Unity SDK

Introduction

Welcome to the WEART Unity SDK documentation.

The WEART SDK allows the creation of haptic experiences in Unity by enabling interfacing with the TouchDIVER devices for both PC and Android Standalone applications and TouchDIVER Pro for PC applications. The new SDK handles both platforms and allows development and testing from the Unity Editor, as well as the ability to build for Windows and standalone headsets:

- Start and Stop device execution
- Calibrate the finger tracking device
- Receive tracking data from the devices
- Retrieve raw data from the device
- Send haptic effects to the devices (actuations)
- Read status information from the device

The minimum setup to use the weart SDK consists of:

- A TouchDIVER device
- An Unity project using the SDK package
- WeArt App running (only on the PC version)

TouchDIVER Pro

PC windows compatibility

- 2021
- 2022

> **Note**
> TouchDIVER Pro does not support Android yet.

TouchDIVER

Android standalone compatibility

- 2021
- 2022

PC windows compatibility

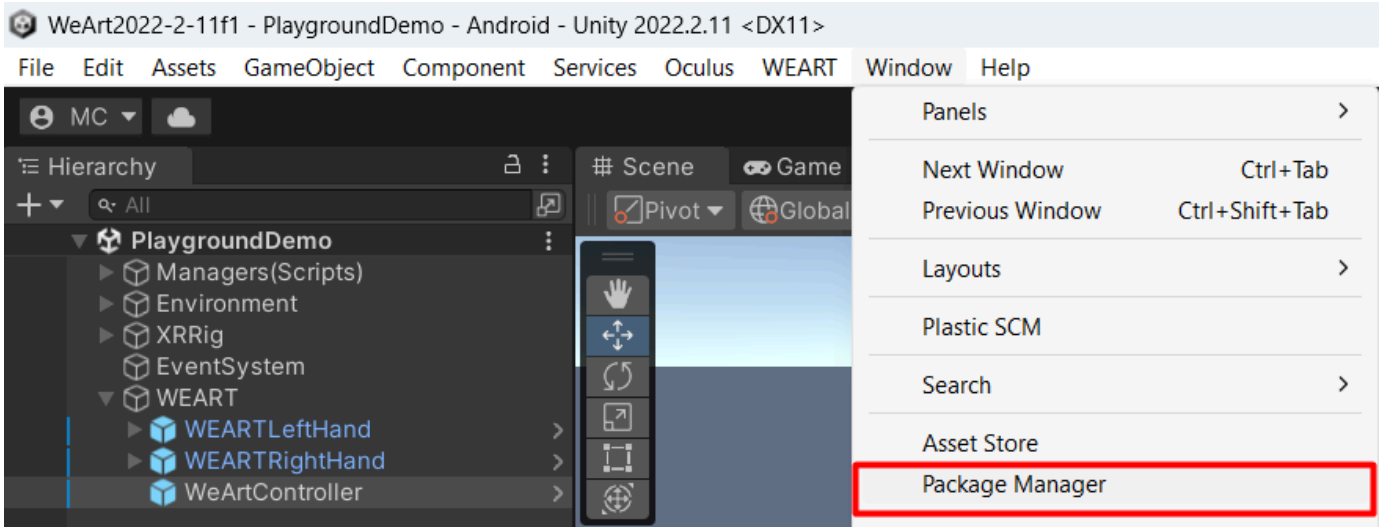
- 2021
- 2022

> **Note**
> For TouchDIVER it is possible to use the same WEART SDK on both platforms (Windows and Android).

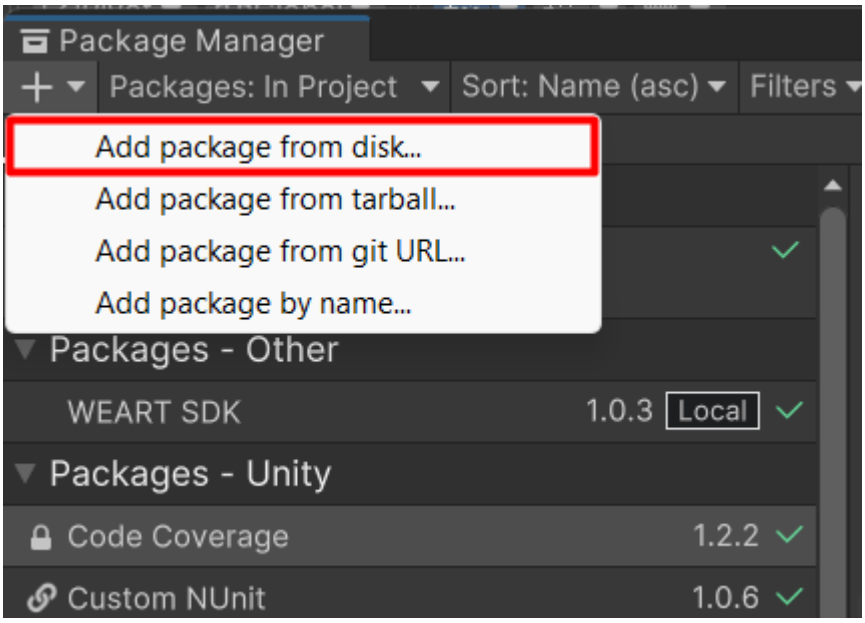
Importing WEART SDK

Create a new project or open an existing one.

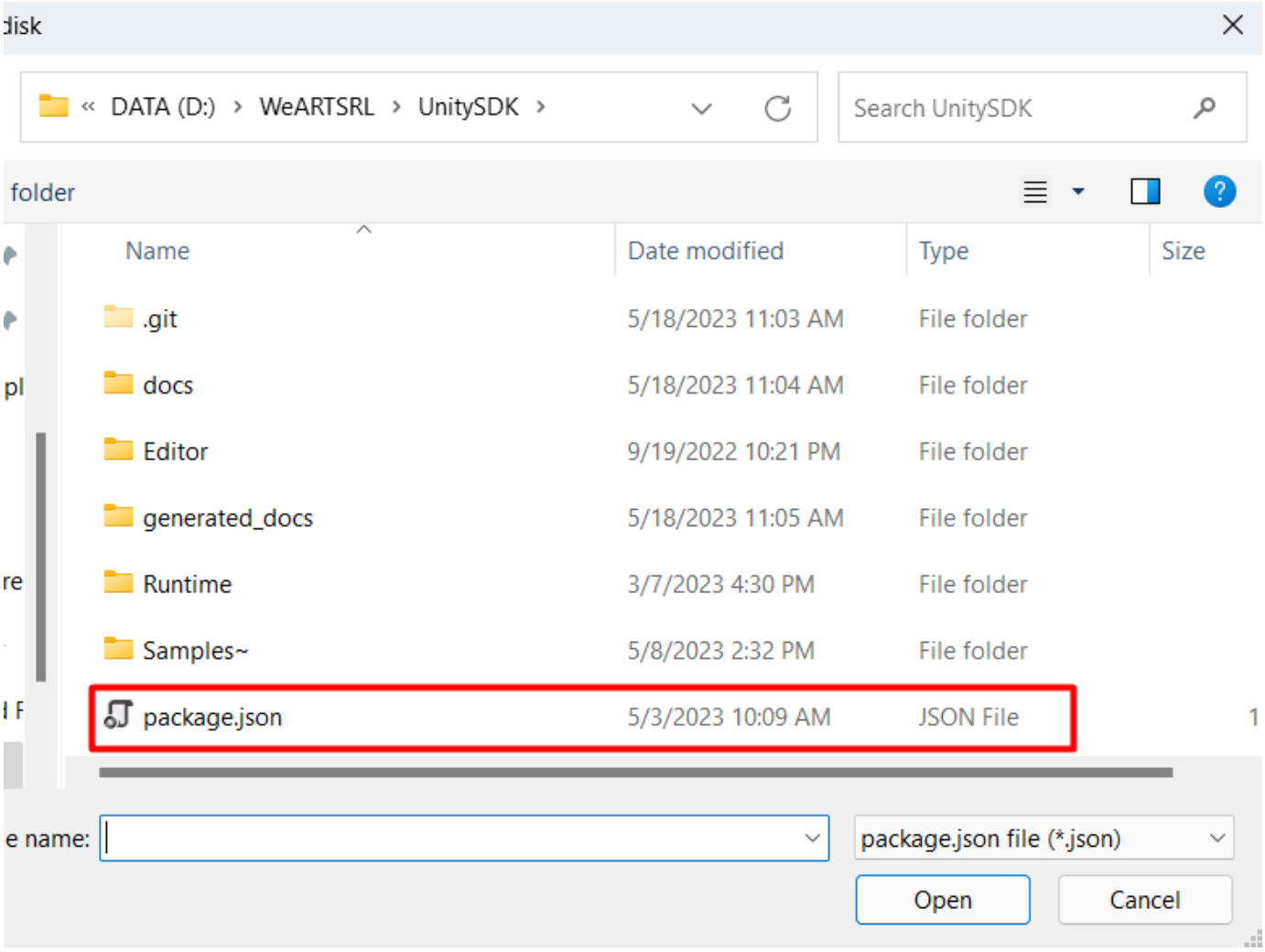
Go to "Window" and then to "Package Manager".



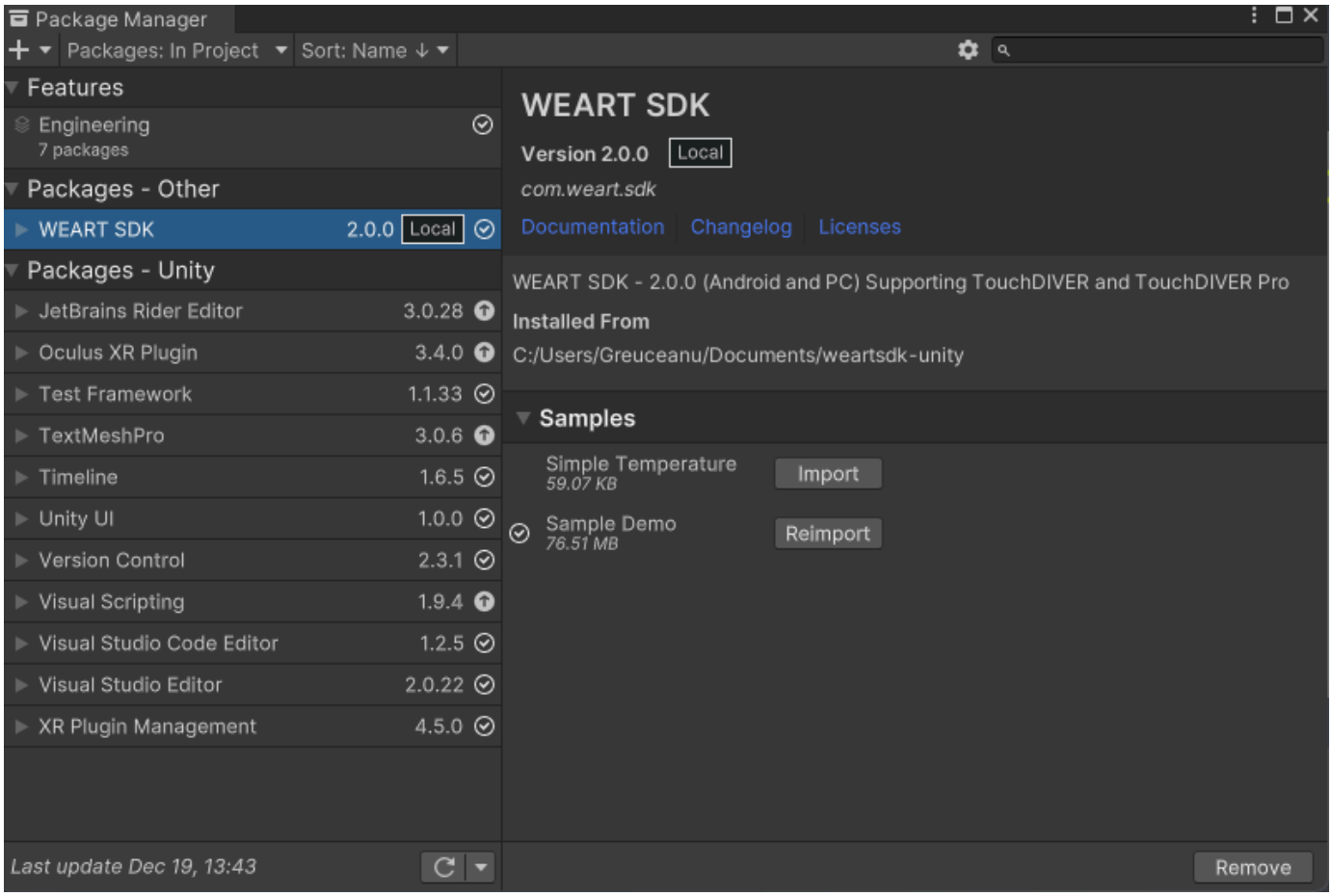
Then press on the "+" and select "Add package from disk".



Find the location of the SDK, select ***package.json*** and press "Open".



The package manager will now contain the WeArt SDK.



PC Windows platform

Unity Editor compatibility:

- 2021
- 2022

PC OS supported:

- Windows 10/11

Architecture

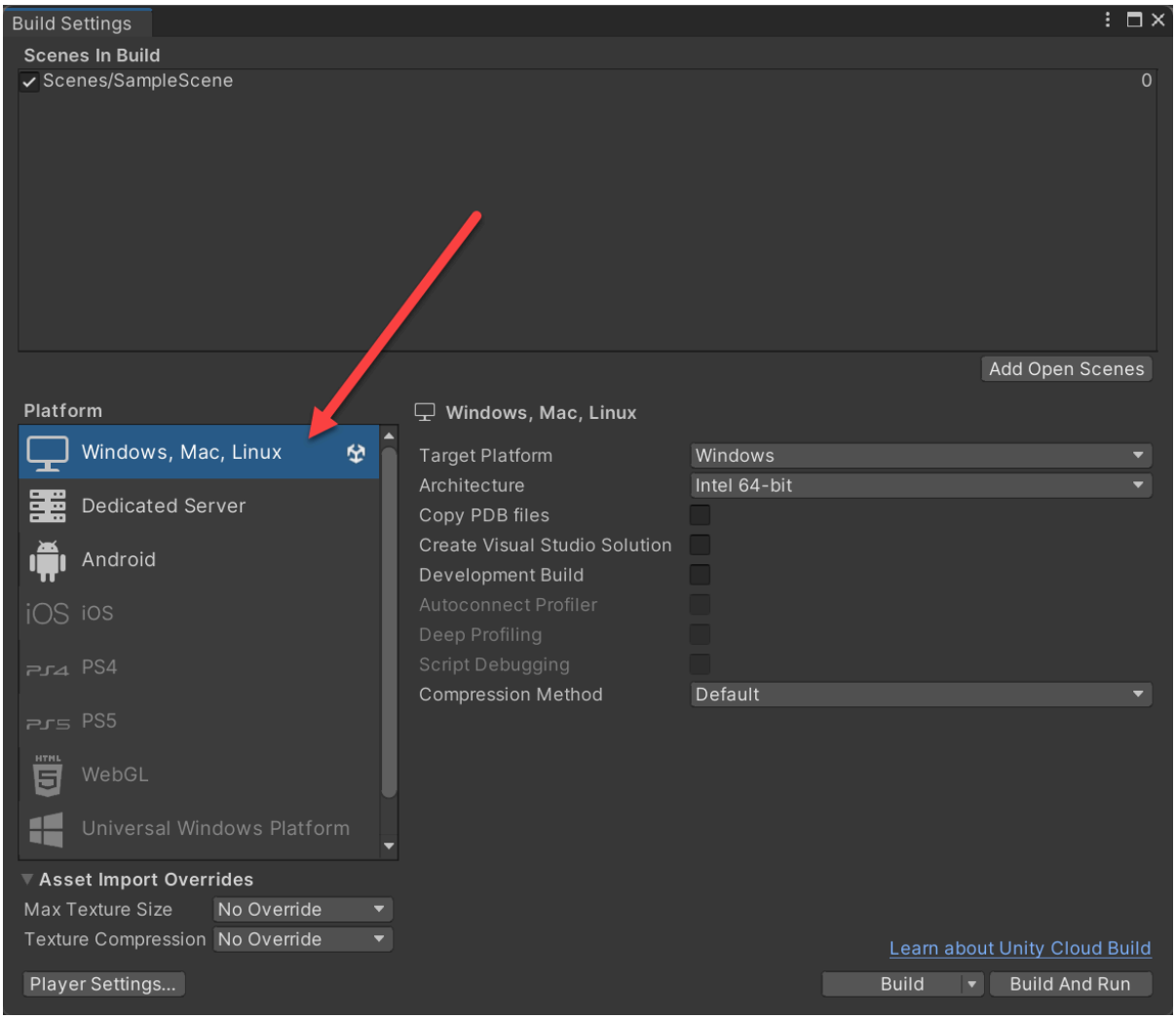
Unity SDK Architecture

Setup PC application

Requirements

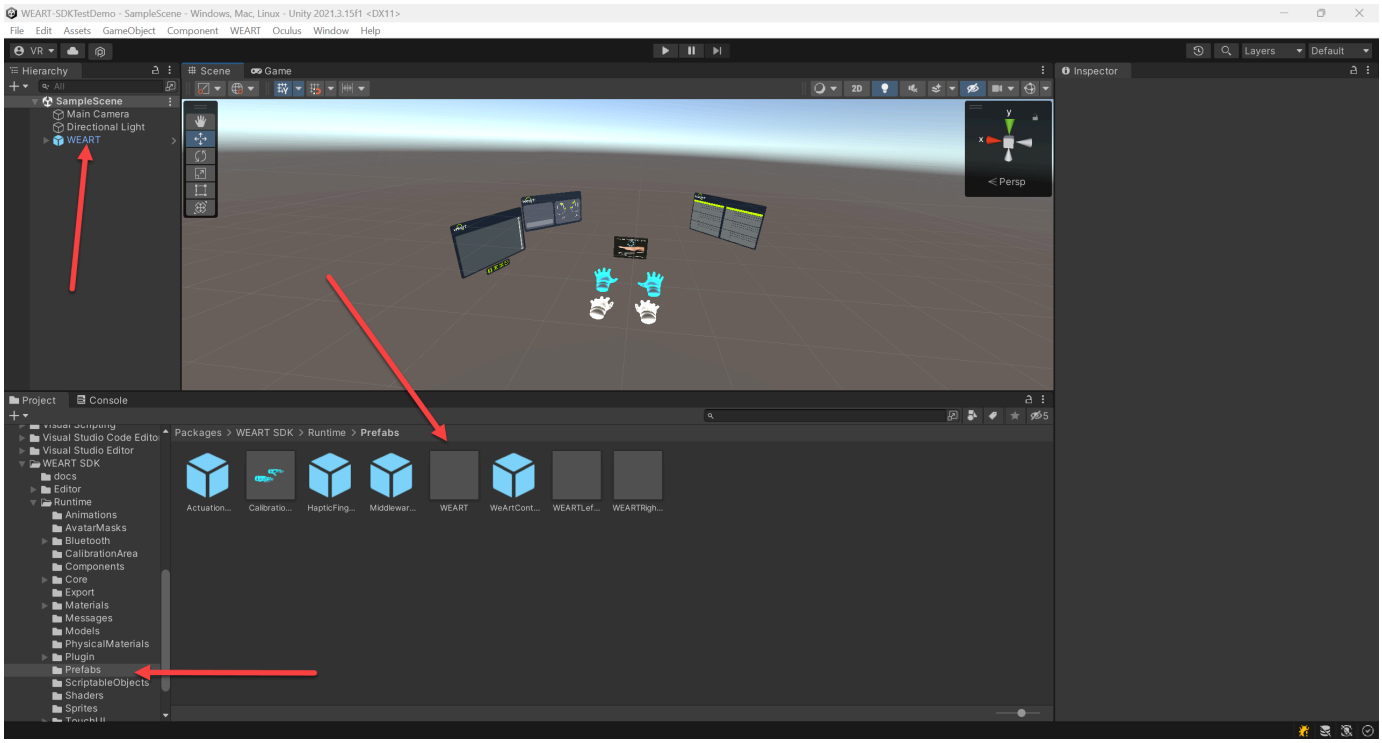
- Run the WeArt App (desktop application)
- Connect at least 1 TouchDIVER
- Run Untiy experience

Ensure the Unity platform is configurated on **Windows, Mac, Linux**



Unity platform

Drag the ***WEART*** prefab on the scene from: *Packages -> WEART SDK -> Runtime -> Prefabs* and enter in play mode.



WEART prefab

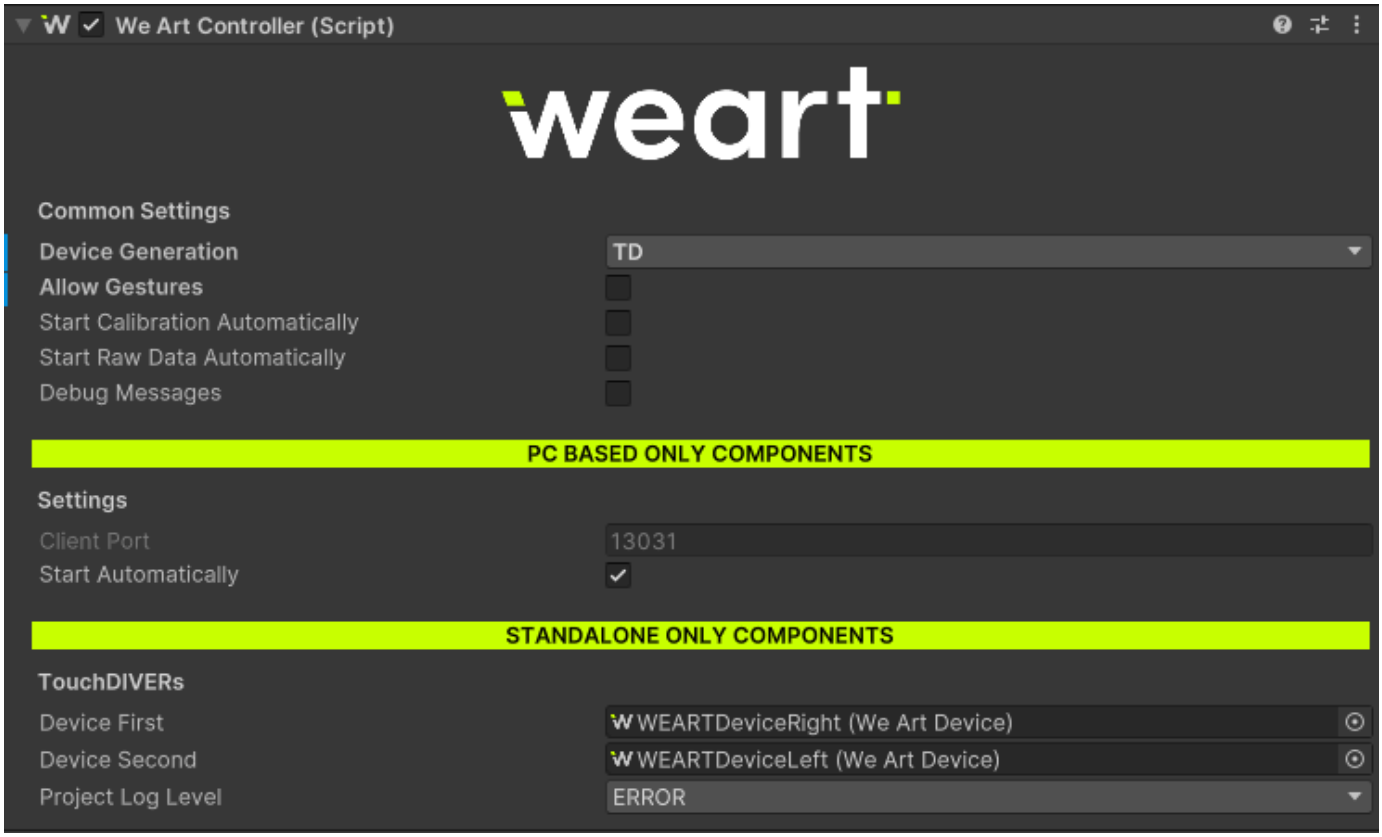
Features Guide

Components

WeArtController

Responsible for communication with the TouchDIVER device through dialogue with WeArt App.
Path: Packages→ WEART SDK → Runtime → Prefabs → WeArtController.

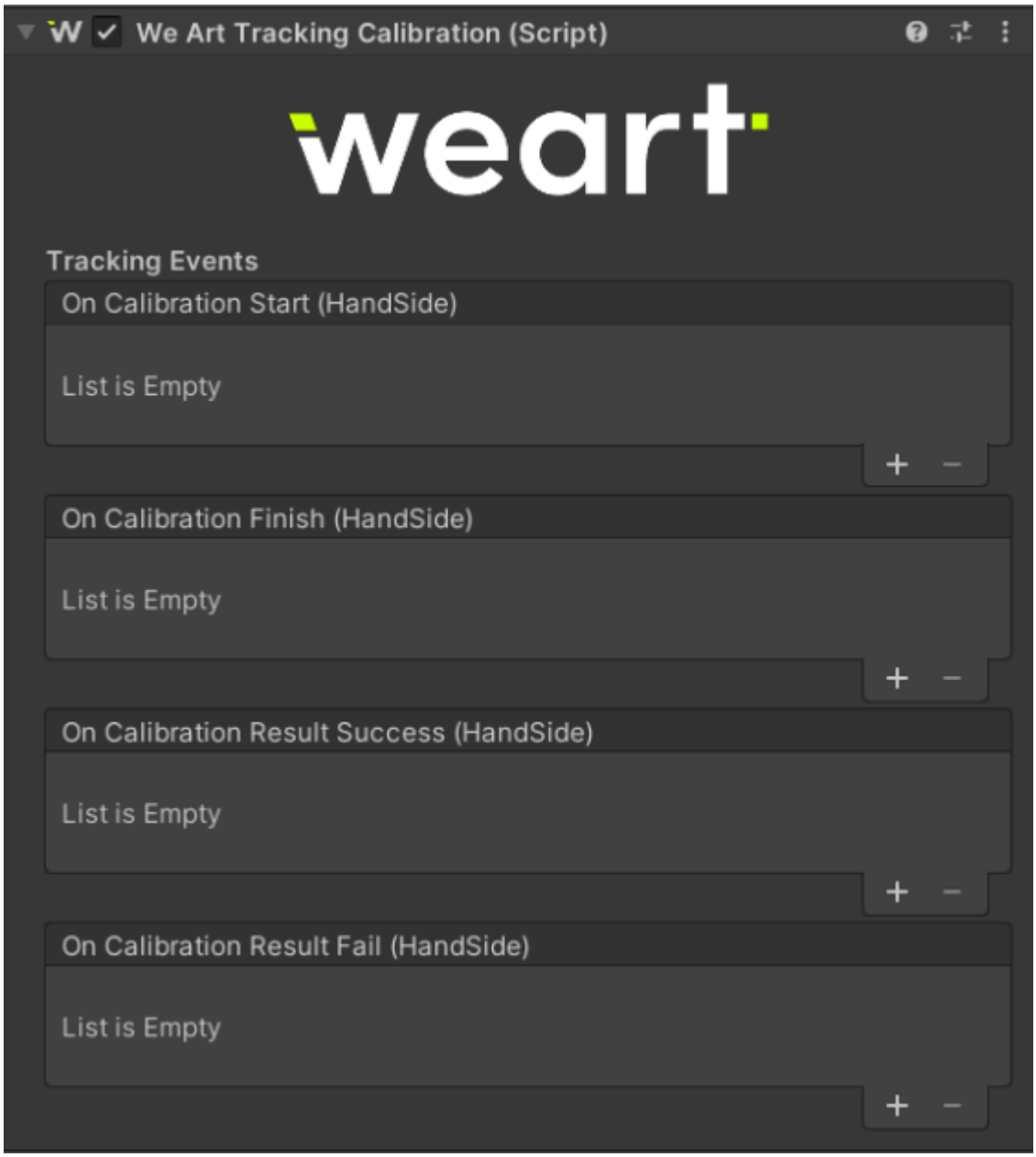
Inspector Properties:



- Device Generation
  - TD - TouchDIVER with three fingers / TD_Pro - TouchDIVER Pro with five fingers
- Allow Gestures
  - Allow the recognition of gestures for features such as teleportation
- Start Calibration Automatically:
  - When this flag is active, the WeArt App calibration begins when the scene starts playing
- Start Raw Data Automatically:
  - When this flag is active, the WeArt App starts sending raw sensors data when the scene starts playing

- Debug Messages:
  - Enable or Disable WEART SDK Logs in console
- Client Port:
  - Socket TCP Port for dialogue with WeArt App (don’t change)
- Start Automatically
  - Starts the SDK when opening the scene in play mode
- TouchDIVERs
  - DeviceFirst - WEARTDeviceRight reference object
  - DeviceSecond - WEARTDeviceLeft reference object
- Project Log Level:
  - Set the level of logs that will be written in you session log file.
  - **ERROR** - print only error messages
  - **DEBUG** - print errors and debug messages
  - **VERBOSE** - print all information available
  - The session logs can be found at your Android device: ***\Android\data\<YourApplicationName>\files\LogFiles***

WeArtTrackingCalibration contains Unity events that correspond to the WeArt App calibration events

