AFFECTIVE COMPUTING AND INTERACTION IN VR

Corso Realtà Virtuale 2023/2024

susanna.brambilla@unimi.it

WITH UNITY V2022.3.5

AFFECTIVE COMPUTING

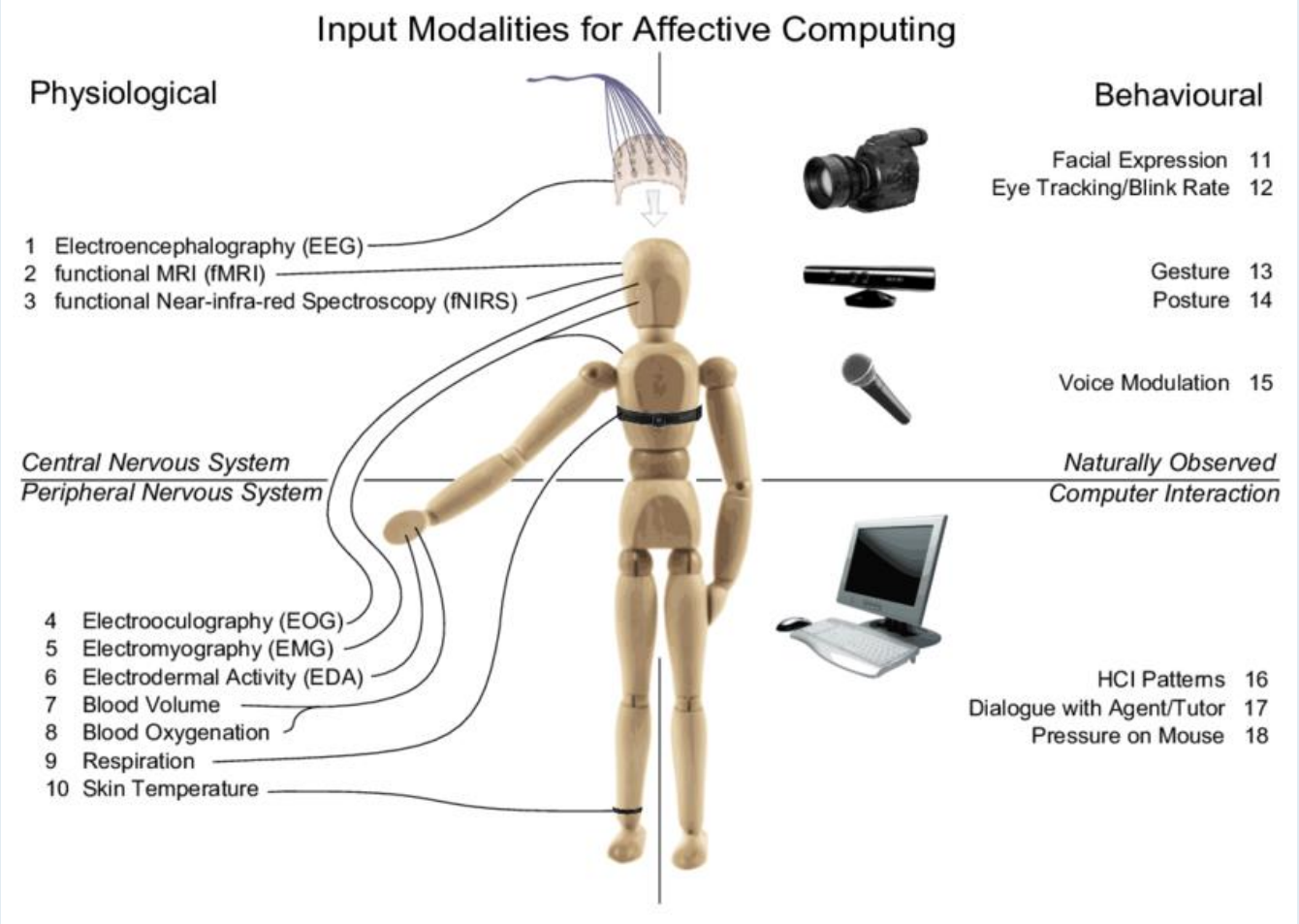
AFFECTIVE COMPUTING (AC)

Picard R. (1995)

Affective Computing (AC) aims at developing systems which can automatically recognize emotions to give adequate responses

Standard methods:

- Face tracking via cameras
- Voice tracking via microphones
- Physiological signals via sensors
- …



AC APPLICATION

- **Education:** cameras or microphones can be used to understand students' emotional states during lessons helping teachers to adapt themselves to tailor class load
- **Healthcare:** bots can monitor physical and emotional well-being of patiens or help doctors in counseling sessions
- **Marketing**: AC can analyze what makes customers engaged or their reactions to new products and companies could organize accordingly
- **Entertainement:** gaming companies can use AC for testing their games monitoring players' satisfaction level, but AC can be also useful to support the game adaptation to players' mental state
- …

AC AND VR

Why VR?

