

# Creating a Hand Tracking Demo for the Quest 2 with Unity: A Step-by-Step Guide

Cast spells like a “Sorcerer Extreme” with hand gestures.



## 1. Project Overview and Goals

Hand tracking is a game-changing feature in the world of virtual reality (VR), allowing developers to create more immersive and interactive experiences. In this tutorial, we will walk you through creating a demo scene that utilizes hand tracking with the Meta Quest 2, Unity and C#. This guide is intended for developers who are new to the Quest 2 platform but have some familiarity with the Unity editor and the C# programming language.

This guide provides step-by-step instructions to develop a VR experience that enables users to cast magical spells using hand gestures. A key feature of this demo is the implementation of a hand tracking system that accurately captures and interprets the state of each finger – whether it is flexed or extended. The state of each finger is represented as a binary digit, with a '1' indicating an extended finger and '0' showing a flexed one. Each unique hand gesture, therefore, encodes a different binary number. The magic comes alive when the user forms different gestures with their hands. Each gesture generates a star-shaped magical effect with as many points as the number encoded by the hand gesture.

Upon concluding this guide, developers will gain valuable insights into VR development with a keen emphasis on hand tracking technology. They will gain familiarity with the Oculus SDK and learn how to effectively configure its various prefabs, which play a crucial role in establishing the VR environment and enabling immersion and interactivity. Developers will acquire a foundational understanding of how to access and interpret data related to the positions of hands and fingers. They will learn how to apply this information to drive events and interactions within the VR world, creating a more immersive and engaging user experience.

Additionally, this guide will provide guidance on the setup requirements and best practices for developing VR applications in Unity for the

Oculus Quest 2. From understanding hardware and software requirements to configuring project settings and optimizing resolution, developers will gain a strong foundation in the end-to-end process of creating, building, and deploying VR applications on the Oculus Quest 2 platform.

## 2. Quest Hardware and Software Requirements

### Hardware

The Meta Quest 2, formerly known as the Oculus Quest 2, is an advanced all-in-one virtual reality (VR) headset developed by Meta Platforms Inc. (previously Facebook Inc.). It offers a powerful and immersive VR experience without the need for an external PC or console. With its high-resolution display, precise head tracking, and intuitive hand controls, the Quest 2 has become a popular choice for developers and enthusiasts alike. It supports a wide variety of applications, including gaming, social networking, and productivity tools, making it a versatile and accessible device for users.



Meta Quest 2 with controllers

Developing VR experiences in Unity for the Meta Quest 2 typically requires testing and iterating directly on the device. However, it is also possible to play Unity scenes from the editor via the “Play” button using a suitable USB cable. To achieve this, you will need a high-quality cable, such as the official Meta Quest Link Cable or any third-party cable that meets the necessary specifications. The cable should support data transfer speeds of at least 5 Gbps and, preferably, have a length of 3 meters or more to provide ample freedom of movement during use. The connector on the Quest 2 is of type USB-C and it needs to be connected to a port on the PC that supports, at a minimum, the USB 3.0 standard.



Meta Quest 2 with USB-C cable for Quest Link

## Software

Please note that while developing for the Oculus Quest 2 on a Mac is possible, this guide will assume that you are using a PC. This is because some features and functionality may be slightly different or not as optimized when using a Mac.

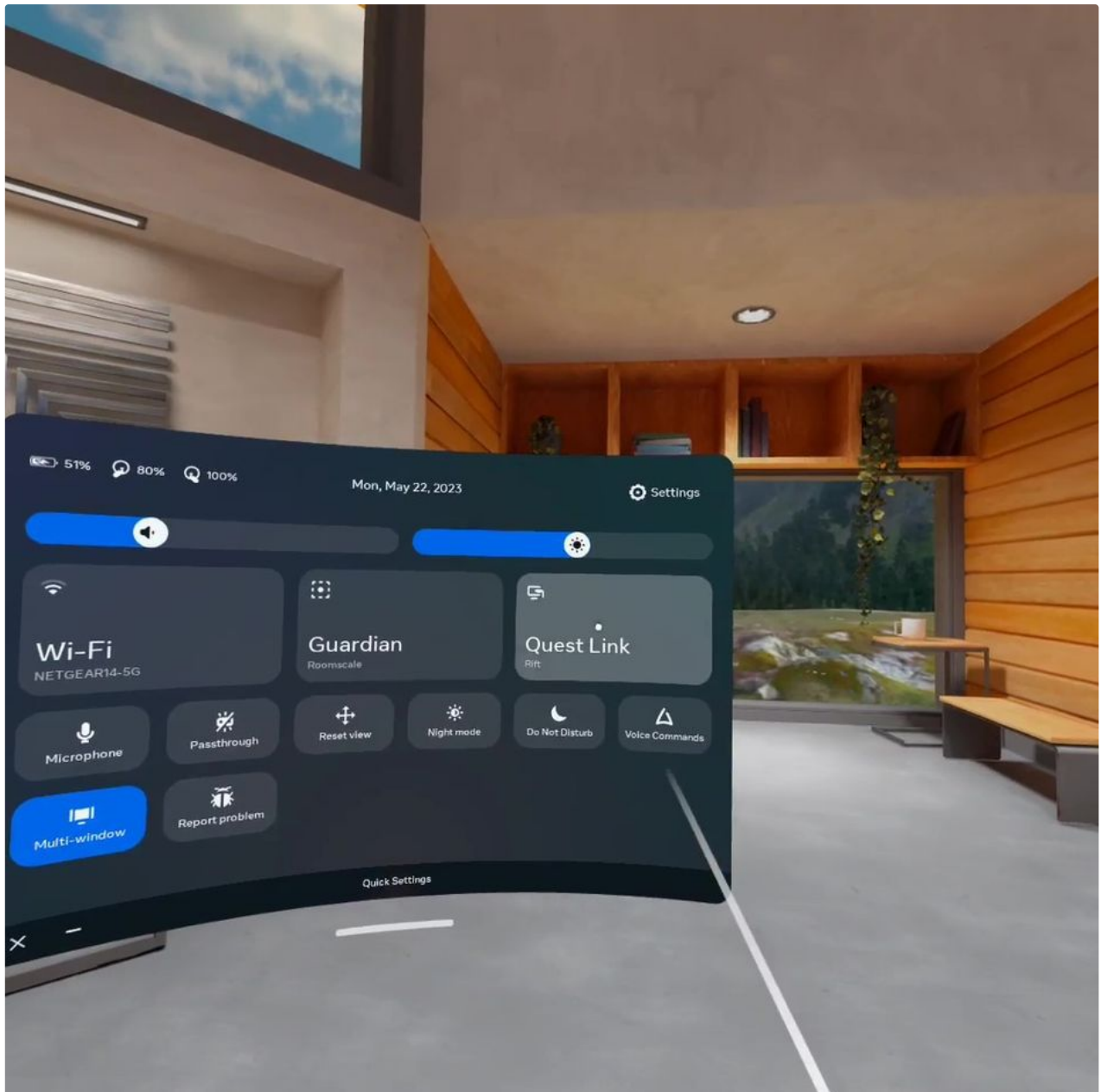
To set up your Meta Quest 2 for testing through a USB cable, follow these step-by-step instructions:

1. Create an Oculus Developer account:
  - a. Visit the Oculus Developer website and sign up for a free account using your Meta/Facebook login credentials:  
<https://developer.oculus.com/>
2. Install the Oculus PC software:
  - a. Download and install the Oculus PC app on your computer:  
<https://www.oculus.com/setup/>
3. Enable Developer Mode on your Quest 2 via your smartphone Quest App (following instructions are for version 215.0):
  - a. Open the Oculus phone app and sign in with your Meta/Facebook account.
  - b. Tap the Menu button at the bottom right of the screen.
  - c. Tap the Devices button and select your Quest 2 device.
  - d. Tap Headset Settings > Developer Mode and enable the Developer mode switch.
4. Enable hand tracking on the Quest 2:
  - a. Put on the Meta Quest 2 headset and press the Oculus button on your right controller to go to the Home screen, if not already there.
  - b. Click on the Quick Settings section (left side) of the Home Screen bar.
  - c. Open the Settings menu by clicking on the gear icon on the top right.
  - d. Navigate to Privacy > Device Permissions.
  - e. Toggle on Hand Tracking to enable the feature.
5. Install the appropriate drivers on your PC:

Download and install the Oculus ADB (Android Debug Bridge) drivers to establish a connection between your Quest 2 and your computer. These drivers allow your PC to communicate with your Android-based Quest 2, so you can install apps, transfer files, and run commands directly on the device from your PC. You can download the drivers here:

<https://developer.oculus.com/downloads/package/oculus-adb-drivers/>

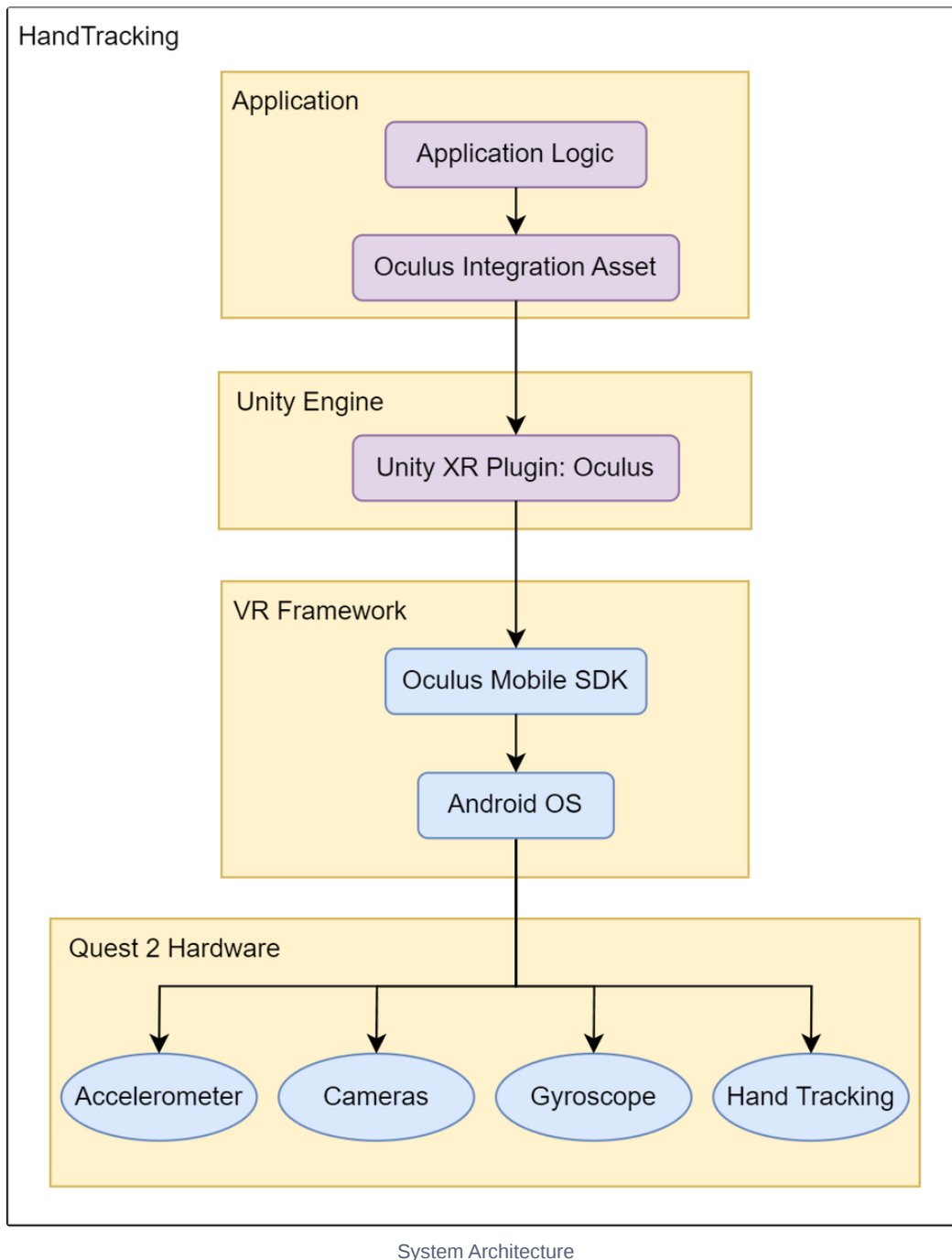
6. Connect your Quest 2 to your PC:
  - a. Use a high-quality USB-C cable to connect your Quest 2 to a USB 3.0 or higher port in your computer.
7. Enable Oculus Link on your Quest 2:
  - a. On your Quest 2 headset, you should see a prompt asking if you want to enable Oculus Link. Confirm by selecting "Enable." If the prompt does not appear, you can manually enable the link from the Quick Settings menu on your Quest 2.



Launch Quest Link from Quick Settings Menu

### 3. System Architecture

The architecture for our hand tracking demo can be broadly divided into four main layers: the Quest hardware, the VR framework, the Unity engine, and the Application layer.



#### Quest hardware

The Meta Quest 2 headset is equipped with a high-resolution display, built-in cameras for inside-out tracking (i.e., tracking the user's position and movement within a physical space), a gyroscope, an accelerometer and hand tracking capabilities. The hardware also includes the Touch Controllers for user input but we will not be using those in our demo. Together, these components enable a standalone VR experience without external sensors or a connection to a PC.

## The VR framework

The Android OS serves as the foundation of the VR framework, offering hardware compatibility and resource management. The Oculus Mobile SDK acts as a bridge, enabling features like stereo rendering, head tracking, hand tracking, and touch controllers tracking.

## Unity engine

Unity is a powerful game engine and development platform that provides a set of tools and APIs for creating interactive 3D and 2D experiences. It includes a rendering engine, physics engine, and scripting capabilities via C#. Unity's XR platform provides a set of APIs and components that enable developers to build VR experiences that can run on multiple VR platforms, including the Meta Quest 2.

## Oculus Integration asset


The Oculus Integration asset is a Unity package that simplifies the process of developing VR experiences for the Meta Quest 2 platform. It includes a set of prefabs, scripts, and tools that are specifically designed for the Quest hardware and the Oculus VR framework. Some key components include:

1. **OVRCameraRig**: Manages head tracking and the camera associated with the user's point of view in the 3D environment.
2. **OVRInputModule**: This component is responsible for handling input events from the Oculus Touch Controllers. It captures various user inputs, including button presses, joystick movements, and touchpad interactions, translating them into events within the VR experience.
3. **OVRHand**, **OVRSkeleton**, and **OVRMesh**: These components are used for the hand tracking capabilities of the Meta Quest. **OVRHand** is used to represent the hand, **OVRSkeleton** is used for skeletal tracking of the fingers, and **OVRMesh** represents the visual 3D mesh of the hand.
4. **OVRAvatar**: Provides a customizable avatar system for multiplayer experiences.
5. **OVRBoundary**: Supports the visualization and management of the Quest's Guardian System.

## 4. Creating the Demo Scene

We will start by creating a new Unity project and setting up the basic scene for the demo with a floor on which to stand and a background environment to provide some visual contrast for the hands and magical effects.

### New Unity project

 Note: The version of Unity used for this guide was 2021.3.11f1

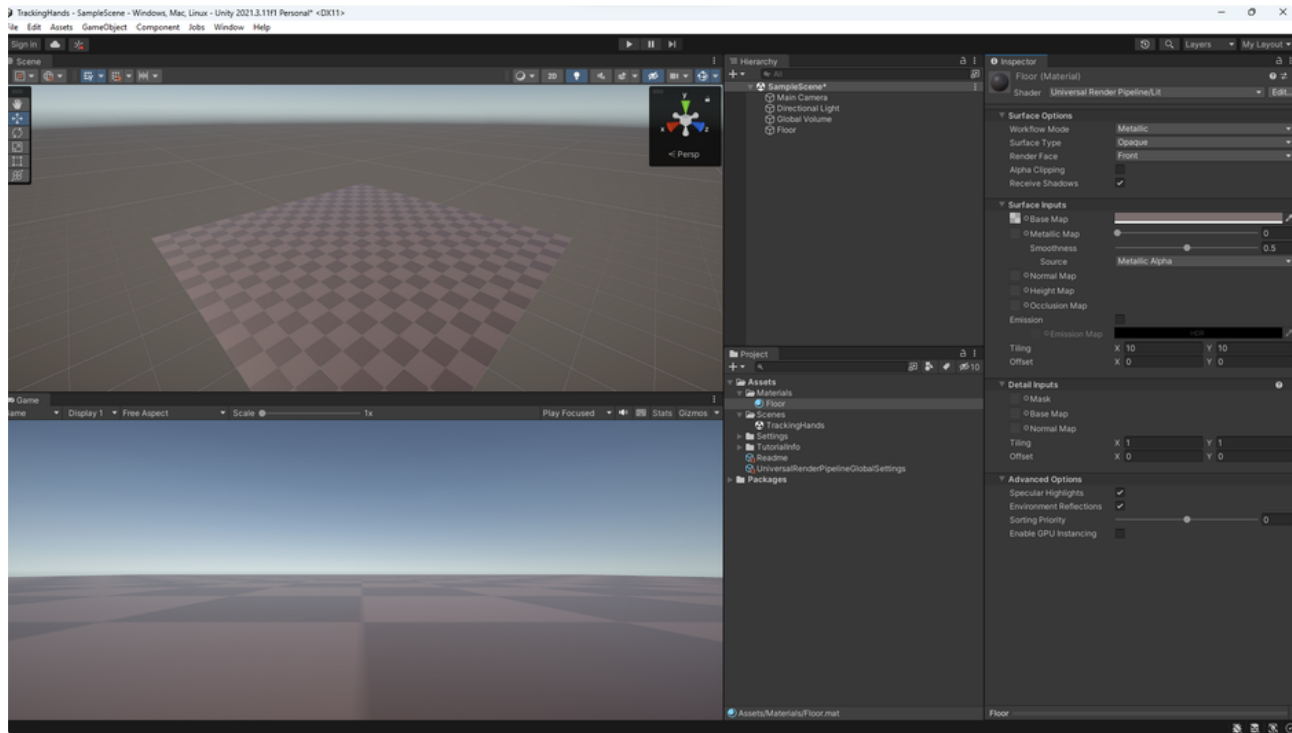
1. Open the Unity Hub and click on "New".
2. Choose the 3D (URP) template.
3. Name your project, select a location to save it and click "Create Project".
4. Adjust the layout of the editor to your preference.
5. In the Project pane, find the Scenes folder and rename SampleScene to your preference. We will call our scene "TrackingHands". You will see a pop up alerting you about the open scene having been changed. Click "Ignore".

