OpenMG 2.0 SDK Setup Guide (version 1.2)

Contents

Introduction

The OpenMG (Open Medical Gesture) is a universal gesture interface. It is a software developer's tool that allows computer programmers to easily incorporate natural hand movements, gestures, and a tactile interface into their immersive 3D simulations and VR/MR applications.

In development from 2016-2019, OpenMG 1.0 SDK has several novel attributes:

- It supports any sensor device be it a 3D camera or a glove type sensor through device virtualization.
- It focuses on low cost sensors (a few hundred dollars) and is designed with an API to readily add new devices to the system as they become commercially available.
- It contains its own universal hand gesture model with an inverse kinematic structure.
- It employs artificial intelligence to learn new gestures using a gesture optimized machine learning console.
- It includes a built-in library of gestures to support medical simulations & procedures, as well as common tasks found in military tactical simulations
- It includes a Software Developer Kit (SDK) and an Application Programming Interface (API) to allow for rapid adoption by programmers into a variety of applications, with specific tools to support Unity3D, C#, Windows .DLL and other game engines.
- It is Open Source (Apache 2.0) and is available for free to everyone.

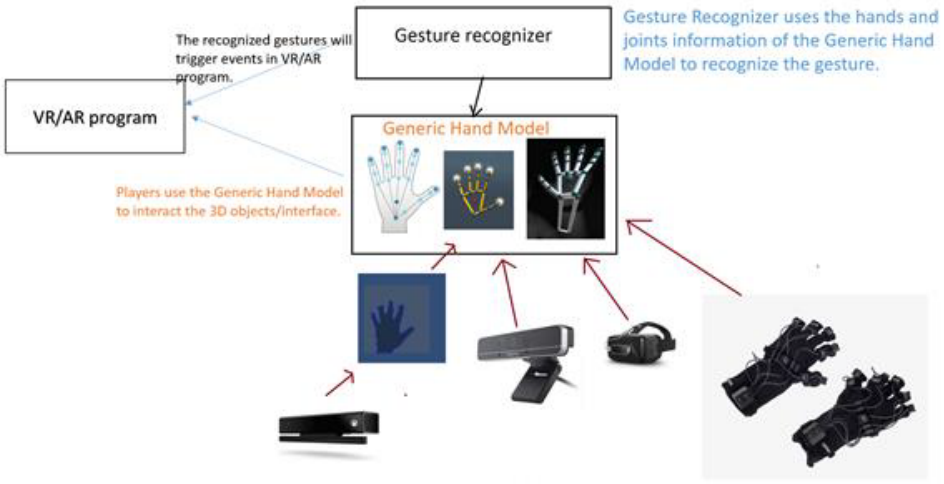


Fig: OpenMG 1.0 design flow

OpenMG 2.0 SDK currently supports two configurations of hand tracking i.e.

1. Oculus Quest 2 HMD by itself.
2. Oculus Quest 2 HMD mounted with a Leap Motion Camera.
3. Leap Motion Camera standalone (facing ceiling).

It also has a set of virtual objects which we call smart objects as they have physics defined to it as well as interaction behaviors to respond to hand gestures as naturally as possible.

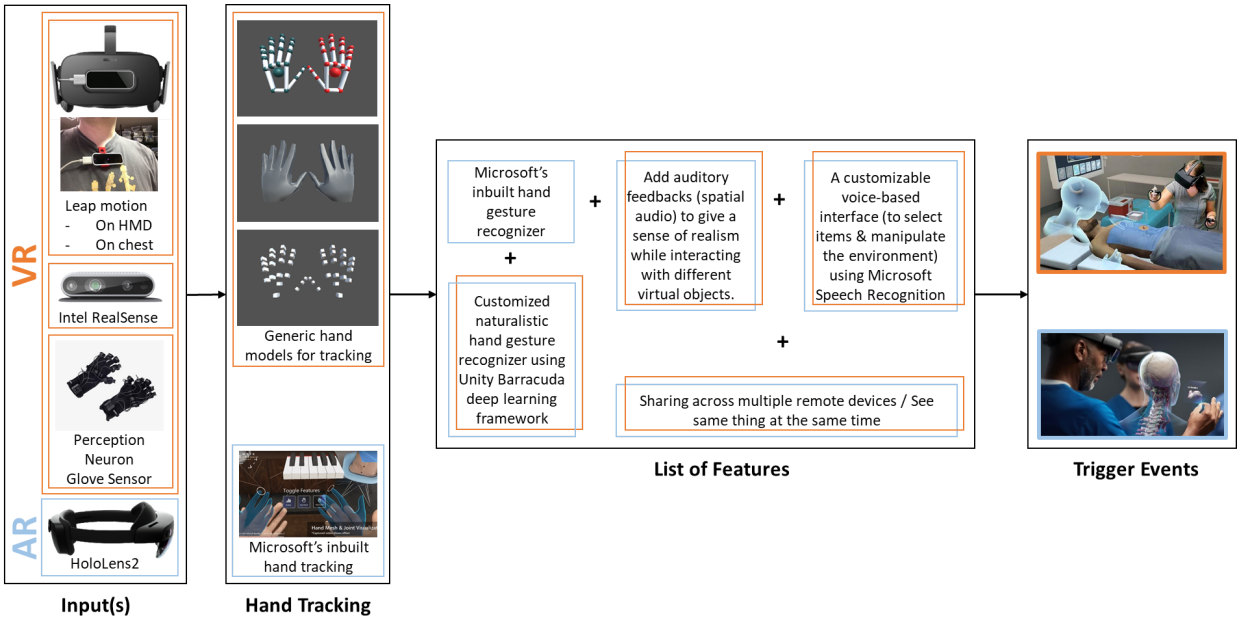


Fig: OpenMG 2.0 design flow (proposed)

System Requirements

This is the minimum system requirement for setting up Leap Motion and Oculus.

- Windows 10 64 bit or newer
- 8GB of RAM or more
- NVIDIA GPU

Device Installation

Leap Motion

Hardware installations required

- Parts required:
    - Leap Motion Controller: https://www.ultraleap.com/product/leap-motion-controller/
    - VR mount (can be 3D printed as well): https://www.ultraleap.com/product/vr-developer-mount/#overview
    - Oculus link cable (to tether it to PC): https://www.meta.com/quest/accessories/link-cable/
- Mount the leap motion sensor to the Oculus headset as per: https://www.youtube.com/watch?v=OUdL3y-mrFM
- Connect the headset and leap motion to the PC using their respective cables.

Software installations/steps required

- Unity version: 2023.1.0a17 or higher.
- Install UltraLeap Orion driver: https://developer.leapmotion.com/releases/leap-motion-orion-410-99fe5-crpgl.

Oculus Quest2

Hardware installations required

- Parts required:
    - Oculus link cable (to tether it to PC): https://www.meta.com/quest/accessories/link-cable/
- Connect the headset to the PC using the cable.

