Lab 4: Interacting With Your Hands

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 4 Files are available on Learning zone under Lab Files
- **THIS IS A TWO-WEEK LAB – Lessons 4 and 5**

Let's make sure we have a scene with an XR camera rig by following these steps:

1. Open the Diorama screen we created earlier using **File | Open Scene** (or double- click the scene file in the **Project** window's Scenes/ folder).
2. Add an **XR Rig** using the XR-Rig Prefab.
3. Save the scene with a new name using **File | Save As**.

Setting up the Scene

For this scene, you could start with a new scene (**File | New Scene**) and then add an **XR Rig** from the **GameObject | XR.**

Start with the **Diorama** scene used in the previous lab and remove all but the **GroundPlane** and **PhotoPlane**, as follows:

1. Open the **Diorama** scene.
2. Remove all the objects, except for **XR Rig, XR Interaction Manager, Directional Light, GroundPlane** and **PhotoPlane**.
3. Position the **XR Rig** a few feet from the scene origin, **Position** (0, 0, -1).
4. Select **File | Save Scene As** and give it a name, such as Balloons.

Defining a Balloon Game Object

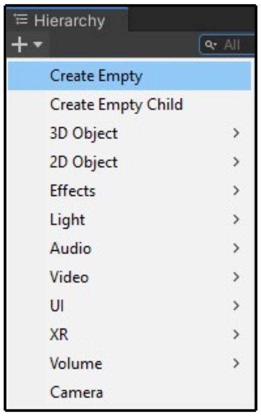
This is provided with the download files for this lab under the name BalloonModel.fbx.
Import the model into your project

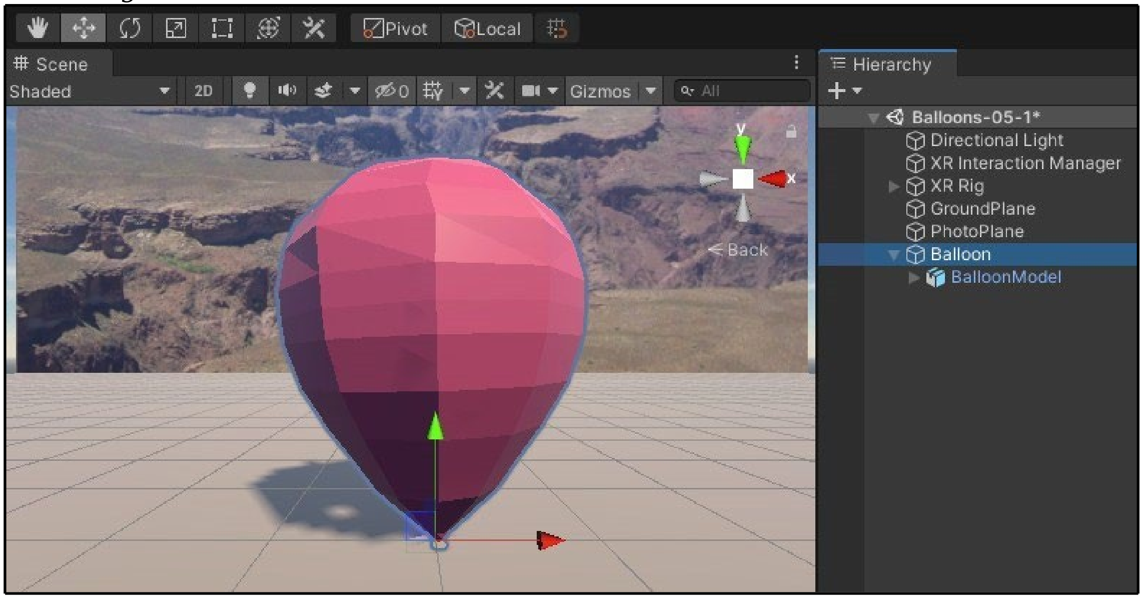
1. Create an asset folder named Models (in the **Project** window, right-click **Create | Folder** and name it).
2. By dragging and dropping, locate the BalloonModel.fbx file in Windows Explorer (or Finder on OSX), and drag it into the **Models** folder you just created. Alternatively, with the **Models** folder selected, navigate to **Assets | Import New Asset** from the main menu.

Now, when we add the model to the scene, we're going to *parent the object* (make it a child of another object) and adjust its position so that the bottom of the balloon model becomes its origin (pivot point), as follows:

1. In **Hierarchy**, create an empty object (**+ | Create Empty**, as shown in the following screenshot) and name it Balloon.
2. Reset its transform (**Transform | 3-dot-icon | Reset**).
3. Drag the balloon model into the **Hierarchy** as a child object of **Balloon** (for example, from the Assets/Models/ folder).
4. In the **Scene** window, use the **Move Tool** to adjust the position of the model so that it sits with its bottom aligned with the origin of its parent, such as (0, 0.5, 0
5. If you do not have a balloon model, then use a sphere (**Create | 3D Object | Sphere**) and add a material, like the **Red Material** we created in the previous lab.

Note that to create the game object you can use the **GameObject** menu in the main menu bar or use the **+** create menu in the **Hierarchy** window:





Making the balloon prefab

Add a Rigidbody and make the object a prefab, as follows:

1. Select your **Balloon** object in **Hierarchy**.
2. From the main menu, **Component | Physics | Rigidbody**.
3. Uncheck the Rigidbody's **Use Gravity** checkbox.
4. Drag the **Balloon** object from the **Hierarchy** to the **Project** window into your Prefabs/ folder to make it a prefab object.
5. Delete the original **Balloon** object from the **Hierarchy**.