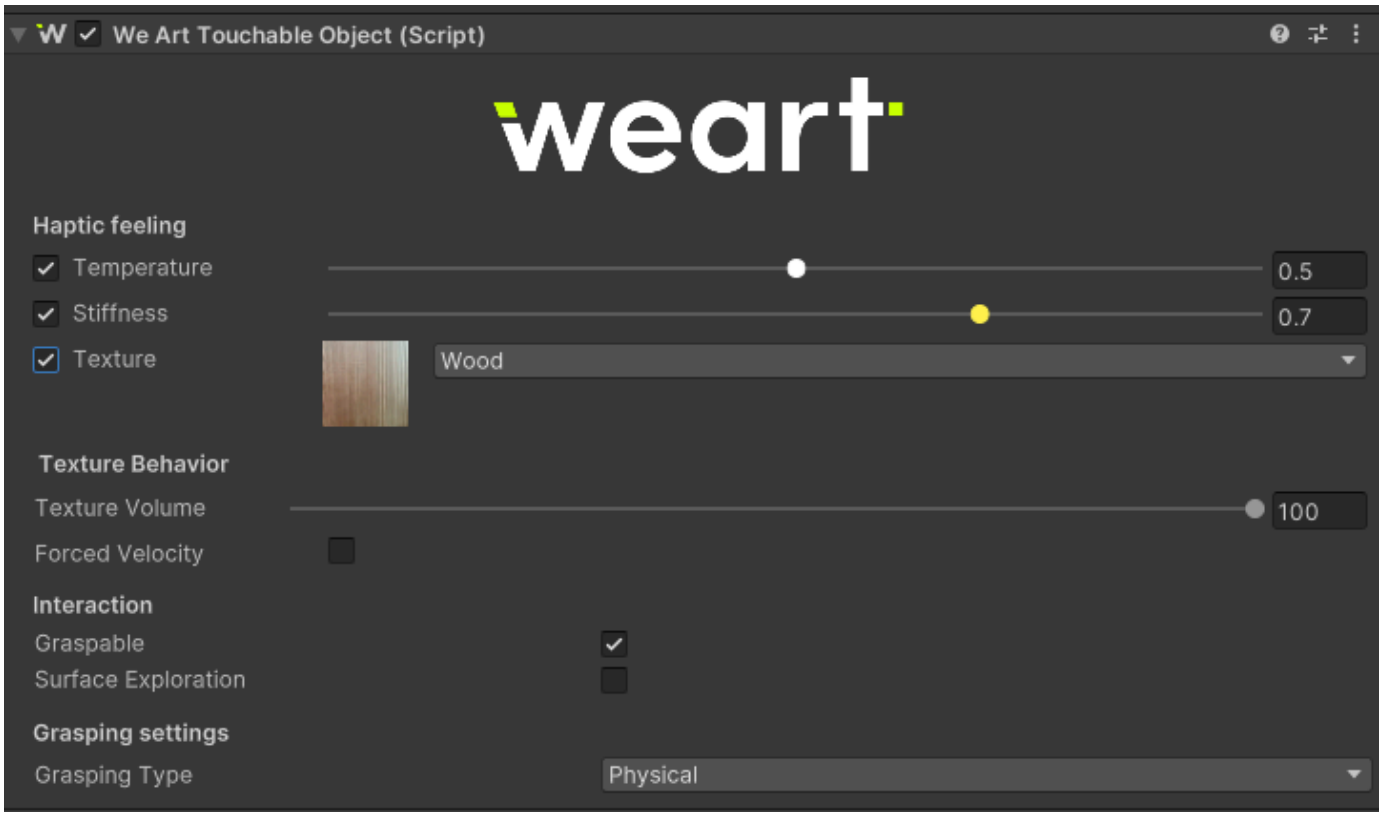


WeArtTouchableObject

Component responsible for the description of the haptic effect to be applied in the event of a collision with the HapticObject actor Properties:

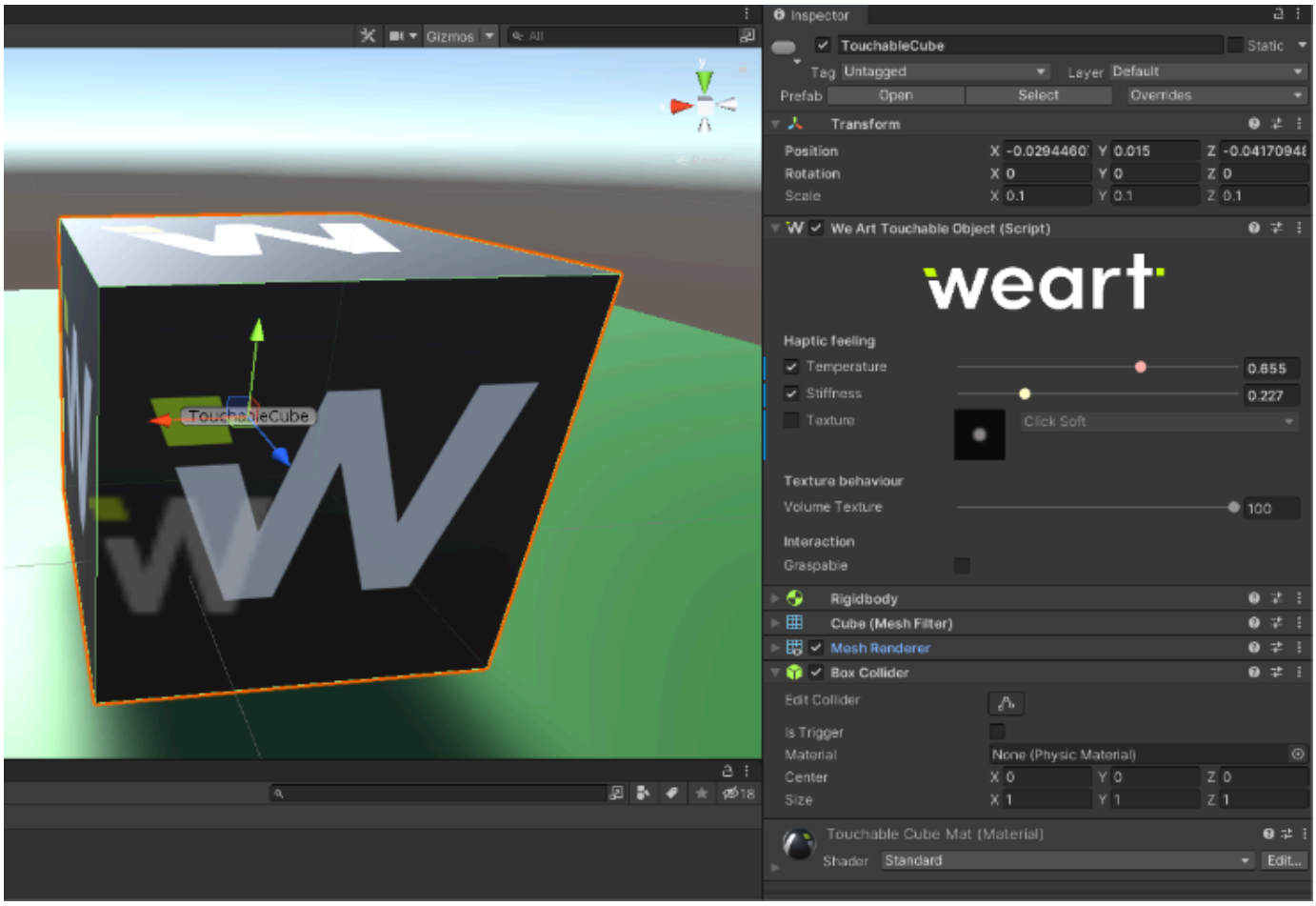
- Temperature – Temperature value implemented on the target thimble or thimbles (from 0.0 to 1.0) [0.5 is environment temp – 0.0 really cold – 1.0 really hot]
- Force – Force value applied on the target thimble (s) (from 0 to 1) [0.0 no force – 1.0 max force]
- Texture – Type of texture rendered on the thimble or target thimbles (from 0 to N)
- Volume Texture: Configure the intensity of texture rendering (from 0 to 100)
- Forced Velocity - Allows the touchable object to have continuous texture at maximum velocity
- Graspable - Enable the ability to grasp the object with virtual hands
- Surface Exploration - When enabled, it makes the exploration of a surface with the palm and fingers easier when the palm is open

- Grasping Type - Define if the touchable object will use the default "Physical" grasping system or the "Snap" grasping system

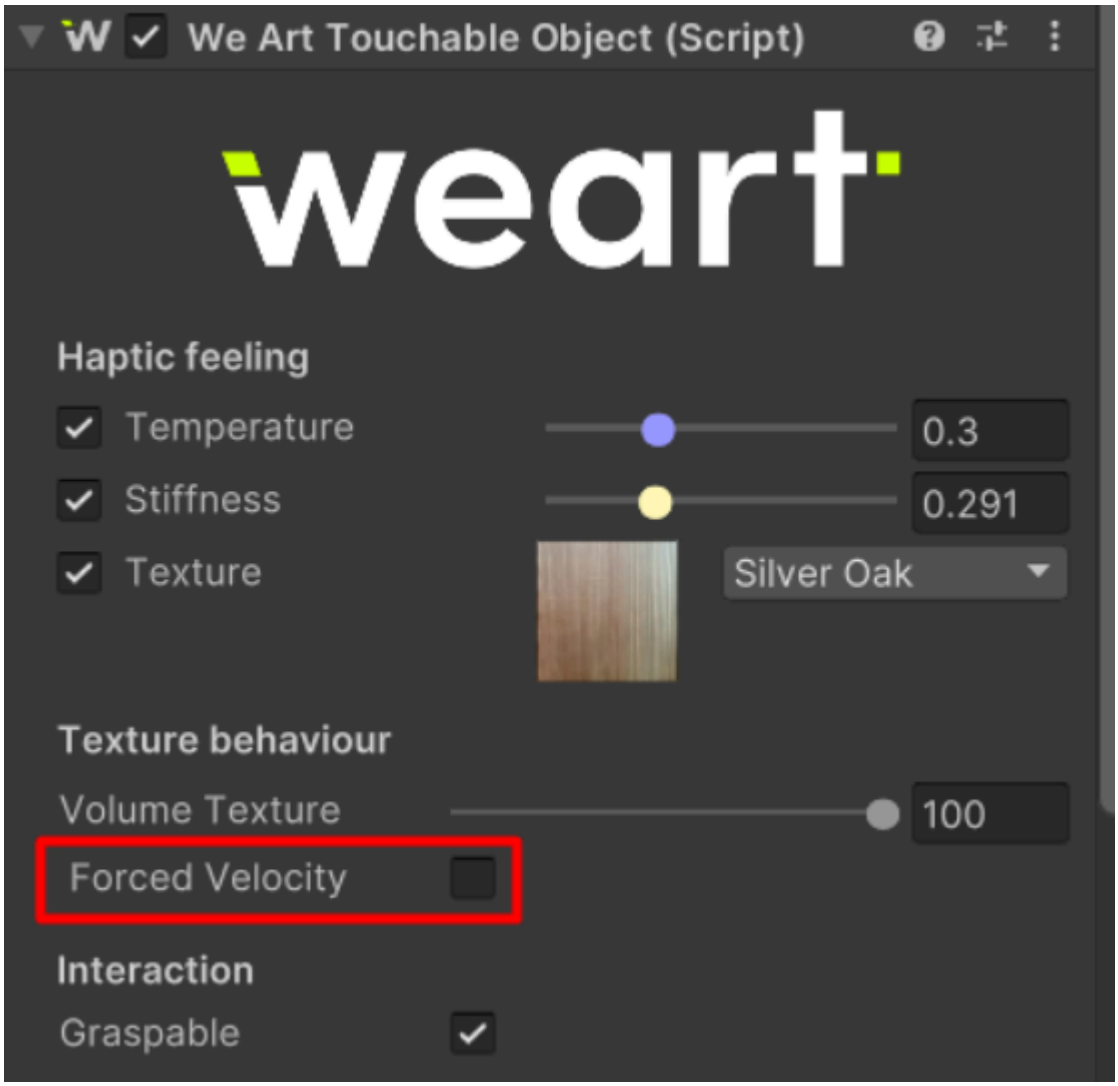


GameObject Requirements:

- Rigidbody
- Collider



The component WeArtTouchableObject has a flag called Forced velocity, if it is enabled, when the hand enters the touchable object, the texture feeling will run at maximum speed. It will not take in consideration the movement of the hand.



WeArtChildCollider

When WeArtTouchableObject gets instantiated, it looks recursively into its children and applies the component WeArtChildCollider to all game objects found. This allows the WeArtTouchableObject to register its children collisions as its own. When touching a child object, we will feel the effect of its referenced WeArtTouchableObject. This works also for grasping and anchored objects. The game objects do not need to be active in the inspector for WeArtChildCollider to be applied. The component is only present during play mode.

Note
If we want to add a touchable object as a child to another touchable object, that parent's transform must have the scale (1f, 1f, 1f) or all three values to be the same otherwise unity will stretch them. It is recommended to use the scale (1, 1, 1) for this scenarios

We can get ParentTouchableObject by code, it is a public method that return the WeArtTouchableObject that it belongs to.

WeArtTouchableObject supports adding children with colliders during play mode, the linking is done by the touchable object that checks if a child is added or removed, and it's children that have WeArtChildCollider, will also check if their children gets added or removed.

WeArtChildCollider must not be added manually during playtime as it will not link correctly to the WeArtTouchableObject.

> **Note**
> Removing or adding collider components to the touchable object or its children during runtime is a bad practice. Enabling and disabling them during runtime is a good practice instead.

Unrecommended use of addition and removal of collider components

If you really need to add or remove collider components here is a guide of how to handle it. It is advised to not do this at all, as we can simply add a game object with colliders instead. A prefab, for example, that has the colliders disabled and then we can just enable the colliders.

WeArtTouchableObject has an important field called "_firstCollider" this is the main reference used for collision handling. It is the first colliders that is found recursively through its own colliders and its children's colliders. The replacement of this "_firstCollider" is done automatically in the case of adding or removing a child from the touchable's objects hierarchy, meaning that even a child of the child of the touchable object will recognize this addition or removal.

> **Note**
> If the "_firstCollider" reference is deleted manually, meaning that the collider is deleted in the inspector or the code, the system will not know that it needs to find a new collider. It is fine to enable or disable these colliders, but deleing and adding colliders manually will not work automatically.

If we want to remove a collider component manually, after removing it we need to call the next method

```
// This will remove the null references and search recursively for a a new "_firstColli
touchable.CheckChildColliderList();
```

If we want to add a collider component manually, after creating it, we need to call this method

```
touchable.AddColliderToTouchableColliders(collider);
```

WeArtAnchoredObject

When the component is applied on a touchable object, it becomes and anchored object. This component needs to be applied to interactions such as levers, wheels, doors and drawers that use Hinge Joints or Configurable Joints.

Properties:

- Grasping Velocity - Velocity used when grasping this anchored Object
- Is Using Screwing Origin - Changes the origin used for velocity for small objects that have a screwing motion

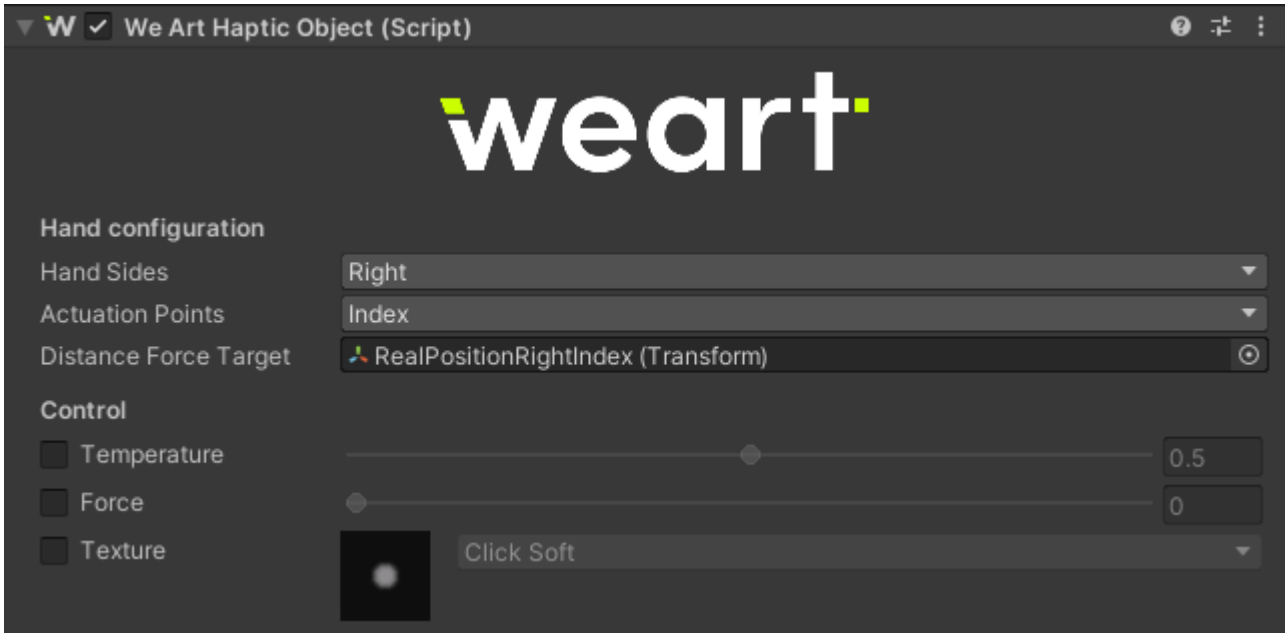
WeArtHapticObject

Component responsible for the haptic actuation of the individual digital devices belonging to the TouchDIVER device Properties:

- Hand Side Flag – Which device it belongs to (RIGHT | LEFT)
- Actuation Point Flag – Target of the thimble or the thimbles on which to apply implementation
  - TouchDIVER Pro (Thumb | Index | Middle | Annular | Pinky | Palm) [Multi selection]
  - TouchDIVER (Thumb | Index | Middle) [Multi selection]
- Distance Force Target - Target for computing dynamic force based on distance, if left empty, there will be no calculation and the touchable object force will be applied as it is to the haptic object
- Control: - (Set by TouchableObject during interaction or manually by developer)
  - Temperature – Temperature value implemented on the target thimble or thimbles (from 0.0 to 1.0) [0.5 is environment temp – 0.0 really cold – 1.0 really hot]
  - Force – Force value applied on the target thimble (s) (from 0 to 1) [0.0 no force – 1.0 max force]
  - Texture – Type of texture rendered on the thimble or target thimbles (from 0 to N)

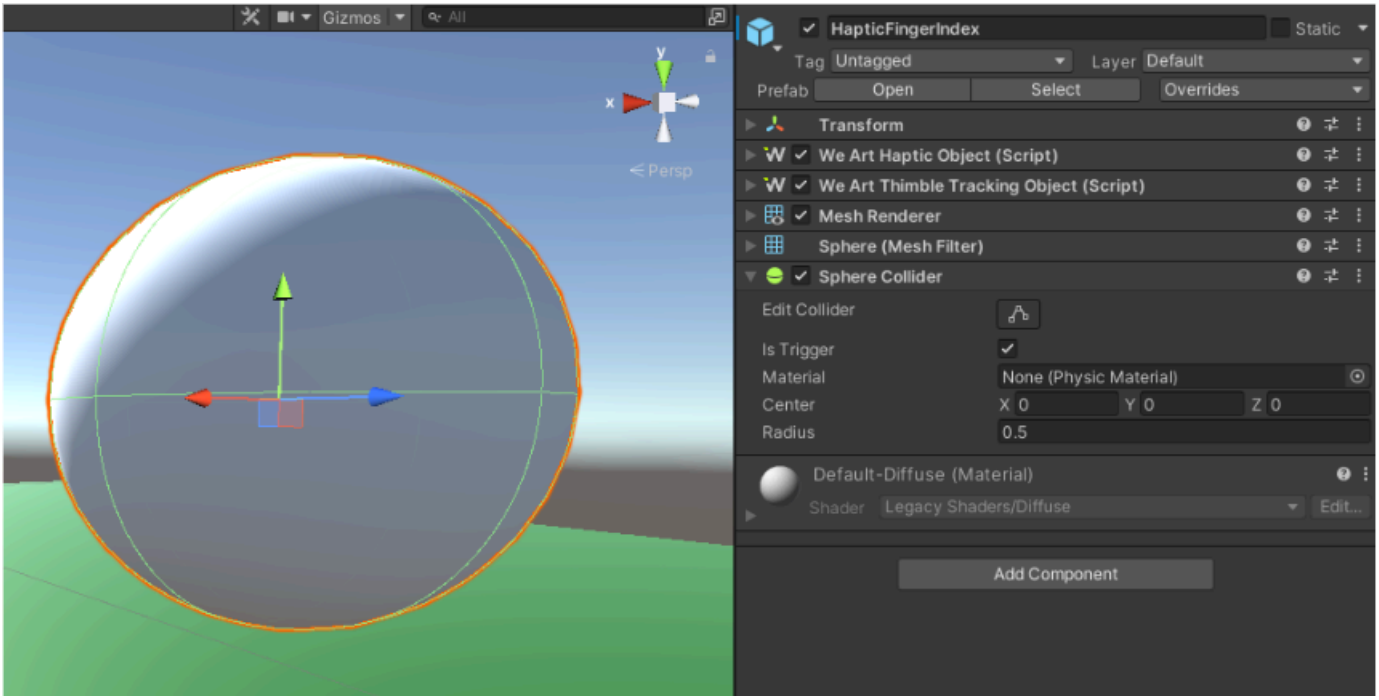
> **Note**

> The sending of actuation data WeArtHapticObject to WeArt App will be started only after calibration process completed.



GameObject Requirements:

- Collider
  - Is Trigger = True



WeArtThimbleTrackingObject

Component responsible for tracking of thimble' movements for quantifying its closed state and animating virtual hands Properties:

- Hand Side Flag – Which device it belongs to (RIGHT | LEFT)
- Actuation Point Flag – Target of the thimble or the thimbles on which to apply implementation
    - TouchDIVER Pro (Thumb | Index | Middle | Annular | Pinky) (Palm's value is fixed, it will always return 1)
    - TouchDIVER (Thumb | Index | Middle)

From this component we can get two variables:

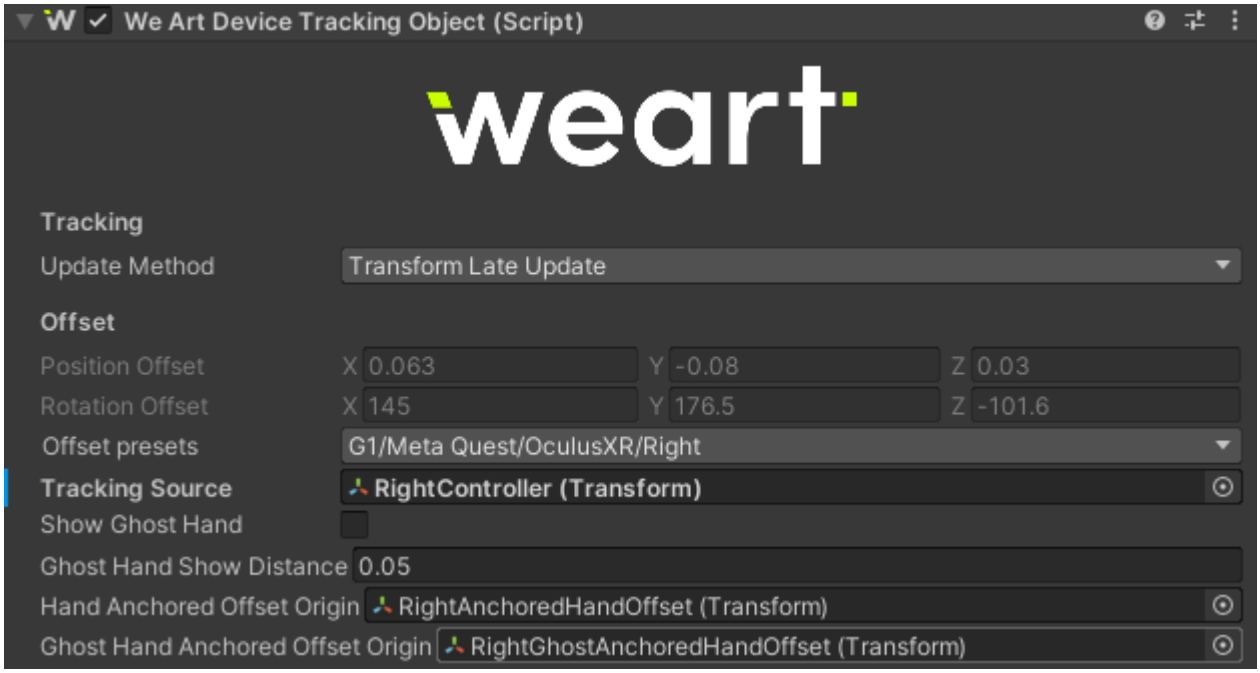
- Closure (the value from 0 to 1 that shows how much the finger is closed in reality)
- Abduction (Applied only to the thumb, a value from 0 to 1 based on the ability of the thumb to become opposable, a value that represents the rotation of the index, middle, annular, pinky and palm will always return 0)

WeArtDeviceTrackingObject

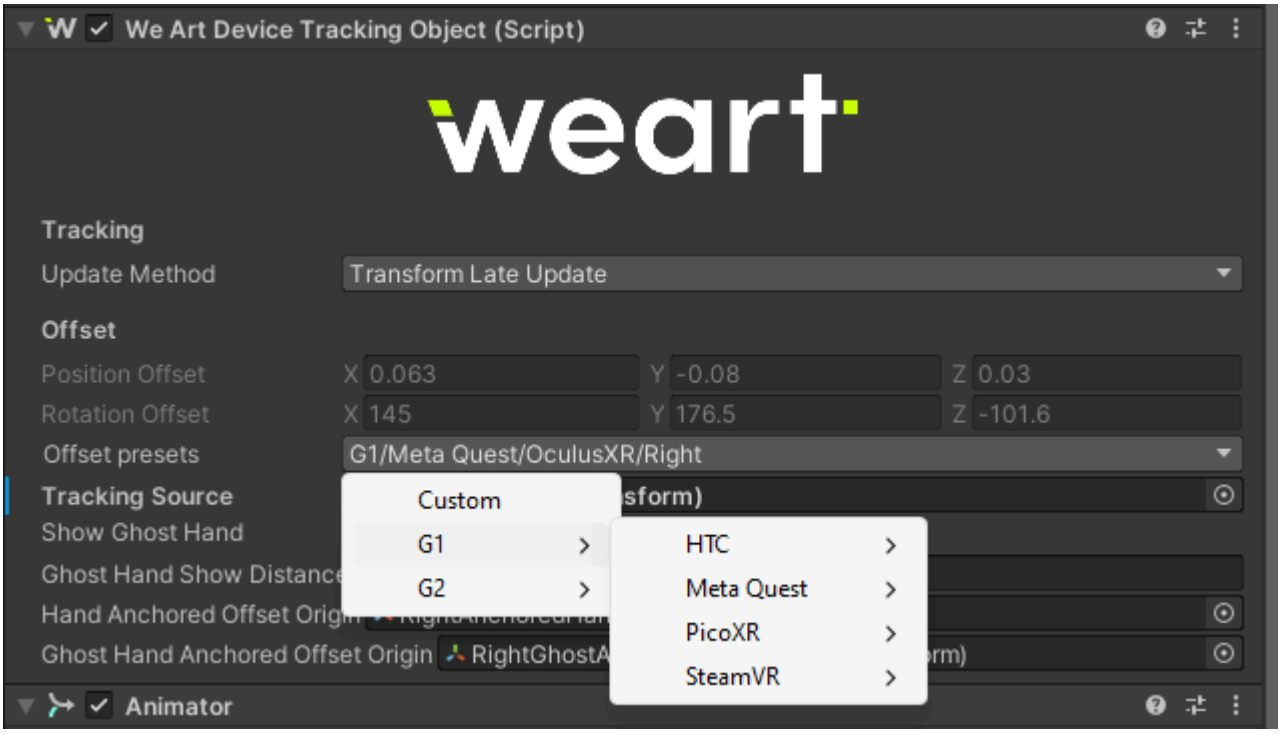
Component responsible for the tracking of the wrist on which the TouchDIVER device is placed Properties:

- Update method – Mode in which the position of the object is updated on the basis of the source tracking
- Position Offset – Transform position offset respect from the tracking source
- Rotation Offset – Transform rotation offset respect from the tracking source
- Offset presets – You can choose a preset offset parameters of specific platform controller like HTC Vive, Oculus 2, Oculus 3, Oculus Pro or in any case a custom configuration
- Tracking Source – Transform reference of motion controller source (ex. VR Controller or Vive Trackers, etc. etc.)
- Show Ghost Hand - When enabled it will show the real position of the hand, not affected by physics

- Ghost Hand Show Distance - The distance required to enable the ghost hand rendering
- Hand Anchored Offset Origin - For small interactable objects, this origin will be used to better apply the velocity
- Ghost Hand Anchored Offset Origin - Offset for calculating the velocity for small interactable objects



Here are the offset presets available:

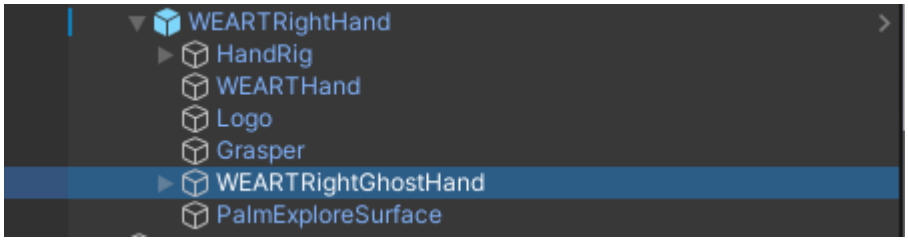


Offset preset

- **HTC** - base on the trackers, XR Elite controllers, Wrist trackers
- **Meta Quest** - base on the XR Plugin: OpenXR or OculusXR
- **PicoXR** - compatible with controllers for Pico 4 Ent and Pico 3 Neo

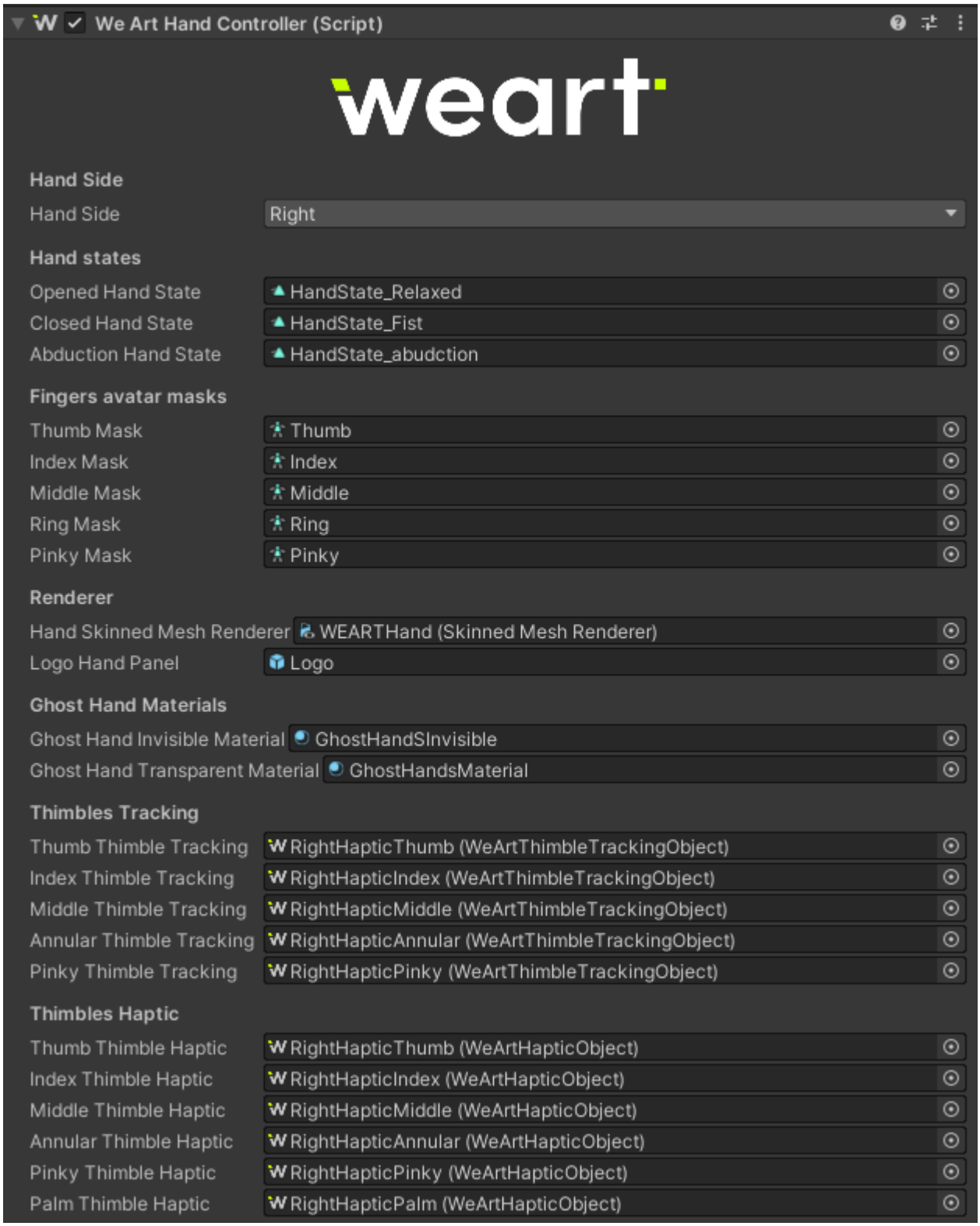
- **SteamVR** - compatible with SteamVR interface

Inside each hand prefab there is a child game object called GhostHand. This game object must not be deleted as it is essential for the hand grasping system.