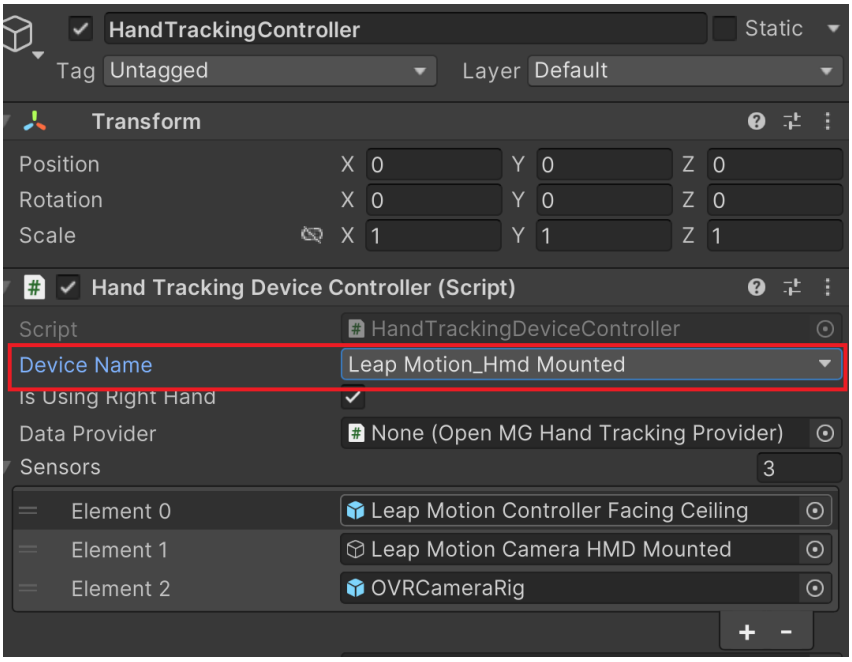
Software installations/steps required

- Unity version: 2023.1.0a17 or higher.
- Oculus XR Plugin - 1.5.1 (from Unity Package Manager) - ***can skip this as it will be included there in the GitHub repo sample project.***

After installing the dependencies above you can just clone the repo and open it in the above version of Unity.

Scene information

- *Scenes/Playground.unity* is the main scene.
- Choose *HandTrackingController* in the scene and under the *HandTrackingDeviceController* component, choose the *Device Name* to specify the type of device you want to use.



- If all the software installations are followed as above, then just run the scene and put on the headset.
- You should see your hands being tracked in the scene.
- Try interacting with the various smart objects there (currently designed to just work with the right hand).
- To pick any of the instruments just touch them and they should be placed with the virtual hand arranged as you would hold in the real world. Go to the Gesture section below to learn how to actuate various instruments.
- You can choose between Leap Motion and Quest2 in Edit mode from the *HandTrackingDeviceController* object *DeviceName* dropdown list in the *Playground* scene and then click on the Play button.
- Leap based demos below shows the interaction of a few of them:
    - Button/Knobs, Ball
    - Auscultation
    - Smooth syringe press
    - Scissors
    - Bandage roll
    - Scalpel
    - All instruments

Gesture Recognition

The framework comprises using hand motions to define a certain action that triggers the smart objects to function as close to the real world as possible.

Defined gestures

1. **Scissors** → Middle finger to Thumb pinch-release action activates the scissor cut-release action.
2. **Syringe** → Middle+Index finger to Thumb pinch-release action activates the syringe press-unpress action.
3. **Bandage roll** → Once picked using the right hand, use the left hand to pull the cloth as seen in the video.
4. **Auscultation** → Place the palm open and facing down on the chest of the mannequin on the bed to hear the heartbeat sound.

The code reference for the gesture functionalities will be under Gesture Scripting Reference.

Core OpenMG Scripting Reference

OpenMGFinger.cs

This script handles the generic hand model finger definition. The finger model we follow is 3 joints per finger, one fingertip per finger, and one joint for the wrist making it a total of 21 points in total per hand. The Bones are Proximal, Intermediate, and Distal.

FUNCTION	DESCRIPTION
public Vector3[] GetJoints()	Returns the array of postions of each of the finger joints
public Vector3 GetTipPosition()	Returns the array of positions of each of the finger tips
public void SetJointPosition(Vector3 pos, int index)	Assigns a position *pos* to each of the joints based on its *index* - *index*: 0 (metacarpal), 1 (proximal), 2 (distal), 3 (finger tip)
public Vector3 GetJointPosition(int index)	Returns the position of the joint based on its *index*

public void SetJointRotation(Quaternion rot, int index)	Assigns a rotation *rot* to each of the joints based on its *index*
public Quaternion GetJointRotation(int index)	Returns the rotation of the joint based on its *index*
public float GetLength()	Returns the physical length (in meters) of a finger measured from a finger's base joint to the tip
public float GetDistanceFromTipTo(Vector3 pos)	Returns the physical length (in meters) of any position *pos* from the tip of a finger
public Vector3 SetFingerDirection(Vector3 dir)	Set the direction the finger is pointing to
public Vector3 GetFingerDirection()	Get the direction the finger is pointing to
public void SetBoneDirection(int bone, Vector3 dir)	Set the direction the bone is pointing to
public Vector3 GetBoneDirection(int bone)	Get the direction the bone is pointing to

OpenMGHand.cs

This script handles the generic hand model defined for OpenMG.

FUNCTION	DESCRIPTION
public void SetPalmQuaternion(Quaternion quat)	Sets the orientation and normalized direction of the palm
public Vector3 GetPalmPosition()	Returns the position of the palm
public Vector3 GetPalmNormalDirection()	Returns the normalized palm direction
public Vector3 GetPalmVelocity()	Returns the velocity at which the palm is moving
public void SetPalmNormalDirection(Vector3 dir)	Sets the direction *dir* of the palm normal
public void SetPalmPosition(Vector3	Sets the position *pos* of the palm

pos)	
public void SetHandRotation(Quaternion quat)	Sets the rotation *quat* of the hand
public Quaternion GetHandRotation()	Returns the rotation of the hand
public Vector3 GetJointPosition(int finger, int index)	Returns the position of a joint in a *finger* based on its *index* - *finger*: 0 (thumb), 1 (index), 2 (middle), 3 (ring), 4 (pinky)
public Quaternion GetJointRotation(int finger, int index)	Returns the rotation of a joint in a *finger* based on its *index*
public void SetThumbJointPosition(Vector3 pos, int joint)	Set the position *pos* of each of the joints *joint* in the thumb
public void SetThumbJointRotation(Quaternion rot, int joint)	Set the rotation *rot* of each of the joints *joint* in the thumb
public Vector3 GetThumbTipPosition()	Returns the position of the tip of the thumb
public float GetThumbLength()	Returns the physical length (in meters) of the thumb
public void SetIndexJointPosition(Vector3 pos, int joint)	Set the position *pos* of each of the joints *joint* in the index finger
public void SetIndexJointRotation(Quaternion rot, int joint)	Set the rotation *rot* of each of the joints *joint* in the index finger
public Vector3 GetIndexTipPosition()	Returns the position of the tip of the index finger
public float GetIndexLength()	Returns the physical length (in meters) of the index finger
public void SetMiddleJointPosition(Vector3 pos, int joint)	Set the position *pos* of each of the joints *joint* in the middle finger
public void SetMiddleJointRotation(Quaternion rot,	Set the rotation *rot* of each of the joints *joint* in the middle finger