### Scene Environment

This step may seem unnecessary for a demo focusing on hand tracking, but the relationship between the real world ground and the perceived ground level in VR is an important contributor to the experience of immersion.

1. In the Hierarchy pane, right-click and select 3D Object > Plane, and name it "**Floor**".
2. Scale the plane as needed (e.g., set **X** and **Z** scale to **10**).
3. Create a material for your floor:

- a. Create a **"Materials"** folder under assets and create a new material named **"Floor"**. The default shader, "Universal Render Pipeline/Lit", will work.
  - b. Configure the Base Map color and texture to your preference. We used color: HEX **#7B6F6F** and texture: **Default-Checker-Gray**.
4. Assign the material by dragging it from the Project pane and dropping it onto the Floor GameObject on the Scene or the Hierarchy window.



Creating the Demo Scene

Next, we will adjust the skybox in the scene to ensure a good contrast against the rest of the elements we will create later. This step is optional, since the same effect could be achieved tweaking the colors of the fonts and materials displayed in the demo.

1. Create a new material and call it **Skybox**.
2. Set its shader to **"Skybox > Procedural"**.
3. Go to Window > Rendering > Lighting : Environment, and drag the Skybox material from the Project window onto the **Skybox Material** slot on the Lighting window.
4. Adjust the **Sky Tint** to a dark blue, such as HEX **#0078C3**.
5. Adjust the **Atmosphere Thickness** to around **0.6** to make sure the horizon is not too bright.

## 5. Configuring the project for Quest 2

### Updating Project settings

1. Go to Edit > Project Settings.
2. Select **XR Plug-in Management** on the left pane and click the **Install XR Plug-in Management** button that will appear on the right pane.
3. Once installed, the right pane will show new options. Under the Android settings tab (Android icon), enable **oculus** in the "Plug-in Providers" section.
4. A new option will become available on the left pane under the **XR Plug-in Management** named **Oculus**.

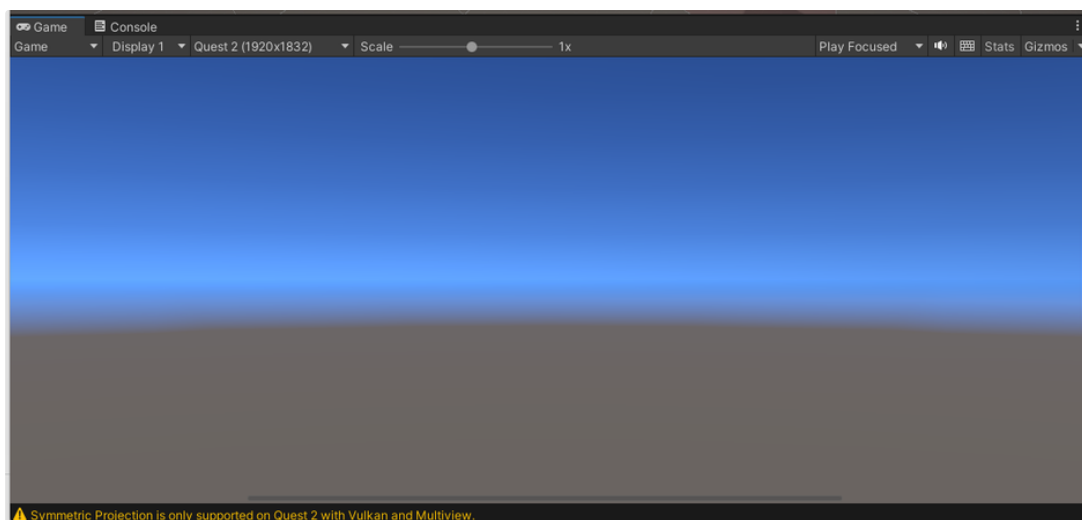


- a. Select **Oculus** on the left pane.
- b. Under the **Android** tab > **Target Devices**, uncheck the box for **Quest** (which is for the Quest 1) and make sure **Quest 2** is selected.
5. In order to be able to test your App from the Unity editor via the “Play” button, go back to the **XR Plug-in Management**, switch to the PC settings tab (monitor icon), and enable the **Oculus** check box.
6. Click on **Player** on the left pane and navigate to the Android settings Tab (Android icon) > Other Settings drop-down > Rendering section. Make sure the **Auto Graphics API** box is unchecked and OpenGL ES2 is not in the list of Graphic APIs. If it is there, select it and remove it with the “-” button. We will keep OpenGL ES3 in that list.
7. Under the Identification section of the same window, we will enter a package name following the convention **com.YourCompanyName.YorProductName**.
8. Still in this section, set the **Minimum API Level** to **Android 10.0 (API level 29)** or higher.
9. In the Configuration section, change the **Scripting Backend** from Mono to **IL2CPP**.
10. Still in this section, check the box for **ARM64** under “Target Architectures”.

### Adjusting the resolution on the Game window

Set the Game window of the Unity editor to match the resolution of the Quest 2. This ensures that tests performed in Play mode are consistent with what you will see when deploying to the device.

1. On the resolution drop down of the Game window, click the + button at the end of the list. The default resolution on the list may just be “Free Aspect”.
2. Label the entry “**Quest 2**” and enter the values **X = 1920** and **Y = 1832**.
3. Next to the resolution setting, keep the **Scale** slider at **1x**.



Quest 2 resolution on Game window

## 6. Setting Up Oculus Integration SDK

### Downloading and installing the Oculus Interaction package

1. Go to the Unity Asset Store and search for “Oculus Integration” in the search bar at the top or click on the link below.  
<https://assetstore.unity.com/packages/tools/integration/oculus-integration-82022>
2. On the asset's page, click the “Add to My Assets” button. If you haven't logged in to your Unity account, you may be prompted to do so.

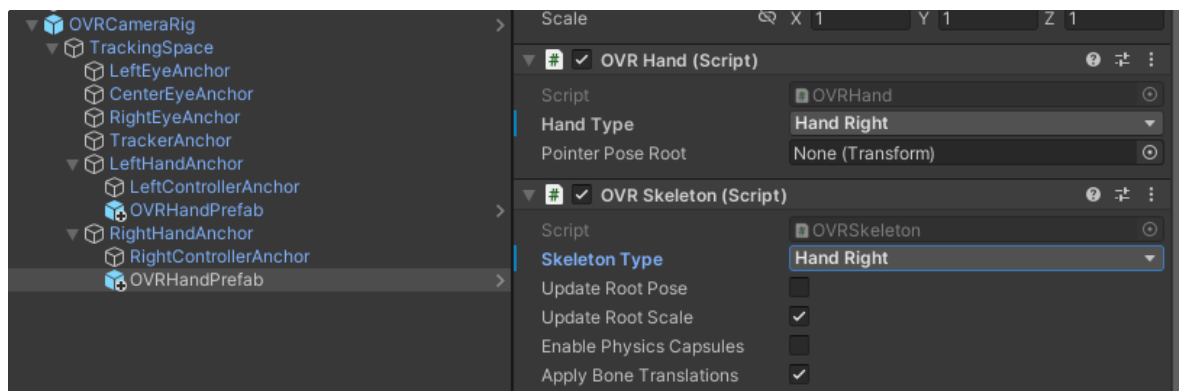


3. A banner will pop on the top of the browser indicating that the asset has been added. Click on the button in the banner labeled "Go to Unity". A window pop-up from the browser may require you to allow it to open the Unity editor.
4. A new Package Manager window in Unity will show you the Oculus Integration package. Click the "Import" button at the bottom right of the window.
5. In the Import Unity Package window, ensure all necessary files are selected (you can leave them all checked by default) and click the "Import" button. Note that the editor may need to restart and this in turn may trigger an "asset cleanup" and plugin updates. Agree to all and let the editor restart.
6. Under Edit > Project Settings, you will see a new option on the left pane called **Oculus**. Click on it and check on the right pane that there are no recommended fixes labeled with a red exclamation mark. If there are, click on the button next to them to apply the fix.