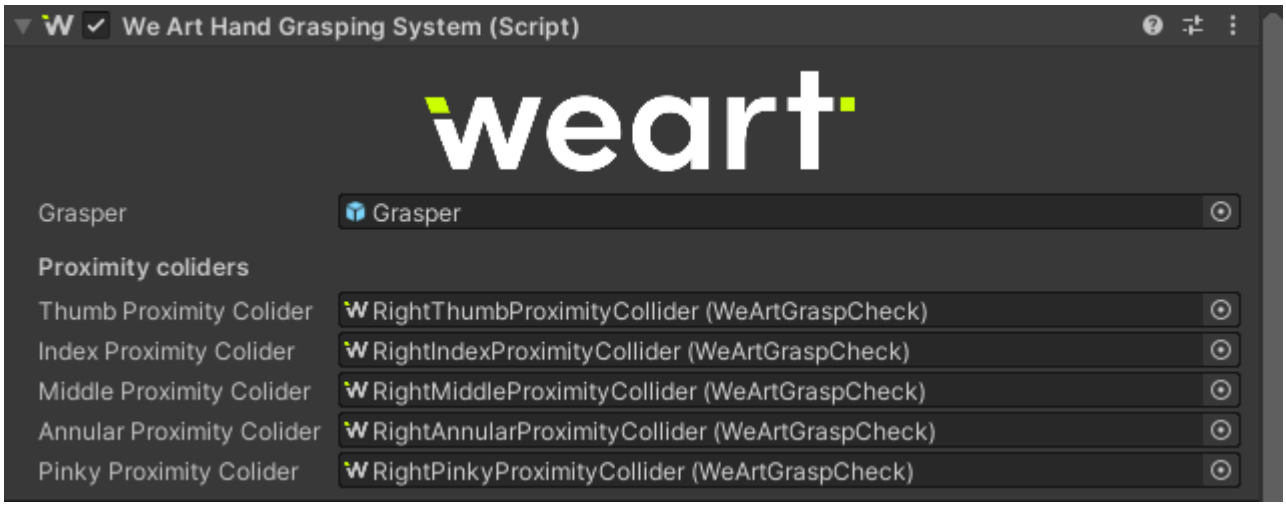
WeArtHandController

Each hand prefab has a script called WeArtHandController that takes care of the physical hand animation and helps the communication with WeArtHandGraspingSystem and WeArtHandSurfaceExploration components that are present on the same game object.



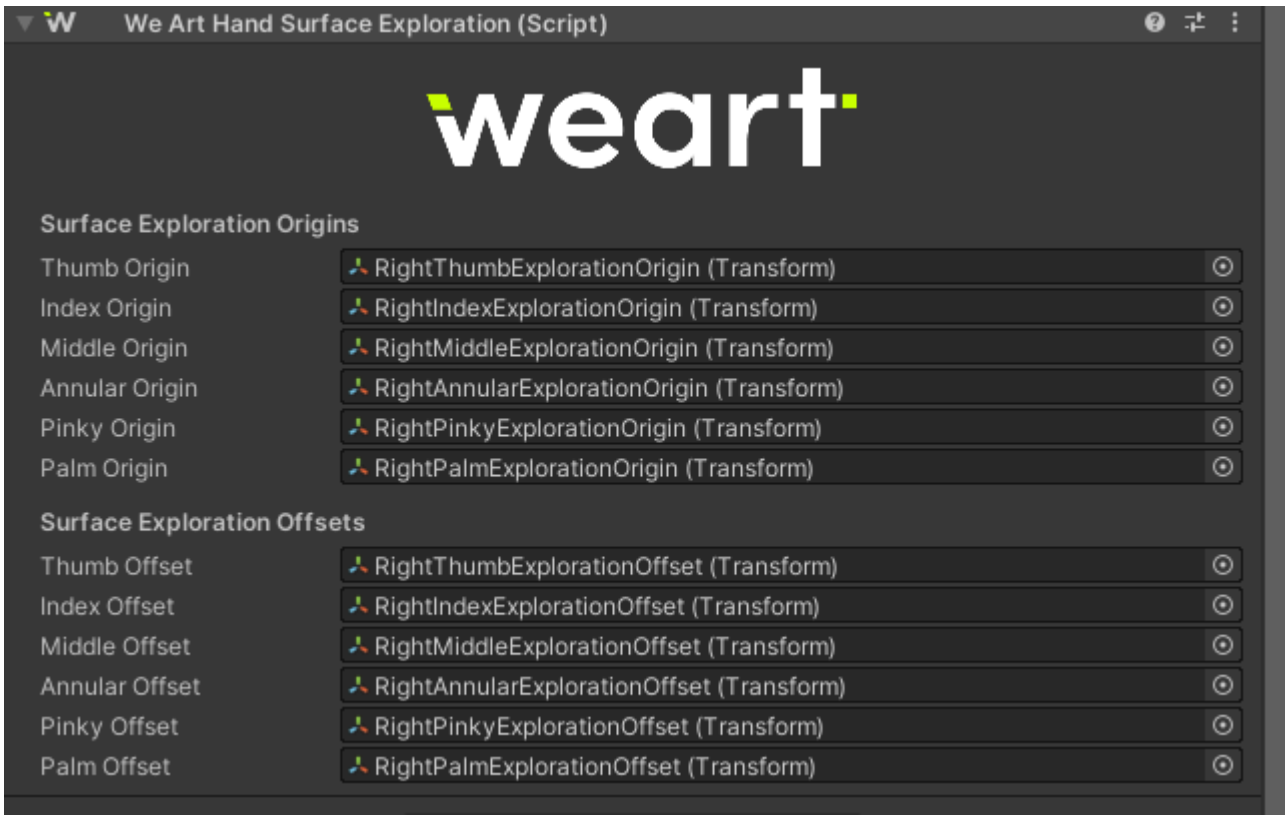
WeArtHandGraspingSystem

The grasping system component is responsible for the interaction with touchable objects and manages the effects that are applied to the haptic objects present on the fingers.

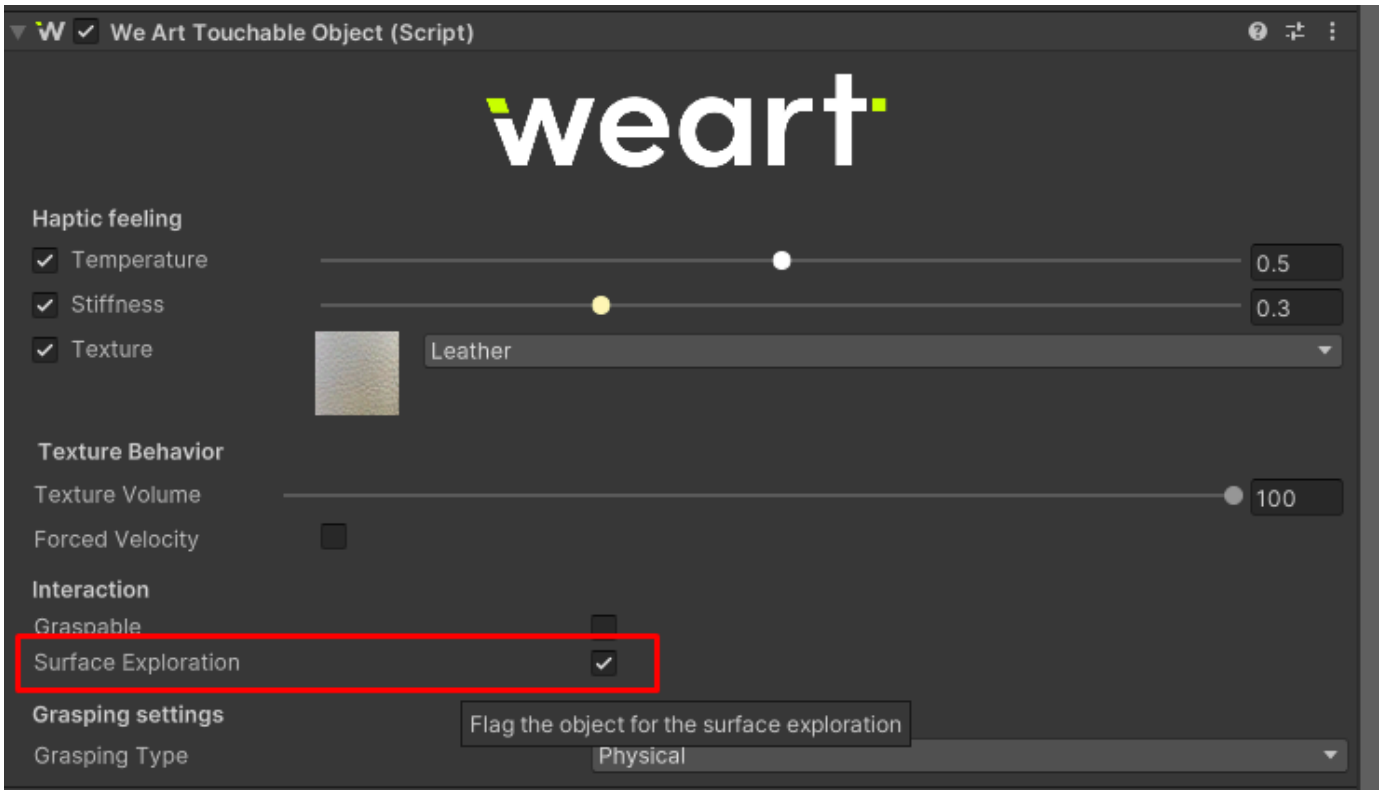


WeArtHandSurfaceExploration

The surface exploration component allows for a better exploration on touchable object with the SurfaceExploration checkbox enabled

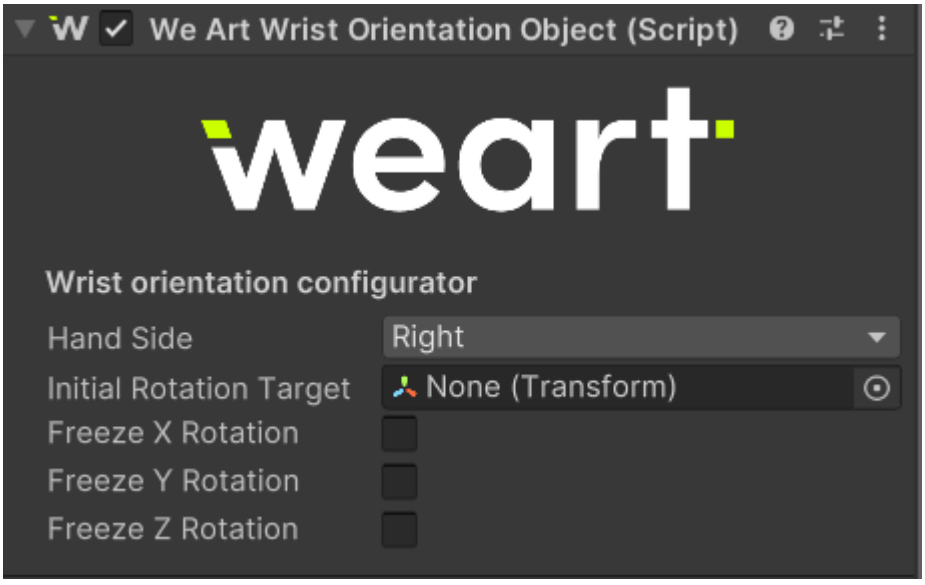


We can enable the surface exploration of a touchable object by enabling the following field



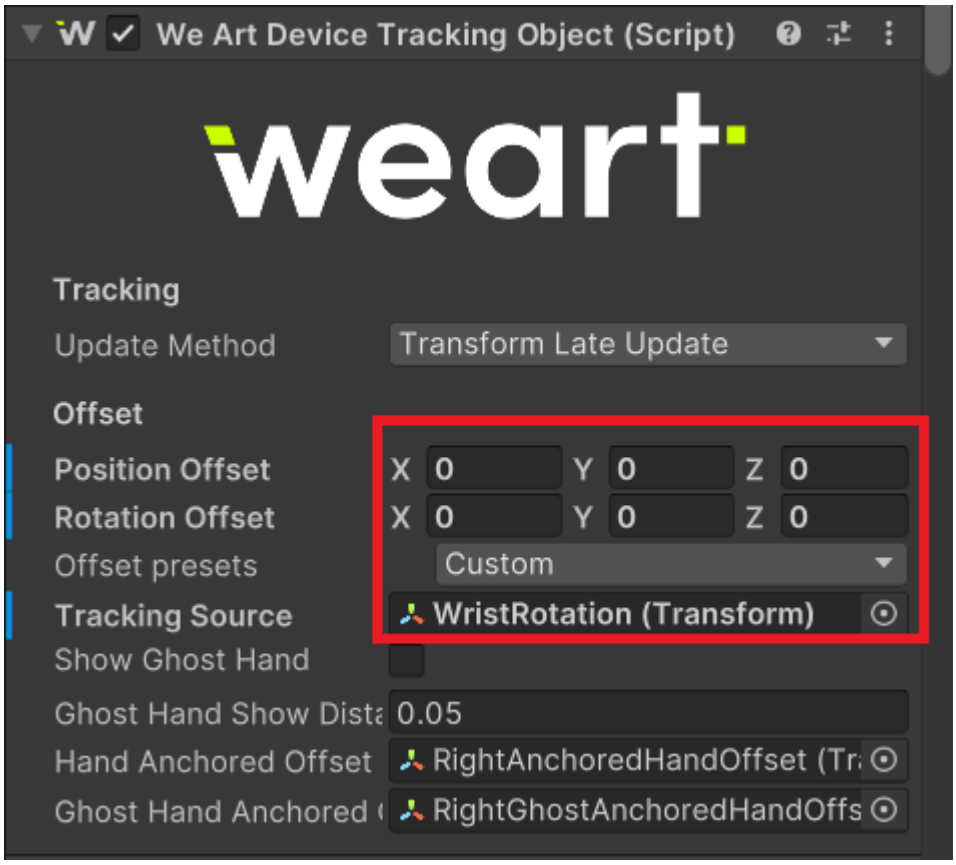
WeArtWristOrientationObject

The component that can read the wrist orientation data directly from Touch Diver Pro. It can be used as an alternative of controller rotation data, but the in this case the implementation of position data receiving has to be implemented by developer itself.



This component can be added to any object to scene and it will be rotated in accordance to TD physical rotation. There is possibility to constrain any axis rotation. Also it is possible to set here any object at scene to use it initial rotation at the start of the application's run and further wrist rotation calculations will include this initial data. In case object is not set, the attached game object initial rotation will be taken into considerations.

GameObject with attached WeArtWristOrientationObject component can be used as data source for WeArtDeviceTrackingObject, but there is should be set the custom offsets.



WeArtStatusTracker

The WeArtStatusTracker component is responsible for tracking and forwarding WeArt App status updates to the interested objects.

The interested objects can add a listener to the _OnMiddlewareStatus event, in order to receive the updates whenever the WeArt App updates its status.

In particular, the event propagates an object of type MiddlewareStatusData, containing:

- The current WeArt App version
- The current status of the WeArt App
- Whether actuations are enabled or not

- The last status code received from the WeArt App and, if not ok, a description of the error
- The status of the connected TouchDIVERs
    - Mac Address
    - Battery level (and will show a bolt symbol if the device is charging)
    - Calibration status during the session
    - Status of each thimble (shown on the hand icon by the three colored dots)

WeArtThimbleSensorObject

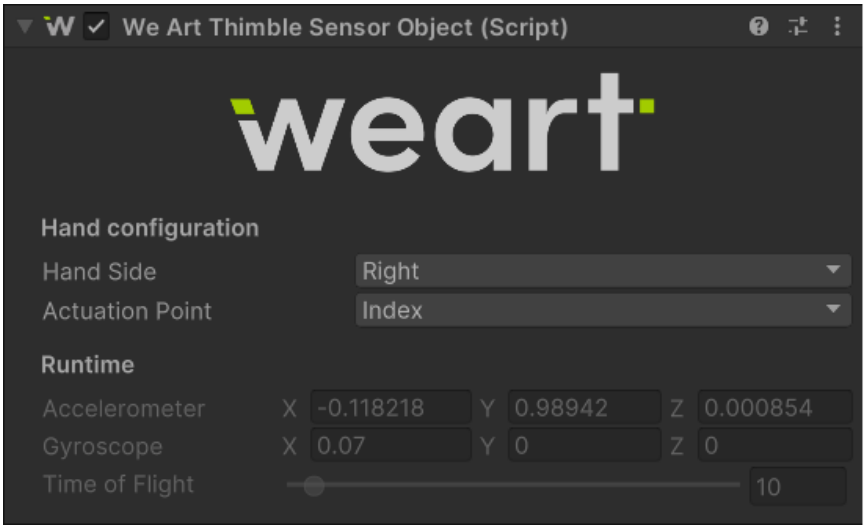
Component responsible for tracking a given thimble raw sensors values. The raw sensors data includes:

- Accelerometer (x,y,z)
- Gyroscope (x,y,z)
- Time Of Flight sensor distance value

Properties:

- Hand Side Flag – Which device it belongs to (RIGHT | LEFT)
- Actuation Point Flag – Target of the thimble or the thimbles from which to get the sensors data
    - TouchDIVER Pro (Thumb | Index | Middle | Annular | Pinky | Palm)
    - TouchDIVER (Thumb | Index | Middle)

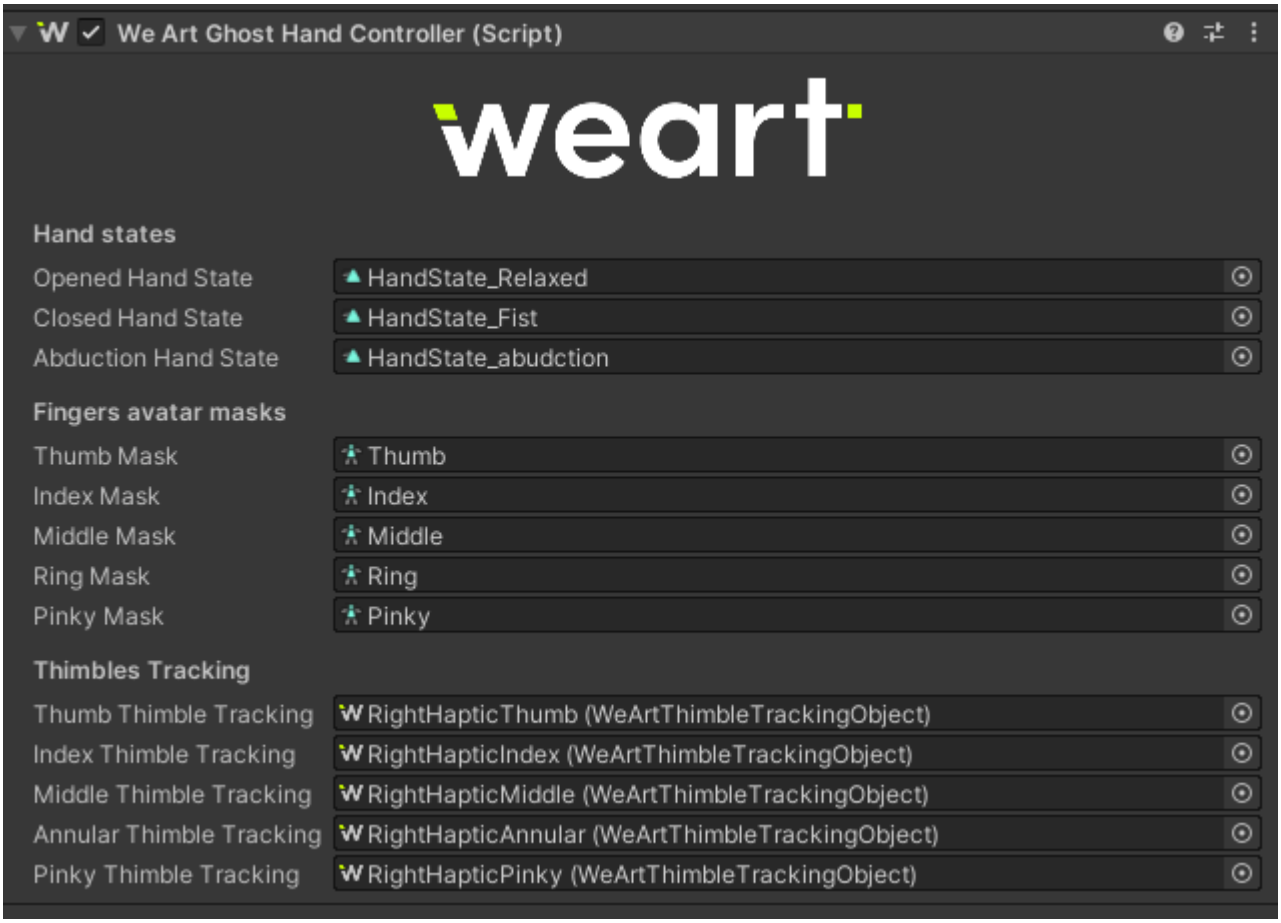
At runtime (while the editor is playing), the component will show the values received for the given thimble, as shown in the picture below.



WeArtGhostHandController

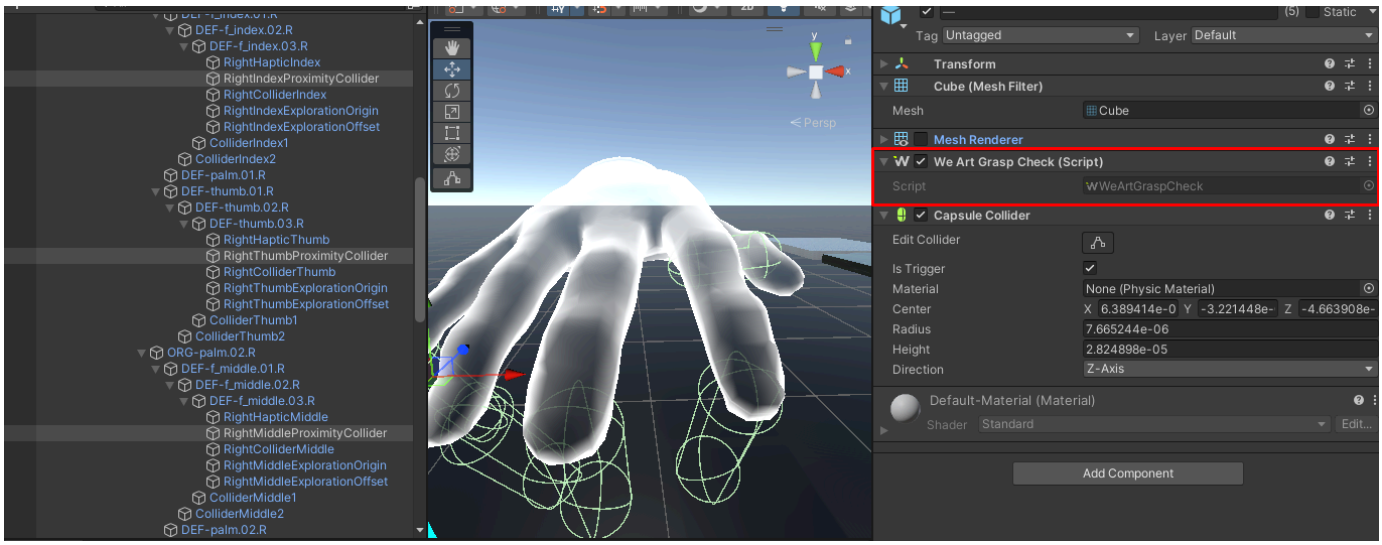
Each ghost hand has a script called WeArtGhostHandController. It is used by the WeArtHandController to animate the ghost hand and report the real positions of fingers in the

scene(not affected by physics)



WeArtGraspCheck

Each hand has three grasp checkers(touchable object proximity check) that have a script called WeArtGraspCheck, this is used by WeArtHandController to manage the grasping system.



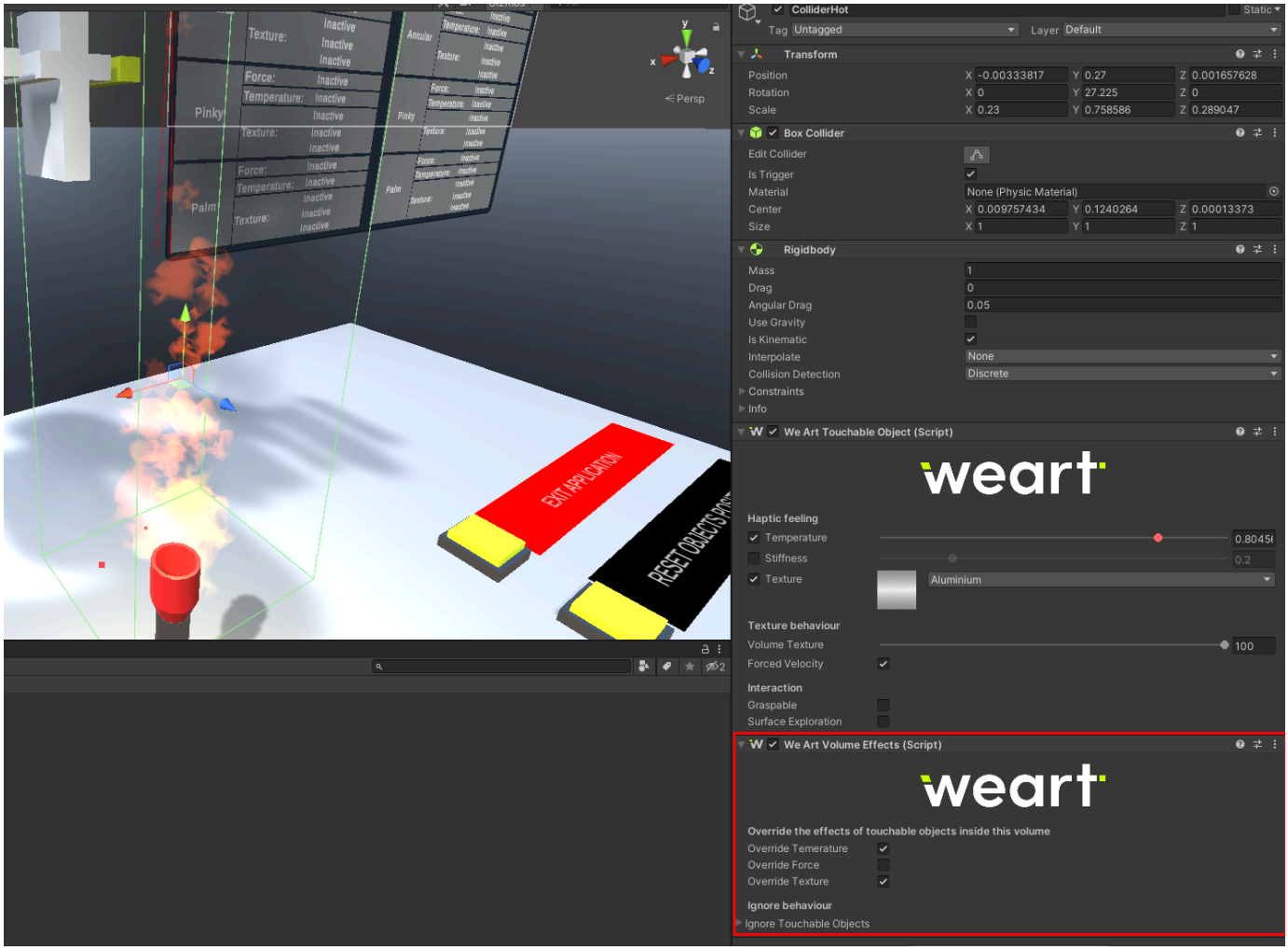
WeArtVolumeEffects

This component can be placed on a game object that contains a trigger collider, WeArtTouchableObject and a rigidbody. The WeArtTouchableObjects that enter this volume will

inherit its "Volume Temperature" as temperature. Even when holding the touchable object with the hand, it will also affect the temperature felt by the fingers. When a touchable object exits this trigger, it resets to the temperature that it had before entering.