int joint)	
public Vector3 GetMiddleTipPosition()	Returns the position of the tip of the middle finger
public float GetMiddleLength()	Returns the physical length (in meters) of the middle finger
public void SetRingJointPosition(Vector3 pos, int joint)	Set the position *pos* of each of the joints *joint* in the ring finger
public void SetRingJointRotation(Quaternion rot, int joint)	Set the rotation *rot* of each of the joints *joint* in the ring finger
public Vector3 GetRingTipPosition()	Returns the position of the tip of the ring finger
public float GetRingLength()	Returns the physical length (in meters) of the ring finger
public void SetPinkyJointPosition(Vector3 pos, int joint)	Set the position *pos* of each of the joints *joint* in the pinky finger
public void SetPinkyJointRotation(Quaternion rot, int joint)	Set the rotation *rot* of each of the joints *joint* in the pinky finger
public Vector3 GetPinkyTipPosition()	Returns the position of the tip of the pinky finger
public float GetPinkyLength()	Returns the physical length (in meters) of the pinky finger
public int GetFingersCount()	Returns the total number of fingers in the hand
public void SetIndexDirection(Vector3 dir)	Set the direction *dir* in which the specified finger is pointing. The direction is expressed as a unit vector pointing in the same direction as the tip
public void SetMiddleDirection(Vector3 dir)	
public void SetPinkyDirection(Vector3 dir)	
public void SetRingDirection(Vector3 dir)	

public void SetThumbDirection(Vector3 dir)	
public void SetFistStrenth(float val)	Set the confidence value *val* between 0 and 1 of how strong a fist is made by a hand
public float GetFistStrength()	Get the confidence value of the fist made by hand
public void SetPinchDistance(float val)	Set the distance *val* between the thumb and index finger of a pinch hand pose
public float GetPinchDistance()	Get the pinch distance above
public void SetPinchStrength(float val)	Set the holding strength *val* of a pinch hand pose. The strength is 0 for an open hand and blends to 1 when a pinching hand pose is recognized.
public float GetPinchStrength()	Get the pinch strength above
public void SetPinchPosition(Vector3 pos)	Set the approximate position *pos* where the thumb and index finger will be if they are pinched together
public Vector3 GetPinchPosition()	Get the pinch position above
public void SetWristPosition(Vector3 pos)	Set the position *pos* of the wrist for the hand
public Vector3 GetWristPosition()	Get the position of the wrist above
public void SetDirectionToFingers(Vector3 pos)	Set the direction *pos* towards the fingers that is perpendicular to the palmar and radial axes
public Vector3 GetDirectionToFingers()	Get the DirectionToFingers above
public void SetDirectionAwayFromPalm(Vector3 pos)	Set the direction *pos* the Hand's palm is facing
public Vector3 GetDirectionAwayFromPalm()	Get the DirectionAwayFromPalm above
public void SetDirectionToThumb(Vector3 pos)	Set the direction *pos* towards the thumb that is perpendicular to the palmar and

	distal axes. Left and right hands will return in opposing directions
public Vector3 GetDirectionToThumb()	Get the DirectionToThumb above
public void SetPinchStatus(bool status)	Set the *status* of pinching performed by hand
public bool GetPinchStatus()	Get the *status* of pinching performed by hand

OpenMGHandTrackingProvider.cs

This script is the parent class for interfacing any hand-tracking model with the generic OpenMG hand model. This basically assigns the sensor-tracked data to the OpenMG hand model.

FUNCTION	DESCRIPTION
public void UpdateThumbPosition(Vector3 pos, int hand, int joint)	Updates the thumb's *joint* position *pos* of a *hand* - *hand*: 0 (left), 1 (right)
public void UpdateIndexPosition(Vector3 pos, int hand, int joint)	Updates the index finger's *joint* position *pos* of a *hand*
public void UpdateMiddlePosition(Vector3 pos, int hand, int joint)	Updates the middle finger's *joint* position *pos* of a *hand*
public void UpdateRingPosition(Vector3 pos, int hand, int joint)	Updates the ring finger's *joint* position *pos* of a *hand*
public void UpdatePinkyPosition(Vector3 pos, int hand, int joint)	Updates the pinky finger's *joint* position *pos* of a *hand*
public void UpdateThumbRotation(Quaternion rot, int hand, int joint)	Updates the thumb's *joint* rotation *rot* of a *hand*
public void UpdateIndexRotation(Quaternion rot, int hand, int joint)	Updates the index finger's *joint* rotation *rot* of a *hand*
public void	Updates the middle finger's *joint* rotation

UpdateMiddleRotation(Quaternion rot, int hand, int joint)	*rot* of a *hand*
public void UpdateRingRotation(Quaternion rot, int hand, int joint)	Updates the ring finger's *joint* rotation *rot* of a *hand*
public void UpdatePinkyRotation(Quaternion rot, int hand, int joint)	Updates the pinky finger's *joint* rotation *rot* of a *hand*
public void UpdatePalmPosition(Vector3 pos, int hand)	Updates the palm postion *pos* of a *hand*
public Vector3 GetPalmPosition(int hand)	Returns the palm postion of a *hand*
public void UpdatePalmVelocity(Vector3 vel, int hand)	Updates the palm velocity *vel* of a *hand*
public Vector3 GetPalmVelocity(int hand)	Returns the palm velocity of a *hand*
public void UpdateHandRotation(Quaternion quat, int hand)	Updates the hand rotation *quat* of a *hand*
public Quaternion GetHandRotation(int hand)	Returns the hand rotation of a *hand*
public Vector3 GetTipPosition(int hand, int finger)	Returns the *finger* tip position of *hand*
public Vector3 GetJointPosition(int hand, int finger, int index)	Returns the joint position of a *finger* based on its *index* and the *hand*
public Quaternion GetJointRotation(int hand, int finger, int index)	Returns the joint rotation of a *finger* based on its *index*and the *hand*
public float GetFingerLength(int hand, int finger)	Returns of the physical length (in meters) of a *finger* in a *hand*
public int GetNumOfHands()	Returns the total number of hands
public OpenMGHand GetHand(int hand)	Returns the object of whether its a left or right hand

<DeviceName>HandTrackingProvider.cs

This script derives from OpenMGHandTrackingProvider.cs:

- *LeapMotionHandTrackingProvider.cs* interfaces the Leap Motion hand tracking model to the generic OpenMG hand model.
- *Quest2HandTrackingProvider.cs* interfaces the Quest2 hand tracking model to the generic OpenMG hand model.

