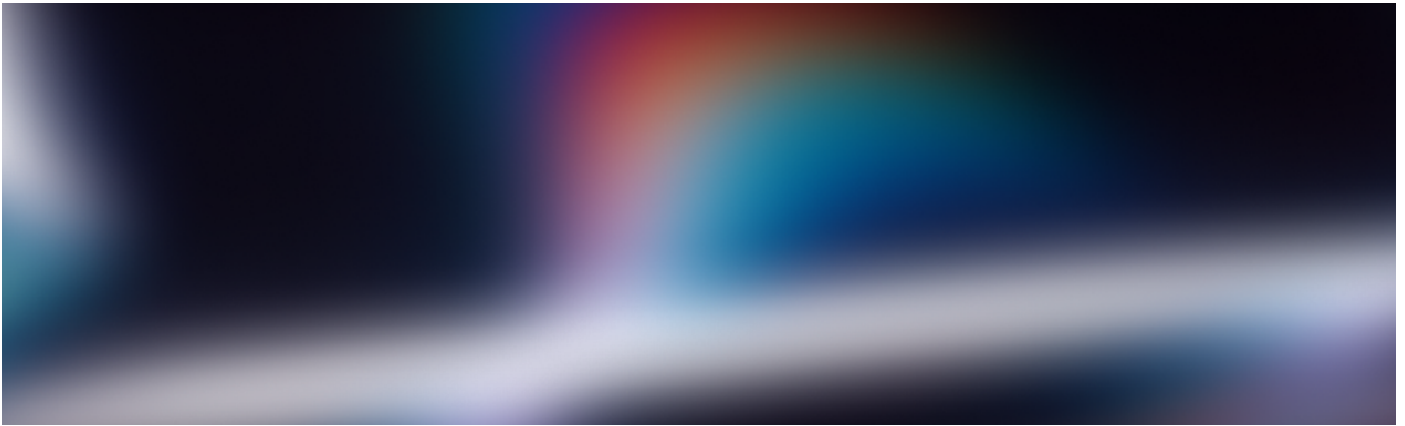


# OctoXR



## OctoXR

Documentation for the OctoXR package for Unity engine (Version 1.0)

It is recommended to access the documentation online at: <https://spectrexr-1.gitbook.io/octoxr/>

OctoXR is a hand-tracking focused plugin that makes it easy to rapidly create a variety of things useful when creating anything from a prototype to a full-scale project.

Whether you want to create interactable objects, world space user interfaces, well-known or experimental locomotion systems or leverage gesture recognition to bring your ideas to life - OctoXR has got you covered!

## Setup

There are a few necessary prerequisites for the package to work so please follow the steps on this page to get everything working nicely.

 These are also some general tips to getting started with VR using the oculus integration package

First, when creating your project, for standalone VR - like the Oculus Quest it is recommended to use the URP project template (this might change with Unity 2021 as there are talks of merging standard and URP together).

### Build settings

The build settings window lets you configure all kinds of platform related settings and later on build the project. This step isn't necessary if you don't intend to build your project, but it is better to do it now rather than later as conversion to a different build platform in later stages of the project might be a painfully long process!

1. In Unity's toolbar, go to File and select Build Settings from the drop down menu

2. Under the platform selection part, change your desired platform from "Pc, Mac & Linux Standalone" to Android
3. Set the texture compression from "Don't override" to ASTC

Your project is ready for Android devices, now let's make it VR ready!

## Project settings

To make your project VR-ready, there are a few things to consider when changing the project settings. First things first we want to enable Oculus support:


1. Open up project settings by going to the Edit tab of Unity's toolbar and selecting Project Settings from the drop down menu - alternatively, if you already had player settings open, this is pretty much the same thing
2. In the project settings window, select the XR Plugin Management option, which should be the last item on the list
3. Press install XR Plugin Management, once it has been installed, some new options will appear in the window
4. Under the android tab, down in the Plug-in Providers section, tick the box next to Oculus, installing this might take some time
5. After it has been installed, a new option under XR Plug-in Management will Appear, named "Oculus", click on it and change Stereo Rendering Mode from Multi Pass to Multiview, this will greatly increase your applications performance once it has actually been built to the device

## Oculus integration

This is the final step before importing OctoXR and probably the most important one. The Oculus integration package provides us with the fundamentals necessary to make hand tracking work, currently, OctoXR is dependent on it to be able to work, but we are working hard on making our package standalone.

Setup:

1. From the Window tab in Unity's toolbar select Asset Store, depending on if you have Unity version 2019 or 2020/21 this will either open inside the editor itself, or lead you to an external link where you'll be able to access the Asset Store
2. While in the Asset Store, use the search bar to look for The Oculus Integration package
3. Once you find it, open it up and click add to Unity, this will add the package to your editor's Package Manager, clicking open in Unity will lead you straight to the Package Manager where you'll download and install the package from
4. Once you've opened the Package Manager, which is located under Window > Package Manager (in case you haven't used the stores direct link) look for The Oculus Integration under My Assets (click the drop down above the list of packages and change from In Project to My Assets) and once you find it press Download in the bottom right corner of the manager's window
5. Once the package has downloaded, you can use the Import button that now appeared in place of the previous Download button and import the package

 You can tick off Oculus Samples, as that part of the package is not required for OctoXR to work and adds unnecessary bloat to our project, if you ever decide to use the samples, you can always just reimport them!

## OctoXR integration

Finally, when you're done setting up all of the prerequisites, if you already haven't, download OctoXR from the Asset Store and then install and import it using the Package Manager.

Once imported, OctoXR will give you an option to automatically set up all layers and physics settings, if by any chance you miss this you can always access it from Window>OctoXR>Setup Window!

Your project is now good to go! We recommend checking out the sample scenes provided in OctoXR > Samples if you want to play around, if you want to get started with building something of your own then head on over to a How To Use section of your desired feature up and running in no time!

# INPUT

## Input

### HandTrackingDataProvider

- Abstract MonoBehaviour that serves as a base class for providing hand tracking data such as hand bone poses and hand tracking confidence status
- This behaviour is usually implemented by platform specific behaviours that read hand tracking state from their corresponding platforms' SDK packages. For now there is only hand tracking data provider for Oculus platform that requires Oculus Integration package to be imported into Unity project

### Public methods

`abstract HandType GetHandType()`

- When implemented in a derived class it should return a value that indicates whether the hand tracking data provider provides tracking data for left or right hand

`abstract bool IsFingerTrackingConfidenceHigh(HandFinger finger)`

- When implemented in a derived class it should return a value that indicates whether the tracking confidence is high for the finger specified by HandFinger parameter

abstract bool IsFingerPinching(HandFinger finger)

- When implemented in a derived class it should return a value that indicates whether the finger specified by HandFinger parameter is currently pinching

abstract float GetFingerPinchStrength(HandFinger finger)

- When implemented in a derived class it should return a quantity that specifies current pinch strength of the finger specified by HandFinger parameter. The returned value should be in [0, 1] range

abstract HandTrackingSkeletonData GetHandTrackingSkeletonData()

- When implemented in a derived class it should return a structure containing data that specifies current tracked hand bone poses along with indicator whether the tracked bone poses and other tracking data is of high confidence. Exact members of HandTrackingSkeletonData structure and their meaning are depicted in the [section below](#)

abstract bool TryGetHandPointerPose(out Pose pointerPose)

- When implemented in a derived class it should try to obtain pose of the tracked hand associated input pointer. Pointer pose is assigned to the output parameter and return value indicates whether the pose is valid and successfully obtained

## HandTrackingSkeletonData

- Structure that provides hand tracking skeleton data

Public members

Pose RootPose

- Position and orientation of hand skeleton root. Hand skeleton root usually corresponds with the hand wrist. It is relative to the tracking space root pose, it does not necessarily correspond to the world space pose of the hand skeleton root. This has to be taken into account mainly when using locomotion and teleport, or any other time tracking space root is moved from the world space origin

float RootScale

- Scale of the hand skeleton root. This scale can be applied to hand skeleton so that the hand bone joints of the hand in the real world approximately correspond with bone joints in the virtual world, i.e. hand size in the virtual world appears approximately the same as the hand from the real world would be in virtual world

ReadOnlyCollection<Quaternion> BoneRotations

- A read-only collection of current hand bone rotations as detected by the underlying hand tracking

system, or are at least considered to be as such. It is expected that this collection contains at least the same number of items as there are different values in HandBoneId enumeration. It is also expected that rotations in this list are sorted in a particular order in order to determine which exact hand bone does each of the quaternions in the list specify rotation of. Mainly, rotation for a bone identified by a value from the HandBoneId enumeration is considered to be located at the index that is equal to the numerical value of the HandBoneId of that bone. Rotations are assumed to be in local space of each of the corresponding bone, relative to parent bone. Root bone's rotation is relative to RootPose rotation. Exact indices and hand bone identities and relations between them are specified in the following table

Hand part / bone	HandBoneId	Parent HandBoneId	Index / ID Number
Hand palm	HandPalm	-	0
Thumb finger metacarpal bone	ThumbFingerMetacarpal	HandPalm	1
Thumb finger proximal phalanx	ThumbFingerProximalPhalanx	ThumbFingerMetacarpal	2
Thumb finger distal phalanx	ThumbFingerDistalPhalanx	ThumbFingerProximalPhalanx	3
Index finger proximal phalanx	IndexFingerProximalPhalanx	HandPalm	4
Index finger middle phalanx	IndexFingerMiddlePhalanx	IndexFingerProximalPhalanx	5
Index finger distal phalanx	IndexFingerDistalPhalanx	IndexFingerMiddlePhalanx	6
Middle finger proximal phalanx	MiddleFingerProximalPhalanx	HandPalm	7
Middle finger middle phalanx	MiddleFingerMiddlePhalanx	MiddleFingerProximalPhalanx	8
Middle finger distal phalanx	MiddleFingerDistalPhalanx	MiddleFingerMiddlePhalanx	9
Ring finger proximal phalanx	RingFingerProximalPhalanx	HandPalm	10
Ring finger middle phalanx	RingFingerMiddlePhalanx	RingFingerProximalPhalanx	11
Ring finger distal phalanx	RingFingerDistalPhalanx	RingFingerMiddlePhalanx	12
Pinky finger proximal phalanx	PinkyFingerProximalPhalanx	HandPalm	13

Pinky finger middle phalanx	PinkyFingerMiddlePhalanx	PinkyFingerProximalPhalanx	14
Pinky finger distal phalanx	PinkyFingerDistalPhalanx	PinkyFingerMiddlePhalanx	15
Thumb finger tip	ThumbFingerTip	ThumbFingerDistalPhalanx	16
Index finger tip	IndexFingerTip	IndexFingerDistalPhalanx	17
Middle finger tip	MiddleFingerTip	MiddleFingerDistalPhalanx	18
Ring finger tip	RingFingerTip	RingFingerDistalPhalanx	19

bool IsDataValid

- Indicates whether the hand tracking data is valid. If this value is false it usually indicates that the hand tracking is currently not working

bool IsDataHighConfidence

- Indicates whether the hand tracking data is considered to be highly accurate. Consequently this value is always false if IsDataValid is false as well

bool IsSystemGestureInProgress

- Indicates whether some built-in platform defined gesture is currently in progress. These kind of gestures usually have some special platform defined functionalities, like exiting the current app and others

## Hand Skeleton

- Collection of parts and bones that together make up a left or right hand starting at wrist
- Obtains hand bone poses from a [HandTrackingDataProvider](#) and sets those poses to the hand bones associated with it
- HandTrackingDataProvider is associated from the same GameObject it is attached to or from parent objects, searching goes up the GameObject hierarchy. If a tracking data provider is not found, HandSkeleton will not generate or cause any errors, it simply will not do anything
- Hand bones that the skeleton consists of must be organized hierarchically and HandSkeleton must be parent/ancestor to them all
- Bones are GameObjects that have a HandBone component attached

- Cannot be added to a GameObject that already has HandSkeleton attached or that is hierarchically located inside an existing HandSkeleton's bone hierarchy
- It will not update bones if the hand tracking skeleton data obtained from hand tracking data provider is not valid
- HandBone poses are set via their Transform components. It is therefore not a good idea to attach a component to a HandBone that will result in HandBone's Transform being under some other system's control, for example physics by attaching a Rigidbody or ArticulationBody
- HandSkeleton is relatively important script, many other behaviours defined in OctoXR depend on it

## Public properties

### HandTrackingDataProvider HandTrackingDataProvider

- Hand tracking data provider the HandSkeleton uses to obtain target bone poses. This property is read-only, HandSkeleton sets it itself in a way that is described above

### HandType HandType

- Defines for which hand the hand skeleton updates bone poses, left or right. This is read-only and depends on HandTrackingDataProvider's value for hand type, it is defined in HandSkeleton for convenience only

### HandSkeletonUpdateModes UpdateMode

- Specifies when should the HandSkeleton update hand bone poses. Possible values are NoUpdate, Update and FixedUpdate. Values can be bit-wise combined so that the HandSkeleton updates bones both in Update and FixedUpdate methods and it can also be set to not update bones at all which is essentially the same as disabling it

### bool UpdateRootPose

- Specifies whether the skeleton should update it's own transform with the root pose obtained from HandTrackingDataProvider (RootPose member in the [HandTrackingSkeletonData](#)). You will probably want to set this value to true, but there may be many cases when skeleton should not update root pose if certain specific role/behaviour is needed from it

### bool UpdateRootScale

- Similar to UpdateRootPose, this specifies whether the HandSkeleton should update it's own scale with the root scale value obtained from HandTrackingDataProvider

### HandBoneCollection<HandBone> Bones

- This is list of all bones that the HandSkeleton consists of. List items can be accessed via zero-based integer index or via HandBoneId key. This list is read-only, bones are added automatically as a

consequence of attaching a HandBone scripts to a particular child objects of the HandSkeleton, bool IsComplete

- Indicates whether the HandSkeleton has all the hand bones that it can possibly have. All hand bones, their identities and other relevant information is listed in a table on [this](#) page. It is important that the HandSkeleton always has all the hand bones since many other scripts in OctoXR rely on that being the case

## Building a HandSkeleton

In most cases manual creation of a hand skeleton will not be necessary as there are prefabs included in OctoXR package that have a pre-built skeleton set up and ready, such as OctoHand\_Left/Right or OctoHandSkeleton\_Left/Right.

However, there may be a need to manually add HandSkeleton script to a GameObject, such as when you have your own custom hand model that you wish to use in a scene. In that case the process is as follows:

- Add HandSkeleton script to a GameObject that would correspond to the hand wrist
- To add bones for the HandSkeleton you need to attach HandBone scripts to objects that are child objects of the one that you attached the HandSkeleton to

Adding HandBone scripts to GameObjects that need to act as hand bones must be performed following certain rules. This is because HandBone scripts, when added, initialize their properties based on their positions under the HandSkeleton's hierarchy. Because of this HandBones should be added only on certain objects and in certain order, depending on which exact GameObject you wish to map to a particular hand bone. Specifically, the rules are:

- HandBone that is added to a GameObject that is direct child of the GameObject with HandSkeleton on it, i.e. it is 1 depth level below HandSkeleton, is initialized as HandPalm bone
- HandBones added to a GameObject 2 depth levels below HandSkeleton (child of a child of the HandSkeleton, or child of the already added HandPalm bone) are initialized as the first phalanx of a finger. Which finger exactly depends on which finger bones are already in the HandSkeleton. If there are no finger bones in the HandSkeleton then it will end up as the first bone of thumb finger (ThumbFingerMetacarpal), if there is one or more thumb bones in the HandSkeleton already present, then it will end up as the first bone of index finger (IndexFingerProximalPhalanx), but if there is one or more index finger bones in the HandSkeleton as well, then it will evaluate to middle finger bone etc. Basically fingers are prioritized in order Thumb -> Index -> Middle -> Ring -> Pinky. The way the bone identity is determined is by searching the first available finger in the HandSkeleton, i.e. finding the first finger that does not have any bones in the HandSkeleton, prioritizing fingers in previously stated order. There is also one more possibility: there exists a HandBone in a GameObject that is hierarchically below the HandBone's GameObject and was previously added to the HandSkeleton. In that case newly added HandBone will determine the finger it belongs to based on which finger the child HandBone belongs to
- HandBones added to a GameObject 3 or more depth levels below HandSkeleton will determine its identity in much the same way as described above for the ones 2 depth levels beneath the HandSkeleton. Finger is determined depending on the current finger bones present in the HandSkeleton and/or which finger does an existing parent or child HandBone(s), if there are any, of the



newly added one belong to. HandBones added 3 levels below HandSkeleton always end up as one of: ThumbFingerProximalPhalanx, (Index/Middle/Ring/Pinky)FingerMiddlePhalanx. HandBones 4 levels below HandSkeleton then evaluate to one of distal phalanxes and finally HandBones 5 level below end up as finger tips

In conclusion, HandBones' identities are determined primarily based on their distance in depth from the HandSkeleton they should be attached to. HandBone's order on its hierarchy level (sibling index) does not matter at all. Note also that you don't need to add HandBones in any particular depth order, you can attach finger tip bones before HandPalm for example. It is depth that matters.

Having described all that, it should be noted that there are certain scenarios that can cause errors. For example attaching HandBone script to a GameObject that is 6 or more depth levels beneath a HandSkeleton or there is no HandSkeleton hierarchically above the HandBone's GameObject at all. There may also be cases when attaching a HandBone results in HandBone determining its identity and then finding the HandSkeleton already contains HandBone with the same identity or sometimes HandBone's identity cannot be determined at all based on its position in the HandSkeleton's hierarchy and existing bones in the HandSkeleton.

All such cases where attaching HandBone to a GameObject end up with HandBone and/or HandSkeleton ending up in some invalid state result in newly attached HandBone being immediately destroyed with error message and details written to the Unity editor's console, so keep an eye on it. Note that the HandSkeleton after any such error remains in perfectly valid state and you don't need to worry about it not working.

Also to note is the fact that you should attach all hand bones to the HandSkeleton to make it complete. As long as that is not the case you will see a warning message written in the HandSkeleton's and any and all of its existing HandBones' inspector window in the Unity editor. Refer to the table on [this page](#) to see which and how many hand bones there should be in the HandSkeleton to make it complete.

