Introduction

Welcome to the documentation for the SenseGlove Unreal Engine Plugin (a.k.a. The SenseGlove Unreal Handbook)!

This handbook is an ongoing effort and a work in progress to document the SenseGlove Unreal Engine Plugin. Feel free to visit this handbook on a regular basis.

Due to superior formatting and frequent updates, we recommend the online version of the handbook; nonetheless, it's also available in PDF and ePub formats as well.

> **Tip**
>
> Feel free to check out the SenseGlove Unreal Engine Plugin landing page on Fab as well.

Overview

To help you navigate the SenseGlove Unreal Engine Handbook, we have organized the content into several key sections. This structured layout aims to simplify your journey through the SenseGlove Unreal Engine Plugin, providing clear and detailed guidance at every step.

🚀 Getting Started

This section covers the basics of the SenseGlove Unreal Engine Plugin:

- Installation
  - Via the Epic Games Launcher
  - Via Microsoft Azure DevOps Repositories
- Enabling and Verifying the Plugin Version
- SenseCom
  - SenseCom on GNU/Linux
    - Connect to Nova gloves using Blueman Bluetooth Manager
    - Connect to Nova gloves using Command-line
  - SenseCom on Microsoft Windows
- Enabling XR_EXT_hand_tracking on VR Headsets
  - PCVR Mode
  - Standalone Mode
  - Third-Party Tutorials
- Setup SenseGlove Default Classes
  - SGPawn
  - SGPlayerController
  - SGGameModeBase
  - SGGameInstance
  - SGGameUserSettings
- Setup the Virtual Hand Meshes
- Setup the Wrist Tracking Hardware
- Setup the Grab/Release System

- Setup the Touch System

⚙️ Plugin Configuration

This section provides detailed information on configuring the plugin:

- Plugin Settings
  - Initialization
  - Game User Settings
    - Hardware-benchmarking
  - Tracking
    - Glove-tracking
    - Hand-tracking
    - HMD-tracking
    - Wrist-tracking
      - Debugging
  - Virtual Hand
    - Animation
    - Debugging
    - Grab
    - Haptics
    - Mesh
    - Touch
- Overriding Settings

💡 Miscellaneous

Toipcs that do not fall under any specific category:

- SenseGlove Console Commands
- Deploying to Android (Standalone)
  - Third-Party Tutorials
- Upgrade Guide
- Optimizing for Higher FPS

🛠️ Advanced Topics

For users familiar with the basics, this section explores advanced features of the plugin:

- Safe Glove Access in Blueprint
- OpenXR
  - Consuming FXRHandTrackingState
    - Blueprint
    - C++
  - Consuming FXRMotionControllerData
    - Blueprint
    - C++
  - Third-Party Tutorials

🔌 Low-Level API

This section delves into the SenseGlove low-level API:

- Low-Level Blueprint API
- Low-Level C++ API

📑 Appendix

The appendix contains various extra useful information:

- Platform Support Matrix
- Planned Features Completion Status
- Changelog
- Directory Structure
- Extra Resources

Plugin Installation

The SenseGlove Unreal Engine Plugin could be installed using various methods:

- Via the Epic Games Launcher by navigating to the SenseGlove Unreal Engine Plugin landing page on Fab.
- Via the SenseGlove Unreal Engine Plugin Microsoft Azure DevOps repository.

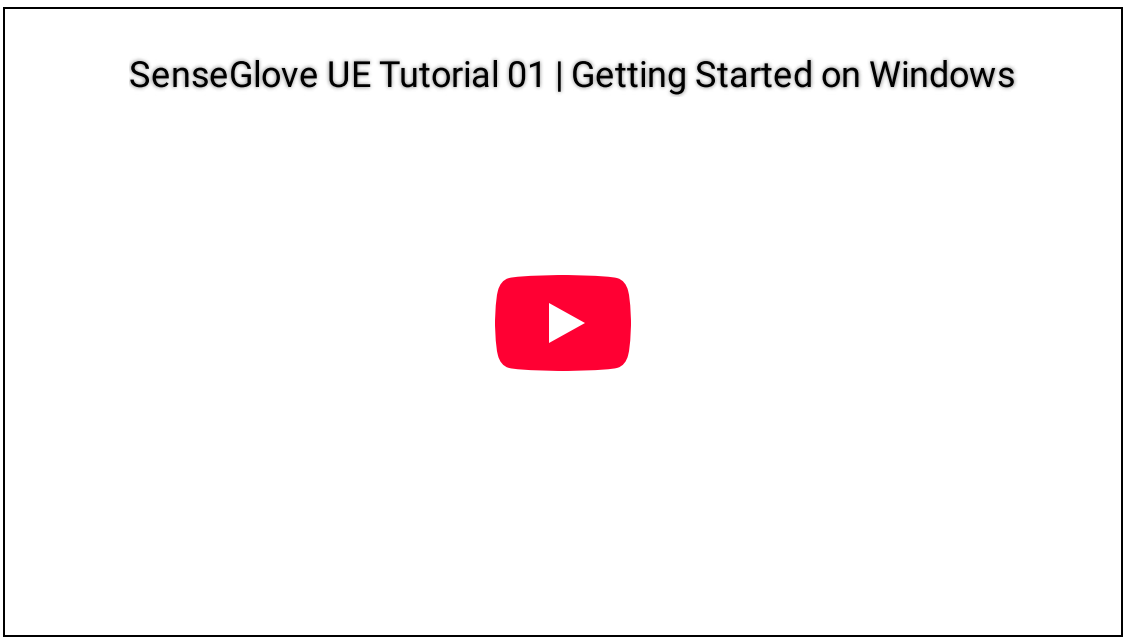
In the following chapters, we discover each of those methods:

- Installation via the Epic Games Launcher
- Installation via Microsoft Azure DevOps Repositories

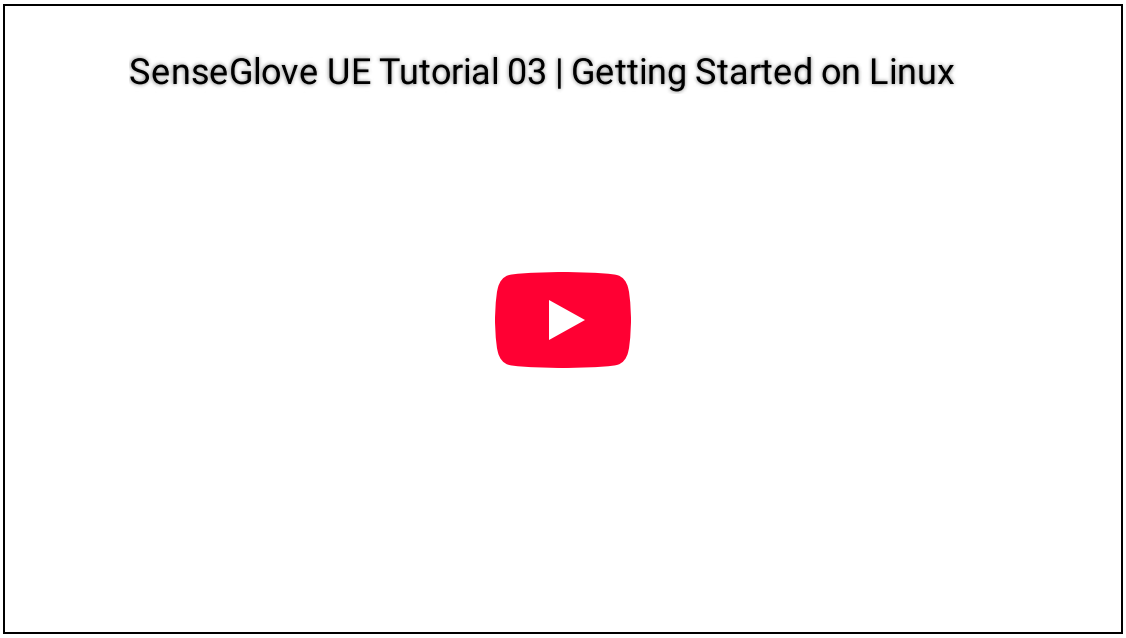
Video Tutorials

We also have older videos demonstrating both installation methods on Microsoft Windows and GNU Linux in more detail.

- Plugin installation guide for Microsoft Windows:



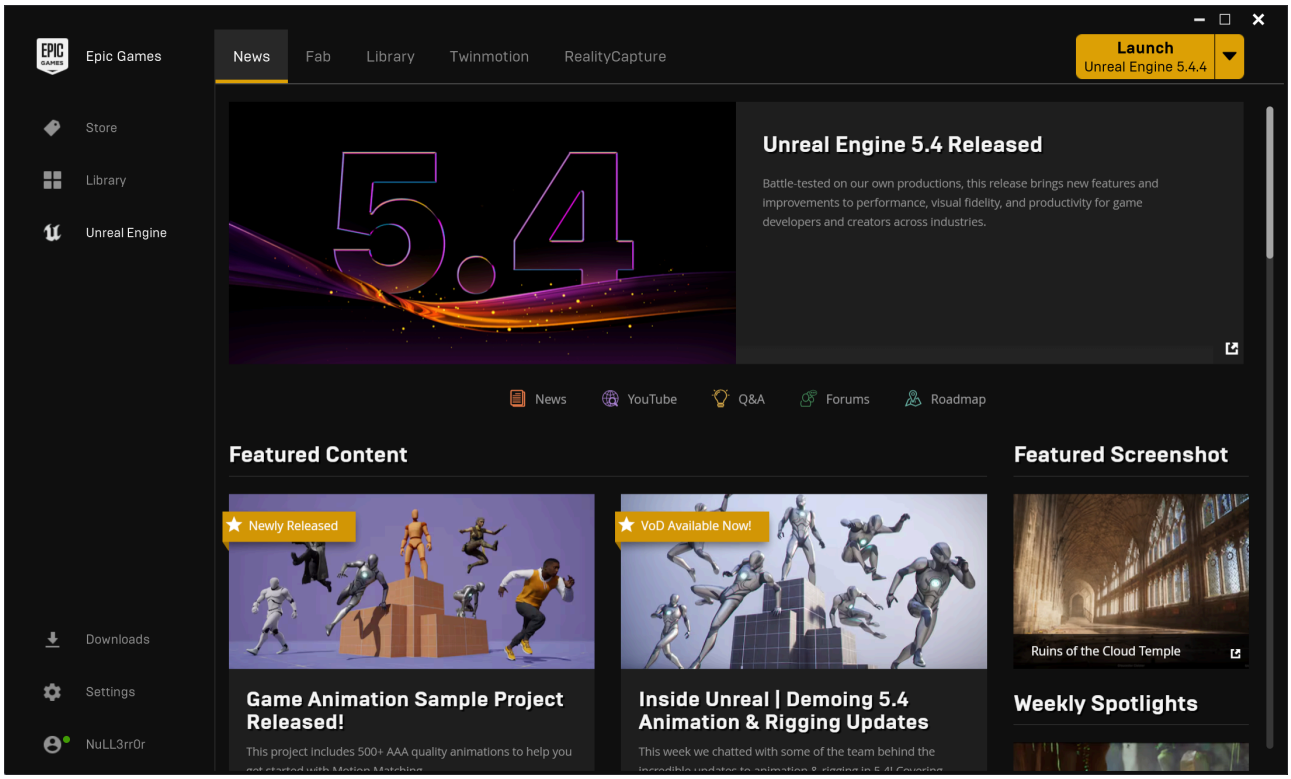
- Plugin and examples installation guide for GNU/Linux:



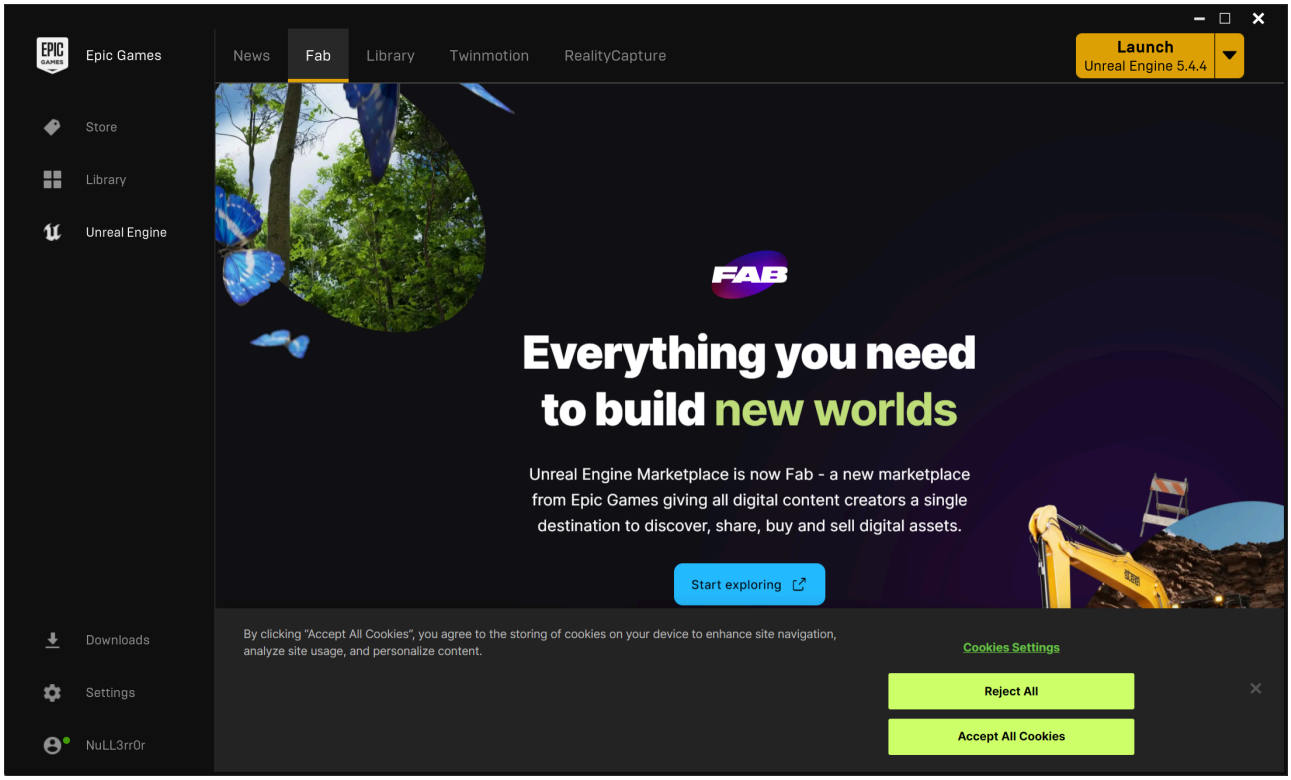
Plugin Installation via the Epic Games Launcher

Before beginning the plugin installation via the Epic Games Launcher, ensure you have signed into your Epic Games account on the Epic Games Launcher and that you have a supported version of Unreal Engine installed. Supported engine versions can be found in the Platform Support Matrix.

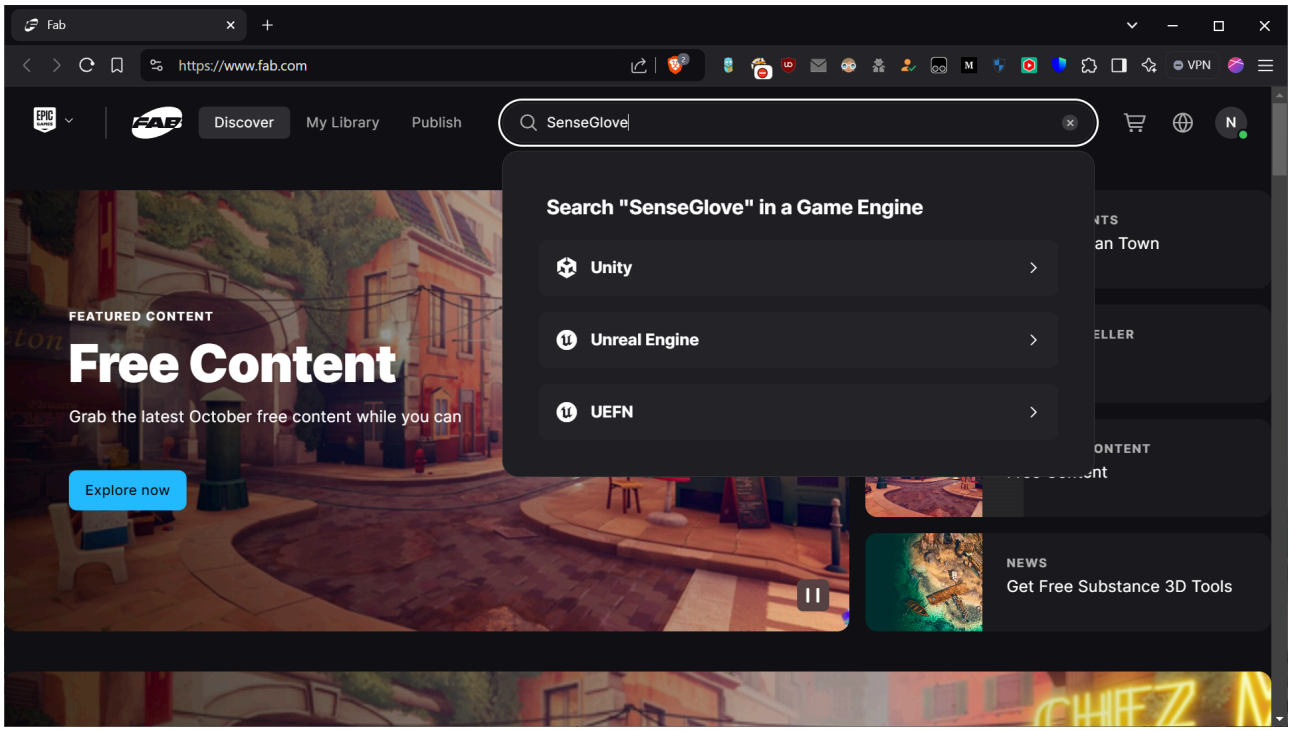
1. Run the Epic Games Launcher.



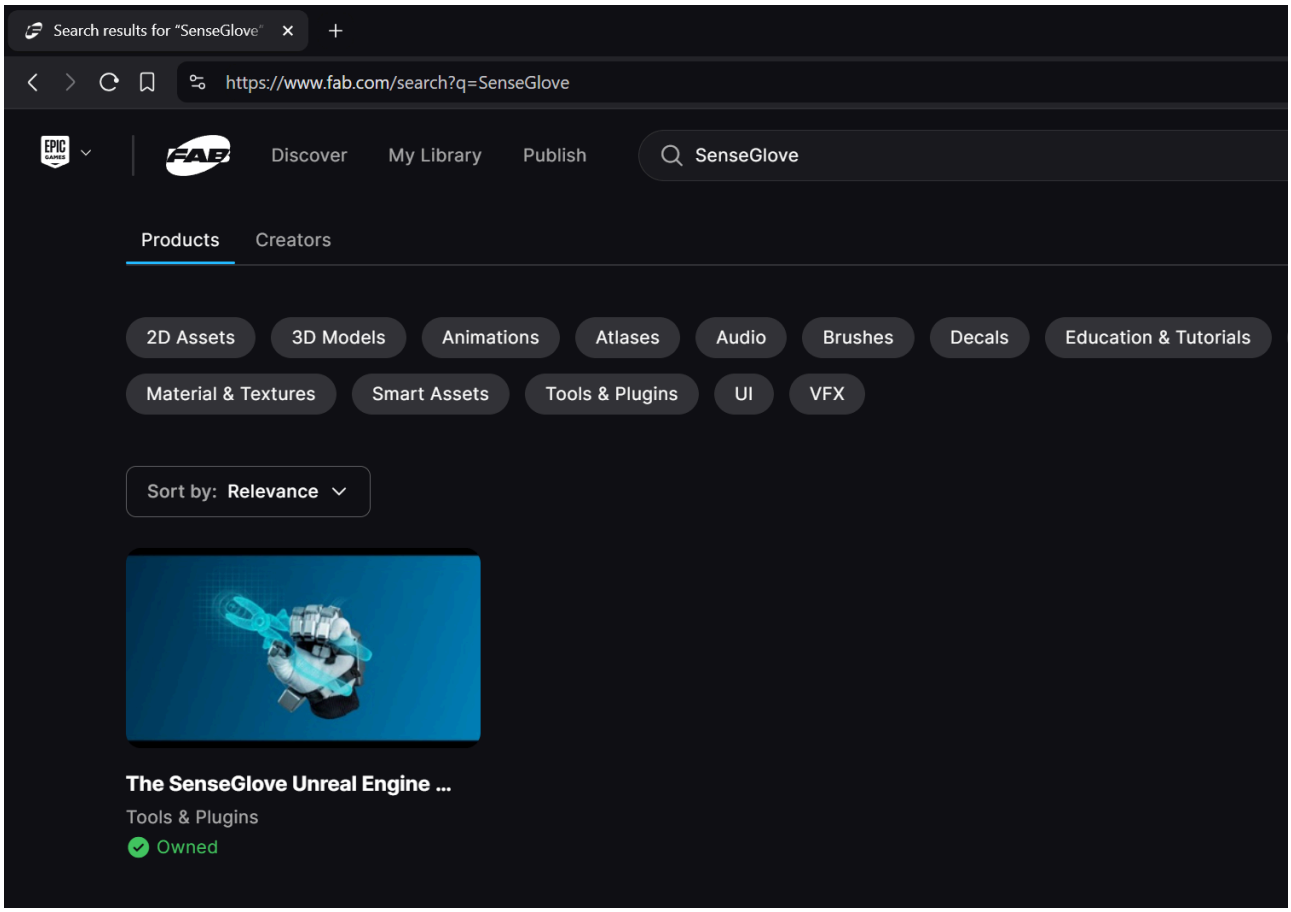
2. Navigate to the `Fab` tab and click `Start exploring` button which in turn opens your default web browser pointing to the Fab home page.



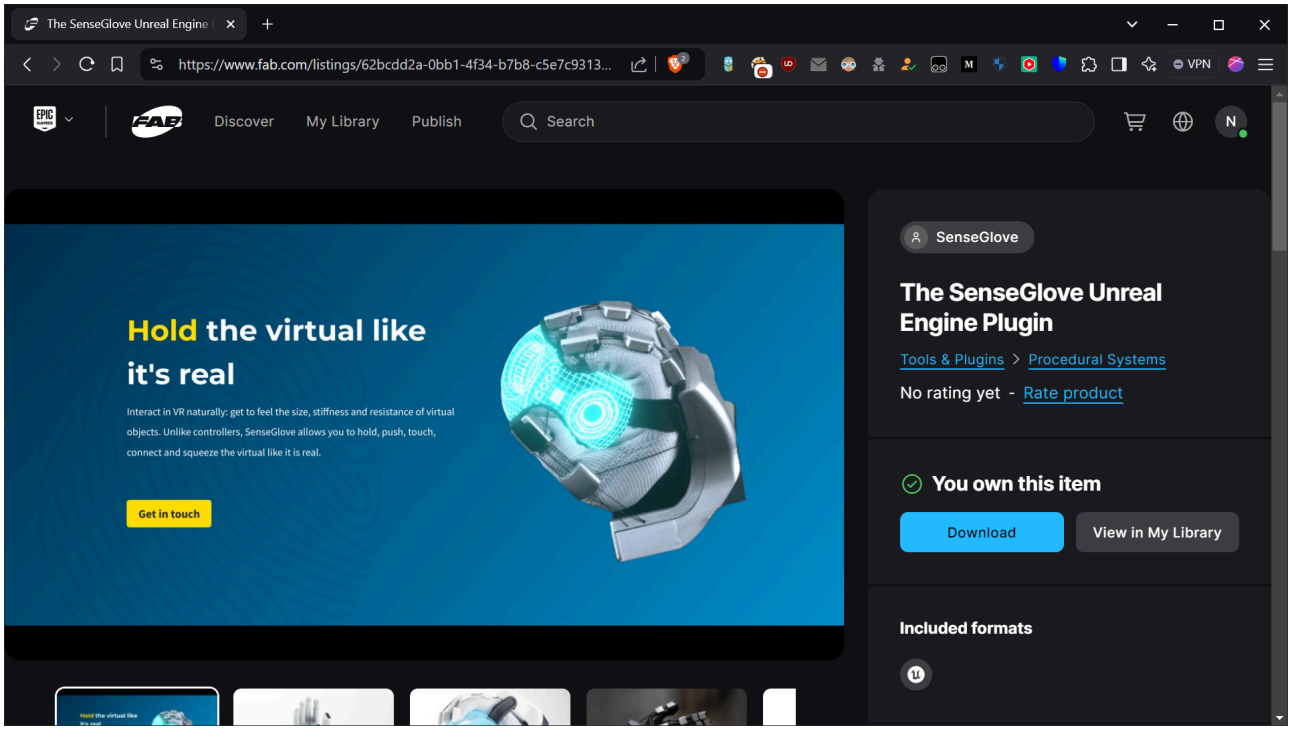
3. On the Fab home page, enter the term `SenseGlove` in the search box and press Enter. Alternatively, you can go directly to the SenseGlove Unreal Engine Plugin landing page on Fab directly instead of taking the above two steps.



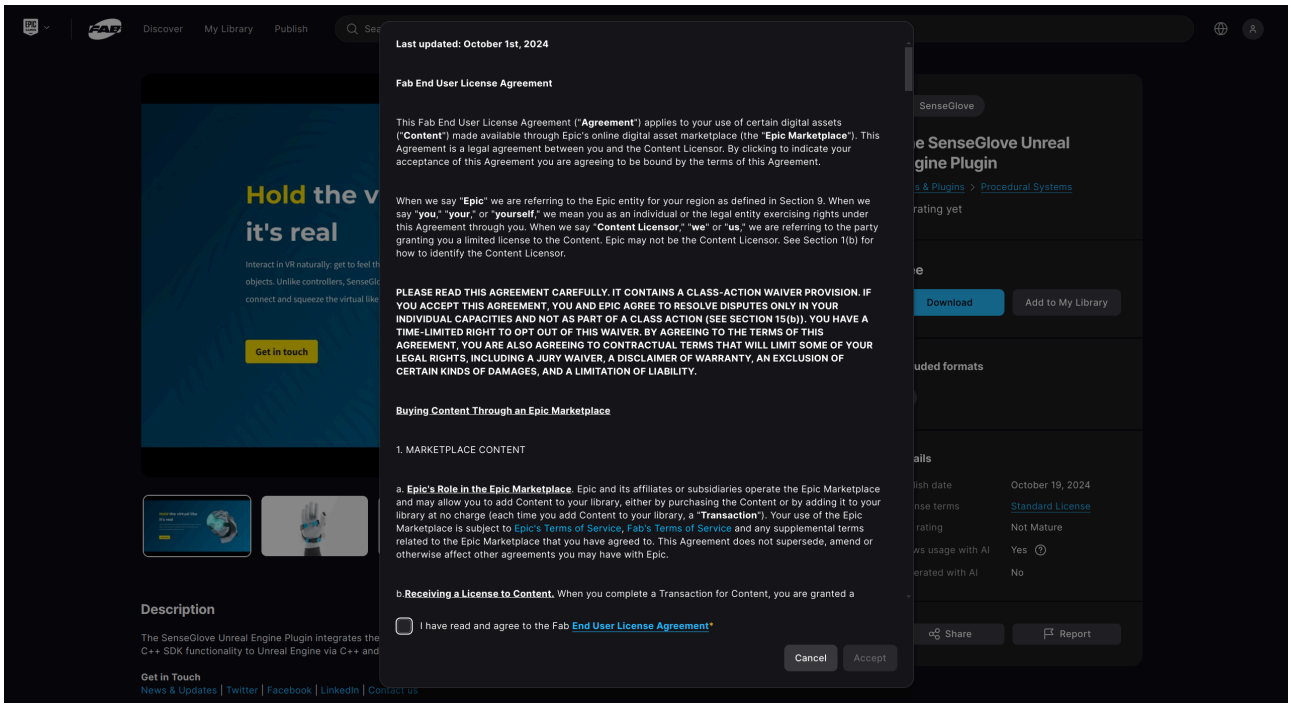
4. Click on the `SenseGlove Unreal Engine Plugin` in the search results to navigate to its dedicated page.



5. On the SenseGlove Unreal Engine Plugin landing page on Fab click the `Download` button.

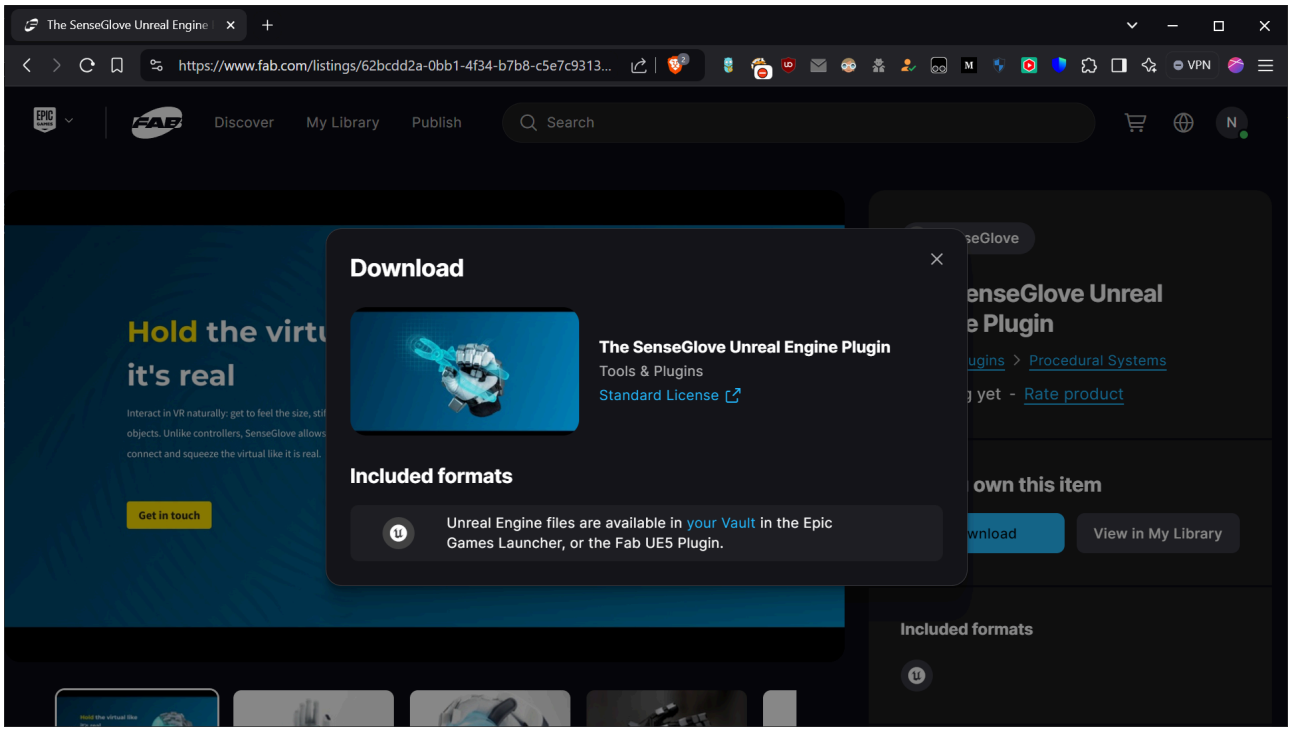


6. If this is your first download from Fab, you will need to agree to the Fab End User License Agreement (EULA) before proceeding.

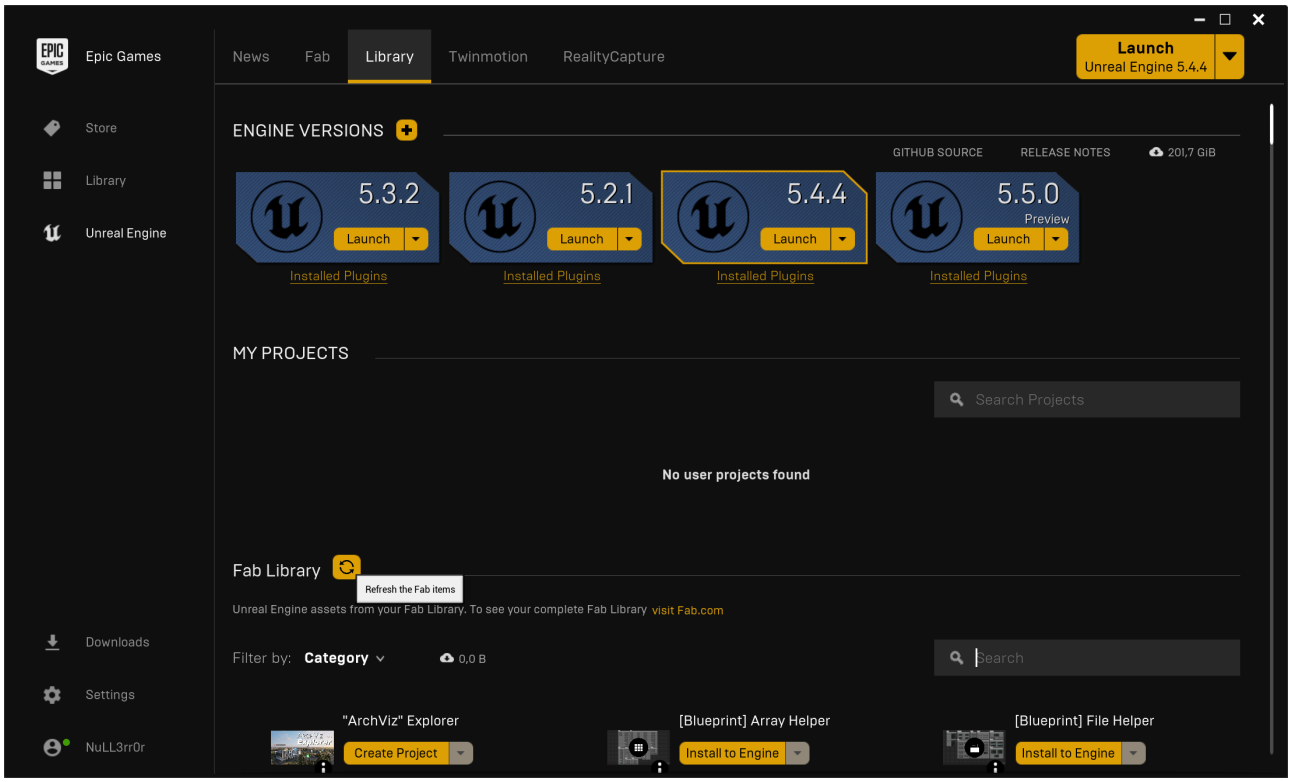


7. After clicking `Download`, a pop-up will notify you that the plugin is available in your Vault in the Epic Games Launcher, or the Fab UE5 Plugin.

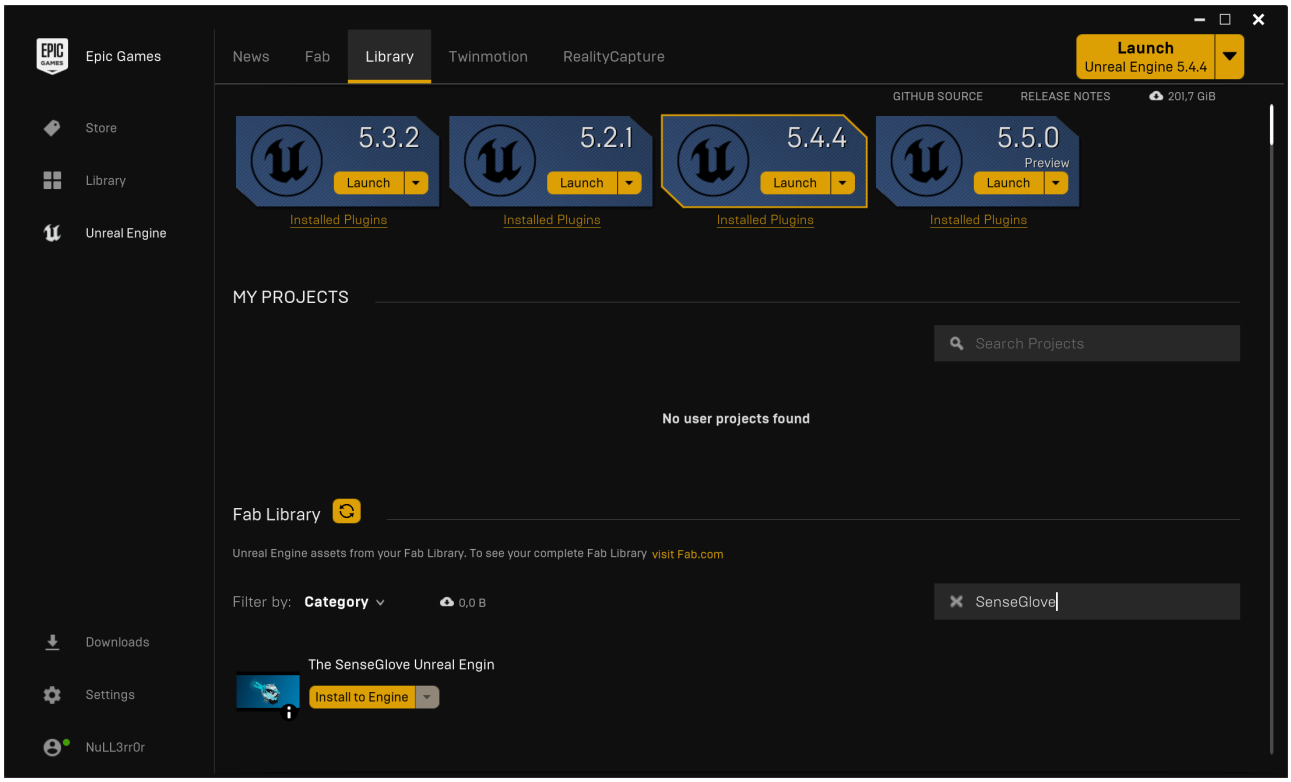
> **Note**
>
> According to the Fab launch announcement:
>
> The Fab integration in UEFN is undergoing maintenance and will be back online shortly, and the Fab integration in the Unreal Engine 5 Editor is coming soon.



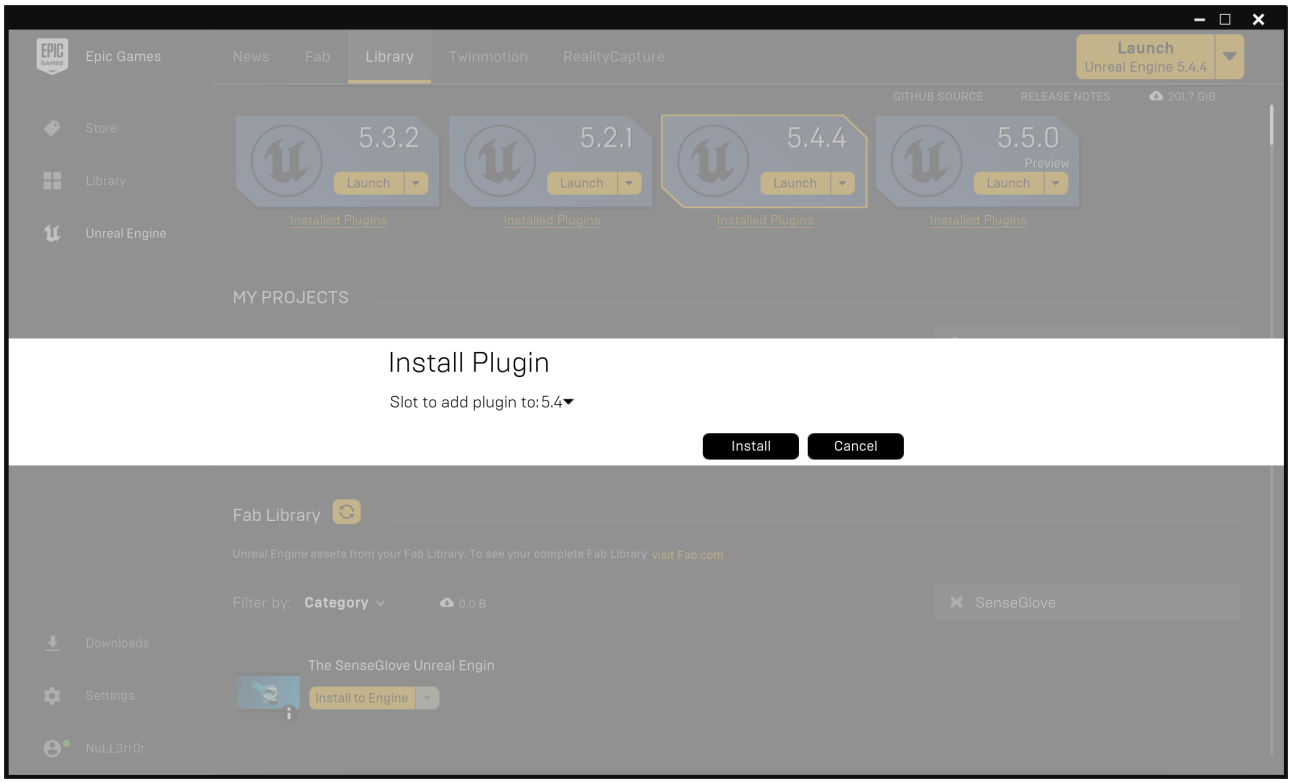
8. Go back to the Epic Games Launcher, navigate to the `Library` tab, and in the `Fab Library` section, click the `Refresh Fab items` button.



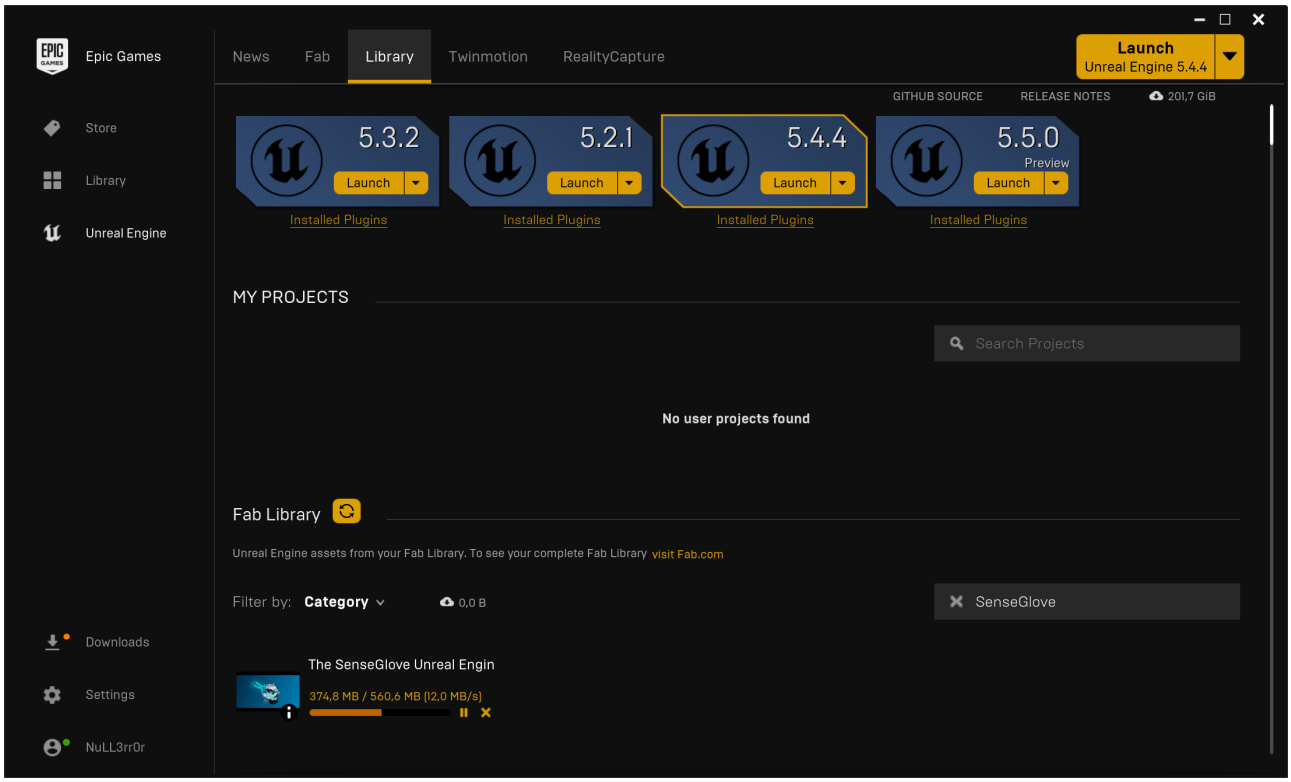
9. Once the Fab library is refreshed and synchronized, use the Vault search box to find the `SenseGlove Unreal Engine Plugin`. Click the `Install to Engine` button.



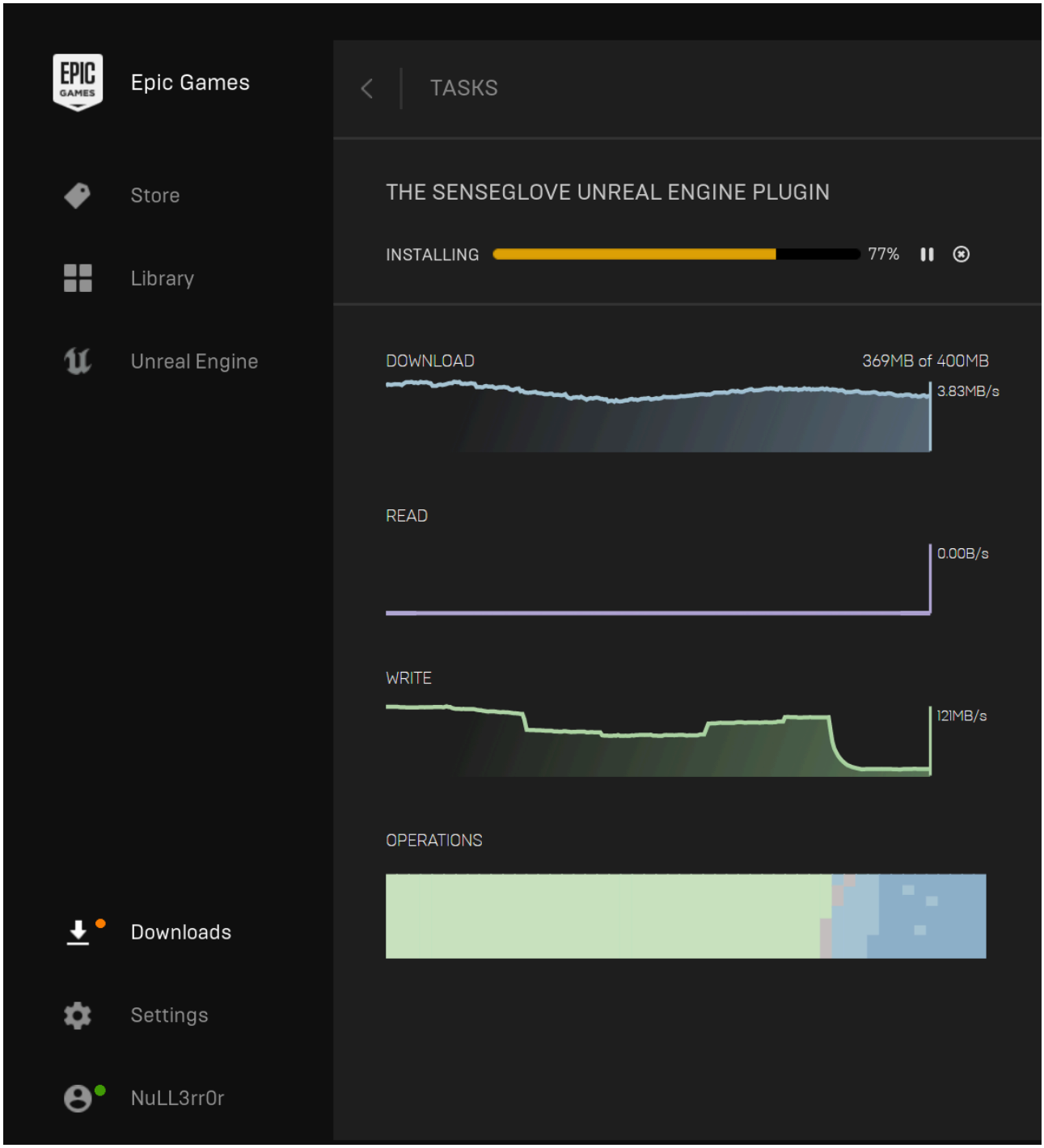
10. You'll be prompted to choose a compatible engine version. Select your desired engine version from the list, then click `Install`.



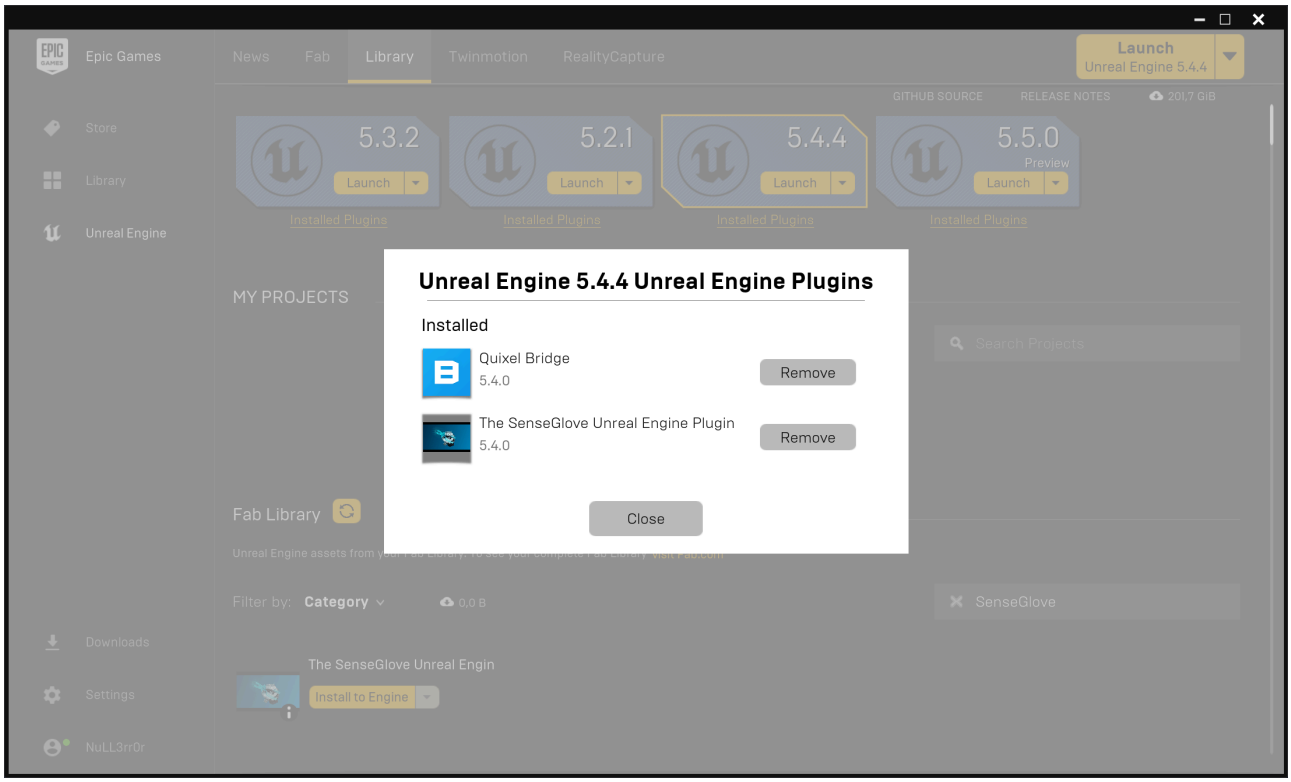
11. The Epic Games Launcher will show the plugin's download and installation progress. Please wait for it to complete.



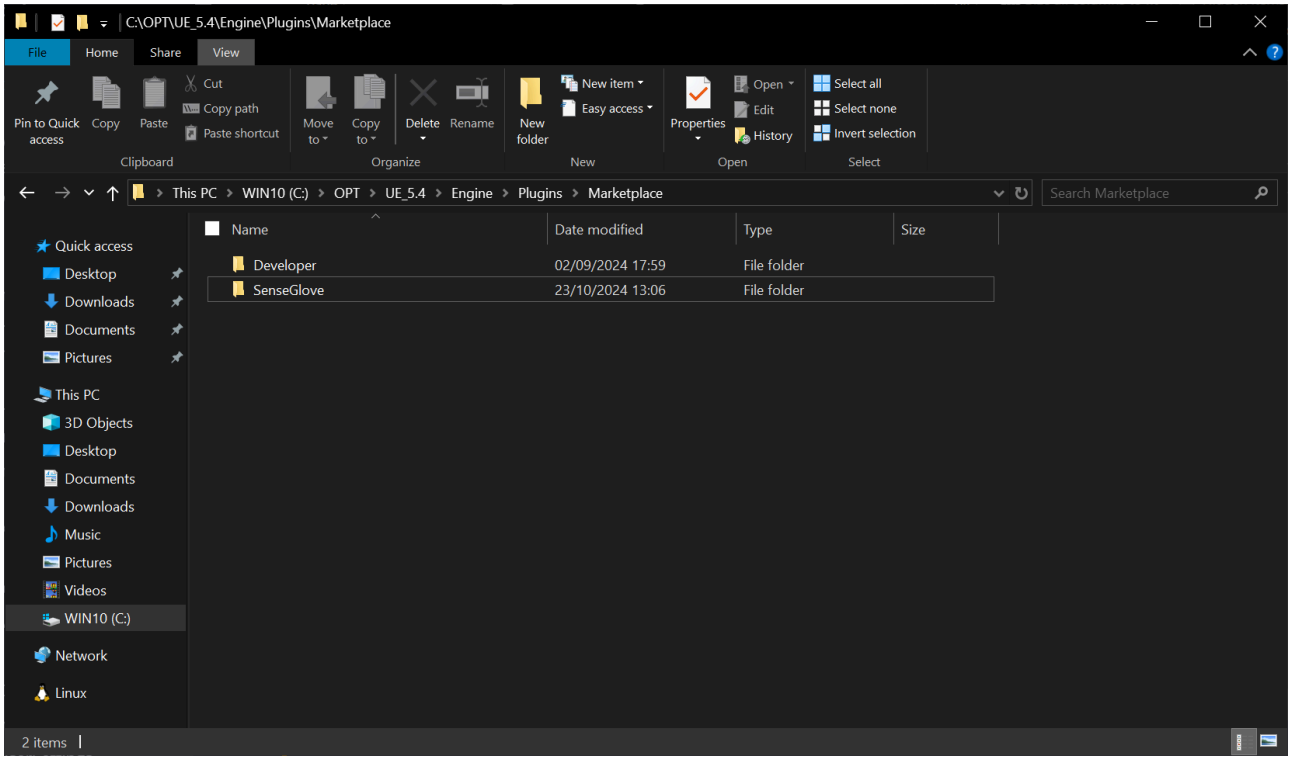
12. While the download and installation are in progress, you can see the progress in more details by clicking on the `Downloads` section on the sidebar.



13. Once the download and installation are complete, verify its installation by clicking `Installed Plugins` under the engine you've just installed it to. The SenseGlove plugin should appear as installed among other currently installed plugins.



14. One last confirmation could be navigating to `YourEngineInstallationPath/Engine/Plugins/Marketplace` directory. The SenseGlove Unreal Engine Plugin source and binaries can be found inside this directory. This is especially useful in case one desires to copy the plugin for example to their own project's source code to run it at the project level instead of running it at the engine level.



> ### Warning
>
> Please note that it is best practice to install the plugin either at the project level or the engine level, but not both. Having the plugin installed in both locations, at the same time, can lead to various issues, especially if the version of the plugin installed at the engine level differs from the one installed at the plugin level. A guide on verifying the plugin version is also available as well.

Plugin Installation via Microsoft Azure DevOps Repositories

While plugin installation via the Epic Games Launcher is the most convenient method for most users to obtain and install the latest version of the SenseGlove Unreal Engine Plugin via Fab, there might be valid reasons to instead download and install the plugin directly from the SenseGlove Unreal Engine Plugin Microsoft Azure DevOps Repository. These reasons may include:

- Downloading an older version that is no longer available on Fab.
- Downloading a recent version that has been submitted to Fab, but is still awaiting approval and publication by the Fab Team.
- Downloading an under-development, unstable release of the plugin for testing purposes.
- Or, any other specific needs that require direct access to the repository.

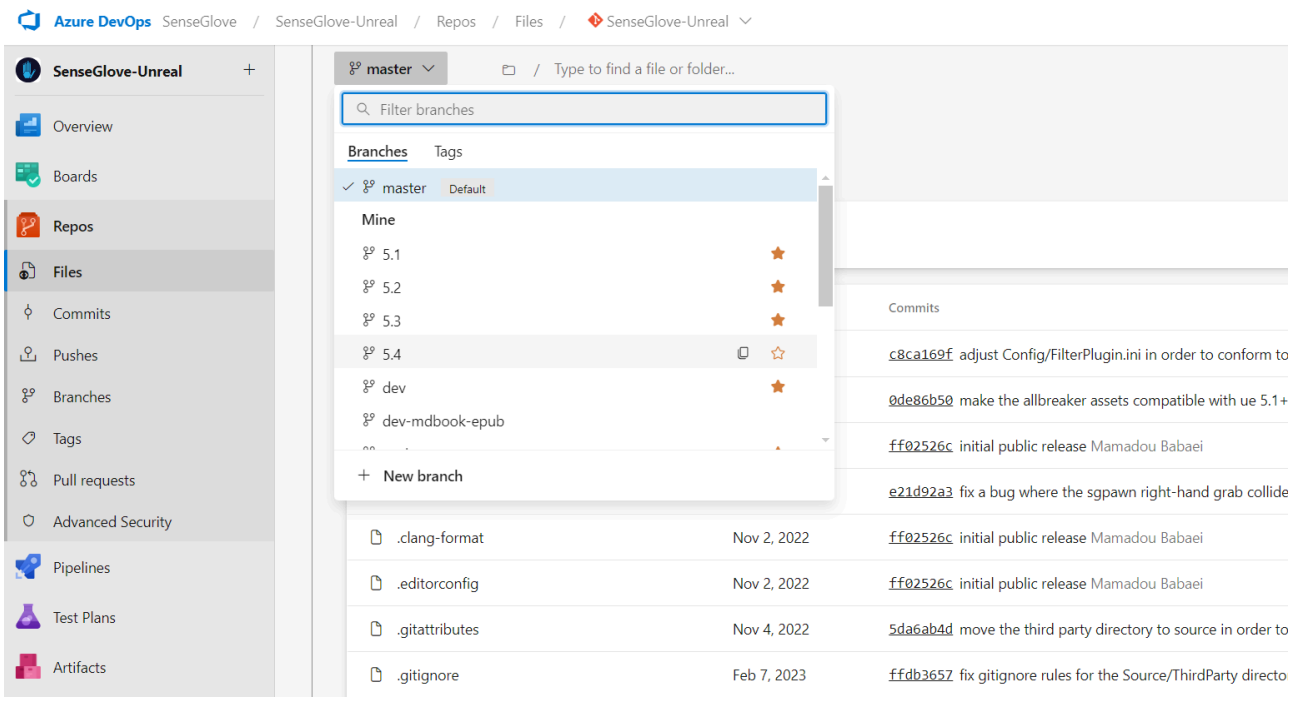
Nonetheless, here is a step-by-step guide to downloading and installing the plugin from the Microsoft Azure DevOps Repositories.

Download a Specific Version

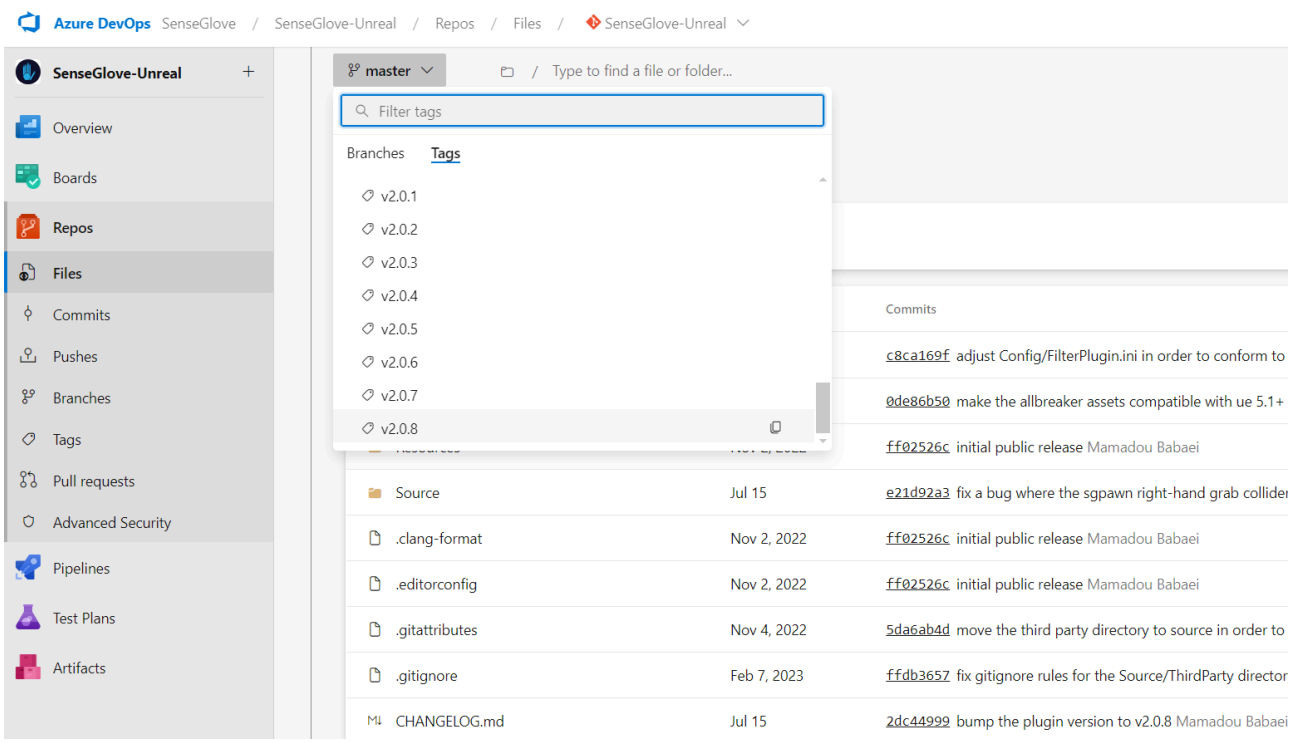
To download a specific version of the plugin, follow these steps:

1. Navigate to the the SenseGlove Unreal Engine Plugin Microsoft Azure DevOps Repository.

2. Locate the branch dropdown menu at the top of the page, just below the navigation bar, and next to the `Copy to clipboard` icon. There you'll find a dropdown menu. By default, it usually selects the master branch.



3. Use the dropdown menu to choose a desired branch containing the source code for a specific version of Unreal Engine or a specific release of the plugin marked with a release tag.



Note

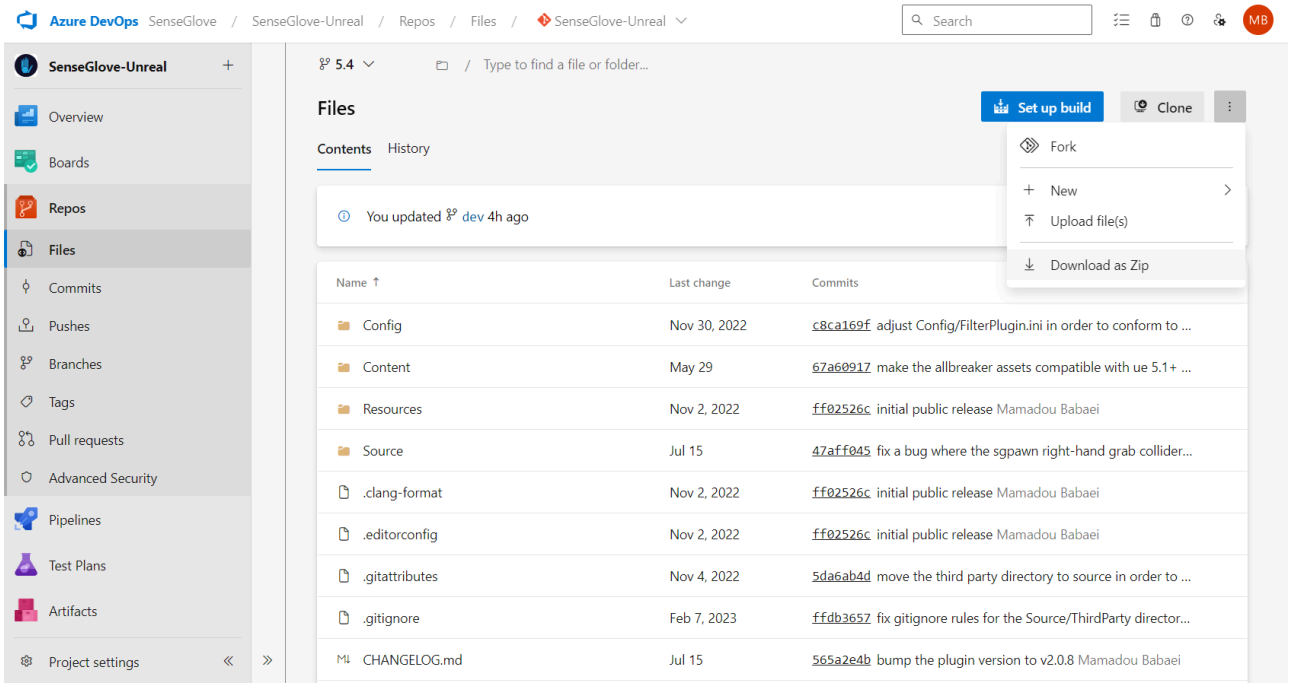
A branch named with engine version numbers, such as `5.4`, `5.3`, etc., ususally contains the source code for the latest stable version of the plugin compatible with that specific Unreal Engine version, provided that version is still supported. For a comprehensive list of supported engine versions please refer to the Platform Support Matrix.

As a general rule of thumb, the `master` branch should work with any supported Unreal Engine version. This is because it does not specify any `EngineVersion` inside the main `.uplugin` file. However, there may be rare exceptions where it does not work due to breaking changes between engine versions that the plugin cannot accommodate. One such a instance occurred with version `2.0.x` of the plugin, where some breaking changes prevented UE `5.1` from sharing similar code with versions `5.2+`. For this reason, it is generally recommended to select a branch specific to the version of the Unreal Engine you intend to use with the plugin.

The same principles that apply to the `master` branch also apply to the `dev` branch, which will discuss later.

We will also cover how to obtain a working version from a tag for scenarios like the one mentioned above.

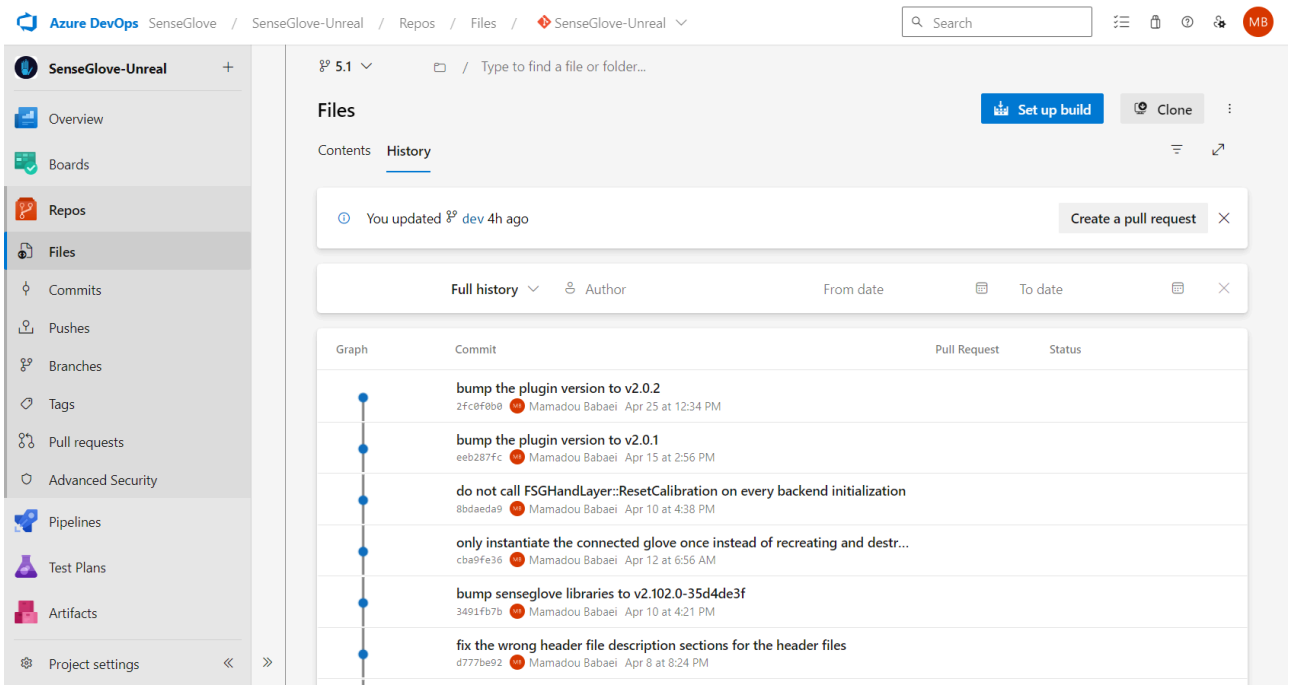
4. After selecting your desired branch or tag, click on the kebab menu (three vertical dots) located at the top right of the screen and choose `Download as Zip` to obtain the source code for that branch or tag.



Download a Specific Version for a Specifc Unreal Engine Version

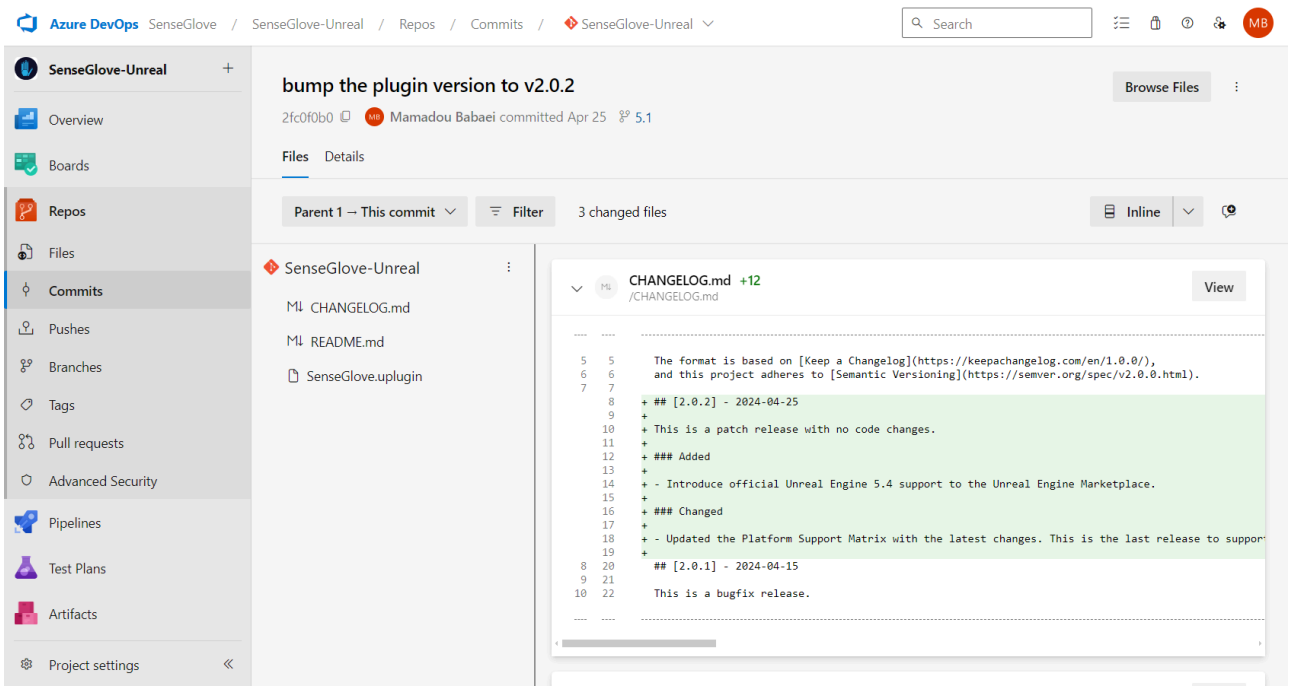
As mentioned earlier, due to breaking changes between Unreal Engine versions, it might not be feasible to share the same source code across different Unreal Engine versions. Since release tags are created from the `master` branch, they contain code compatible only with the latest version of Unreal Engine. Therefore, the instructions for downloading a specific version from a release tag might not work with some Unreal Engine versions. In such cases, you can use an alternative approach:

1. First, choose the appropriate branch for your desired Unreal Engine version from the branch dropdown menu, as discussed earlier. Then navigate to the `History` tab.

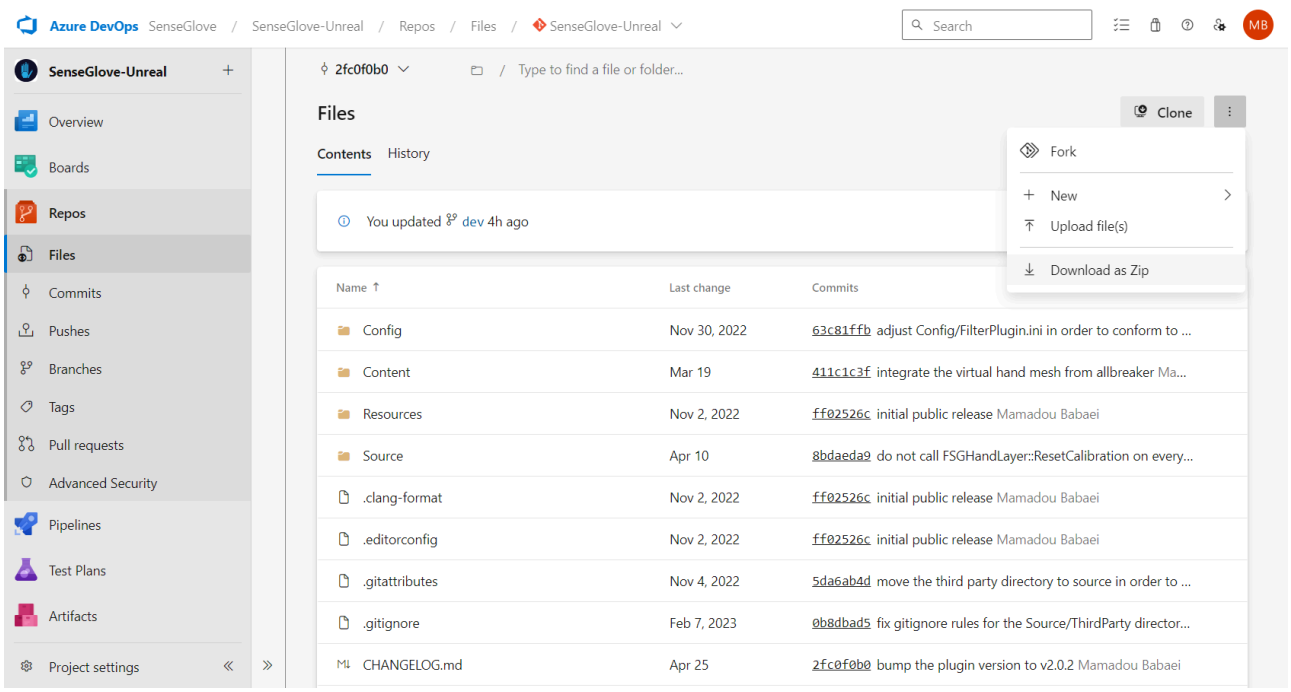


2. Look via the commit history for a commit message that says `bump the plugin version to vX.X.X` as all releases are finalized with this exact commit message and the plugin version. Next, click on the commit message for the version you are looking for.

3. Once you've selected the correct commit, click on the `Browse Files` button next to the kebab menu (three vertical dots) at the top right of the screen.



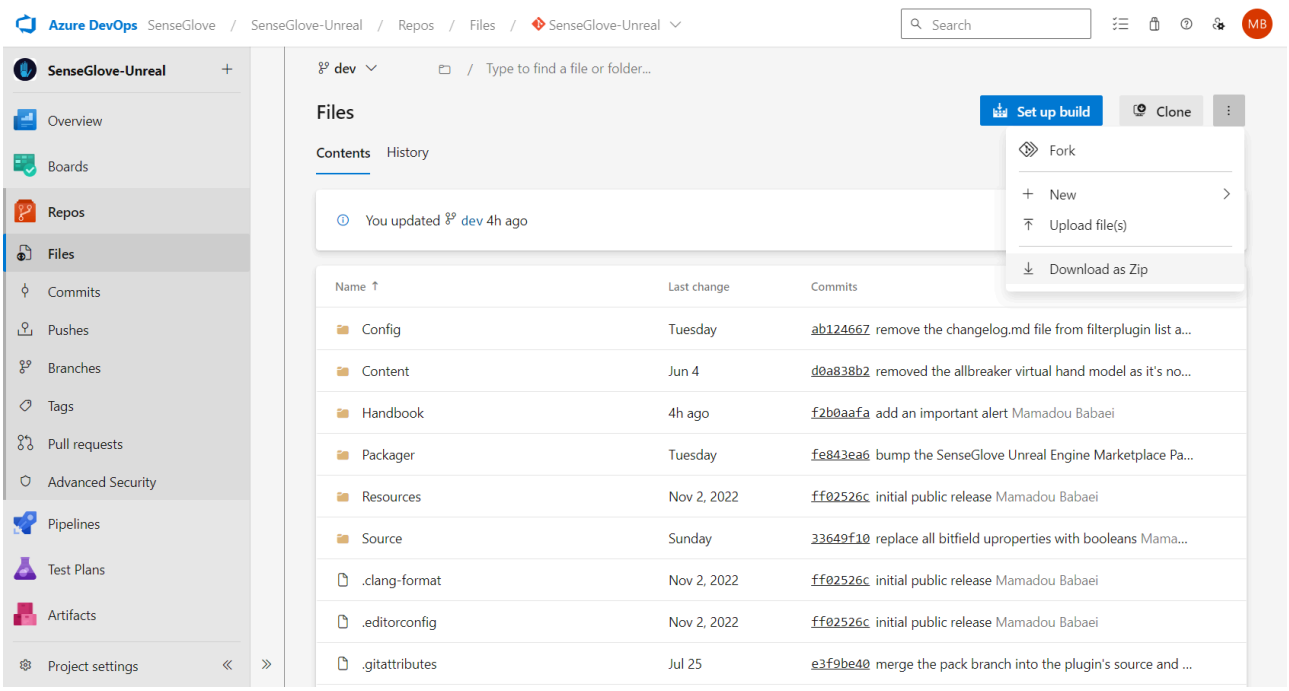
4. You should now be in the `Content` tab, with the branch dropdown menu displaying the commit hash instead of a branch name or tag. Click on the kebab menu (three vertical dots) again, and select `Download as Zip`. This will give you a zip file containing the exact release you need, compatible with your chosen Unreal Engine version.



Download the Bleeding-edge Development Branch

> **Caution**
>
> The `dev` branch is an active development branch that is constant and ongoing changes. As a result, the code on this branch is primarily untested and therefore not production-ready. It may not even compile successfully or may lack comprehensive documentation. For any serious development, it is generally recommended to use a stable release of the plugin. The `dev` branch is publicly accessible to give you a preview of upcoming features and for trial purposes only.
>
> The most up-to-date documentation for the dev branch can usually be found at: at: https://unreal.dev.senseglove.com/next.

Downloading the `dev` branch is as easy as choosing the `dev` branch from the branch dropdown menu (as discussed earlier) and then choosing `Download as Zip` from the kebab menu (three vertical dots).



Installation

Once you have obtained the desired plugin version compatible with the Unreal Engine version you have in mind using any of the methods mentioned above, it's time to build and install the plugin. There are two ways to install the SenseGlove Unreal Engine Plugin, one is at the engine level, and the other is per project.

- **Engine-level installation**: this method makes the plugin accessible to any project within that Unreal Engine version.
- **Per-project installation**: this method makes the plugin accessible only to a specific project.

> **Warning**
>
> Please note that it is best practice to install the plugin either at the project level or the engine level, but not both. Having the plugin installed in both locations, at the same time, can lead to various issues, especially if the version of the plugin installed at the engine level differs from the one installed at the plugin level. A guide on verifying the plugin version is also available as well.

Engine-level installation

Per-project installation

1. Locate your existing C++ or Blueprint project, or create a new project from scratch.

> **Important**
>
> Before proceeding, make sure your project's Unreal Editor is closed, and you do not have your project open in your C++ IDE to avoid any issues.

2. Inside your project's root directory create a new `Plugins` directory if you don't have one already.

3. Inside the `Plugins` directory create a new directory named `SenseGlove`.
4. Extract the content of your downloaded zip file into the `SenseGlove` directory.
5. Remove any directories or files that are only meant for use by the SenseGlove Unreal Engine Plugin maintainers. These are not part of the distributed plugin package and are not required by either Unreal Engine or the SenseGlove Unreal Engine Plugin to function correctly.

The mandatory files and folders to stay are as follows:

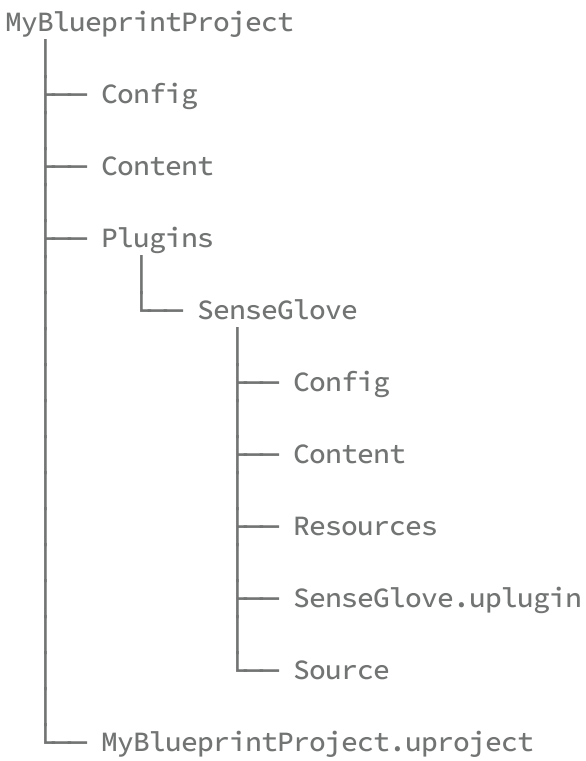
```
Config
Content
Resources
Source
SenseGlove.uplugin
```

Anything else can be safely removed. For example, these files and folders can be safely deleted:

```
Handbook
Packager
.clang-format
.editorconfig
.gitattributes
.gitignore
README.md
```

6. Ensure your project has the correct structure.

For a Blueprint-only project, it should look something like this:



For a C++ project, the structure should look like this:

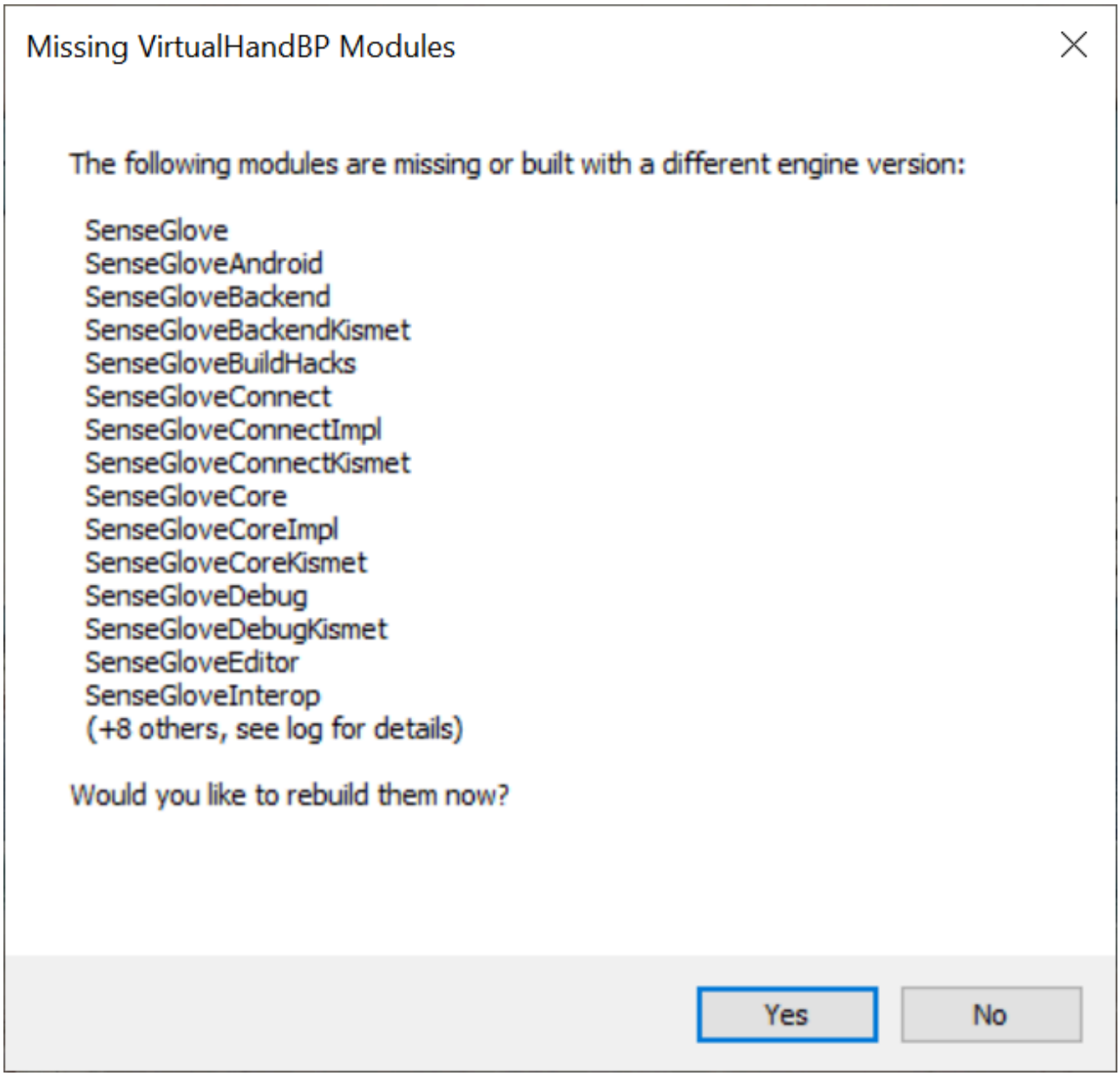
```
MyCppProject
├── Config
├── Content
├── Plugins
│   └── SenseGlove
│       ├── Config
│       ├── Content
│       ├── Resources
│       ├── SenseGlove.uplugin
│       └── Source
├── Source
└── MyCppProject.uproject
```

> **Tip**
>
> If you are keeping your project under Git and Git LFS, consider keeping the `.gitignore` and `.gitattributes` as they help keep irrelevant files out of the remote repository, or manage binary blobs efficiently.

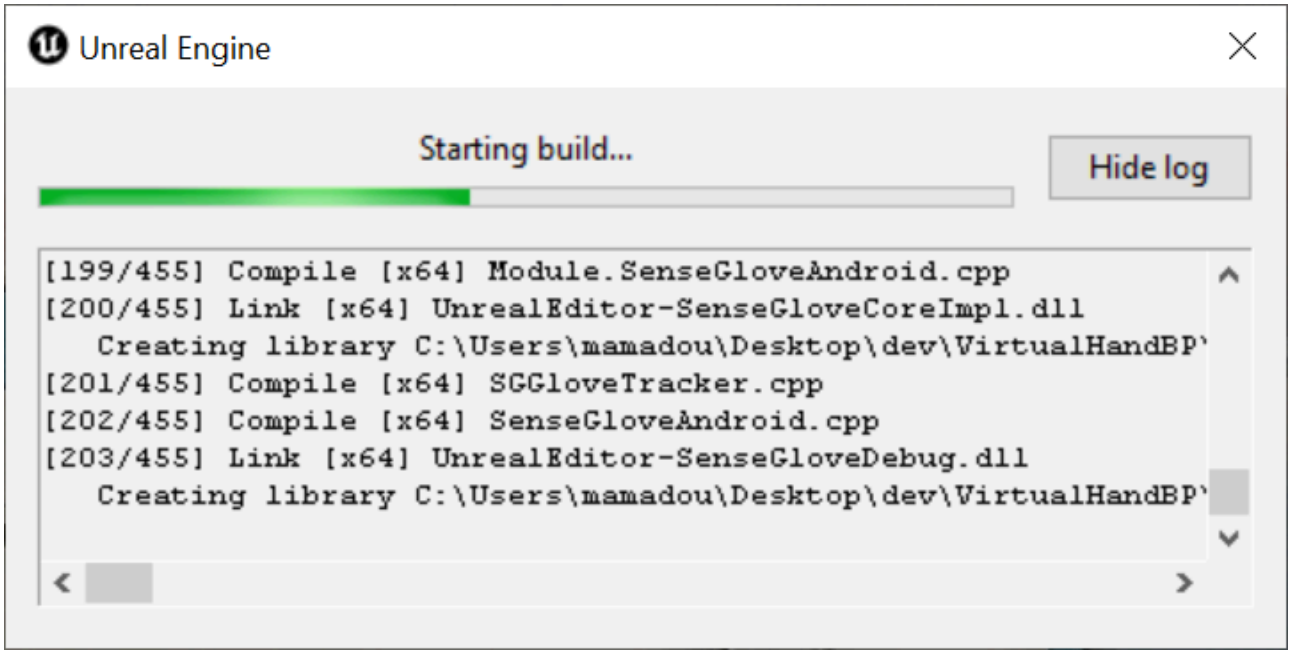
7. OK, now it's time to build the plugin.

> **Note**
>
> For Linux build instructions see the Linux Build Instructions section.

For a Blueprint-only project, on Microsoft Windows simply double-clicking the project's `.uproject` file should present you with a pop-up informing you that some binary modules are missing.



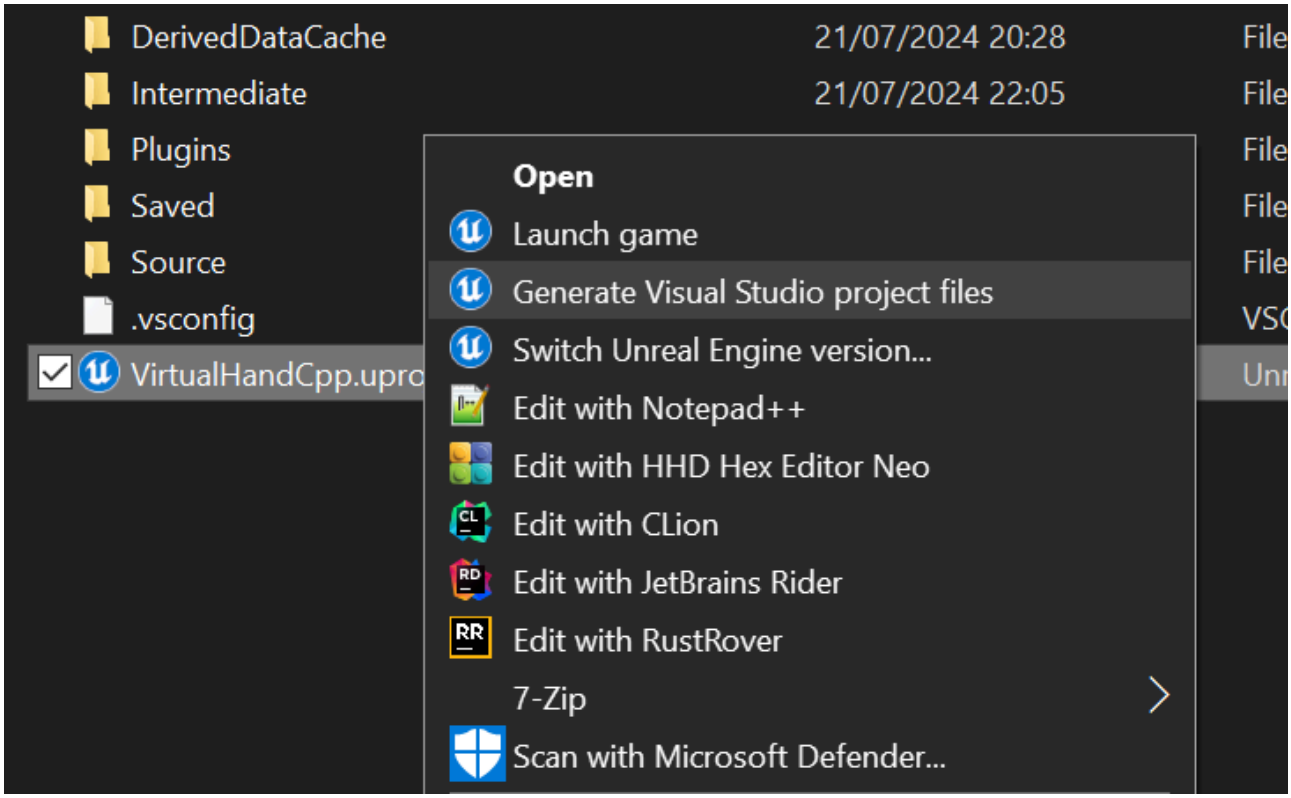
After confirming, the build process will start automatically, and a dialog indicating the build progress will be shown:



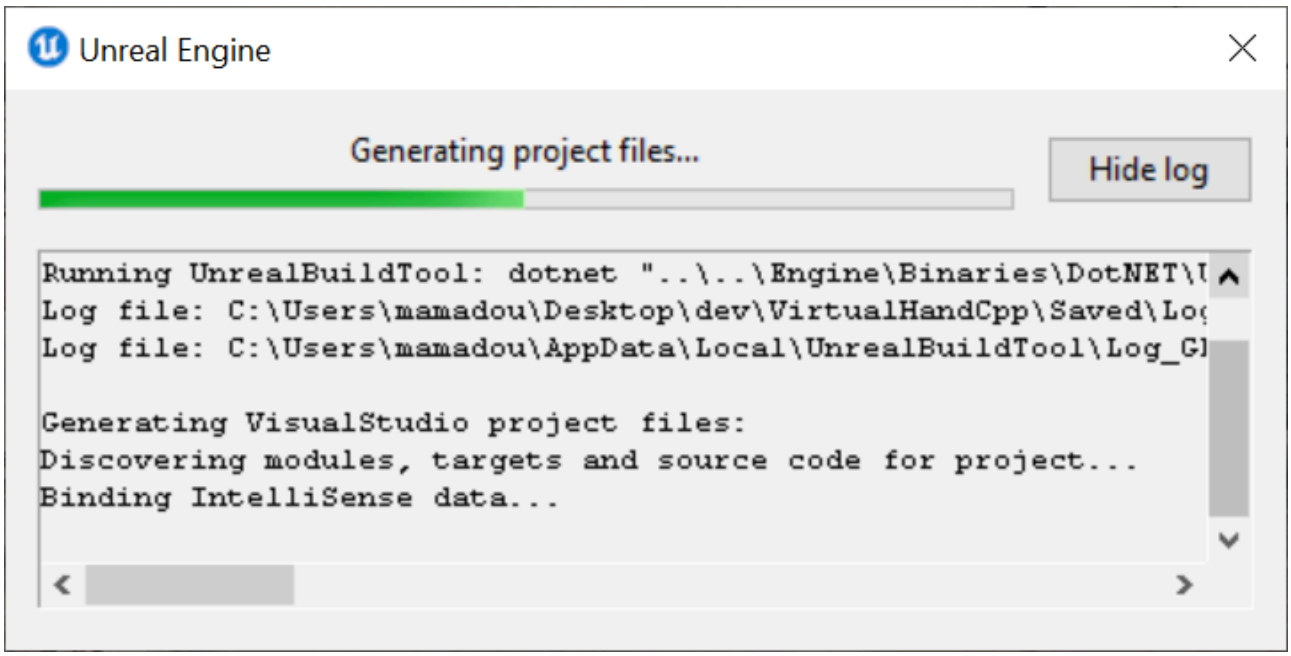
Once finished successfully, the project will be loaded.

> **Note**
>
> Sometimes, due to an esoteric bug in some versions of Unreal Engine, the build process for Blueprint-only projects may immediately fail after choosing `Yes` in the `Missing Modules` dialog. If this happens, one workaround would be to try to build the plugin inside a temporary C++ project, then copy the `Plugins/SenseGlove` folder containing the binaries, from the C++ project to your Blueprint project and then try to reopen the project again.

For C++ projects, on Microsoft Windows, right-click on your C++ `.uproject` file and choose `Generate Visual Studio project files`:



A dialog will pop up shows you the progress of generating the Visual Studio project files:



Once the project files are generated, open up the C++ project in your preferred C++ IDE and build the project. After this, the project can be loaded in the Unreal Editor.

8. Once the plugin has been built successfully, ensure the SenseGlove Unreal Engine is enabled and verify the plugin version matches the expected version.

Linux Build Instructions

When building the SenseGlove Unreal Engine Plugin on Linux, you won't encounter the `Missing Modules` dialog that appears on Microsoft Windows. Instead, examining the Unreal Editor logs reveals that the Unreal Editor automatically chooses `No` in response to the `Would you like to rebuild them now?` question as the `No is implied` states.

```
$ /path/to/UnrealEngine/Engine/Binaries/Linux/UnrealEditor \
    /path/to/MyBlueprintProject/MyBlueprintProject.uproject

LogLinux: Warning: MessageBox: The following modules are missing or built
with a different engine version:

  SenseGlove
  SenseGloveAndroid
  SenseGloveBackend
  SenseGloveBackendKismet
  SenseGloveBuildHacks
  SenseGloveConnect
  SenseGloveConnectImpl
  SenseGloveConnectKismet
  SenseGloveCore
  SenseGloveCoreImpl
  SenseGloveCoreKismet
  SenseGloveDebug
  SenseGloveDebugKismet
  SenseGloveEditor
  SenseGloveInterop
  (+8 others, see log for details)

Would you like to rebuild them now?: Missing MyBlueprintProject Modules: No
is implied.
LogCore: Engine exit requested (reason: EngineExit() was called)
LogExit: Preparing to exit.
LogPakFile: Destroying PakPlatformFile
LogExit: Exiting.
LogInit: Tearing down SDL.
Exiting abnormally (error code: 1)
```

If your Unreal Engine installation on Linux was obtained from the GitHub Sources y you can generate the project files using the following command:

```
$ /path/to/UnrealEngine/GenerateProjectFiles.sh \
    /path/to/MyProject/MyProject.uproject \
    -editor -game -makefile
```

However, if you are using a prebuilt Linux version of Unreal Engine, the main `GenerateProjectFiles.sh` script at the engine root does not exists. Instead, we have to invoke the underlying `GenerateProjectFiles.sh` script located elsewhere. This is a different script which shares the same name and is also present in the GitHub

sources. The main `GenerateProjectFiles.sh` script at the engine root is actually a wrapper around this script.

```
$ /path/to/UnrealEngine/Engine/Build/BatchFiles/Linux/GenerateProjectFiles.sh
\
    /path/to/MyProject/MyProject.uproject \
    -editor -game -makefile
```

Still, running the any of the above commands on a Blueprint project results in the following error:

```
$
/path/to/UnrealEngine/Engine/Build/BatchFiles/Linux/GenerateProjectFiles.sh \
    /path/to/MyBlueprintProject/MyBlueprintProject.uproject \
    -editor -game -makefile

Setting up Unreal Engine project files...

Setting up bundled DotNet SDK
Log file: /home/mamadou/.config/Epic/UnrealBuildTool/Log_GPF.txt
Project file formats specified via the command line will be ignored when
generating
project files from the editor and other engine tools.

Consider setting your desired IDE from the editor preferences window, or
modify your
BuildConfiguration.xml file with:

<?xml version="1.0" encoding="utf-8" ?>
<Configuration xmlns="https://www.unrealengine.com/BuildConfiguration">
  <ProjectFileGenerator>
    <Format>Make</Format>
  </ProjectFileGenerator>
</Configuration>


Generating Make project files:
Discovering modules, targets and source code for project...
Total execution time: 0.35 seconds
Directory '/path/to/MyBlueprintProject/MyBlueprintProject' is missing
'Source' folder.
```

For a C++ project, however, the project files will generate without any issues:

```
$ /path/to/UnrealEngine/Engine/Build/BatchFiles/Linux/GenerateProjectFiles.sh
\
    /path/to/MyCppProject/MyCppProject.uproject \
    -editor -game -makefile

Setting up Unreal Engine project files...

Setting up bundled DotNet SDK
Log file: /home/mamadou/.config/Epic/UnrealBuildTool/Log_GPF.txt
Project file formats specified via the command line will be ignored when
generating
project files from the editor and other engine tools.

Consider setting your desired IDE from the editor preferences window, or
modify your
BuildConfiguration.xml file with:

<?xml version="1.0" encoding="utf-8" ?>
<Configuration xmlns="https://www.unrealengine.com/BuildConfiguration">
  <ProjectFileGenerator>
    <Format>Make</Format>
  </ProjectFileGenerator>
</Configuration>


Generating Make project files:
Discovering modules, targets and source code for project...
Generating data for project indexing... 100%

Generating QueryTargets data for editor...
Total execution time: 2.98 seconds
```

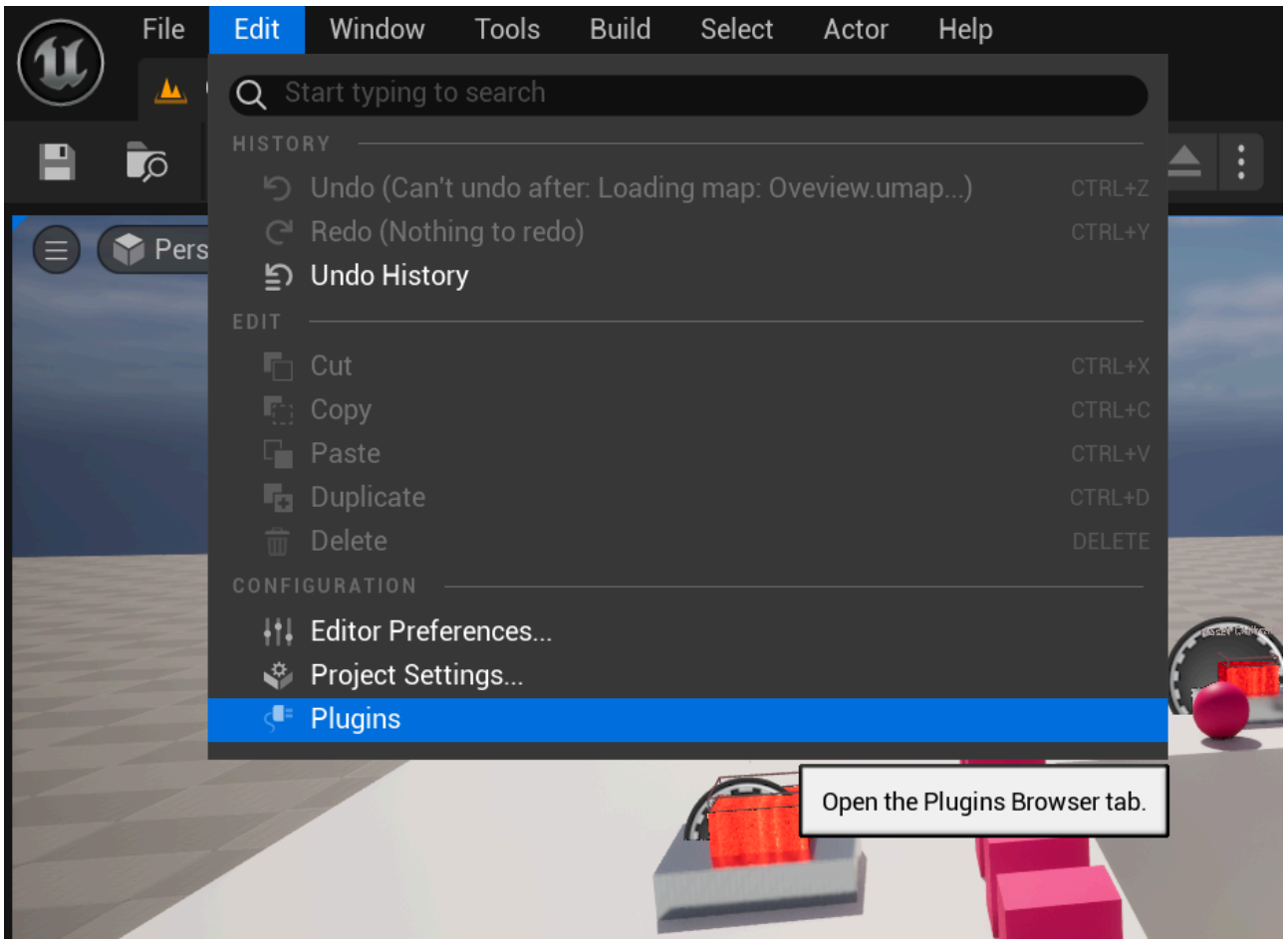
So, the workaround for Blueprint projects is to build the plugin inside a C++ project and then copy the `Plugin/SenseGlove` directory, which contains the built binary modules, to the corresponding directory in your Blueprint project.

```
$ /path/to/UnrealEngine/Engine/Build/BatchFiles/Linux/GenerateProjectFiles.sh
\
    /path/to/MyCppProject/MyCppProject.uproject \
    -editor -game -makefile
$ make MyCppProjectEditor -C /path/to/MyCppProject/
$ cp -vr \
    /path/to/MyCppProject/Plugins/SenseGlove \
    /path/to/MyBlueprintProject/Plugins/
$ /path/to/UnrealEngine/Engine/Binaries/Linux/UnrealEditor \
    /path/to/MyBlueprintProject/MyBlueprintProject.uproject
```

Enabling The SenseGlove Unreal Engine Plugin and Veirfying the Plugin Version

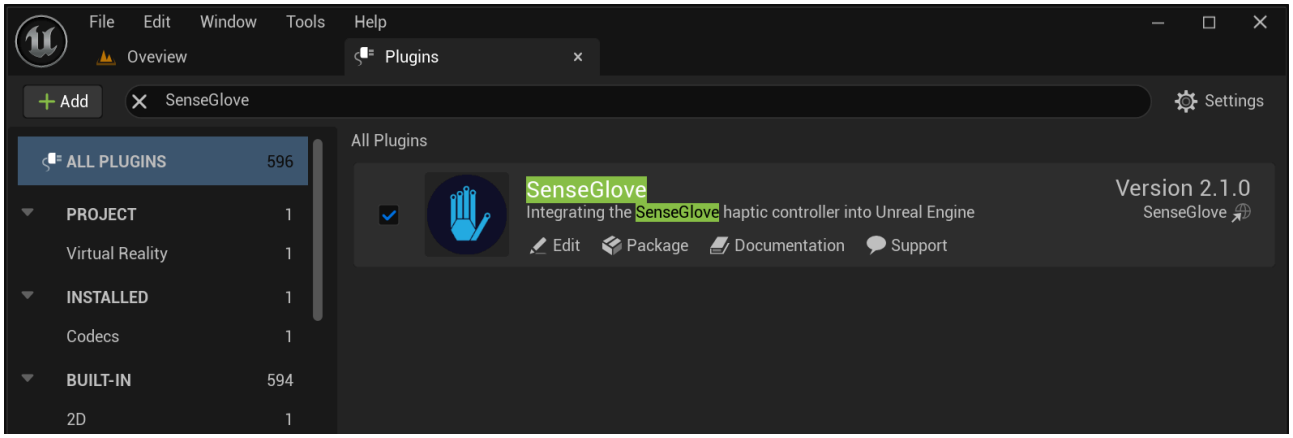
Enabling the SenseGlove Unreal Engine Plugin is a very simple and straightforward procedure. Furthermore, checking which version of the plugin your project is using may sometimes come in handy, especially if you have multiple versions of the plugin installed on different engine versions or various projects.

1. Inside the Unreal Editor for your project, select the `Plugins` from the `Edit` menu.

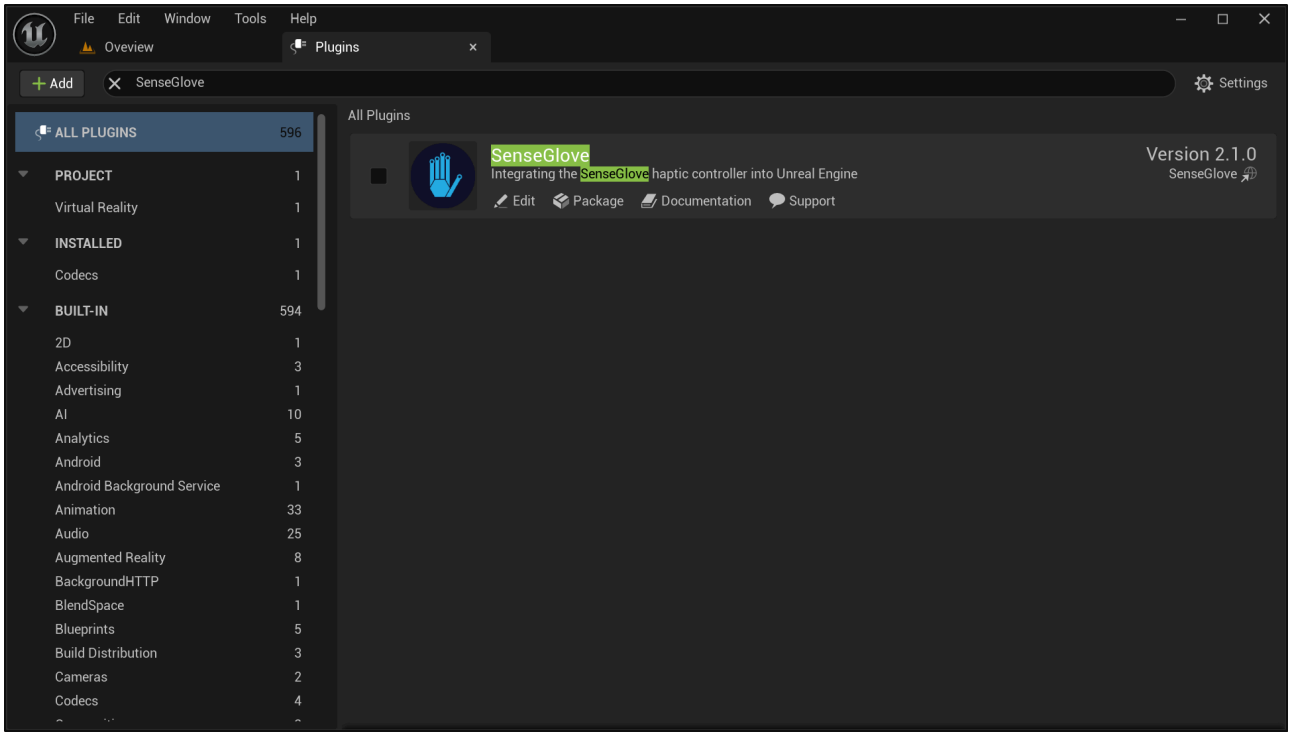


2. Once the plugin window/tab is open, start typing `SenseGlove` until you're able to spot the SenseGlove Unreal Engine Plugin. There you could find the plugin

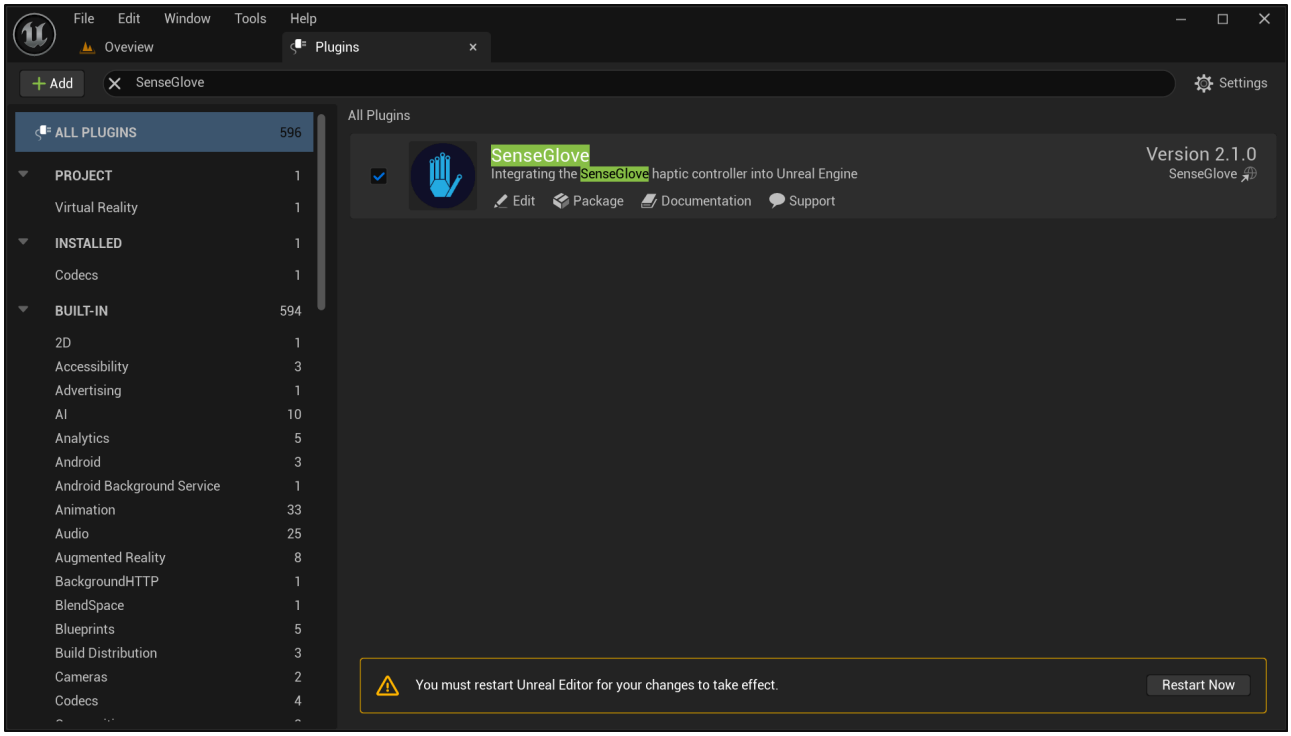
version, and other useful resources, such as the documentation website or support contact.



3. If the plugin is not enabled, it does not have the checkmark next to it.



4. It should be easy to click the checkmark and enable the plugin if that's not the case. Once the plugin is enabled, the Unreal Editor asks to be restarted. Click on the `Restart Now` button as this is mandatory to activate the plugin inside your project.



5. The source code for the plugin might be required to be rebuilt depending on how you have obtained and installed the plugin, usually the Unreal Editor lets you know and does this automatically. If it's required to build the plugin source, and it fails to do so, it usually suggests an alternative approach such as opening your regenerating the project files and rebuilding the project inside a C++ IDE. Once this is done the Editor for your projects re-opens and you can follow steps `1` and `2` in order to verify the plugin's version and availability inside your project.

Video Tutorial

A video demonstrating the same instructions in more detail is also available on the SenseGlove YouTube channel.

SenseGlove UE Tutorial: Finding your Plugin Version



SenseCom

SenseCom (short for SenseGlove Communications) is a background program that runs alongside your Unreal Engine application. Its primary function is to discover, and connect to SenseGlove devices on your system, exchanging data with them, much like a "SteamVR for Haptic Gloves." The SenseGlove Unreal Engine Plugin relies on SenseCom to communicate with any SenseGlove hardware.

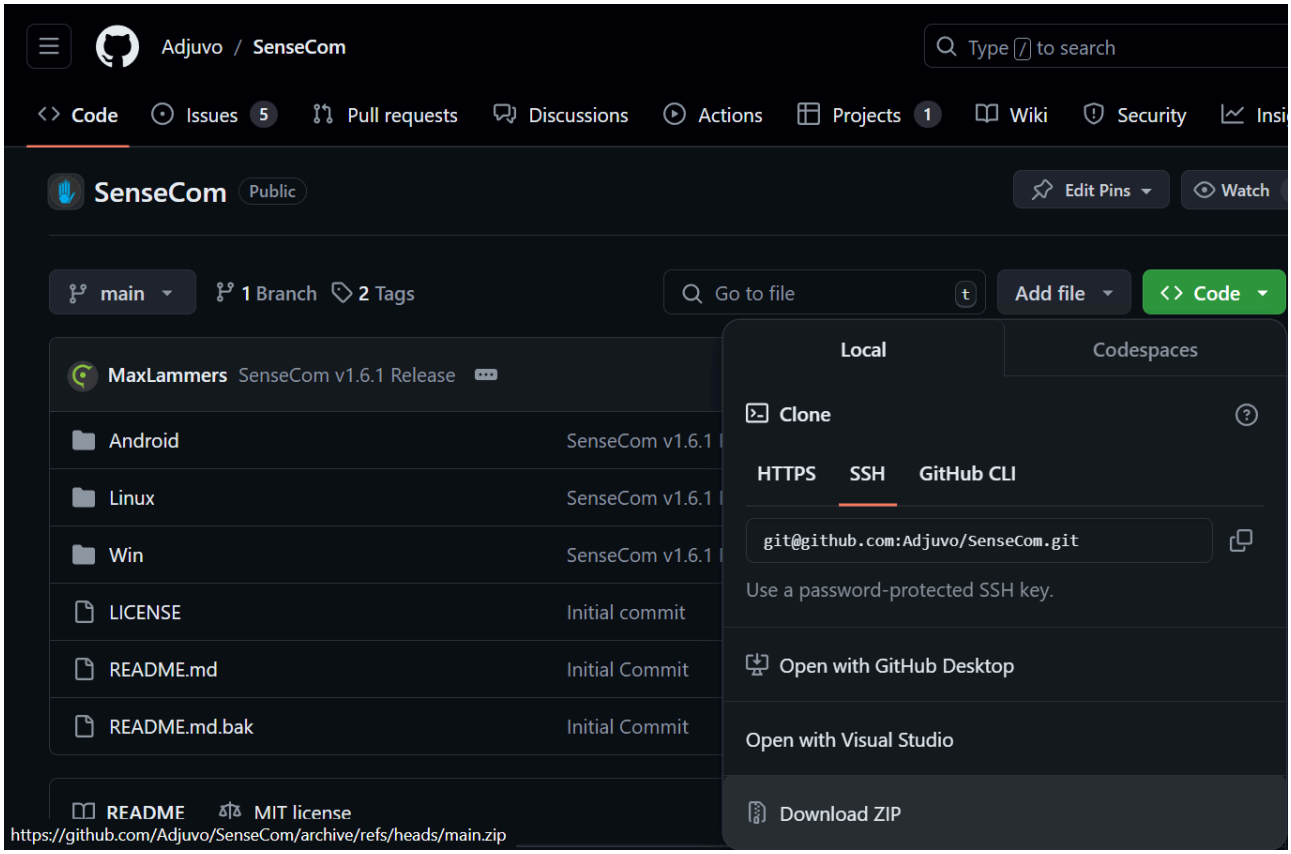
> **Note**
>
> SenseCom is required only for communication on Windows or Linux. For standalone Android devices, the communication functionality is embedded directly into your application.

> **Note**
>
> For more detailed information and troubleshooting, consult the SenseCom documentation page on SGDocs, please.

SenseCom on GNU/Linux

Follow these steps to quickly set up and run SenseCom on GNU/Linux:

1. First, obtain the SenseCom binaries from its GitHub repository.



2. Extract the SenseCom `.zip` file to a location on your computer.

```
$ unzip SenseCom-main.zip -d /some/path/
```

3. Navigate to the `SenseCom_Linux_Latest` folder containing the SenseCom binaries for GNU/Linux:

```
$ cd /some/path/SenseCom-main/Linux/SenseCom_Linux_Latest/
```

4. List the files and check the executable permissions for the main SenseCom binary, `SenseCom.x86_64` :

```
$ ls -ahl

total 20M
drwxr-xr-x 3 mamadou mamadou   5 Apr 10 11:24 .
drwxr-xr-x 3 mamadou mamadou   5 Apr 10 11:24 ..
drwxr-xr-x 7 mamadou mamadou  34 Apr 10 11:24 SenseCom_Data
-rw-r--r-- 1 mamadou mamadou 15K Apr 10 11:24 SenseCom.x86_64
-rw-r--r-- 1 mamadou mamadou 33M Apr 10 11:24 UnityPlayer.so
```

5. As seen above the `SenseCom.x86_64` binary does not have the executable permission. Run the following command to set the executable permission for all users:

```
$ chmod a+x SenseCom.x86_64
```

6. Veirfy the executable permission has been set on `SenseCom.x86_64` :

```
$ ls -l SenseCom.x86_64

-rwxr-xr-x 1 mamadou mamadou 14720 Apr 10 11:24 SenseCom.x86_64
```

7. Time to run the SenseCom executable:

```
$ ./SenseCom.x86_64

[UnityMemory] Configuration Parameters - Can be set up in boot.config
    "memorysetup-bucket-allocator-granularity=16"
    "memorysetup-bucket-allocator-bucket-count=8"
    "memorysetup-bucket-allocator-block-size=4194304"
    "memorysetup-bucket-allocator-block-count=1"
    "memorysetup-main-allocator-block-size=16777216"
    "memorysetup-thread-allocator-block-size=16777216"
    "memorysetup-gfx-main-allocator-block-size=16777216"
    "memorysetup-gfx-thread-allocator-block-size=16777216"
    "memorysetup-cache-allocator-block-size=4194304"
    "memorysetup-typetree-allocator-block-size=2097152"
    "memorysetup-profiler-bucket-allocator-granularity=16"
    "memorysetup-profiler-bucket-allocator-bucket-count=8"
    "memorysetup-profiler-bucket-allocator-block-size=4194304"
    "memorysetup-profiler-bucket-allocator-block-count=1"
    "memorysetup-profiler-allocator-block-size=16777216"
    "memorysetup-profiler-editor-allocator-block-size=1048576"
    "memorysetup-temp-allocator-size-main=4194304"
    "memorysetup-job-temp-allocator-block-size=2097152"
    "memorysetup-job-temp-allocator-block-size-background=1048576"
    "memorysetup-job-temp-allocator-reduction-small-platforms=262144"
    "memorysetup-temp-allocator-size-background-worker=32768"
    "memorysetup-temp-allocator-size-job-worker=262144"
    "memorysetup-temp-allocator-size-preload-manager=262144"
    "memorysetup-temp-allocator-size-nav-mesh-worker=65536"
    "memorysetup-temp-allocator-size-audio-worker=65536"
    "memorysetup-temp-allocator-size-cloud-worker=32768"
    "memorysetup-temp-allocator-size-gfx=262144"
Loading in SingleInstance mode
```

8. If you have already paired any glove with your system, SenseCom should recognize and connect to your glove(s) shortly. If not, please follow the instructions on How to connect to Nova gloves using Blueman Bluetooth Manager or How to connect to Nova gloves using Command-line.



> **Note**
>
> For more detailed information and troubleshooting, consult the SenseCom documentation page on SGDocs, please.

Connect to Nova gloves using Blueman Bluetooth Manager

Follow these steps to pair a Nova glove with your PC on GNU/Linux usng the Blueman Bluetooth Manager:

1. Install Blueman Bluetooth Manager on your Linux distribution using the appropriate package manager:

```
# Gentoo
$ emerge -atuv net-wireless/blueman

# Arch, Manjaro
$ sudo pacman -S blueman

# CentOS, Fedora, AlmaLinux, Rocky Linux
$ sudo dnf install blueman

# CentOS/RHEL
$ sudo yum install epel-release
$ sudo yum install blueman

# Debian, Ubuntu
$ sudo apt install blueman

# openSUSE
sudo zypper install blueman

# Solus
$ sudo eopkg install blueman

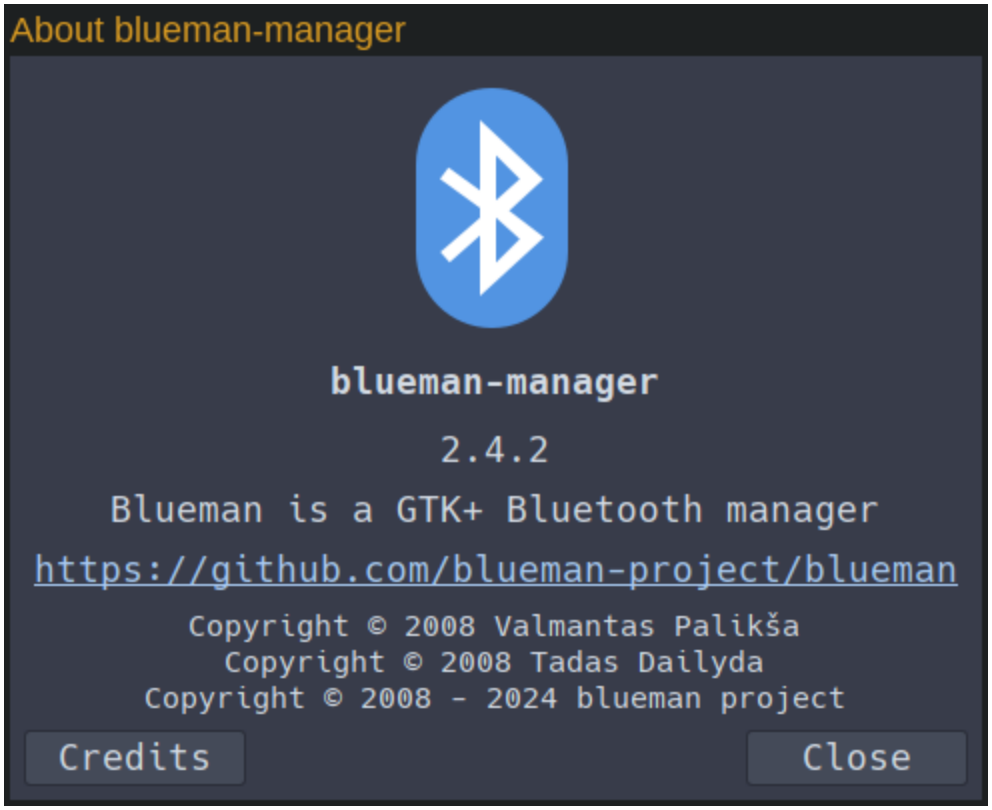
# Void Linux
$ sudo xbps-install -S blueman
```

> **Important**
>
> To properly set up the Bluetooth stack on your Linux distribution, additional steps may be required. For example, on Gentoo and Arch consult each distribution's official guide.