HandTrackingDeviceController.cs

This script handles the integration of various hand tracking sensors and opens it for the designer to use it as needed e.g. expose it through a UI for creating a selection menu for the device to be used in play mode.

FUNCTION	DESCRIPTION
public enum DeviceName	Used to list down the names of various hand-tracking sensors
void Awake()	This function defines which device from the list is activated as default
public void SwitchDevice(string device)	This allows the user to switch between devices at runtime

SmartObjectBehaviour.cs

This is the parent class that holds few properties that make the object it is attached to smart for fine control of the object and the way we interact with it.

PUBLIC VARIABLES	DESCRIPTION
public Transform leftHand	Transform pointing to the Left Hand
public Transform rightHand	Transform pointing to the Right Hand
public Material opaqueMat	Opaque material
public Material fadeMatPartial	Partially faded material
public Material fadeMatFull	Fully faded material

public Transform homeTransform	The base transform of the smart object
public Transform mainControl	Control Activate/Deactivate time for contact
public GameObject fakeHand	The fake hand holding the smart object
public bool showFakeHand	Show/Hide the fake hand holding the smart object

RUNTIME ASSIGNED VARIABLES	**DESCRIPTION**
public Renderer leftHandRenderer	to toggle between left tracked hand and positioned hand render
public Renderer rightHandRenderer	to toggle between right tracked hand and positioned hand render
public Transform leftHandAnchor	Anchor about left wrist
public Transform rightHandAnchor	Anchor about right wrist
public bool touchFlag	For picking up smart object (default = true)
public bool pressFlag	For instruments with press functionality e.g. syringes (default = false)
public Rigidbody rigidBody	get the attached Rigidbody
public Vector3 origPos	capture start postion of object (may come to use later)
public Quaternion origRot	capture start rotation of object (may come to use later)

FUNCTION	**DESCRIPTION**
public virtual void Awake()	Capture the start postion and rotation of the smart object
public virtual void Start()	Assign the runtime variables *leftHandRender, rightHandRenderer, leftHandAnchor, rightHandAnchor,*

	rigidBody
public virtual void AttachToHand()	Attach instrument to Hand
public virtual void RemoveFromHand()	Remove instrument from Hand
public GameObject FindChild(GameObject parent, string childName)	Helper function to find child of a gameobject *parent* with certain name *childName*
public IEnumerator LerpScale(Transform transformToChange, Vector3 startScale, Vector3 targetScale, float duration)	Helper function to change the scale of an object's transform *transformToChange* over a period of time *duration*
public IEnumerator LerpPosition(Transform transformToChange, Vector3 startPosition, Vector3 targetPosition, float duration)	Helper function to change the position of an object's transform *transformToChange* over a period of time *duration*
public IEnumerator LerpRotation(Transform transformToChange, Quaternion startValue, Quaternion endValue, float duration)	Helper function to change the rotation of an object's transform *(transformToChange)* over a period of time *(duration)*

Gesture Scripting Reference

OmgDetectionFinger.cs

This defines a few *OpenMGFinger* functionalities required specifically for gesture detection.

FUNCTION	DESCRIPTION
public void SetFinger(OpenMGFinger a_Finger)	Set the current finger mapping it to *OpenMGFinger* type
public EFinger GetFingerType()	Get the type of finger (thumb/index/middle/ring/pinky)
public Quaternion GetFingerRotation()	Get the rotation of the finger

public Vector3 GetFingerDirection()	Get the direction the finger is pointing to
public Vector3 GetTipPosition()	Get the position of the fingertip
public Quaternion GetBoneRotation(ESpecificBone a_Bone)	Get the rotation of a specific bone in the finger (proximal/intermediate/distal)
public Vector3 GetBoneDirection(ESpecificBone a_Bone)	Get the direction the bone is pointing to
public Vector3 GetBonePosition(ESpecificBone a_Bone)	Get the Center position of the bone
public bool IsExtended()	See if finger is extended or not

OmgDetectionHand.cs

This defines a few *OpenMGHand* functionalities required specifically for gesture detection.

FUNCTION	DESCRIPTION
public void SetHand(OpenMGHand a_Hand)	Set the current finger mapping it to *OpenMGHand* type
public int NumberOfFingersExtended()	Count the number of fingers extended
public bool IsFingerExtended(EFinger a_finger)	Check if a finger is extended
public bool IsSet()	Check if all fingers and hands are valid
public bool IsClosed(float a_fTolerence)	Check if hand is closed
public bool IsPinching()	Check for hand pinching
public Vector3 GetDirectionToFingers()	Get the direction towards the fingers that is perpendicular to the palmar and radial axes
public Vector3 GetDirectionToThumb()	Get the direction towards the thumb that is perpendicular to the palmar and distal

	axes. Left and right hands will return in opposing directions
public Vector3 GetDirectionAwayFromPalm()	Get the direction the Hand's palm is facing
public Vector3 GetPositionBetweenPinch()	Get the rough position where pinching is occurring
public Vector3 GetHandAxis(EHandAxis a_HandAxis)	Get the axes of the hand (distal, radial, palmar)
public Vector3 GetHandPosition()	Get the position of the hand (palm)
public Quaternion GetRotation()	Get the rotation of the hand
public Vector3 GetWristPosition()	Get the position of the wrist
public Vector3 GetVelocity()	Get the velocity vector of the palm
public OmgDetectionFinger GetFinger(EFinger a_FingerType)	Get the current finger in the hand being used for gesture detection
public float DistanceBetweenFingers(EFinger a_Finger0, EFinger a_Finger1)	Calculate the distance between fingers

OmgGestureDetectionManager.cs

This script handles the core gesture detection framework to translate certain hand and finger movements into meaningful gestures to trigger smart objects to perform certain actuations.

This also has a lot of enums for easy access to some named constants across the gesture detection framework.

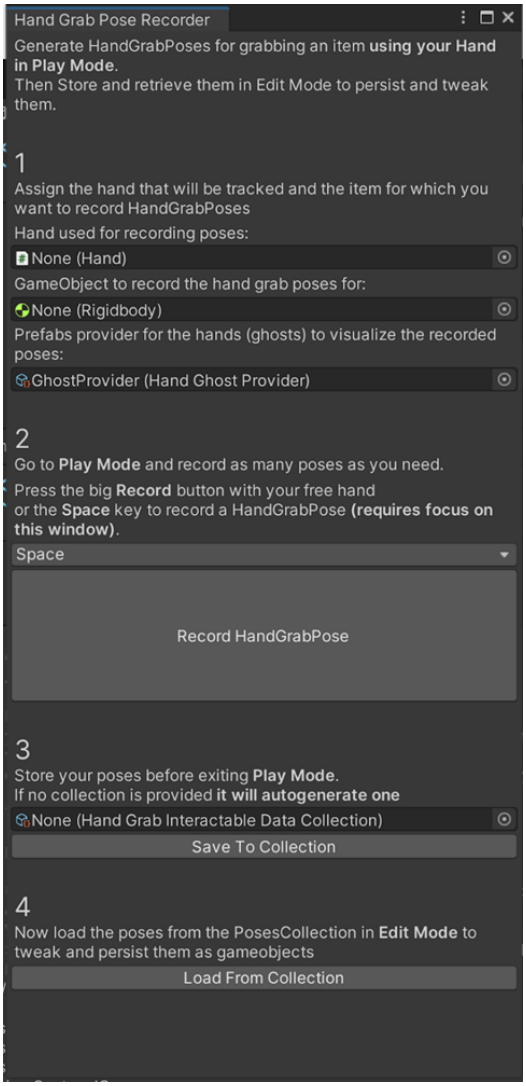
FUNCTION	DESCRIPTION
public static OmgGestureDetectionManager Get()	Define static for shared access of data
public OmgDetectionHand GetHand(EHand a_hand)	Get the gesture detection hand
public bool IsHandSet(EHand a_hand)	Check if the hand is set and ready to go

