Unity Bootcamp
Day 2: Unity & Virtual Reality

CAP 6121 - 3D User Interfaces for Games and Virtual Reality
CAP 6119 - Advanced Virtual Reality

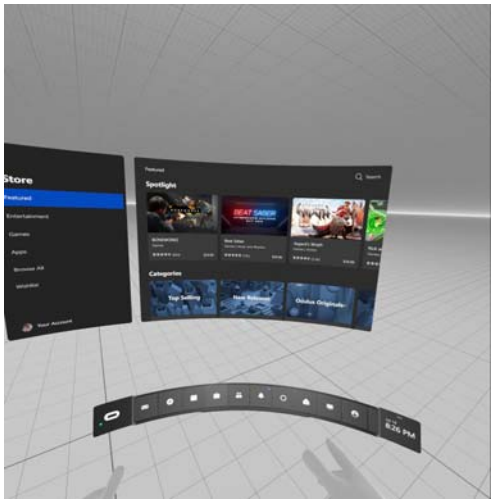
Mykola Maslych
Spring 2026
https://github.com/maslychm/UnityBootcamp-VR

Unity & VR Bootcamp Overview

- What to expect
  - Basic VR scene setup
  - Extension of the game from Day 1
    - Meta Quest 2/3/3s
    - Addition of XR interactions using XRIT
    - Custom Input Actions and controller input
  - Resource links and useful tips
- Assumptions and Requirements
  - Unity Editor basics covered
  - VR equipment available
  - Powerful PC available

Unity Development – Quest Setup

- On PC
  - Meta Horizon Link App https://www.meta.com/help/quest/1517439565442928/
  - USB-C Cable or 'AirLink' (requires good Wi-Fi bandwidth)
- On Quest
  - Quick Menu → Link → Connect to PC
- Allows streaming Unity Editor to Quest



Hardware and Software

- **We will use XR Interaction Toolkit (XRIT) - Native to Unity** https://docs.unity3d.com/Packages/com.unity.xr.interaction.toolkit@3.4
- Other options
  - SteamVR https://steamcommunity.com/app/250820
  - Meta XR SDK https://developers.meta.com/horizon/downloads/package/meta-xr-sdk-all-in-one-upm/
  - Run SteamVR apps on Quest or Windows MR HMDs using Windows Mixed Reality for SteamVR https://store.steampowered.com/app/719950

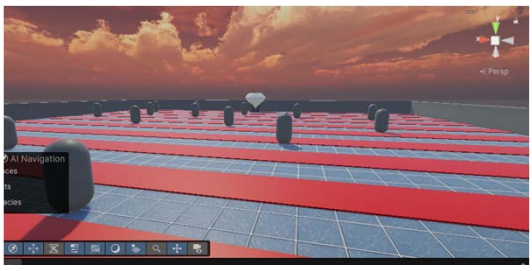
Setting up a Cross-Platform VR Project

- Package Manager
  - o **XR Plugin Management**
  - o **OpenXR Plugin**
  - o **XR Interaction Toolkit**
    - Samples → Starter Assets → Import
- Project Settings
  - o XR Plugin Management → Check OpenXR
  - o OpenXR: Enabled Interaction Profiles
    - + Oculus Touch Controller Profile (for Quest 2, 3, 3s)
    - + Meta Quest Touch Pro Controller Profile (For Quest Pro)
  - o Project Validation → Fix All
- "Enable new Input System" prompt → Yes and Restart
- Note: if creating a new project, choose "Universal 3D" (rendering pipeline)

Modifying scene to VR mode: Player Setup

- Duplicate the scene
- Move Force Field outside of main player (we will reuse it later)
- Delete the desktop object for **Player**
- Add the XR Origin prefab
- Play!