## Platforms

## Oculus

### OculusHandTrackingDataProvider

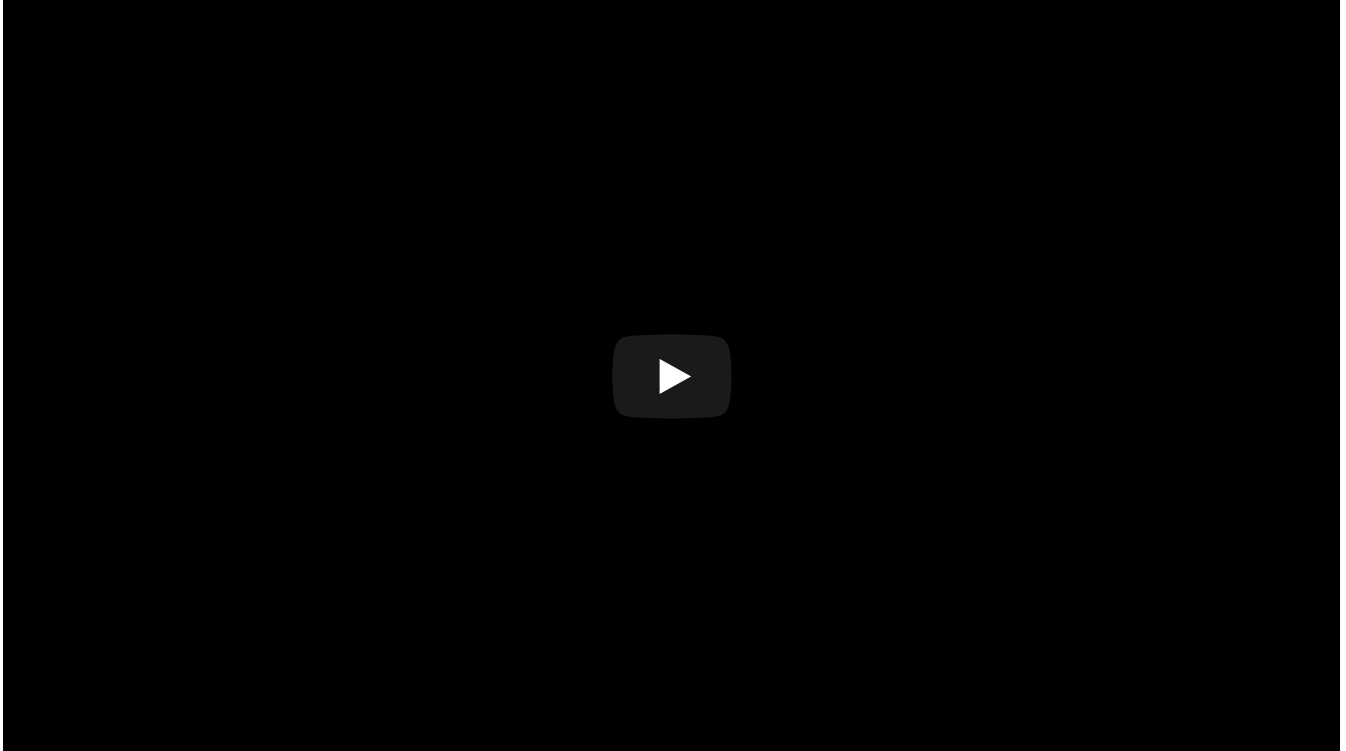
- Oculus platform specific implementation of [HandTrackingDataProvider](#)
- Requires a OVRHand script from Oculus Integration package attached to the same GameObject the OculusHandTrackingDataProvider is attached to

#### Public interface

- Being an implementation of abstract HandTrackingDataProvider, public interface of OculusHandTrackingDataProvider matches the public interface of that MonoBehaviour, it conforms to the description of [abstract members of HandTrackingDataProvider](#)

# Gesture Recognition

## How To Use



Short video tutorial explaining how to implement gesture recognition using OctoXR

## Setting Up

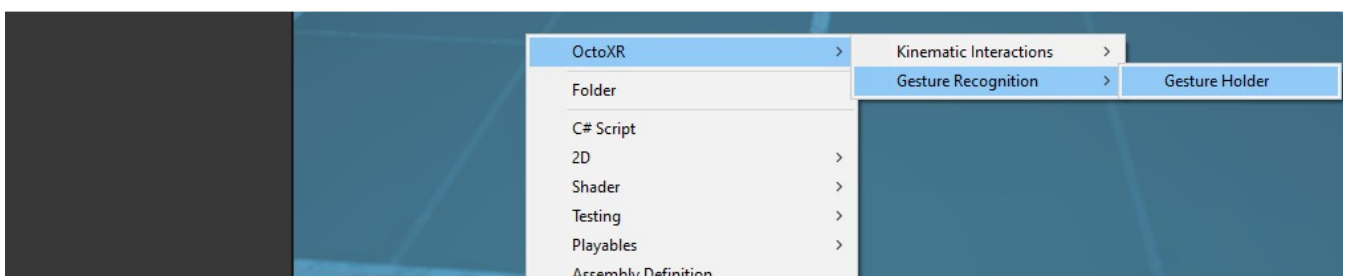
Drag and drop *OctoHand\_Left* and *OctoHand\_Right* prefabs as a child of *TrackingSpace* and tick the *Update Root Pose* box on *Hand Skeleton*.

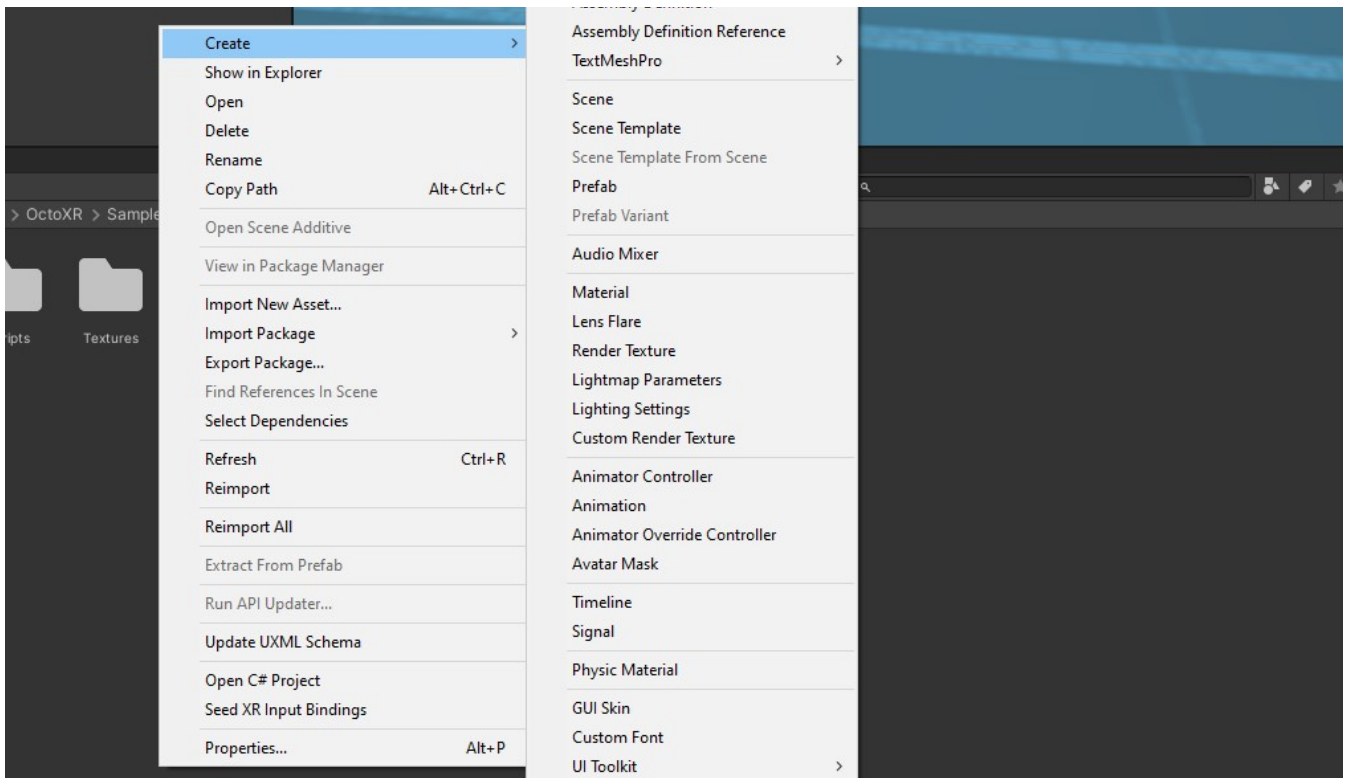
## Creating Gestures

Gestures can be created with or without the use of *GestureHolder* scriptable object.

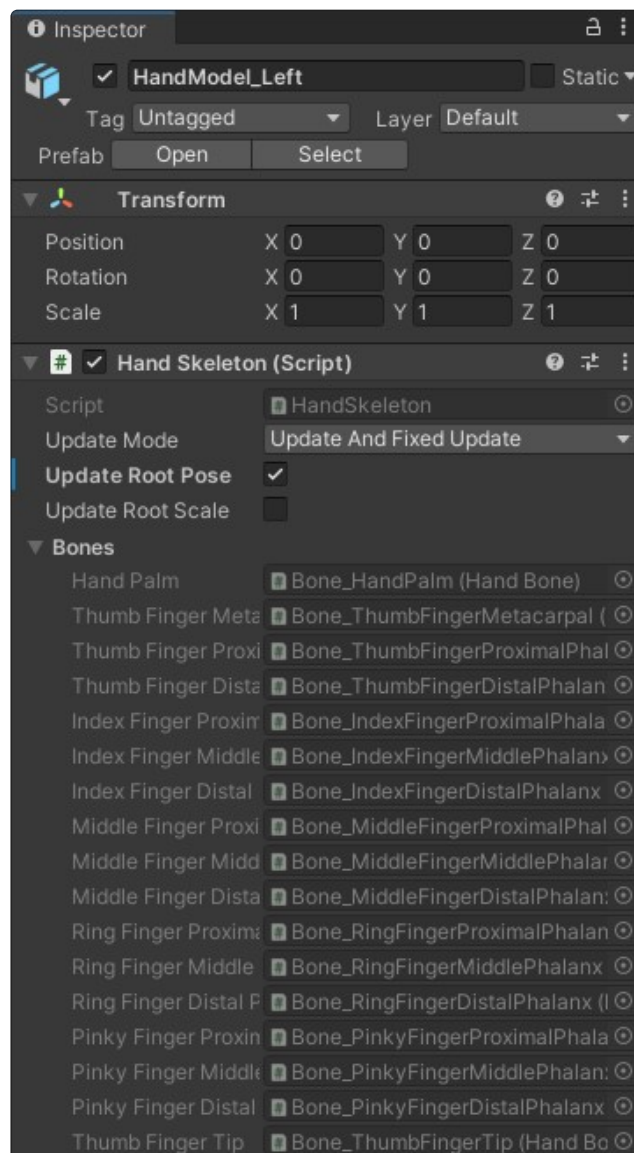
### With *GestureHolder*

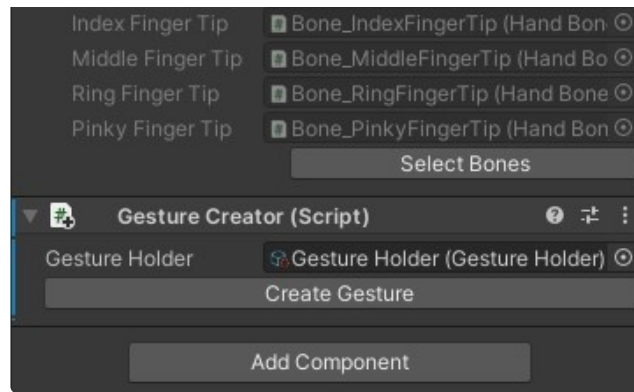
Right click somewhere in the project's assets folder and click on  
Create/Octo XR/Gesture Recognition/Gesture Holder



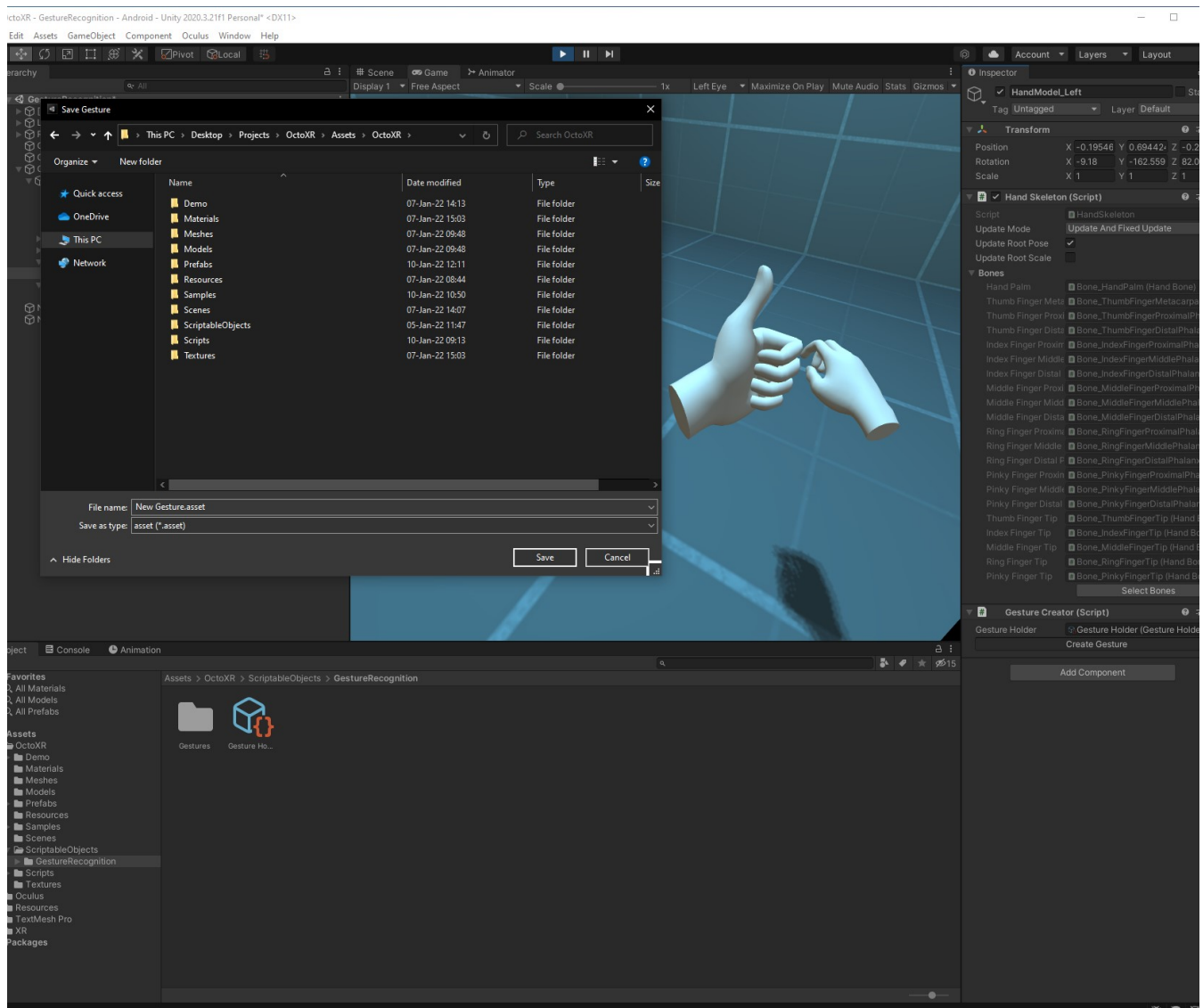


Attach the *GestureCreator* component to *HandSkeleton* and drag and drop previously made *GestureHolder*.





Start the play mode and click the "Create Gesture" button while doing the gesture with hand. Save the gesture in desired folder. All of gestures are saved both in the assets folder and in the *GestureHolder* scriptable Object.



## Without *GestureHolder*

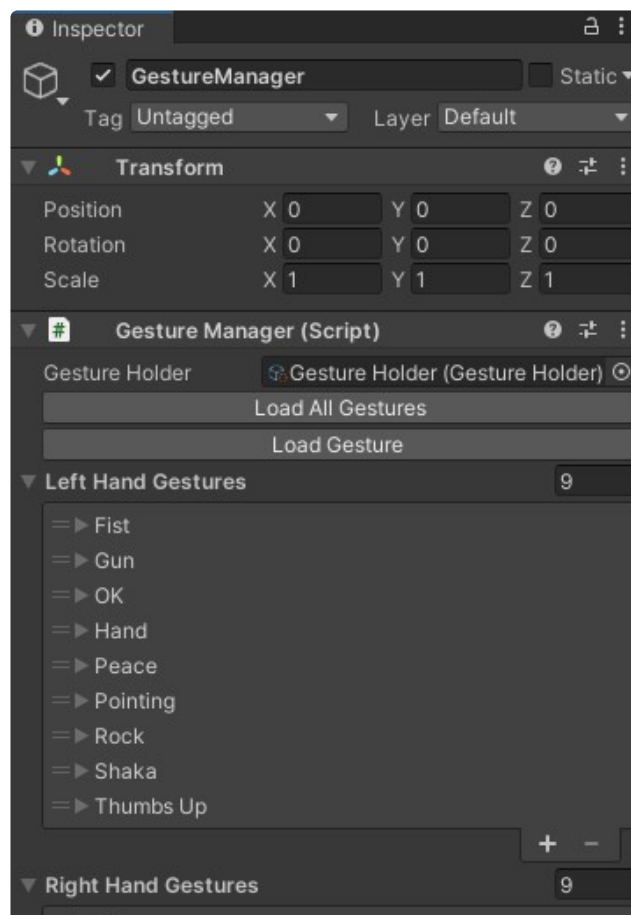
You can choose not to use *GestureHolder* scriptable object. If you do not provide *GestureCreator* with the *GestureHolder*, your gestures will be saved and stored in the projects assets folder but not in the *GestureHolder*.

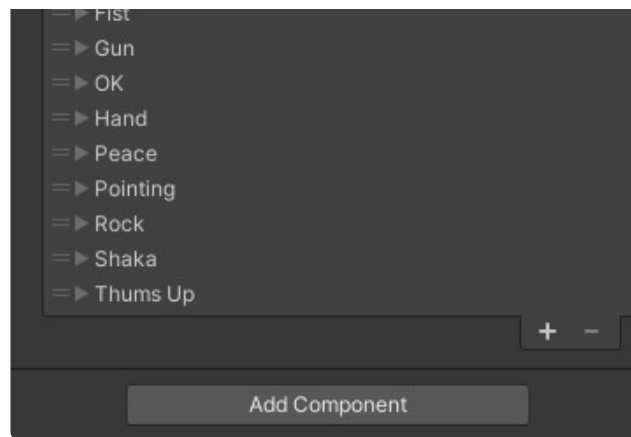
## Setting Up *GestureManager*

Create empty game object and attach the *GestureManager* component.