public bool IsBothHandsSet()	Check if both hands are set and ready to go
public bool AreBothHandsVisible()	Check if both hands are in the FOV
public float GetDistanceBetweenHands()	Get distance between hands
GameObject CreateCollisionSphere()	Create small sphere colliders for detecting finger collisions
void UpdateHand(OpenMGHand hand)	Update the hand data being received

Make a new art asset/object/gameobject a SmartObject

For Oculus Quest2 (Cutomized over Oculus Integration SDK)

1. Make sure the SmartObject has a *RigidBody* component and a *Collider*.
2. Add the *Grabbable.cs* to the SmartObject.
    a. For creating a grab pose for the object, use the *Hand Grab Pose Recorder* menu under *Oculus -> Interaction*.



b. Drag and drop the Hand model reference (*LeftHand* or *RightHand*) from under the *OVRCameraRig-> OVRInteraction->OVRHands* component to *None (Hand)* field.

c. Drag the SmartObject with the RigidBody attached as per 1 to the *None (Rigidbody)* field.

d. Leave the *GhostProvider (Hand Ghost Provider)* field as is.

e. In the Play mode and headset on, grab the SmartObject using your hand that you specified above.

f. Once you make a near resemblance of how you would want to hold the object, press the *RecordHandPose* button or click *Space* (whichever is easier as you have to do it with your headset on). The pose gets saved under *Assets/HandGrabInteractableDataCollection* with the name *<SmartObjectName>_HandGrabInteractable*.

g. You can make as many poses as you want for the SmartObject and it will all be clubbed under the same *<SmartObjectName>_HandGrabInteractable>* pose name created.

h. Once you are happy, get your headset off and click on *Save To Collection* button and the *None* field there gets replaced by *<SmartObjectName>_HandGrabInteractable>*.

i. Exit Play mode and with the SmartObject selected click on *Load From Collection*.

j. This should create the *HandGrabInteractable* gameobject as a child of the SmartObject.

k. There could be a possibility that the dummy hand (*HandGrabInteractable* gameobject) that just got created above may appear weird (out of shape or very tiny or very big). Just to note that it just appears so as it totally depends on the scale of the SmartObject but while grabbing in play mode again it will grab perfectly. One way to rectify the visual is to locally play around with the scale of the *HandGrabInteractable* gameobject which is the child of the SmartObject till it encapsulates right. (Again, this is not needed but if you want to create clean visuals then go for it).

l. If you want to create something similar for the other hand either you can follow all the steps *a to l* above or just go to the *HandGrabInteractable* gameobject and inside the *HandGrabInteractable* component under it, click *Create Mirrored HandGrabInteractable.* I personally prefer the latter as it gives an already well aligned way which might need tweaks as in *k*.

m. Again, manual fine-tuning may be required for proper alignment of the hand to the surface of the object.

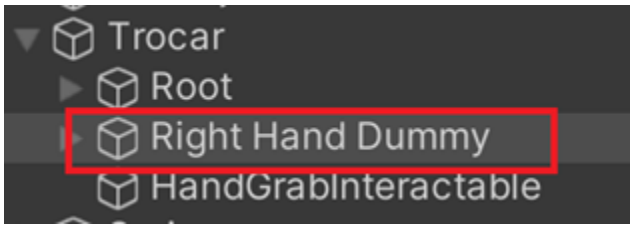
- Under *HandGrabInteractable gameobject->HandGrabPose* component, make sure the finger which you want to change has the *Fingers Freedom* set to *Free* from the drop-down list (Note the *Max* there refers to the Pinky finger).
- That way you will be able to use the *Joint Angles* section just below it to make the necessary change to the specific joints of the finger.

Note: We can jointly work on creating poses specific to your needs.

For Leap Motion

1. Make sure the SmartObject has a *RigidBody* component and a *Collider*.

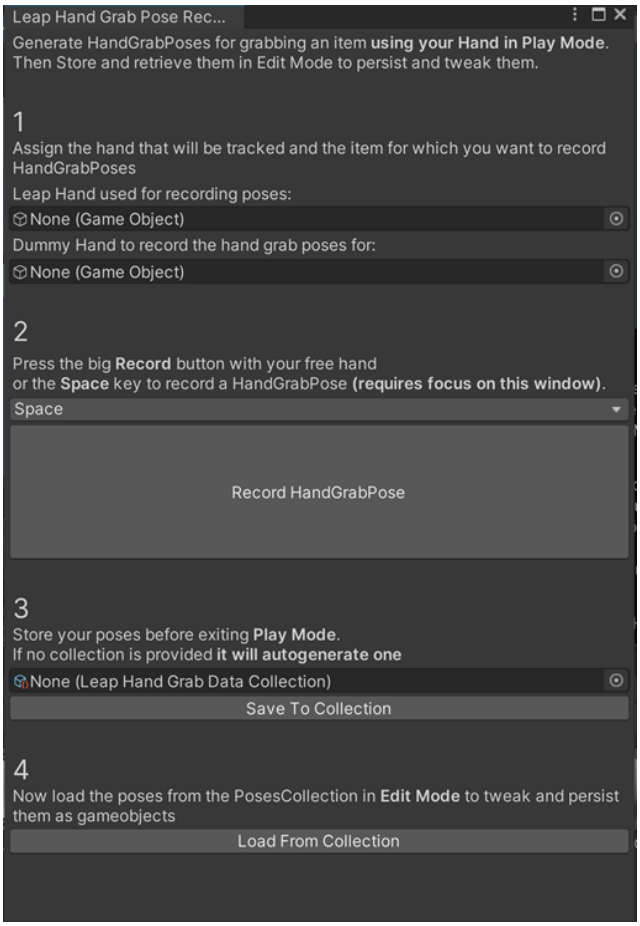
2. From *Assets/Prefabs* add either *Left Hand Dummy* or *Right Hand Dummy* or both as child of the SmartObject.