## Adding OVR components to the scene

Next, we will add the **OVRCameraRig** from the Oculus Integration asset to our scene:

1. In the Project window navigate to the folder:  
Assets / Oculus / VR / Prefabs /
2. Locate the **OVRCameraRig** prefab in the folder.
3. Click and drag the **OVRCameraRig** prefab into your scene's Hierarchy window.
4. The **OVRCameraRig** comes with a script called **OVRManager** where we can configure a number of things.
  - a. Under **Tracking > Tracking Origin Type**, select **Floor Level** for an accurate position of the user's point of view based on the distance between the VR headset and the floor. This distance is defined in the "Guardian" feature of the Quest device.
  - b. In the **Quest Features > General** section, set **Hand Tracking Support** to **Controllers And Hands**.
5. Delete the default Main Camera from the scene, as the OVRCameraRig includes its own camera setup.
6. Add the OVRHand prefabs from the Oculus Integration asset:
  - a. Locate the Oculus Integration asset folder containing the **OVRHandPrefab** at:  
Assets / Oculus / VR / Prefabs /
  - b. Expand the **OVRCameraRig** object in the Hierarchy window.
  - c. Locate the **LeftHandAnchor** and **RightHandAnchor** child objects under the **TrackingSpace** object. These objects represent the positions of the left and right hands in the VR scene.
  - d. Drag the **OVRHandPrefab** from the Oculus Integration asset folder onto the **LeftHandAnchor** and **RightHandAnchor** objects in the Hierarchy window. Note that the prefab is configured for the left hand by default, so we will need to make some adjustments for the right hand.
  - e. Under **RightHandAnchor > OVRHandPrefab > OVR Hand (script)**, set **Hand Type** to **Hand Right**.
  - f. Under **RightHandAnchor > OVRHandPrefab > OVR Skeleton (script)**, set **Skeleton Type** to **Hand Right**.



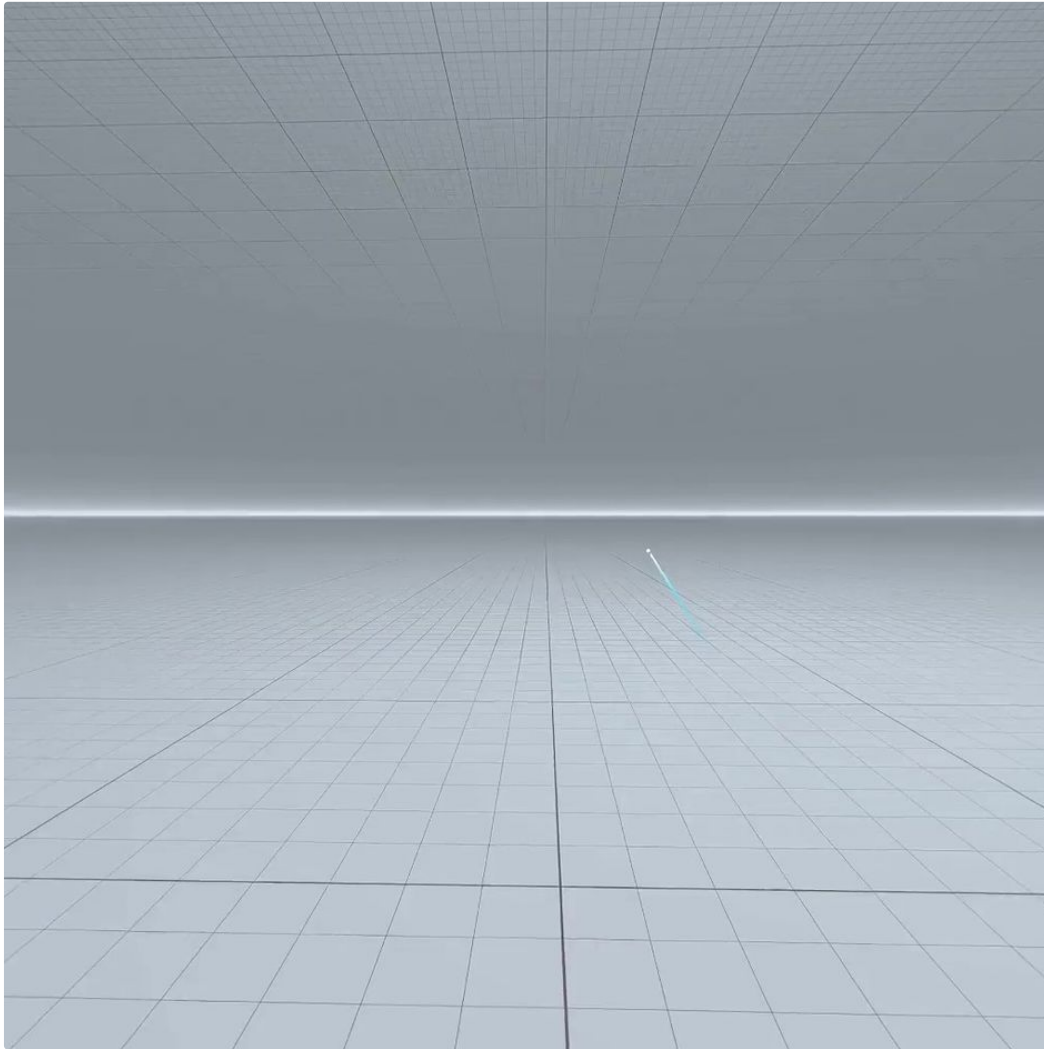
OVRHandPrefab Right Hand Configuration

7. Update the default material for the hand meshes so it works with URP:

- a. Find the material for the OVR Hands in the Project window at:  
Assets / Oculus / VR / Materials / BasicHandMaterial
- b. In the **Shader** drop-down (which will say "Legacy Shaders/Diffuse") select a shader compatible with URP like **"Universal Render Pipeline > Unlit"**.
- c. In the **Surface Inputs** section, click the radio button next to **Base Map** and choose the texture called **"HandTracking\_uvmap\_2048"**. This texture will help visualize the mesh applied to the OVRHand.

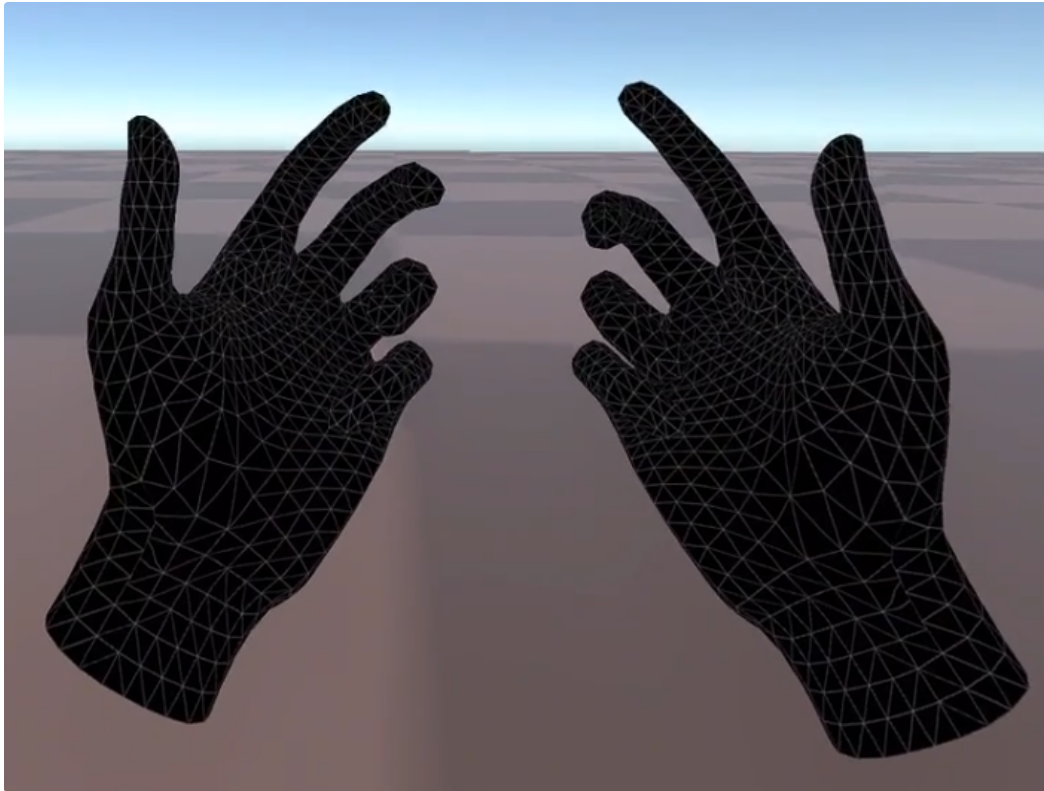
### Testing our scene in Unity Play Mode

With your USB-C cable connected to the Quest 2 and your PC, go to your Quest home and enable Quest Link.



