**Lab 5: UI and the Canvas**

# Technical Requirements

To implement the projects and exercises in this lab, you will need the following:

* A PC or Mac capable of running Unity 2021.3.18 LTS or later, along with an internet connection to download files.
* A VR headset supported by the Unity XR platform.
* Lab 5 Files are available on LZ under Learning Materials/LabFils

# Making a Reusable Default Canvas

Start this lab the downloaded Lab5 files from LZ

Save a working copy of the scene for this chapter, named 06-UI, as follows.

1. Open the 04-Navmesh scene.
2. Go to **File** | **Save As** and name it 06-UI.
3. You may need to rebake the NavMesh by selecting **Window** | **Navigation** | **Bake**.

We now have a working scene named 06-UI for this lab.

# Creating a default canvas prefab

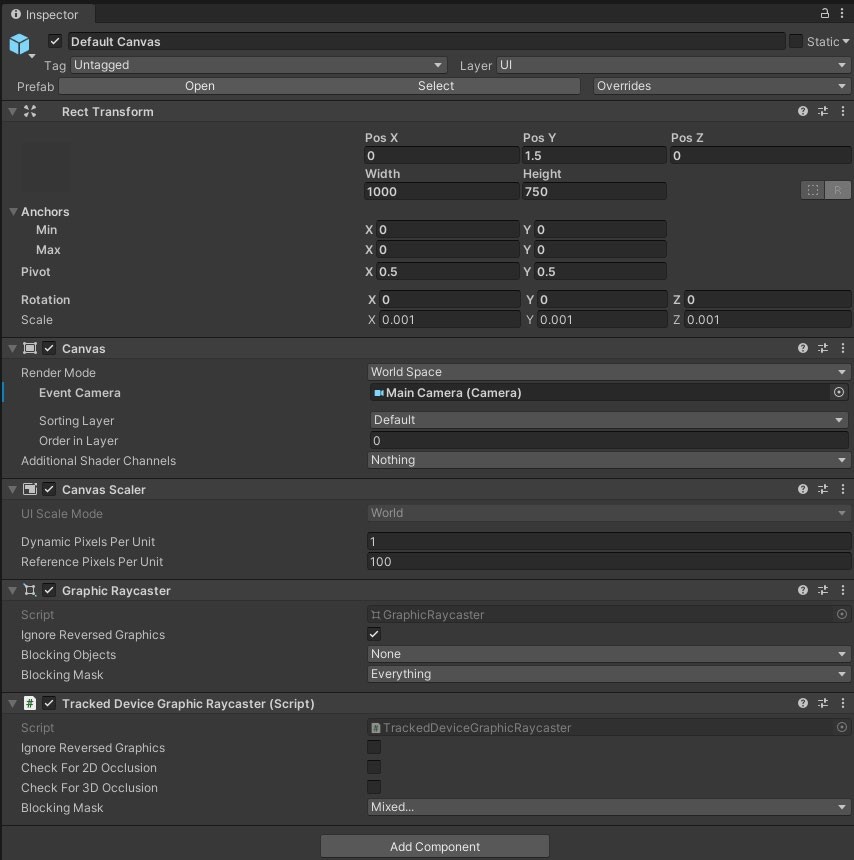
Create a new world space XR UI canvas, as follows:

1. Add a new canvas to the scene (**GameObject** | **XR** | **UI Canvas**).
2. Rename the canvas Default Canvas.
3. Notice that **Render Mode** is set to **World Space** (if not, change it now).
4. Also, note the canvas **Event Camera** setting will be preset to **Main Camera** in **XR Rig**.

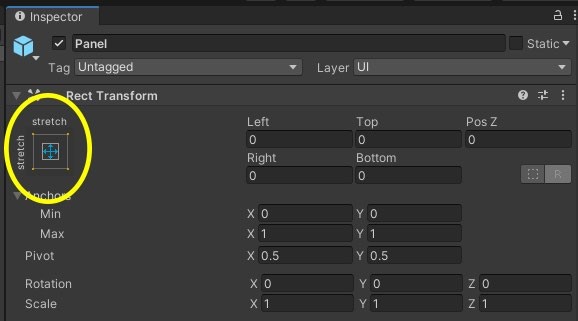
Set it to a value of 1000 x 750, giving it a 0.75 aspect ratio, as follows:

1. In **Rect Transform**, set **Width** to 1000 and **Height** to 750.
2. In **Scale**, set **X**, **Y**, and **Z** to 0.001, 0.001, and 0.001, respectively. This is the size for one of our canvas units in world space units.
3. Now, position the canvas centered above the ground plane. In **Rect Transform**, set **Pos X**, **Pos Y**, and **Pos Z** to 0, 1.5, and 0, respectively.