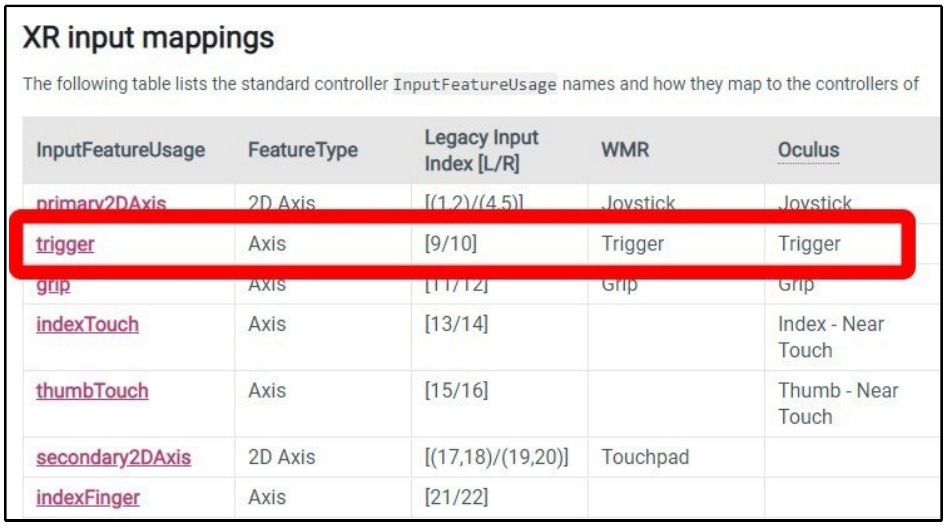
Creating a Balloon Controller

The last step in our initial scene setup is to create an empty game object named balloon controller and attach a script correspondingly named BalloonController.cs.

1. In **Hierarchy**, create an empty game object (**+** | **Create Empty**), reset its transform (**Transform | 3dot-icon | Reset**), and name it Balloon Controller.
2. Create a new script on the object named BalloonController (from the inspector, go to **Add Component | New Script |** BalloonController).
3. Open it for editing (double-click the script file).

Using an Input Manager button

The hand controllers for VR often have a lot of different buttons and axes, and it can get pretty complex. For example, you may think of the (index finger) trigger as a binary button that's either pressed or not, but on some controllers, you may also be able to detect just a finger touch and the amount of pressure you're putting on the trigger (read as an axis value between 0.0 and 1.0).



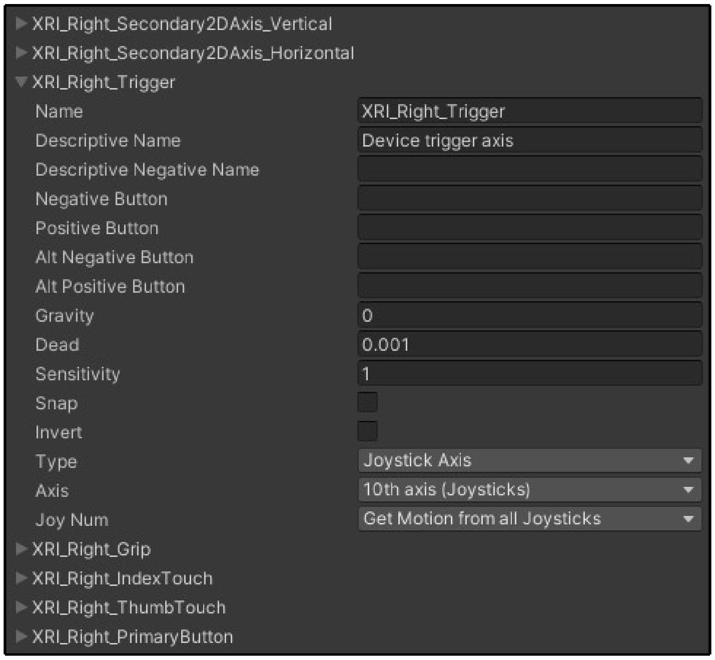
XR input mappings

The following table lists the standard controller `InputFeatureUsage` names and how they map to the controllers of

InputFeatureUsage	FeatureType	Legacy Input Index [L/R]	WMR	Oculus
primary2DAxis	2D Axis	[(1,2)/(4,5)]	Joystick	Joystick
trigger	Axis	[9/10]	Trigger	Trigger
grip	Axis	[11/12]	Grip	Grip
indexTouch	Axis	[13/14]		Index - Near Touch
thumbTouch	Axis	[15/16]		Thumb - Near Touch
secondary2DAxis	2D Axis	[(17,18)/(19,20)]	Touchpad	
indexFinger	Axis	[21/22]		

The Unity XR Platform has simplified this by providing a rich set of input bindings. Let's add these bindings now to our project as follows:

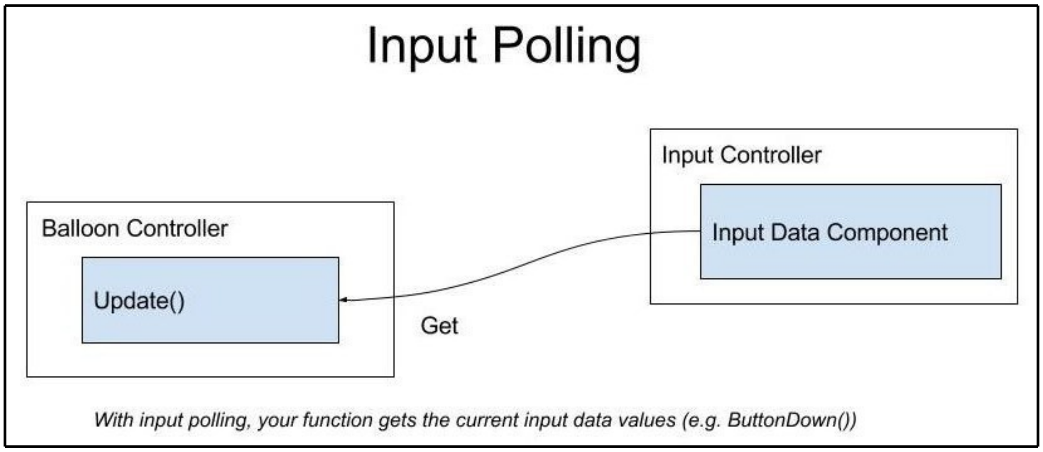
1. Open the Input Manager settings (**Project Settings | Input Manager**).
2. From the main menu, select **Assets | Seed XR Input Bindings**.
3. In **Input Manager**, if necessary, click to unfold the **Axes** settings.