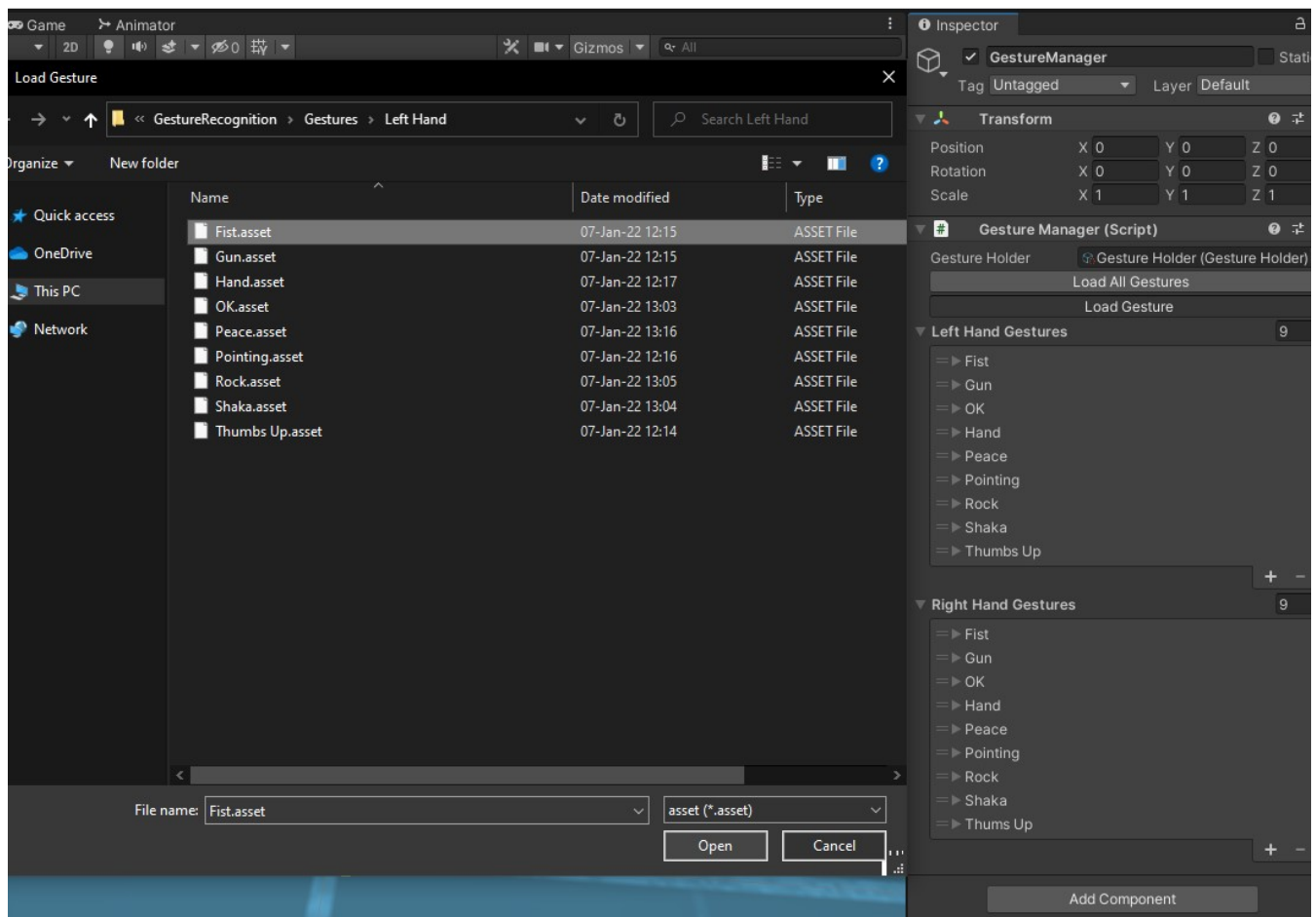


If you drag and drop previously created *GestureHolder* and click the "Load All Gestures" button, your lists will be filled with gestures.



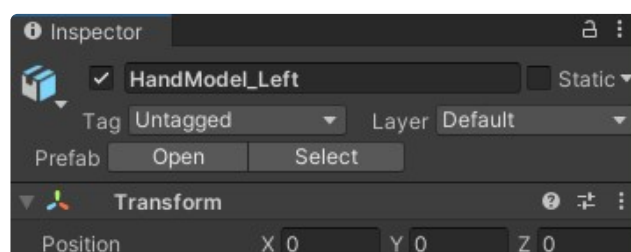


If you did not create *GestureHolder*, you can load your gestures by clicking "Load Gesture" button and browsing for gesture in the project's assets folder.

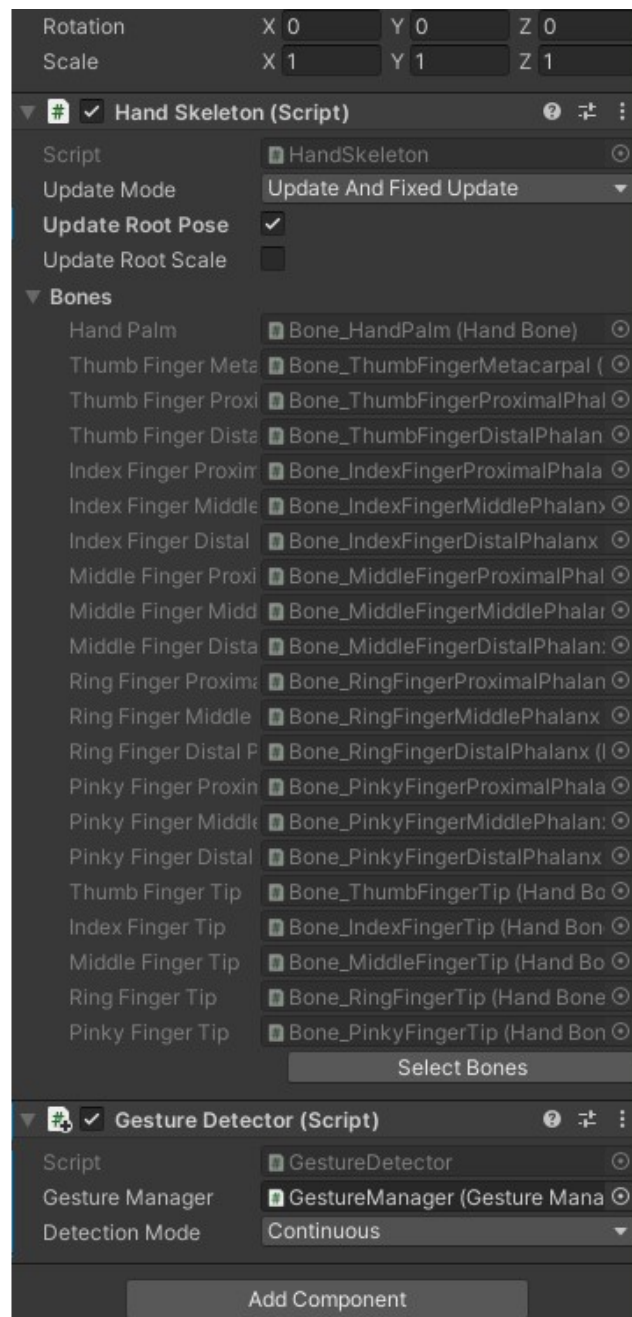


## Gesture Detection

Remove *GestureCreator* component from *HandSkeleton* and add the *GestureDetector*. Drag and drop previously made *GestureManager*. You can read more about Detection Modes in next section.

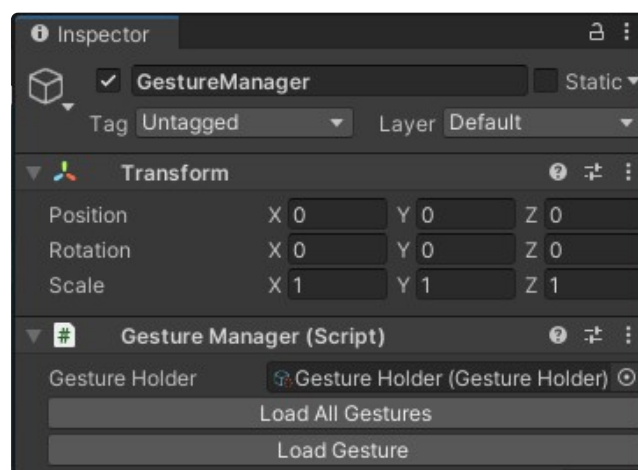


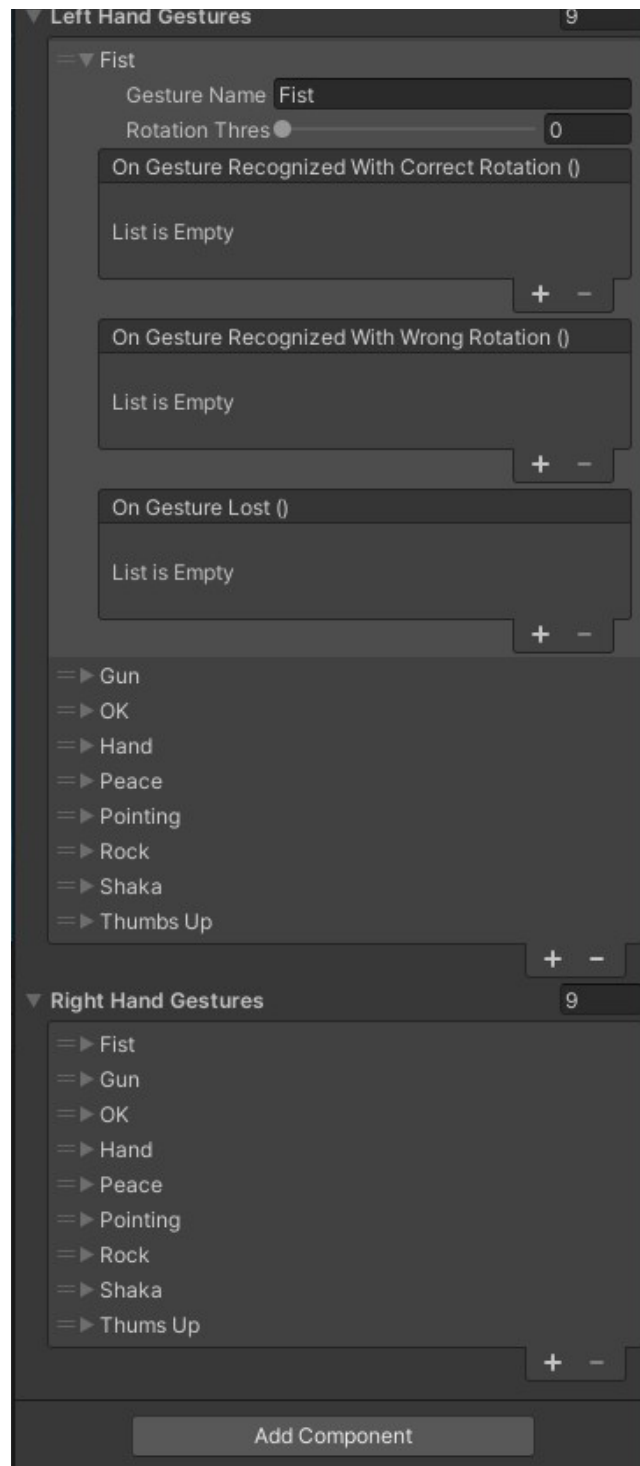




## Gesture Recognized Events

Every gesture has three events you can connect your functionalities to. More about them and what is Rotation Threshold in the next section.





## Scripting

### GestureScriptableObject

- Used for storing and sharing gestures through multiple scenes or projects
- Translated into *Gesture* when used in scripting
- Instantiated by *GestureCreator* when *CreateGesture()* method is called



## Public Fields

string GestureName

float DotValueUp

float DotValueRight

float DotValueForward

List<Vector3> HandBonePositions

HandType HandType

## GestureHolder

- Scriptable object which stores *GestureScriptableObjects* in a list
- Instantiated in Assets/Create/OctoXR/Gesture Recognition/Gesture Holder

## Public Methods

void AddLeftHandGesture(GestureScriptableObject)

void AddRightHandGesture(GestureScriptableObject)

List<GestureScriptableObject> GetLeftHandGestures()

List<GestureScriptableObject> GetRightHandGestures()

## Gesture

- Structure that represents a single gesture
- Created by *GestureManager*
- Translated from *GestureScriptableObject*

## Public Fields

string GestureName

- Gesture name given when saving the *Gesture*

float DotValueUp

- Dot product from *GestureCreator.transform.up* and *Vector3.Up*

float DotValueRight

- Dot product from *GestureCreator.transform.right* and *Vector3.Up*

float DotValueForward

- Dot product from *GestureCreator.transform.Forward* and *Vector3.Up*

float RotationThreshold

- When detecting gestures, *GestureDetector* compares current hand rotation with rotation when gesture was created (by comparing dot values). *RotationThreshold* is a value that represents a tolerance for deviation from rotation when gesture was created
- Range [0, 1] where 0 is minimal tolerance and 1 is maximum tolerance

List<Vector3> HandBonePositions

- List of hand bone positions relative to *HandSkeleton*

HandType HandType

- Enum that marks left or right hand

## Public Events

UnityEvent OnGestureRecognizedWithCorrectRotation

- Will be invoked when delta from current hand rotation and created hand rotation is **BELLOW** *RotationThreshold*
- if *RotationThreshold* is set to 0, will never invoke
- if *RotationThreshold* is set to 1, will always invoke

UnityEvent OnGestureRecognizedWithWrongRotation

- Will be invoked when delta from current hand rotation and created hand rotation is **ABOVE** *RotationThreshold*
- if *RotationThreshold* is set to 0, will always invoke
- if *RotationThreshold* is set to 1, will never invoke

UnityEvent OnGestureLost

- Will be invoked when current gesture is different than the gesture from last frame

## GestureCreator

- Requires component of type *HandSkeleton*
- *GestureHolder* - Optional reference. If provided, gestures will be saved to it

### Public Methods

GestureScriptableObject CreateGesture()

- Creates and returns *GestureScriptableObject* with all fields set to current hand values

void AddGestureToGestureHolder(GestureScriptableObject)

## GestureManager

- Manages all the gestures in the current scene
- *GestureDetector* uses it for comparing and finding gestures

### Public Methods

Gesture ConvertGestureScriptableObjectToGesture(GestureScriptableObject)

List<Gesture> GetLeftHandGestures()

List<Gesture> GetRightHandGestures()

void AddLeftHandGesture(Gesture)

void AddRightHandGesture(Gesture)

Gesture FindGestureByName(string, HandType)

- If *Gesture* with name is found, returns it. If it's not found, returns new empty *Gesture*

## GestureDetector

- Requires component of type *HandSkeleton*
- It compares current hand bone positions with the created gestures and finds the closest match

### Public Fields

GestureDetectionMode GestureDetectionMode

- Continuous mode will invoke events every frame if gesture is recognized
- Discrete mode will invoke events only when new gesture (different from last detected) is recognized

## Public Methods

Gesture GetPreviousGesture()

- Returns previously detected *Gesture*
- If the *GestureDetectionMode* is set to Continuous, it will return *Gesture* from previous frame
- if the *GestureDetectionMode* is set to Discrete, it will return last detected *Gesture*

Gesture DetectGesture()

- Finds current shown *Gesture*. If no *Gesture* is shown, returns new empty *Gesture*
- If the *GestureDetectionMode* is set to Continuous, it is recommended not to use this method since it has some heavy computation. Calling *GetPreviousGesture()* will suffice

void AddGesture(Gesture)

- Used for adding gestures created in runtime

# User Interface

## How To Use

### Setting Up

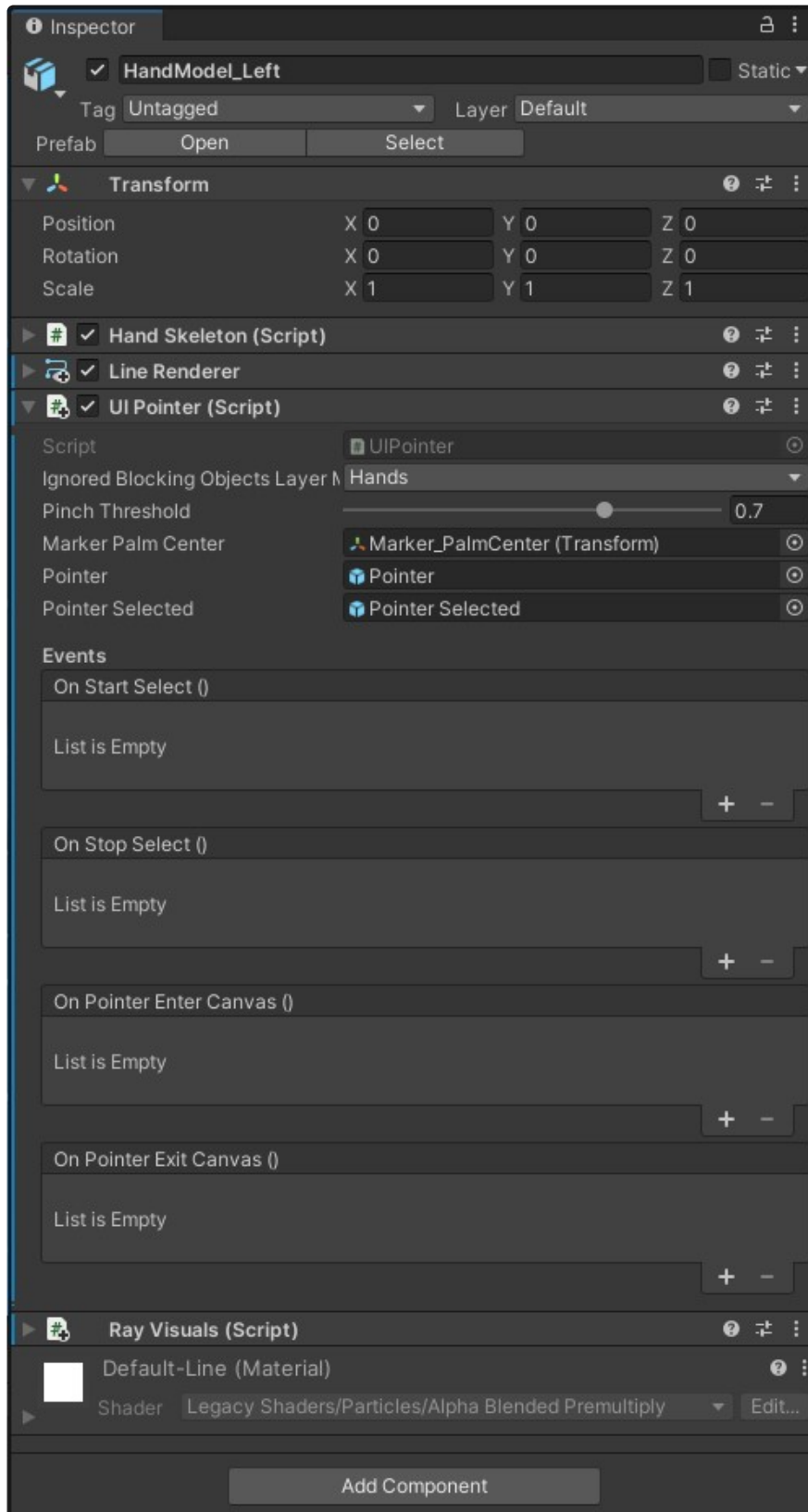
Drag and drop *OctoHand\_Left* and *OctoHand\_Right* prefabs as a child of *TrackingSpace* and tick the *Update Root Pose* box on *Hand Skeleton*.

Drag and drop *OctoInputModule* prefab somewhere in scene hierarchy. Make sure that there are no other input modules or event systems in the scene.