Quest 2 view with Quest Link enabled

Click Play on the Unity editor and wait for your app to load. After a few seconds, you should be able to see your hands in VR clad in the the default black material from the Oculus Integration asset.



Tracking your hands with the Oculus Integration Asset

### Building and running the scene on the Quest 2

Next we will build the scene and upload the APK (Android Application Package) file to the Quest 2 and test our scene running directly on the device. We will need the USB-C cable to upload the APK file but we do not need to be running Quest Link. Once the app is loaded, the USB-C cable is not needed since the software is running directly on the Quest hardware.

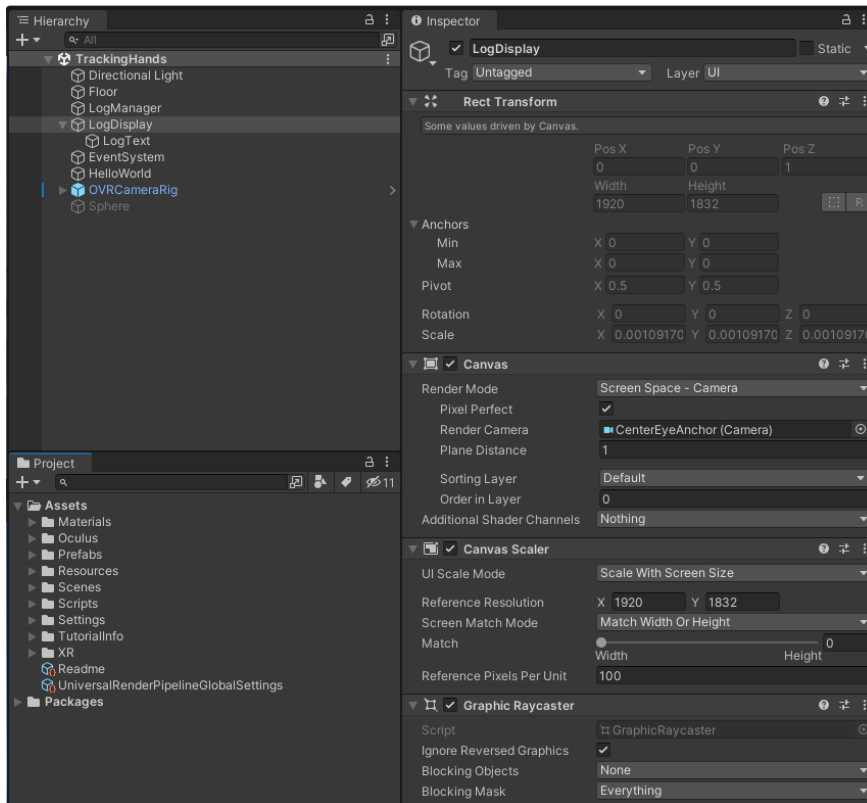
1. Go to File > Build Settings.
2. Select “Android” as the target platform and click “Switch Platform”.
3. Click “Add Open Scenes” button to add your current scene.
4. Under “Run Device” select your Oculus Quest 2 device.
5. Click “Build And Run” button.
6. Create and select a folder for your builds and give a name to this build.
7. Once you press “Save”, the project will compile and get uploaded to your Quest device.
8. Put your headset on and test the app.

## 7. Implementing a HUD for in-game debug logging

When testing VR apps, it is useful to be able to log messages within the game to avoid having to remove the headset to take a look at the console on the Unity editor. In the next steps we will set up a UI element to serve as a heads-up display (HUD) and we will code the logic to pass messages from any script in the game to this display.

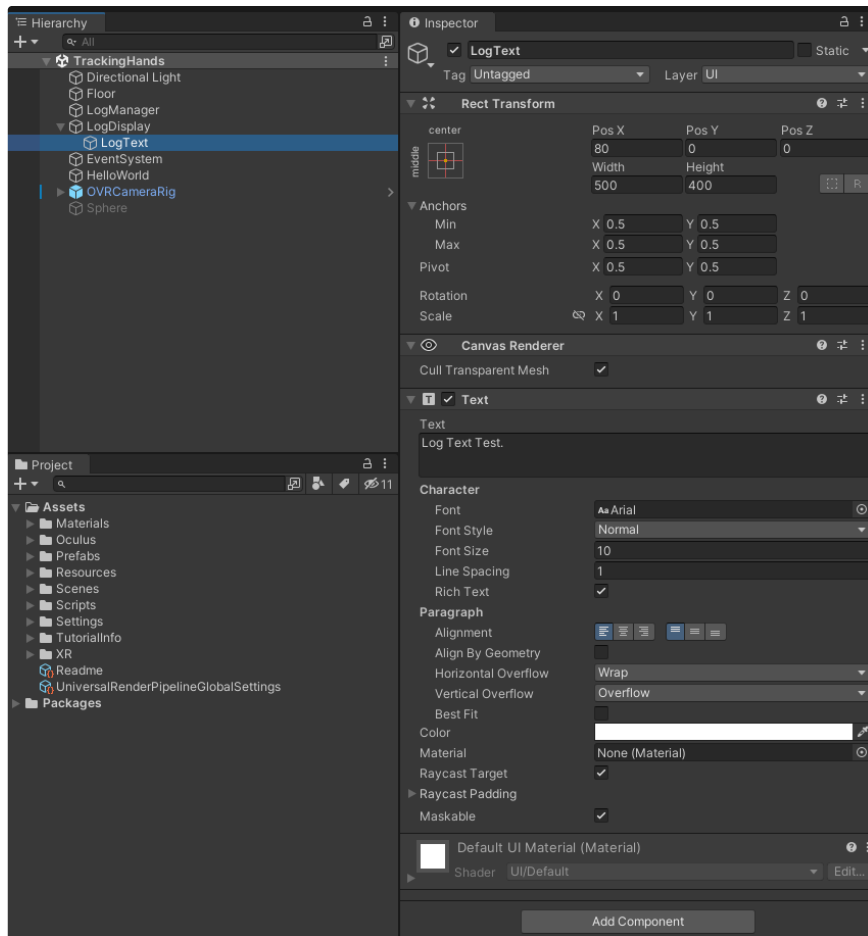
1. Create an empty object in the scene called `LogManager`. This object will hold a script that will act as a hub for all the logs sent by the different scripts in our project.

2. Create a UI > Canvas object in the scene called **LogDisplay** :
  - a. Select the **LogDisplay** object and under **Canvas > Render Mode** , select **Screen Space - Camera**. This ensures that the UI elements are positioned and rendered in screen space, allowing them to overlay the game view and remain consistent regardless of camera movement or perspective.
  - b. Check the box for **Pixel Perfect** for a more crisp font.
  - c. From the **OVRCameraRig** hierarchy, drag and drop the **CenterEyeAnchor** (which contains a camera component) onto the **LogDisplay > Canvas > Render Camera** slot.
  - d. Set **Plane Distance** to **1** unit to achieve the HUD effect.
  - e. On the **LogDisplay > Canvas Scaler** component, set the **UI Scale Mode** to **Scale With Screen Size**.
  - f. Set the **Reference Resolution** to **1920 x 1832** to match the resolution on the Quest 2.



LogDisplay settings

3. Select the **LogDisplay** GameObject in the Hierarchy and add an empty child object called **LogText** :
  - a. Adjust the **LogText -> Rect Transform** to have **Pos X = 80** , **Width = 500** and **Height = 400**. This will keep the text centered in the player's view in VR while still being out of the way.
4. Add a **Text** component to the **LogText** GameObject:
  - a. Set the **LogText > Text > Font Size** to **10**.
  - b. In the Text box of the Text component, enter a default text to see how it will be displayed on screen.



LogText settings

5. Set up the messaging infrastructure to allow other pieces of code to send information to the log display:
  - a. Create a script called `LogManager` and attach it to the empty object we created earlier of the same name.
  - b. Open `LogManager.cs` with your favorite editor by double-clicking on the script.
  - c. Replace the existing code with the following implementation of a singleton, to allow other scripts in your project to access and log messages. This script not only logs messages to the console but also maintains a list of messages that can be accessed by any other script. A script attached to our UI object, `LogDisplay`, will retrieve and display these messages.

```

1 using UnityEngine;
2 using System.Collections.Generic;
3
4 public class LogManager : MonoBehaviour
5 {
6     public static LogManager Instance { get; private set; }
7
8     private List<string> logMessages = new List<string>();
9
10    private void Awake()
11    {
12        if (Instance == null)
13        {
14            Instance = this;
15            DontDestroyOnLoad(gameObject);
16        }
17        else

```

```

18     {
19         Destroy(gameObject);
20     }
21 }
22
23 public void Log(string message)
24 {
25     logMessages.Add(message);
26     Debug.Log(message);
27 }
28
29 public List<string> GetLogMessages()
30 {
31     return logMessages;
32 }
33 }

```

The `LogManager` maintains a list called `logMessages` to store the logged messages. Upon initialization, the `Awake()` method checks if an instance of the `LogManager` already exists. If not, it sets the current instance as the singleton and ensures that the `LogManager` object persists between scene loads. If an instance already exists, the duplicate instance is destroyed. The `Log()` method adds a message to the `logMessages` list and logs it to the console using `Debug.Log()`. The `GetLogMessages()` method returns the list of log messages.