- High degree of immersion
- Elicit stronger emotional reactions
- Gather different types of data
- Realistic experience in a controlled and safe setting

It provides simulated experiences that create the sensation of being in the real world. Thus, VR makes it possible to simulate and evaluate spatial environments under controlled laboratory conditions

AC TECHNIQUES IN VR

- Face tracking
- Eye tracking
- Voice signal
- Motion-behavioral data
- …

OVRInput

OVR INPUT

With OVRInput it is possible to track input features that you can take advantage of when you design user interactions

Data:

- Positions
- Rotations
- Touch
- Buttons
- Joysticks

Control	Enumerates
`OVRInput.Button`	Traditional buttons found on gamepads, controllers, and back button.
`OVRInput.Touch`	Capacitive-sensitive control surfaces found on the controller.
`OVRInput.NearTouch`	Proximity-sensitive control surfaces found on the controller.
`OVRInput.Axis1D`	One-dimensional controls such as triggers that report a floating point state.
`OVRInput.Axis2D`	Two-dimensional controls including thumbsticks. Reports a Vector2 state.

Map Controllers | Oculus Developers

OVR INPUT USAGE

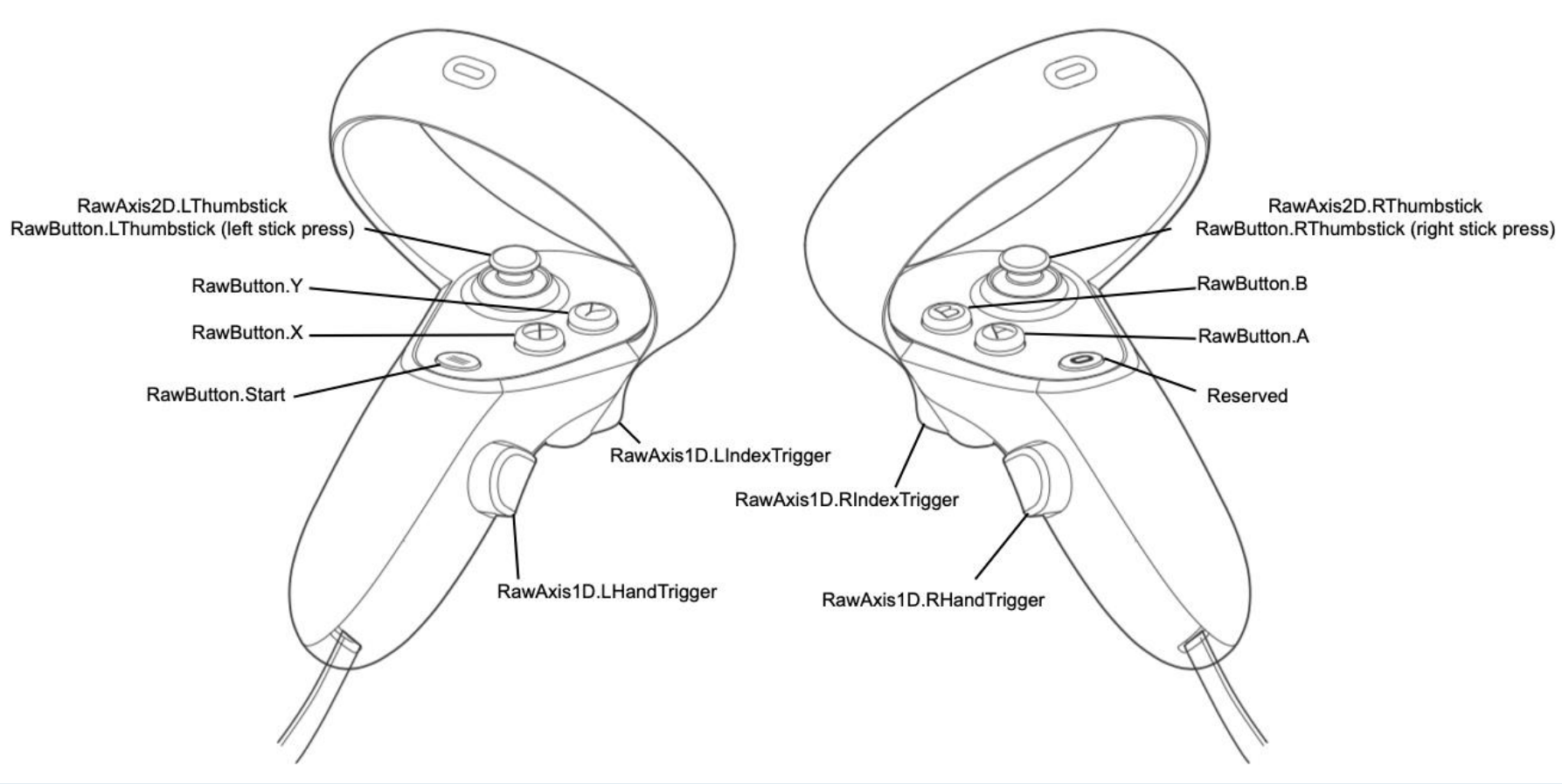
The primary usage of OVRInput is to access controller input state through:

- **Get()** queries the current state of a controller
- **GetDown()** queries if a controller was pressed this frame
- **GetUp()** queries if a controller was released this frame

You can access:

- Virtual mapping: provides a virtualized input mapping (smoothed)
- Raw mapping: directly exposes the controllers

RAW CONTROLLER MAPPING



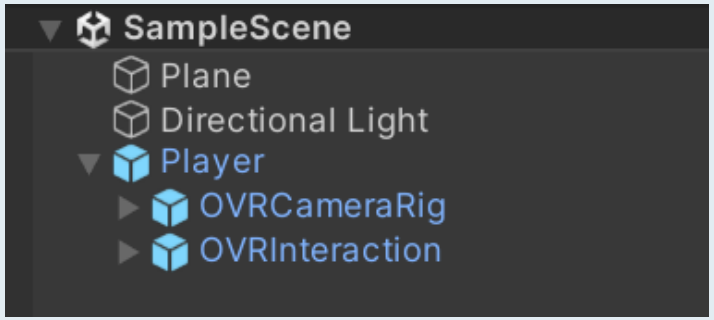
INITIALIZATION 1/2

1. Download the ‘Ex05-01’ folder from github and open the OVRInput_Demo Unity project

You will find a SampleScene without camera and with a plane and a Player prefab in the OVRInput_demo folder

2. Drag the **Player prefab** in the Hierarchy

3. Create a script and call it ‘**OVRInput_tracker**’



INITIALIZATION 2/2

We need a function to save the timestamp:

```
public static string GetTimestamp()
{
    return DateTime.UtcNow.ToString(format: "yyyy-MM-dd' 'HH:mm:ss.fff");
}
```

We need to check the presence of the controllers (both left and right) and the head:

- If left/right controller/head found -> data are valid
- If left/right controller/head not found -> data are not valid

```
if (OVRInput.IsControllerConnected(OVRInput.Controller.RTouch) && OVRInput.IsControllerConnected(OVRInput.Controller.LTouch))
```

```
if (OVRPlugin.GetNodePositionTracked( OVRPlugin.Node.Head ))
```

Put them in the FixedUpdate()

FEATURES

Different values from controllers:

- Position, velocities -> Vector3
- Orientation -> Quaternion
- Button pressure -> float (range 0-1)
- Button pressed -> int [0-1]
- Thumbstick position -> Vector2

Different values from HMD:

- Position, velocities -> Vector3
- Orientation -> Quaternion

HEAD

In order to take the reference of the player's head:

```
OVRPlugin.Node.Head
```

In order to take the head position, orientation, and velocities:

```
OVRPose pose = OVRPlugin.GetNodePose(node, OVRPlugin.Step.Render).ToOVRPose();
Vector3 position = pose.position;
Vector3 velocity = OVRPlugin.GetNodeVelocity(node, OVRPlugin.Step.Render).FromVector3f();
Vector3 angVelocity = OVRPlugin.GetNodeAngularVelocity(node, OVRPlugin.Step.Render).FromVector3f();
Quaternion orientation = pose.orientation;
```