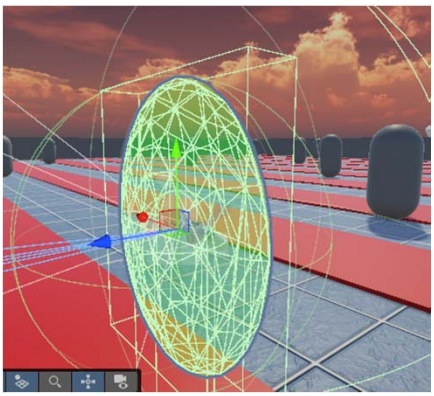
§ Adding Teleportation
  - o Add **Teleportation Area** Component to "Floor-base" object
  - o Add floor's collider to the tracked colliders
  - o Add Everything to "Include Layers" on Teleportation Area



Modifying scene to VR: Locomotion

- Enemies to target player
  - On **WaveManager**, set target to **Main Camera**
  - On **Enemy** prefab, set **AimHeight** = **0**
- Level Completion
  - Create an invisible object with collider and **RigidBody**, attached to camera and set its tag to **Player**
  - Raise the collider on **FinishLine**
  - Jumping issue? Window → Analysis → Physics Debugger
    - § Assign "HeadCollider" to a new layer IgnoreDefault
    - § Assign "FinishLine" to a new layer "FinishLine"
    - § Project Settings → Physics Settings → These two need to only collide with each other

Modifying scene to VR: Carry Shield



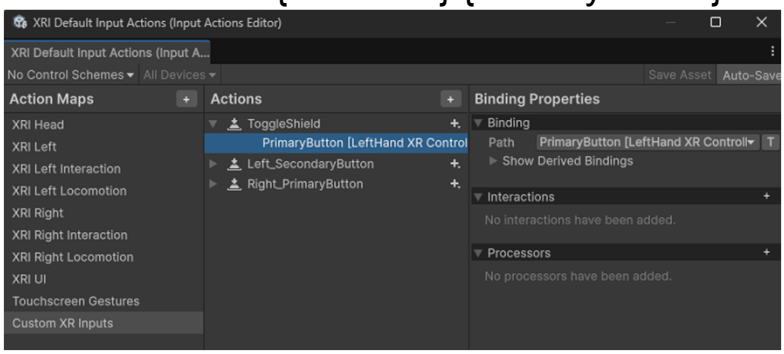
- Shield
  - Resize to (.5, .6, .1)
  - Add Convex Mesh Collider and an extra Box Collider (isTrigger = true)
  - Disable Sphere Collider
  - Attach the object to Left Controller
  - Toggle the shield on/off via button press?
    Input System time!

Unity Input System & XR Interaction Toolkit

- Input Action References
  https://docs.unity3d.com/Packages/com.unity.inputsystem@1.18/api/UnityEngine.InputSystem.InputActionReference.html
- Allow mapping physical hardware input to detectable events
  https://docs.unity3d.com/6000.3/Documentation/Manual/xr_input.html
- Cross-platform
  - Mouse, keyboard
  - XR Controllers, Game Controllers
  - Tracked hands
  - Etc.

Create Input Actions

- Open the "XRI Default Input Actions" object
- Create a new Action "ToggleShield" and assign path: <XRController>{LeftHand}/{PrimaryButton}



Reading Input Actions: Functions

```
[SerializeField] private InputActionReference actionReference;

Unity Message | 0 references
private void Update()
{
    if (actionReference == null || actionReference.action == null) return;

    // True as long as the button is held down
    actionReference.action.IsPressed();

    // True only on the frame button state transitions from "not pressed" to "pressed"
    actionReference.action.WasPressedThisFrame();

    // True only on the frame button state transitions from "pressed" to "not pressed"
    actionReference.action.WasReleasedThisFrame();

    // True only on the frame the action is performed (for button, identical to WasPressedThisFrame())
    actionReference.action.WasPerformedThisFrame();
}
```

Reading Input Actions: Usage Example

```
public class CustomXRInputs : MonoBehaviour
{
    [SerializeField] private InputActionReference actionReference;

    Unity Message | 0 references
    private void Update()
    {
        if (actionReference == null || actionReference.action == null) return;

        if (actionReference.action.WasPressedThisFrame())
        {
            Debug.Log("Action button pressed!");
        }
    }
```

Reading Input Actions: Callback Method

```
public class CustomXRInputs : MonoBehaviour
{
    [SerializeField] private InputActionReference customActionReference;

    Unity Message | 0 references
    private void OnEnable()
    {
        customActionReference.action.Enable();
        customActionReference.action.performed += OnCustomActionPerformed;
    }

    Unity Message | 0 references
    private void OnDisable()
    {
        customActionReference.action.performed -= OnCustomActionPerformed;
        customActionReference.action.Disable();
    }

    2 references
    private void OnCustomActionPerformed(InputAction.CallbackContext context)
    {
        if (context.performed)
        {
            Debug.Log("Custom action performed!");
        }
    }
```