- d. Create a new script called `LogDisplay` and attach it to the `GameObject` of the same name.
- e. Open `LogDisplay.cs` and replace the default code with the following implementation:

```

1 using UnityEngine;
2 using UnityEngine.UI;
3
4 public class LogDisplay : MonoBehaviour
5 {
6     [SerializeField] private Text logText;
7     [SerializeField] private int maxDisplayedMessages = 20;
8
9     private void Update()
10    {
11        DisplayLog();
12    }
13
14    private void DisplayLog()
15    {
16        var logMessages = LogManager.Instance.GetLogMessages();
17        int startIdx = Mathf.Max(0, logMessages.Count - maxDisplayedMessages);
18        string displayedLog = "";
19
20        for (int i = startIdx; i < logMessages.Count; i++)
21        {
22            displayedLog += logMessages[i] + "\n";
23        }
24
25        logText.text = displayedLog;
26    }
27 }

```

The `LogDisplay` class is responsible for displaying log messages in a UI text element. It retrieves log messages from the `LogManager` singleton and updates the UI with the latest messages based on the specified maximum display limit.

f. Select the `LogDisplay` GameObject and click on the radio button next to the `Log Text` field.

g. From the selection window pop-up, select the `LogText` object.

6. Test the log screen with a Hello World message:

a. Create an empty object on the scene and name it `HelloWorld`.

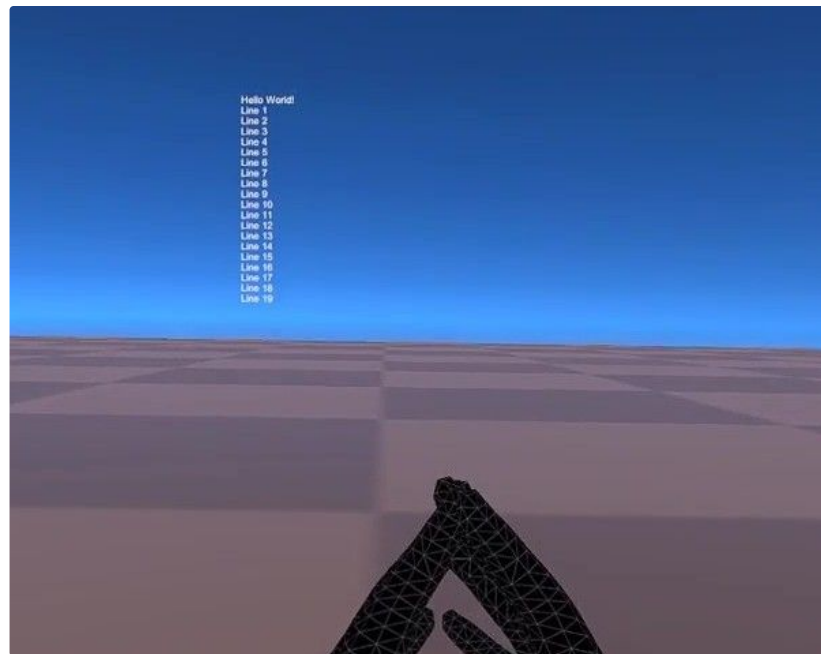
b. Create a new script called `HelloWorld` and attach it to the GameObject of the same name.

c. Edit the `HelloWorld.cs` file and replace its content with the following:

```
1 using System.Collections;
2 using System.Collections.Generic;
3 using UnityEngine;
4
5 public class HelloWorld : MonoBehaviour
6 {
7     // Start is called before the first frame update
8     void Start()
9     {
10         LogManager.Instance.Log("Hello World!");
11
12         // Fill 19 additional lines to give a sense of space taken on screen.
13         for (int i = 1; i <= 19; i++)
14         {
15             LogManager.Instance.Log("Line " + i);
16         }
17     }
18 }
```

Press the Play button and confirm that the the message Hello World! appears in the Game window of the editor. The script will also show some debug text for each additional line the display can show, so you can get an idea of how much screen real estate will be covered.





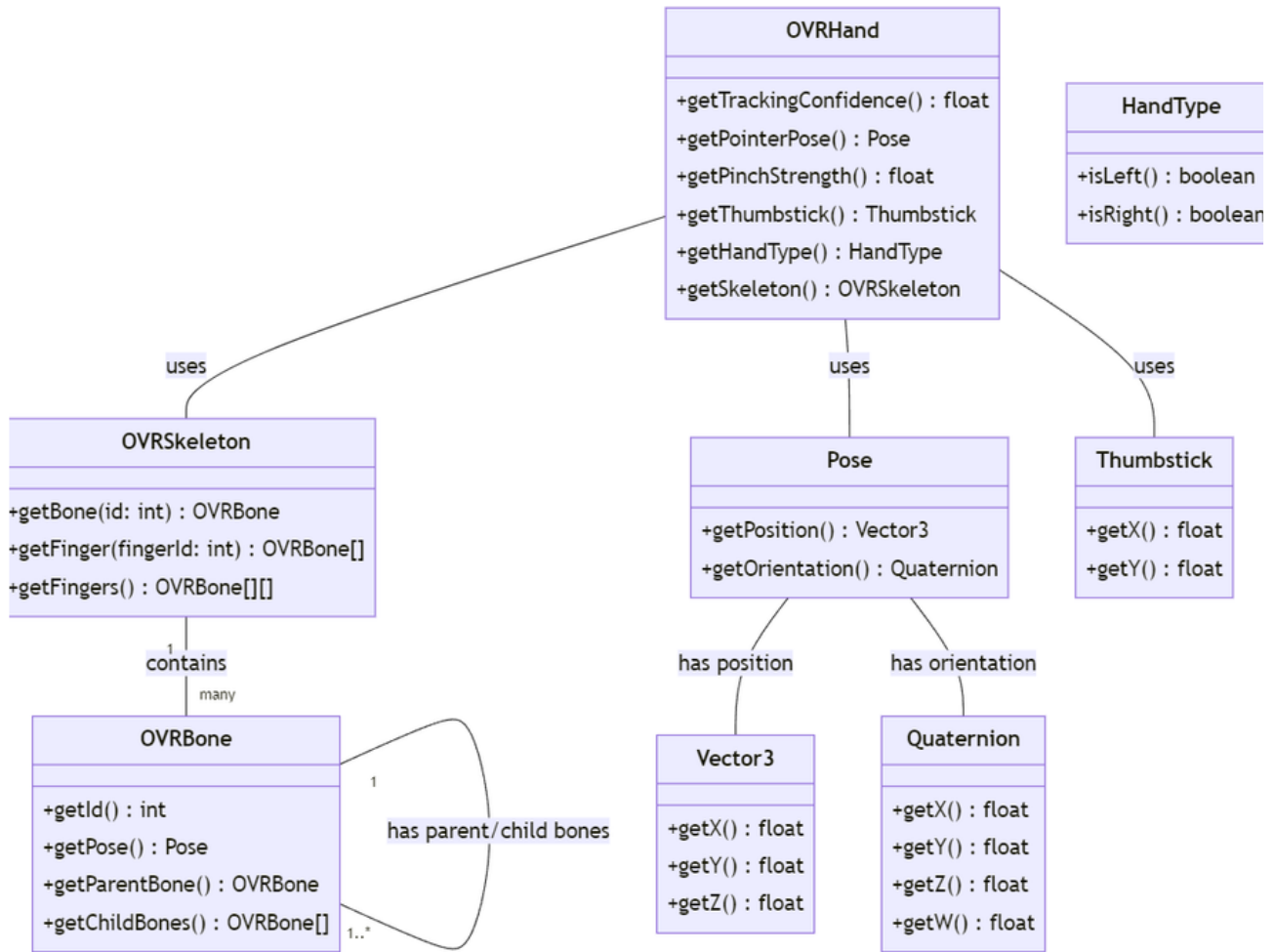
20 lines of text in LogDisplay

## 8. Building the Hand Tracking Demo

We are finally ready to bringing together the VR environment and hardware configuration we have set up with the power of the Oculus SDK and Unity engine. We will delve into the construction of our hand tracking demo, showing how we can capture, interpret, and use the state of the user's fingers to interact with the VR world. Through transforming binary representations of hand gestures into magical star-shaped spells, we will explore how hand tracking can elevate user experience, making the VR world not just immersive but also truly interactive and magical.