CONTROLLERS

In order to reference, e.g., the left controller of the Quest Pro:

```
OVRInput.Controller.LTouch
```

In order to take the controller position, orientation, and velocities:

```
Vector3 position = OVRInput.GetLocalControllerPosition(controller);
Vector3 velocity = OVRInput.GetLocalControllerVelocity(controller);
Vector3 angVelocity = OVRInput.GetLocalControllerAngularVelocity(controller);
Quaternion orientation = OVRInput.GetLocalControllerRotation(controller);
```

BUTTONS

```
/// Raw button mappings that can be used to directly query the state of a controller.
public enum RawButton
{
    None                    = 0,          ///< Maps to Physical Button: [Gamepad, Touch, LTouch, RTouch, Remote: None]
    A                       = 0x00000001, ///< Maps to Physical Button: [Gamepad, Touch, RTouch: A], [LTouch, Remote: None]
    B                       = 0x00000002, ///< Maps to Physical Button: [Gamepad, Touch, RTouch: B], [LTouch, Remote: None]
    X                       = 0x00000100, ///< Maps to Physical Button: [Gamepad, Touch, LTouch: X], [RTouch, Remote: None]
    Y                       = 0x00000200, ///< Maps to Physical Button: [Gamepad, Touch, LTouch: Y], [RTouch, Remote: None]
    Start                   = 0x00100000, ///< Maps to Physical Button: [Gamepad, Touch, LTouch, Remote: Start], [RTouch: None]
    Back                    = 0x00200000, ///< Maps to Physical Button: [Gamepad, Remote: Back], [Touch, LTouch, RTouch: None]
    LShoulder               = 0x00000800, ///< Maps to Physical Button: [Gamepad: LShoulder], [Touch, LTouch, RTouch, Remote: None]
    LIndexTrigger           = 0x10000000, ///< Maps to Physical Button: [Gamepad, Touch, LTouch: LIndexTrigger], [RTouch, Remote: None]
    LHandTrigger            = 0x20000000, ///< Maps to Physical Button: [Touch, LTouch: LHandTrigger], [Gamepad, RTouch, Remote: None]
    LThumbstick             = 0x00000400, ///< Maps to Physical Button: [Gamepad, Touch, LTouch: LThumbstick], [RTouch, Remote: None]
    LThumbstickUp           = 0x00000010, ///< Maps to Physical Button: [Gamepad, Touch, LTouch: LThumbstickUp], [RTouch, Remote: None]
    LThumbstickDown         = 0x00000020, ///< Maps to Physical Button: [Gamepad, Touch, LTouch: LThumbstickDown], [RTouch, Remote: None]
    LThumbstickLeft         = 0x00000040, ///< Maps to Physical Button: [Gamepad, Touch, LTouch: LThumbstickLeft], [RTouch, Remote: None]
    LThumbstickRight        = 0x00000080, ///< Maps to Physical Button: [Gamepad, Touch, LTouch: LThumbstickRight], [RTouch, Remote: None]
    LTouchpad               = 0x40000000, ///< Maps to Physical Button: [Gamepad, Touch, LTouch, RTouch, Remote: None]
    RShoulder               = 0x00000008, ///< Maps to Physical Button: [Gamepad: RShoulder], [Touch, LTouch, RTouch, Remote: None]
    RIndexTrigger           = 0x04000000, ///< Maps to Physical Button: [Gamepad, Touch, RTouch: RIndexTrigger], [LTouch, Remote: None]
    RHandTrigger            = 0x08000000, ///< Maps to Physical Button: [Touch, RTouch: RHandTrigger], [Gamepad, LTouch, Remote: None]
    RThumbstick             = 0x00000004, ///< Maps to Physical Button: [Gamepad, Touch, RTouch: RThumbstick], [LTouch, Remote: None]
    RThumbstickUp           = 0x00001000, ///< Maps to Physical Button: [Gamepad, Touch, RTouch: RThumbstickUp], [LTouch, Remote: None]
    RThumbstickDown         = 0x00002000, ///< Maps to Physical Button: [Gamepad, Touch, RTouch: RThumbstickDown], [LTouch, Remote: None]
    RThumbstickLeft         = 0x00004000, ///< Maps to Physical Button: [Gamepad, Touch, RTouch: RThumbstickLeft], [LTouch, Remote: None]
    RThumbstickRight        = 0x00008000, ///< Maps to Physical Button: [Gamepad, Touch, RTouch: RThumbstickRight], [LTouch, Remote: None]
    RTouchpad  = unchecked((int)0x80000000),///< Maps to Physical Button: [Gamepad, Touch, LTouch, RTouch, Remote: None]
    DpadUp                  = 0x00010000, ///< Maps to Physical Button: [Gamepad, Remote: DpadUp], [Touch, LTouch, RTouch: None]
    DpadDown                = 0x00020000, ///< Maps to Physical Button: [Gamepad, Remote: DpadDown], [Touch, LTouch, RTouch: None]
    DpadLeft                = 0x00040000, ///< Maps to Physical Button: [Gamepad, Remote: DpadLeft], [Touch, LTouch, RTouch: None]
    DpadRight               = 0x00080000, ///< Maps to Physical Button: [Gamepad, Remote: DpadRight], [Touch, LTouch, RTouch: None]
    Any                     = ~None,      ///< Maps to Physical Button: [Gamepad, Touch, LTouch, RTouch, Remote: Any]
}
```