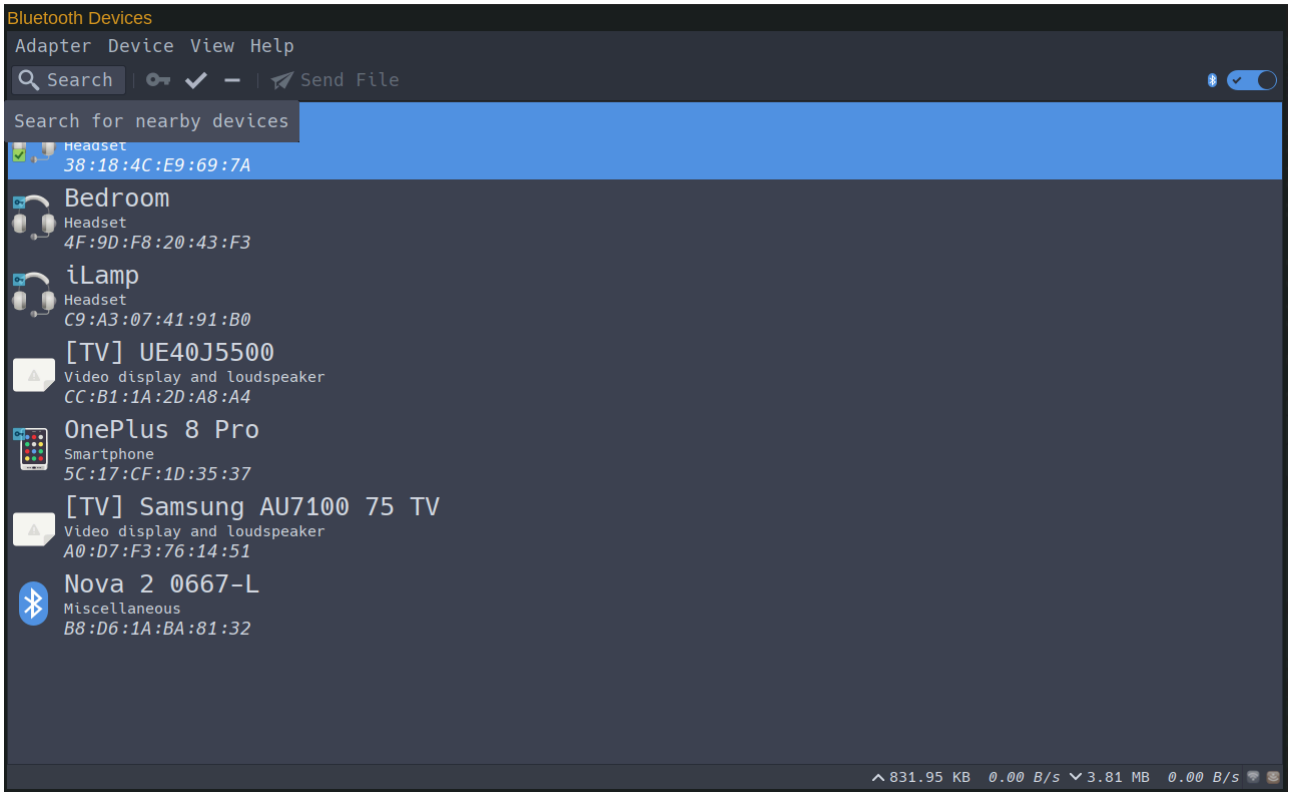
2. Ensure any glove you would like to pair with and connect to your system is not paired, or connected to any other device, such as another PC or VR headset.

3. Make sure the glove is turned on.

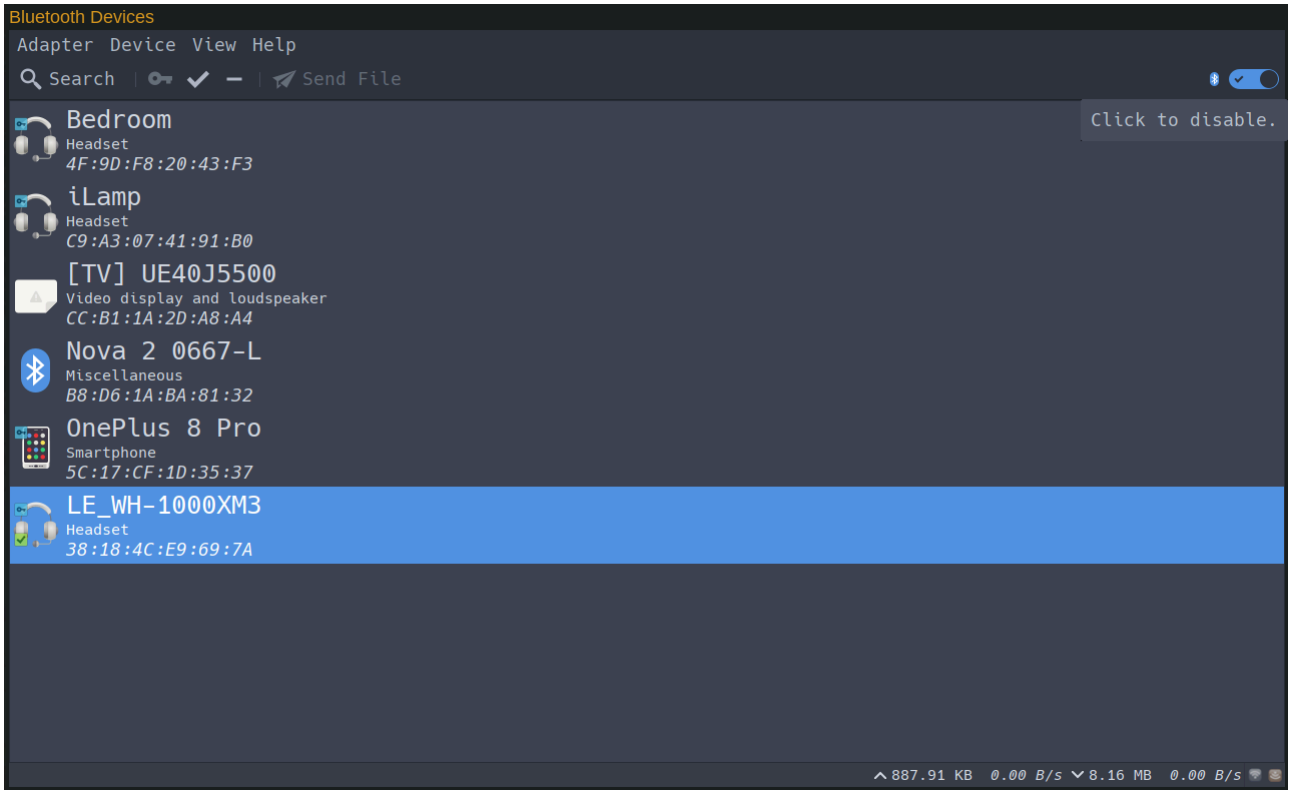
4. Start the Blueman Bluetooth Manager and verify you have a recent version installed by selecting `Help` > `About` from the application's menu.



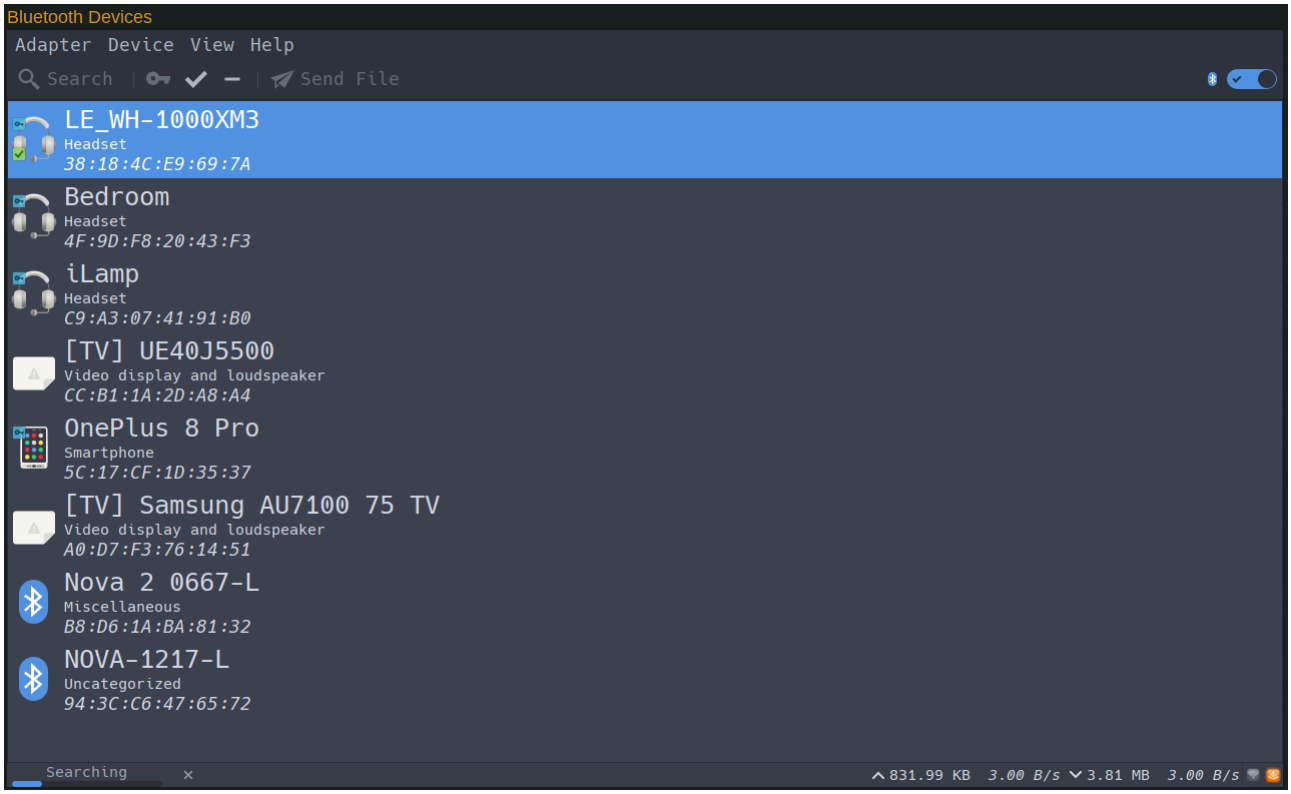
5. If you don't see your glove, click the `Search` button on the toolbar or select `Adapter` > `Search` from the application's menu to look for new Bluetooth devices.



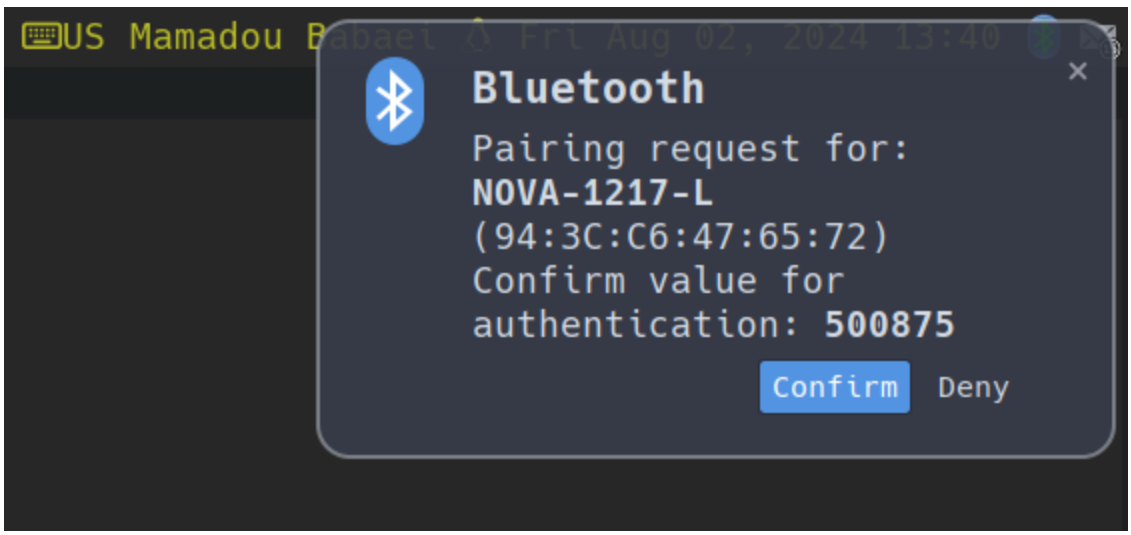
> **Important**
>
> Before starting the search operation, ensure that your PC's Bluetooth controller is turned on by verifying its status on the right side of the toolbar next to the Bluetooth logo. If disabled, the Search button will be grayed out.



6. A progress bar will appear on the application's status bar. If a new device is found, it will be listed in the main device list area.

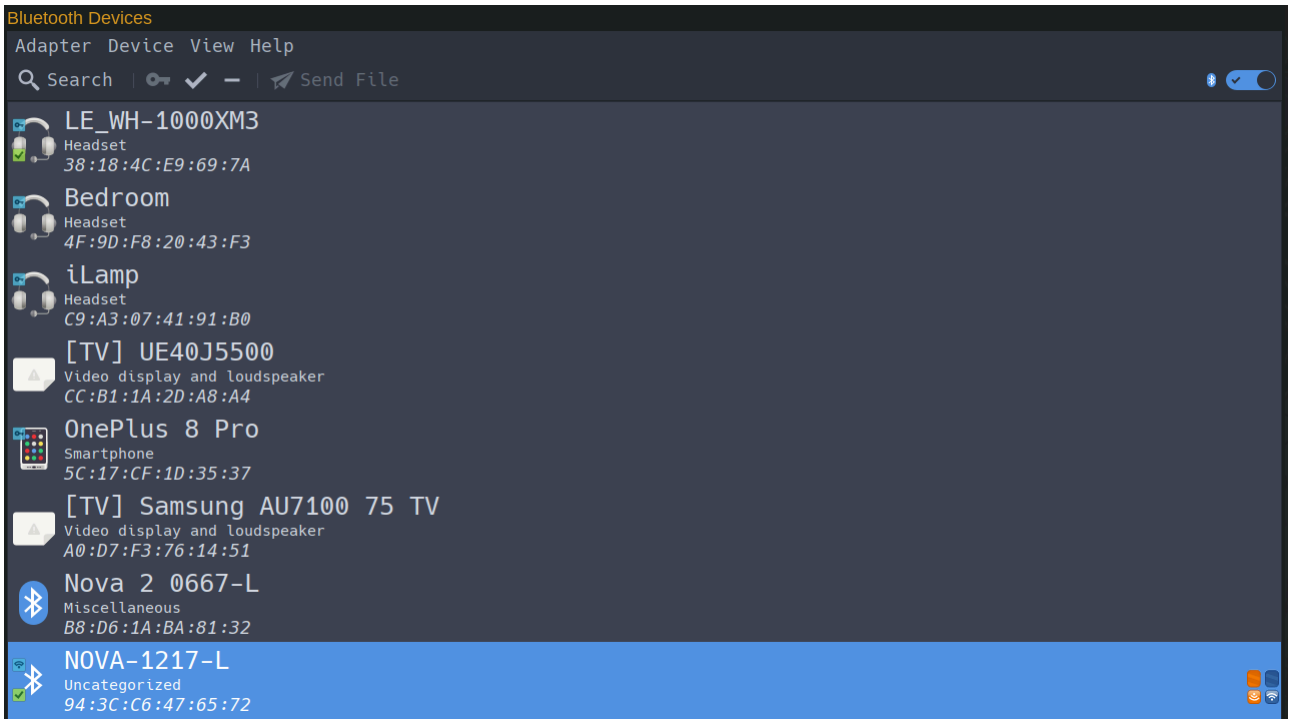


7. Once the glove is found, click on it to select it.
8. Either right-click on the device, or go to the `Device` menu, then choose `Pair`.
9. Blueman will prompt you to pair the glove with a notification. Click `Confirm` to proceed.



9. After pairing, either right-click on the device again, or go to the `Device` menu, then choose `Trust`.

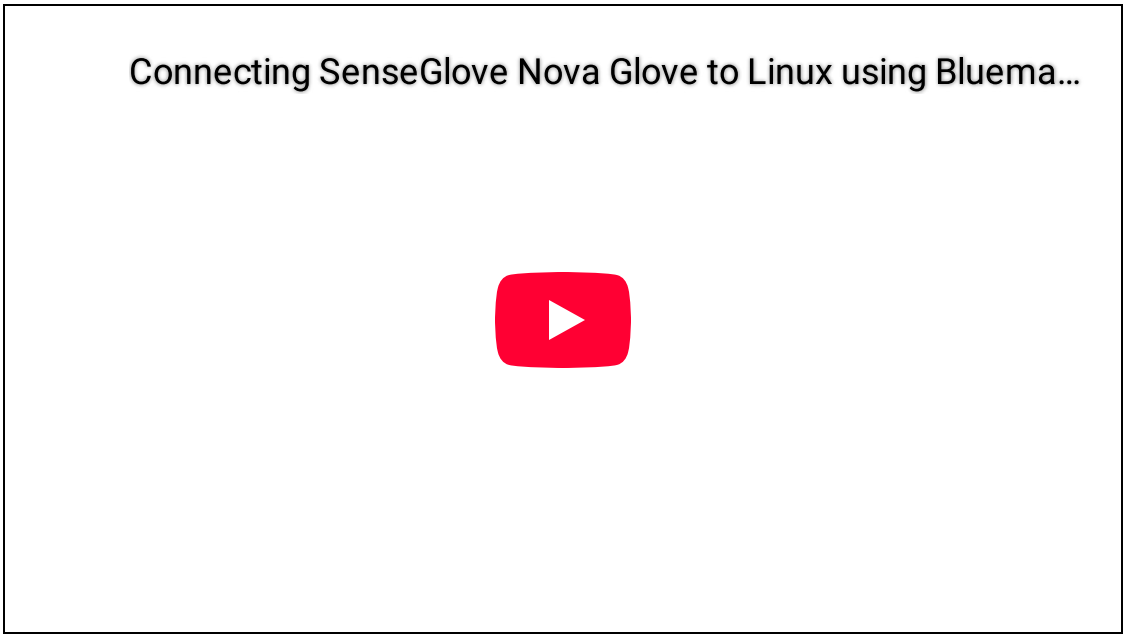
10. If everything has been successful, the key icon indicates successful pairing, and the checkmark confirms the device is trusted.



11. Follow the SenseCom on GNU/Linux instructions and you should be able to successfully connect to the newly paired glove from SenseCom.

Video Tutorial

There is also a video tutorial demonstrating how to connect to Nova gloves on GNU/Linux using Blueman Bluetooth Manager.


Connecting SenseGlove Nova Glove to Linux using Bluema...

Connect to Nova gloves using Command-line

Follow these steps to pair a Nova glove to your PC on GNU/Linux usng command-line and Bluez:

1. Some Linux distributions include BlueZ in their default installation. If yours doesn't, install it using the appropriate package manager:

```
# Gentoo
$ emerge -atuv net-wireless/bluez

# Arch, Manjaro
$ sudo pacman -S bluez

# CentOS, Fedora, AlmaLinux, Rocky Linux
$ sudo dnf install bluez

# CentOS/RHEL
$ sudo yum install bluez

# Debian, Ubuntu
$ sudo apt install bluez

# openSUSE
sudo zypper install bluez

# Solus
$ sudo eopkg install bluez

# Void Linux
$ sudo xbps-install -S bluez
```

> **Important**
>
> To properly set up the Bluetooth stack on your Linux distribution, additional steps may be required. For example, on Gentoo and Arch consult each distribution's official guide.

2. Run the following command to ensure that BlueZ is installed and check your `bluetoothctl` version:

```
bluetoothctl version
Version 5.77
```

3. Ensure that the `bluetooth` service is started and running. For example, on Gentoo Linux:

```
$  rc-service bluetooth start
```

You might see one of these outputs based on whether it's already running or not:

```
 * Starting bluetooth ...
# or
 * WARNING: bluetooth has already been started
```

4. Ensure any glove you would like to pair with and connect to your system is not paired, or connected to any other device, such as another PC or VR headset.
5. Make sure the glove is turned on.
6. Use `bluetoothctl list` or `bluetoothctl show` command to extract your PC's Bluetooth Controller MAC Address which is useful for later on:

```
$ bluetoothctl list

Controller CC:15:31:90:69:87 BlueZ 5.77 [default]

$ bluetoothctl show
Controller CC:15:31:90:69:87 (public)
    Manufacturer: 0x0002 (2)
    Version: 0x0b (11)
    Name: BlueZ 5.77
    Alias: BlueZ 5.77
    Class: 0x007c010c (8126732)
    Powered: yes
    PowerState: on
    Discoverable: no
    DiscoverableTimeout: 0x0000003c (60)
    Pairable: no
    UUID: Message Notification Se.. (00001133-0000-1000-8000-00805f9b34fb)
    UUID: A/V Remote Control        (0000110e-0000-1000-8000-00805f9b34fb)
    UUID: OBEX Object Push          (00001105-0000-1000-8000-00805f9b34fb)
    UUID: Message Access Server     (00001132-0000-1000-8000-00805f9b34fb)
    UUID: PnP Information           (00001200-0000-1000-8000-00805f9b34fb)
    UUID: IrMC Sync                 (00001104-0000-1000-8000-00805f9b34fb)
    UUID: Headset                   (00001108-0000-1000-8000-00805f9b34fb)
    UUID: A/V Remote Control Target (0000110c-0000-1000-8000-00805f9b34fb)
    UUID: Generic Attribute Profile (00001801-0000-1000-8000-00805f9b34fb)
    UUID: Phonebook Access Server   (0000112f-0000-1000-8000-00805f9b34fb)
    UUID: Audio Sink                (0000110b-0000-1000-8000-00805f9b34fb)
    UUID: Device Information        (0000180a-0000-1000-8000-00805f9b34fb)
    UUID: Generic Access Profile    (00001800-0000-1000-8000-00805f9b34fb)
    UUID: Handsfree Audio Gateway   (0000111f-0000-1000-8000-00805f9b34fb)
    UUID: Audio Source              (0000110a-0000-1000-8000-00805f9b34fb)
    UUID: OBEX File Transfer        (00001106-0000-1000-8000-00805f9b34fb)
    Modalias: usb:v1D6Bp0246d054D
    Discovering: no
    Roles: central
    Roles: peripheral
Advertising Features:
    ActiveInstances: 0x00 (0)
    SupportedInstances: 0x0c (12)
    SupportedIncludes: tx-power
    SupportedIncludes: appearance
    SupportedIncludes: local-name
    SupportedSecondaryChannels: 1M
    SupportedSecondaryChannels: 2M
    SupportedCapabilities.MinTxPower: 0xfffffffe (-34)
    SupportedCapabilities.MaxTxPower: 0x0007 (7)
    SupportedCapabilities.MaxAdvLen: 0xfb (251)
```

```
    SupportedCapabilities.MaxScnRspLen: 0xfb (251)
    SupportedFeatures: CanSetTxPower
    SupportedFeatures: HardwareOffload
```

7. Ensure the controller is powered on:

```
$ bluetoothctl power on

Changing power on succeeded
```

8. Enable the agent to listen for Bluetooth events that require user interaction, such as pairing requests and managing device authorizations:

```
$ bluetoothctl agent on
```

9. Set the current agent as the default agent:

```
$ bluetoothctl default-agent

No agent is registered
```

10. Set the controller to be discoverable for `180` seconds:

```
$ bluetoothctl discoverable on

bluetoothctl discoverable on
hci0 new_settings: powered connectable ssp br/edr le secure-conn wide-band-
speech
hci0 new_settings: powered connectable discoverable ssp br/edr le secure-conn
wide-band-speech
Changing discoverable on succeeded
```

> **Note**
>
> To change the default discoverable timeout, you can set it manually using the `bluetoothctl discoverable-timeout` command.

```
$ bluetoothctl discoverable-timeout 300

Changing discoverable-timeout 300 succeeded
```

11. Then, make the controller pairable as well:

```
$ bluetoothctl pairable on

hci0 new_settings: powered connectable discoverable bondable ssp br/edr le
secure-conn wide-band-speech
Changing pairable on succeeded
```

12. Begin scanning for devices:

```
$ bluetoothctl scan on

SetDiscoveryFilter success
```

13. After a few seconds, list the discovered devices:

bluetoothctl devices

```
Device 78:D2:52:42:33:2F 78-D2-52-42-33-2F
Device 94:3C:C6:47:65:72 NOVA-1217-L
Device AC:F1:08:37:9F:93 LG DSN7CY(93)
Device 70:D6:10:9D:73:8F 70-D6-10-9D-73-8F
Device 7F:2C:8C:8D:09:9F 7F-2C-8C-8D-09-9F
Device F9:56:4B:86:1E:13 F9-56-4B-86-1E-13
Device C9:A3:07:41:91:B0 iLamp
Device 4F:9D:F8:20:43:F3 Bedroom
Device CC:B1:1A:2D:A8:A4 [TV] UE40J5500
Device A0:D7:F3:76:14:51 [TV] Samsung AU7100 75 TV
Device 5C:17:CF:1D:35:37 OnePlus 8 Pro
Device E2:F8:03:F6:D8:CB E2-F8-03-F6-D8-CB
Device 38:18:4C:E9:69:7A LE_WH-1000XM3
Device B8:D6:1A:BA:81:32 Nova 2 0667-L
```

> **Note**
>
> If your device is not listed yet, you can run this command multiple times as `bluetoothctl` continues the device discovery in the background.

14. Use the following command to pair with the discoved glove:

```
$ bluetoothctl pair GLOVE_MAC_ADDRESS
```

For example:

```
$ bluetoothctl pair 94:3C:C6:47:65:72

Attempting to pair with 94:3C:C6:47:65:72
[CHG] Device 94:3C:C6:47:65:72 Connected: yes
[CHG] Device 94:3C:C6:47:65:72 Bonded: yes
[CHG] Device 94:3C:C6:47:65:72 UUIDs: 00001101-0000-1000-8000-00805f9b34fb
[CHG] Device 94:3C:C6:47:65:72 ServicesResolved: yes
[CHG] Device 94:3C:C6:47:65:72 Paired: yes
Pairing successful
```

> **Note**
>
> If you encounter the `Failed to pair: org.bluez.Error.AuthenticationFailed` error message, it might be misleading. Check if there is a line with the glove's MAC address followed by `Connected: yes`, which indicates that the connection was actually successful.

```
Attempting to pair with 94:3C:C6:47:65:72
[CHG] Device 94:3C:C6:47:65:72 Connected: yes
Failed to pair: org.bluez.Error.AuthenticationFailed
```

15. Mark the device as trusted by issuing the following command:

```
$ bluetoothctl trust GLOVE_MAC_ADDRESS
```

For example:

```
$ bluetoothctl trust 94:3C:C6:47:65:72

[CHG] Device 94:3C:C6:47:65:72 Trusted: yes
Changing 94:3C:C6:47:65:72 trust succeeded
```

16. Attempt to connect to the glove again:

```
$ bluetoothctl connect GLOVE_MAC_ADDRESS
```

For example:

```
$ bluetoothctl connect 94:3C:C6:47:65:72

Attempting to connect to 94:3C:C6:47:65:72
[CHG] Device 38:18:4C:E9:69:7A RSSI: 0xffffffd0 (-48)
[CHG] Device 94:3C:C6:47:65:72 Connected: yes
[CHG] Device 94:3C:C6:47:65:72 UUIDs: 00001101-0000-1000-8000-00805f9b34fb
[CHG] Device 94:3C:C6:47:65:72 ServicesResolved: yes
Failed to connect: org.bluez.Error.NotAvailable br-connection-profile-
unavailable
```

> **Note**
>
> Again, the error message may be misleading. The connection is often successful despite the error.

17. If desired, you can extract some information from the glove using:

```
$ bluetoothctl info GLOVE_MAC_ADDRESS
```

For example:

```
bluetoothctl info 94:3C:C6:47:65:72
Device 94:3C:C6:47:65:72 (public)
    Name: NOVA-1217-L
    Alias: NOVA-1217-L
    Class: 0x00001f00 (7936)
    Paired: yes
    Bonded: yes
    Trusted: yes
    Blocked: no
    Connected: yes
    LegacyPairing: no
    UUID: Serial Port               (00001101-0000-1000-8000-00805f9b34fb)
```

18. Create an RFCOMM device:

```
$ sudo rfcomm connect /dev/rfcommX GLOVE_MAC_ADDRESS CHANNEL_NUMBER
```

For example:

```
$ sudo rfcomm connect /dev/rfcomm0 94:3C:C6:47:65:72 1

Connected /dev/rfcomm0 to 94:3C:C6:47:65:72 on channel 1
Press CTRL-C for hangup
```

> **Note**
>
> The `rfcomm` command requires root permision, so it must be run with `sudo`.

> **Tip**
>
> To determine the channel number, run the following command:

```
$ sdptool browse GLOVE_MAC_ADDRESS
```

```
$ sdptool browse 94:3C:C6:47:65:72
Browsing 94:3C:C6:47:65:72 ...
Service Name: SPP_SERVER
Service RecHandle: 0x10000
Service Class ID List:
  "Serial Port" (0x1101)
Protocol Descriptor List:
  "L2CAP" (0x0100)
  "RFCOMM" (0x0003)
    Channel: 1
Profile Descriptor List:
  "Serial Port" (0x1101)
    Version: 0x010
```

> **Note**
>
> If you have more than one glove or in general multiple serial Bluetooth devices connected to your device connected to your PC, then `/dev/rfcomm0` may already be allocated to another device. In that case, increment the number until finding

> a free rfcomm device. You can query the existing rfcomm devices using the command: `ls /dev/rfcomm*`.

19. Follow the SenseCom on GNU/Linux instructions and you should be able to successfully connect to the newly paired glove from SenseCom.
20. Once the SenseCom is closed and we are done with the gloves, we can disconnect the gloves using:

```
$ bluetoothctl disconnect ${SG_DEVICE}
$ sudo rfcomm release ${SG_RFCOMM}
```

For example:

```
$ bluetoothctl disconnect 94:3C:C6:47:65:72
$ sudo rfcomm release /dev/rfcomm0
```

> **Note**
>
> Again, the `rfcomm` command requires elevated permissions, so it must be run with the `sudo` command.

Scripts to Easily Connect and Disconnect from a Glove

You can automate the above tedious process using scripts for connecting and disconnecting gloves.

`sg-connect.sh`:

```
#!/usr/bin/env sh

CTRL_DEVICE="YOUR_BLUETOOTH_CONTROLLER_MAC_ADDRESS"
SG_DEVICE="YOUR_SENSEGLOVE_MAC_ADDRESS"
SG_RFCOMM="/dev/rfcomm0"

bluetoothctl pairable on
bluetoothctl discoverable on
bluetoothctl pair ${SG_DEVICE}
bluetoothctl trust ${SG_DEVICE}
bluetoothctl connect ${SG_DEVICE}
rfcomm connect ${SG_RFCOMM} ${SG_DEVICE} 1 &
```

sg-disconnect.sh：

```
#!/usr/bin/env sh

SG_DEVICE="YOUR_SENSEGLOVE_MAC_ADDRESS"
SG_RFCOMM="/dev/rfcomm0"

bluetoothctl disconnect ${SG_DEVICE}
rfcomm release ${SG_RFCOMM}
```

Example Scripts for a Left-Handed Glove

```
$ cat sg-connect-left.sh

#!/usr/bin/env sh

CTRL_DEVICE="CC:15:31:90:69:87"
SG_DEVICE="94:3C:C6:47:65:72"
SG_RFCOMM="/dev/rfcomm0"

bluetoothctl pairable on
bluetoothctl discoverable on
bluetoothctl pair ${SG_DEVICE}
bluetoothctl trust ${SG_DEVICE}
bluetoothctl connect ${SG_DEVICE}
rfcomm connect ${SG_RFCOMM} ${SG_DEVICE} 1 &

$ cat sg-disconnect-left.sh

#!/usr/bin/env sh

SG_DEVICE="94:3C:C6:47:65:72"
SG_RFCOMM="/dev/rfcomm0"

bluetoothctl disconnect ${SG_DEVICE}
rfcomm release ${SG_RFCOMM}

# Set the executable permissions for all users:
$ chmod a+x sg-connect-left.sh
$ chmod a+x sg-disconnect-left.sh

# Before running SenseCom:

$ sudo ./sg-connect-left.sh

Password:

Changing pairable on succeeded
hci0 new_settings: powered connectable bondable ssp br/edr le secure-conn
wide-band-speech
hci0 new_settings: powered connectable discoverable bondable ssp br/edr le
secure-conn wide-band-speech
Changing discoverable on succeeded
Attempting to pair with 94:3C:C6:47:65:72
Failed to pair: org.bluez.Error.AlreadyExists
Changing 94:3C:C6:47:65:72 trust succeeded
Attempting to connect to 94:3C:C6:47:65:72
```

```
hci0 94:3C:C6:47:65:72 type BR/EDR connected eir_len 18
[CHG] Device 94:3C:C6:47:65:72 Connected: yes
[CHG] Device 94:3C:C6:47:65:72 ServicesResolved: yes
Failed to connect: org.bluez.Error.NotAvailable br-connection-profile-
unavailable

# Run SenseCom in between!

# Once SenseCom is closed:

$ sudo ./sg-disconnect-left.sh

sudo ./sg-disconnect-left.sh

Password:

Attempting to disconnect from 94:3C:C6:47:65:72
hci0 94:3C:C6:47:65:72 type BR/EDR disconnected with reason 2
[CHG] Device 94:3C:C6:47:65:72 ServicesResolved: no
Successful disconnected
Can't release device: No such device
```

Video Tutorial

There is also a video tutorial demonstrating how to connect to Nova gloves on GNU/Linux using the command line.

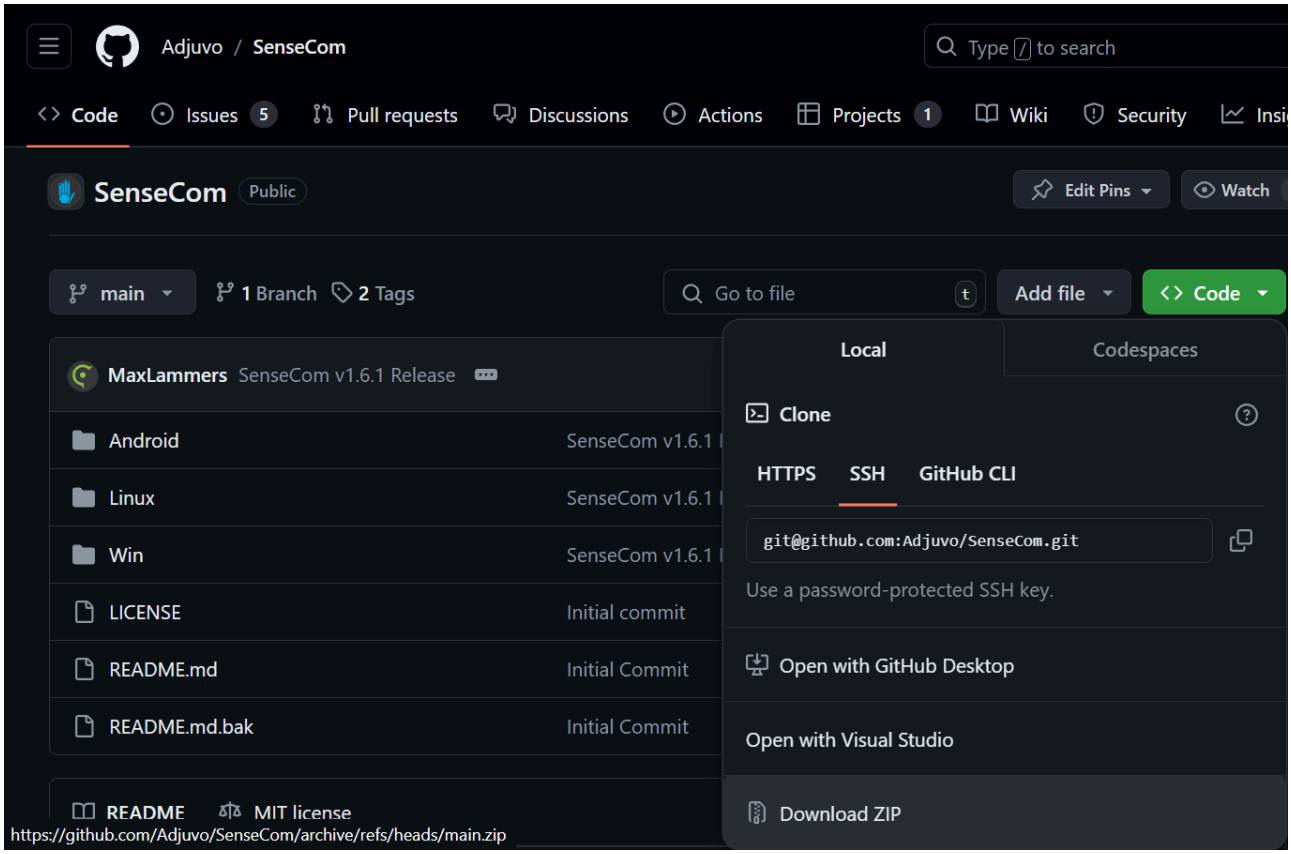
SenseGlove Tutorial | Connecting to Nova Gloves on Linux



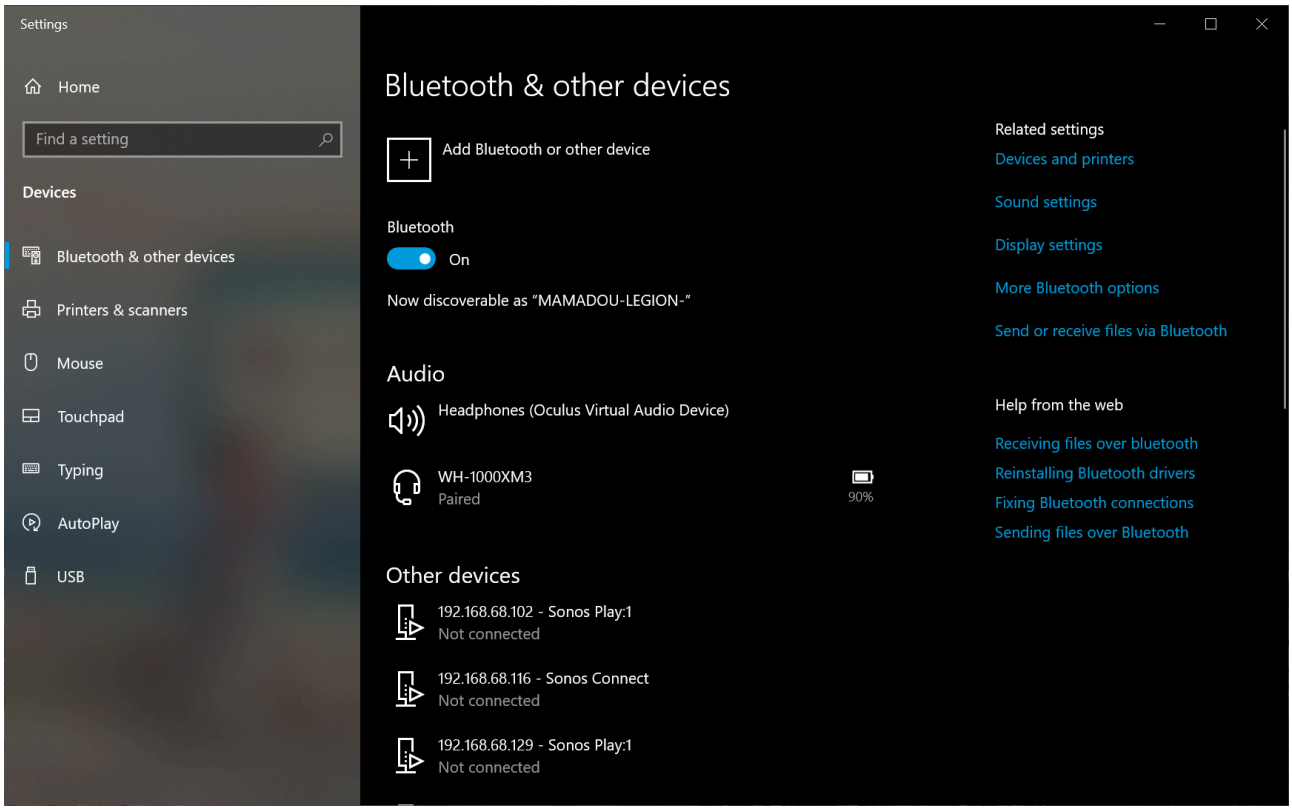
SenseCom on Microsoft Windows

Follow these steps to quickly set up and run SenseCom on Microsoft Windows:

1. First, obtain the SenseCom binaries from its GitHub repository.



2. Extract the SenseCom `.zip` file to a location on your computer after downloading it.
3. Ensure any glove you would like to pair with and connect to your system is not paired, or connected to any other device, such as another PC or VR headset.
4. Make sure the glove is powered on.
5. Access Windows Bluetooth Settings by navigating to `Settings > Devices > Bluetooth & other devices`.

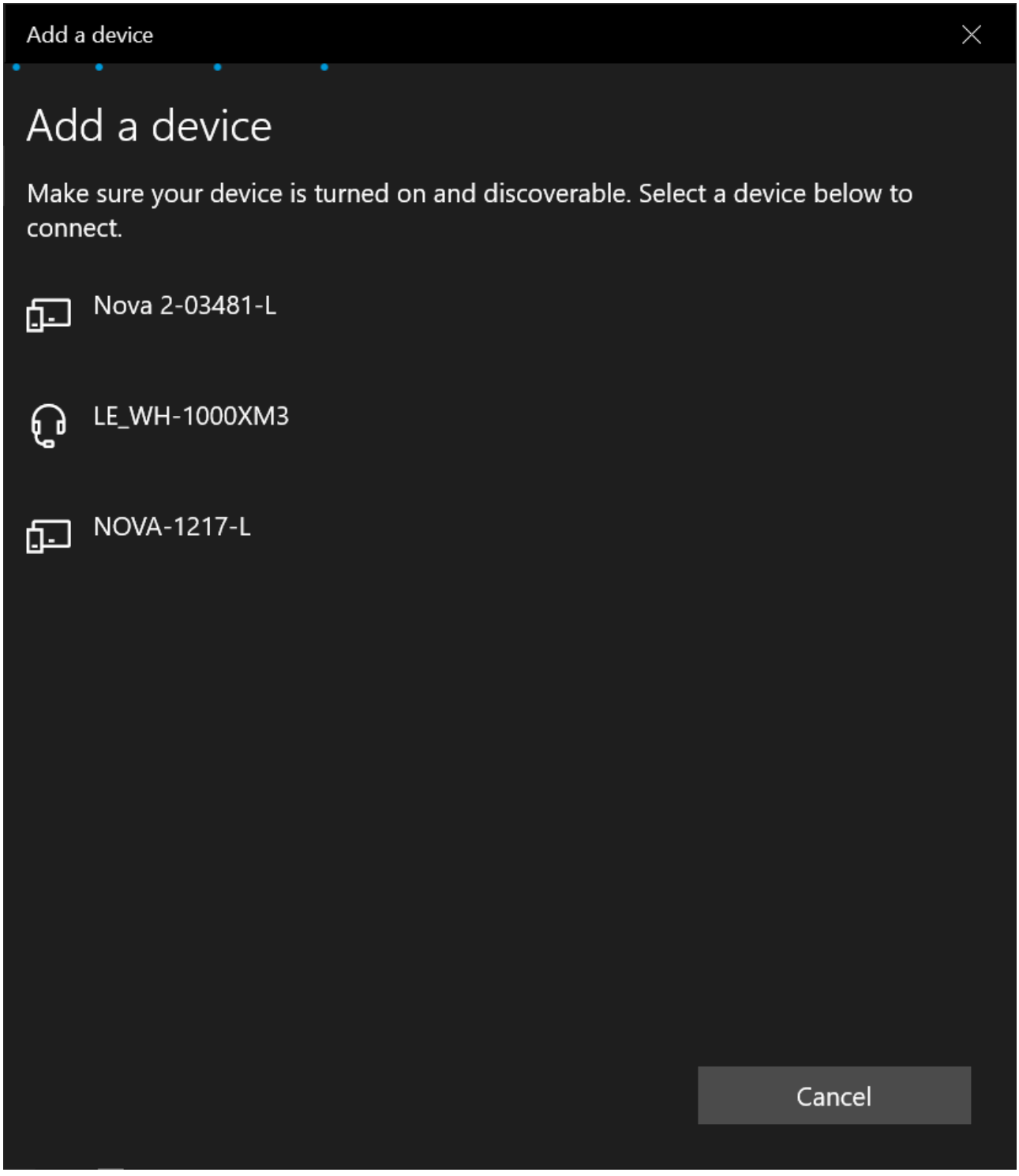


6. Click on `Add Bluetooth or other devices`.

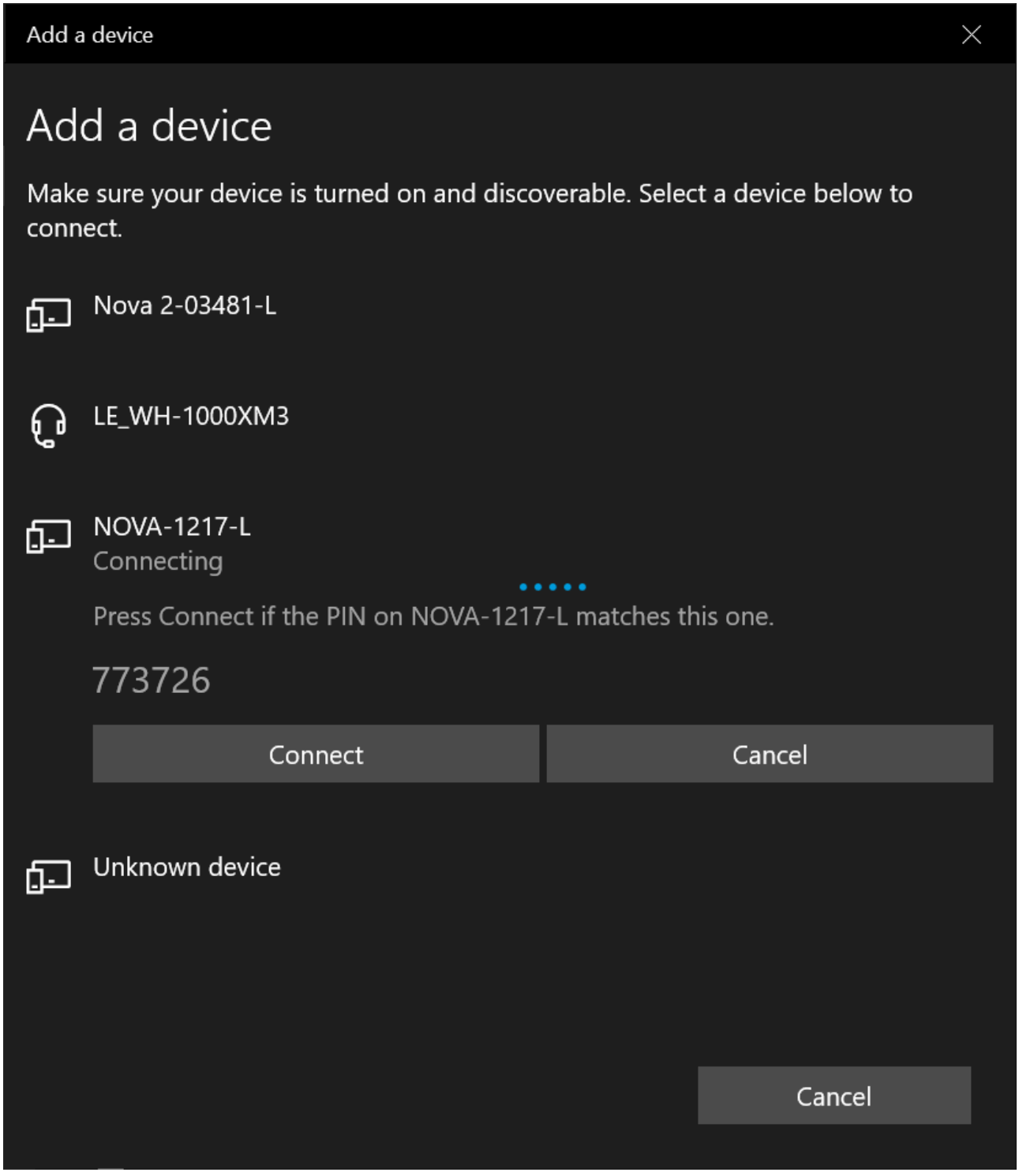
7. In the new window click on `Bluetooth`.



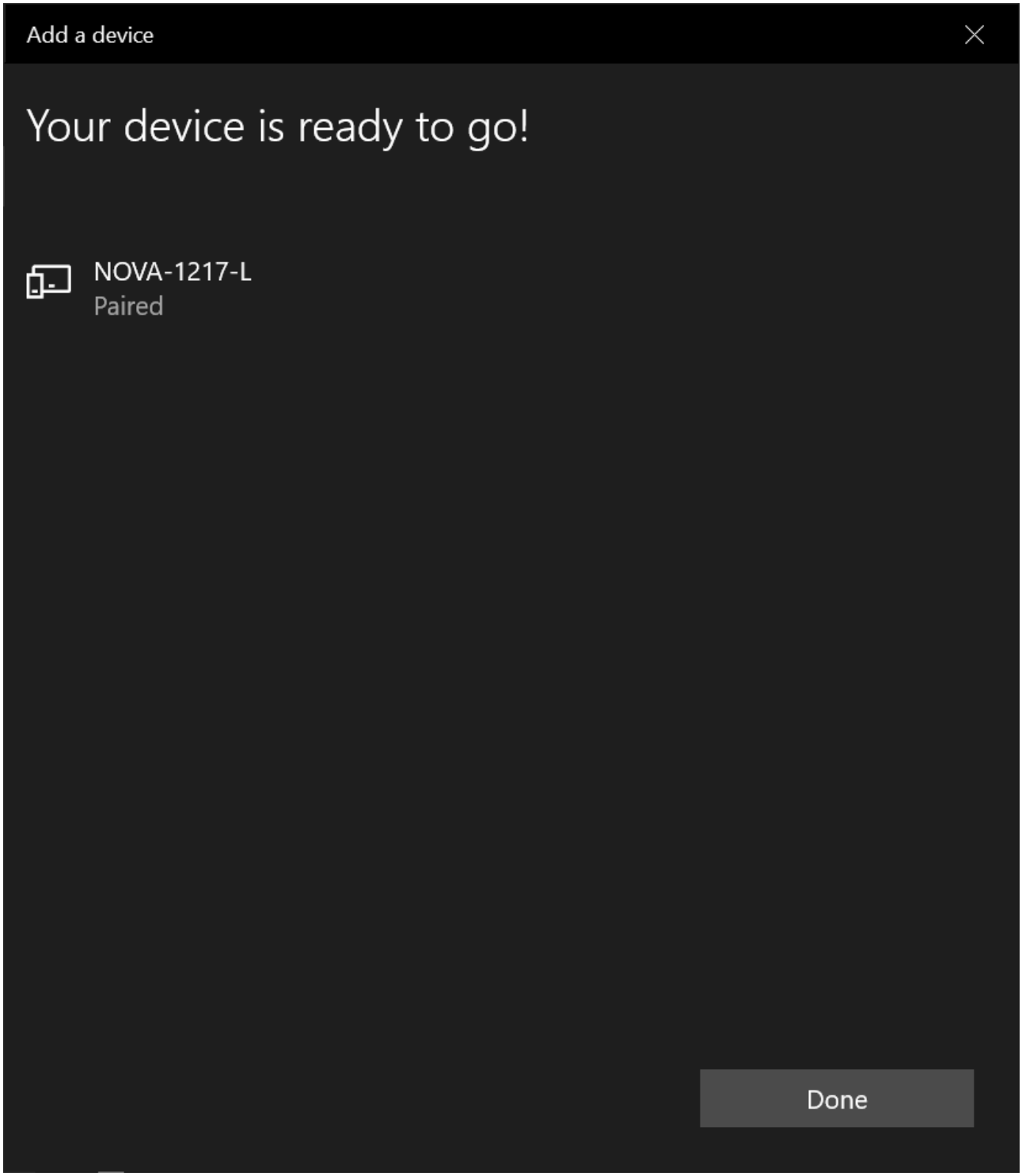
8. Wait for the glove to be discovered, then click on it.



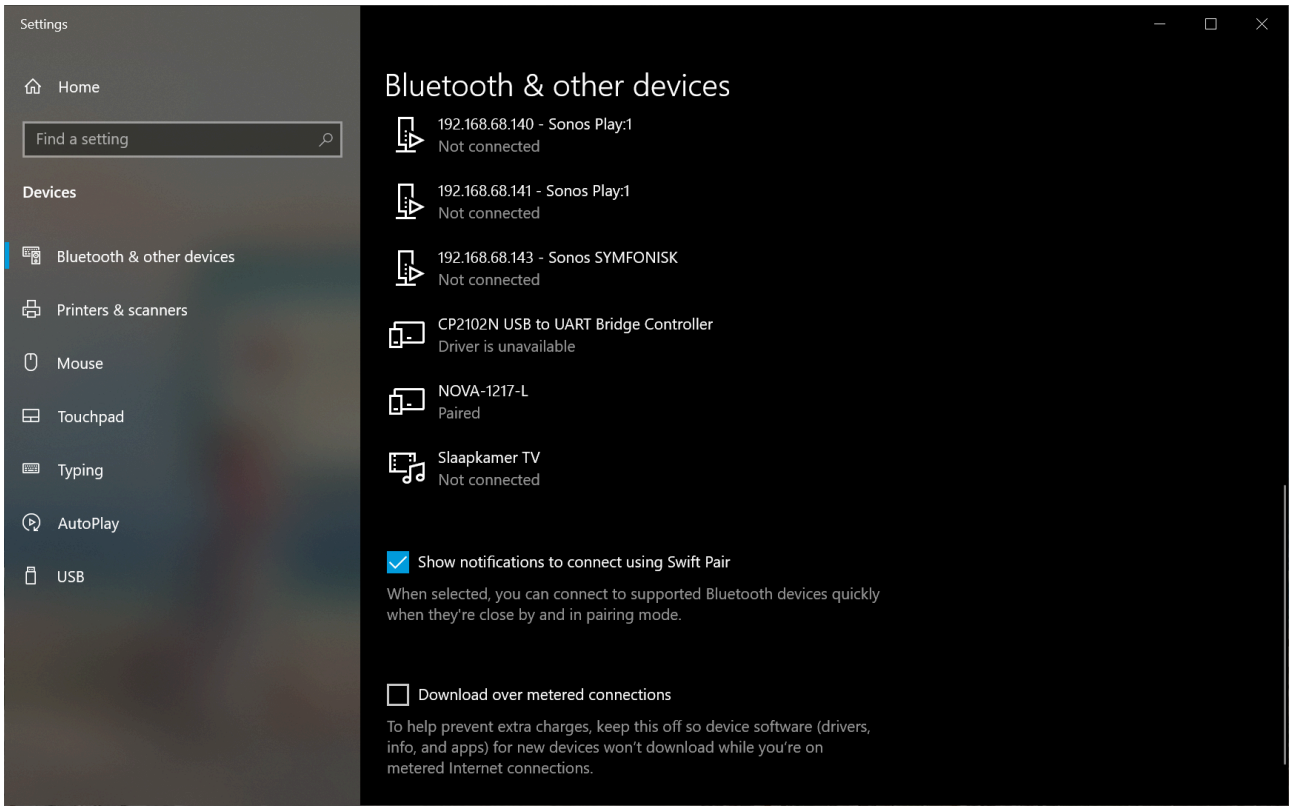
9. Click `Connect` to connect and pair the glove.



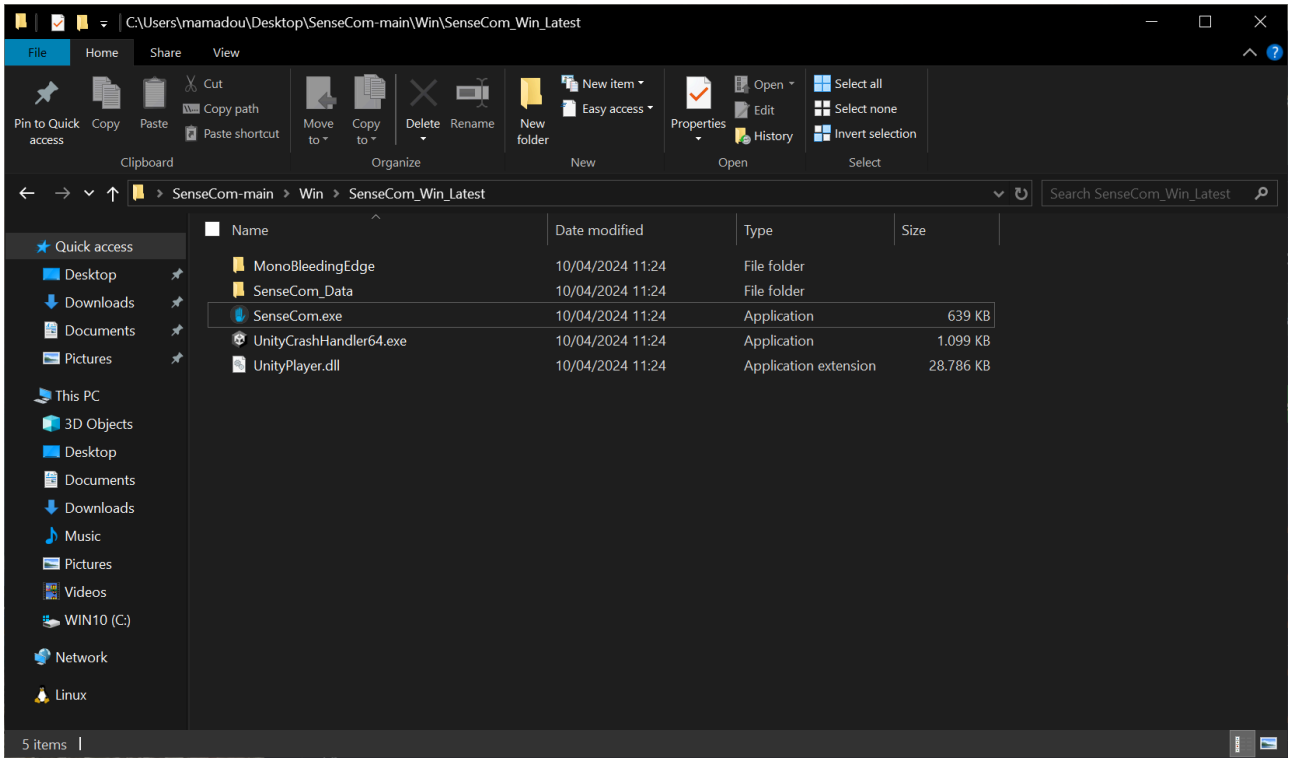
10. Once the glove is paired, you're good to go. Click on `Done` .



11. Once you are back to Windows Bluetooth settings, verify that the glove is listed as a paired device.



12. After successfully paring your glove, it's time to run SenseCom. Navigate to the folder where you extracted SenseCom and go to to `/path/to/extracted/SenseCom/directory/Win/SenseCom_Win_Latest`.



> **Note**
>
> Inside the `/path/to/extracted/SenseCom/directory/Win/` folder, a SenseCom installer is available if you wish to permanently install it on your operating system.

13. In a moment, SenseCom should recognize and connect to your glove(s):



> **Note**
>
> For more detailed information and troubleshooting, consult the SenseCom documentation page on SGDocs, please.

14. At this stage, SenseCom is ready and you should be able to connect to and communicate with SenseGlove devices from inside your Unreal Engine applications.

Enabling XR_EXT_hand_tracking OpenXR Extension on VR Headsets

> **Important**
>
> Starting from version `v2.1.0`, the SenseGlove Unreal Engine Plugin requires the `XR_EXT_hand_tracking` OpenXR extension to function. Without this OpenXR extension the plugin won't output any glove data.

Since version `v2.1.0`, the SenseGlove Unreal Engine Plugin requires the `XR_EXT_hand_tracking` OpenXR extension to function. Whether you are streaming your immersive 3D VR application from your PC to your VR headset, or deploying it to your VR headset in standalone mode, enabling `XR_EXT_hand_tracking` support, might require additional plugins or settings depending on the HMD's vendor or model.

PCVR Mode

For instructions on how to setup `XR_EXT_hand_tracking` support in PCVR mode please refer to the relevant section.

Standalone Mode

For instructions on how to setup `XR_EXT_hand_tracking` support in standalone mode on Android please refer to the relevant section.

Third-Party Tutorials

As a part of this OpenXR comprehensive tutorial series, you will learn how to enable the developer runtime features, set up the OpenXR runtime, and the `XR_EXT_hand_tracking` support in PCVR mode. Furthermore, it will show you how to enable hand-tracking on Android (standalone mode) using the Meta XR and VIVE OpenXR plugins.

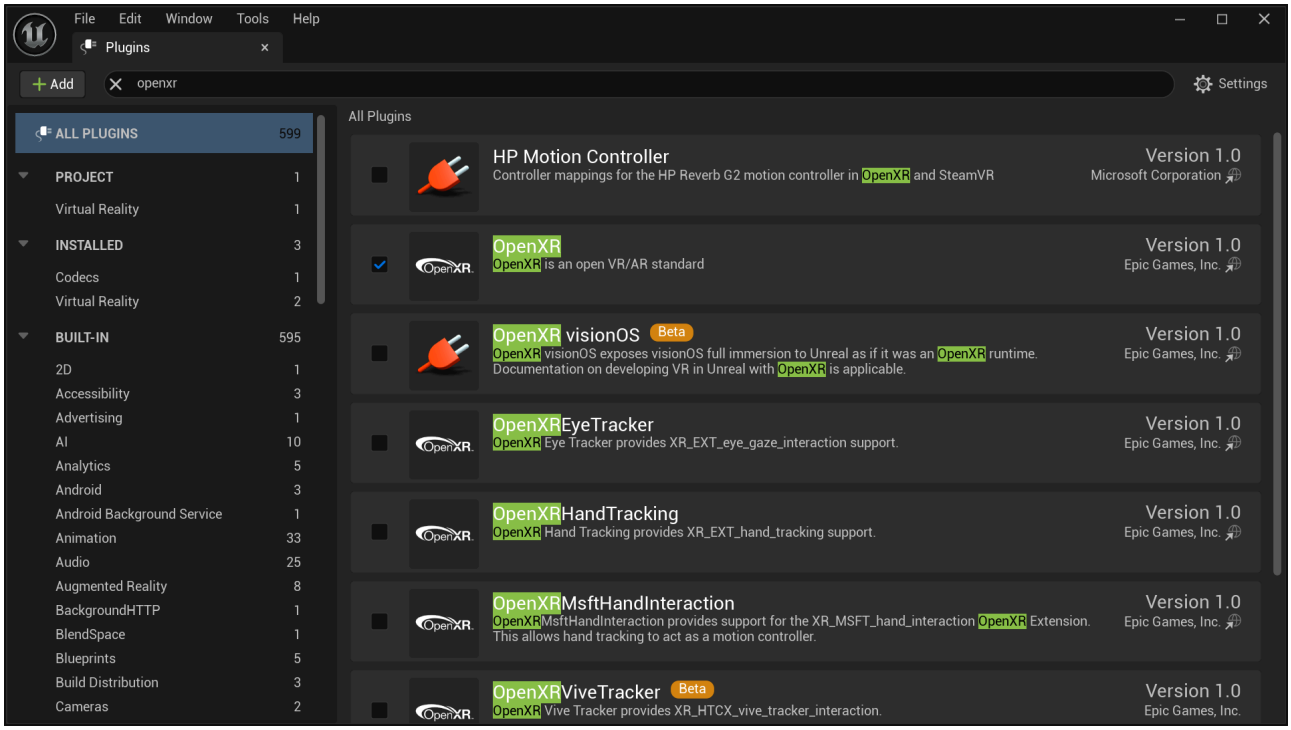
Enabling XR_EXT_hand_tracking OpenXR Extension on VR Headsets in PCVR Mode

Starting from version `v2.1.0`, the SenseGlove Unreal Engine Plugin requires the `XR_EXT_hand_tracking` OpenXR extension to function. If you are streaming your immersive 3D VR application from your PC to your VR headset, enabling `XR_EXT_hand_tracking` support, requires additional plugins and settings depending on the HMD's vendor or model.

Enabling OpenXR Plugin and Disabling OpenXRHandTracking Plugin

Regardless of the type or vendor of the HMD you have in mind for development or deployment purposes, the `OpenXR` plugin is required as a prerequisite. Also, ensure the `OpenXRHandTracking` is disabled as it conflicts with the SenseGlove Unreal Engine Plugin since both implement the same `XR_EXT_hand_tracking` OpenXR extension.

Though enabling the SenseGlove Unreal Engine Plugin should enable the `OpenXR` plugin automatically, it is recommended to ensure this plugin is enabled, and most importantly `OpenXRHandTracking` is disabled, by navigating to `Edit > Plugins` in the Unreal Editor menus.



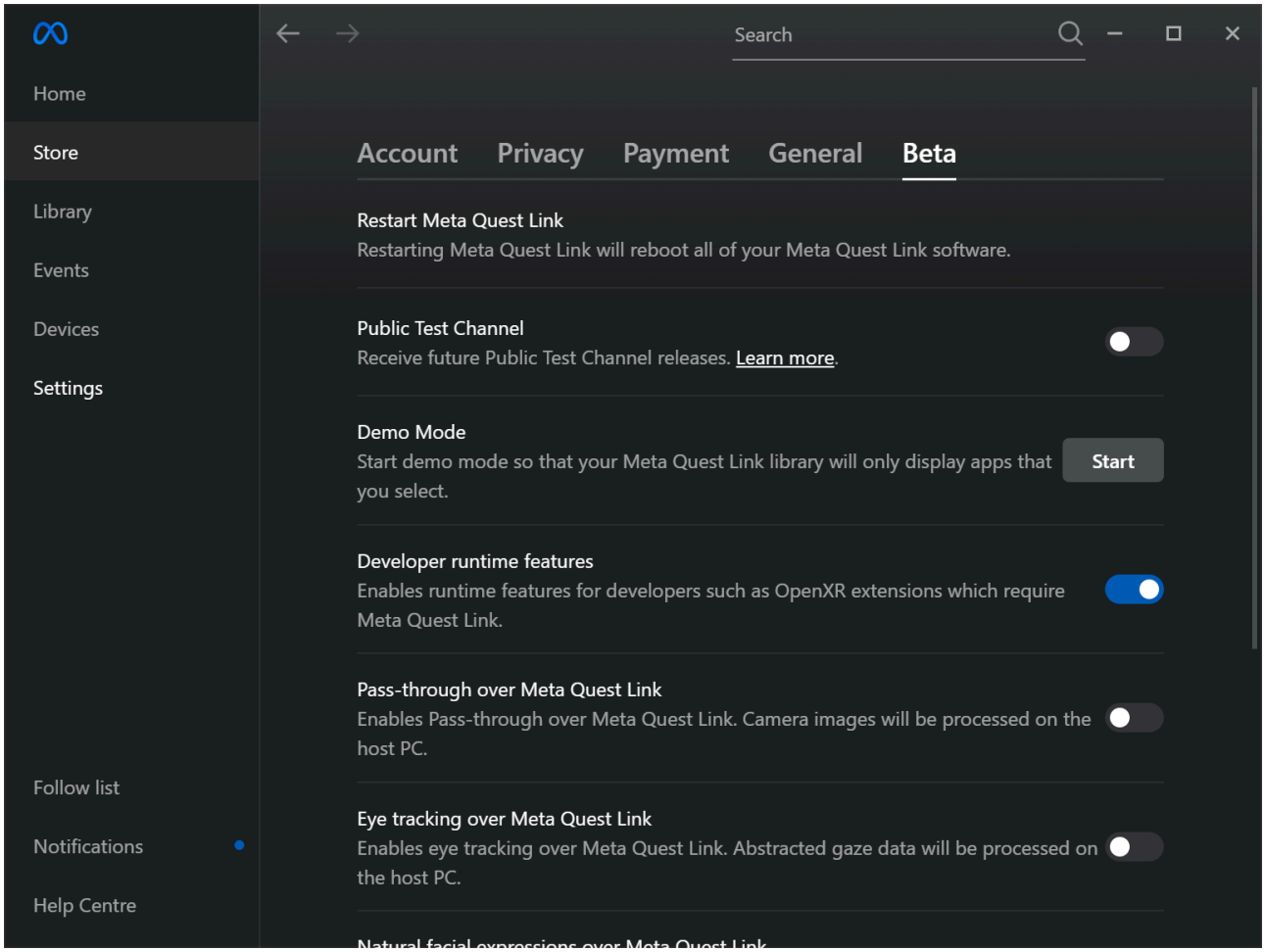
The `OpenXRHandTracking` plugin implements the `XR_EXT_hand_tracking` OpenXR extension.

Meta Quest

To set up `XR_EXT_hand_tracking` support on Meta Quest HMDs in PCVR mode, depending on your project requirements (e.g. whether you rely on the `Meta XR` plugin or not), additional setup steps are required.

Meta Quest Link App

For Meta Quest headsets, enable the `Developer runtime features` under the `Settings > Beta` section inside the Meta Quest Link app:



Important

Enabling `Developer runtime features` in the Meta Quest Link requires a Meta Developer Account. If you are not signed in using a Meta Developer Account, this option won't be shown to you inside Meta Quest Link.

Caution

Streaming to Meta Quest headsets from SteamVR is no longer supported because the migration to OpenXR has caused controller offsets for Meta Quest HMDs to break on SteamVR. One possible reason is that SteamVR lists `XR_FB_hand_tracking` as an unsupported feature. Further investigation is needed to identify the exact underlying cause.

Meta XR Plugin

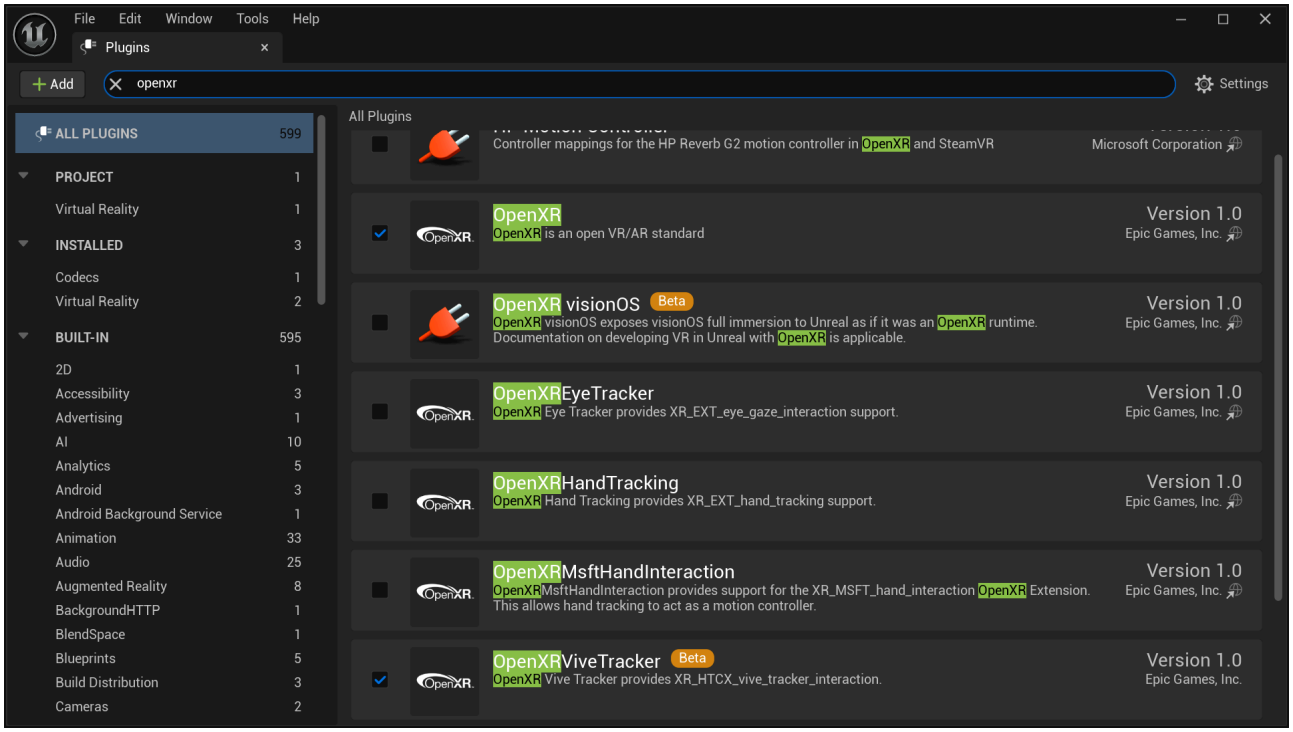
> **Caution**
>
> Please note that enabling the `Meta XR` plugin alongside the SenseGlove plugin will result in crashes or unexpected behavior. `Meta XR` plugin compatibility is being worked on and might be supported in the future.

HTC VIVE

To set up `XR_EXT_hand_tracking` support on HTC VIVE HMDs in PCVR mode, additional plugins or configuration steps are required.

OpenXRViveTracker Plugin

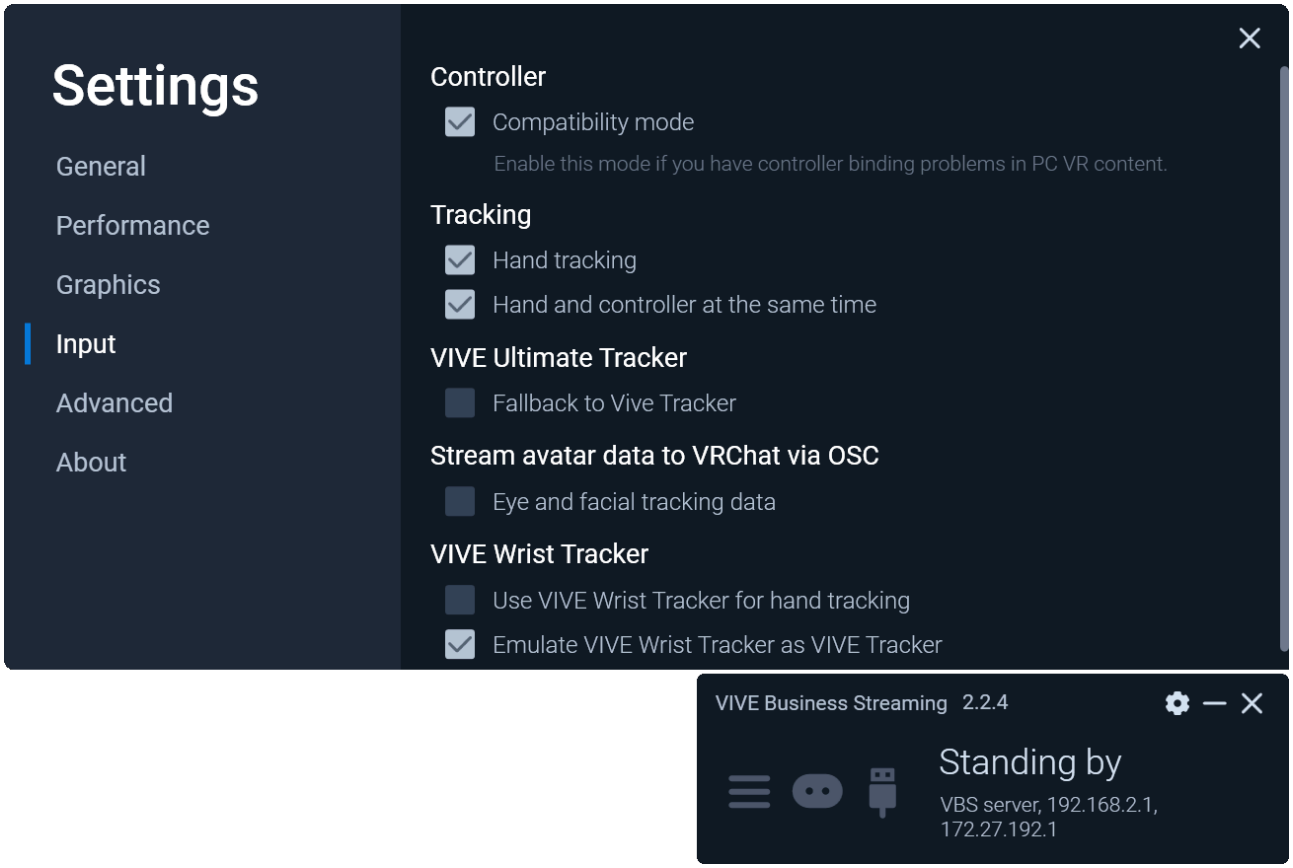
To enable VIVE Trackers support ensure the `OpenXRViveTracker` plugin is enabled by navigating to `Edit > Plugins` in the Unreal Editor menus. This plugin should be enabled, or wrist tracking won't function on VIVE devices at all.



The `OpenXRViveTracker` plugin implements the `XR_HTCX_vive_tracker_interaction` OpenXR extension which is necessary to use VIVE Trackers or to emulate the VIVE Wrist Trackers as VIVE Trackers on Windows.

VIVE Business Streaming App

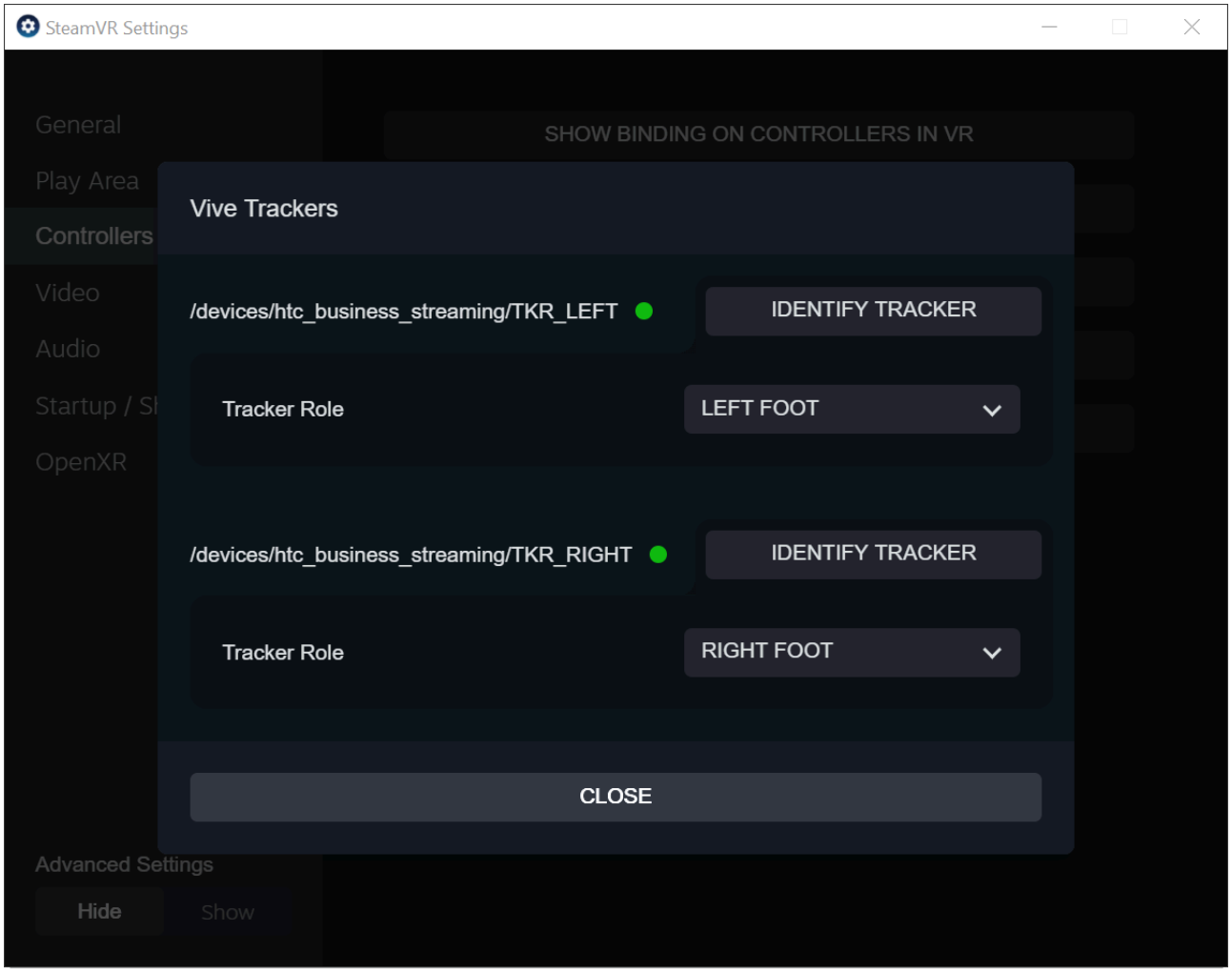
For VIVE headsets relying on the VIVE Business Streaming application, ensure the Hand Tracking settings under `Input` are enabled and `XR_HTCX_vive_tracker_interaction` is enabled for VIVE Wrist Trackers by enabling `Emulate VIVE Wrist Tracker as VIVE Tracker`:

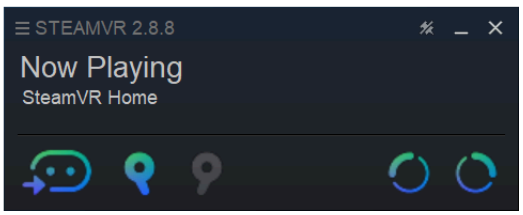


> **Note**
>
> Tracking and accessing `FXRMotionControllerData` output from SenseGlove devices do not require Hand and Body Tracking to be enabled on the HMD device. Enabling this feature is only necessary if you wish to use hand-tracking as a fallback option when no glove is connected to your PC.

SteamVR App

After enabling the OpenXRViveTracker plugin and enabling `Emulate VIVE Wrist Tracker as VIVE Tracker` for VIVE HMDs relying on the VIVE Business Streaming, you need to perform one final setup in the SteamVR app for the SenseGlove Unreal Engine Plugin to be able to retrieve the correct wrist-tracking offsets. Once you have paired your VIVE Trackers or VIVE Wrist Trackers, navigate to `SteamVR Settings > Controllers > MANAGE TRACKERS` and make sure your left tracker is set to `LEFT FOOT` and the right tracker is set to `RIGHT FOOT`:





SenseGlove Wrist Tracking Settings

Once you have set up everything, it's time to adjust the SenseGlove Wrist Tracking Settings inside the project-wide plugin's settings. For detailed information, please visit the Wrist Tracking Hardware and HMD auto-detection configuration section as well.

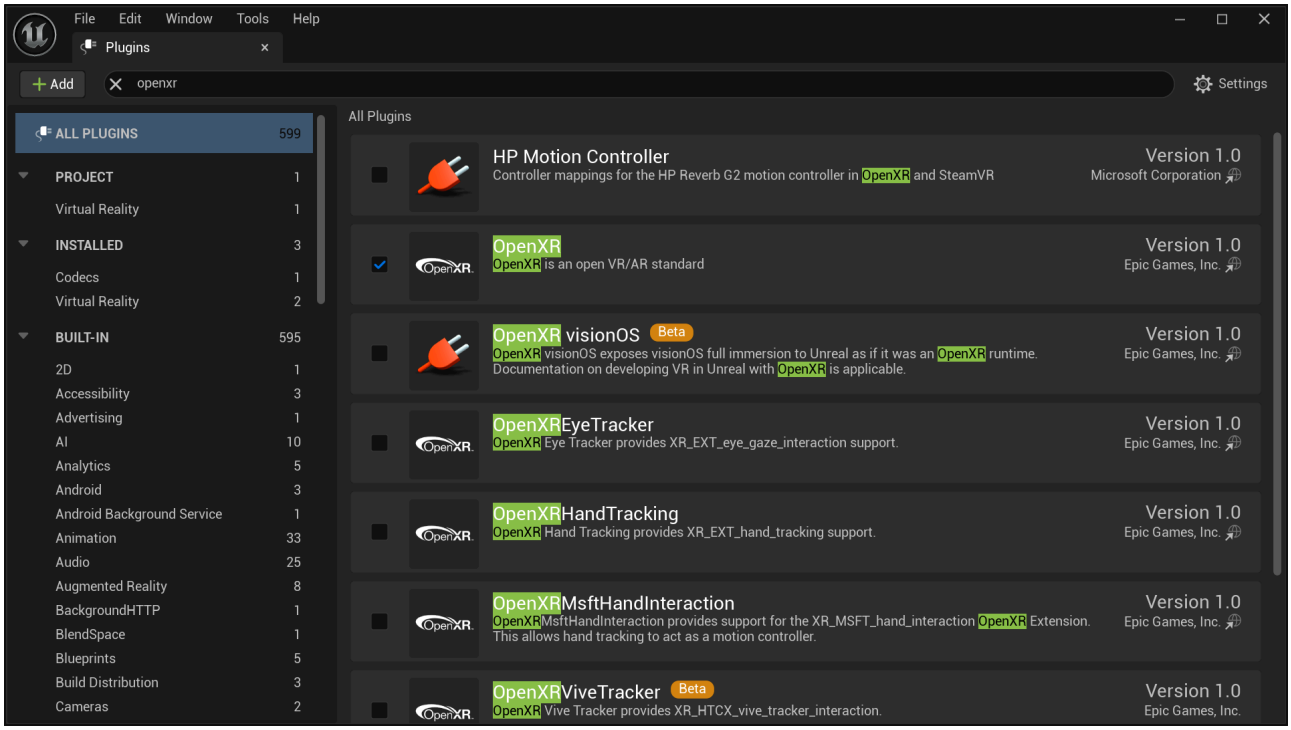
Enabling XR_EXT_hand_tracking OpenXR Extension on VR Headsets in Standalone Mode

Starting from version `v2.1.0`, the SenseGlove Unreal Engine Plugin requires the `XR_EXT_hand_tracking` OpenXR extension to function. If you are deploying your immersive 3D VR application to your VR headset in standalone mode, enabling `XR_EXT_hand_tracking` support, requires additional plugins and settings depending on the HMD's vendor or model.

Enabling OpenXR Plugin and Disabling OpenXRHandTracking Plugin

Regardless of the type or vendor of the HMD you have in mind for development or deployment purposes, the `OpenXR` plugin is required as a prerequisite. Also, ensure the `OpenXRHandTracking` is disabled as it conflicts with the SenseGlove Unreal Engine Plugin since both implement the same `XR_EXT_hand_tracking` OpenXR extension.

Though enabling the SenseGlove Unreal Engine Plugin should enable the `OpenXR` plugin automatically, it is recommended to ensure this plugin is enabled, and most importantly `OpenXRHandTracking` is disabled, by navigating to `Edit > Plugins` in the Unreal Editor menus.



The `OpenXRHandTracking` plugin implements the `XR_EXT_hand_tracking` OpenXR extension.

Meta Quest

To set up `XR_EXT_hand_tracking` support on Meta Quest HMDs in Standalone mode, depending on your project requirements (e.g. whether you rely on the `Meta XR` plugin or not), additional setup steps are required.

> **Important**
>
> Although, the SenseGlove plugin does not require the `Meta XR` plugin to function, and relying solely on the `OpenXR` and `OpenXRHandTracking` plugins would suffice for functional glove data retrieval using OpenXR, hand-tracking as a fallback mechanism won't work on Android without the `Meta XR` plugin availability.

Meta XR Plugin

> **Caution**
>
> Please note that enabling the `Meta XR` plugin alongside the SenseGlove plugin will result in crashes or unexpected behavior. `Meta XR` plugin compatibility is being worked on and might be supported in the future.

HTC VIVE

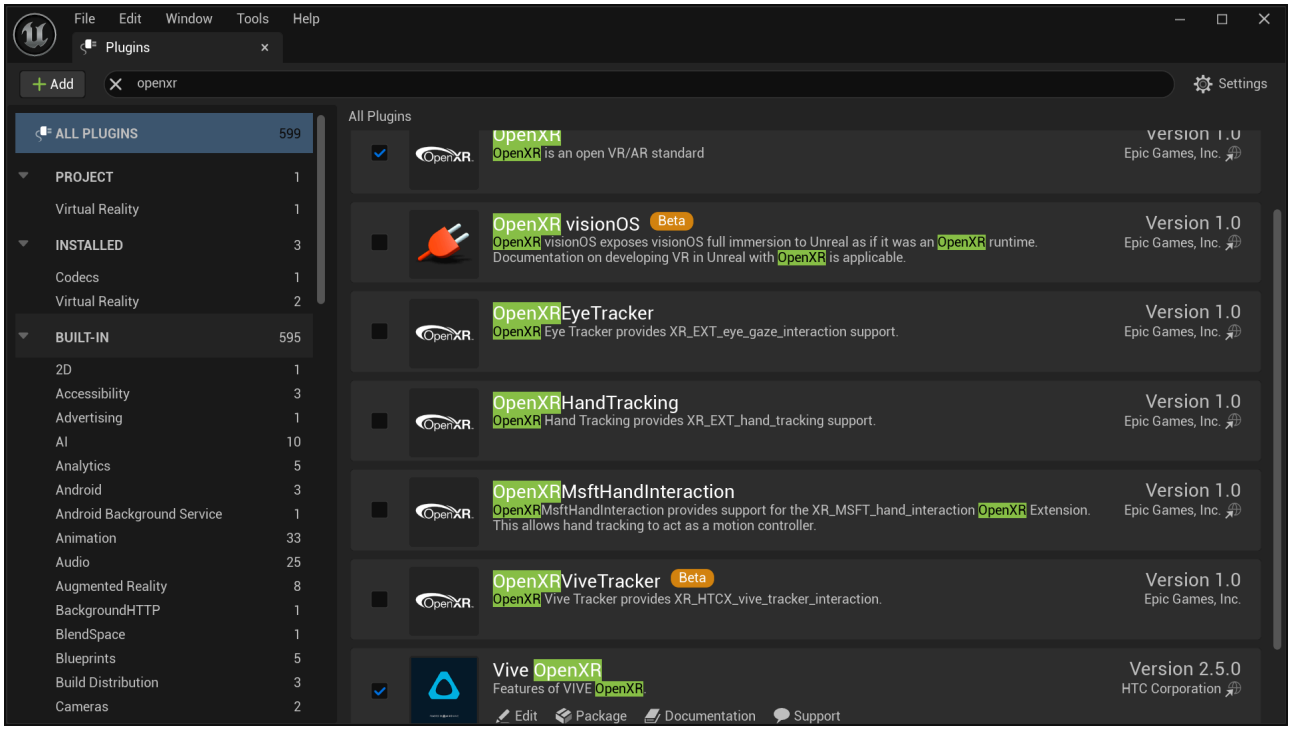
To set up `XR_EXT_hand_tracking` support on HTC VIVE HMDs in Standalone mode, additional plugins or configuration steps are required.

OpenXRViveTracker Plugin

Unlike the PCVR-mode on Windows, the `OpenXRViveTracker` plugin is not required on Android since it only provides an implementation of the `XR_HTCX_vive_tracker_interaction` OpenXR extension which is necessary when we use VIVE Trackers on Windows or we emulate the VIVE Wrist Trackers as VIVE Trackers on Windows. Instead, we require the `XR_HTCX_vive_wrist_tracker_interaction` OpenXR extension to be able to use VIVE Wrist Trackers on Android, which is provided by the `ViveOpenXR` plugin. So, you can safely ignore enabling this plugin for Android standalone deployments.

ViveOpenXR Plugin

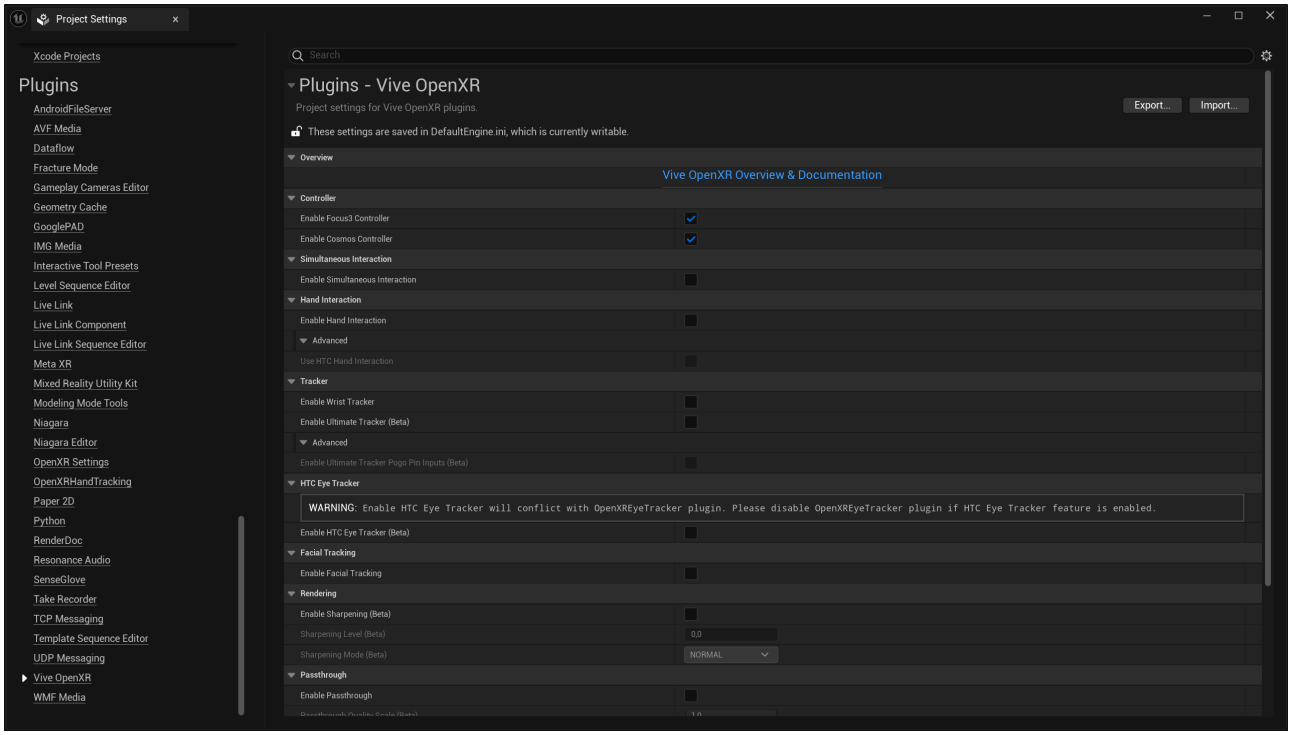
To enable VIVE Wrist Trackers support ensure the `ViveOpenXR` plugin is enabled by navigating to `Edit > Plugins` in the Unreal Editor menus. This plugin should be enabled, or wrist tracking won't function on VIVE devices at all.

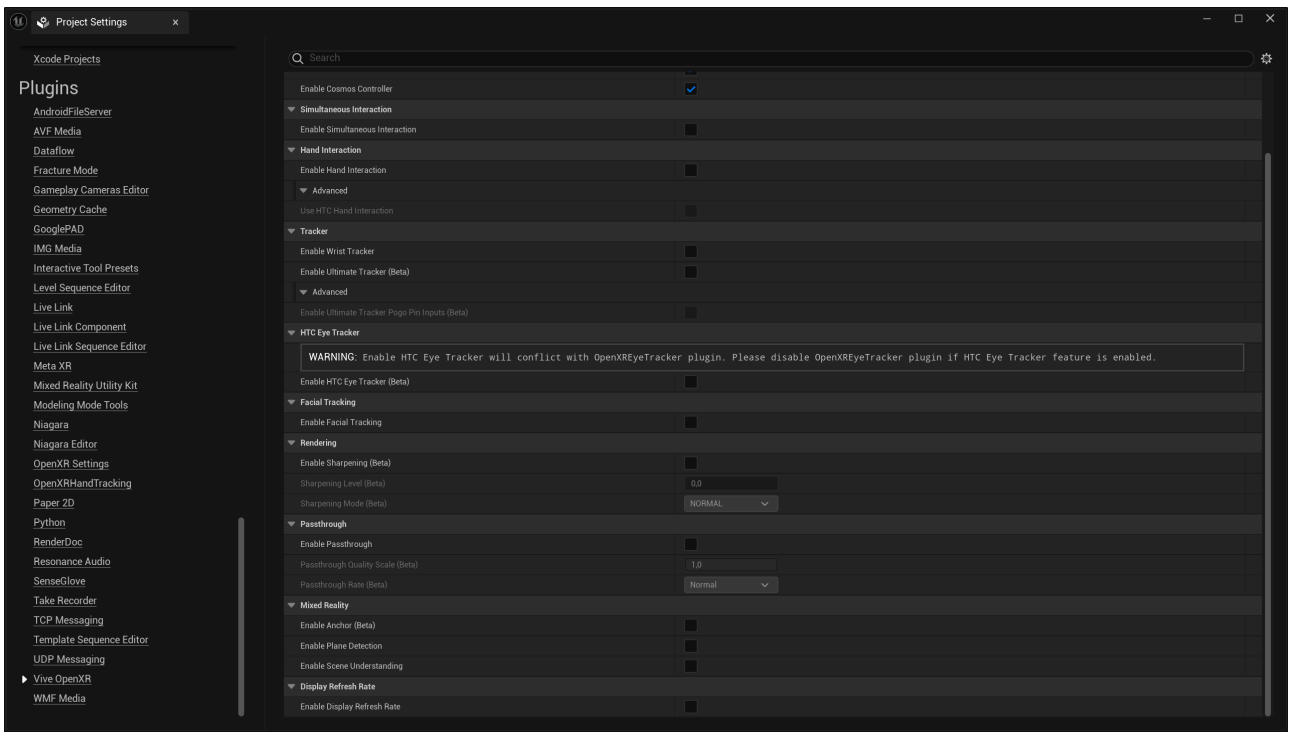


The `ViveOpenXR` plugin implements the `XR_HTCX_vive_wrist_tracker_interaction` OpenXR extension.

> **Important**
>
> Without the `ViveOpenXR` plugin deploying an immersive 3D VR application to your HTC VIVE in Standalone mode won't be possible, whether you enable `bStartInVR`, or not. Without it your app will be deployed and recognized as a 2D app.

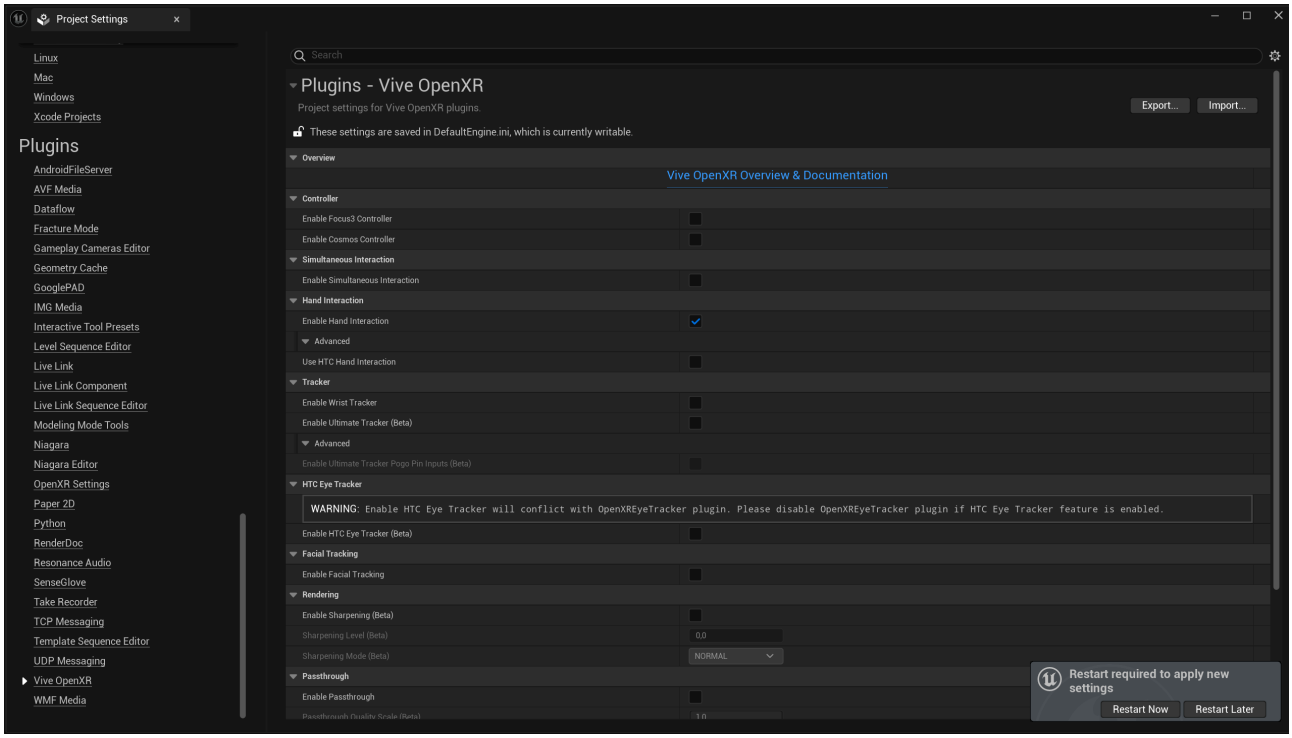
By default, the `ViveOpenXR` plugin settings located in `Edit > Project Settings > Plugins > Vive OpenXR` look something like these:





The following settings control the availability of hand-tracking when the `ViveOpenXR` plugin is enabled:

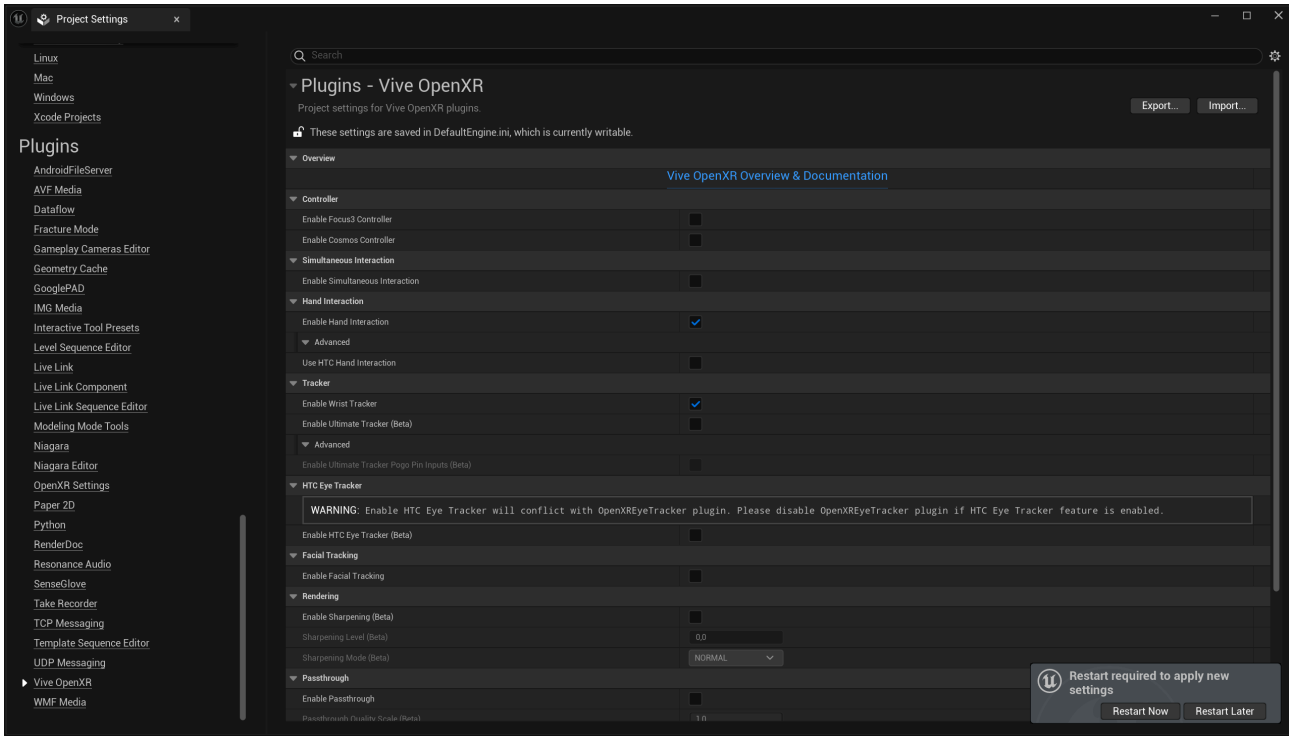
- **Enable Hand Interaction**: This enables the hand interactions with the OpenXR hand interaction extension `XR_HTC_hand_interaction`. Changing this setting will prompt you to restart the engine to apply the new settings. This setting should be enabled.



- **Use HTC Hand Interaction**: This selects which OpenXR hand interaction extension to use. If enabled, `XR_HTC_hand_interaction` will be used, effectively breaking SenseGlove glove and hand-tracking data output. If disabled, `XR_EXT_hand_interaction` will be used, which is compatible with the SenseGlove Unreal Engine Plugin. Changing this setting will prompt you to restart the engine to apply the new settings. This setting should be disabled.

The following settings control the availability of wrist-tracking when the `ViveOpenXR` plugin is enabled:

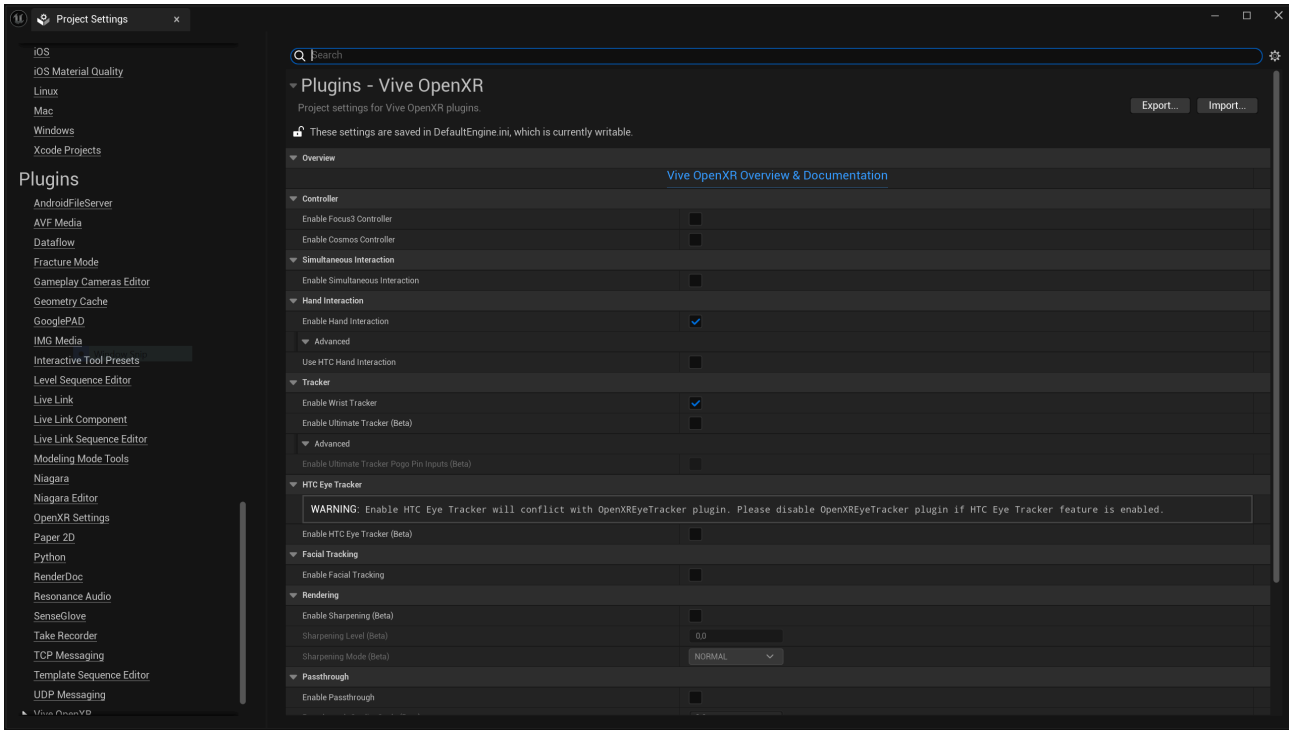
- **Enable Wrist Tracker**: This option controls the `XR_HTC_vive_wrist_tracker_interaction` OpenXR extension. If enabled, in turn, it enables the use of HTC Wrist Tracker interaction profiles in OpenXR. Changing this setting will prompt you to restart the engine to apply the new settings.

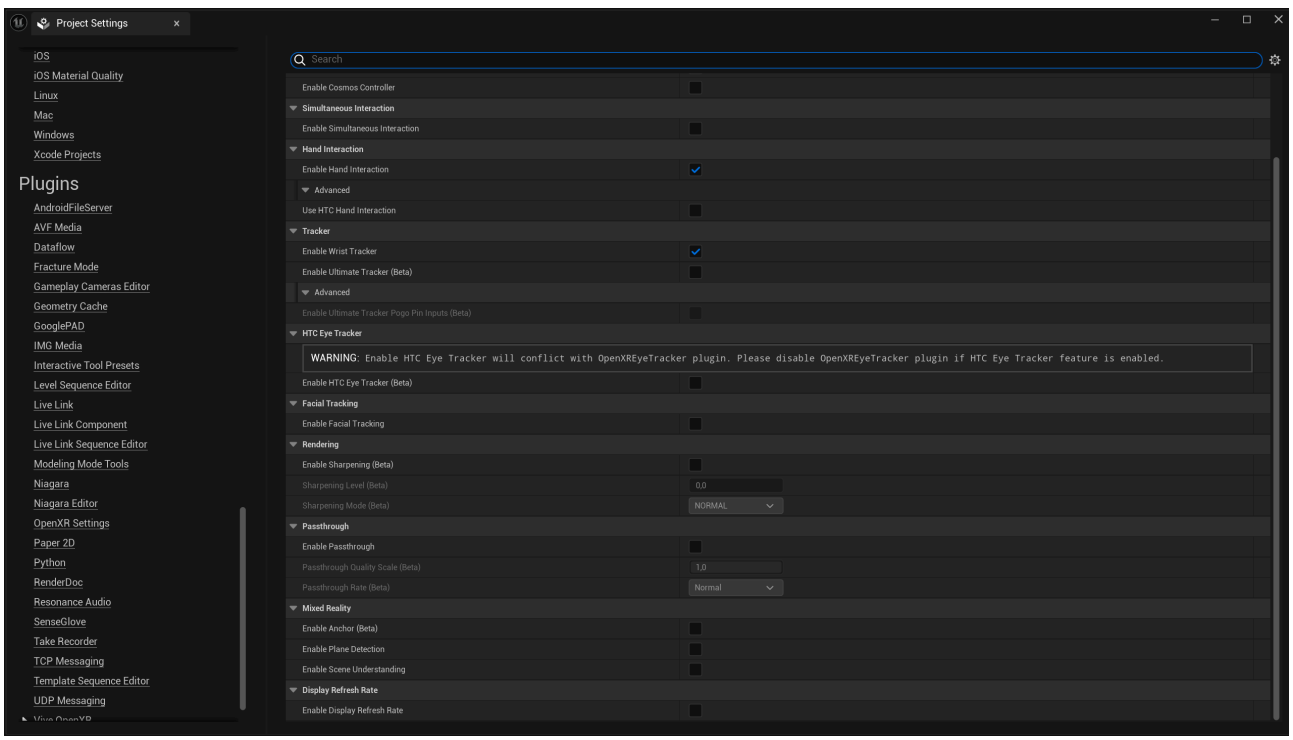


- **Enable Ultimate Tracker (Beta)**: This option controls the `XR_HTC_path_enumeration` and `XR_HTC_vive_xr_tracker_interaction` OpenXR extensions. If enabled, in turn, they enable the use of HTC Xr Tracker interaction profiles in OpenXR. Changing this setting will prompt you to restart the engine to apply the new settings.

- **Enable Ultimate Tracker Pogo Pin Inputs (Beta)**: Enables or disables the input options for Unreal's Enhanced Input System. Changing this setting will prompt you to restart the engine to apply the new settings.

The following setup demonstrates a functional immersive 3D VR application with the minimum `ViveOpenXR` required features enabled to make it compatible with the SenseGlove Unreal Engine Plugin:


Project Settings
iOS
iOS Material Quality
Linux
Mac
Windows
Xcode Projects
Plugins
AndroidFileServer
AVF Media
Dataflow
Fracture Mode
Gameplay Cameras Editor
Geometry Cache
GooglePAD
IMG Media
Interactive Tool Presets
Level Sequence Editor
Live Link
Live Link Component
Live Link Sequence Editor
Modeling Mode Tools
Niagara
Niagara Editor
OpenXR Settings
Paper 2D
Python
RenderDoc
Resonance Audio
SenseGlove
Take Recorder
TCP Messaging
Template Sequence Editor
UDP Messaging
Search
Plugins - Vive OpenXR
Project settings for Vive OpenXR plugins.
Export...
Import...
These settings are saved in DefaultEngine.ini, which is currently writable.
Overview
Vive OpenXR Overview & Documentation
Controller
Enable Focus3 Controller
Enable Cosmos Controller
Simultaneous Interaction
Enable Simultaneous Interaction
Hand Interaction
Enable Hand Interaction
Advanced
Use HTC Hand Interaction
Tracker
Enable Wrist Tracker
Enable Ultimate Tracker (Beta)
Advanced
Enable Ultimate Tracker Pogo Pin Inputs (Beta)
HTC Eye Tracker
WARNING: Enable HTC Eye Tracker will conflict with OpenXREyeTracker plugin. Please disable OpenXREyeTracker plugin if HTC Eye Tracker feature is enabled.
Enable HTC Eye Tracker (Beta)
Facial Tracking
Enable Facial Tracking
Rendering
Enable Sharpening (Beta)
Sharpening Level (Beta)
0,0
Sharpening Mode (Beta)
NORMAL
Passthrough
Enable Passthrough


Project Settings
iOS
iOS Material Quality
Linux
Mac
Windows
Xcode Projects
Plugins
AndroidFileServer
AVF Media
Dataflow
Fracture Mode
Gameplay Cameras Editor
Geometry Cache
GooglePAD
IMG Media
Interactive Tool Presets
Level Sequence Editor
Live Link
Live Link Component
Live Link Sequence Editor
Modeling Mode Tools
Niagara
Niagara Editor
OpenXR Settings
Paper 2D
Python
RenderDoc
Resonance Audio
SenseGlove
Take Recorder
TCP Messaging
Template Sequence Editor
UDP Messaging
Search
Enable Cosmos Controller
Simultaneous Interaction
Enable Simultaneous Interaction
Hand Interaction
Enable Hand Interaction
Advanced
Use HTC Hand Interaction
Tracker
Enable Wrist Tracker
Enable Ultimate Tracker (Beta)
Advanced
Enable Ultimate Tracker Pogo Pin Inputs (Beta)
HTC Eye Tracker
WARNING: Enable HTC Eye Tracker will conflict with OpenXREyeTracker plugin. Please disable OpenXREyeTracker plugin if HTC Eye Tracker feature is enabled.
Enable HTC Eye Tracker (Beta)
Facial Tracking
Enable Facial Tracking
Rendering
Enable Sharpening (Beta)
Sharpening Level (Beta)
0,0
Sharpening Mode (Beta)
NORMAL
Passthrough
Enable Passthrough
Passthrough Quality Scale (Beta)
1,0
Passthrough Rate (Beta)
Normal
Mixed Reality
Enable Anchor (Beta)
Enable Plane Detection
Enable Scene Understanding
Display Refresh Rate
Enable Display Refresh Rate

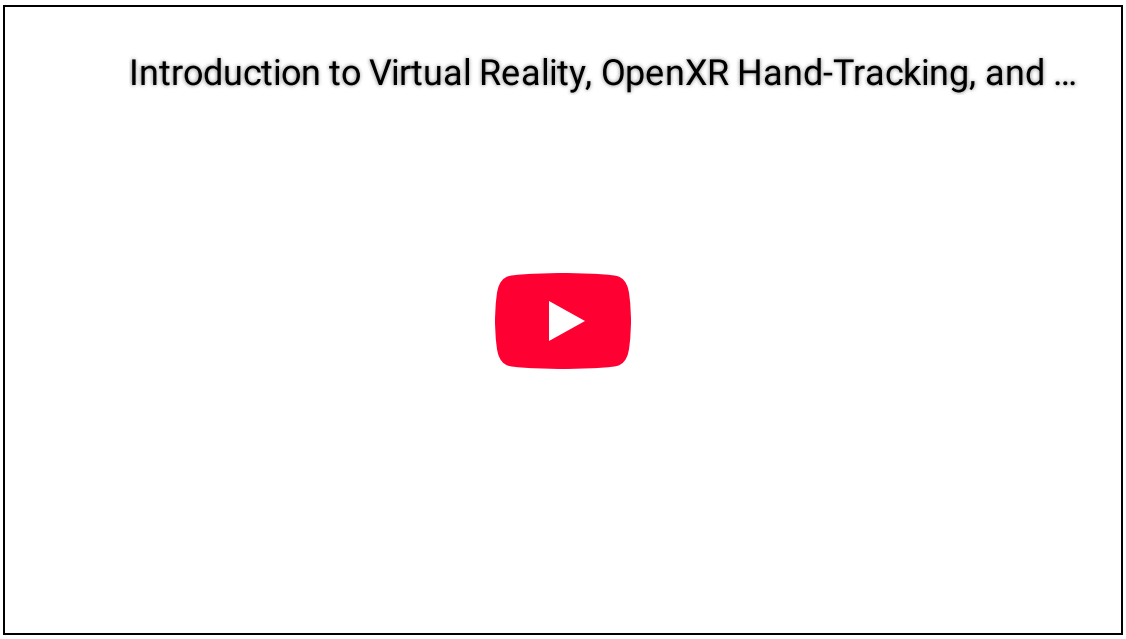
SenseGlove Wrist Tracking Settings

Once you have set up everything, it's time to adjust the SenseGlove Wrist Tracking Settings inside the project-wide plugin's settings. For detailed information, please visit the Wrist Tracking Hardware and HMD auto-detection configuration section as well.

Third-Party Tutorials: XR_EXT_hand_tracking Setup

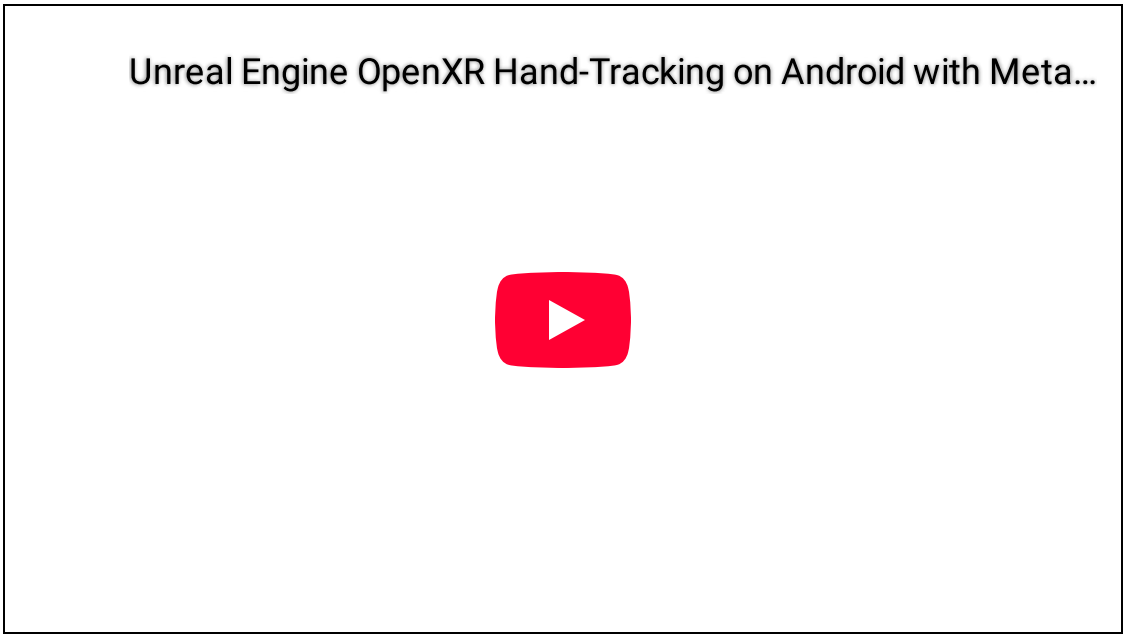
Introduction to Virtual Reality, OpenXR Hand-Tracking, and Gesture Detection in Unreal Engine

A part of this comprehensive tutorial will guide you through setting up the OpenXR runtime and enabling developer runtime features:



Unreal Engine OpenXR Hand-Tracking on Android with Meta XR (Quest 3S/3/Pro/2) and HTC VIVE OpenXR (Focus Vision/XR Elite/Focus 3) Plugins

A part of this comprehensive tutorial will guide you through setting up the Meta XR and VIVE OpenXR plugins for Android standalone-mode deployment:


Unreal Engine OpenXR Hand-Tracking on Android with Meta...

Setting Up the SenseGlove Default Classes

Setting up the default SenseGlove classes is recommended if you want to take full advantage of the quality-of-life features provided by the SenseGlove Unreal Engine Plugin. These features are designed to streamline the development process within the Unreal Engine environment. For instance, if you need a quick setup with a virtual hand mesh already integrated into a pawn, enabling you to get started with your project in just a few minutes, it is essential to configure the default classes and familiarize yourself with these classes.

If you wish to extend the functionality of these classes, you can do so by subclassing them. The default SenseGlove classes, which are prefixed with `SG`, include:

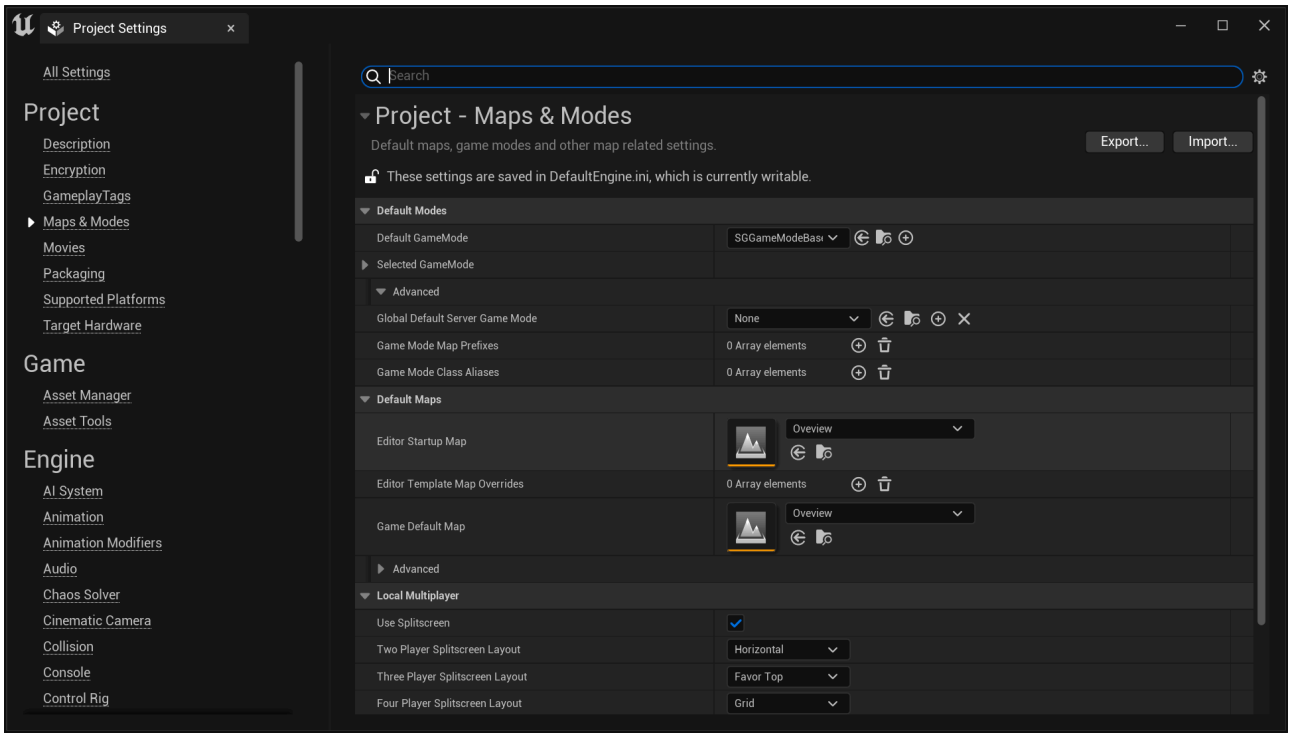
- SGGameModeBase
- SGPawn
- SGPlayerController
- SGGameInstance
- SGGameUserSettings

However, if you prefer a different approach or do not require the functionality provided by the default SenseGlove classes, you can opt to utilize individual components like `SGVirtualHandComponent`, `SGWristTrackerComponent`, etc., directly within your own actors. Alternatively, you can develop a completely custom system from scratch, leveraging the low-level SenseGlove C++ or Blueprint APIs.

Additionally, you can enforce setting the default SenseGlove classes during initialization via the plugin settings, if desired.

Setting Up SGGameModeBase

After installing and enabling the SenseGlove Unreal Engine Plugin, the easiest and most straightforward approach to get started is to just set the default GameMode to `SGGameModeBase` from `Edit > Project Settings... > Maps & Modes > Default Mode > Default GameMode`. By doing this, the `Default Pawn Class` is automatically set to `SGPawn`, and the `Player Controller Class` is set to `SGPlayerController`. This setup ensures that a SenseGlove pawn will automatically spawn when you hit the play button in the editor.



> **Tip**
>
> For greater control and customization, consider extending the SGGameModeBase.

> **Note**

Currently, setting `SGGameModeBase` or a subclass of it as the `Default GameMode` is not a strict requirement. Its primary function is to ensure that a default `SGPawn` and `SGPlayerController` are set. However, this might change in the future, and it could become a mandatory setting.

Important

While setting `SGGameModeBase` as the `Default GameMode` will automatically spawn the default SGPawn at BeginPlay and initiate communication with the SenseGlove devices, it will not display any virtual hands in your simulation by default. You might still need to configure the Virtual Hand Meshes and the Wrist Tracking Hardware separately.

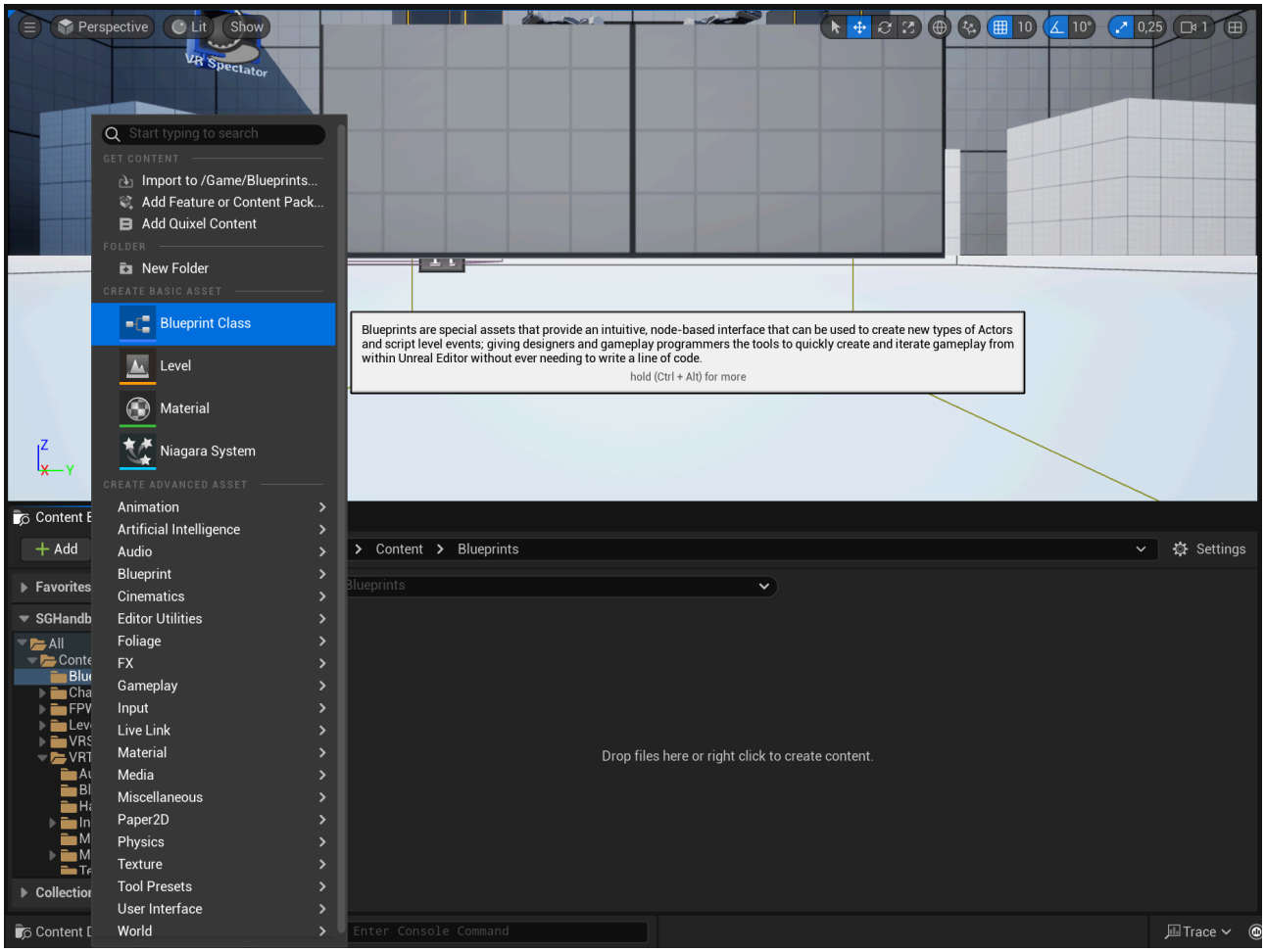
Important

Before starting the simulation in the editor, make sure that SenseCom is running and XR_EXT_hand_tracking is enabled. Without these, your simulation will not receive hand pose data from the SenseGlove devices.