Reading Input Actions: Shield Toggle Implementation

```
    1 reference
    private void ProcessControllerInput()
    {
        if (toggleShield == null || toggleShield.action == null) return;

        if (toggleShield.action.WasPressedThisFrame())
        {
            if (shieldObject != null)
            {
                shieldObject.SetActive(!shieldObject.activeSelf);
            }
        }
    }
```

XR Interactors and Interactables

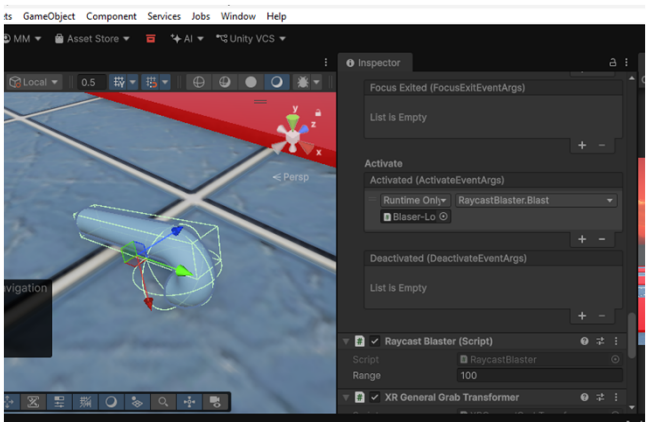
- Common XR Interactors
  - Near-Far Interactor
  - Ray Interactor
  - Poke Interactor
  - Teleport Interactor
- Common Interactables
  - XR Grab Interactable
  - XR Select Interactable

```
public class MyHoverHandler : MonoBehaviour
{
    private XRGrabInteractable grabInteractable;

    private void Awake() {
        grabInteractable = GetComponent<XRGrabInteractable>();
        grabInteractable.hoverEntered.AddListener(OnHoverEntered);
        grabInteractable.hoverExited.AddListener(OnHoverExited);
    }

    private void OnDestroy() {
        grabInteractable.hoverEntered.RemoveListener(OnHoverEntered);
        grabInteractable.hoverExited.RemoveListener(OnHoverExited);
    }

    private void OnHoverEntered(HoverEnterEventArgs args) { /*Hover logic*/ }
    private void OnHoverExited(HoverExitEventArgs args) { /*Hover exit logic*/ }
}
```

- Produce interaction events: **Hover**, **Select**, **Activate**... Methods can subscribe to these events

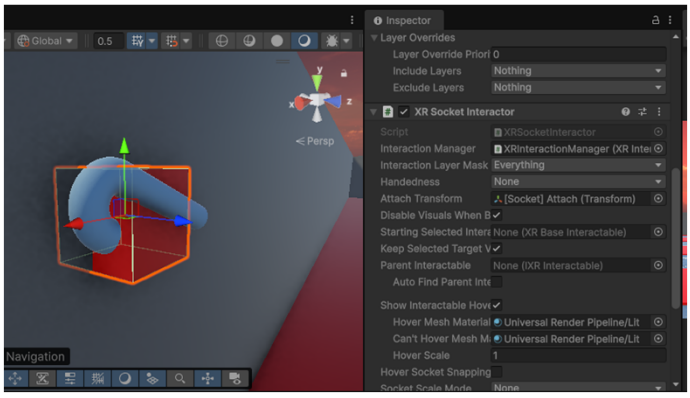
Grabbable objects: Grabbing a Blaser

- Blaser-Long from the XRIT Demo scene
- Attach a new script with a public **Blast()** function
- Call the function on XR Grab Interactable's **Activate** Event (trigger button while holding the object)
- Enemies (tag check) take damage on **Raycast** hits