You can reference a specific button of your controller, by accessing controllers mapping

For example, if you want have a reference of the left trigger:

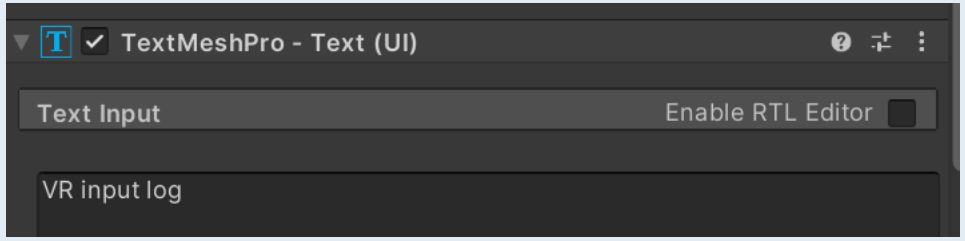
```
(OVRInput.RawButton.LIndexTrigger)
```

OVR INPUT CONSOLE

For example, if we want to get the float value of the pressure on the left controller trigger:

```
OVRInput.RawAxis1D.LIndexTrigger
```

1. In your scene, create a new UI with right mouse click in the Hierarchy > Canvas and set its render mode to World Space
2. Select the Canvas object and right mouse click > UI > TextMeshPro, when promped click Import TMP Essentials
3. Scale the Canvas and the Text and place them in front of the Player prefab
4. Select the Text object and, in TextMeshPro - Text component, change ‘New Text’ into ‘VR input log’



LOGS

1. You need two fields:
    1. A public TextMeshProUGUI which defines the area where we want to display logs
    2. A private int to define the max num of lines allowed

2. You need also a function to print the logs and a function to clear the lines where the maximum allowed is reached:

```
2 riferimenti
public static string GetTimestamp()
{
    return DateTime.UtcNow.ToString(format: "yyyy-MM-dd' 'HH:mm:ss.fff");
}

1 riferimento
private void LogInfo(string message)
{
    ClearLines();
    logsArea.text += $"{GetTimestamp()} {message}\n";
}

1 riferimento
private void ClearLines()
{
    if (logsArea.text.Split('\n').Count() >= maxLines)
    {
        logsArea.text = string.Empty;
    }
}
```

PRINT LEFT CONTROLLER LOGS

1. In the FixedUpdate() function, we need to call the CollectOVRInputControllerButtonData() function

```
CollectOVRInputControllerButtonData();
```

2. We will print the logs calling the LogInfo() function and giving it as input the triggerValue

```
private void CollectOVRInputControllerButtonData()
{
    float triggerValue = OVRInput.Get( OVRInput.RawAxis1D.LIndexTrigger );

    LogInfo(triggerValue.ToString());
}
```

N.B. we call the functions only if the target device has been found

RESULTS

- In the Hierarchy, create a new empty GO with: right mouse click > Create Empty and call it ‘Input Manager’
- Attach the ‘OVRInput_tracker’ script to the Input Manager GO by dragging it to the GO
- Assign the ‘Text (TMP)’ object (children of Canvas) to the ‘Log Area’ variable
- Press Play