Properties:

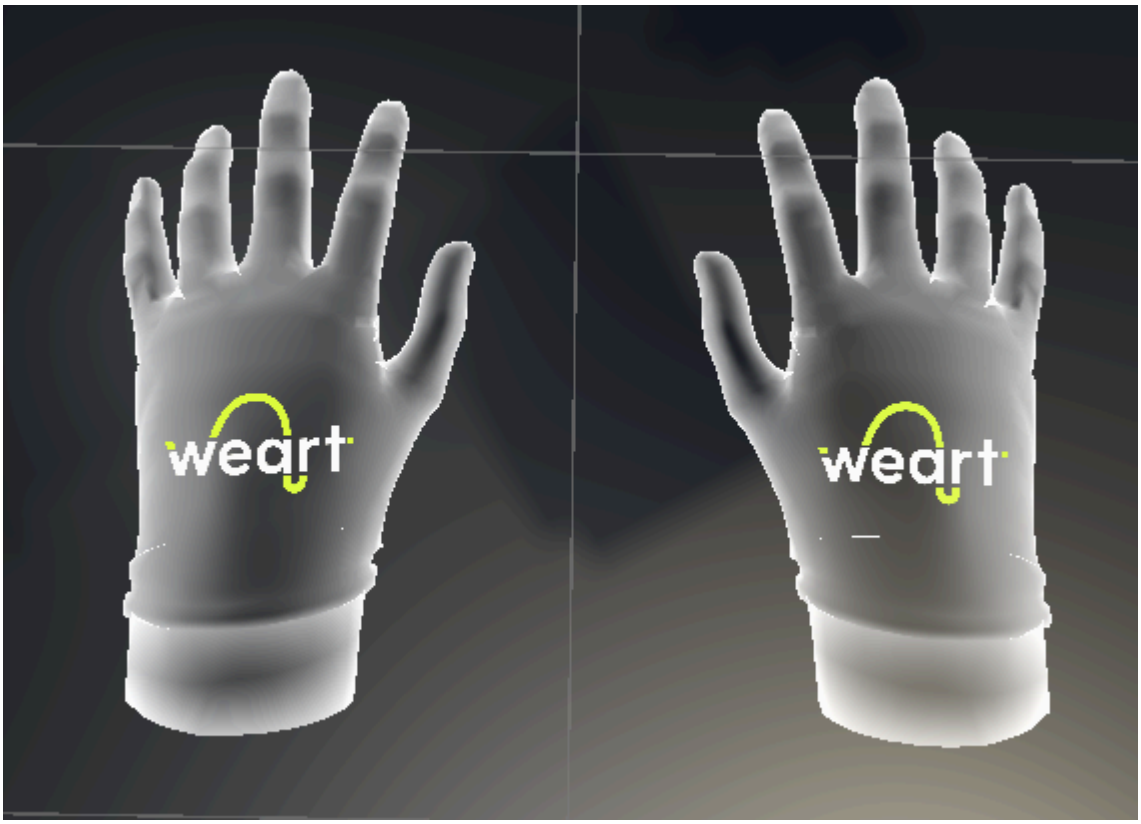
- Override Temperature (WeArtTouchableObjects that enter this volume will inherit this value as temperature)
- Override Force (WeArtTouchableObjects that enter this volume will inherit this value as Force)
- Override Texture (WeArtTouchableObjects that enter this volume will inherit this value as texture)
- Ignore Touchable Objects - A list of touchable objects that should not interact with this trigger



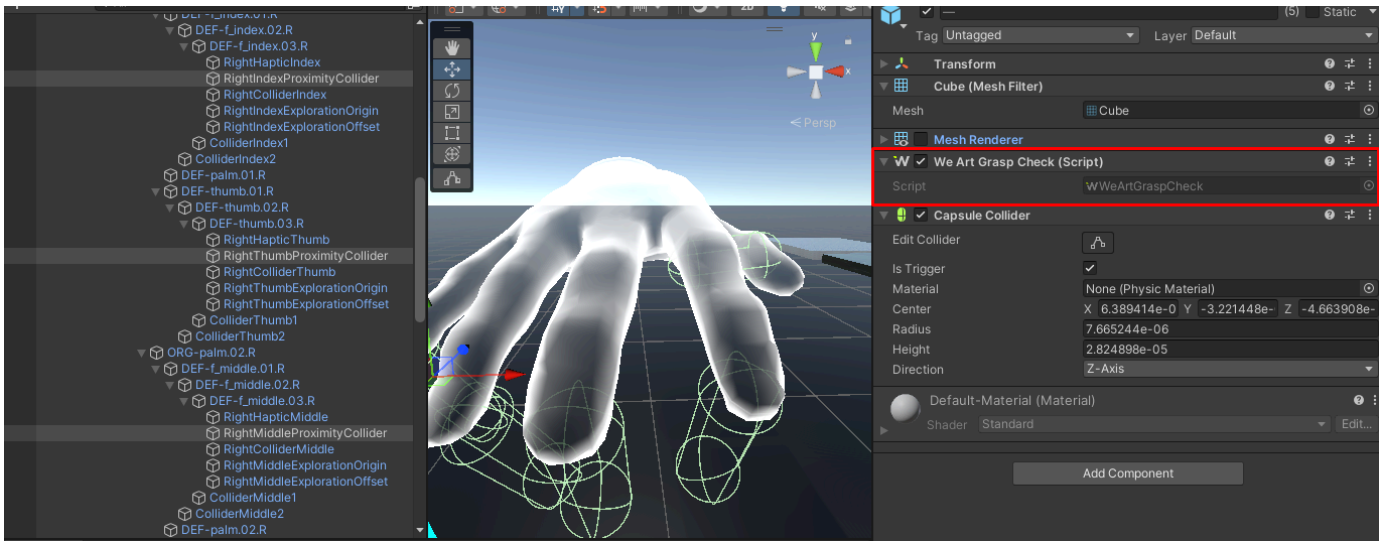
Hands System

Represents virtual hands in your experience linked to TouchDIVER and able to update finger tracking for activation hand animation and haptic for thimbles. In base of the project configuration,

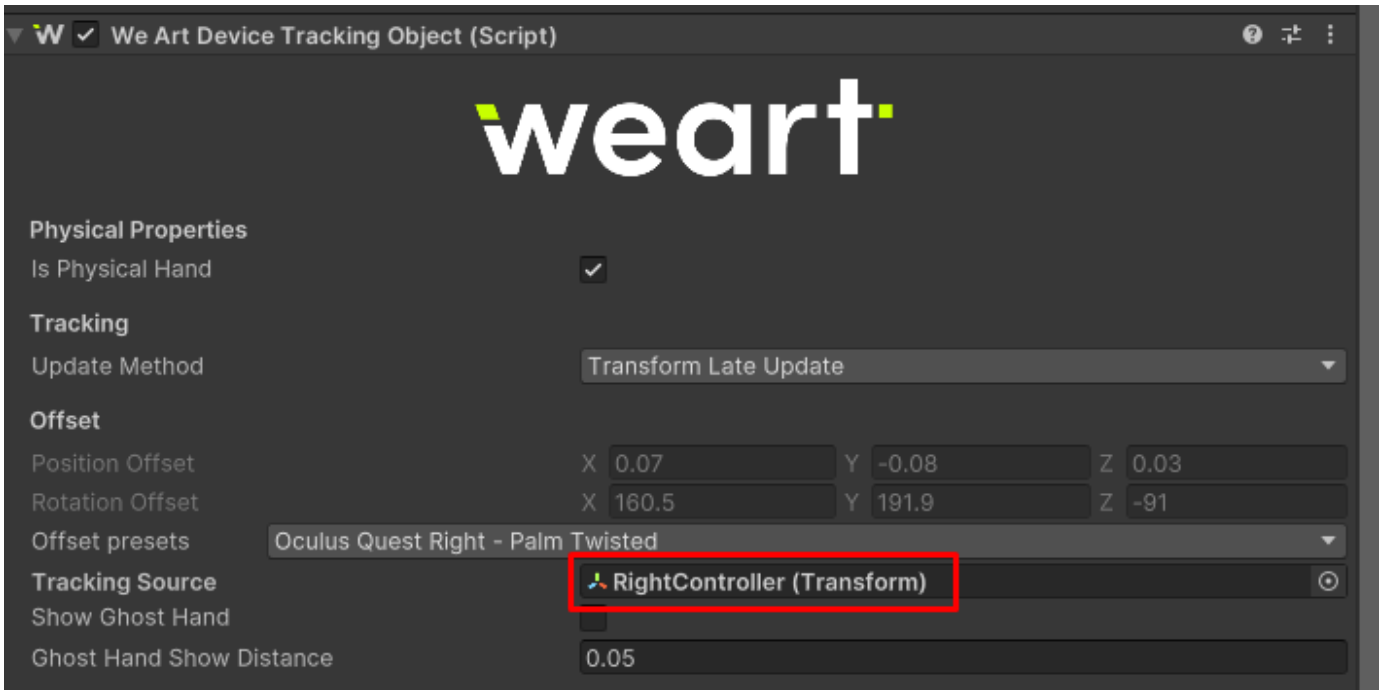
the hands can move with your motion controller. Path: Packages→WEART SDK → Runtime → Prefabs → WEARTLeftHand | WEARTRightHand



Each hand has three grasp checkers(touchable object proximity check) that have a script called WeArtGraspCheck, this is used by WeArtHandController to manage the grasping system.



If you are not using a sample scene, you can drag and drop the prefabs in the scene, make sure to apply the transform that it needs to follow of your VR rig controller. Make sure to add WeArtController to the scene.



Hand Grasp Events

Each WeArtHandController has two events. One fires when an object is grasped and the other when it is released.

Create two functions that will be subscribed to the events

```
void OnRightGrasp(HandSide handSide, GameObject gameObject)
  {

  }

void OnRightRelease(HandSide handSide, GameObject gameObject)
  {

  }
```

And then subscribe them to the hand controller.

```
rightHandController.GetGraspingSystem().OnGraspingEvent += OnRightGrasp;
rightHandController.GetGraspingSystem().OnReleaseEvent += OnRightRelease;
```

> **Note**
> Do not subscribe to the events in an Awake() method as the GraspingSystem will not be linked yet to the HandController

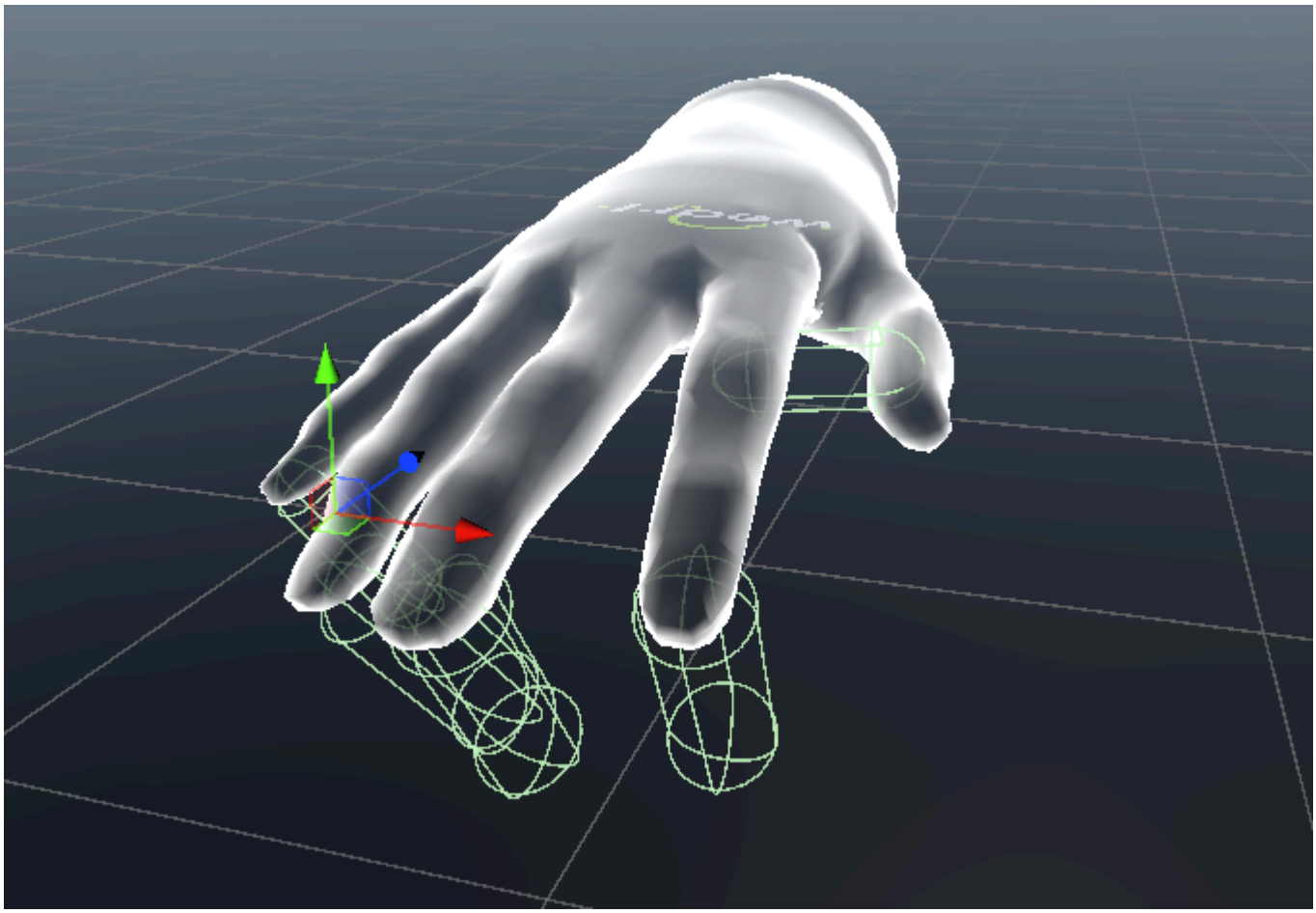
Grasping System

The physical hand is animated using the closure values of the WeArtThimbleTrackingObjects present on the Thumb, Index, Middle, Annular and Pinky fingers. Through out the hand there are colliders, the palm has colliders that never move while the fingers have colliders that move because they are placed on the transforms of the phalanges of the fingers that are moved by the animation. Each Fingertip of thumb, index, middle, annular, pinky and palm have a capsule collider with a WeArtHapticObject component. When the hand does not grab anything, the haptic objects interact with the touchable objects in the scene and receive their effects.

Each hand also contains a palm collider(box).

Moreover, each fingertip of thumb, index, middle, annular and pinky have capsule colliders(Proximity Checker Collider) that also contain WeArtGraspChecker. These Objects do not affect the effects felt by the Touch Diver. They are only present to check what WeArtTouchableObjects are in contact with them.

Here you can see the "Proximity Checker Colliders"



The Unity SDK now integrate a new hand system, ready-to-use, that enables physical interaction with objects, making the experience more realistic and enhancing tactile feedback during the interaction.

With the new hand asset, it is possible to grasp objects naturally, without predefining static poses and speeding up the development of experiences that enable manipulation. Tactile feedback is synchronized precisely with object manipulation, increasing the realism of the feedback.

The new hand integrated into the SDK can be adopted in any project and allows for handling collisions with objects, applying forces, and making interactions more realistic.

Collider best practices

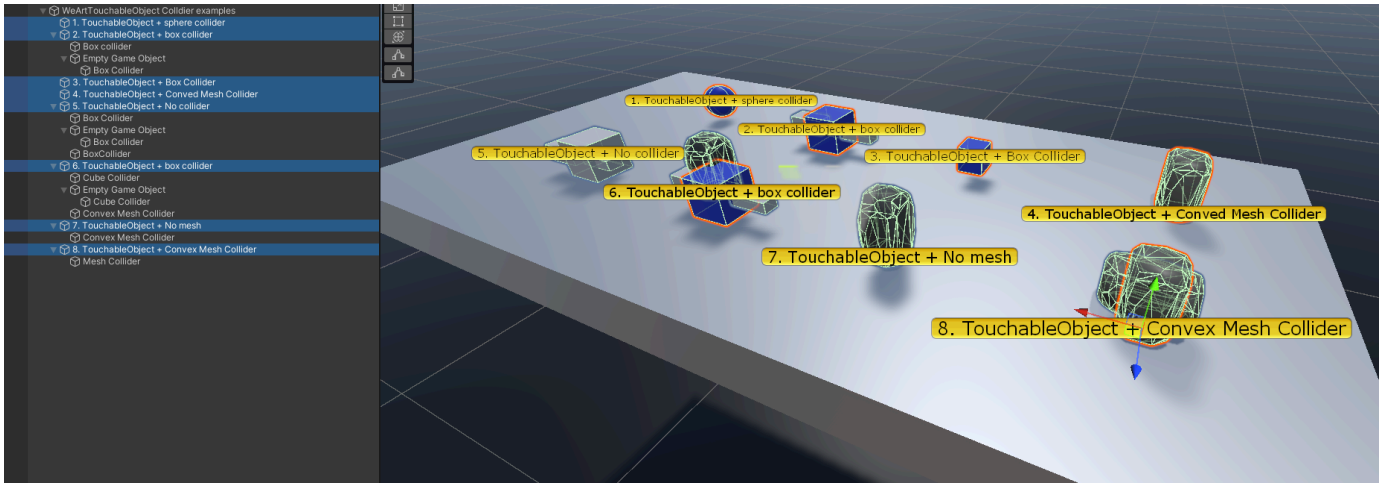
> **Note**
> The main requirement of the SDK is to have a main parent object where the WeArtTouchableObject will be located and the colliders can be placed on the game object itself or on its children. Mesh colliders must be convex.

The sdk allows for many flexible ways of setting up colliders as long as there is a main game object that has the WeArtTouchableObjects. The SDK counts disabled colliders as valid colliders. WeArtTouchable object has a firstCollider reference that will be used to direct the other components to a reliable collider. The firstCollider is searched first on the components of the game object that has the component WeArtTouchableObject. If this game object does not have a collider, it will continue to search recursively through its children and the first collider that is found, will be saved as firstCollider.

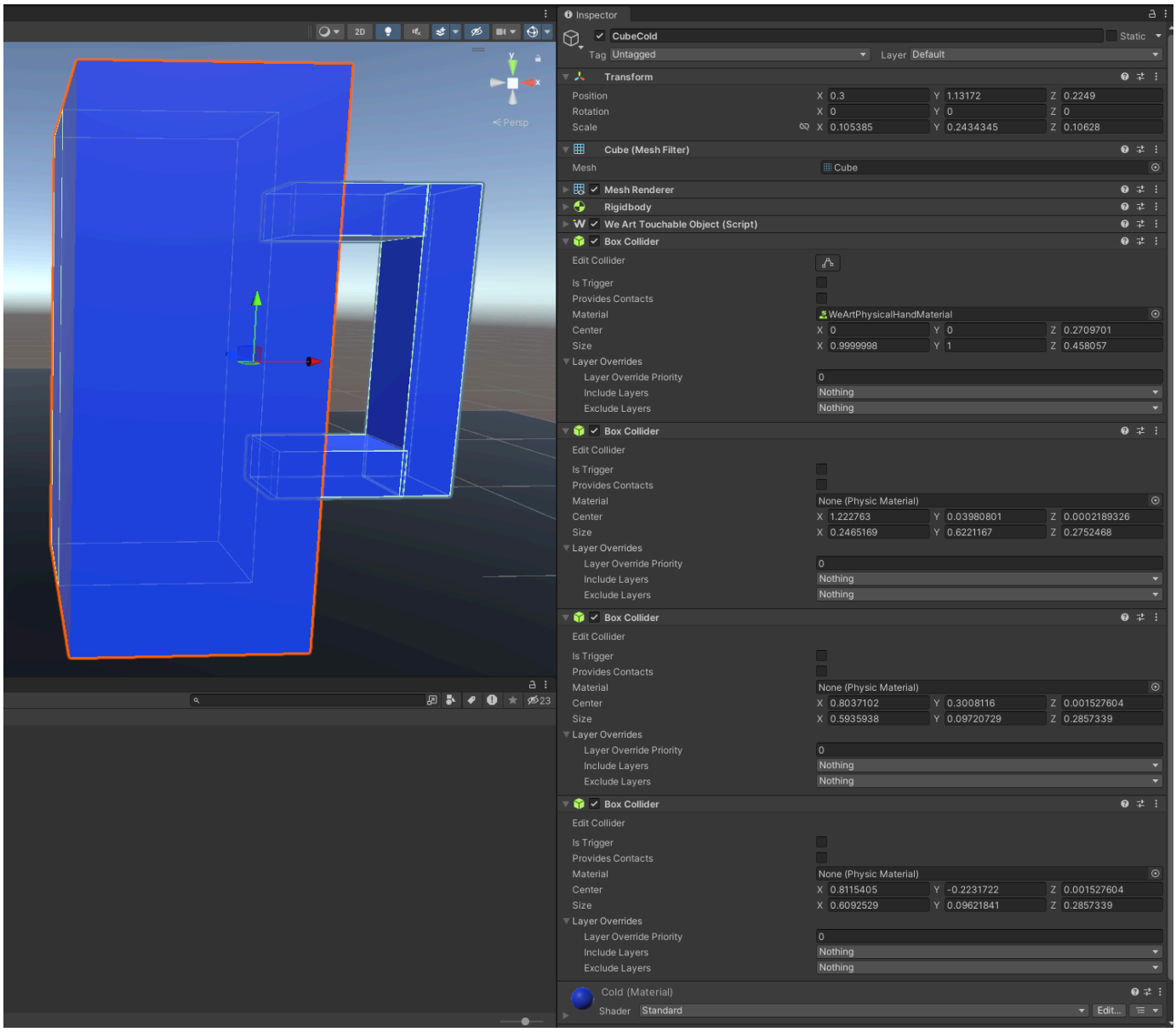
This enables us to create a lot of types of setups regarding the touchable objects' layouts.

> **Note**
> FirstCollider is essential for WeArtTouchableObject and it must not be deleted during play mode.

Here are some example of layouts for the touchable objects. Everything is allowed as long as there is a parent object with WeArtTouchable object component and as long as all mesh colliders are convex.

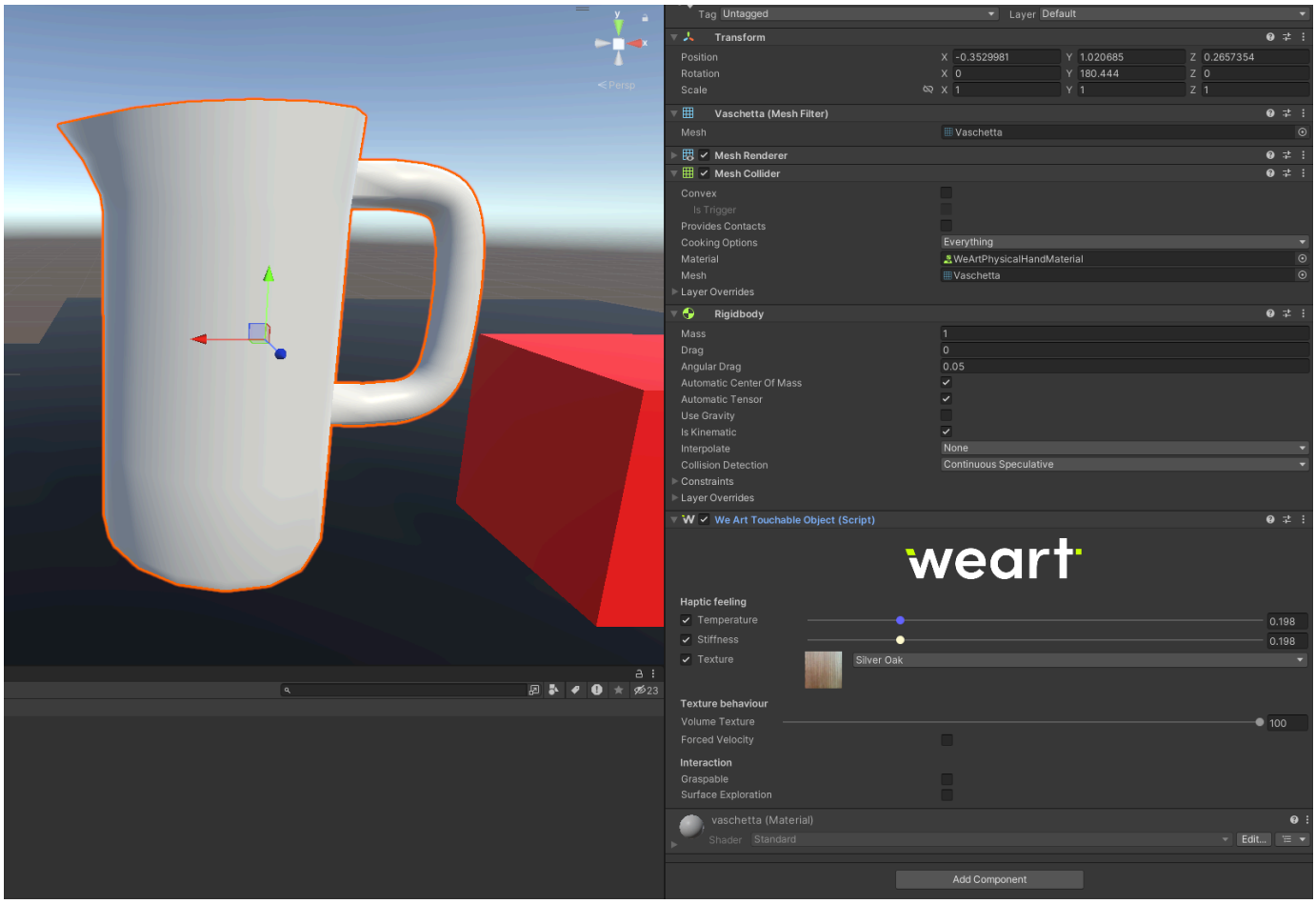


Here is an example of multiple colliders on the same game object.

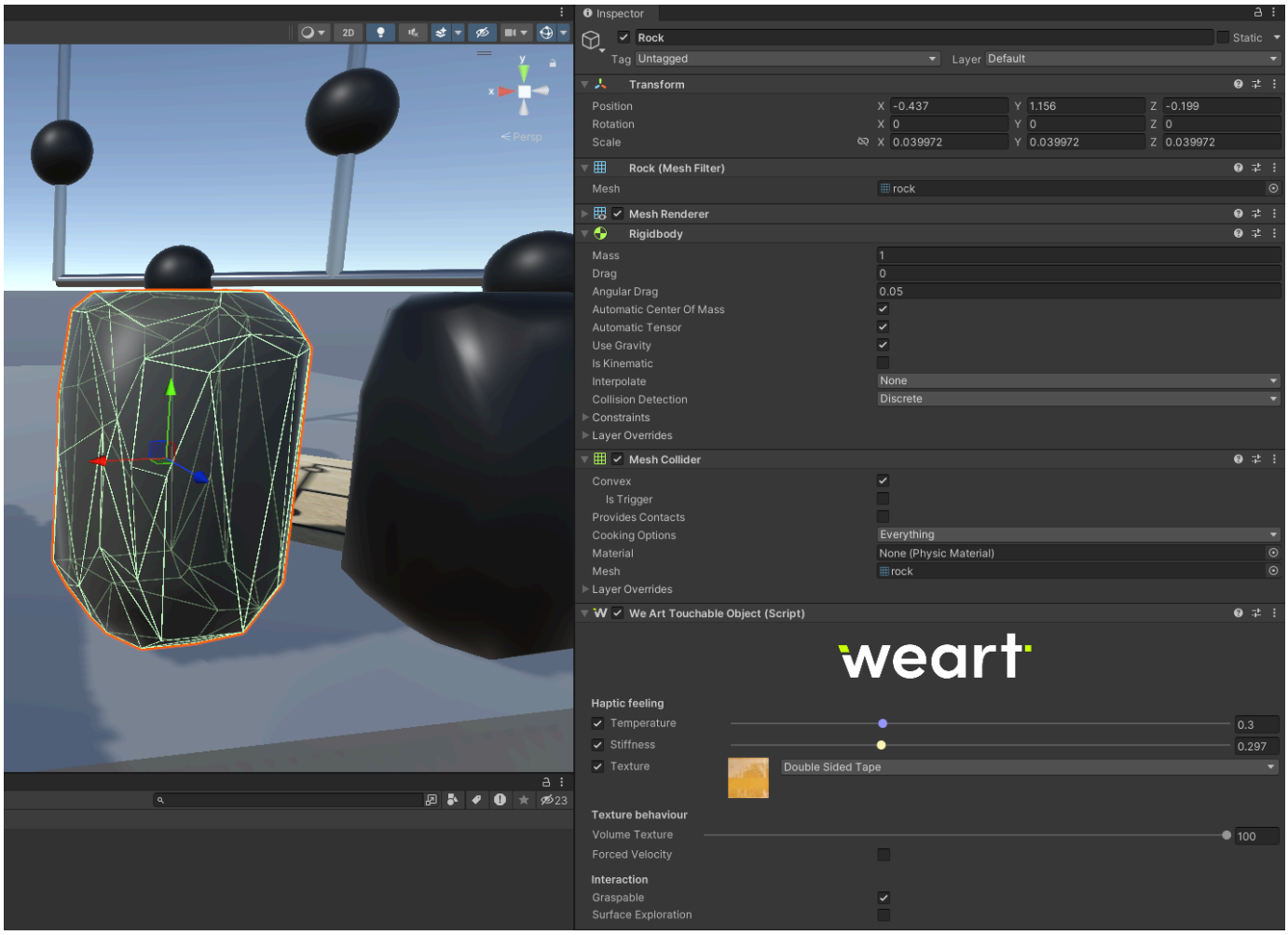


Unity does not allow moving Non Convex Meshes, when the non convex mesh collider is moved, it ignores any kind of collision. The mesh colliders must always be set to "convex".

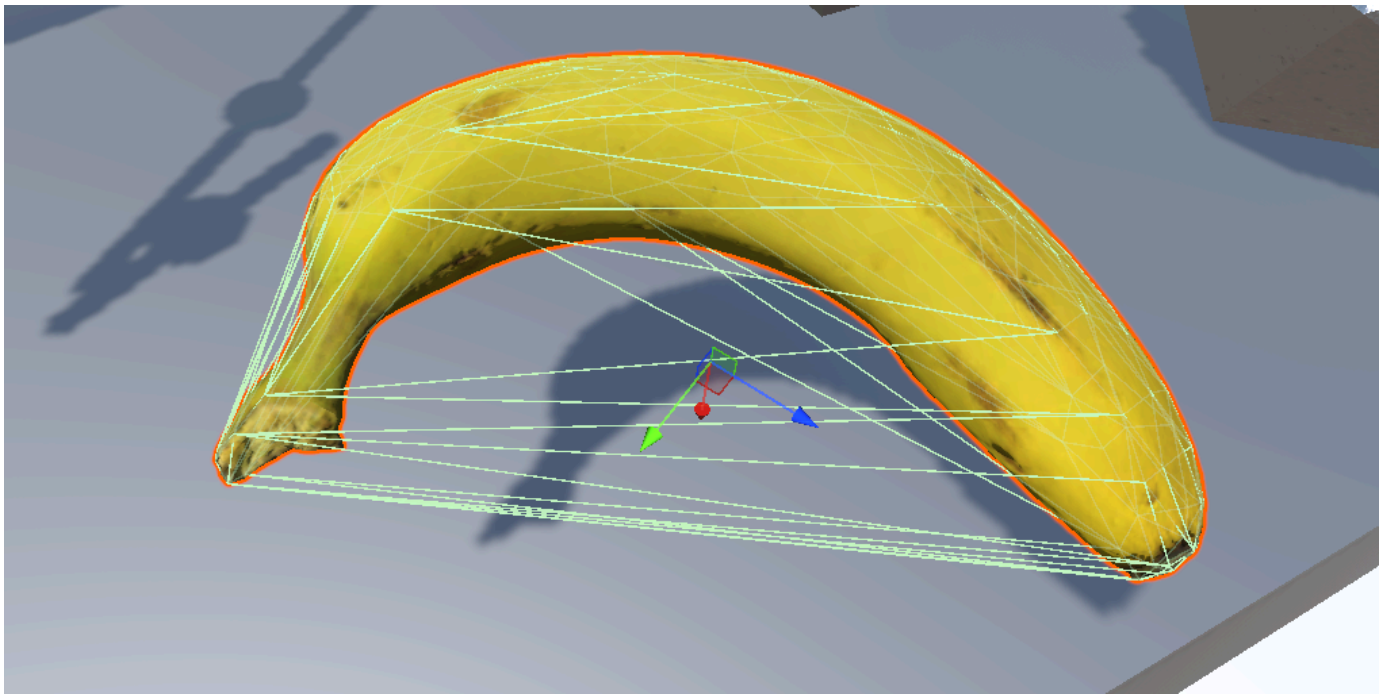
If you want to have static non convex mesh colliders. you can add them and set their rigidbody to kinematic. This will allow you to touch and feel the solid mesh, but you will not be allowed to move(or grab) this object. They will look like this.