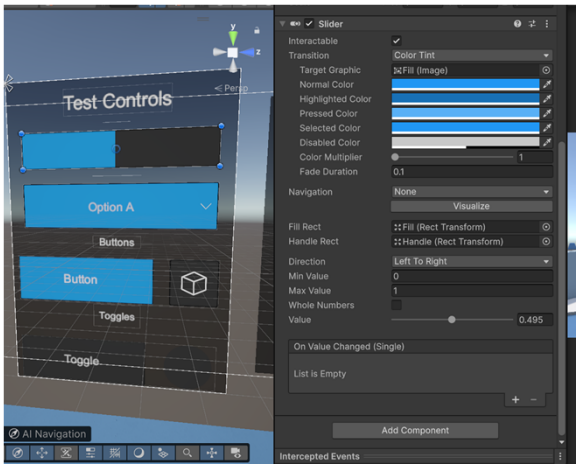
XR Socket Interactor

- **XR Socket Interactor** allows attaching/detaching **XR Grab Interactable** objects to it
- Use-cases:
  - Inventory systems
  - Tool belt
  - Activation



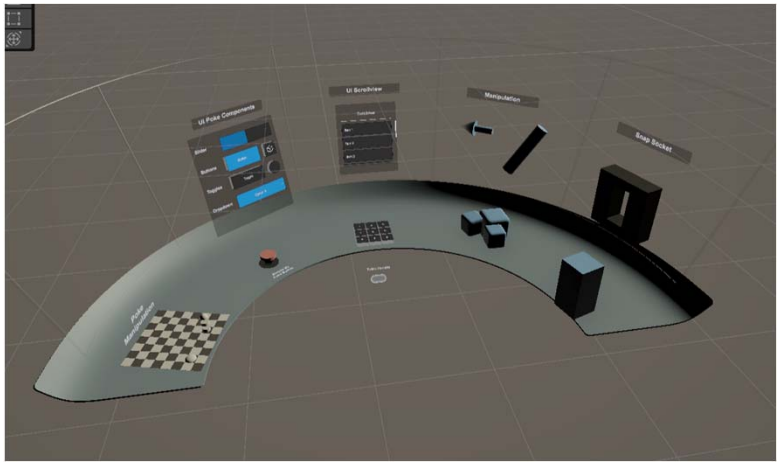
UI Interactions

- Work well with Ray Interactor, Poke Interactor
- Trigger subscribable events
  - **Buttons:** OnClick()
  - **Sliders:** OnValueChanged(int value)
  - **Toggles:** OnValueChanged(bool)



Hand Tracking with XRIT

- Install **XR Hands plugin** and **"Hand Visualizer"** sample
  - https://docs.unity3d.com/Manual/com.unity.xr.hands.html
  - https://docs.unity3d.com/Packages/com.unity.xr.interaction.toolkit@3.4/manual/samples-hands-interaction-demo.html
- Can also be installed together from **XR Interaction Toolkit** Package under Samples → Hands Interactions Demo
- Requires Hand and Body tracking permissions on Quest



More Unity: Shortcuts and QoL

- Use Version control (Git, VSC, Unity Devops)
- Singleton & Command patterns
  https://unity.com/resources/design-patterns-solid-ebook
- Hierarchy search for objects with a specific Component type
- Asset filtering by file type
- Print statements in color
  `Debug.Log($"<color=red>Height {transform.position.y}</color>");`
- Refer to Demo Scenes of packages

More Unity: Optimizations

- Enable GPU instancing for repeating materials
- Keep poly count low
  - o <2 million for powerful PC
  - o <1 million for less-powerful PC
- Object Pooling (reduce instantiation)
- Event Function Execution Order
  https://docs.unity3d.com/6000.3/Documentation/Manual/execution-order.html

More Unity: Debugging

- **Window → Analysis → Physics Debugger** to visualize all active colliders
- **Window → Analysis → Profiler** to for per-frame function call durations
- Scene Editor Draw Modes
  - o **Wireframe**: quick way to find high-poly areas
- **Preferences → General → Recompile After Finished Playing**!!!

Data capture: HMD, Hands/Controllers, Buttons

- Track the raw Vector3 positions and Quaternion (not euler angles!!) rotations of these objects – useful for data logging
- Use the Input Action References to record button clicks
- Log every frame into CSV for analysis

More VR Development SDKs tradeoffs

- **XRIT:** stable and cross-platform, but less 'new' features
- **Meta XR SDK:** less stable, only compatible with Quest, more 'new' features
  - Physical environment mesh API
  - Lipsync
  - Inverse Kinematics (for self-embodiment)
- **SteamVR:** reliable but limited to PC-only platforms
- **VRTK:** cross-platform, but outdated and bloated