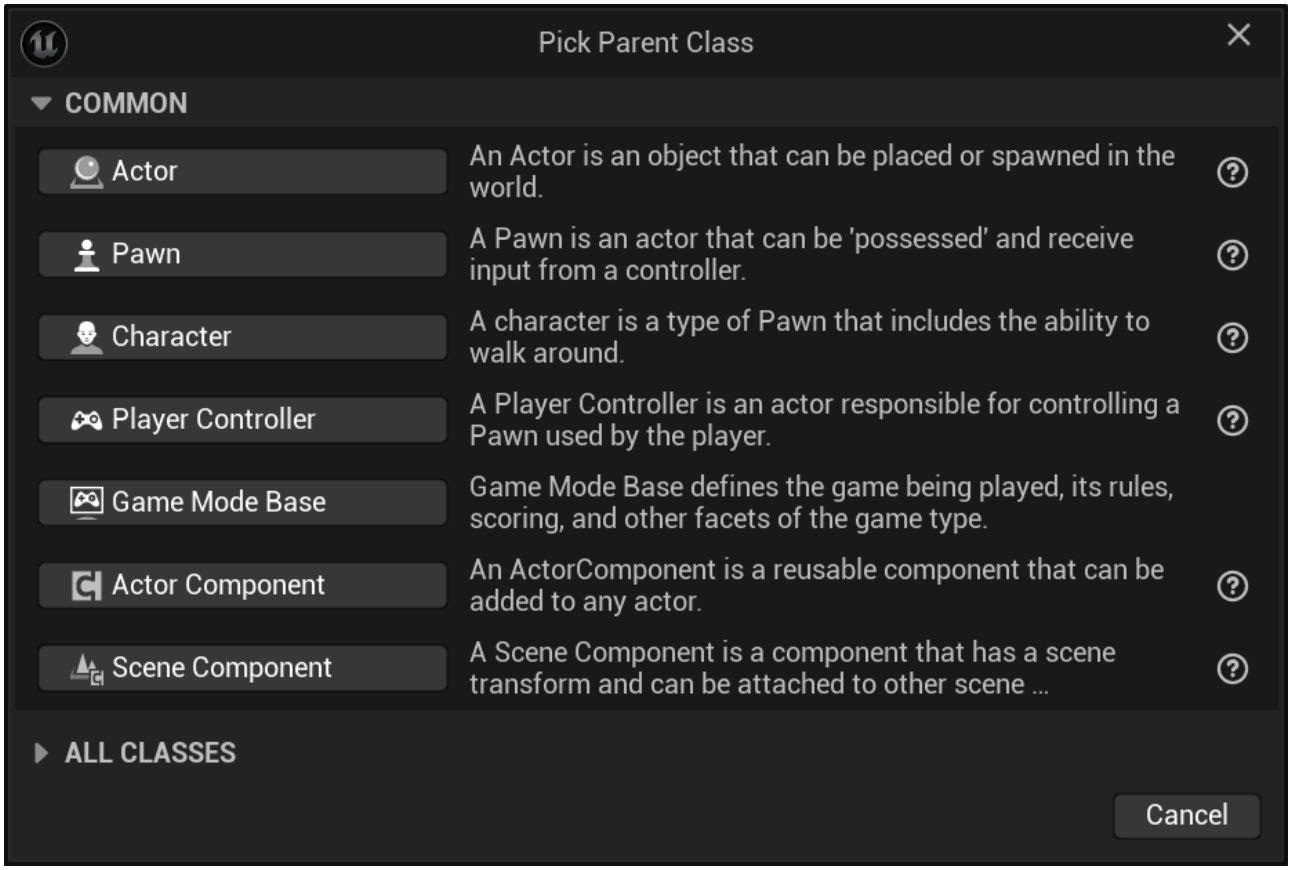
Extending SGGameModeBase

Follow these steps to extend and set up your own version of `SGGameModeBase`:

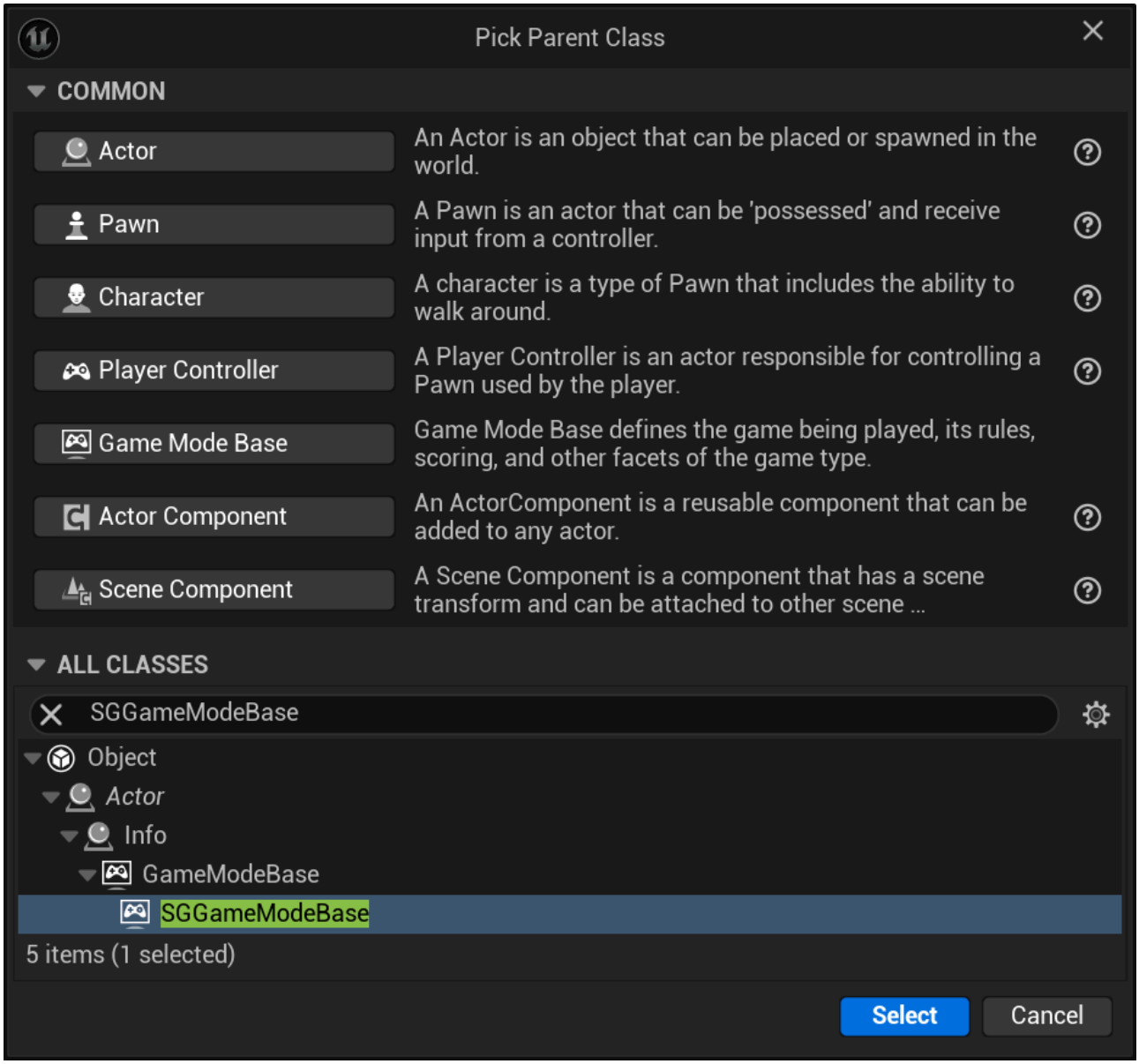
1. In the Content Browser, click the `+ Add` button, then select `Blueprint Class from the menu`. Alternatively, right-click inside the Content Browser and choose `Blueprint Class` from the context menu.



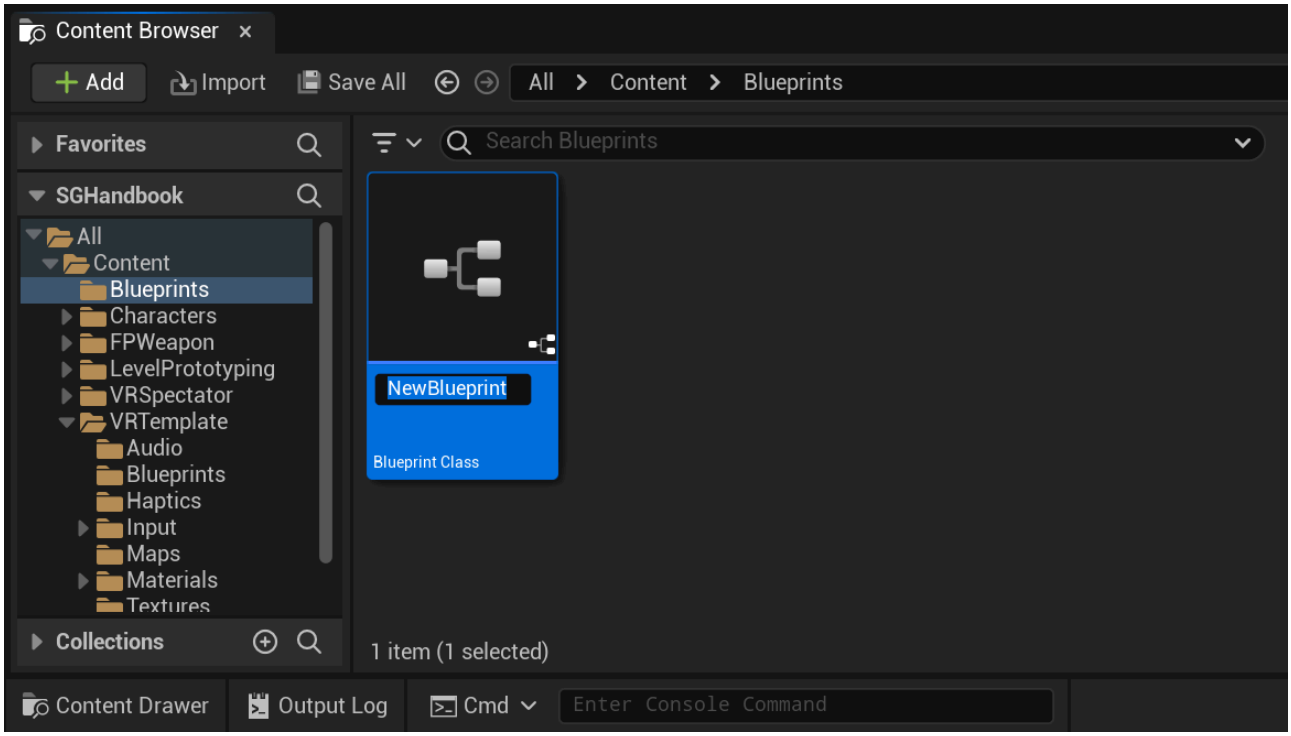
2. A dialog will appear asking you to choose a parent class. Click on the ALL CLASSES section to expand the list of available classes.



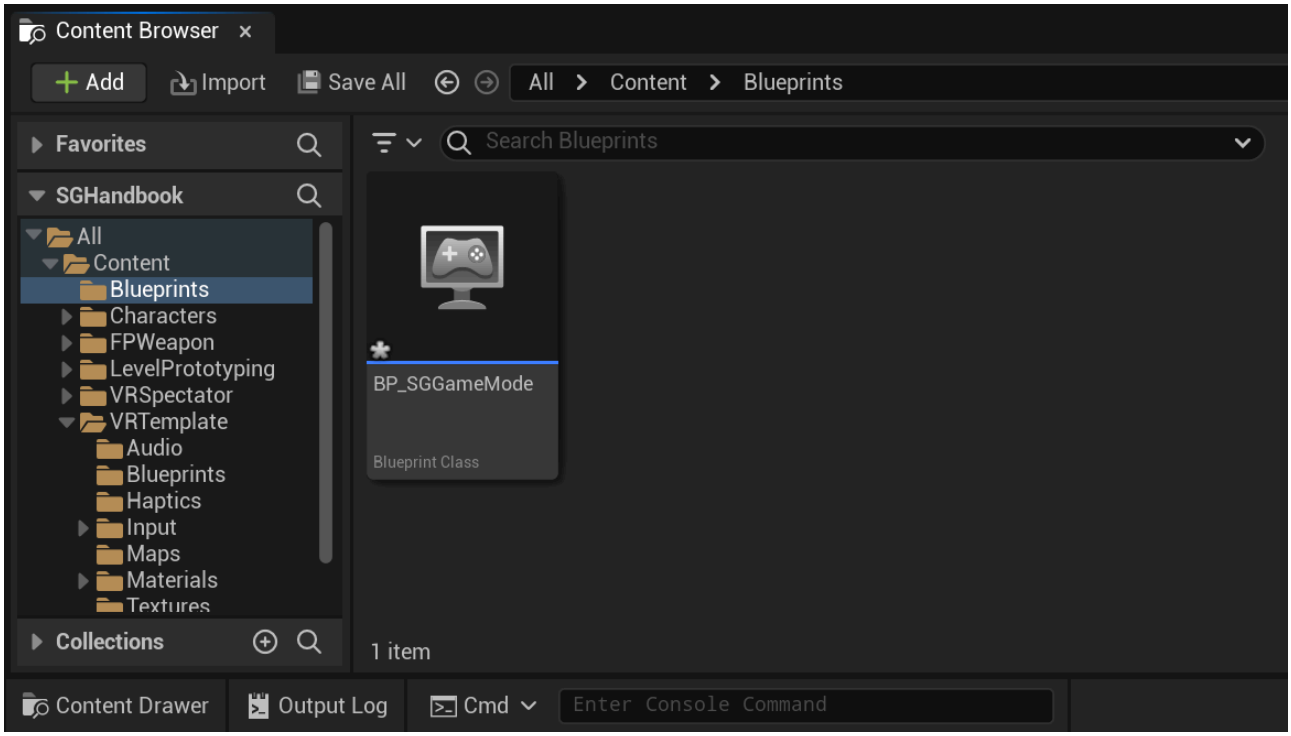
3. In the expanded `ALL CLASSES` section, start typing `SGGameModeBase` in the Search box. When `SGGameModeBase` appears, select it and click the `Select` button to create your new Blueprint class based on it.



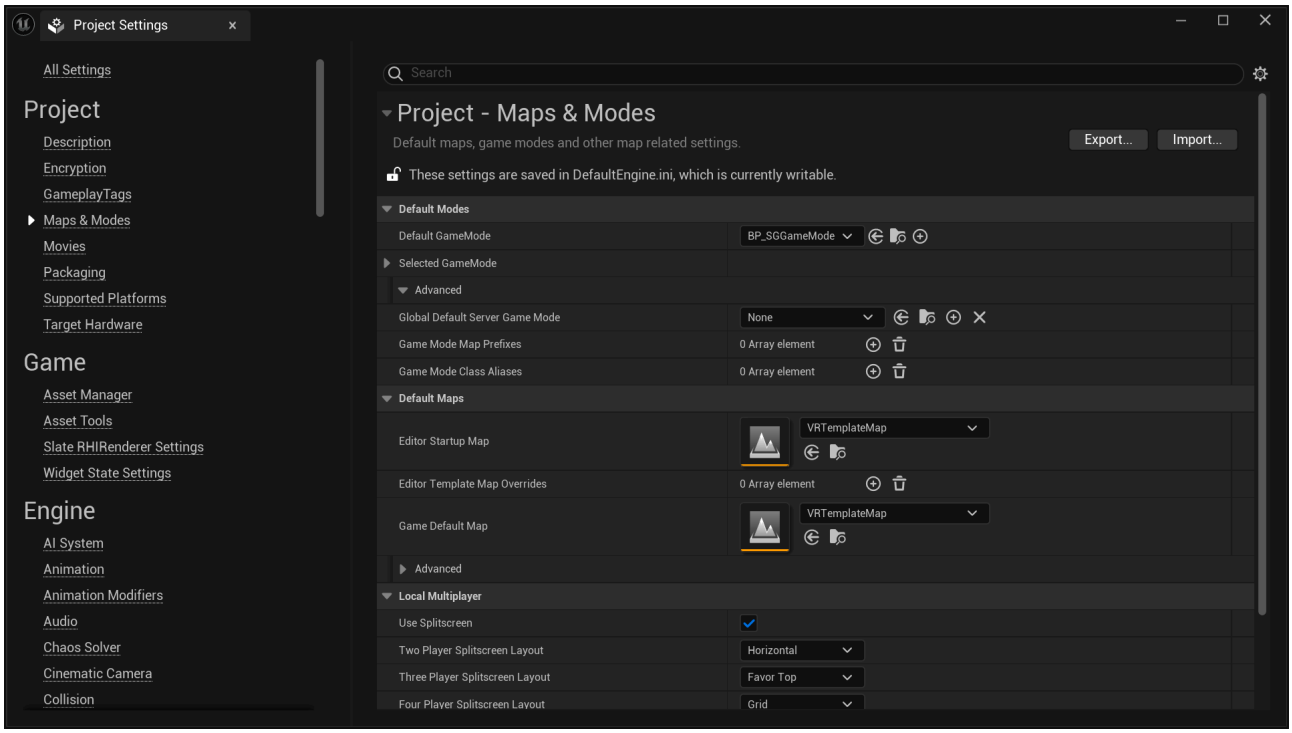
4. After returning to the Content Browser, the Unreal Editor will prompt you to rename `NewBlueprint` to your desired class name. You can rename the class at any time by pressing `F2` or by right-clicking on it and selecting `Rename` from the context menu.



5. Once you have renamed the `NewBlueprint` class to your desired name, click on `Save All` to save the new class to disk.

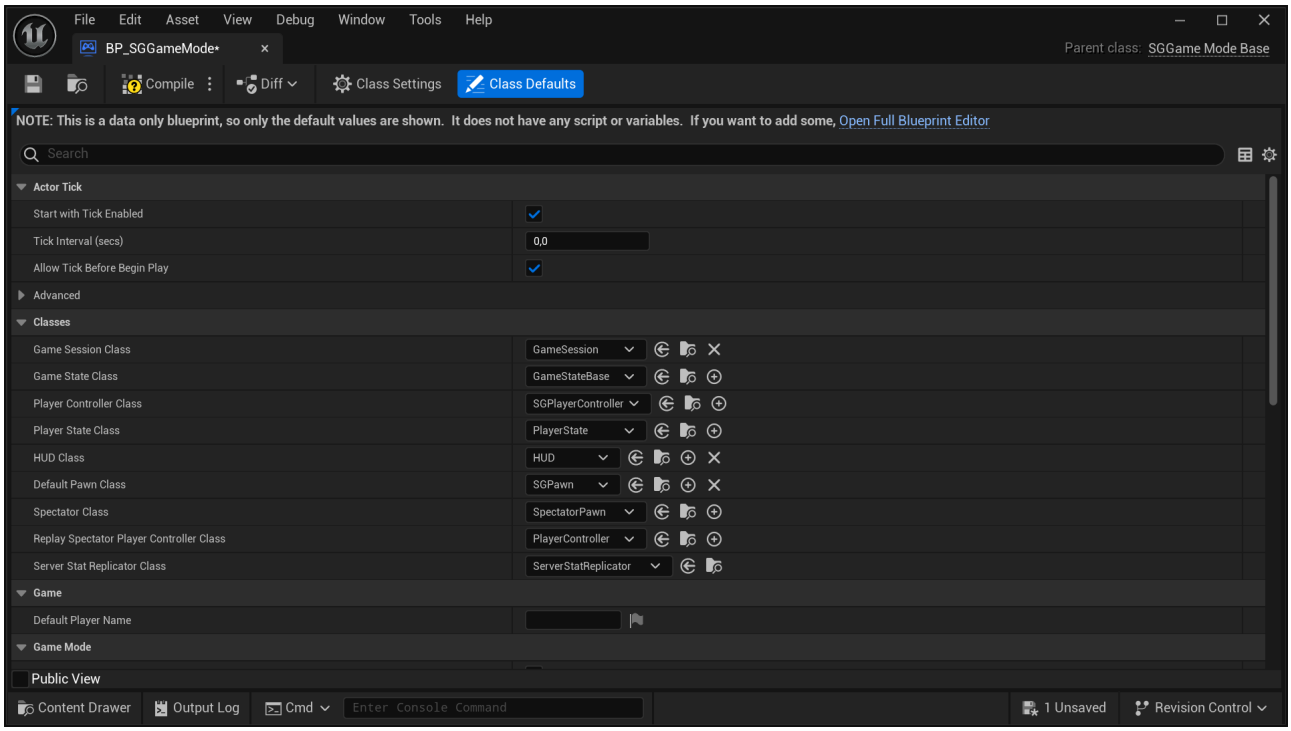


6. Finally, set your newly created subclass of `SGGameModeBase` as the `Default GameMode`. You can do this by navigating to `Project Settings > Project > Maps & Modes > Default Modes > Default GameMode`.



Setting Up SGPawn

Depening on the Unreal Engine version and your project's type and configuration, you might be able to set `SGPawn` as the `Default Pawn Class` by navigating to `Project Settings > Project > Maps & Modes > Default Modes > Selected GameMode > Default Pawn Class`. However, regardless of the engine version or project type and configuration, you can always configure this by opening your `Default GameMode` and setting the `Default Pawn Class` directly from there. Once set, click on the `Compile` button and save your game mode Blueprint asset.



> **Tip**
>
> For greater control and customization, consider extending the SGPawn.

> **Caution**
>
> Setting `SGPawn` or a subclass of it as the `Default Pawn Class` without setting `SGPlayerController` or a subclass of it as the default `Player Controller Class`

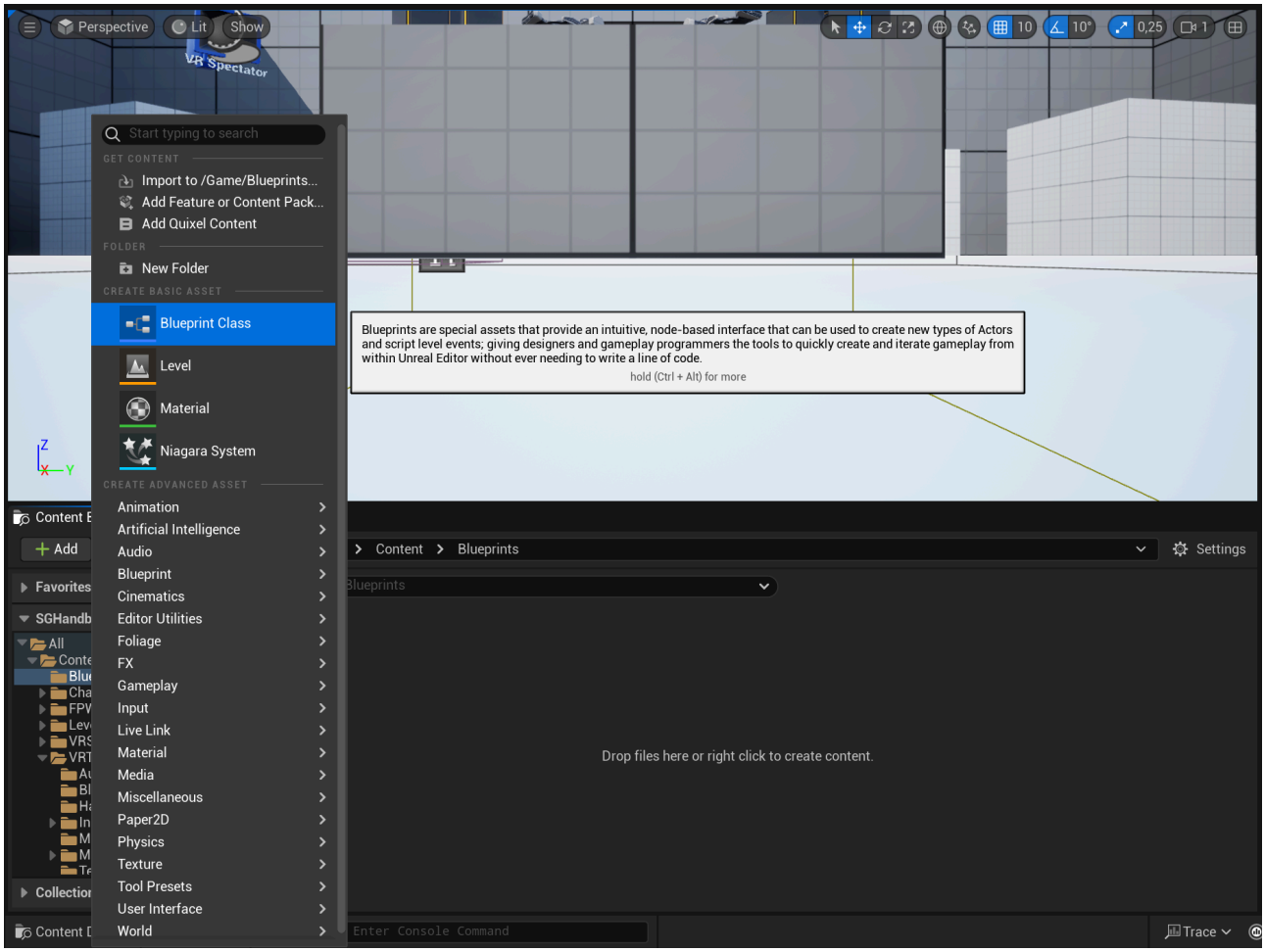
will cause the `SGPawn` to not function properly. So, it's a strict requirement.

> **Important**
>
> To have a fully functional `SGPawn`, simply setting it up is not enough. You still need to setup the Virtual Hand Meshes and setup the Wrist Tracking Hardware.

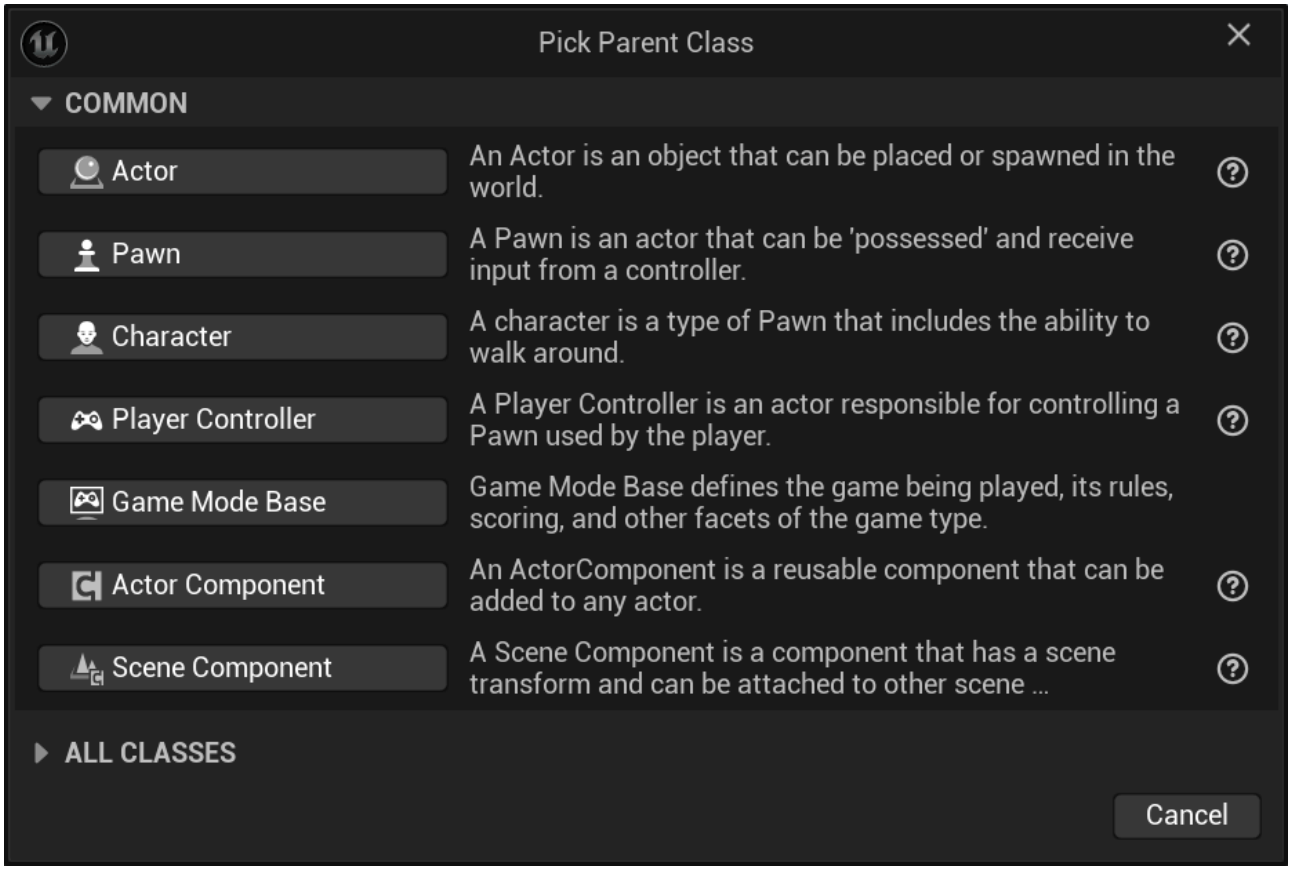
Extending SGPawn

Follow these steps to extend and set up your own version of `SGPawn`:

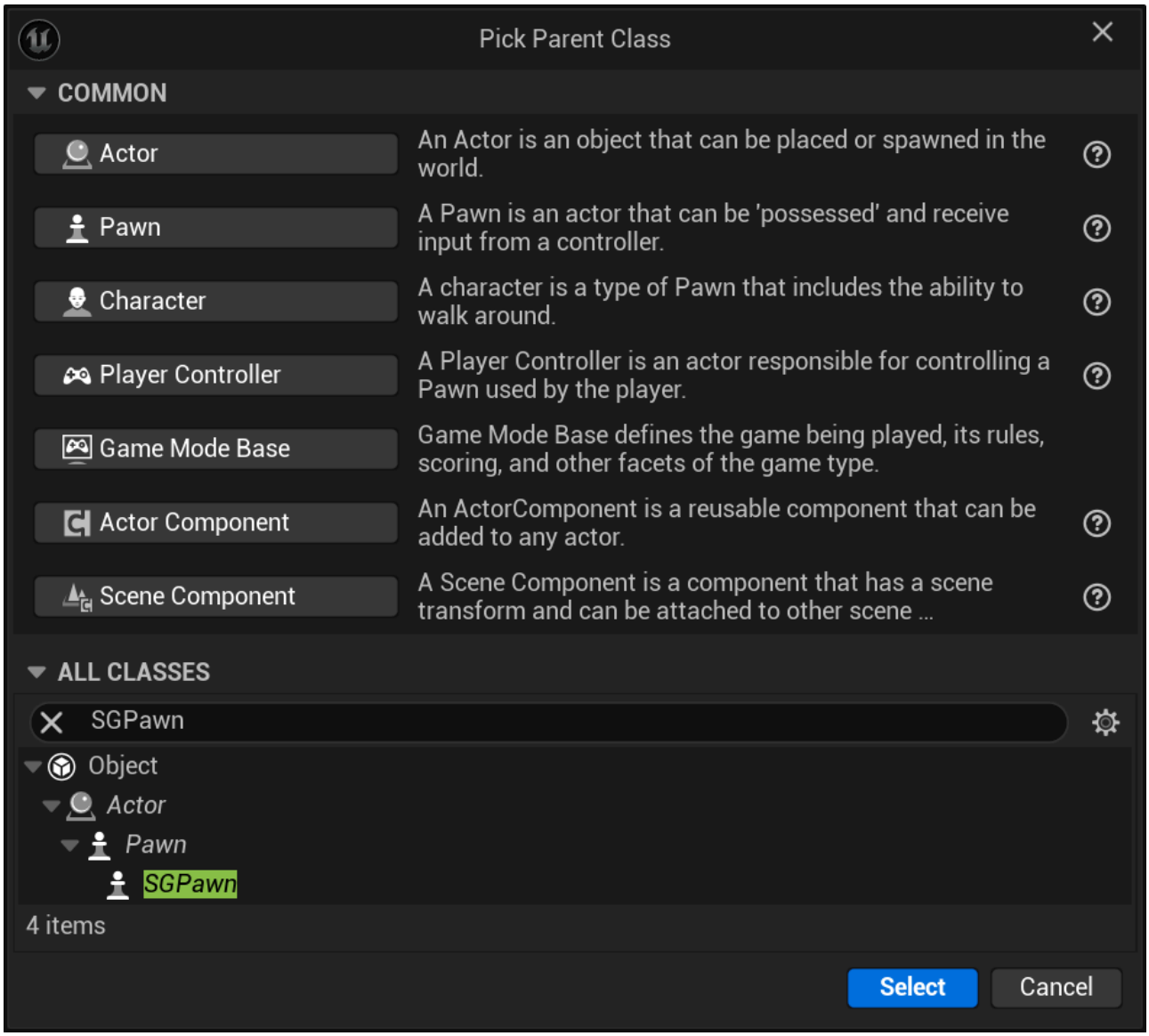
1. In the Content Browser, click the `+ Add` button, then select `Blueprint Class from the menu`. Alternatively, right-click inside the Content Browser and choose `Blueprint Class` from the context menu.



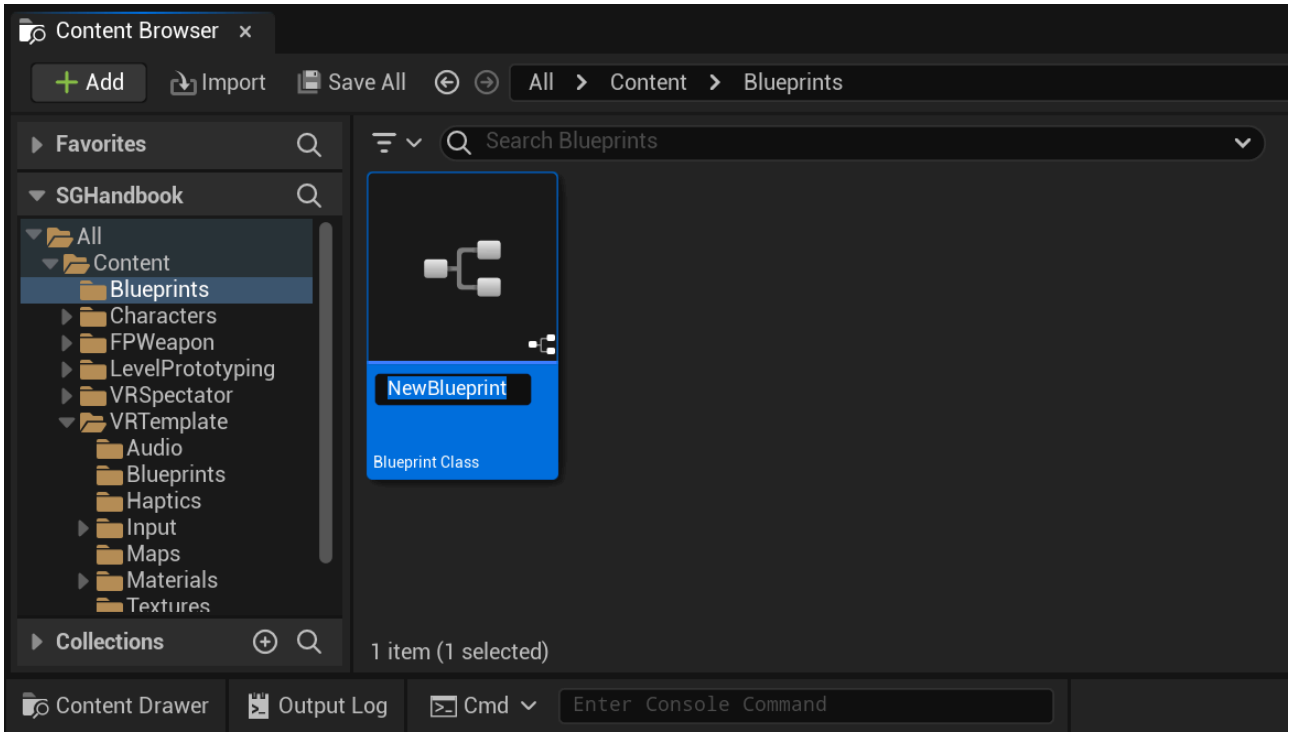
2. A dialog will appear asking you to choose a parent class. Click on the ALL CLASSES section to expand the list of available classes.



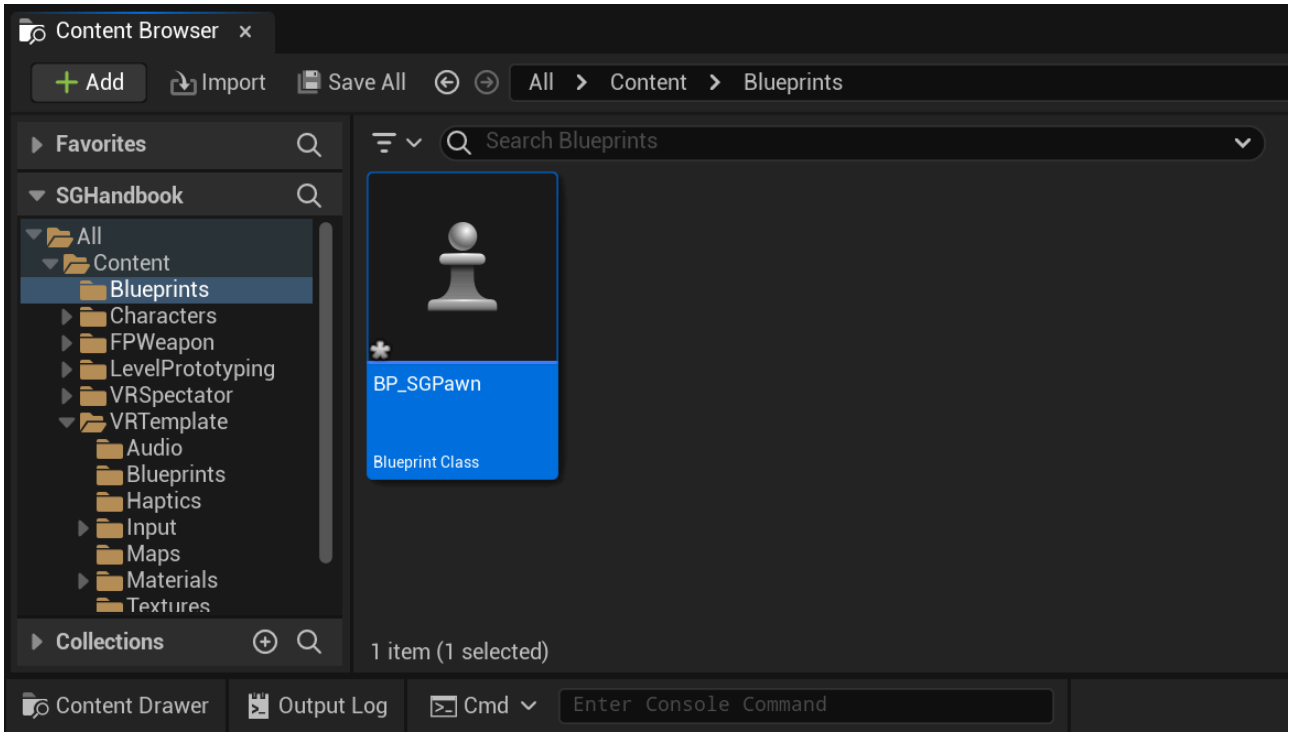
3. In the expanded `ALL CLASSES` section, start typing `SGPawn` in the Search box. When `SGPawn` appears, select it and click the `Select` button to create your new Blueprint class based on it.



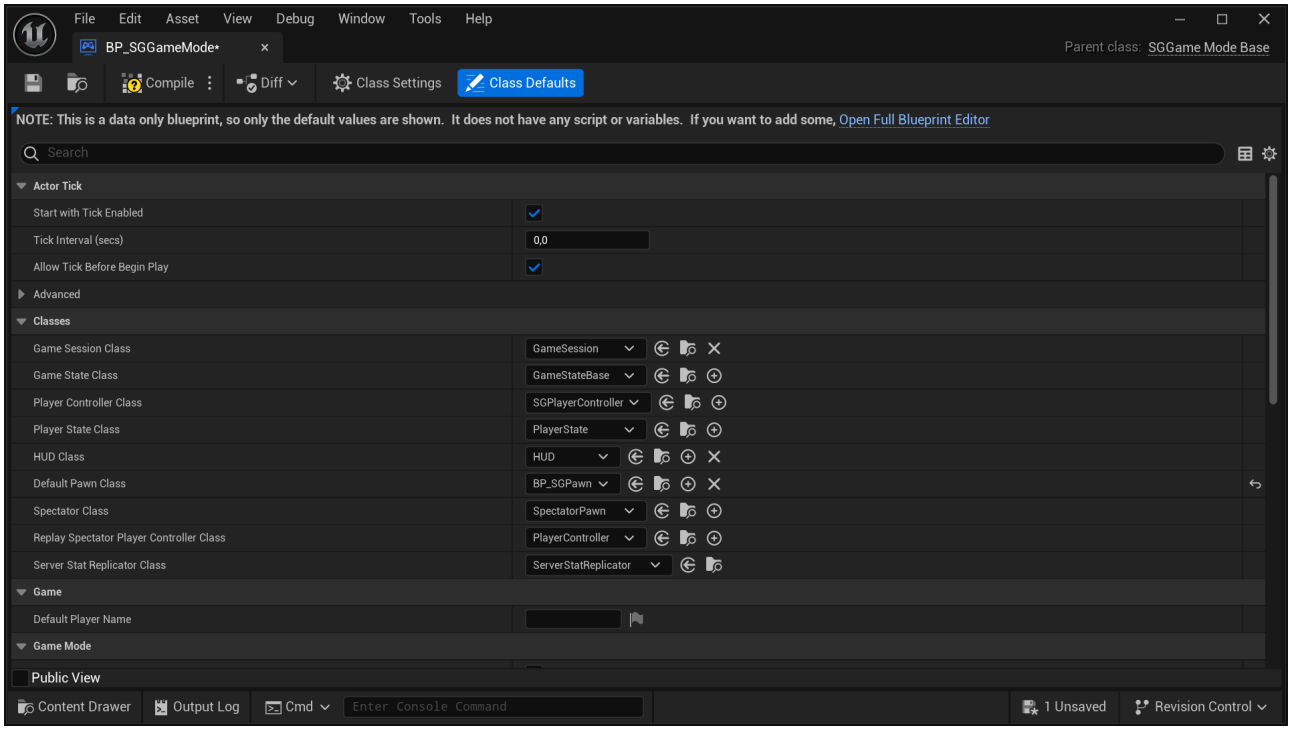
4. After returning to the Content Browser, the Unreal Editor will prompt you to rename `NewBlueprint` to your desired class name. You can rename the class at any time by pressing `F2` or by right-clicking on it and selecting `Rename` from the context menu.



5. Once you have renamed the `NewBlueprint` class to your desired name, click on `Save All` to save the new class to disk.



6. Finally, set your newly created subclass of `SGPawn` as the `Default Pawn Class`. Depening on the Unreal Engine version and your project's type and configuration, you might be able do this by navigating to `Project Settings > Project > Maps & Modes > Default Modes > Selected GameMode > Default Pawn Class`. However, regardless of the engine version or project type and configuration, you can always configure this by opening your `Default GameMode` and setting the `Default Pawn Class` directly from there. Once set, click on the `Compile` button and save your game mode Blueprint asset.



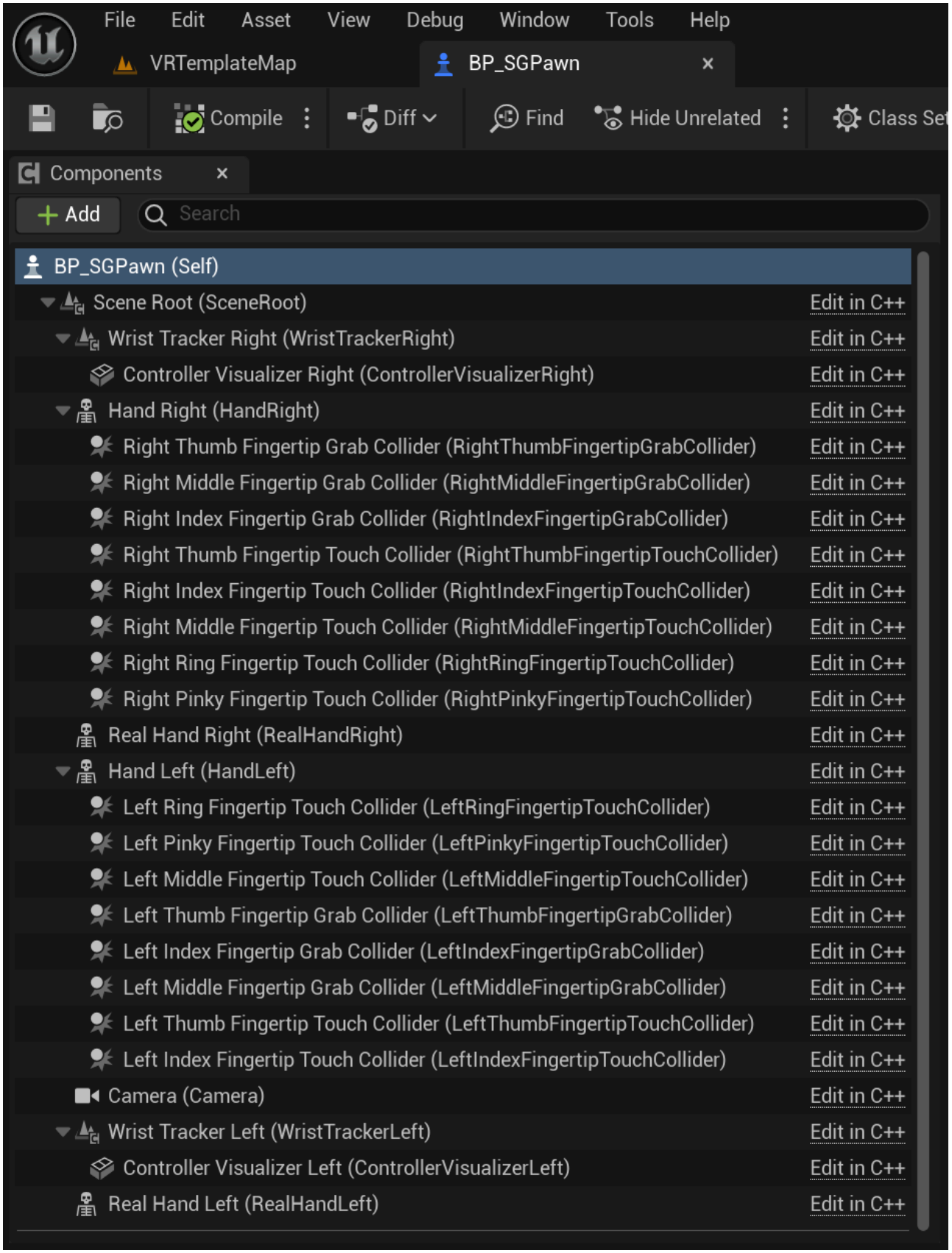
> **Important**
>
> To have a fully functional `SGPawn`, simply setting it up is not enough. You still need to setup the Virtual Hand Meshes and setup the Wrist Tracking Hardware.

Customizing SGPawn

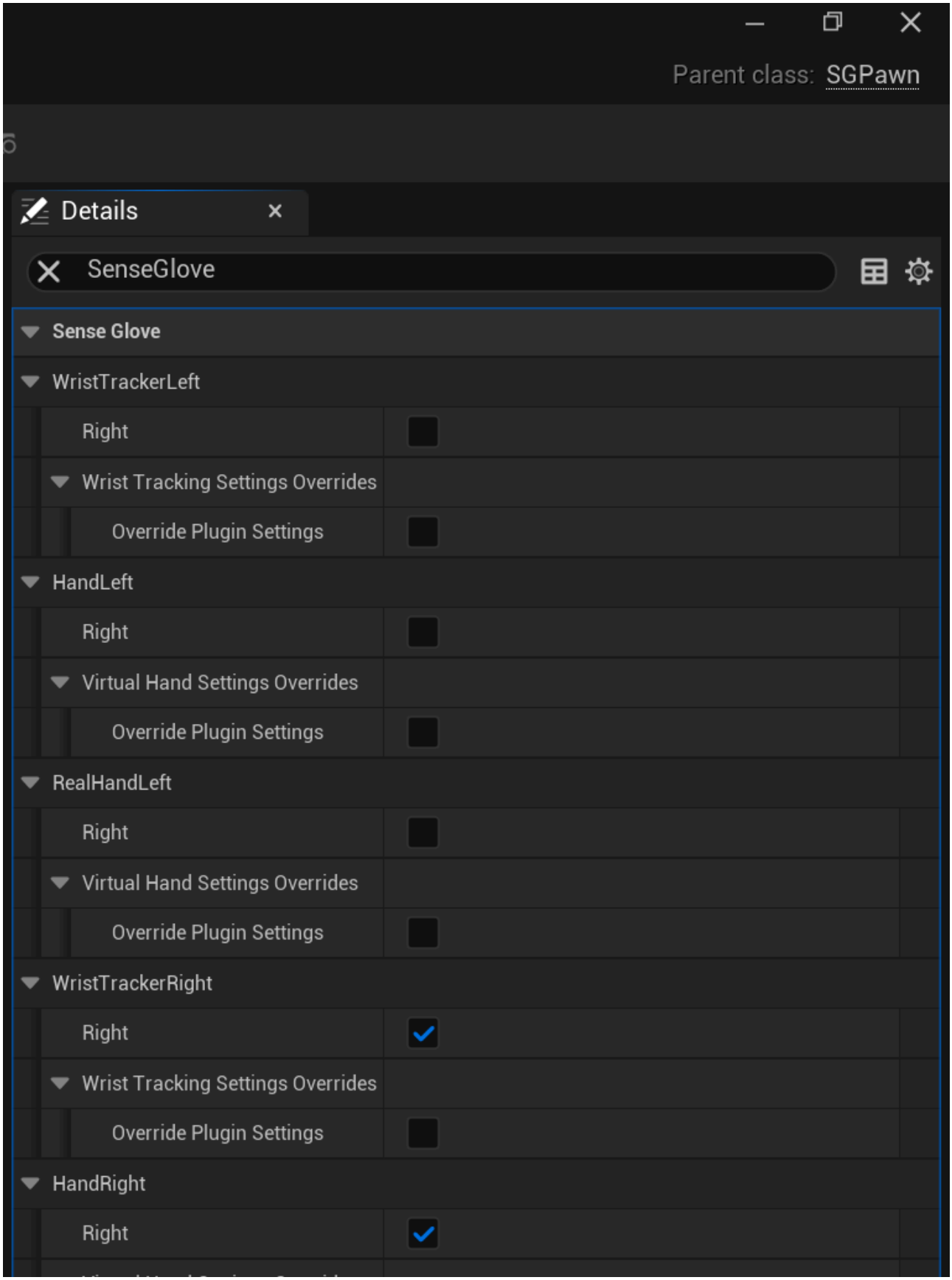
Customizing the `SGPawn` after subclassing is straightforward and flexible.

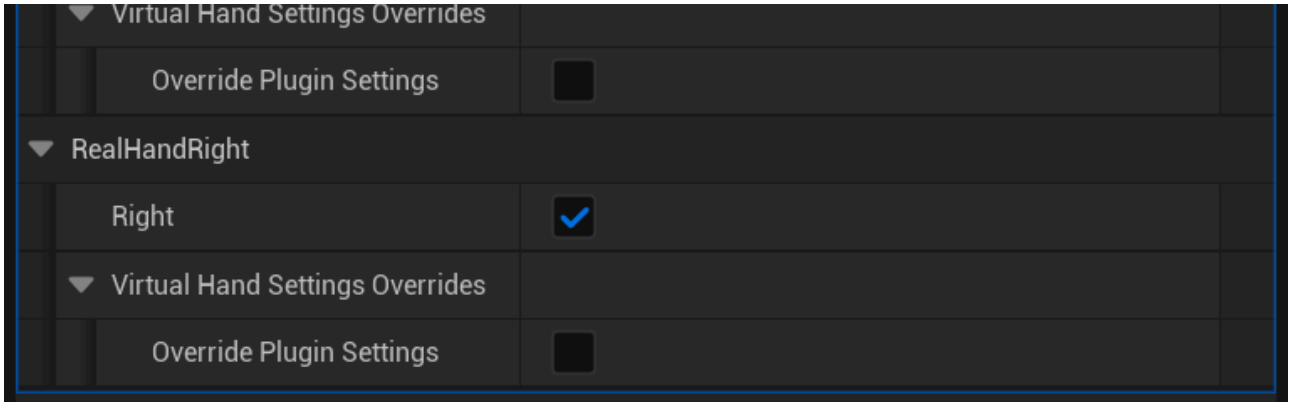
The `SGPawn` class includes several key subcomponents:

- `Wrist Tracker Left` and `Wrist Tracker Right` of type `SGWristTrackerComponent`.
- `HandLeft` and `HandRight` of type `SGVirtualHandComponent` and represent the virtual hand models visible to the user in the simulation.
- `RealHandLeft` and `RealHandRight` of type `SGVirtualHandComponent`. By default, these are hidden and represent the real hands within the simulation. These components are useful if you need to separate the rendering of the virtual hands from the real hands. For instance, the virtual hands typically have collisions and cannot pass through objects, while the real hands are not constrained in this way.


File
Edit
Asset
View
Debug
Window
Tools
Help
VRTemplateMap
BP_SGPawn
Compile
Diff
Find
Hide Unrelated
Class Se
Components
Add
Search
BP_SGPawn (Self)
Scene Root (SceneRoot)
Wrist Tracker Right (WristTrackerRight)
Controller Visualizer Right (ControllerVisualizerRight)
Hand Right (HandRight)
Right Thumb Fingertip Grab Collider (RightThumbFingertipGrabCollider)
Right Middle Fingertip Grab Collider (RightMiddleFingertipGrabCollider)
Right Index Fingertip Grab Collider (RightIndexFingertipGrabCollider)
Right Thumb Fingertip Touch Collider (RightThumbFingertipTouchCollider)
Right Index Fingertip Touch Collider (RightIndexFingertipTouchCollider)
Right Middle Fingertip Touch Collider (RightMiddleFingertipTouchCollider)
Right Ring Fingertip Touch Collider (RightRingFingertipTouchCollider)
Right Pinky Fingertip Touch Collider (RightPinkyFingertipTouchCollider)
Real Hand Right (RealHandRight)
Hand Left (HandLeft)
Left Ring Fingertip Touch Collider (LeftRingFingertipTouchCollider)
Left Pinky Fingertip Touch Collider (LeftPinkyFingertipTouchCollider)
Left Middle Fingertip Touch Collider (LeftMiddleFingertipTouchCollider)
Left Thumb Fingertip Grab Collider (LeftThumbFingertipGrabCollider)
Left Index Fingertip Grab Collider (LeftIndexFingertipGrabCollider)
Left Middle Fingertip Grab Collider (LeftMiddleFingertipGrabCollider)
Left Thumb Fingertip Touch Collider (LeftThumbFingertipTouchCollider)
Left Index Fingertip Touch Collider (LeftIndexFingertipTouchCollider)
Camera (Camera)
Wrist Tracker Left (WristTrackerLeft)
Controller Visualizer Left (ControllerVisualizerLeft)
Real Hand Left (RealHandLeft)
Edit in C++

Also, it's possible to filter the properties for these SenseGlove components inside the `Details` panel inside the `SGPawn` Blueprint Editor by typing the word `SenseGlove` inside `Search` box of the `Details` panel.

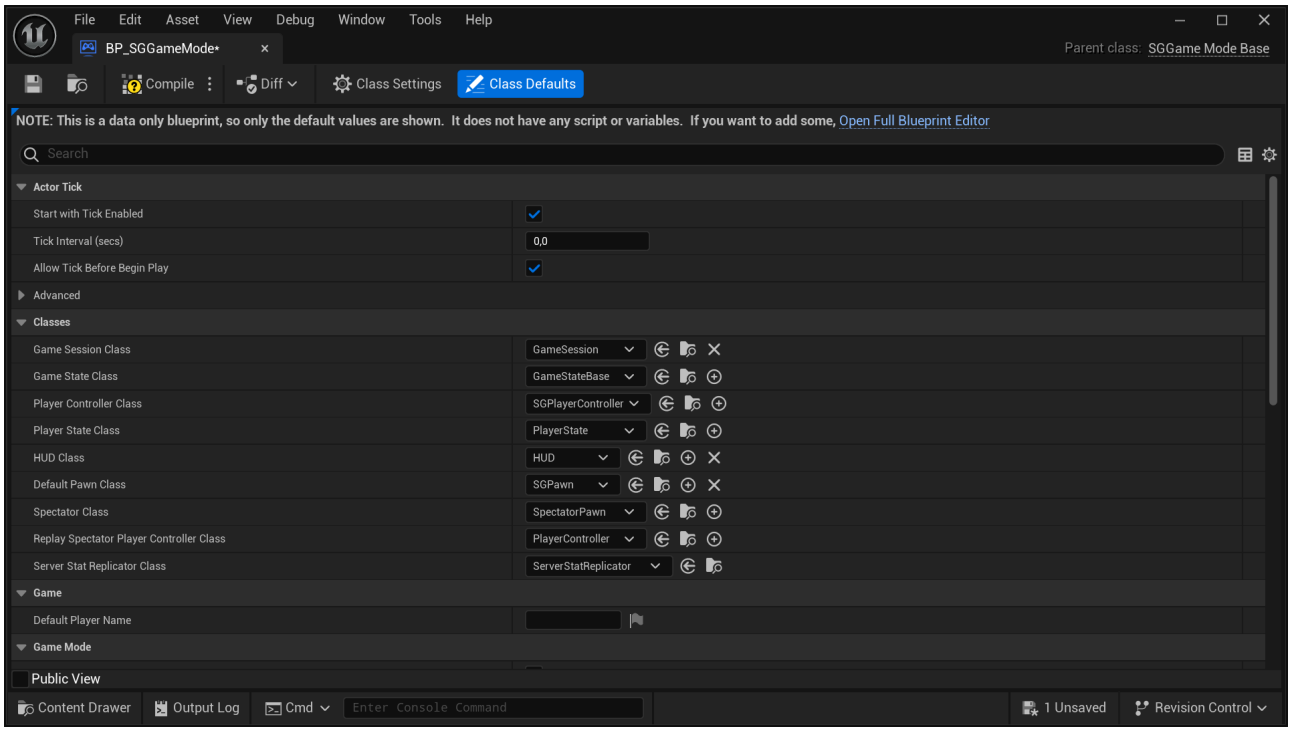

Parent class: SGPawn
Details
SenseGlove
Sense Glove
WristTrackerLeft
Right
Wrist Tracking Settings Overrides
Override Plugin Settings
HandLeft
Right
Virtual Hand Settings Overrides
Override Plugin Settings
RealHandLeft
Right
Virtual Hand Settings Overrides
Override Plugin Settings
WristTrackerRight
Right
Wrist Tracking Settings Overrides
Override Plugin Settings
HandRight
Right



Please visit how to setup the Virtual Hand Meshes, The Virtual Hand Mesh Settings, and how to setup the Wrist Tracking Hardware sections for more information.

Setting Up SGPlayerController

Depening on the Unreal Engine version and your project's type and configuration, you might be able to set `SGPlayerController` as the default `Player Controller Class` by navigating to `Project Settings > Project > Maps & Modes > Default Modes > Selected GameMode > Player Controller Class`. However, regardless of the engine version or project type and configuration, you can always configure this by opening your `Default GameMode` and setting the default `Player Controller Class` directly from there. Once set, click on the `Compile` button and save your game mode Blueprint asset.



> **Tip**
>
> For greater control and customization, consider extending the SGPlayerController.

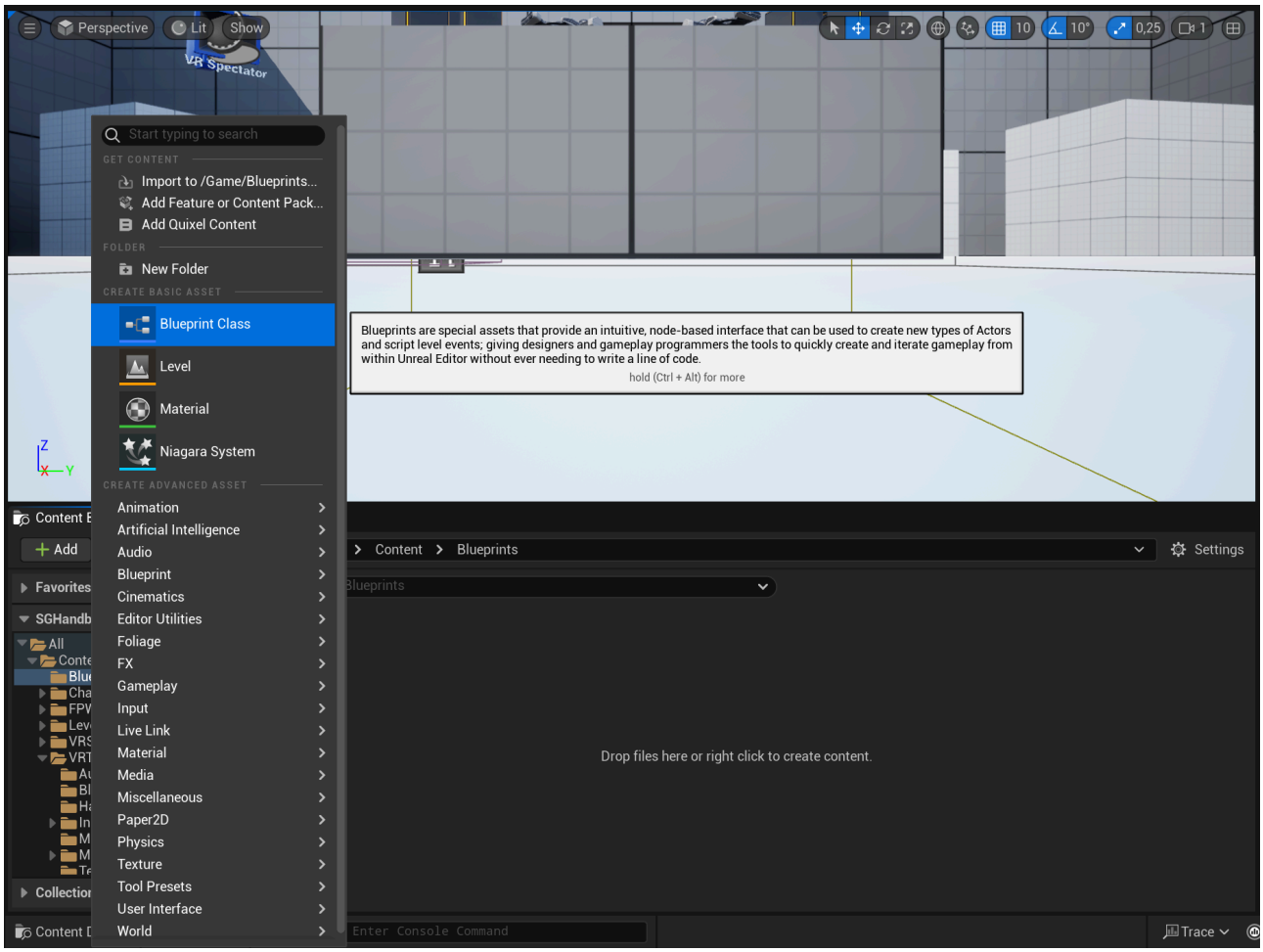
> **Caution**

> Setting `SGPlayerController` or a subclass of it as the default `Player Controller Class` without setting `SGPawn` or a subclass of it as the `Default Pawn Class` will cause your simulation or editor to crash. So, it's a strict requirement.

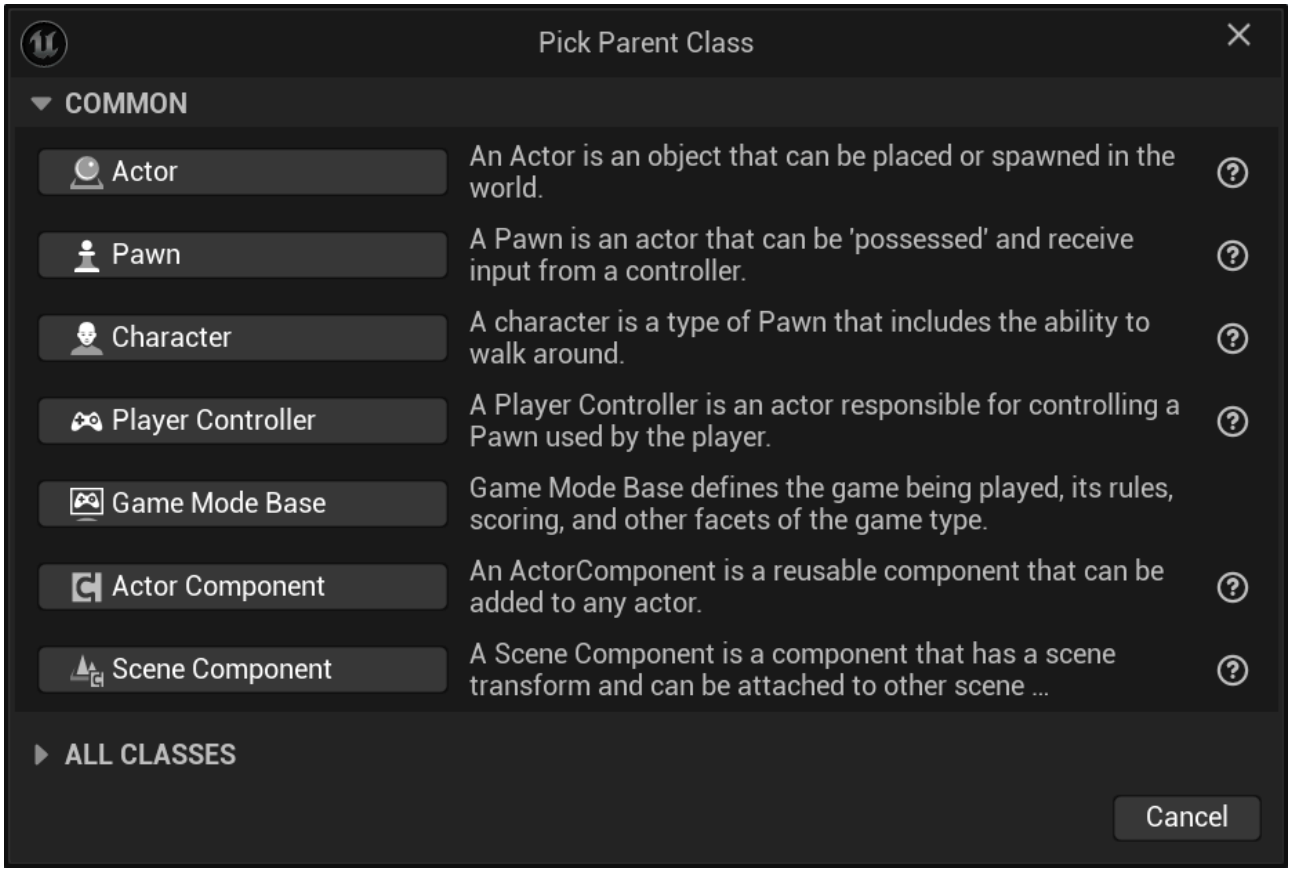
Extending SGPlayerController

Follow these steps to extend and set up your own version of `SGPlayerController` :

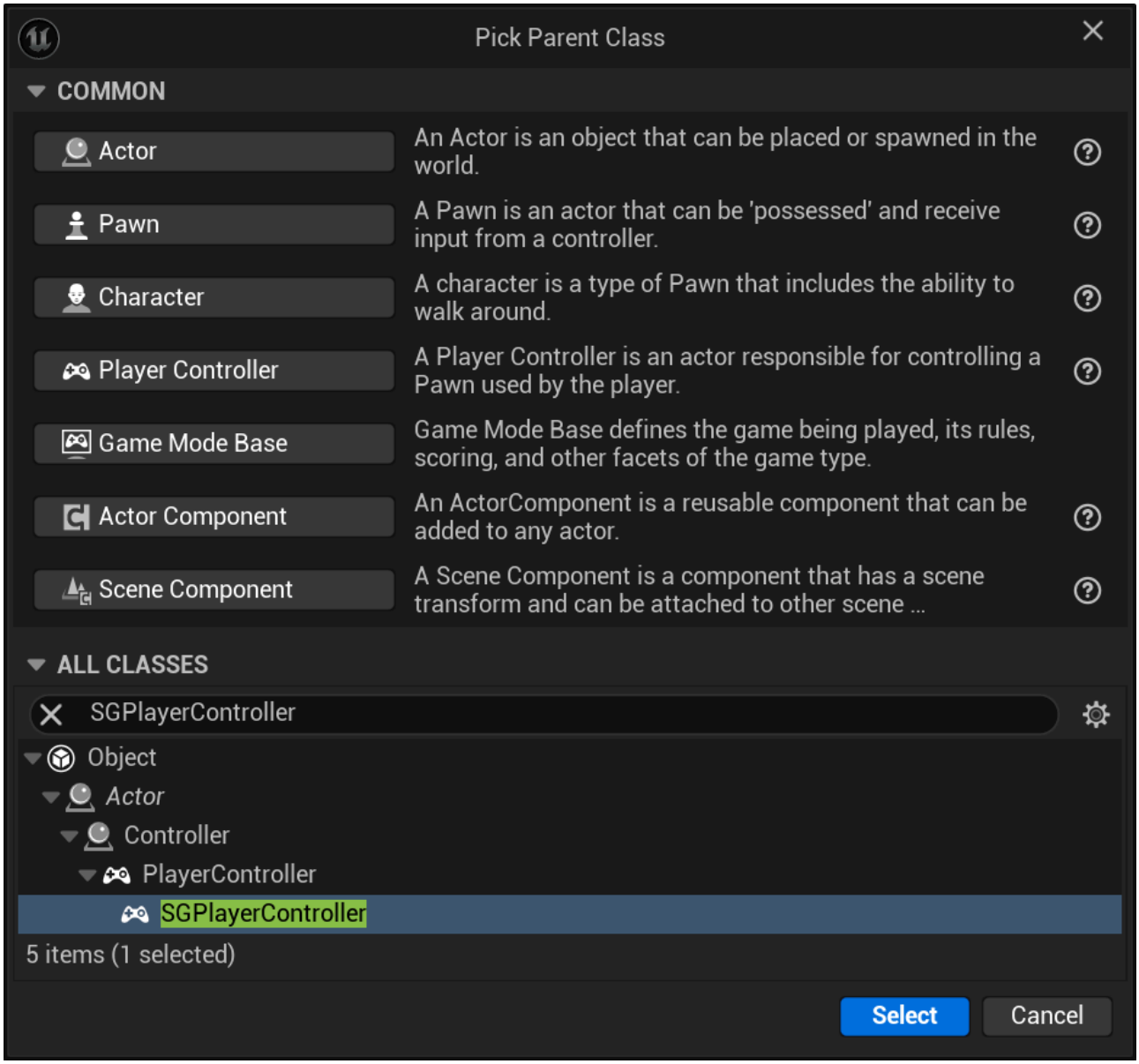
1. In the Content Browser, click the `+ Add` button, then select `Blueprint Class from the menu` . Alternatively, right-click inside the Content Browser and choose `Blueprint Class` from the context menu.



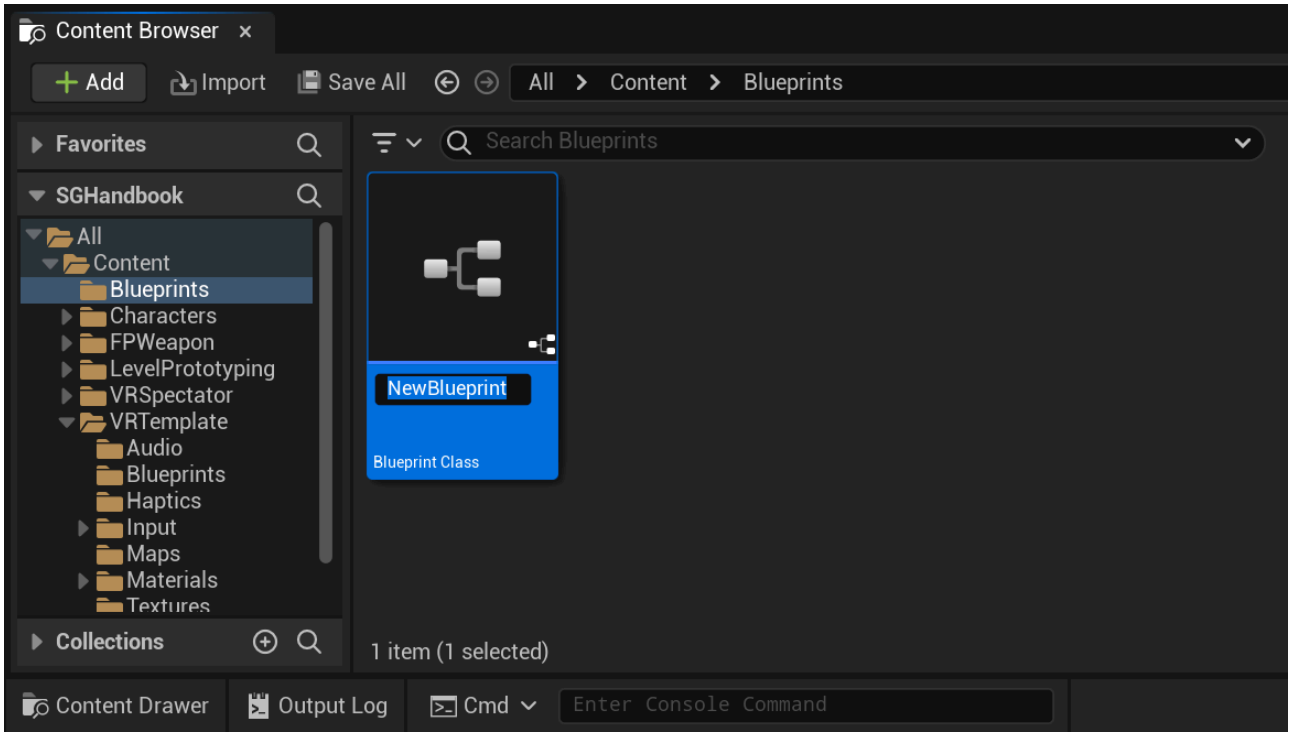
2. A dialog will appear asking you to choose a parent class. Click on the `ALL CLASSES` section to expand the list of available classes.



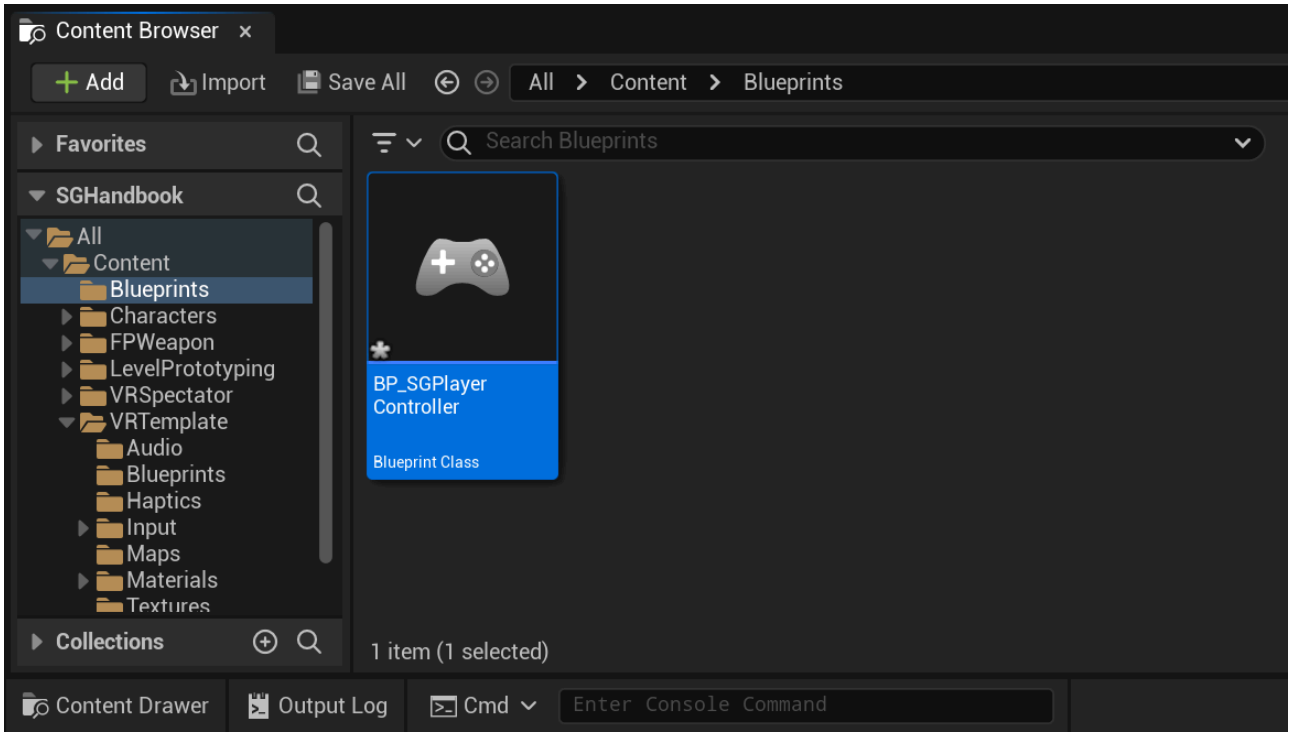
3. In the expanded `ALL CLASSES` section, start typing `SGPlayerController` in the Search box. When `SGPlayerController` appears, select it and click the `Select` button to create your new Blueprint class based on it.



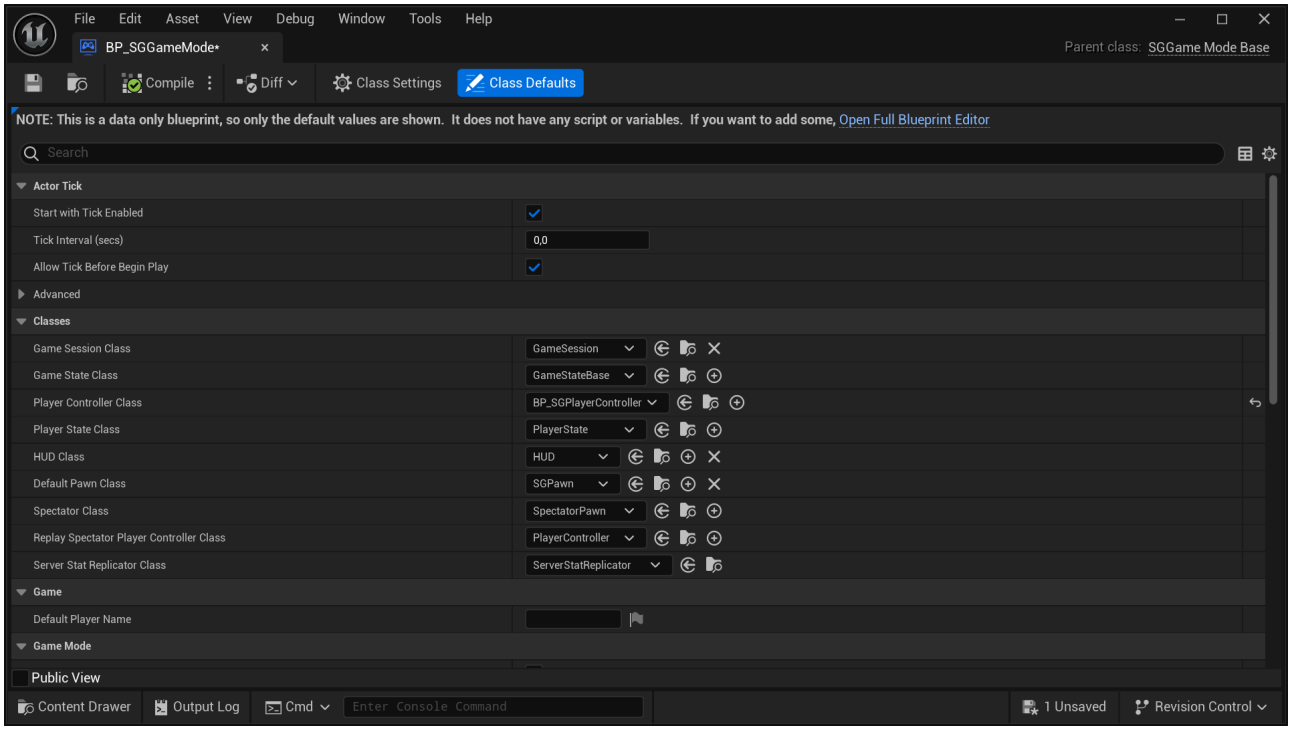
4. After returning to the Content Browser, the Unreal Editor will prompt you to rename `NewBlueprint` to your desired class name. You can rename the class at any time by pressing `F2` or by right-clicking on it and selecting `Rename` from the context menu.



5. Once you have renamed the `NewBlueprint` class to your desired name, click on `Save All` to save the new class to disk.

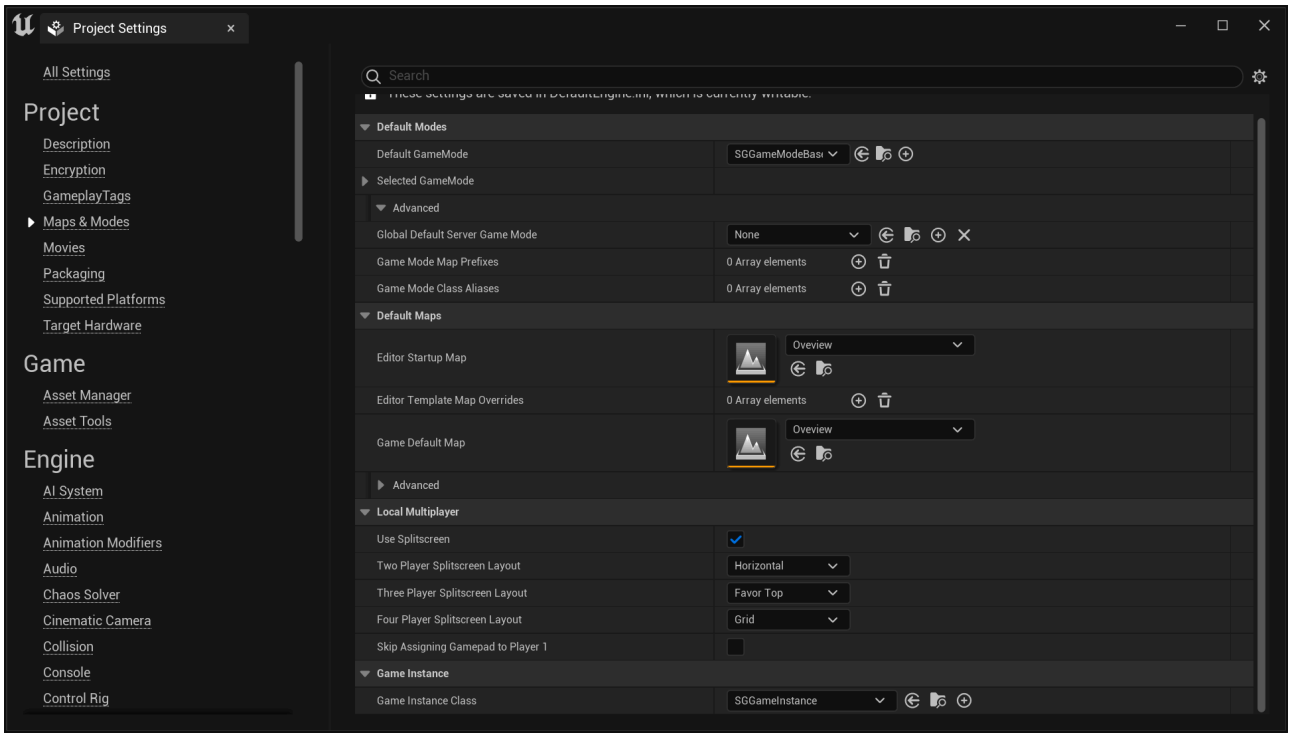


6. Finally, set your newly created subclass of `SGPlayerController` as the default `Player Controller Class`. Depening on the Unreal Engine version and your project's type and configuration, you might be able do this by navigating to `Project Settings > Project > Maps & Modes > Default Modes > Selected GameMode > Player Controller Class`. However, regardless of the engine version or project type and configuration, you can always configure this by opening your `Default GameMode` and setting the default `Player Controller Class` directly from there. Once set, click on the `Compile` button and save your game mode Blueprint asset.



Setting Up SGGameInstance

Setting `SGGameInstance` as the default `Game Instance Class` is very straightforward. You can do this by navigating to `Project Settings > Project > Maps & Modes > Game Instance > Game Instance Class`.



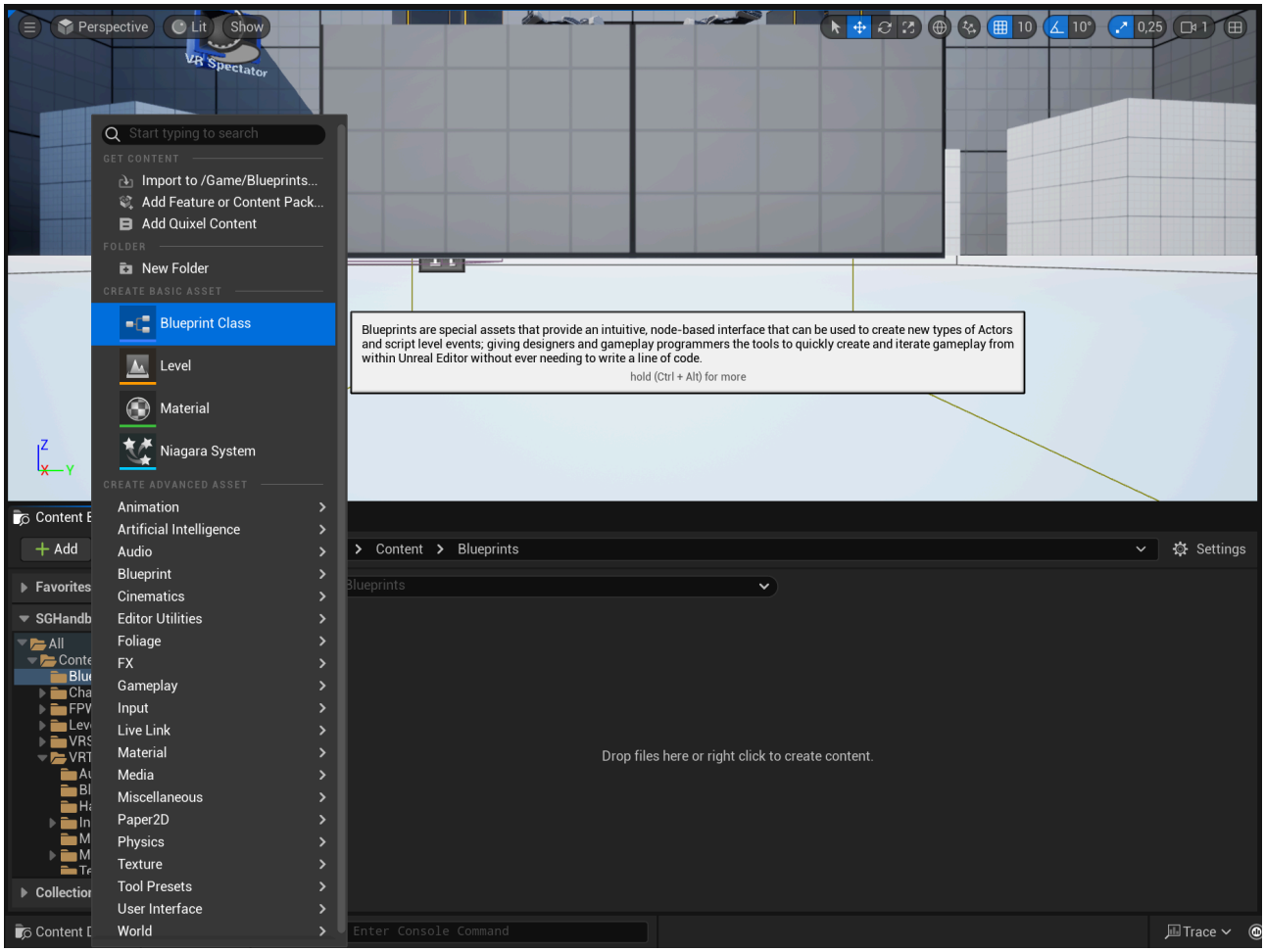
> **Tip**
>
> For greater control and customization, consider extending the SGGameInstance.

> **Important**
>
> Currently, setting `SGGameModeBase` or a subclass of it as the default `Game Instance Class` is not a strict requirement. However, if you intend to use any SenseGlove console command it becomes mandatory. If not set, SenseGlove console commands will not be recognized by Unreal Engine.

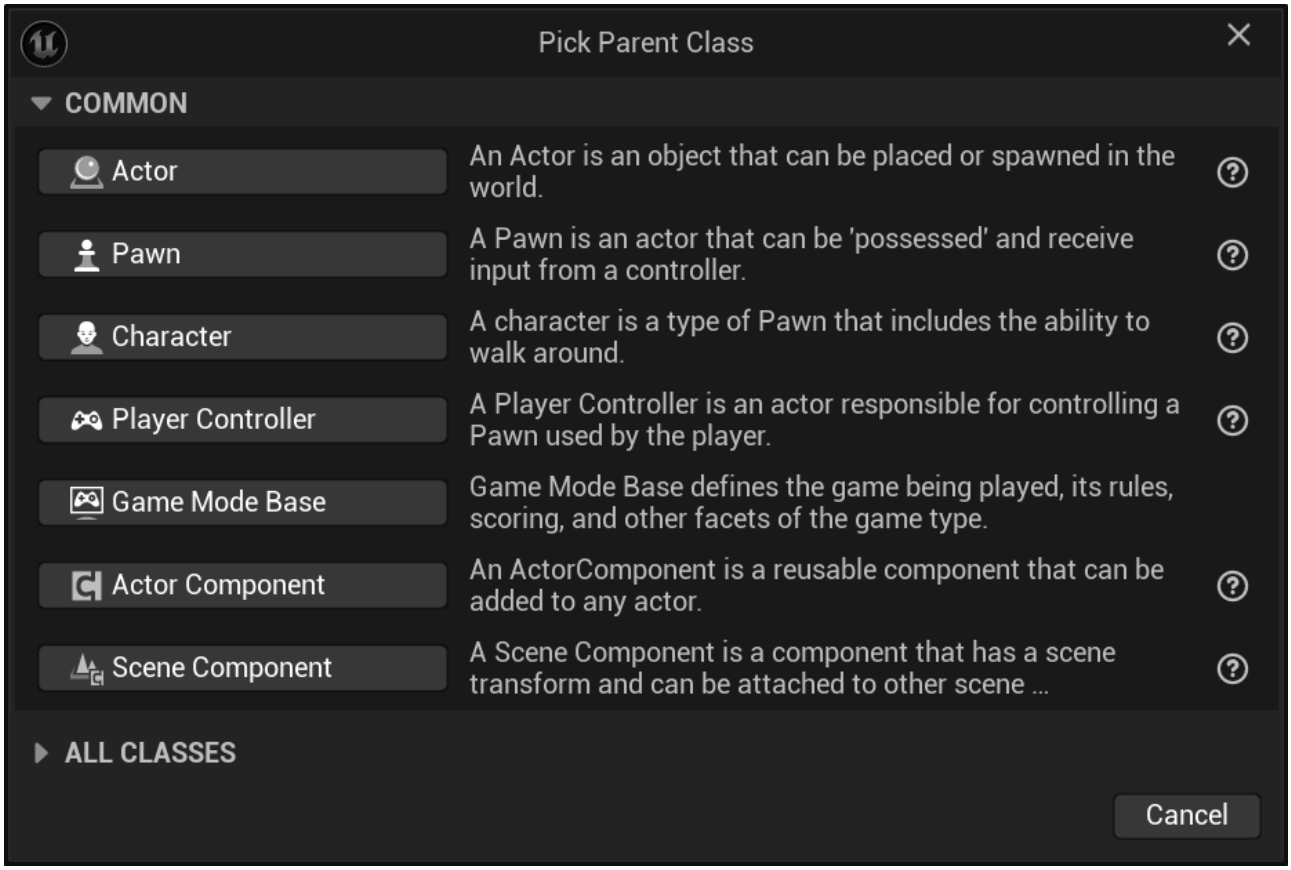
Extending SGGameInstance

Follow these steps to extend and set up your own version of `SGGameInstance`:

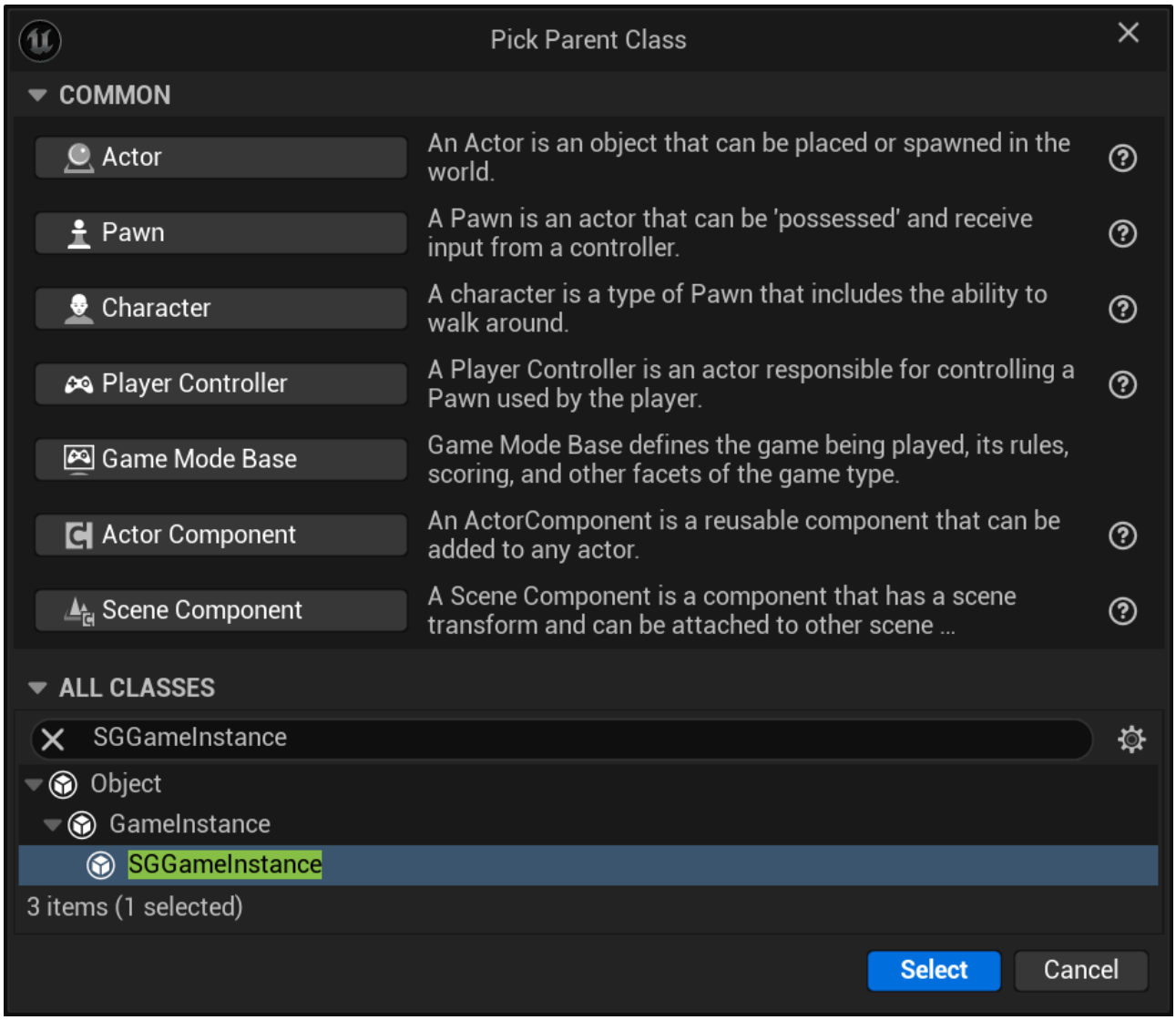
1. In the Content Browser, click the `+ Add` button, then select `Blueprint Class from the menu`. Alternatively, right-click inside the Content Browser and choose `Blueprint Class` from the context menu.



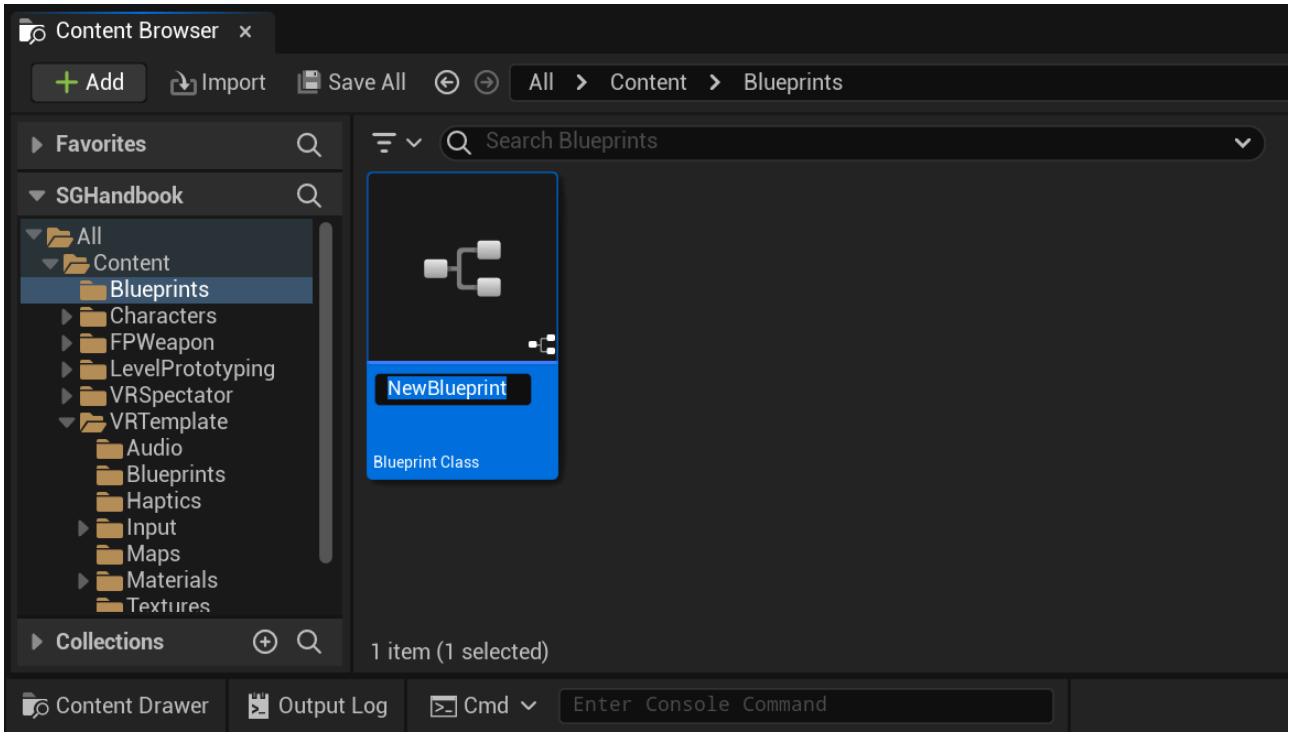
2. A dialog will appear asking you to choose a parent class. Click on the `ALL CLASSES` section to expand the list of available classes.



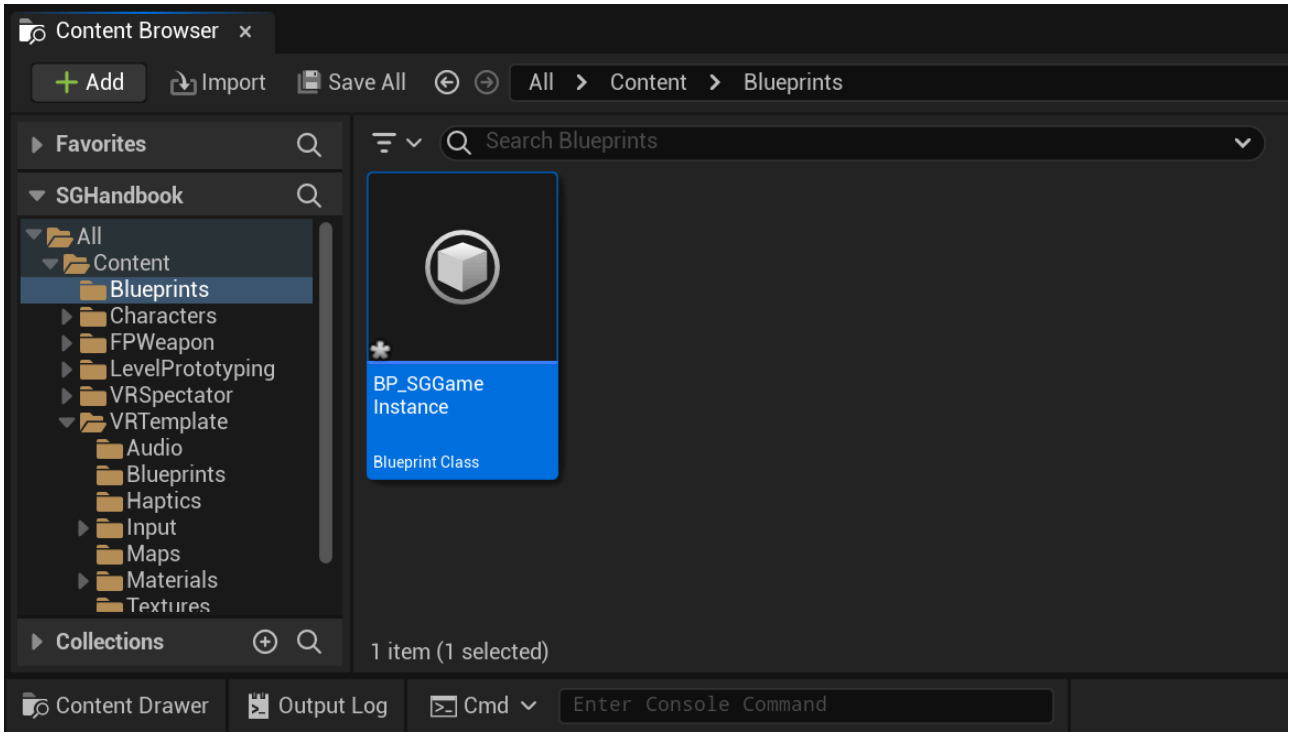
3. In the expanded `ALL CLASSES` section, start typing `SGGameInstance` in the Search box. When `SGGameInstance` appears, select it and click the `Select` button to create your new Blueprint class based on it.



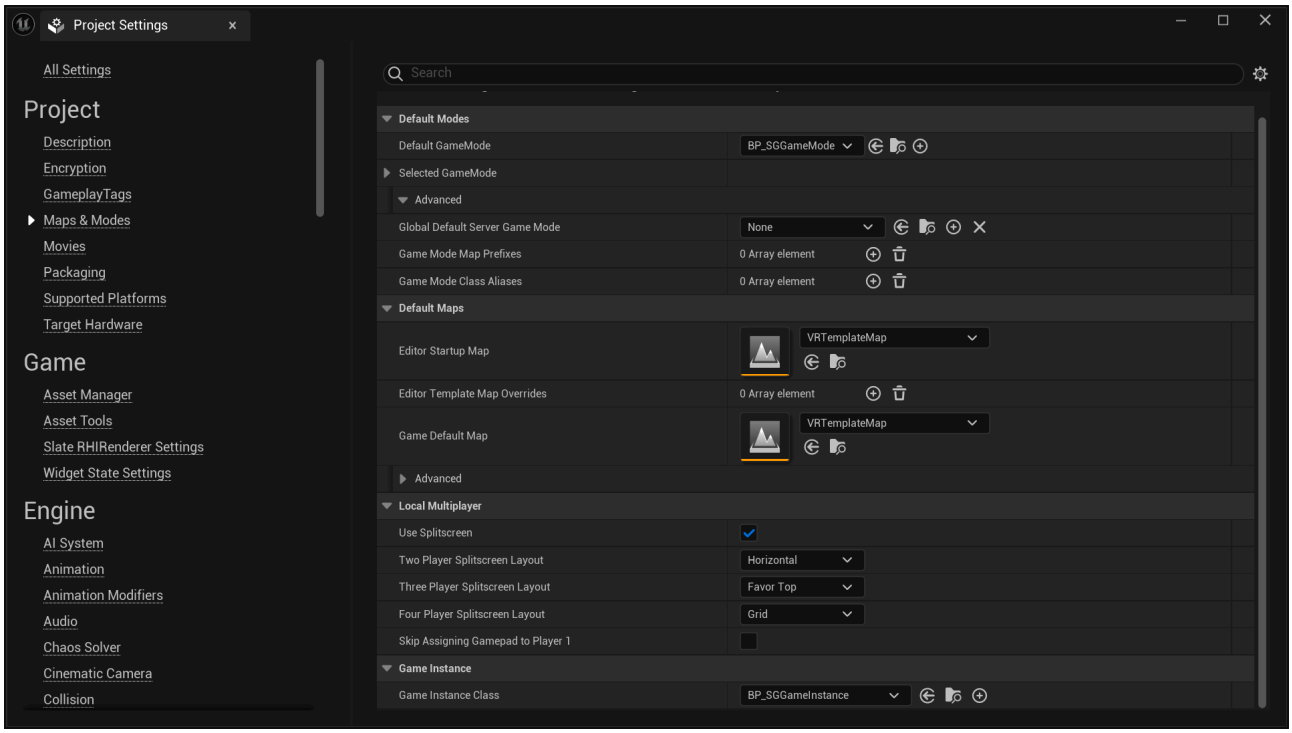
4. After returning to the Content Browser, the Unreal Editor will prompt you to rename `NewBlueprint` to your desired class name. You can rename the class at any time by pressing `F2` or by right-clicking on it and selecting `Rename` from the context menu.



5. Once you have renamed the `NewBlueprint` class to your desired name, click on `Save All` to save the new class to disk.

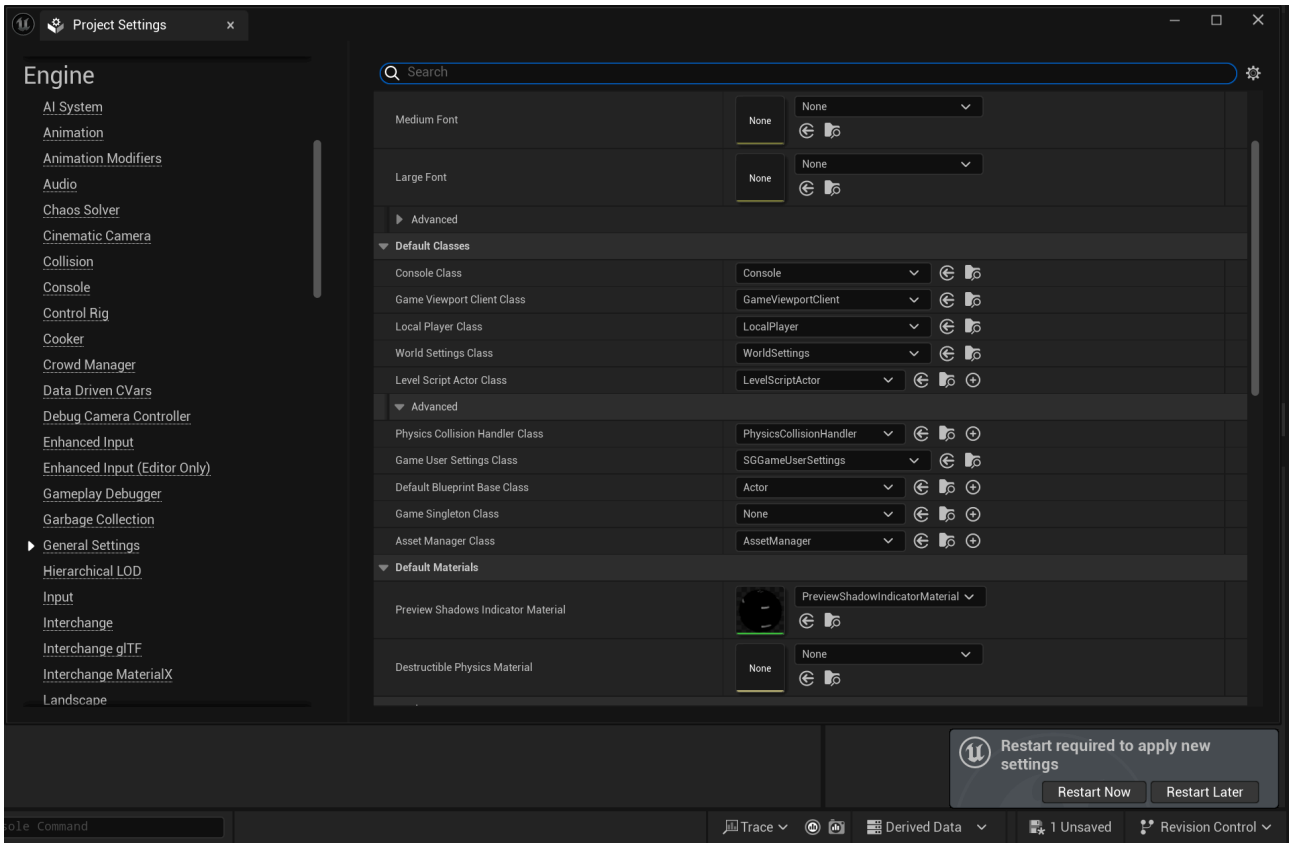


6. Finally, set your newly created subclass of `SGGameInstance` as the default `Game Instance Class`. You can do this by navigating to `Project Settings > Project > Maps & Modes > Game Instance > Game Instance Class`.



Setting Up SGGameUserSettings

Setting `SGGameUserSettings` as the default `Game User Settings Class` is very straightforward. You can do this by navigating to `Project Settings > Engine > General Settings > Default Classes > Advanced > Game User Settings Class`. Once you change the default `Game User Settings Class` the Unreal Editor will prompt you with `Restart required to apply new settings`. For the changes to take effect, click on the `Restart Now` button and wait for the editor to reopen.



> **Tip**
>
> For greater control and customization, consider extending the SGGameUserSettings.

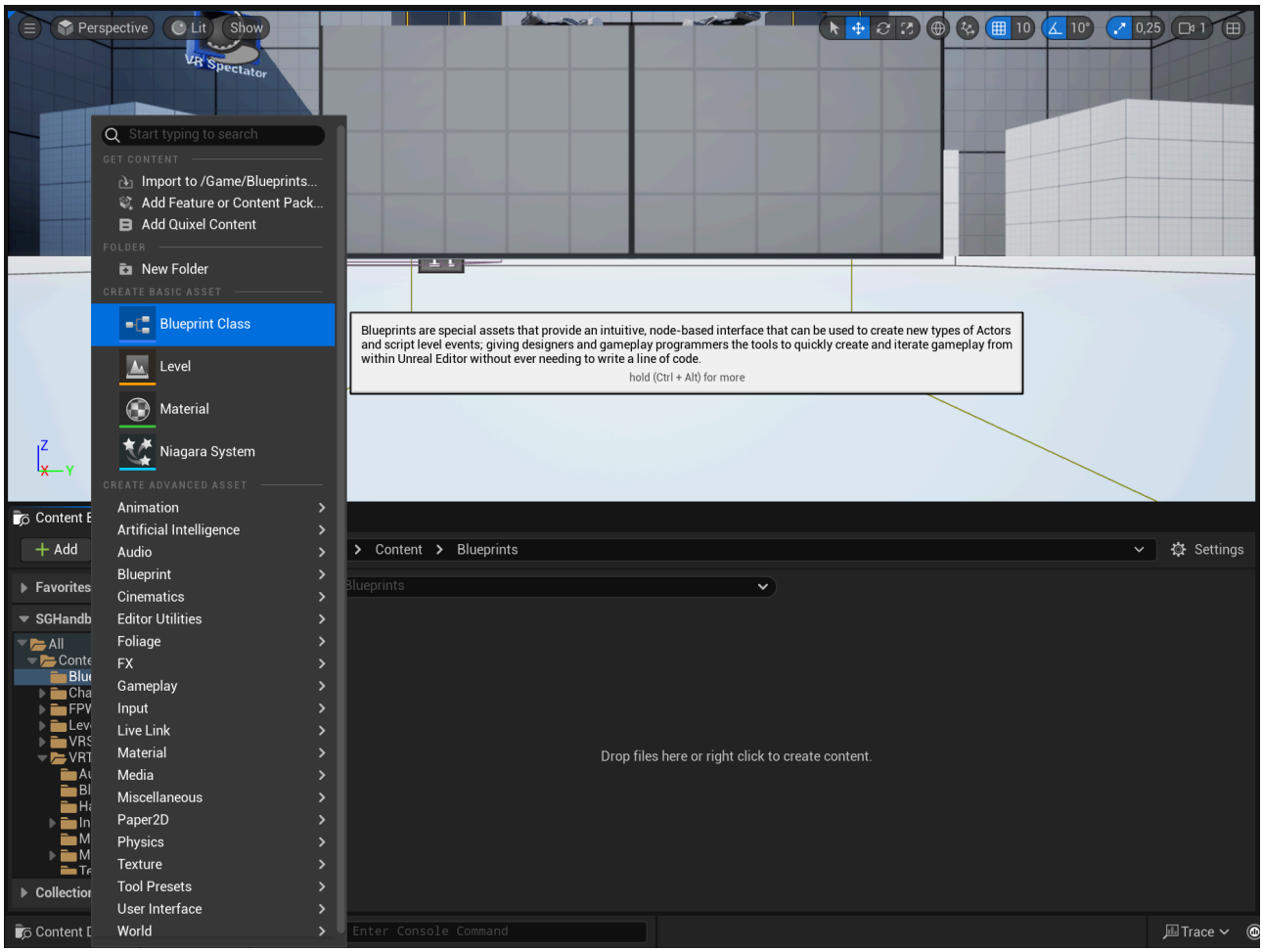
> **Important**

> Currently, setting `SGGameUserSettings` or a subclass of it as the default `Game User Settings Class` is not a strict requirement. However, if you intend to use any SGGameUserSettings-related SenseGlove console command it becomes mandatory. If not set, calling any SGGameUserSettings-related SenseGlove console command will cause your simulation or editor to crash.

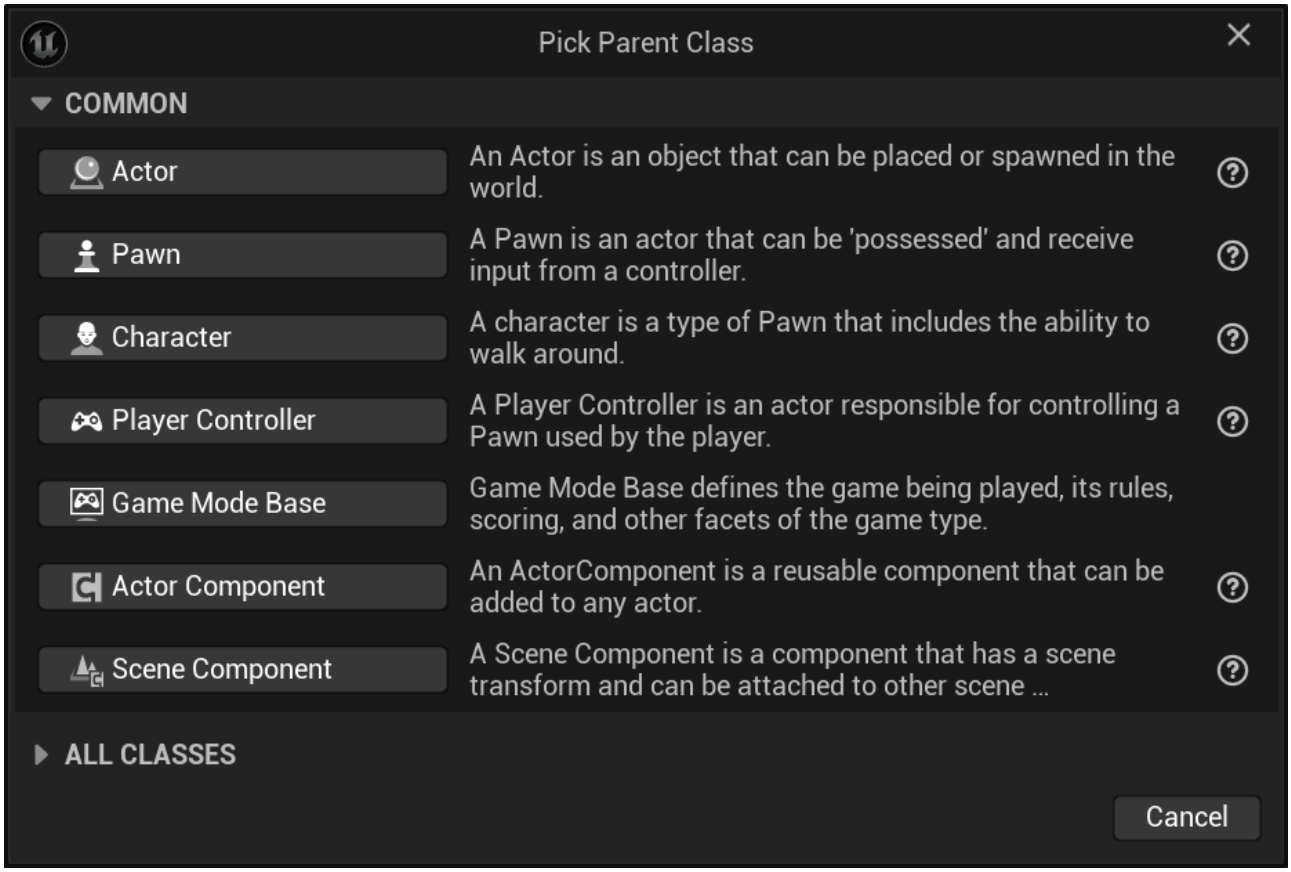
Extending SGGameUserSettings

Follow these steps to extend and set up your own version of `SGGameUserSettings`:

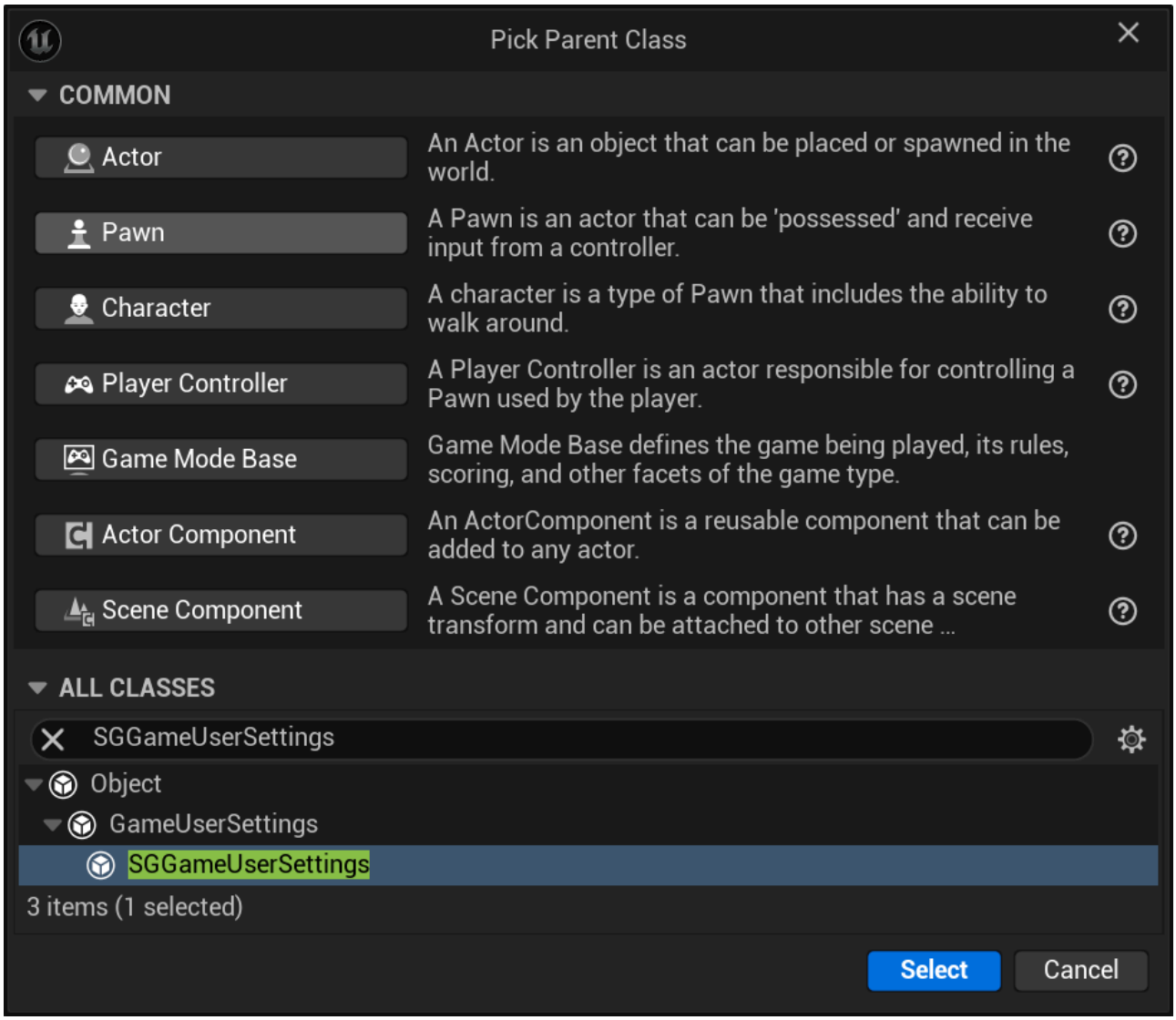
1. In the Content Browser, click the `+ Add` button, then select `Blueprint Class from the menu`. Alternatively, right-click inside the Content Browser and choose `Blueprint Class` from the context menu.



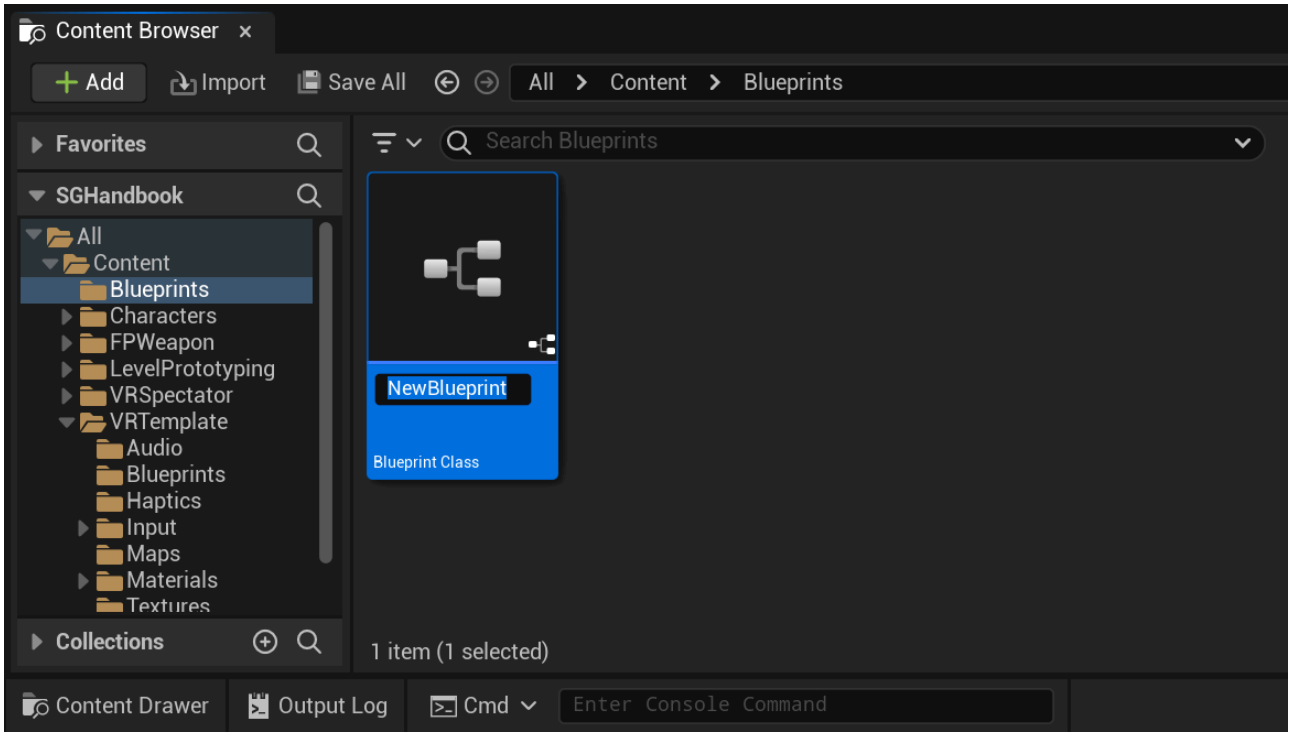
2. A dialog will appear asking you to choose a parent class. Click on the `ALL CLASSES` section to expand the list of available classes.



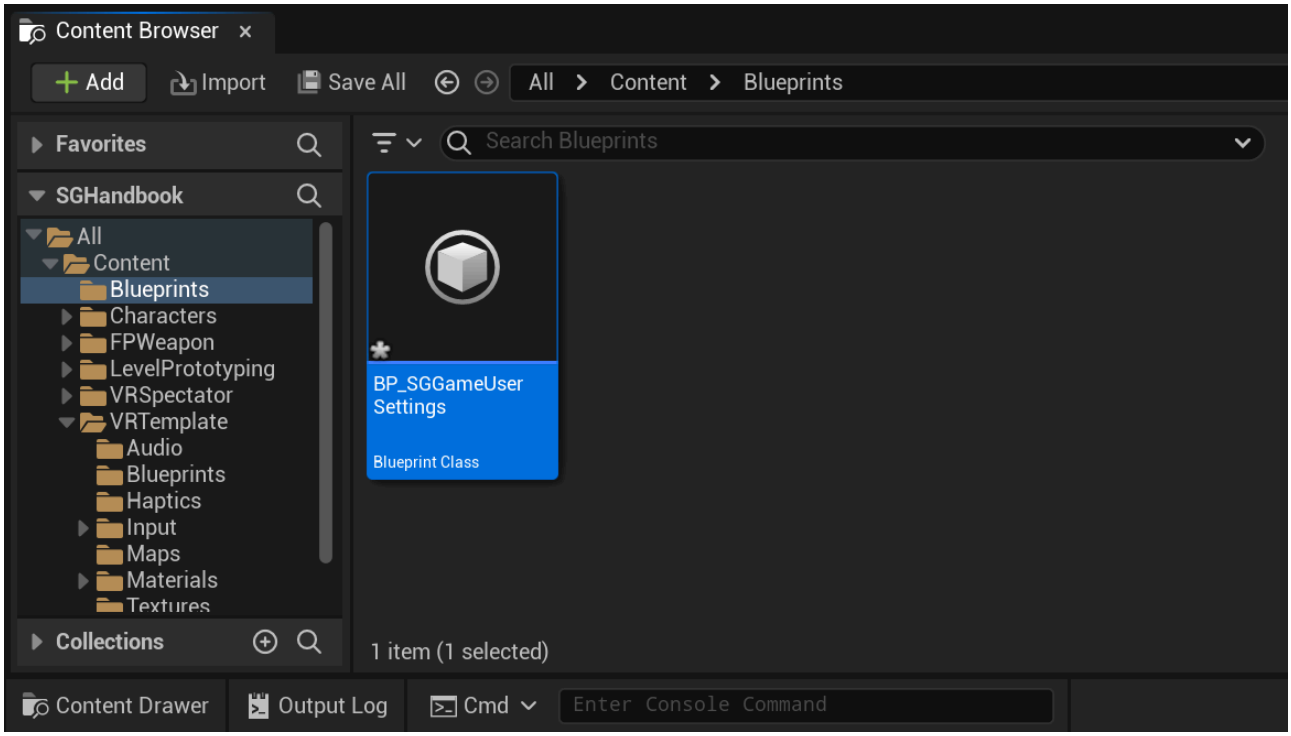
3. In the expanded `ALL CLASSES` section, start typing `SGGameUserSettings` in the Search box. When `SGGameUserSettings` appears, select it and click the `Select` button to create your new Blueprint class based on it.