There are situations in which the convex mesh collider works directly, but in most situations with more complex meshes this will not work. Here is an example of a convex mesh collider that works.

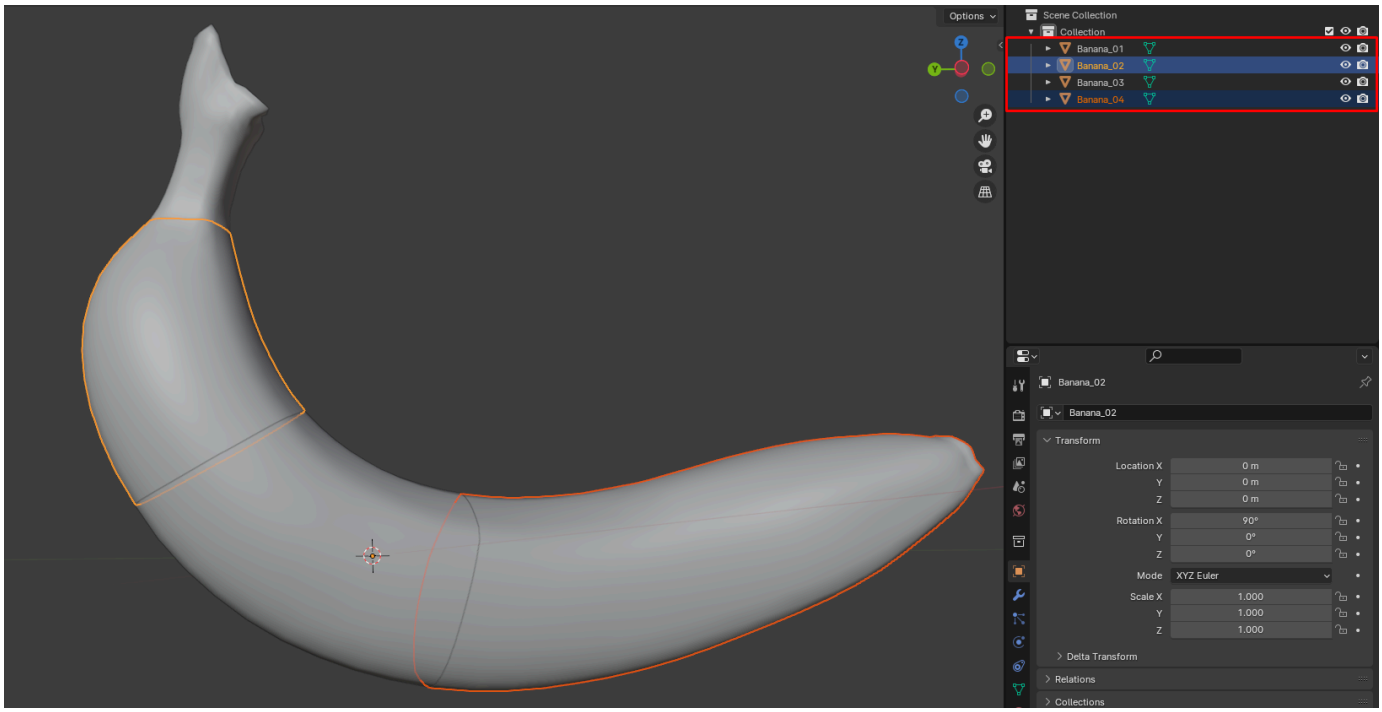


This is how the collider looks like if we enable the convex property on the mesh collider that is more complex.



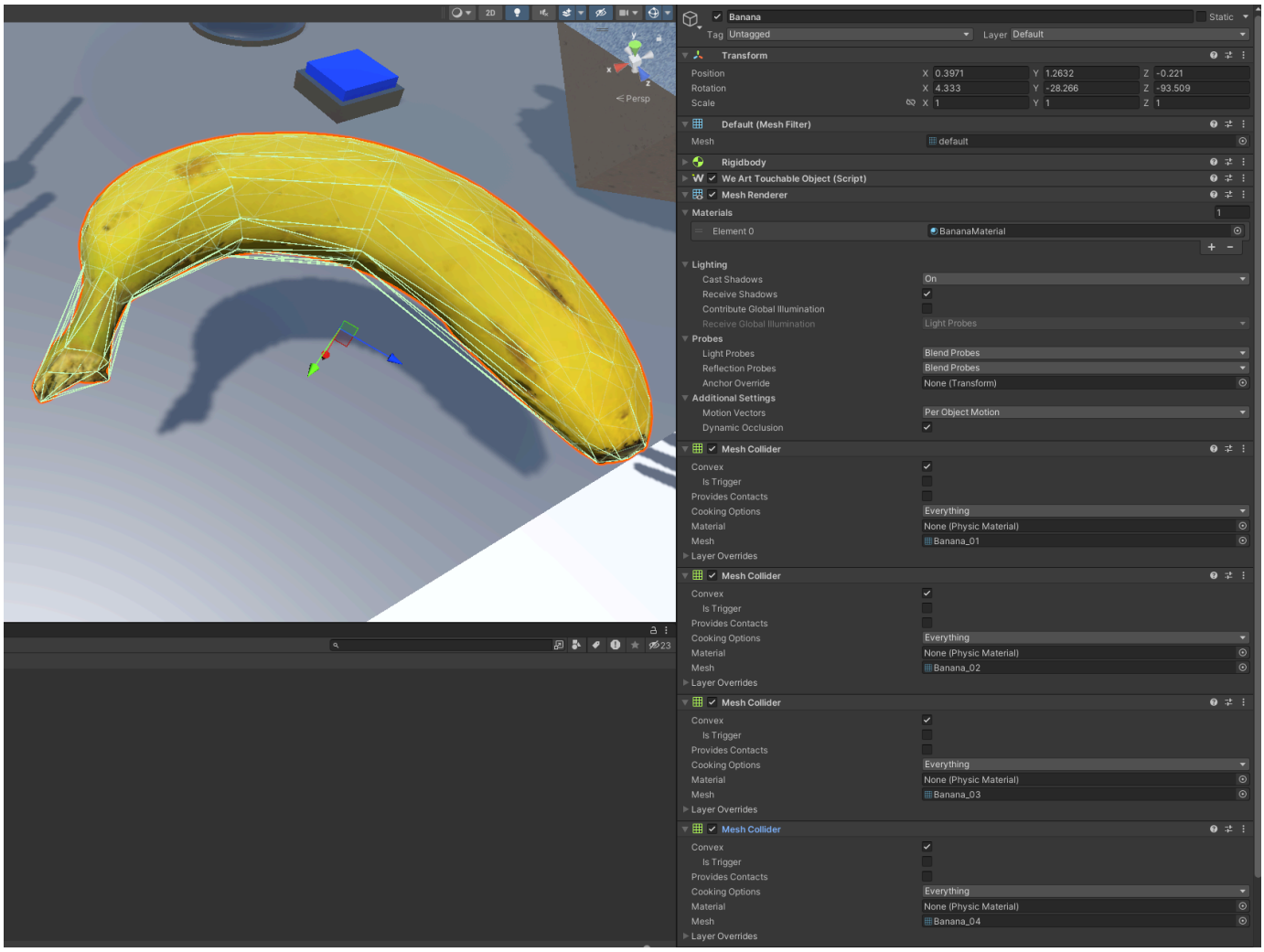
The convex function tries to wrap the whole mesh into a convex shape, and by doing this, the collider will be able to be touched and grabbed in places that the mesh does not exist.

We need to follow the next steps in order to have a good setup for our mesh. Take the mesh that you want to make grabbable into a model editing software. Divide the mesh into smaller parts that will make the convex shapes accurately.



Now in unity, keep the mesh renderer and Mesh filter the same. Add the mesh colliders components to the same game object and add the submeshes of the new model to the colliders' meshes. Make sure that the import size is right, you can check it by clicking on the model inside the project folder. Remember to make them convex.

The final result will look like this.



Easy Grasp

Easy Grasp is a type of snap grasping that was created in order to record a pose quickly with an intuitive procedure.

Easy Grasp is a feature used to record a grasping pose on a touchable object. Both hands can be recorded on the same touchable object. There cannot be more than one pose saved per hand per object. Recording the same hand again will override the existing pose. The fingers cannot move with this grasping type.

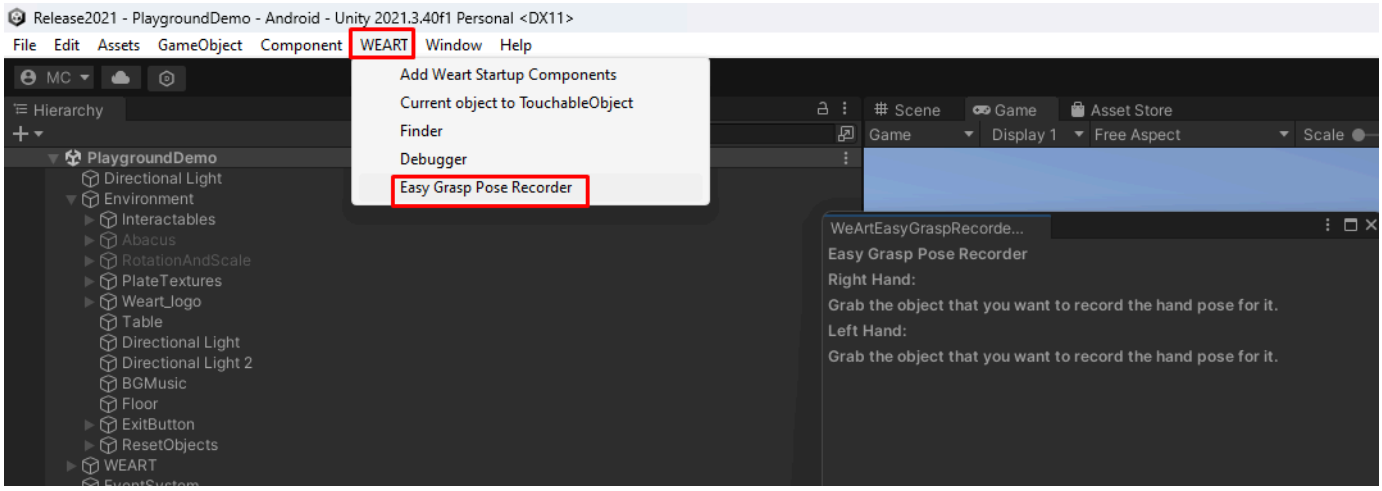
WeArtEasyGraspManager

WeArtEasyGraspManager takes care of recording and applying the poses during and after play mode. It is already present in WeArt Prefab. In case of not having the component in a scene and we want to use the component, we need to add the component to a game object of the scene. This component works only in editor and not in a build.

Recording an Easy Grasp pose

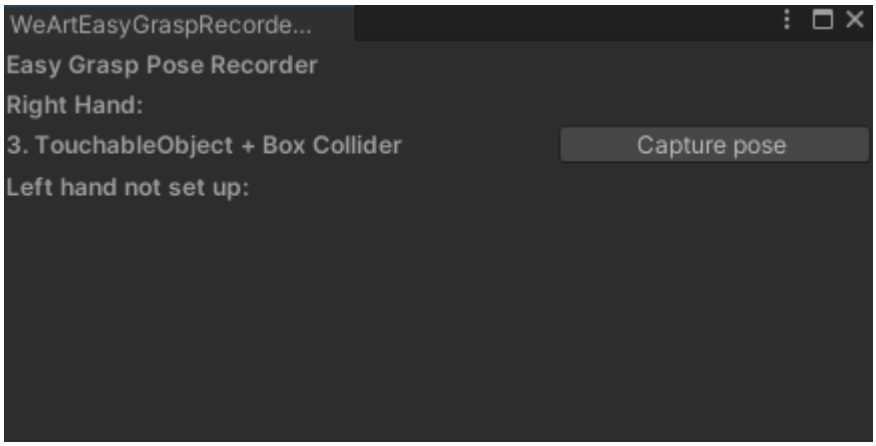
WeArtTouchableObject has 2 types of grasping systems: Physical and Snap.

First make sure that the touchable object grasping type is set to "Physical". Then grasp the touchable object with one of the hands. Now start the scene in "Play mode" and go to WEART -> Easy Grasp Pose Recorder.



This will show what are the hands grasping.

While grasping a touchable object, the window will show a "Capture Pose" Button.



Pressing the "Capture Pose" button during "Play mode", will save this pose when exiting "play mode". The system was created to save a pose during runtime and then to save it permanently. It is an automatic system that also allows the users to record multiple poses of different touchable objects and keep them after exiting "play mode".

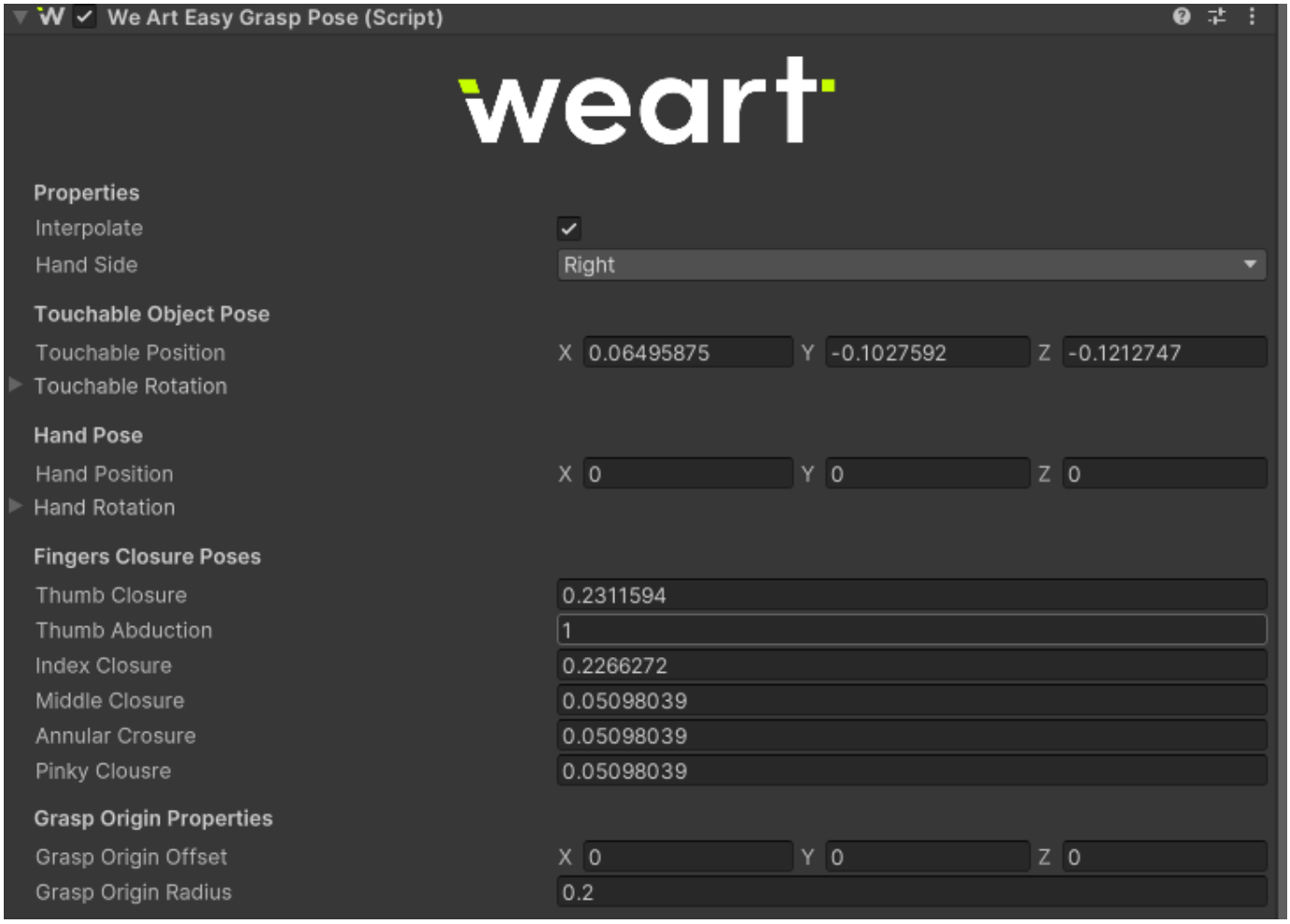
Finally after exiting play mode, set the touchable's grasping type to "Snap". This will allow the hand to "snap" grasp the touchable object.

> **Note**
> A touchable object that its grasping type is set to snap and does not have poses recorded on it, will make the hands unable to grasp the object. If the touchable object only has a pose for the right hand, the left hand will not be able to grasp it, it will require to record a pose with the left hand as well.

Easy Grasp Pose Recording Video Guide

Here is a video that shows the steps needed to record a easy grasp pose.

WeArtEasyGraspPose

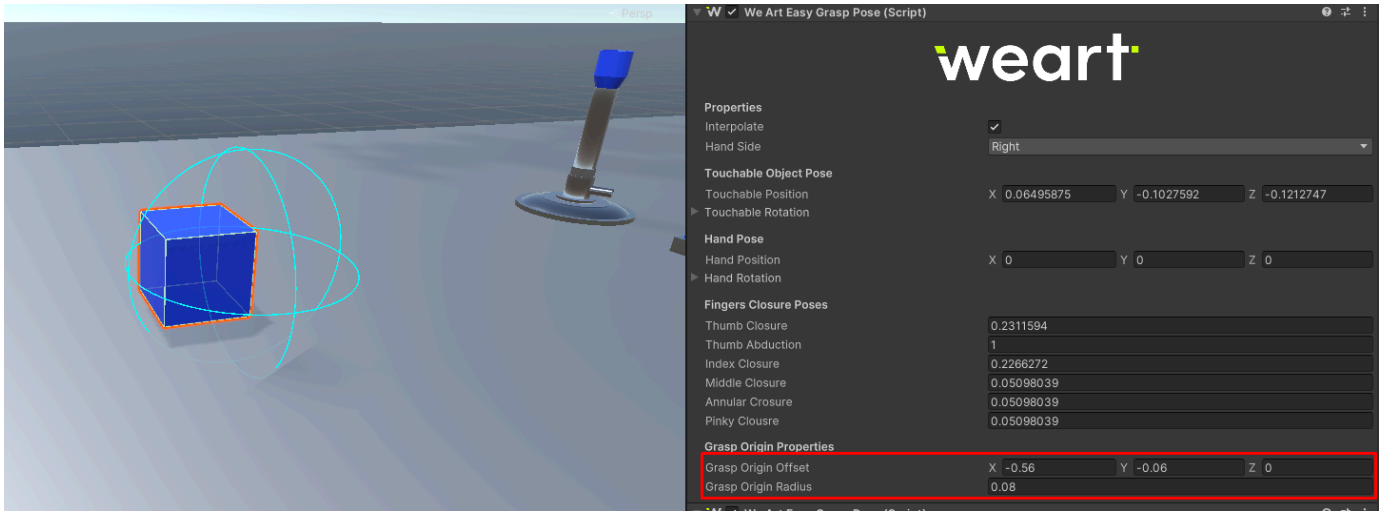


This component is created when the "Capture Pose" button is pressed, and it remains there after exiting "play mode". It stores data about the hand that was used to create the pose. As well as information about the position and rotation of the touchable objects when they were grasped, depending if the touchable objects were anchored or normal.

The component also keeps the closure of all fingers as well as the abduction value of the thumb.

GraspOriginOffset and GraspOriginRadius define the place where the touchable objects can be "snap" grasped. It will create a sphere in this position based on the radius. GraspOriginOffset works in local space.

Here you can see how GraspOriginOffset and GraspOriginRadius affect the grasping area of "Snap" objects



Note

Touchable objects with grasping type set to "Snap" can only be grasped in the area created by GraspOriginOffset and GraspOriginRadius. Physical grasping will not work on an object that its grasping type is set to "Snap". Adding child game objects with colliders or even complete touchable objects will allow the user to feel the effect but it will not allow grasping in that location if it is not in the snap grasp radius.

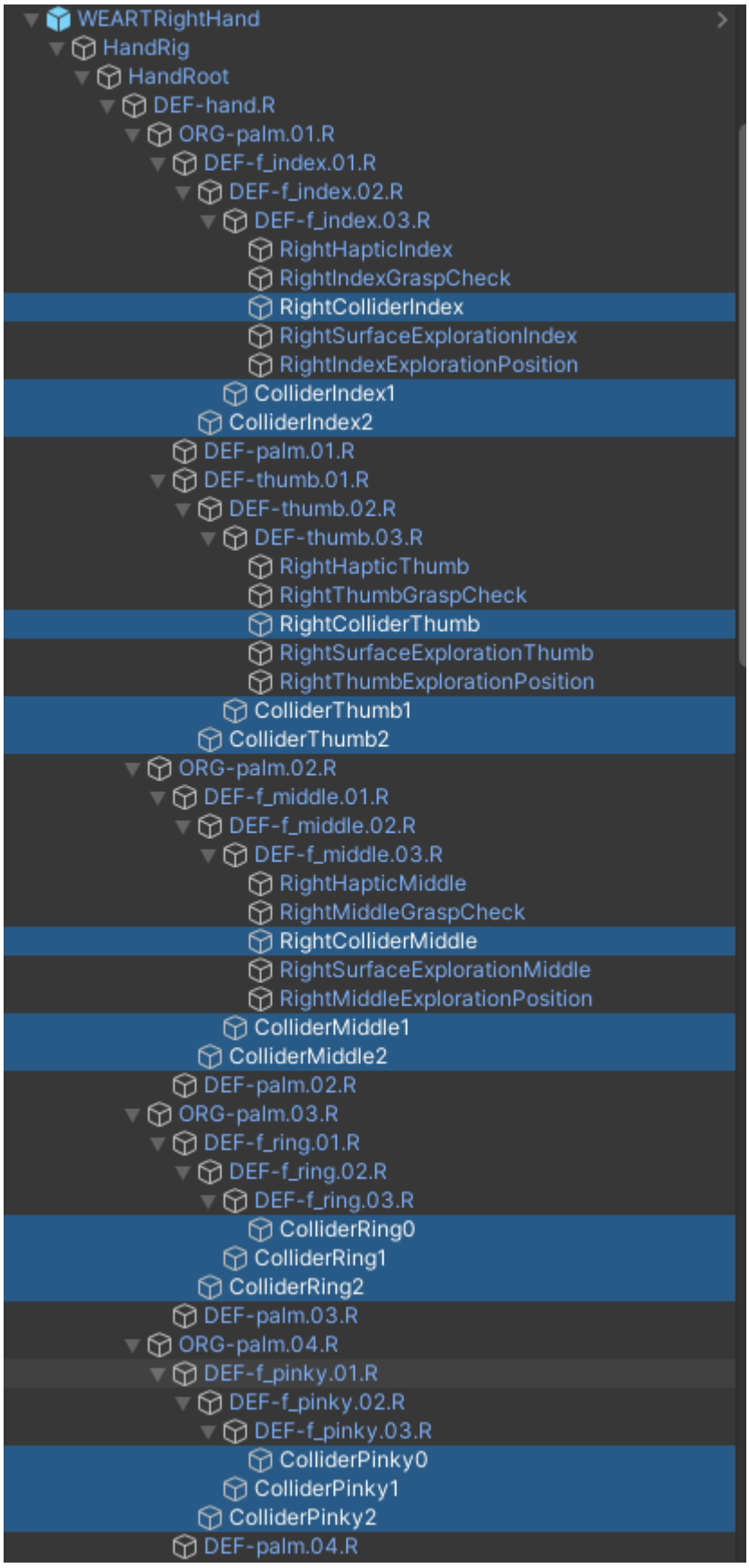
Surface Exploration

When a WeArtTouchableObject has the flag "Surface Exploration" enabled, it allows for a better touch experience on a flat surface.

If any two fingertips(thumb, index, middle, annular and pinky) or palm raycast in the direction of the palm and hit a touchable object with "Surface Exploration" flag enabled, the system will displace the haptic objects in order to enable a better exploration experience.

Remove Collisions Form The Hand Prefab / Pass Through Hands

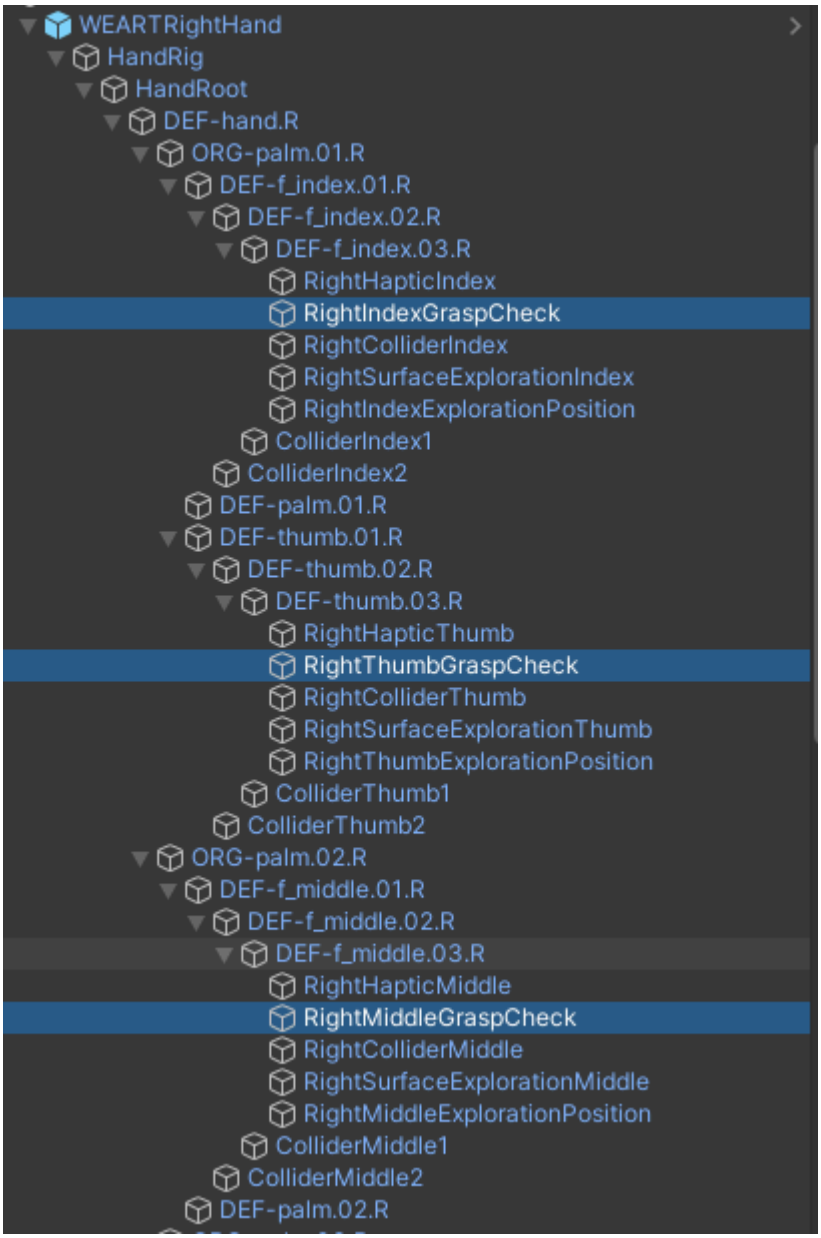
In order to remove the physics from the hand, select the game objects displayed in the next picture and disable their colliders.



Remember to also disable the colliders from the next image (they are present in the same hand).

Remove the Grasping Functionality

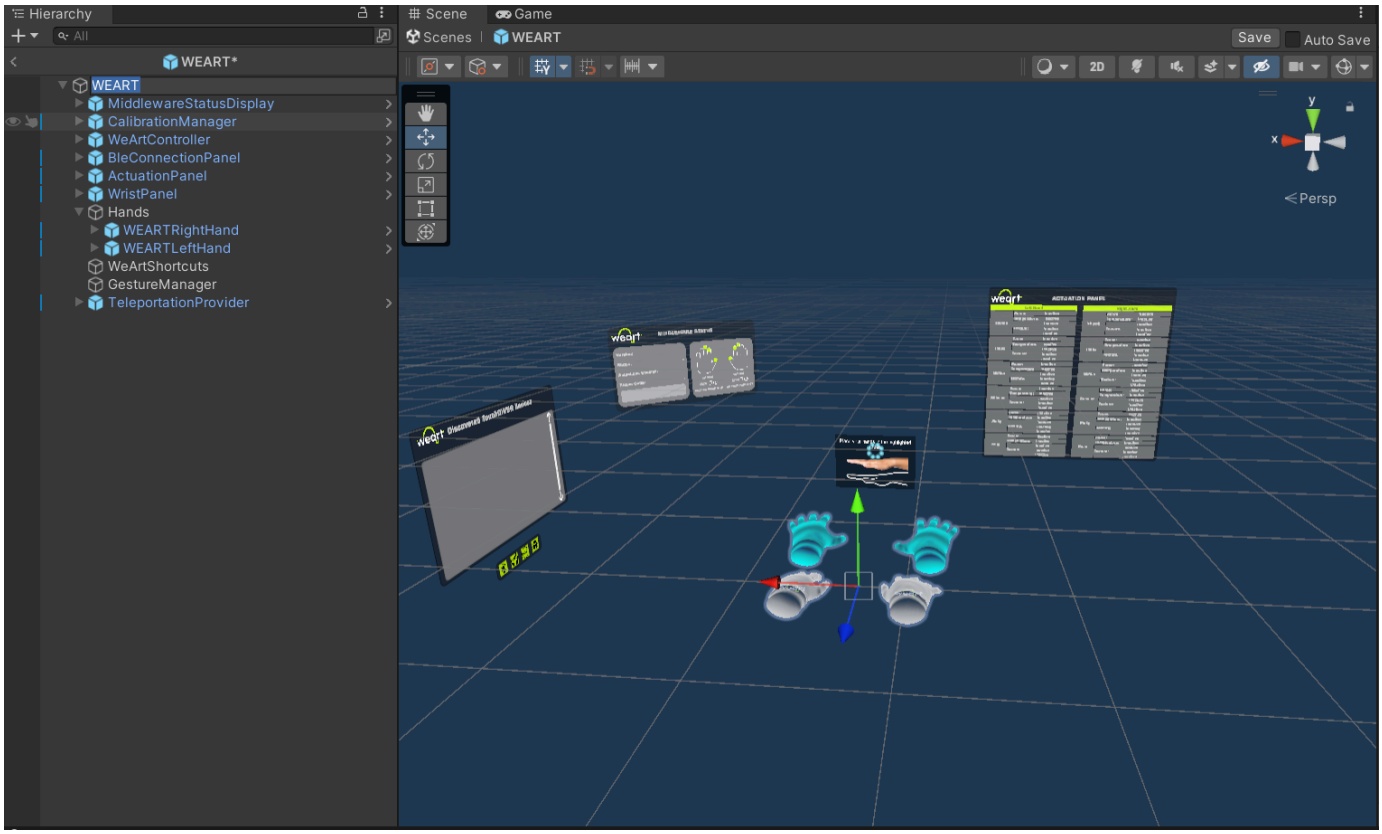
If you also want to remove the grabbing capabilities, disable the colliders from these game objects.