The **Default Canvas** scope's **Inspector** window is shown in the following screenshot. Note the **Width**, **Height**, and **Scale** values, as well as the additional default components—**Canvas**, **Canvas Scaler**, **Graphic Raycaster**, and **Tracked Device Graphic Raycaster**:

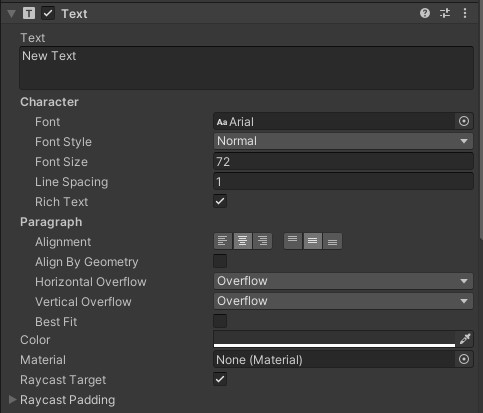


1. With **Default Canvas** selected, right-click and choose **UI** | **Panel** (ensure that it's created as a child of **Default Canvas**; if not, move it under **Default Canvas**).
2. With **Panel** selected, note that at the upper left of its **Rect Transform** pane, there is an **Anchor Presets** button (shown in the following screenshot) that is set by default to **stretchstretch** so that it fills the canvas. Selecting this opens the **Anchor Presets** dialog box.
3. Press and hold the *Alt* key to see and set the **Anchors** position and/or the *Shift* key for the **Pivot** setting:



Add a **Text** element under the panel with useful new settings, as follows:

1. With **Panel** selected, right-click and choose **UI** | **Text** (ensure that it's created as a child of **Panel**; if not, move it under **Panel**). New Text should appear on the canvas, although the text may be too small to easily notice.
2. Set its **Anchors** setting to **stretch**-**stretch** using the **Anchor Presets** widget at the upper left of its **Rect Transform** pane.
3. Also, set its **Position** setting to **stretch**-**stretch** using the *Alt* key while the **Anchor Presets** box is still open.
4. Choose a larger font. With **Text** selected, set **Font Size** to 72.
5. Center the text on the panel. With **Text** selected, set **Alignment** to **Center Align** and **Middle Align**.



# Initializing the Default Main Camera

1. Open the **Default Canvas** prefab for editing (double-click on **Default Canvas** in the **Project** window's Prefabs/ folder.)
2. Select the root **Default Canvas** object.
3. Create a new C# script on the object, named AssignCanvasEventCamera.

Assign Camera.main to the Canvas if it isn't already assigned:

public class AssignCanvasEventCamera : MonoBehaviour

{ void Start()

{

Canvas canvas = GetComponent<Canvas>(); if (canvas && canvas.worldCamera == null)

{ canvas.worldCamera = Camera.main;

}

}

}

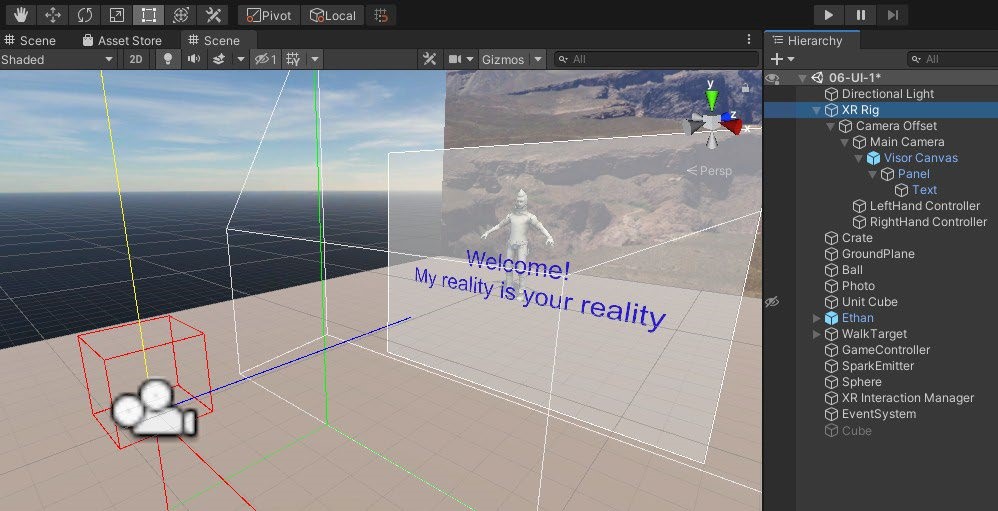
# Creating a Visor HUD

Make a visor HUD with a welcome message and see for ourselves how it feels, as follows:

1. In the **Hierarchy** window, unfold the **XR Rig** object and then drill down to the **Main Camera** object.
2. From the **Project** window, drag the **Default Canvas** prefab onto the camera object so that it becomes a child of it, and rename it Visor Canvas.
3. In the **Inspector** window for the canvas, change the **Rect Transform** component's **Pos X**, **Pos Y**, and **Pos Z** values to 0, 0, and 1, respectively.
4. Unfold **Visor Canvas** and select the child **Text** object.
5. In the **Inspector** window, change the text to Welcome! My reality is your reality. (You can enter line breaks in the input text area.)
6. Change its **Horizontal Overflow** setting to **Wrap**
7. Change the text **Color** setting to something bright and colorful.
8. Select the parent **Panel** object and disable its **Image** object so that only the text shows by un-checking its **enable** checkbox in **Inspector**.

9.

Save the scene and try it in VR.