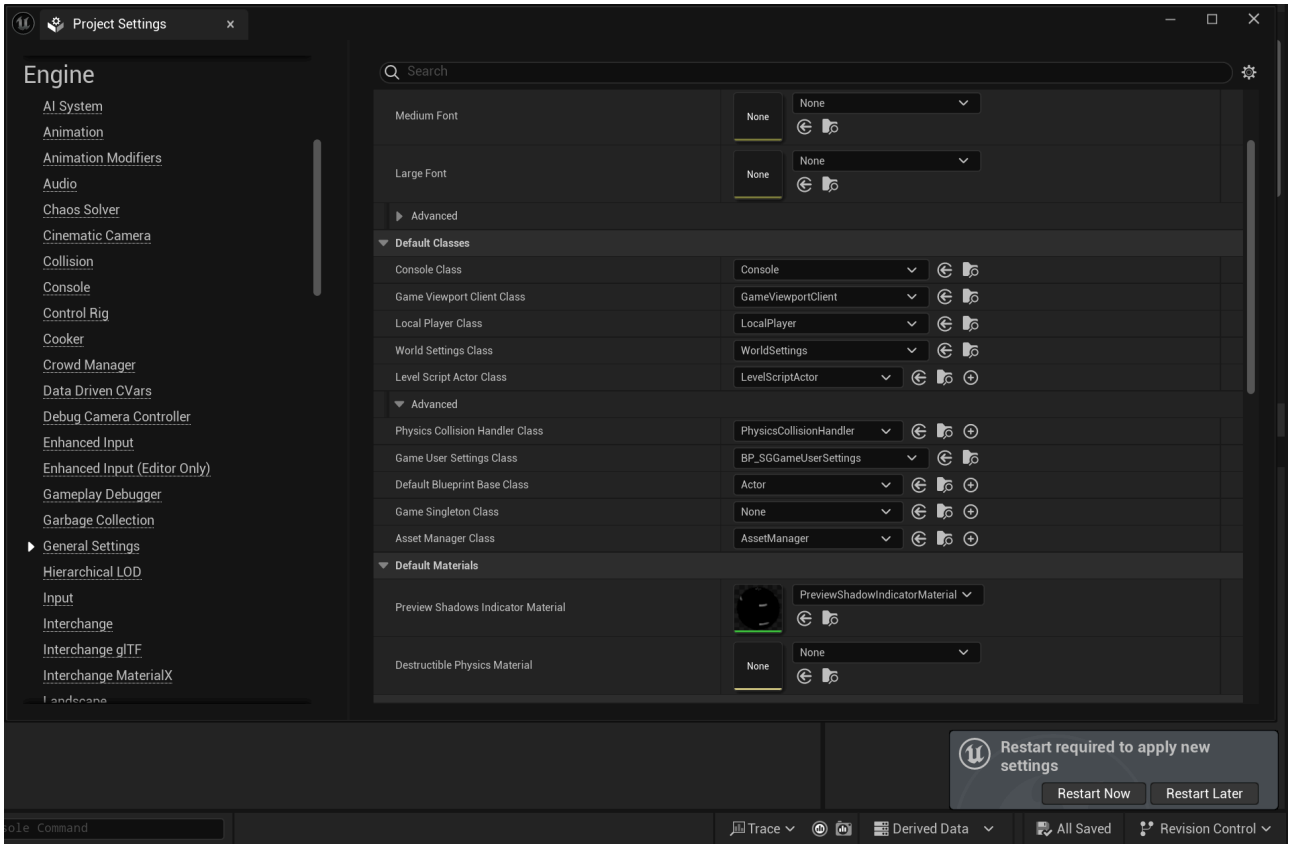
4. After returning to the Content Browser, the Unreal Editor will prompt you to rename `NewBlueprint` to your desired class name. You can rename the class at any time by pressing `F2` or by right-clicking on it and selecting `Rename` from the context menu.



5. Once you have renamed the `NewBlueprint` class to your desired name, click on `Save All` to save the new class to disk.



6. Finally, set your newly created subclass of `SGGameUserSettings` as the default `Game User Settings Class`. You can do this by navigating to `Project Settings > Engine > General Settings > Default Classes > Advanced > Game User Settings Class`. Once you change the default `Game User Settings Class` the Unreal Editor will prompt you with `Restart required to apply new settings`. For the changes to take effect, click on the `Restart Now` button and wait for the editor to reopen.



Setting Up the Virtual Hand Meshes

Setting up Virtual Hand Meshes involves two key steps:

1. Importing the virtual hand meshes into your project.
2. Configuring the virtual hand settings.

In this section we focus on the first part. For detailed information on step two, please visit the Virtual Hand configuration section.

Compatible Virtual Hand Meshes

The SenseGlove Unreal Engine Plugin is compatible with any virtual hand mesh that adheres to the Epic rig and bone structure. Additionally, the virtual hand meshes must be exported with specific settings to meet all requirements. If you're planning to model and rig your own virtual hand meshes, the Epic FBX Skeletal Mesh Pipeline is a useful starting point.

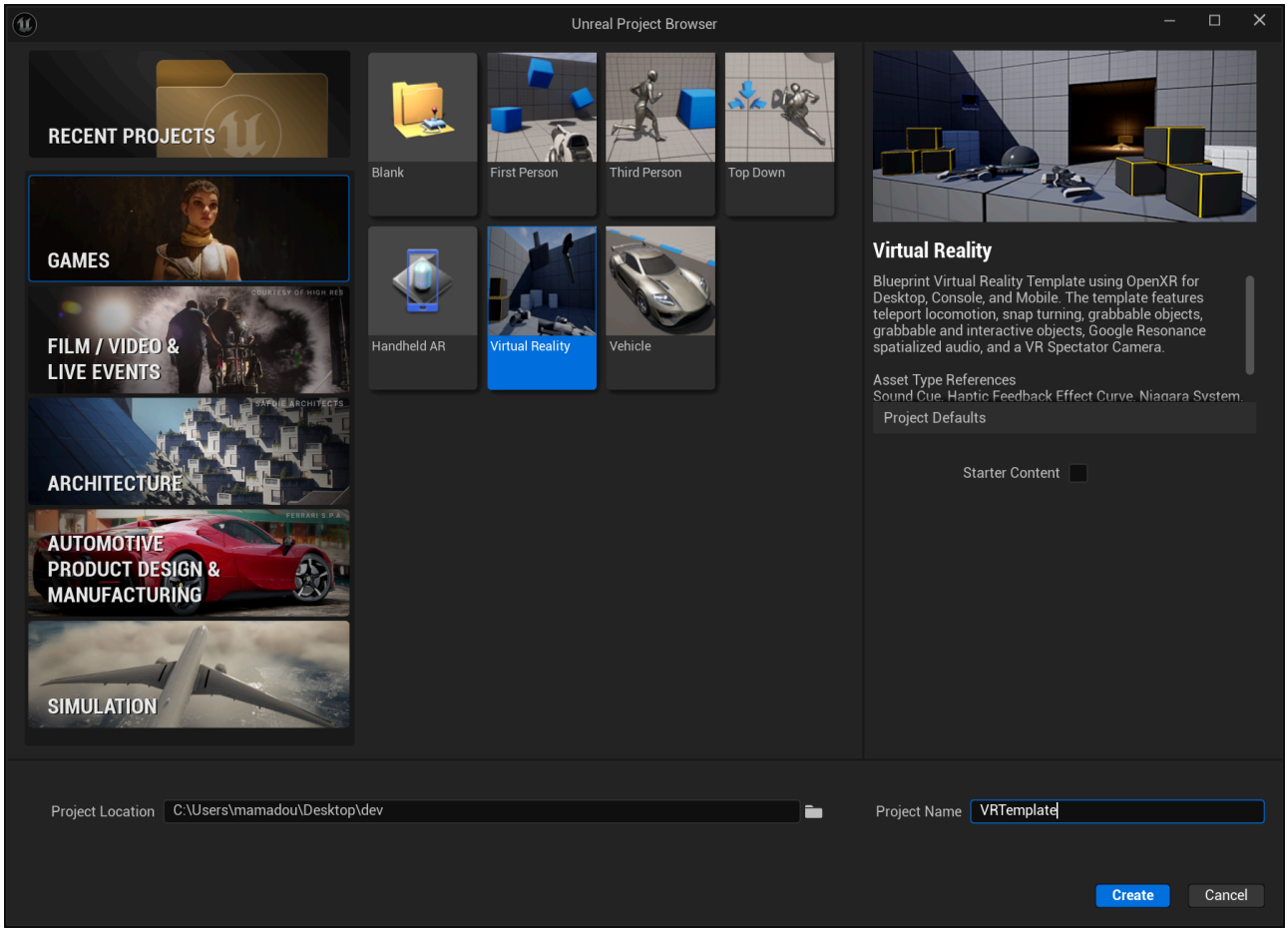
However, if you're looking to get up and running with the SenseGlove Unreal Engine Plugin quickly, the process is much simpler. Unreal Engine has included two sets of compatible virtual hand models with the Unreal Engine VR Template since version `5.1`. This guide will walk you through how to export these virtual hand models from the VR Template and import them into your VR simulation.

> **Caution**
>
> While it is possible to migrate the virtual hand meshes directly from the Content Browser of the VR Template, this approach is not recommended. As part of the setup process, it is necessary to configure the SenseGlove Grab and Touch sockets. Although it's possible to set up these sockets manually, as demonstrated in one of our older tutorials, we no longer recommend doing so. Since version `v2.1.0` of the SenseGlove Unreal Engine Plugin, we've included a tool that automates the socket setup with a single click, eliminating the need for the tedious manual process.

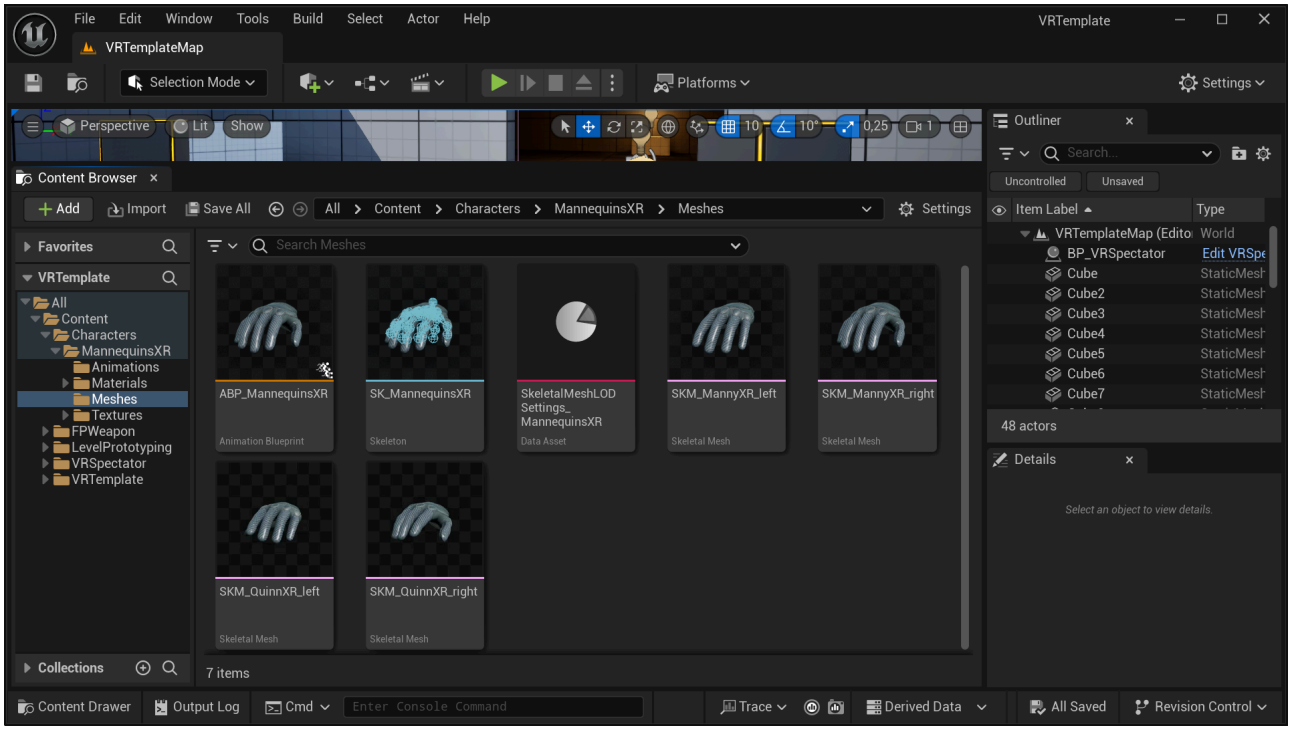
> Unfortunately, the SenseGlove Sockets Editor tool does not support skeletal meshes that share their skeleton. This is the case with the hand models included in the VR Template. Because of this limitation, we will be reimporting the virtual hand meshes with separate skeletons to ensure full compatibility with the SenseGlove Sockets Editor.

Exporting the Virtual Hand Meshes from the VRTemplate

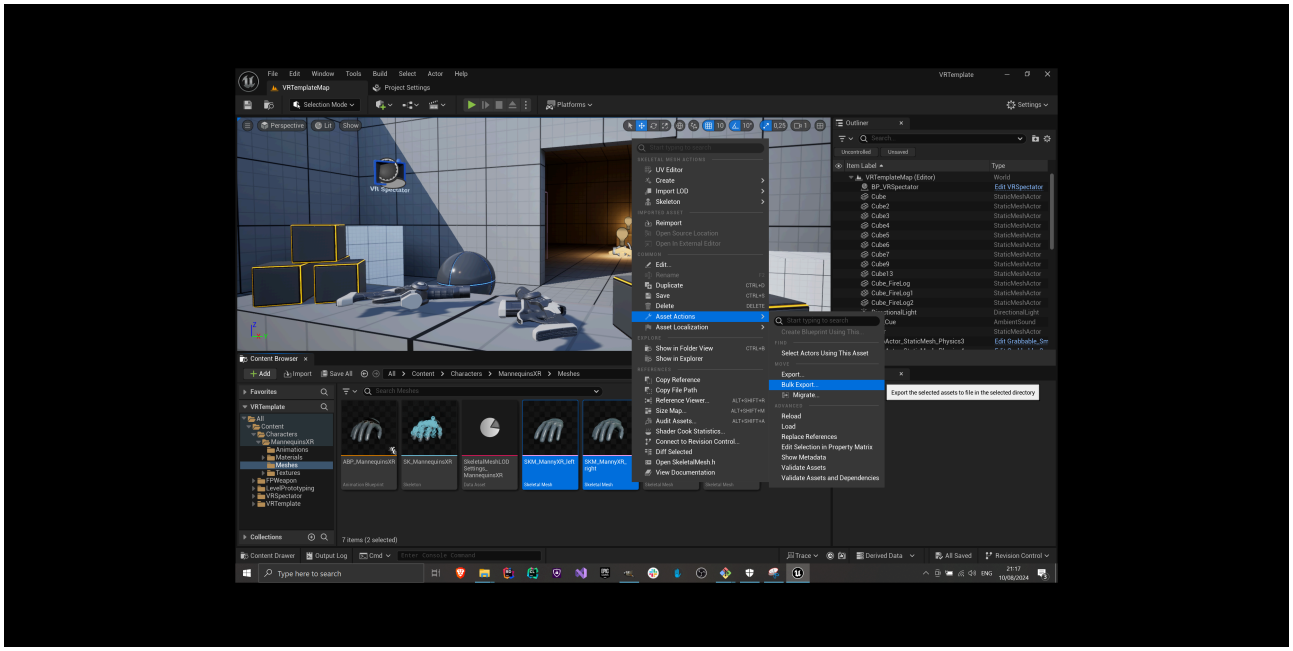
1. Start by creating a new Unreal Engine project using the VR Template. In the Unreal Project Browser, select `GAMES > Virtual Reality`.



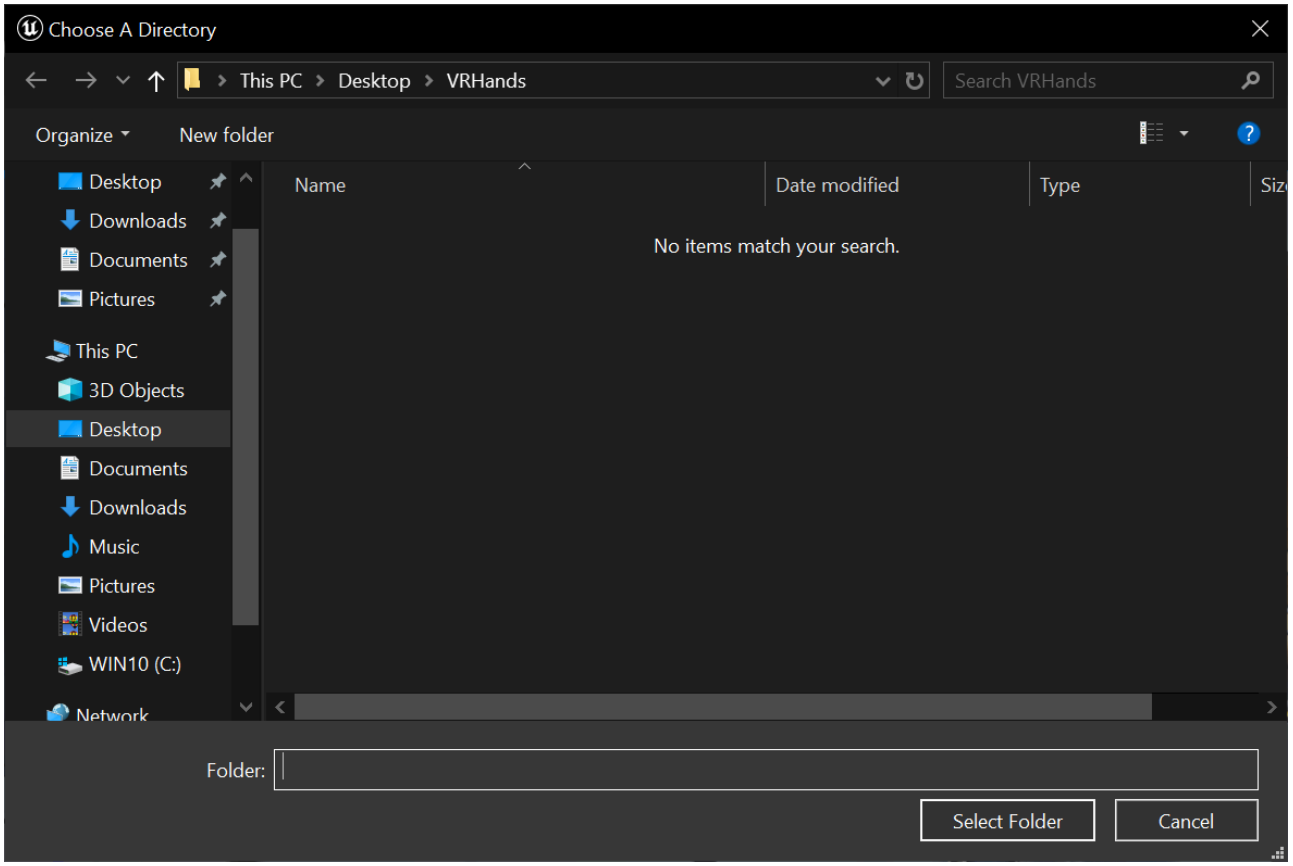
2. Once the Unreal Editor opens with your new project, navigate to the Content Browser. Go to `All > Content > Characters > MannequinsXR > Meshes`. Here, you'll find two sets of virtual hand meshes: `SDKM_MannyXR_left` and `SDKM_MannyXR_right` (male hands), and `SDKM_QuinnXR_left` and `SDKM_QuinnXR_right` (female hands).



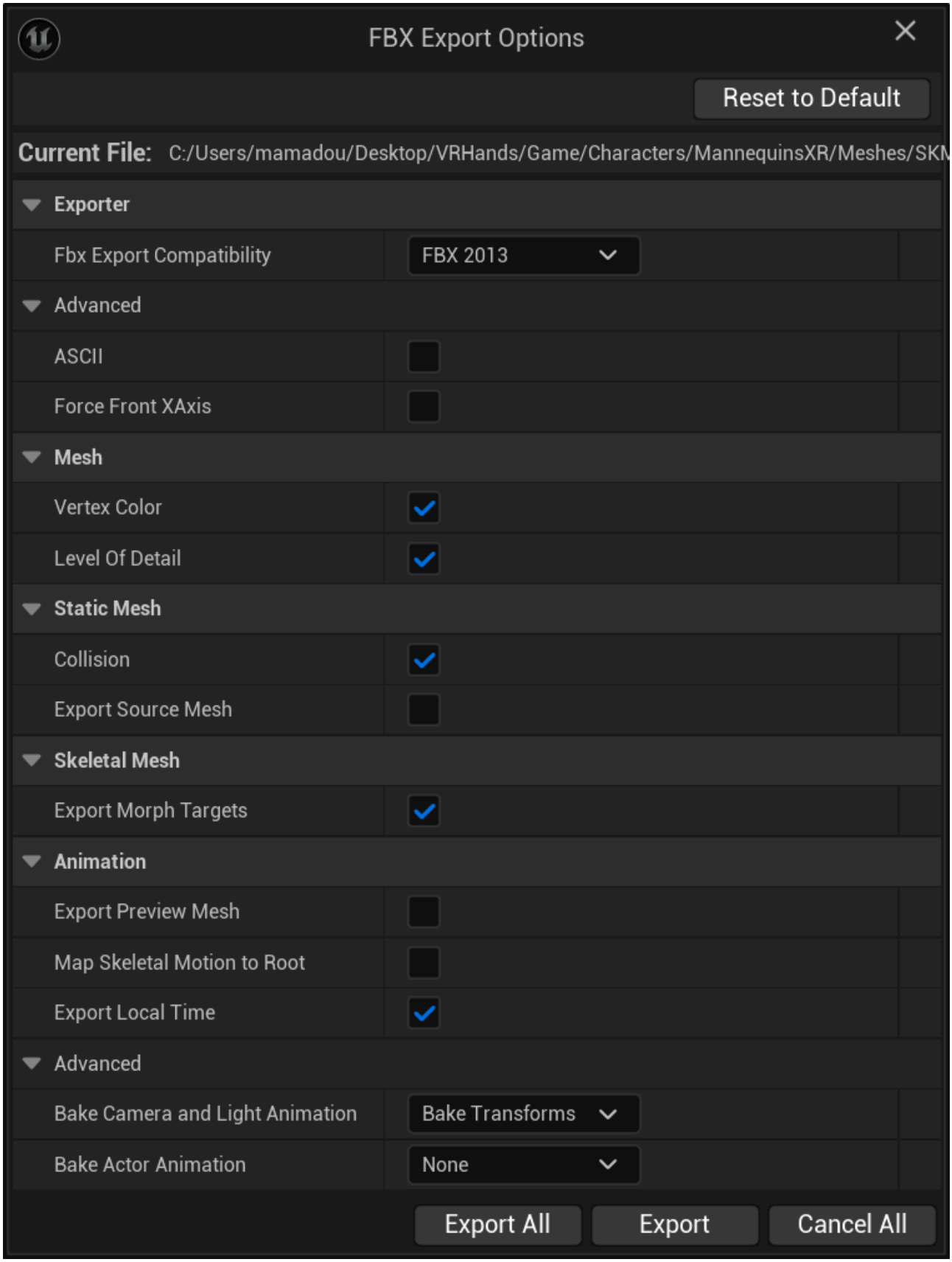
3. Choose the pair of hand meshes you want to export. Right-click on them, then select `Asset Actions` followed by `Bulk Export...` from the context menu.



4. In the file dialog that appears, choose a folder to save the exported hands, and click the `Select Folder` button to export the meshes in FBX format.

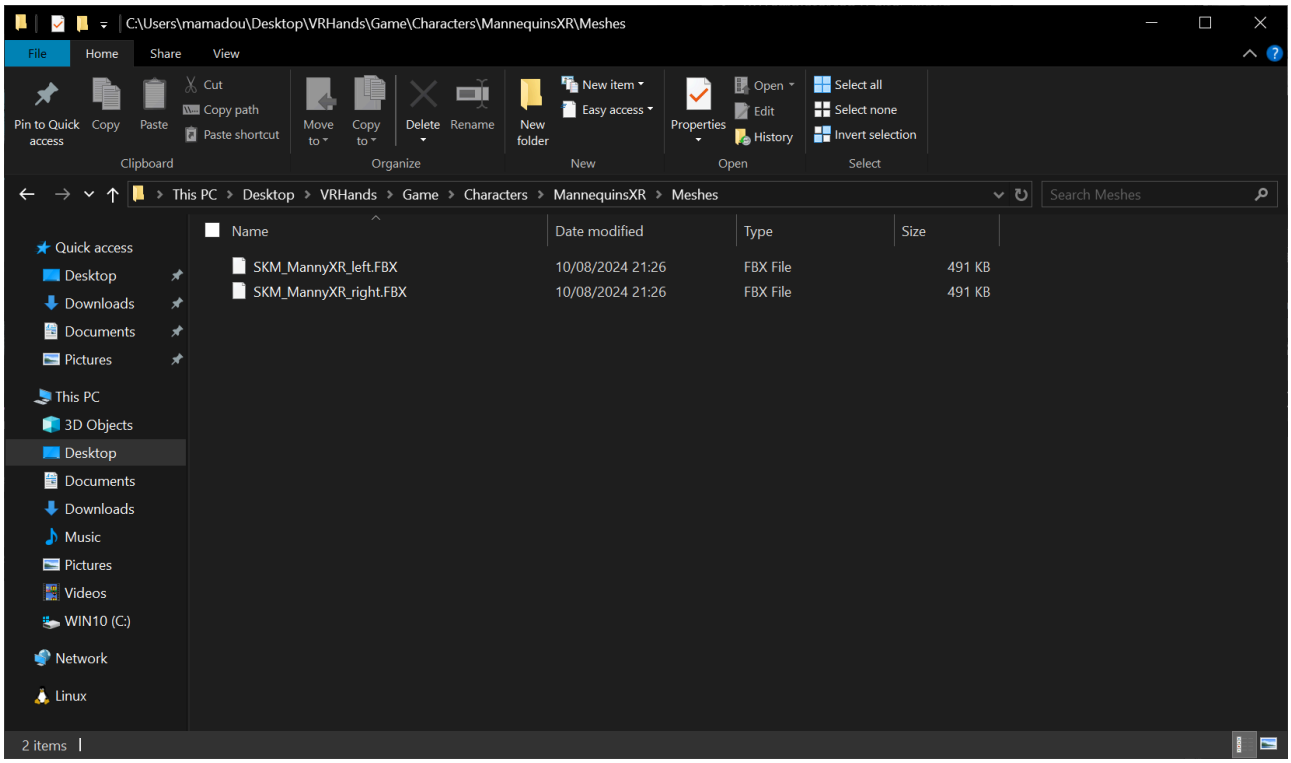


5. The Unreal Editor will then display the `FBX Export Options` dialog. Leave the default settings unchanged and click `Export All` to proceed.


FBX Export Options
Reset to Default
Current File: C:/Users/mamadou/Desktop/VRHands/Game/Characters/MannequinsXR/Meshes/SKM
Exporter
Fbx Export Compatibility
FBX 2013
Advanced
ASCII
Force Front XAxis
Mesh
Vertex Color
Level Of Detail
Static Mesh
Collision
Export Source Mesh
Skeletal Mesh
Export Morph Targets
Animation
Export Preview Mesh
Map Skeletal Motion to Root
Export Local Time
Advanced
Bake Camera and Light Animation
Bake Transforms
Bake Actor Animation
None
Export All
Export
Cancel All

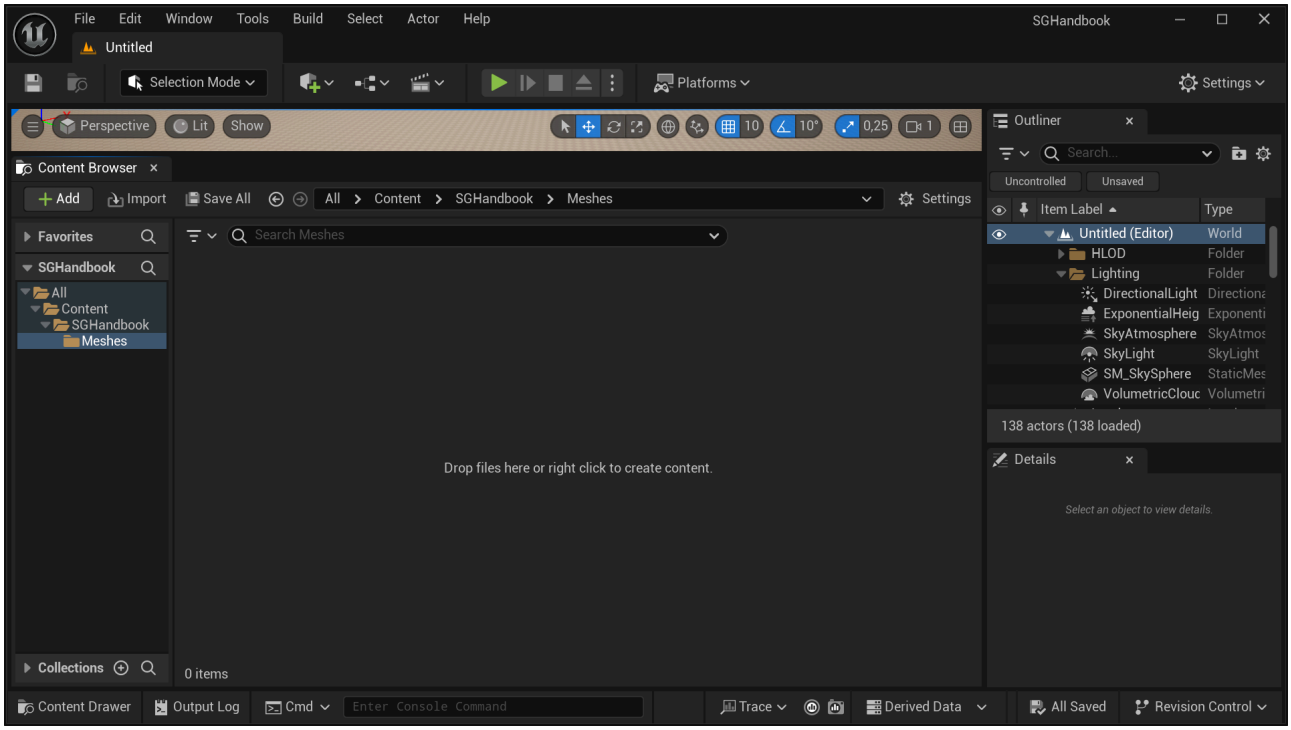
> **Tip**
>
> If you're unsure whether the options are set to their defaults, you can click the `Reset to Default` button in the top-right corner of the dialog to restore the default settings.

6. After exporting, you can find the FBX files for both hands in the directory you selected:
   `/path/you/chose/for/bulk/export/Game/Characters/MannequinsXR/Meshes/`.

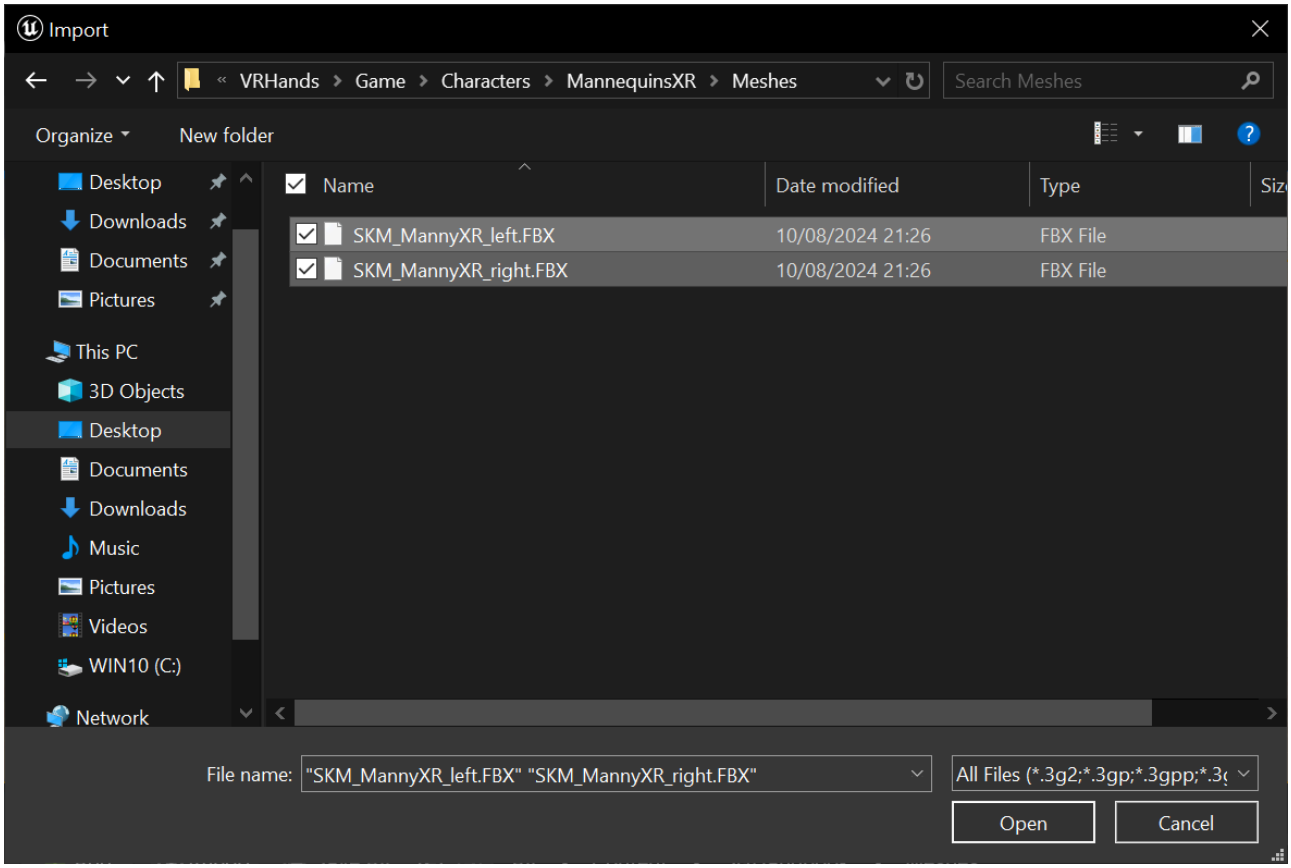


Importing the Virtual Hand Meshes into Your Own Project

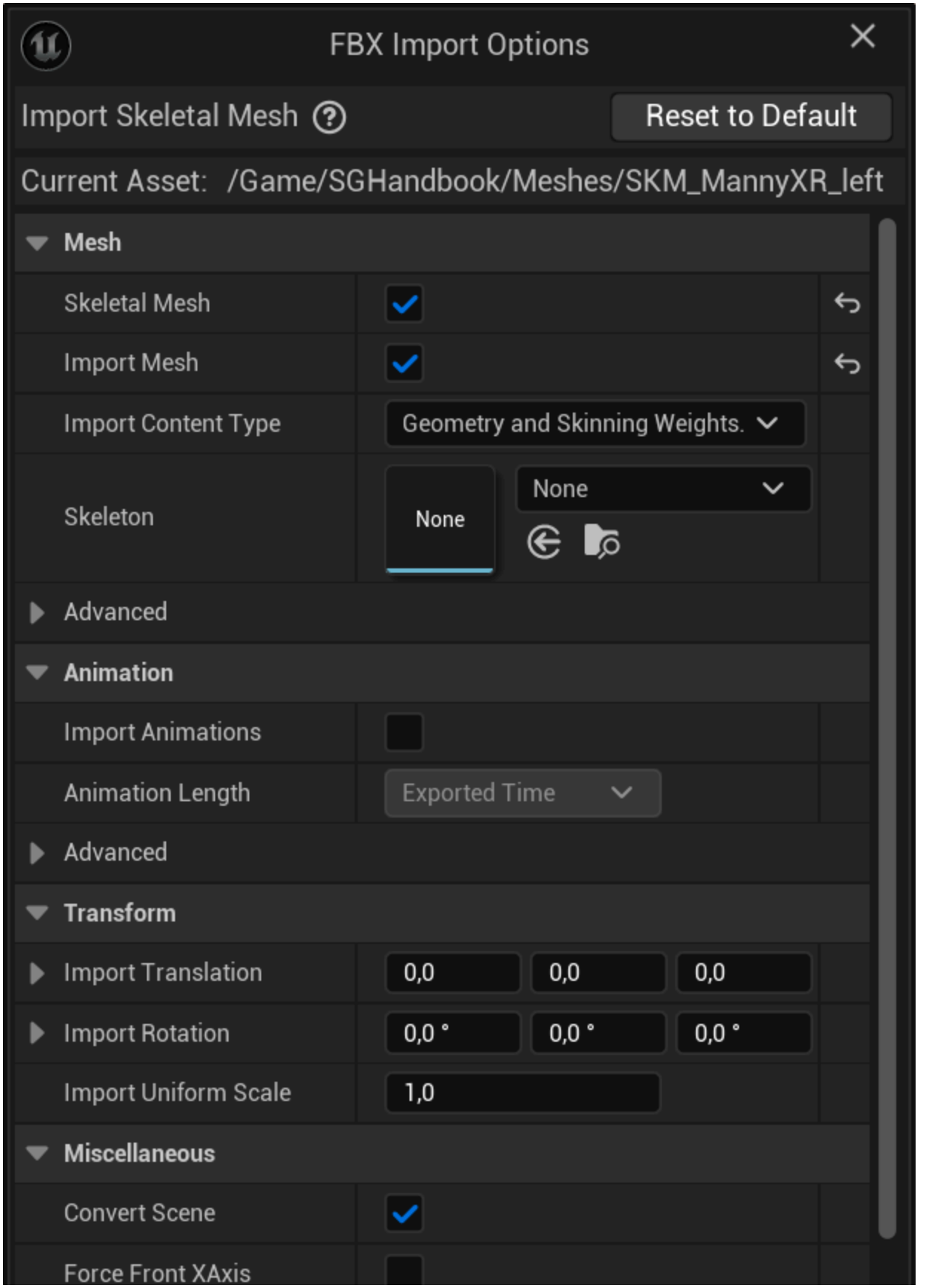
1. Start by creating a new folder inside your project's Content Browser. Navigate to that folder, then press the `Import` button next to the `+ Add` button at the top of the Content Browser.

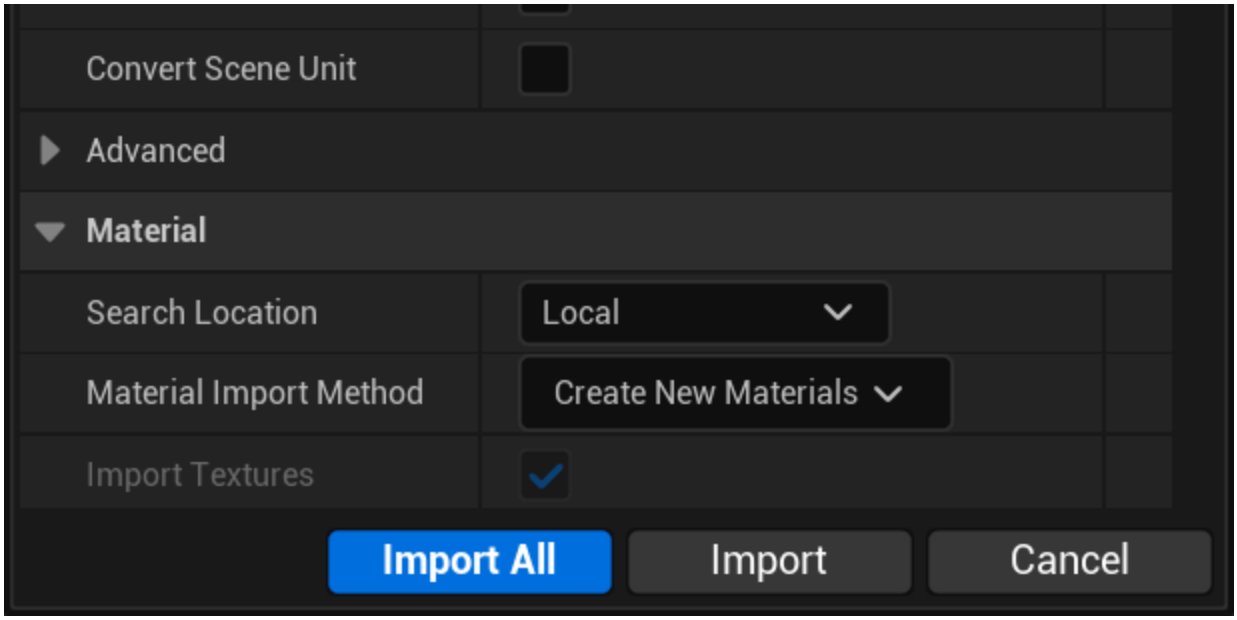


2. In the `Import` dialog that appears, navigate to the folder containing the virtual hand meshes. Select both FBX files and click the `Open` button.



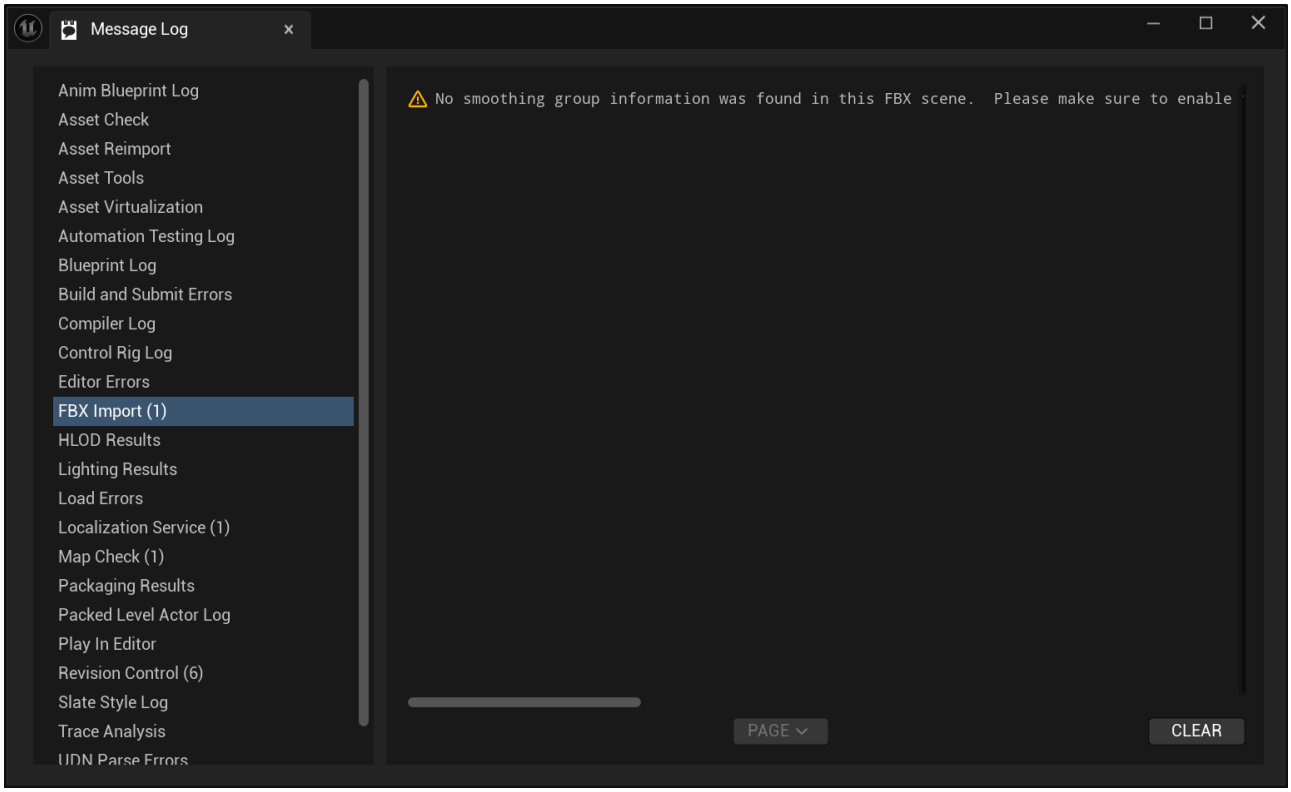
3. The Unreal Editor will display the `FBX Import Options` dialog. Leave the default settings unchanged and click `Import All` to proceed.


FBX Import Options
Import Skeletal Mesh
Reset to Default
Current Asset: /Game/SGHandbook/Meshes/SKM_MannyXR_left
Mesh
Skeletal Mesh
Import Mesh
Import Content Type
Geometry and Skinning Weights.
Skeleton
None
None
Advanced
Animation
Import Animations
Animation Length
Exported Time
Advanced
Transform
Import Translation
0,0
0,0
0,0
Import Rotation
0,0 °
0,0 °
0,0 °
Import Uniform Scale
1,0
Miscellaneous
Convert Scene
Force Front XAxis



> **Tip**
>
> If you're unsure whether the options are set to their defaults, you can click the `Reset to Default` button in the top-right corner of the dialog to restore the default settings.

4. After the import process is done, a dialog will display the import logs. Any errors or warnings encountered during the import process will be shown here.

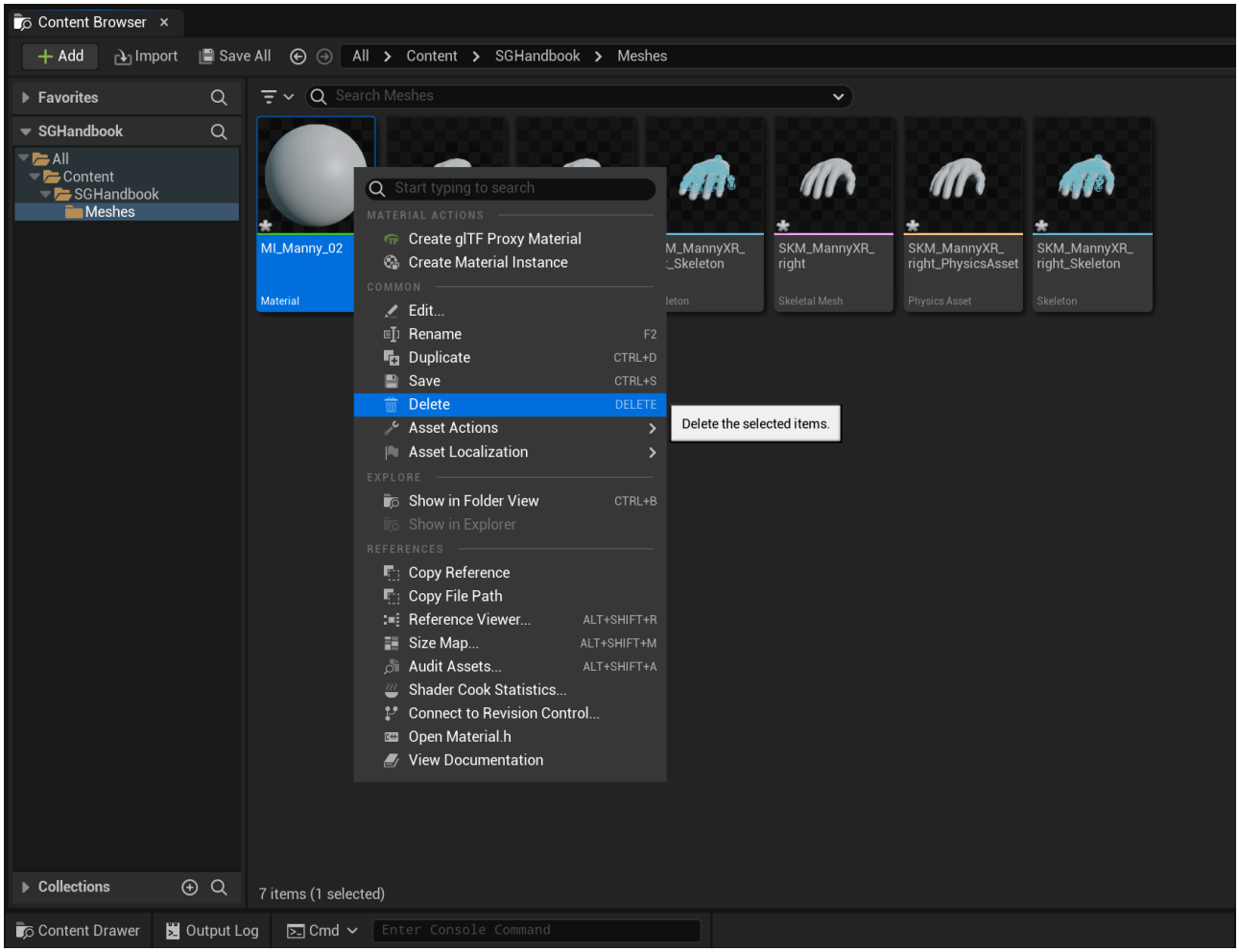


> **Note**
>
> The following warning can be safely ignored:
>
> `FBXImport: Warning: No smoothing group information was found in this FBX scene. Please make sure to enable the 'Export Smoothing Groups' option in the FBX Exporter plug-in before exporting the file. Even for tools that don't support smoothing groups, the FBX Exporter will generate appropriate smoothing data at export-time so that correct vertex normals can be inferred while importing.`

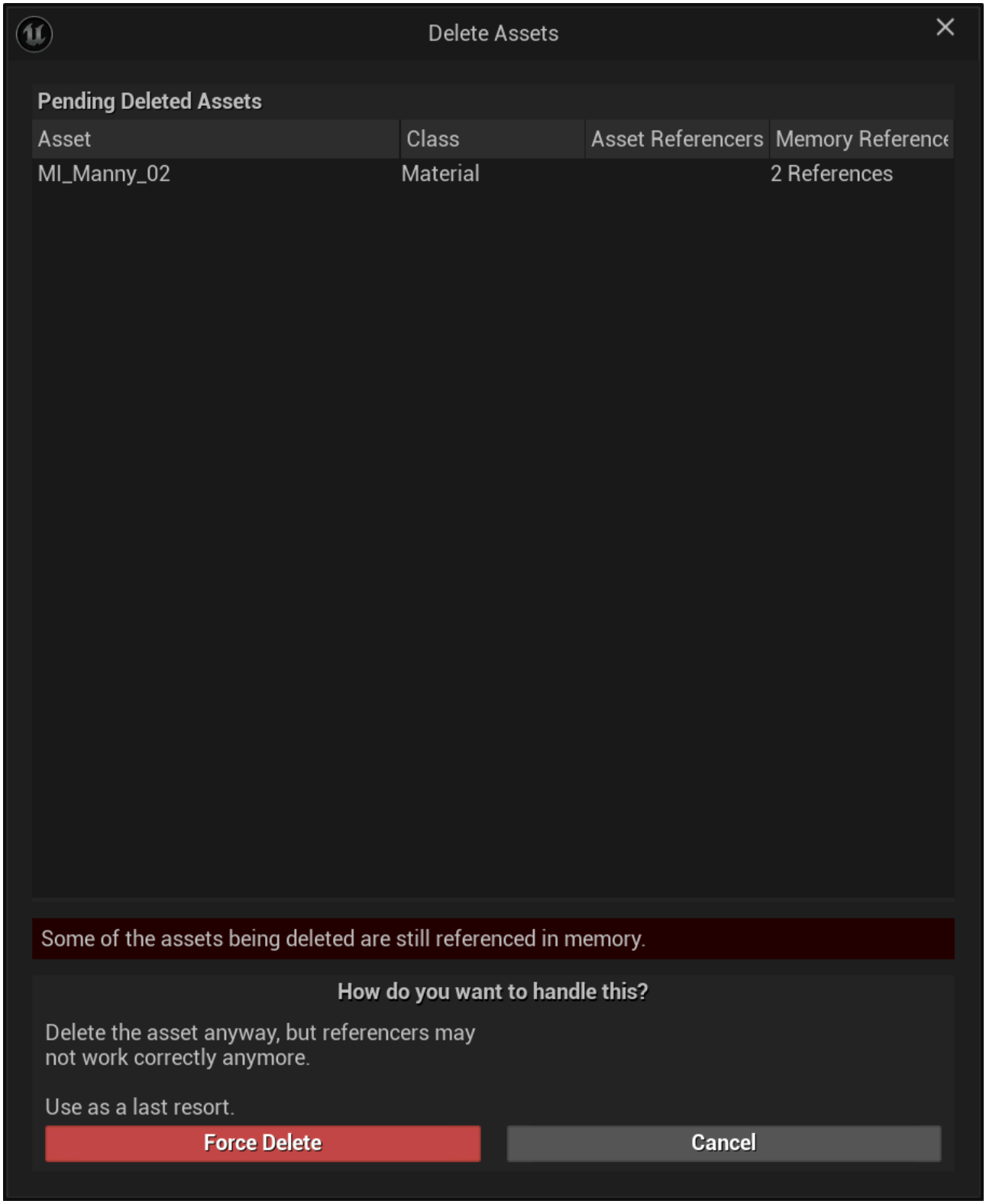
5. The imported virtual hand meshes should now appear in the folder you selected in the Content Browser. Unreal Engine will create a Skeletal Mesh, a Skeleton, and a Physics Asset for each imported mesh, along with a default Material asset shared between both virtual hand meshes.



6. You can choose to keep or modify the default material. However, since the SenseGlove Unreal Engine Plugin provides a default material, we choose to delete the default material created by Unreal Engine during the import process. We'll assign the SenseGlove default material to the imported virtual hand meshes in the next steps. Right-click on the default material and select `Delete`.



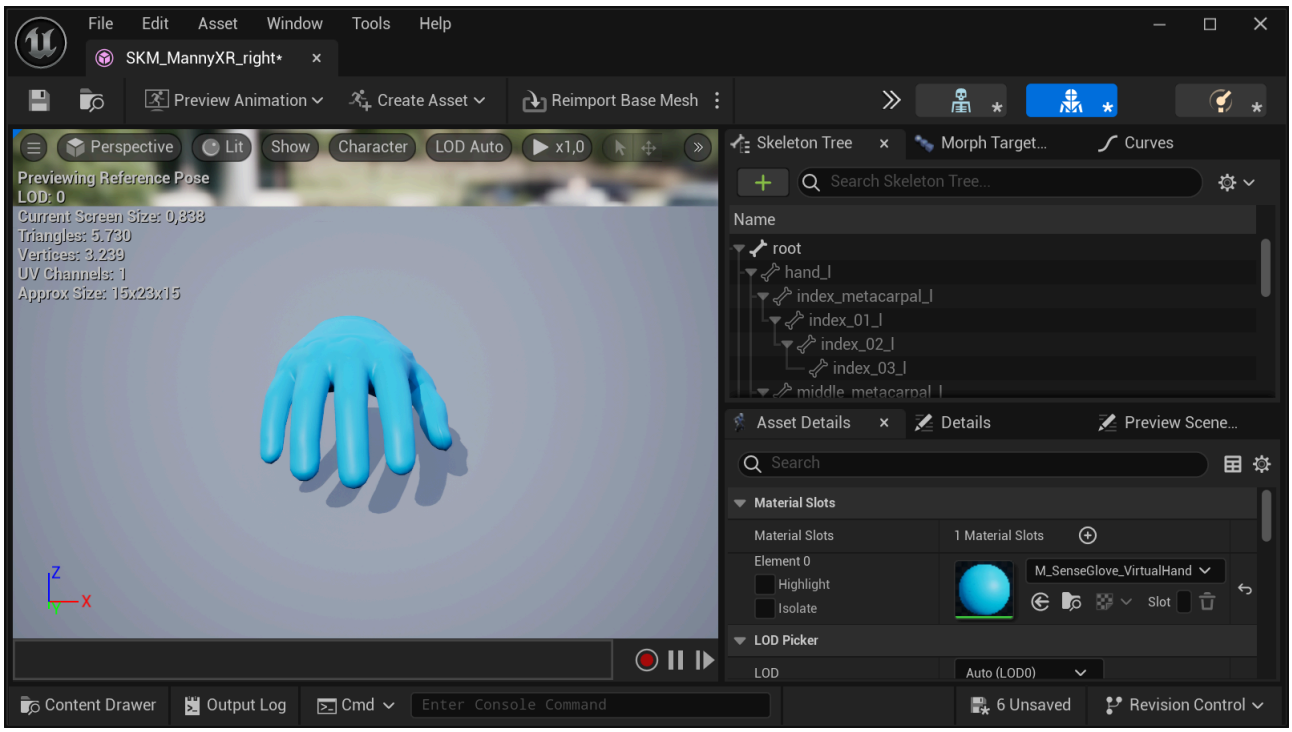
7. In the `Delete Assets` dialog, click `Force Delete` to confirm the deletion of the default material.



8. Open the Skeletal Mesh asset for the left hand and assign the `M_SenseGlove_VirtualHand` material from the `Asset Details` panel.



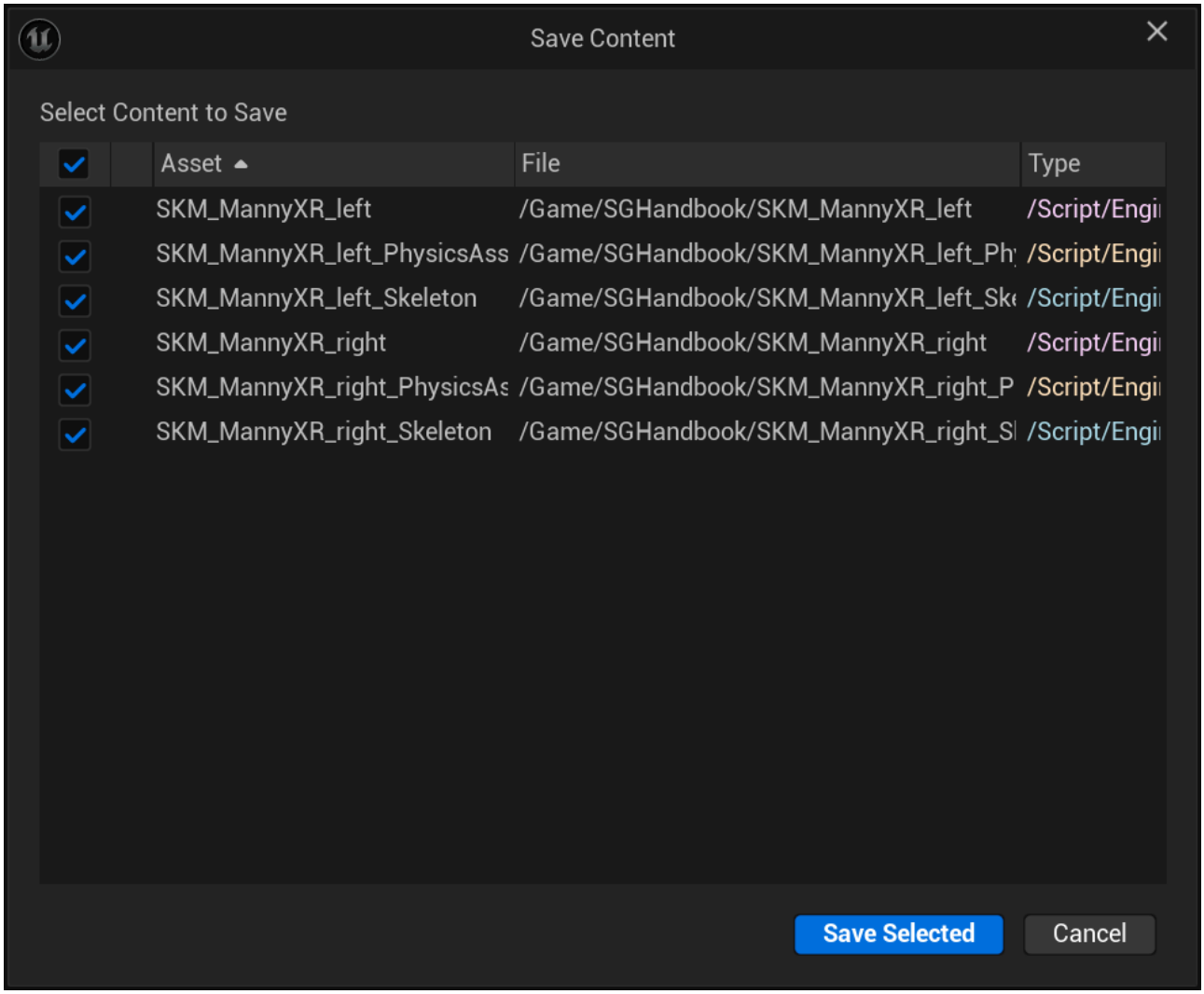
9. Repeat the process for the Skeletal Mesh asset of the right hand, and assign the `M_SenseGlove_VirtualHand` material in the `Asset Details` panel.



10. Return to the Content Browser by closing all asset windows and click the `Save All` button to save all imported virtual hand mesh assets to disk.

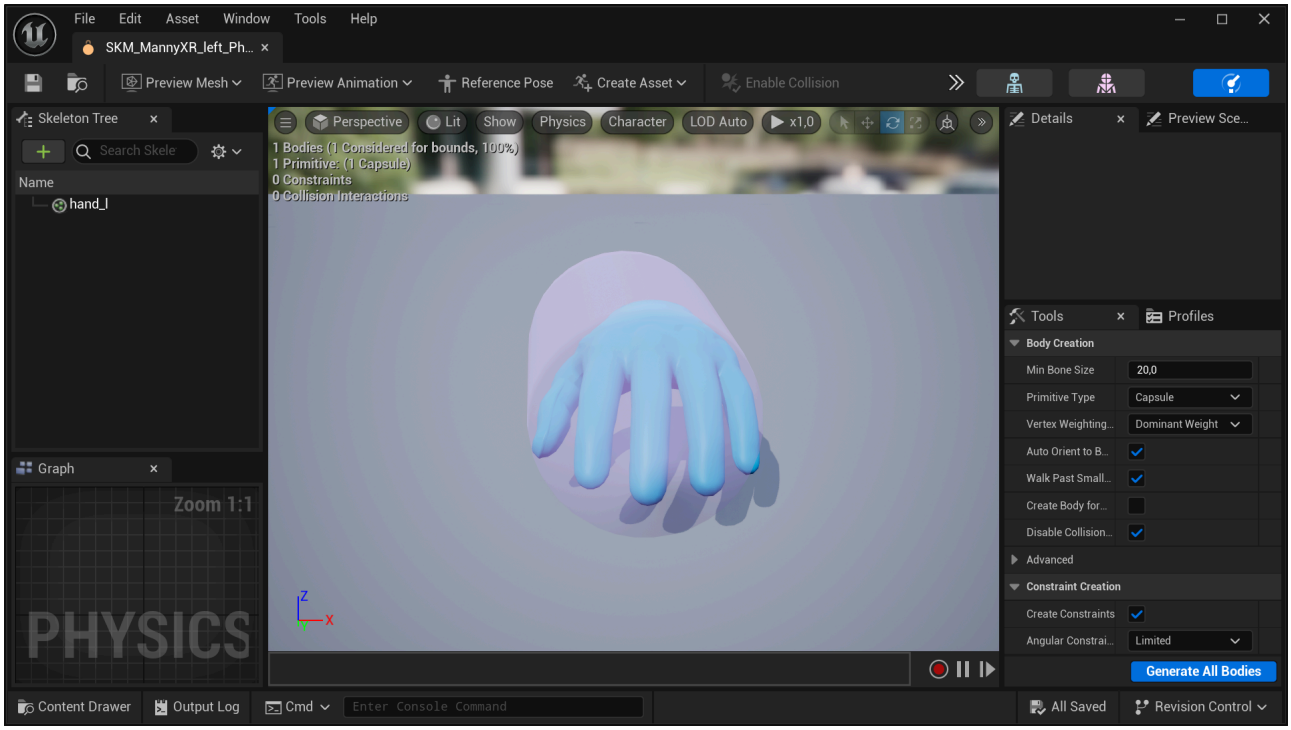


11. In the `Save Content` dialog, choose `Save Selected` to confirm the saving all action.

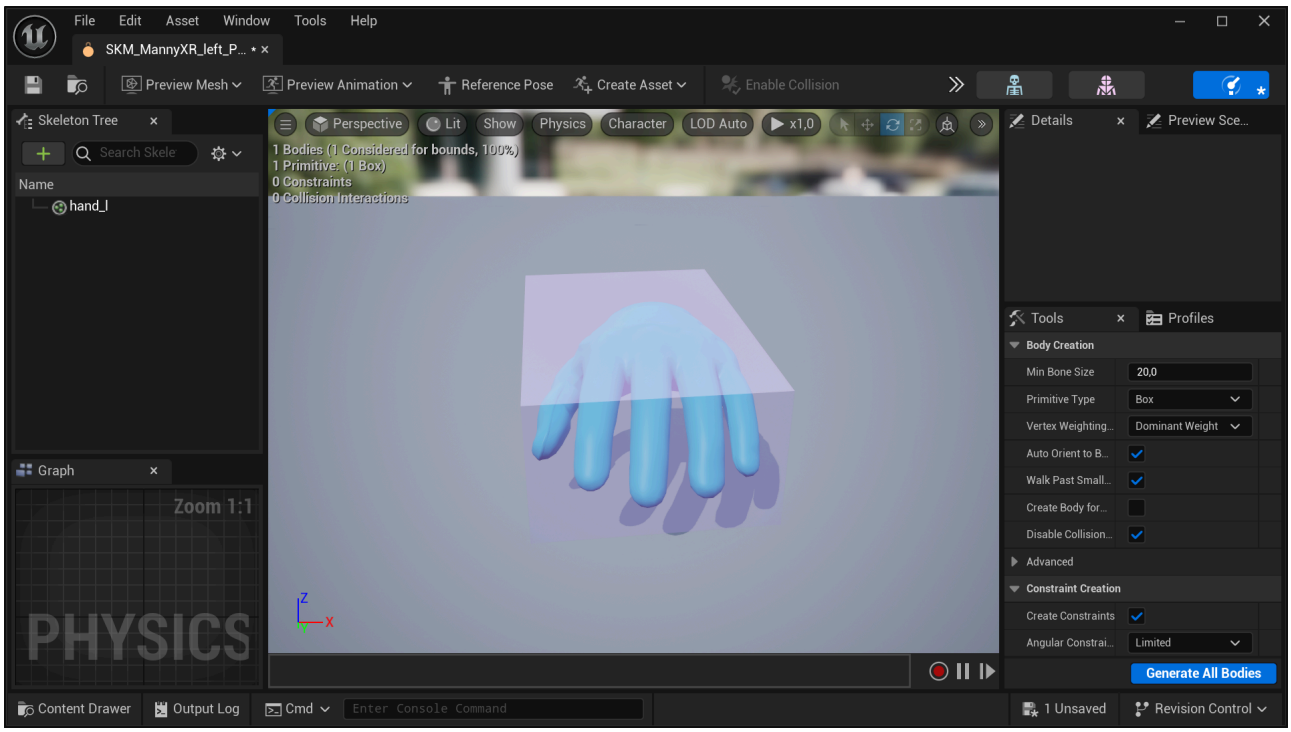


Setting up the Rigid Bodies

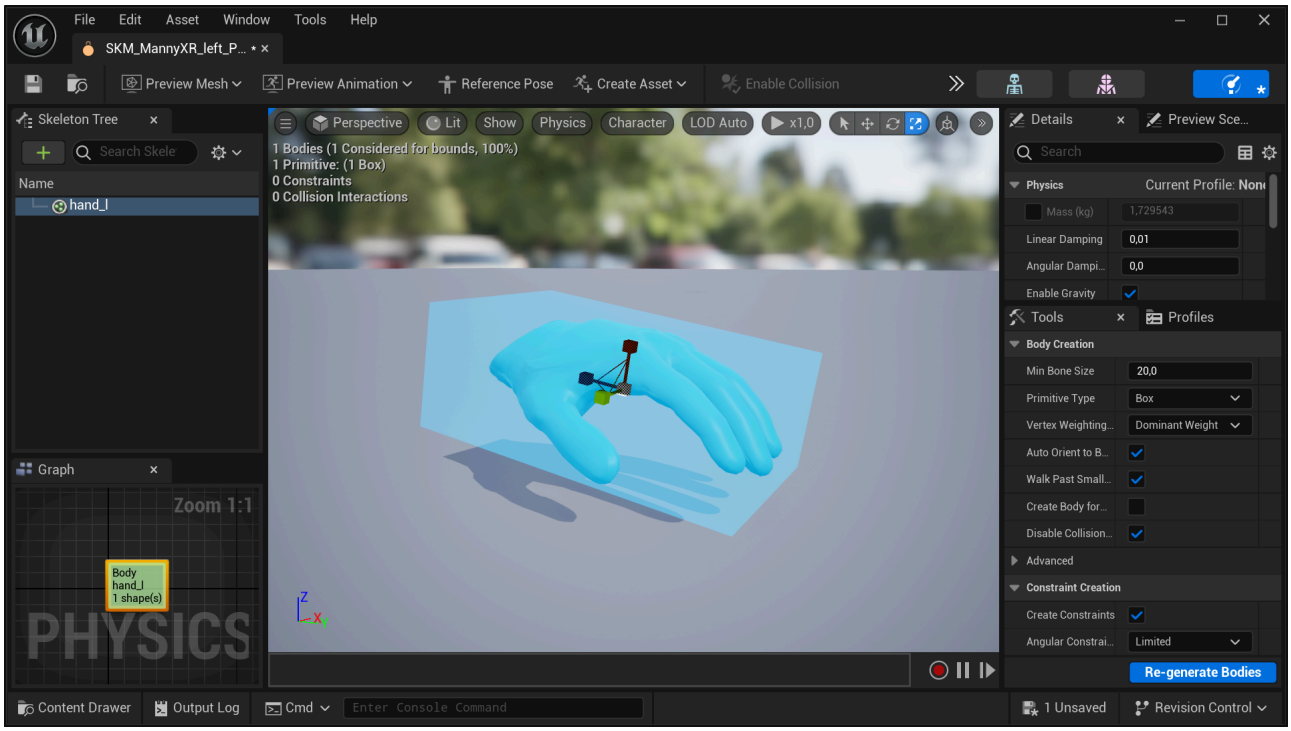
1. Open the Physics Asset for the left virtual hand mesh by double-clicking it in the Content Browser. This will open the PhAT (Physics Asset Tool) editor, where the virtual hand mesh for the left hand will appear with a default physics body, usually shaped as a capsule.



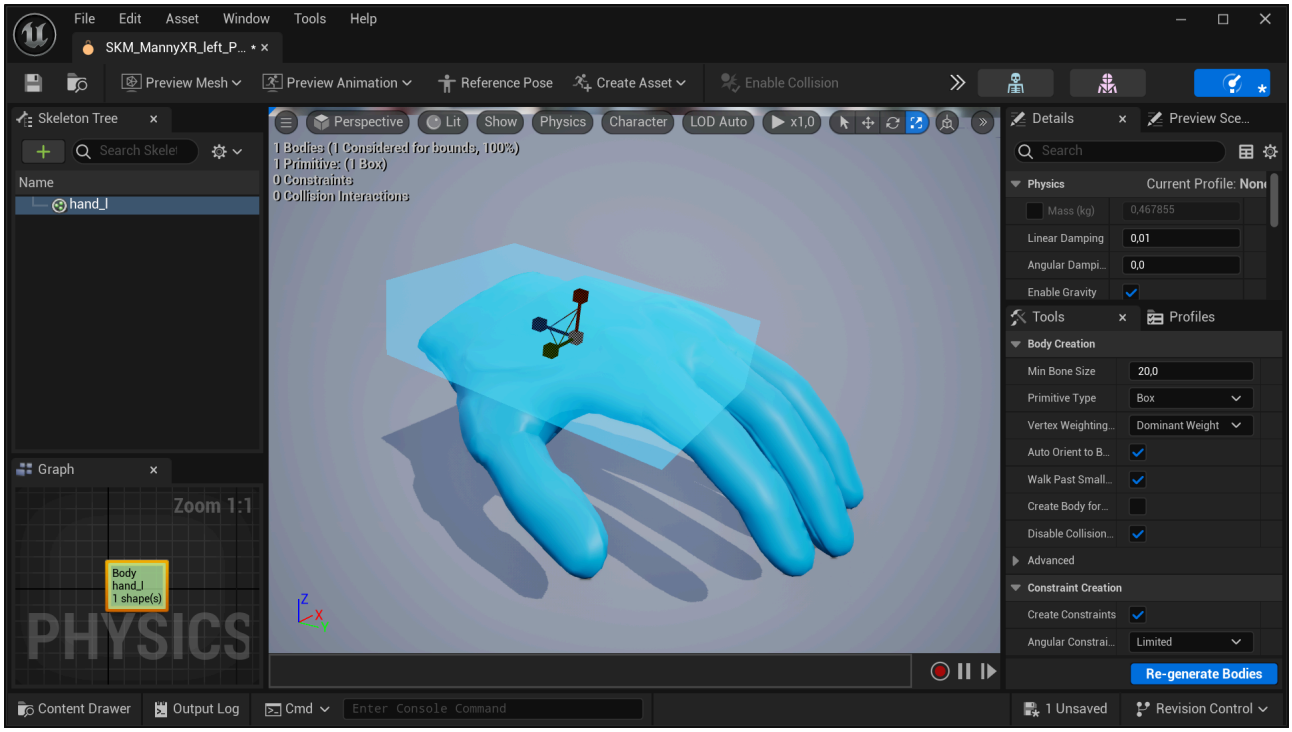
2. In the `Tools` panel, under the `Body Creation` section, locate the `Primitive Type` dropdown and select `Box` instead of the default `Capsule` shape. Then, click the `Generate All Bodies` button at the bottom of the `Tools` panel to create a new physics body.



3. After generating the new body, some adjustments are required for optimal interactions inside your VR simulations. Press the `r` key on your keyboard to enter scaling mode and use the arrows to resize the physics body. To reposition the body, press the `w` key to switch to translation mode. For adjusting the rotation, press the `e` key. Toggle between these modes as needed to fine-tune the physics body to your requirements.

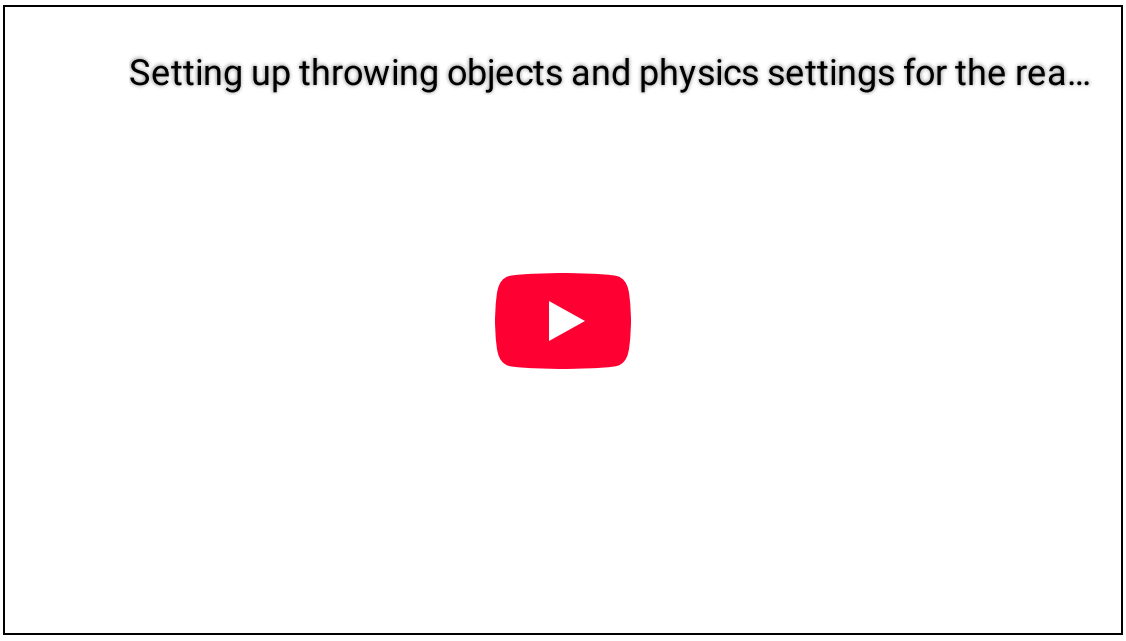


4. You can always revisit and adjust the rigid body later after testing its impact in your VR simulations. For now, save the asset and close the PhAT editor.



5. Repeat the same procedure for the right virtual hand mesh.

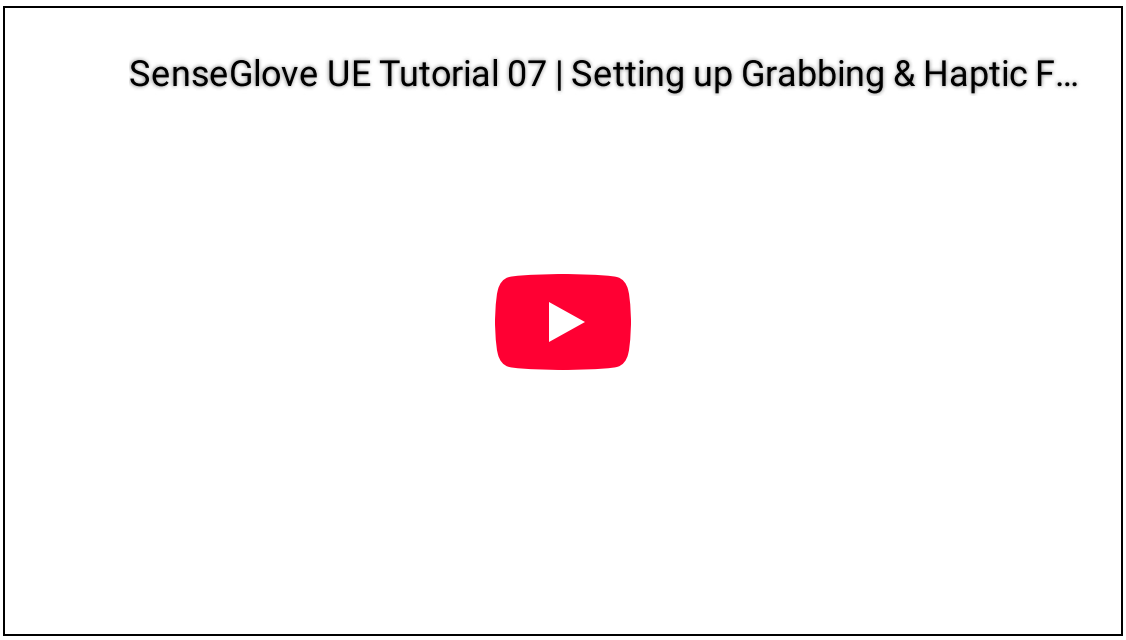
> **Note**
>
> An older yet still relevant video tutorial demonstrating a similar procedure is also available.



Setting up the SenseGlove Grab and Touch Sockets

To ensure the Grab/Release and Touch systems function correctly, multiple sockets must be set up on each virtual hand mesh with precise locations and rotations. Before version `v2.1.0` of the SenseGlove Unreal Engine Plugin, this was a manual and time-consuming process. However, with the `v2.1.0` release, the plugin now includes the SenseGlove Sockets Editor, a built-in tool specifically designed for this task.

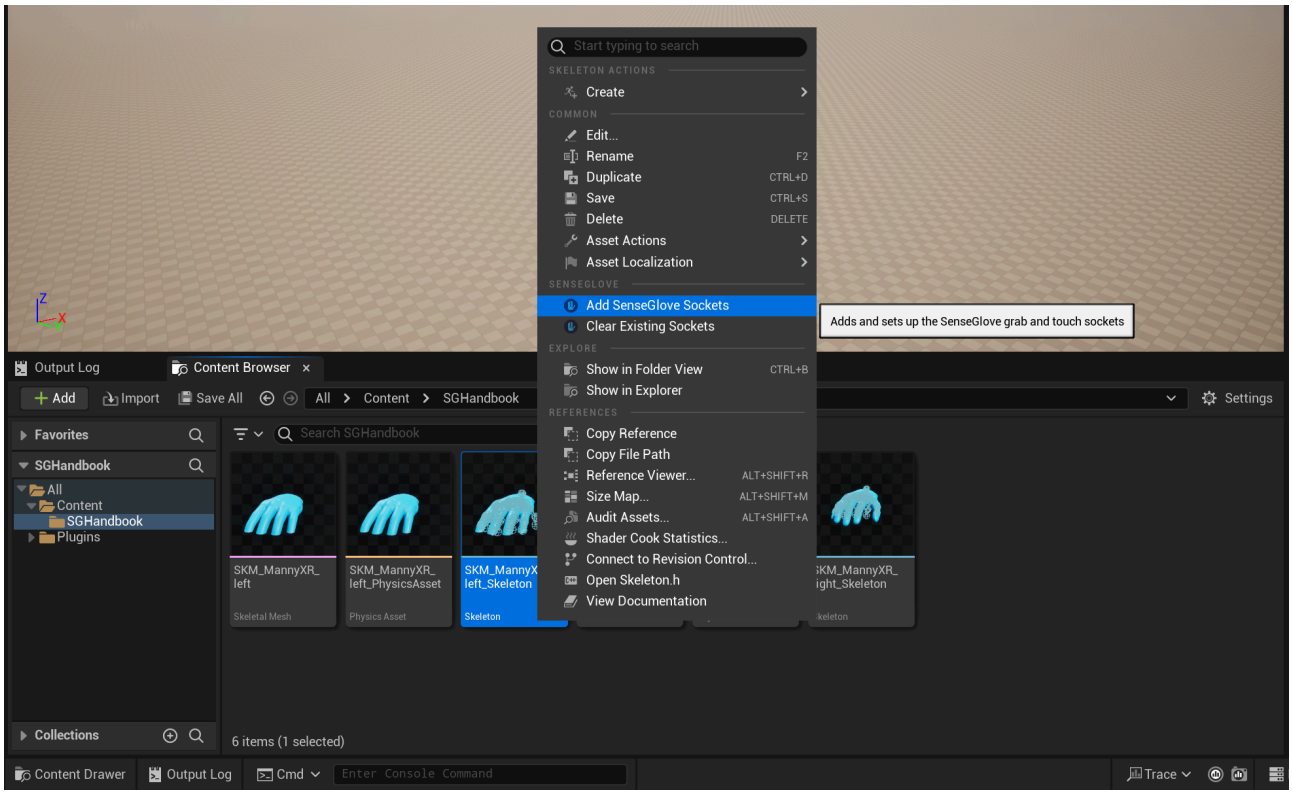
> **Note**
>
> If for any reason you still prefer to manually set up the sockets, a detailed video tutorial is available.



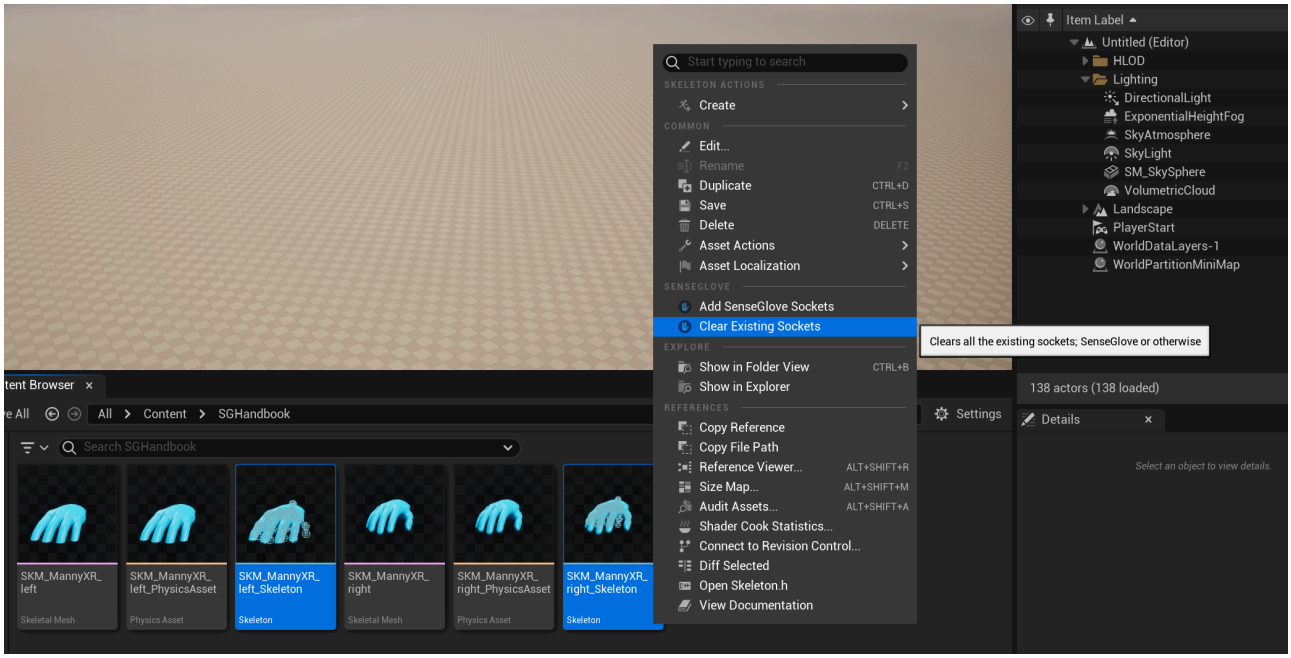
Accessing the SenseGlove Sockets Editor

The SenseGlove Sockets Editor can be utilized in three ways:

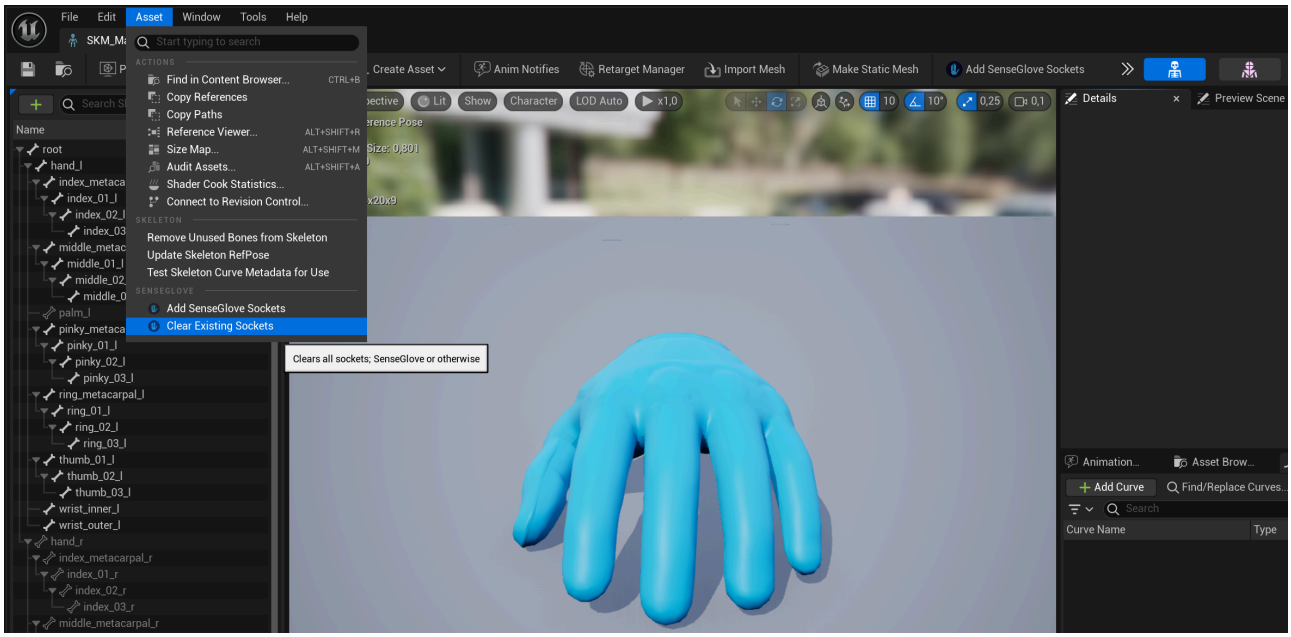
1. By right-clicking on any Skeleton or Skeletal Mesh asset inside the Unreal Content Browser.



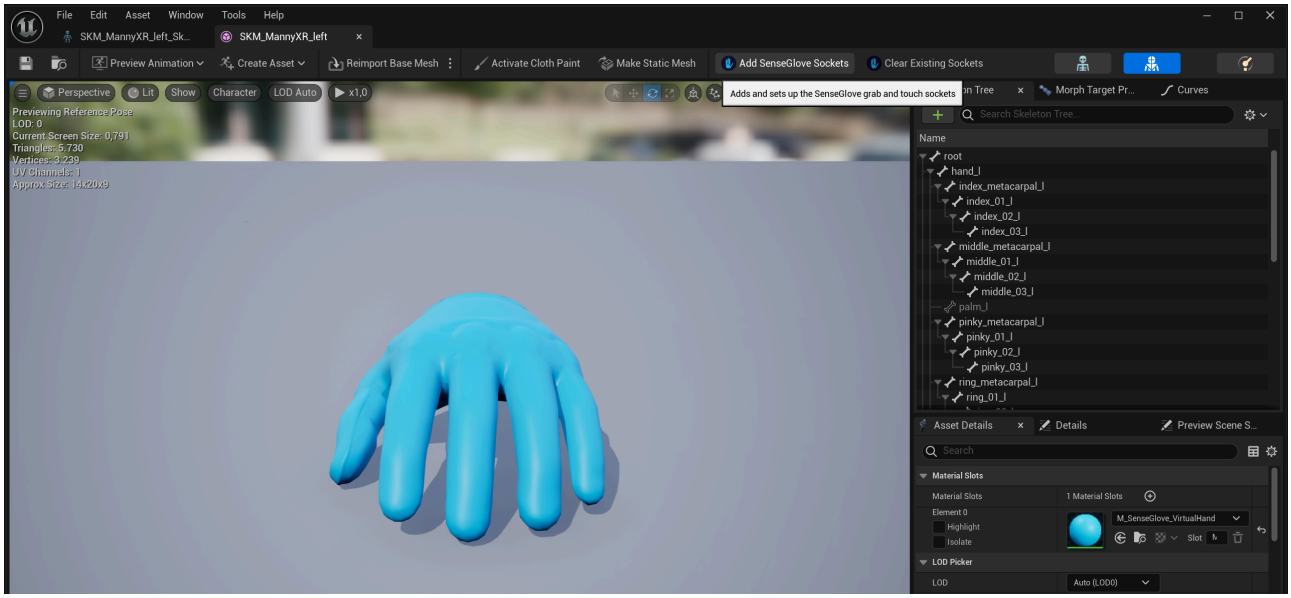
> ### Tip
>
> You can also perform Sockets Editor actions in bulk by selecting multiple assets of the same type and right-clicking on one of them. Note that if the selected assets are not all of the same type, Sockets Editor actions will not appear (e.g. selecting assets of type Skeletons and Skeletal Meshes together).



2. From the `Asset` menu in the Skeleton Editor or Skeletal Mesh Editor for any open Skeleton or Skeletal Mesh asset.



3. From the Skeleton Editor or Skeletal Mesh Editor toolbar for any open Skeleton or Skeletal Mesh asset.



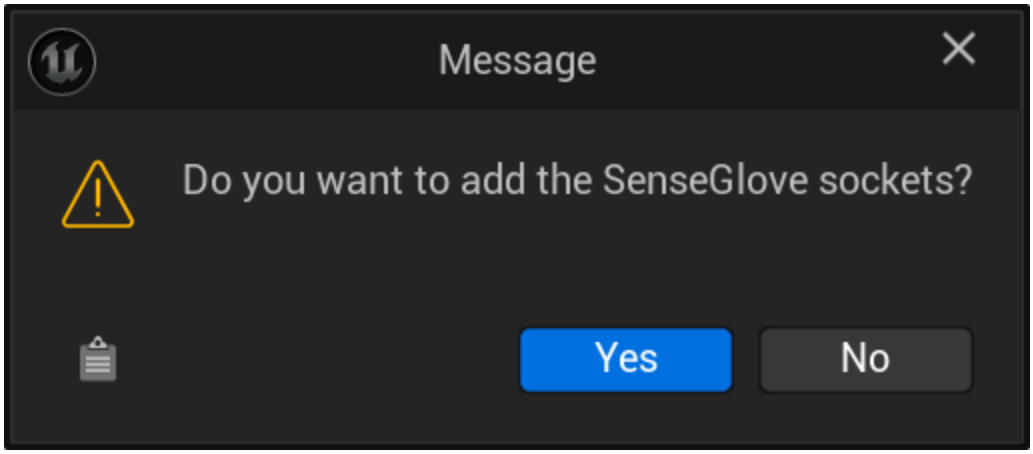
The SenseGlove Sockets Editor currently offers two actions:

1. `Add SenseGlove Sockets` : which adds and sets up the SenseGlove grab and touch sockets to any virtual hand mesh that adheres to the Epic rig and bone structure.
2. `Clear Existing Sockets` : which destructively clears all existing sockets; SenseGlove or otherwise, from any mesh.

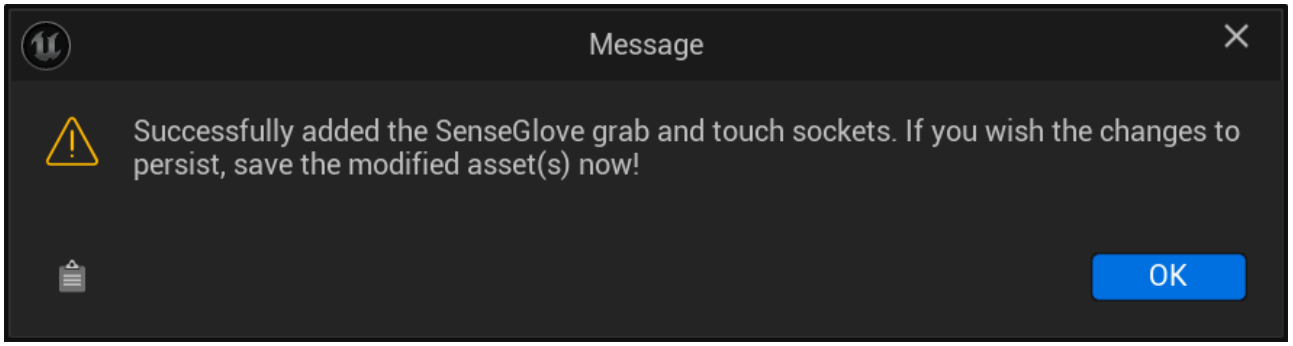
> **Important**
>
> Simply performing any of these actions won't permanently modify your assets. In fact, if you close the Unreal Editor without saving your assets first, all changes performed by the SenseGlove Sockets Editor will be lost forever. This is by design and the plugin will leave this final choice to the user. So, in order to apply the changes permanently, you must save the assets manually.

Adding the SenseGlove Sockets

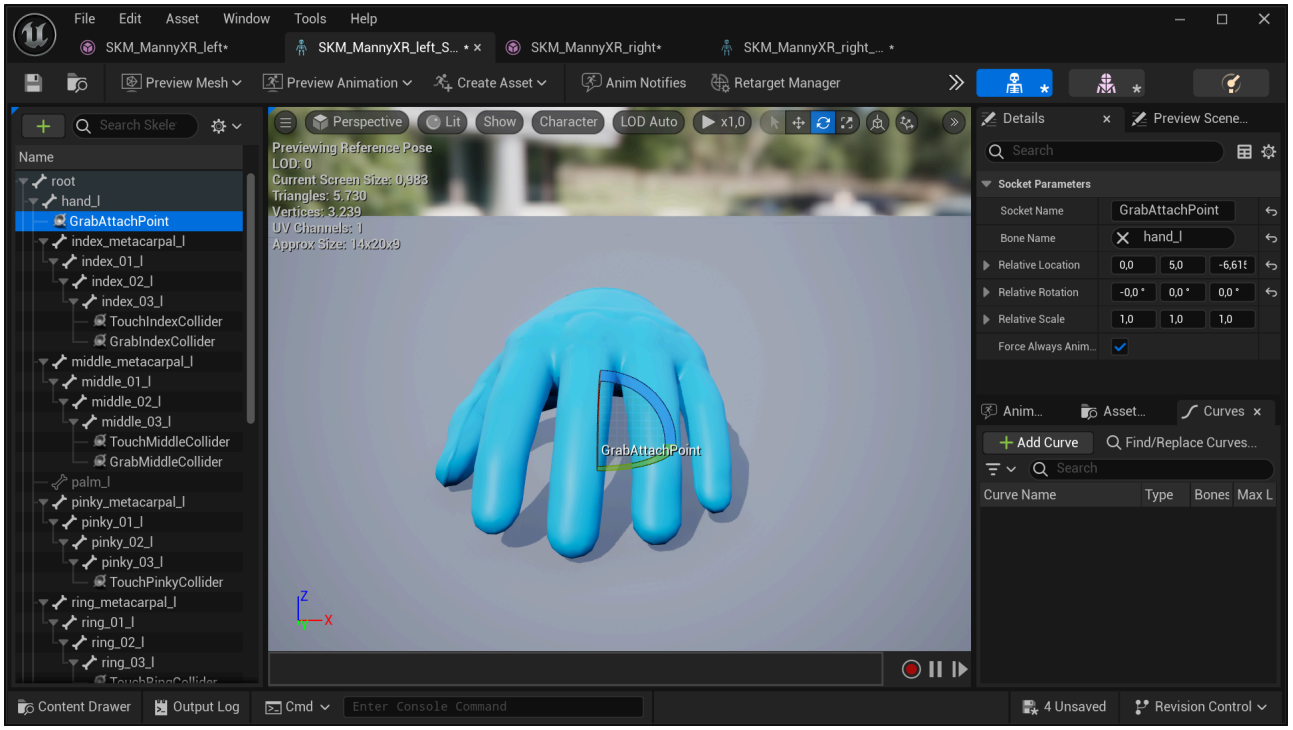
When you invoke the `Add SenseGlove Sockets` action, the Sockets Editor will prompt you for confirmation:



If it succeeds at adding the standard SenseGlove sockets, you will receive a confirmation message:

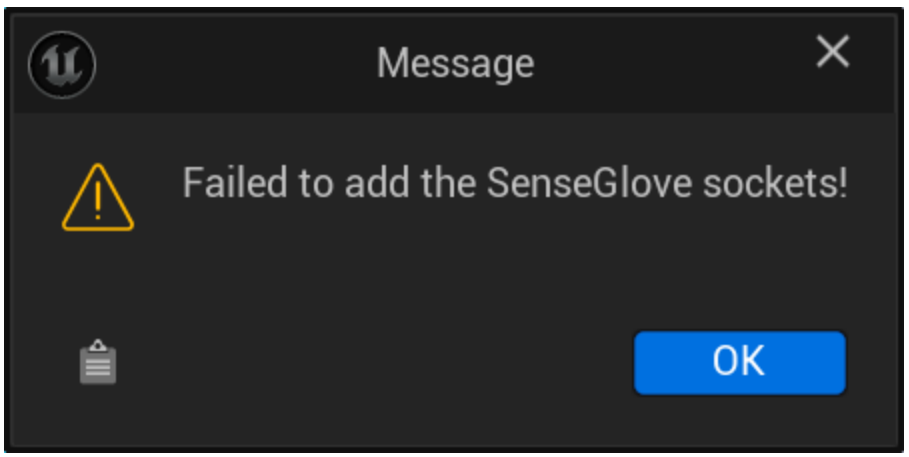


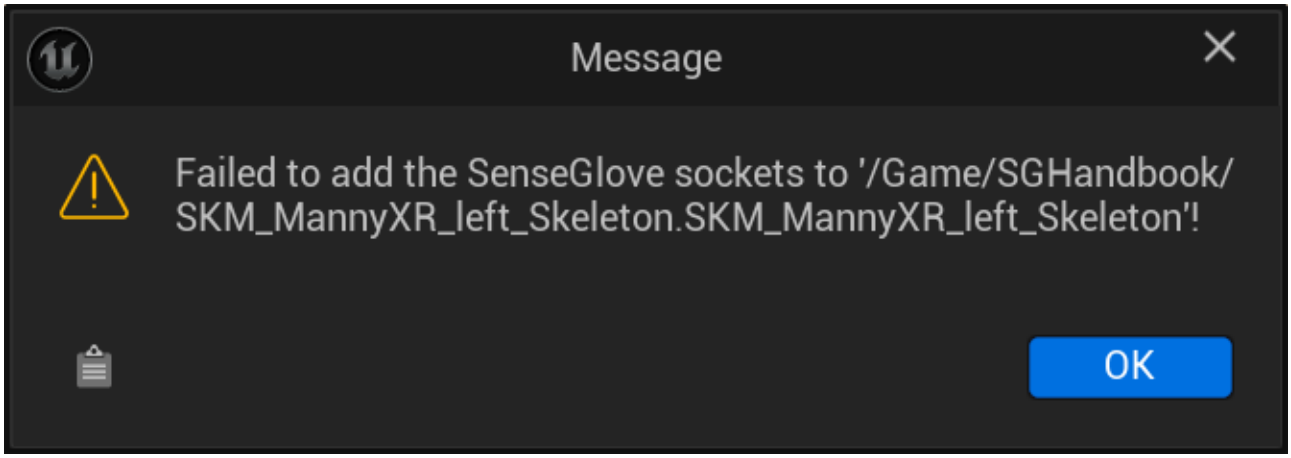
After closing the dialog, the editors for the affected Skeleton and Skeletal Mesh assets will open, displaying the newly added sockets:



To ensure the changes persist, save the assets to disk.

> **Note**
>
> The `Add SenseGlove Sockets` action can fail for various reasons, so it's important to investigate and identify the cause if an issue arises.

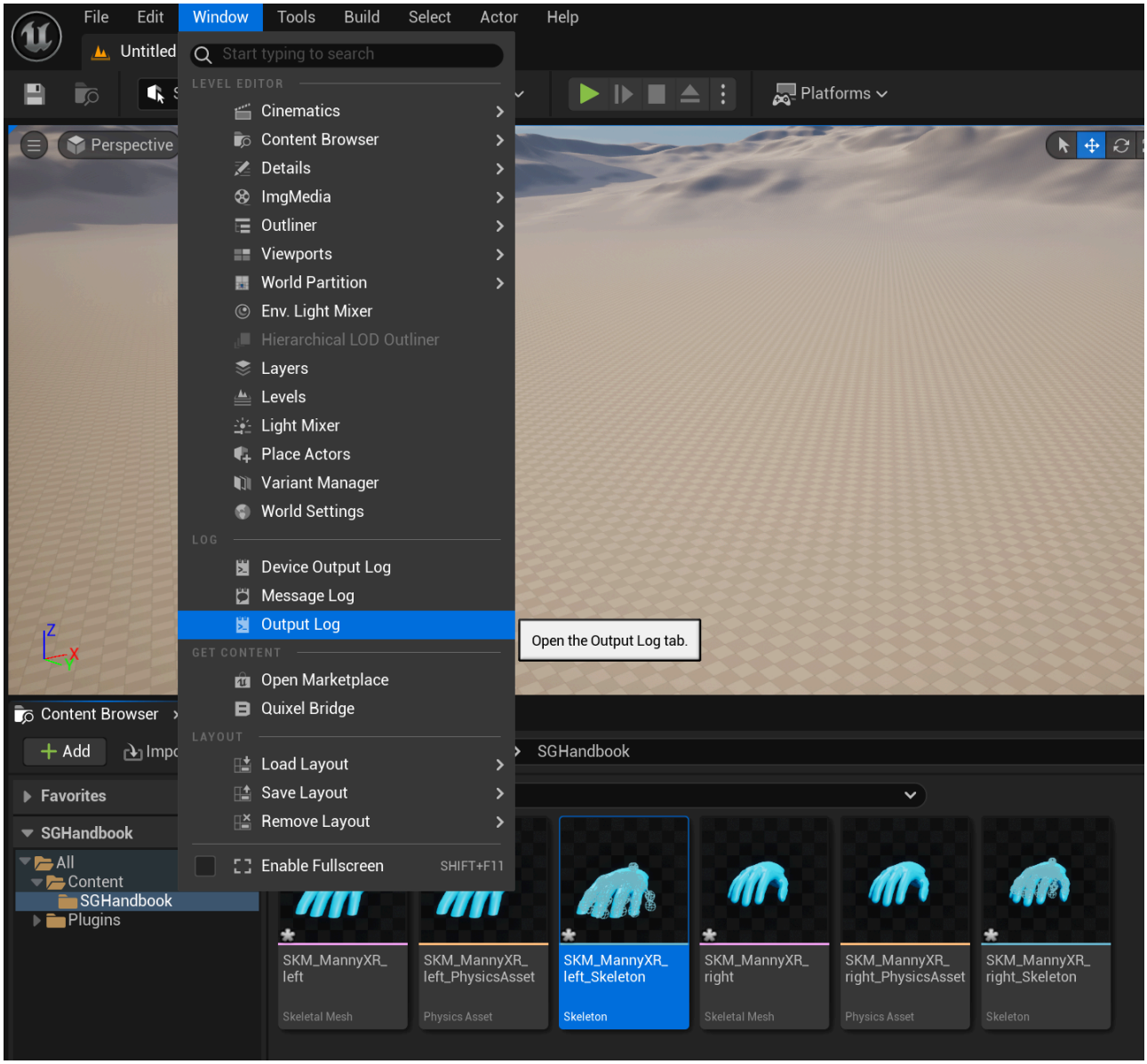




> **Important**
>
> A common cause of failure is that the SenseGlove sockets have already been set up, or the meshes you're using already have the necessary sockets. In this case, consider using the `Clear Existing Sockets` action first.

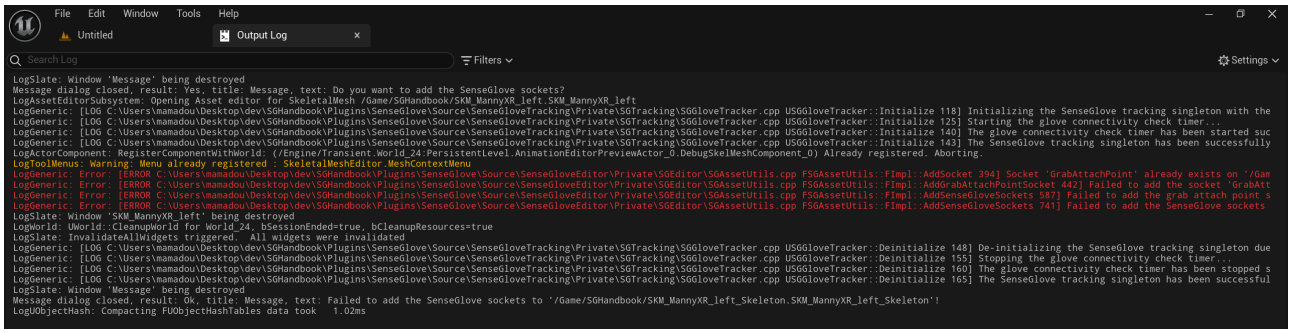
> **Caution**
>
> Another common cause of failure is if your virtual hand meshes share a skeleton. As noted in the Compatible Virtual Hand Meshes section, the SenseGlove Sockets Editor does not support skeletal meshes that share their skeleton. You may need to export and re-import the virtual hand meshes in in a compatible manner first.

In any case, the SenseGlove Sockets Editor reports all failures in the Unreal Editor logs. To view and investigate the logs, simply head to the `Window` menu and click on `Output Log`:



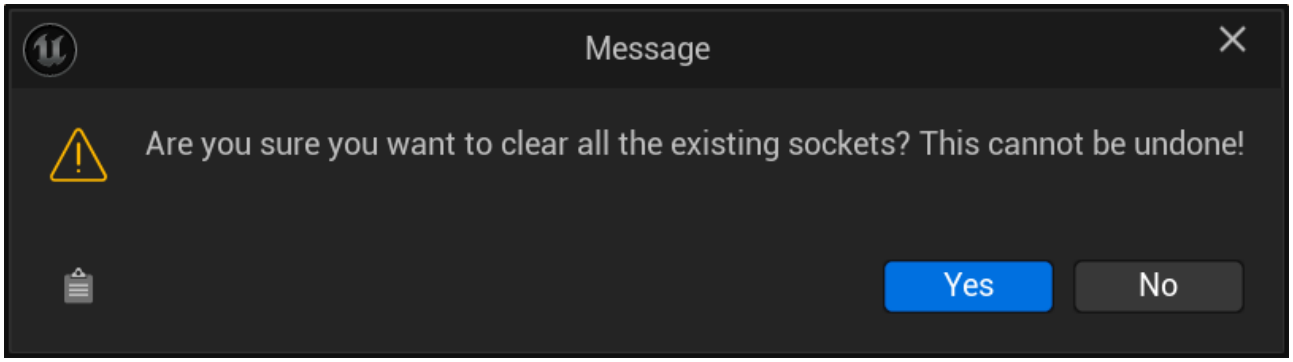
For example, in the following screenshots the following errors are stated: `Socket 'GrabAttachPoint' already exists on '/Game/SGHandbook/SKM_MannyXR_left.SKM_MannyXR_left'; refuse to add a duplicate!.`

```
LogGeneric: Error: [ERROR
C:\Users\mamadou\Desktop\dev\SGHandbook\Plugins\SenseGlove\Source\SenseGloveE
ditor\Private\SGEditor\SGAssetUtils.cpp FSGAssetUtils::FImpl::AddSocket 394]
Socket 'GrabAttachPoint' already exists on
'/Game/SGHandbook/SKM_MannyXR_left.SKM_MannyXR_left'; refuse to add a
duplicate!
LogGeneric: Error: [ERROR
C:\Users\mamadou\Desktop\dev\SGHandbook\Plugins\SenseGlove\Source\SenseGloveE
ditor\Private\SGEditor\SGAssetUtils.cpp
FSGAssetUtils::FImpl::AddGrabAttachPointSocket 442] Failed to add the socket
'GrabAttachPoint' to '/Game/SGHandbook/SKM_MannyXR_left.SKM_MannyXR_left'!
LogGeneric: Error: [ERROR
C:\Users\mamadou\Desktop\dev\SGHandbook\Plugins\SenseGlove\Source\SenseGloveE
ditor\Private\SGEditor\SGAssetUtils.cpp
FSGAssetUtils::FImpl::AddSenseGloveSockets 587] Failed to add the grab attach
point socket to asset '/Game/SGHandbook/SKM_MannyXR_left.SKM_MannyXR_left'!
LogGeneric: Error: [ERROR
C:\Users\mamadou\Desktop\dev\SGHandbook\Plugins\SenseGlove\Source\SenseGloveE
ditor\Private\SGEditor\SGAssetUtils.cpp
FSGAssetUtils::FImpl::AddSenseGloveSockets 741] Failed to add the SenseGlove
sockets to the asset '/Game/SGHandbook/SKM_MannyXR_left.SKM_MannyXR_left'!
```

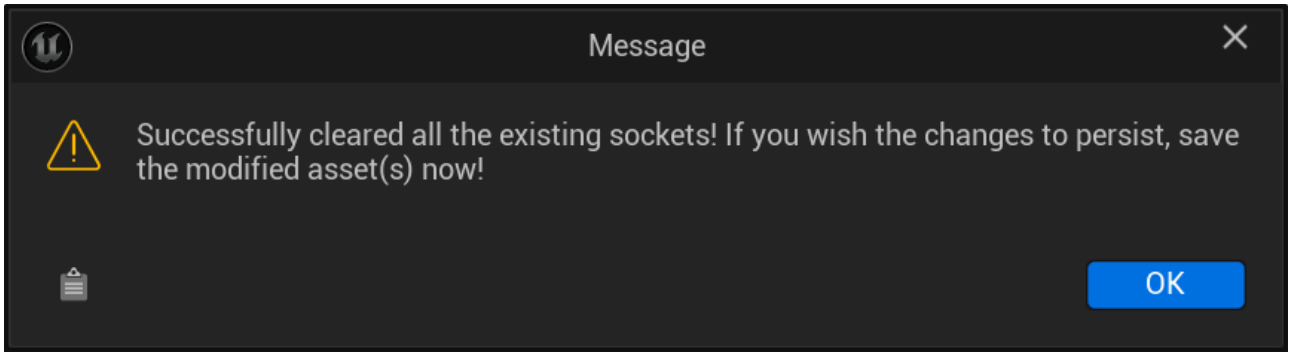


Clearing All Existing Sockets

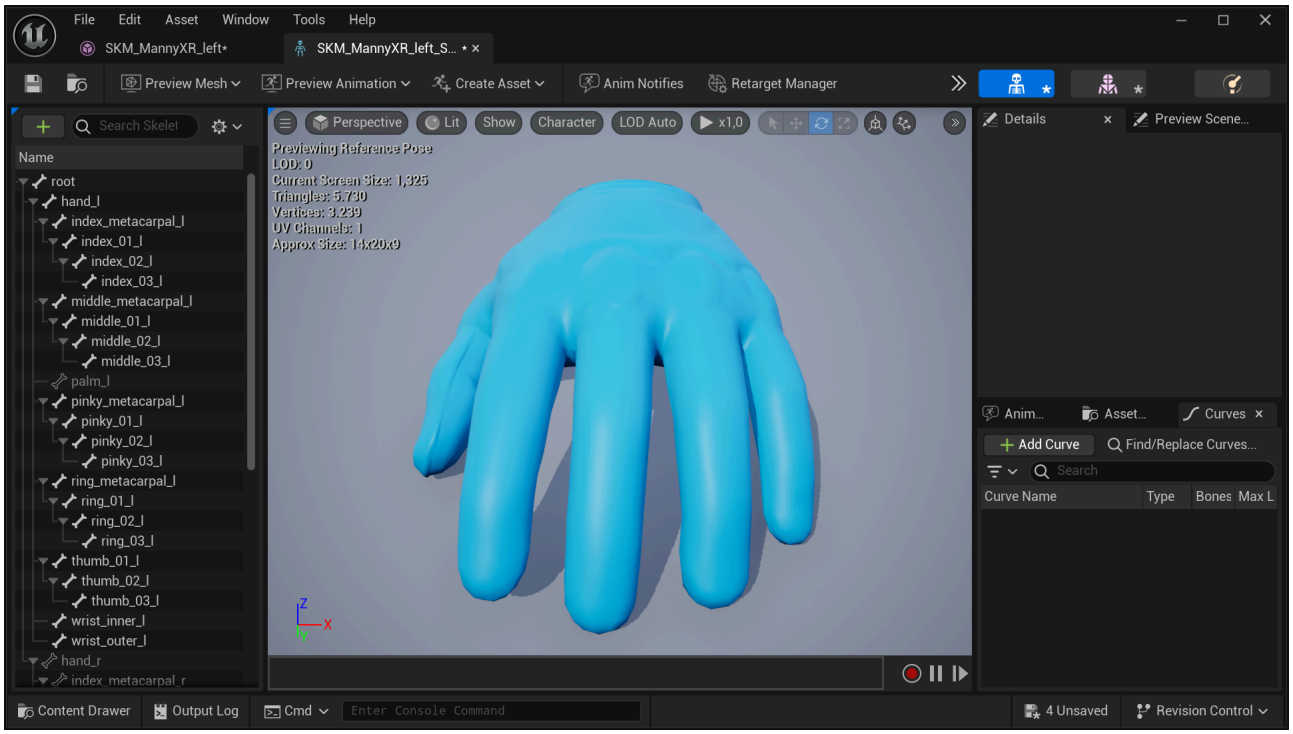
When you invoke the `Clear Existing Sockets` action, the Sockets Editor will ask for your confirmation:



If successful, you will receive a message indicating all the existing sockets have been cleared:



After closing the dialog, the editors for the affected Skeleton and Skeletal Mesh assets will open, displaying the affected assets with all sockets cleared:



Configuring the SGPawn and Plugin Virtual Hand Mesh Settings

The final step in setting up the virtual hand meshes is to configure the `SGPawn` and Plugin `Virtual Hand Mesh Settings` to ensure they utilize the newly created virtual hand meshes.

Please visit Setting Up SGPawn, The Virtual Hand Mesh Settings, and how to setup the Wrist Tracking Hardware sections for more information.

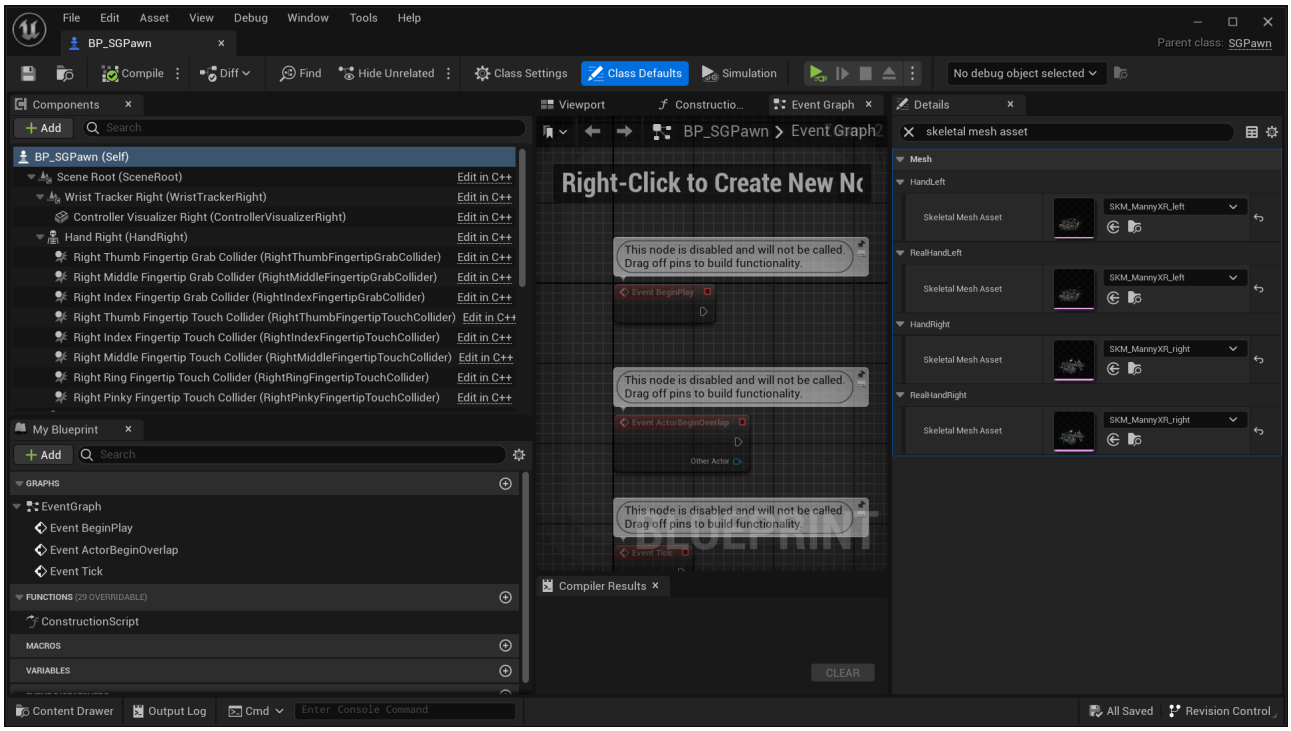
SGPawn Configuration

In the `SGPawn` Blueprint class, make sure to assign the appropriate `Skeletal Mesh Asset` to the following components:

- `HandLeft`
- `HandRight`

- `RealHandLeft`
- `RealHandRight`

This ensures that the correct hand meshes are used for both virtual and real hands.

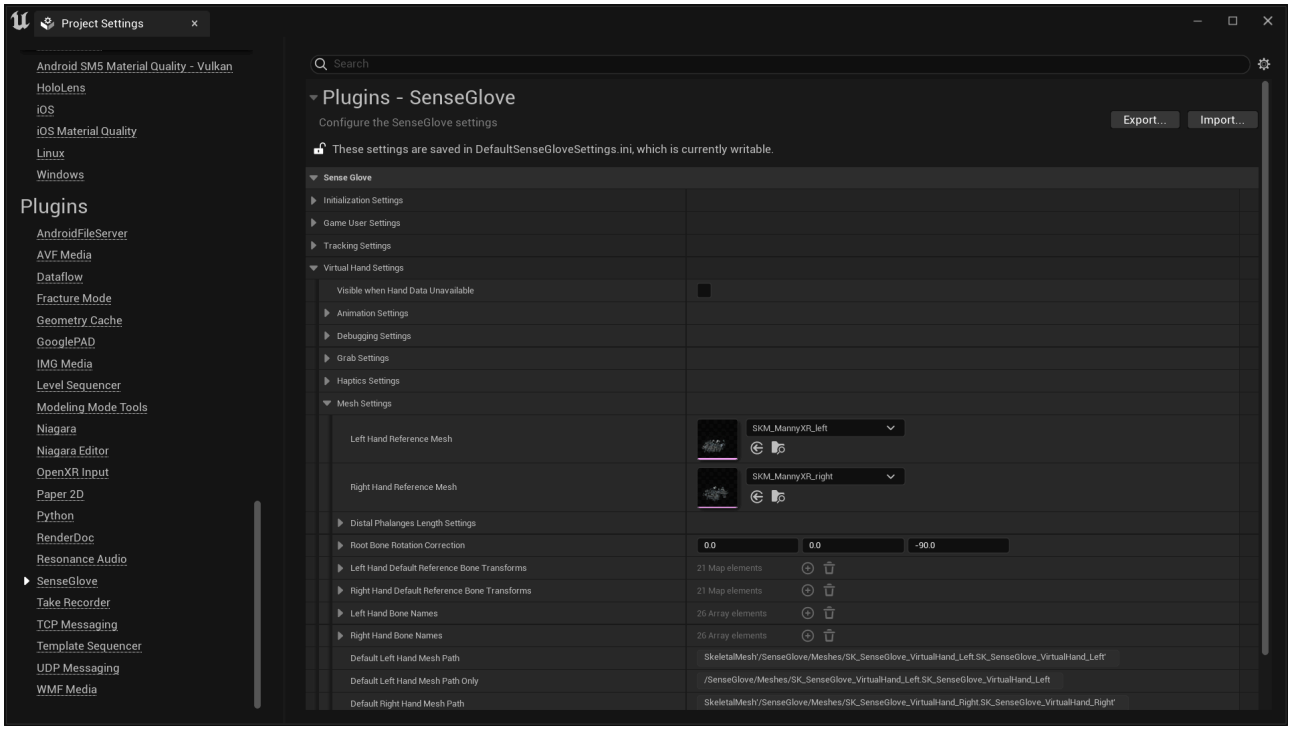


Plugin Virtual Hand Mesh Settings

Next, navigate to `Project Settings > Plugins > SenseGlove > Virtual Hand Settings > Mesh Settings` and specify the correct left and right-hand meshes for:

- `Left Hand Reference Mesh`
- `Right Hand Reference Mesh`

This configuration guarantees that the tracking system correctly interprets the bone transforms of the virtual hand meshes when generating `FXRMotionControllerData`. Additionally, it allows the animation system to accurately use these bone transforms when processing `FXRMotionControllerData` and animating the virtual hand meshes.


Project Settings
Android SM5 Material Quality - Vulkan
HoloLens
iOS
iOS Material Quality
Linux
Windows
Plugins
AndroidFileServer
AVF Media
Dataflow
Fracture Mode
Geometry Cache
GooglePAD
IMG Media
Level Sequencer
Modeling Mode Tools
Niagara
Niagara Editor
OpenXR Input
Paper 2D
Python
RenderDoc
Resonance Audio
SenseGlove
Take Recorder
TCP Messaging
Template Sequencer
UDP Messaging
WMF Media
Search
Plugins - SenseGlove
Configure the SenseGlove settings
Export...
Import...
These settings are saved in DefaultSenseGloveSettings.ini, which is currently writable.
Sense Glove
Initialization Settings
Game User Settings
Tracking Settings
Virtual Hand Settings
Visible when Hand Data Unavailable
Animation Settings
Debugging Settings
Grab Settings
Haptics Settings
Mesh Settings
Left Hand Reference Mesh
SKM_MannyXR_left
Right Hand Reference Mesh
SKM_MannyXR_right
Distal Phalanges Length Settings
Root Bone Rotation Correction
0.0
0.0
-90.0
Left Hand Default Reference Bone Transforms
21 Map elements
Right Hand Default Reference Bone Transforms
21 Map elements
Left Hand Bone Names
26 Array elements
Right Hand Bone Names
26 Array elements
Default Left Hand Mesh Path
SkeletalMesh'/SenseGlove/Meshes/SK_SenseGlove_VirtualHand_Left.SK_SenseGlove_VirtualHand_Left'
Default Left Hand Mesh Path Only
/SenseGlove/Meshes/SK_SenseGlove_VirtualHand_Left.SK_SenseGlove_VirtualHand_Left
Default Right Hand Mesh Path
SkeletalMesh'/SenseGlove/Meshes/SK_SenseGlove_VirtualHand_Right.SK_SenseGlove_VirtualHand_Right'

Setting Up the Wrist Tracking Hardware

To enable the SenseGlove Unreal Engine Plugin to track the gloves position and rotation in the world, you need to specify a positional tracking hardware, referred to as Wrist Tracking Hardware within the plugin. By default, if the Wrist Tracking Hardware is not explicitly set, the plugin will attempt to automatically detect it by identifying your Head-mounted display (HMD) hardware. However, this auto-detection feature may not be entirely reliable, as it is still experimental, and it may occasionally fail.