▶ XRI_Right_Secondary2DAxis_Vertical	
▶ XRI_Right_Secondary2DAxis_Horizontal	
▼ XRI_Right_Trigger	
Name	XRI_Right_Trigger
Descriptive Name	Device trigger axis
Descriptive Negative Name	
Negative Button	
Positive Button	
Alt Negative Button	
Alt Positive Button	
Gravity	0
Dead	0.001
Sensitivity	1
Snap	☐
Invert	☐
Type	Joystick Axis
Axis	10th axis (Joysticks)
Joy Num	Get Motion from all Joysticks
▶ XRI_Right_Grip	
▶ XRI_Right_IndexTouch	
▶ XRI_Right_ThumbTouch	
▶ XRI_Right_PrimaryButton	

Polling the XRI_Right_Trigger button

The simplest way to obtain user input is to just *get* the current state from the Input Manager. This process of polling the input state is shown in the following diagram:


Input Polling
Input Controller
Input Data Component
Balloon Controller
Update()
Get
With input polling, your function gets the current input data values (e.g. ButtonDown())

Open the BalloonController.cs file in your editor and edit the script as follows:

```
public class BalloonController : MonoBehaviour
{ void
  Update()
  { if
    (Input.GetButtonDown("XRI_Right_TriggerButton"
    ))
    {
      Debug.Log("Trigger down");
    } else if
    (Input.GetButtonUp("XRI_Right_TriggerButton"))
    {
      Debug.Log("Trigger up");
    }
  }
}
```

Let's try it out by going through the following steps:

1. Press **Play** in the Unity editor to run the scene.
2. When you press the trigger on your input controller, you will see the **Input: Trigger down** message as output.
3. When you release the trigger button, you will see the **Input: Trigger up** message.

Controlling Balloons with the Input Trigger

The BalloonController.cs script should now create a new balloon when the trigger button gets pressed. In your code editor, change the Update function to the following:

```
void Update()
  { if (Input.GetButtonDown("XRI_Right_TriggerButton"))
      {
          CreateBalloon();
       }
   }
```

Declare a public GameObject variable named balloonPrefab at the top of the BalloonController class and add a private variable to hold the current instance of the balloon, as follows:

```
public class BalloonController : MonoBehaviour
{ public GameObject balloonPrefab;
     private GameObject balloon;
     ...
```

Add the CreateBalloon function that calls the Unity Instantiate function to create a new instance of the balloonPrefab and assign it to the balloon variable:

```
public void CreateBalloon()
{ balloon = Instantiate(balloonPrefab); }
```

Releasing Balloons

The ReleaseBalloon function should be called when the player releases the trigger button. Let's add that to Update now, as follows:

```
void Update()
   { if (Input.GetButtonDown("XRI_Right_TriggerButton"))
      {
        CreateBalloon();
      }                         else                          if
        (Input.GetButtonUp("XRI_Right_TriggerButton"))
         {
           ReleaseBalloon();
          }
       }
```

Add this variable with the others declared at the top of the BalloonController script:

```
public    GameObject    balloonPrefab;
public   float   floatStrength   =
20f;  private GameObject balloon;
```

Write the ReleaseBalloon function as follows:

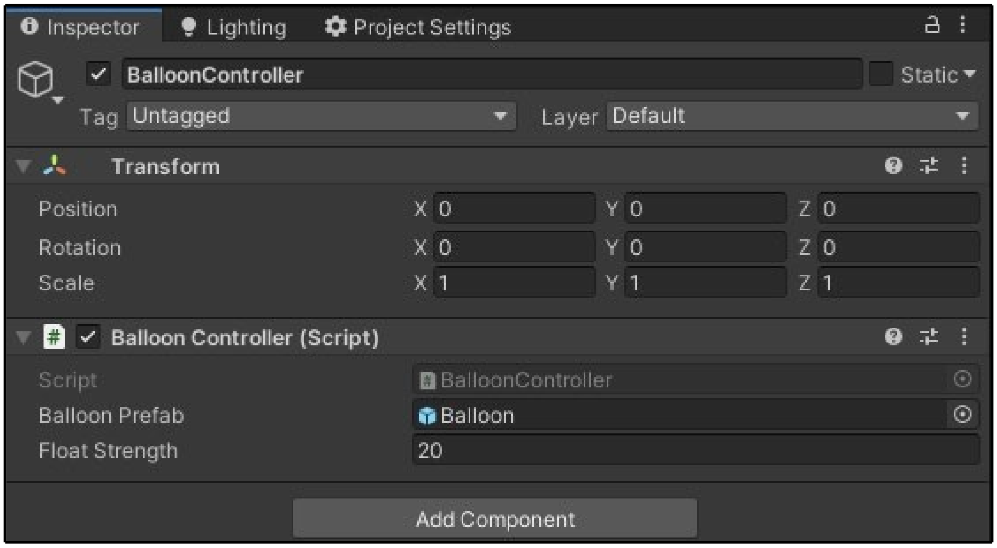
```
public void ReleaseBalloon()
{
     Rigidbody rb = balloon.GetComponent<Rigidbody>(); Vector3
     force = Vector3.up * floatStrength; rb.AddForce(force);

     GameObject.Destroy(balloon, 10f); balloon =
     null;
}
```

Save the file. Go back into Unity and assign the Balloon prefab to the variable in our script as follows:

1. In the **Hierarchy** window, make sure that the **Balloon Controller** is currently selected.
2. In the **Project** window, locate the **Balloon Prefab** that we created earlier.
3. Drag the **Balloon** prefab from the **Project** window to the **Inspector** window and onto the BalloonController's **Balloon Prefab** slot.