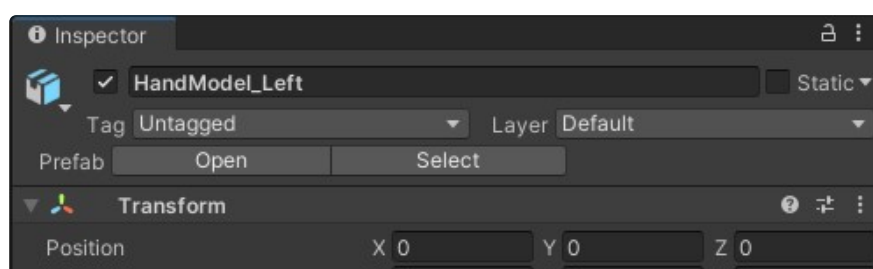
### Adjusting UIPointer

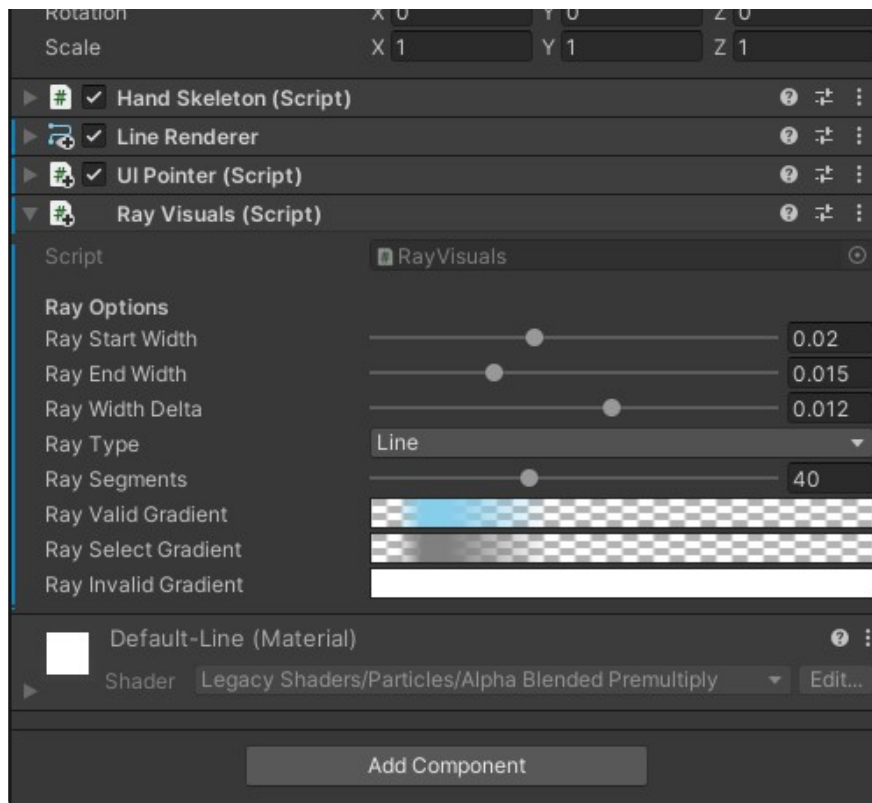
Add *UIPointer* component to the *HandSkeleton*. It will automatically add *RayVisuals* and *LineRenderer* components.

Drag and drop *Pointer* and *PointerSelected* from *OctoInputModule* to *UIPointer*. Find *Marker\_PalmCenter* in the bones hierarchy. If you do not provide it with *Marker\_PalmCenter*, it will find it in the scene automatically.



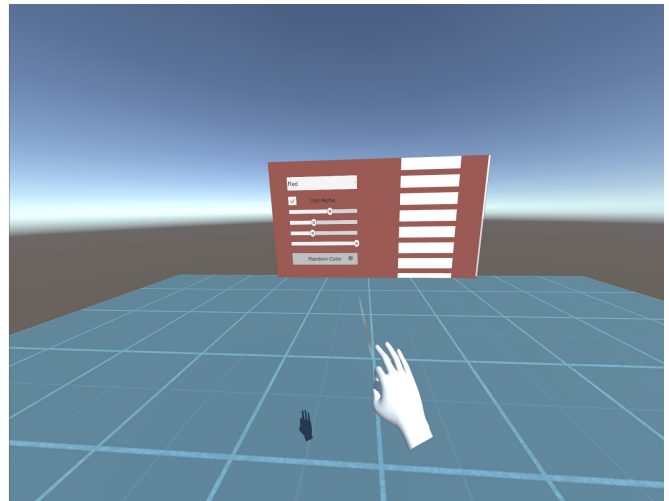
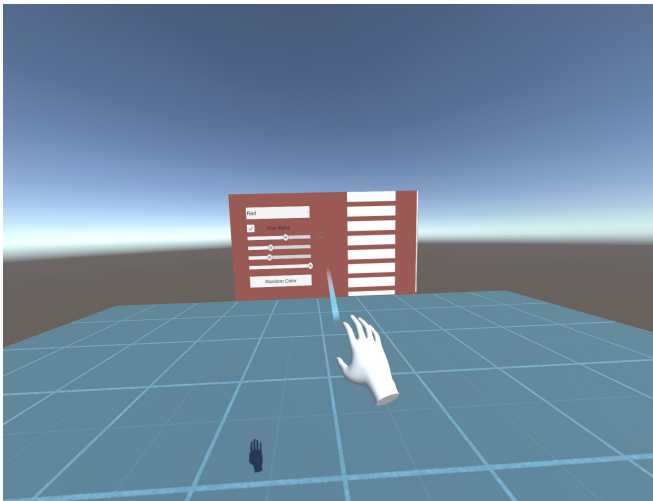
Add material to *LineRenderer* and style your ray using *RayVisuals*.





## Testing

Use pinch gestures to operate UI elements



## Scripting

## UIPointer

## Public Events

UnityEvent OnStartSelect

- Will be invoked when pointer is hovering over UI element and pinches

UnityEvent OnStopSelect

- Will be invoked when pinch release is detected and start action was previously detected

UnityEvent OnPointerEnterCanvas

UnityEvent OnPointerExitCanvas

# Locomotion

## How To Use

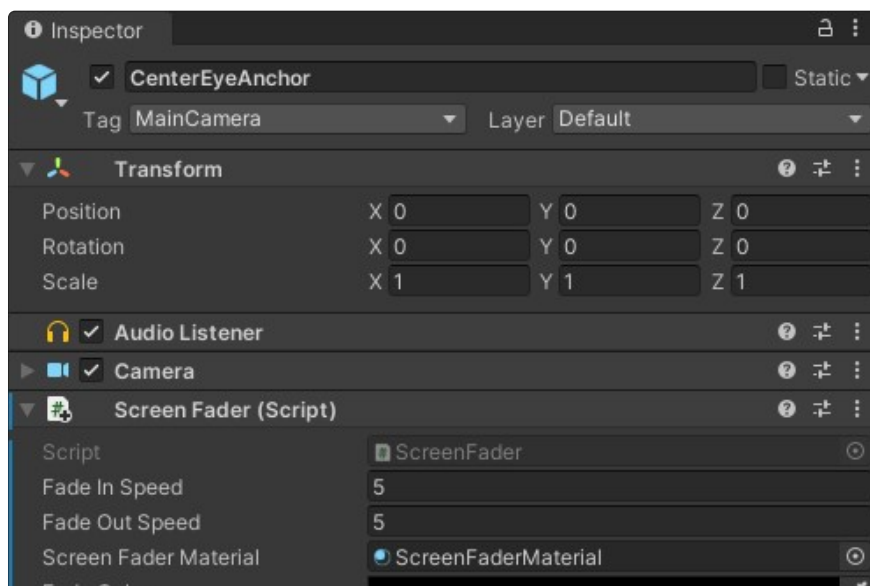
### Setting Up

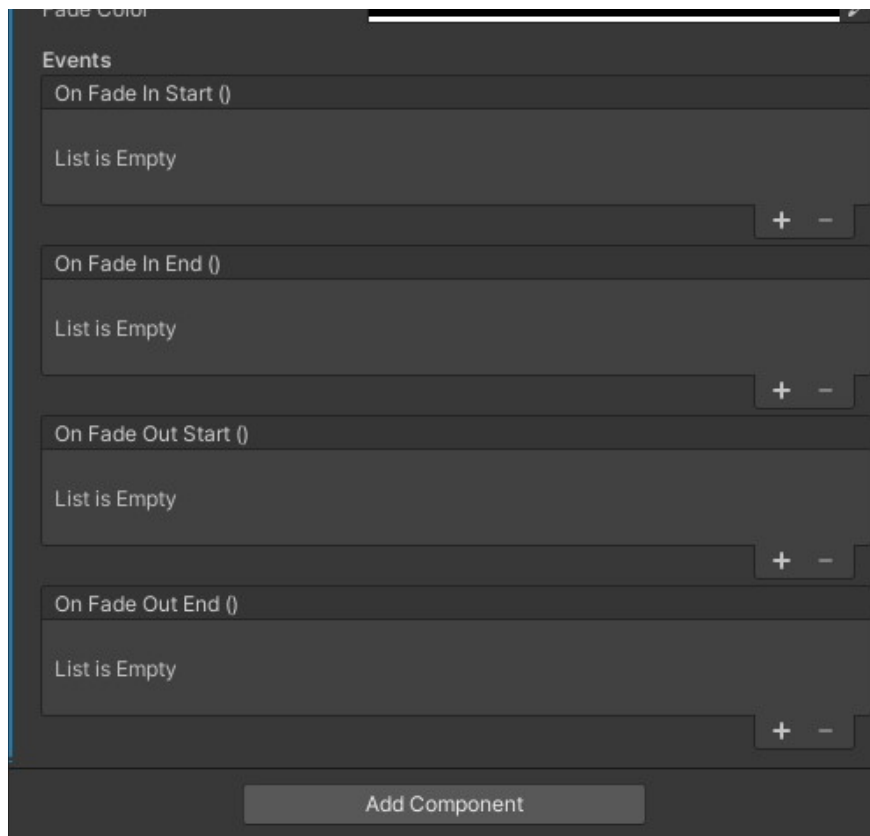
Drag and drop *OctoHand\_Left* and *OctoHand\_Right* prefabs as a child of *TrackingSpace* and tick the *Update Root Pose* box on *Hand Skeleton*.

Drag and drop *Footprints* and *Teleport Reticle* prefabs somewhere in the scene hierarchy. Note that you can use your own custom ones.

### Screen Fader

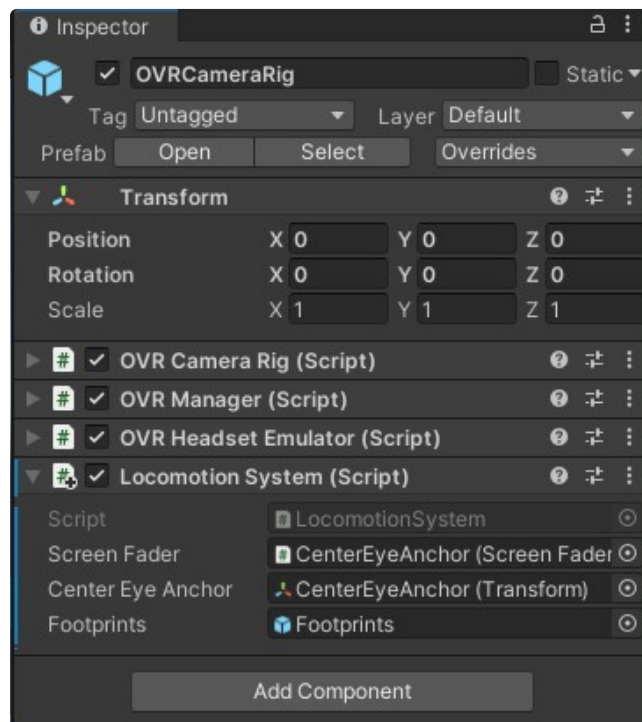
In order to ease the motion sickness caused by immersive VR world, *ScreenFader* was implemented. In the sample scenes, It is mostly used to fade in and out while teleporting (blink teleport) and to fade in when the player is in an uncomfortable situation (perhaps on the edge of the building). Attach it to main camera (in this example *CenterEyeAnchor*). Provide it with the material from OctoXR/Materials folder.





## Locomotion System

Attach *LocomotionSystem* component to the *OVRCameraRig*. Drag *ScreenFader*, *CenterEyeCamera* transform and *Footprints* game object.

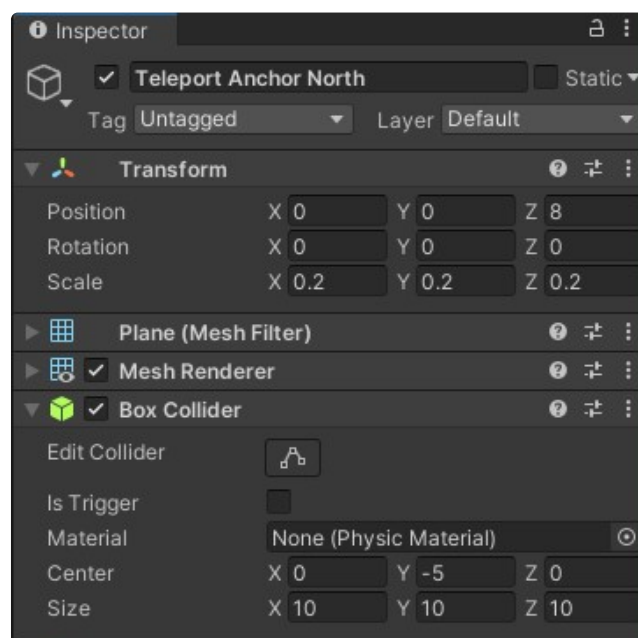
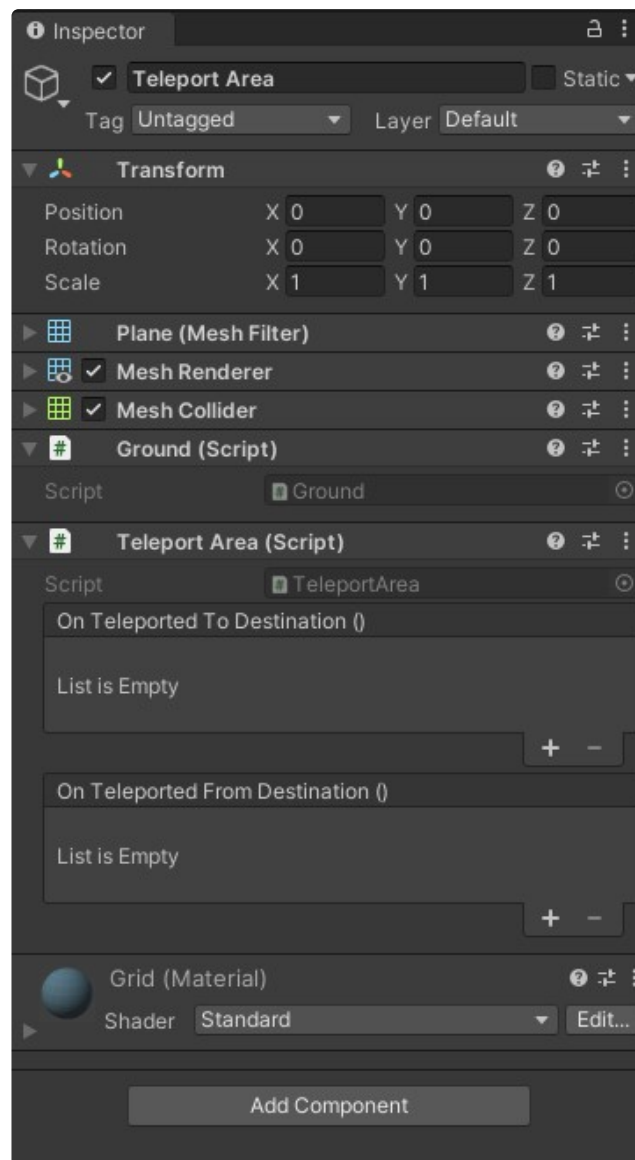


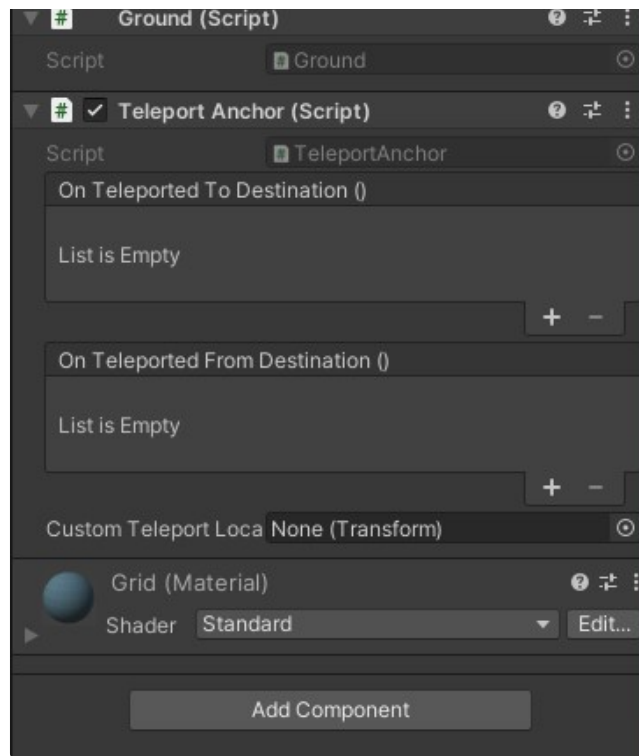
## Create Teleport Destinations

There are two types of teleport destinations. *TeleportArea* which represents larger area where player can



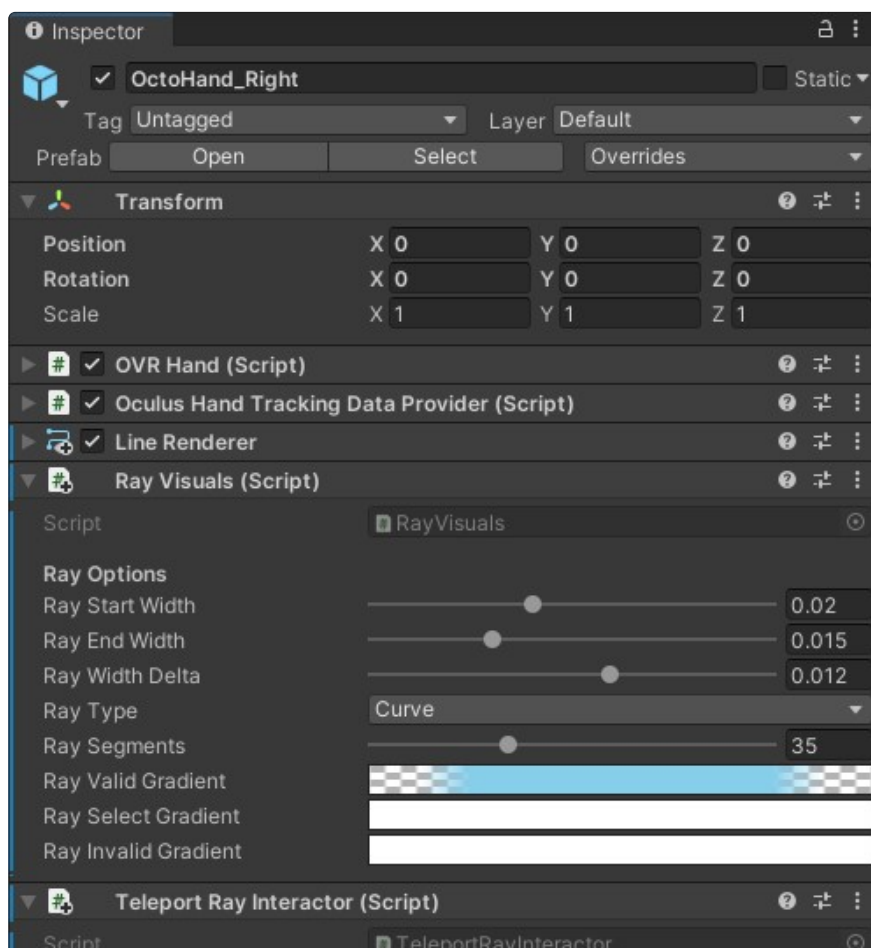
freely teleport and *TeleportAnchor* where player teleports to the fixed location every time. Add a plane to the scene. Attach it with *TeleportArea* component and *Ground* component. Add another plane to the scene. Scale it down. This time attach it with *TeleportAnchor* component as well as *Ground* component. Planes should look somehow like this now:

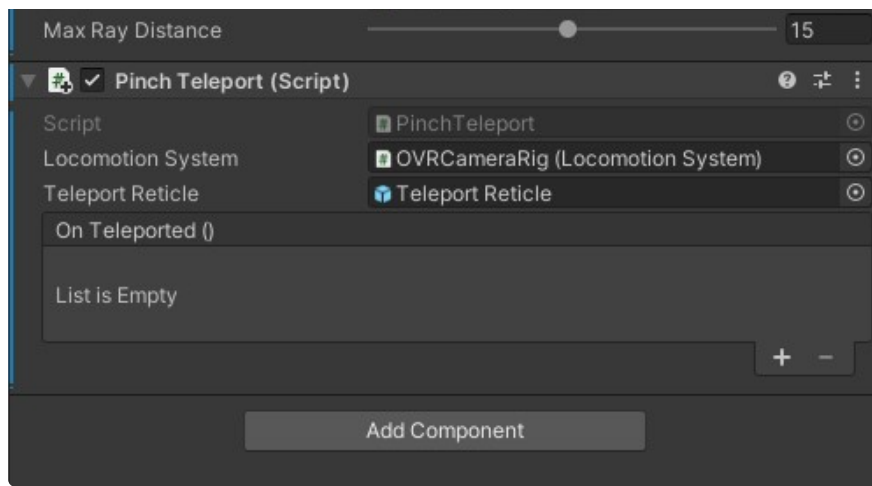




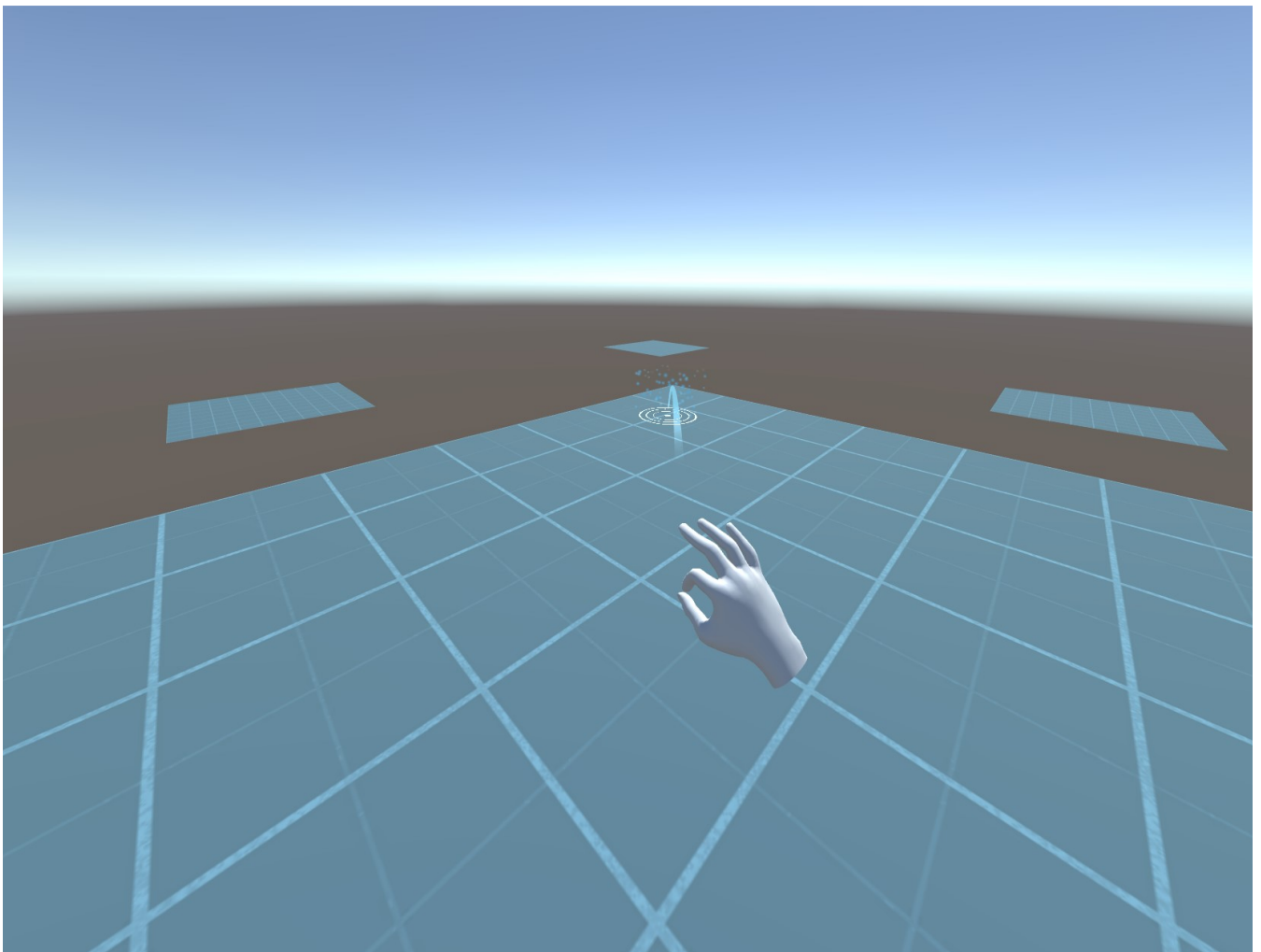
## Teleport

There are multiple predefined types of teleport in the OctoXR. For the purpose of this example we will only show you how to implement *PinchTeleport*. Find *OctoHand* game object and attach it with *PinchTeleport* component. It will automatically attach *TeleportRayInteractor*, *RayVisuals* and *LineRenderer* component. Set it up like in the picture:

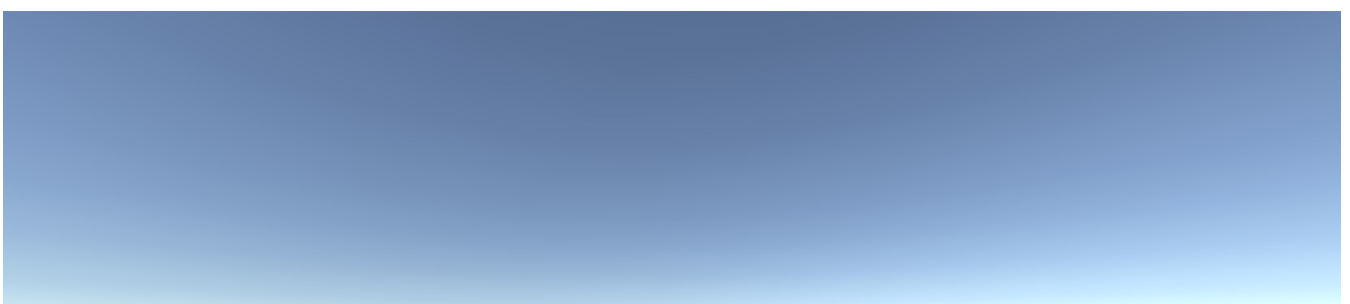


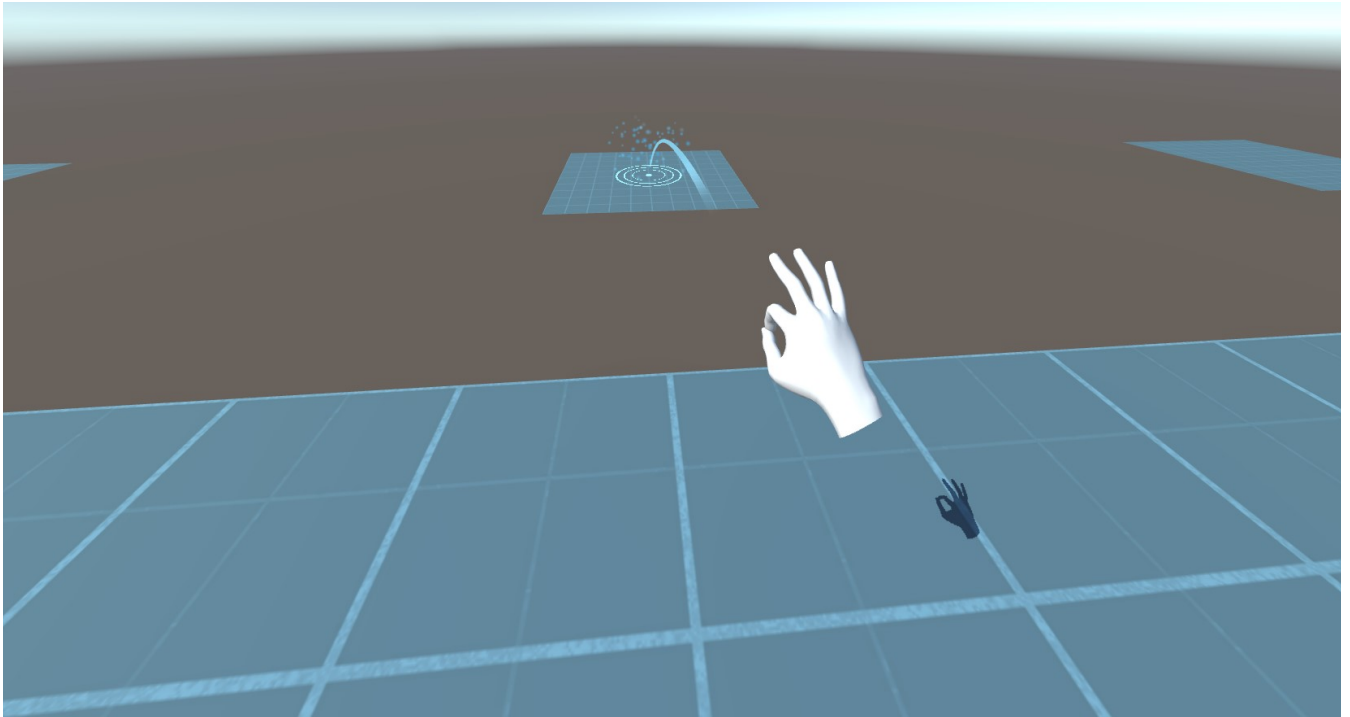


## Results



Teleport Area Example





Teleport Anchor Example

## Scripting

### LocomotionSystem

- Used for checking if player is grounded. If it's not, screen will fade black and *Footprints* will appear on the place player has left the ground
- Holds references to all objects essential for teleport and continuous locomotion to work
- Requires component of type `OVRCameraRig`

#### Public Properties

Transform Transform

- Cached transform component

Transform TeleportRayStartLeft

Transform TeleportRayStartRight

- TeleportRayStarts are created to help stabilize ray casting

Transform CenterEyeAnchor

ScreenFader ScreenFader

TeleportDestination OldTeleportDestination

- Used to keep track of where player was and invoking events

### Public Methods

Vector3 TryGetGroundNormal()

- Returns normal if the camera is above object containing Ground
- Returns Vector3.Zero if the camera is not above object containing Ground

## TeleportRayInteractor

- Used for finding TeleportDestination with raycasts

### Public Methods

TeleportDestination TryGetTeleportDestination(Vector3, Vector3, out bool)

TeleportDestination TryGetTeleportDestination(Vector3, Vector3, Vector3, RayVisuals, out bool)

## Teleport

- Abstract class. There are three concrete implementations for it. Pinch, Gaze and Gesture teleport but you can inherit from it and make your own
- Requires components of type *TeleportRayInteractor* and *HandTrackingDataProvider*

### Protected fields

LocomotionSystem locomotionSystem

GameObject teleportReticle

UnityEvent onTeleported

HandTrackingDataProvider handTrackingDataProvider

TeleportRayInteractor teleportRayInteractor

TeleportDestination teleportDestination

HandTrackingSkeletonData handTrackingSkeletonData

HandType handType  
RayVisuals rayVisuals

Transform teleportReticleTransform

bool isTeleportAllowed

### Protected methods

virtual void CheckForTeleportDestination()

- Override it and add your logic to finding teleport destinations

void TeleportPlayer()

- Teleports player to *TeleportDestination* location

void ResetTeleportVariables()

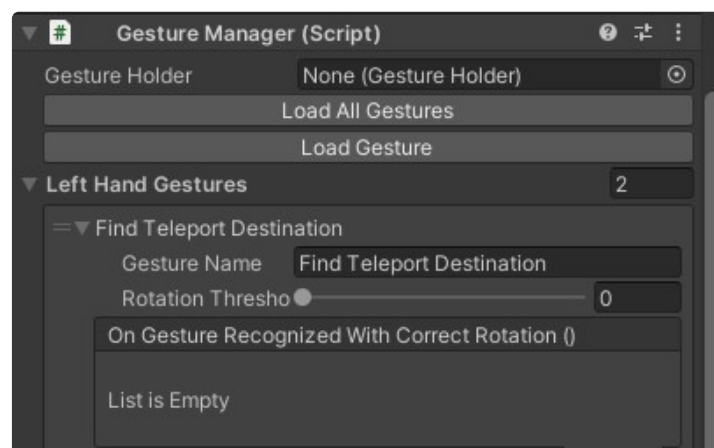
void ActivateTeleportReticle()

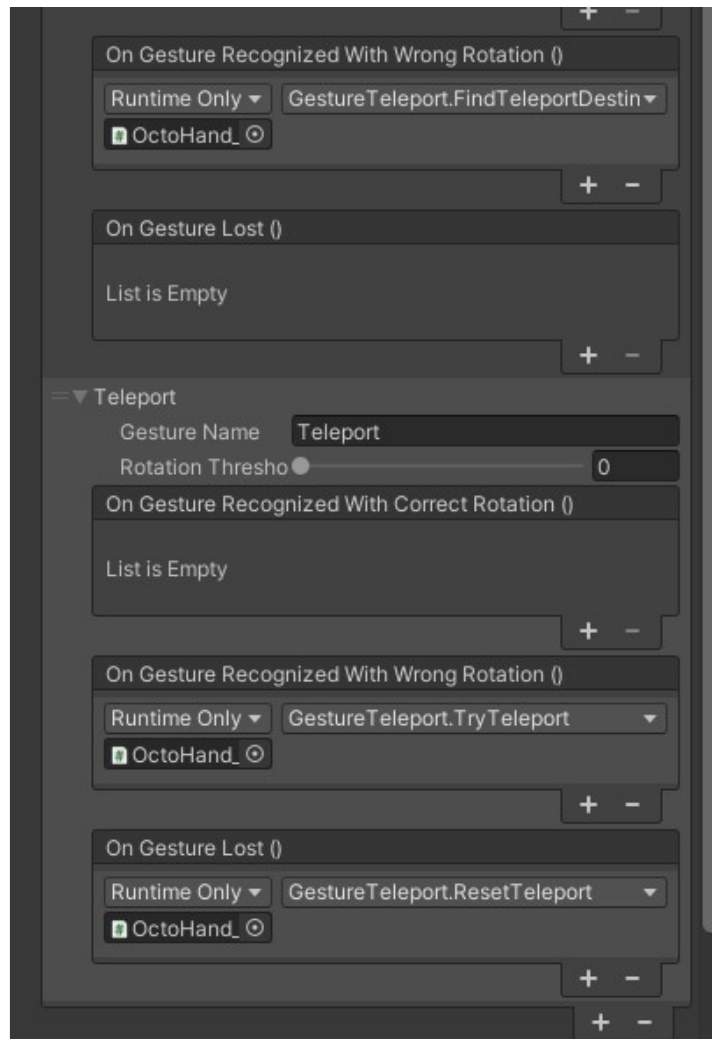
## PinchTeleport

- Finds teleport destination while pinching forward. If teleport destination is found, will do the blink teleport on pinch release
- Uses RayVisuals to draw the line

## GestureTeleport

- Gesture teleport is made for your own custom made gestures
- Recommend setting it up like in the following picture:





## Public Methods

void FindTeleportDestination()

void TryTeleport()

void ResetTeleport()

## GazeTeleport

- Finds teleport destination while pinching in the direction of players look. If teleport destination is found, will do the blink teleport on pinch release

## ContinuousLocomotion

- Moves continuously in the direction of looking while pinching
- Requires component of type *HandTrackingDataProvider*
- Can move only above objects containing *Ground* component

## TeleportDestination

- Abstract class. Has two concrete implementations: TeleportArea and TeleportAnchor

### Public Fields

bool CanTeleportHere

Vector3 HitNormal

UnityEvent OnTeleportedToDestination

UnityEvent OnTeleportedFromDestination

### Public Methods

Vector3 GetTeleportLocation()

virtual void SetTeleportLocation(RaycastHit)

## TeleportArea

- Teleport freely on objects containing this component
- Sets teleport location to raycastHit.point

## TeleportAnchor

- Teleport to a specific location
- Main difference between anchor and area is that if you provide anchor with custom teleport location, it will teleport the player there. Else it will teleport the player to raycastHit.transform.position

# Kinematic Interactions

## How To Use

This is a quick start guide on how to rapidly create kinematic interactions. Each title leads to the page guiding you through the process.



If you're just starting out, you might want to head over to OctoXR > Samples > KinematicInteractions and check out the scene provided to see a simple example of some stuff you can do with this module!

Use this page as a quick reference in case you need to quick refresher.

## Interactors

Learn how to create hands that interact with grabbable objects quickly by using the kinematic rig.

## Grabbable Objects

Learn how to quickly and easily create grabbable objects.

## Grab Points / Hand Posing

Learn how to manipulate grab points and create custom hand poses.

## Snap zone

Learn how to create snap zones.

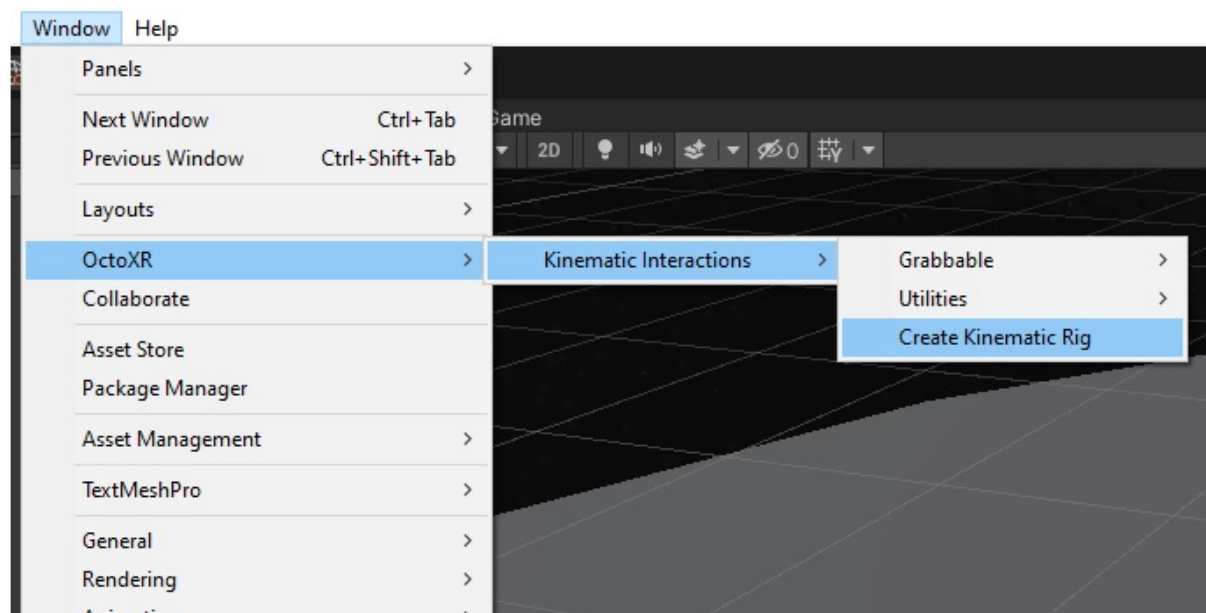
## Scripting

A deeper dive into the core components and useful utilities of the kinematic interactions module

# Grab Controllers / Kinematic Rig

## Setting Up Hands

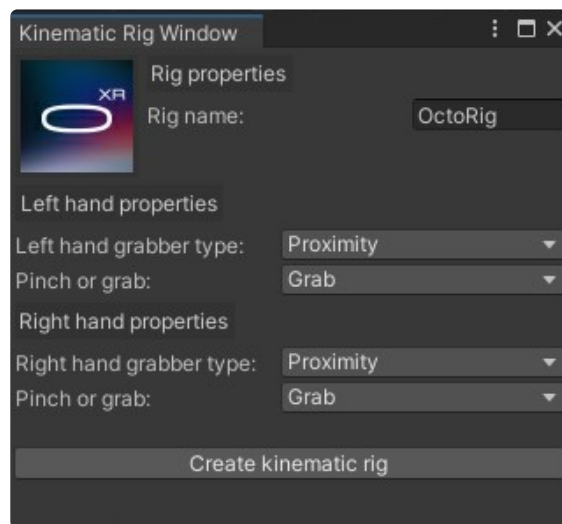
The easiest way to set up your kinematic hands is to right click on the hierarchy or by clicking the window tab in the toolbar and selecting OctoXR > Kinematic Interactions > Create Kinematic Rig.