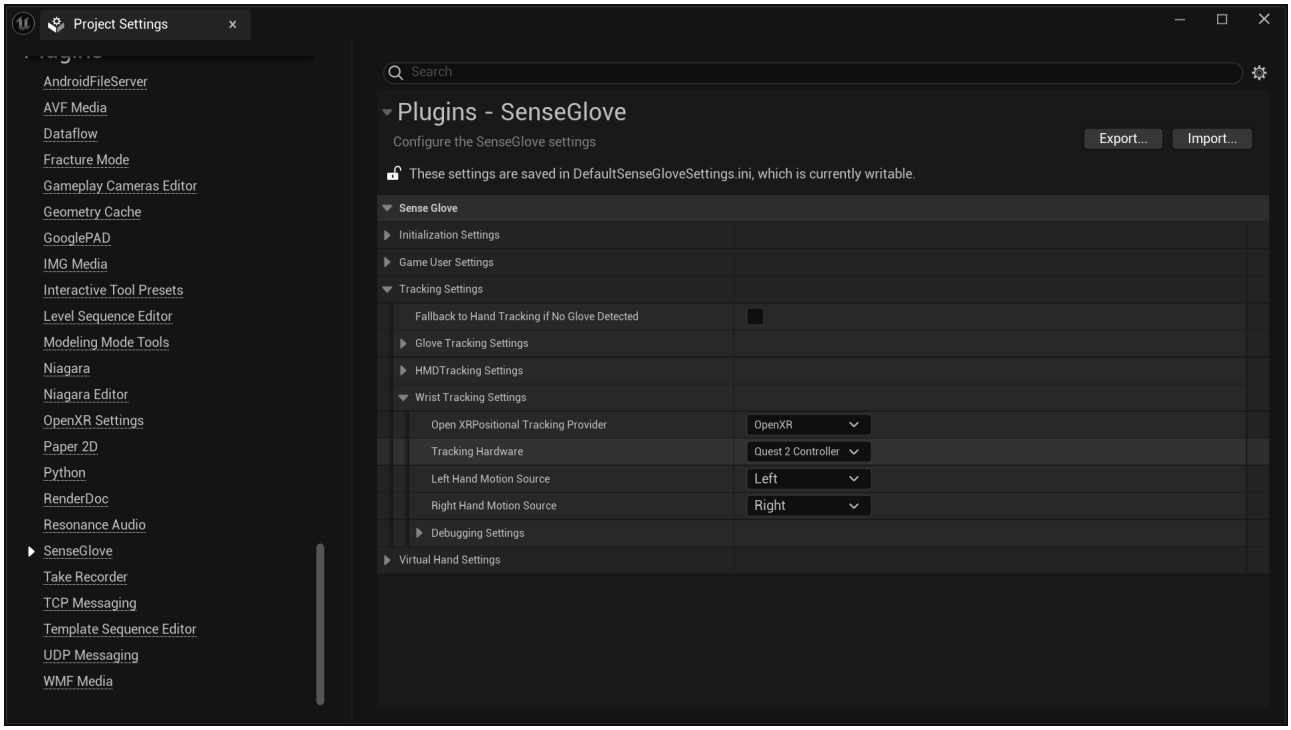
For detailed information, please visit the Wrist Tracking Hardware and HMD auto-detection configuration section.

Prerequisites

Before you even consider setting up the Wrist Tracking Settings for the SenseGlove Unreal Engine Plugin, please make sure that you have already taken all the necessary steps in the Enabling XR_EXT_hand_tracking OpenXR Extension on VR Headsets section for PCVR or Standalone modes. Otherwise, there's no guarantee that the plugin can access the location data from the wrist-tracking hardware of your choice. So, as the first troubleshooting measure, we always recommend double-checking the relevant prerequisite guides above.

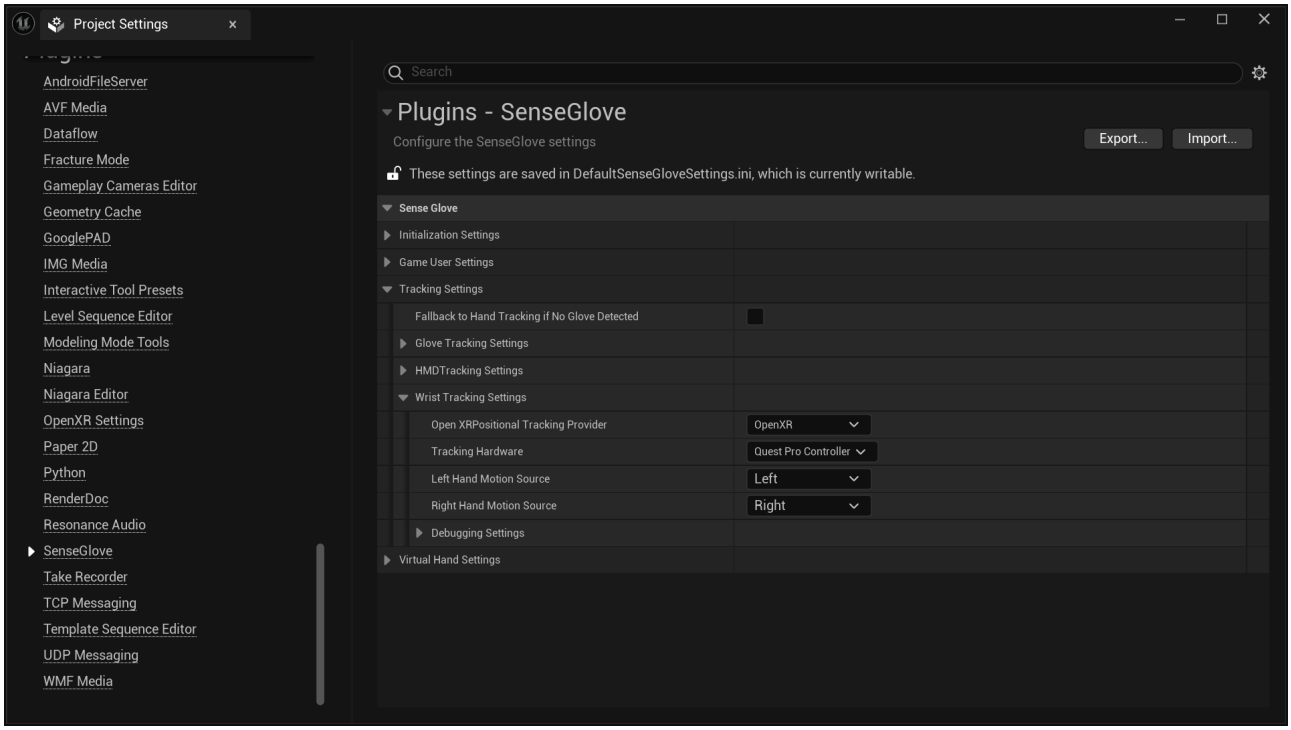
Meta Quest 2 Controller

For the Meta Quest 2, whether in PCVR or Standalone mode, the functional wrist-tracking settings looks something like this:



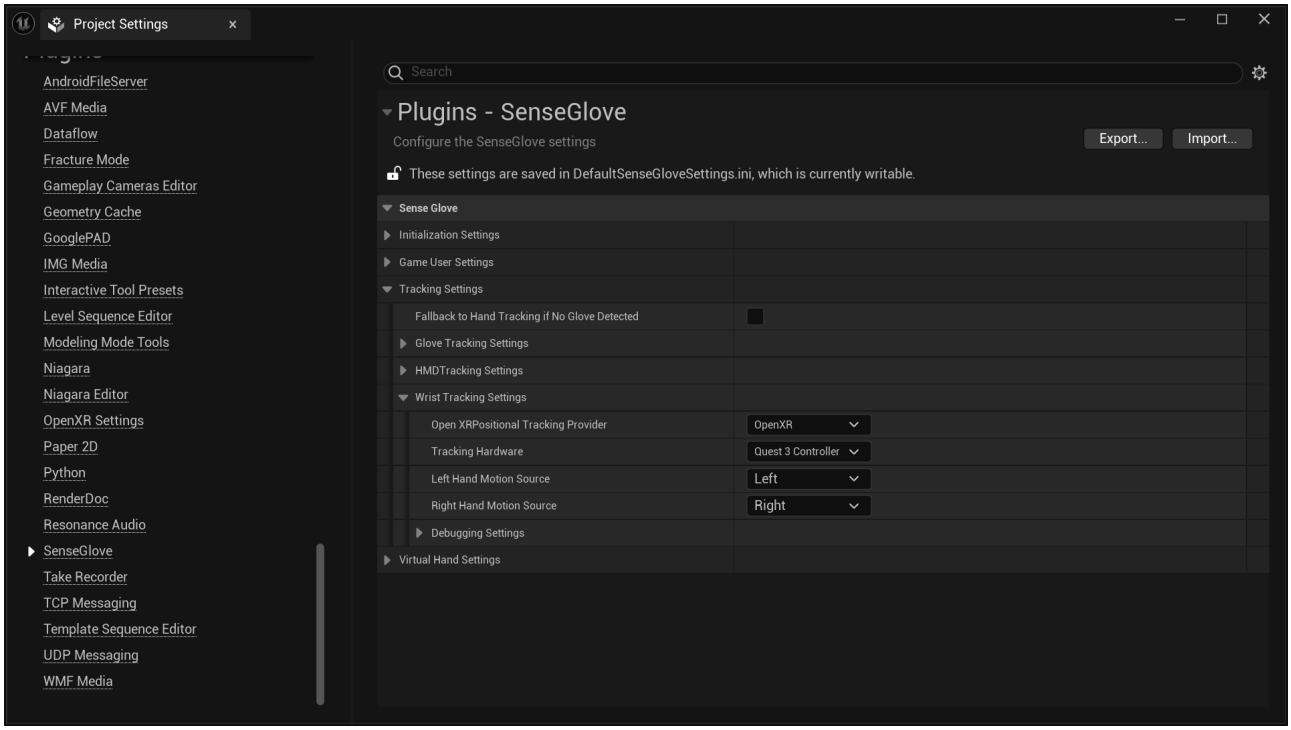
Meta Quest Pro Controller

For the Meta Quest 3, whether in PCVR or Standalone mode, the functional wrist-tracking settings looks something like this:



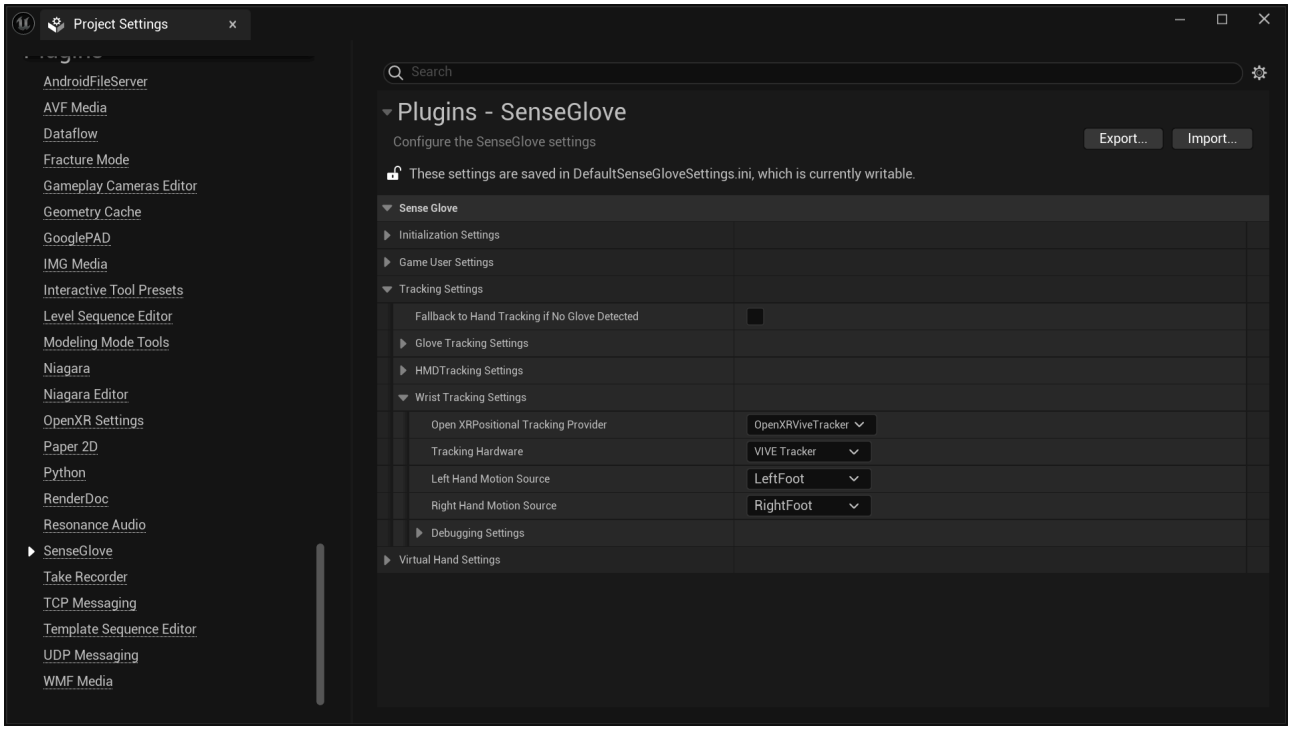
Meta Quest 3 Controller

For the Meta Quest 3, whether in PCVR or Standalone mode, the functional wrist-tracking settings looks something like this:



HTC VIVE Tracker

For the HTC VIVE Pro using VIVE Trackers, which only supports the PCVR mode, the functional wrist-tracking settings looks something like this:

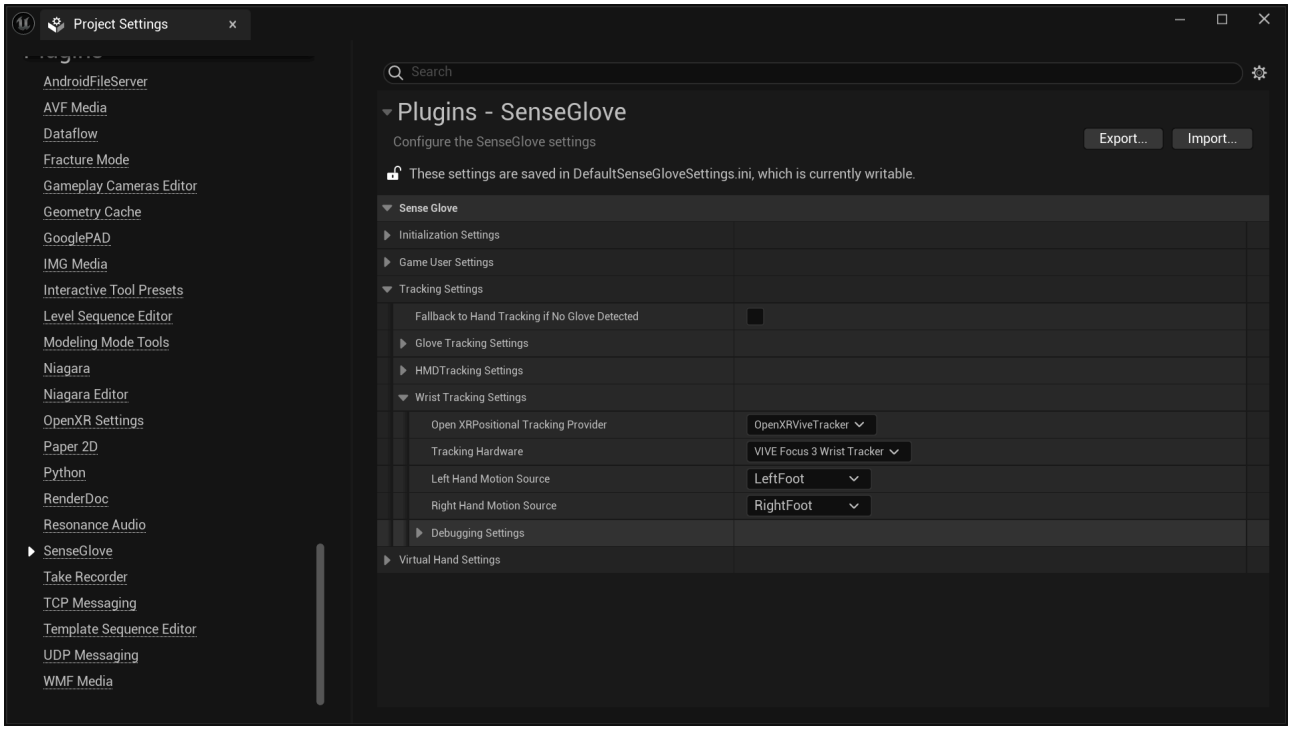


HTC VIVE Focus 3 Wrist Tracker

The wrist-tracking settings for the HTC VIVE Focus Vision, VIVE XR Elite, and VIVE Focus 3, when using VIVE Wrist Trackers, will vary depending on the platform and configuration in use.

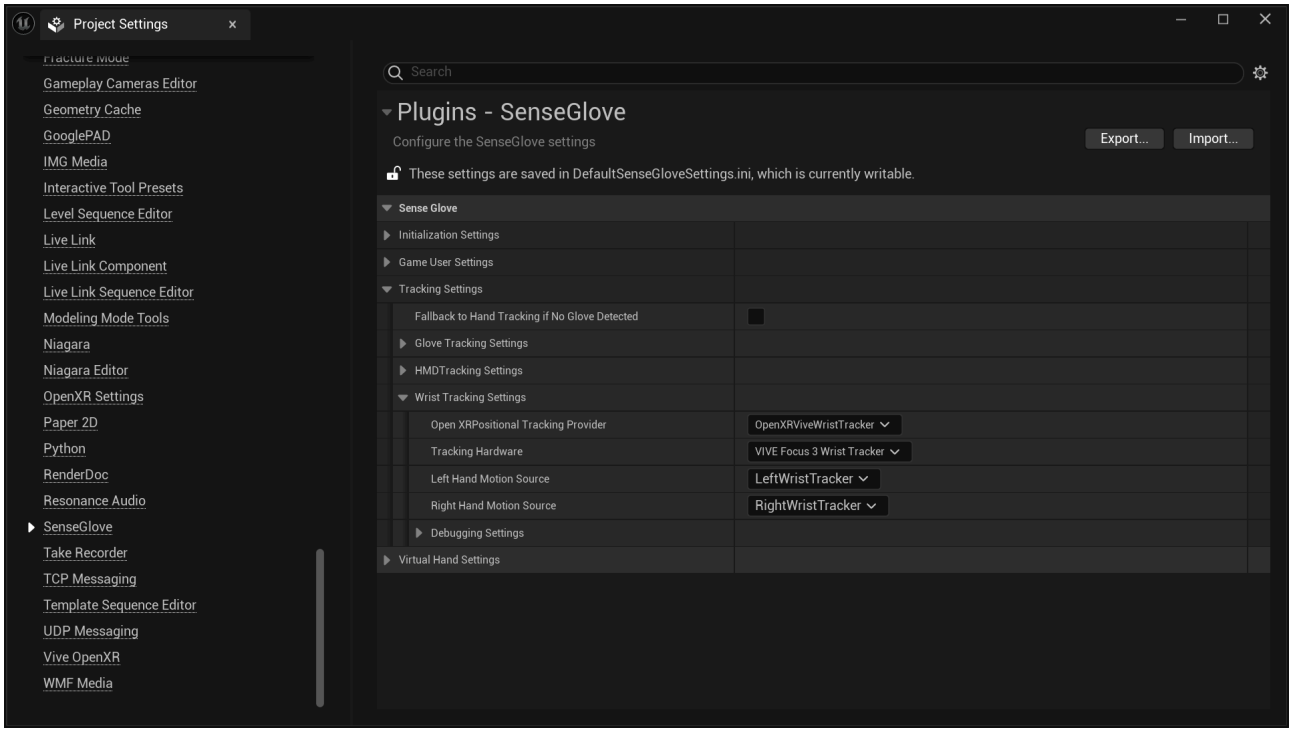
PCVR Mode

When running in PCVR mode, the functional wrist-tracking settings looks something like this:



Standalone Mode

When running in Standalone mode, the functional wrist-tracking settings looks something like this:

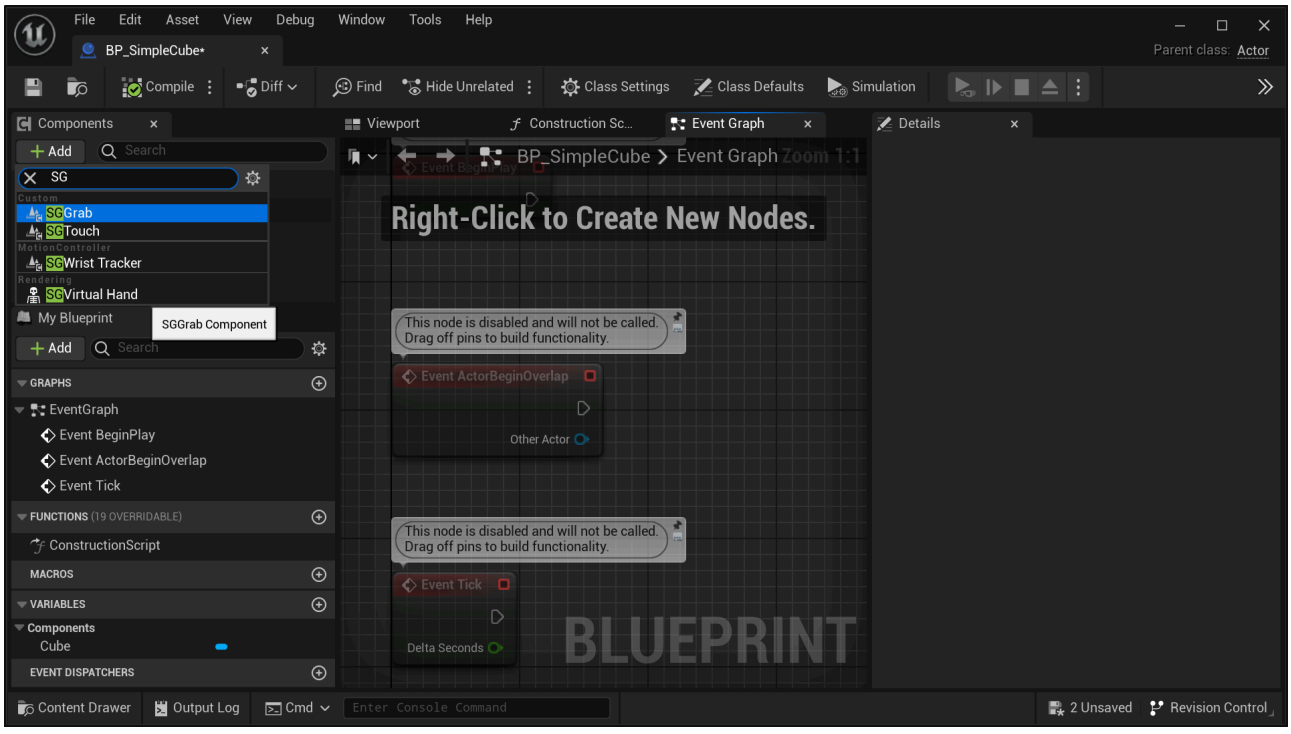

Project Settings
Gameplay Cameras Editor
Geometry Cache
GooglePAD
IMG Media
Interactive Tool Presets
Level Sequence Editor
Live Link
Live Link Component
Live Link Sequence Editor
Modeling Mode Tools
Niagara
Niagara Editor
OpenXR Settings
Paper 2D
Python
RenderDoc
Resonance Audio
SenseGlove
Take Recorder
TCP Messaging
Template Sequence Editor
UDP Messaging
Vive OpenXR
WMF Media
Search
Plugins - SenseGlove
Configure the SenseGlove settings
Export...
Import...
These settings are saved in DefaultSenseGloveSettings.ini, which is currently writable.
Sense Glove
Initialization Settings
Game User Settings
Tracking Settings
Fallback to Hand Tracking if No Glove Detected
Glove Tracking Settings
HMDTracking Settings
Wrist Tracking Settings
Open XRPositional Tracking Provider
OpenXRViveWristTracker
Tracking Hardware
VIVE Focus 3 Wrist Tracker
Left Hand Motion Source
LeftWristTracker
Right Hand Motion Source
RightWristTracker
Debugging Settings
Virtual Hand Settings

Setting up the Grab/Release System

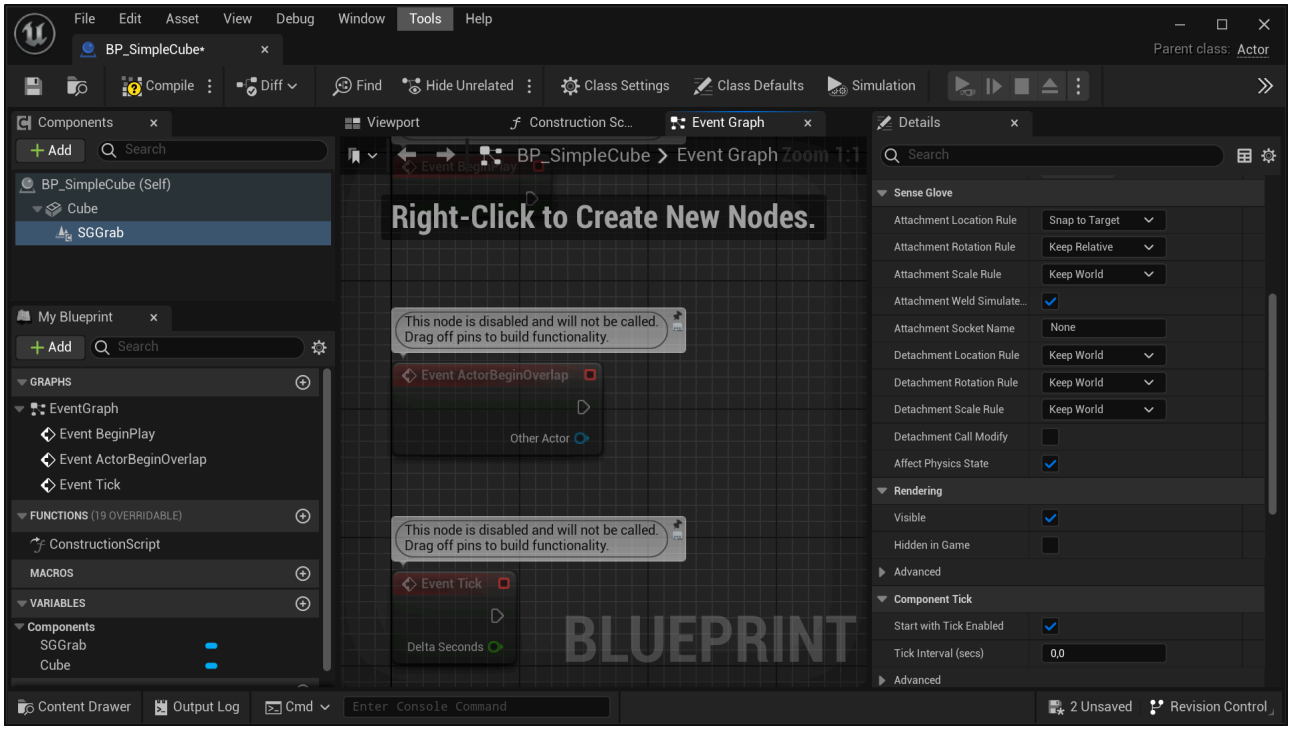
Setting up the SenseGlove Grab/Release System involves two main steps. The first step, configuring the virtual hand meshes for both real and virtual hands, is handled automatically by the plugin. The second step, which is also straightforward, involves setting up any existing actor in the Unreal Blueprint Editor that you want to respond to with haptic feedback when your SenseGlove device comes into contact with it:

1. Open any existing actor in the Unreal Blueprint Editor that you would like to respond to with haptic feedback when your SenseGlove device comes into contact with it.

2. In the `Components` panel, click the `+ Add` button, then type `SGGrab` into the `Search Components` input field. Once found, click on `SGGrab` to add it to the current actor. You can rename the `SGGrab` component to your desired name.



3. With the `SGGrab` component selected in the `Components` panel, navigate to the `Details` panel. Under the `SenseGlove` section, adjust the settings for the grab/release system to suit your needs.



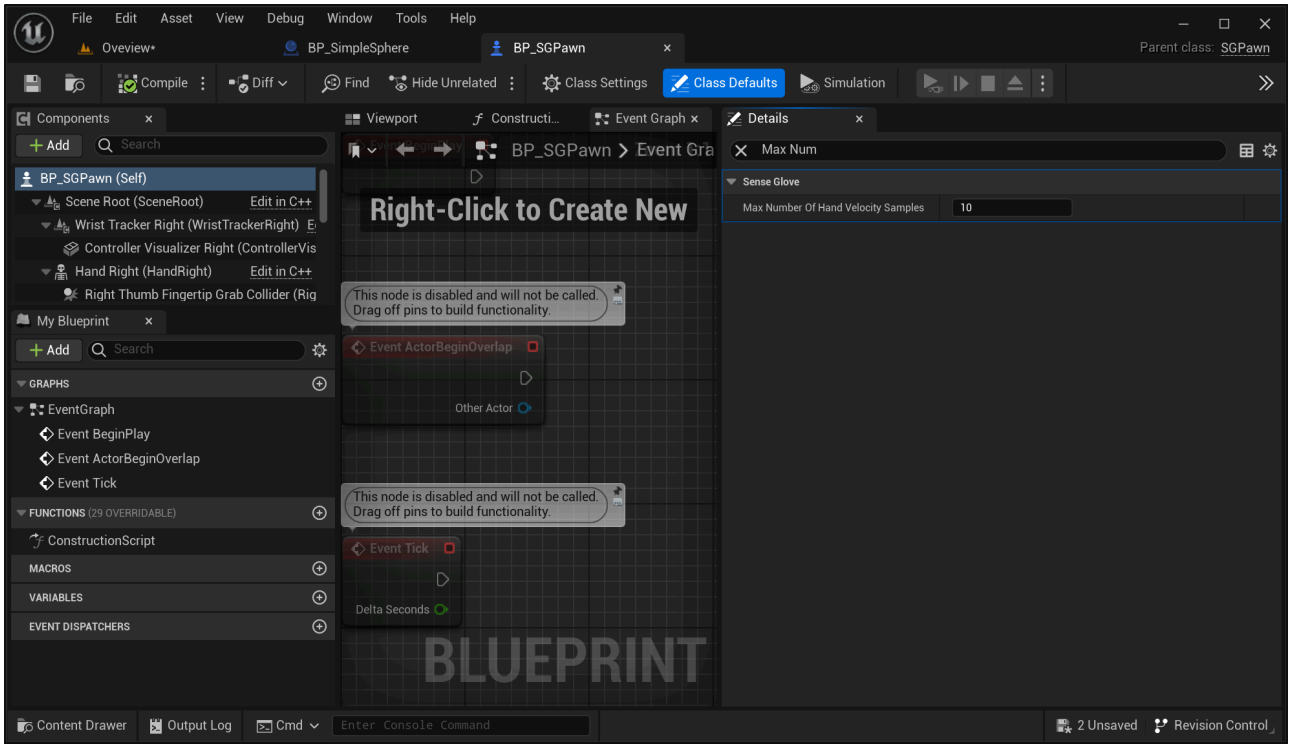
Note

Any property prefixed with `Attachment` is a parameter directly passed to Unreal's `FAttachmentTransformRules` during the grab process, while any property prefixed with `Detachment` is a parameter directly passed to Unreal's `FDetachmentTransformRules` during the release process.

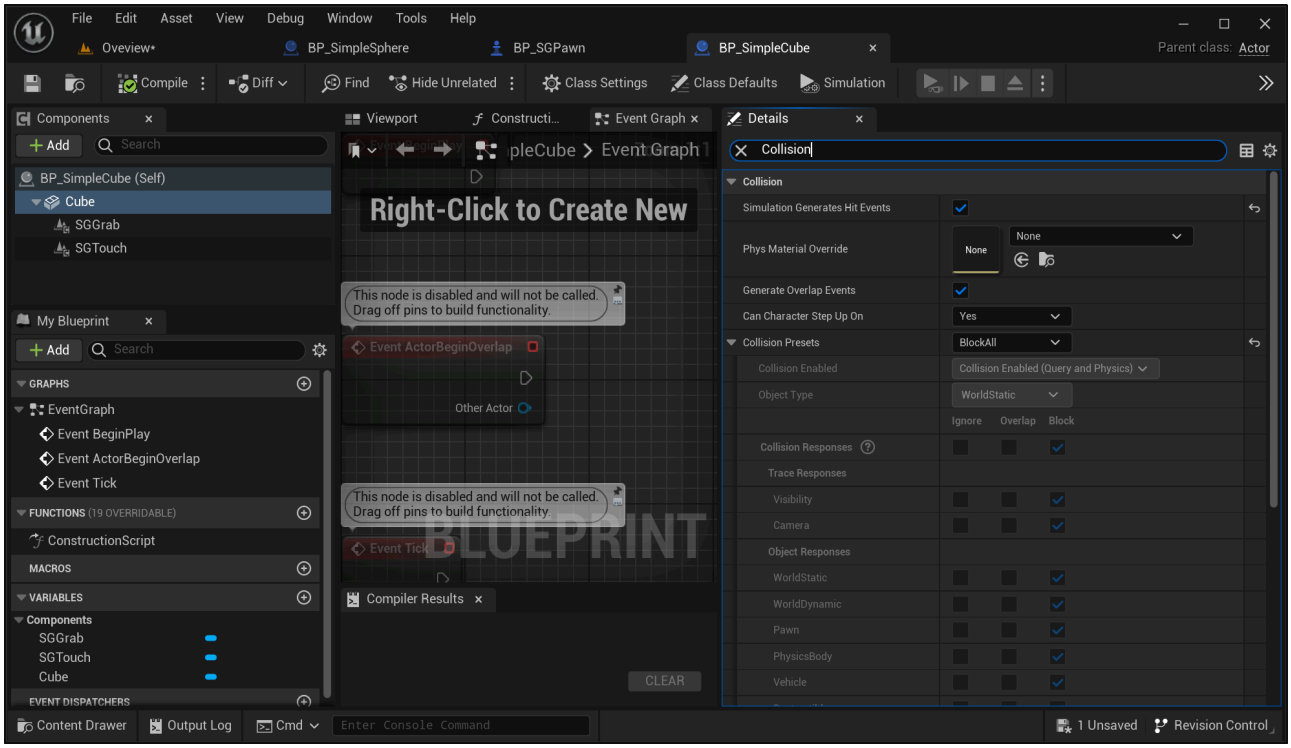
Caution

If `AttachmentSocketName` is unspecified, or incorrect the grabbable object will be attached to the root bone of the virtual hand mesh, which probably is not ideal.

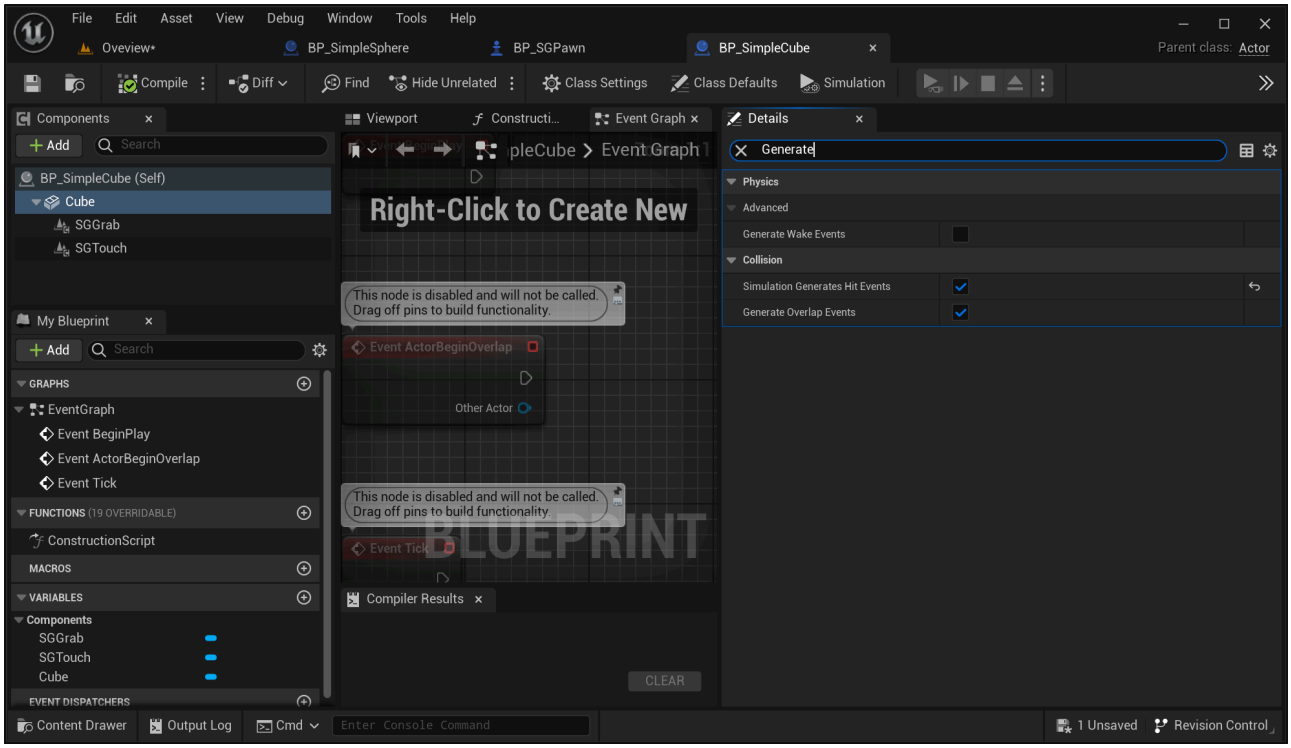
4. A key setting for the release system is located within your `SGPawn` instance. In the `Details` panel for your `SGPawn`, find the `Max Number of Hand Velocity Samples` setting and adjust it according to your needs. This setting determines the velocity of objects released from the hands by averaging the specified number of frames. Optimizing this value depends on the framerate of your simulation at runtime.



5. One last aspect of the grabbable actors to take into account for the grab system to function properly is the collision settings of their mesh components. If you'd like to prevent the virtual hand meshes from passing through a grabbable actor, it's necessary to set the `Collision Presets` to `Block All` inside the `Details` panel for the actor's mesh components.



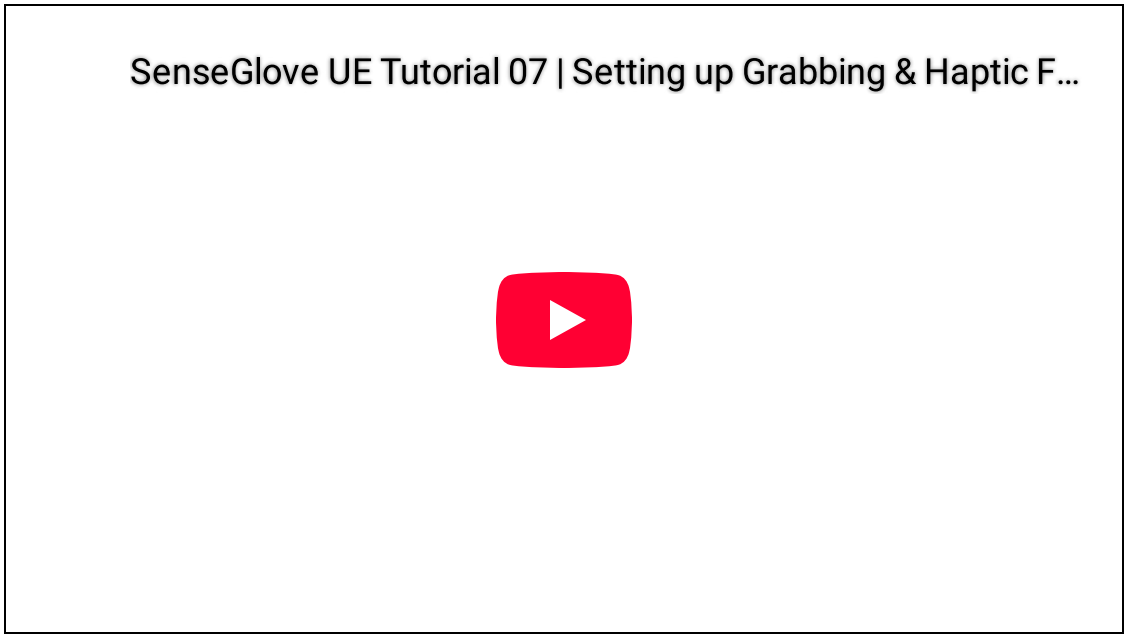
6. Additionally, enabling `Simulation Generates Hit Events` and `Generate Overlap Events` on the actors mesh components is mandatory. These settings are crucial for notifying the grab system when the virtual hand meshes come into contact with the actor.



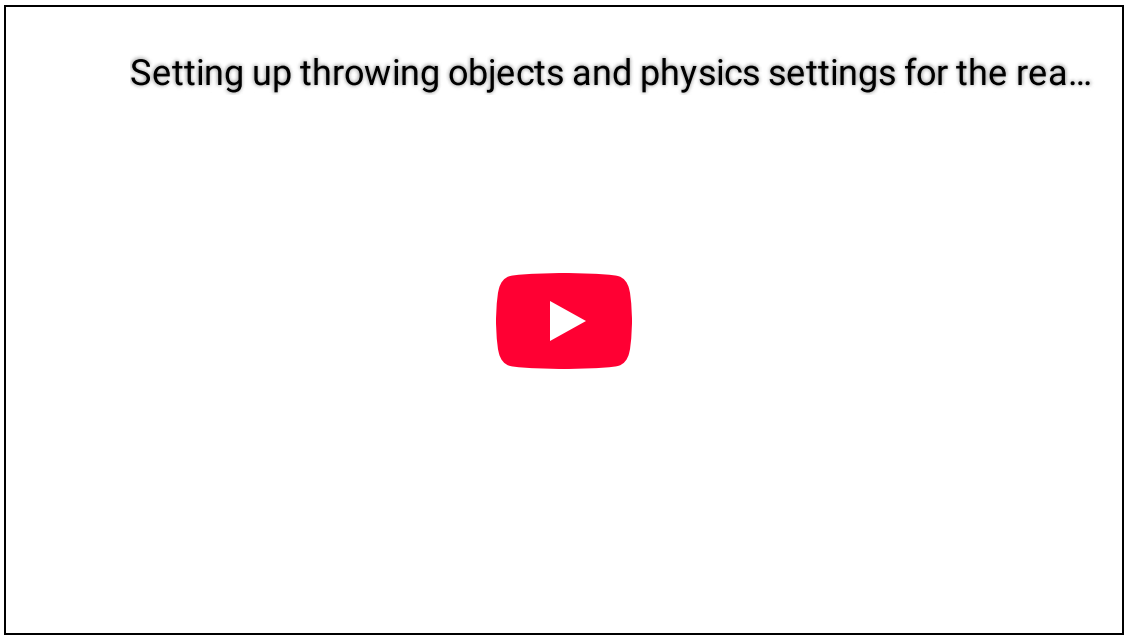
Video Tutorials

The following tutorials, though for much older releases of the plugin, still provide in-depth guidance on the same process:

- Setting up Grabbing and Haptic Feedback functionalities (SGBasicDemo)



- SGBasicDemo: setup throwing objects and physics settings for the real and virtual hands

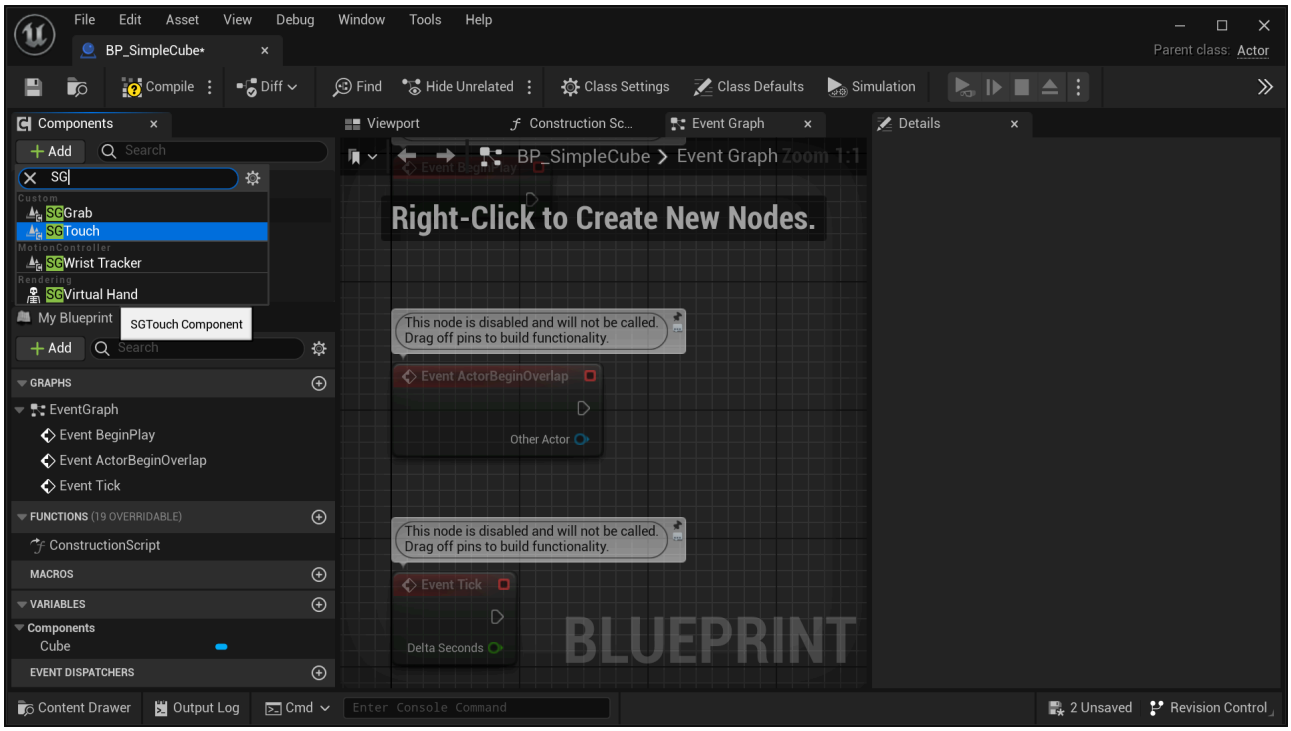


Setting up the Touch System

Configuring the SenseGlove Touch System involves two key steps. The first step, which is automatically handled by the plugin, is configuring the virtual hand meshes for both real and virtual hands. The second step, which is also straightforward, involves setting up any existing actor in the Unreal Blueprint Editor that you want to respond to with haptic feedback when your SenseGlove device comes into contact with it:

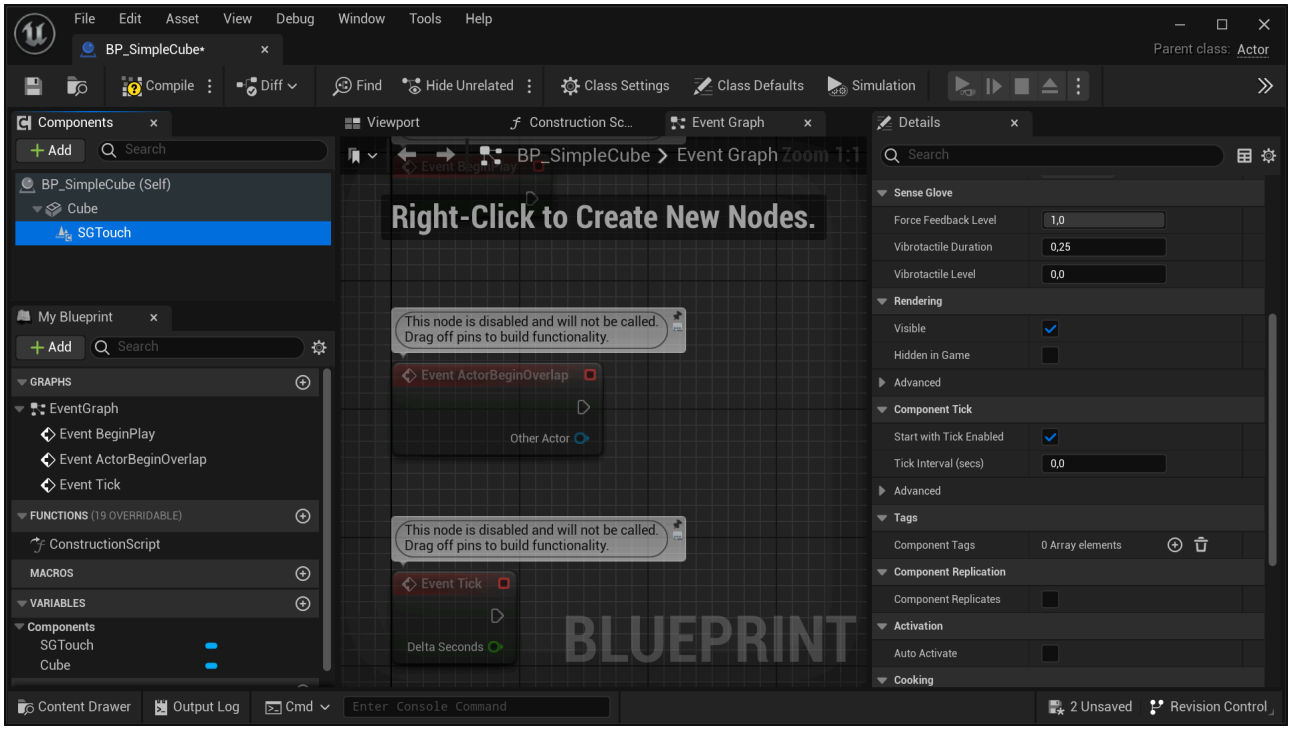
1. Open any existing actor in the Unreal Blueprint Editor that you would like to respond to with haptic feedback when your SenseGlove device comes into contact with it.

2. In the `Components` panel, click the `+ Add` button, then type `SGTouch` into the `Search Components` input field. Once found, click on `SGTouch` to add it to the current actor. You can rename the `SGTouch` component to your desired name.

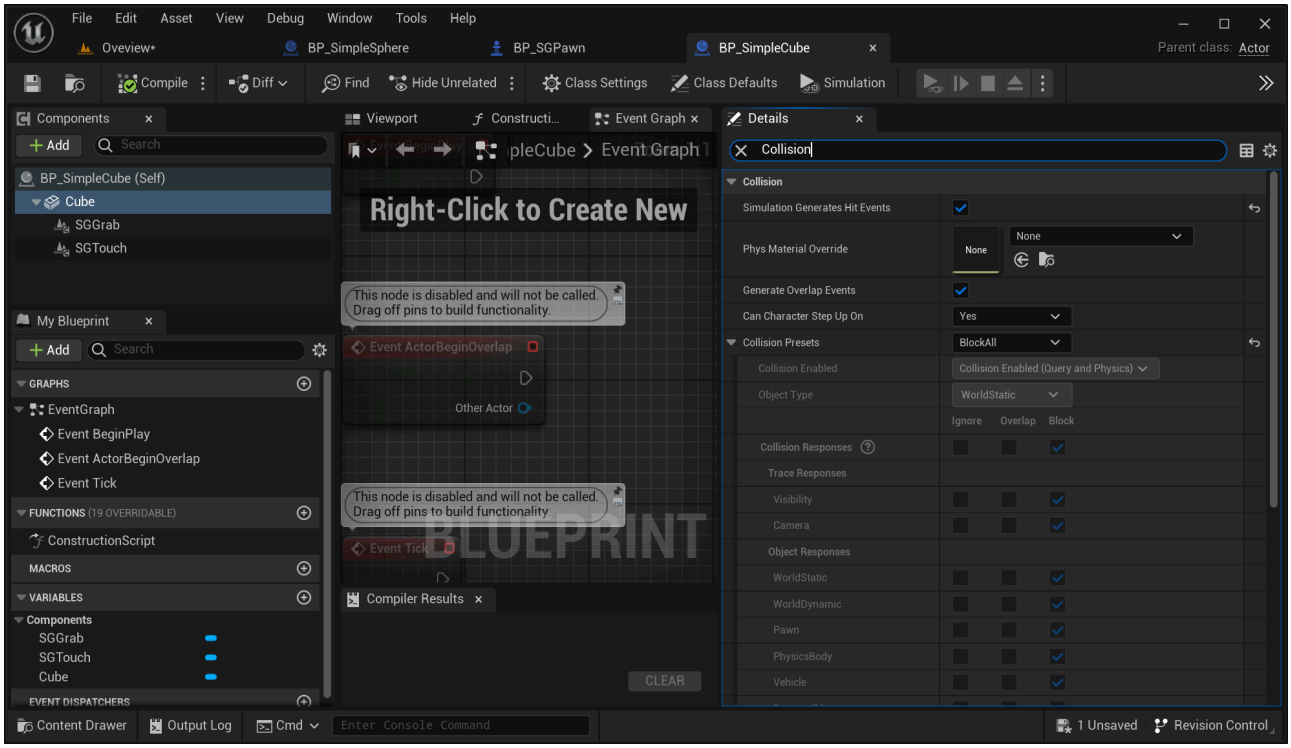


3. With the `SGTouch` component selected in the `Components` panel, navigate to the `Details` panel. Under the `SenseGlove` section, adjust the settings for the touch

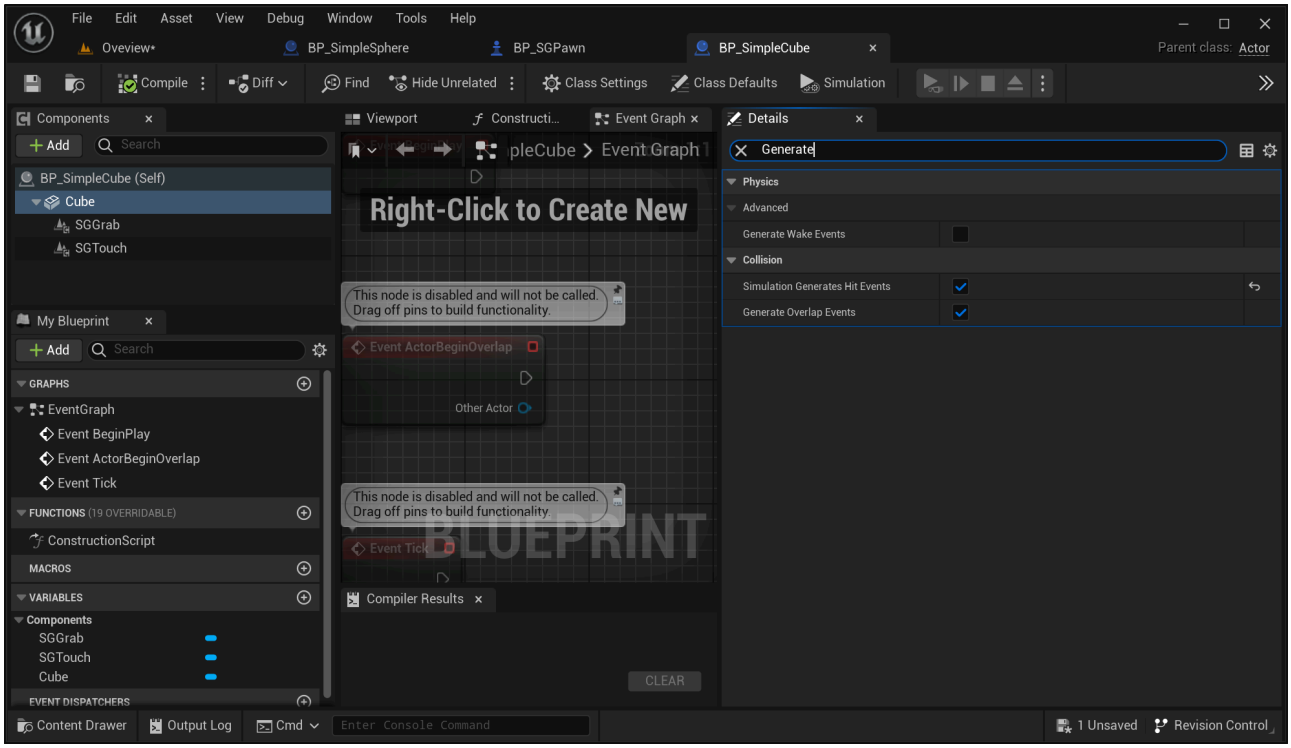
system to suit your needs.



4. One last aspect of the touchable actors to take into account for the touch system to function properly is the collision settings of their mesh components. If you'd like to prevent the virtual hand meshes from passing through a touchable actor, it's necessary to set the `Collision Presets` to `Block All` inside the `Details` panel for the actor's mesh components.



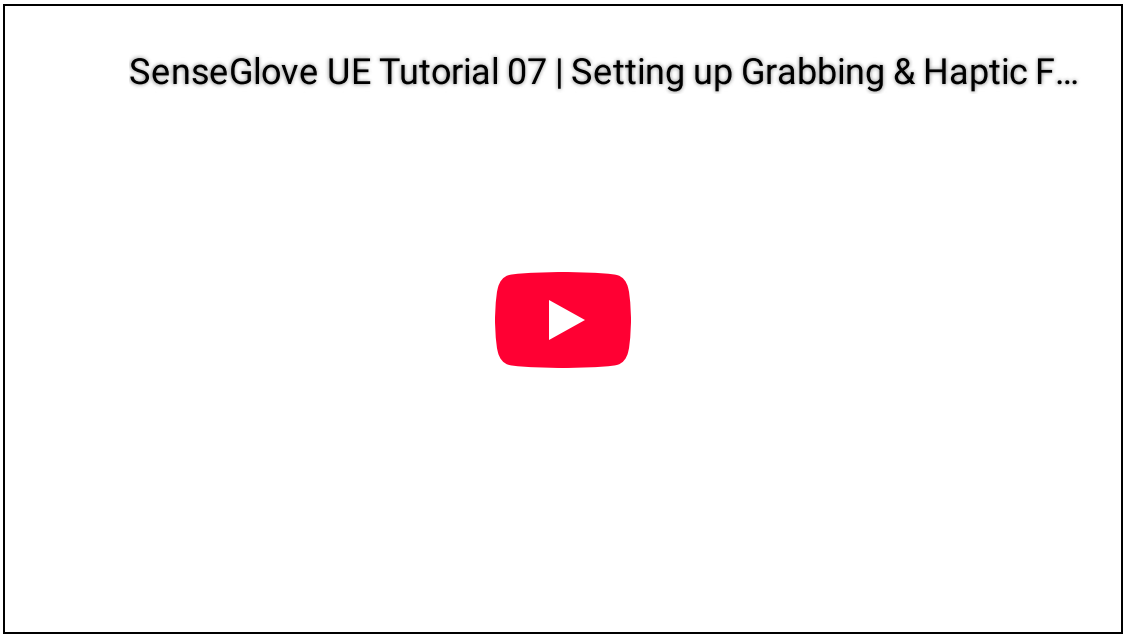
5. Additionally, enabling `Simulation Generates Hit Events` and `Generate Overlap Events` on the actors mesh components is mandatory. These settings are crucial for notifying the touch system when the virtual hand meshes come into contact with the actor.



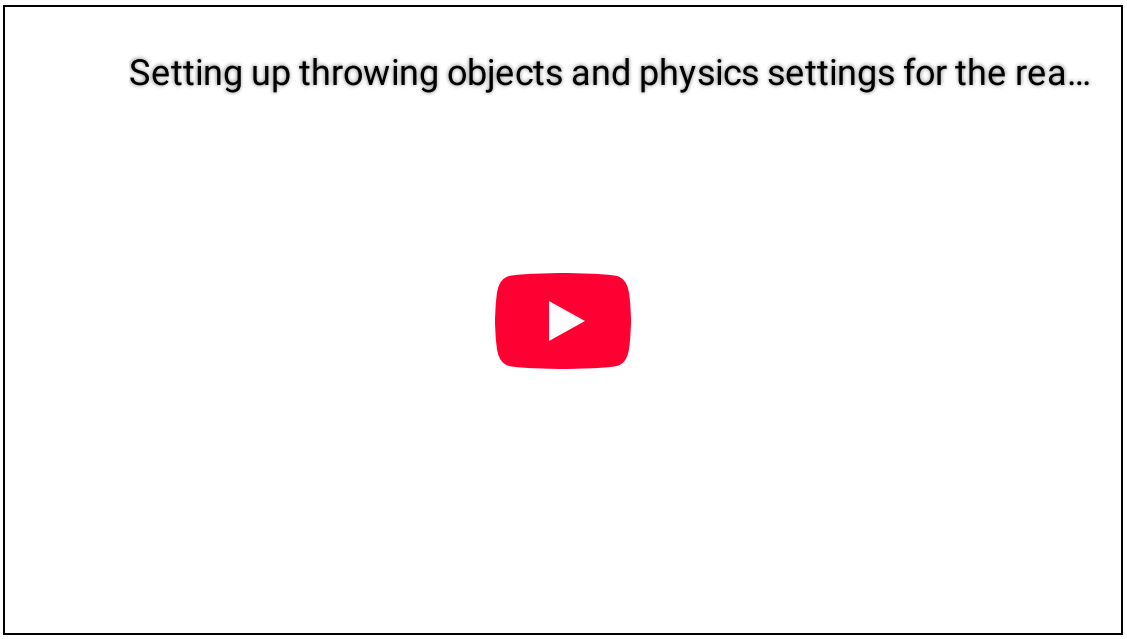
Video Tutorials

The following tutorials, though for much older releases of the plugin, still provide in-depth guidance on the same process:

- Setting up Grabbing and Haptic Feedback functionalities (SGBasicDemo)

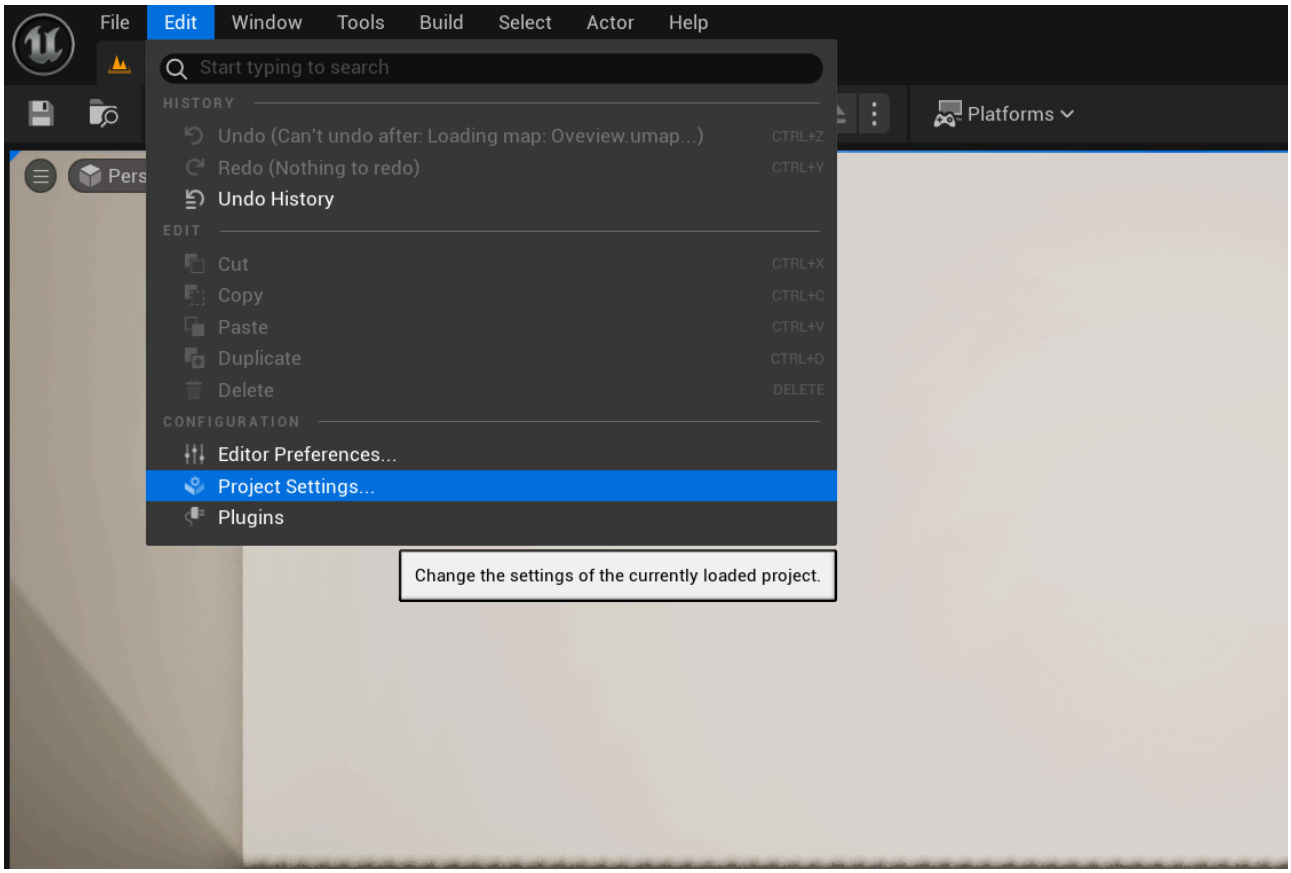


- SGBasicDemo: setup throwing objects and physics settings for the real and virtual hands

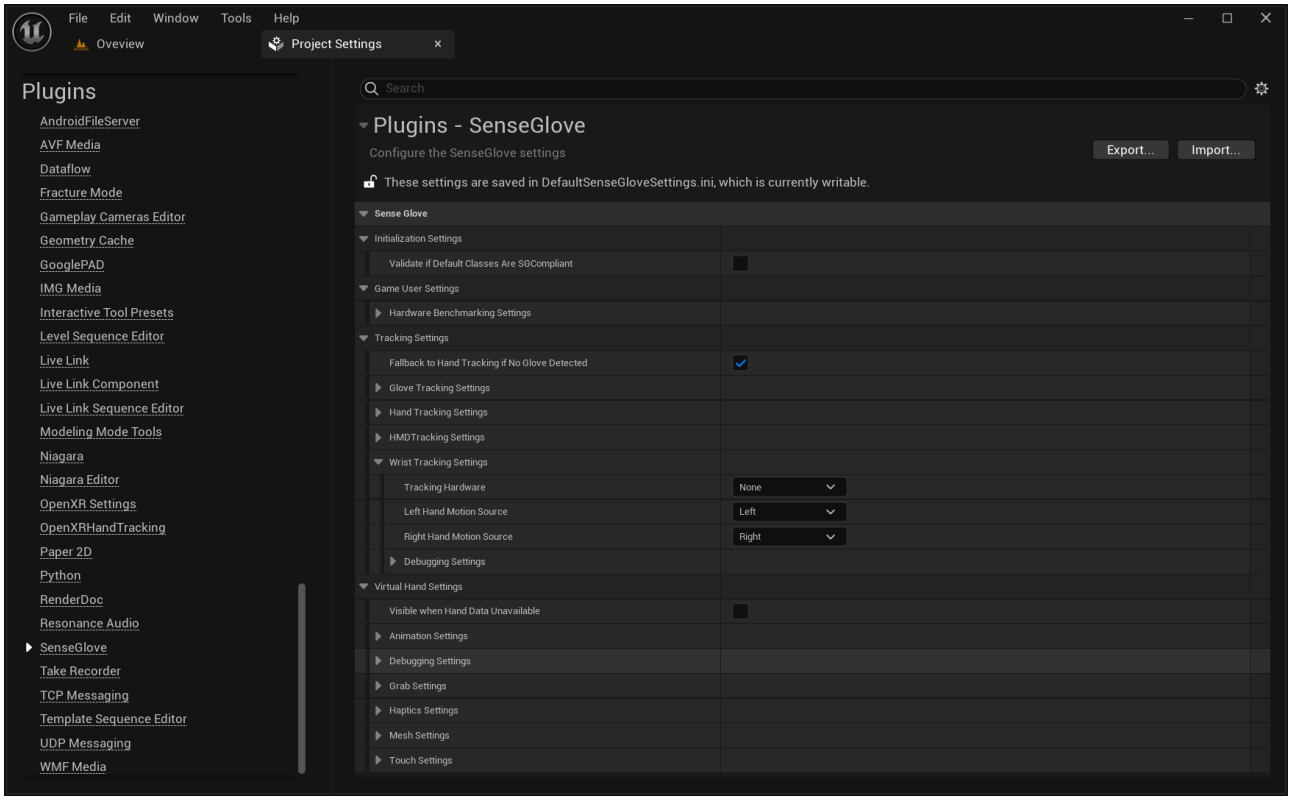


The Plugin Settings

Once the SenseGlove Unreal Engine Plugin is enabled the plugin settings can be accessed through `Edit > Project Setting...` inside your project's Unreal Editor.



The SenseGlove Unreal Engine Plugin offers fine-grained control over various aspects of its functionality through its settings system. It also allows you to override specific settings from subcomponents when possible. In the following sections, we will explore the settings and the override system in detail.



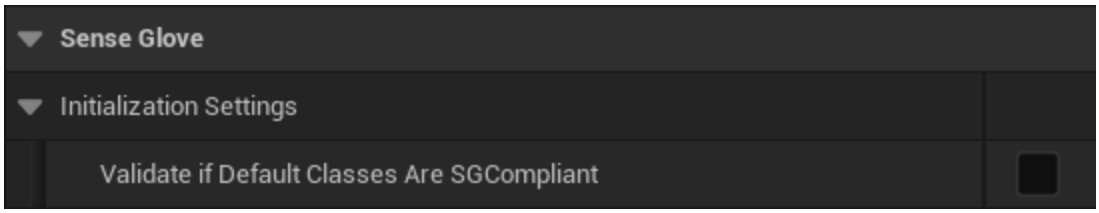
Settings Categories

The plugin settings are organized into four main categories, and each of those might contain its own sub-categories. These main categories are as follows:

- The Initialization Settings
- The Game User Settings
- The Tracking Settings
- The Virtual Hand Settings

The Plugin Initialization Settings

The Initialization Settings section is designed to control how the SenseGlove Unreal Engine Plugin is initialized, allowing you to customize its behavior to suit your project's needs.



bValidateIfDefaultClassesAreSGCompliant

If enabled, the plugin tries to check and validate whether the default for classes such as GameMode, GameInstance, etc. are indeed SenseGlove classes or SenseGlove-derived classes. If not, it attempts to set them. If you don't like this behavior for whatever reason, consider disabling this option.

By default, this option is disabled.

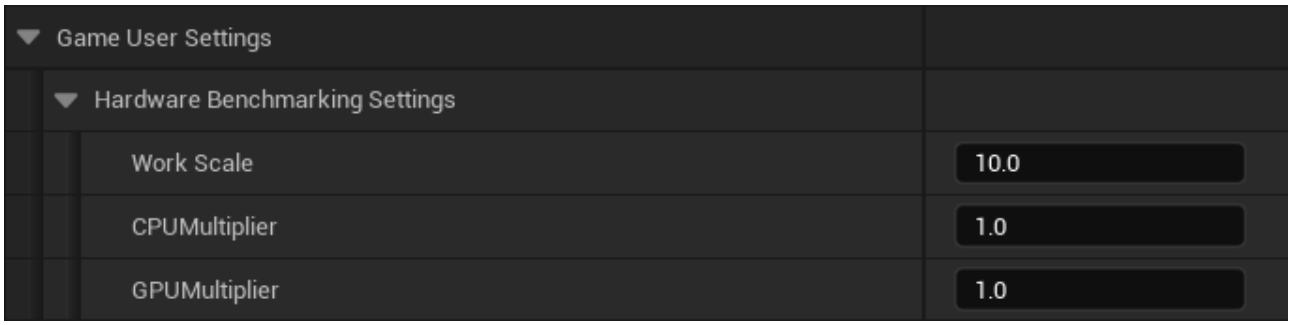
> **Caution**
>
> Due to the current initialization mechanism, setting the default classes might occasionally fail. Therefore, it's essential to verify that the default classes have been correctly set. You can do this by checking the following sections in the project settings:
>
> - `Project Settings > Project > Maps & Modes > Default Modes > Default GameMode`
> - `Project Settings > Project > Maps & Modes > Default Modes > Selected GameMode > Default Pawn Class`
> - `Project Settings > Project > Maps & Modes > Default Modes > Selected GameMode > Player Controller Class`

- `Project Settings > Project > Maps & Modes > Game Instance > Game Instance Class`
- `Project Settings > Engine > General Settings > Default Classes > Advanced > Game User Settings Class`

For more information visit the SenseGlove default classes.

The Game User Settings

The Game User Settings control the behavior of the SenseGlove instance of `UGameUserSettings`. The `USGGameUserSettings` class extends the functionality of `UGameUserSettings` to provide enhanced customization options specifically for applications that utilize the SenseGlove Unreal Engine Plugin.



The Hardware-benchmarking Settings

The settings in this section are utilized by the `USGGameUserSettings::SetEngineScalabilitySettings()` method when the `Scalability` parameter is set to `ESGEngineScalabilitySettings::Auto`. When the engine scalability settings set to auto the graphics settings are determined by running a hardware benchmark by calling the `UGameUserSettings::RunHardwareBenchmark()`. The settings listed here are basically the parameters passed to `UGameUserSettings::RunHardwareBenchmark()`.



WorkScale

The `WorkScale` parameter determines the intensity of the benchmark test. Higher values result in more intensive testing, which can help achieve more accurate scalability settings.

The default value is `10`.

CPUMultiplier

The `CPUMultiplier` parameter allows you to adjust the impact of CPU performance on the benchmark results. Increasing this value will emphasize CPU performance more heavily in determining scalability settings.

The default value is `1.0f`.

GPUMultiplier

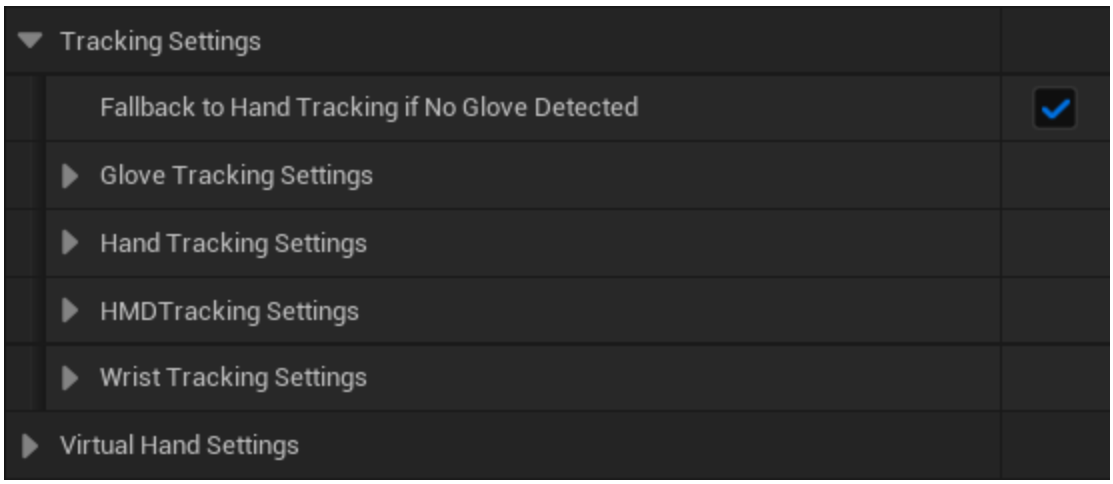
The `GPUMultiplier` parameter lets you modify the influence of GPU performance on the benchmark outcomes. A higher value will increase the weight of GPU performance in setting scalability.

The default value is `1.0f`.

The Tracking Settings

The tracking settings are primarily used by the SenseGlove `Tracking` module and are divided into various subsections, each focusing on a specific aspect of tracking. These subsections, along with the other settings directly provided by this section, provide comprehensive control over the tracking functionalities. The subsections are as follows:

- The Glove-tracking Settings
- The Hand-tracking Settings
- The HMD-tracking Settings
- The Wrist-tracking Settings



bFallbackToHandTrackingIfNoGloveDetected

Determines whether to fallback to hand-tracking, or not, when no SenseGlove device is detected:

- If disabled, only a real glove will be tracked.
- If enabled, the plugin will fall back to hand-tracking when it's available and supported by the HMD device.

> **Note**

> Disabling this option hides the hand-tracking settings section, while enabling it makes the hand-tracking settings visible.

Glove Tracking Settings

Provides the tracking settings related to SenseGlove devices.

Hand Tracking Settings

The settings in this section only affects the hand-tracking functionality when it's enabled and available. When enabled the bare hands can be used instead of SenseGlove devices to interact within the VR simulation, of course without the haptics feedback provided by the SenseGlove devices.

> **Important**
>
> If you don't see the hand-tracking settings, ensure that the option `bFallbackToHandTrackingIfNoGloveDetected` is checked.

HMD Tracking Settings

Provides the tracking settings related to head-mounted displays (HDMs) and their auto-detection functionality.

Wrist Tracking Settings

Provides the tracking settings applicable to wrist-tracking hardware.

The Glove-tracking Settings

Provides the tracking settings related to SenseGlove devices.



GloveConnectivityCheckInterval

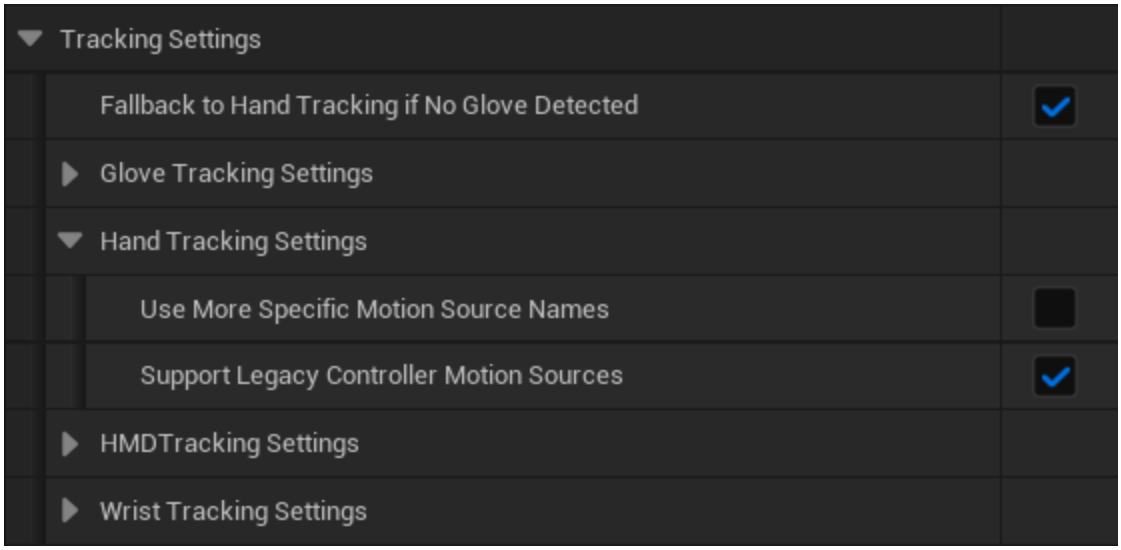
The interval in which the tracking module checks for glove connectivity.

The default is `16.666666f` which means `60` times per second.

The Hand-tracking Settings

The settings in this section only affects the hand-tracking functionality when it's enabled and available. When enabled the bare hands can be used instead of SenseGlove devices to interact within the VR simulation, of course without the haptics feedback provided by the SenseGlove devices.

> **Important**
>
> If you don't see the hand-tracking settings, ensure that the option `bFallbackToHandTrackingIfNoGloveDetected` is checked.



bUseMoreSpecificMotionSourceNames

If disabled, (the default) the motion sources for hand tracking will be of the form `[Left|Right][Keypoint]`. If enabled, they will be of the form `HandTracking[Left|Right][Keypoint]`. It is recommended to be enabled to avoid collisions between motion sources from different device types.

bSupportLegacyControllerMotionSources

If enabled, hand tracking supports the `Left` and `Right` legacy motion sources. If disabled, it does not. It is recommended to be disabled unless you need legacy compatibility in older unreal projects.

The HMD-tracking Settings

Provides the tracking settings related to head-mounted displays (HDMs) and their auto-detection functionality.



ViveHMDDetectionPriority

Determines which VIVE HMD to prioritize for detection, as the current detection mechanism cannot differentiate between HTC VIVE Focus Vision, HTC VIVE XR Elite, and HTC VIVE Focus 3.

The following values are possible:

```
/*
 * The HTC VIVE HMD detection priority for HTC devices that we cannot
distinguish.
 */
UENUM(BlueprintType)
enum class ESGViveHMDDetectionPriority : uint8
{
    /* First try to detect HTC VIVE Focus Vision, then XR Elite, and then
Focus 3. */
    FocusVision_XRElite_Focus3 UMETA(DisplayName = "Focus Vision (1st), XR
Elite (2nd), Focus3 (3rd)"),

    /* First try to detect HTC VIVE Focus Vision, then Focus 3, and then XR
Elite. */
    FocusVision_Focus3_XRElite UMETA(DisplayName = "Focus Vision (1st),
Focus3 (2nd), XR Elite (3rd)"),

    /* First try to detect HTC VIVE XR Elite, then Focus Vision, and then
Focus 3. */
    XRElite_FocusVision_Focus3 UMETA(DisplayName = "XR Elite (1st), Focus
Vision (2nd), Focus3 (3rd)"),

    /* First try to detect HTC VIVE XR Elite, then Focus 3, and then Focus
Vision. */
    XRElite_Focus3_FocusVision UMETA(DisplayName = "XR Elite (1st), Focus3
(2nd), Focus Vision (3rd)"),

    /* First try to detect HTC VIVE Focus 3, then Focus Vision, and then XR
Elite. */
    Focus3_FocusVision_XRElite UMETA(DisplayName = "Focus3 (1st), Focus
Vision (2nd), XR Elite (3rd)"),

    /* First try to detect HTC VIVE Focus 3, then XR Elite, and then Focus
Vision. */
    Focus3_XRElite_FocusVision UMETA(DisplayName = "Focus3 (1st), XR Elite
(2nd), Focus Vision (3rd)"),
};
```

The Wrist-tracking Settings

Provides the tracking settings applicable to wrist-tracking hardware.

OpenXRPositionalTrackingProvider

Specifies the type of OpenXR provider to use in order to extract the positional tracking data from the underlying XR system. If set to None, the plugin attempts to set this automatically by considering a combination of approaches including, the current value of `TrackingHardware` specified below, the platform it's targeted at, the available OpenXR plugins, along with HMD auto-detection mechanism to specify a compatible OpenXR tracking provider. Please note that the OpenXR provider depends on the combination of plugins, platform, and the settings you use. For example, it is possible to use a Vive Focus 3 Wrist Tracker on Microsoft Windows along with the Epic `OpenXRViveTracker` plugin and check the option `Emulate VIVE Wrist Tracker as VIVE Tracker` inside the VIVE Business Streaming application's Input settings. In that case, the correct OpenXR positional tracking provider to use would be `OpenXRViveTracker`. However, using the official ViveOpenXR plugin on Android, the correct OpenXR provider would be `OpenXRViveWristTracker`.

> **Caution**
>
> HMD auto-detection is currently an experimental feature and may fail because HMD vendors occasionally change the properties utilized by the plugin for HMD detection. If you encounter issues, such as incorrect tracker offsets, it is recommended to explicitly specify the tracking hardware.

> **Caution**
>
> Due to highly experimental nature of the HMD auto-detection feature, the HTC VIVE Focus Vision, HTC VIVE Focus 3, and HTC VIVE XR Elite cannot be distinguished from each other in the current iteration. However, since the tracker devices and offsets for all these headsets are the same, this should not

affect the performance or any functionality. The order in which the HMD is detected can be specified through the HMD-tracker setting `ViveHMDDetectionPriority`.

TrackingHardware

Specifies the type of tracking hardware to use. If set to `None`, the plugin attempts at HMD auto-detection to automatically specify a compatible tracking hardware. If set to `Custom`, any desired location and rotation can be specified.

At the moment the following hardware are supported:

- Quest 2 Controllers
- Quest 3 Controllers
- Quest Pro Controllers
- VIVE Focus 3 Wrist Trackers
- VIVE Trackers



Caution

HMD auto-detection is currently an experimental feature and may fail because HMD vendors occasionally change the properties utilized by the plugin for HMD detection. If you encounter issues, such as incorrect tracker offsets, it is recommended to explicitly specify the tracking hardware.

Caution

> Due to highly experimental nature of the HMD auto-detection feature, the HTC VIVE Focus Vision, HTC VIVE Focus 3, and HTC VIVE XR Elite cannot be distinguished from each other in the current iteration. However, since the tracker devices and offsets for all these headsets are the same, this should not affect the performance or any functionality. The order in which the HMD is detected can be specified through the HMD-tracker setting `ViveHMDDetectionPriority`.

TrackingHardwareLocationOffsetLeftHand

Sets a custom location offset for left hand's wrist-tracking hardware.

> **Note**
>
> This setting is visible and valid only if TrackingHardware is set to `Custom`.

TrackingHardwareLocationOffsetRightHand

Sets a custom location offset for right hand's wrist-tracking hardware.

> **Note**
>
> This setting is visible and valid only if TrackingHardware is set to `Custom`.

TrackingHardwareRotationOffsetLeftHand

Sets a custom rotation offset for left hand's wrist-tracking hardware.

> **Note**

> This setting is visible and valid only if TrackingHardware is set to `Custom`.

TrackingHardwareRotationOffsetRightHand

Sets a custom rotation offset for right hand's wrist-tracking hardware.

> **Note**
>
> This setting is visible and valid only if TrackingHardware is set to `Custom`.

LeftHandMotionSource

Determines the motion source for the left hand. For Oculus HMDs, this is usually `Left`, and for VIVE HMDs using VIVE Wrist Trackers, VIVE Business Streaming, and SteamVR, it's typically `LeftFoot`. For the `OpenXRVive` plugin on Android Standalone Mode using the VIVE Wrist Trackers, this typically is `LeftWristTracker`.

> **Note**
>
> For VIVE devices using SteamVR, the motion source hardware for the left hand can be specified by the user through the SteamVR app.

RightHandMotionSource

Determines the motion source for the right hand. For Oculus HMDs, this is usually `Right`, and for VIVE HMDs using VIVE Wrist Trackers, VIVE Business Streaming, and SteamVR, it's typically `RightFoot`. For the `OpenXRVive` plugin on Android Standalone Mode using the VIVE Wrist Trackers, this typically is `RightWristTracker`.

> **Note**

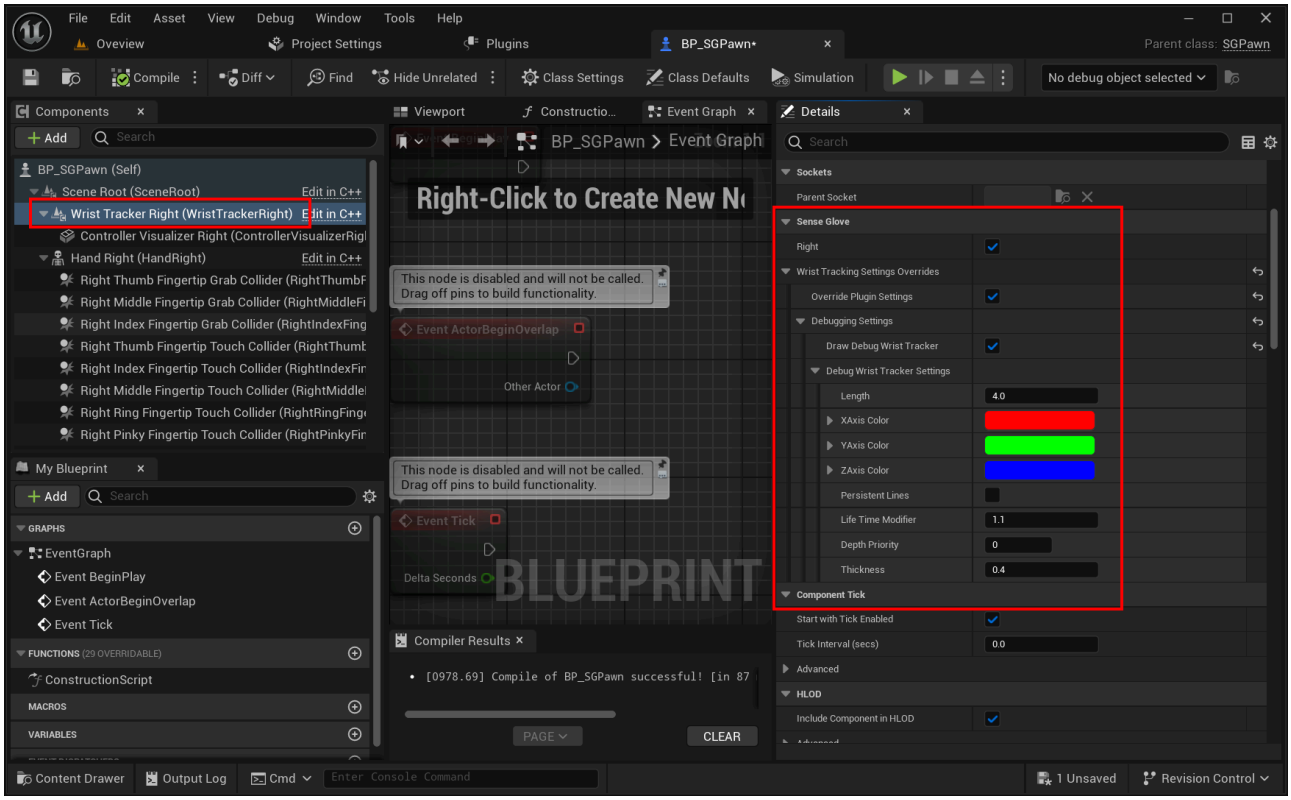
> For VIVE devices using SteamVR, the motion source hardware for the right hand can be specified by the user through the SteamVR app.

DebuggingSettings

Provides debugging options for visually debugging the wrist tracker.

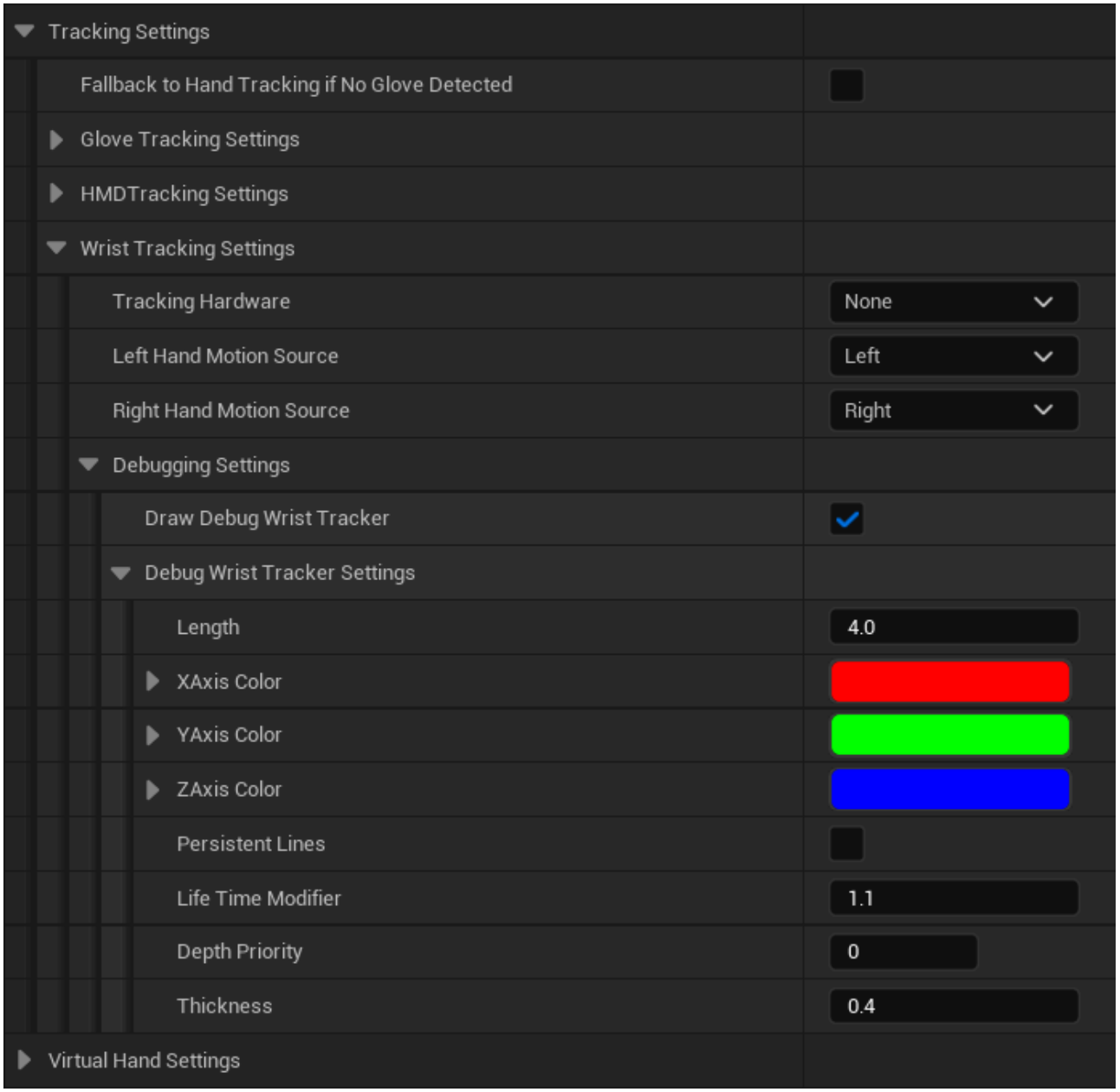
Overriding the Wrist-tracking Settings from the Wrist Tracker Component

It's possible to override some of the wrist tracker settings through the details panel of any specific Wrist Tracker Component. When overriden by enabling the `SenseGlove > Wrist Tracking Settings Override > Override Plugin Settings` option inside the details panel, these settings take precedence over the plugin's global settings.


File Edit Asset View Debug Window Tools Help
Oveview
Project Settings
Plugins
BP_SGPawn•
Parent class: SGPawn
Compile
Diff
Find
Hide Unrelated
Class Settings
Class Defaults
Simulation
No debug object selected
Components
Add
Search
BP_SGPawn (Self)
Scene Root (SceneRoot) Edit in C++
Wrist Tracker Right (WristTrackerRight) Edit in C++
Controller Visualizer Right (ControllerVisualizerRigl
Hand Right (HandRight) Edit in C++
Right Thumb Fingertip Grab Collider (RightThumbF
Right Middle Fingertip Grab Collider (RightMiddleFi
Right Index Fingertip Grab Collider (RightIndexFing
Right Thumb Fingertip Touch Collider (RightThumt
Right Index Fingertip Touch Collider (RightIndexFir
Right Middle Fingertip Touch Collider (RightMiddle
Right Ring Fingertip Touch Collider (RightRingFing
Right Pinky Fingertip Touch Collider (RightPinkyFir
My Blueprint
Add
Search
GRAPHS
EventGraph
Event BeginPlay
Event ActorBeginOverlap
Event Tick
FUNCTIONS (29 OVERRIDABLE)
ConstructionScript
MACROS
VARIABLES
Viewport
Constructio...
Event Graph
BP_SGPawn > Event Graph
Right-Click to Create New N
This node is disabled and will not be called.
Drag off pins to build functionality.
Event ActorBeginOverlap
Other Actor
This node is disabled and will not be called.
Drag off pins to build functionality.
Event Tick
Delta Seconds
BLUEPRINT
Compiler Results
[0978.69] Compile of BP_SGPawn successful! [in 87
PAGE
CLEAR
Details
Search
Sockets
Parent Socket
Sense Glove
Right
Wrist Tracking Settings Overrides
Override Plugin Settings
Debugging Settings
Draw Debug Wrist Tracker
Debug Wrist Tracker Settings
Length 4.0
XAxis Color
YAxis Color
ZAxis Color
Persistent Lines
Life Time Modifier 1.1
Depth Priority 0
Thickness 0.4
Component Tick
Start with Tick Enabled
Tick Interval (secs) 0.0
Advanced
HLOD
Include Component in HLOD
Content Drawer
Output Log
Cmd
Enter Console Command
1 Unsaved
Revision Control

The Wrist-tracking Debugging Settings

Provides debugging options for visually debugging the wrist tracker.



bDrawDebugWristTracker

If enabled, visualizes the debug wrist trackers where possible.

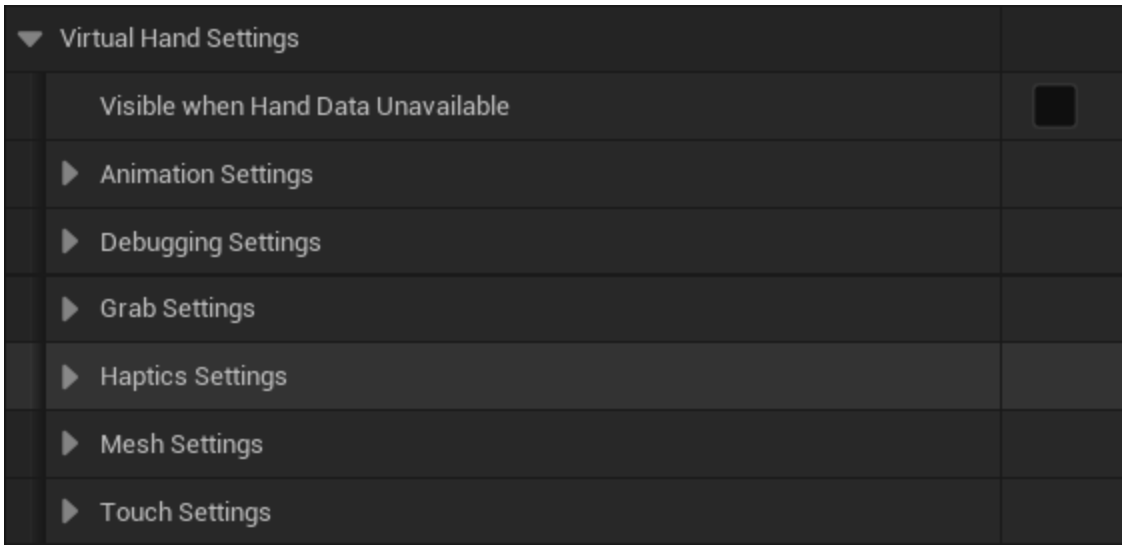
DebugWristTrackerSettings

Visible and valid only if `bDrawDebugGizmo` is enabled.

The Virtual Hand Settings

The Virtual Hand Settings are utilized by various SenseGlove modules such as `Debug`, `Editor`, `Tracking`, and the main module. These settings are divided into several subsections, each focusing on a specific aspect of the virtual hand functionality. Together with the settings provided directly in this section, they offer comprehensive control over any system or component that utilizes the virtual hand. The subsections are as follows:

- The Animation Settings
- The Debugging Settings
- The Grab Settings
- The Haptics Settings
- The Mesh Settings
- The Touch Settings



bVisibleWhenHandDataUnavailable

Used by the Virtual Hand Component to determine its visibility when no hand data, either from a SenseGlove or hand-tracking, is available. If enabled, the virtual hand mesh remains visible even when no data is available. By default, this setting is

disabled, providing users of the simulation with a clear indicator that no hand data is currently available.

Animation Settings

Controls how the virtual hand model is animated by the animation system.

Debugging Settings

Primarily used for visually debugging low-level hand data. When enabled, the Virtual Hand Component visualizes a debug virtual hand by drawing all individual hand joints.

Grab Settings

Utilized by the SenseGlove Sockets Editor to automatically generate the hand sockets required by the Grab system to function.

The `SGPawn` also utilizes these settings to set up the grab colliders on the virtual hand components.

Haptics Settings

Utilized by the haptics system.

Mesh Settings

Utilized by the SenseGlove `Tracking` module to account for the current virtual hand mesh when generating hand pose data, resulting in more accurate glove or hand data representation and also smoother animations.

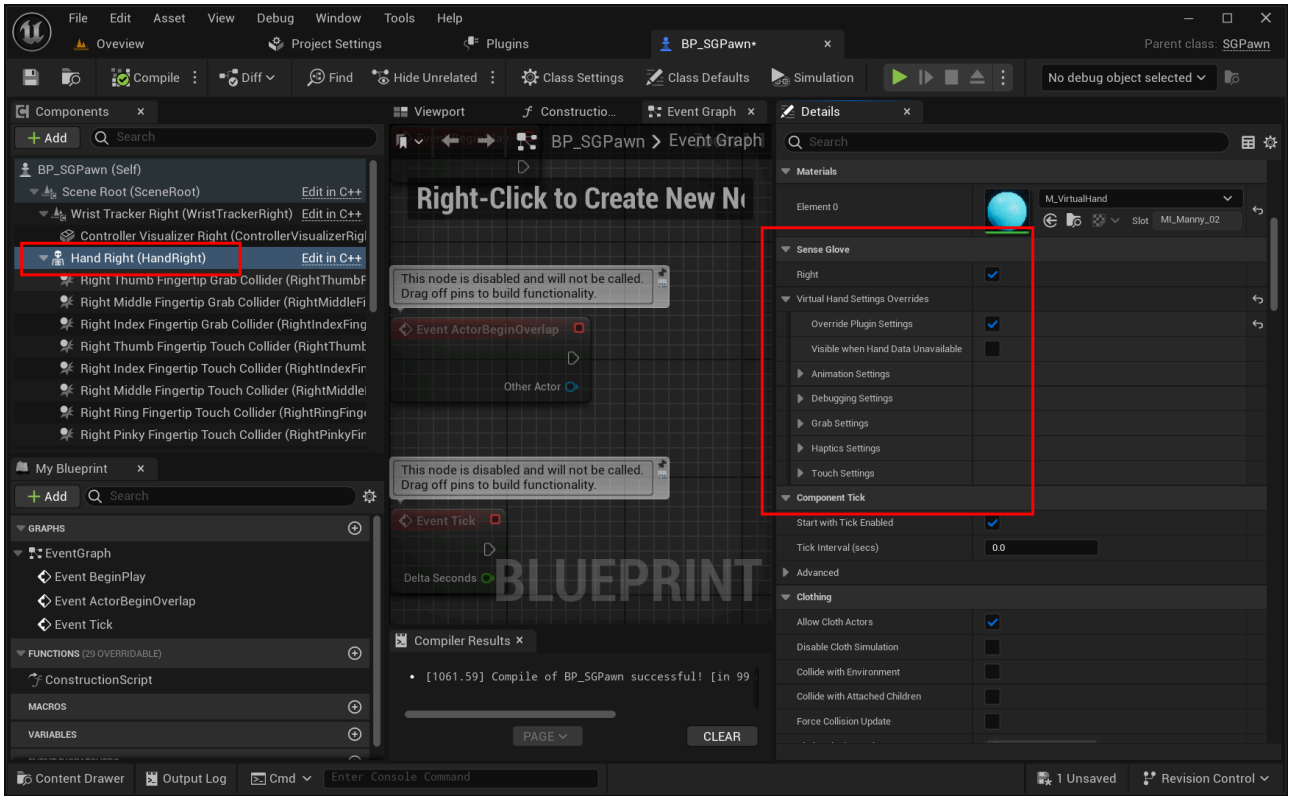
Touch Settings

Utilized by the SenseGlove Sockets Editor to automatically generate the hand sockets required by the Touch system to function.

The `SGPawn` also utilizes these settings to set up the touch colliders on the virtual hand components.

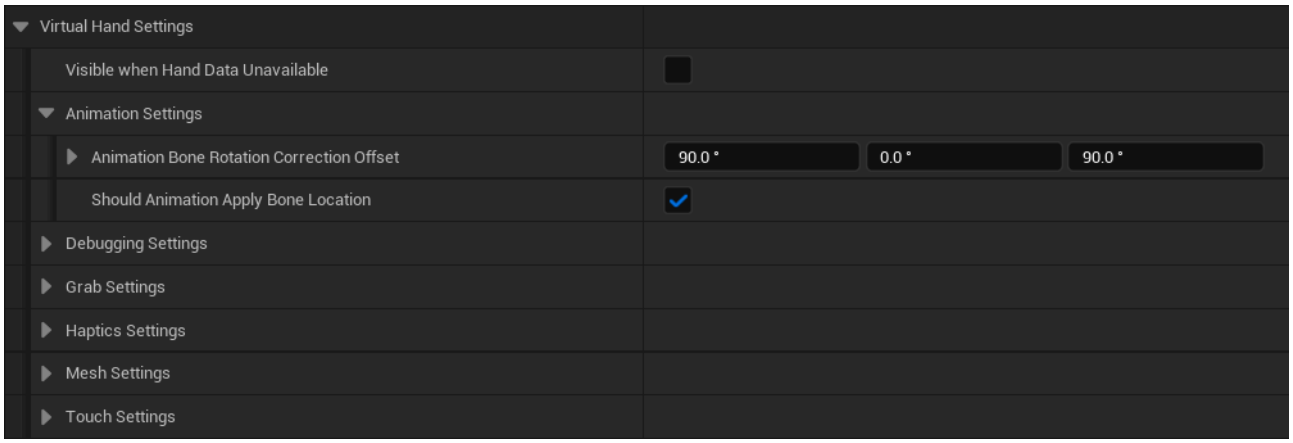
Overriding the Virtual Hand Settings from the Wrist Tracker Component

It's possible to override some of the virtual hand settings through the details panel of any specific Virtual Hand Component. When overriden by enabling the `SenseGlove > Virtual Hand Settings Override > Override Plugin Settings` option inside the details panel, these settings take precedence over the plugin's global settings.


File Edit Asset View Debug Window Tools Help
Oveview
Project Settings
Plugins
BP_SGPawn•
Parent class: SGPawn
Compile
Diff
Find
Hide Unrelated
Class Settings
Class Defaults
Simulation
No debug object selected
Components
Add
Search
BP_SGPawn (Self)
Scene Root (SceneRoot) Edit in C++
Wrist Tracker Right (WristTrackerRight) Edit in C++
Controller Visualizer Right (ControllerVisualizerRigl
Hand Right (HandRight) Edit in C++
Right Thumb Fingertip Grab Collider (RightThumbF
Right Middle Fingertip Grab Collider (RightMiddleFi
Right Index Fingertip Grab Collider (RightIndexFing
Right Thumb Fingertip Touch Collider (RightThumt
Right Index Fingertip Touch Collider (RightIndexFir
Right Middle Fingertip Touch Collider (RightMiddle
Right Ring Fingertip Touch Collider (RightRingFing
Right Pinky Fingertip Touch Collider (RightPinkyFir
My Blueprint
Add
Search
GRAPHS
EventGraph
Event BeginPlay
Event ActorBeginOverlap
Event Tick
FUNCTIONS (29 OVERRIDABLE)
ConstructionScript
MACROS
VARIABLES
Viewport
Constructio...
Event Graph
BP_SGPawn > Event Graph
Right-Click to Create New N
This node is disabled and will not be called.
Drag off pins to build functionality.
Event ActorBeginOverlap
Other Actor
This node is disabled and will not be called.
Drag off pins to build functionality.
Event Tick
Delta Seconds
BLUEPRINT
Compiler Results
[1061.59] Compile of BP_SGPawn successful! [in 99
PAGE
CLEAR
Details
Search
Materials
Element 0
M_VirtualHand
Slot
MI_Manny_02
Sense Glove
Right
Virtual Hand Settings Overrides
Override Plugin Settings
Visible when Hand Data Unavailable
Animation Settings
Debugging Settings
Grab Settings
Haptics Settings
Touch Settings
Component Tick
Start with Tick Enabled
Tick Interval (secs)
0.0
Advanced
Clothing
Allow Cloth Actors
Disable Cloth Simulation
Collide with Environment
Collide with Attached Children
Force Collision Update
Content Drawer
Output Log
Cmd
Enter Console Command
1 Unsaved
Revision Control

The Virtual Hand Animation Settings

Controls how the virtual hand model is animated by the animation system.



AnimationBoneRotationCorrectionOffset

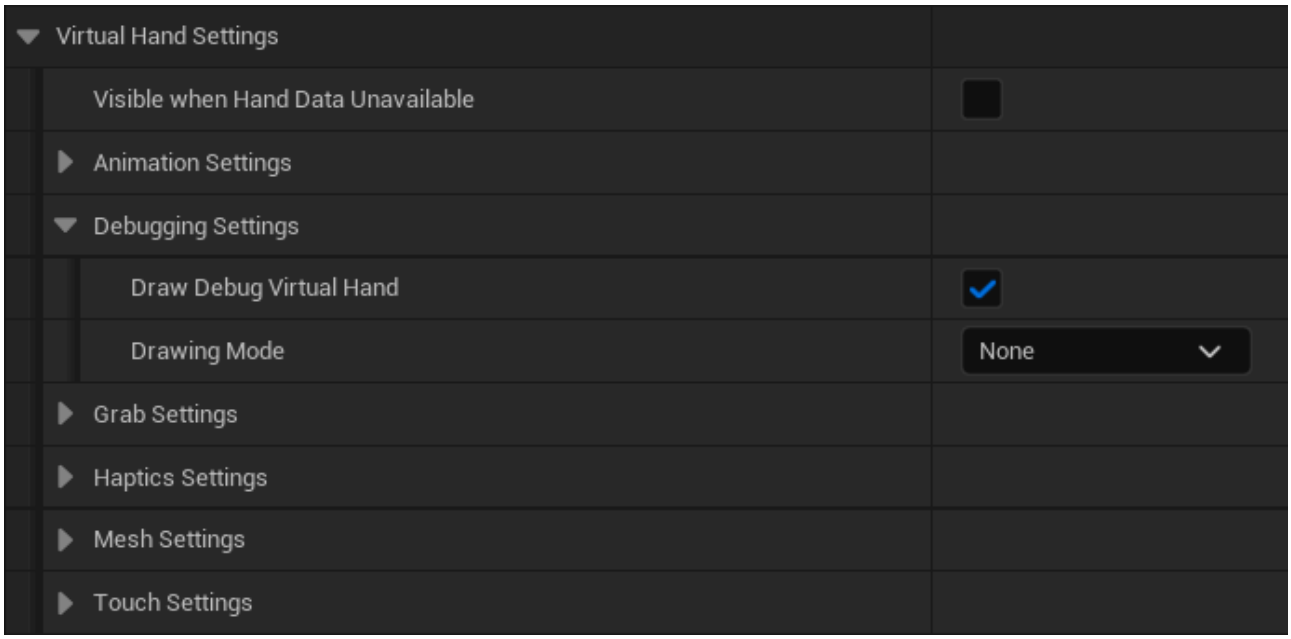
Specifies the offset to apply to each bone's rotation when translating hand pose data to the virtual hand bones. This is useful if the virtual hand mesh was imported with an initial rotation. For example, the virtual hand model shipped with Unreal Engine's VRTemplate typically has an initial `90.0f` degrees rotation on the `Yaw` axis. By default, this option has been set up with the Unreal Engine's VRTemplate virtual hand model in mind.

bShouldAnimationApplyBoneLocation

When enabled, the animation system applies the joint locations to the current virtual hand mesh bones in addition to the joint rotation. Otherwise, only the joint rotations are applied, and joint locations are ignored, leaving the bone locations untouched on the virtual hand mesh when animating it. Enabling this option typically improves the virtual hand animation. By default, this option is enabled.

The Virtual Hand Debugging Settings

Primarily used for visually debugging low-level hand data. When enabled, the Virtual Hand Component visualizes a debug virtual hand by drawing all individual hand joints.



bDrawDebugVirtualHand

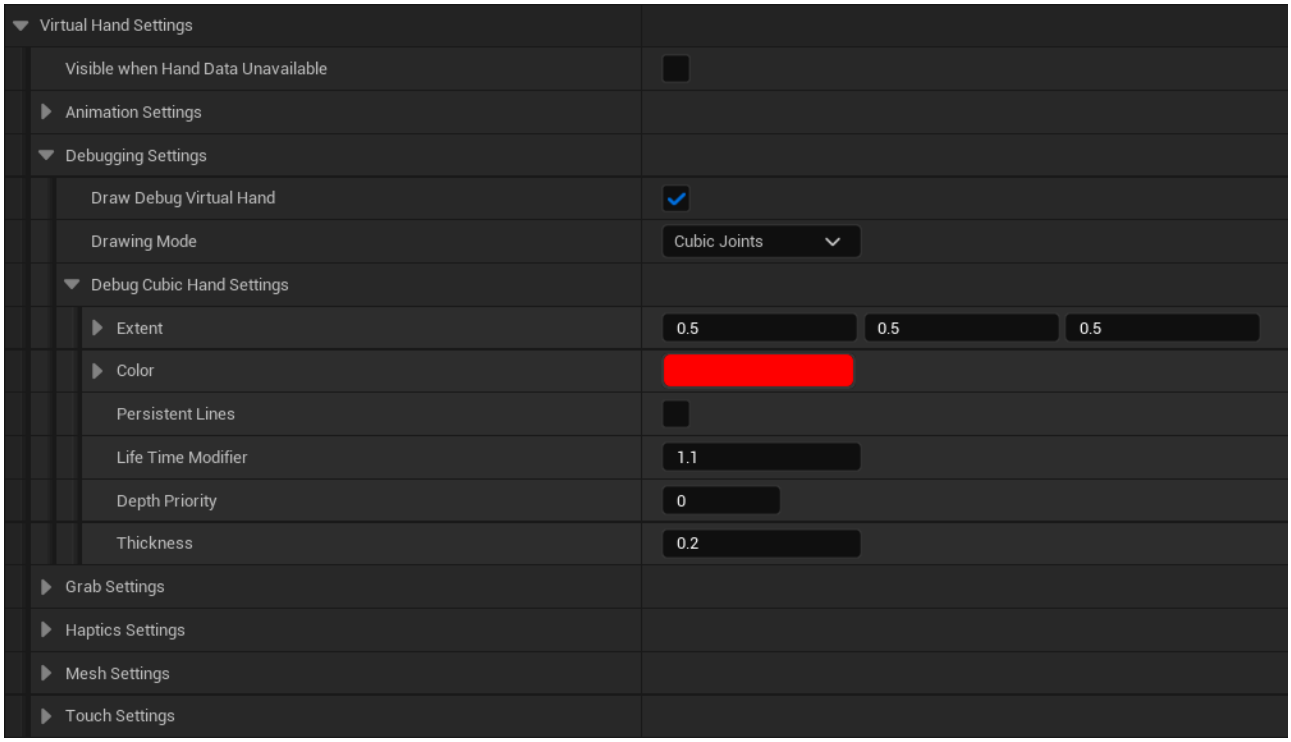
If enabled, visualizes the debug virtual hand where possible.

DrawingMode

Determines the virtual hand drawing mode. If set to `CubicJoints`, for every joint a debug cube will be drawn. If set to `GizmoJoints`, for every joint a debug gizmo will be drawn.

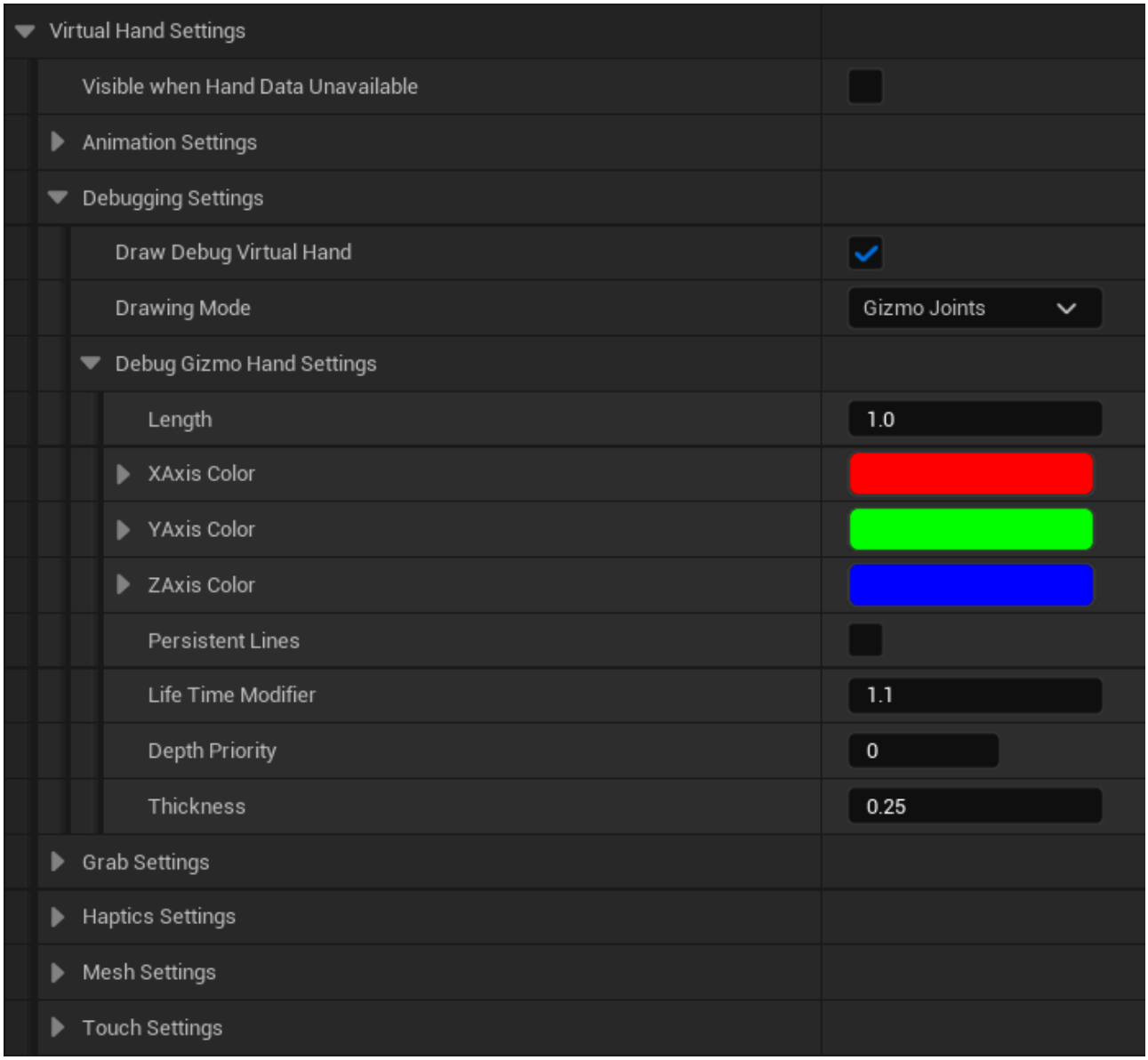
DebugCubicHandSettings

Visible and valid only if `bDrawDebugVirtualHand` is enabled and `DrawingMode` has been set to `ESGDebugVirtualHandDrawingMode::CubicJoints`.



DebugGizmoHandSettings

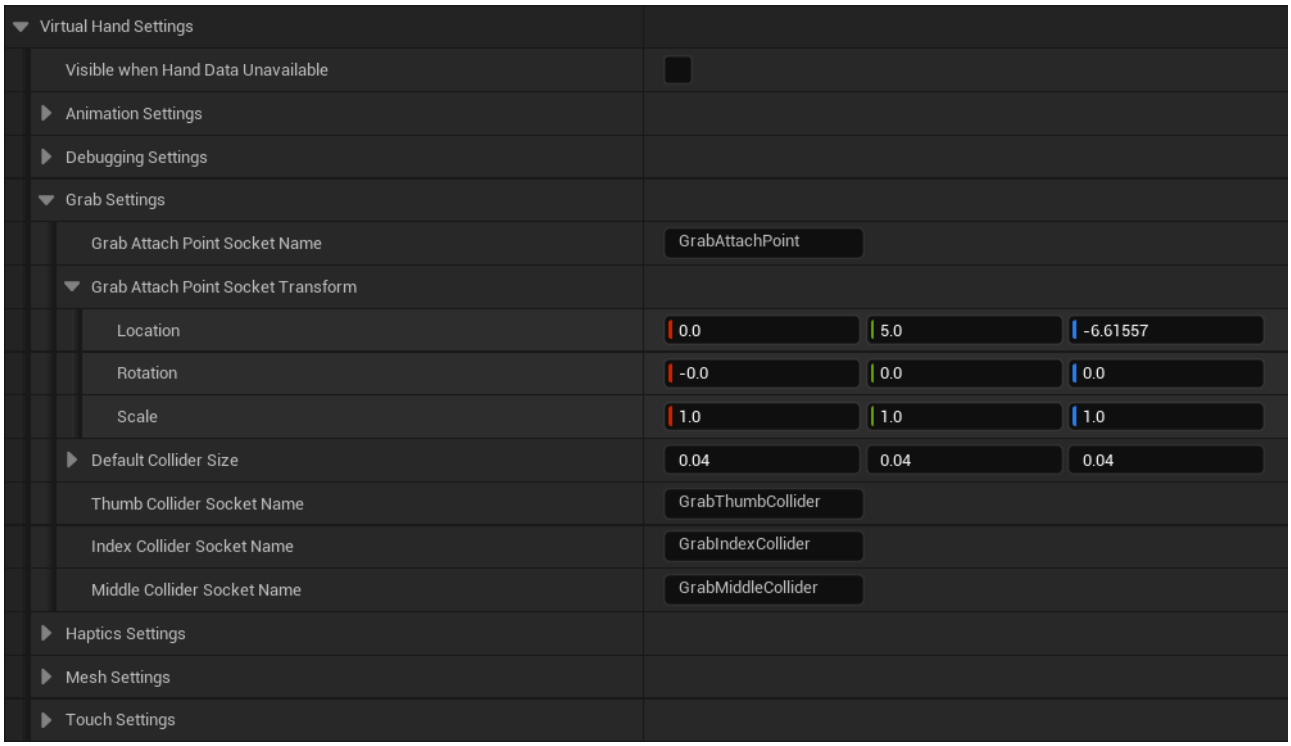
Visible and valid only if `bDrawDebugVirtualHand` is enabled and `DrawingMode` has been set to `ESGDebugVirtualHandDrawingMode::GizmoJoints`.


Virtual Hand Settings
Visible when Hand Data Unavailable
Animation Settings
Debugging Settings
Draw Debug Virtual Hand
Drawing Mode
Gizmo Joints
Debug Gizmo Hand Settings
Length
1.0
XAxis Color
YAxis Color
ZAxis Color
Persistent Lines
Life Time Modifier
1.1
Depth Priority
0
Thickness
0.25
Grab Settings
Haptics Settings
Mesh Settings
Touch Settings

The Virtual Hand Grab Settings

Utilized by the SenseGlove Sockets Editor to automatically generate the hand sockets required by the Grab system to function.

The `SGPawn` also utilizes these settings to set up the grab colliders on the virtual hand components.



GrabAttachPointSocketName

The default socket name for the grab attach point, usually located at the palm of the hand.

GrabAttachPointSocketTransform

The default socket transform (location, rotation, scale) for the grab attach point, usually located at the palm of the hand.

DefaultColliderSize

The default collider size for the fingers' grab colliders.

ThumbColliderSocketName

The default socket name for the thumb finger's grab collider, usually located at the tip of the thumb finger.

IndexColliderSocketName

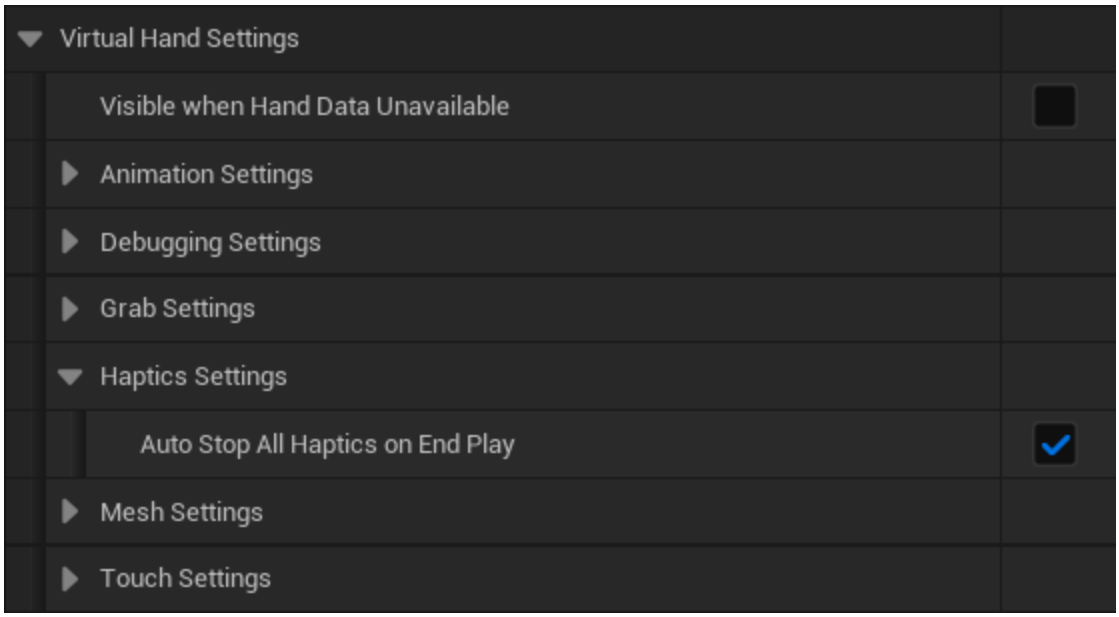
The default socket name for the index finger's grab collider, usually located at the tip of the index finger.

MiddleColliderSocketName

The default socket name for the middle finger's grab collider, usually located at the tip of the middle finger.

The Virtual Hand Haptics Settings

Utilized by the haptics system.

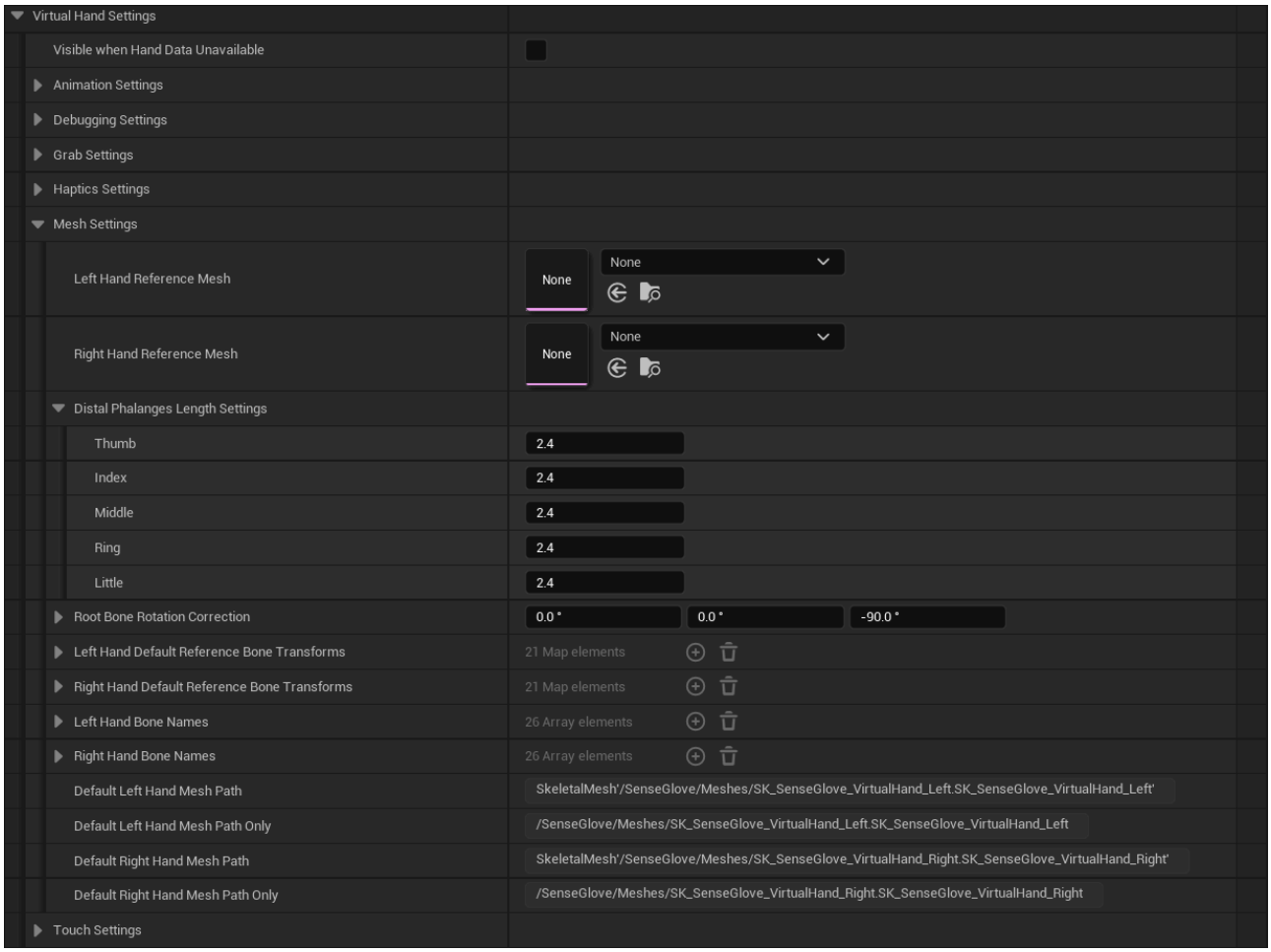


bAutoStopAllHapticsOnEndPlay

Forces all haptics to stop automatically on the `EndPlay` event. This is useful for situations where the simulation has ended, but ongoing haptic feedback might remain active on the glove indefinitely. By default, this setting is enabled.