Typically, gesture tracking in virtual reality applications utilizes higher-level tools and prefabs that readily interpret hand movements into predefined gestures. These solutions offer simplicity and accessibility, allowing developers to easily integrate gesture-based interactions into their VR experiences. However, our approach in this tutorial is somewhat different - we aim to provide a more in-depth look into how hand tracking works at a more fundamental level. We will work directly with the `OVRSkeleton` component which will allow us to access and manipulate data related to the positions of hand skeleton bones. This approach not only provides a more nuanced understanding of hand tracking but also opens up greater possibilities for customization and precision, offering a unique insight into the mechanics of VR development.

The following class diagram provides additional details on the implementation of relevant classes within the Oculus Integration SDK.



OVRHand Class Diagram

## OVRSkeleton

OVRSkeleton is a component that provides a skeleton representation of the user's hand. It consists of a hierarchy of GameObjects that represent the bones in a hand. Each GameObject has a transform that can be used to understand the position and orientation of the corresponding bone in the hand. The skeleton's overall structure mimics the physical structure of a hand, including the wrist, palm, and individual finger bones.

## OVRHand

OVRHand is another Oculus Integration component that provides a high-level interface for hand tracking. It uses the OVRSkeleton to visualize the hand's current state, and provides additional functionality, such as tracking whether the hand is currently making a fist or pointing, and so on.

It provides information about tracking confidence, whether the hand is left or right, and it has specific functions to estimate whether a certain hand pose is being performed (for example, a pinch), although we will not be using those gesture features in our project.

## OVRMesh

OVRMesh is a visualization of the tracked hand data provided by the Oculus Quest's onboard sensors. It is a 3D, skinned mesh that is rigged to the OVRSkeleton, providing a more realistic and visually pleasing representation of the user's hand in VR. OVRMeshRender is a component that should be attached to the same GameObject as OVRMesh to handle the rendering of the mesh.

## Hand Tracking

In this section, we will dig into the heart of the hand tracking operation. Our `HandTracking` class contains a set of functions that directly interact with the `OVRSkeleton` data, specifically measuring the distances from the root of the hand (near the wrist) to the tip of each finger. These distances provide us with a quantitative way to determine if a finger is extended or flexed. The method `GetFingerExtension` is crucial in this aspect, calculating this distance for a given finger and returning it. Depending on the measurement, we can then classify each finger as flexed or extended.

Create a new script named `HandTracking` and drag and drop it on the `OVRHandPrefab` objects on the left and right hand anchors. This class is thoroughly commented and we highly recommend studying these comments for an in-depth understanding of its behavior.

### 1. Class definition, properties and fields:

```
1  using UnityEngine;
2  using System.Collections;
3  using System;
4
5  public class HandTracking : MonoBehaviour
6  {
7      // Oculus Integration SDK objects for tracking hands.
8      private OVRHand hand;
9      private OVRSkeleton handSkeleton;
10     private OVRSkeleton.SkeletonType handType;
11
12     // ID's to identify the tip bone for each finger.
13     private int[] tips =
14     {
15         (int)OVRSkeleton.BoneId.Hand_PinkyTip,
16         (int)OVRSkeleton.BoneId.Hand_RingTip,
17         (int)OVRSkeleton.BoneId.Hand_MiddleTip,
18         (int)OVRSkeleton.BoneId.Hand_IndexTip,
19         (int)OVRSkeleton.BoneId.Hand_ThumbTip
20     };
21
22     // Arrays to store calibration data for each finger's
23     // tip threshold, minimum and maximum flexion, and the number of calibration cycles.
24     private float[] tipThresholds = new float[5];
25     private float[] minFlexions = new float[5];
26     private float[] maxFlexions = new float[5];
27     private int[] cycles = new int[5];
28
29     // Boolean flags to track the state of the calibration process
30     private bool calibrationActive = false;
31     private bool allFingersCalibrated = false;
32
33     // Store the sum of all fingers interpreted as bits:
34     // finger extended -> 1 ; finger flexed -> 0.
35     // Thumb is the most significant bit and pinky the least.
36     private bool[] isFingerExtended = new bool[5];
37     private int _handSum;
38     public int HandSum
39     {
40         get { return _handSum; }
41     }
42     //...
```

## 2. Start() method:

```
1  //...
2  void Start()
3  {
4      // Get these components from the same GameObject this script is attached to.
5      hand = GetComponent<OVRHand>();
6      handSkeleton = GetComponent<OVRSkeleton>();
7
8      // Read whether this is a right hand or left hand.
9      handType = handSkeleton.GetSkeletonType();
10 }
11 //...
```

## 3. Update() method:

```
1  //...
2  void Update()
3  {
4      if (!hand.IsTracked)
5      {
6          LogHUD("Not Tracking ...");
7          return;
8      }
9
10     if (hand.IsTracked && !calibrationActive)
11     {
12         calibrationActive = true;
13         StartCoroutine(CalibrateFingers());
14     }
15
16
17     foreach (int tip in tips)
18     {
19         // Get the OVRSkeleton id for the tip bone of the current finger.
20         int id = Array.IndexOf<int>(tips, tip);
21
22         if (calibrationActive)
23         {
24             // During calibration, the GetFingerExtension method is responsible for
25             // adjusting the minFlexions and maxFlexions.
26             GetFingerExtension(id);
27         }
28         else
29         {
30             // Not in calibration, use tipThresholds to determine if finger
31             // is extended or not.
32             float extension = GetFingerExtension(id);
33
34             // store the corresponding booleans.
35             isFingerExtended[id] = extension >= tipThresholds[id];
36         }
37     }
38
39     if (!calibrationActive && allFingersCalibrated)
40     {
41         // Calculate the sum of the fingers.
```

```

42     _handSum = 0;
43     foreach (int tip in tips)
44     {
45         int id = Array.IndexOf(tips, tip);
46
47         if (isFingerExtended[id])
48         {
49             _handSum += (int)Mathf.Pow(2, id);
50         }
51     }
52     LogHUD("Sum of fingers: " + _handSum);
53 }
54
55 }
56 //...

```

#### 4. Co-routine:

```

1 //...
2 IEnumerator CalibrateFingers()
3 {
4     // Set up base flexion values per finger and initialize counter.
5     for (int i = 0; i < 5; i++)
6     {
7         // min starts at max value, max starts at min and they evolve towards
8         // each other to find the mid-point for each tip to wrist distance.
9         minFlexions[i] = float.MaxValue;
10        maxFlexions[i] = float.MinValue;
11        cycles[i] = 0;
12    }
13
14
15    float[] prevExtension = new float[5];
16
17    while (!allFingersCalibrated)
18    {
19        // Each finger will get a chance to contradict that all fingers are calibrated.
20        allFingersCalibrated = true;
21        LogHUD("Calibrating...");
22
23        for (int i = 0; i < 5; i++)
24        {
25            // Set new min and max flexions if current finger extension is smaller
26            // or greater than the current min and max.
27            float extension = GetFingerExtension(i);
28            minFlexions[i] = Mathf.Min(minFlexions[i], extension);
29            maxFlexions[i] = Mathf.Max(maxFlexions[i], extension);
30
31            // Threshold is the average of min and max flexion.
32            tipThresholds[i] = (minFlexions[i] + maxFlexions[i]) / 2f;
33
34            // Detect if the current extension crosses over the current thresholds
35            // and count that as a completed calibration cycle.
36            if (prevExtension[i] < tipThresholds[i] && extension >= tipThresholds[i])
37            {
38                cycles[i]++;
39            }

```

```

40         // Same detection for crossing under current threshold.
41         else if (prevExtension[i] >= tipThresholds[i] && extension < tipThresholds[i])
42         {
43             cycles[i]++;
44         }
45
46         // Update previous extensions.
47         prevExtension[i] = extension;
48
49         // Each finger can deny calibration complete and keep loop running.
50         if (cycles[i] < 6)
51         {
52             allFingersCalibrated = false;
53         }
54     }
55     yield return null;
56 }
57 // All fingers calibrated at this point.
58 calibrationActive = false;
59 }
60 //...

```

## 5. Utility methods:

```

1  //...
2  float GetFingerExtension(int _fingerId)
3  {
4      // Get tip bone for current finger.
5      OVRBone bone = handSkeleton.Bones[(int)tips[_fingerId]];
6
7      // Calculate the position of tip bone relative to the handSkeleton object
8      // (i.e, the root bone of the hand, near the wrist).
9      Vector3 distance = handSkeleton.transform.InverseTransformPoint(bone.Transform.position);
10
11      Vector3 distAbs = Abs(distance);
12
13      // The thumb flexes on a different plane than the rest of the fingers
14      // so we look at a different coordinate.
15      return bone.Id == OVRSkeleton.BoneId.Hand_ThumbTip ? distAbs.z : distAbs.x;
16  }
17
18  Vector3 Abs(Vector3 v)
19  {
20      return new Vector3(Mathf.Abs(v.x), Mathf.Abs(v.y), Mathf.Abs(v.z));
21  }
22
23  void LogHUD(string message)
24  {
25      LogManager.Instance.Log(this.GetType() + ":" + handType + ": " + message);
26  }
27 }
28 // END

```

This would be a good time to test and practice the hand calibration process on the Quest 2. With the USB-C cable connected to your PC, launch Quest Link on the headset and press Play on the Unity editor. As soon as the app loads, quickly extend and flex all fingers several

times. Each hand will log its own encoded number so pay attention to the label prior to the number in the logs. As a reference, remember that the the fingers, from pinky to thumb, encode 1, 2, 4, 8 and 16, respectively. Therefore, an open hand should read 31, the victory sign is  $8+4=12$ , and so on.