# The Windshield HUD

Create a simple windshield HUD by performing the following steps:

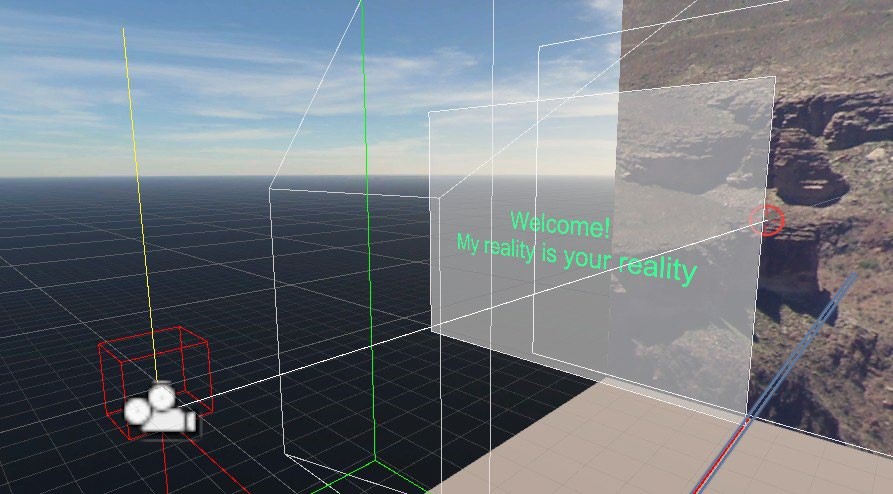
1. From the **Project** window, drag the **Default Canvas** prefab onto the XR

Rig object in the **Hierarchy** panel so that it becomes an immediate child of XR Rig (not under the camera this time).

1. Rename it Windshield Canvas.
2. With Windshield Canvas selected, set the **Rect Transform** component's **Pos X**, **Pos Y**, and **Pos Z** values to 0, 1.5, and 1, respectively.
3. Now, we'll set the **Text** component. With **Text** under Windshield Canvas selected, change the text to Welcome! My reality is your reality.

The HUD canvas will stay in front of you, relative to the position of your body object, XR Rig. You can try it now in the editor:

1. Select **XR Rig** in **Hierarchy**.
2. Press **Play**.
3. In the **Scene** window, using the **Move** gizmo, move the **XRRig** object's position. In VR, you'll see that the HUD follows along like it's part of your body or a spaceship's cockpit, as in the following **Scene** view:



**Hiding the Panel Using Canvas Group**

Write a script that removes the welcome message after 5 seconds.

1. With **Windshield Canvas** selected, add a **Canvas Group** component (go to the main menu, then **Component | Layout | Canvas Group**).
2. Note that **Canvas Group** has an **Alpha** parameter. Try changing its value to somewhere between 1 and 0 to fade the canvas.
3. Add a new script, HideAfterDelay, to the canvas (go to **Add Component | New Script**, name it HideAfterDelay, and then click **Create And Add**).
4. Open the script for editing.

Here's the HideAfterDelay.cs script:

public class HideAfterDelay : MonoBehaviour

{ public float delayInSeconds = 5f; public float fadeRate = 0.25f; private CanvasGroup canvasGroup; private float startTimer;

private float fadeoutTimer;

void OnEnable()

{ canvasGroup = GetComponent<CanvasGroup>(); canvasGroup.alpha = 1f;

startTimer = Time.time + delayInSeconds; fadeoutTimer = fadeRate;

}

void Update()

{

// time to fade out? if (Time.time >= startTimer)

{ fadeoutTimer -= Time.deltaTime;

// fade out complete? if (fadeoutTimer <= 0)

{ gameObject.SetActive(false);

} else

{

// reduce the alpha value

canvasGroup.alpha = fadeoutTimer / fadeRate; }

}

}

}

# Making a Scoreboard

Add a scoreboard into the scene at the top-left corner of the backdrop **Photo** image:

1. From the **Project** window, drag the Default Canvas prefab directly into the **Scene** view and rename it Scoreboard.
2. Position and size the canvas in the corner of the image (with Scoreboard selected, set the **Rect Transform** component's **Pos X**, **Pos Y**, and **Pos Z** values to(-2.8, 7, and 4.9, respectively, and the **Width** and **Height** values to 2000 and 750, respectively).
3. Hide the **Panel** background (select the child **Panel** object and in the **Inspector** window, un-check the **Image** component **enable** checkbox).
4. Enlargen the text (select the **Text** object and in the **Inspector** window, set **Font Size** to 300).
5. Choose a noticeable color, such as red, for the **Text** object.
6. Enter the Score: 0 sample string for **Text**.



Update the KillTarget.cs script, as follows:

**using UnityEngine.UI;**

* 1. Add a public variable for scoreText at the top of the class:

**public Text scoreText;**

* 1. Add a line to Start() to initialize the score text:

**scoreText.text = "Score: 0";**

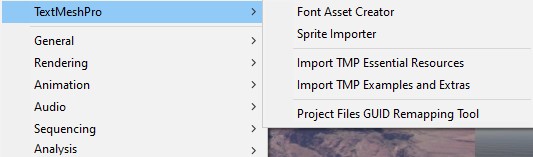
* 1. Then, add a line to Update() to change the score text when the score changes:

score += 1; **scoreText.text = "Score: " + score;**

* 1. Select GameController from the **Hierarchy** panel, and then drag and drop the **Text** object under **Scoreboard** from **Hierarchy** onto the **Score Text** field in the **Kill Target** component.
  2. Run the scene in VR. Each time you kill Ethan (by staring at him), your score will be updated on the Scoreboard component at the upper left of the Photo plane.