The Virtual Hand Mesh Settings

Utilized by the SenseGlove `Tracking` module to account for the current virtual hand mesh when generating hand pose data, resulting in more accurate glove or hand data representation and also smoother animations.



LeftHandReferenceMesh

The virtual hand model for the left hand is to be used by the SenseGlove `Tracking` module to generate all the `26` joint data present in the `FXRMotionControllerData`. The main reason the `Tracking` module requires a virtual hand mesh as a reference is the SenseGlove Hand Pose format only provides `15` joints. So, the remaining joint data for `FXRMotionControllerData` are calculated from a virtual hand mesh

compatible with the Epic rig and also the values specified by `DistalPhalangesLengthSettings`. Furthermore, when calculating the existing joints data, their current locations and rotations are taken into account in calculating the resulting `FXRMotionControllerData`.

By default, no virtual hand mesh is set.

> **Caution**
>
> If no virtual hand mesh is set, the `Tracking` module will fall back to hard-coded values extracted from the standard virtual hand model shipped by Unreal Engine VRTemplate. This may result in distorted hand mesh while animating a hand in case a different hand mesh other than the default Epic virtual hand mesh is being set on the virtual hand components.

RightHandReferenceMesh

The virtual hand model for the right hand is to be used by the SenseGlove `Tracking` module to generate all the `26` joint data present in the `FXRMotionControllerData`. The main reason the `Tracking` module requires a virtual hand mesh as a reference is the SenseGlove Hand Pose format only provides `15` joints. So, the remaining joint data for `FXRMotionControllerData` are calculated from a virtual hand mesh compatible with the Epic rig and also the values specified by `DistalPhalangesLengthSettings`. Furthermore, when calculating the existing joints data, their current locations and rotations are taken into account in calculating the resulting `FXRMotionControllerData`.

By default, no virtual hand mesh is set.

> **Caution**
>
> If no virtual hand mesh is set, the `Tracking` module will fall back to hard-coded values extracted from the standard virtual hand model shipped by Unreal Engine VRTemplate. This may result in distorted hand mesh while animating a hand in case a different hand mesh other than the default Epic virtual hand mesh is being set on the virtual hand components.

DistalPhalangesLengthSettings

The length of distal phalanges that cannot be retrieved from any virtual hand mesh compliant with the Epic standard rig. Also, the SenseGlove Hand Pose format does not provide these. This is used by SenseGlove `Tracking` module to calculate an `FXRMotionControllerData` the all `26` joints. The values you specify here depend on the shape of the virtual hand mesh and the defaults are approximated for the virtual hand model shipped with the Unreal Engine VRTemplate.

RootBoneRotationCorrection

Used mostly by the SenseGlove `Tracking` module and `SGPawn` to offset for any initial rotation during the virtual hand mesh import process. This is the case for example with the virtual hand model shipped with Unreal Engine's VRTemplate, which typically has an initial `-90.0f` degrees rotation on the `Yaw` axis. By default, this option has been set up with the Unreal Engine's VRTemplate virtual hand model in mind.

LeftHandDefaultReferenceBoneTransforms

Read-only and for internal use only.

RightHandDefaultReferenceBoneTransforms

Read-only and for internal use only.

LeftHandBoneNames

Read-only and for internal use only.

RightHandBoneNames

Read-only and for internal use only.

DefaultLeftHandMeshPath

Read-only and for internal use only.

DefaultLeftHandMeshPathOnly

Read-only and for internal use only.

DefaultRightHandMeshPath

Read-only and for internal use only.

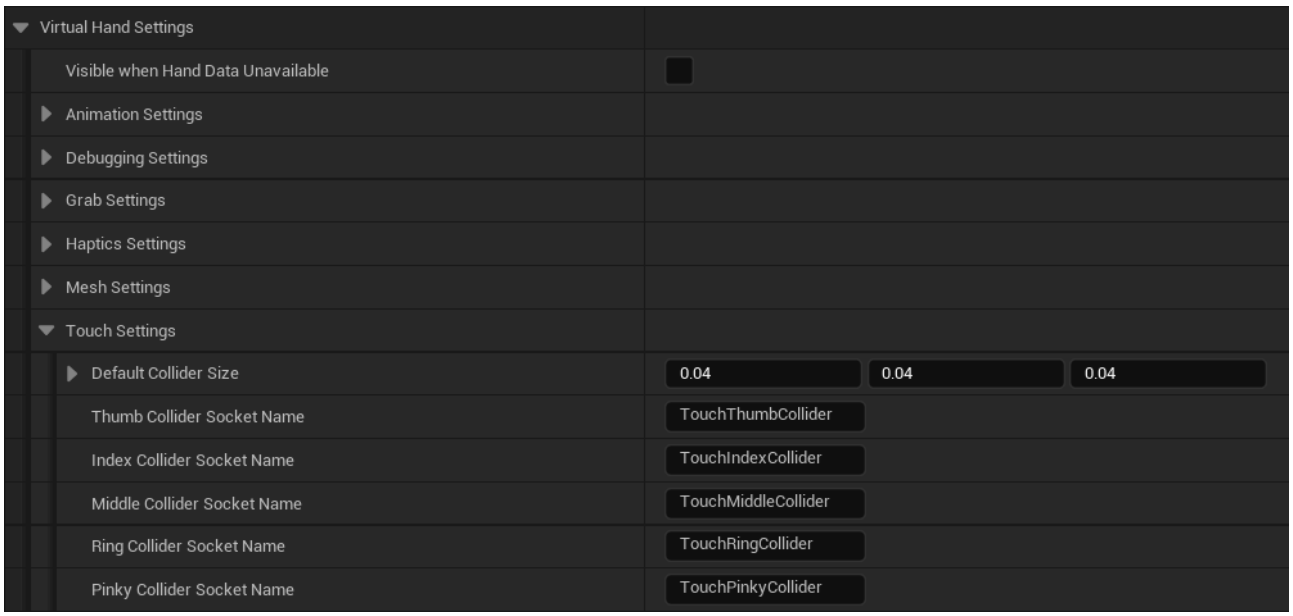
DefaultRightHandMeshPathOnly

Read-only and for internal use only.

The Virtual Hand Touch Settings

Utilized by the SenseGlove Sockets Editor to automatically generate the hand sockets required by the Touch system to function.

The `SGPawn` also utilizes these settings to set up the touch colliders on the virtual hand components.



DefaultColliderSize

The default collider size for the fingers' touch colliders.

ThumbColliderSocketName

The default socket name for the thumb finger's touch collider, usually located at the tip of the thumb finger.

IndexColliderSocketName

The default socket name for the index finger's touch collider, usually located at the tip of the index finger.

MiddleColliderSocketName

The default socket name for the middle finger's touch collider, usually located at the tip of the middle finger.

RingColliderSocketName

The default socket name for the ring finger's touch collider, usually located at the tip of the ring finger.

PinkyColliderSocketName

The default socket name for the pinky finger's touch collider, usually located at the tip of the pinky finger.

Overriding The Plugin Settings

The override system allows you to customize and override the global settings for the SenseGlove Unreal Engine Plugin through specific subcomponents where applicable. This feature enables more precise control over the behavior of individual components within your project.

The SenseGlove Virtual Hand Component

The Virtual Hand Component provides the ability to override certain aspects of the global plugin settings, allowing for tailored interactions and behaviors specific to virtual hands. For more details, refer to the Virtual Hand Settings section.

The SenseGlove Wrist Tracker Component

The Wrist Tracker Component enables overriding of specific global plugin settings, providing flexibility in wrist tracking configurations. For additional information, see the Wrist-tracker Settings section.

The SenseGlove Console Commands

The SenseGlove Unreal Engine Plugin offers a variety of utility console commands to enhance your development experience.

> **Important**
>
> To ensure the SenseGlove console commands are registered and recognized by Unreal Engine, set the default Game Instance class to `SGGameInstance` or a subclass of it. This can be done through: `Project Settings > Project > Maps & Modes > Game Instance > Game Instance Class`. Failing to do so will result in the error: `Command not recognized: SG_*` in the logs. For more details, refer to `SGGameInstance`.

SGGameUserSettings Console Commands

> **Caution**
>
> Before running any of the following console commands, ensure that the default Game User Settings class is set to `SGGameUserSettings` or a subclass of it. This can be configured via: `Project Settings > Engine > General Settings > Default Classes > Advanced > Game User Settings Class`. Failure to set this correctly will cause your simulation or editor to crash upon calling any of the following console commands. For more information, refer to `SGGameUserSettings`.

SG_GetEngineScalabilitySettings

This console command prints the current Engine Scalability Settings to the logs.

SG_SetEngineScalabilitySettings

This console command sets the Engine Scalability Settings for both the current game and the editor. It accepts a `Scalability` parameter with the following valid values:

- `Low`
- `Medium`
- `High`
- `Epic`
- `Cinematic`
- `Auto`

> **Note**
>
> The `Auto` option is used for benchmarking purposes. It will adjust the engine scalability settings to one of the other levels based on the benchmarking results.

Deploying to Android (Standalone)

Epic Games provides official documentation for setting up Unreal projects targeting Android:

- Setting Up Android SDK and NDK for Unreal
- Android Quick Start

Here are a few important notes to consider:

- Since SenseGlove provides native libraries built for Android, it's crucial to consult the Platform Support Matrix before deciding to deploy your project to Android.
- Currently, all third-party native libraries are built against Android NDK API Level `29`.
- On Meta Quest devices, building against Android SDK API Level `29` or `32` has been tested and is supported.
- A video tutorial on deploying to Oculus Quest devices and Android is also available.



Caution

> As of the `v2.1.0` release of the SenseGlove Unreal Engine Plugin, the `XR_EXT_hand_tracking` OpenXR extension is required for the plugin to function. Without this OpenXR extension, the plugin won't output any glove data. Unreal Engine provides `XR_EXT_hand_tracking` support through the `OpenXR` and `OpenXRHandTracking` plugins. However, for this to function on Android in standalone mode in conjunction with other plugins such as `Meta XR` or `ViveOpenXR` plugins, or hand-tracking as a fallback mechanism when no glove data is available, extra configuration steps and considerations are required.

Third-Party Tutorials

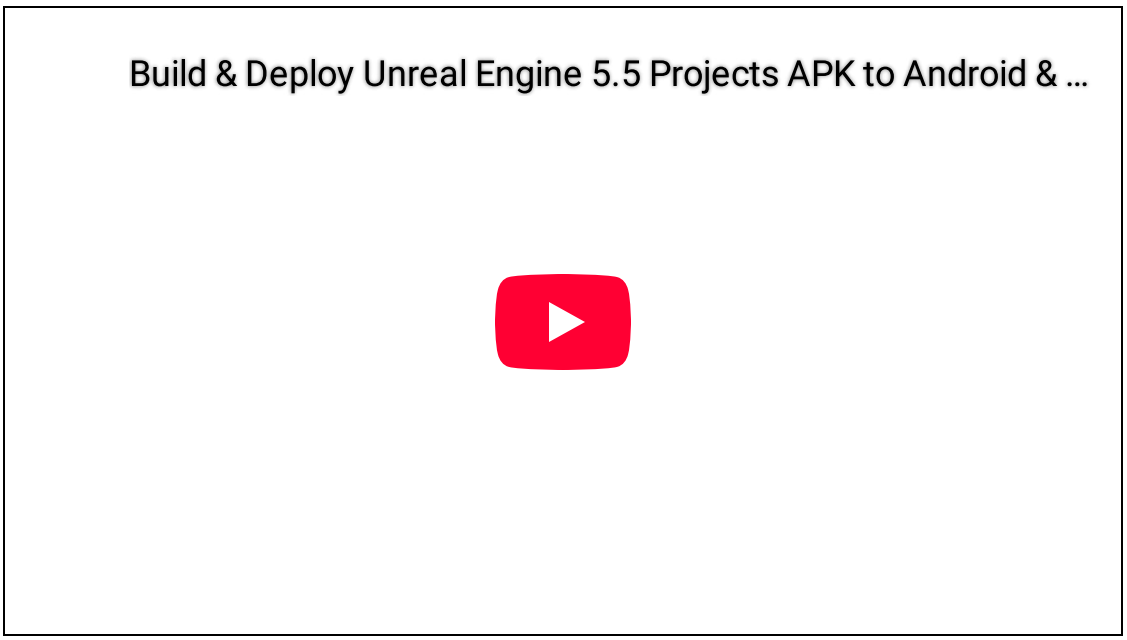
This Android Standalone Mode Deployment tutorials series covers how to build and deploy Unreal Engine `5.5` Projects APK to Android and Meta Quest 3S/3/Pro/2 in standalone mode. Furthermore, it covers the how-to on enabling OpenXR hand-tracking on Android with Meta XR (Quest 3S/3/Pro/2) and HTC VIVE OpenXR (Focus Vision/XR Elite/Focus 3) plugins.

Third-Party Tutorials: Android Standalone Mode Deployment

Build & Deploy Unreal Engine 5.5 Projects APK to Android & Meta Quest 3S/3/Pro/2 in Standalone Mode

This video will guide you through the process of building and deploying an Unreal Engine 5.5 project (or any version newer than `4.24`) to Android Standalone Mode for Meta Quest 2 and Quest 3 devices. The steps outlined here should also apply to other Android-based VR headsets.

It will show you where and how to download and install the necessary prerequisites, such as the Android SDK, NDK, Java Development Kit (JDK), and Microsoft Visual Studio. Next, it will configure both the development environment and Unreal Engine for a successful build. Finally, it will walk you through deploying your project to your VR headset and troubleshooting common errors to ensure a smooth experience.

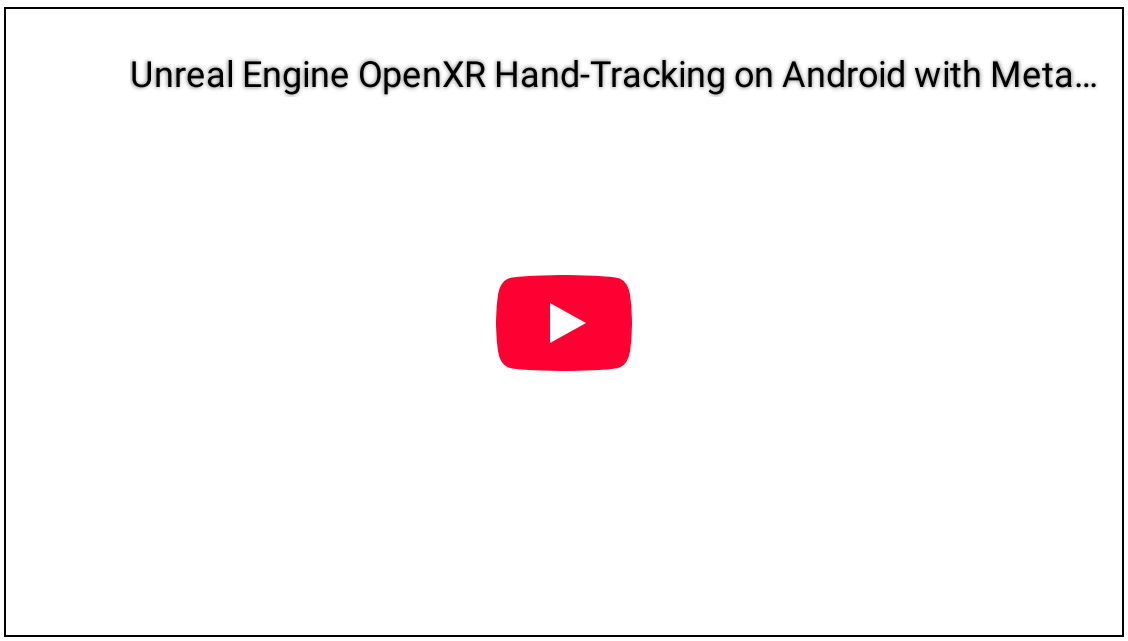


Unreal Engine OpenXR Hand-Tracking on Android with Meta XR (Quest 3S/3/Pro/2) and HTC VIVE OpenXR (Focus Vision/XR Elite/Focus 3) Plugins

The OpenXR hand-tracking provided by Epic won't work out of the box on Android when projects are deployed in standalone mode to HMDs such as Meta Quest or HTC VIVE devices.

In this tutorial you'll learn how to deploy your Unreal Engine projects to Android with functional hand-tracking on the following head-mounted displays:

- Meta Quest 3S
- Meta Quest 3
- Meta Quest Pro
- Meta Quest 2
- HTC VIVE Focus Vision
- HTC VIVE XR Elite
- HTC VIVE Focus 3



Upgrade Guide

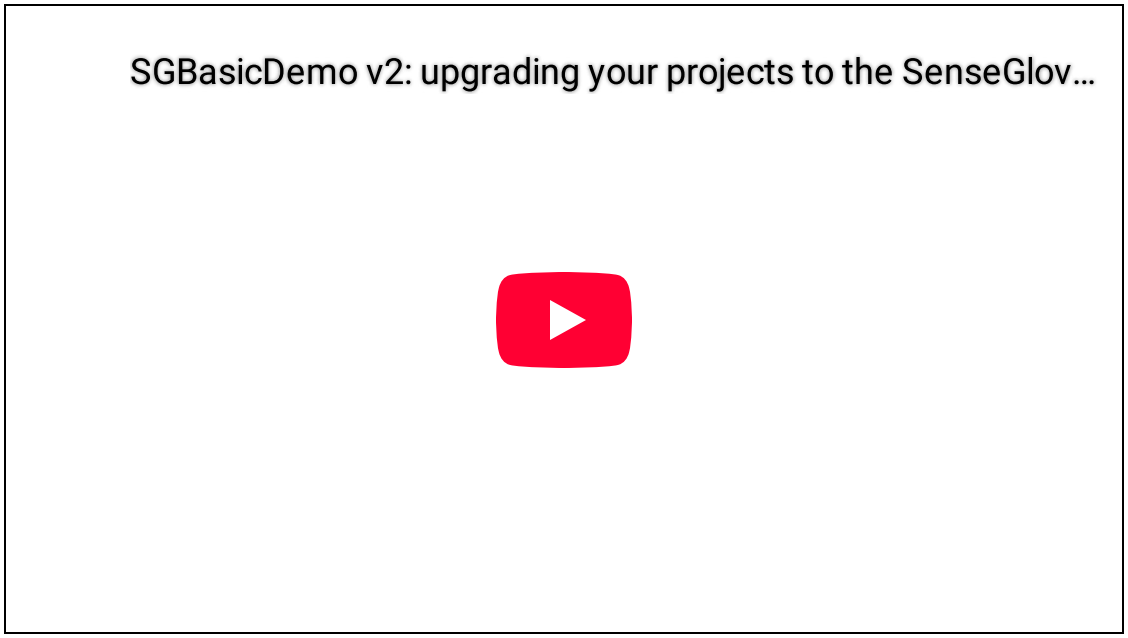
The transition from `v2.0.x` to `v2.1.x` introduces numerous changes, including several breaking changes. The effort required to upgrade your project will vary depending on its complexity and which features of the SenseGlove Unreal Engine Project you are using. However, if you are working with a simple Blueprint project like SGBasicDemo, the upgrade process is quite straightforward. We successfully upgraded SGBasicDemo to SGBasicDemo-OpenXR by following the procedure outlined below.

These are the notable changes that might affect your project:

- The SenseGlove Virtual Hand and Wrist Tracker components no longer rely on the SenseGlove Hand Pose data from the underlying SenseGlove API. Instead, they use `FXRMotionControllerData`.
- The virtual hand animation system has been revamped to use `FXRMotionControllerData` and no longer relies on SenseGlove Hand Angles. This means the virtual hand meshes are animated using world space transforms instead of parent bone space transforms.
- The Allbreaker virtual hand meshes have been removed and are no longer supported as they are incompatible with the new OpenXR tracking and animation system.

> **Caution**
>
> Please consult the changelog before upgrading your project to see if any change affects or breaks your current project.

> **Note**
>
> For upgrading older versions of the plugin to `v2.0.0`, a YouTube tutorial is available.



1. Remove the existing `Plugins/SenseGlove` folder from your project.
2. Obtain the latest `v2.1.x` version of the plugin either from the Epic Games Launcher or Microsoft Azure DevOps Repositories and place it in the `Plugins/SenseGlove` folder that you've just removed.
3. It might be best to clean up and remove the following folders from your project before generating the project files or attempting to open your project with the Unreal Editor. This might prevent a certain class of build issues:

```
- Binaries
- Intermediate
- Saved
```

4. Build your project using your favorite IDE if it's a C++ project, or open your project's `.uproject` file with the Unreal Editor and wait for the Editor to build the necessary binaries and open the project.
5. Remove the Allbreaker virtual hand meshes if you are using them, as they are no longer compatible with the new animation system.
6. Import and set up a set of compatible virtual hand meshes such as the VRTemplate virtual hand meshes, and configure the materials, rigid bodies, and

the SenseGlove Grab and Touch Sockets using the SenseGlove Sockets Editor.

7. Set up the SGPawn to use the new virtual hand meshes for the `HandLeft`, `HandRight`, `RealHandLeft`, and `RealHandRight` components.

8. Adjust the Virtual Hand Mesh Settings and ensure the `Left Hand Reference Mesh` and `Right Hand Reference Mesh` are set correctly.

9. Check and adjust the Virtual Hand Animation Settings as needed.

10. You might also want to set up the Wrist Tracking Hardware to use the new experimental HMD auto-detection feature. This allows the plugin to automatically configure the wrist tracking hardware at runtime, rather than limiting your builds to a specific HMD.

11. Set up the `SGGameInstance` and `SGGameUserSettings` if you want to use the new SenseGlove console commands or take advantage of the Engine Scalability Settings to achieve higher framerates in your project.

12. Additionally, the latest release introduces the ability to use hand-tracking as an alternative to SenseGlove hardware—albeit without haptic feedback—for rapid development and testing. It's also recommended to enable the Fallback to HandTracking if No Glove Detected feature to seamlessly switch to hand-tracking when a glove isn't connected.

13. If all steps have been followed correctly, your project should now be fully compatible with the new plugin release.

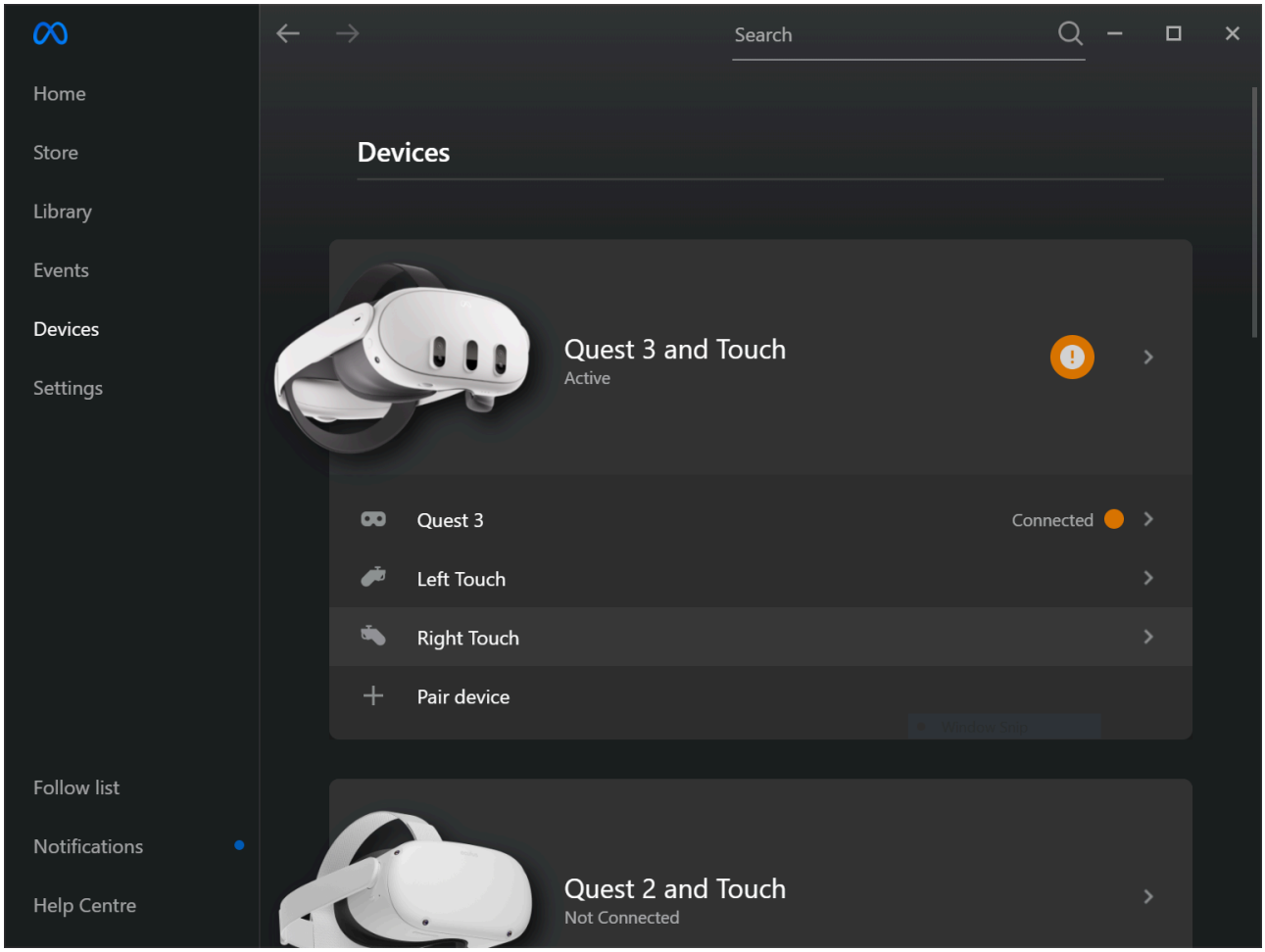
Optimizing Your Project for Higher FPS

Enhancing the performance and framerate of Unreal Engine VR applications, whether running standalone or streaming from a PC, can sometimes be challenging depending on the nature of your project. This guide will walk you through generic strategies that can significantly boost your project's performance and framerate with minimal effort.

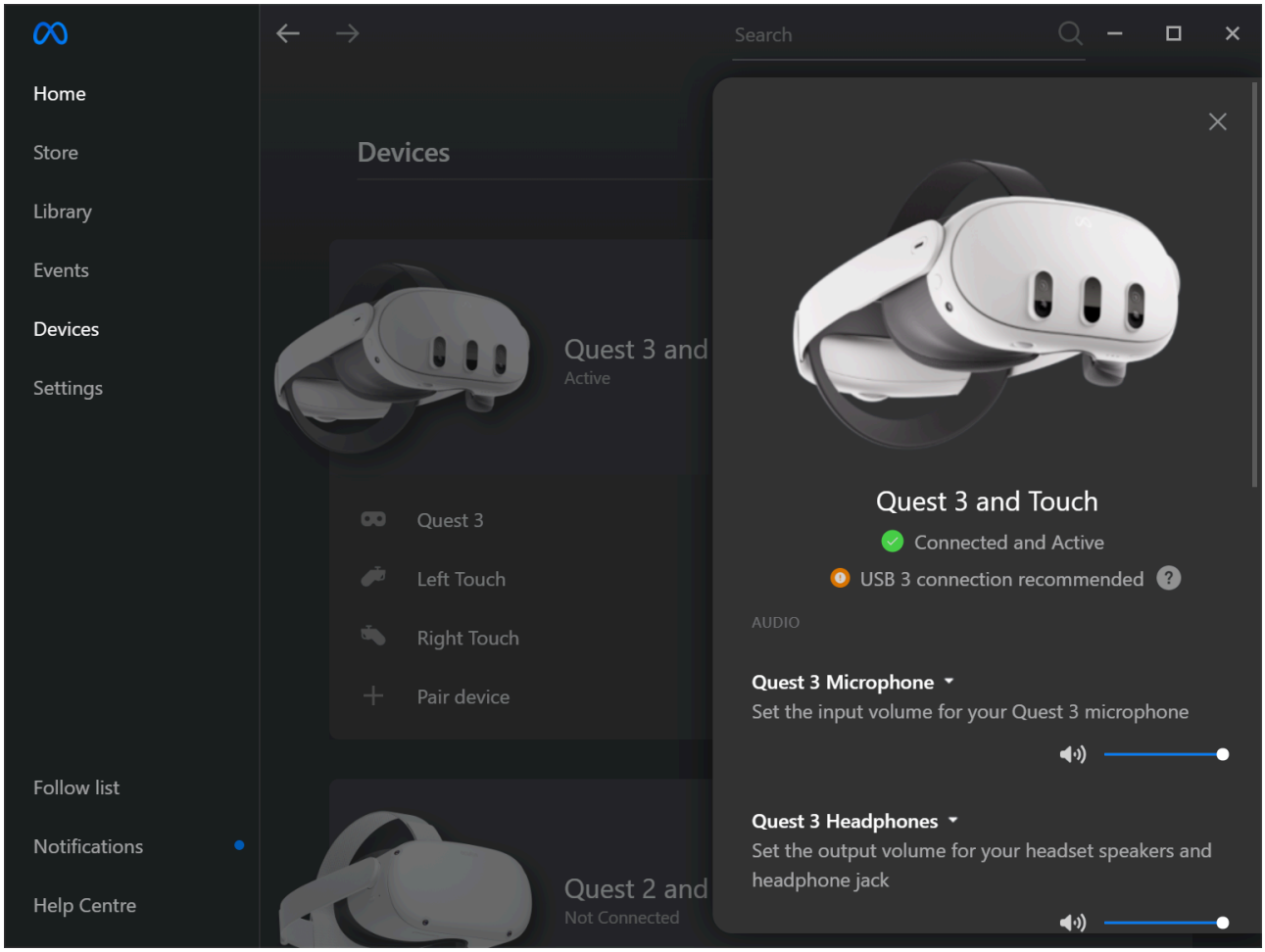
Meta Quest Link Advanced Graphics Preferences

When streaming from a PC to Meta Quest devices, the default refresh rate is set at `72 Hz`. However, you can increase this to `120 Hz`, which not only enhances the refresh rate but also reduces the rendering resolution, potentially improving performance. Follow these steps to make the adjustment:

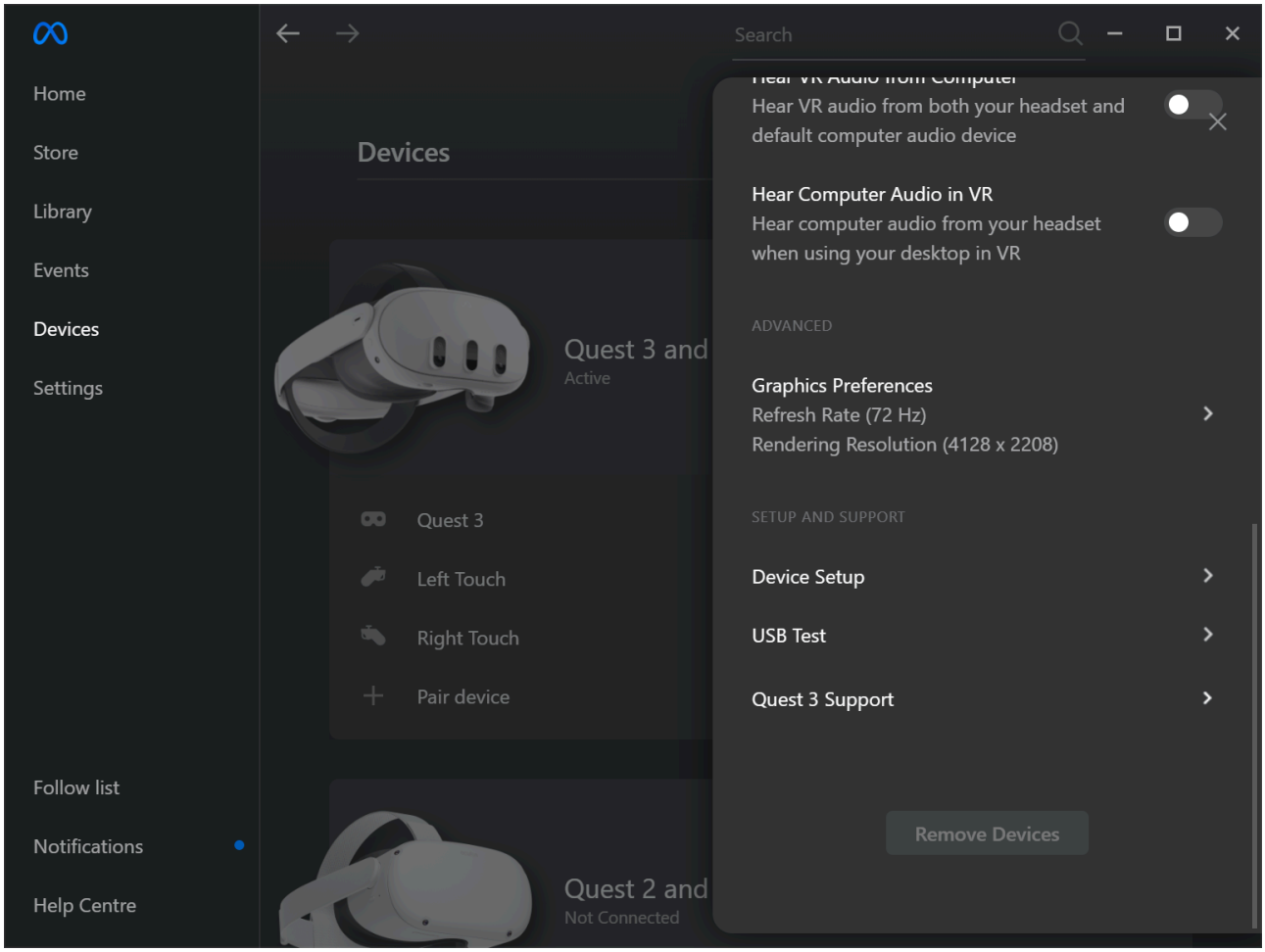
1. Open the Meta Quest Link app and navigate to the `Devices` tab.



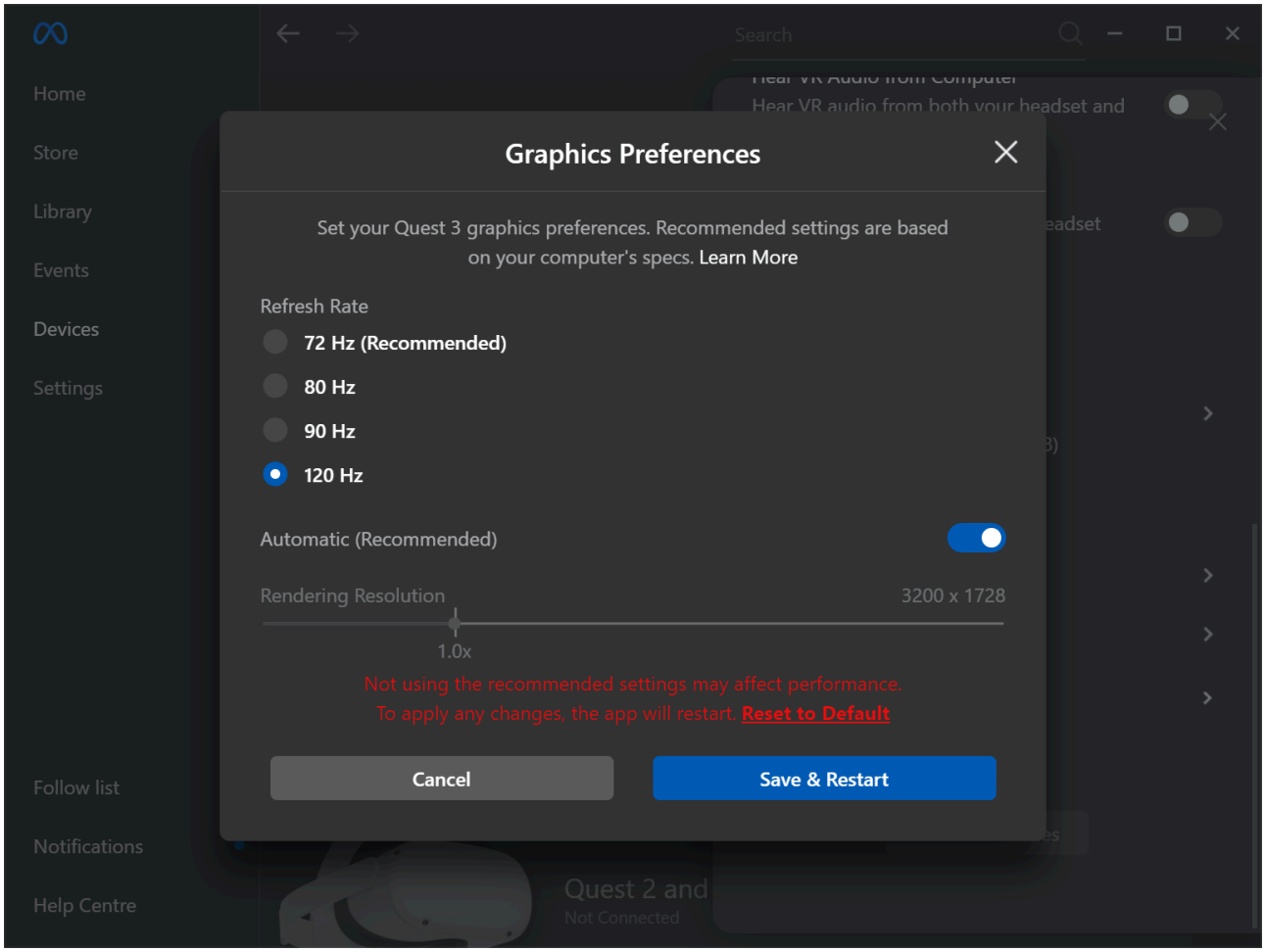
2. Choose the device for which you would like to tweak the refresh rate.



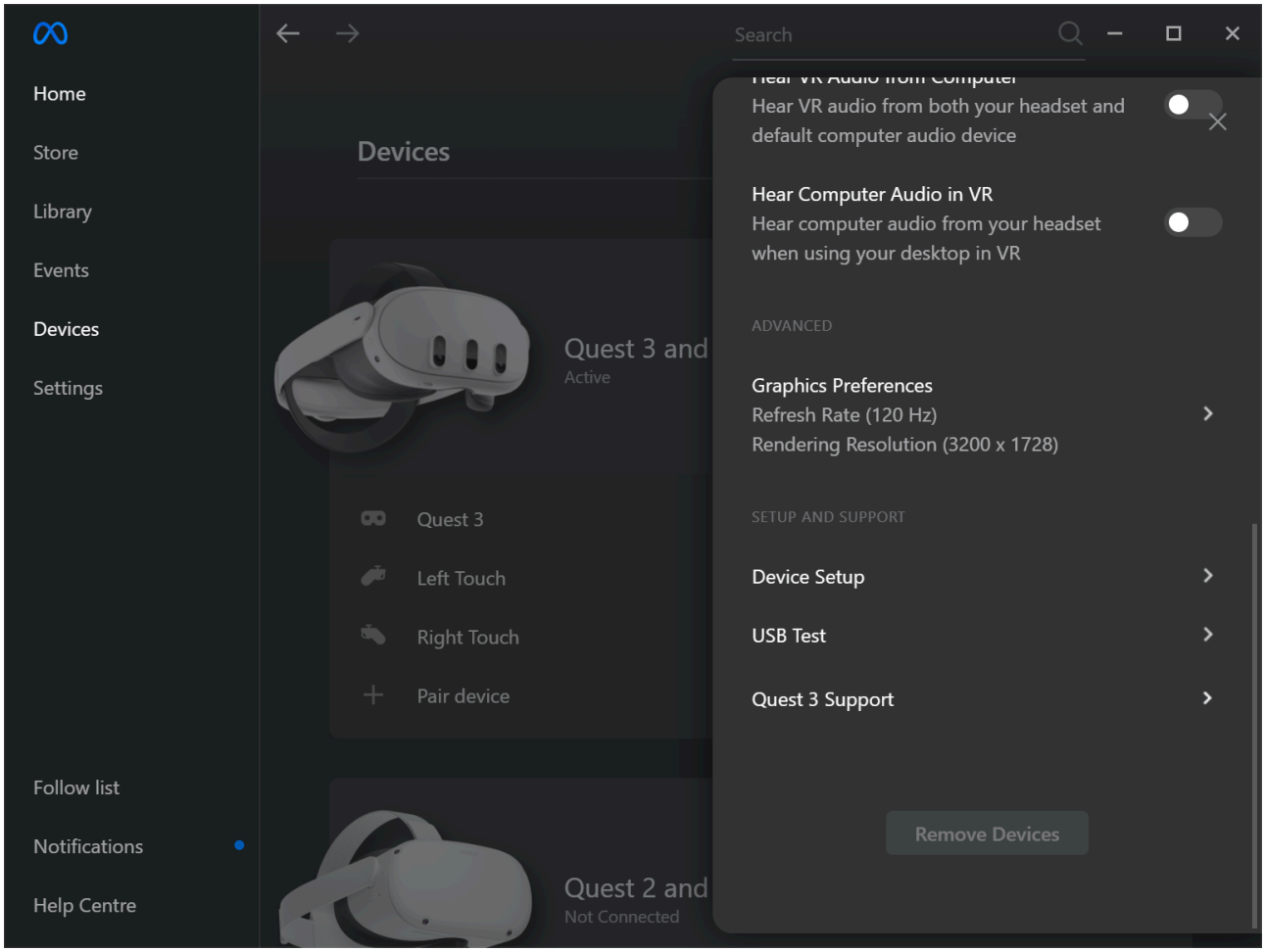
3. In the device settings, scroll down to the `Advanced` section and select `Graphics Preferences`.



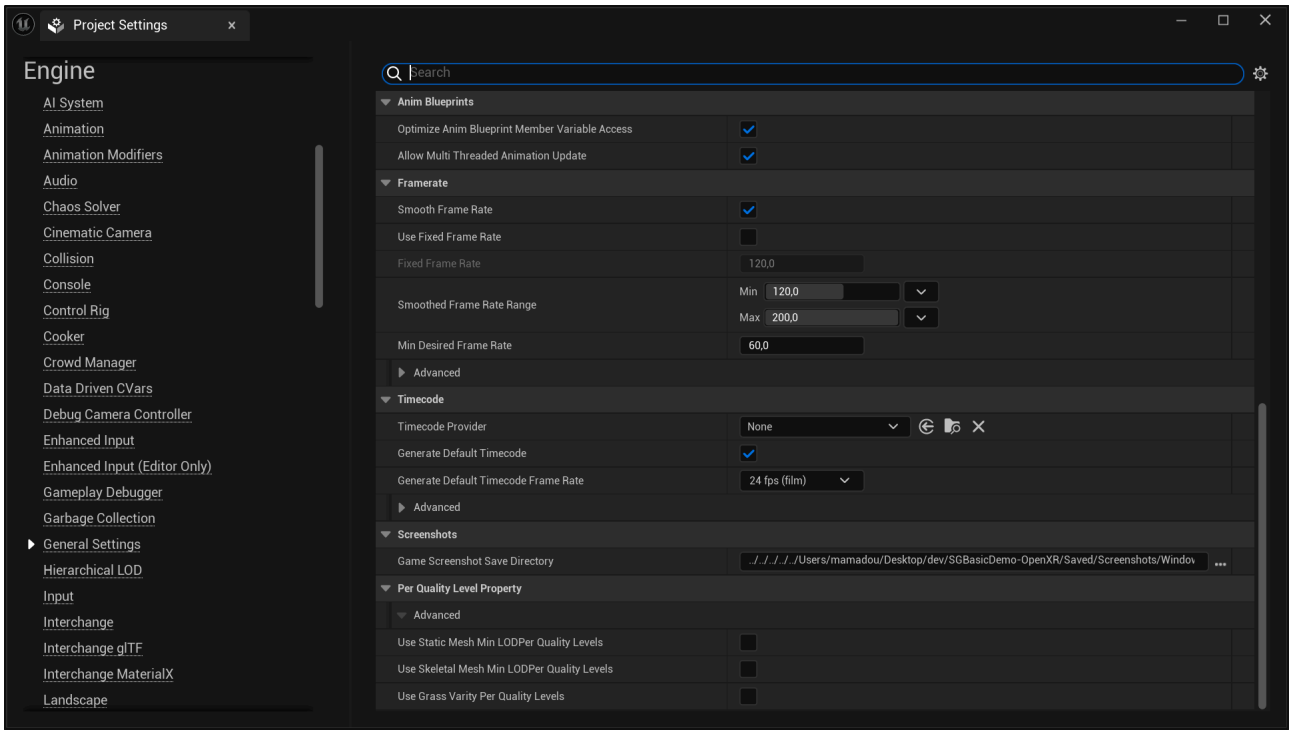
4. Choose your desired refresh rate. In this case select a refresh rate of `120 Hz`. After making your selection, click `OK`, and the Meta Quest Link app will restart to apply the changes.



5. Once the Meta Quest Link app restarts, go back to the `Devices` tab, select your device, and confirm the refresh rate setting under `Advanced > Graphics Preferences`.



6. Now, open your Unreal Engine project and navigate to `Project Settings`. Under `Engine > General Settings > Framerate`, you can fine-tune and experiment with the framerate settings to match your project's requirements.

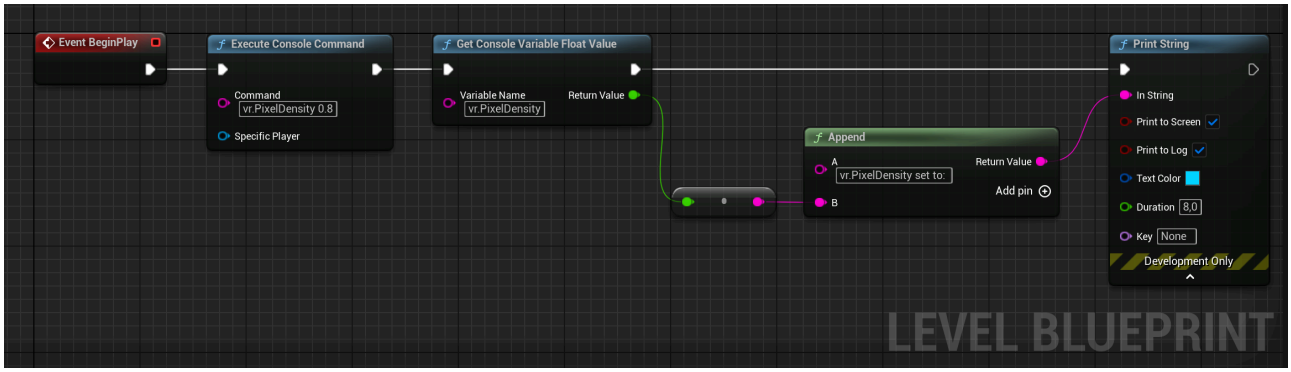


HTC VIVE Specific Optimizations in Standalone Mode

Headsets like the HTC VIVE Focus 3 and Focus Vision have larger framebuffers, higher eye-buffer size, wider fields of view (FOV), and varying refresh rates compared to HMDs like the Meta Quest series. As a result, performance can differ when running the same content across these devices.

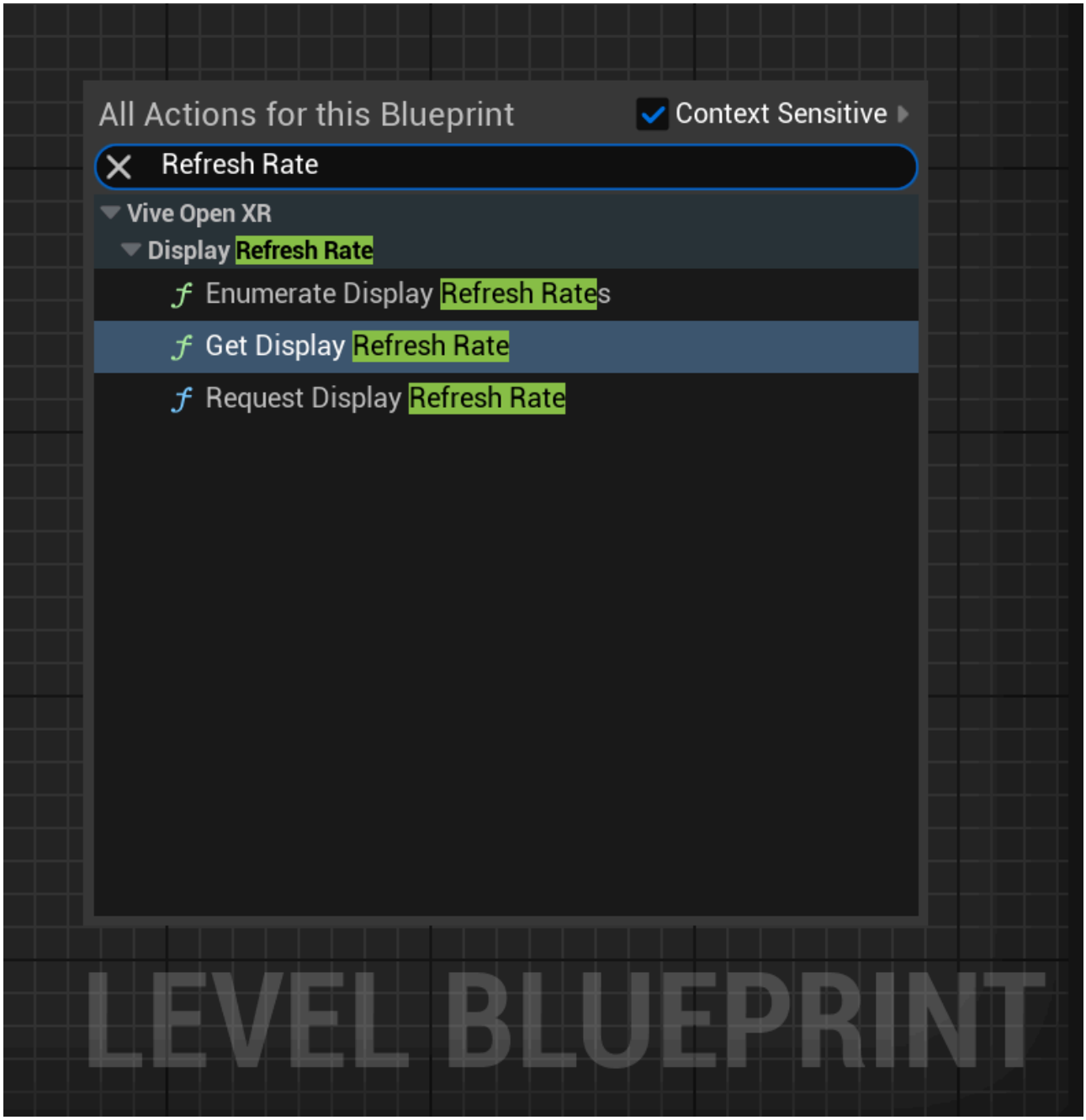
Setting a Custom Pixel Density

If performance on the Focus 3 or Focus Vision is lower than on the Quest, we recommend adjusting the **Pixel Density** to approximately `0.8` (or a value suited to your content). This reduces the eye buffer size, and matches the resolution of Quest 2, hence increasing FPS:

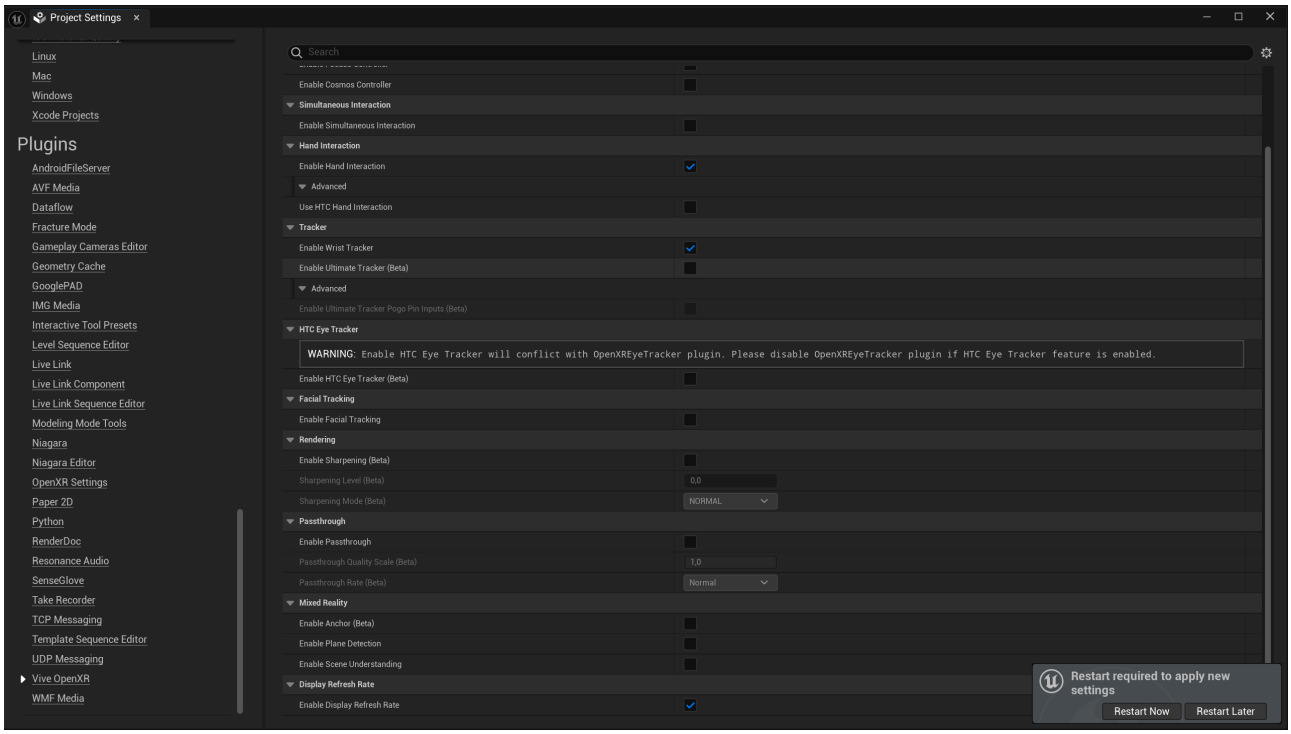


Setting a Lower Refresh Rate

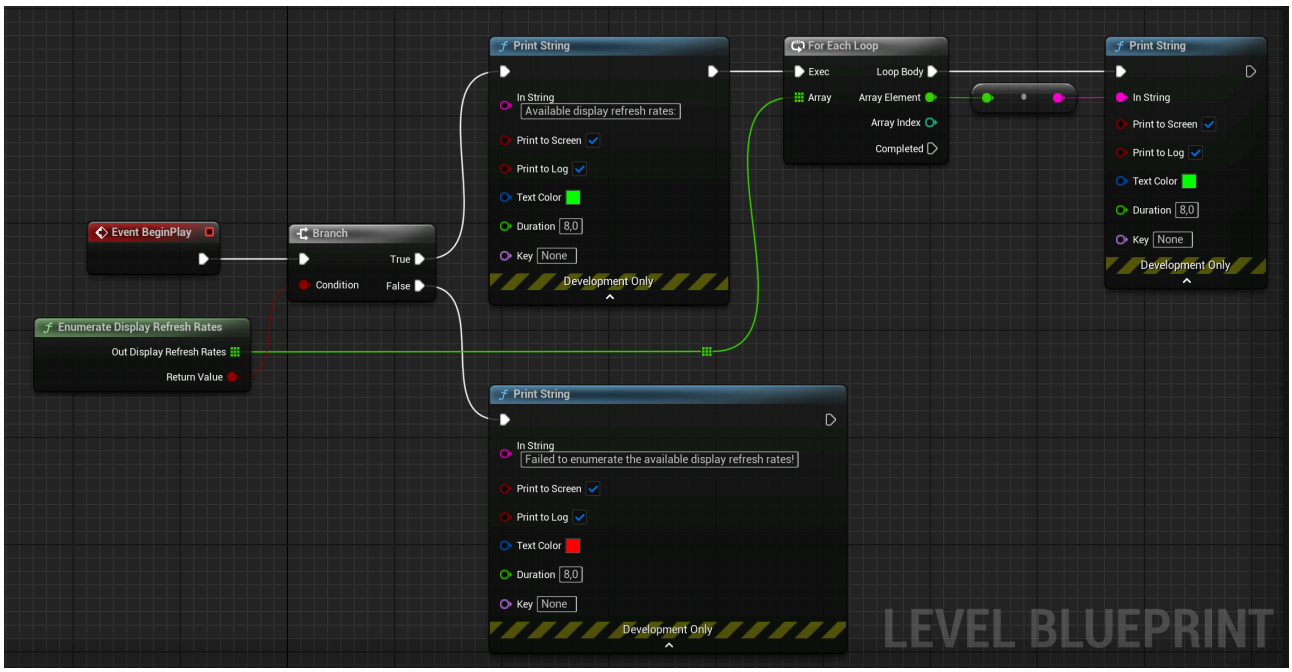
The `ViveOpenXR` plugin provides various Blueprint functions to query the supported refresh rates and adjust the current refresh rate according to your needs:



But, before you'd be able to make these adjustments you have to enable the relevant settings by navigating to `Edit > Project Settings > Plugins > Vive OpenXR` and ensure the `Enable Display Refresh Rate` option is checked. This option enables the OpenXR extension `XR_FB_display_refresh_rate` support which allows your application to dynamically adjust the display refresh rate in order to improve the overall user experience. Please note that You need to restart the engine to apply new settings after changing this setting.

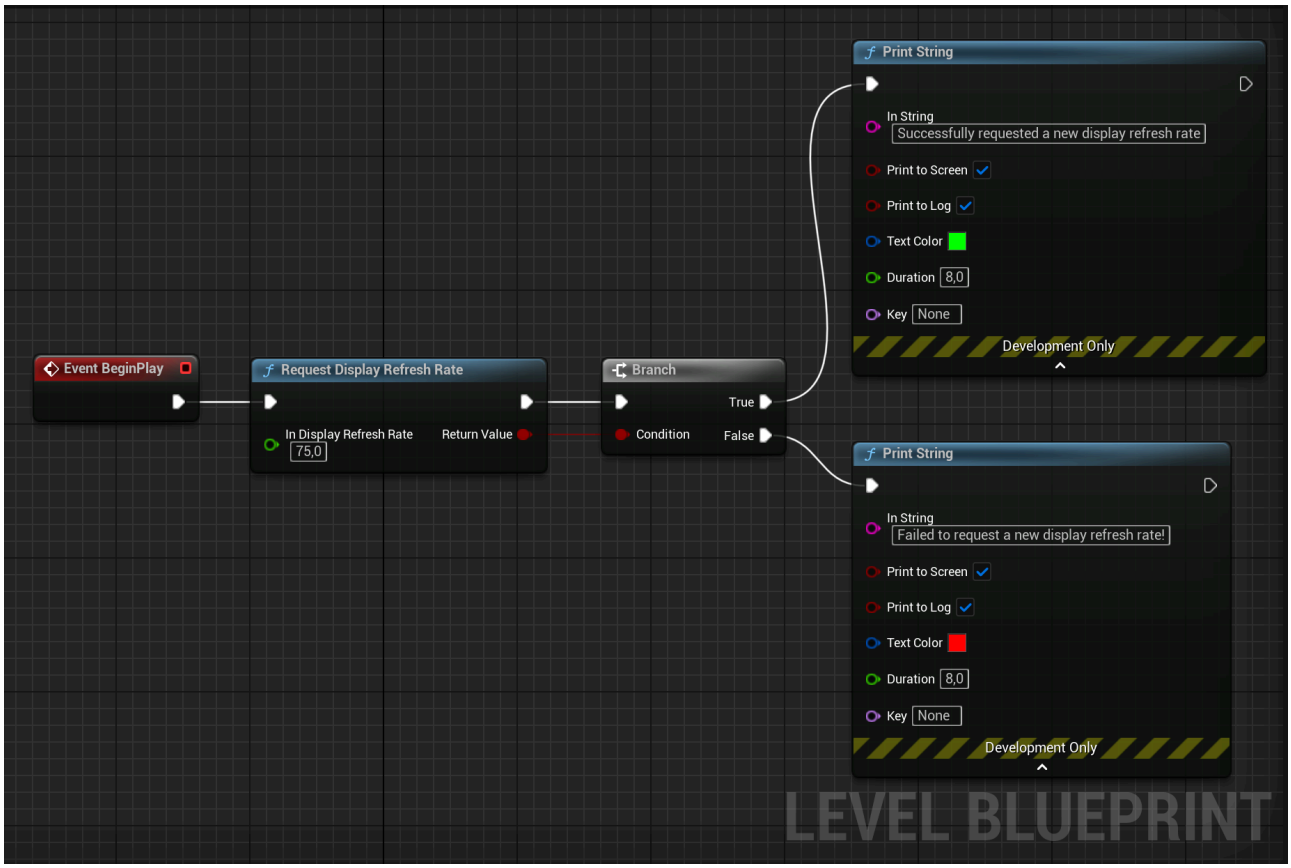


In order to query all the available display refresh rates use the Blueprint function `Enumerate Display Refresh Rates`:

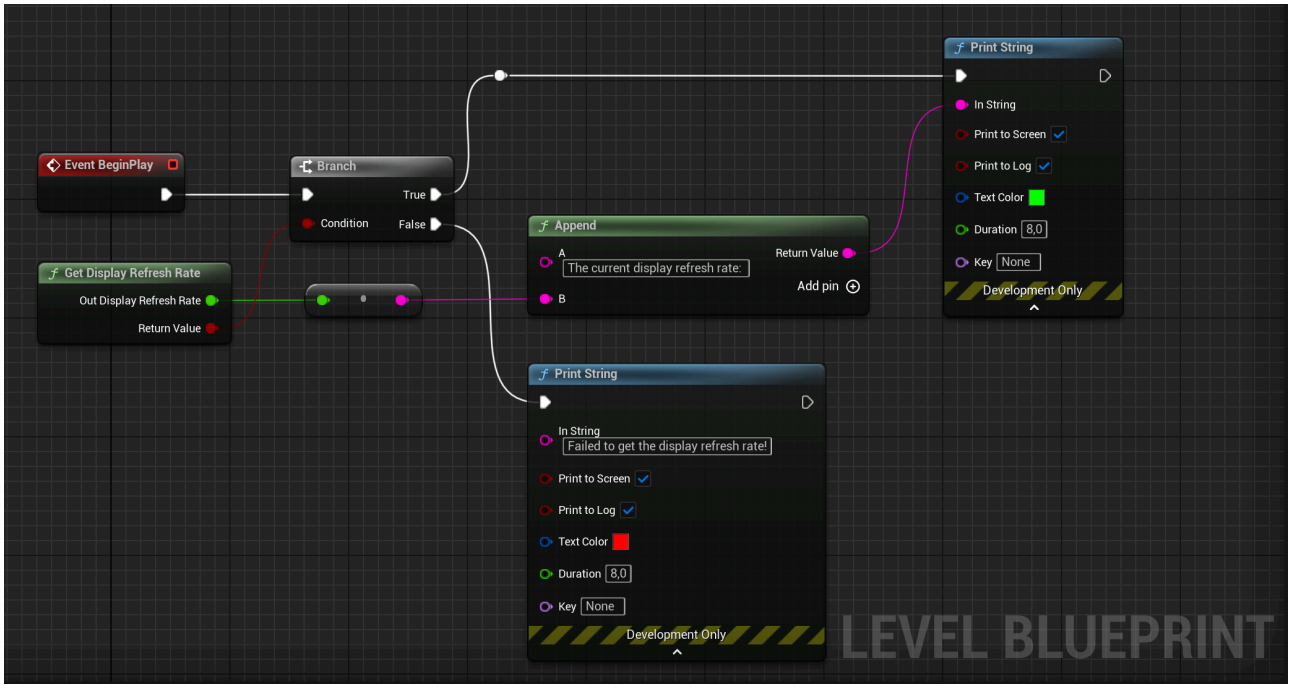


At the moment with most devices it returns `75.0` and `90.0` hz values.

To change the current display refresh rate use the Blueprint function `Request Display Refresh Rate`:



To obtain the current display refresh rate use the Blueprint function `Get Display Refresh Rate`:



If you're looking to squeeze more performance out of your HTC VIVE HMD in standalone mode, consider lowering the target refresh rate from `90hz` to `75hz`. This can help reduce GPU workload and improve overall stability while still maintaining a somewhat smooth experience.

HTC VIVE Optimal Rendering Settings

As a last resort consider adding the following rendering settings to your project's `DefaultEngine.ini` under the `/Script/Engine.RendererSettings` section and experiment with them until you obtain your desired results:

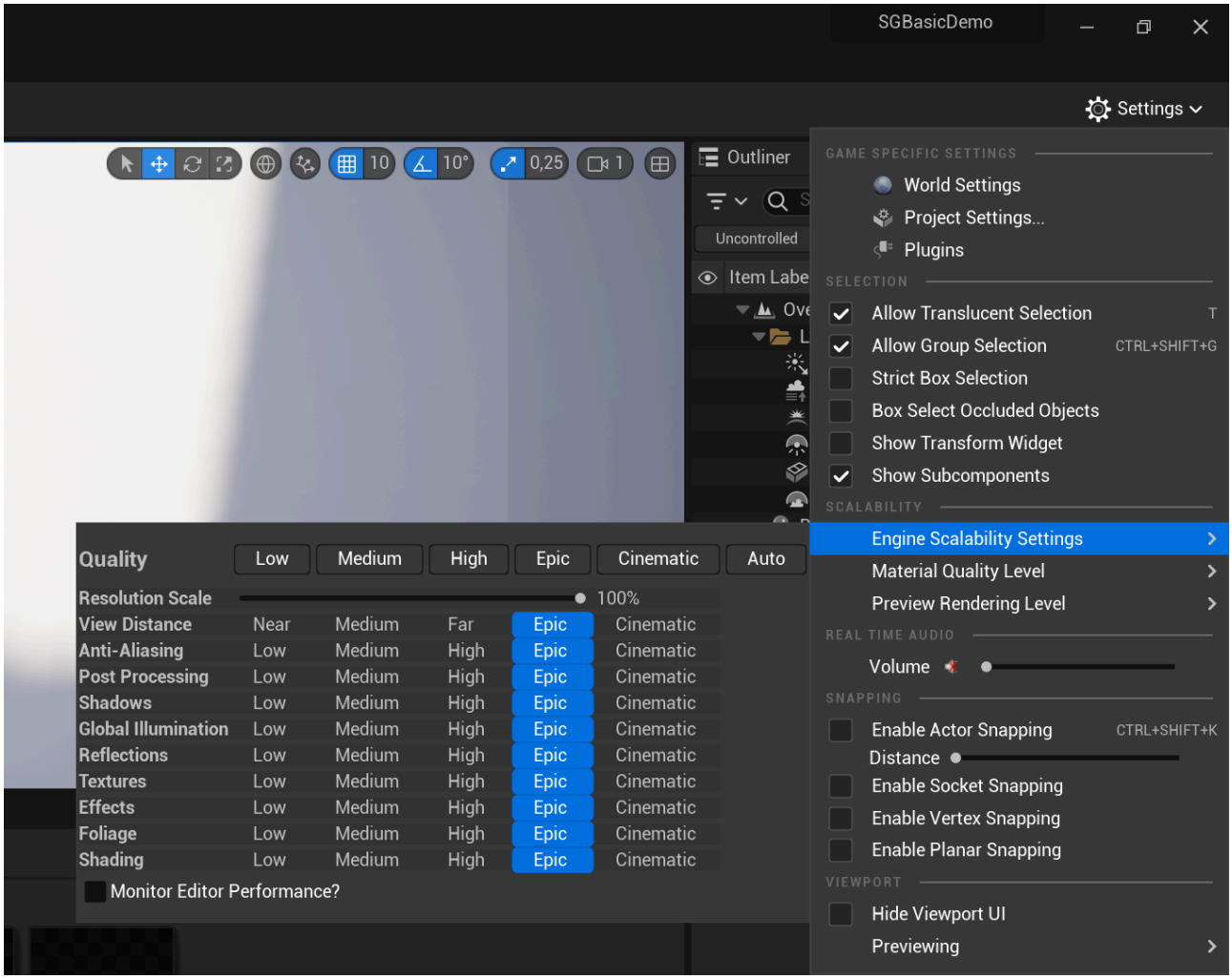
```
[/Script/Engine.RendererSettings]
r.Mobile.DisableVertexFog=True
r.Mobile.AllowDitheredLODTransition=False
r.Mobile.AllowSoftwareOcclusion=False
r.Mobile.VirtualTextures=False
r.DiscardUnusedQuality=False
r.AllowOcclusionQueries=True
r.MinScreenRadiusForLights=0.030000
r.MinScreenRadiusForDepthPrepass=0.030000
r.MinScreenRadiusForCSMDepth=0.010000
r.PrecomputedVisibilityWarning=False
r.TextureStreaming=True
Compat.UseDXT5NormalMaps=False
r.VirtualTextures=False
r.VirtualTexturedLightmaps=False
r.VT.TileSize=128
r.VT.TileBorderSize=4
r.vt.FeedbackFactor=16
r.VT.EnableCompressZlib=True
r.VT.EnableCompressCrunch=False
r.ClearCoatNormal=False
r.ReflectionCaptureResolution=128
r.ReflectionEnvironmentLightmapMixBasedOnRoughness=True
r.ForwardShading=True
r.VertexFoggingForOpaque=True
r.AllowStaticLighting=True
r.NormalMapsForStaticLighting=False
r.GenerateMeshDistanceFields=False
r.DistanceFieldBuild.EightBit=False
r.GenerateLandscapeGIData=False
r.DistanceFieldBuild.Compress=False
r.TessellationAdaptivePixelsPerTriangle=48.000000
r.SeparateTranslucency=True
r.TranslucentSortPolicy=0
TranslucentSortAxis=(X=0.000000,Y=-1.000000,Z=0.000000)
r.CustomDepth=1
r.CustomDepthTemporalAAJitter=True
r.PostProcessing.PropagateAlpha=2
r.DefaultFeature.Bloom=False
r.DefaultFeature.AmbientOcclusion=False
r.DefaultFeature.AmbientOcclusionStaticFraction=True
r.DefaultFeature.AutoExposure=False
r.DefaultFeature.AutoExposure.Method=0
r.DefaultFeature.AutoExposure.Bias=1.000000
r.DefaultFeature.AutoExposure.ExtendDefaultLuminanceRange=True
r.DefaultFeature.AutoExposure.ExtendDefaultLuminanceRange=True
r.EyeAdaptation.EditorOnly=False
```

```
r.DefaultFeature.MotionBlur=False
r.DefaultFeature.LensFlare=False
r.TemporalAA.Upsampling=False
r.SSGI.Enable=False
r.AntiAliasingMethod=3
r.DefaultFeature.LightUnits=1
r.DefaultBackBufferPixelFormat=4
r.Shadow.UnbuiltPreviewInGame=True
r.StencilForLODDither=False
r.EarlyZPass=3
r.EarlyZPassOnlyMaterialMasking=False
r.DBuffer=True
r.ClearSceneMethod=1
r.VelocityOutputPass=0
r.Velocity.EnableVertexDeformation=0
r.SelectiveBasePassOutputs=False
bDefaultParticleCutouts=False
fx.GPUSimulationTextureSizeX=1024
fx.GPUSimulationTextureSizeY=1024
r.AllowGlobalClipPlane=False
r.GBufferFormat=1
r.MorphTarget.Mode=True
r.GPUCrashDebugging=False
vr.InstancedStereo=True
r.MobileHDR=False
vr.MobileMultiView=True
r.Mobile.UseHWsRGBEncoding=True
vr.RoundRobinOcclusion=False
vr.ODSCapture=False
r.MeshStreaming=False
r.WireframeCullThreshold=5.000000
r.RayTracing=False
r.RayTracing.UseTextureLod=False
r.SupportStationarySkylight=True
r.SupportLowQualityLightmaps=True
r.SupportPointLightWholeSceneShadows=True
r.SupportAtmosphericFog=True
r.SupportSkyAtmosphere=True
r.SupportSkyAtmosphereAffectsHeightFog=False
r.SkinCache.CompileShaders=False
r.SkinCache.DefaultBehavior=1
r.SkinCache.SceneMemoryLimitInMB=128.000000
r.Mobile.EnableStaticAndCSMShadowReceivers=True
r.Mobile.EnableMovableLightCSMShaderCulling=True
r.Mobile.AllowDistanceFieldShadows=True
r.Mobile.AllowMovableDirectionalLights=True
r.MobileNumDynamicPointLights=4
r.MobileDynamicPointLightsUseStaticBranch=True
```

```
r.Mobile.EnableMovableSpotlights=False
r.Mobile.EnableMovableSpotlightsShadow=False
r.GPUSkin.Support16BitBoneIndex=False
r.GPUSkin.Limit2BoneInfluences=False
r.SupportDepthOnlyIndexBuffers=True
r.SupportReversedIndexBuffers=True
r.LightPropagationVolume=False
r.Mobile.AmbientOcclusion=False
r.GPUSkin.UnlimitedBoneInfluences=False
r.GPUSkin.UnlimitedBoneInfluencesThreshold=8
r.Mobile.PlanarReflectionMode=0
bStreamSkeletalMeshLODs=(Default=False,PerPlatform=())
bDiscardSkeletalMeshOptionalLODs=(Default=False,PerPlatform=())
VisualizeCalibrationColorMaterialPath=None
VisualizeCalibrationCustomMaterialPath=None
VisualizeCalibrationGrayscaleMaterialPath=None
r.Mobile.AntiAliasing=3
r.Mobile.FloatPrecisionMode=2
r.OpenGL.ForceDXC=0
r.DynamicGlobalIlluminationMethod=1
r.ReflectionMethod=1
r.Shadow.Virtual.Enable=0
r.Lumen.TranslucencyReflections.FrontLayer.EnableForProject=False
```

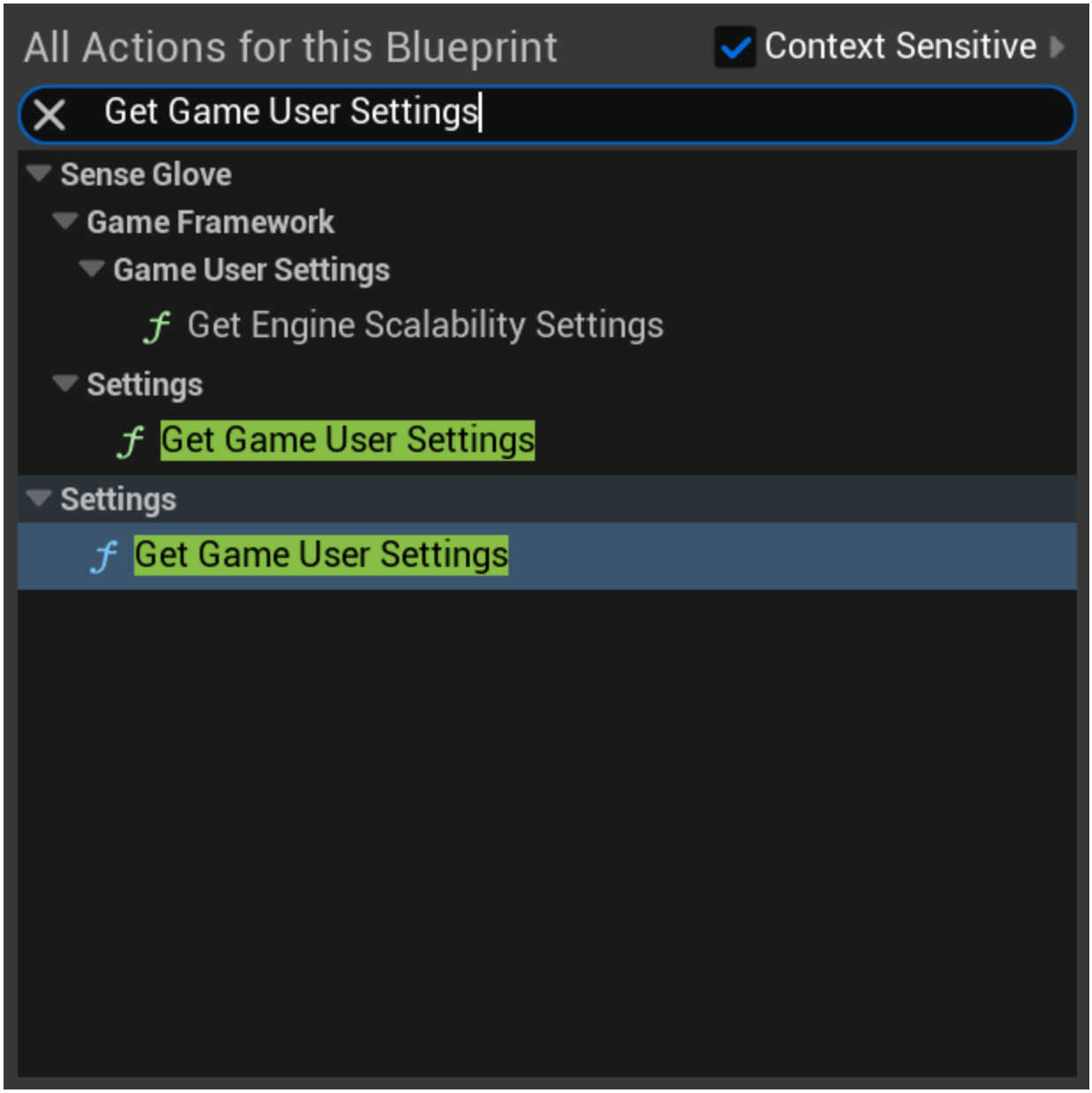
Game User Settings and Engine Scalability Settings

Unreal Engine offers predefined graphics quality profiles known as Engine Scalability Settings, which can be easily adjusted to optimize performance. These settings can be modified directly within the Unreal Editor through the Settings menu on the toolbar or dynamically at runtime using code. Importantly, these settings are universal, meaning changes made in the Unreal Editor will apply to the game when run in PIE (Play In Editor) mode, and settings adjusted via code will also affect the editor itself.



> ### Note
>
> The SenseGlove Unreal Engine Plugin includes specialized console commands that allow you to switch between different Engine Scalability Settings on the fly. Please note that these commands require you to set up SGGameInstance and `SGGameUserSettings`.

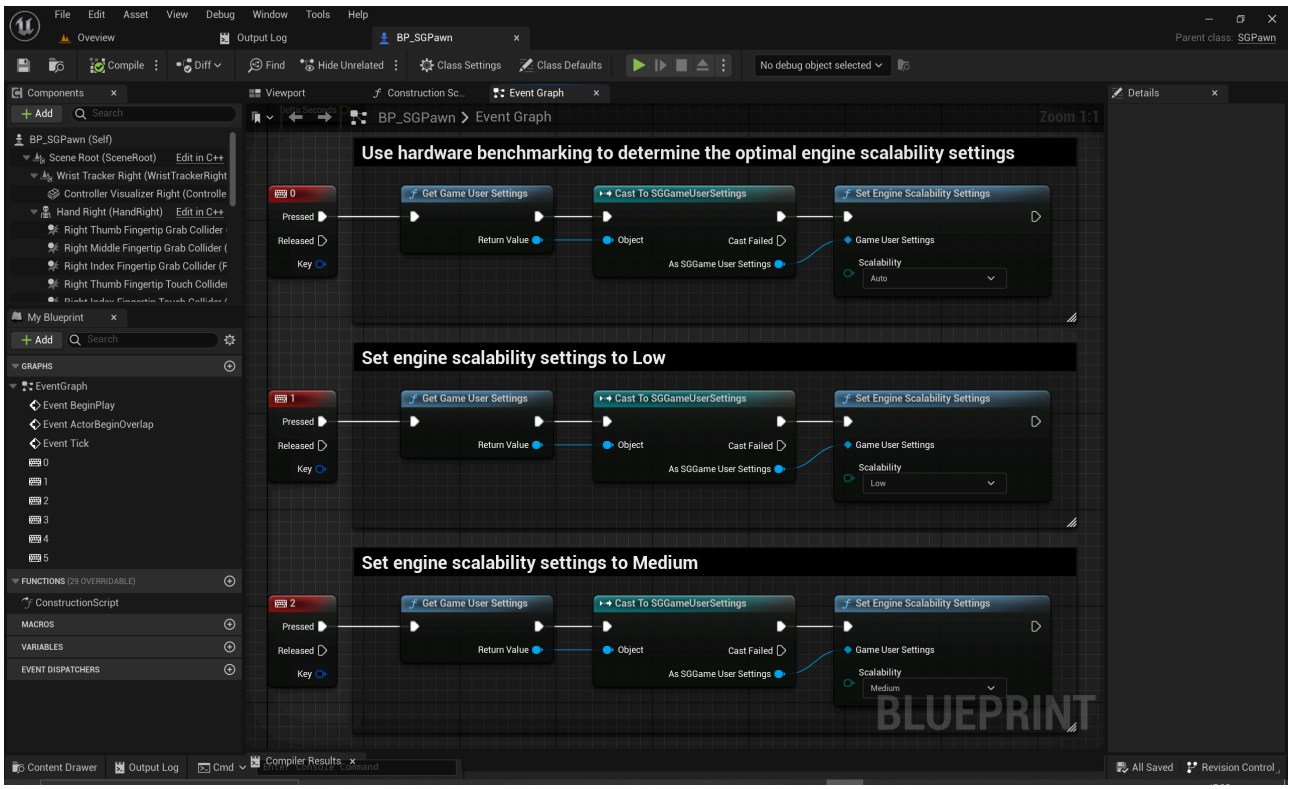
In order to switch between various Engine Scalability Settings, you can use the `Get Game User Settings` Blueprint function and then cast it to `SGGameInstance`.

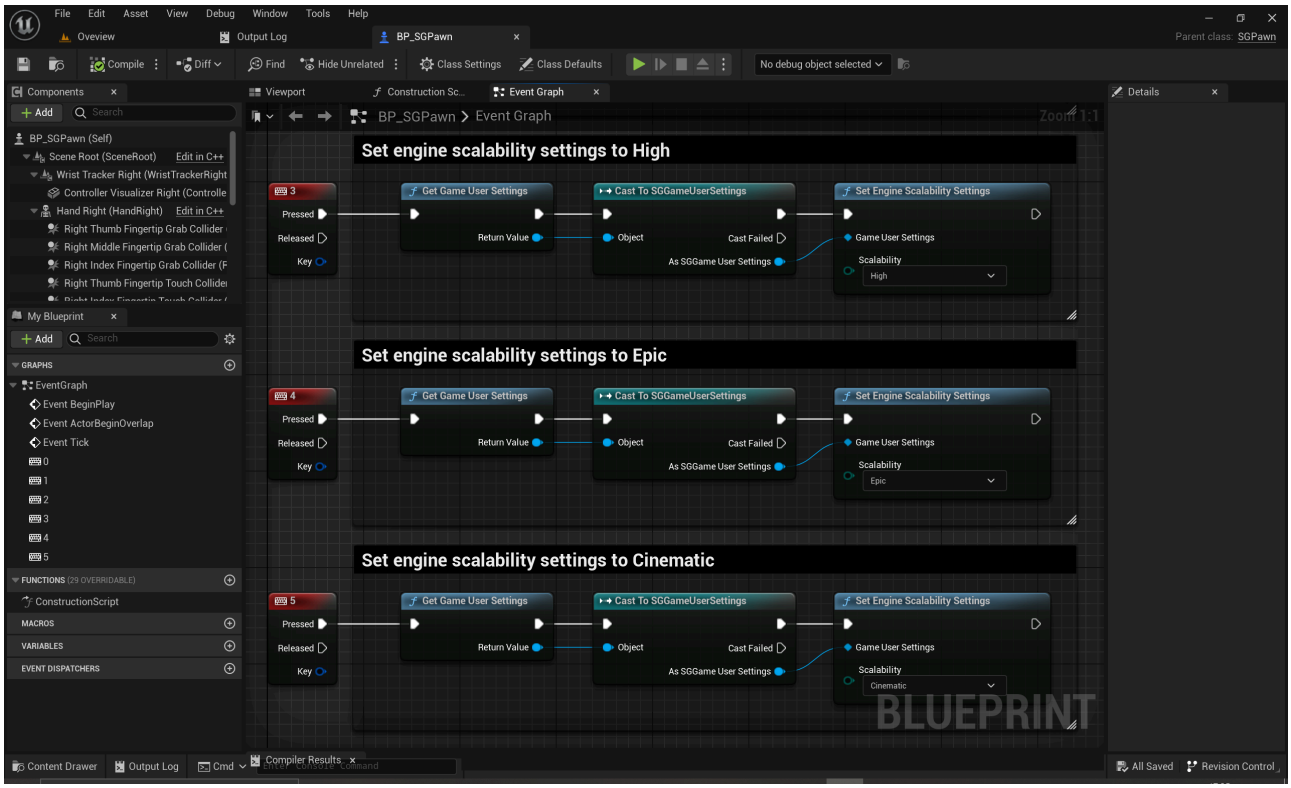


> **Important**
>
> Unreal Engine's default Blueprint functions only allow you to set Engine Scalability Settings to `Low` or `Epic`. To access the full range of settings, `SGGameUserSettings` extends Blueprint access to all Engine Scalability Settings and includes hardware benchmarking to detect the optimal settings. Therefore, it's essential to make `SGGameUserSettings` or a subclass of it the default Game User Settings class to utilize all these features.

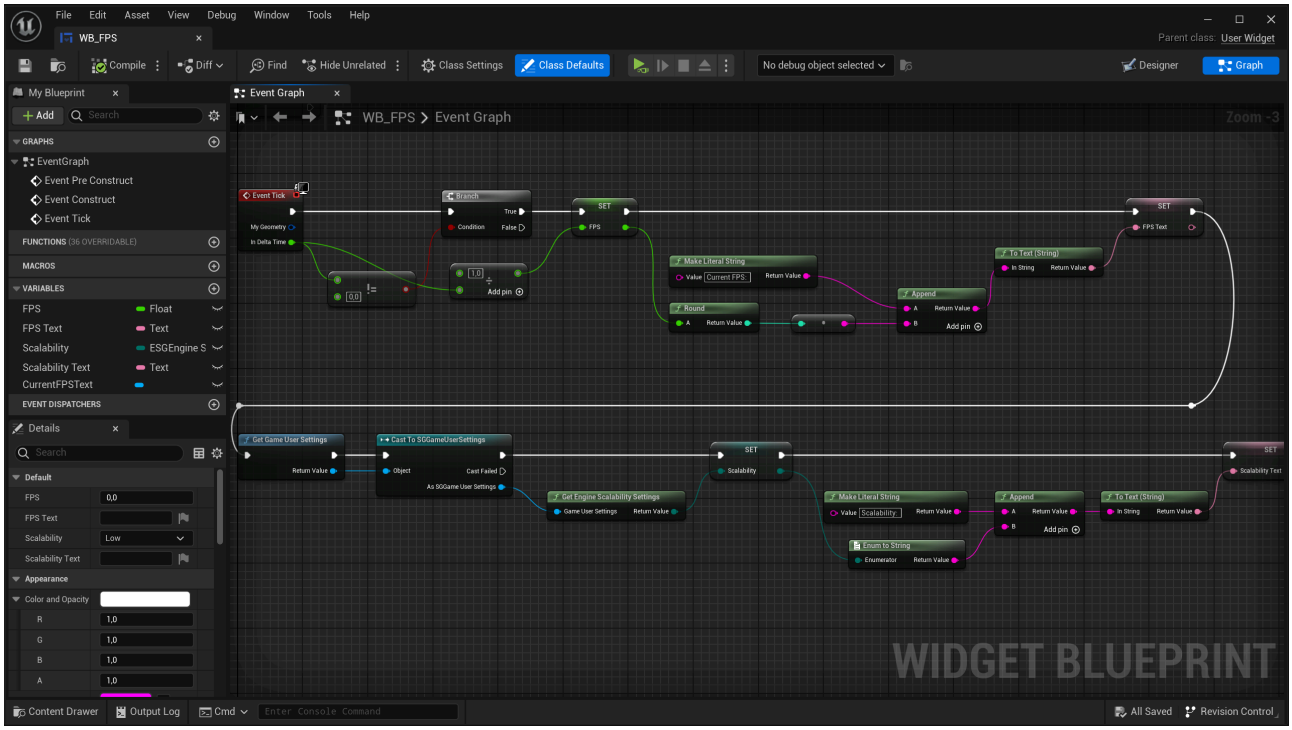
The following Blueprint code from the SGBasicDemo-OpenXR example scene demonstrates how to bind numeric keys `1` to `5` to set various Engine Scalability Settings, and key `0` to utilize hardware benchmarking to determine the optimal Engine Scalability Settings:

- `0` : Use hardware-benchmarking to determine the optimal Engine Scalability Settings.
- `1` : Set Engine Scalability Settings to `Low`.
- `2` : Set Engine Scalability Settings to `Medium`.
- `3` : Set Engine Scalability Settings to `High`.
- `4` : Set Engine Scalability Settings to `Epic`.
- `5` : Set Engine Scalability Settings to `Cinematic`.





> ## Tip
>
> The SGBasicDemo-OpenXR includes an example 3D widget actor that displays the current FPS and Engine Scalability Settings. This widget can be placed within a VR scene and is located in `All > Content > Blueprints > UI > BP_FPS3DWidget`. The underlying UMG widget can be found at `All > Content > Blueprints > UI > WB_FPS` within the Content Browser for the SGBasicDemo-OpenXR example scene.

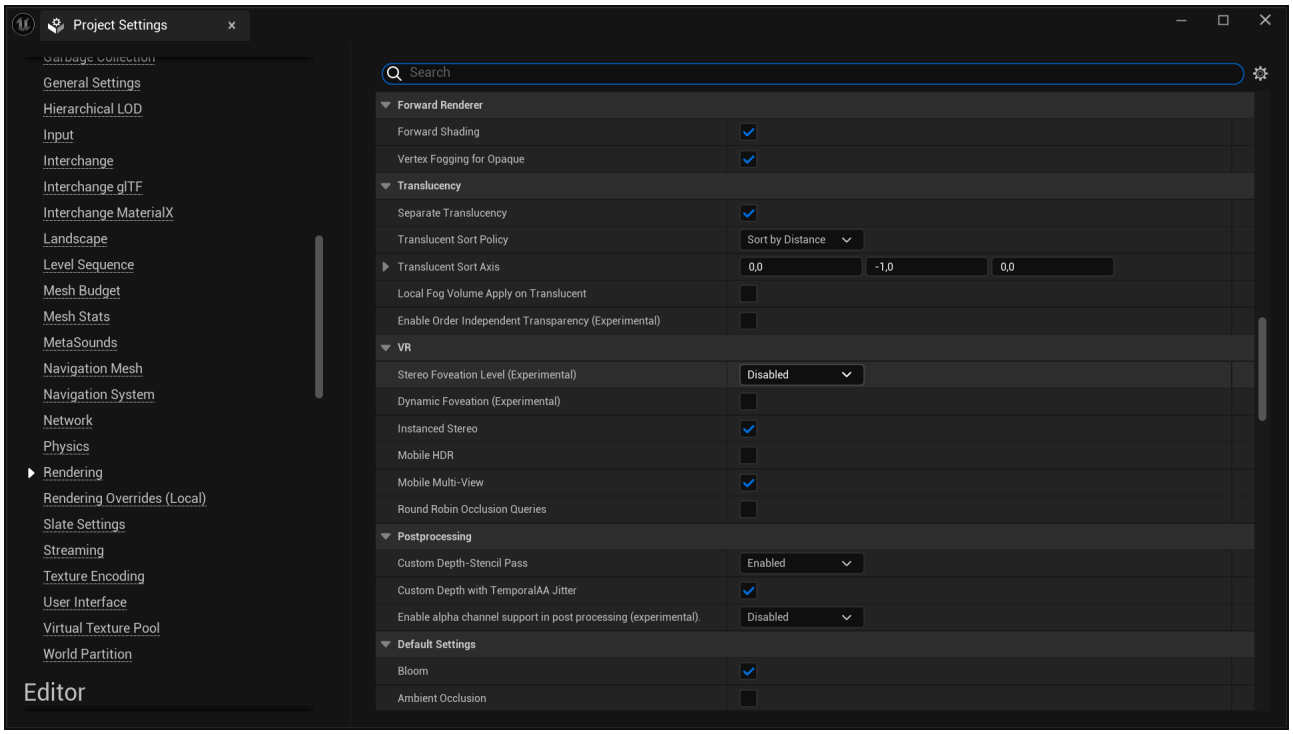


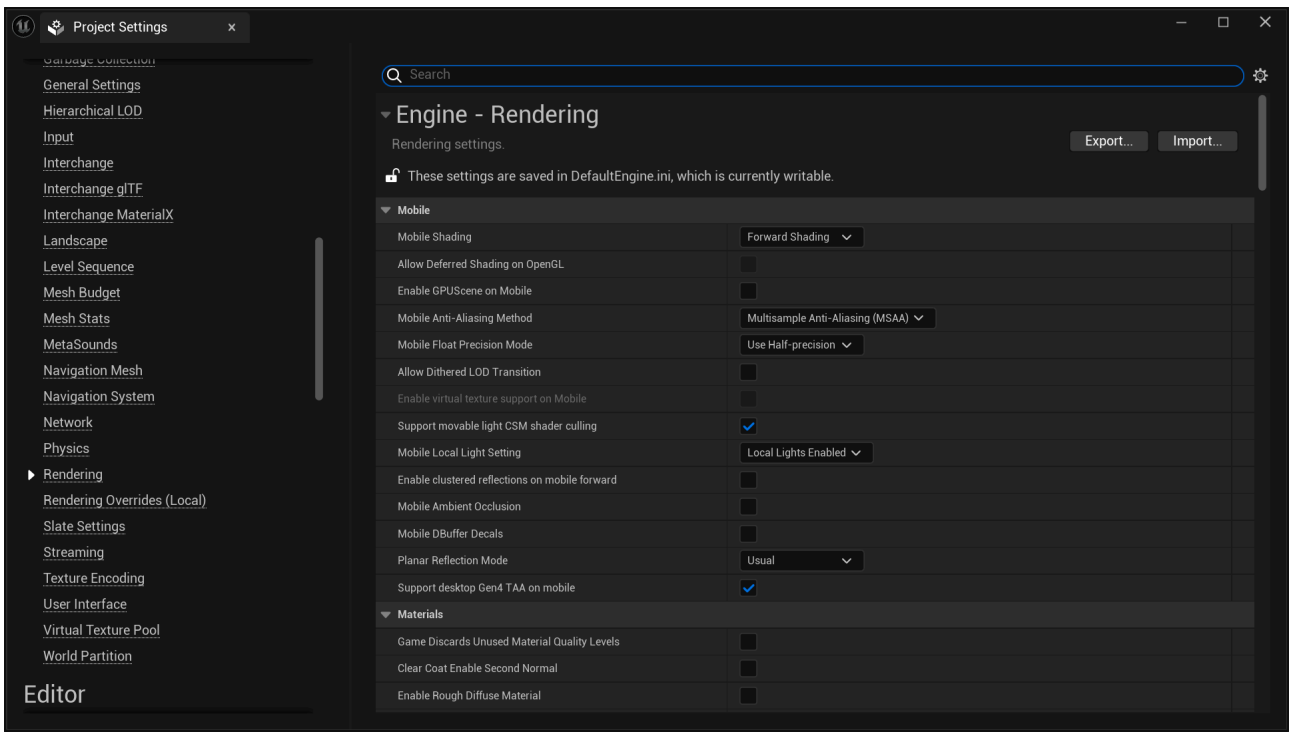
Optimizing Unreal Projects for Mobile

We have the SGBasicDemo-OpenXR project, which has been optimized for mobile. You can explore the project configuration by reviewing the settings inside the `Config` folder and compare them with your own project settings. In addition, here are some crucial guidelines and settings that you may want to adjust for further optimization:

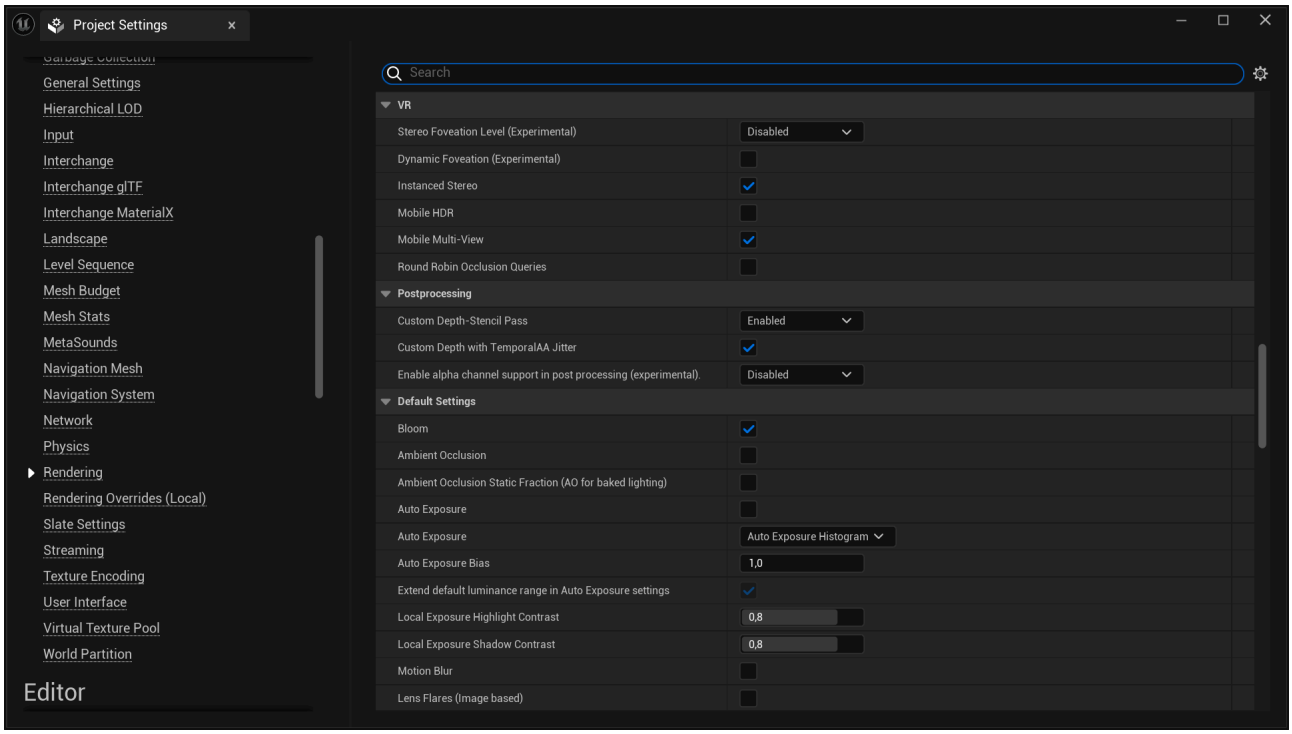
General Rendering Settings

Forward Shading: Enable Forward Shading for better performance. It's more efficient on mobile platforms.

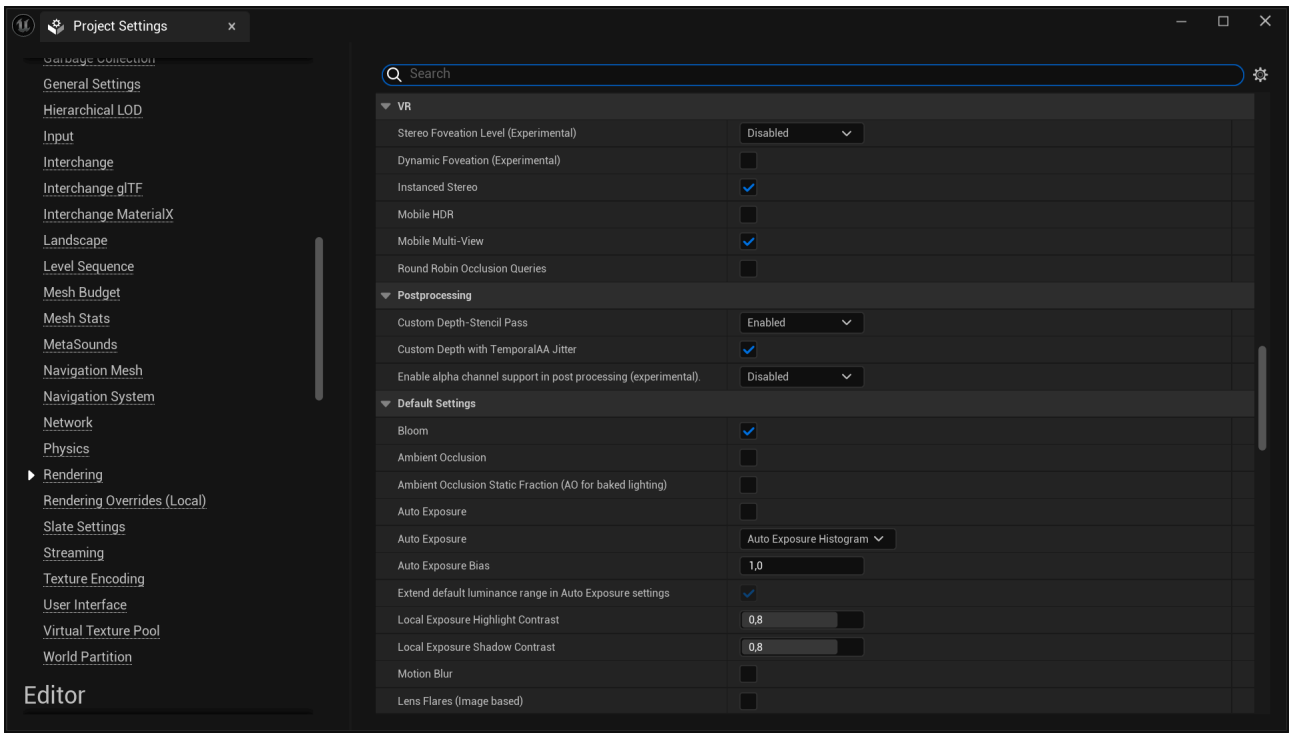




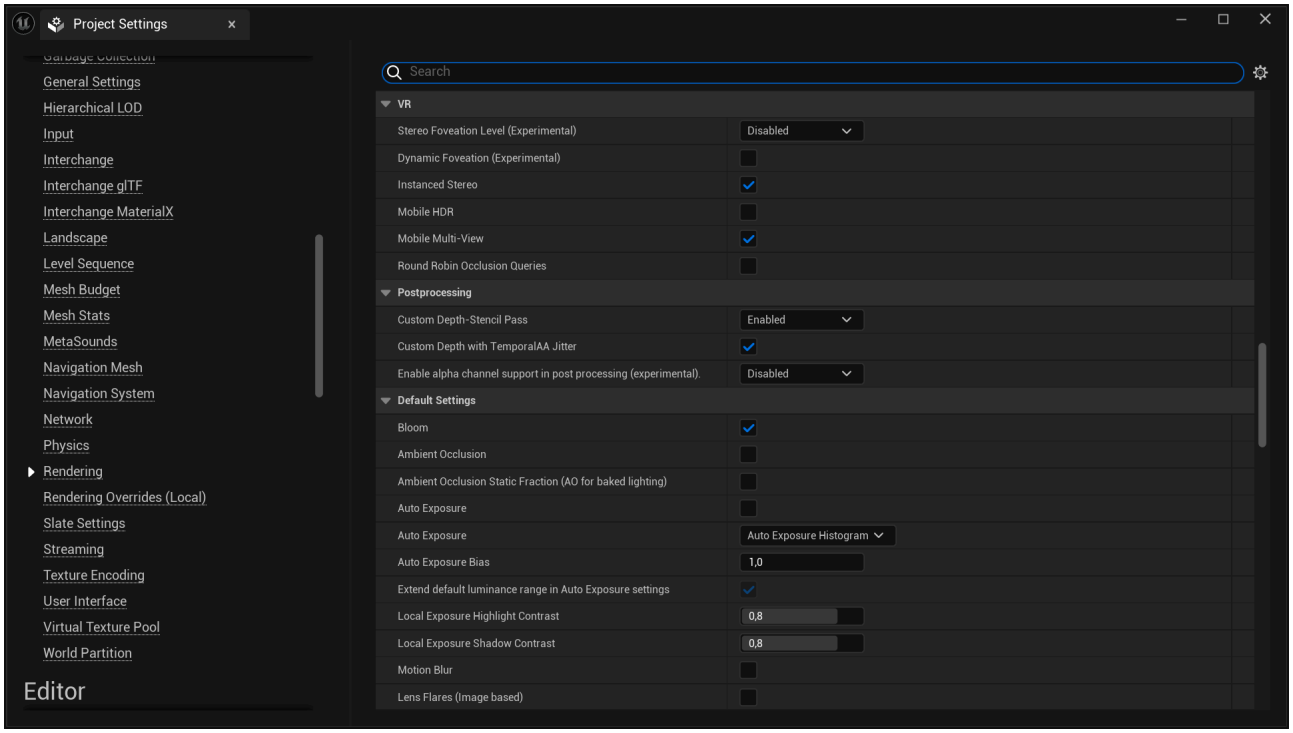
Mobile HDR: Disable this setting. Mobile HDR can significantly affect performance, especially on lower-end devices.



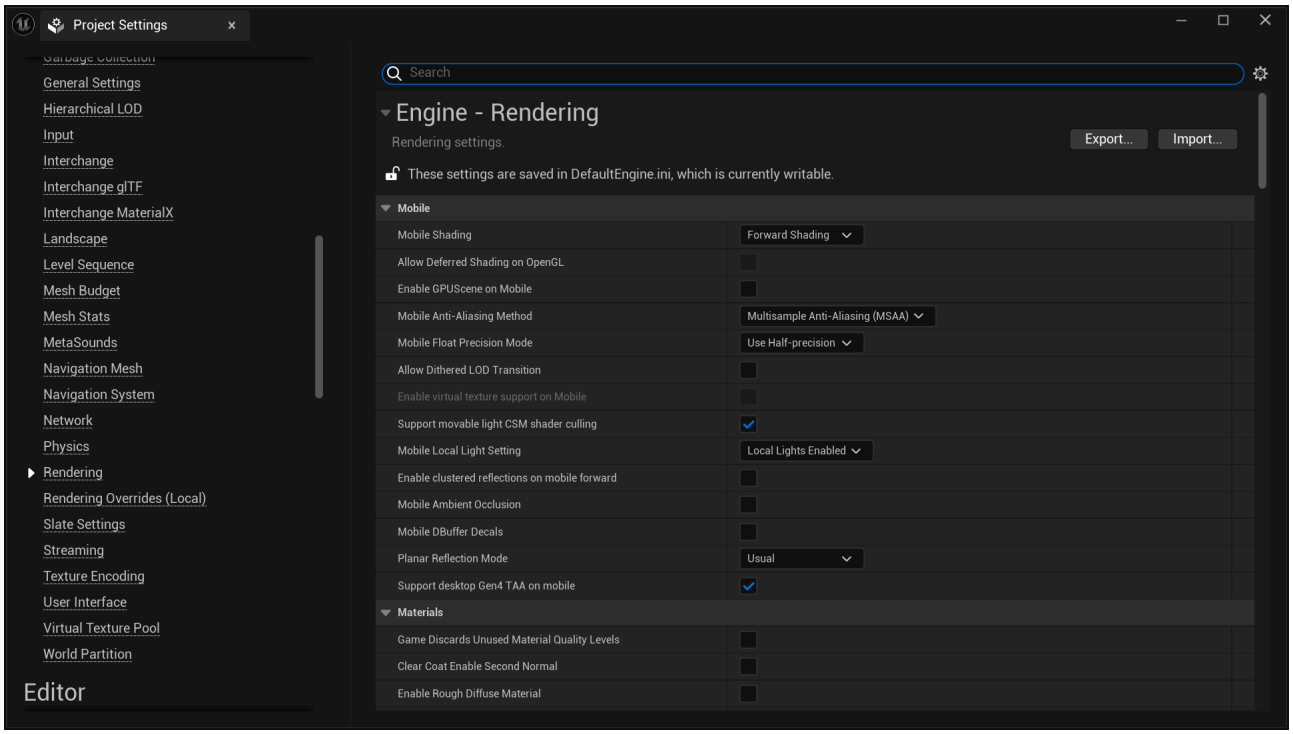
Instanced Stereo: Enable this setting. It is a rendering technique used in Unreal Engine primarily for virtual reality (VR) applications. Its main purpose is to optimize the rendering process when creating VR experiences by reducing the workload associated with rendering two slightly different images for each eye.



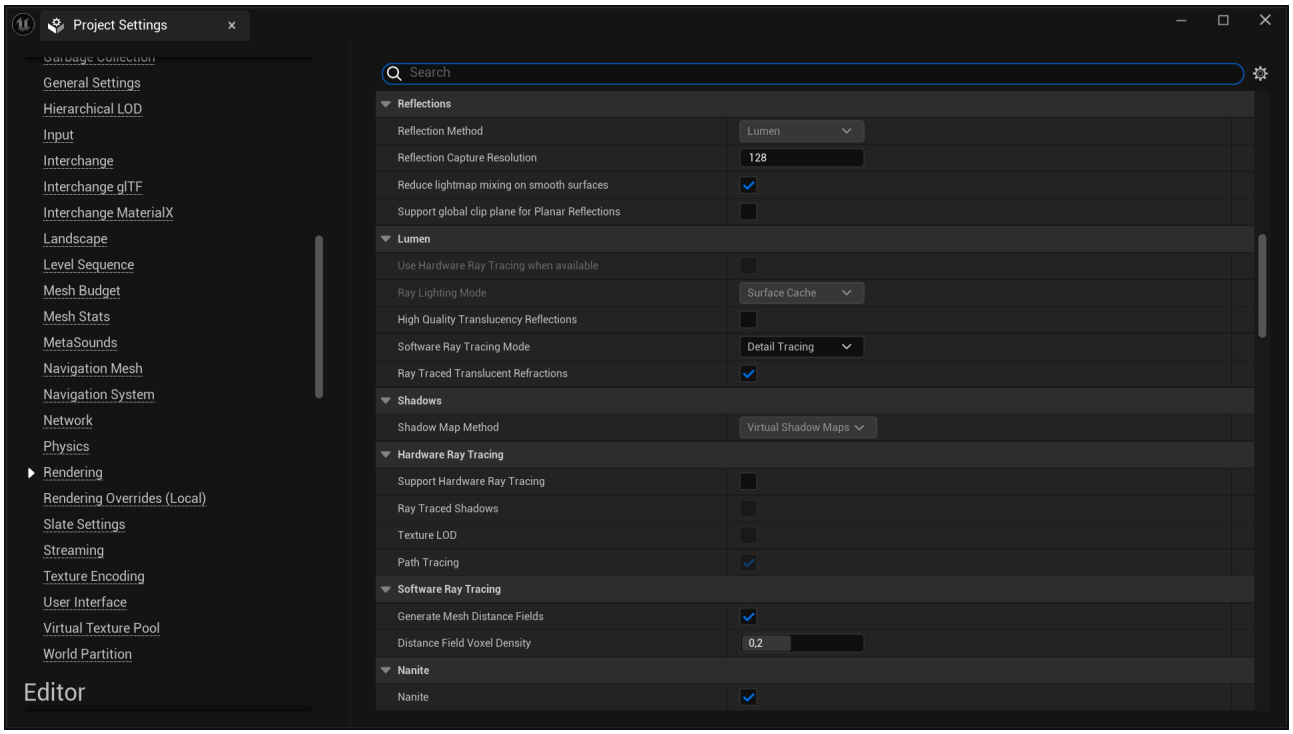
Mobile Multi-View: Enable this setting. It is a rendering feature in Unreal Engine designed to optimize the performance of Virtual Reality (VR) applications on mobile devices, particularly when using VR platforms like Google Daydream or Samsung Gear VR. It is similar in concept to Instanced Stereo, but specifically optimized for mobile hardware.



Mobile Anti-Aliasing Method: Use `FXAA (Fast Approximate Anti-Aliasing)` or `MSAA (Multisample Anti-Aliasing)`. `MSAA` is often preferred for mobile as it gives better visual quality without a huge performance hit.

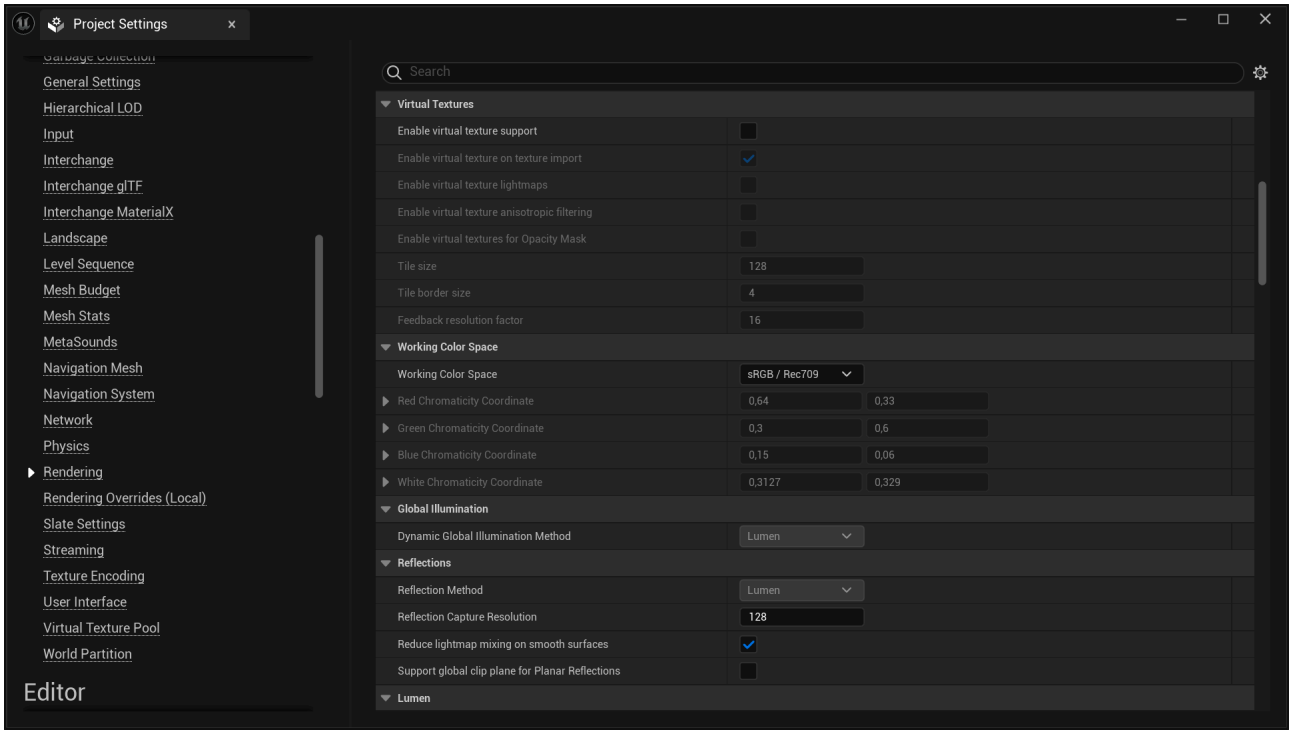


Reflection Capture Resolution: Reduce this value (e.g., 128 or 256) to decrease the memory usage.

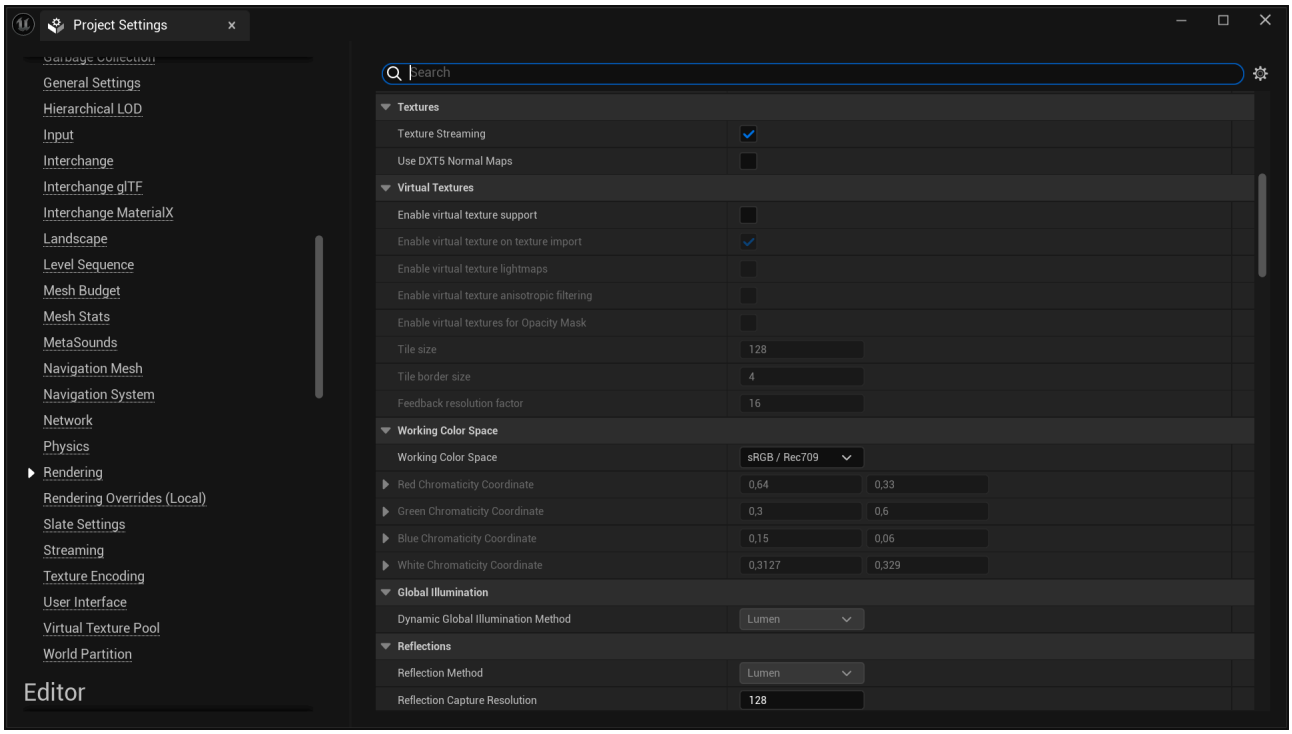


Texture Settings

Enable virtual texture support: Disable this setting.



Texture Streaming: Enable texture streaming to ensure textures load progressively, which helps in reducing memory usage.



Texture Quality: Lower the overall texture quality to Medium or Low depending on the target device capabilities.

Texture Compression: Use ASTC compression for Android to ensure the textures are optimized for mobile devices.

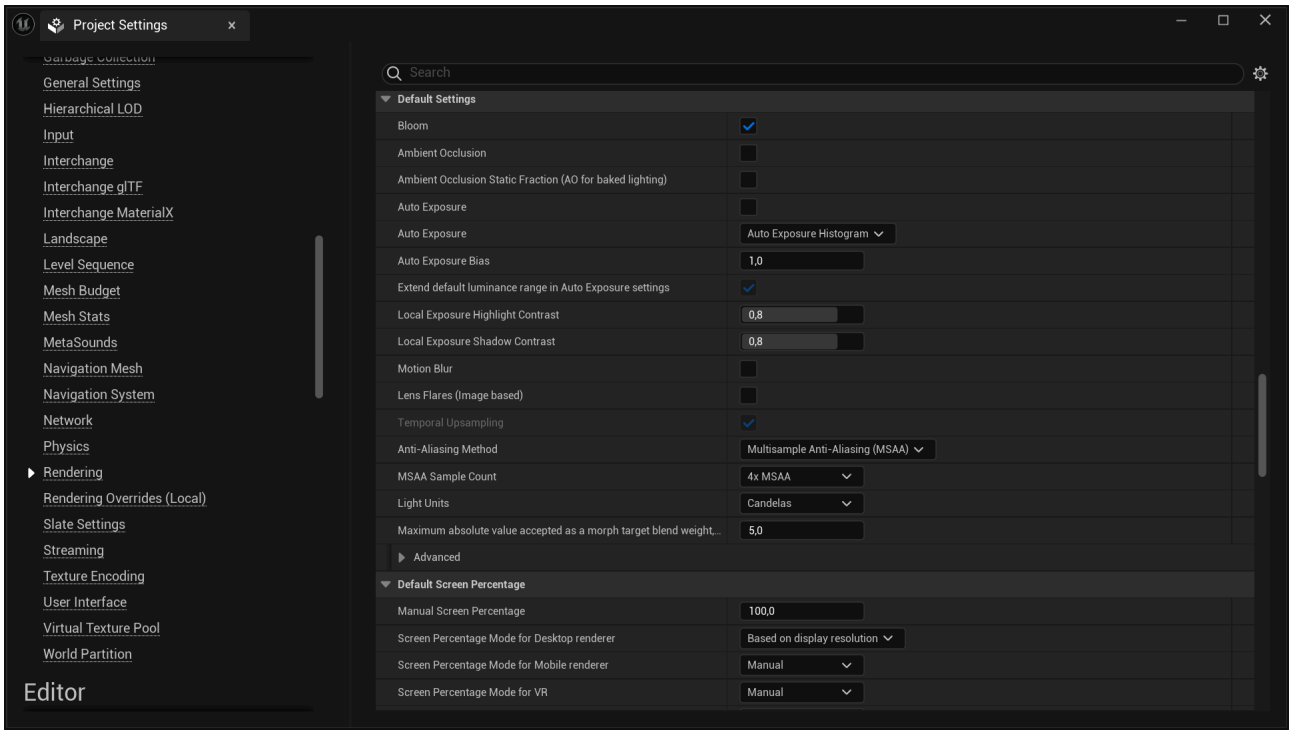
Lighting Settings

Use Static Lighting: Prefer static lighting over dynamic lighting for better performance.

Lightmap Resolution: Use a lower lightmap resolution (e.g., 32 or 64) for mobile to reduce memory usage.

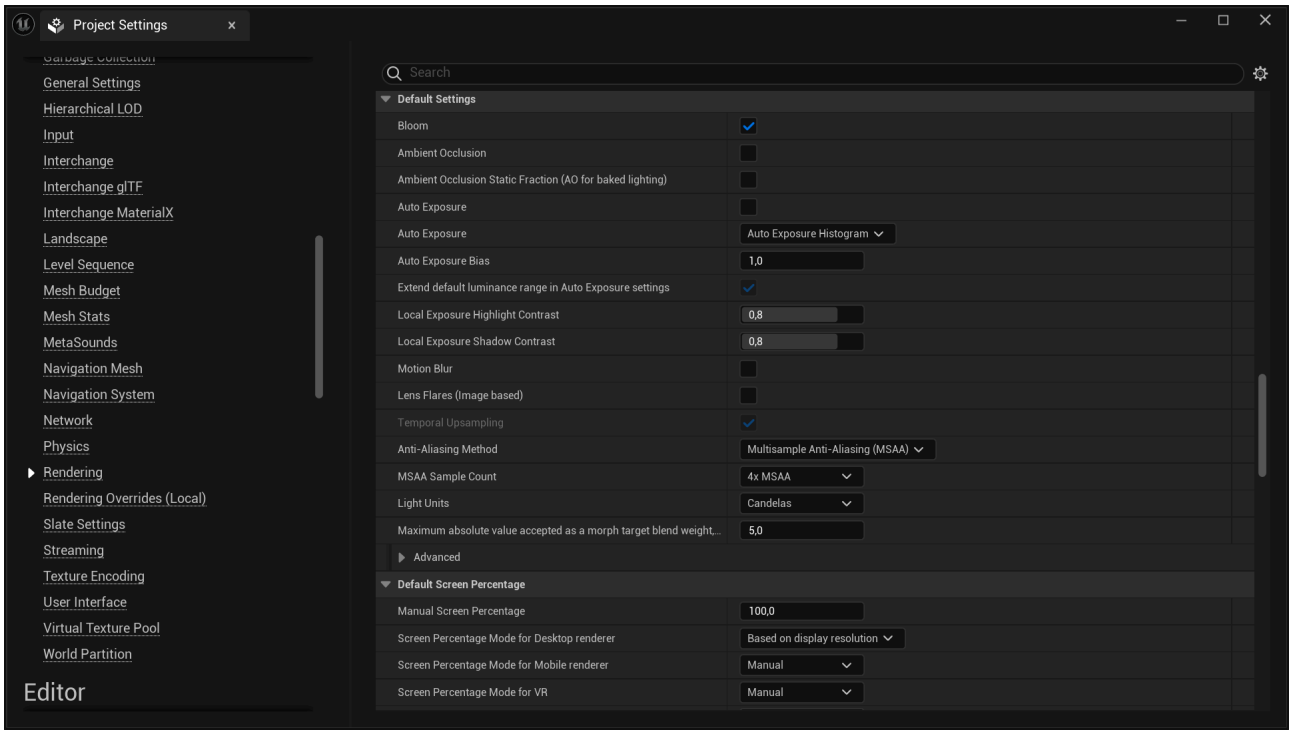
Dynamic Shadows: Disable or minimize the use of dynamic shadows. If required, use CSM (Cascaded Shadow Maps) with low resolution and distance.

Distance Field Shadows/Ambient Occlusion: Disable these features as they are costly on mobile platforms.



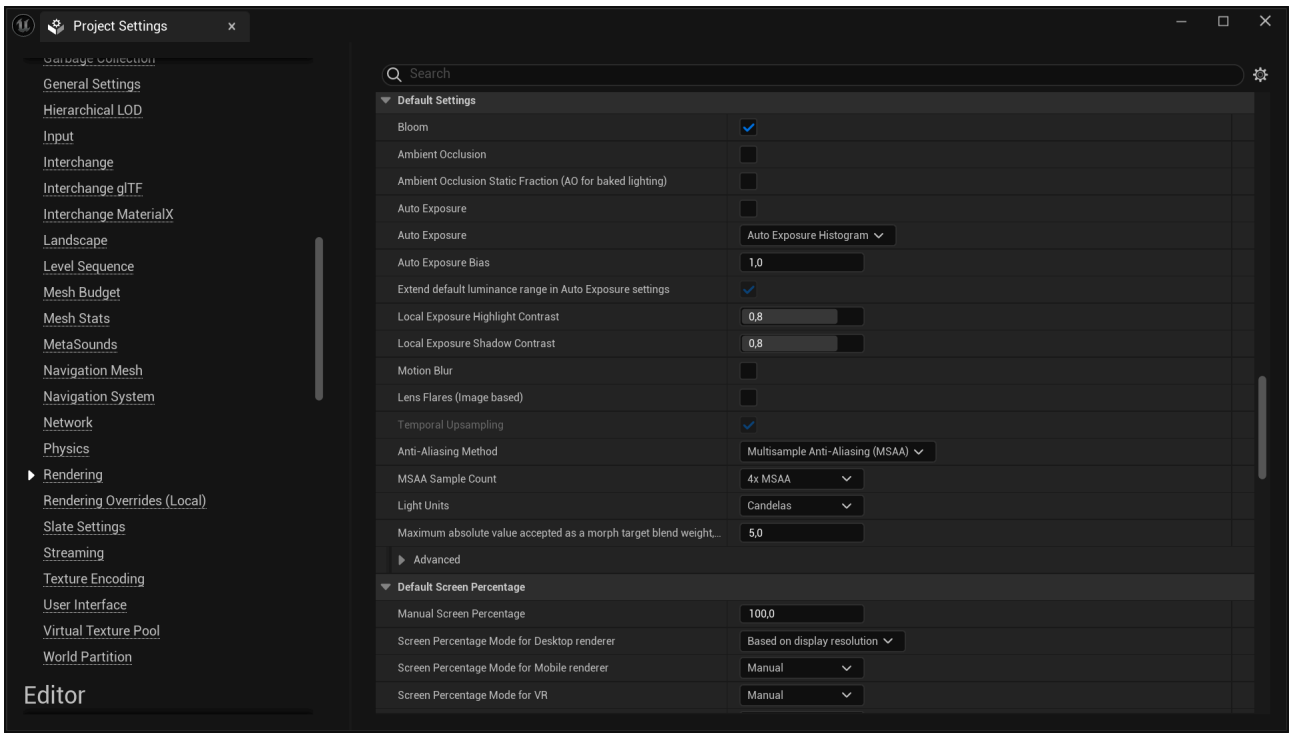
Post-Processing Settings

Bloom, Lens Flares, and Auto Exposure: Minimize or disable these effects as they can be performance-intensive.