WEART Prefab

WEART prefab allows for a fast implementation of the system. It contains the MiddlewareStatusDisplay, CalibrationManager,WeArtController, BLEConnectionPanel, ActuationPanel, WristPanel and the hands prefabs.

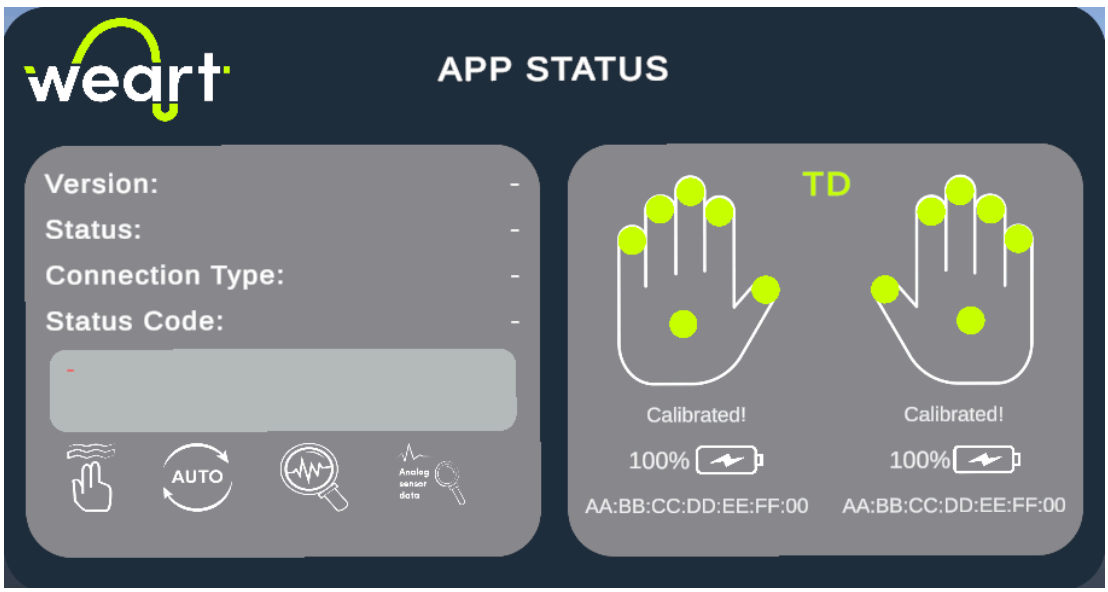
This prefab can be found in WEART SDK→ Runtime→ Prefabs → WEART



WeArt App Status Display

This tool will add the status display item prefab to the current scene.

The "Application Status Display" (shown below) is a 2D canvas containing all the status information received by the WeArt App.



In particular, the object will display:

- The current application version
- The current status of the application (as shown in the weart application UI)

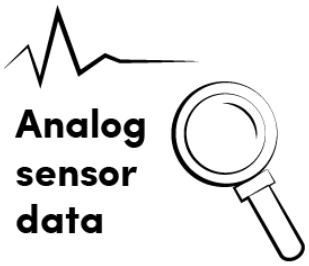
- The last status code received from the weart application and, if not ok, a description of the error
- The status of the connected TouchDIVERs
    - Mac Address
    - Battery level (and will show a bolt symbol if the device is charging)
    - Calibration status during the session
    - Status of each thimble (shown on the hand icon by colored dots: 3 for TD and 6 for TD Pro)

The icons indicate:

Actuations Enabled

Autoconnection Enabled

Raw Data Log Enabled



Sensor On Mask Enabled

Status Codes

The current status codes (along with their description) are:

Status Code		Description
0	OK	Ok
100	START_GENERIC_ERROR	Can't start generic error: Stopping
101	CONNECT_THIMBLE	Unable to start, connect at least one thimble and retry
102	WRONG_THIMBLES	Unable to start, connect the right thimbles matched to the bracelet and retry
103	BATTERY_TOO_LOW	Battery is too low, cannot start
104	RUNNING_DEVICE_CHARGING	Can't start while the devices are connected to the power supply
105	FIRMWARE_COMPATIBILITY	You have an incompatible firmware, finger tracking will not work properly!
106	SET_IMU_SAMPLE_RATE_ERROR	Error while setting IMU Sample Rate! Device Disconnected!
107	RUNNING_SENSOR_ON_MASK	Inconsistency on Analog Sensors raw data! Please try again or Restart your device/s!
108	BATTERY_REMOVED	Battery is removed, please insert a battery or device will shut down
109	BATTERY_LOW_WARNING	Battery low, please connect the device to a power supply
200	CONSECUTIVE_TRACKING_ERRORS	Too many consecutive running sensor errors, stopping session
201	DONGLE_DISCONNECT_RUNNING	BLE Dongle disconnected while running, stopping session
202	TD_DISCONNECT_RUNNING	TouchDIVER disconnected while running, stopping session
203	DONGLE_CONNECTION_ERROR	Error on Dongle during connection phase!
204	USB_CONNECTION_ERROR	Can not use the device through Bluetooth while connected to a USB port
300	STOP_GENERIC_ERROR	Generic error occurred while stopping session

Note
The description of each status code might change between different WeArt App versions, use the status code to check instead of the description.

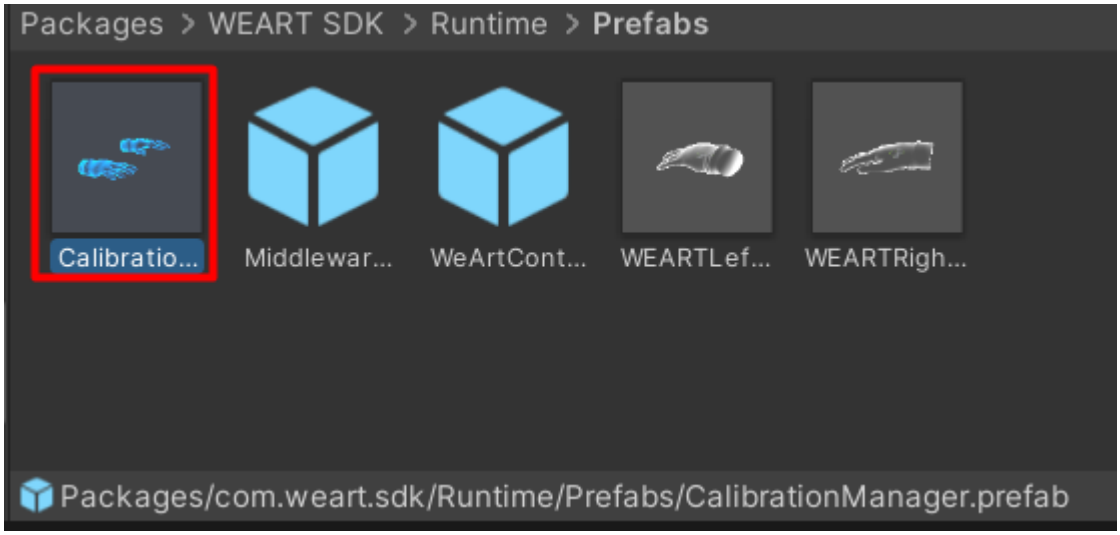
Calibration Manager

The calibration manager offers an easy and precise way of calibrating the touch diver at the start of the experience.

The calibration manager checks how many touch divers have been connected and changes the number of the hands required to calibrate and their hand sides automatically.

You need to put your hands in the position specified and it will signal to the WeArt App that the calibration process is starting. The CalibrationManager prefab needs to have in the same scene the WeArtController prefab and the two hands prefabs.

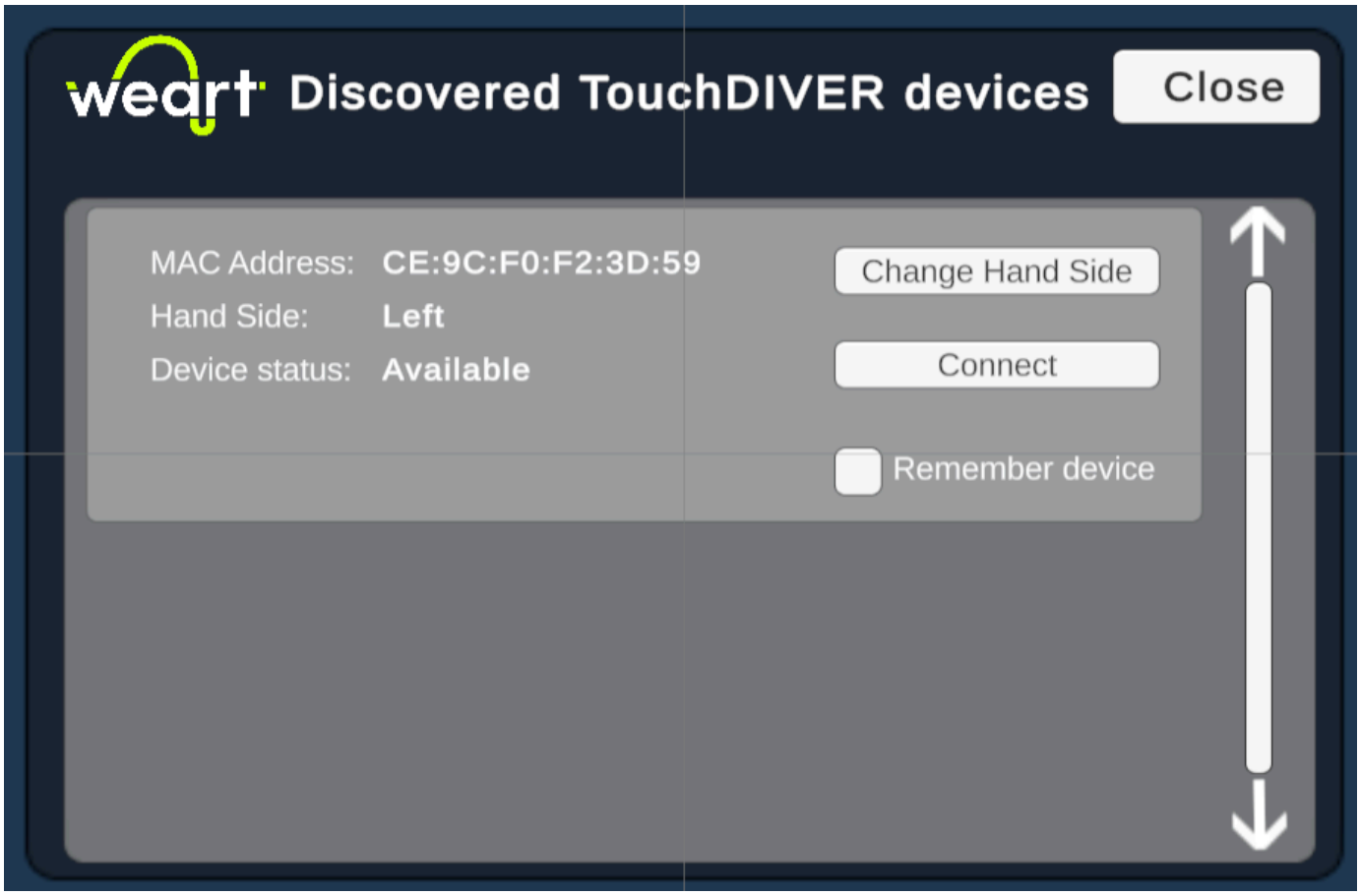
> **Note**
> If inside the WeArtController, the StartRawDataAutomatically is checked, the calibration will be forced. If you want to use Calibration Manager Prefab, disable StartRawDataAutomatically.



An example of this can be found in the PlaygroundDemo scene.

Ble Connection Panel

The "Weart TouchDIVER Connection Panel" is a 2D canvas provided all information about scanned Weart TouchDIVER devices with possibility to connect them, set the hand side and remember them for auto connection within next sessions.



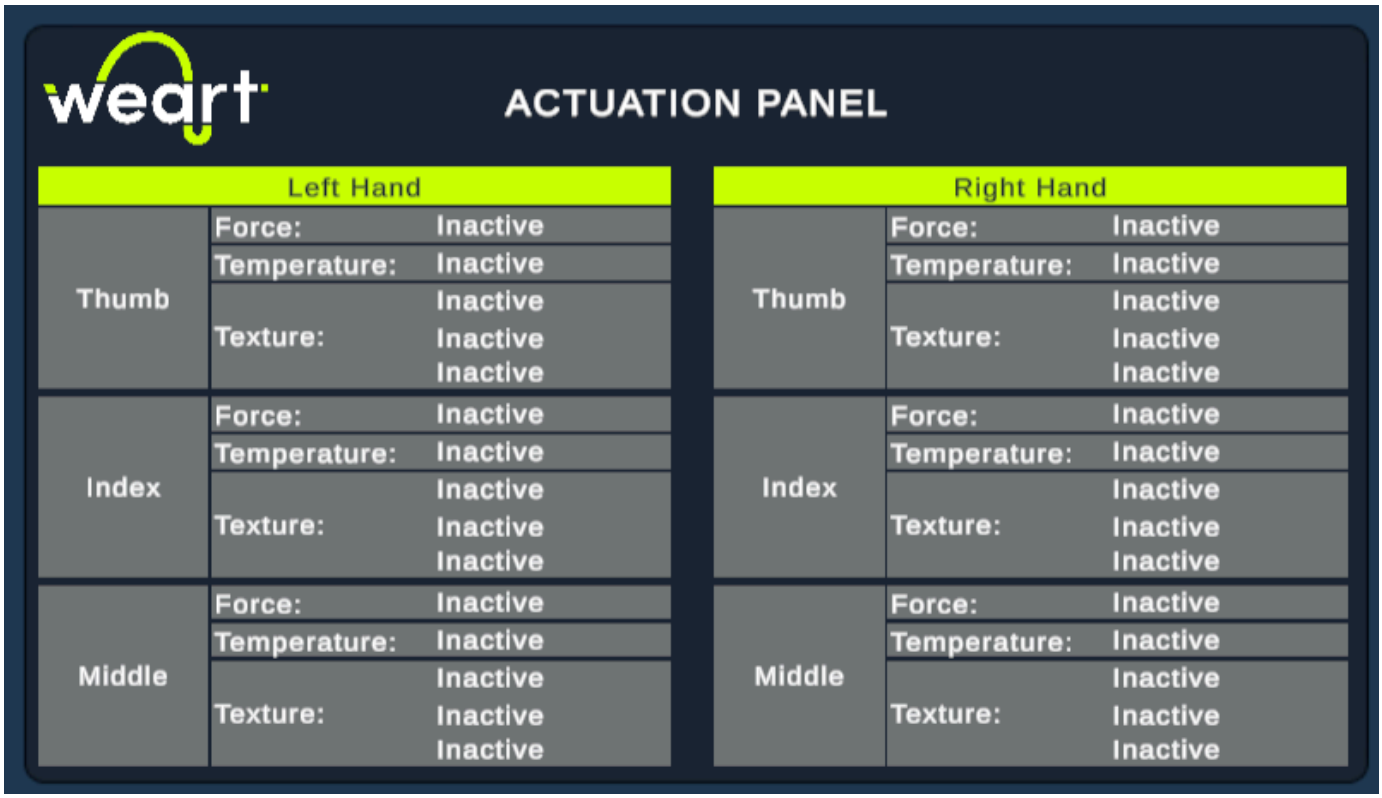
Panel contains:

- HandSide — the actual device hand side, Left or Right, with possibility to change it in accordance to your needs: just press the related button "Change Hand Side".
- Device Status — status of bluetooth connection.
- Connection Button — pressing of this button initiates the connection process, after successful connection, this button will be replaced by Disconnection Button.
- Remember device toggle — if this toggle is selected the TouchDIVER device will be saved and when you will found it again within your next session this device will be connected automatically.

All buttons and toggle at this panel are touchable, to press them just poke them by your virtual hand.

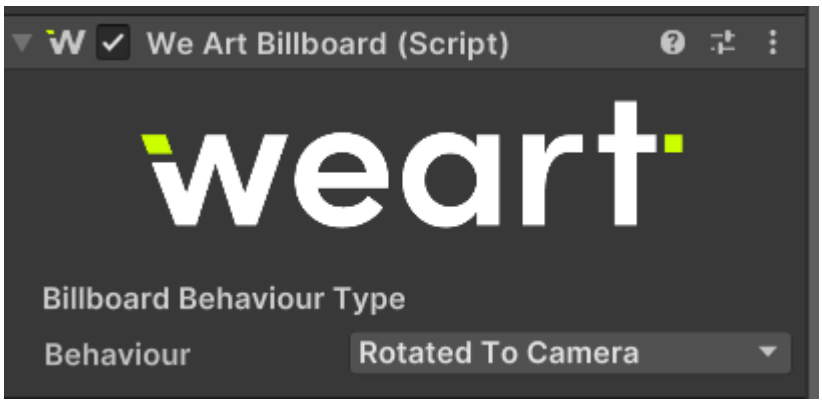
Actuation Panel

The "WEART Actuation Debug Panel" is a 2D canvas provided the runtime actuation data from haptic objects at WEARTHands: TouchDIVER: thumb, index, middle or TouchDIVER Pro: thumb, index, middle, annular, pinky and palm (Right hand and Left hand). It will show 3 haptic objects for TouchDiver and 6 haptic objects for TouchDIVER Pro.

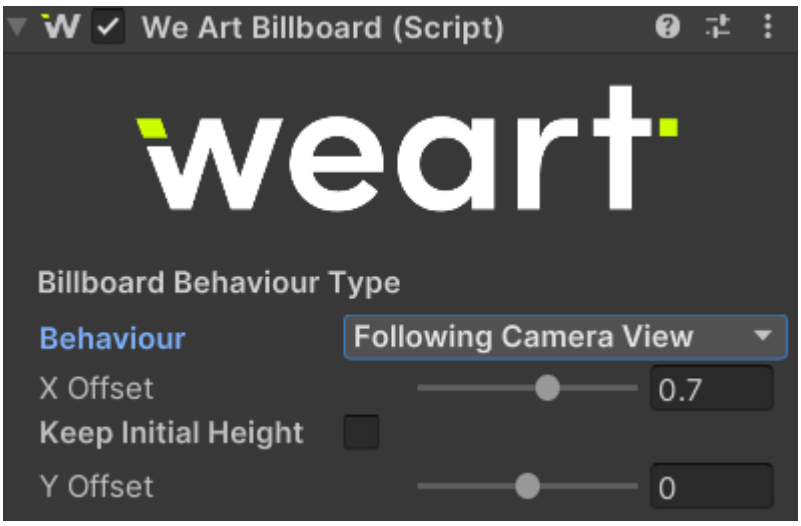


Billboard Component

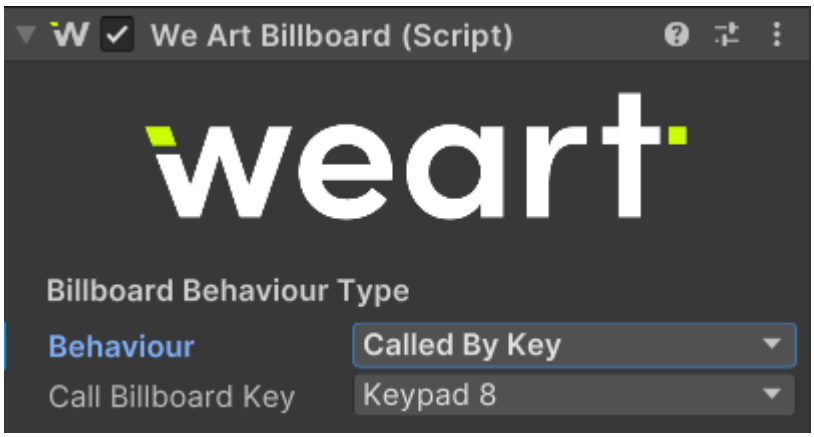
The "WEART Actuation Debug Panel" has the WeArt Billboard Component that provided a several behaviors:



- Rotated to Camera — the object will be always rotated to camera (player) with its front side.



- Followed Camera View — the object will follow the camera. The value X Offset sets how far it will be from camera. The flag Keep Initial Height will keep the Y position that object had at the start of scene. The value Y Offset sets Y position of the object related to the center of camera view.



- Called by Key — the object will positioned in from of camera view when the user will press the specific key at the keyboard. The key sets at the inspector.
- Static — does nothing.

Wrist Panel

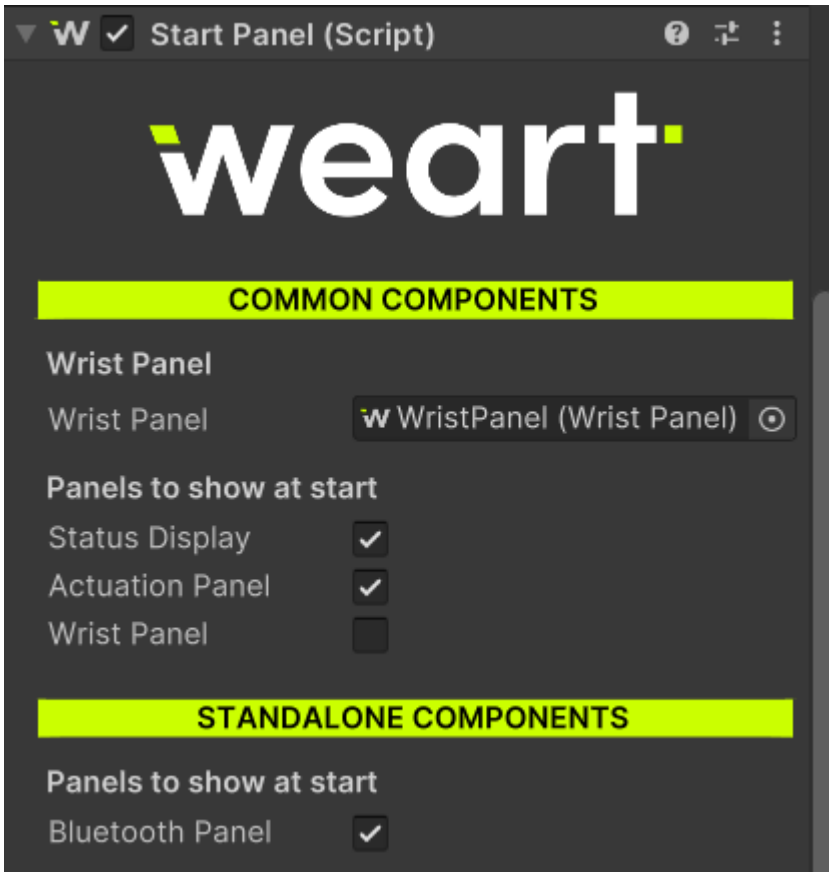
The "WEART Wrist Panel" is a part of TouchUI system with buttons used for:

- Show/Hide the WEART TouchDIVER Connection Panel.
- Show/Hide the WeArt App Status Panel.
- Restart the WEART Hands calibration process.
- Show/Hide the WEART Actuation Debug Panel.

This panel could be placed anywhere at the scene in accordance your project requirements.

> **Note**
> DO NOT DISABLE THE LISTED HERE WEART PANELS AT THE UNITY SCENE DIRECTLY!!!

If you would like to hide them at the start of your session(including the wrist panel itself), just deselect them at the StartPanel component at the WEART Wrist Panel:



Shortcuts

The WeArt shortcuts component provides the possibility to assign the key shortcuts for useful SDK services. At this moment there is implemented the restart of calibration.

Gesture System

The SDK includes a Gesture Recognition system. While it comes with predefined gestures, you can also create custom gestures by following these steps:

1. Define the Gesture Name

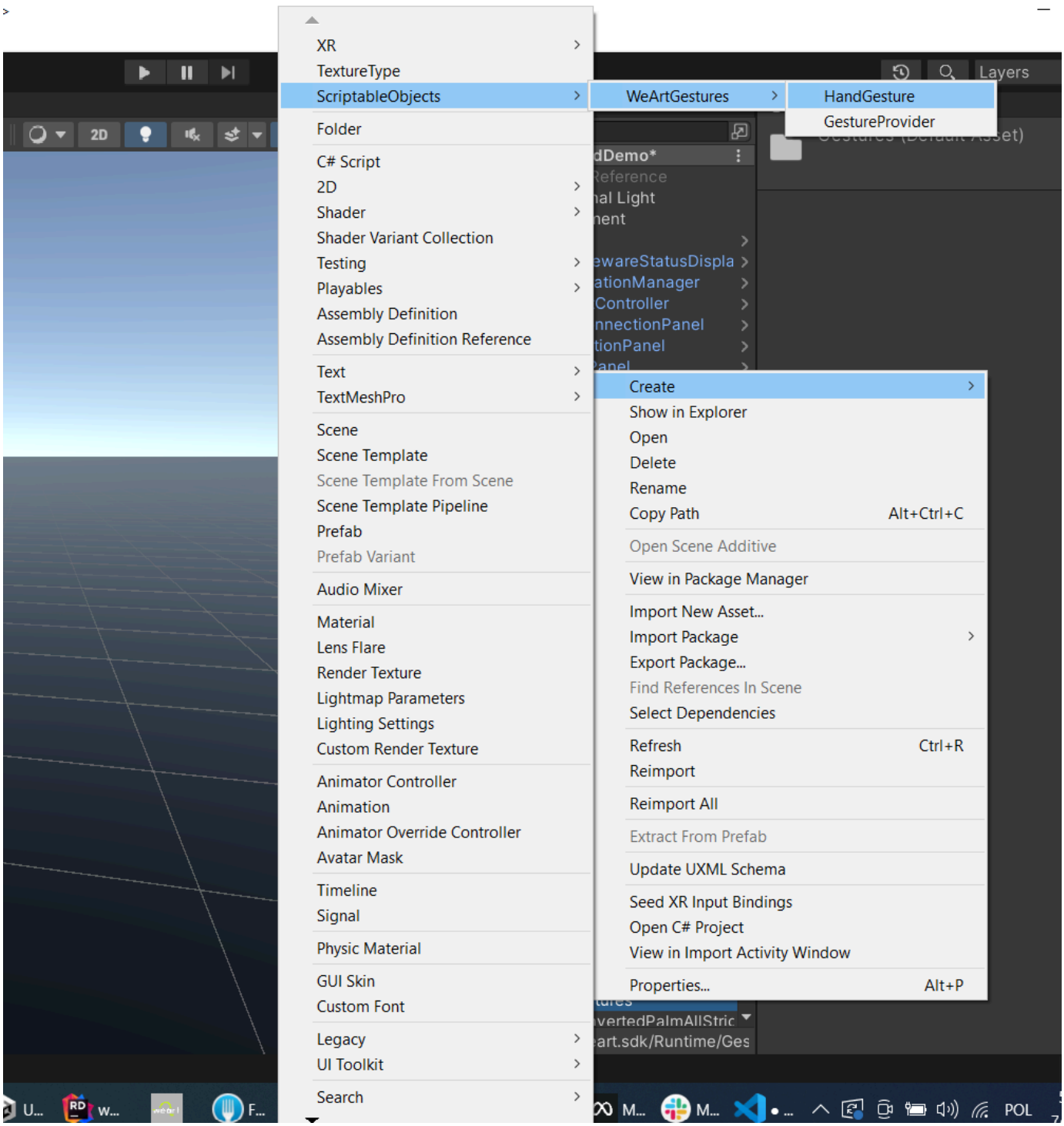
Choose a name for your new gesture. Locate the GestureName.cs script in the following path:

Packages/WEART SDK/Runtime/GestureInteractions/Core/Scripts/GestureName.cs

Add your new gesture name to this script.

2. Create the Gesture

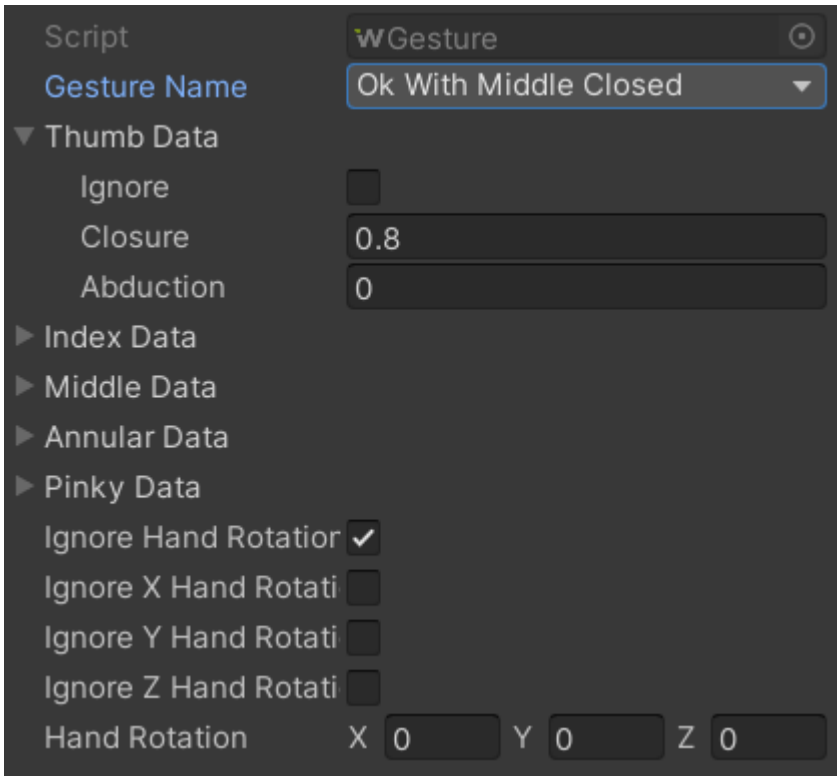
Right-click on your asset folder and navigate to: Create => ScriptableObjects => WeArtGestures => HandGesture



3. Configure Your Gesture

Gestures are defined based on finger closure and abduction data:

- A closure value of 0 indicates an open finger, while 1 indicates a closed finger.
- Optionally, the gesture can include hand rotation data.
- Each finger or rotation axis can be set to "ignore" as needed.