# Using TextMesh Pro

1. Open **Package Manager** (**Window | Package Manager**).
2. In the search bar, search for TextMeshPro. If necessary, in the left-side filter dropdown, choose **All** instead of **Packages In Project**.
3. Click **Install** to install the package, or if necessary, select **Update** to install the current version.
4. Import the **TMP essential assets** into your project (**Window| TextMeshPro | Import TMP Essential Resources**) and then select **Import** in the import box.
5. This should open a new **TMP Importer** window. Press the **Import TMP Essentials** button to complete the import.
6. Likewise, import the **TMP examples and extras** by selecting **Import TMP Examples and Extras**, and carry out the previous step:



Complete the following:

1. With **Scoreboard Panel** selected in **Hierarchy**, create a new TMP text element (rightclick, then go to **UI | Text - TextMeshPro)**. Notice its default name is Text (TMP).
2. This replaces our standard UI text element, so disable the **Text** object (un-check the **enable** checkbox at the top left of **Inspector**).
3. Let's set up the **Text (TMP)** object like we had the original text one. Go to **Rect Transform** | **Anchor Presets** | **Stretch-Stretch**, and then press *Shift* + *Alt* and click **Stretch**-**Stretch**.
4. For the **Text** string, set the **Score** value to 0, **Font Size** to 300,

**Wrapping** to **Disabled**, **Overflow** to **Overflow**, and **Alignment** to **Left, Middle**.

Experiment with some fonts. Let's try **Bangers**:

* 1. In the **Font Asset** area, pick the doughnut icon to open the **Select TMP Font Asset** window and choose **Bangers SDF**.
  2. For its **Material Preset** setting, use **Bangers SDF Glow**.
  3. Scroll to the **Glow** settings in the **Material** pane to adjust the colors and other

settings as you desire.

We now need to tell the KillTarget script that we're using a TMP text object rather than the standard UI text object. Modify KillTarget.cs as follows.

1. Declare that we are using the Unity Engine TMP classes:

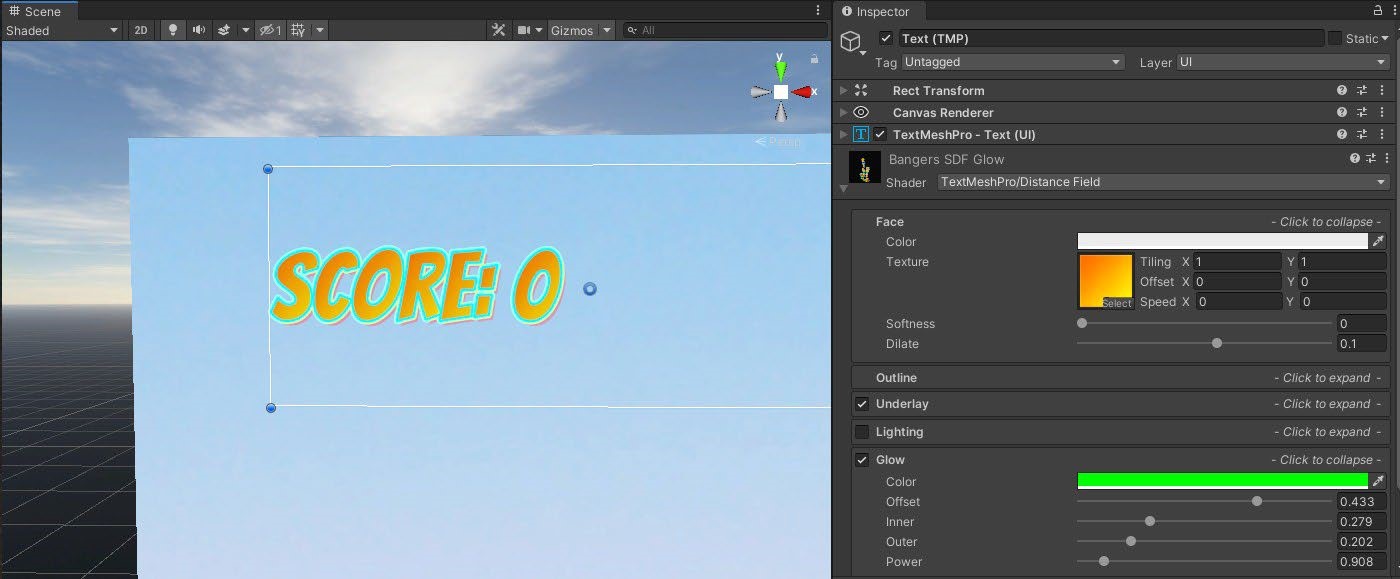
using TMPro;

1. Replace the data type of the scoreText variable with TMP\_Text:

public

TMP\_Text

scoreText;



# Info Bubbles

Display the *X* and *Z* location of the WalkTarget object, controlled by the LookMoveTo.cs script. To add the info bubble, take the following steps:

1. From the **Project** window, drag the Default Canvas prefab directly into the **Hierarchy** window so that it's a child of WalkTarget.
2. Rename it InfoBubble.
3. Use **Rect Tool** in the **Scene** window to adjust the **InfoBubble** canvas size. Set **Pos X**, **Pos Y**, and **Pos Z** to 0, 0.2, and 0, respectively, and **Width** and **Height** to 600 and 150, respectively.
4. On the child **Text** element, enter the X:00.00, Z:00.00 string for **Text**.

Verify that the canvas and text look good.