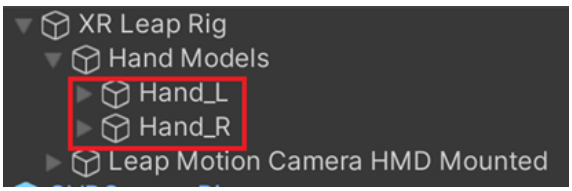
3. There could be a possibility that the dummy hand gameobject that just got created above may appear weird (out of shape or very tiny or very big). Just to note

that it just appears so as it totally depends on the scale of the SmartObject it is a child of. One way to rectify the visual is to locally play around with the scale of the dummy hand gameobject which is the child of the SmartObject till it encapsulates right.

4. For creating a grab pose for the SmartObject, use the *Leap Hand Grab Pose Recorder* menu under *Leap -> Interaction* editor menu.



5. Drag and drop the Hand model reference whose pose is to be recorded (*Hand_L* or *Hand_R as shown in figure below*) from under the *LeapMotionHandler/Desktop Leap Rig* or *LeapMotionHandler/ XR Leap Rig* (depending on how you are using the leap motion) to *Leap Hand used for recording poses: None (Game Object)* field.



6. Drag and drop the Dummy Hand Model from the child of the SmartObject which was added in 2 to the *Dummy Hand to record the hand grab poses for: None (Game Object)* field.

7. In the Play mode and headset on, grab the SmartObject using your hand that you specified above.

8. Once you make a near resemblance of how you would want to hold the object, press the *Record HandGrabPose* button or click *Space* (whichever is easier as you might have to do it with your headset on). The pose gets saved under *Assets/LeapHandGrabDataCollection* with the name *<SmartObjectName>_LeapHandGrabData*.

9. Once you are happy, (if headset was on get your headset off) click on *Save To Collection* button and the *None* field there gets replaced by *<SmartObjectName>_LeapHandGrabData*.

10. Exit Play mode and with the SmartObject selected click on *Load From Collection*.

11. This should update the poses for the dummy hand added as per 2 with the saved poses as per 8.

12. Manual fine-tuning may be required for proper alignment of the hand to the surface of the object.

13. Once you are happy with the fine-tuning, turn the dummy hand inactive in hierarchy.

14. The steps above solve the hand pose for grabbing the object. For proper interaction with the tracked hand we need to align the SmartObject to it.

15. Create and attach a script called <*SmartObjectName*>*Behaviour.cs* to the SmartObject (For reference refer to *ScalpelBehaviour.cs*).

16. Study how *ScalpelBehaviour* component how it has all the things attached and add them as under for the new one created in 15.
    a) Leave the *Left Hand, Right Hand* as they get assigned at runtime.
    b) Set *Opaque Mat, Fade Mat Partial, Fade Mat Full* as is.
    c) Set the *Home Transform* as the immediate parent of the SmartObject.
    d) Leave the *Main Control* unassigned.
    e) Set the *Fake Hand* as the Right Dummy Hand created in 2 ((if not created leave it unassigned).
    f) Set the *Fake Hand Left* as the Left Dummy Hand created in 2 (if not created leave it unassigned).
    g) Set the *Show Fake Hand* to checked.
    h) Set the *Device* to as is.
    i) For *Position at Right Hand Anchor* and *Rotation At Right Hand Anchor*:
        - Drag the SmartObject in the hierarchy to being a child of the *LeapMotionHandler/Desktop Leap Rig* or *LeapMotionHandler/ XR Leap Rig* (depending on how you are using the leap motion)/ */HandModels/Hand_R/HandContainer/R_Arm_Cut/R_Wrist* as we want to anchor the object about the tracked hand wrist.
        - Manually align it to the hand so that it sits properly anchored.
        - Note the position and rotation of the SmartObject after proper alignment.
        - Drag back the SmartObject to where ever it was dragged from (child of the *Home Transform* in 16-c)
        - Now set the values of *Position at Right Hand Anchor* and *Rotation At Right Hand Anchor* with the values noted above.
    j) For *Position at Left Hand Anchor* and *Rotation At Left Hand Anchor* follow the similar steps as 16-i. (if at all left hand is created).

17. Attach the *InteractionBehaviour.cs* to the SmartObject.
    a) *Ignore Grasping* should be marked as checked.
    b) *InteractionManager* component gets assigned at runtime.
    c) Set the *ContactBegin()* unity event with the *AttachToHand()* function inside the <*SmartObjectName*>*Behaviour.cs*. Make sure its *Runtime Only*. This attaches the SmartObject to the hand when it hovers near it.

18. In *the Omg_RightHandFistGesture & RightHandFacingDown (Release Object)* gameobject in hierarchy, Set the *OneGestureEnd()* unity event with the *RemoveFromHand()* function inside the *<SmartObjectName>Behaviour.cs*. On performing this gesture, the object is released from the hand.

Note: We can jointly work on creating poses specific to your needs.

Smart Objects in-built in OpenMG

Scissors, Bandage Scissors, Needle Forceps

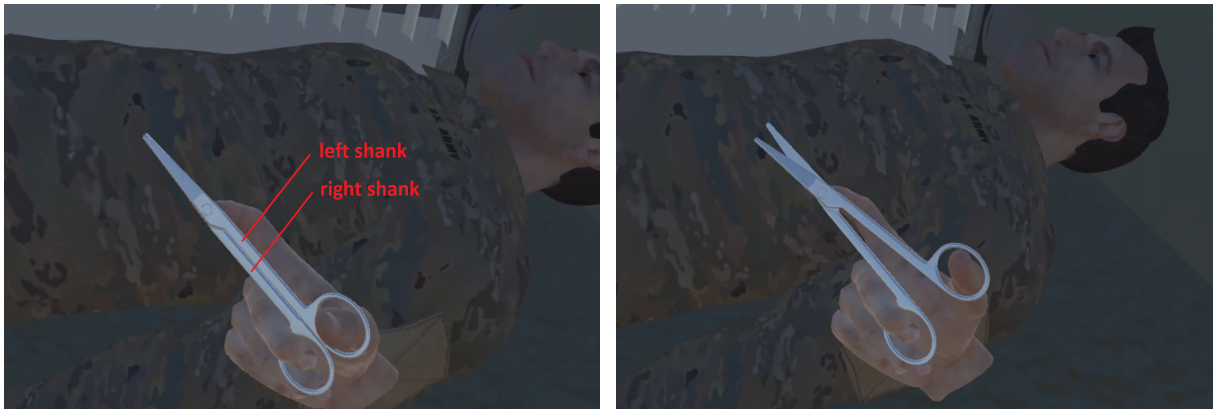


Fig: Scissors close (left) and open (right)

Fig: Bandage Scissors close (left) and open (right)

Fig: Needle Forceps close (left) and open (right)

1. These prefabs have two main parts:
    a. left shank
    b. right shank
2. This has the *ScissorForcepBehaviour.cs, BandageScissorsBehaviour.cs* and *NeedleForcepsBehaviour.cs* all derive from *SmartObjectBehaviour.cs* attached to it which makes these objects smart.