> **Note**
> Please, be sure that you new gesture has its new unique name.

4. Add Your Gesture to the GestureProvider

Locate the GestureProvider asset at:

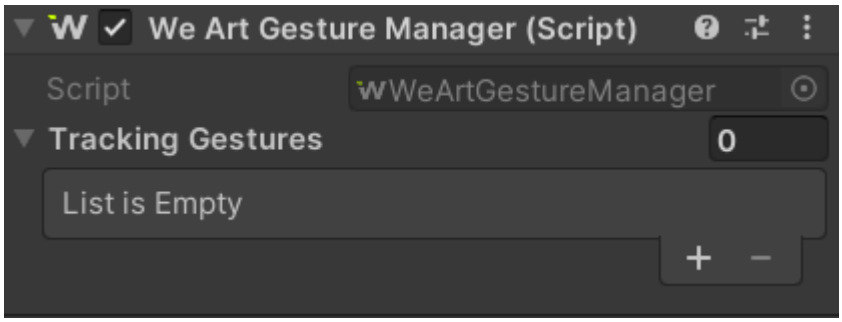
Packages/WEART SDK/Runtime/GestureInteractions/Core/Resources/GestureProvider.asset

Add your new gesture to this asset.

GestureManager

The WeArtGestureManager is part of the WEART prefab and is used to track gestures and subscribe methods to their recognition stages. Currently, it works exclusively with WeArtHandControllers.

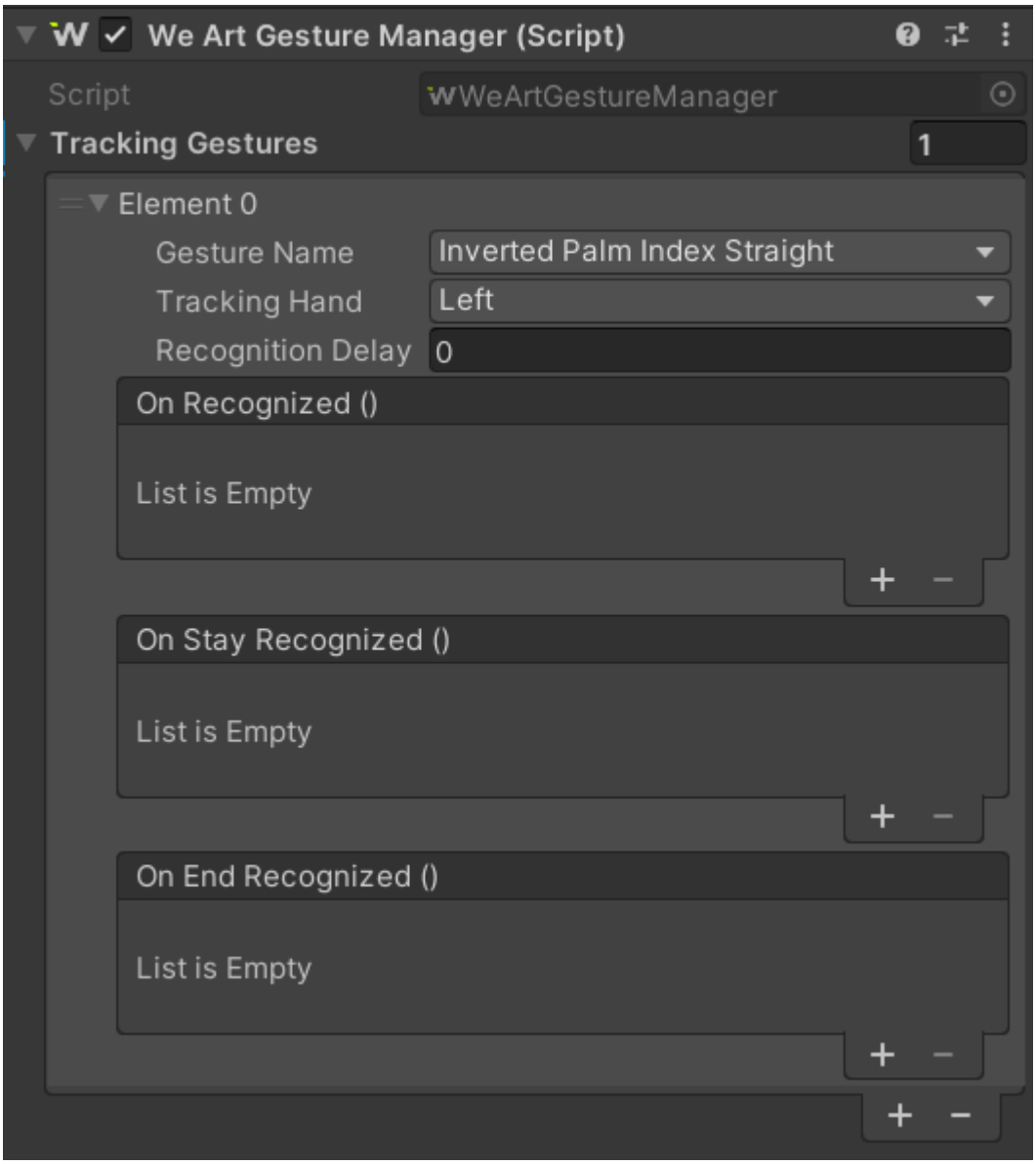
To add a new gesture for recognition, press “+” to expand the gesture list:



- Gesture Name — The specific gesture to track.
- Tracking Hand — Specifies which hand(s) to track: left, right, any, or both. Selecting "both" requires both hands to perform the same gesture simultaneously.
- Tracking Delay — The time delay (in seconds) that the hand(s) must hold the gesture before it begins to be recognized.

You can subscribe to the following events:

- OnRecognized — Triggered after the gesture recognition delay is met.
- OnStayRecognized — Triggered every frame while the gesture remains recognized.
- OnEndRecognized — Triggered when the recognized gesture is no longer detected.

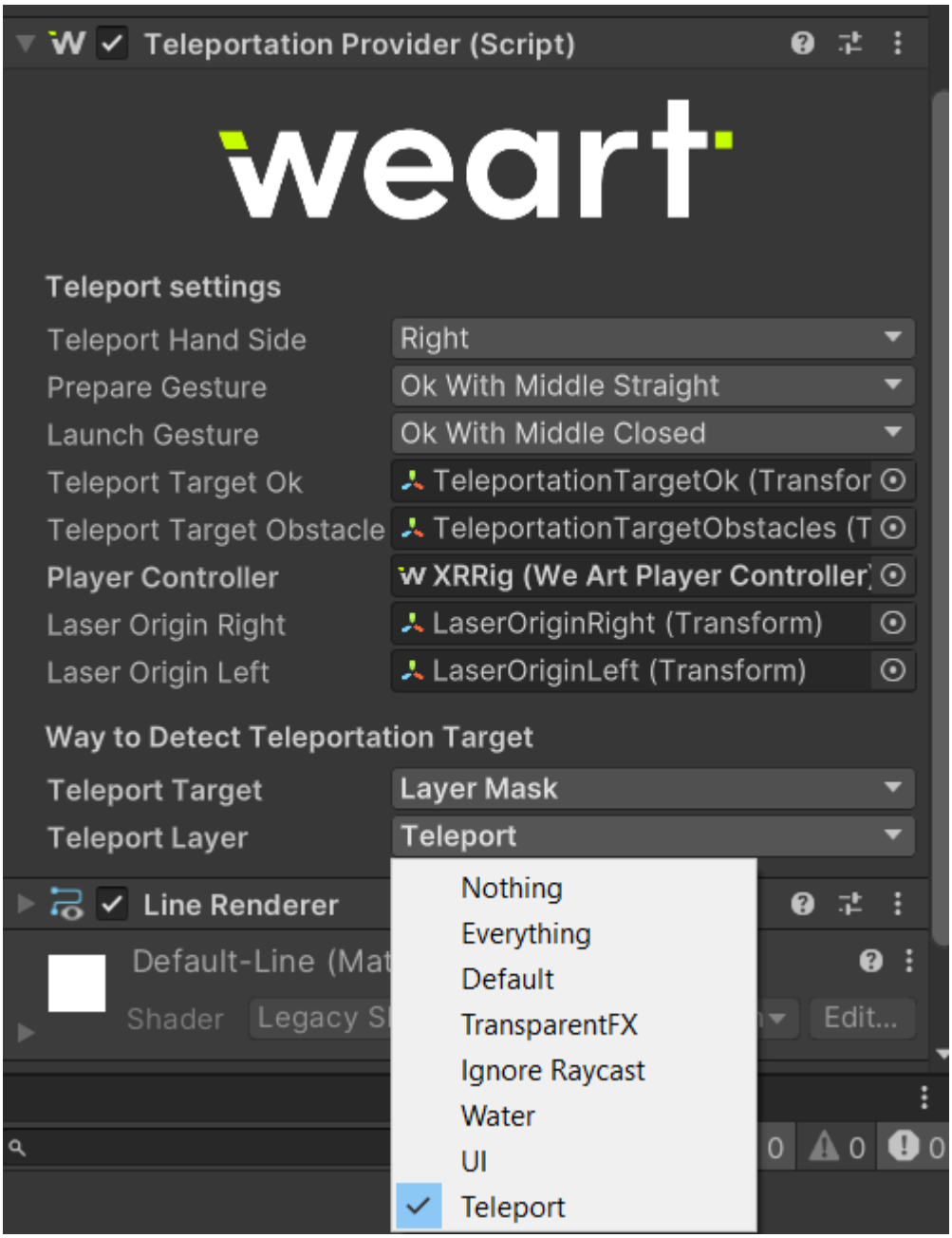


Teleportation Provider

Teleportation Provider — provides the teleportation feature based on gesture recognition. Currently it works only wih WeArtHandControllers. The prepare gestures invokes the laser that checks where it is possible to teleport. At this moment you can the target teleportation by its name. The green laser and green teleport target indicate that teleport is possible. The red laser — that teleport is not possible.

You may set your own teleportation target for possible places and for places with obstacles.

To be able to teleport, please, put here reference of your XR Rig with WeArtPlayerController component (or your own implementation).



API

> **Note**
> A WeArtHapticObject that is affected by WeArtHandController(part of the physical hand prefab) will not respond properly to API changes to the effects. It will be managed by the WeArtHandController instead.

Start/Stop Client

Manage Start and Stop WeArt App communication Import in your C# class this namespace:

```
using WeArt.Components;
```

From WeArtController GameObject get Instance

```
WeArtController.Instance.Client.Start(); //Start WeArt App communication
WeArtController.Instance.Client.Stop(); //Stop WeArt App communication
```

Start/Stop Calibration and events

From WeArtController GameObject get Instance

```
WeArtController.Instance.StartCalibration();
WeArtController.Instance.StopCalibration();
```

Start/Stop Raw Data from WeArt App

To start and stop the WeArt App from sending raw sensors data, use the WeArtController GameObject instance:

```
WeArtController.Instance.StartRawData();
WeArtController.Instance.StopRawData();
```

Tracking

After starting the WeArt App and performing the device calibration, it's possible to receive tracking data related to the TouchDIVER thimbles.

There are two hand prefabs in the scene. Every hand prefab has a WeArtHandController. Every WeArtHandController has three WeArtThimbleTrackingObject references. We can get the closure and abduction from these components. The values are from 0 to 1, 0 representing no closure or abduction and the 1 representing the maximum value.

Getting Closure and Abduction:

```
WeArtThimbleTrackingObject _thumbThimbleTracking;
float closure = _thumbThimbleTracking.Closure.Value;
float abduction = _thumbThimbleTracking.Abduction.Value;
```

WeArtTouchEffect

The SDK contains a basic WeArtTouchEffect class to apply effects to the haptic device. The TouchEffect class contains the effects without any processing. For different use cases (e.g. values not

directly set, but computed from other parameters), create a different effect class by implementing the WeArtEffect interface.

Create Custom effect

WeArtHandController takes care of the realistic transitions of effects in the scene, custom effects(creating, updating, adding and removing) are not compatible with the hand prefabs. Haptic objects can be used outside of it as long and their field called IsPhysical is disabled.

Instantiate new effect:

```
WeArtTouchEffect effect = new WeArtTouchEffect();
```

Update effect

Create and activate actuations

```
Temperature temperature = Temperature.Default;
temperature.Active = true; //you have to active
temperature.Value = 0.6f; //hot

Force force = Force.Default;
force.Active = true;
force.Value = 0.8f;

WeArt.Core.Texture texture = WeArt.Core.Texture.Default;
texture.Active = true;
texture.ForcedVelocity =false;
texture.TextureType = TextureType.CrushedRock;
texture.Volume = 100f;
```

Set actuation to effect:

```
effect.Set(Temperature, Force, Texture, new WeArtTouchEffect.WeArtImpactInfo());

// In case of dynamic force on haptic object that uses a distance target
effect.Set(Temperature, hapticObject.CalculatePhysicalForce(touchableObject), Texture,
```

Texture Velocity

We can change the velocity of the texture of an effect by changing its value [0.0 to 0.1] 0.5 is recommended for the normal speed of the texture. This will give the effect a constant speed for the

texture without moving the hand

```
Texture texture = Texture.Default;
texture.TextureType = (WeArt.Core.TextureType)11;
texture.Active = true;
texture.Velocity = 0.5f;

touchEffect.Set(temperature, force, texture, new WeArtTouchEffect.WeArtImpactInfo());
```

Add effect

Apply to your HapticObject (finger/thimble) effect:

```
// Add effect and subscribe it to a touchable object
hapticObject.AddEffect(effect, touchableObject);
// Add effect with no reference
hapticObject.AddEffect(effect);
```

Remove effect

To remove effect and restore actuation, get the same instance of effect and call “Remove” for the same HapticObject:

```
hapticObject.RemoveEffect();
```

Change Effect of WeArtTouchableObject

We can change directly the effect of a WeArtTouchableObject by using the public methods. In this way, we don't need to provide the other parameters to the effect.

```
// Set Temperature
touchableObject.SetTemperature(float value);

// Set Force
touchableObject.SetHapticForce(0.6f);

// Set Texture
touchableObject.SetTextureType(TextureType.Aluminium);

// Set Texture Velocity
touchableObject.SetTextureVelocity(0.5f);

// Set Texture Volume
```

```
touchableObject.SetTextureVolume(90);
```

Adding new custom texture type

In WeArtCommon.cs we can find enum called TextureType We can add a new texture like this:

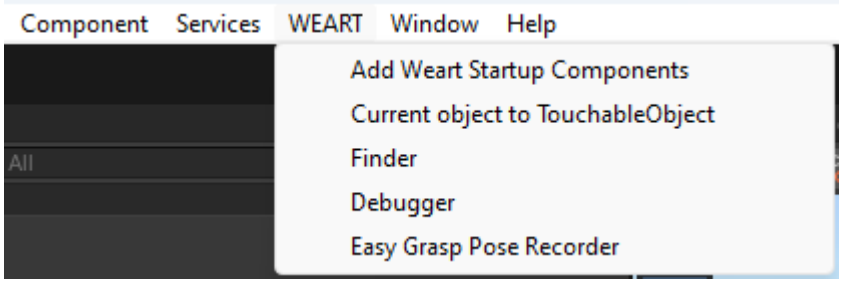
```
public enum TextureType : int
{
    Click = 0,
    SoftClick = 1,
    DoubleClick = 2,
    FineAluminiumSlow = 3,
    FineAluminumFast = 4,
    PlasticSlow = 5,
    ProfiledAluminiumMedium = 6,
    ProfiledAluminiumFast = 7,
    RhombAluminiumMedium = 8,
    TextileMedium = 9,
    CrushedRock = 10,
    Granite = 11,
    Wood = 12,
    Laminate = 13,
    ProfiledRubber = 14,
    VelcroHooks = 15,
    VelcroLoops = 16,
    PlasticFoil = 17,
    Leather = 18,
    Cotton = 19,
    Aluminium = 20,
    DoubleSidedTape = 21
}
```

And we can access it by index:

```
Texture texture = Texture.Default;
texture.TextureType = (WeArt.Core.TextureType)50;
texture.Active = true;
```

Tools

You can find the tools at the top of the editor.



- Add WEART Startup Components - Spawn on your scene the main WEART components ready to use
- Current objects to TouchableObject - Selecting a GameObject on your scene you could make it touchable adding any WEART component automatically
- Finder - Will show you any GameObject with the WeArtTouchableObject attached
- Debugger - Show actuations for any WeArtHapticObject in the scene
- Easy Grasp Pose Recorder - Records snap grasping pose while grasping a touchable object

Add WEART Startup Components

This tool will add to the scene the Weart WeArt App Status Display, CalibrationManager, WeArtController, BleConnectionPanel, ActuationPanel,WristPanel and the hands prefabs. The hand prefabs will not be connected to any VR headset controllers in the scene.

Current object to TouchableObject

This tool will transform the selected game object into a touchable object and it will apply the necessary components.

Finder

This tool will open a window that will show the touchable objects present in the scene

Debugger

This tool will open a window that will show the haptic objects in the scene and its parameters

Easy Grasp Pose Recorder

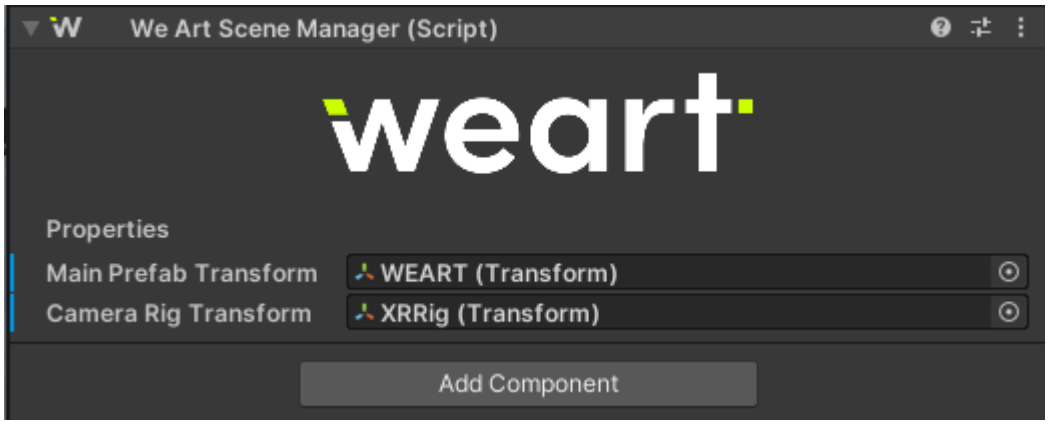
This tool will open a window that will allow the user to record a pose for the snap pose system, while grasping an object

Scene Changing System

Components

WeArtSceneManager

The goal of this prefab is to be placed in the main scene with the WeArt prefab(a prefab that contains all the weart components such as WeArtController, calibration and the hands) and the VR camera rig(contains the camera and the references for the controllers) The script attached to the prefab has two fields: one for the main prefab(which will be in the normal case, the weart prefab) and the second field which takes a reference to the VR camera rig transform.



These two field are empty by default, and the user will have to pace them in the corresponding fields. WeArtSceneManager Prefab was created as a optional component. It does not appear in the scene that is imported with the SDK. It is a drag and drop solution for changing or reloading a scene that contains the WeArt components and camera rig.

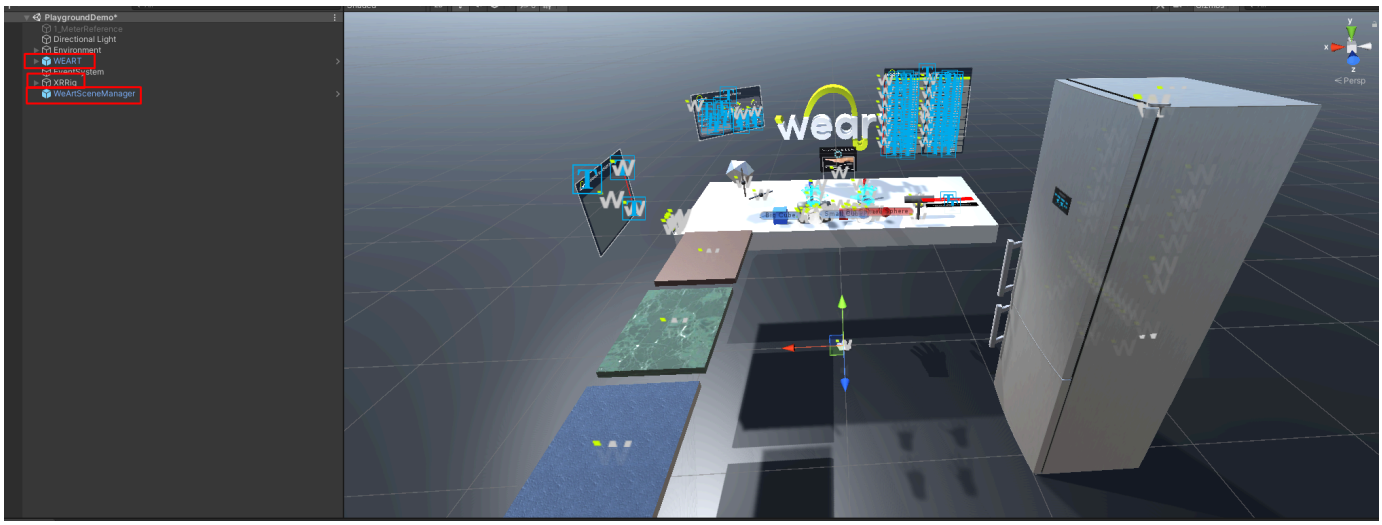
We can also leave one of the fields empty, in case we are have in an experience that does not require VR.

MainPrefabTransform is recommended to be filled in, if we are using WeArtSceneManager. In the case that the developer wants to create their own scene with haptic objects and touchable objects, they will need to have everything under a main parent, so we can have the reference to put in the field MainPrefabTransform, so that WeArtSceneManager, will know to make it persistent between scenes.

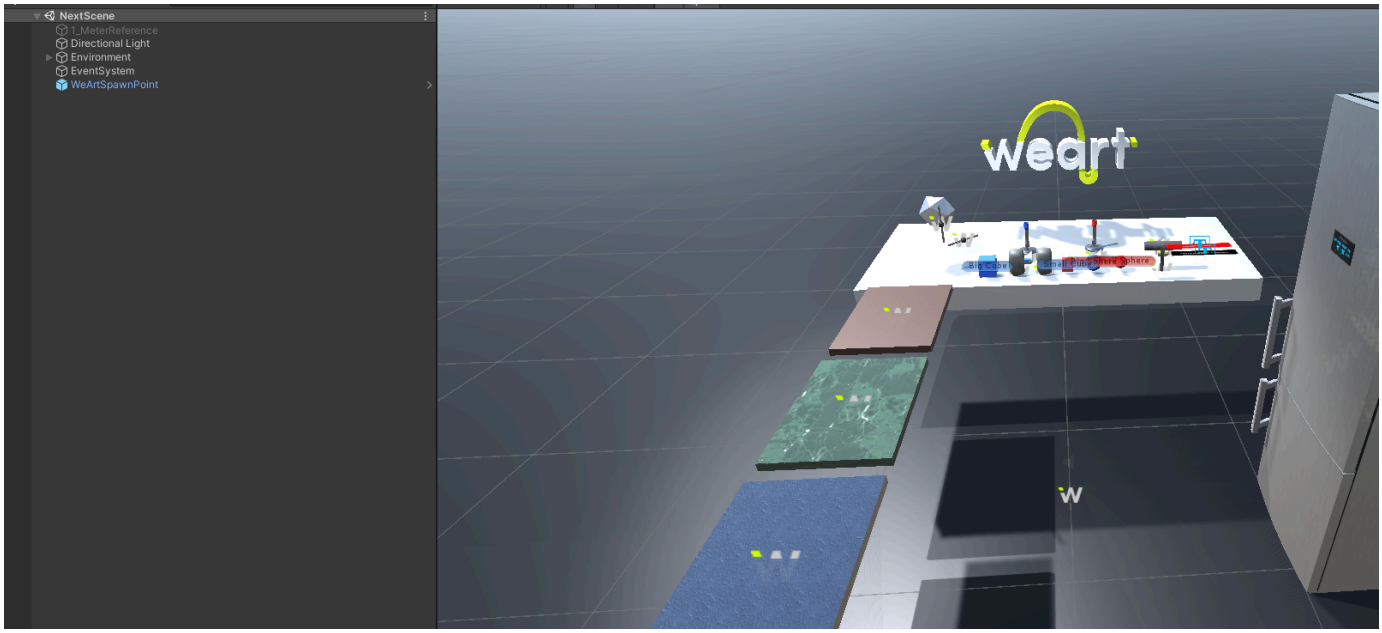
Both the transforms that will be placed into MainPrefabTransform and CameraRigTransform will become persistent by internally using the Unity standard method DontDestroyOnLoad() which will make them persistent in the case we call Unity's standard method for changing scenes called LoadScene(). WeArtSceneManager was not created to work with additive scenes, it was created to work with direct loading of new scenes or the same scene. But there can be build functionality around it, or the code can be easily changed.

WeArtSceneManager is meant to be placed into the scene that has the WeArt prefab and the vr camera rig. This is important for the setup. once this scene starts and makes persistent the weart

prefab, VR camera rig and WeArtSceneManager itself, this scene will act as a setup scene. This scene must not be loaded again. The scenes that must be loaded, should not contain haptic objects. These scenes should be only environments. For example this would be a Setup Scene(that we never go back to).



And this a scene that we can go to, or even reload.



WeArtSpawnPoint

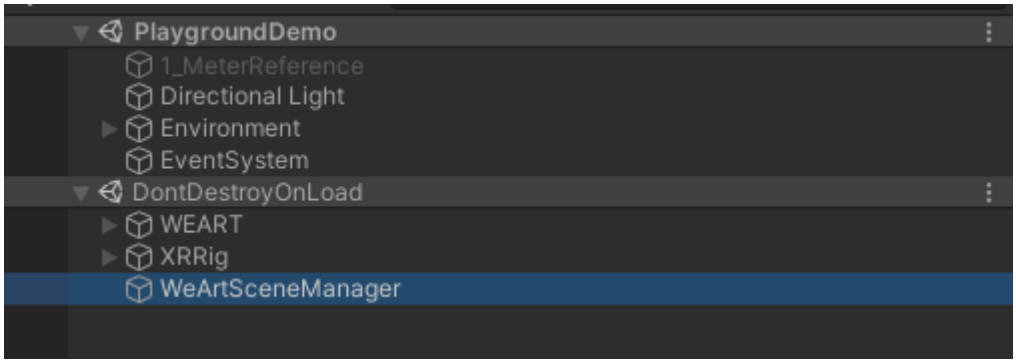
WeArtSpawnPoint is a script attached to the prefab with the same name. It's meant to be placed in a scene that will be loaded(not the setup scene). This prefab is also meant to drag and drop in the "next" scene and placed in the corresponding location and rotation .When this scene will be loaded, the VR camera rig reference from the WeArtSceneManager, will teleport and mimic the rotation of this WeArtSpawnPoint game object. This is optional in the case that CameraRigTransform form WeArtSceneManager gets left empty, then it will not run.

Functionality

WeArtSceneManager uses Unity's standard SceneManager library. The script wraps its functionality.

When the setup scene first starts, the script will set itself as a singleton(instance). From its fields in the inspector it will take MainPrefabTransform and CameraRigTransform and make them persistent by adding them to "DontDestroyOnLoad" which makes the game objects persistent even if the scene is changed. The game object of WeArtSceneManager will also make itself persistent. In total, the script is responsible for making 3 game objects persistent.

This is how the scene looks when it starts and it contains "WeArtSceneManager" prefab with the 2 fields filled, the three game objects become persistent.



The script has a public Singleton called Instance that can be accessed by other scripts. It return the unique instance of WeArtSceneManager.

The script has 2 events that can be called.

```
/// <summary>
/// Event fired after the pre requiremts are finished and before the scene starts to load
/// </summary>
public SceneDelegate OnSceneLoadStart;

// TODO Unity event

/// <summary>
/// Event fired after post requirements are finished and scene is loaded
/// </summary>
public SceneDelegate OnSceneLoadFinish;
```

LoadScene public method

This method takes the name of the scene you want to go to, optionally you can tell it if in the next scene you want to use the WeArt SDK, it is considered by default that we want to use it in the next scene. There is no additive scene in this setup. It deletes the old scene and loads the new scene while keeping the persistent game objects in "DontDestroyOnLoad".

```
public void LoadScene(string sceneName, bool nextSceneHasWeArtSDK = true)
```

```
{
  // Handled by the SDK
}
```

It makes sure that the hands release any kind of object (normal or anchored) and disables the hands and then makes sure that the hands are teleported directly to the next scene without interpolation to the real VR controller position. This is done to teleport the hands and not have them stuck in objects.

It checks to see if the new scene has a prefab called WeArtSpawnPoint, and if it does, it will teleport the camera rig to that position and rotation of the WeArtSpawnPoint. Then based of the optional “nextSceneHasWeArtSDK”(true by default) that the user entered in LoadScene.

Changing the scene

> **Note**
> Make sure to have the scenes listed in the build settings' "Scenes In Build". Or Unity will not allow the scenes to be changed.

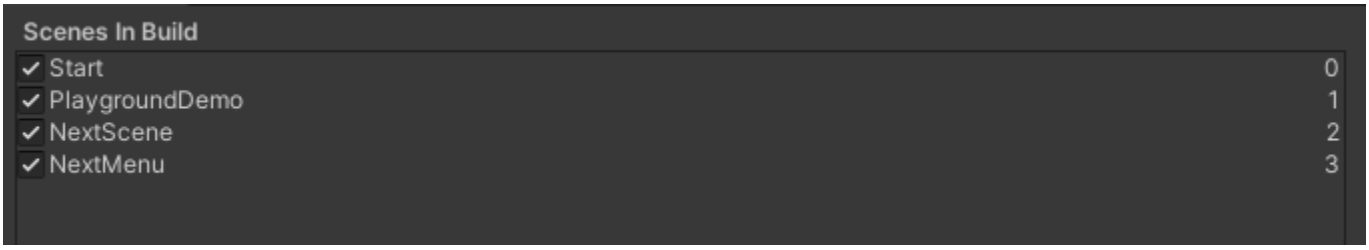
In a newly created script, we can simply call the load method like this:

```
if (WeArtSceneManager.Instance != null)
{
    WeArtSceneManager.Instance.LoadScene("NextScene");
}
```

“NextScene” is a scene that only contains the demo’s environment without the WeArt prefab or camera rig. In this scene we can grab objects and interact with the environment. If we call the method above, again, WeArtSceneManager will make sure that the hands will release anything it holds, loads the scene again (Reloading in this case) and teleport the camera rig to the WeArtSpawnPoint. And teleports the hands without interpolation to this new position, avoiding getting the hands stuck in walls or objects.