Modify the LookMoveTo.cs script to show the current WalkTarget object's *X* and *Z* position. Open the script in the editor and add the following code (highlighted in bold). Declare the infoBubble and infoText variables, then initialize infoText in Start:

using UnityEngine; **using UnityEngine.UI;**

public class LookMoveTo : MonoBehaviour

{ public GameObject ground; private Transform camera;

**public Transform infoBubble; private Text infoText;**

void Start()

{ camera = Camera.main.transform;

**if (infoBubble != null)**

**{ infoText = GetComponentInChildren<Text>(); }**

Add Update to the code, which updates the infoText object with the current position of the target. Rotate the canvas to always face the camera, as follows:

void Update()

{

Ray ray;

RaycastHit[] hits; GameObject hitObject;

ray = new Ray(camera.position, camera.rotation \* Vector3.forward); hits = Physics.RaycastAll(ray); for (int i = 0; i < hits.Length; i++)

{

RaycastHit hit = hits[i]; hitObject = hit.collider.gameObject; if (hitObject == ground)

{ **if (infoBubble != null)**

**{ infoText.text = "X:" + hit.point.x.ToString("F2") +**

", " +

**"Z:" + hit.point.z.ToString("F2");**

**infoBubble.LookAt(camera.position); infoBubble.Rotate(0, 180f, 0);**

**}** transform.position = hit.point;

}

}

# Adding a Canvas Reticle to Gaze-based Interaction

Use the sprite image named GUIReticle.png with the files in Lab 5 in Blackboard

Add the reticle by taking the following steps:

1. Find your **Main Camera** object in the **Hierarchy** window under **XR Rig**, as we did previously.
2. From the **Project** window, drag the Default Canvas prefab onto the camera object so that it becomes a child of the camera. Name it Reticle Canvas.
3. Set the **Rect Transform** component's **Width** and **Height** values to

128 and 128 respectively, and position **Pos X**, **Pos Y**, and **Pos Z** to 0, 0, and 1, respectively.

1. Disable the child **Text** object by un-checking its **enable** checkbox in **Inspector.**
2. Set the panel's **Source Image** setting to your reticle image (drag the GUIReticle image from the Project folder onto the panel's **Source Image** slot, or use its doughnut icon to browse).
3. To change its color, use the **Image** | **Color** property. I picked green. Set it to opaque (set the alpha value to 255).
4. Save the scene and try it in VR.

.Write a script to handle reticle distance:

1. With Reticle Canvas selected, click on **Add Component** | **New Script** and name it CursorPositioner, and click on **Create And Add**.
2. Open the script by double-clicking on the name.

Here's the CursorPositioner.cs script:

public class CursorPositioner : MonoBehaviour

{ private float defaultPosZ; private Transform camera;

void Start()

{ camera = Camera.main.transform; defaultPosZ = transform.localPosition.z; }

void Update()

{

Ray ray = new Ray(camera.position, camera.rotation \* Vector3.forward); RaycastHit hit; if (Physics.Raycast(ray, out hit))

{ if (hit.distance > defaultPosZ)

{ transform.localPosition = new Vector3(0, 0, hit.distance \*

0.95f); } else

{ transform.localPosition = new Vector3(0, 0, defaultPosZ); }

}

}

}

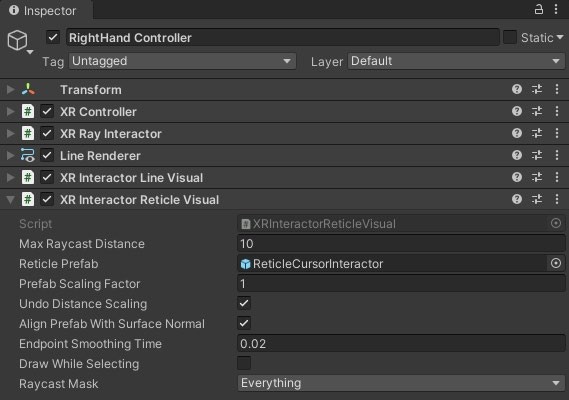
# Adding a Reticle to the XR interactor Hand Controller

Make a prefab of our reticle graphic, contained in an empty parent game object.

1. In the **Hierarchy** root, select **GameObject | Create Empty** and name it Reticle Prefab, then reset its **Transform** value (click on the three-dots icon and click **Reset**).
2. Drag the **Default Canvas** prefab from its **Project** folder as a child of **Reticle Prefab** and rename it Reticle.
3. Set the **Reticle** position's **X**, **Y**, and **Z** values to 0, 0.01, and 0, respectively, **Width** and **Height** to 64 and 64, respectively, and **Rotation** to 90, 0, 0 so that it's upright.
4. Set its child, by going to **Panel | Source Image**, then to **GUIReticle**, setting **Color** to green, and setting no alpha transparency (255).
5. Disable its child **Text** object (un-check its **enable** checkbox).
6. Drag **Reticle Prefab** into your **Project** window's Prefabs/ folder to save it as a prefab.
7. Delete the working copy of **Reticle Prefab** from the **Hierarchy** panel.

Add the reticle to the hand controller using the following steps:

1. In the **Hierarchy** window, unfold the **XR Rig** object and select the **RightHand Controller** object.
2. In the main menu, select **Components** | **XR** | **Helpers** | **XR Interactor Reticle Visual**.
3. Drag **Reticle Prefab** from the **Project** window onto the **Reticle Prefab** slot of the **XR Interactor Reticle Visual** component.
4. Repeat these steps for the **LeftHand Controller** object if you want.



When you press **Play** and use the hand controller to point at objects in the scene, there's now a reticle at the end of the laser ray.