Creating rig from toolbar

Creating this rig will ensure that you have met all the necessary prerequisites for different supported types of kinematic grabbing.



Kinematic rig window

Once you open the rig creator, a window like this one will pop up.

**Hand grabber type:** lets you select between proximity (close range), distance grabbing or both options together.

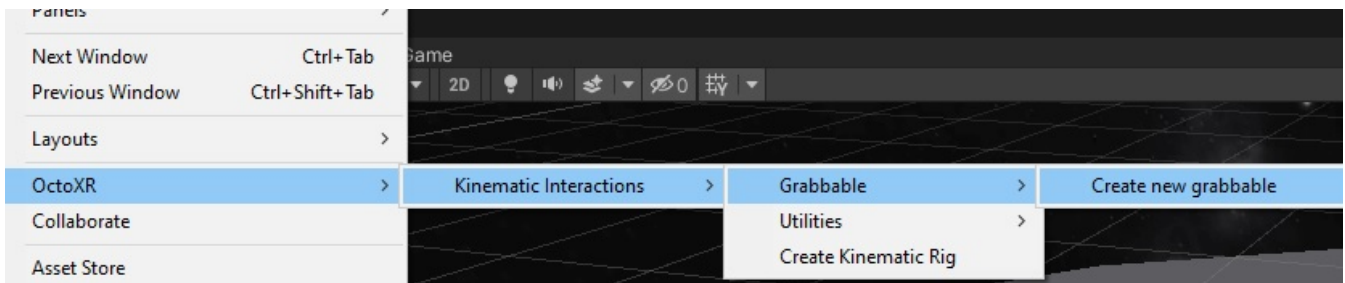
**Pinch or grab:** very self-explanatory, it lets you choose between a grabbing or pinching gesture in order to trigger interactions.

**i** While having both types of interactors on one grabber type works - it's not recommended as it might produce some wonky results

## Grabbable Objects

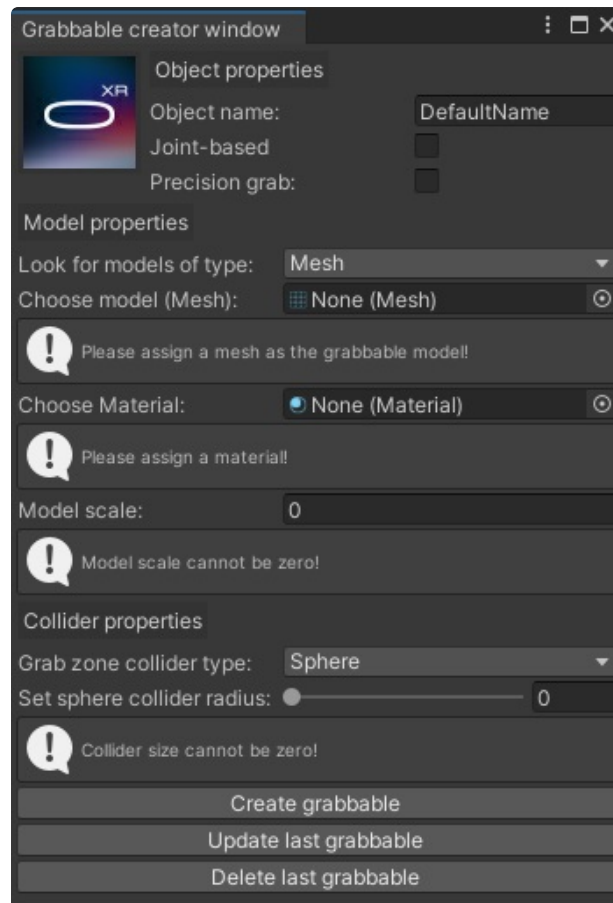
### Setting Up Grabbable Objects

Much like the interactors, grabbable objects can easily be set up by right clicking the hierarchy window or by clicking the window tab in the toolbar and selecting OctoXR > Kinematic Interactions > Grabbable > Create new grabbable.



Creating a grabbable from the toolbar

Creating grabbable objects this way ensures that you have all the necessary prerequisites for the object to be grabbable.



Grabbable creator window

Once you open the grabbable creator, a window like this will pop up.

**Joint-based:** Leaving joint-based unticked will make the object ignore collision with the world while interacting with it. Ticking the option will allow the object to collide with the world.

**Precision-grab:** Leaving the precision grab option unticked will snap the object to your hand while grabbing. Ticking the option will allow you to manipulate the object exactly at the point where you're grabbing, but will disable hand posing.

**Look for models of type:** allows you to choose between meshes and prefabs while picking objects to represent your grabbable.

**colliders** attached which might mess with your grabbable, always double check your models after creating the object.

If you choose to import a mesh as the model for your object, you will be able to set its material with the **Choose material** option.

**Model scale:** sets the scale of your model (**NOT THE OBJECT** - the object scale should **always** be **1:1:1**).

**Grab zone collider type:** creates the designated trigger collider around your object.

## Grab Points / Hand Posing

This is a step-by-step guide on setting up your grab points

### Setting up grab points

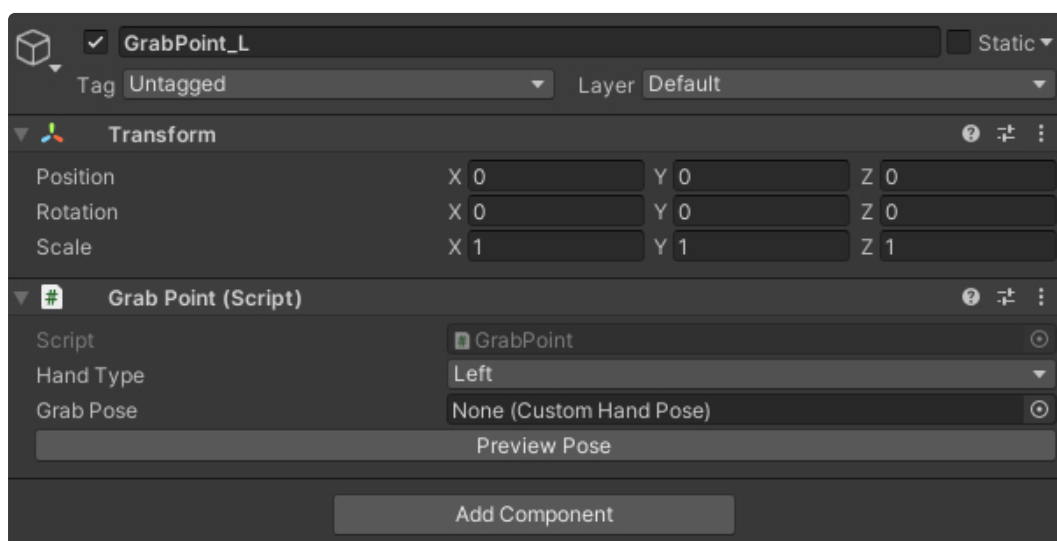
After creating a grabbable object, your objects hierarchy should look something like this:



Grabbable hierarchy with grab points

As you might've noticed, the grabbable has kindly provided us with two grab points. Open one of them and let's see what's up.

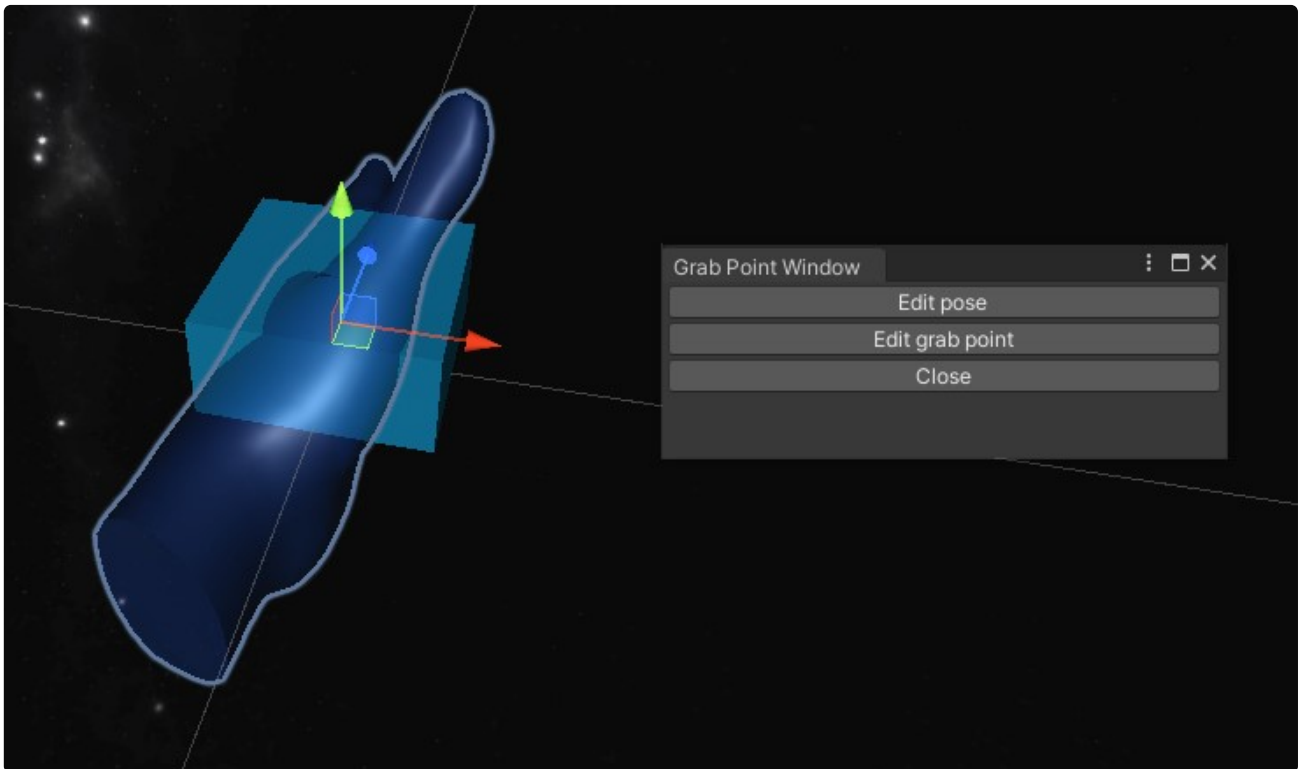
**i** Grab points are still required for grabbable detection even if your object is a precision grabbable!



Grab point in the inspector

Clicking on preview pose will open a new window, instantiate a preview hand and focus on our grab point.

- i** Note that if you've already created a grab pose, you can always assign it through the inspector under Grab Pose.



Grab point, preview hand and grab point window

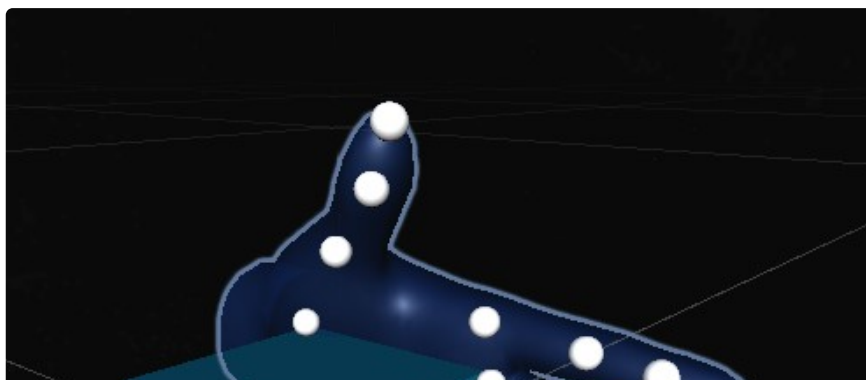
Simply adjust the position and rotation of the grab point by manipulating its transform in the editor or using the scene positional and rotational gizmos.

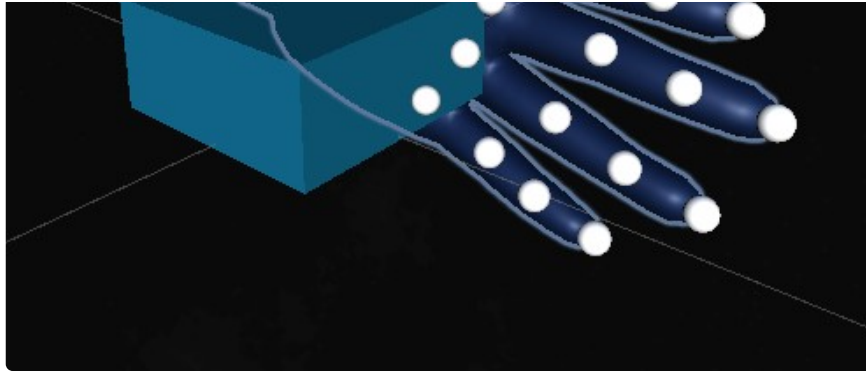
This is enough if you want to create an object with just grab points and no hand posing.

- i** Be careful not to move or rotate the actual preview hand, as that will achieve nothing. Always move and rotate the grab points.

## Creating hand poses

If you want to take this a step further click on edit pose. This will focus in on the preview hand and allow you to create custom poses your hand will mimic once you grab an object.

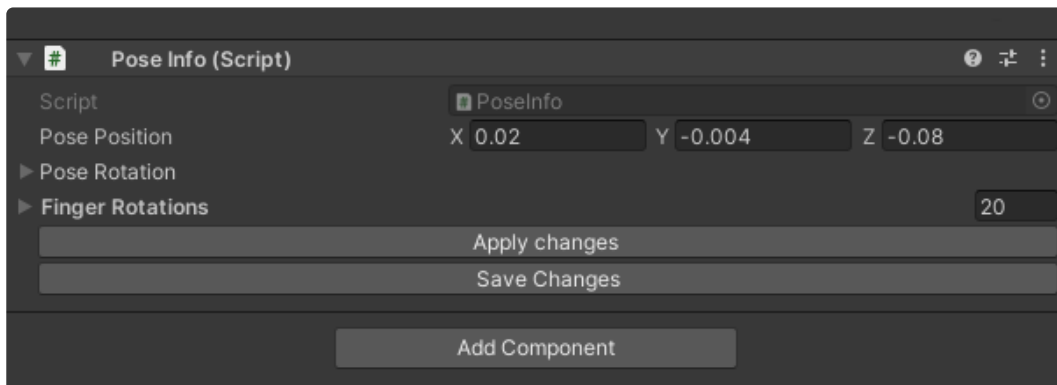




Manipulating the preview hand joints

- What you see in the image is the default pose our preview hand appears in. If you have already created and assigned the pose to the grab point, the preview hand will load that pose instead.

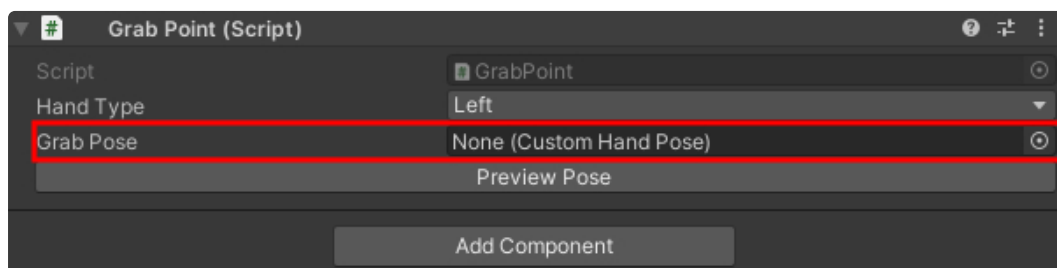
Once you are done posing the fingers of the hand to your liking, look over to the inspector, you will find the pose info component.



Pose info component

Apply and save changes. The editor will prompt you to choose your file location and name. After doing so, you can close the grab point window.

- If you are happy with your pose, always apply changes before saving. If you do not apply the changes, the pose will have nothing new to save.



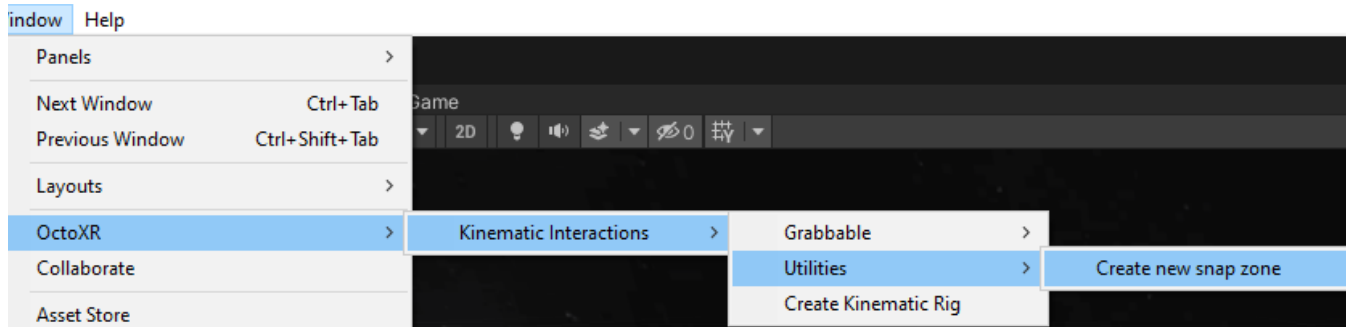
Grab point component with marked grab pose

Your newly created grab point should now be assigned to the grab pose field of the grab point component!