Screen Space Reflections: Disable this setting as it is costly in terms of performance on mobile devices.

Motion Blur: Disable this feature to save on processing power.



Materials and Shaders

Material Complexity: Use simple materials with few instructions and limit the number of textures and shader nodes.

Specular Highlights: Consider reducing or disabling specular highlights on materials to save on performance.

LOD (Level of Detail) Models: Ensure that LODs are set up correctly for all models, with appropriate reduction in polygon count for distant objects.

Level of Detail (LOD) Settings

Mesh LODs: Configure LODs for all meshes to reduce polygon count at distances.

Screen Size: Adjust screen size settings for LODs to ensure they switch at appropriate distances for mobile screens.

Engine Scalability Settings

Resolution Scale: Lower the resolution scale (e.g., 70% or 80%) to improve performance while maintaining visual quality.

View Distance: Set to Medium or Low to reduce the amount of detail rendered at long distances.

Shadows: Set to Low or Off for better performance.

Textures: Set to Medium or Low depending on the device's capabilities.

Effects: Set to Low to reduce the complexity of visual effects.

> **Note**
>
> See Game User Settings and Engine Scalability Settings for more details.

Physics and Collision

Physics Simulation: Limit the use of physics simulation where possible, as it can be expensive on mobile devices.

Collision Complexity: Use simple collision meshes instead of complex ones to improve performance.

Audio Settings

Sample Rate: Lower the sample rate to reduce memory usage and processing load.

Number of Audio Channels: Limit the number of audio channels used in the project to reduce CPU usage.

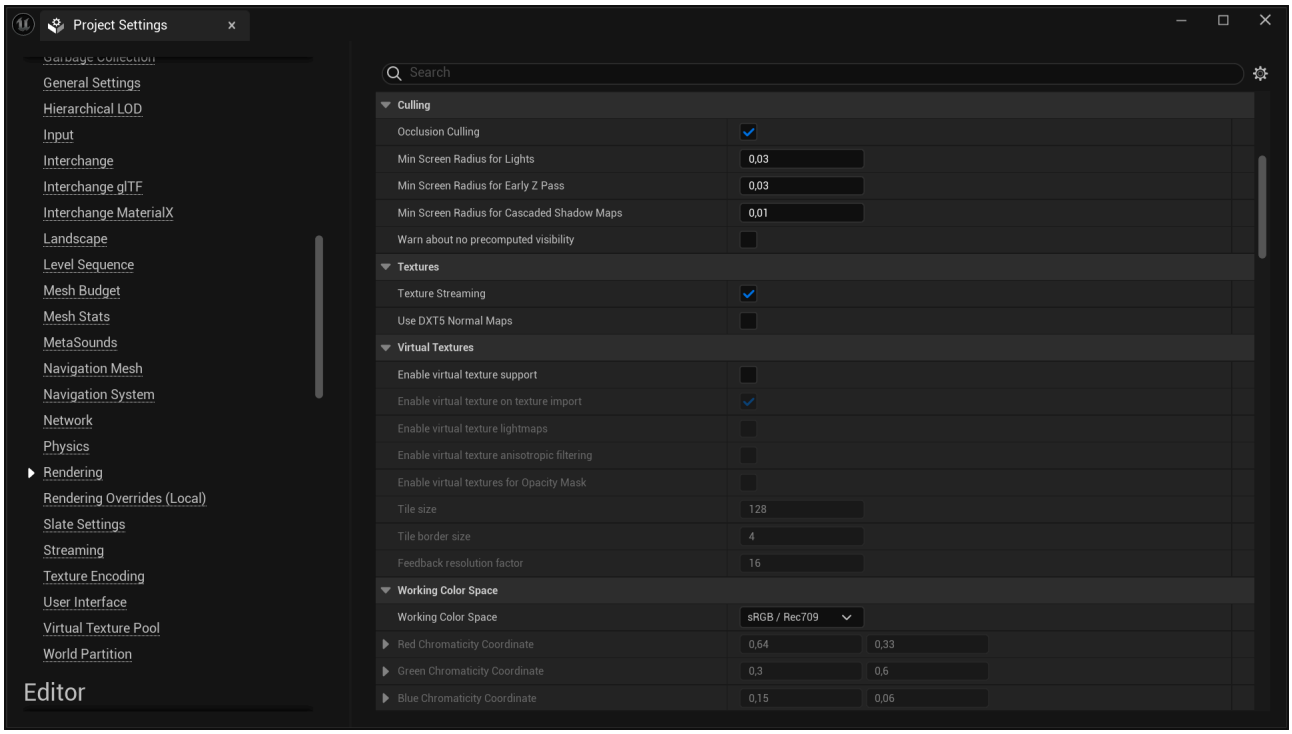
Rendering API

Vulkan vs OpenGL ES: Test your project with both Vulkan and OpenGL ES to see which provides better performance on your target devices. Vulkan often offers better performance but may not be supported on all devices.

Culling

Frustum Culling: Ensure that frustum culling is enabled to avoid rendering objects outside of the camera's view.

Occlusion Culling: Enable occlusion culling to avoid rendering objects that are not visible due to being blocked by other objects.


Project Settings
General Settings
Hierarchical LOD
Input
Interchange
Interchange glTF
Interchange MaterialX
Landscape
Level Sequence
Mesh Budget
Mesh Stats
MetaSounds
Navigation Mesh
Navigation System
Network
Physics
Rendering
Rendering Overrides (Local)
Slate Settings
Streaming
Texture Encoding
User Interface
Virtual Texture Pool
World Partition
Editor
Search
Culling
Occlusion Culling
Min Screen Radius for Lights
0,03
Min Screen Radius for Early Z Pass
0,03
Min Screen Radius for Cascaded Shadow Maps
0,01
Warn about no precomputed visibility
Textures
Texture Streaming
Use DXT5 Normal Maps
Virtual Textures
Enable virtual texture support
Enable virtual texture on texture import
Enable virtual texture lightmaps
Enable virtual texture anisotropic filtering
Enable virtual textures for Opacity Mask
Tile size
128
Tile border size
4
Feedback resolution factor
16
Working Color Space
Working Color Space
sRGB / Rec709
Red Chromaticity Coordinate
0,64
0,33
Green Chromaticity Coordinate
0,3
0,6
Blue Chromaticity Coordinate
0,15
0,06

Safe and Reliable Glove Access in Blueprint

Since the Blueprint API uses the underlying C++ API to access the SenseGlove hardware, it often has to deal with C++ pointers. Those who are familiar with C++ and in particular with the Unreal Engine UObject Garbage Collection System are aware that:

- As a general rule of thumb, a pointer should be validated before dereferenced, meaning before accessing the pointer a `NULL` check should be performed, otherwise if the pointer is `NULL` the program is going to crash upon access.
- Unreal implements a garbage collection scheme whereby UObjects that are no longer referenced or have been explicitly flagged for destruction will be cleaned up at regular intervals. The engine builds a reference graph to determine which UObjects are still in use and which ones are orphaned. The ones that are orphaned will be evaluated to `NULL` on the next GC cycle and their allocated memory will be released. Hence, `NULL` checks on UObjects are always mandatory.

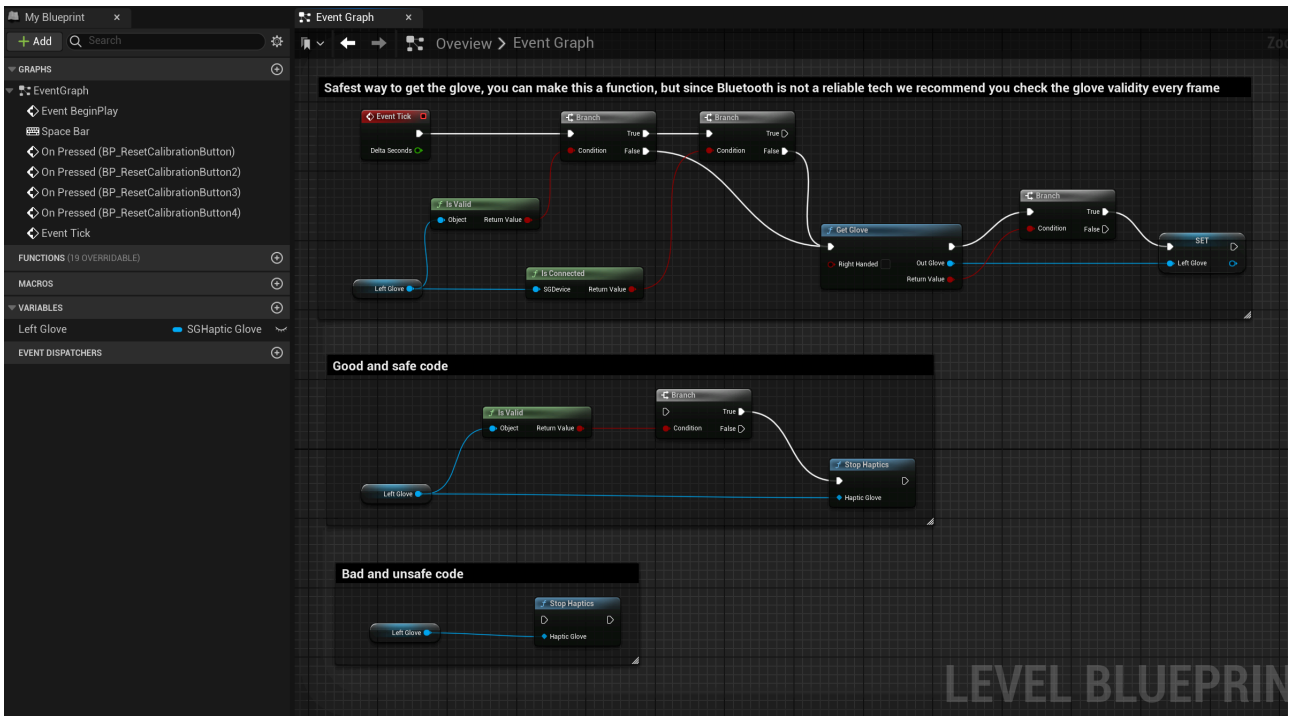
Glove objects inside the SenseGlove Unreal Engine Plugin, utilize the UObject system, and since communication for Nova gloves happens over SenseCom and the Bluetooth protocol, and also the gloves are running on battery, there's always the possibility for a glove variable to become `NULL` and therefore invalidated when the glove hardware for any reason is not accessible.

The recommended way to work with a glove instance without any performance penalty, and in a safe manner in Blueprint is:

1. Cache the glove instance inside a global variable if it passes certain tests so that you don't have to perform all those checks on every access. This usually could happen inside the `Tick` function.
2. The first check inside the `Tick` function is to check whether the cached glove instance is valid. If it's valid we continue to the next step, if not, we ask the API for a new glove instance.

3. If the glove instance is valid, then it's best to perform a connectivity check next. If the glove is connected we don't have to do anything else in regards to obtaining a new glove instance and caching it. If however the glove is not connected, we might ask the API for a new glove instance.
4. If any of the above steps fail, then we can actually ask the API for a new glove instance, and if the result is successful we're going to cache the new glove instance.
5. From here on, anywhere else inside your code, whenever you need to access the glove data or perform an operation like for example sending or stopping haptics you always perform a validity check and only proceed when the glove instance is valid. This way you will always ensure you are accessing the glove instances in a safe and reliable manner, thus avoiding any unexpected behaviors or crashes.

The following Blueprint examples implement the above approach and also demonstrate good and bad glove instance accesses:



OpenXR

The SenseGlove Unreal Engine Plugin has provided OpenXR-compatible hand tracking by implementing `XR_EXT_hand_tracking` since `v2.1.0`.

Typically a user does not need to know anything about OpenXR to use the plugin, so this section of the handbook is for advanced users who are looking for a way to directly consume the OpenXR data coming directly from either a SenseGlove device or if enabled in the plugin settings from hand-tracking.

Since the SenseGlove Unreal Engine Plugin registers itself as an `OpenXRHandTracking` motion controller device it becomes a hand-tracking provider for Unreal Engine, thus the OpenXR data from SenseGlove could always be retrieved from the Unreal Engine's `IXTrackingSystem` with one caveat. The caveat is if another OpenXR-compatible hand-tracking plugin, e.g. Epic's own OpenXRHandTracking, is enabled simultaneously it's not guaranteed that the `FXRMotionControllerData` and `FXRHandTrackingState` structs retrieved from the `IXTrackingSystem::GetMotionControllerData()` and `IXTrackingSystem::GetHandTrackingState()` methods are coming from SenseGlove, as these methods return the first hand-tracking plugin they could find. Thus, SenseGlove provides its own implementation of `GetMotionControllerData()` and `GetHandTrackingState()` which guarantee the retrieved `FXRMotionControllerData` or `FXRHandTrackingState` are coming from the SenseGlove Unreal Engine Plugin; and this is the preferred way to that.

> **Note**
>
> In order to retrieve the latest `FXRMotionControllerState` available, The SenseGlove Unreal Engine Plugin provides an alternative implementation for `IXTrackingSystem::GetMotionControllerState()` as well . However, since this method does not rely on the `OpenXRHandTracking` provider, it may become redundant. As a result, we might consider removing this functionality in future updates in favor of the one that Unreal Engine provides.

> **Important**

> Unreal Engine versions `5.2`, `5.3`, and `5.4` are limited to `FXRMotionControllerData` since at the time of their release no `FXRHandTrackingState` was available. Also please keep in mind that, while `FXRMotionControllerData` is pretty much usable and functional in Unreal Engine `5.5`, it is recommended to utilize `FXRHandTrackingState` instead. This is because this version of UE has deprecated `FXRMotionControllerData` in favor of the `FXRMotionControllerState` and `FXRHandTrackingState` structs. Prior to version `5.5`, `FXRMotionControllerData` handled both motion controller and hand tracking data. From `5.5` onward, these responsibilities have been separated into the two distinct structs, providing clearer and more specialized handling of each.

In the next sections we'll see:

- How we can directly consume the `FXRMotionControllerData` on UE `5.2`, `5.3`, `5.4`, and `5.5` to draw and animate debug virtual hands in both Blueprint and C++.
- How we can directly consume the `FXRHandTrackingState` on UE `5.5` to draw and animate debug virtual hands in both Blueprint and C++.
- The Consuming OpenXR Hand-Tracking Data tutorial series provides a comprehensive introduction to virtual reality, OpenXR hand-tracking, and gesture detection in Unreal Engine. Additionally, this tutorial series covers procedural virtual hand mesh animation using the OpenXR hand-tracking data.

Consuming FXRHandTrackingState

> **Important**
>
> Unreal Engine versions `5.2`, `5.3`, and `5.4` are limited to `FXRMotionControllerData` since at the time of their release no `FXRHandTrackingState` was available. Also please keep in mind that, while `FXRMotionControllerData` is pretty much usable and functional in Unreal Engine `5.5`, it is recommended to utilize `FXRHandTrackingState` instead. This is because this version of UE has deprecated `FXRMotionControllerData` in favor of the `FXRMotionControllerState` and `FXRHandTrackingState` structs. Prior to version `5.5`, `FXRMotionControllerData` handled both motion controller and hand-tracking data. From `5.5` onward, these responsibilities have been separated into the two distinct structs, providing clearer and more specialized handling of each.

Taking a closer look at the `FXRHandTrackingState` declaration inside the Unreal Engine's `HeadMountedDisplay` module at `Engine/Source/Runtime/HeadMountedDisplay/Public/HeadMountedDisplayTypes.h`, figuring out the data structure might not seem very straightforward:

```
USTRUCT(BlueprintType)
struct FXRHandTrackingState
{
    GENERATED_USTRUCT_BODY();

    // The state is valid if poses have ever been provided.
    UPROPERTY(BlueprintReadOnly, Category = "XR")
    bool bValid = false;
    UPROPERTY(BlueprintReadOnly, Category = "XR")
    FName DeviceName;
    UPROPERTY(BlueprintReadOnly, Category = "XR")
    FGuid ApplicationInstanceID;

    UPROPERTY(BlueprintReadOnly, Category = "XR")
    EXRSpaceType XRSpaceType = EXRSpaceType::UnrealWorldSpace;

    UPROPERTY(BlueprintReadOnly, Category = "XR")
    EControllerHand Hand = EControllerHand::Left;

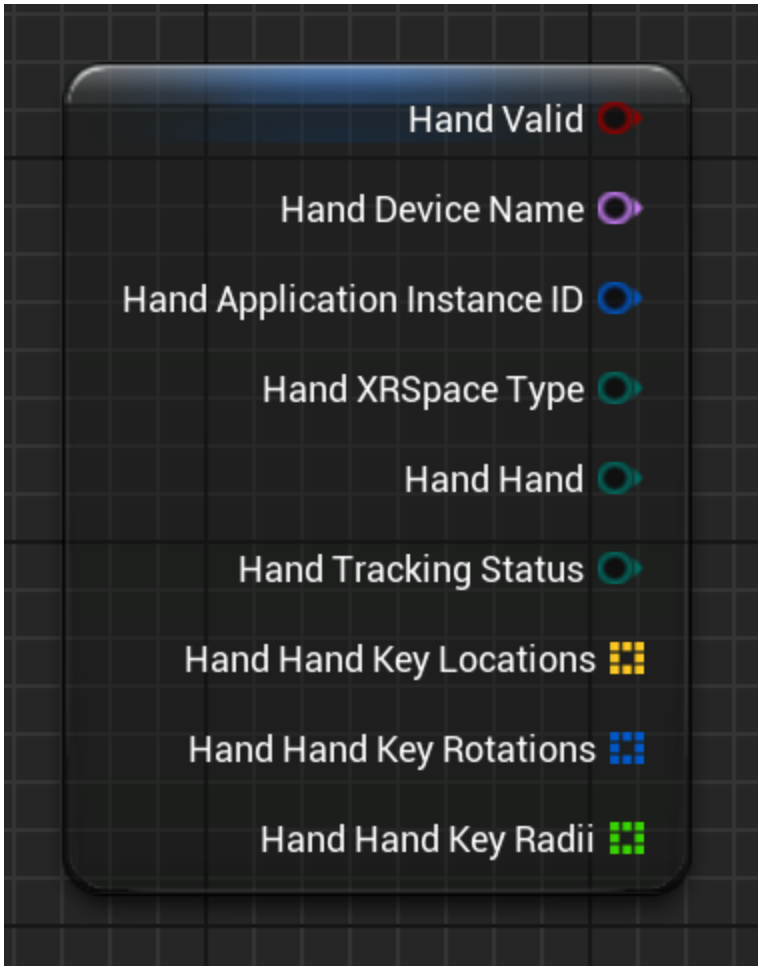
    UPROPERTY(BlueprintReadOnly, Category = "XR")
    ETrackingStatus TrackingStatus = ETrackingStatus::NotTracked;

    // The indices of this array are the values of EHandKeypoint (Palm,
Wrist, ThumbMetacarpal, etc).
    UPROPERTY(BlueprintReadOnly, Category = "XR")
    TArray<FVector> HandKeyLocations;

    // The indices of this array are the values of EHandKeypoint (Palm,
Wrist, ThumbMetacarpal, etc).
    UPROPERTY(BlueprintReadOnly, Category = "XR")
    TArray<FQuat> HandKeyRotations;

    // The indices of this array are the values of EHandKeypoint (Palm,
Wrist, ThumbMetacarpal, etc).
    UPROPERTY(BlueprintReadOnly, Category = "XR")
    TArray<float> HandKeyRadii;
};
```

Which on the Blueprint side it looks like this:



But, fear not, we've got you covered!

FXRHandTrackingState in Unreal Engine

`FXRHandTrackingState` is a structure in Unreal Engine designed to hold detailed information about the state of a hand-tracking device at a given moment. This structure is essential for handling hand-tracking inputs in virtual reality (VR) applications, providing the necessary data to accurately track and represent the user's hand movements and actions within the virtual environment.

Structure Members of FXRHandTrackingState

- **bValid**
  - **Description**: A boolean flag indicating whether the data is valid or not.
  - **Usage**: This is used to check if the motion controller data is correctly initialized and can be used for further processing.
- **DeviceName**
  - **Type**: `FName`
  - **Description**: The name of the device.
  - **Usage**: Identifies which device the data is coming from, useful when multiple devices are in use.
- **ApplicationInstanceID**
  - **Type**: `FString`
  - **Description**: A unique identifier for the application instance.
  - **Usage**: Helps in differentiating data from different instances of an application, ensuring the correct instance processes the data.
- **XRSpaceType**
  - **Type**: `EXRSpaceType`
  - **Description**: Enum specifying the type of XR space being used (e.g., unreal world or tracking space).
  - **Usage**: Specifies the coordinate system the XR Device is tracking itself in.
- **Hand**
  - **Type**: `EControllerHand`
  - **Description**: Enum indicating which hand is being tracked (left or right).
  - **Usage**: Helps identify whether the hand-tracking data pertains to the left or right hand, essential for hand-specific actions or interactions.
- **TrackingStatus**
  - **Type**: `EXRTrackingStatus`

    - **Description**: Enum indicating the tracking status of the hand-tracking device.
    - **Usage**: Shows whether the hand-tracking device is being tracked accurately, with possible statuses like `Tracked`, `NotTracked`, etc.

- **HandKeyLocations**

    - **Type**: `TArray<FVector>`
    - **Description**: An array of vectors representing key locations of the hand.
    - **Usage**: Provides detailed locations of key points on the hand, useful for precise hand-tracking and interaction.

- **HandKeyRotations**

    - **Type**: `TArray<FQuat>`
    - **Description**: An array of quaternions representing key rotations of the hand.
    - **Usage**: Complements the hand key locations with rotational data, ensuring accurate representation of hand movements.

- **HandKeyRadii**

    - **Type**: `TArray<float>`
    - **Description**: An array of floats representing the radii of key points of the hand.
    - **Usage**: Gives the size of the hand key points, aiding in collision detection and interaction fidelity.

Organization of FXRHandTrackingState

The structure is organized to encapsulate all relevant data needed for hand-tracking in a coherent and accessible manner. Boolean flag `bValid` provides quick checks on the state of the controller data. Identifiers `DeviceName` and `ApplicationInstanceID` ensure the correct association of data. Arrays `HandKeyLocations`, `HandKeyRotations`, and `HandKeyRadii` allow detailed hand-tracking, which is critical for immersive VR experiences. Lastly, the tracking status `TrackingStatus` informs the system of the

reliability of the data being processed and whether the hands are actively being tracked or they are inactive at the moment.

Processing the Data for Drawing and Animating a Virtual Hand

In order to draw and animate a virtual hand in real-time whether the data is coming from hand-tracking or a SenseGlove device, we could consume the data from the `HandKeyLocations` and `HandKeyRotations` fields of the `FXRHandTrackingState` struct.

Both `HandKeyLocations` and `HandKeyRotations` contain 26 elements as defined by OpenXR's `XR_HAND_JOINT_COUNT_EXT` and `XrHandJointLocationsEXT`, etc.

Unreal Engine also provides an enum called `EHandKeypoint` naming the 26 joints, and the equivalent of `XR_HAND_JOINT_COUNT_EXT` as `EHandKeypointCount` inside `Engine/Source/Runtime/HeadMountedDisplay/Public/HeadMountedDisplayTypes.h` as follows:

```
/**
 * Transforms that are tracked on the hand.
 * Matches the enums from WMR to make it a direct mapping
 */
UENUM(BlueprintType)
enum class EHandKeypoint : uint8
{
    Palm,
    Wrist,
    ThumbMetacarpal,
    ThumbProximal,
    ThumbDistal,
    ThumbTip,
    IndexMetacarpal,
    IndexProximal,
    IndexIntermediate,
    IndexDistal,
    IndexTip,
    MiddleMetacarpal,
    MiddleProximal,
    MiddleIntermediate,
    MiddleDistal,
    MiddleTip,
    RingMetacarpal,
    RingProximal,
    RingIntermediate,
    RingDistal,
    RingTip,
    LittleMetacarpal,
    LittleProximal,
    LittleIntermediate,
    LittleDistal,
    LittleTip
};

const int32 EHandKeypointCount = static_cast<int32>(EHandKeypoint::LittleTip)
+ 1;
```

So, getting the any joint's location or rotation is as easy as casting the enum value and passing it as the array index.

```
    FXRHandTrackingState HandTrackingState;
    const bool bGotHandTrackingState = FSGXRTracker::GetHandTrackingState(
        GetWorld(), EXRSpaceType::UnrealWorldSpace, EControllerHand::Left,
HandTrackingState);

    // Return if the struct data is invalid!
    if (!bGotHandTrackingState || !HandTrackingState.bValid)
    {
        return;
    }

    // Return if the device is not being tracked!
    if (HandTrackingState.TrackingStatus == ETrackingStatus::NotTracked)
    {
        return;
    }

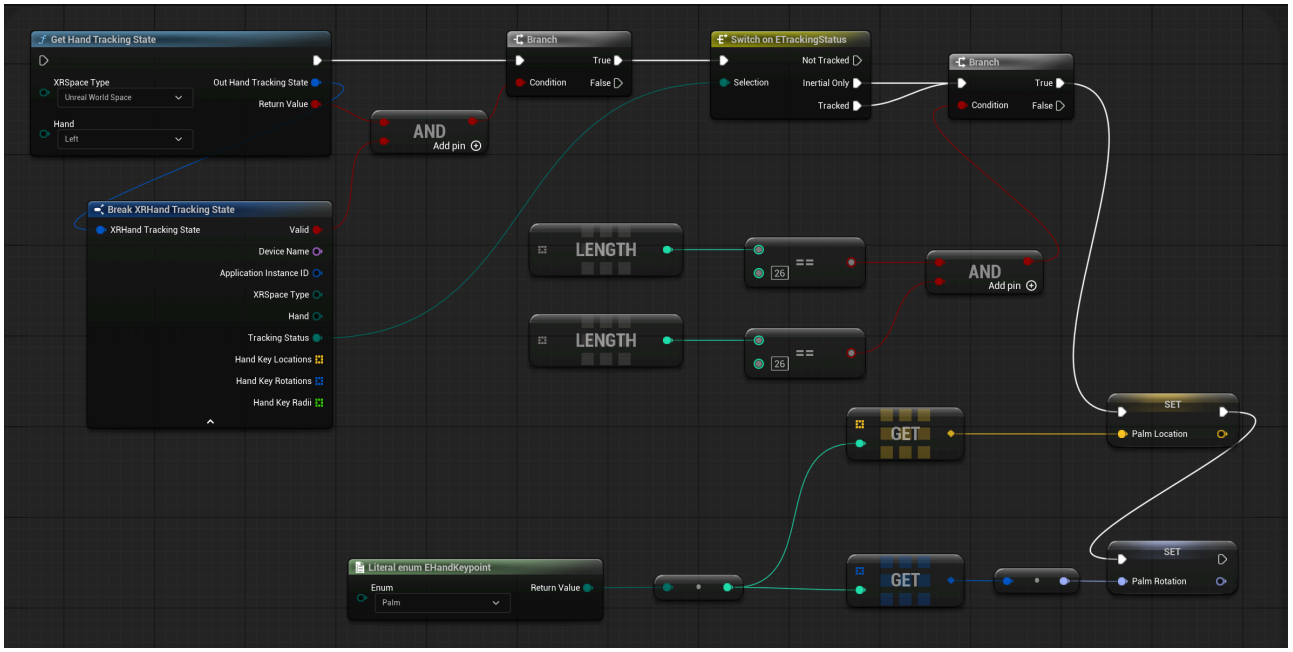
    // Ensure that HandTrackingState.HandKeyLocations has the location data
    // for 26 joints!
    if (!ensureAlwaysMsgf(HandTrackingState.HandKeyLocations.Num()
                          == EHandKeypointCount,
                          TEXT("Invalid HandKeyLocations count!")))
    {
        return;
    }

    // Ensure that HandTrackingState.HandKeyRotations has the rotation data
    // for 26 joints!
    if (!ensureAlwaysMsgf(HandTrackingState.HandKeyRotations.Num()
                          == EHandKeypointCount,
                          TEXT("Invalid HandKeyRotations count!")))
    {
        return;
    }

    static constexpr int32 PalmIndex = static_cast<int32>
(EHandKeypoint::Palm);

    const FVector& PalmLocation{
        HandTrackingState.HandKeyLocations[PalmIndex]
    };
    const FRotator& PalmRotation{
        HandTrackingState.HandKeyRotations[PalmIndex].Rotator()
    };
```

The equivalent Blueprint code for the above looks something like this:



OK, now that we've got a glimpse of how the virtual hand's joint data could be processed we are going to draw and animate a virtual hand in both Blueprint and C++ in the upcoming sections.

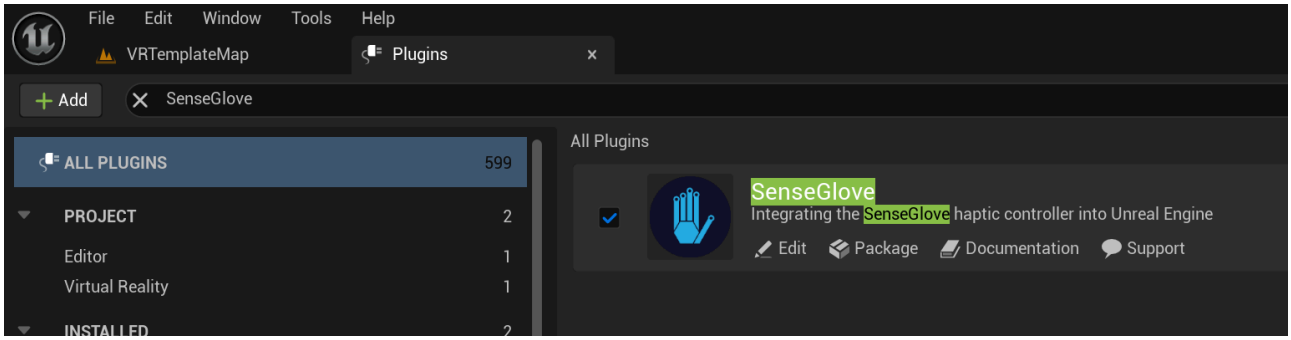
Consuming FXRHandTrackingState in Blueprint

> **Important**
>
> Unreal Engine versions `5.2`, `5.3`, and `5.4` are limited to `FXRMotionControllerData` since at the time of their release no `FXRHandTrackingState` was available. Also please keep in mind that, while `FXRMotionControllerData` is pretty much usable and functional in Unreal Engine `5.5`, it is recommended to utilize `FXRHandTrackingState` instead. This is because this version of UE has deprecated `FXRMotionControllerData` in favor of the `FXRMotionControllerState` and `FXRHandTrackingState` structs. Prior to version `5.5`, `FXRMotionControllerData` handled both motion controller and hand tracking data. From `5.5` onward, these responsibilities have been separated into the two distinct structs, providing clearer and more specialized handling of each.

Before continuing this section, please ensure you've studied the Consuming FXRHandTrackingState section, first.

Drawing and Animating Virtual Hands

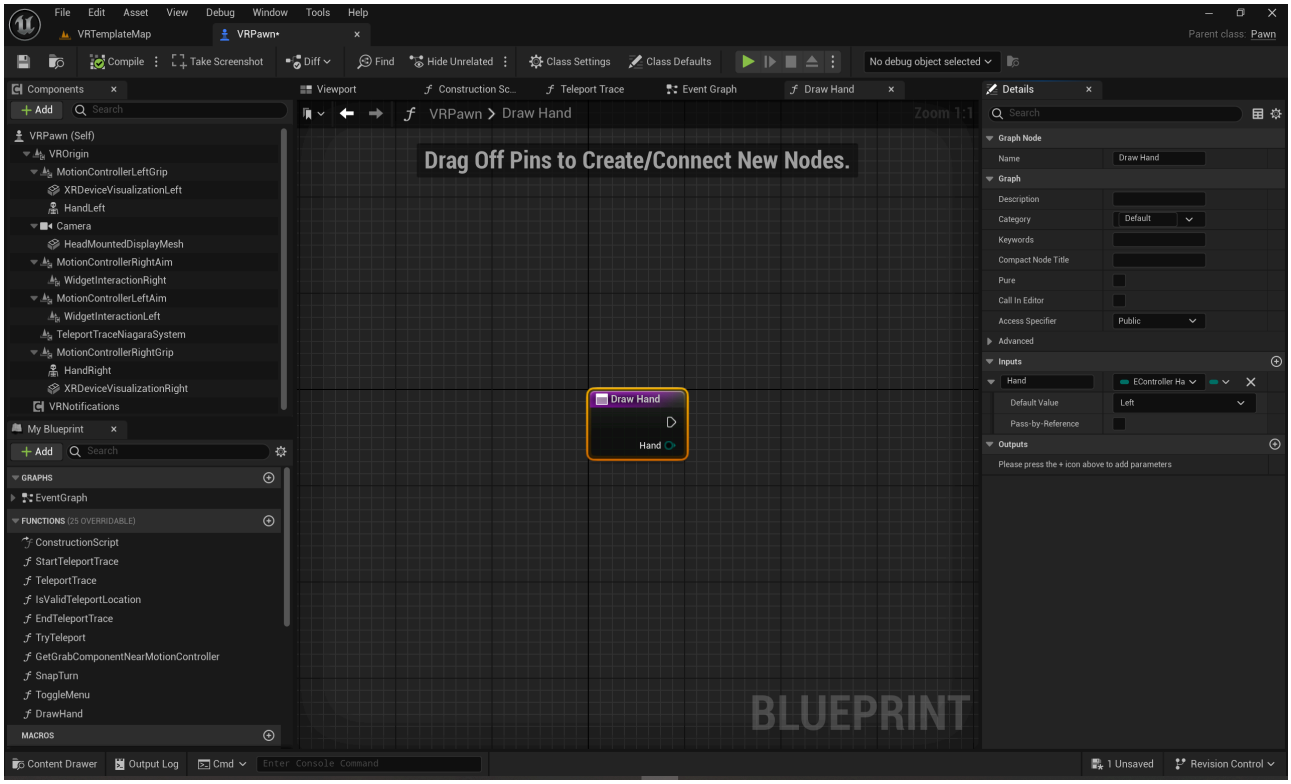
1. Create a new Virtual Reality project based the Unreal VR Template.
2. Make sure the SenseGlove UnrealEngine plugin is installed and enabled inside your new project.



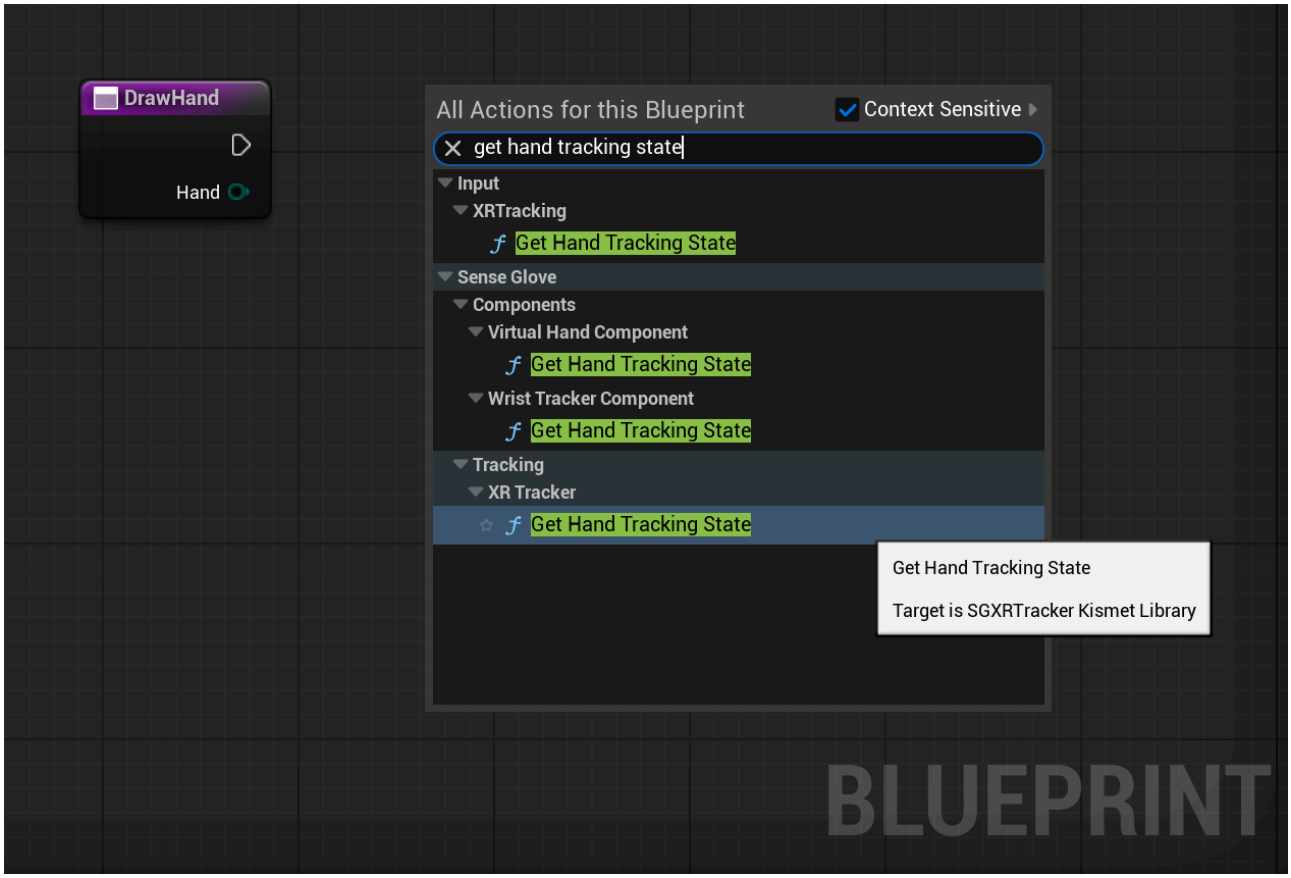
3. You could use either hand-tracking or a SenseGlove device as the input data, or both of the inside the same project. Whether you would like to use hand-tracking or a SenseGlove device, please make sure the required steps are taken for each of those first.

4. You could add the required Blueprint code for drawing virtual hands to either your Level Buleprint or the VRPawn Blueprint Class located at `/Content/VRTemplate/Blueprints/VRPawn`. In this guide we are going to add the code to our VRPawn.

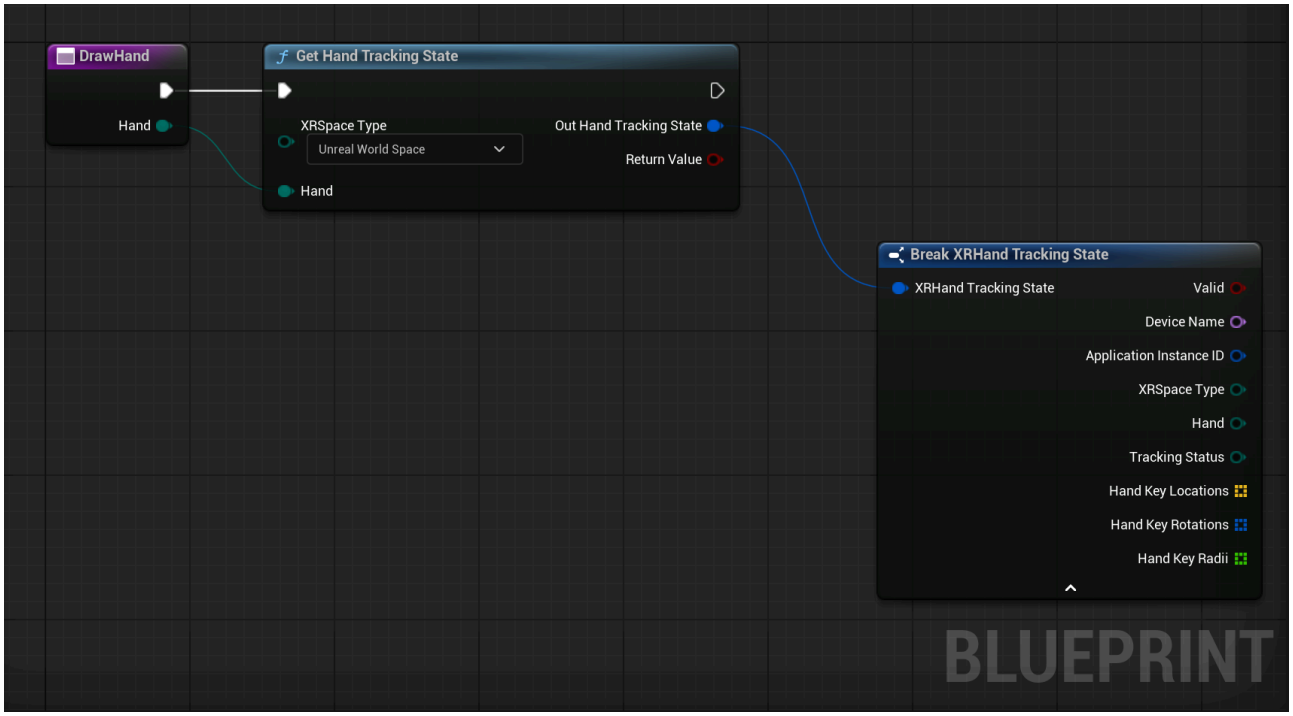
5. Add a new function named `Draw Hand` with an input parameter of type `EController Hand` named `Hand`.



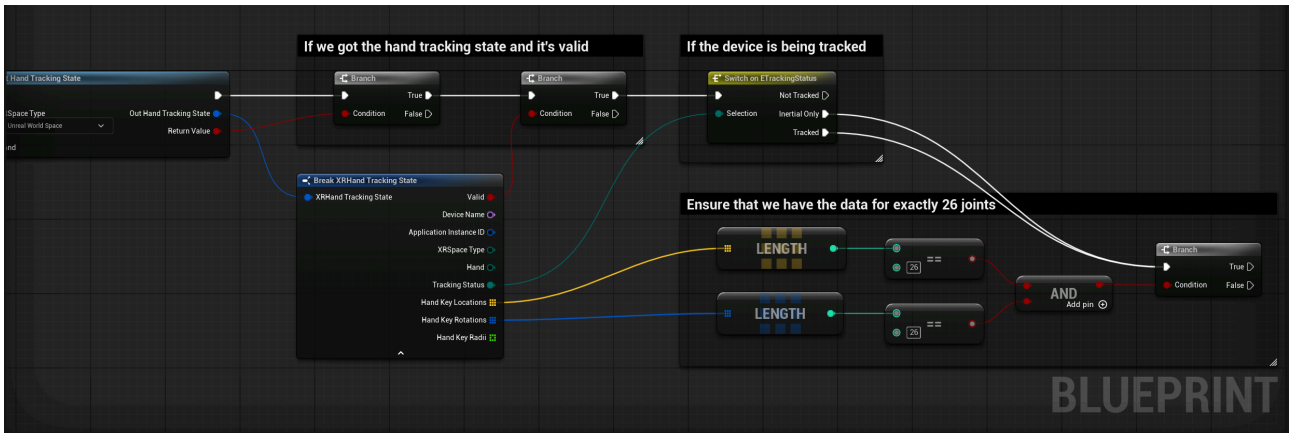
6. Inside this function's event graph add a `Get Hand Tracking State` node from `SenseGlove > Tracking > XR Tracker > Get Hand Tracking State`.



7. Then connect the functions `Hand` input parameter to the `Get Hand Tracking State` 's `Hand` input and right-click on the `OutHandTrackingState` parameter and use the `Break XRHandTrackingState` node to break the struct to it's fields.



8. After this, we need to perform data validation by checking the return status of the `Get Hand Tracking State` function and `FXRHandTrackingState's Valid` field. Then, we check if the hand-tracking device is being tracked and indeed coming from a hand-tracking source. And, finally, we check whether we have the positions and rotations for exactly `26` joints or not.



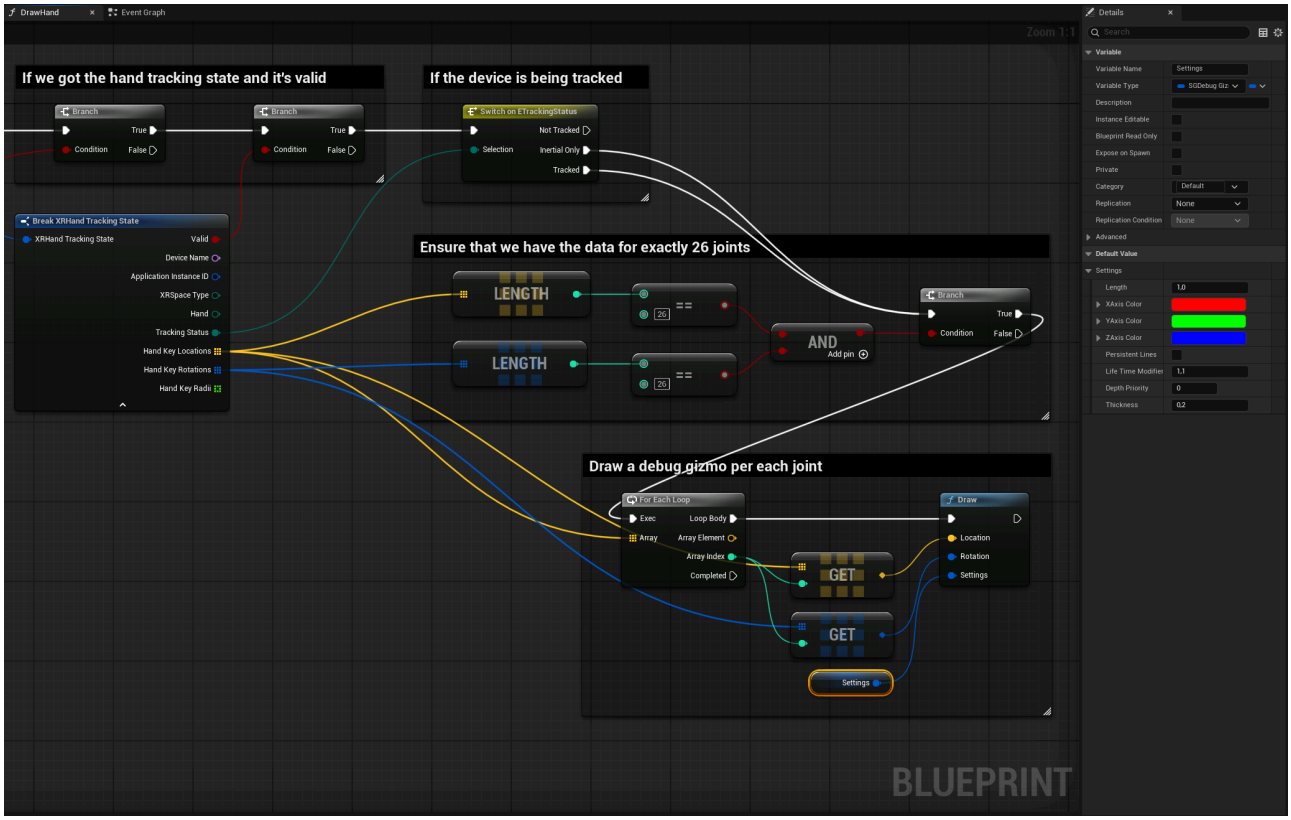
9. OK, now it's time to draw the joints! If we check out the SenseGlove Debug module's draw option, we notice there are various ways to draw the debug virtual hand. Drawing a cube or a gizmo per joint, or draw the whole hand all at once by passing the retrieved `FXRHandTrackingState` to the

`DebugVirtualHand::Draw` function! But, since the point of this tutorial is to learn how to consume the `FXRHandTrackingState` we ignore the last option. Between the debug cubes or gizmos, we are going to choose the gizmos since they better represent the rotations than the cubes.

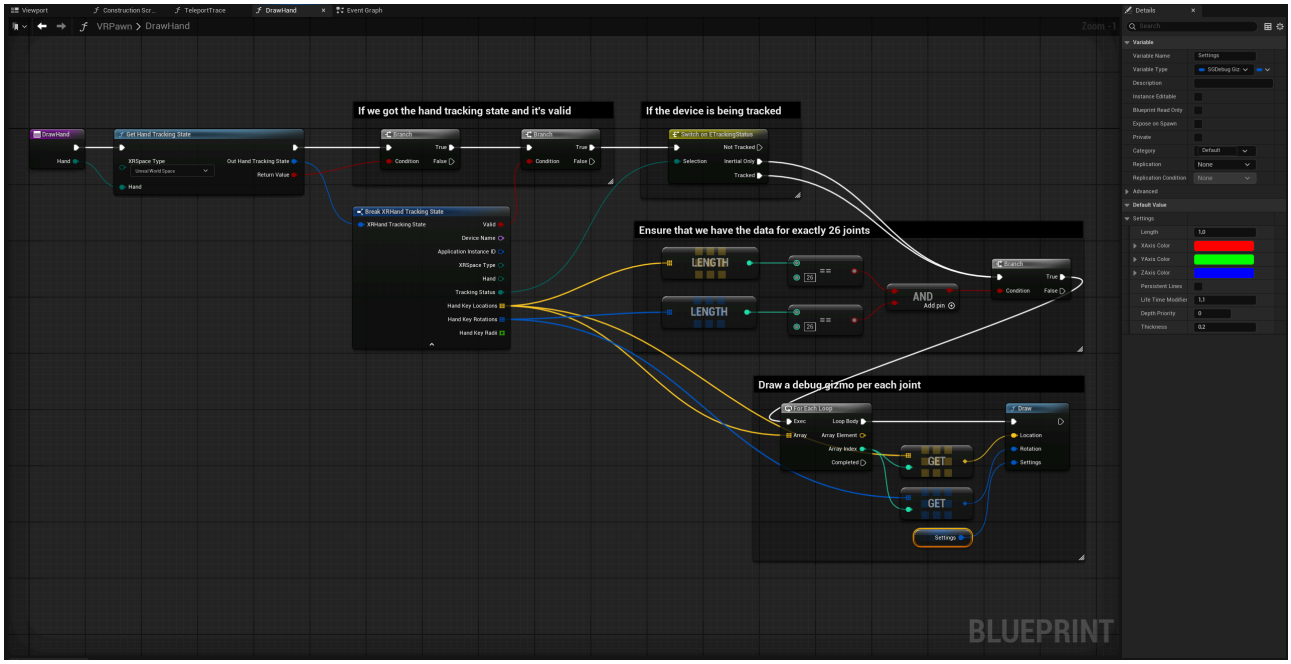


10. In the last step inside the `Draw Hand` function, in order to draw a virtual hand with `26` joints, we have to first iterate through either of the `Hand Key Positions` or `Hand Key Rotations` arrays from the `FXRHandTrackingState` struct.

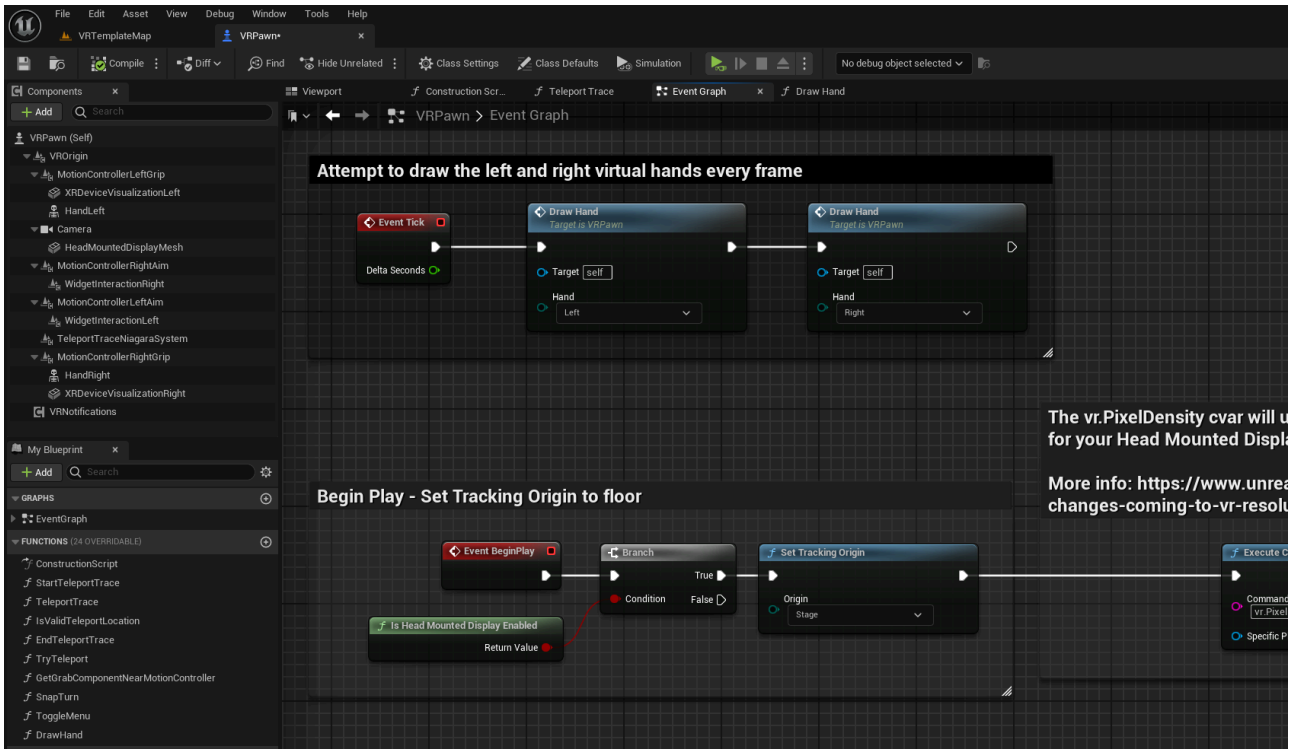
Since we made sure both arrays have `26` elements before we reached this step, it's safe to just iterate over one and use the `Array Index` inside a `For Each Loop` or a `For Loop` to access the position and rotation of every joint. Then we use each array `Get (a ref)` method to access the position and rotation data inside the loop and call the `Draw` function from `SenseGlove > Debug > Gizmo` per every joint. Please note that there are two `Draw` functions and the only difference between the two is that one accepts an `FQuat` and the other a `FRotator` for its `Rotation` input parameter. In this case, we use the `FQuat` variant to avoid an extra conversion to `FRotator`. Also, please adjust the `Thickness` option for the `Settings` parameter from `1.0` to `0.2`, as the default value might be too thick for drawing a joint gizmo.



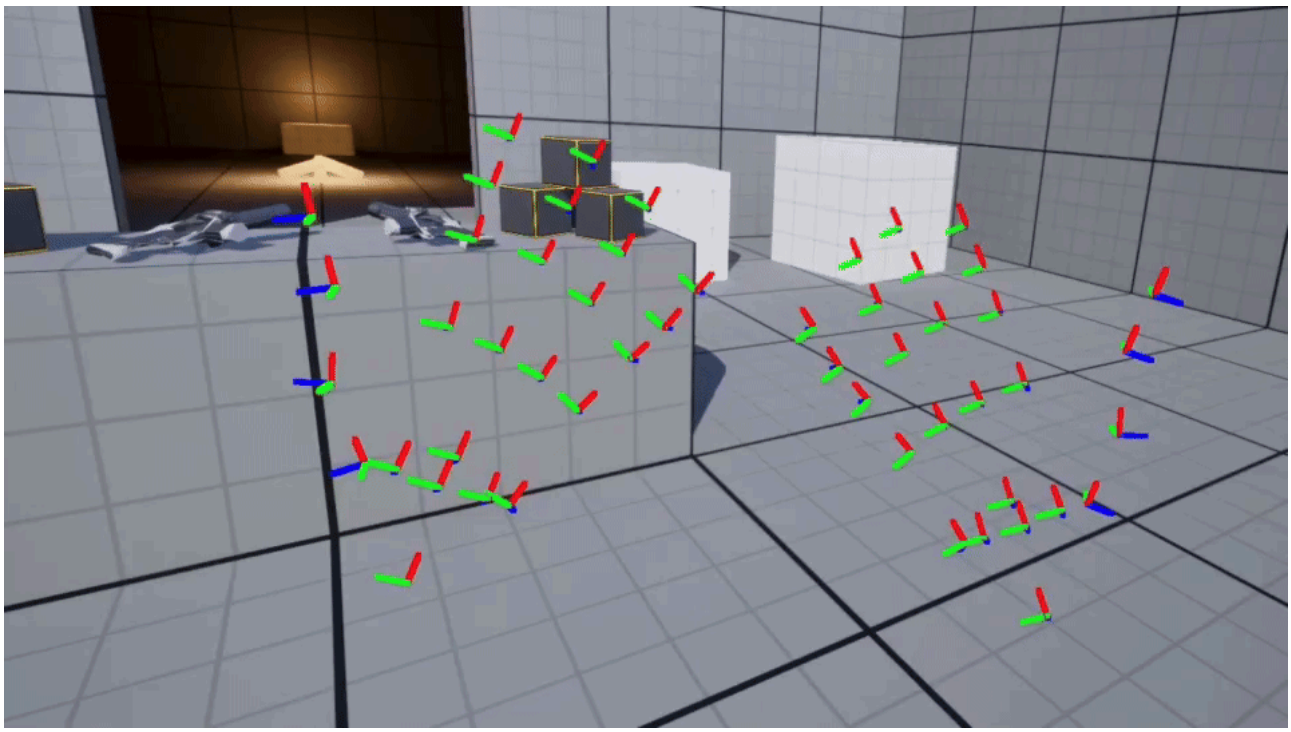
11. Well, now the full implementation for the `Draw Hand` function insde the `VRPawn` should look something like this:



12. Finally, go back to `VRPawn`'s event graph and the following code to the `Tick` event. Basically what we do here is call our newly implemented `Draw Hand` twice, once for each hand.



13. Now, go back to the `VRTemplateMap` and use the VR Preview button to run the game. If everything's done correctly, you should be able to see the virtual hands inside your VR simulation.



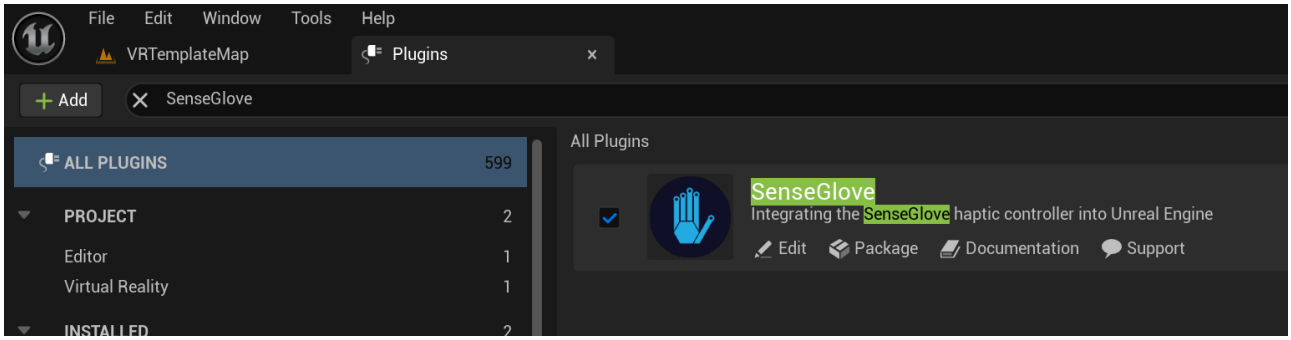
Consuming FXRHandTrackingState in C++

> **Important**
>
> Unreal Engine versions `5.2`, `5.3`, and `5.4` are limited to `FXRMotionControllerData` since at the time of their release no `FXRHandTrackingState` was available. Also please keep in mind that, while `FXRMotionControllerData` is pretty much usable and functional in Unreal Engine `5.5`, it is recommended to utilize `FXRHandTrackingState` instead. This is because this version of UE has deprecated `FXRMotionControllerData` in favor of the `FXRMotionControllerState` and `FXRHandTrackingState` structs. Prior to version `5.5`, `FXRMotionControllerData` handled both motion controller and hand tracking data. From `5.5` onward, these responsibilities have been separated into the two distinct structs, providing clearer and more specialized handling of each.

Before continuing this section, please ensure you've first studied the Consuming FXRHandTrackingState section.

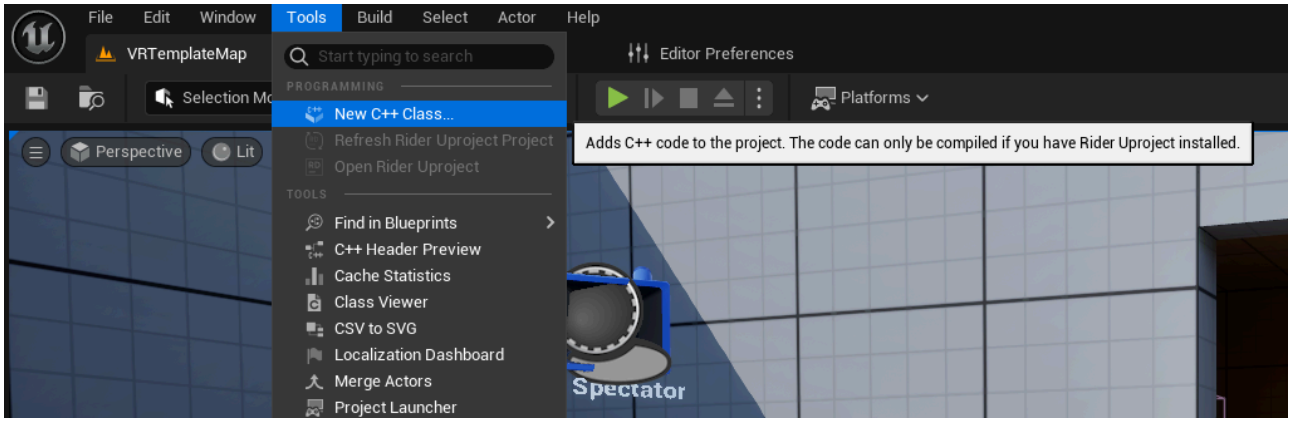
Drawing and Animating Virtual Hands

1. Create a new Virtual Reality project based the Unreal VR Template.
2. Make sure the SenseGlove UnrealEngine plugin is installed and enabled inside your new project.

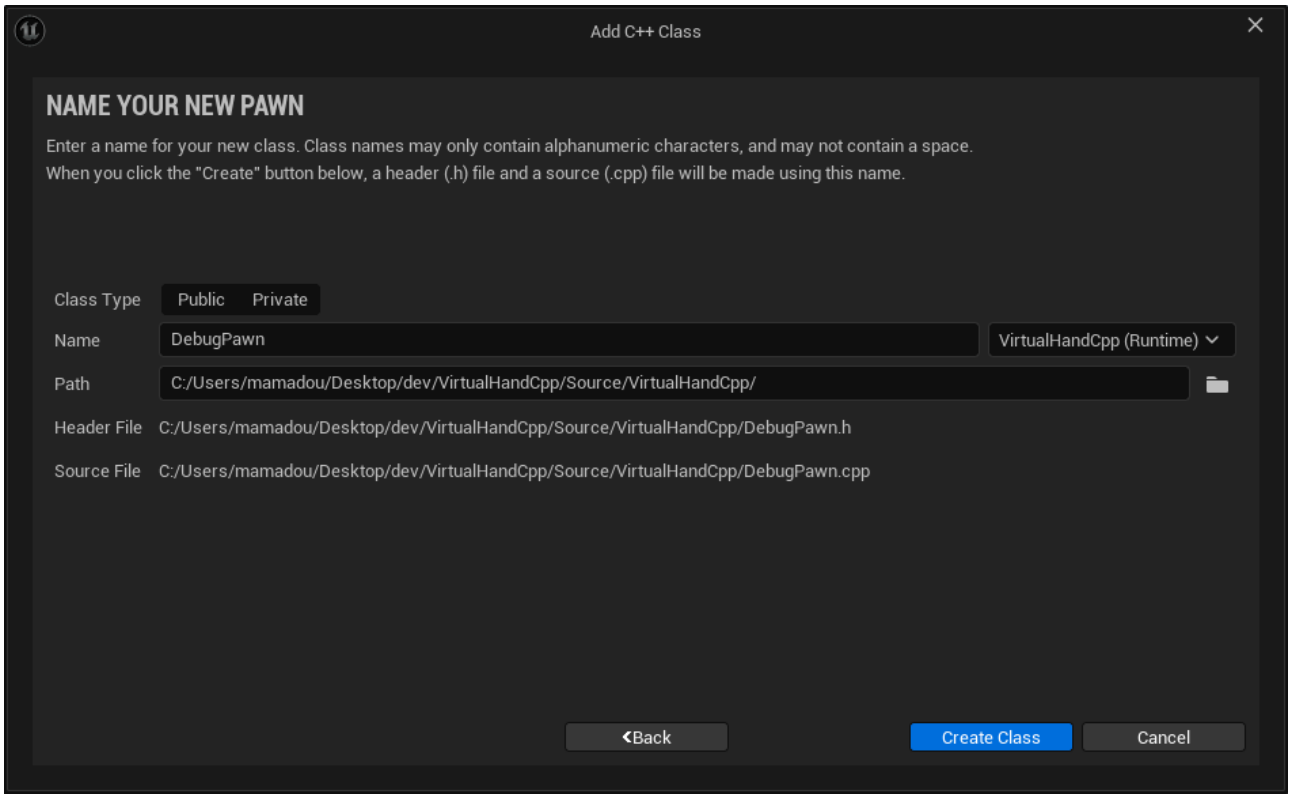


3. You could use either hand-tracking or a SenseGlove device as the input data, or both of the inside the same project. Whether you would like to use hand-tracking or a SenseGlove device, please make sure the required steps are taken for each of those first.

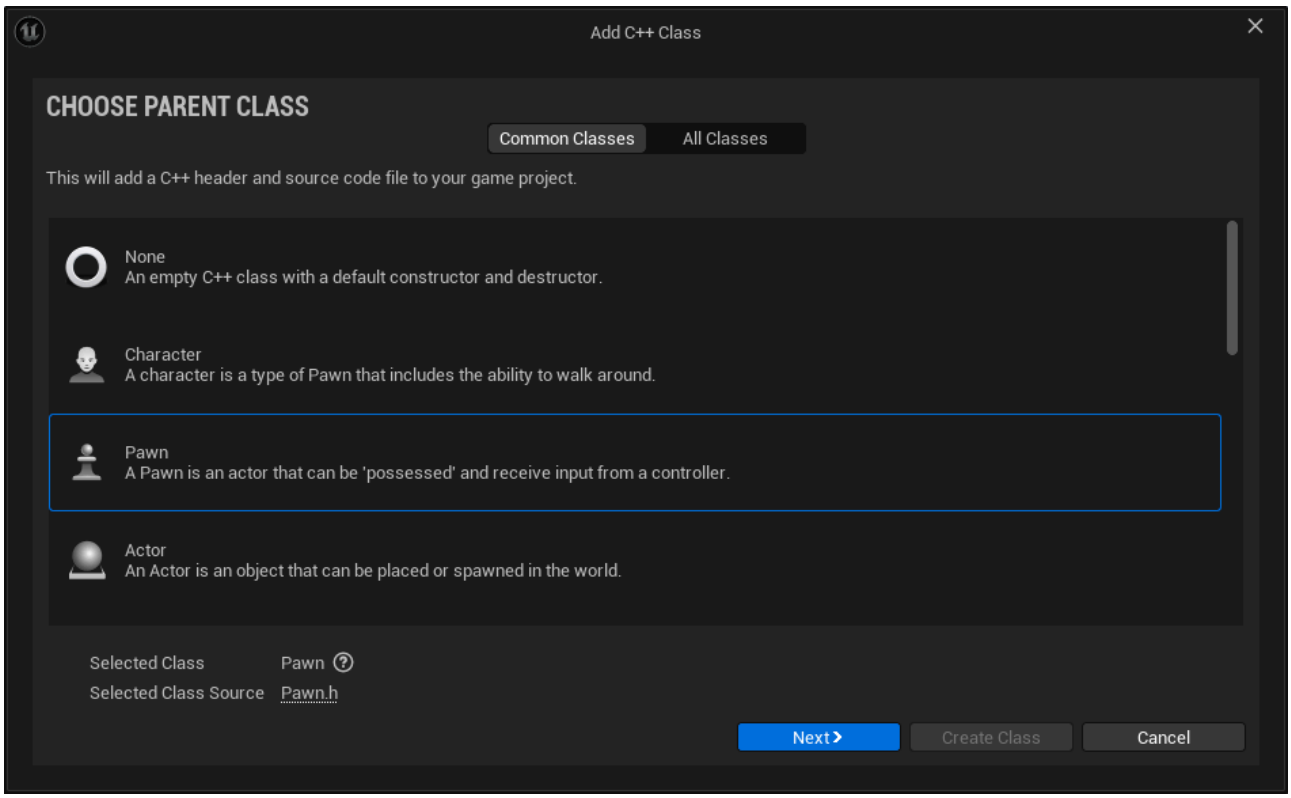
4. From the `Tools` menu choose `New C++ class...`.



5. Choose the Unreal Engine's `APawn` class as the parent class for the new C++ pawn class.

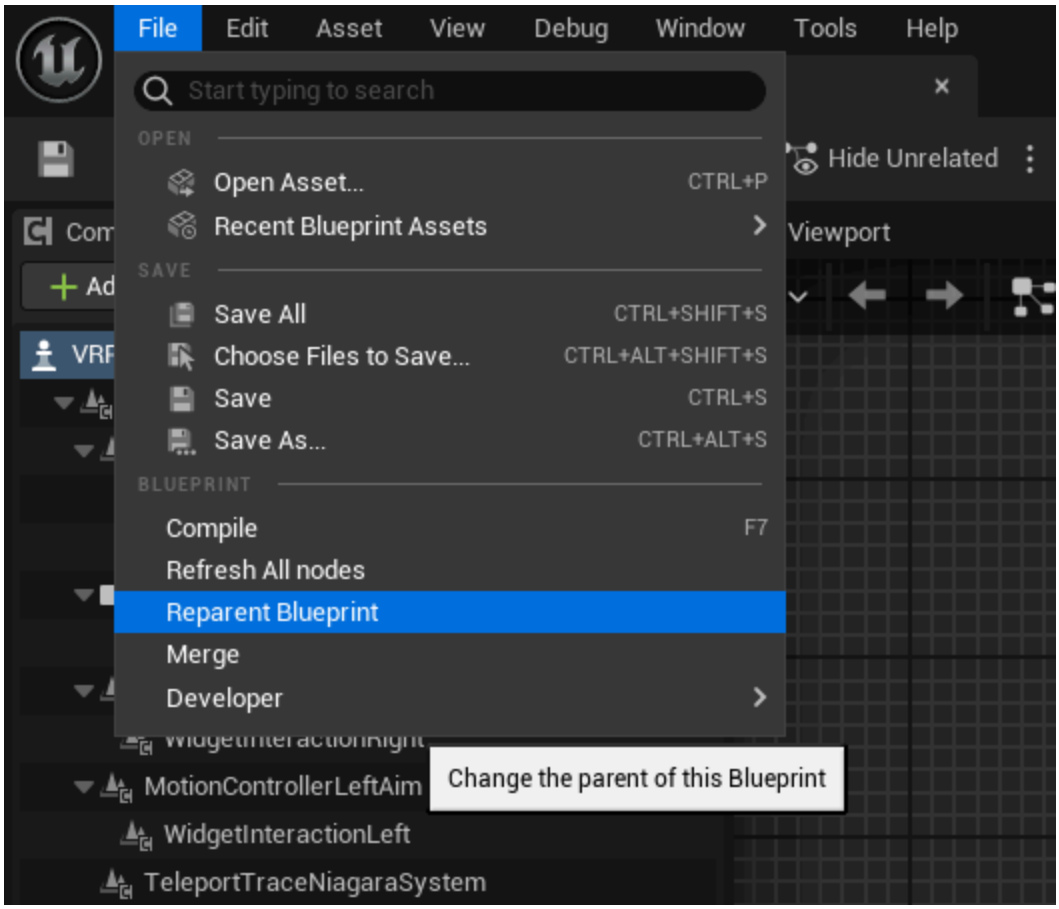


6. Name the new pawn class `DebugPawn`.



7. Since we have created a new C++ class, this converts the current Blueprint VRTemplateMap project to a C++ one. That's why the Unreal Editor will give us a few prompts regarding opening the project in the default IDE and rebuilding the code. It might be simpler to just close the editor, then rebuild the source code inside your favorite IDE, and then start the editor with the converted project again.

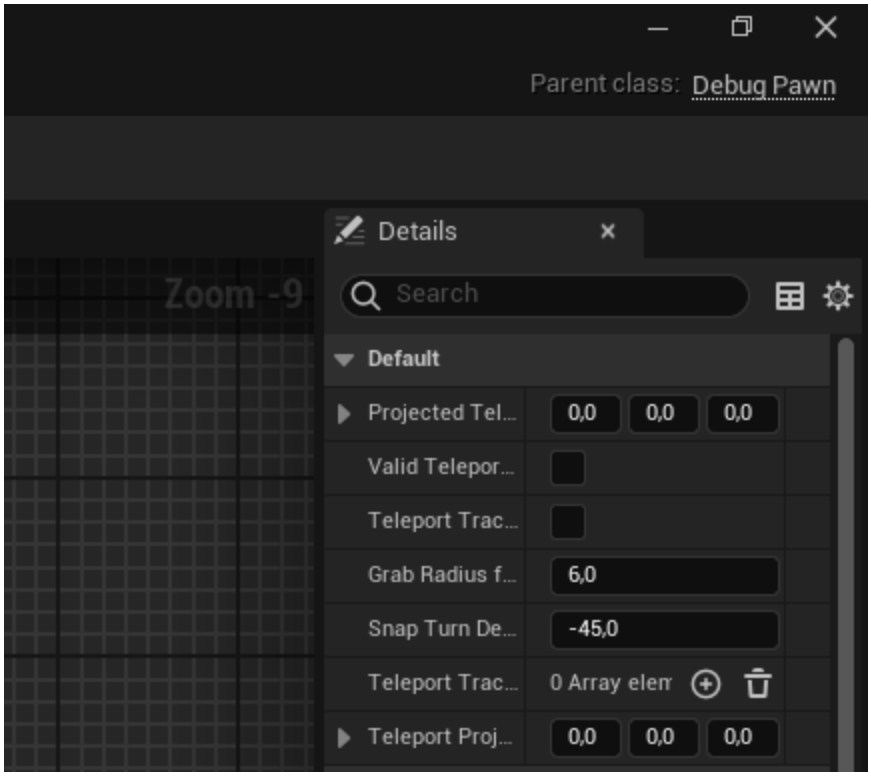
8. Find and open the VRPawn Blueprint Class located at `/Content/VRTemplate/Blueprints/VRPawn` inside the Blueprint Editor and from the `File` menu choose the `Reparent Blueprint` class.



9. In the new `Reparent blueprint` window choose `DebugPawn` as the new parent.



10. By looking at the `Parent Class` label located under the Blueprint Editor window control buttons verify that the `ADebugPawn` class has been set as the new parent.



11. Locate the project's main Build file, in our case `VirtualHandCpp/Source/VirtualHandCpp/VirtualHandCpp.Build.cs` and add the `InputDevice`, `OpenXRHMD`, `SenseGloveBuildHacks`, `SenseGloveDebug`, `SenseGloveSettings`, and `SenseGloveTracking` modules as either a private or public dependency.

```
// Fill out your copyright notice in the Description page of Project
Settings.

using UnrealBuildTool;

public class VirtualHandCpp : ModuleRules
{
    public VirtualHandCpp(ReadOnlyTargetRules Target) : base(Target)
    {
        PCHUsage = PCHUsageMode.UseExplicitOrSharedPCHs;

        PublicDependencyModuleNames.AddRange(new string[] { "Core",
"CoreUObject", "Engine", "InputCore" });

        PrivateDependencyModuleNames.AddRange(new string[]
        {
            "InputDevice",
            "OpenXRHMD",
            "SenseGloveBuildHacks",
            "SenseGloveDebug",
            "SenseGloveSettings",
            "SenseGloveTracking"
        });

        // Uncomment if you are using Slate UI
        // PrivateDependencyModuleNames.AddRange(new string[] { "Slate",
"SlateCore" });

        // Uncomment if you are using online features
        // PrivateDependencyModuleNames.Add("OnlineSubsystem");

        // To include OnlineSubsystemSteam, add it to the plugins section in
your uproject file with the Enabled attribute set to true
    }
}
```

12. Locate the C++ header and source file for the `ADebugPawn` inside the project in your C++ IDE. In our case they are located at `VirtualHandCpp/Source/VirtualHandCpp/DebugPawn.h` and `VirtualHandCpp/Source/VirtualHandCpp/DebugPawn.cpp`.

13. Modify the `DebugPawn.h` header file to look like this:

```
// Fill out your copyright notice in the Description page of Project
Settings.

#pragma once

#include "CoreMinimal.h"
#include "GameFramework/Pawn.h"

#include "SGSettings/SGDebugGizmoSettings.h"

#include "DebugPawn.generated.h"

UCLASS()
class VIRTUALHANDCPP_API ADebugPawn : public APawn
{
    GENERATED_BODY()

private:
    // The virtual hand drawing settings.
    UPROPERTY(EditDefaultsOnly, Category="DebugPawn",
        meta=(AllowPrivateAccess="false"))
    FSGDebugGizmoSettings HandDrawingSettings;

public:
    // Sets default values for this pawn's properties
    ADebugPawn();

protected:
    // Called when the game starts or when spawned
    virtual void BeginPlay() override;

public:
    // Called every frame
    virtual void Tick(float DeltaTime) override;

    // Called to bind functionality to input
    virtual void SetupPlayerInputComponent(class UInputComponent*
PlayerInputComponent) override;

private:
    // The method responsible for drawing a virtual hand.
    void DrawHand(EControllerHand Hand) const;
};
```

14. Modify the `DebugPawn.cpp` implementation file to look like this:

```
// Fill out your copyright notice in the Description page of Project
Settings.


#include "DebugPawn.h"

#include "SGDebug/SGDebugGizmo.h"
#include "SGTracking/SGXRTracker.h"

// Sets default values
ADebugPawn::ADebugPawn()
{
    // Set this pawn to call Tick() every frame.  You can turn this off to
improve performance if you don't need it.
    PrimaryActorTick.bCanEverTick = true;

    // Set the default virtual hand drawing settings.
    HandDrawingSettings = FSGDebugGizmoSettings{
        1.0f,
        FColor{255, 0, 0, 255},
        FColor{0, 255, 0, 255},
        FColor{0, 0, 255, 255},
        false,
        1.1f,
        0,
        0.2f,
    };
}

// Called when the game starts or when spawned
void ADebugPawn::BeginPlay()
{
    Super::BeginPlay();
}

// Called every frame
void ADebugPawn::Tick(float DeltaTime)
{
    Super::Tick(DeltaTime);

    // Attempt at drawing the left/right virtual hands every frame.
    DrawHand(EControllerHand::Left);
    DrawHand(EControllerHand::Right);
}

// Called to bind functionality to input
void ADebugPawn::SetupPlayerInputComponent(UInputComponent*
```

```
PlayerInputComponent)
{
    Super::SetupPlayerInputComponent(PlayerInputComponent);
}

void ADebugPawn::DrawHand(const EControllerHand Hand) const
{
    // Get the world and cache it, if it's null we return early.
    UWorld* World{GetWorld()};
    if (!IsValid(World))
    {
        return;
    }

    FXRHandTrackingState HandTrackingState;
    const bool bGotHandTrackingState = FSGXRTracker::GetHandTrackingState(
        World, EXRSpaceType::UnrealWorldSpace, Hand, HandTrackingState);

    // Return if the struct data is invalid!
    if (!bGotHandTrackingState || !HandTrackingState.bValid)
    {
        return;
    }

    // Return if the device is not being tracked!
    if (HandTrackingState.TrackingStatus == ETrackingStatus::NotTracked)
    {
        return;
    }

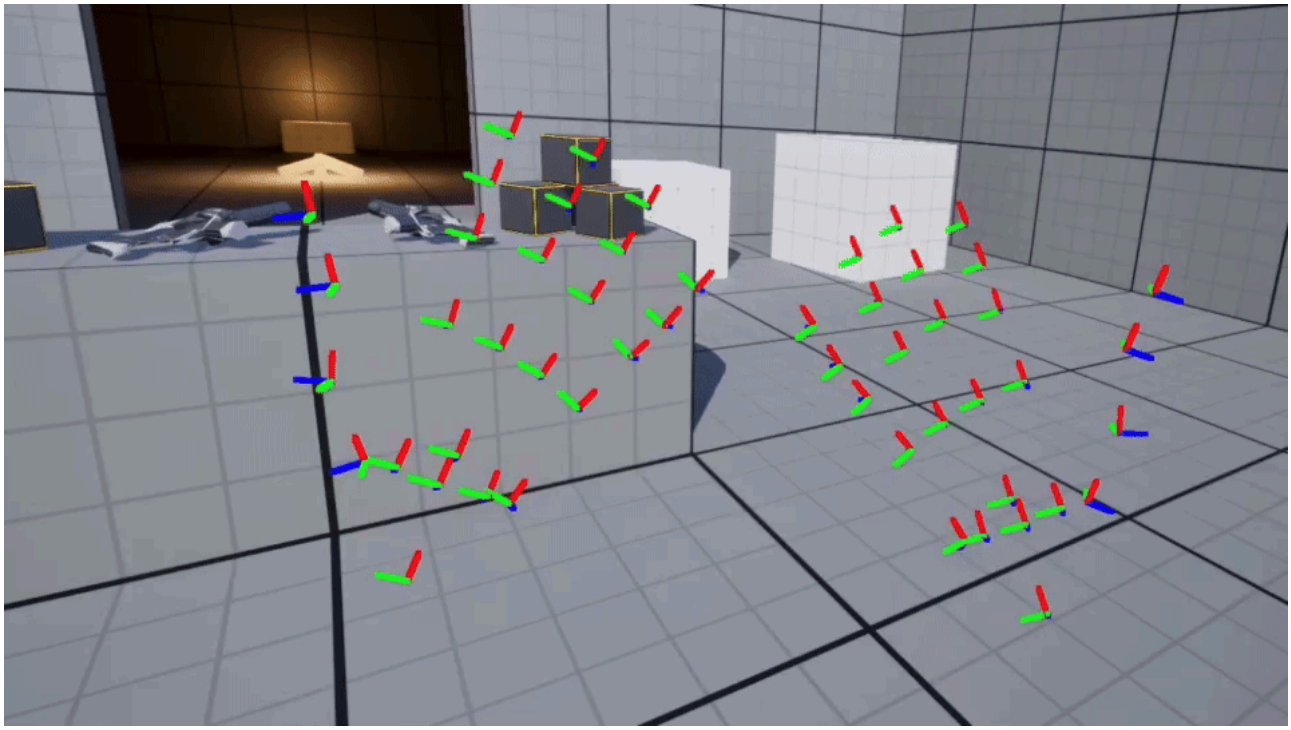
    // Ensure that HandTrackingState.HandKeyLocations has the location data
    // for 26 joints!
    if (!ensureAlwaysMsgf(HandTrackingState.HandKeyLocations.Num()
                          == EHandKeypointCount,
                          TEXT("Invalid HandKeyLocations count!")))
    {
        return;
    }

    // Ensure that HandTrackingState.HandKeyRotations has the rotation data
    // for 26 joints!
    if (!ensureAlwaysMsgf(HandTrackingState.HandKeyRotations.Num()
                          == EHandKeypointCount,
                          TEXT("Invalid HandKeyRotations count!")))
    {
        return;
    }
```

```
    // Iterate over the hand joint locations and rotations!
    for (int32 JointIndex = 0; JointIndex < EHandKeypointCount; ++JointIndex)
    {
        const FVector& JointLocation{
            HandTrackingState.HandKeyLocations[JointIndex]
        };
        const FQuat& JointRotation{
            HandTrackingState.HandKeyRotations[JointIndex]
        };

        // Draw a single joint's gizmo!
        // Please note that we could alternatively:
        // Use FSGDebugCube::Draw() to draw a cube.
        // Or use the FSGDebugVirtualHand::Draw() method and pass the
        // HandTrackingState directly to draw the virtual hand
        // all at once without iterating the joints. But, that's not
        // goal of this tutorial.
        FSGDebugGizmo::Draw(World, JointLocation, JointRotation,
HandDrawingSettings);
    }
}
```

15. Now, rebuild the source code and go back to the `VRTemplateMap`, then use the VR Preview button to run the game. If everything's done correctly, you should be able to see the virtual hands inside your VR simulation.



Consuming FXRMotionControllerData

> **Important**
>
> Unreal Engine versions `5.2`, `5.3`, and `5.4` are limited to `FXRMotionControllerData` since at the time of their release no `FXRHandTrackingState` was available. Also please keep in mind that, while `FXRMotionControllerData` is pretty much usable and functional in Unreal Engine `5.5`, it is recommended to utilize `FXRHandTrackingState` instead. This is because this version of UE has deprecated `FXRMotionControllerData` in favor of the `FXRMotionControllerState` and `FXRHandTrackingState` structs. Prior to version `5.5`, `FXRMotionControllerData` handled both motion controller and hand tracking data. From `5.5` onward, these responsibilities have been separated into the two distinct structs, providing clearer and more specialized handling of each.

Taking a closer look at the `FXRMotionControllerData` declaration inside the Unreal Engine's `HeadMountedDisplay` module at `Engine/Source/Runtime/HeadMountedDisplay/Public/HeadMountedDisplayTypes.h`, figuring out the data structure might not seem very straightforward:

```
USTRUCT(BlueprintType)
struct FXRMotionControllerData
{
    GENERATED_USTRUCT_BODY();

    UPROPERTY(BlueprintReadOnly, Category = "XR")
    bool bValid = false;
    UPROPERTY(BlueprintReadOnly, Category = "XR")
    FName DeviceName;
    UPROPERTY(BlueprintReadOnly, Category = "XR")
    FGuid ApplicationInstanceID;
    UPROPERTY(BlueprintReadOnly, Category = "XR")
    EXRVisualType DeviceVisualType = EXRVisualType::Controller;

    UPROPERTY(BlueprintReadOnly, Category = "XR")
    EControllerHand HandIndex = EControllerHand::Left;

    UPROPERTY(BlueprintReadOnly, Category = "XR")
    ETrackingStatus TrackingStatus = ETrackingStatus::NotTracked;

    // Vector representing an object being held in the player's hand
    UPROPERTY(BlueprintReadOnly, Category = "XR")
    FVector GripPosition = FVector(0.0f);
    // Quaternion representing an object being held in the player's hand
    UPROPERTY(BlueprintReadOnly, Category = "XR")
    FQuat GripRotation = FQuat(EForceInit::ForceInitToZero);

    // For handheld controllers, gives a vector for pointing at objects
    UPROPERTY(BlueprintReadOnly, Category = "XR")
    FVector AimPosition = FVector(0.0f);
    // For handheld controllers, gives a quaternion for pointing at objects
    UPROPERTY(BlueprintReadOnly, Category = "XR")
    FQuat AimRotation = FQuat(EForceInit::ForceInitToZero);

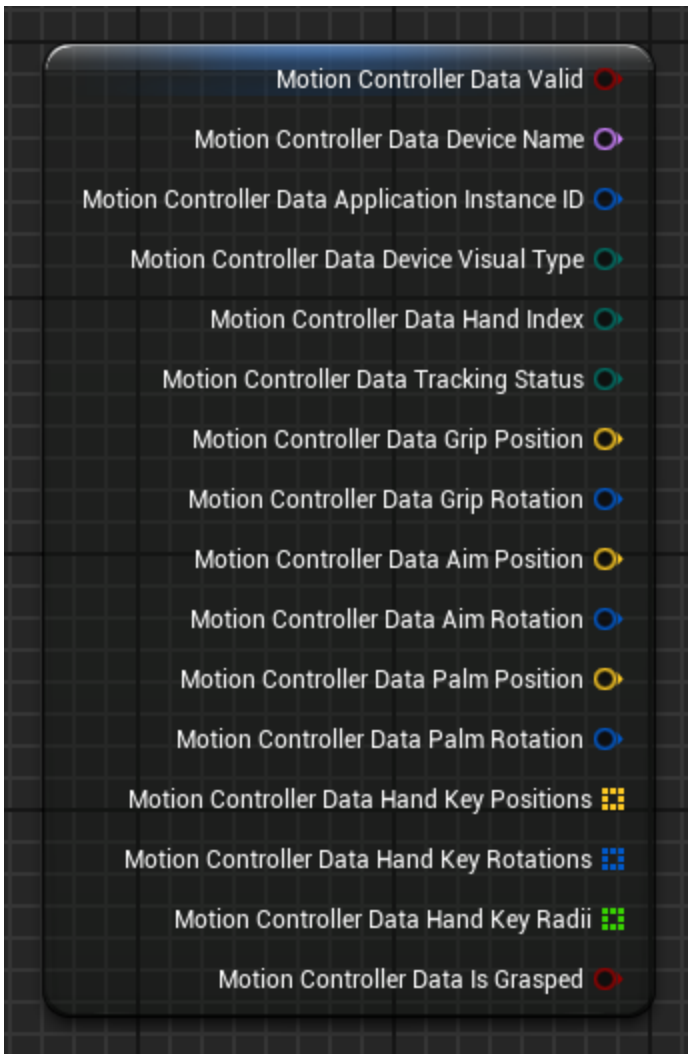
    // For handheld controllers, gives a vector for representing the hand
    UPROPERTY(BlueprintReadOnly, Category = "XR")
    FVector PalmPosition = FVector(0.0f);
    // For handheld controllers, gives a quaternion for representing the hand
    UPROPERTY(BlueprintReadOnly, Category = "XR")
    FQuat PalmRotation = FQuat(EForceInit::ForceInitToZero);

    // The indices of this array are the values of EHandKeypoint (Palm,
Wrist, ThumbMetacarpal, etc).
    UPROPERTY(BlueprintReadOnly, Category = "XR")
    TArray<FVector> HandKeyPositions;
    // The indices of this array are the values of EHandKeypoint (Palm,
Wrist, ThumbMetacarpal, etc).
```

```
    UPROPERTY(BlueprintReadOnly, Category = "XR")
    TArray<FQuat> HandKeyRotations;
    // The indices of this array are the values of EHandKeypoint (Palm,
Wrist, ThumbMetacarpal, etc).
    UPROPERTY(BlueprintReadOnly, Category = "XR")
    TArray<float> HandKeyRadii;

    UPROPERTY(BlueprintReadOnly, Category = "XR")
    bool bIsGrasped = false;
};
```

Which on the Blueprint side it looks like this:



But, fear not, we've got you covered!

FXRMotionControllerData in Unreal Engine

`FXRMotionControllerData` is a structure in Unreal Engine designed to hold detailed information about the state of a motion controller device at a given moment. This structure is essential for handling motion controller inputs in virtual reality (VR) applications, providing the necessary data to accurately track and represent the user's hand movements and actions within the virtual environment.

Structure Members of FXRMotionControllerData

- **bValid**
    - **Description**: A boolean flag indicating whether the data is valid or not.
    - **Usage**: This is used to check if the motion controller data is correctly initialized and can be used for further processing.
- **DeviceName**
    - **Type**: `FName`
    - **Description**: The name of the device.
    - **Usage**: Identifies which motion controller device the data is coming from, useful when multiple devices are in use.
- **ApplicationInstanceID**
    - **Type**: `FString`
    - **Description**: A unique identifier for the application instance.
    - **Usage**: Helps in differentiating data from different instances of an application, ensuring the correct instance processes the data.
- **DeviceVisualType**
    - **Type**: `EXRVisualType`
    - **Description**: Enum specifying the visual type of the device (e.g., controller, hand).
    - **Usage**: Used to differentiate between various motion controller devices or hand-tracking representations for rendering and interaction purposes.

- **HandIndex**
  - **Type**: `EControllerHand`
  - **Description**: Enum indicating which hand is being tracked (left or right).
  - **Usage**: Helps identify whether the motion data pertains to the left or right hand, essential for hand-specific actions or interactions.
- **TrackingStatus**
  - **Type**: `EXRTrackingStatus`
  - **Description**: Enum indicating the tracking status of the motion controller.
  - **Usage**: Shows whether the controller is being tracked accurately, with possible statuses like `Tracked`, `NotTracked`, etc.
- **GripPosition**
  - **Type**: `FVector`
  - **Description**: The position of the grip in world coordinates.
  - **Usage**: Provides the 3D coordinates of the controller's grip, essential for positioning the virtual representation of the controller.
- **GripRotation**
  - **Type**: `FQuat`
  - **Description**: The rotation of the grip in world coordinates.
  - **Usage**: Provides the orientation of the controller's grip, allowing for accurate rotation and alignment in the virtual space.
- **AimPosition**
  - **Type**: `FVector`
  - **Description**: The position of the aim point in world coordinates.
  - **Usage**: Specifies where the controller is aiming, useful for aiming or pointing actions.
- **AimRotation**
  - **Type**: `FQuat`
  - **Description**: The rotation of the aim point in world coordinates.

    - **Usage**: Determines the orientation of the aim direction, important for actions like shooting or selecting objects in VR.

- **PalmPosition**

    - **Type**: `FVector`
    - **Description**: The position of the palm in world coordinates.
    - **Usage**: Provides the 3D location of the palm, important for determining hand gestures or interactions in VR.

- **PalmRotation**

    - **Type**: `FQuat`
    - **Description**: The rotation of the palm in world coordinates.
    - **Usage**: Defines the orientation of the palm, crucial for hand-based interaction accuracy and realism in VR experiences.

- **HandKeyPositions**

    - **Type**: `TArray<FVector>`
    - **Description**: An array of vectors representing key positions of the hand.
    - **Usage**: Provides detailed positions of key points on the hand, useful for precise hand tracking and interaction.

- **HandKeyRotations**

    - **Type**: `TArray<FQuat>`
    - **Description**: An array of quaternions representing key rotations of the hand.
    - **Usage**: Complements the hand key positions with rotational data, ensuring accurate representation of hand movements.

- **HandKeyRadii**

    - **Type**: `TArray<float>`
    - **Description**: An array of floats representing the radii of key points of the hand.
    - **Usage**: Gives the size of the hand key points, aiding in collision detection and interaction fidelity.

- **bIsGrasped**
    - **Type**: `bool`
    - **Description**: A boolean indicating whether the controller is currently grasping an object.
    - **Usage**: Determines if the user is holding something, affecting interactions and animations.

Organization of FXRMotionControllerData

The structure is organized to encapsulate all relevant data needed for hand and motion controller tracking in a coherent and accessible manner. Boolean flags `bValid` and `bIsGrasped` provide quick checks on the state of the controller data. Identifiers `DeviceName` and `ApplicationInstanceID` ensure the correct association of data. Positional and rotational data `GripPosition`, `GripRotation`, `AimPosition`, and `AimRotation` offer precise tracking of the controller's movement. Arrays `HandKeyPositions`, `HandKeyRotations`, and `HandKeyRadii` allow detailed hand tracking, which is critical for immersive VR experiences. Lastly, the tracking status `TrackingStatus` informs the system of the reliability of the data being processed and whether the motion controller is actively being tracked or it's inactive at the moment.

Processing the Data for Drawing and Animating a Virtual Hand

In order to draw and animate a virtual hand in real-time whether the data is coming from hand-tracking or a SenseGlove device, we could consume the data from the `HandKeyPositions` and `HandKeyRotations` fields of the `FXRMotionControllerData` struct.

Both `HandKeyPositions` and `HandKeyRotations` contain 26 elements as defined by OpenXR's `XR_HAND_JOINT_COUNT_EXT` and `XrHandJointLocationsEXT`, etc.

Unreal Engine also provides an enum called `EHandKeypoint` naming the 26 joints, and the equivalent of `XR_HAND_JOINT_COUNT_EXT` as `EHandKeypointCount` inside

`Engine/Source/Runtime/HeadMountedDisplay/Public/HeadMountedDisplayTypes.h` as follows:

```
/**
 * Transforms that are tracked on the hand.
 * Matches the enums from WMR to make it a direct mapping
 */
UENUM(BlueprintType)
enum class EHandKeypoint : uint8
{
    Palm,
    Wrist,
    ThumbMetacarpal,
    ThumbProximal,
    ThumbDistal,
    ThumbTip,
    IndexMetacarpal,
    IndexProximal,
    IndexIntermediate,
    IndexDistal,
    IndexTip,
    MiddleMetacarpal,
    MiddleProximal,
    MiddleIntermediate,
    MiddleDistal,
    MiddleTip,
    RingMetacarpal,
    RingProximal,
    RingIntermediate,
    RingDistal,
    RingTip,
    LittleMetacarpal,
    LittleProximal,
    LittleIntermediate,
    LittleDistal,
    LittleTip
};

const int32 EHandKeypointCount = static_cast<int32>(EHandKeypoint::LittleTip)
+ 1;
```

So, getting the any joint's position or rotation is as easy as casting the enum value and passing it as the array index.

```
    FXRMotionControllerData MotionControllerData;
    const bool bGotMotionControllerData =
FSGXRTracker::GetMotionControllerData(
        GetWorld(), EControllerHand::Left, MotionControllerData);

    // Return if the struct data is invalid!
    if (!bGotMotionControllerData || !MotionControllerData.bValid)
    {
        return;
    }

    // Return if the device is not being tracked!
    if (MotionControllerData.TrackingStatus == ETrackingStatus::NotTracked)
    {
        return;
    }

    // Ensure that MotionControllerData.DeviceVisualType is a hand!
    if (!ensureAlwaysMsgf(MotionControllerData.DeviceVisualType
                          == EXRVisualType::Hand,
                          TEXT("Invalid DeviceVisualType type!")))
    {
    }

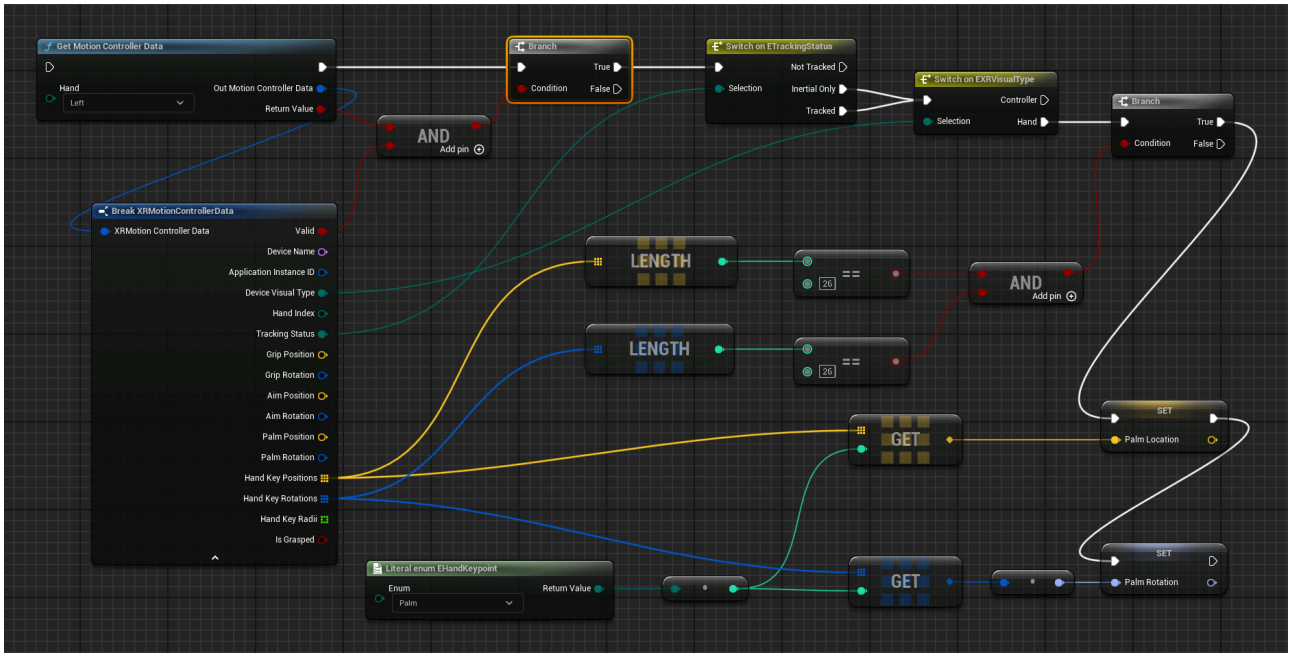
    // Ensure that MotionControllerData.HandKeyPositions has the position
data
    // for 26 joints!
    if (!ensureAlwaysMsgf(MotionControllerData.HandKeyPositions.Num()
                          == EHandKeypointCount,
                          TEXT("Invalid HandKeyPositions count!")))
    {
        return;
    }

    // Ensure that MotionControllerData.HandKeyRotations has the rotation
data
    // for 26 joints!
    if (!ensureAlwaysMsgf(MotionControllerData.HandKeyRotations.Num()
                          == EHandKeypointCount,
                          TEXT("Invalid HandKeyRotations count!")))
    {
        return;
    }

    static constexpr int32 PalmIndex = static_cast<int32>
(EHandKeypoint::Palm);
```

```
const FVector& PalmPosition{
    MotionControllerData.HandKeyPositions[PalmIndex]
};
const FRotator& PalmRotation{
    MotionControllerData.HandKeyRotations[PalmIndex].Rotator()
};
```

The equivalent Blueprint code for the above looks something like this:



OK, now that we've got a glimpse of how the virtual hand's joint data could be processed we are going to draw and animate a virtual hand in both Blueprint and C++ in the upcoming sections.

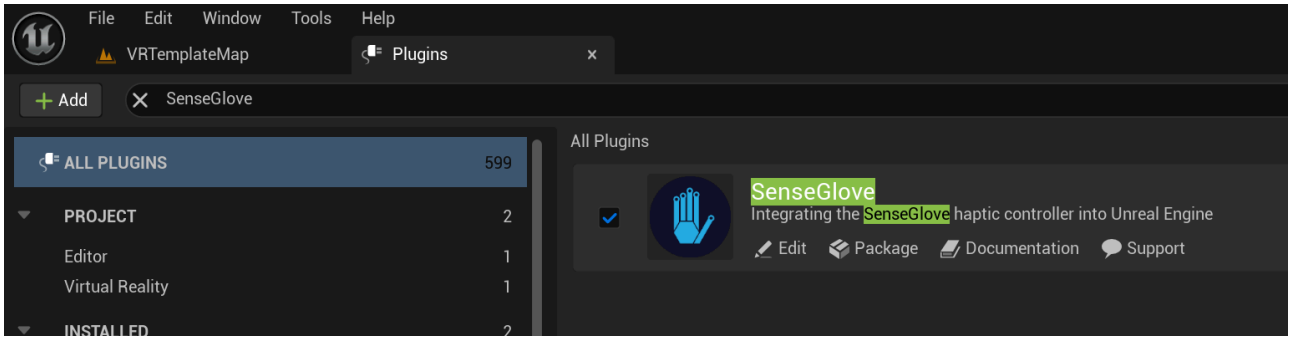
Consuming FXRMotionControllerData in Blueprint

> **Important**
>
> Unreal Engine versions `5.2`, `5.3`, and `5.4` are limited to `FXRMotionControllerData` since at the time of their release no `FXRHandTrackingState` was available. Also please keep in mind that, while `FXRMotionControllerData` is pretty much usable and functional in Unreal Engine `5.5`, it is recommended to utilize `FXRHandTrackingState` instead. This is because this version of UE has deprecated `FXRMotionControllerData` in favor of the `FXRMotionControllerState` and `FXRHandTrackingState` structs. Prior to version `5.5`, `FXRMotionControllerData` handled both motion controller and hand tracking data. From `5.5` onward, these responsibilities have been separated into the two distinct structs, providing clearer and more specialized handling of each.

Before continuing this section, please ensure you've studied the Consuming FXRMotionControllerData section, first.

Drawing and Animating Virtual Hands

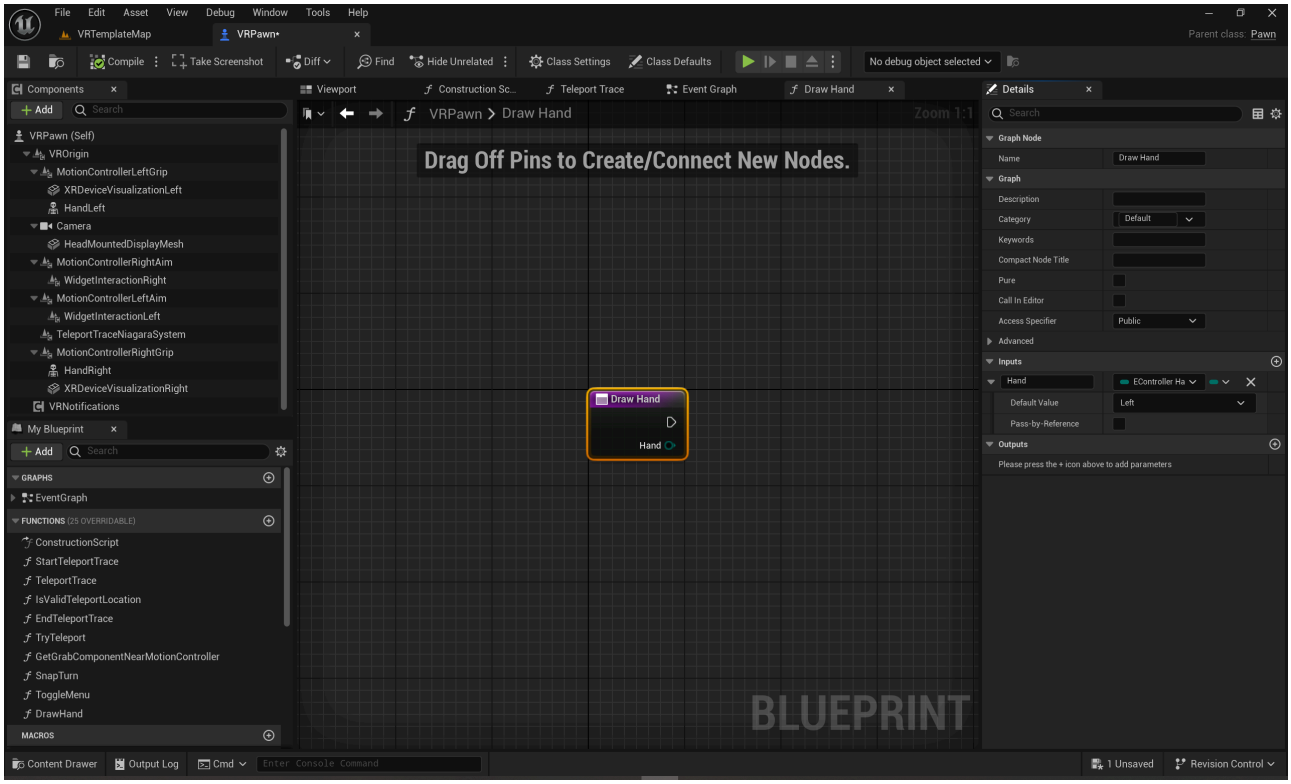
1. Create a new Virtual Reality project based the Unreal VR Template.
2. Make sure the SenseGlove UnrealEngine plugin is installed and enabled inside your new project.



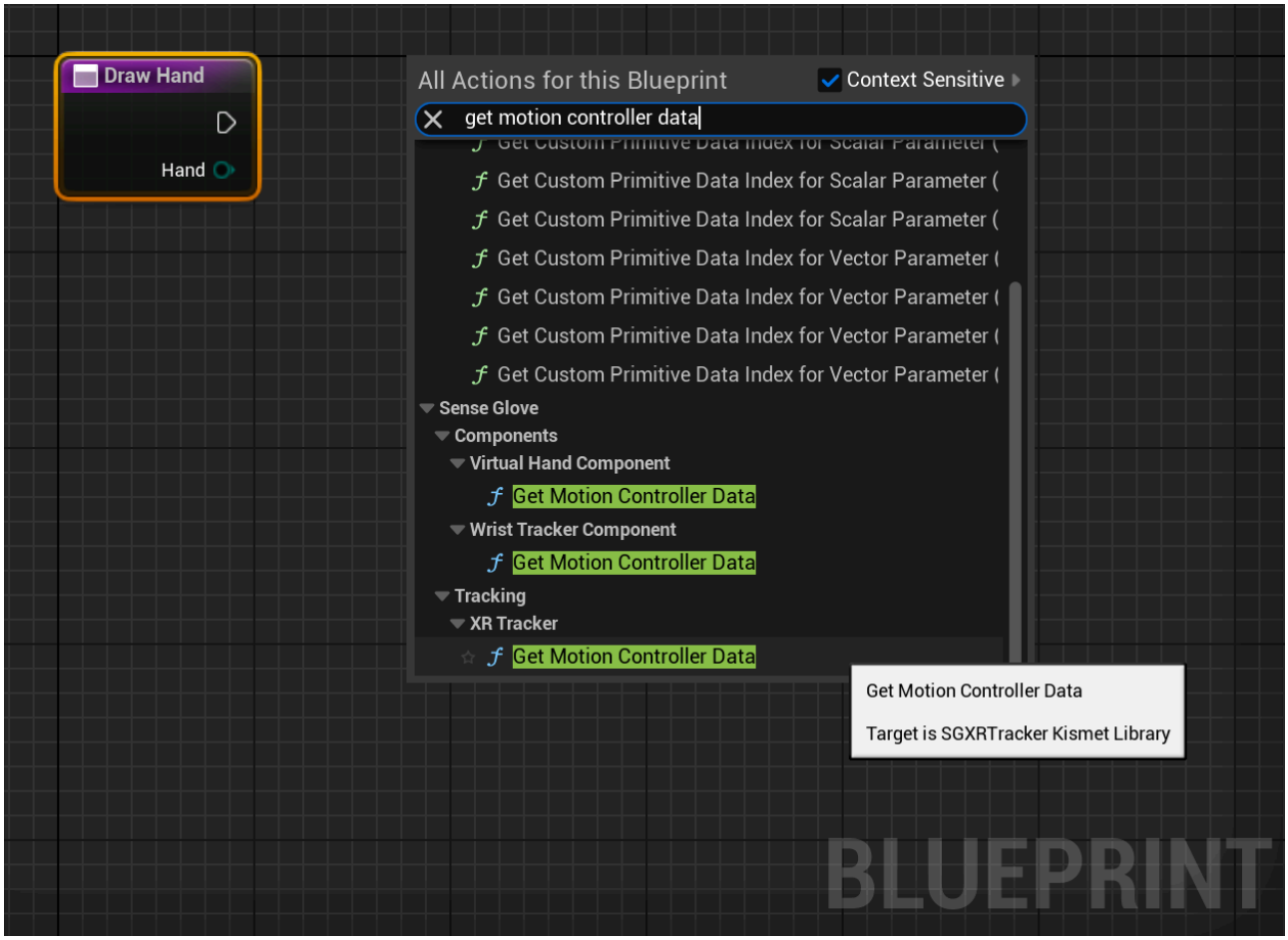
3. You could use either hand-tracking or a SenseGlove device as the input data, or both of the inside the same project. Whether you would like to use hand-tracking or a SenseGlove device, please make sure the required steps are taken for each of those first.

4. You could add the required Blueprint code for drawing virtual hands to either your Level Buleprint or the VRPawn Blueprint Class located at `/Content/VRTemplate/Blueprints/VRPawn`. In this guide we are going to add the code to our VRPawn.

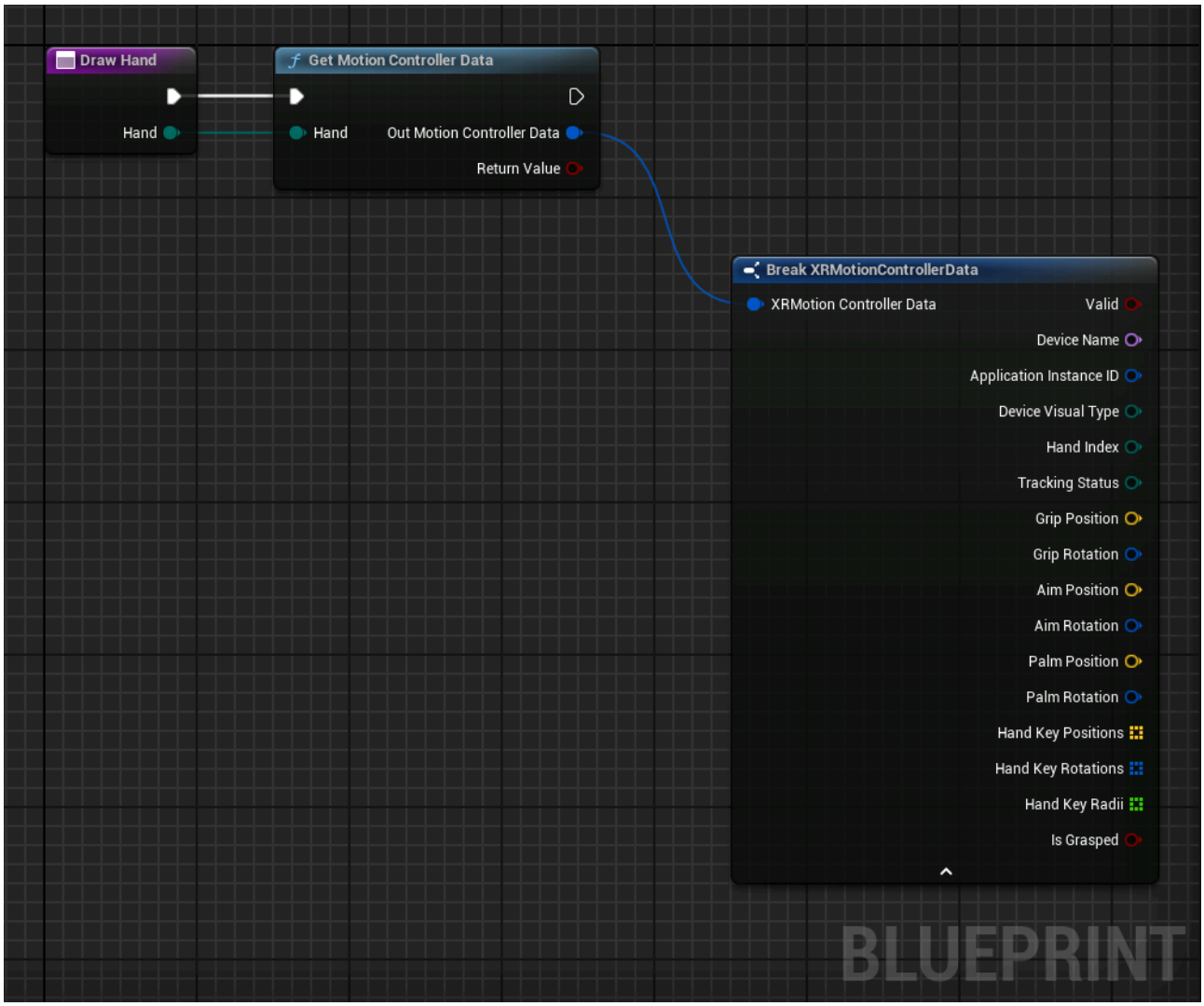
5. Add a new function named `Draw Hand` with an input parameter of type `EController Hand` named `Hand`.



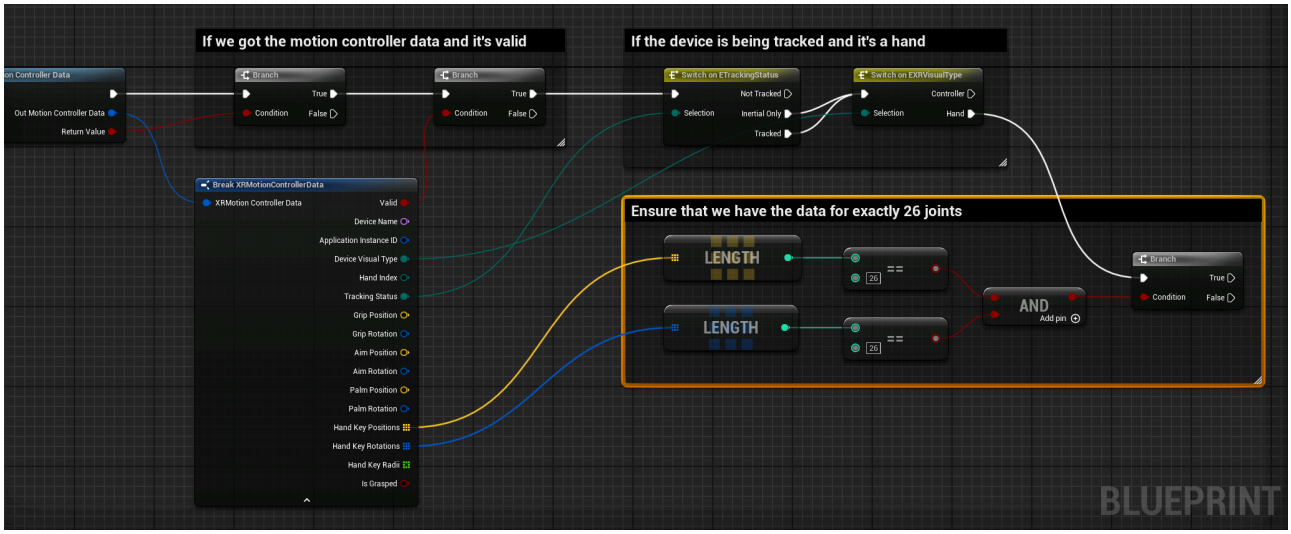
6. Inside this function's event graph add a `Get Motion Controller Data` node from `SenseGlove > Tracking > XR Tracker > Get Motion Controller Data`.



7. Then connect the functions `Hand` input parameter to the `Get Motion Controller Data` 's `Hand` input and right-click on the `OutMotionControllerData` parameter and use the `Break XRMotionControllerData` node to break the struct to it's fields.



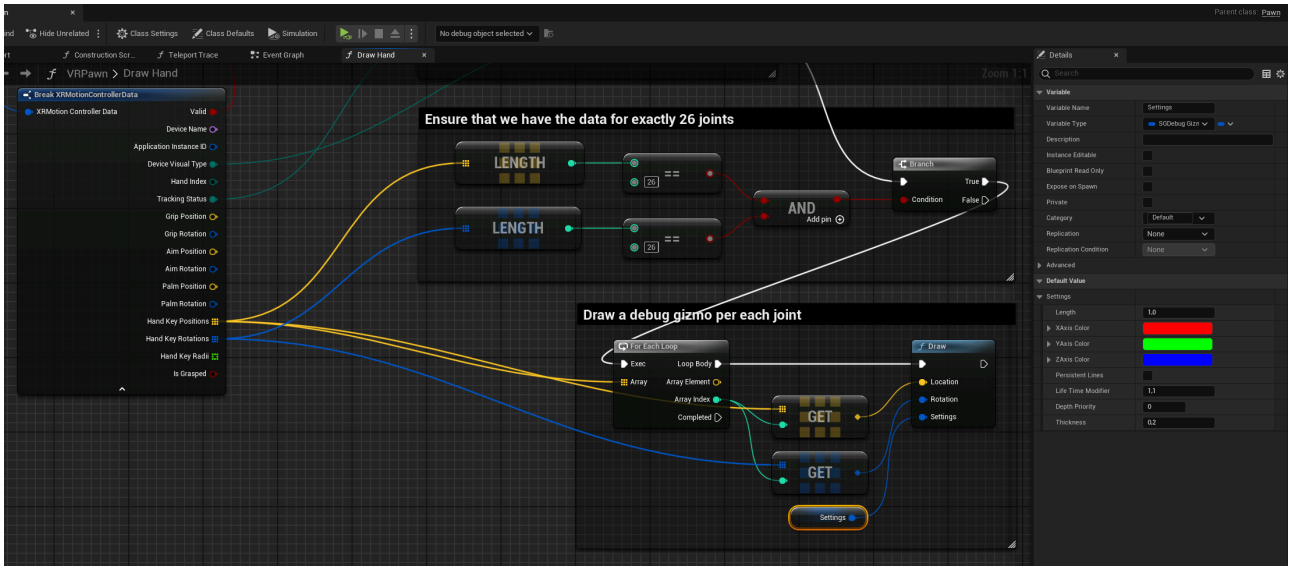
8. After this, we need to perform data validation by checking the return status of the `Get Motion Controller Data` function and `FXRMotionControllerData`'s `Valid` field. Then, we check if the motion controller device is being tracked and indeed coming from a hand-tracking source. And, finally, we check whether we have the positions and rotations for exactly `26` joints or not.



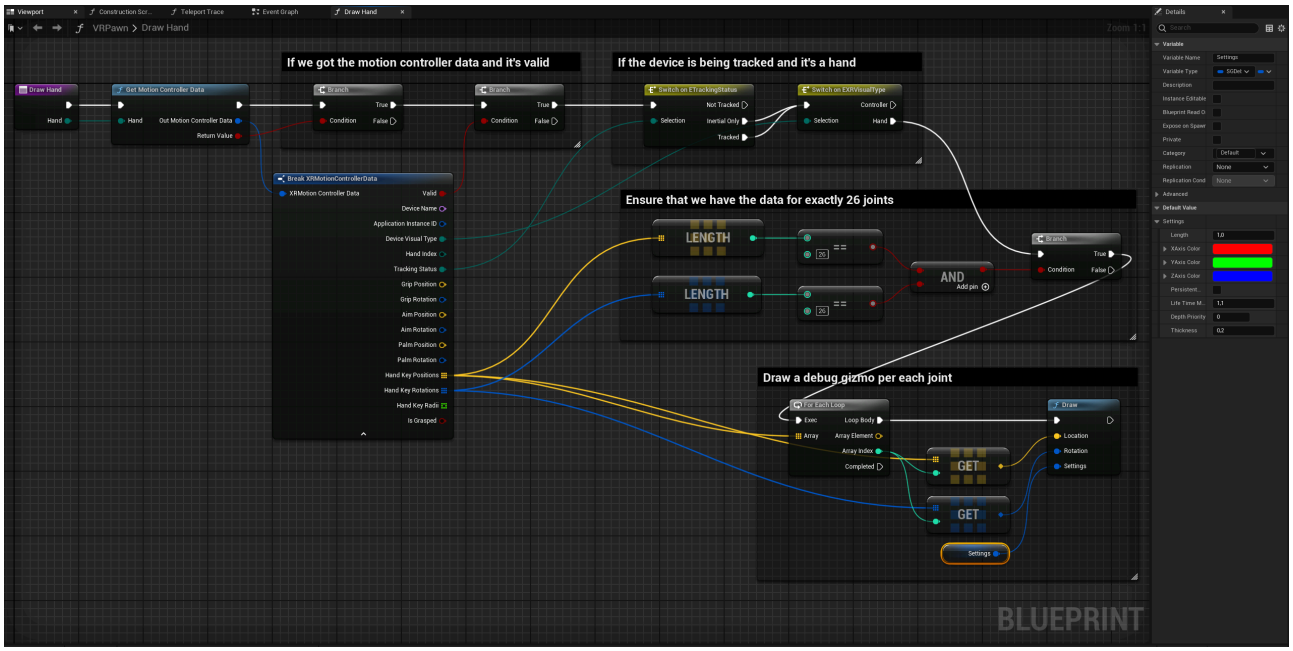
9. OK, now it's time to draw the joints! If we check out the SenseGlove Debug module's draw option, we notice there are various ways to draw the debug virtual hand. Drawing a cube or a gizmo per joint, or draw the whole hand all at once by passing the retrieved `FXRMotionControllerData` to the `DebugVirtualHand::Draw` function! But, since the point of this tutorial is to learn how to consume the `FXRMotionControllerData` we ignore the last option. Between the debug cubes or gizmos, we are going to choose the gizmos since they better represent the rotations than the cubes.



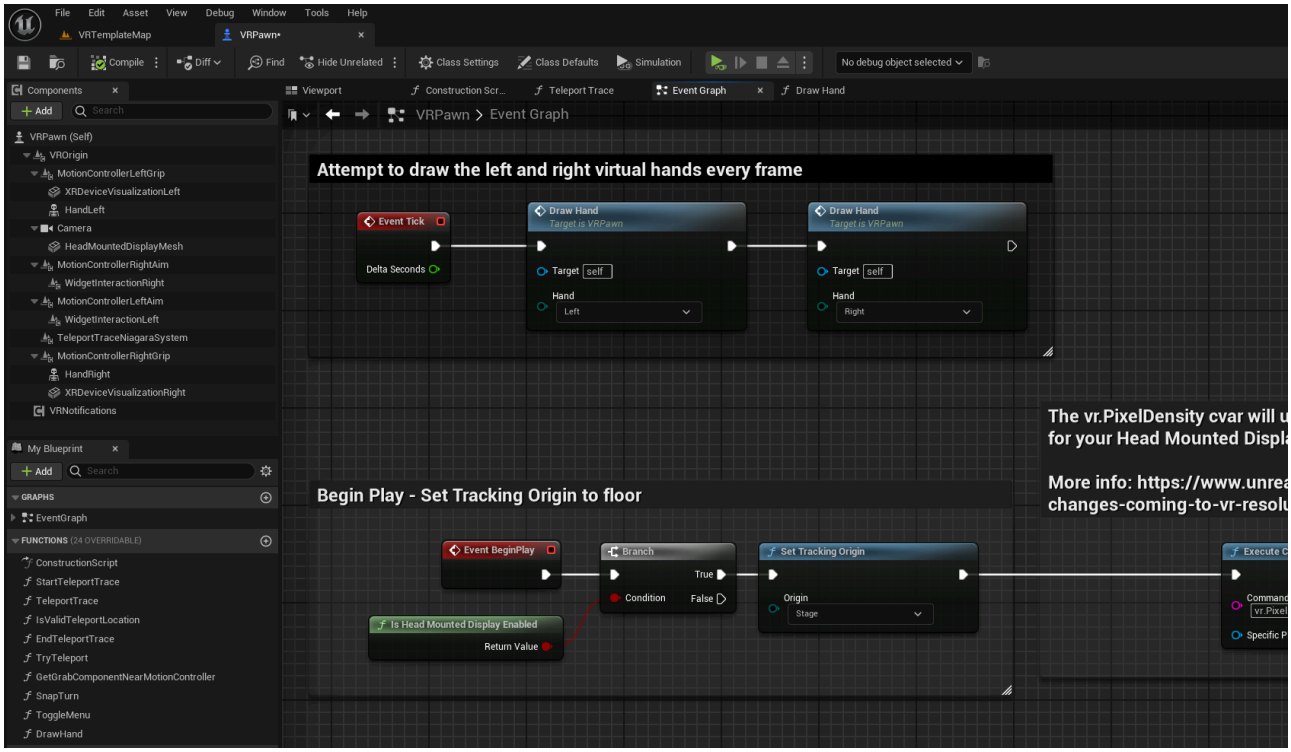
10. In the last step inside the `Draw Hand` function, in order to draw a virtual hand with `26` joints, we have to first iterate through either of the `Hand Key Positions` or `Hand Key Rotations` arrays from the `FXRMotionControllerData` struct. Since we made sure both arrays have `26` elements before we reached this step, it's safe to just iterate over one and use the `Array Index` inside a `For Each Loop` or a `For Loop` to access the position and rotation of every joint. Then we use each array `Get (a ref)` method to access the position and rotation data inside the loop and call the `Draw` function from `SenseGlove > Debug > Gizmo` per every joint. Please note that there are two `Draw` functions and the only difference between the two is that one accepts an `FQuat` and the other a `FRotator` for its `Rotation` input parameter. In this case, we use the `FQuat` variant to avoid an extra conversion to `FRotator`. Also, please adjust the `Thickness` option for the `Settings` parameter from `1.0` to `0.2`, as the default value might be too thick for drawing a joint gizmo.