# Adding a gaze-based reticle using XRI

Create a new C# script named BasicInteractor

The following code simply implements an XRBaseControllerInteractor class with no support for interactable targets:

using System.Collections.Generic;

using UnityEngine.XR.Interaction.Toolkit;

public class BasicInteractor : XRBaseControllerInteractor

{ public override void GetValidTargets(List<XRBaseInteractable> validTargets) { }

protected override List<XRBaseInteractable> ValidTargets { get; } }

In Unity, we can use this new script, as follows:

1. If you haven't already, disable or delete the **Reticle Canvas** child of **Main Camera** since we're going to use a different implementation now.
2. With **Main Camera** selected, from the main menu, select **Component** | **XR** | **XR Interactor Reticle Visual**.
3. Notice how this forces an **XR Controller** component to also be added. Let's disable its interactive features. (If not, please add the component now using **Component | XR | XR Controller**.) Un-check the **Enable Input Actions** checkbox.
4. Set its **Controller Node** setting to **Center Eye**. Ignore the other settings.
5. With **Main Camera** still selected, drag the BasicInteractor script onto it.
6. Drag **Reticle Prefab** from the **Project** panel's Prefabs/ folder onto the reticle component's **Reticle Prefab** slot.

When you press **Play**, you should see a full-featured reticle cursor following your gaze, maintaining its scale, and aligning with the surface normals.

# Adding a Dynamic Water Hose

1. In the **Project** window, find the Assets/Standard Assets/Particle Systems/Prefabs/Hose prefab and drag it into the **Hierarchy** window.
2. Set its **Transform** component's **X**, **Y**, and **Z** values to -3, 0, and 1.5, respectively, and **Rotation** to -20, 90, and 0.
3. Ensure that **Hose** is enabled (check its **Enable** checkbox).
4. The hose comes with two control scripts—Hose and Simple Mouse Rotator—that we will not use, so remove both components (click on the three- dots icon and select **Remove Component**).
5. Unfold the **Hose** object in **Hierarchy** so that you can see its child **WaterShower** particle system. Select it.
6. In the **Scene** window, you'll now notice a **Particle Effect** control. If necessary, press **Restart** to see the hose spray in action.
7. In the **Inspector** window, in the **Particle System** properties for **WaterShower**, ensure the **Play On Awake** checkbox is checked.

# Creating a Dashboard with a Toggle Button

The dashboard will be a canvas positioned as a dashboard about 1 meter off the ground and a little out in front of the player, as follows:

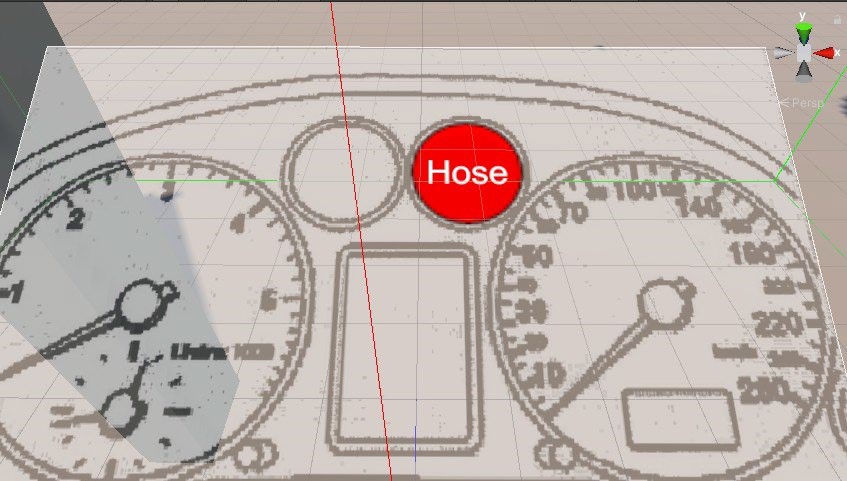
1. From the **Project** window, drag the Default Canvas prefab onto the XR Rig object in the **Hierarchy** window so that it becomes a child.
2. Rename it Dashboard.
3. With Dashboard selected, set the **Rect Transform** setting to a comfortable size, then position and angle it. For example, set **Width** and **Height** to 800 and 600, respectively, **Pos X**, **Pos Y**, and **Pos Z** to 0, 0.8, and 0.15, respectively,and **Rotation** to 60, 0, and 0. Feel free to adjust the position for your preferred comfort zone and specific VR device camera rig.
4. Disable the **Text** child object of Dashboard/Panel/.

Use the DashboardSketch.png in the Lab 5 download

1. If necessary, import the DashboardSketch.png file into your **Project** panel (such as the Assets/Textures folder). Set it as a sprite (set **Import Settings | Texture Type** to **Sprint (2D and UI)** and press **Apply**).
2. Select the **Panel** child of **Dashboard** and drag **DashboardSketch** onto its **Source Image** slot (replacing the default white **Background** image).
3. Adjust its **Color** alpha value if you want to make it opaque.