# Snap Zone

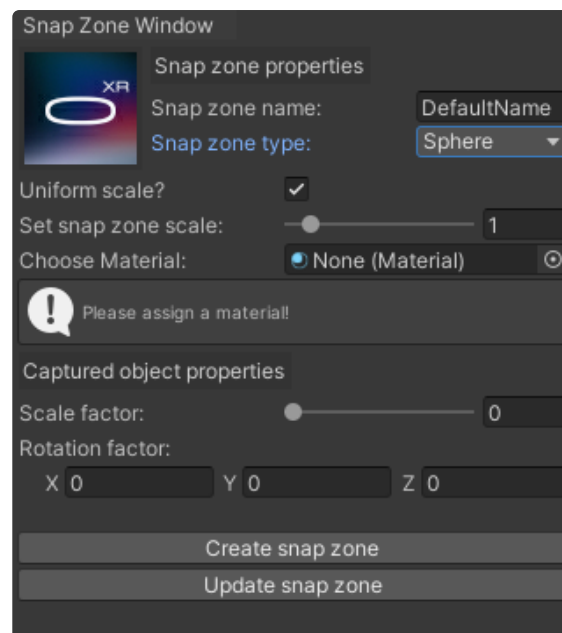
## Setting up a snap zone

Creating a snap zone is again, fairly straight forward. Use the option provided for you in the toolbar or access it by right clicking the hierarchy. This will provide you with an option to create basic spherical or box snap zones.



Creating a snap zone from the toolbar

Creating snap zones this way ensures that you have all the necessary prerequisites for the object to be grabbable.



Snap zone window

Once you open the snap zone creator, a window like this will pop up.

## Snap zone properties

**Snap zone type:** lets you choose between a sphere and a box for the shape of your snap zone.

**Uniform Scale:** ticking this will scale the snap zone uniformly on all axes, ticking it off lets you scale it differently on each axis.

[Set snap zone scale:](#) changes the scale of the snap zone

[Choose Material:](#) lets you choose the material for your snap zone.

### Captured object properties

[Scale factor:](#) uniformly scales your object when it is snapped (leaving it at 0 will not scale it).

[Rotation factor:](#) sets the rotation of your object on each axis (leaving it at 0 will not rotate it).

## Scripting

Deeper dive into the inner workings of kinematic interactions

## Core

Core components of the kinematic interactions module

## InteractionHand

### Description

- Core component for hands used in kinematic interactions
- Contains grab and pinch detection
- Mainly used for kinematic interactions, but It's logic can easily be used in other scenarios
- One important thing to remember is that you should have one for each of the hands you want to do interaction with. This component uses vital hand data important for the functionality of any variant of a [GrabController](#)

### Public properties

[HandSkeleton](#) HandSkeleton

- Reference to the hand skeleton, necessary in order for the script to get the transform of each individual fingertip

[HandType](#) HandSkeletonType

- Enum responsible for setting the type of the current interaction hand (left or right). It automatically gathers this data from the provided hand skeleton in the Awake() method
- Publicly accessible because It's used for hand type detection in multiple interaction scripts



[HandTrackingDataProvider](#) [HandDataProvider](#)

- Reference to the hand data provider, necessary to check for pinching

[bool](#) ShouldGrab

- Publicly accessible property which returns true when the average distance between the collected fingertips and hands goes below a certain threshold

[bool](#) IsGrabbing

- Publicly accessible property which returns true only when the hand has actually grabbed an object
- Note that this is not changed in [InteractionHand](#), but instead in the [HandleGrab\(\)](#) method of class [GrabController\(\)](#)

[bool](#) IsPinch

- If true, stops using the method to detect grabbing and instead uses the one used to detect pinching with a certain finger

## Public methods

[void](#) DetectGrab()

- Calculates the distance between finger tips and wrist and returns true on [ShouldGrab](#) if the distance is below or equal to a certain threshold

[bool](#) DetectPinch([HandFinger](#) finger)

- Gathers data from the hand data provider and returns true on [ShouldGrab](#) if the provided finger is pinching

# GrabController

## Description

- Core component for grab types
- Checks conditions for grabbing and releasing an object
- Gathers information about grabbable objects and grab points in reach
- Inherited by [ProximityGrabController](#) and [DistanceGrabController](#)

## Public properties

[InteractionHand](#) [InteractionHand](#)

- Reference to the appropriate interaction hand, the before mentioned script which houses grab detection logic

#### [GrabPointController](#) GrabPointController

- Reference to the grab point controller which is necessary for snapping kinematic grabbables (not joint based-grabbables, their snapping functions differently)

#### [List<Grabbable>](#) ObjectsInReach

- A list of objects in reach which the GrabPointController populates

#### [List<GrabPoint>](#) GrabPointsInReach

- A list of grab points in reach which the GrabPointController populates

#### [Grabbable](#) GrabbedObject

- The object that the hand is grabbing will be referenced here

#### [Grabbable](#) ClosestObject

- The closest object in reach of the trigger of the proximity grab controller or in the path of the ray casted by the distance grab controller

#### [Grabbable](#) LastClosestObject

- If the closest grabbable is no longer in reach of the proximity or distance grab controller it is referenced here
- This reference is used to trigger OnUnhover event available in the grabbable component

### Public methods

#### [void](#) GrabCheck()

- Used in the Update() method to check whether all of the conditions for triggering HandleGrab() or HandleRelease() were met

#### [void](#) HandleGrab()

- Uses the Attach() method available in grabbable objects to handle object grabbing
- Invokes OnGrab event

#### [void](#) HandleRelease()

- Uses the Detach() method available in grabbable objects to handle object release
-

Invokes OnRelease event

## Unity events

[UnityEvent](#) OnGrab

[UnityEvent](#) OnRelease

# ProximityGrabController

## Description

- Uses inherited logic from GrabController in conjunction with Unity's OnTrigger events to find grab points and grabbable objects in reach and add them to the list of grab candidates using logic available in GrabPointController.
- Finding a grab candidate triggers OnHover event available in the grabbable component

# DistanceGrabController

## Description

- Uses inherited logic from GrabController in conjunction with raycasting and spherecasting to find the closest grabbable which the ray source is pointing at and assigns it as the interaction candidate.
- Finding a grab candidate triggers OnHover event available in the grabbable component

# GrabPointController

## Description

- Moves a kinematic grabbable object to match the position and rotation of the grab controller transform
- Controls adding and removing grab points to a list of grab points in reach available in the grab controller

## Public methods

[void](#) MoveToGrabPoint([Grabbable](#) connectedObject, [GrabPoint](#) grabPoint, [float](#) maxDelta)

- Adjusts the position and rotation of the given grabbable to the position and rotation of the given grab point

`void AddGrabPoint(Collider grabPointCollider)`

- Adds a grab point to the array of grab points within reach

`void RemoveGrabPoint(Collider grabPointCollider)`

- Removes all grab points from the array of grab points within reach

`void AddGrabbableObject(Collider grabbableCollider)`

- Adds a grabbable object to an array of grabbable objects within reach

`void RemoveGrabbableObject(Collider grabbableCollider)`

- Removes all grab points from the array of grab points within reach

## DistanceCheck

### Description

- Abstract class used by the grab point controller to check for the closest grabbable and/or grab point

### Public methods

`void CheckGrabbableObjectDistance(GrabController grabController)`

- Checks distance between the given grab controller and all grabbable objects in reach and assigns the current closest one to the grab controller

`void CheckGrabPointDistance(GrabController grabController)`

- Checks distance between the given grab controller and all grab points in reach and assigns the grab point's parent, the grabbable object whose grab point it is, to the grab controller

## Grabbable

### Description

### Public properties

`bool IsPrecisionGrab`

- Determines whether or not the grabbable will snap to its grab controller's transform

**bool** IsRespawnable

- Determines whether the object will respawn to its original position once it collides with the given object

**float** MaxDelta

- Speed at which the kinematic grabbable should move towards the hand

**Transform** GrabPointRight

- Reference to the right grab point of this object

**Transform** GrabPointLeft

- Reference to the left grab point of this object

**GrabController** CurrentGrabber

- Current grab controller in control of this object

**bool** IsGrabbed

- Signifies that the object is currently being grabbed by a grab controller, preventing other grab controllers to take control of it

**Rigidbody** Rigidbody

**Vector3** InitialLocalScale

## Public methods

**virtual void** Attach(**Transform** grabParent)

- Attaches the object to the grab controller
- Starts estimating velocity
- Assigns the grab controller as the current grabber
- Invokes OnAttach event

**virtual void** Detach()

- Detaches the object from the grab controller
- Stops estimating velocity
- Removes current grabber
- Invokes OnDetach event

## Unity events

[UnityEvent](#) OnHover

[UnityEvent](#) OnUnhover

[UnityEvent](#) OnAttach

[UnityEvent](#) OnHeld

[UnityEvent](#) OnDetach

## KinematicGrabbable

### Description

- Inherits logic from the grabbable script and overrides Attach/Detach scripts
- Parents itself to the transform of the grab controller and resets It's rotation (in case its been modified previously by a joint-based grabbable)

## JointBasedGrabbable

### Description

- Inherits logic from the grabbable script and overrides Attach/Detach scripts
- Creates a joint between the object and the grab controller
- Has It's own snapping logic separate from the one used in GrabPointController to snap kinematic grabbable objects

### Public methods

void SnapPositionAndRotation(GrabController grabController)

- Used in case you want to snap a joint-grabbable object

## Grab Point

### Description

- Grab point of the object you create, contains and updates data about positional and rotational offset of

It's parent object - the grabbable object

- Used to store a hand pose for the grab controller to look for
- Has a helper script to instantiate a preview pose when manipulating grab points and creating hand poses

## Public properties

**HandType** HandType

- Enum responsible for setting the type of the interaction hand (left or right) this object should respond to
- This information must be set up manually

**CustomHandPose** GrabPose

- Grab pose - scriptable that will be applied to the interacting grab controller's poseable mesh once this object is grabbed
- Contains information about the rotation of individual bones under the root of each finger, as well as the positional and rotational data of the pose itself

**Grabbable** ParentGrabbable

- The grabbable object acting as a parent to this grab point

**Vector3** Offset

- Positional offset used by GrabPointController when snapping the object's position

**Quaternion** RotationOffset

- Rotational offset used by GrabPointController when snapping the object's rotation

**bool** isGrabbed

- True when the object is grabbed

**GameObject** InstantiatedPreviewHand

- Reference used by the method for instantiating preview hand poses
- This reference is necessary in order to possibly shuffle through multiple grab poses without compromising the actual asset or scriptable object

## Public methods

**void** UpdatePositionOffset()

- Multiplies the inverse of the grabbable object's rotation with the sum of the negative position of the

grabbable and the position of the grab point to calculate positional offset

`void UpdateRotationOffset()`

- Multiplies the rotation of the grabbable object's rotation with the grab points rotation in order to calculate the rotational offset

`void InstantiateHandPose()`

- Instantiates a preview hand prefab from the resource folder and offsets it to the palm of the hand
- This method is used by the editor script for hand posing to give you a visual representation of the pose for easier grab point positioning and hand posing

## VelocityEstimator

### Description

- Used to estimate release velocities of grabbable objects, without it, the objects would behave as if no external forces are impacting them (excluding the non-precision joint grabbable) on throw

### Public methods

`void StartVelocityEstimation()`

- Ends any previous velocity estimation
- Starts a coroutine which calculates linear and angular velocities

`void FinishVelocityEstimation()`

- Stops the coroutine - called in the previously mentioned method

`Vector3 GetLinearVelocityAverage()`

- Gets the linear velocity average or the average rate at which the object should be moved according to the velocity calculation coroutine started upon grabbing it

#### ✓ Linear velocity calculation

Calculated by simply multiplying the given velocity factor with the current position of the object subtracted by its previous position

`Vector3 GetAngularVelocityAverage()`



- Gets the angular velocity average or the average rate at which the object should be rotated according to the velocity calculation coroutine started upon grabbing it

#### ✓ Angular velocity calculation

A bit more complex than linear velocity but it basically boils down to these steps:

- First, the delta rotation is calculated, which takes into account the multiplication of the current rotation of the object and the inverse of its previous rotation
- Next, the theta is calculated by multiplying the cosine arc of a clamped delta rotation of  $w(\text{rotation around the vector})$  value by two
- The angular velocity is then stored as a vector3 of delta rotation on each axis and in the case of its square magnitude being above zero, multiplied by the set velocity factor and a normalized value of the angular velocity
- Finally, the angular velocity is stored in an array of angular velocity samples later used by `GetAngularVelocityAverage` to calculate the rotation average

## HandPosing

### CustomHandPose

#### Description

- Scriptable object used to store info about newly created hand poses

#### Public properties

[Vector3](#) PosePosition

[Quaternion](#) PoseRotation

[List<Quaternion>](#) FingerRotations

#### Public methods

[void](#) SetHandPoseData([PoseData](#) poseData)

- Used to save the pose data, namely the pose position, rotation and finger rotations to the public

properties of the scriptable object for later use

## PoseableHand

### Description

- Abstract class which runtime and preview hands inherit most of the logic from
- Collects data about joint transforms from finger roots
- Uses that joint data to collect rotation data from individual joints
- Applies the pose and finger rotations

### Protected properties

[HandType](#) HandType

[List<Transform>](#) FingerRoots

[CustomHandPose](#) CurrentHandPose

### Public properties

[List<Transform>](#) Joints

### Protected methods

[List<Transform>](#) CollectJoints()

- Finds all available transforms under each finger root

### Public methods

[List<Quaternion>](#) GetJointRotations()

- Gets rotations of each joint from the list of collected joints

[void](#) ApplyPose([CustomHandPose](#) customHandPose)

- Applies the pose data available on this component to the given custom hand pose
- Used in both the preview and runtime hands to apply poses when necessary

[void](#) ApplyFingerRotations([List<Quaternion>](#) rotations)

- Applies the rotation data of every joint to the rotation of the given custom hand pose

# PreviewHand

## Description

- When enabled, collects joints
- Looks for any available grab poses in the grab point it is assigned to
- If a grab pose is found, applies it

# RuntimeHand

## Description

- When enabled, adds an event listener to the OnGrab and OnRelease events of the grab controller currently interacting with it
- Detects whether the left or the right hand should be posed upon interaction and applies the pose accordingly
- What basically happens here is that, upon interaction, the original mesh is disabled, and the mesh with the applied pose is enabled

# PoseData

## Description

- Used to save transform and finger rotation data of the current preview hand while the hand is being posed so you can save it as a scriptable object later
- Also used to get pose info from the targeted custom hand pose

## Public properties

[Vector3](#) PosePosition

[Quaternion](#) PoseRotation

[List<Quaternion>](#) FingerRotations

## Public methods

void Save([PreviewHand](#) hand)

- Saves the pose data from the current preview hand to this component
- This method is used from the editor (apply button) to temporarily store data of a preview hand which is currently being posed before actually saving the pose to your project folder

void GetPoseData([CustomHandPose](#) customHandPose)

- Gets the transform and finger rotation from the given pose and applies the data to this component
- This method is used in runtime to actually apply the hand pose when an interaction is achieved

## PoseDataRuntime

### Description

- Used in the runtime posable hand
- Basically does the same thing as pose data, but is used at runtime
- Hides editor functionality to avoid bloat in the scene's inspector

## Utilities

Optional useful utilities used in kinematic interactions

## Interaction indicator

### Description

- Controls the color and scale of an object
- Used for interaction indicator objects, like a mesh/sprite renderer that would indicate where to put your hand
- Note that this is an extensible abstract class, meant to be inherited from for the shared properties and methods

### Public properties

[Color32](#) InteractionColor

- Color that you want the object to change to

[float](#) ScaleFactor

- How much you want to scale the object

#### Color StartingColor

- Color that the object's renderer will be reverted to
- Automatically set in Start() methods of inheritor classes

#### Vector3 StartingScale

- Scale that the object's transform will be reverted to
- Automatically set in Start() methods of inheritor classes

#### bool HasEntered

- True if the scale and color have been changed, false if they've been reverted

### Public methods

#### virtual void ChangeColor()

- Scales the transform of the object uniformly according to the scale factor on each axis
- Changes the color through the methods in inheritor classes

#### virtual void RevertColor()

- Reverts scale and color

## MeshInteractionIndicator

### Description

- Overrides the base color changing and reverting methods to change the color of a referenced mesh renderer

## SpriteInteractionIndicator

### Description

- Overrides the base color changing and reverting methods to change the color of a referenced sprite renderer

# FollowObject

## Description

- Can be used to make an object follow another object's rotation and position
- Optionally you can set a positional offset of the follower object
- There is also an option to set up a line renderer which will be drawn from the follower object to the object that is being followed, the starting point of the curve can also be offset
- This script can be constantly updated or tick only once
- If the update is constant, the object will follow its rotation/position parent the whole time
- If the update is not constant, the object will stay at the place where the parent was at (plus offset) in the moment it was enabled

# ShiftFocusToParent

## Description

- Simple and useful script you can drag and drop to an object so when you click it in the scene view, you select its parent instead
- This is useful, for example, when an object has a model as a child of it in the hierarchy and you want to avoid selecting the model all the time

# ResetObject

## Description

- Resets an object back to its original position after a set amount of time
- The information about the original position of the object as well as if it's respawnable is gathered from the grabbable object itself

## Unity events

[UnityEvent](#) OnReset

# HoldToActivate

## Description

- Detects when this object is overlapping with a set collider to trigger events
- Useful for non-raycast UI interactions when you want to prevent the user from accidentally clicking a button and triggering its functionality randomly
- Sound and visual triggers for these interactions can also be easily set up here

## Unity events

- [UnityEvent](#) OnHoldStart
- [UnityEvent](#) OnHoldCompleted
- [UnityEvent](#) OnHoldCancelled

# PrefabSpawner

## Description

- Spawns a referenced prefab at the chosen position with the chosen parent and destroys it after set time if its value is not set to zero, if it is zero, the prefab won't be destroyed
- Useful for spawning particle prefabs

## Public methods

[void](#) SpawnPrefabAtTransform([GameObject](#) gameObject, [Transform](#) spawnPosition, [Transform](#) parent, [float](#) destroyTime)

- Spawns at the transform of a certain object

[void](#) SpawnPrefabAtVector([GameObject](#) prefab, [Vector3](#) spawnPosition, [Transform](#) parent, [float](#) destroyTime)

- Spawns at a certain vector3

# Physics (Experimental)

## How To Use

This is a quick start guide on how to rapidly create physics interactions. Note that this whole section describes functionality that is considered experimental and is subject to constant changes

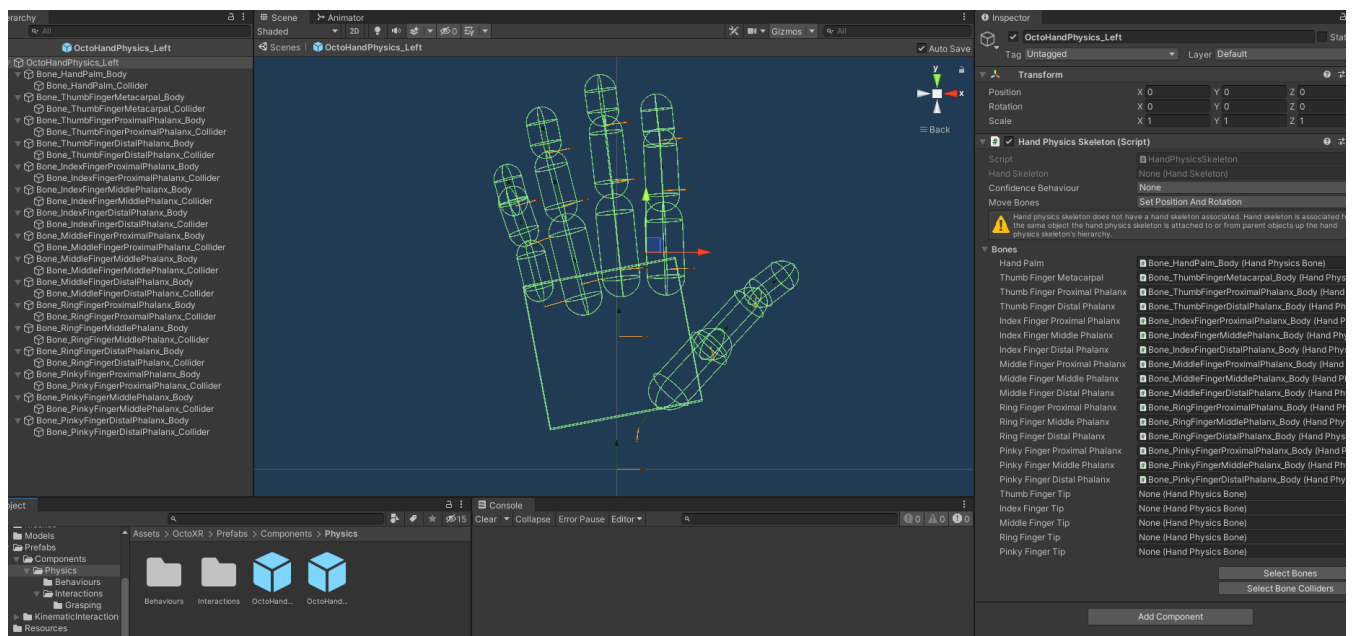
## Hand Physics Skeleton

Learn how to quickly setup physics enabled hand skeleton and important information about it

# Hand Physics Skeleton

## Setting up

Assuming you have a [HandSkeleton](#) already set up in your scene, the easiest way to add HandPhysicsSkeleton is to drag and drop it under the HandSkeleton object from a prefab located in Prefabs\Components\Physics, either OctoHandPhysics\_Left for the left hand or OctoHandPhysics\_Right for the right hand. Both prefabs have default settings and colliders set up and ready.