When you're ready, press **Play**. Then, inside the VR world, when you press the trigger button, a new balloon will be instantiated.



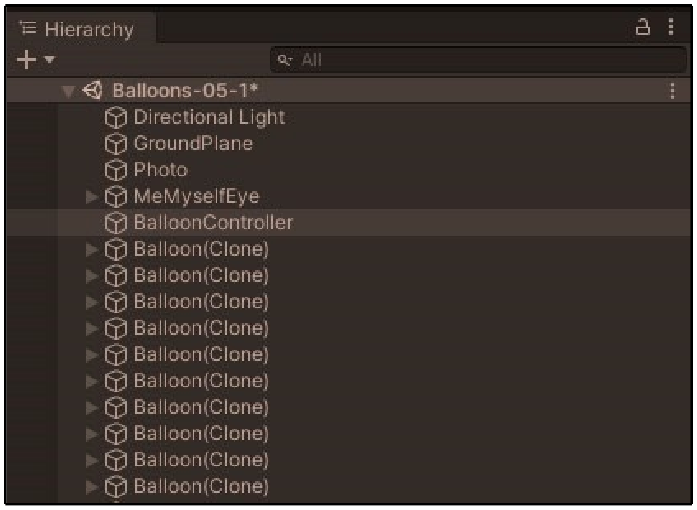
BalloonController
Tag Untagged Layer Default

Transform

	X	Y	Z
Position	0	0	0
Rotation	0	0	0
Scale	1	1	1

Balloon Controller (Script)

Script	BalloonController
Balloon Prefab	Balloon
Float Strength	20



```
Balloons-05-1*
  Directional Light
  GroundPlane
  Photo
  MeMyselfEye
  BalloonController
  Balloon(Clone)
  Balloon(Clone)
  Balloon(Clone)
  Balloon(Clone)
  Balloon(Clone)
  Balloon(Clone)
  Balloon(Clone)
  Balloon(Clone)
  Balloon(Clone)
  Balloon(Clone)
```

Inflating a balloon while pressing the trigger

Create a small balloon at one-tenth of its original size.

```
public void CreateBalloon()
{ balloon = Instantiate(balloonPrefab); balloon.transform.localScale =
new Vector3(0.1f, 0.1f, 0.1f); }
```

Declare a growRate variable that will be used to change the balloon scale.

```
public float growRate = 1.5f;
```

Modify the Update function with a third if condition as follows:

```
else if (balloon != null)
{
      GrowBalloon();
}
```

Add the GrowBalloon function like this:

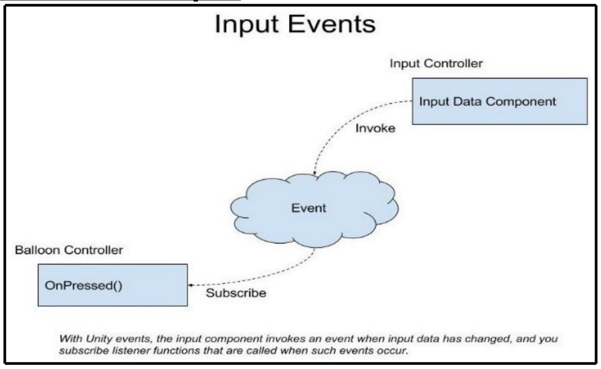
```
public void GrowBalloon()
{ float growThisFrame = growRate * Time.deltaTime;
   Vector3 changeScale = balloon.transform.localScale * growThisFrame;
balloon.transform.localScale += changeScale; }
```

Refactor into a single line of code, as shown in the following code:

```
public void GrowBalloon()
{ balloon.transform.localScale += balloon.transform.localScale
          * growRate * Time.deltaTime;
}
```

Press **Play** in Unity. When you press the controller button, you'll create a balloon, which continues to inflate until you release the button. Then the balloon floats up.

Using Unity Events for Input



With Unity events, the input component invokes an event when input data has changed, and you subscribe listener functions that are called when such events occur.

To begin, create an empty **Input Controller** game object and add a corresponding new C# script:

1. In **Hierarchy**, create an empty game object (**+ | Create Empty**), reset its transform (**Transform | 3dot-icon | Reset**), and name it Button Input Controller.
2. Create a new script on the object named ButtonInputController (**Add Component | New Script**).
3. Open it for editing (double-click the script file).

Invoking our Input Action Events

To implement our example using events, we'll first have the ButtonInputController trigger events when the trigger button is pressed and another event when the button is released.

The entire ButtonInputController.cs is as follows:

```
using UnityEngine;
using UnityEngine.Events;

public class ButtonInputController : MonoBehaviour
{ public UnityEvent ButtonDownEvent = new UnityEvent(); public
      UnityEvent ButtonUpEvent = new UnityEvent();

      void Update()
      { if (Input.GetButtonDown("XRI_Right_TriggerButton")) {
                  ButtonDownEvent.Invoke();
            } else if (Input.GetButtonUp("XRI_Right_TriggerButton"))
            {
                  ButtonUpEvent.Invoke();
            }
      }
}
```

Subscribing to Input Events

Modify the BaloonControoler Update function so it reads as follows:

```
void Update()
{ if (balloon != null)
      {
            GrowBalloon();
      }
}
```

To wire up the input events to the Balloon Controller, go through the following steps:

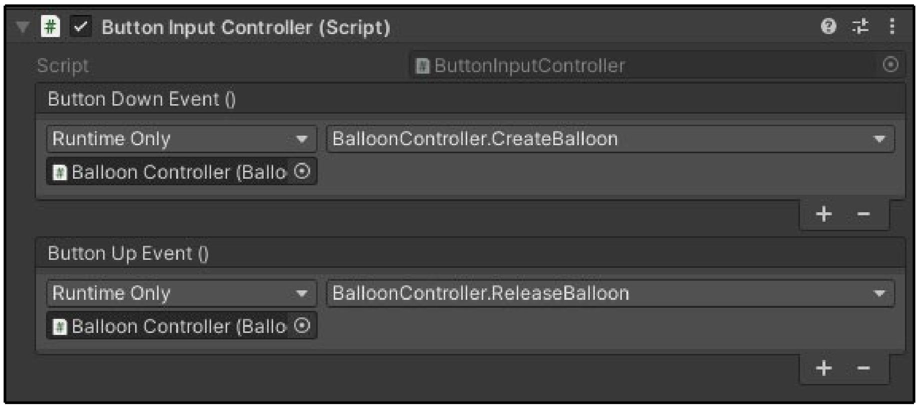
1. In **Hierarchy**, select **Button Input Controller** and look at its **Inspector** window.

2. You will see that the script component now has two event lists, as we declared them in its script.
3. On the **Button Down Event** list, click the + in the lower-right corner to create a new item.
4. Drag the **BalloonController** from **Hierarchy** into the empty object slot.
5. In the function select list, choose **BalloonController | CreateBalloon**.

Repeat the process for the Button Up Event as follows:

1. On the **Button Up Event** list, press the **+** in the lower-right corner to create a new item.
2. Drag the **BalloonController** from **Hierarchy** into the empty object slot.
3. In the function select list, choose **BalloonController | ReleaseBalloon**.

The component should now look like this:



Button Input Controller (Script)

Script: ButtonInputController

Button Down Event ()

Runtime Only — BalloonController.CreateBalloon

Balloon Controller (Ballo

Button Up Event ()

Runtime Only — BalloonController.ReleaseBalloon

Balloon Controller (Ballo

Now, when you press **Play** and press the trigger, the input controller invokes the **Button Down Event**, and the CreateBalloon function that is listening for these events gets called (likewise for the **Button Up Event**). When the trigger is released, the input controller invokes the other event, and ReleaseBalloon gets called.

If you wanted to subscribe to the button events as follows:

```
void Start()
{               buttonController.ButtonDownEvent.AddListener(CreateBalloon);
buttonController.ButtonUpEvent.AddListener(ReleaseBalloon); }
```

Tracking Your Hands

To start taking advantage of the positional tracking of your hands, we simply need to parent the balloon prefab to the hand model. Our scene includes an **XR Rig** that contains not only the **Main Camera** that is positionally tracked with the player's head-mounted display, but it also contains a **LeftHand Controller** and **RightHand Controller** that are tracked with the player's hand controllers.

Parenting the Balloon to Your Hand

Pass the hand GameObject to theYCreateBalloon function and then pass its transform to the Instantiate function, as follows:

```
public void CreateBalloon(GameObject parentHand)
{ balloon = Instantiate(balloonPrefab, parentHand.transform);
balloon.transform.localScale = new Vector3(0.1f, 0.1f, 0.1f); }
```

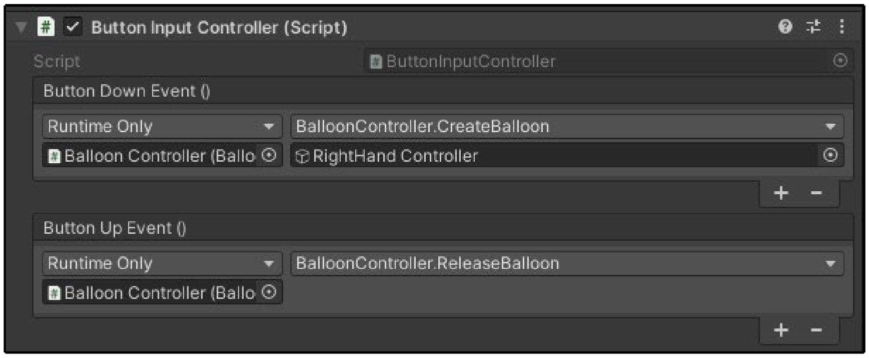
ReleaseBalloon must detach the balloon from the hand before sending it on its way, as follows:

```
public void ReleaseBalloon()
{ balloon.transform.parent = null;
      balloon.GetComponent<Rigidbody>().AddForce(Vector3.up
         * floatStrength);
      GameObject.Destroy(balloon, 10f); balloon =
      null;
}
```

In the **Inspector,** we need to update the **Button Down Event** function, since it now requires the game object argument:

1. In the Unity editor, select the **Button Input Controller** object.
2. In the **Button Down Event** list, the function may now say something like <Missing BalloonController.CreateBalloon>.
3. Select the **Function** dropdown and choose **BalloonController | CreateBalloon(GameObject)**.
4. Unfold the **XR Rig** object in **Hierarchy** and look for the **RightHand Controller** object, then drag it onto the empty **Game Object** slot.

The **Button Input Controller** component now looks like this:


Button Input Controller (Script)

Script: ButtonInputController

Button Down Event ()