VARIABLES	DESCRIPTION
public Vector3 positionAtLeftHandAnchor	Neutral position of the scissor in left hand where it gets anchored to
public Vector3 rotationAtLeftHandAnchor	Neutral rotation of the scissor in left hand where it gets anchored to
public Vector3 positionAtRightHandAnchor	Neutral position of the scissor in right hand where it gets anchored to
public Vector3 rotationAtRightHandAnchor	Neutral rotation of the scissor in right hand where it gets anchored to
public Vector3 unpressRotationLeftShank	The unpressed scissor's left shank rotation component
public Vector3 pressRotationLeftShank	The pressed scissor's left shank rotation component
public Vector3 unpressRotationRightShank	The unpressed scissor's right shank rotation component
public Vector3 pressRotationRightShank	The pressed scissor's right shank rotation component

public Vector3 unpressRotationThumb	The dummy hand thumb's rotation component when the scissor is in neutral state
public Vector3 pressRotationThumb	The dummy hand thumb's activated rotation component on pressing the scissor

FUNCTIONS	DESCRIPTION
override public void AttachToHand()	Handles the smart object's placement in the the hand when it is touched
override public void RemoveFromHand()	Handles the smart object's removal from the hand when release gesture is perfomed
public void Press()	Performs the closing action of the scissor
public void UnPress()	Performs the opening action of the scissor

Syringe

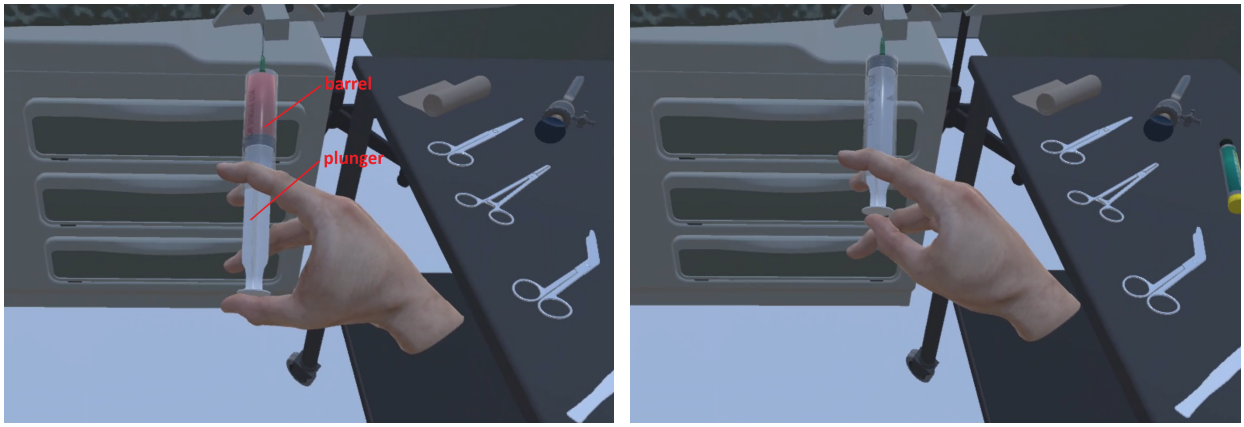


Fig: Syringe neutral (left) and actuated (right)

1. This prefab has two main parts:
    a. plunger
    b. barrel

2. This has the *SyringeBehaviour.cs* derives from *SmartObjectBehaviour.cs* attached to it which makes this object smart.

VARIABLES	DESCRIPTION
public Vector3 syringeUnpressPosition	Used to set the position of the hand and plunger when the syringe is in neutral state
public Vector3 syringePressPosition	Used to set the position of the hand and plunger when the syringe is in actuated state
public Vector3 syringeUnpressRotation	Used to set the rotation of the hand and plunger when the syringe is in neutral state
public Vector3 syringePressRotation	Used to set the rotation of the hand and plunger when the syringe is in actuated state

FUNCTIONS	DESCRIPTION
override public void AttachToHand()	Handles the smart object's placement in the the hand when it is touched
override public void RemoveFromHand()	Handles the smart object's removal from the hand when release gesture is perfomed
public void PressSyringe()	Performs the actuated action of the syringe
public void UnPressSyringe()	Performs the return to neutral action of the syringe

Scalpel

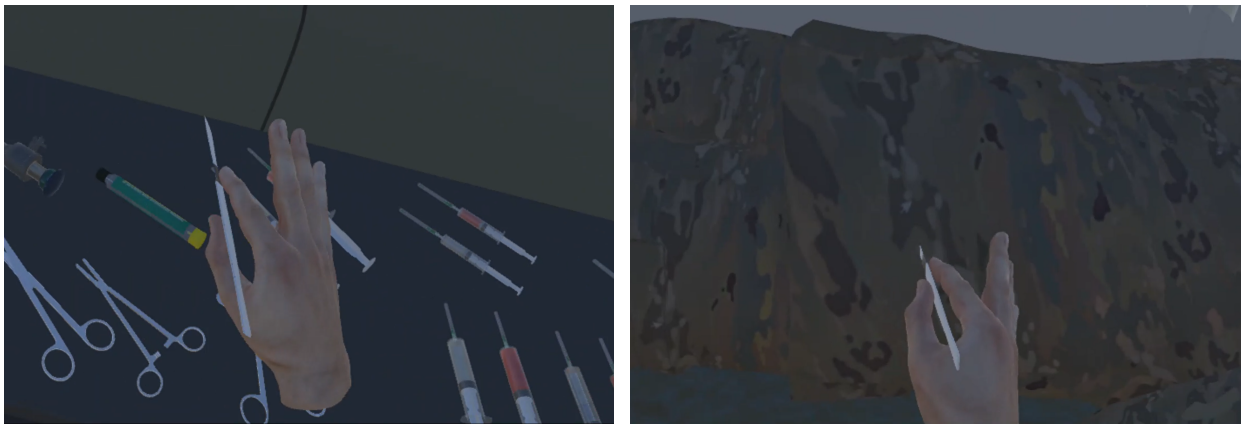


Fig: Scalpel manipulation

1. This has the *ScalpelBehaviour.cs* derives from *SmartObjectBehaviour.cs* attached to it which makes this object smart.

VARIABLES	DESCRIPTION
public Vector3 _positionAtLeftHandAnchor	Neutral position of the scalpel in left hand where it gets anchored to
public Vector3 _rotationAtLeftHandAnchor	Neutral rotation of the scalpel in left hand where it gets anchored to
public Vector3 _positionAtRightHandAnchor	Neutral position of the scalpel in right hand where it gets anchored to
public Vector3 _rotationAtRightHandAnchor	Neutral rotation of the scalpel in right hand where it gets anchored to

FUNCTIONS	DESCRIPTION
override public void AttachToHand()	Handles the scalpel's placement in the the hand when it is touched
override public void RemoveFromHand()	Handles the scalpel's removal from the hand when release gesture is perfomed

Trocar

Fig: Trocar manipulation

1. This has the *TrocarBehaviour.cs* derives from *SmartObjectBehaviour.cs* attached to it which makes this object smart.