Prefab for the left hand physics skeleton

HandPhysicsSkeleton requires a HandSkeleton in the same GameObject it is attached to or parent objects up the physics skeleton's hierarchy. Physics skeleton uses the HandSkeleton to read the target poses for its bones. The relation between HandPhysicsSkeleton and its HandSkeleton is similar to that between a HandSkeleton and its HandTrackingDataProvider - one provides bone poses for the other.

## Setting up from scratch

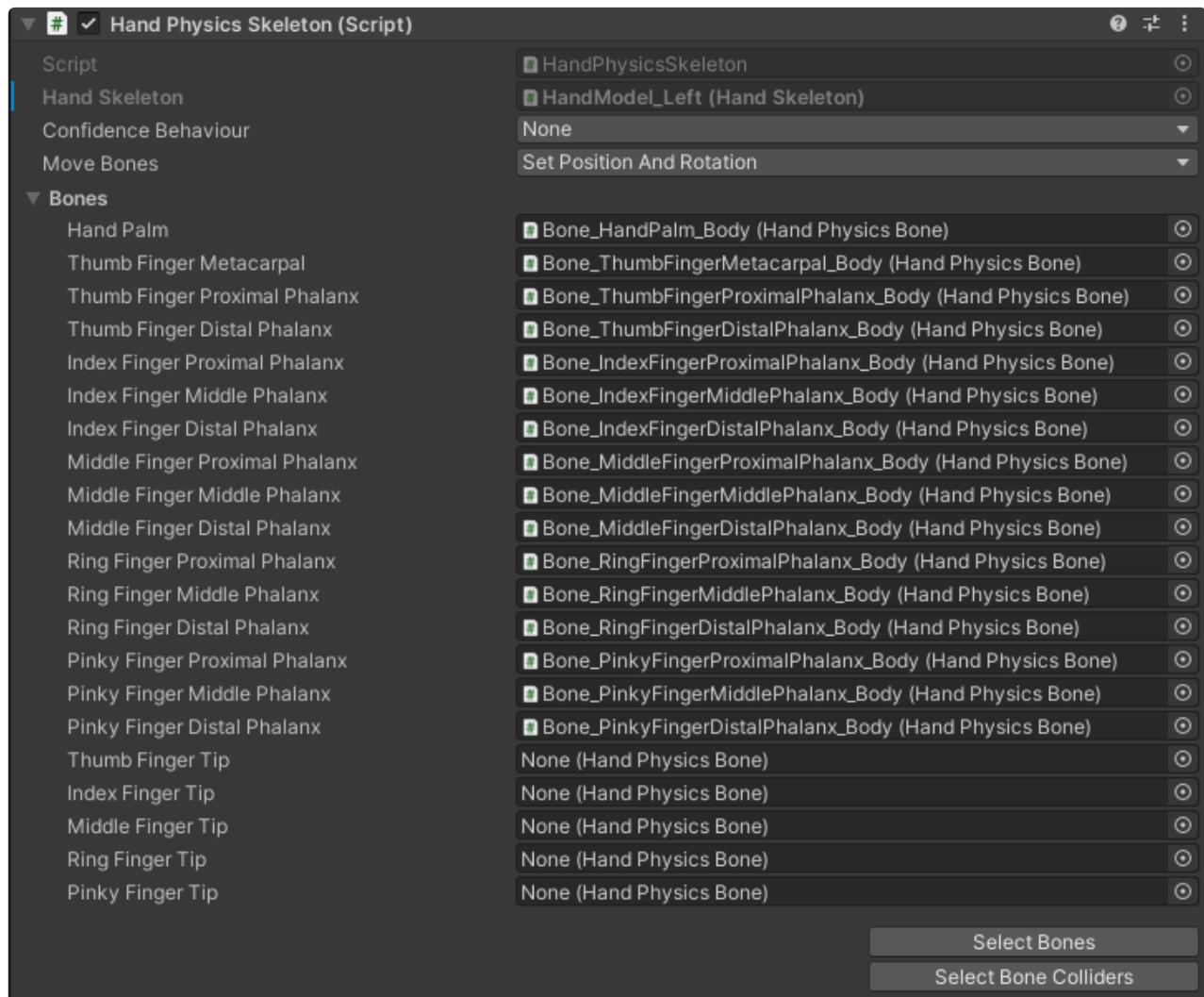
To set up a physics skeleton from start:

- Add HandPhysicsSkeleton script to the GameObject that has HandSkeleton attached or to any of the HandSkeleton's child GameObjects
- Add HandPhysicsBone script to all objects that you want to act as physics skeleton's bones
- Add target HandPhysicsBones to the HandPhysicsSkeleton by dragging and dropping them to the HandPhysicsSkeleton's list of bones in its inspector



HandPhysicsBone represents a bone of the HandPhysicsSkeleton, it has a similar relation with the HandPhysicsSkeleton as a HandBone has with its HandSkeleton. Unlike HandBone, HandPhysicsBone does not have any requirements pertaining to the hierarchy state of the GameObject it is attached to, but it needs to be manually added to a HandPhysicsSkeleton. Note that it does not matter how many bones does a HandPhysicsSkeleton have, it isn't required to have all possible hand bones at all.

## Properties



Inspector for the HandPhysicsSkeleton

Property	Description
Confidence Behaviour	<p>Specifies how to handle situations where hand tracking data is low confidence or there is no hand tracking at all. Possible values are:</p> <ul style="list-style-type: none"> <li>None - no specific action is taken</li> <li>Toggle Physics Bones - GameObjects to which physics skeleton's bones are attached will be deactivated when tracking data is low confidence and re-activated when it is not</li> </ul>

	<ul style="list-style-type: none"> <li>• Toggle Bone Collision Detection - Rigid bodies attached to the physics skeleton's bones will have its collision detection disabled and re-enabled depending on the tracking data confidence.</li> </ul>
Move Bones	<p>Provides certain options that define how will the HandPhysicsSkeleton move its bones to target poses. Possible values are:</p> <ul style="list-style-type: none"> <li>• Set Position And Rotation - physics skeleton's bones will be moved to target poses by directly setting positions and rotations of the rigid bodies attached to them</li> <li>• Set Velocity And Angular Velocity - physics skeleton's bones will be moved to target pose by setting linear and angular velocities of the rigid bodies attached to them</li> </ul> <p>Note that bones whose rigid bodies are kinematic will always be moved by setting their positions and rotations, this property does not affect them at all.</p>
Bones	<p>List of HandPhysicsSkeleton's bones. Bones can be added by assigning them to the target references.</p>

## Hand Physics Bone

HandPhysicsBone represents a hand bone of the HandPhysicsSkeleton the physics bone is associated with. When you want to add a physics bone to physics skeleton you need to attach HandPhysicsBone script to the GameObject you want to act as a bone of the HandPhysicsSkeleton and then assign that object in the physics skeleton's Inspector window to the target reference.

HandPhysicsBone requires Rigidbody and ConfigurableJoint components. You can set properties of the physics bone's rigid body as you see fit. The joint is used to connect the bone to its parent bone in the physics skeleton. If the bone does not have a parent in its physics skeleton or it has a parent, but it is not the bone's immediate parent in the standard hand bone hierarchy then the bone will not be connected to its parent via joint. Joints that connect two bones have their linear motion locked and angular motion limited by default. Angular motion is set to Limited to allow you to set angular limits between bones via the joints that connect them.

Note that bones that have their rigid bodies set as kinematic will still be connected via joints, although the joints will have no effect on them. Collisions between connected bones are disabled in their joints by

default. Regardless, you should probably put all the bones attached to the same physics skeleton in the special layer that is configured to not allow collisions with other objects in it.

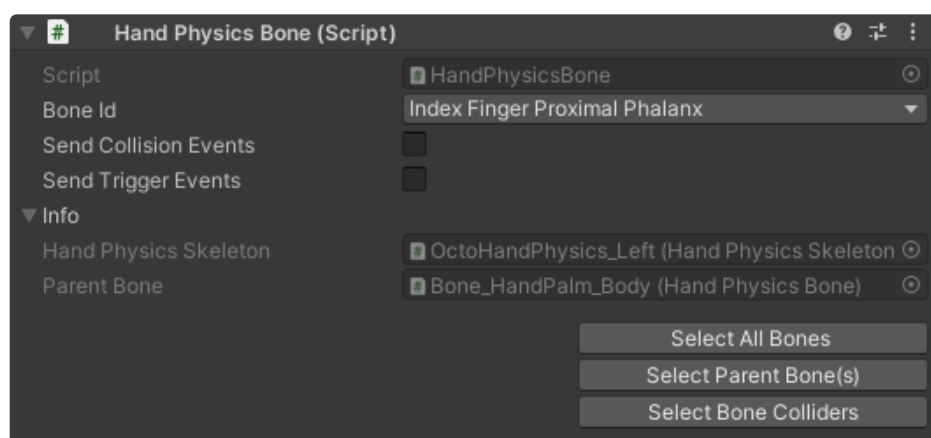
There may be scenarios where you want to have multiple physics skeletons associated with the same source hand skeleton. Having collisions between the bones in those physics skeletons is probably going to produce unwanted behaviour so you should probably always set all physics bones associated with the same source hand skeleton in a layer that prevents collisions between them, or prevent collisions between them in some other way.

You are free to add any kind of colliders to the hand physics bone, compound colliders are fully supported as well.

One more thing to note is that physics bones can be removed from a physics skeleton. To remove a bone from the skeleton you can either assign null (None) reference in its current place in the Inspector of the bone's HandPhysicsSkeleton or you can also reset the HandPhysicsBone in its Inspector - clicking three dots in the upper right corner of the bone's Inspector window and then clicking on Reset from the opened menu. In either case, HandPhysicsBone will be removed from its HandPhysicsSkeleton. After that you can re-add it if you wish, or add it to some other HandPhysicsSkeleton. In addition, the bone can be implicitly removed from its physics skeleton by dropping another bone on its place in the HandPhysicsSkeleton's Inspector.

## Properties

HandPhysicsBone defines a number of properties that can be edited and some read-only properties that provide certain information that may be of interest.



Inspector window of the HandPhysicsBone

Property	Description
Bone Id	Identity of the hand bone the physics bone represents. This property can be changed as long as the physics bone is not in a HandPhysicsSkeleton, if you try to change it while the bone is in one you will get an error generated and bone's identity will not change. Note that this property does not matter at all when assigning the bone to a physics skeleton in the

	editor, the identity will be assigned based on which exact place you drop the bone in the HandPhysicsSkeleton's Inspector window.
Send Collision Events	Determines whether the hand physics bone will send collision events. Events sent can be handled by IHandCollisionEnterHandler, IHandCollisionStayHandler and IHandCollisionExitHandler interfaces. Events are always sent to the object the hand physics bone is attached to, the hand physics skeleton the hand physics bone is attached to and its parent objects up hierarchy and also the object the hand physics bone is colliding with and its parents upwards. Sending event up a hierarchy is stopped if event is used by a handler in that hierarchy. Note that even sending will not work for a hand physics bone not added to a physics skeleton.
Send Trigger Events	This is basically the same as Send Collision Events, only this works with trigger callbacks. Event handlers for this are IHandTriggerEnterHandler, IHandTriggerStayHandler and IHandTriggerExitHandler interfaces.
Hand Physics Skeleton	Read-only property that tells which hand physics skeleton this bone currently belongs to. Obviously this will be null for bones not added to a HandPhysicsSkeleton.
Parent Bone	Current parent bone of the physics bone that is in the same physics skeleton. This does not need to be bone's natural parent bone. For example, you can have IndexFingerMiddlePhalanx and HandPalm bones in the skeleton, but no IndexFingerProximalPhalanx - in that case index middle phalanx will have hand palm set as its parent bone.

## Scripting

## HandPhysicsSkeleton

Public interface of the HandPhysicsSkeleton is already in large part described in its [How To Use section](#), every property in the HandPhysicsSkeleton's Inspector window is available in code as well. This section will describe some of its methods that are available in code only.

## Public methods

`void AddBone(HandPhysicsBone bone)`

- This method is used to add a new bone to the physics skeleton. When adding a bone in Unity editor via HandPhysicsSkeleton's Inspector, identity of the bone instance does not matter; however, when adding a new bone via this method then the identity of the bone must be set to the desired value (using `BoneId` property of the `HandPhysicsBone`). An error is generated if physics skeleton already contains a bone with the same identity or if the specified bone instance is already part of another physics skeleton

`void ClearBones()`

- Removes all bones from the hand physics skeleton

`HandPhysicsBone GetParentBone(HandBoneId bone)`

- Finds and returns the closest parent bone for the specified bone identity contained in the `HandPhysicsSkeleton`. If the parent bone for the specified bone identity is not found, null is returned. Note that it does not matter whether the `HandPhysicsSkeleton` contains a bone with the specified identity or not, this method will always return the bone that is the closest ancestor for a bone with the specified identity, i.e. it will always return a bone that is on upper level in hierarchy from the specified bone identity, never the one at the same level

`HandBoneId? GetParentBoneId(HandBoneId bone)`

- Variation of the previously described method, this one just returns the identity of the parent bone rather than the bone instance

`bool RemoveBone(HandBoneId bone)`

- Removes physics bone with the specified identity from the `HandPhysicsSkeleton`. Return value indicates whether the physics skeleton contained a bone with the specified identity before its removal in the first place

`void RemoveBoneAt(int index)`

- This method is similar to the previously described, except the bone to remove is specified by its index in the `HandPhysicsSkeleton`'s list of bones (`Bones` property). Note that this method might result in error if the specified index is out of range

## HandPhysicsBone

Public interface of the `HandPhysicsBone` is already in large part described in its [How To Use section](#), every

property in the HandPhysicsBone's Inspector window is available in code as well. This section will describe some of its other public members available in code.

## Public properties

Transform Transform { get; }

- Cached reference to the transform component of the HandPhysicsBone

Rigidbody Rigidbody { get; }

- Cached reference to the rigid body component of the HandPhysicsBone

ConfigurableJoint Joint { get; }

- Cached reference to the joint that the HandPhysicsBone uses to connect its rigid body to its parent bone's rigid body. If the physics bone's parent bone is not its immediate natural parent bone in the hand bone hierarchy or the bone has no parent bone, then this joint won't be connected to another physics bone, but will instead be connected to world space origin

bool IsParentBoneClosestAncestor { get; }

- This property indicates whether the physics bone's parent bone is its immediate parent, i.e. its closest possible ancestor in the hand bone hierarchy. If the bone has no parent bone in the physics skeleton it is attached to, then this will be false

ReadOnlyCollection<HandPhysicsBone> ChildBones { get; }

- A read-only collection of physics bones that are child bones of the HandPhysicsBone in the same HandPhysicsSkeleton. All bones in this list will have the HandPhysicsBone as their parent bone. Bones in this list are sorted by their HandBoneId

Quaternion BindPoseRotation { get; set; }

- Rotation of the hand physics bone relative to the bone's parent that serves as bind pose rotation. This rotation is reset every time bone is added to a physics skeleton. In edit mode it gets updated constantly with every change to bone's transform so setting it then will probably have no effect whatsoever. In addition, this value plays important role when handling bone joint's angular limits so that they remain relatively same during hand physics skeleton's lifetime. Note that this rotation is relative to the bone's parent bone that is also it's closest possible ancestor, it only changes when bone is added to a physics skeleton where it has such kind of parent bone

# General

# ScreenFader

- Creates a canvas as a child of Camera which will be used to fade in and out
- Requires component of type Camera

## Public Fields

float FadeInSpeed

float FadeOutSpeed

bool IsFadeInProgress

UnityEvent onFadeInStart

UnityEvent onFadeInEnd

UnityEvent onFadeOutStart

UnityEvent onFadeOutEnd

## Public Methods

void DoFadeIn()

void DoFadeOut()

float GetCanvasGroupAlphaValue()

## Example

```
1 private IEnumerator DoTeleport()
2 {
3     screenFader.DoFadeIn();
4
5     while (screenFader.GetCanvasGroupAlphaValue() != 1f)
6     {
7         yield return new WaitForEndOfFrame();
8     }
9
10    Teleport();
11
12    screenFader.DoFadeOut();
13 }
```

# RayVisuals

- Used for visualizing rays
- Requires a component of type LineRenderer

### Public Methods

void EnableLineRenderer()

void DisableLineRenderer()

void DrawValidRay(Vector3, Vector3)

- Draws a ray from start to end with ray valid gradient

void DrawInvalidRay(Vector3, Vector3)

- Draws a ray from start to end with ray invalid gradient

void DrawSelectRay(Vector3, Vector3)

- Draws a ray from start to end with ray select gradient

ReduceLineWidthByPercentage(float)

- Scales line width dynamically in threshold of rayStartWidth and rayStartWidth - rayWidthDelta by percentage

## BadHandTrackingDetection

### Description

- Visual indicator for loss of confidence in hand tracking
- Uses available data from hand tracking data providers to disable the corresponding hand
- HandSkeleton - reference to the hand skeleton you want to disable when bad hand tracking is detected
- HandDataProvider - reference to the hand tracking data provider, necessary to check for confidence of hand tracking data
- BadTrackingCanvas - object used to notify the user when loss of tracking occurs
- BadTrackingMaterial - material which the hand will change to when the hand skeleton is disabled

### Public fields

[Renderer](#) ObjectRenderer

- Hand renderer whose material needs to be changed upon detecting bad tracking



## Public Methods

`void DetectLowTrackingConfidence()`

- Detects when the assigned data provider's tracked hand data confidence is low and disables the referenced hand skeleton, freezing the hand in place - this prevents loss of interaction and dropping of held objects to some degree

`void EnableHand()`

- Used to disable the referenced hand skeleton, change It's original material and enable the visual indicator for loss of tracking

`void DisableHand()`

- Used to enable the referenced hand skeleton, revert It's material to the original one and disable the visual indicator for loss of tracking