Runtime Only | BalloonController.CreateBalloon

Balloon Controller (Ballo | RightHand Controller

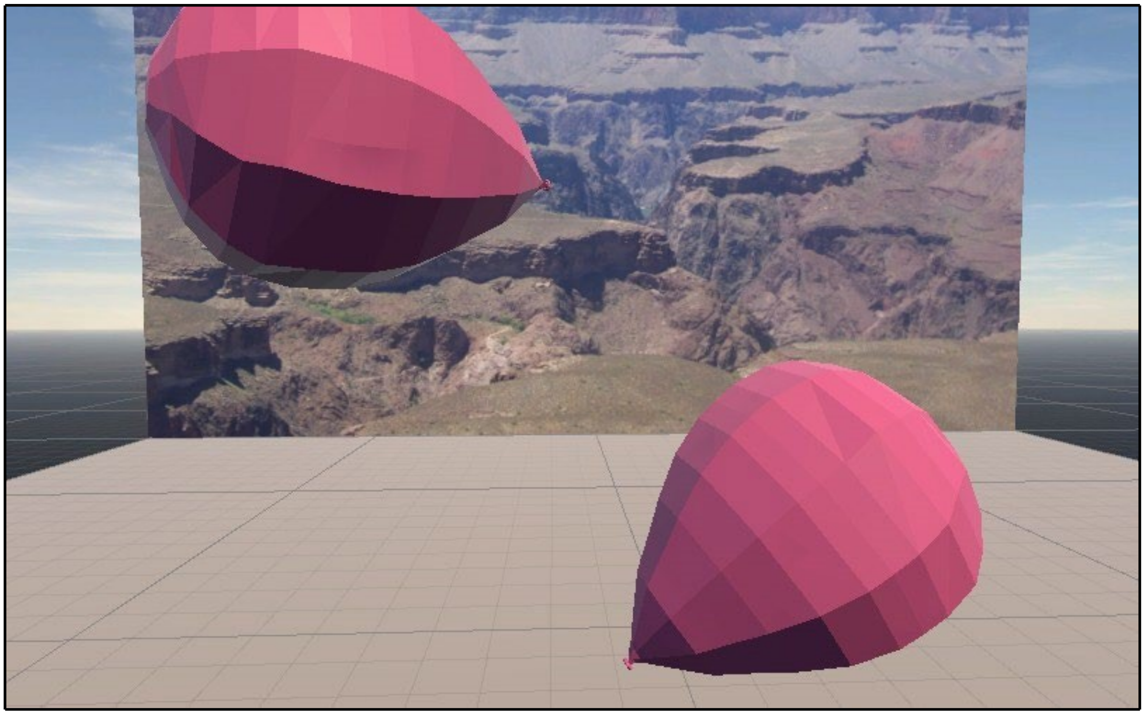
Button Up Event ()

Runtime Only | BalloonController.ReleaseBalloon

Balloon Controller (Ballo

Press **Play**. When you press the trigger button, the balloon is created attached to your hand, and when released, it is released into the scene from that hand location at the time.

Forcing Balloons to Float Upright



Write a component script, KeepUpright, that orients the balloon to remain upright.

Let's write this script now and add it to the **Balloon** prefab object:

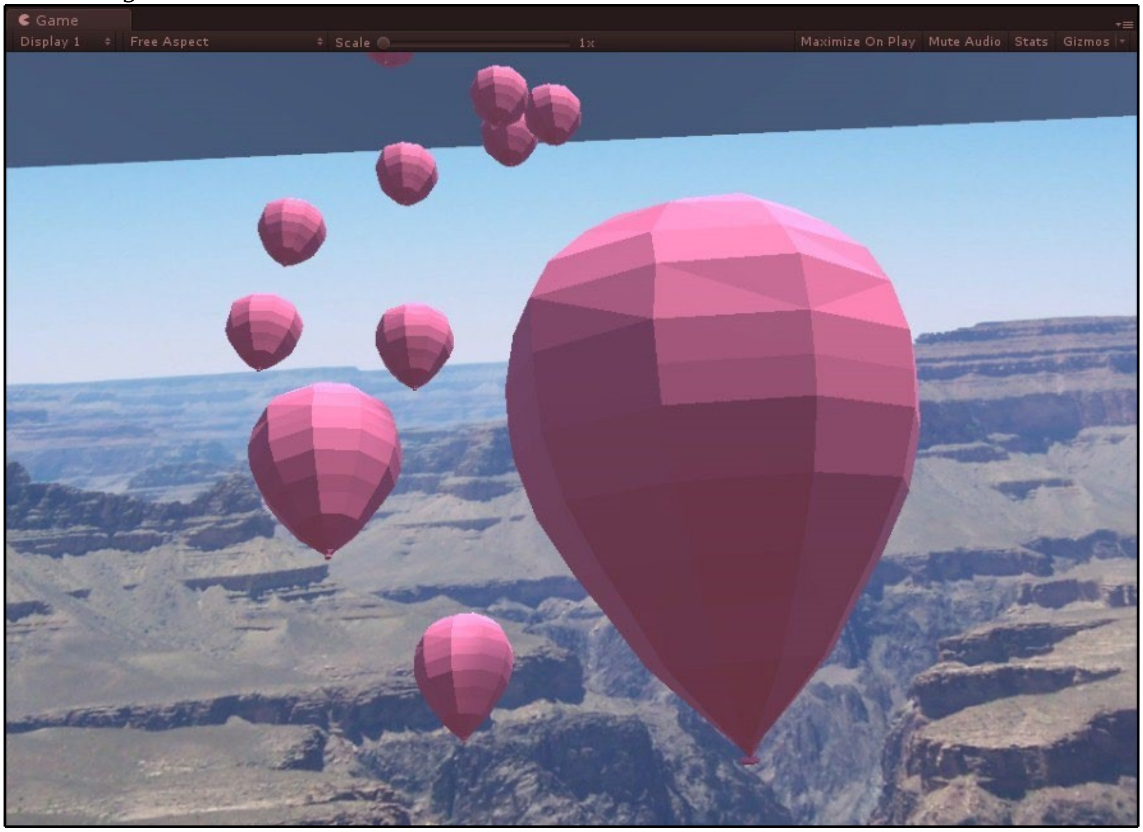
1. Select your **Balloon** prefab in the **Project** window (in the Assets/Prefabs/ folder).
2. Create and add a new script named KeepUpright (by navigating to **Add Component | New Script |** KeepUpright **| Create And Add**).
3. For tidiness, in the **Project** window, drag the new script into your scripts folder (Assets/Scripts/).
4. Double-click the script to open it for editing.