Create a **Toggle** element and modify it to look like a button instead of its default checkbox graphics:

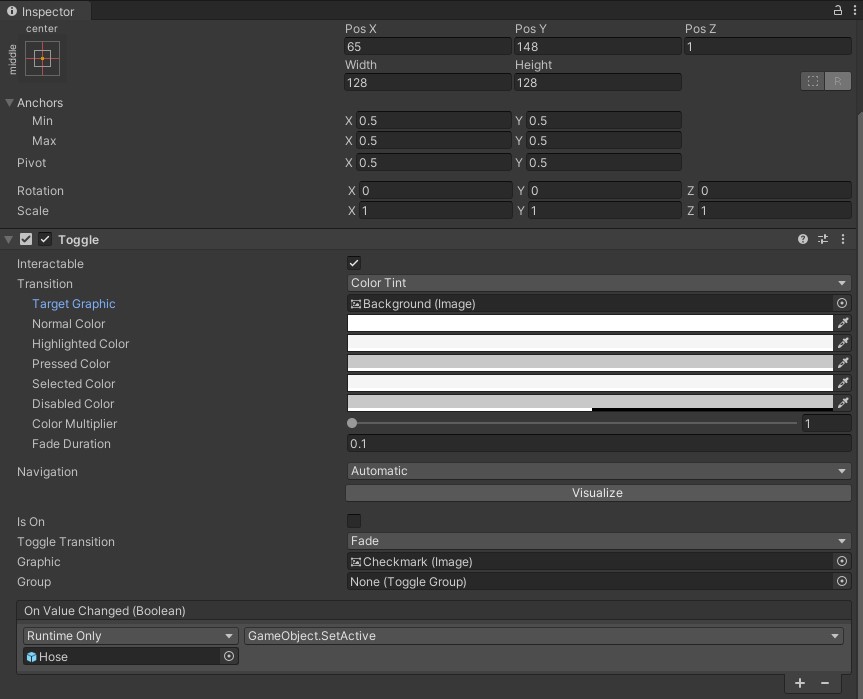
1. With the **Panel** object selected, choose **GameObject** | **UI** | **Toggle**, move it to a child of **Panel**, and rename it WaterHose Toggle.
2. Set its **Width** and **Height** values to 128 and 128, respectively, and **PosX**, **PosY**, and **PosZ** to 65, 148, and 1, respectively.
3. On its child **Background** object, stretch its anchor to **stretch**-**stretch** (click on the anchor icon, then **stretch**-**stretch**, and then press *Alt* + *Shift* and click **stretch**- **stretch**).
4. Change the **Background Source Image** setting to **Knob** (use the doughnut icon to select a new sprite) and change its **Color** property to a red (#FF0000) for its off state.
5. Instead of the checkmark image for the on state, we'll overlay a green circle. Select the child **Checkmark** object, set its anchor to **stretch**-**stretch**, then press *Alt* + *Shift* and select **stretch**-**stretch**.
6. Change the **Source Image** setting to **Knob** and set its **Color** property to green (#00FF00).
7. Change **Label** to Hose (select the **Label** object, set **Text** | **Text** to Hose, set **Text** | **Alignment** | **Middle** and **Center**, and **Color** to white, #FFFFFF).
8. We'll start the scene with the toggle disabled (with **WaterHose Toggle** selected, uncheck the **Is On** checkbox).
9. Likewise, disable **Hose** (select **Hose** in **Hierarchy** and uncheck its **enable** checkbox).



Connect the button to the hose, as follows:

* 1. Ensure **WaterHose Toggle** is selected in **Hierarchy**.
  2. In its **Toggle** component's **On Value Changed** events, click the **+** button to add an event response.
  3. Drag the **Hose** object from **Hierarchy** onto the **Runtime object** slot.
  4. From the **Function** dropdown, select **GameObject | SetActive**.

The resulting **WaterHose Toggle** settings look as in the following screenshot in the **Inspector** window:



Press **Play** and try it in VR. You can point the hand controller to the toggle button and pull the trigger to interact. Press once to turn on the hose and again to turn it off.

# Stopping the ray interactor at the Canvas

Make these adjustments:

1. Select the **Panel** child of the **Dashboard** canvas.
2. Add a child **Cube** object (right-click and select **3D Object Cube**) and name it Backstop.
3. The default **Cube** object already has a **Box Collider** component, which we want to be the size of the canvas. Set the cube's **Rect Transform Scale** property to match its parent canvas (800, 600, 1).
4. Position it so that it is a little behind the panel and set **Position** to 0, 0, 5.
5. Disable its **Renderer** component (un-check the **enable** checkbox of its **Mesh Renderer** component.

When you press **Play** and use the hand controller with its ray interactor, the ray line extends from your hand to the dashboard, but not beyond the Backstop object we just added.

# Direct interaction with UI Elements

Up to now in this chapter, we've been using the XR ray interactor on hand controllers to

1. In **Hierarchy**, select **RightHand Controller** (under **XR Rig/Camera Offset/**), duplicate it (press *Ctrl* + *D*), and rename the copy RightHand Direct.
2. Disable the original **RightHand Controller** object (uncheck its **enable** checkbox).
3. On **RightHand Direct**, remove all of its components other than **XR Controller** (right-click on the components and select **Remove Component**).
4. Add a **Direct Interactor** component (go to the main menu and select **Component**

| **XR** | **XR Direct Interactor**).

For the interactor to detect an object, it needs a collider.:

1. With **RightHand Direct** selected, add a sphere collider (**Component** | **Physics** | **Sphere Collider**).
2. Set the collider's **Radius** property to 0.1.
3. Check its **Is Trigger** checkbox.

Make a sphere to visualize where your hand is.

1. With **RightHand Direct** selected, add a sphere (right-click and select **3D Object** | **Sphere**).
2. Set its **Scale** value to 0.1, 0.1, 0.1.