Loading scene without WeArt SDK

These are the scenes used to explain the feature



Start scene has nothing in it but a button that takes us to “PlaygroundDemo” using Unity’s default SceneManager.

Playground demo is the scene where we make the calibration. This scene also contains the WeArtSceneManager. We will never come back to this scene, as it is only a setup scene.

Then we can use the WeARtSceneManager to go to “NextScene”

```
if (WeArtSceneManager.Instance != null)
{
    WeArtSceneManager.Instance.LoadScene("NextScene");
}
```

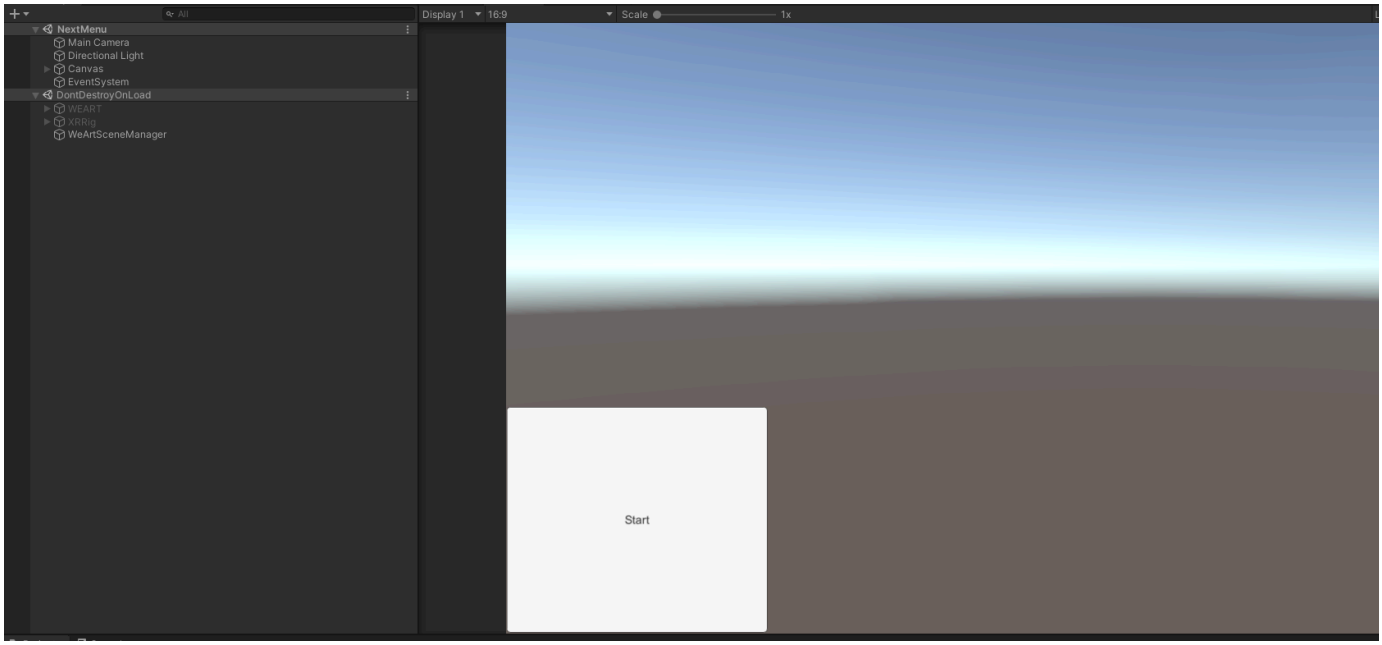
“NextScene” is a scene that only contains the demo’s environment without the WeArt prefab or camera rig. In this scene we can grab objects and interact with the environment. If we call the method above, again, WeArtSceneManager will make sure that the hands will release anything it holds, loads the scene again (Reloading in this case) and teleport the camera rig to the WeArtSpawnPoint. And teleports the hands without interpolation to this new position, avoiding getting the hands stuck in walls or objects.

“NextMenu” is a scene that only has a button to go to “NextScene” it was created to demonstrate what would happen if we wanted to go to an empty scene while we had the WeArt hands calibrated and ready to interact.

This is the code that loads “NextMenu”. The false is there to tell the WeArtSceneManager that we want to go to a scene where we want the WeArt prefab and camera rig to be disabled.

```
if (WeArtSceneManager.Instance != null)
{
    WeArtSceneManager.Instance.LoadScene("NextMenu",false);
}
```

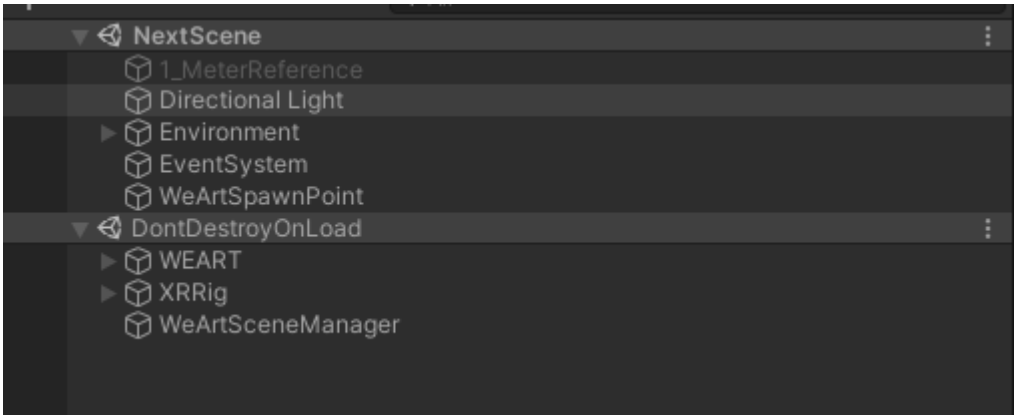
This is the result of this method:



From this scene we can call again:

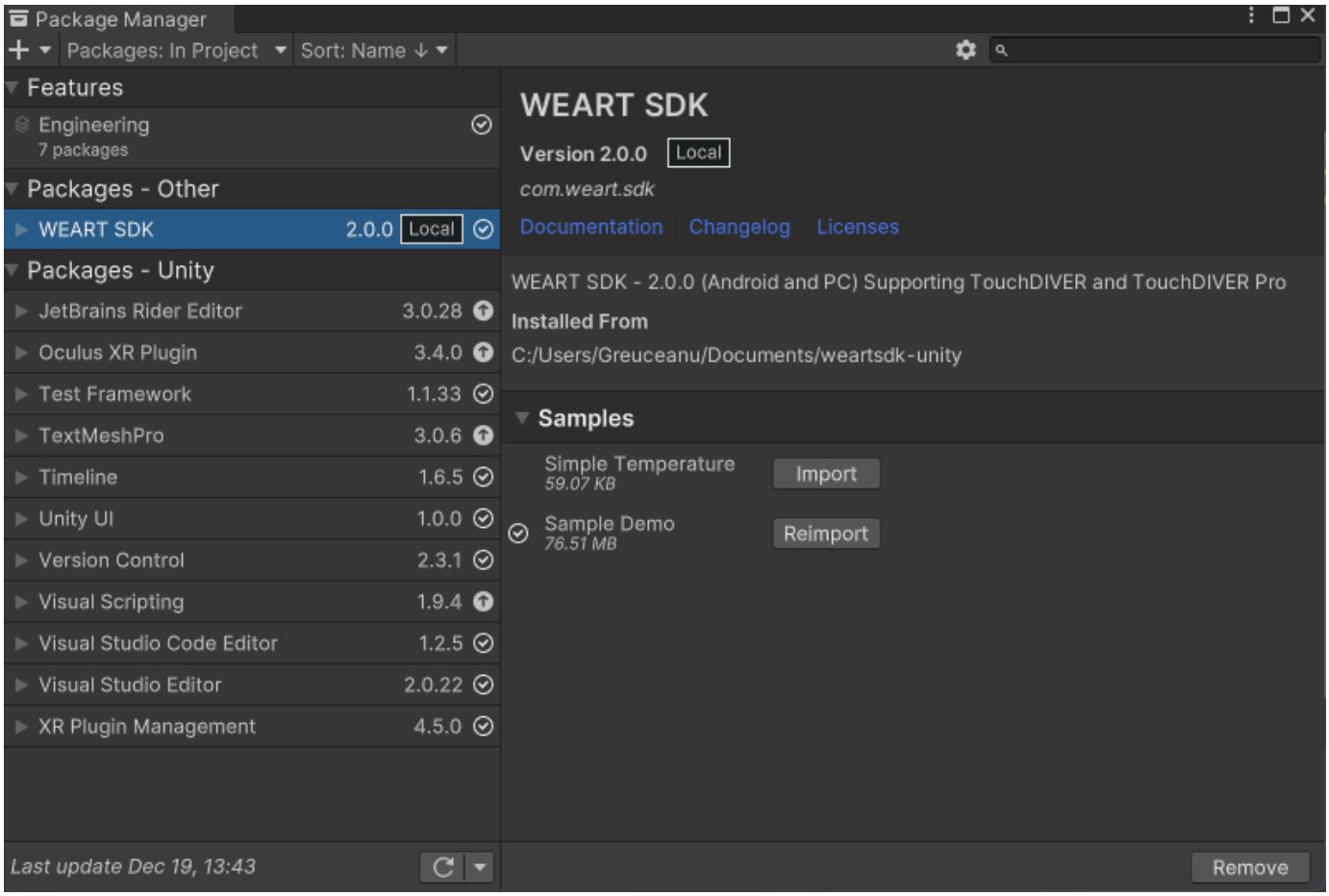
```
if (WeArtSceneManager.Instance != null)
{
    WeArtSceneManager.Instance.LoadScene("NextScene");
}
```

And have the hands back and ready to interact with the environment:



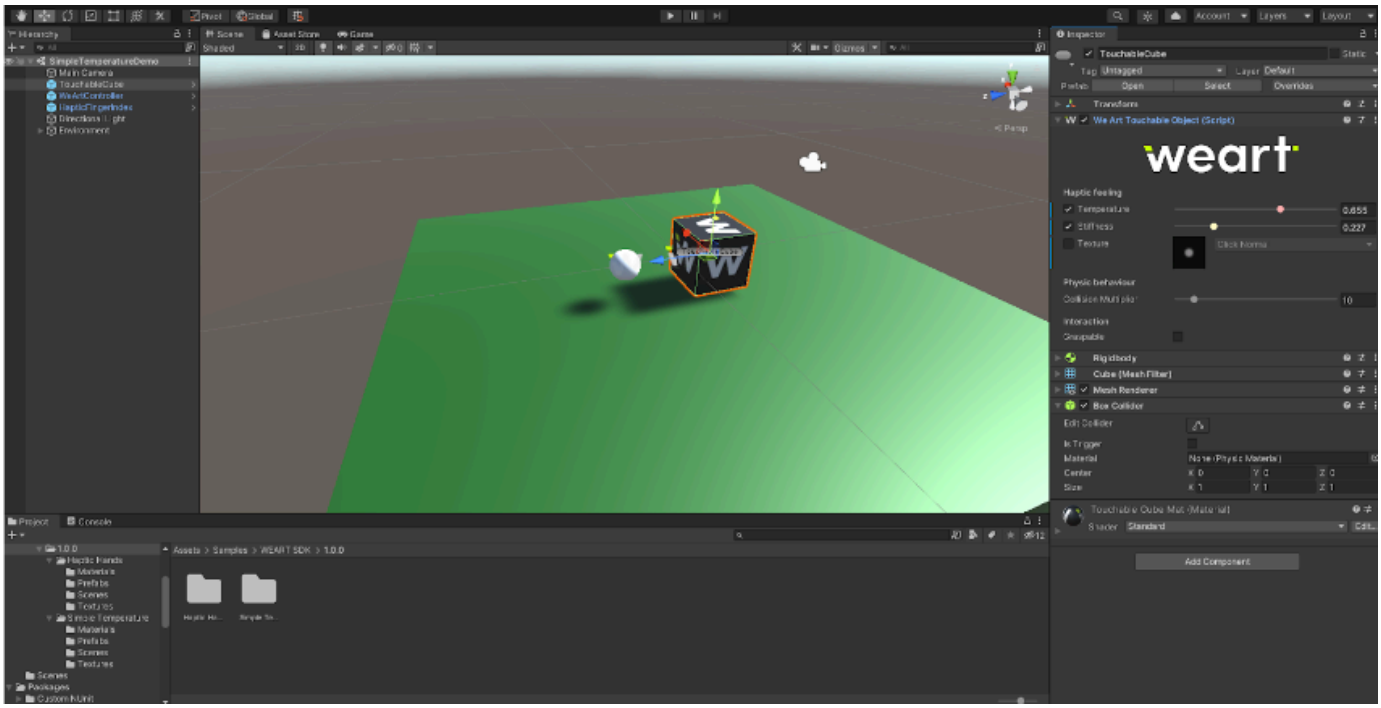
Example

You could import a sample scene to understand SDK components and simplify your test using ready scenes with any components you need.



Simple Temperature

This sample contains a very basic scene with one Touchable Cube and one finger Haptic Object for testing actuation

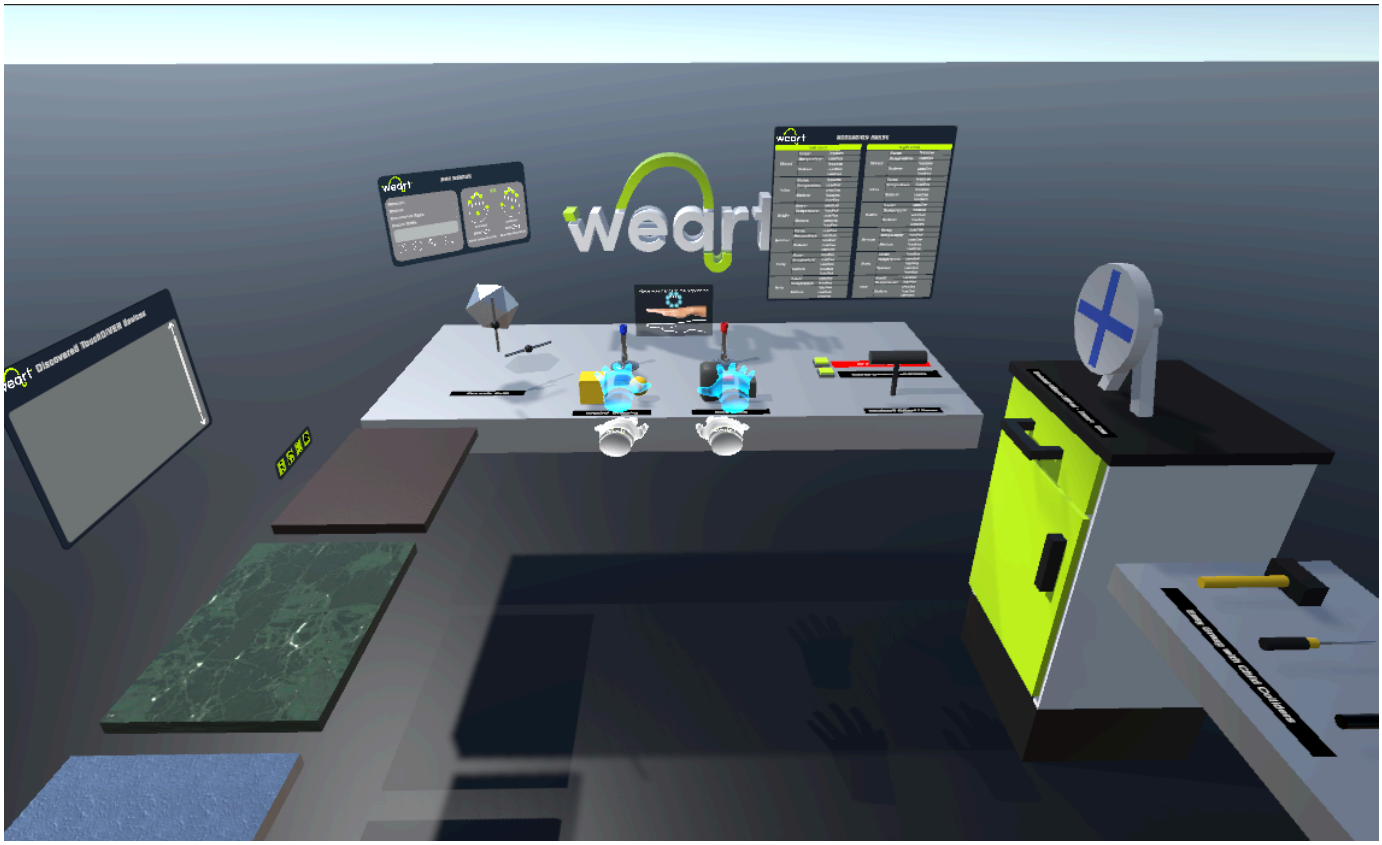


Sample Demo

This sample contains a ready-to-use scene which shows the capabilities of the sdk. It has three scenes:

PlaygroundDemo

This scene contains four material plates that have different textures, four objects on the table that can be grabbed and have different temperatures and a group of slider objects that we can interact with and two objects that create a continuous temperature and texture effect



URP Materials Upgrade

Here you can find the URP Materials:

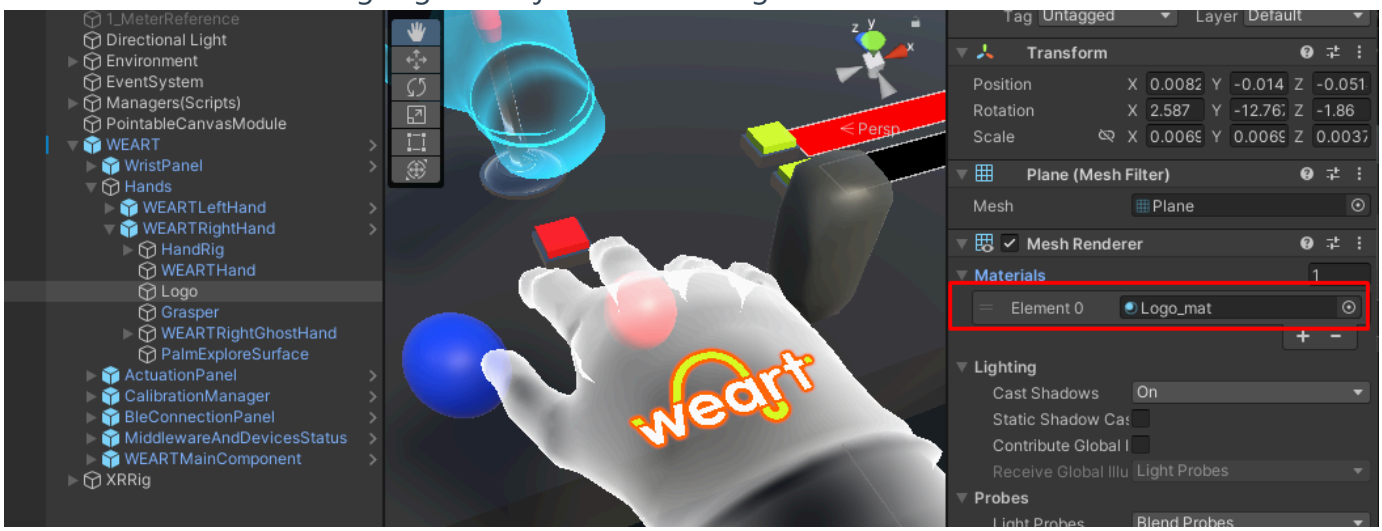
Remember to follow the steps for both hands.

If you are having a project that uses URP or want to create a new project with it, and you want to use the sample scene make sure to upgrade the materials present in:
Packages→ WEART SDK→ Runtime→ Materials
Assets→ Samples→ WEART SDK→ 1.3.0→ Sample Demo→ Materials
Make sure to upgrade also the "Plates" folder from the location above.

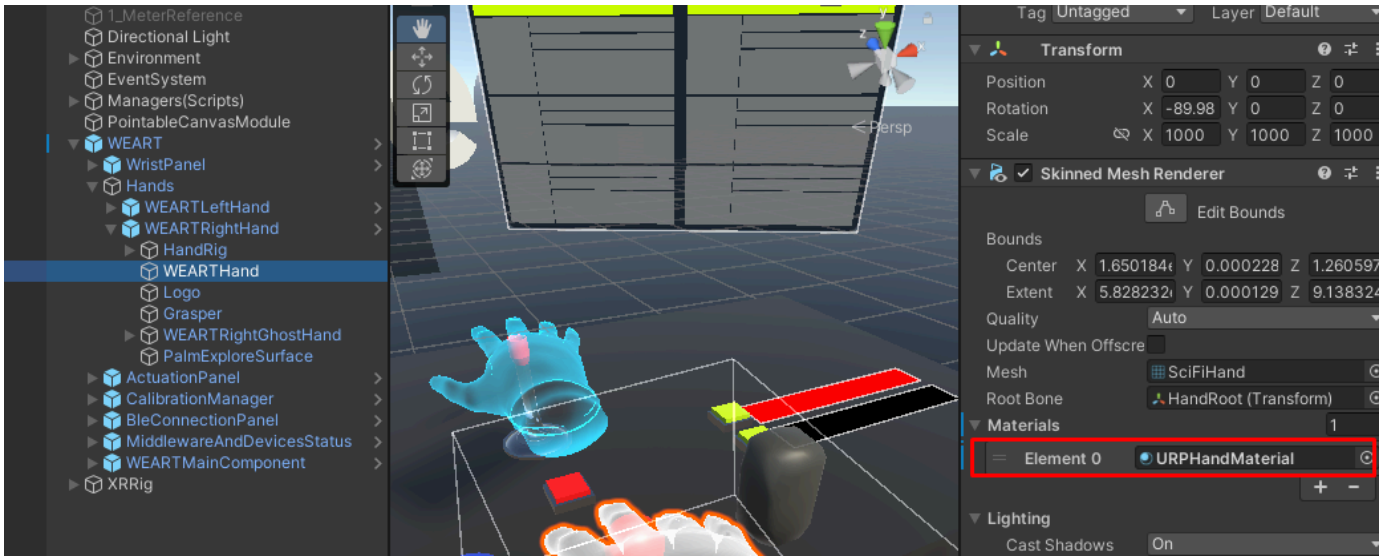
Update Hand's Materials

Starting from now we will use only the materials from WeArt SDK Standalone→ Runtime→ Materials→ URPMaterials

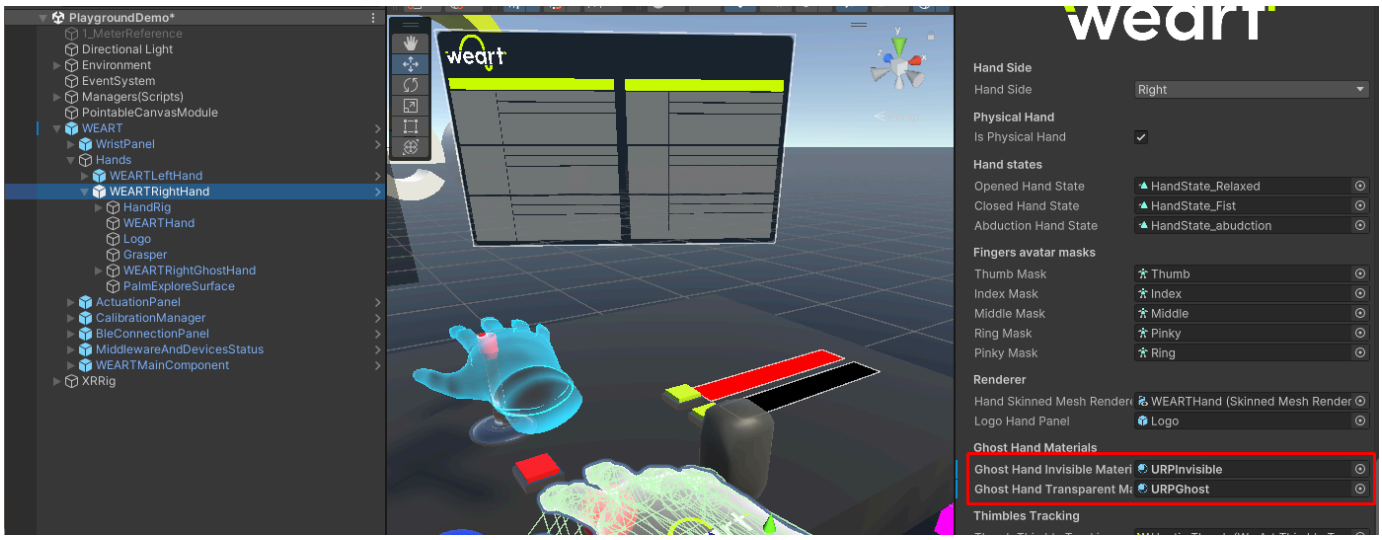
Switch the material of "Logo" game object with URPLogo



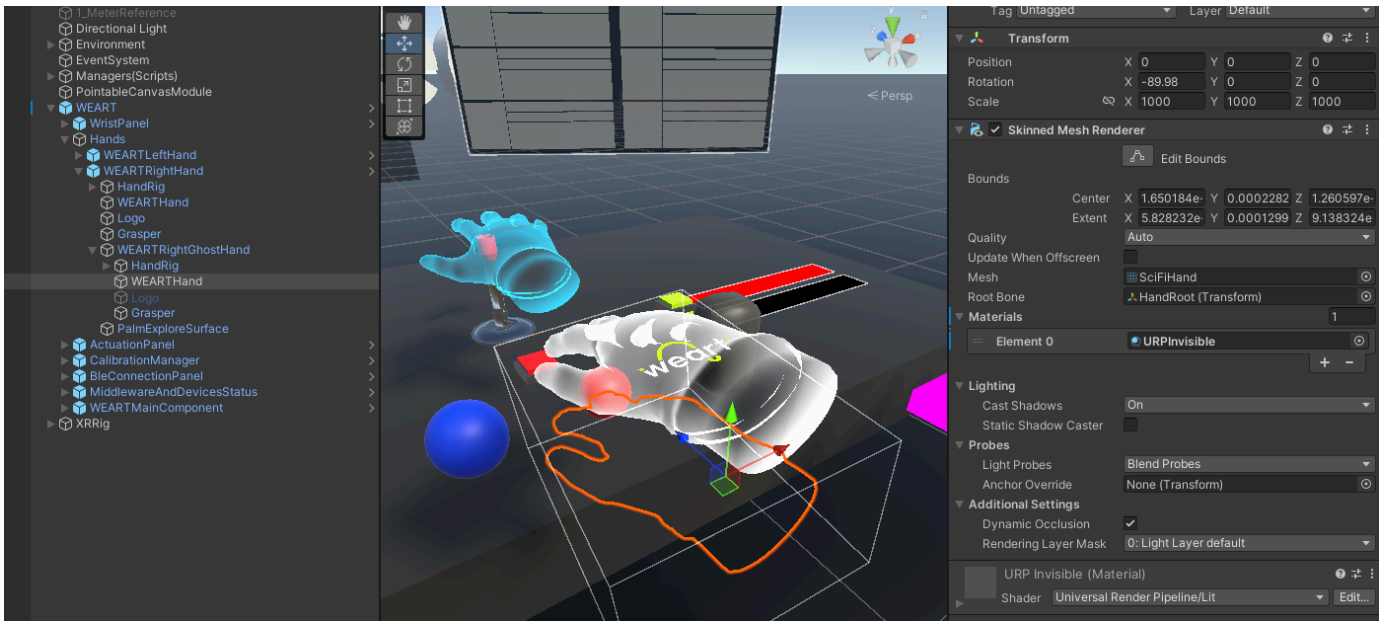
Switch the material of WEARTHand to URPHandMaterial



In WEARTRightHand in the component WeArtHandController replace the GhostHandInvisibleMaterial with URPInvisible and then GhostHandTransparentMaterial with URPGhost.

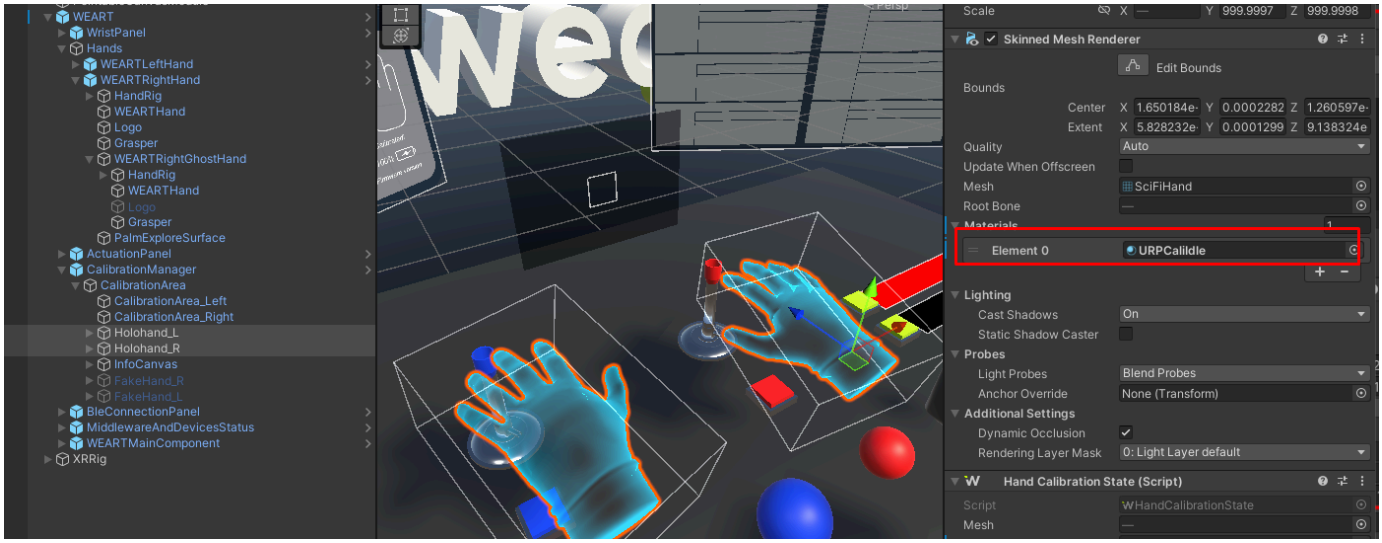


Inside WEARTRightGhostHand there is a child called WEARTHand. Replace its material with URPInvisible



Update Calibration UX Materials

For both Holohand_L and Holohand_R, replace their material to URPCaliIdle



For both Holohand_L and Holohand_R, inside the component HandCalibrationState, replace IdleMaterial with URPCaliIdle, CalibrationMaterial with URPCaliCalibrating, SuccessMaterial with URPCaliSuccess and FailedMaterial with URPCaliFail.