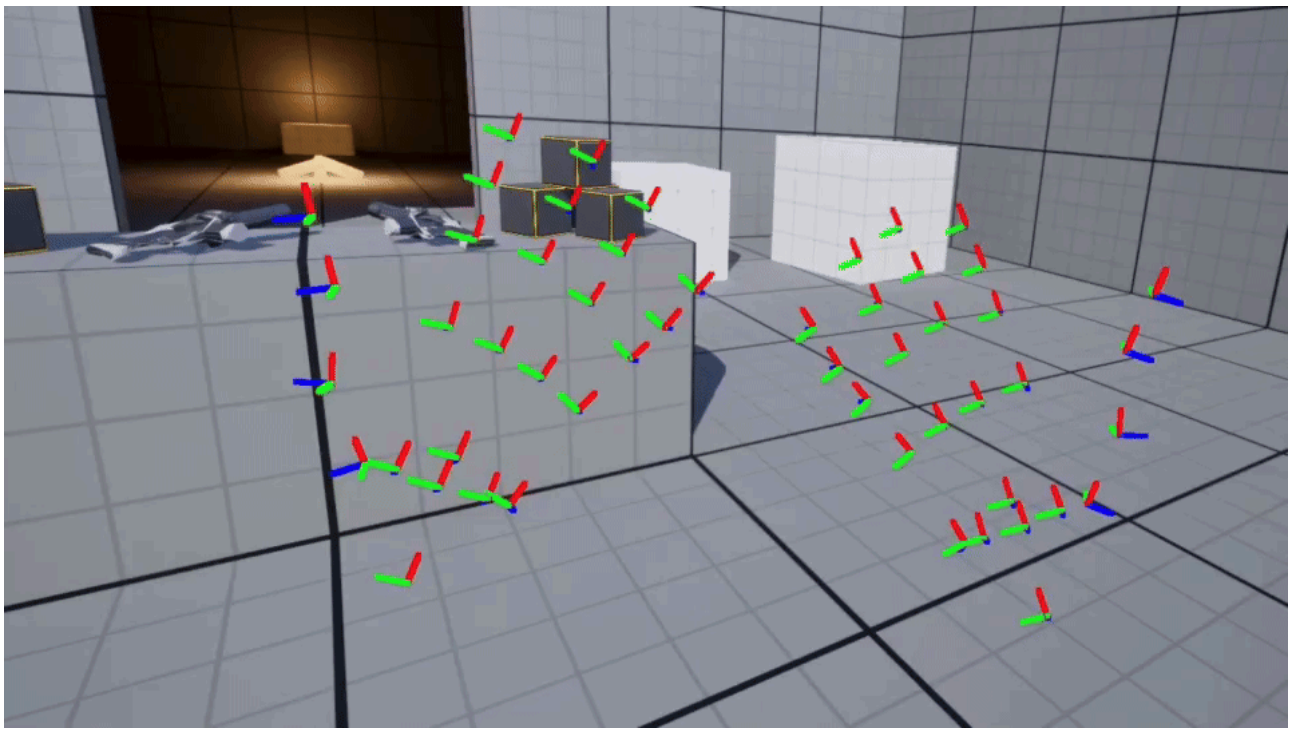
11. Well, now the full implementation for the `Draw Hand` function insde the `VRPawn` should look something like this:



12. Finally, go back to `VRPawn` 's event graph and the following code to the `Tick` event. Basically what we do here is call our newly implemented `Draw Hand` twice, once for each hand.



13. Now, go back to the `VRTemplateMap` and use the VR Preview button to run the game. If everything's done correctly, you should be able to see the virtual hands inside your VR simulation.



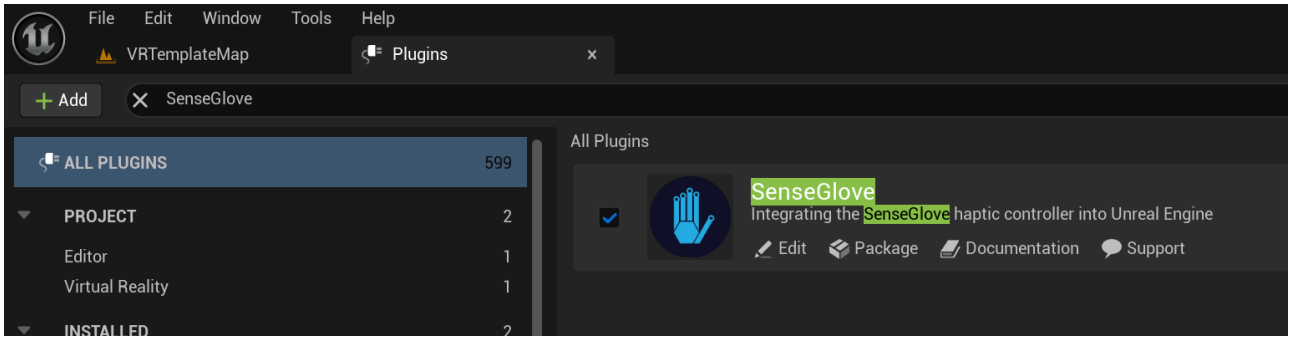
Consuming FXRMotionControllerData in C++

> **Important**
>
> Unreal Engine versions `5.2`, `5.3`, and `5.4` are limited to `FXRMotionControllerData` since at the time of their release no `FXRHandTrackingState` was available. Also please keep in mind that, while `FXRMotionControllerData` is pretty much usable and functional in Unreal Engine `5.5`, it is recommended to utilize `FXRHandTrackingState` instead. This is because this version of UE has deprecated `FXRMotionControllerData` in favor of the `FXRMotionControllerState` and `FXRHandTrackingState` structs. Prior to version `5.5`, `FXRMotionControllerData` handled both motion controller and hand tracking data. From `5.5` onward, these responsibilities have been separated into the two distinct structs, providing clearer and more specialized handling of each.

Before continuing this section, please ensure you've first studied the Consuming FXRMotionControllerData section.

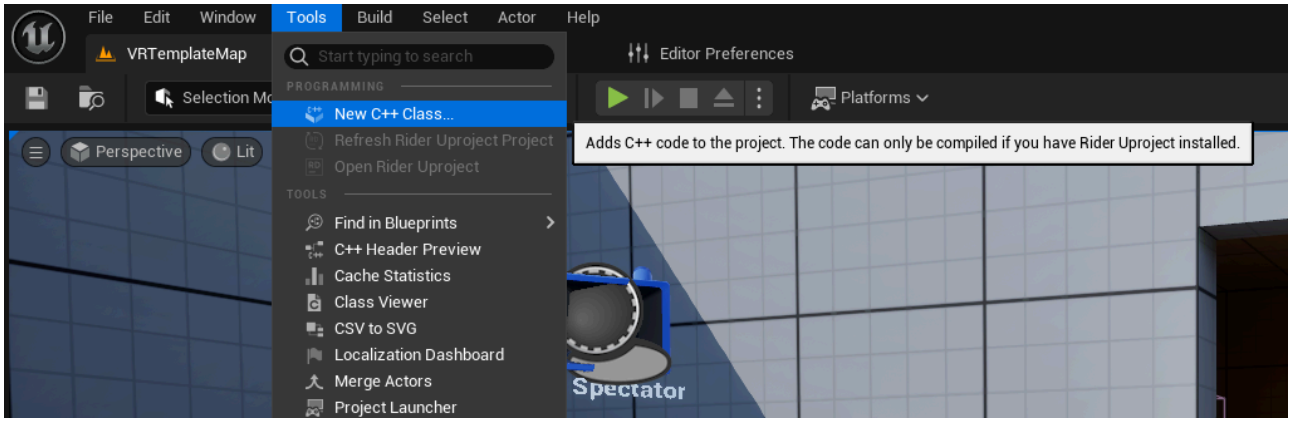
Drawing and Animating Virtual Hands

1. Create a new Virtual Reality project based the Unreal VR Template.
2. Make sure the SenseGlove UnrealEngine plugin is installed and enabled inside your new project.

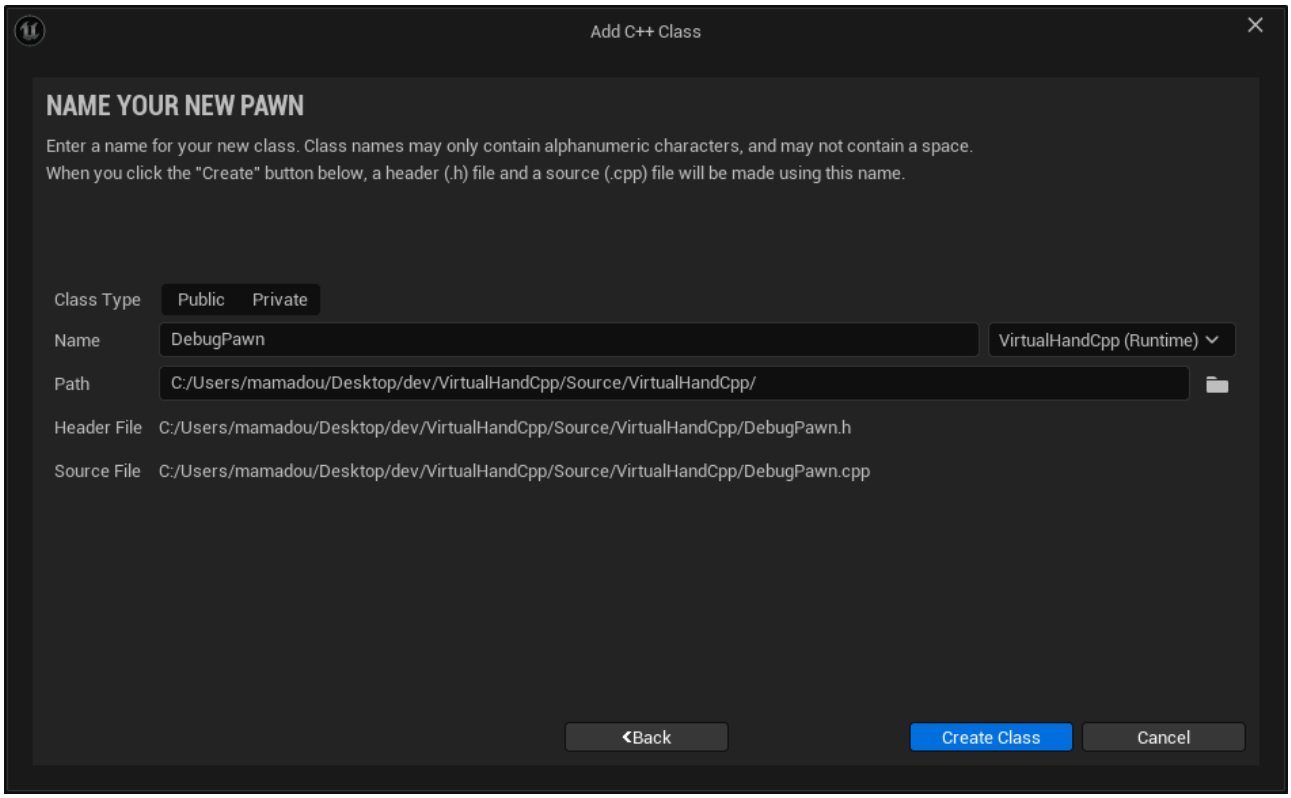


3. You could use either hand-tracking or a SenseGlove device as the input data, or both of the inside the same project. Whether you would like to use hand-tracking or a SenseGlove device, please make sure the required steps are taken for each of those first.

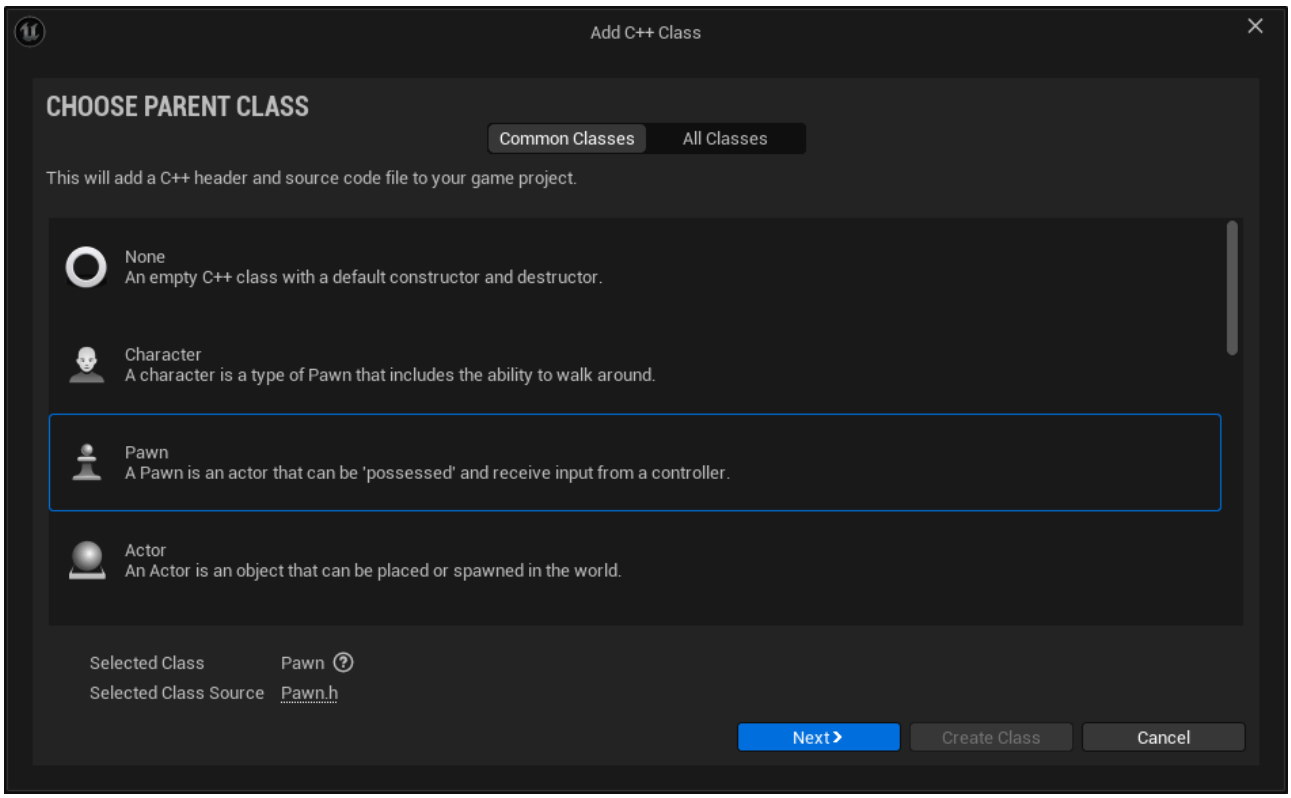
4. From the `Tools` menu choose `New C++ class...`.



5. Choose the Unreal Engine's `APawn` class as the parent class for the new C++ pawn class.

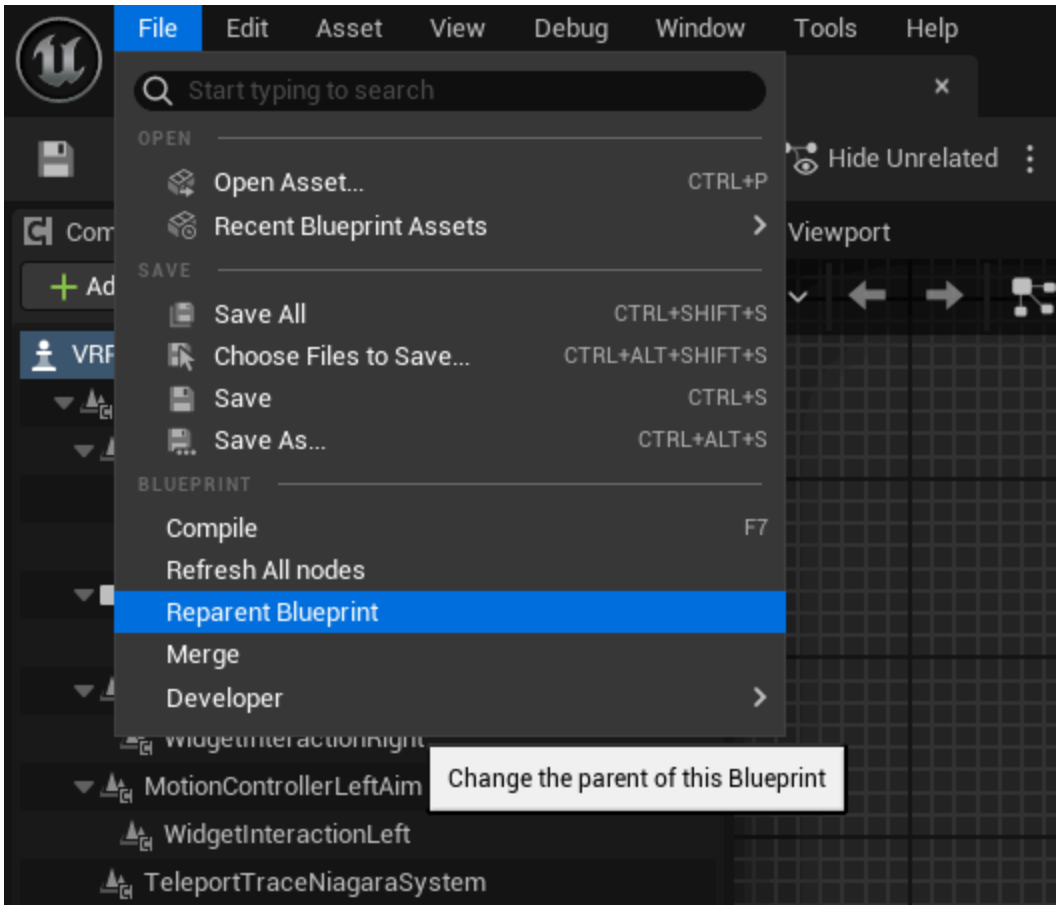


6. Name the new pawn class `DebugPawn`.



7. Since we have created a new C++ class, this converts the current Blueprint VRTemplateMap project to a C++ one. That's why the Unreal Editor will give us a few prompts regarding opening the project in the default IDE and rebuilding the code. It might be simpler to just close the editor, then rebuild the source code inside your favorite IDE, and then start the editor with the converted project again.

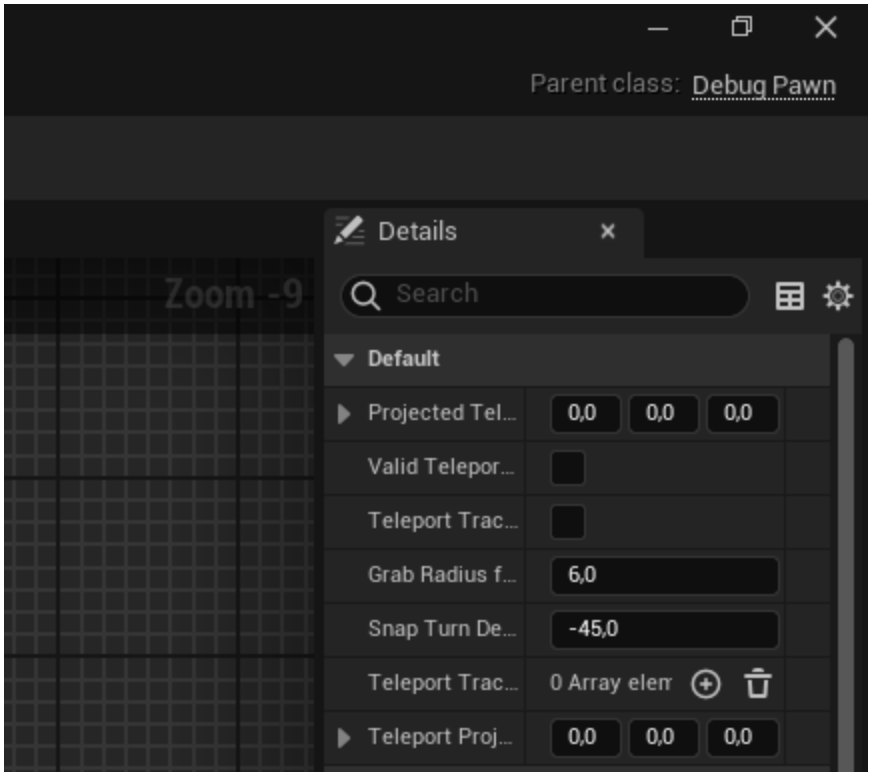
8. Find and open the VRPawn Blueprint Class located at `/Content/VRTemplate/Blueprints/VRPawn` inside the Blueprint Editor and from the `File` menu choose the `Reparent Blueprint` class.



9. In the new `Reparent blueprint` window choose `DebugPawn` as the new parent.



10. By looking at the `Parent Class` label located under the Blueprint Editor window control buttons verify that the `ADebugPawn` class has been set as the new parent.



11. Locate the project's main Build file, in our case `VirtualHandCpp/Source/VirtualHandCpp/VirtualHandCpp.Build.cs` and add the `InputDevice`, `OpenXRHMD`, `SenseGloveBuildHacks`, `SenseGloveDebug`, `SenseGloveSettings`, and `SenseGloveTracking` modules as either a private or public dependency.

```
// Fill out your copyright notice in the Description page of Project
Settings.

using UnrealBuildTool;

public class VirtualHandCpp : ModuleRules
{
    public VirtualHandCpp(ReadOnlyTargetRules Target) : base(Target)
    {
        PCHUsage = PCHUsageMode.UseExplicitOrSharedPCHs;

        PublicDependencyModuleNames.AddRange(new string[] { "Core",
"CoreUObject", "Engine", "InputCore" });

        PrivateDependencyModuleNames.AddRange(new string[]
        {
            "InputDevice",
            "OpenXRHMD",
            "SenseGloveBuildHacks",
            "SenseGloveDebug",
            "SenseGloveSettings",
            "SenseGloveTracking"
        });

        // Uncomment if you are using Slate UI
        // PrivateDependencyModuleNames.AddRange(new string[] { "Slate",
"SlateCore" });

        // Uncomment if you are using online features
        // PrivateDependencyModuleNames.Add("OnlineSubsystem");

        // To include OnlineSubsystemSteam, add it to the plugins section in
your uproject file with the Enabled attribute set to true
    }
}
```

12. Locate the C++ header and source file for the `ADebugPawn` inside the project in your C++ IDE. In our case they are located at `VirtualHandCpp/Source/VirtualHandCpp/DebugPawn.h` and `VirtualHandCpp/Source/VirtualHandCpp/DebugPawn.cpp`.

13. Modify the `DebugPawn.h` header file to look like this:

```
// Fill out your copyright notice in the Description page of Project
Settings.

#pragma once

#include "CoreMinimal.h"
#include "GameFramework/Pawn.h"

#include "SGSettings/SGDebugGizmoSettings.h"

#include "DebugPawn.generated.h"

UCLASS()
class VIRTUALHANDCPP_API ADebugPawn : public APawn
{
    GENERATED_BODY()

private:
    // The virtual hand drawing settings.
    UPROPERTY(EditDefaultsOnly, Category="DebugPawn",
        meta=(AllowPrivateAccess="false"))
    FSGDebugGizmoSettings HandDrawingSettings;

public:
    // Sets default values for this pawn's properties
    ADebugPawn();

protected:
    // Called when the game starts or when spawned
    virtual void BeginPlay() override;

public:
    // Called every frame
    virtual void Tick(float DeltaTime) override;

    // Called to bind functionality to input
    virtual void SetupPlayerInputComponent(class UInputComponent*
PlayerInputComponent) override;

private:
    // The method responsible for drawing a virtual hand.
    void DrawHand(EControllerHand Hand) const;
};
```

14. Modify the `DebugPawn.cpp` implementation file to look like this:

```
// Fill out your copyright notice in the Description page of Project
Settings.


#include "DebugPawn.h"

#include "SGDebug/SGDebugGizmo.h"
#include "SGTracking/SGXRTracker.h"

// Sets default values
ADebugPawn::ADebugPawn()
{
    // Set this pawn to call Tick() every frame.  You can turn this off to
improve performance if you don't need it.
    PrimaryActorTick.bCanEverTick = true;

    // Set the default virtual hand drawing settings.
    HandDrawingSettings = FSGDebugGizmoSettings{
        1.0f,
        FColor{255, 0, 0, 255},
        FColor{0, 255, 0, 255},
        FColor{0, 0, 255, 255},
        false,
        1.1f,
        0,
        0.2f,
    };
}

// Called when the game starts or when spawned
void ADebugPawn::BeginPlay()
{
    Super::BeginPlay();
}

// Called every frame
void ADebugPawn::Tick(float DeltaTime)
{
    Super::Tick(DeltaTime);

    // Attempt at drawing the left/right virtual hands every frame.
    DrawHand(EControllerHand::Left);
    DrawHand(EControllerHand::Right);
}

// Called to bind functionality to input
void ADebugPawn::SetupPlayerInputComponent(UInputComponent*
```

```
PlayerInputComponent)
{
    Super::SetupPlayerInputComponent(PlayerInputComponent);
}

void ADebugPawn::DrawHand(const EControllerHand Hand) const
{
    // Get the world and cache it, if it's null we return early.
    UWorld* World{GetWorld()};
    if (!IsValid(World))
    {
        return;
    }

    FXRMotionControllerData MotionControllerData;
    const bool bGotMotionControllerData =
FSGXRTracker::GetMotionControllerData(
        World, Hand, MotionControllerData);

    // Return if the struct data is invalid!
    if (!bGotMotionControllerData || !MotionControllerData.bValid)
    {
        return;
    }

    // Return if the device is not being tracked!
    if (MotionControllerData.TrackingStatus == ETrackingStatus::NotTracked)
    {
        return;
    }

    // Ensure that MotionControllerData.DeviceVisualType is a hand!
    if (!ensureAlwaysMsgf(MotionControllerData.DeviceVisualType
                          == EXRVisualType::Hand,
                          TEXT("Invalid DeviceVisualType type!")))
    {
    }

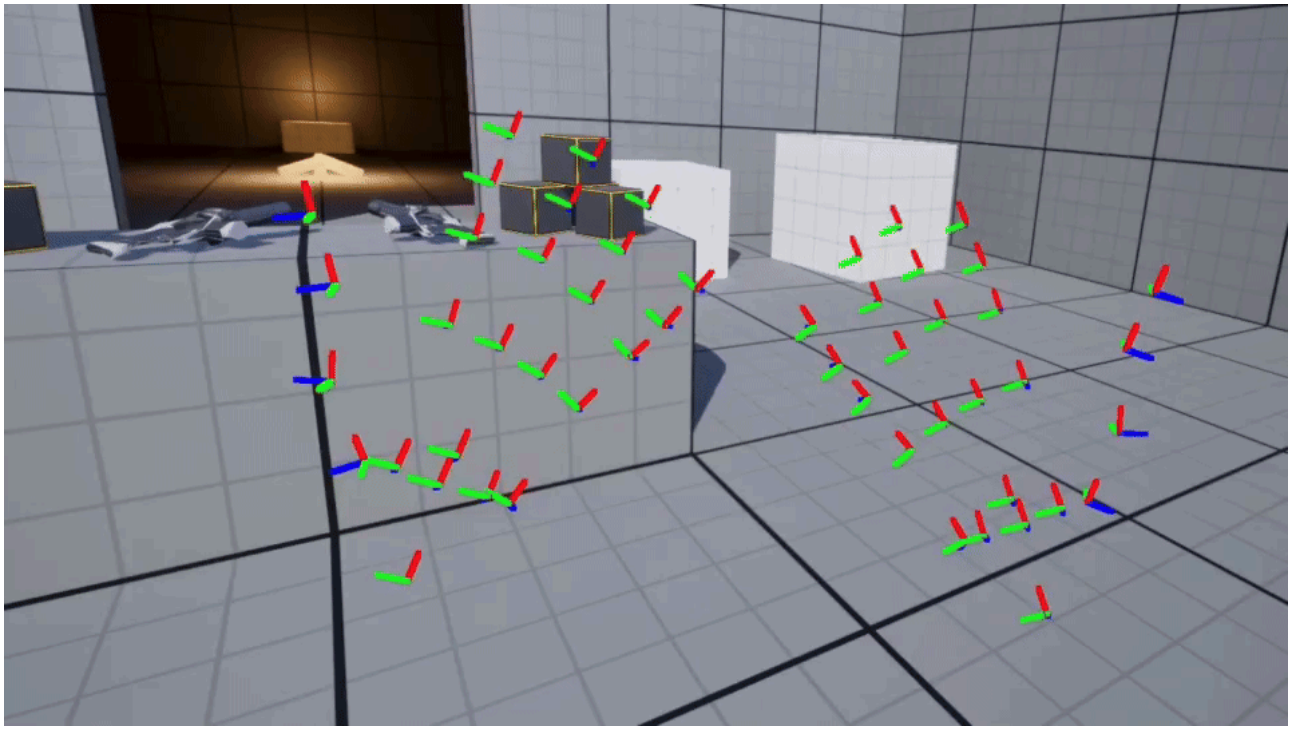
    // Ensure that MotionControllerData.HandKeyPositions has the position
data
    // for 26 joints!
    if (!ensureAlwaysMsgf(MotionControllerData.HandKeyPositions.Num()
                          == EHandKeypointCount,
                          TEXT("Invalid HandKeyPositions count!")))
    {
        return;
    }
```

```
    // Ensure that MotionControllerData.HandKeyRotations has the rotation
data
    // for 26 joints!
    if (!ensureAlwaysMsgf(MotionControllerData.HandKeyRotations.Num()
                          == EHandKeypointCount,
                          TEXT("Invalid HandKeyRotations count!")))
    {
        return;
    }

    // Iterate over the hand joint positions and rotations!
    for (int32 JointIndex = 0; JointIndex < EHandKeypointCount; ++JointIndex)
    {
        const FVector& JointPosition{
            MotionControllerData.HandKeyPositions[JointIndex]
        };
        const FQuat& JointRotation{
            MotionControllerData.HandKeyRotations[JointIndex]
        };

        // Draw a single joint's gizmo!
        // Please note that we could alternatively:
        // Use FSGDebugCube::Draw() to draw a cube.
        // Or use the FSGDebugVirtualHand::Draw() method and pass the
        // MotionControllerData directly to draw the virtual hand
        // all at once without iterating the joints. But, that's not
        // goal of this tutorial.
        FSGDebugGizmo::Draw(World, JointPosition, JointRotation,
HandDrawingSettings);
    }
}
```

15. Now, rebuild the source code and go back to the `VRTemplateMap`, then use the VR Preview button to run the game. If everything's done correctly, you should be able to see the virtual hands inside your VR simulation.

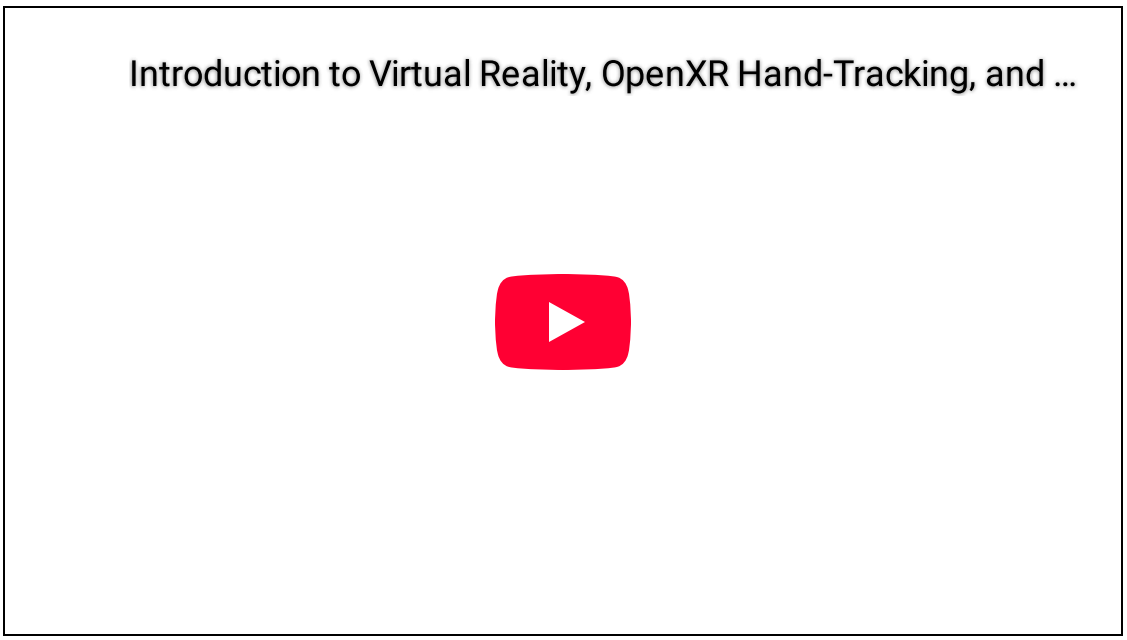


Third-Party Tutorials: Consuming OpenXR Hand-Tracking Data

Introduction to Virtual Reality, OpenXR Hand-Tracking, and Gesture Detection in Unreal Engine

In this tutorial, you'll learn how to get the OpenXR Hand-Tracking Data from the Unreal Engine API and how to consume it to draw virtual hand models using cubes (as hand joints). It will also dive into gesture recognition by implementing a simple pinch gesture recognition.

In the first part, it will focus on UE `4.26` to `5.4` API. And, in the second part, you'll learn how to update the project to work with `5.5`.

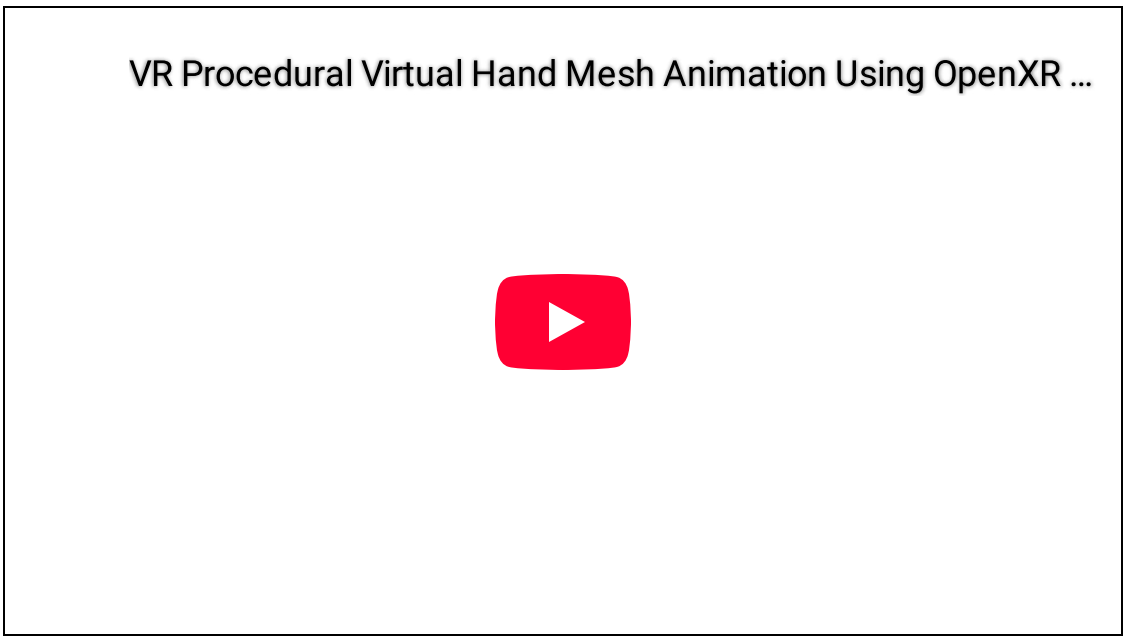


Procedural Virtual Hand Mesh Animation Using OpenXR Hand-Tracking Data

Building on the Introduction to Virtual Reality, OpenXR Hand-Tracking, and Gesture Recognition in Unreal Engine tutorial, this slightly more advanced tutorial will dive deeper into the following topics:

- Transitioning seamlessly between motion controller and hand-tracking modes in Unreal Engine.
- Adding custom debugging gizmos to improve development and testing workflows.
- Visualizing debug virtual hands by incorporating the custom gizmos.
- Animating virtual hand meshes with OpenXR hand-tracking data, moving beyond basic joint representation with cubes.
- Re-using and adapting the gesture recognition code from the introductory tutorial to integrate with the new animated virtual hand meshes. This guide will help you take your VR projects to the next level with polished and practical implementations.

Part 1:



Part 2:

VR Procedural Virtual Hand Mesh Animation Using OpenXR ...



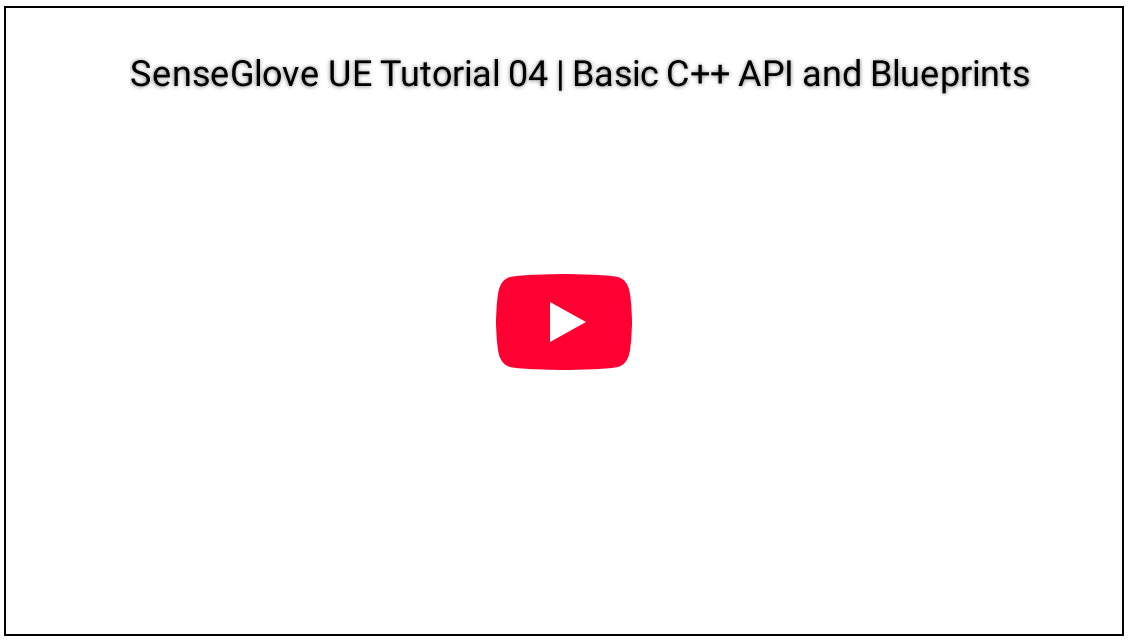
Low-level Blueprint API

Unfortunately, due to Unreal Engine's limited availability of automated documentation generation tools, there is no updated online documentation for the SenseGlove Blueprint API. However, this does not mean that no documentation is available. In fact, most of the Blueprint code is already documented within the relevant header files. Any modules with the `Kismet` postfix in the name contain the Blueprint documentation. For example, the Blueprint documentation for the `Core` module can be found inside the `Source/SenseGloveCoreKismet/Public/SGCoreKismet` directory.

There is also an outdated Blueprint documentation hosted on GitLab. This documentation was generated for the early releases of the plugin using kamrann/KantanDocGenPlugin and kamrann/KantanDocGenTool, which is no longer maintained.

Efforts are ongoing to generate comprehensive documentation using PsichiX/unreal-doc, but progress has been hindered by various known issues.

There are also other outdated materials that might still be partially relevant. These include an example Unreal Engine Blueprint project and a video tutorial:



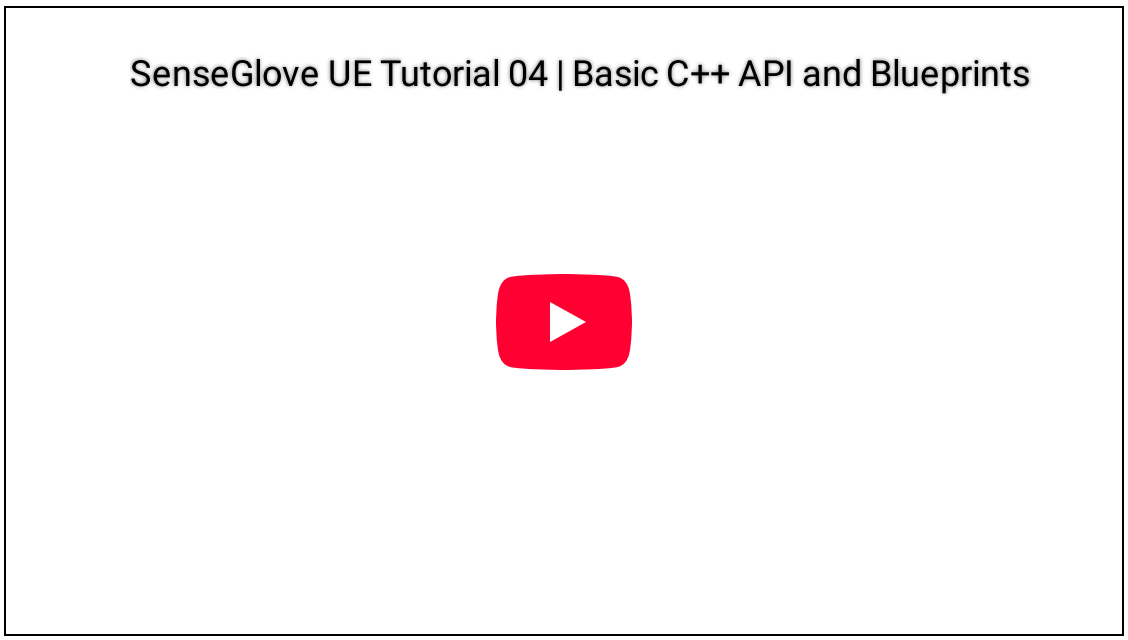
Low-level C++ API

Due to Unreal Engine's limited availability of automated documentation generation tools, there is no updated online documentation for the SenseGlove Unreal Engine C++ API. However, this does not mean that no documentation is available. A significant portion of the API is documented within the relevant header files. For example, the C++ API documentation for the `Core` module can be found inside the `Source/SenseGloveCore/Public/SGCore` directory.

Efforts are ongoing to generate comprehensive documentation using PsichiX/unreal-doc, but progress has been hindered by various known issues.

Nevertheless, since this plugin builds on top of the SGConnect and SGCoreCpp third-party C++ libraries, the upstream documentation provides detailed information on various aspects of the underlying SenseGlove C++ API.

There are also other outdated materials that might still be partially relevant. These include an example Unreal Engine C++ project and a video tutorial:



Platform Support Matrix

	Windows (MSVC 2017)	Windows (MSVC 2019)	Windows (MSVC 2022)	Linux x86-64 (Native Toolchain)	Linux AArch64 (Native Toolchain)
5.5	❌	❌	✅ v2.4.x	✅ v2.4.x	✅ v2.4.x
5.4	❌	❌	✅ v2.4.x	✅ v2.4.x	✅ v2.4.x
5.3	❌	✅ v2.4.x	✅ v2.4.x	✅ v2.4.x	✅ v2.4.x
5.2	❌	✅ v2.4.x	✅ v2.4.x	✅ v2.4.x	✅ v2.4.x
5.1	❌	⚠️ v2.0.x	⚠️ v2.0.x	⚠️ v2.0.x	⚠️ v2.0.x
5.0	❌	⚠️ v1.6.x	⚠️ v1.6.x	⚠️ v1.6.x	⚠️ v1.6.x
4.27	⚠️ v1.4.x	⚠️ v1.4.x	⚠️ v1.4.x	⚠️ v1.4.x	⚠️ v1.4.x
4.26	⚠️ v1.0.x	⚠️ v1.0.x	❌	⚠️ v1.0.x	❌
4.25	⚠️ v1.0.x	⚠️ v1.0.x	❌	⚠️ v1.0.x	❌
4.24	⚠️ v1.0.x	⚠️ v1.0.x	❌	⚠️ v1.0.x	❌
4.23	⚠️ v1.0.x	⚠️ v1.0.x	❌	⚠️ v1.0.x	❌
4.22	⚠️ v1.0.x	⚠️ v1.0.x	❌	⚠️ v1.0.x	❌

- ✅ Supported
- ⚠️ Not supported by the latest release and might be lacking features
- ❌ Not supported at all
- ❓ Unknown or untested

Remarks:

- Per Epic's Marketplace Guidelines in regards to Code Plugins (sections `2.6.3.d` and `3.1.b`), we are only able to distribute or update the SenseGlove plugin for the last `3` stable versions of Unreal Engine. As a result, we won't be able to publish updates or bug fixes for the older versions of the Engine except on rare occasions and only through our official repository on Microsoft Azure DevOps.
- All third-party libraries on Windows built against Windows SDK `10.0`.
- Oculus and VIVE support is only provided through the recommended Android NDK versions by Epic Games.
- wjwwood/serial requires Android NDK API Level `28+` in order to be built successfully.
- All third-party libraries target Android NDK API Level `29`, thus any project relying on the plug-in should be build with the same NDK API Level.
- Meta Quest PCVR-mode support is only provided through the Epic OpenXR plugin. Furthermore, the Standalone-mode support is also provided through the Epic OpenXR plugin only. Please note that enabling the Meta XR plugin will result in crashes or unexpected behavior. Meta XR plugin compatibility is being worked on at the moment and might be supported in the future.
- While HTC VIVE PCVR-mode support is only provided through the Epic OpenXR plugin, the standalone-mode support is only provided through the official HTC `ViveOpenXR` plugin and no other plugin is supported.

Planned Features Completion Status

Implemented as of v2.4.x

- ☑ Full SenseGlove low-level core API access through Unreal C++.
- ☑ Full SenseGlove low-level core API access through Blueprint.
- ☑ DK 1 Support.
- ☑ Nova 1 Support.
- ☑ Nova 2 Support.
- ☑ Support for Microsoft Windows as a development platform.
- ☑ Support for GNU/Linux as a development platform.
- ☑ Support for Microsoft Windows as a deployment platform.
- ☑ Support for GNU/Linux x64 as a deployment platform.
- ☑ Support for GNU/Linux AArch64 as a deployment platform.
- ☑ Support for Android as a deployment platform.
- ☑ Support for Oculus Quest 2, Oculus Quest Pro, and Oculus Quest 3.
- ☑ Support for HTC VIVE Pro, HTC VIVE Focus 3, HTC XR Elite, and HTC VIVE Focus Vision.
- ☑ Support for HTC VIVE Trackers and HTC VIVE Wrist Trackers.
- ☑ On-device calibration for Android without the need for SenseCom.
- ☑ Haptic feedback including force feedback, buzz, and thumper commands.
- ☑ A customizable Grab component that could be added to any actor.
- ☑ A customizable Touch component that could be added to any actor.
- ☑ Ability to grab, release, and throw objects around.
- ☑ Separation of the real and virtual hand rendering.
- ☑ An out-of-the-box customizable SGPawn with the ability to be extended in C++ and Blueprint.
- ☑ Easy wrist/hand tracking debugging using the SenseGlove Debug module.
- ☑ A generic Settings module with the ability to override settings.
- ☑ C++/Blueprint interaction events such as OnGrabStateUpdated, OnTouchStateUpdated, OnActorGrabbed, OnActorReleased, OnActorBeginTouch, and OnActorEndTouch.

- ☑ A fall back to HMD and wrist tracker hardware auto-detection mechanism when automatic detection of the wrist tracker hardware is desired.
- ☑ OpenXR-compatible hand tracking (XR_EXT_hand_tracking) support.
- ☑ `FXRMotionControllerData` compatible hand animation system on UE versions `5.2`, `5.3`, and `5.4`.
- ☑ `FXRHandTrackingState` compatible hand animation system on UE versions `5.5+`.
- ☑ `FXRMotionControllerData` compatible wrist tracking system on UE versions `5.2`, `5.3`, and `5.4`.
- ☑ `FXRHandTrackingState` compatible wrist tracking system on UE versions `5.5+`.
- ☑ `FXRMotionControllerData` compatible hand interaction manipulation system on UE versions `5.2`, `5.3`, and `5.4`.
- ☑ `FXRHandTrackingState` compatible hand interaction manipulation system on UE versions `5.5+`.
- ☑ Ability to fallback to hand tracking when a glove is not present and use the bare hands for interactions, or a combination of glove and hand tracking if no motion controller input is detected.
- ☑ The SenseGlove grab/touch sockets one-click-setup ability on any Epic-compliant virtual hand mesh from within the Unreal Editor's Content Browser, Skeleton Editor, or Skeletal Mesh Editor.
- ☑ A flexible virtual hand animation system that can take the mesh bone's transforms into account for a more reliable hand animation.
- ☑ Ability to manage the Engine Scalability Settings through the SenseGlove plugin in order to change the graphics settings on the fly.
- ☑ Ability to automatically ask for the required permissions on Android when the plugin is enabled in any UE project.
- ☑ `ViveOpenXR` plugin compatibility.

Upcoming features planned for the v2.5.x release

Planned features long-term

- ☐ Get tracking input from sources other than a SenseGlove device.

- ⬛ Be able to assign behaviors to different objects (meshes) in the scene (e.g. Slider, Hinge, basic Grabables, etc).
- ⬛ Make it so developers can define or extend their own behavior(s) to an object through Code / Blueprint (e.g. I want a car door that is like a slider, but follows a path rather than a straight line).
- ⬛ Make the hand(s) able to push around physics-driven objects (for as much as their behaviors allow) (in backlog).
- ⬛ Be able to grab objects with up to 2 hands (and move them around with both hands at the same time in a way that seems realistic).
- ⬛ Ensure that our virtual hands (and the objects they hold) do not phase through other physics objects (e.g. walls and tables).
- ⬛ Allow other scripts to force a grab and/or release to occur (for example, when you place it apart at the designated location, it gets removed from your hand and snaps into place).
- ⬛ Have some form of weight simulation by making certain objects harder to push, lowering manipulation speed, or making objects only moveable with two hands.
- ⬛ (Optional) Make it so the fingers of your virtual hands do not clip inside the meshes you are holding (certain people see this as an indicator of how fast the Force-Feedback activates - but it's basically just rendering).

Changelog

All notable changes to this project will be documented in this file.

The format is based on Keep a Changelog, and this project adheres to Semantic Versioning.

[2.4.2] - 2025-02-17

This is a patch release to address minor issues in the SenseGlove Unreal Engine Handbook, with no modifications to the plugin code.

Documentation

- List the third-party tutorials with a description in their corresponding parent sections.
- Additional minor fixes and improvements that may not be listed here.

[2.4.1] - 2025-02-14

This is a patch release to address minor issues in the SenseGlove Unreal Engine Handbook, with no modifications to the plugin code.

Documentation

- Fix a bug that breaks the custom CSS styles on the Handbook's release URLs (e.g. `https://unreal.docs.senseglove.com/2.4/`) by reverting an unintentional change from the `v2.4.0`.
- Applied a minor Handbook Makefile fix.
- Additional minor fixes and improvements that may not be listed here.

[2.4.0] - 2025-02-14

This minor release brings various improvements and, notably, the first version to add support for VIVE standalone mode with ViveOpenXR compatibility.

Added

- Added the ePub version of the SenseGlove Unreal Engine Handbook.
- Added `FSGConnectJNI::Initialize()` and `FSGCoreJNI::Initialize()` return codes to Android logs for detailed debugging purposes through `adb logcat`.
- Introduced compatibility with the `ViveOpenXR` plugin.
- Added the `FSGPluginUtils` utility class for other plugins or modules availability detection such as `Meta XR` and `ViveOpenXR`.
- Added various HTC HMDs auto-detection support on Android.
- Added support for HTC VIVE Focus Vision HMD auto-detection.
- Added support for HTC Vive Wrist Trackers support on Android using the official `ViveOpenXR` plugin's OpenXR positional tracking provider `OpenXRViveWristTracker`.
- Added enum `ESGOpenXRPositionalTrackingProvider`.
- Added the `OpenXRPositionalTrackingProvider` option to the plugin's wrist-tracker settings to either manually set the positional tracking provider or auto-detect it based on a combination of tracker hardware settings, targeted platform, available OpenXR plugins, or the auto-detected HMD hardware.

Fixed

- Fix a critical issue where `HandStates->GetTransform(KeyPoint)` was incorrectly resolving to `(&HandStates[0])->GetTransform(KeyPoint)`, causing both hands to use the left hand's wrist transform under specific conditions. This occurred when the `bFallbackToHandTrackingIfNoGloveDetected` option was enabled, two gloves were present, and no hardware wrist-tracking device was active, resulting in both hands overlapping at the same transform.
- `FSGArrayUtils` optimizations that affect the plugin performance as a whole.
- Additional minor fixes and improvements that may not be listed here.

Changed

- Now the motion sources for the wrist-tracking hardware or hand-tracking are queried and populated dynamically rather than relying on the hardcoded `EControllerHand` enum. This allows the SenseGlove Unreal Engine Plugin to integrate better into other plugins such as `ViveOpenXR`, which when enabled, provides many more options as the motion source for their various wrist-tracking hardware.
- `FSGWristTrackingSettings::LeftHandMotionSource` and `FSGWristTrackingSettings::RightHandMotionSource` types have changed from `EControllerHand` to `FName`.
- Bumped the SenseGlove libraries to `v2.105.3-97ea18cb`.
- Bumped the SenseGlove Unreal Engine Marketplace Packager `v0.5.0-7df1183`.
- Bumped the copyright years.
- This is the last release to support Unreal Engine `5.2`. From `v2.5.x` onwards only UE `5.3` and newer will be supported.
- The `ESGViveHMDDetectionPriority` enum items have changed and are no longer backward-compatible.

Deprecated

- This is the last minor release to support Unreal Engine `5.2` and its support will be removed from the next minor or major releases.

Documentation

- Reintroduced the Handbook in ePub format.
- Revamped the Enabling XR_EXT_hand_tracking OpenXR Extension on VR Headsets and Deploying to Android (Standalone) documentation, and added the `ViveOpenXR`-compatibile instructions as well.
- Significantly improved the Setting Up the Wrist Tracking Hardware section by providing more detailed documentation and example configuration per HMD and wrist tracking hardware.
- Clarified how to set up the VIVE Wrist Trackers in various configurations.
- Added HTC VIVE specific optimization tips for running in standalone mode

- Fixed a few broken URLs caused by bad markdown formatting.
- Applied various Handbook Makefile fixes.
- Clarified the HTC VIVE standalone support status in the platform support matrix.
- Bumped the mdbook-alerts crate to `v0.7.x`.
- Reintroduced mdbook-epub at `d1536bbbdc1ca00320522ad73a967e15057ef573` from the `master` branch as the blocking issues in #115 have been address in `1ca2a860f6ed405c00914a3aadd8057d5050b29b`.
- Added third-party tutorials to the following sections: Enabling XR_EXT_hand_tracking on VR Headsets, Deploying to Android (Standalone), and OpenXR.
- Additional minor fixes and improvements that may not be listed here.
- List the VRExpansionPlugin demo in the Extra Resources section.

[2.3.2] - 2025-01-28

This patch release addresses a critical issue backported from the upcoming `2.4.x` release to `2.3.x`.

Fixed

- Fix a critical issue backported from the upcoming `2.4.x` release where `HandStates->GetTransform(KeyPoint)` was incorrectly resolving to `(&HandStates[0])->GetTransform(KeyPoint)`, causing both hands to use the left hand's wrist transform under specific conditions. This occurred when the `bFallbackToHandTrackingIfNoGloveDetected` option was enabled, two gloves were present, and no hardware wrist-tracking device was active, resulting in both hands overlapping at the same transform.

[2.3.1] - 2024-11-27

This patch release addresses a few issues with SenseGlove Sockets Editor.

Fixed

- Additional minor fixes and improvements that may not be listed here.

Changed

- The SenseGlove Sockets Editor now calculates hand bone reference transforms using the current virtual hand mesh being edited, rather than the reference mesh, when adding SenseGlove sockets.

[2.3.0] - 2024-11-13

This minor release includes some improvements and adds official Unreal Engine `5.5` Fab support.

Added

- Added `USGAndroidPermissions` to the `SenseGloveAndroid` module, enhancing the plugin's permission request process on Android. Now, a pop-up prompts the user to grant permissions, preventing silent crashes when permissions haven't been granted beforehand.
- Added Unreal Engine `5.5` Fab support.

Fixed

- Fix UE `5.5` deprecation warnings inside `USGVirtualHandComponent`.
- Additional minor fixes and improvements that may not be listed here.

[2.2.2] - 2024-11-08

This patch release addresses a few issues with both glove and hand-tracking.

Fixed

- Fixed a chain of critical bugs that gets triggered due to `GloveConnectivityCheckInterval` getting passed as seconds to the engine rather than milliseconds. Thus, the default or any large value for `GloveConnectivityCheckInterval` causes noticeable long delays between glove-connectivity-check intervals and consequently renders the hand-tracking state invalid in certain situations when the `bFallbackToHandTrackingIfNoGloveDetected` option is false.

[2.2.1] - 2024-10-23

This patch release focuses exclusively on updates to the documentation.

Documentation

- Updated all URLs, screenshots, and tutorials to reflect the transition from the Unreal Engine Marketplace to Fab, Epic's new unified content marketplace.
- Revised documentation now points to the new home of the SenseGlove Unreal Engine Plugin on Fab, ensuring users have access to the latest resources and information.

[2.2.0] - 2024-10-22

This is a minor release with some breaking API and ABI changes, focusing mainly on migrating away from the deprecated `FXRMotionControllerData` in favor of `FXRMotionControllerState` and `FXRHandTrackingState` on Unreal Engine `5.5+`.

Added

- Completed support for the upcoming Unreal Engine `5.5` release.
- Added `USGVirtualHandComponent::GetMotionControllerState()` and the equivalent Blueprint function `UVirtualHandComponentKismetLibrary::GetMotionControllerState` on UE `5.5+`.
- Added `USGVirtualHandComponent::GetHandTrackingState()` and the equivalent Blueprint function `UVirtualHandComponentKismetLibrary::GetHandTrackingState` on UE `5.5+`.
- Added `USGWristTrackerComponent::GetMotionControllerState()` and the equivalent Blueprint function `UWristTrackerComponentKismetLibrary::GetMotionControllerState` on UE `5.5+`.
- Added `USGWristTrackerComponent::GetHandTrackingState()` and the equivalent Blueprint function `UWristTrackerComponentKismetLibrary::GetHandTrackingState` on UE `5.5+`.
- Added a variant of `FSGDebugVirtualHand::Draw()` and the equivalent Blueprint function `USGDebugVirtualHandKismetLibrary::Draw_FXRHandTrackingState()` which accept `FXRHandTrackingState` on UE `5.5+`.
- Added the new member `bTracked` to the `FSGXRHandState` struct.
- Added `FSGXRTracker::GetMotionControllerState()` and the equivalent Blueprint function `USGXRTrackerKismetLibrary::GetMotionControllerState()`.
- Added `FSGXRTracker::GetHandTrackingState()` and the equivalent Blueprint function `USGXRTrackerKismetLibrary::GetHandTrackingState()`.

Fixed

- Additional minor fixes and improvements that may not be listed here.

Changed

- Replaced all internal usages of the `FXRMotionControllerData` struct with either `FXRMotionControllerState` or `FXRHandTrackingState` on UE `5.5+`.
- Deprecated `USGVirtualHandComponent::GetMotionControllerData()` on UE `5.5+`.

- Deprecated `USGWristTrackerComponent::GetMotionControllerData()` on UE `5.5+`.
- Deprecated the variant of `FSGDebugVirtualHand::Draw()` which accepts `FXRMotionControllerData` as a parameter on UE `5.5+`.
- Renamed `USGDebugVirtualHandKismetLibrary::Draw` to `USGDebugVirtualHandKismetLibrary::Draw_FXRMotionControllerData` for more clarification.
- Renamed an `FSGXRHandState` member from `bReceivedJointPoses` to `bHasReceivedJointPoses`.
- Changed the `FSGXRTracker::GetAllKeypointStates()` signature on UE `5.5+` to match the `IHandTracker` interface API changes.
- The animation system on UE `5.5+` has been revamped to utilize `FXRHandTrackingState` instead of `FXRMotionControllerData`.
- The wrist tracking system on UE `5.5+` has been revamped to utilize `FXRHandTrackingState` instead of `FXRMotionControllerData`.
- The hand interaction manipulation on UE `5.5+` has been revamped to utilize `FXRHandTrackingState`.
- The virtual hand debugging system on UE `5.5+` has been revamped to utilize `FXRHandTrackingState`.

Documentation

- Added the documentation on consuming the `FXRHandTrackingState` struct in both Blueprint and C++.
- Updated the documentation on consuming the `FXRMotionControllerData` struct.
- Additional minor documentation fixes and improvements that may not be listed here.

[2.1.4] - 2024-10-22

This is a bugfix release that delivers some documentation fixes.

Documentation

- Updated the documentation on consuming the `FXRMotionControllerData` struct.
- Additional minor documentation fixes and improvements that may not be listed here.

[2.1.3] - 2024-10-11

This bugfix release centers on adding initial support for the upcoming Unreal Engine `5.5`.

Added

- Added initial support for the upcoming Unreal Engine `5.5` release. Please note that, while the plugin is functional, a few adjustments are still required to address deprecation warnings. Specifically, the `FXRMotionControllerData` struct needs to be replaced with the newly introduced `FXRMotionControllerState` and `FXRHandTrackingState` structs, along with adjustments to adhere to the new hand-tracking API changes.
- Added support for Epic Native Toolchain `v23`.

Fixed

- Fix a bug inside `USGVirtualHandComponent::PostEditChangeProperty()` where the get member name check happens against the wrong class and member names.
- Additional minor fixes and improvements that may not be listed here.

Changed

- The SenseGlove libraries have been updated to `v2.105.0-02a2e508`.

[2.1.2] - 2024-09-02

This is a bugfix release that addresses a few non-critical issues and documentation fixes.

Fixed

- Fix a bug where the hands are always visible even when `bVisibleWhenHandDataUnavailable` is disabled.
- Fix a bug where the `HandVisibilityChangedEvent` event is not triggered on the virtual hand component visibility changes.
- Fix the wrong script name for `USGHMDTrackerKismetLibrary`.
- Fix the wrong script name for `USGXRTrackerKismetLibrary` .
- Fix `LogPython: Warning: 'SGHMDTrackerKismetLibrary' and 'SGXRTrackerKismetLibrary' have the same name (SenseGloveHeadMountDisplayKismetLibrary) when exposed to Python. Rename one of them using 'ScriptName' meta-data` when packaging the game.
- Fix the non-existent default hand-mesh warnings polluting the logs when packaging the game.
- Expanded the clickable area on the handbook index page revision buttons.
- Minor documentation fixes.

[2.1.1] - 2024-08-18

This is a bugfix release with no actual plugin code changes, mostly addressing issues in the documentation and third-party dependencies caused by source control merge conflicts.

Fixed

- Fix the messed up changelog file caused by cherry-picking merge conflicts between the dev branch and the master branch.

- Fix a bug that causes a handbook revision mismatch when deploying the handbook from the dev branch.
- Fix a bug where `SG_GIT_IS_SHALLOW_CLONE` while building the handbook is always set to `yes` even if it's not a shadow clone because `SG_DOT_GIT_SHALLOW_FILE` evaluates to an empty string when the `.git/shallow` file does not exist.
- Fix some documentation typos.

Removed

- Removed Android NDK r25 armv7 and x86 dependencies brought back by mistake while merging v2.1.0 from the dev branch to the master branch.

[2.1.0] - 2024-08-16

This is a minor release focusing mainly on bringing OpenXR-compatible hand tracking support (`XR_EXT_hand_tracking`) and Head-mounted Display automatic detection for adjusting wrist tracker offsets automatically at runtime.

Added

- Added SenseGloveTracking and module which provides OpenXR-compatible hand tracking by implementing `XR_EXT_hand_tracking` support, HMD auto-detection, and SenseGlove device tracking.
- Added USenseGloveTrackingKismet module in order to expose part of the SenseGloveTracking functionality to Blueprint.
- Added FSGXRTracker, the underlying main class that implements the OpenXR compatibility.
- Added USGXRTrackerKismetLibrary in order to allow Blueprint to retrieve the FXRMotionControllerData directly from our tracking module.
- Added the SGTrackingTypes header to the SenseGloveTypes module in order to define and share SenseGloveTracking module types through this header across the plugin modules.

- A fallback to HMD and wrist tracker hardware auto-detection mechanism has been added to be triggered in situations when automatic detection of the wrist tracker hardware is desired, e.g., either by not setting it explicitly, or setting it to the default None value. Please note that this is still highly experimental and HTC VIVE Focus 3 and HTC XR Elite cannot be distinguished in the current iteration. Though, since the tracker devices and offsets for both headsets are the same in the end it does not make a difference if both headsets are detected as each other.
- Added ESGHeadMountedDisplayDevice enum with supported HMDs list.
- Added ESGViveHMDDetectionPriority enum in order to choose which headset we attempt to detect between VIVE Focus 3 and VIVE XR Elite as we cannot distinguish them, yet.
- Added the FSGHMDTracker utility class, in order to easily gather information about the HMD device at runtime.
- Added USGHMDTrackerKismetLibrary which exposes the equivalent C++ HMD auto-detection functionality to Blueprint.
- Added FSGHMDTrackingSettings config struct.
- Added the FSGGloveTracer utility class, in order to easily check the left or right glove connectivity or retrieve the connected glove instances.
- Added USGGloveTrackerKismetLibrary which exposes the equivalent C++ functionality to Blueprint.
- Added FSGGloveTrackingSettings config struct.
- Added FSGTrackingSettings config struct.
- Added FSGHandTrackingSettings config struct.
- Added FSGWristTrackingDebuggingSettings config struct.
- Added FSGVirtualHandSettings config struct.
- Added FSGVirtualHandAnimationSettings config struct.
- Added FSGVirtualHandDebuggingSettings config struct.
- Added FSGVirtualHandGrabSettings config struct.
- Added FSGVirtualHandHapticsSettings config struct.
- Added FSGVirtualHandMeshSettings config struct.
- Added FSGVirtualHandPhalangesLengthSettings config struct.
- Added FSGVirtualHandTouchSettings config struct.
- Added USGVirtualHandComponent::OnHandVisibilityChanged() event in order to notify other components/actors whenever the virtual hand mesh appears or disappears (for example, this could happen when a glove is connected/disconnected).

- GetMotionControllerData() has been introduced to the USGVitualHandComponent in order to retrieve the OpenXR-compatible glove data in Unreal's FXRMotionControllerData format.
- Added FSGVirtualHandAnimInstanceProxy::GetMotionControllerData and many more accessor methods usable only by child classes to allow consumption of the data required for manipulating the virtual hand mesh animations.
- GetMotionControllerData() has been introduced to the USGWristTrackerComponent in order to retrieve the OpenXR-compatible glove data in Unreal's FXRMotionControllerData format.
- Added USGGrabComponent::SimulatePhysics() method.
- Added FSGDebugCube.
- Added FSGDebugCubeSettings.
- Added the SenseGloveDebugKismet module in order to allow drawing of debugging, cubes, gizmos, and virtual hands from Blueprint.
- Added USGDebugCubeKismetLibrary in order to expose the FSGDebugCube functionalities to Blueprint.
- Added USGDebugGizmoKismetLibrary in order to expose the FSGDebugGizmo functionalities to Blueprint.
- Added USGDebugVirtualHandKismetLibrary in order to expose the FSGDebugVirtualHand functionalities to Blueprint.
- Added a new static Draw() method overload to DebugGizmo which allows passing an FQuat instead of a FRotator.
- Introduced a new FXRMotionControllerData compatible hand animation system with the ability to take the mesh bone's transforms into account for a more reliable hand animation.
- Introduced a new FXRMotionControllerData compatible wrist tracking system.
- Introduced a new FXRMotionControllerData compatible hand interaction manipulation system.
- Added the ability to fallback to hand tracking when a glove is not present and use the bare hands for interactions, or a combination of glove and hand tracking if no motion controller input is detected.
- Added the SenseGlove grab/touch sockets one-click-setup ability on any Epic-compliant virtual hand mesh from within the Unreal Editor's Content Browser, Skeleton Editor, or Skeletal Mesh Editor by extending the Unreal Editor.
- Added FSGAssetUtils editor-only class.
- Added FSGContentBrowserExtension editor-only class.
- Added FSGPluginStyle editor-only class.

- Added FSGSocketsEditor editor-only class.
- Added FSGSocketsEditorCommands editor-only class.
- Added the FSGInitializationSettings config struct in order to control how the plugin is initialized.
- Introduced the FSGGameUserSettings for managing the Engine Scalability Settings through the SenseGlove plugin in order to change the graphics settings on the fly.
- Added USGGameUserSettingsKismetLibrary in order to allow all the Engine Scalability Settings to be managed from the Blueprint side.
- Added FSGGameUserSettingsSettings config struct.
- Added the SenseGlove console commands: SG_GetEngineScalabilitySettings() and SG_SetEngineScalabilitySettings(Scalability).
- Added SGHardwareBenchmarkingSettings config struct.
- Introduced ESGEngineScalabilitySettings enum.
- Added FSGVirtualHandSettingsOverrides config struct used by the new settings override system.
- Added SGWristTrackingSettingsOverrides config structured by the new settings override system.
- Added support for Android API level 32 in addition to the API level 29.
- Introduced the SenseGlove Unreal Engine Handbook as an attempt at documenting the SenseGlove Unreal Engine Plugin.
- Merged the `pack` utility branch to the plugin's source code at `/Packager` which adds the SenseGlove Unreal Engine Marketplace Packager v0.4.0-a65bb20 binaries and configurations.

Fixed

- Fixed a bug when the virtual hand inside the game is not visible but still collides with other objects inside the scene, mistakenly triggering events like OnGrabStateUpdated and OnTouchStateUpdated.
- Fixed a bug where USGGrabComponent's `bAffectPhysicsState` does not enables physics on its owning actor at `BeginPlay()`.
- Fixed various wrong Kismet script names and their class exports.
- Fixed the display name for various overloads of the Blueprint-exposed function `Queue Command Vibro Level` to expose sensible display names.
- Some Android UPL tweaks, permission, and build fixes.

- Many other large and small fixes and improvements that might not be listed here.
- A few small bugfixes that have already been backported to the v2.0.x series.

Changed

- Now, if bValidateIfDefaultClassesAreSGCompliant option from FSGInitializationSettings is enabled (default) the SenseGlove plugin checks for default SenseGlove-compliant GameMode, GameInstance, etc, at module initialization and tries to set to default, native SenseGlove classes, if any of those default classes are not a SenseGlove or a SenseGlove-derived class.
- The USGSettings has been fully revamped with more customizations added and categorized in a different manner adding many new structs and removing some, in order to have fine-grained control over the various aspects and functionality of the plugin components.
- The USGSettings constructor visibility has been changed from public to private.
- The Settings override system has been overhauled as well affecting how we override settings from the USGVirtualHandComponent and USGWristTrackerComponent.
- The SenseGlove libraries have been updated to v2.104.1-55fddbd2.
- GetHandPose() has been replaced by GetMotionControllerData inside USGVirtualHandComponent (see the relevant entry in the Added and Removed sections).
- Many functions inside USGVirtualHandComponent for retrieving bone names or reference transforms has been renamed to return different data types; e.g. GetLeftHandFingerBoneNames(), GetRightHandFingerBoneNames(), GetLeftHandFingerBoneName(), and GetRightHandFingerBoneName() renamed to GetLeftHandBoneNames(), GetRightHandBoneNames(), GetLeftHandBoneName(), and GetRightHandBoneName() respectively.
- bHiddenInGameIfNoGloveDetected UPROPERTY from USGVirtualHandComponent has been renamed to bVisibleWhenHandDataUnavailable and accordingly all of its getters and setters; bVisibleWhenHandDataUnavailable = false now acts as bHiddenInGameIfNoGloveDetected = true, and vice-versa.
- USGWristTrackerComponent now uses FXRMotionControllerData for wrist tracking instead of calculating the wrist location by calling the SenseGlove API.

- FSGVirtualHandAnimInstanceProxy now relies on FXRMotionControllerData to animate the hands instead of a TMap of bone names and rotations which allows it to also apply the bone locations.
- The new OpenXR animation system now takes into account the mesh bone's transforms for a more reliable hand animation.
- FSGDebugVirtualHand::Draw now accepts a FXRMotionControllerData parameter instead of all WristLocation, WristRotation, JointPositions, and JointRotations parameters.
- FSGDebugVirtualHandSettings has been renamed to FSGVirtualHandDebuggingSettings.
- The value for USGGrabComponent's AttachmentSocketName uproperty now defaults to the value of the plugin's GrabAttachPointSocketName instead of Name_NONE.
- The USGGrabComponent now enables `bGravityEnabled`, `bSimulatePhysics`, and calls `WakeRigidBody` on its owning actor at `BeginPlay()` if `bAffectPhysicsState` is enabled.
- Updated the Directory Structure section of the main README file to reflect the latest toolchain support status.
- The `/CHANGELOG.md` file has been migrated to `/Handbook/src/overview/changelog.md`
- The `/LICENSE.md` file has been migrated to `/Handbook/src/license/senseglove-unreal-engine-plugin.md`
- The `/LICENSE-THIRD-PARTY.md` file has been migrated to `/Handbook/src/license/third-party.md` and every third-party component's license has been split; adding `/Handbook/src/license/senseglove-sdk.md` for the SenseGlove SDK, `/Handbook/src/license/boost-cpp-libraries.md` for the Boost C++ Libraries, and `/Handbook/src/license/serial-communication-library.md` for the Serial Communication Library.
- The Platform Support Matrix section of the main README file has been migrated to `/Handbook/src/overview/platform-support-matrix.md`.
- The Planned Features Completion Status section of the main README file has been migrated to `/Handbook/src/overview/planned-features-completion-status.md`.
- The Directory Structure section of the main README file has been migrated to `/Handbook/src/overview/directory-structure.md`.

- The SenseGlove settings' main config struct is now marked as DefaultConfig which means it does not require to be saved when settings are changed and they take effect immediately as the user updates them.
- Replaced all bitfield uproperties with booleans.
- Changed the DocsURL from the old Blueprint docs website to the new SenseGlove Unreal Engine Handbook website.
- The Blueprint signature for various overloads of the Blueprint-exposed function `Queue Command Vibro Level` has been changed to expose sensible display names.

Removed

- Dropped support for Unreal Engine 5.1 and Epic Native Toolchain v20 (used to build UE 5.0 and 5.1 Linux dependencies).
- Removed the Allbreaker virtual hand model as it's no longer compatible with the SenseGlove plugin.
- Removed ASGVirtualHandActor as it was experimental and we no longer maintain it and haven't been doing so for a long time.
- Removed FSGVirtualHandAnimInstanceProxy::GetBonesRotations().
- Removed USGVirtualHandComponent::GetHandPose() and it's no longer possible to get the hand pose data from USGVirtualHandComponent as GetHandPose() has been removed. If you need it, you could always use the SenseGlove low-level API to retrieve it from the glove.
- Removed also GetFingerBoneName(), GetFingerBoneRefTransform(), GetFingerBoneRefRotation() and GetFingerBoneRefRotation() from USGVirtualHandComponent.
- Removed some remnants of UE 5.1 and older releases from the C++ code.
- Removed the `pack` utility branch and merge it to the plugin's source code at `/Packager`.

Known Issues

- With the new OpenXR release, the separation of the real and virtual hand rendering is broken. The reason is the animation system now uses the OpenXR data in the world transforms which yields better animations, but comes at the

cost of overriding the the hand position set by the wrist tracker component's position and rotation. If `FXRMotionControllerData` is invalid and `bVisibleWhenHandDataUnavailable` is enabled for example, the system works as expected, since the animation system won't proceed to animate the hand meshes without valid `FXRMotionControllerData`. Since the animation system is only aware of the hand mesh it's animating versus the real hand and virtual hand meshes it means either it should become aware of the physics events like begin and end overlap events and also the real vs virtual hands, or it should resort back to animating the virtual hand meshes in local or component space. This release marks this feature as broken for now until we come up with a reasonable solution in the future.

- The `UXRDeviceVisualizationComponent` provided by Unreal Engine is used in the SGPawn class as ControllerVisualizerLeft and ControllerVisualizerRight for implementing the wrist tracking hardware visualization feature. However, it is not compatible with the new OpenXR system in certain scenarios. For instance, when the motion controllers serve as wrist tracking hardware since the SenseGlove plugin is now introduced to the engine as an `OpenXRHandTracking` system, it causes the UXRDeviceVisualizationComponent to visualize the wrist tracking hardware at coordinates (0.0f, 0.0f, 0.0f) instead of their actual location and rotation in the world. This happens because the component incorrectly registers them as inactive, possibly because it's assumed hand tracking and motion controllers cannot be in use at the same time. Currently, we use this feature solely for debugging, and we have an alternative in the form of wrist-tracking debug gizmos, which can be toggled on or off via the settings system. In future releases, we might remove this feature due to its incompatibility, unless we find a solution to make the `UXRDeviceVisualizationComponent` work with the new system. Alternatively, we may develop our own version of the `UXRDeviceVisualizationComponent`.
- Although the SenseGlove OpenXR implementation is fully compatible with the `IOpenXRHMD` interface and the `FOpenXRHMD XRTrackingSystem`, it is not compatible with the `FOculusXRHMD` backend provided by the Meta XR plugin. The same issue likely applies to the VIVE OpenXR plugin. So, if these plugins are enabled in your project, the SenseGlove OpenXR will not function as intended, effectively breaking the plugin's functionality. It seems these plugins are necessary in order to make the fallback to the hand-tracking feature work on Android. While we may add support and compatibility with Meta XR and VIVE

OpenXR plugins in the future, for the time being, if your project requires these plugins, we advise continuing with the `v2.0.x` release of the SenseGlove Unreal Engine plugin until this issue is addressed.

[2.0.8] - 2024-07-15

This is a bugfix release that contains a somewhat important bugfix backported from the next release of the plugin as documented below.

Fixed

- Fix a bug where the SGPawn right-hand grab colliders' default size is mistakenly set to the default value for the left-hand grab colliders at CDO initialization time.

[2.0.7] - 2024-05-29

This is a bugfix release with no actual plugin code changes, only fixing issues with binary assets incompatible with UE versions earlier than 5.4.

Fixed

- Make the Allbreaker assets compatible with UE5.1+ again as the v2.0.5 update breaks compatibility with UE versions earlier than 5.4, thus leaving the engine unable to load those assets.

[2.0.6] - 2024-05-29

This is a bugfix release with no actual plugin code changes, only removing development/test assets from UE 5.3 that were never meant to be shipped.

Removed

- Removed the dev/test virtual hand models that leaked into the 5.3 branch.

Fixed

[2.0.5] - 2024-05-22

This is a bugfix release with no actual plugin code changes, only focusing on fixing the Allbreaker virtual hand model issues.

Fixed

- Fix the wrong palm bone names on the Allbreaker virtual hand models.

[2.0.4] - 2024-05-17

This is a bugfix release with no actual plugin's code change.

Fixed

- Fix our in-house Unreal Engine Marketplace submission tool's configurations where the Content folder (containing the Allbreaker hand model) is mistakenly ignored during the submission. This release reintroduces the Virtual Hand Model and its material missing from the previous release.
- Fix the SenseGlove.uproject's wrong versioning submitted to the Unreal Engine Marketplace.

[2.0.3] - 2024-05-15

This is a bugfix release addressing mostly RunUAT build issues on Unreal Engine 5.4.

Fixed

- Fix UE 5.4 RunUAT build issue: "Asking CppCompileEnvironment for a single Architecture, but it has multiple Architectures (arm64, x64)", affecting SenseGloveConnectImpl and SenseGloveCoreImpl modues.
- Improved target platform detection when building SenseGloveConnectImpl and SenseGloveCoreImpl modules and also distinguishing the x64 builds from arm64 on Microsoft Windows.
- Fix other UE 5.4 RunUAT build issues, mostly caused by missing headers.

Removed

- Removed support for Android armeabi-v7a and x86 architectures as they are no longer supported by the supported engine versions.

[2.0.2] - 2024-04-25

This is a patch release with no code changes.

Added

- Introduce official Unreal Engine 5.4 support to the Unreal Engine Marketplace.

Changed

- Updated the Platform Support Matrix with the latest changes. This is the last release to support Unreal Engine 5.1 as we no longer are able to push updates

for this release to the Unreal Engine Marketplace. The v2.0.1 release for Unreal Engine 5.1 can be obtained from the Unreal Engine Marketplace, and v2.0.2 through our Microsoft Azure DevOps repositories. Please note that there are no actual code changes between these two releases and in terms of functionality they are almost identical.

[2.0.1] - 2024-04-15

This is a bugfix release.

Fixed

- Fix a bug inside both SGVirtualHandComponent and SGWristTrackerComponent where the connected glove's UObject instance gets destroyed and re-instantiated every frame. With this fix now the glove instance will be created or destroyed only when a glove connects to or disconnects from the system.
- Update the outdated Platform Support Matrix and its remarks section to reflect the latest status information.
- Fix the wrong header file description sections for the header files inside `SenseGloveKismet/Public/SGKismet/`.

Changed

- SenseGlove libraries have been updated to v2.102.0-35d4de3f.
- Together, SenseGlove libraries v2.102.0-35d4de3f and SenseCom v1.6.1 remove the need to call ResetCalibration every time and are able to store and load calibration profiles from disk.
- SesenGloveBackend module is no longer calling FSGHandLayer::ResetCalibration on every backend initialization.

[2.0.0] - 2024-03-22

This is the second major release of the SenseGlove Unreal Engine Plugin adding support for Nova 2 with enormous breaking changes to the current C++ and Blueprint APIs.

Added

- Added support for the SenseGlove Nova 2 devices.
- Added support for Quest 3 controllers.
- Various classes have been added to the API in order to implement the new functionalities and features from the latest upstream SenseGlove libraries.
- Added initial support for the upcoming Unreal Engine 5.4 release.
- Added a pair of default production-ready virtual hand meshes for the left and right hands, courtesy of Allbreaker LLC Columbia. For usage and redistribution, please consult the LICENSE-THIRD-PARTY.md file.

Fixed

- A few critical bug fixes that have already been backported to the v1.x.x series through v1.9.3 to v1.9.8 releases.
- Revamped the way we do FVector <-> SGVect3D, FQuat <-> SGQuat, and SenseGlove <-> Unreal Engine angles conversions in order to properly translate between the SenseGlove and Unreal Engine coordinate systems.
- Allow the C++ compiler the opportunity to perform RVO/NRVO if applicable.
- Fix the modules' order inside the .uplugin file.
- Fix a build issue inside FSGArrayUtils::FromStdVector introduced by newer MVSC updates due to stricter implicit uint64 to int32 conversions.
- Fix a build issues inside FSGArrayUtils when performing non-Unity builds due to the missing header.
- Fix other build issues in USGDevice, USGNovaGloveSensorData, FSGDeviceImpl, and FSGSenseGloveVarsImpl when performing non-Unity builds due to the missing relevant headers.
- Fix changelog formatting.
- Some other improverment and fixes.

Changed

- SenseGlove libraries have been updated to v2.101.12-62b1be11.
- The SenseGlove Unreal Engine Plugin now declares the OpenXR plugin as a dependency, so that the OpenXR plugin will be enabled automatically as soon as the SenseGlove Unreal Engine Plugin gets enabled.
- Various classes and parts of the API have been changed in order to reflect and adhere to upstream SenseGlove libraries.
- Reverse the Platform Support Matrix order from newer Unreal Engine versions to the older ones.
- Clarify the engine support policy in the main readme file by adding the corresponding references from the Epic Marketplace Guidelines and a URL to their guidelines page.
- The SGTouchComponent uproperties BuzzDuration and BuzzLevel now utilize different different names in order to correspond to the underlying API changes. They have been renamed to VibrotactileDuration and VibrotacktileLevel.
- The SGTouchComponent uproperties ForceFeedbackLevel and BuzzLevel (now VibrotacktileLevel) parameters type have changed from int32 to float with the value range varying between 0.0f to 1.0f instead of 1 to 100 in order to correspond to the underlying API changes.
- The SGVirtualHandComponent now assumes the default grab point's name as GenericGrabPoint instead of GrabPoint as default if not specified in the Unreal Blueprint Editor.
- The SGPawn on UE 5.2+ now utilizes UXRDeviceVisualizationComponent in order to properly display the controller meshes shipped with Unreal Engine's OpenXR plugin, or a user-provided mesh. On UE 5.1 this could still be set on the WristTrackerLeft and WristTrackerRight components. Please note that despite the fact that on UE 5.2+ it's still possible to utilize the WristTrackerLeft and WristTrackerRight for setting the controller meshes, this has been deprecated in UE 5.2+ and is no longer supported.

Removed

- Various classes and parts of the API have been removed in order to reflect and adhere to upstream SenseGlove libraries.
- Removed the redundant SGIC_int32_Ref interop type.

[1.9.8] - 2024-03-12

This is a bugfix release that contains bugfixes backported from the next major release of the plugin as documented below.

Fixed

- Fix a bug where the right-hand mesh is always hidden inside the game no matter whether the right glove is connected or not.
- Fix a crash inside the USGHandPose::FromHandAngles method.
- Some performance optimizations by utilizing MoveTemp in return statements.
- Some improvements applied to the source code.
- Some other minor fixes.

Changed

- The BonesRotations TMap is no longer a public field of FSGVirtualHandAnimInstanceProxy and instead could be retrieved by calling the GetBonesRotations() method.

[1.9.7] - 2024-02-18

This is a bugfix release that contains bugfixes backported from the next major release of the plugin as documented below.

Fixed

- Fix various bugs inside the SGPlayerController which occur when the thumb and pinky fingers are simultaneously touching different SGTouchComponents, or only one of them is in touch with such a component. In this case pinky's buzz and force-feedback levels are determined from the SGTouchComponent that is in collision with the thumb instead of the one that is touched by the pinky. Or,

the pinky could ignore the buzz and force-feedback level if the thumb is not in collision with an SGTouchComponent. Or, the pinky could have reacted with a buzz or force feedback while only the thumb is in contact with an SGTouchComponent.
- Fix the BuzzDuration UPROPERTY range in order not to get clamped at 100.0f and also use float values for ClampMin and UIMin specifiers instead of integer values.

[1.9.6] - 2024-02-14

This is a bugfix release.

Fixed

- Fix a few critical bugs inside the NovaGlove class where the higher levels of the API including constructors, Parse, and NewNovaGlove methods mistakenly instantiate a SenseGloveImpl class instead of a NovaGloveImpl class.

[1.9.5] - 2024-02-09

This is a bugfix release.

Fixed

- Fix a wrong type-casting inside SGDeviceModel::ParseFirmware where OutMainVersion and OutSubVersion arguments are getting passed to the lower levels of the API. This could potentially result in a segfault at the FFI boundary between lower and higher levels of the API.

[1.9.4] - 2024-02-08

This is a bugfix release addressing mostly Blueprint API issues with ABI breaking changes inside the Blueprint layer, backported from the next major release of the plugin as documented below.

Fixed

- Fix the Blueprint Parse function signature for the NovaGloveInfoKismetLibrary where the OutGloveInfo passed by the caller was never actually assigned as it was not getting passed by reference.
- Changelog formatting.

[1.9.3] - 2024-02-03

This is a hotfix release addressing a few critical issues that might result in crashes or malfunctions for users of the low-level SenseGlove API, backported from the next major release of the plugin as documented below.

Fixed

- Fix a potential memory corruption inside one of the SGBasicHandModel constructors where the StartPositions parameter gets passed as the StartRotations parameter to lower levels of the API.
- Fix a potential memory corruption inside one of the SGSenseGloveInfo constructors where the StartPositions parameter gets passed as the Functions parameter to lower levels of the API.
- Fix a potential memory corruption where inside the SGHapticGloveCalibrationSequence::GetCurrentInstruction method, the return statement of the function is getting assigned to the const parameter NextStepKey, thus the return statement of the function will always be empty as well.

- Fix a potential memory corruption where inside one of the overloads of the SGSenseGloveImpl::GetGlovePose method, the out parameter of the method is getting passed as the SensorData parameter to the lower levels of the API.
- Fix multiple Equals methods for a few classes such as SGInterpolationSet, SGNovaGloveHandProfile, SGNovaGloveInfo, SGSenseGloveHandProfile, SenseGloveInfo, SenseGlovePose, where the Equal method compares the current instance against itself instead of the other instance passed to as the parameter to the method.
- Removed a redundant code statement inside the SGNovaGloveImpl::GetSubFirmwareVersion method.
- Some minor const correctness fixes.
- Some other minor code fixes and improvements.
- Fix the wrong version numbers inside the paltform support matrix and the main .uplugin file.
- Minor changelog fixes.
- Bumped the copyright years.

[1.9.2] - 2023-11-03

Added

- Added a list of planned features and their completion status to the main README file.

Fixed

- A bug where the released actor is going to be NULL whenever the OnActorReleased event fires.

[1.9.1] - 2023-10-11

Fixed

- Add the missing Unreal Engine C++ header to files that rely on the ENGINE_*_VERSION macros in order to fix the Epic Store build failures on UE 5.3.

[1.9.0] - 2023-10-10

Changed

- The BlueprintImplementableEvent UFUNCTION specifier for the OnGrabStateUpdated, OnTouchStateUpdated, OnActorGrabbed, OnActorReleased, OnActorBeginTouch, and OnActorEndTouch events have been changed to BlueprintNativeEvent in order to allow them to be implemented from the child C++ classes as well. This won't break any existing Blueprint code that relies on the previous BlueprintImplementableEvent signature.

Fixed

- Add a missing release note entry for the v1.8.0 release to the changelog file.

[1.8.0] - 2023-10-10

Added

- Introduced new SGPawn events: OnActorGrabbed, OnActorReleased, OnActorBeginTouch, and OnActorEndTouch.
- Exposed OnGrabStateUpdated, OnTouchStateUpdated, OnActorGrabbed, OnActorReleased, OnActorBeginTouch, and OnActorEndTouch events to Blueprint as BlueprintImplementableEvent.

Fixed

- Fix a bug where the OnTouchStateUpdated event is mistakenly triggered instead of the OnGrabStateUpdated when the right thumb fingertip grab collider overlaps with a grabbable actor.
- Fix the DECLARE_EVENT macro signature for OnGrabStateUpdated and OnTouchStateUpdated events.

[1.7.0] - 2023-09-14

Added

- Introduce SGGameInstance, a customized SenseGlove game instance for future use.
- Added the new SenseGloveBackend and SenseGloveBackendKismet modules.
- Added SG_CPP20 C++ macro for C++20 detection, which is now default from UE 5.3 onwards.
- Added SG_CAPTURE_THIS C++ macro as a workaround for error C4855: implicit capture of 'this' via '[=]' is deprecated in /std:c++20 in order to build the same lambda captures without extra #ifdefs on all supported engine versions.

Changed

- SenseGlove libraries have been updated to v2.12.0-19c9854.
- SGCoreImpl/SGPlatform has been moved to SGBuildHacks/SGPlatform.

Fixed

- Proper initialization of the SenseGlove backend in order to fix a bug in certain situations where SGConnect::Init() gets called every frame.
- Some other minor fixes and improvements.

[1.6.1] - 2023-08-14

Fixed

- Fix Unreal Engine 5.0 build issues.
- Minor documentation fixes.

[1.6.0] - 2023-08-14

Added

- Added support for the upcoming Unreal Engine 5.3.
- Now, the hand's velocity is applied to grabbed actors after being released from the hand.
- Introduce the real hands to the SenseGlove module (SGPawn) API.
- Added separation of the virtual and real hand rendering.

Fixed

- Fix the wrong default debug virtual hand gizmo colors when initialized using the default constructor.
- Some minor performance fixes and improvements.

Changed

- SenseGlove libraries have been updated to v2.11.0-b775a05.

[1.5.3] - 2023-07-19

This is a hotfix release mostly addressing Android Bluetooth performance issues.

Fixed

- Minor changelog fixes.

Changed

- SenseGlove libraries have been updated to v2.10.1-3b0e7c9.

[1.5.2] - 2023-07-19

This is a hotfix release mostly addressing Android-related issues.

Fixed

- Fix a build issue with Android shipping builds due to sgconnect.jar not getting copied automatically in the AFSProject which is compiled for shipping builds

when AndroidFileServer (AFS) is enabled.
- Minor changelog fixes and some source code formatting fixes.

[1.5.1] - 2023-07-13

This is a hotfix release addressing a few critical issues introduced by the recent changes.

Fixed

- Fix a wrist tracker bug where left and right hands' wrist trackers are mistakenly tracking the opposite hand's motion source.
- Fix a bug where the right hand is not able to do grab or release.

[1.5.0] - 2023-06-16

This release breaks ABI/API compatibility with the previous versions in some areas as documented below.

Added

- Added HTC VIVE Focus 3 positional tracking hardware enum.
- Added support for the Meta Quest Pro, HTC VIVE, and HTC VIVE Focus 3 positional tracking hardware.
- Added two options to the wrist tracker settings (to the global plugin settings and the overrides in the wrist tracker component) in order to be able to specify a custom motion source for the left and right hands, so that it allows SteamVR-based trackers such as HTC VIVE or HTC VIVE Focus 3 to operate with the SGPawn.

Fixed

- Fix a bug where SteamVR trackers such as HTC VIVE and HTC VIVE Focus 3's wrist orientation and location were not being tracked.

Changed

- Fully refactored the top-level configurations in the settings system into USTRUCTs.
- SenseGlove libraries have been updated to v2.10.0-12133ac.

Removed

- Dropped support for the Epic Native Toolchain v19, MSVC v141 (Visual Studio 2017), and thus Unreal Engine 4.27 as it has been marked as deprecated since v1.4.x.
- Removed any kind of support for Oculus Touch (Oculus Rift S and Oculus Quest 1) positional tracking hardware, thus the enum as well.
- Removed any kind of support for Pico Neo 2 positional tracking hardware, thus the enum as well.
- Removed any kind of support for Pico Neo 3 positional tracking hardware, thus the enum as well.

[1.4.3] - 2023-06-01

This is a hotfix release addressing a critical Android crash.

Fixed

- Fix a critical Android crash that happens where the default development hand meshes are not found, which means almost always since we don't ship any default virtual hand mesh at the moment.

- Minor changelog release formatting fix in order to stay consistent.

[1.4.2] - 2023-06-01

This is a hotfix release addressing a few critical issues.

Fixed

- Fix build issues with certain compilers when the Unreal Engine version is older than 5.2.
- Reintroduced the Virtual Hand and the Wrist Tracker debug gizmos which have temporarily been disabled due to a bug in the settings system.
- Some minor changelog fixes.

[1.4.1] - 2023-05-29

This is a bugfix release with a focus on Android build issues.

Fixed

- Fix an Android Gradle build issue that happens when the game's package name won't start with com.senseglove.*.
- Suppress a grade warning for non-arm64 architectures when the build target is Android.

Removed

- Remove dead Gradle code from the Android module.

[1.4.0] - 2023-05-19

This release breaks ABI/API compatibility with the previous versions.

Added

- Added support for the stable release of Unreal Engine 5.2 (the preview release has been supported since v1.2.0).
- Added Linux AArch64 platform support.
- Added a new Grab component that can turn any actor into a grabbable object.
- Added a new Touch component that enables haptic feedback such as Buzz and Force-Feedback commands.
- Added an optional feature in order to automatically stop all haptics on the EndPlay event, wherever the virtual hand component is used. By default, it's enabled.

Fixed

- Fix Blueprint signatures for USGVirtualHandComponentKismetLibrary and make all the Blueprint exposed functions static.

Changed

- SenseGlove libraries have been updated to v2.7.1-965f90c with support for Linux AArch64.
- The Virtual Hand and the Wrist Tracker debug gizmos (the intended use is only for SenseGlove developers for really low-level stuff; thus won't affect the users of the plugin at all) have been disabled and will be ignored due to an esoteric bug in the settings systems which has been scheduled to be fixed in the future releases.

Removed

- Removed the redundant SenseGloveCoreTypes module which causes all kinds of packaging issues with certain versions of the engine.

Deprecated

- This is the last release to support Unreal Engine 4.27 and please keep in mind that the current release is not obtainable through the Unreal Engine Marketplace. The latest published version on the Marketplace for 4.27 is v1.3.1. Per Epic's Marketplace policy regarding Code Plugins, we are only able to distribute or update the SenseGlove plugin for the last 3 stable versions of Unreal Engine. As a result, we won't be able to publish updates or bug fixes for the older versions of the Engine except on rare occasions and only through our official repository on Microsoft Azure DevOps.

[1.3.1] - 2023-04-28

Fixed

- Fix RunUAT build issues caused by missing headers.
- Minor documentation fixes.

[1.3.0] - 2023-04-28

This release breaks ABI/API compatibility with the previous versions in addition to breaking coordinates systems conversions between Unreal Engine and the SenseGlove libraries.

Added

- A new generic SenseGlove Debug module.
- A debug virtual hand.

Fixed

- Fix the wrist tracker miscalculations for the Quest 2 controllers (other headsets might need fixing as well, in that case, future releases will address that).
- Minor code improvement and fixes.
- Minor documentation fixes.

Changed

- Breaking API/ABI changes in the Settings and the main SenseGlove module due to some settings refactoring.
- Breaking changes in the SenseGlove/Unreal coordinates systems conversions due to underlying changes in the SenseGlove Core Libraries.
- SenseGlove libraries have been updated to v2.6.0-aac3d56.

[1.2.1] - 2023-03-30

Fixed

- Fix RunUAT build issues with Android.

[1.2.0] - 2023-03-28

This release breaks ABI/API compatibility with the previous versions.

Added

- Android / Oculus on-device glove calibration.
- Introduced the animated Virtual Hand Model (as a set of virtual hand and wrist tracker components and an actor) with in-editor animation availability.
- Introduced SGPawn, SGPlayerController, SGGameModeBase, etc classes.
- Added an internal SenseGloveCoreTypes module in order to share common SenseGloveCore types between various modules.
- Segregated Android binaries for NDK r21e (UE 4.27 and 5.0) and r25b (UE 5.1, 5.2).
- Fully functional and stable Linux development support.
- Fully functional and stable Unreal Engine 5.2 preview support has been added.
- Added a Plugin's settings manager and two new modules SenseGloveSettings and SenseGloveSettingsKismet.

Changed

- SenseGlove libraries have been updated to the Linux-aware version: v2.5.0-8069342.
- API has changed to use degrees instead of radians.
- SGCoordinates utility class name has been changed to SGAngles and now the plugin API uses degrees in contrast of SenseGlove libraries by default.
- Migrate common nested array types into the SenseGloveTypes module from the SenseGloveCore module.

Removed

- Removed a few thousand lines of archaic pre-public-release dead code.
- Dropped Android NDK r21b binaries used by the older engine versions.
- Purged the dead code for dropped engine versions by v1.1.1 (4.22, 4.23, 4.24, 4.25, and 4.26) that carried over to the current version.
- Removed redundant SGConnectImpl/SGPlatform.
- Removed redundant SGTypes/SGConnectTypes.

Known Issues

- Wrist Tracker's offsets are a bit off (e.g. on Quest 2), scheduled to be fixed in the next patch release.

[1.1.1] - 2023-02-07

Added

- Initial support for the upcoming Unreal Engine 5.2.
- Add support for Android armeabi-v7a with neon, x86-64, and x86 builds in addition to arm64-v8a.

Fixed

- Fix various Android build issues.
- Some minor fixes and improvements.

Changed

- Bump SenseGlove libraries to v2.1.2-95ec6e7.

[1.1.0] - 2023-02-03

Added

- Whitelist Android as a target platform.
- Introduce Android support.
- Add third-party library SGConnect for Android v1.1.0.

Fixed

- Fix Android build issues caused by the log module.

Changed

- SGConnect and SGCore libraries have been updated to v2.1.1-0569c74.

Removed

- Removed the enum utils class due to ANY_PACKAGE deprecation warnings in Unreal Engine 5.1.
- Support for older versions of the Engine (namely, 4.22, 4.23, 4.24, 4.25, and 4.26) has been dropped.

[1.0.4] - 2022-12-02

This is a minor release focusing mostly on adherence to the Unreal Engine Marketplace Guidelines based on the feedback from Epic Games.

Added

- Added support for MSVC 2017

Changed

- Updated SenseGlove libraries (SGCore/SGConnect) to v2.0.4.

[1.0.3] - 2022-11-29

This is a minor release focusing on adherence to the Unreal Engine Marketplace Guidelines based on the feedback from Epic Games.

Changed

- Adjust Config/FilterPlugin.ini in order to conform to Epic's Market Place Guidelines.

[1.0.2] - 2022-11-27

This is a minor release focusing on adherence to the Unreal Engine Marketplace Guidelines based on the feedback from Epic Games.

Added

- Added the newly acquired Unreal Engine Market Place Offer ID to the .uplugin file.
- List the dotfiles inside the FilterPlugin.ini file as well.
- Add the copyright notice to the source files missing it.
- Add the SenseGlove SDK license to the third-party license file.

Fixed

- Fix the readme typos and errors.
- Minor fixes in the changelog for previous releases.

[1.0.1] - 2022-11-25

Changed

- Exposed SenseGloveTypes as a public dependency in SenseGloveConnect and SenseGloveCore modules, so that the C++ users of the API don't need to explicitly add it as a dependency.
- Cleaned up the redundant headers/modules dependencies from SGCore headers.

Fixed

- Fix RunUAT build issues prior to Epic Store submission.

[1.0.0] - 2022-11-24

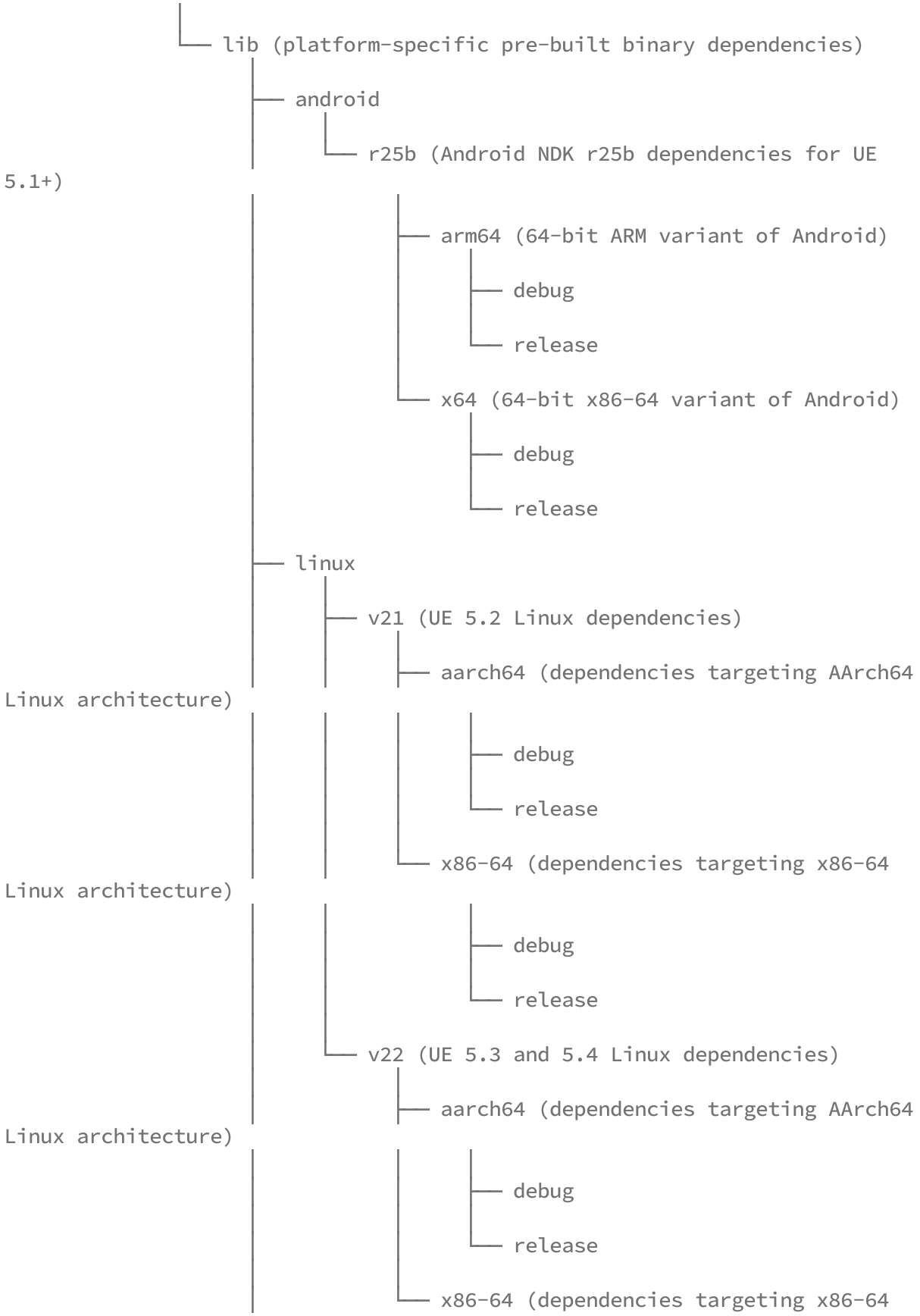
Added

- Initial public release of the SenseGlove haptic API for Unreal Engine with support for Microsoft Windows and GNU/Linux.

Directory Structure

```
/
   ├── Config
   ├── Documentation (this will be generated by running the <code>make</code>
command inside the Handbook directory)
   ├── Handbook (this is the mdBook source code, used to generate the
Documentation folder and not distributed to [Fab](https://www.fab.com/))
   ├── Resources
   └── Source (various plug-in modules)
         ├── SenseGlove (the UE-specific high-level API)
         ├── SenseGloveAndroid (the Android-specific module)
         ├── SenseGloveBackend (responsible for initialization and
deinitialization of the backend libraries)
         ├── SenseGloveBackendKismet (exposes Blueprint-specific functionality
from the SenseGloveBackend module)
         ├── SenseGloveBuildHacks (uses Exceptions and RTTI, internally used
for compiler-specific build hacks)
         ├── SenseGloveConnect (exposes part of the SGConnect low-level API to
C++)
         ├── SenseGloveConnectImpl (uses Exceptions and RTTI, intended for
internal use only)
         ├── SenseGloveConnectKismet (SGConnect functionality exposed to
Blueprint)
         ├── SenseGloveCore (exposes part of the SGCoreCpp low-level API to
C++)
         ├── SenseGloveCoreImpl (uses Exceptions and RTTI, intended for
internal use only)
         ├── SenseGloveCoreKismet (SGCoreCpp functionality exposed to
```

```
Blueprint)
        ├── SenseGloveDebug (a utility debug module)
        ├── SenseGloveDebugKismet (exposes Blueprint-specific functionality
from the SenseGloveDebug module)
        ├── SenseGloveEditor (the Editor module)
        ├── SenseGloveInterop (internally used for interoperability between
RTTI disabled/enabled modules)
        ├── SenseGloveKismet (exposes Blueprint-specific functionality from
the SenseGlove module)
        ├── SenseGloveLog (the internal log module)
        ├── SenseGloveSettings (the plugin's settings manager)
        ├── SenseGloveSettingsKismet (exposes Blueprint-specific
functionality from the SenseGloveSettings module)
        ├── SenseGloveTracking (provides XR_EXT_hand_tracking support, HMD
auto-detection, and SenseGlove device tracking)
        ├── SenseGloveTrackingKismet (exposes Blueprint-specific
functionality from the SenseGloveTracking module)
        ├── SenseGloveTypes (exposes various enums from the backend libraries
and also types from the SenseGlove module)
        ├── SenseGloveUtils (the internal utility module)
        └── ThirdParty (3rd-party dependencies)
                ├── android (.jar file Java libraries for Android)
                ├── include (header files)
                │       ├── boost
                │       ├── SenseGlove
                │       │       ├── Connect (SGConnect headers)
                │       │       └── Core (SGCoreCpp headers)
                │       └── serial
```


lib (platform-specific pre-built binary dependencies)
android
r25b (Android NDK r25b dependencies for UE 5.1+)
arm64 (64-bit ARM variant of Android)
debug
release
x64 (64-bit x86-64 variant of Android)
debug
release
linux
v21 (UE 5.2 Linux dependencies)
aarch64 (dependencies targeting AArch64 Linux architecture)
debug
release
x86-64 (dependencies targeting x86-64 Linux architecture)
debug
release
v22 (UE 5.3 and 5.4 Linux dependencies)
aarch64 (dependencies targeting AArch64 Linux architecture)
debug
release
x86-64 (dependencies targeting x86-64

```
Linux architecture)
                    │                   ├── debug
                    │                   └── release
                    └── win64
                        ├── msvc142 (Microsoft Visual Studio 2019
dependencies)
                        │   ├── debug
                        │   └── release
                        └── msvc143 (Microsoft Visual Studio 2022
dependencies)
                            ├── debug
                            └── release
```

Extra Resources

There are various resources available for older versions of the SenseGlove Unreal Engine Plugin prior to `v2.1.x` that might still be partially relevant. These include example projects, demo scenes, and tutorials. Plans are underway to provide new example projects, demo scenes, and tutorials for the latest release. In the meantime, the outdated resources can still be beneficial

Examples and Demo Projects

- A basic OpenXR-compatible Blueprint demo demonstrating basic functionality such as grab/release, touch with buzz and force-feedback, etc (compatible with versions v2.1.0+).
- A basic Blueprint demo demonstrating basic functionality such as grab/release, touch with buzz and force-feedback, etc (compatible with versions >= v1.4.x and <= v2.0.x).
- Example C++ API Project (only compatible with early v1.x.x releases)
- Example Blueprint API Project (only compatible with early v1.x.x releases)

OpenXR Integration Demos

- A VRExpansionPlugin Integration Demo for UE `5.4`

Tutorials

- Finding out your SenseGlove plugin version
- Plugin installation guide for Microsoft Windows
- C++ & Blueprint examples for Microsoft Windows
- Plugin and examples installation guide for GNU/Linux

- How to connect to Nova gloves on GNU/Linux using Blueman Bluetooth Manager
- How to connect to Nova gloves on GNU/Linux using command-line
- The basic C++ and Blueprint API usage
- How to setup the virtual hand model & the SenseGlove pawn
- How to deploy to Oculus Quest 2 and Android
- Setting up Grabbing and Haptic Feedback functionalities (SGBasicDemo)
- Setting up VIVE Pro & VIVE Trackers in Unreal Engine
- Setting up VIVE Focus 3 & VIVE Wrist Trackers in Unreal Engine
- SGBasicDemo: setup throwing objects and physics settings for the real and virtual hands
- SGBasicDemo v2: upgrading your projects to the SenseGlove Unreal Engine Plugin v2.0.0

SenseGlove Unreal Engine Plugin License

The SenseGlove Unreal Engine Plugin is licensed under the terms of the MIT License. Below is the MIT License:


```
MIT License

Copyright (c) 2020 - 2025 SenseGlove

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in
all
copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
SOFTWARE.
```


Please note that while the SenseGlove Unreal Engine Plugin is made available under the MIT License, it utilizes a few third-party libraries with permissive free licenses as well, in order to power various components. For a list of these libraries and their own respective open-source licenses take a look at the third-party licenses, please.

SenseGlove Unreal Engine Handbook License

The SenseGlove Unreal Engine Handbook is licensed under the terms of the CC BY (Creative Commons Attribution) License. Below is the CC BY License:

Attribution 4.0 International

Copyright (c) 2020 - 2025 SenseGlove

===

Creative Commons Corporation ("Creative Commons") is not a law firm and does not provide legal services or legal advice. Distribution of Creative Commons public licenses does not create a lawyer-client or other relationship. Creative Commons makes its licenses and related information available on an "as-is" basis. Creative Commons gives no warranties regarding its licenses, any material licensed under their terms and conditions, or any related information. Creative Commons disclaims all liability for damages resulting from their use to the fullest extent possible.

Using Creative Commons Public Licenses

Creative Commons public licenses provide a standard set of terms and conditions that creators and other rights holders may use to share original works of authorship and other material subject to copyright and certain other rights specified in the public license below. The following considerations are for informational purposes only, are not exhaustive, and do not form part of our licenses.

> Considerations for licensors: Our public licenses are intended for use by those authorized to give the public permission to use material in ways otherwise restricted by copyright and certain other rights. Our licenses are irrevocable. Licensors should read and understand the terms and conditions of the license they choose before applying it. Licensors should also secure all rights necessary before applying our licenses so that the public can reuse the material as expected. Licensors should clearly mark any material not subject to the license. This includes other CC-licensed material, or material used under an exception or limitation to copyright. More considerations for licensors: wiki.creativecommons.org/Considerations_for_licensors

> Considerations for the public: By using one of our public licenses, a licensor grants the public permission to use the licensed material under specified terms and conditions. If the licensor's permission is not necessary for any reason--for example, because of any applicable exception or limitation to copyright--then that use is not regulated by the license. Our licenses grant only permissions under copyright and certain other rights that a licensor has authority to grant. Use of

the licensed material may still be restricted for other reasons, including because others have copyright or other rights in the material. A licensor may make special requests, such as asking that all changes be marked or described. Although not required by our licenses, you are encouraged to respect those requests where reasonable. More considerations for the public:
wiki.creativecommons.org/Considerations_for_licensees

==========

Creative Commons Attribution 4.0 International Public License

By exercising the Licensed Rights (defined below), You accept and agree to be bound by the terms and conditions of this Creative Commons Attribution 4.0 International Public License ("Public License"). To the extent this Public License may be interpreted as a contract, You are granted the Licensed Rights in consideration of Your acceptance of these terms and conditions, and the Licensor grants You such rights in consideration of benefits the Licensor receives from making the Licensed Material available under these terms and conditions.

Section 1 -- Definitions.

a. Adapted Material means material subject to Copyright and Similar Rights that is derived from or based upon the Licensed Material and in which the Licensed Material is translated, altered, arranged, transformed, or otherwise modified in a manner requiring permission under the Copyright and Similar Rights held by the Licensor. For purposes of this Public License, where the Licensed Material is a musical work, performance, or sound recording, Adapted Material is always produced where the Licensed Material is synched in timed relation with a moving image.

b. Adapter's License means the license You apply to Your Copyright and Similar Rights in Your contributions to Adapted Material in accordance with the terms and conditions of this Public License.

c. Copyright and Similar Rights means copyright and/or similar rights closely related to copyright including, without limitation, performance, broadcast, sound recording, and Sui Generis Database Rights, without regard to how the rights are labeled or categorized. For purposes of this Public License, the rights specified in Section 2(b)(1)-(2) are not Copyright and Similar Rights.

d. Effective Technological Measures means those measures that, in the

absence of proper authority, may not be circumvented under laws fulfilling obligations under Article 11 of the WIPO Copyright Treaty adopted on December 20, 1996, and/or similar international agreements.

e. Exceptions and Limitations means fair use, fair dealing, and/or any other exception or limitation to Copyright and Similar Rights that applies to Your use of the Licensed Material.

f. Licensed Material means the artistic or literary work, database, or other material to which the Licensor applied this Public License.

g. Licensed Rights means the rights granted to You subject to the terms and conditions of this Public License, which are limited to all Copyright and Similar Rights that apply to Your use of the Licensed Material and that the Licensor has authority to license.

h. Licensor means the individual(s) or entity(ies) granting rights under this Public License.

i. Share means to provide material to the public by any means or process that requires permission under the Licensed Rights, such as reproduction, public display, public performance, distribution, dissemination, communication, or importation, and to make material available to the public including in ways that members of the public may access the material from a place and at a time individually chosen by them.

j. Sui Generis Database Rights means rights other than copyright resulting from Directive 96/9/EC of the European Parliament and of the Council of 11 March 1996 on the legal protection of databases, as amended and/or succeeded, as well as other essentially equivalent rights anywhere in the world.

k. You means the individual or entity exercising the Licensed Rights under this Public License. Your has a corresponding meaning.

Section 2 -- Scope.

a. License grant.

1. Subject to the terms and conditions of this Public License, the Licensor hereby grants You a worldwide, royalty-free, non-sublicensable, non-exclusive, irrevocable license to exercise the Licensed Rights in the Licensed Material to:

a. reproduce and Share the Licensed Material, in whole or in part; and

b. produce, reproduce, and Share Adapted Material.

2. Exceptions and Limitations. For the avoidance of doubt, where Exceptions and Limitations apply to Your use, this Public License does not apply, and You do not need to comply with its terms and conditions.

3. Term. The term of this Public License is specified in Section 6(a).

4. Media and formats; technical modifications allowed. The Licensor authorizes You to exercise the Licensed Rights in all media and formats whether now known or hereafter created, and to make technical modifications necessary to do so. The Licensor waives and/or agrees not to assert any right or authority to forbid You from making technical modifications necessary to exercise the Licensed Rights, including technical modifications necessary to circumvent Effective Technological Measures. For purposes of this Public License, simply making modifications authorized by this Section 2(a)(4) never produces Adapted Material.

5. Downstream recipients.

a. Offer from the Licensor -- Licensed Material. Every recipient of the Licensed Material automatically receives an offer from the Licensor to exercise the Licensed Rights under the terms and conditions of this Public License.

b. No downstream restrictions. You may not offer or impose any additional or different terms or conditions on, or apply any Effective Technological Measures to, the Licensed Material if doing so restricts exercise of the Licensed Rights by any recipient of the Licensed Material.

6. No endorsement. Nothing in this Public License constitutes or may be construed as permission to assert or imply that You are, or that Your use of the Licensed Material is, connected with, or sponsored, endorsed, or granted official status by, the Licensor or others designated to receive attribution as provided in Section 3(a)(1)(A)(i).

b. Other rights.

1. Moral rights, such as the right of integrity, are not licensed under this Public License, nor are publicity, privacy, and/or other similar personality rights; however, to the extent possible, the Licensor waives and/or agrees not to assert any such rights held by the Licensor to the limited extent necessary to allow You to exercise the Licensed Rights, but not otherwise.

2. Patent and trademark rights are not licensed under this Public License.

3. To the extent possible, the Licensor waives any right to collect royalties from You for the exercise of the Licensed Rights, whether directly or through a collecting society under any voluntary or waivable statutory or compulsory licensing scheme. In all other cases the Licensor expressly reserves any right to collect such royalties.

Section 3 -- License Conditions.

Your exercise of the Licensed Rights is expressly made subject to the following conditions.

a. Attribution.

1. If You Share the Licensed Material (including in modified form), You must:

a. retain the following if it is supplied by the Licensor with the Licensed Material:

i. identification of the creator(s) of the Licensed Material and any others designated to receive attribution, in any reasonable manner requested by the Licensor (including by pseudonym if designated);

ii. a copyright notice;

iii. a notice that refers to this Public License;

iv. a notice that refers to the disclaimer of warranties;

v. a URI or hyperlink to the Licensed Material to the extent reasonably practicable;

b. indicate if You modified the Licensed Material and retain an indication of any previous modifications; and

c. indicate the Licensed Material is licensed under this Public License, and include the text of, or the URI or hyperlink to, this Public License.

2. You may satisfy the conditions in Section 3(a)(1) in any reasonable manner based on the medium, means, and context in which You Share the Licensed Material. For example, it may be reasonable to satisfy the conditions by providing a URI or hyperlink to a resource that includes the required information.

3. If requested by the Licensor, You must remove any of the information required by Section 3(a)(1)(A) to the extent reasonably practicable.

4. If You Share Adapted Material You produce, the Adapter's License You apply must not prevent recipients of the Adapted Material from complying with this Public License.

Section 4 -- Sui Generis Database Rights.

Where the Licensed Rights include Sui Generis Database Rights that apply to Your use of the Licensed Material:

a. for the avoidance of doubt, Section 2(a)(1) grants You the right to extract, reuse, reproduce, and Share all or a substantial portion of the contents of the database;

b. if You include all or a substantial portion of the database contents in a database in which You have Sui Generis Database Rights, then the database in which You have Sui Generis Database Rights (but not its individual contents) is Adapted Material; and

c. You must comply with the conditions in Section 3(a) if You Share all or a substantial portion of the contents of the database.

For the avoidance of doubt, this Section 4 supplements and does not replace Your obligations under this Public License where the Licensed Rights include other Copyright and Similar Rights.

Section 5 -- Disclaimer of Warranties and Limitation of Liability.

a. UNLESS OTHERWISE SEPARATELY UNDERTAKEN BY THE LICENSOR, TO THE EXTENT POSSIBLE, THE LICENSOR OFFERS THE LICENSED MATERIAL AS-IS AND AS-AVAILABLE, AND MAKES NO REPRESENTATIONS OR WARRANTIES OF ANY KIND CONCERNING THE LICENSED MATERIAL, WHETHER EXPRESS, IMPLIED, STATUTORY, OR OTHER. THIS INCLUDES, WITHOUT LIMITATION, WARRANTIES OF TITLE, MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, NON-INFRINGEMENT, ABSENCE OF LATENT OR OTHER DEFECTS, ACCURACY, OR THE PRESENCE OR ABSENCE OF ERRORS, WHETHER OR NOT KNOWN OR DISCOVERABLE. WHERE DISCLAIMERS OF WARRANTIES ARE NOT ALLOWED IN FULL OR IN PART, THIS DISCLAIMER MAY NOT APPLY TO YOU.

b. TO THE EXTENT POSSIBLE, IN NO EVENT WILL THE LICENSOR BE LIABLE TO YOU ON ANY LEGAL THEORY (INCLUDING, WITHOUT LIMITATION, NEGLIGENCE) OR OTHERWISE FOR ANY DIRECT, SPECIAL, INDIRECT, INCIDENTAL, CONSEQUENTIAL, PUNITIVE, EXEMPLARY, OR OTHER LOSSES, COSTS, EXPENSES, OR DAMAGES ARISING OUT OF THIS PUBLIC LICENSE OR USE OF THE LICENSED MATERIAL, EVEN IF THE LICENSOR HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH LOSSES, COSTS, EXPENSES, OR DAMAGES. WHERE A LIMITATION OF LIABILITY IS NOT ALLOWED IN FULL OR IN PART, THIS LIMITATION MAY NOT APPLY TO YOU.

c. The disclaimer of warranties and limitation of liability provided above shall be interpreted in a manner that, to the extent possible, most closely approximates an absolute disclaimer and waiver of all liability.

Section 6 -- Term and Termination.

a. This Public License applies for the term of the Copyright and Similar Rights licensed here. However, if You fail to comply with this Public License, then Your rights under this Public License terminate automatically.

b. Where Your right to use the Licensed Material has terminated under Section 6(a), it reinstates:

    1. automatically as of the date the violation is cured, provided it is cured within 30 days of Your discovery of the violation; or

    2. upon express reinstatement by the Licensor.

    For the avoidance of doubt, this Section 6(b) does not affect any right the Licensor may have to seek remedies for Your violations of this Public License.

c. For the avoidance of doubt, the Licensor may also offer the

Licensed Material under separate terms or conditions or stop distributing the Licensed Material at any time; however, doing so will not terminate this Public License.

d. Sections 1, 5, 6, 7, and 8 survive termination of this Public License.

Section 7 -- Other Terms and Conditions.

a. The Licensor shall not be bound by any additional or different terms or conditions communicated by You unless expressly agreed.

b. Any arrangements, understandings, or agreements regarding the Licensed Material not stated herein are separate from and independent of the terms and conditions of this Public License.

Section 8 -- Interpretation.

a. For the avoidance of doubt, this Public License does not, and shall not be interpreted to, reduce, limit, restrict, or impose conditions on any use of the Licensed Material that could lawfully be made without permission under this Public License.

b. To the extent possible, if any provision of this Public License is deemed unenforceable, it shall be automatically reformed to the minimum extent necessary to make it enforceable. If the provision cannot be reformed, it shall be severed from this Public License without affecting the enforceability of the remaining terms and conditions.

c. No term or condition of this Public License will be waived and no failure to comply consented to unless expressly agreed to by the Licensor.

d. Nothing in this Public License constitutes or may be interpreted as a limitation upon, or waiver of, any privileges and immunities that apply to the Licensor or You, including from the legal processes of any jurisdiction or authority.

==========

Creative Commons is not a party to its public
licenses. Notwithstanding, Creative Commons may elect to apply one of its public licenses to material it publishes and in those instances will be considered the “Licensor.” The text of the Creative Commons

public licenses is dedicated to the public domain under the CC0 Public Domain Dedication. Except for the limited purpose of indicating that material is shared under a Creative Commons public license or as otherwise permitted by the Creative Commons policies published at creativecommons.org/policies, Creative Commons does not authorize the use of the trademark "Creative Commons" or any other trademark or logo of Creative Commons without its prior written consent including, without limitation, in connection with any unauthorized modifications to any of its public licenses or any other arrangements, understandings, or agreements concerning use of licensed material. For the avoidance of doubt, this paragraph does not form part of the public licenses.

Creative Commons may be contacted at creativecommons.org.

Third Party Licenses

Please note that while the SenseGlove Unreal Engine Plugin is made available under the MIT License, it utilizes a few third-party libraries with permissive free licenses as well, in order to power various components.

The following third-party software are used and shipped with the SenseGlove Unreal Engine Plugin:

- The SenseGlove SDK (a.k.a. SenseGlove Backend Libraries, or SenseGlove Core Libraries)
- The Boost C++ Libraries
- The Serial Communication Library

For more information consult their own respective open-source licenses, please.

SenseGlove SDK License

SENSEGLOVE SDK LICENSE

Copyright (c) 2020 - 2025 SenseGlove

Purchase of the Product does not entitle you to ownership or a license to any software generated by SenseGlove for use with the Product (the “Software”). To the extent that SenseGlove, in its sole discretion, grants you access to any such Software, the Software is licensed by us or by the relevant licensor/owner subject to the relevant end-user license agreement or other license terms included with the Product and/or on the SenseGlove Websites including the Github page of SenseGlove (the “License Terms“).

Specifically, SenseGlove shall have sole discretion to determine and change the availability, nature, features, content, versioning of any Software that it makes available to you, for download through the the Github page of SenseGlove or otherwise (including the SenseGlove software developer kit (“SDK”)). Purchase of a Product does not entitle you to access to any specific features, content or version of the SDK, including and especially versions of the SDK that have not yet been made available to the public. SenseGlove will have no obligation to provide any updates or upgrades to any Software it makes available to you, but in the event that it does, such updates, upgrades and any documentation will be subject to the License Terms available at https://www.senseglove.com/solutions/.

Except to the extent expressly provided by us in writing or under the License Terms, the Software is provided “AS IS” without any warranties, terms or conditions as to quality, fitness for purpose, non-infringement, performance or correspondence with description and we do not offer any warranties or guarantees in relation to the Software installation, configuration or error/defect correction.

Boost C++ Libraries License

Boost Software License - Version 1.0 - August 17th, 2003

Permission is hereby granted, free of charge, to any person or organization obtaining a copy of the software and accompanying documentation covered by this license (the "Software") to use, reproduce, display, distribute, execute, and transmit the Software, and to prepare derivative works of the Software, and to permit third-parties to whom the Software is furnished to do so, all subject to the following:

The copyright notices in the Software and this entire statement, including the above license grant, this restriction and the following disclaimer, must be included in all copies of the Software, in whole or in part, and all derivative works of the Software, unless such copies or derivative works are solely in the form of machine-executable object code generated by a source language processor.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT. IN NO EVENT SHALL THE COPYRIGHT HOLDERS OR ANYONE DISTRIBUTING THE SOFTWARE BE LIABLE FOR ANY DAMAGES OR OTHER LIABILITY, WHETHER IN CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Serial Communication Library License

Copyright (c) 2012 William Woodall, John Harrison

Permission is hereby granted, free of charge, to any person obtaining a copy of
this software and associated documentation files (the "Software"), to deal in
the Software without restriction, including without limitation the rights to
use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of
the Software, and to permit persons to whom the Software is furnished to do so,
subject to the following conditions:

The above copyright notice and this permission notice shall be included in all
copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS
FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR
COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER
IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN
CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Build Information

	The SenseGlove Unreal Engine Handbook
Handbook Revision	2.4
Handbook Revision URL	https://unreal.docs.senseglove.com/2.4
Handbook PDF URL	https://unreal.docs.senseglove.com/2.4/the-senseglove-unreal-engine-handbook-2.4.pdf
Handbook ePub URL	https://unreal.docs.senseglove.com/2.4/the-senseglove-unreal-engine-handbook-2.4.epub
Git Branch	HEAD
Git Tag	v2.4.2
Git Commit	094854c2
Git Commits Since Tag	0
Git Tree State	clean
Git Is Shallow Clone	no
Git Latest Remote Tag	v2.4.2
Git Version	v2.4.2
Git Version Major	2
Git Version Minor	4
Git Version Patch	2
Plugin Version	v2.4.2
Plugin Version Major	2

The SenseGlove Unreal Engine Handbook	
Plugin Version Minor	4
Plugin Version Patch	2
Build Host	mamadou-legion
Build Time	Mon Feb 17, 2025 13:31 CET +0100