Edit the script as follows:

```
public class KeepUpright : MonoBehaviour
{ public float speed = 5f;

  void Update()
  {
  transform.rotation         =         Quaternion.Slerp(transform.rotation,
  Quaternion.identity, Time.deltaTime * speed); }
}
```

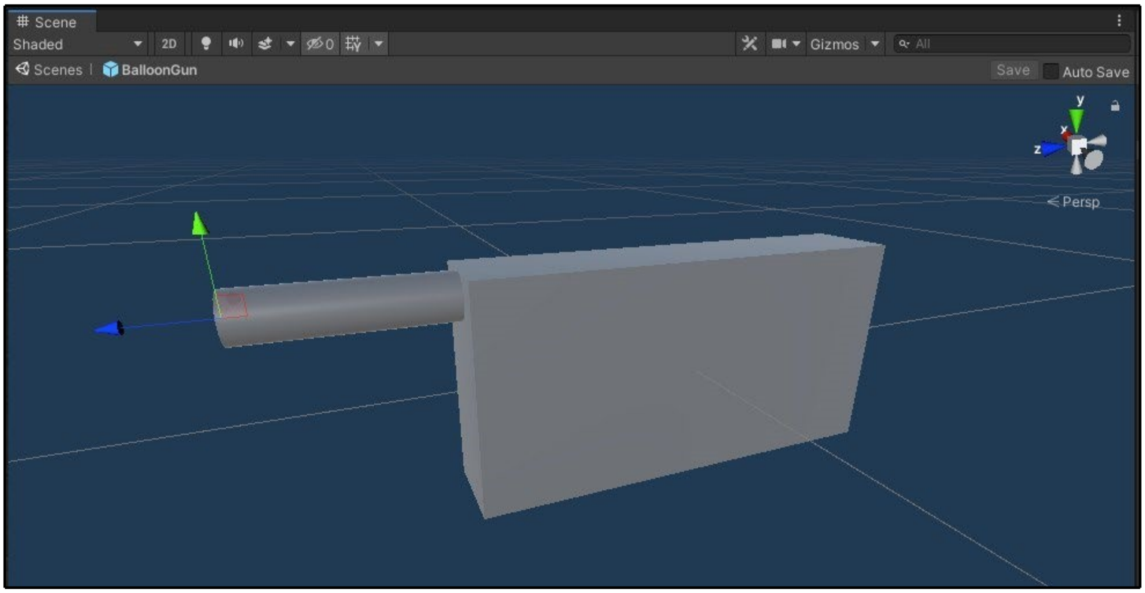
Press **Play**. Here is a screenshot of generating a bunch of balloons that appear to be flying over the Grand Canyon.



Creating a grabbable balloon gun

We'll create a grabbable gun. If you have a gun model, you can use that.
The important things are to make this model into a prefab and make sure that it includes a Transform that we can use to define the tip of the gun where new balloons will be instantiated:

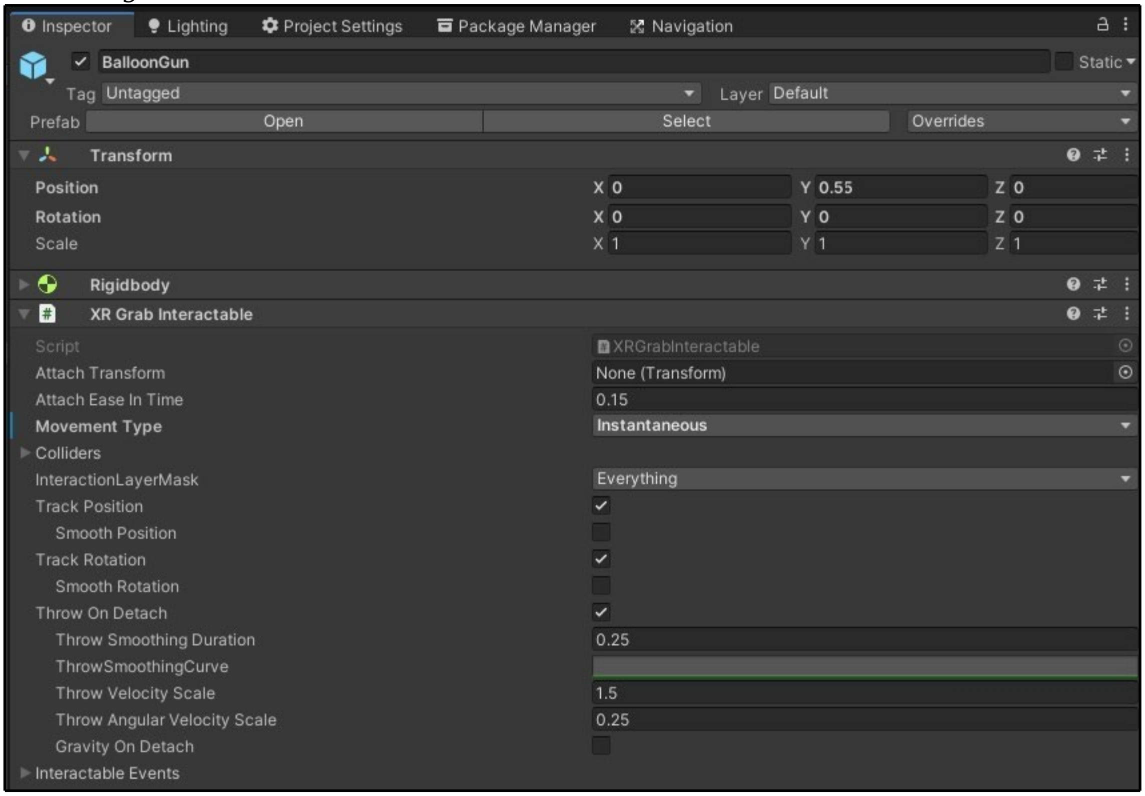
1. In **Hierarchy**, create an empty game object named BalloonGun (**+ | Create Empty**), and reset its **Transform** (**3-dot-icon | Reset**).
2. For its Body, create a child 3D **Cube** and **Scale** it to (0.04, 0.1, 0.2).
3. For its Barrel, create a child 3D **Cylinder**, with **Scale** (0.02, 0.04, 0.02), **Rotation** (90, 0,0), and **Position** (0, 0.04, 0.14).
4. Define the tip of the barrel where balloons will be instantiated and create an **Empty GameObject** named Tip at **Position** (0, 0.04, 0.18).