## Spell Effect

The next step will be to use the information from the hand gestures captured by the `HandTracking` class to control a spell. The spell will consist of a star shape floating in front of the player based on the number encoded by the hands.

1. Click on `LeftHandAnchor` and create an empty child named `SpellEffect`.
  - a. In the Inspector, set its `Transform > Position` to X, Y, Z = **0.1, 0, 0.1**.
  - b. Set its `Transform > Rotation` to X, Y, Z = **90, 90, 0**.
2. Create a new material in your Materials folder and call it `SpellEffect`.
  - a. Make sure the shader is "Universal Render Pipeline/Lit".
  - b. Give the Base Map a color of your choice. For example, HEX **#EFF300**.
  - c. Turn on Emission.
  - d. Click on the Emission Map color box and select another color. For example, RGB = **(191, 0, 3)**.
  - e. For `Intensity`, enter **3.4**.
3. Add a `Line Renderer` component to the `SpellEffect` GameObject.
  - a. Assign the new material to `Line Renderer > Materials > Element 0`. You may need to expand the Materials section to see the list.
  - b. Uncheck the box for `Use World Space` so the effect follows the hand movements.
4. Create a new script named `SpellEffect` and drag and drop it onto the `SpellEffect` GameObject.
5. We will want an exact copy of the spell effect on the right hand:
  - a. Copy and paste the `SpellEffect` GameObject onto the `RightHandAnchor`.
  - b. In the Inspector, set its `Transform > Position` to X, Y, Z = **-0.1, 0, 0.1**.

Now that we have created a material with emission, we can take it a step further by adding a bloom post-processing effect. The bloom effect will enhance the glow of our material, creating a magical visual effect. In the following steps, we will configure the bloom effect using Unity's post-processing stack.

1. Create a sphere in the scene and give it the `SpellEffect` material. We will use this object to test and adjust the glow effect.
2. On the `Global Volume` GameObject (created for you automatically when using the URP project template), click on the "New" button next to the `Volume > Profile` property. This will create a new volume profile called `Global Volume Profile` and assign it automatically to the `Profile` slot. Double click on it to see its details in the Inspector.
3. Click the "Add Override" button on the `Volume` component and select **Post-processing > Bloom**.
4. On the `Camera` component of the `CenterEyeAnchor`, we will adjust the following settings to be able to see the bloom effect:
  - a. Check the box for the `CenterEyeAnchor > Camera > Rendering > Post Processing` property.
  - b. In the `CenterEyeAnchor > Camera > Output > HDR` property, select **"Use settings from Render Pipeline Asset"**.



5. You can now go back to the global volume profile and play with the settings of the Bloom effect to adjust it to your liking. For example:
  - a. Increase `Intensity` to **0.7** (second parameter) to see the scene glow.
  - b. Increase `Threshold` to **0.6** (first parameter) to reduce the glow of the sky and horizon and leave the sphere glowing.
6. Delete or disable the sphere object when you are happy with the bloom effect.

We are now ready to write the code for our spell effect. The effect is dependent on `hand sum`, which we will read from the associated `HandTracking` class. The visual representation is updated each frame, providing a real-time, interactive magical spell effect that responds to the player's hand gestures.

We will again divide the script into sections:

1. Class definition and fields:

The class `SpellEffect` has been decorated with the attribute `[RequireComponent]`, ensuring that the `LineRenderer` component is attached to any `GameObject` this script is applied to.

```
1 using UnityEngine;
2
3 [RequireComponent(typeof(LineRenderer))]
4 public class SpellEffect : MonoBehaviour
5 {
6     LineRenderer lineRenderer;
7
8     public HandTracking hand;
9     //...
```

2. `Awake()` method:

This method runs once when the `GameObject` this script is attached to is initialized. In this method, the `lineRenderer` field is assigned by getting the `LineRenderer` component attached to the same `GameObject`. Then, we set the thickness of the line that the `LineRenderer` will draw.

```
1     //...
2     void Awake()
3     {
4         lineRenderer = GetComponent<LineRenderer>();
5         lineRenderer.startWidth = 0.01f;
6         lineRenderer.endWidth = 0.01f;
7     }
8     //...
```

3. `Update()` method:

In this method, we start by storing the current hand sum in a local variable `sum`. The position and forward direction of the player's hand are calculated and the spell effect is offset by 1 unit forward from the hand's position. The position of the `GameObject` is then updated and the `GenerateSpellEffect` method is called with the hand sum.

```

1  //...
2  void Update()
3  {
4      int sum = hand.HandSum;
5
6      // Calculate hand position
7      Vector3 pos = hand.transform.position;
8
9      // Get forward direction of the player's headset
10     Vector3 forward = OVRManager.instance.transform.forward;
11
12     // Offset the position N (float) meters forward from the midpoint
13     Vector3 offset = forward * 1f;
14
15     // Update GameObject's position
16     transform.position = pos + offset;
17
18     GenerateSpellEffect(sum);
19 }
20 //...

```

#### 4. GenerateSpellEffect( ) method:

This method uses the line renderer to generate a star pattern with a number of points defined by its parameter, `points`. If `points` is zero, the line renderer is cleared. If `points` is one, a small dash is drawn. For any other number of points, the algorithm spreads the points over a circle of radius 1 and connects them following a specific order to create intricate geometric patterns. The method tries to avoid making regular polygons by stepping over a number of consecutive points according to the first relative prime of `points`.

```

1  //...
2  void GenerateSpellEffect(int points)
3  {
4      if (points == 0)
5      {
6          // Clear the line renderer if there are no points
7          lineRenderer.positionCount = 0;
8          return;
9      }
10     else if (points == 1)
11     {
12         // Draw a small dash
13         lineRenderer.positionCount = 2;
14         lineRenderer.SetPosition(0, new Vector3(0, 0, 0));
15         lineRenderer.SetPosition(1, new Vector3(0.1f, 0, 0));
16         return;
17     }
18
19     float radius = 0.5f;
20     int steps = FindFirstRelativePrime(points);
21     lineRenderer.positionCount = points + 1;
22
23     for (int i = 0; i <= points; i++)
24     {
25         float radian = Mathf.Deg2Rad * 360f * (i * steps % points) / points;
26         float x = Mathf.Cos(radian) * radius;

```

```

27         float y = Mathf.Sin(radian) * radius;
28         lineRenderer.SetPosition(i, new Vector3(x, y, 0));
29     }
30 }
31 //...

```

## 5. Utilities:

The utilities section contains two functions for handling mathematical calculations related to relative primes, also known as coprimes. The `GCD` method calculates the greatest common divisor of two integers, which is used to determine if two numbers are coprimes (i.e., their GCD is 1).

The `FindFirstRelativePrime` method finds the first number that is relatively prime to the provided number 'n' by iterating through integers smaller than 'n', and checking if their GCD with 'n' is 1.

```

1  //...
2  int GCD(int a, int b)
3  {
4      while (b != 0)
5      {
6          int temp = b;
7          b = a % b;
8          a = temp;
9      }
10     return a;
11 }
12
13 int FindFirstRelativePrime(int n)
14 {
15     for (int i = 2; i < n; i++)
16     {
17         if (GCD(n, i) == 1)
18         {
19             return i;
20         }
21     }
22     return -1; // no relative prime found, should never happen for n > 2
23 }
24
25 }
26 // END

```

Save the script and assign the value for the `hand` field in the Inspector:

1. Locate the `LeftHandAnchor > SpellEffect (Script) > Hand` field, and drag and drop the `OVRHandPrefab` object for the left hand onto the slot.
2. Perform the same action for the right hand.

We are now ready to test the project on the Quest 2 via Unity's play button. As we did before, ensure that the device is connected to a USB 3.0 (or higher) port on your PC, via the appropriate USB-C cable, and launch Quest Link from the Quick Settings bar in your Quest 2 home

Once your hands are calibrated, you should immediately see n-pointed stars floating in front of your hands. Try moving and rotating your arms and hands to see how the shapes follow your movements. Try different hand gestures and compare the hand sum logged in the HUD with the number of points in your stars.



Congratulations! You are now on your way to becoming a VR “Sorcerer Extreme”!