Make the UI button an interactable object by adding a corresponding interactable and collider components to the button. We'll use a **Simple Interactable** component to do this:

1. With the **WaterHose Toggle** setting in the **Dashboard** panel selected, add a

**Simple Interactable** component (**Component** | **XR** | **XR Simple Interactable**).

1. This also automatically adds a **Rigidbody** component, which is required. Since we don't really want to apply any physics to the button, we should constrain it. Uncheck the **Use Gravity** checkbox.
2. Then, check all six of the **Constraints** checkboxes—the **Freeze Position** property's **X**, **Y**, and **Z** settings and the **Freeze Rotation** property's **X**, **Y**, and **Z** settings.
3. Add a collider so that the hand controller can find it. We'll use a **Box**

**Collider** object, which can be more easily flattened than a sphere. (**Component** |**Physics** | **Box Collider**) and set its **Size** property to 100, 100, 5.

Connect the **Interactable On Hover Enter** event, which is triggered when the hand touches the button, to equate to a toggle button selection. We need a little helper script to translate the interactable events to toggle events and ensure the toggle's OnValueChanged event is invoked:

1. On the **WaterHose Toggle** object, create a new C# script named ForceToggle.
2. Open the script for editing and write it as follows:

using UnityEngine; using UnityEngine.UI;

public class ForceToggle : MonoBehaviour

{ private Toggle toggle;

private void Start()

{ toggle = GetComponent<Toggle>();

}

public void Toggle()

{ toggle.isOn = !toggle.isOn;

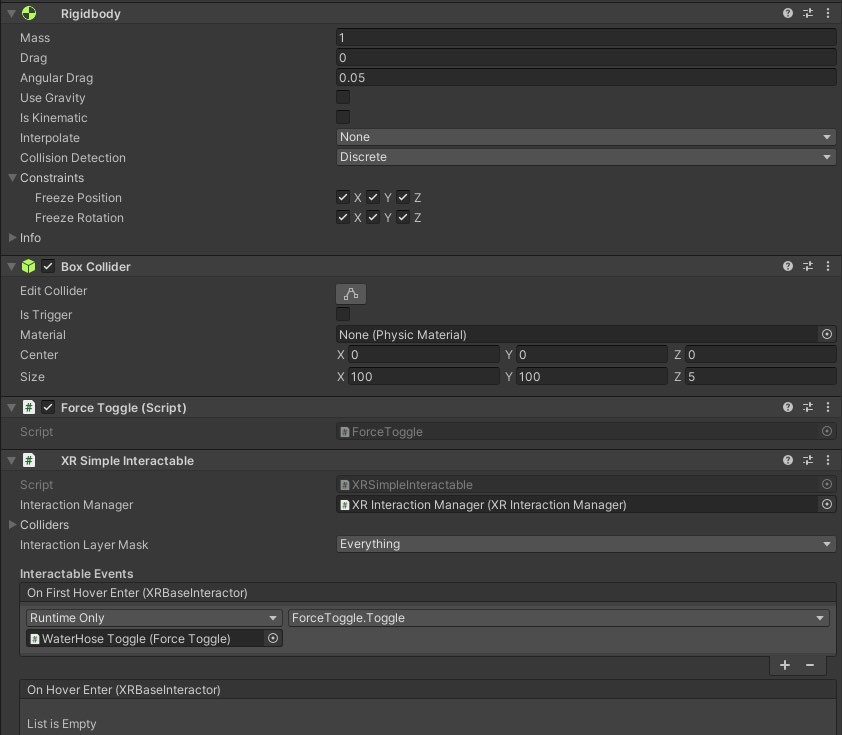
}

}

Modifying this isOn property will automatically invoke the OnValueChanged event. Now, we can use it in our UI:

1. Ensure this new script is added to the **WaterHose Toggle** object.
2. On the **XR Simple Interactable | On First Hover Enter** event actions, click the **+** button to add a new event response.
3. Drag the same **WaterHose Toggle** object from **Hierarchy** onto the **Runtime Only** object slot.
4. From the **Function** dropdown, select **ForceToggle | Toggle()**.

The new components we added to the **WaterHose Toggle** look like this:



Press **Play**. Move your hand toward the dashboard's **Hose** button. When it touches it, the button will turn green and the hose will start shooting water. Pull away and reach again to touch the button again. It toggles off to red and the water stops.

# Building a Wrist-based Menu Palette

Some VR applications that are designed for two-handed setups give you a virtual menu palette attached to the player's wrist while the other hand selects buttons or items from it.

Duplicate and re-purpose the Dashboard object to use it on your left wrist:

1. In **Hierarchy**, right-click on **Dashboard** and choose **Duplicate**.
2. Rename the new object Palette.
3. Disable the old Dashboard object.
4. Drag the **Palette** object so that it is a child of the **LeftHand Controller** object (under **XR Rig/Camera Offset**).

Modify the **Palette** graphics,

1. On the **Palette** obejct itself, set its **PosX**, **PosY**, and **PosZ** values to 0, 0.1, and -0.1, respectively, **Rotation** to 90, -150, -115, and **Width** and **Height** to 240 and 180, respectively.
2. In the **Panel** child of **Palette**, replace **Source Image** with **Background** (using the doughnut icon).
3. Move the **WaterHose Toggle** button to **Pos XYZ** (0, 0, 1).

END OF LAB 5