VARIABLES	DESCRIPTION
public Vector3 _positionAtLeftHandAnchor	Neutral position of the trocar in left hand where it gets anchored to
public Vector3 _rotationAtLeftHandAnchor	Neutral rotation of the trocar in left hand where it gets anchored to
public Vector3 _positionAtRightHandAnchor	Neutral position of the trocar in right hand where it gets anchored to
public Vector3 _rotationAtRightHandAnchor	Neutral rotation of the trocar in right hand where it gets anchored to

FUNCTIONS	DESCRIPTION
override public void AttachToHand()	Handles the trocar's placement in the the hand when it is touched
override public void RemoveFromHand()	Handles the trocar's removal from the hand when release gesture is perfomed

Bandage Roll

Fig: Bandage Roll neutral (left) and stretched (right)

1. This has the *BandageRollBehaviour.cs* derives from *SmartObjectBehaviour.cs* attached to it which makes this object smart.

VARIABLES	DESCRIPTION
public Vector3 _positionAtLeftHandAnchor	Neutral position of the bandage roll in left hand where it gets anchored to
public Vector3 _rotationAtLeftHandAnchor	Neutral rotation of the bandage roll in left hand where it gets anchored to
public Vector3 _positionAtRightHandAnchor	Neutral position of the bandage roll in right hand where it gets anchored to
public Vector3 _rotationAtRightHandAnchor	Neutral rotation of the bandage roll in right hand where it gets anchored to

FUNCTIONS	DESCRIPTION
override public void AttachToHand()	Handles the bandage roll's placement in the the hand when it is touched
override public void RemoveFromHand()	Handles the bandage roll's removal from the hand when release gesture is perfomed

Autoinjector

Fig: Autoinjector actuation

1. This has the *BandageRollBehaviour.cs* derives from *SmartObjectBehaviour.cs* attached to it which makes this object smart.

VARIABLES	DESCRIPTION
public Vector3 _positionAtLeftHandAnchor	Neutral position of the autoinjector in left hand where it gets anchored to
public Vector3 _rotationAtLeftHandAnchor	Neutral rotation of the autoinjector in left hand where it gets anchored to
public Vector3 _positionAtRightHandAnchor	Neutral position of the autoinjector in right hand where it gets anchored to
public Vector3 _rotationAtRightHandAnchor	Neutral rotation of the autoinjector in right hand where it gets anchored to

FUNCTIONS	DESCRIPTION
override public void AttachToHand()	Handles the autoinjector's placement in the the hand when it is touched
override public void RemoveFromHand()	Handles the autoinjector's removal from the hand when release gesture is perfomed

Otoscope

Drawers

Fig: Drawer closed (left) and open (right)

1. This has the *DrawerBehaviour.cs* derives from *SmartObjectBehaviour.cs* attached to it which makes this object smart.

VARIABLES	DESCRIPTION
public Vector3 _positionAtLeftHandAnchor	Neutral position of the drawer in left hand where it gets anchored to
public Vector3 _rotationAtLeftHandAnchor	Neutral rotation of the drawer in left hand where it gets anchored to
public Vector3 _positionAtRightHandAnchor	Neutral position of the drawer in right hand where it gets anchored to
public Vector3 _rotationAtRightHandAnchor	Neutral rotation of the drawer in right hand where it gets anchored to

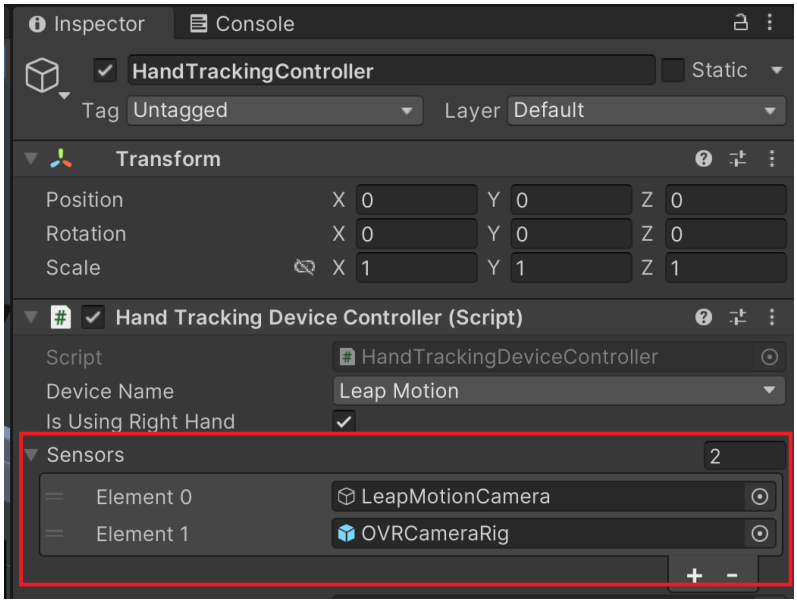
FUNCTIONS	DESCRIPTION
override public void AttachToHand()	Handles the drawer's placement in the the hand when it is touched
override public void RemoveFromHand()	Handles the drawer's removal from the hand when release gesture is perfomed

Connecting new devices with hand tracking

1. Locate/download the hand tracking module inbuilt to the new device from the Unity Asset Store and add the hand tracking packaged prefab to the main scene.
2. Create a *<DeviceName>HandTrackingProvider.cs* script of type *OpenMGHandTrackingProvider* parent class.
3. Interface the tracking functionalities of the new device with the generic hand model (see *LeapMotionHandTrackingProvider.cs* for example reference).
4. Under the *HandTrackingController* gameobject in the scene, open the *HandTrackingDeviceController* script:
    a. Add the new device name under the *DeviceName* enum.
    b. Create a similar case as the highlighted section below for the new device.

```
void Awake()
{
    switch (m_DeviceName)
    {
        case DeviceName.LeapMotion:
            {
                m_DataProvider = new LeapMotionHandTrackingProvider();
                m_DataProvider.m_DataSource = sensors[(int)DeviceName.LeapMotion];
                sensors[(int)DeviceName.Quest2].SetActive(false);
                break;
            }
        case DeviceName.Quest2:
            {
                m_DataProvider = new Quest2HandTrackingProvider();
                m_DataProvider.m_DataSource = sensors[(int)DeviceName.Quest2];
                sensors[(int)DeviceName.LeapMotion].transform.parent.gameObject.SetActive(false);
                //display_joints = true;
                break;
            }
    }
```

5. Now you should have your new device placeholder added to the list of sensors in the *HandTrackingDeviceController* gameobject.



6. Pull the hand tracking packaged prefab added to scene in step 2 to the sensor placeholder in step 6.

Licensing

MIT License