Make it an Interactable object as follows:

1. Ensure that the root **BalloonGun** object is selected in **Hierarchy**.
2. Add a **XR Grab Interactable** component (from the main menu, select **Components | XR | XR Grab Interactable**).
3. Change the Interactable **Movement Type** to **Instantaneous**.

Note that adding the Interactable also adds a **Rigidbody** to the object, if one is not already there. The **BalloonGun**'s **Inspector** now looks as follows:



BalloonGun Static
Tag Untagged Layer Default
Prefab

Transform

	X	Y	Z
Position	0	0.55	0
Rotation	0	0	0
Scale	1	1	1

Rigidbody

XR Grab Interactable

Script	XRGrabInteractable
Attach Transform	None (Transform)
Attach Ease In Time	0.15
Movement Type	Instantaneous
Colliders	
InteractionLayerMask	Everything
Track Position	✓
Smooth Position	
Track Rotation	✓
Smooth Rotation	
Throw On Detach	✓
Throw Smoothing Duration	0.25
ThrowSmoothingCurve	
Throw Velocity Scale	1.5
Throw Angular Velocity Scale	0.25
Gravity On Detach	
Interactable Events	

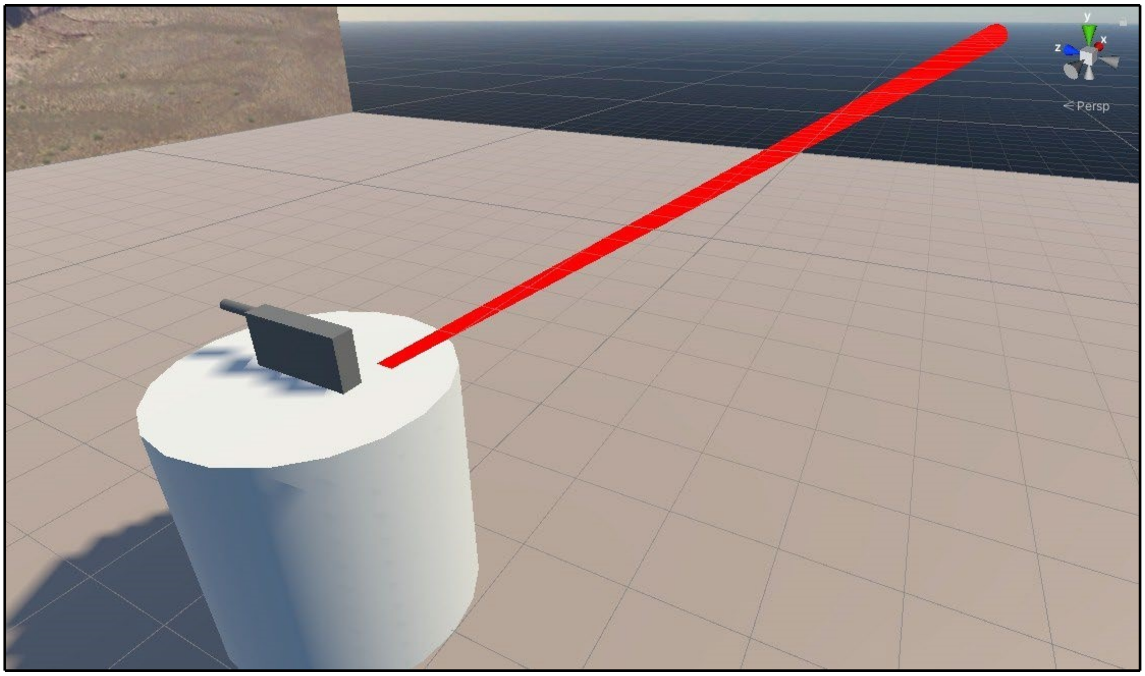
Add the gun to the scene. Let's build a pedestal to place our objects on:

1. Create a cylinder at the root of your **Hierarchy** (**+ | 3D Object | Cylinder**) and rename it Pedestal.
2. Set its **Scale** (0.5, 0.25, 0.5) and **Position** at the origin (0, 0.25, 0).
3. Make it white by creating a new material or using an existing one (by dragging **White Material** from **Project** Assets/Materials/ onto the **Pedestal**).
4. Place the gun on the pedestal by setting its **Position** to (0, 0.55, 0).

Replace the capsule with a mesh that matches the cylinder's actual shape, as follows:

1. Disable or remove the Pedestal's **Capsule Collider** component (**3-dot-icon | Remove Component**).
2. Add the **Mesh Collider** (**Component | Physics | Mesh Collider**).

Press **Play** to try it out in VR now. Using the ray interactor with either hand, you can point the laser at the gun, then squeeze the grip button (middle finger) (as Select Usage designates) and it will fly into your hand. Release the grip button and it drops to the ground, or you can throw it. The following is a screen capture of the Scene view using the laser ray just before grabbing the gun:



Handling Activate Events

Copy the existing **Balloon Controller (Script)** component onto the **BalloonGun** GameObject by going through the following steps:

1. In **Hierarchy**, select the **Balloon Controller**.
2. In **Inspector**, select the **3-dot-icon** for the **Balloon Controller** component and select **Copy Component**.
3. In **Hierarchy**, select the **BalloonGun**.
4. In **Inspector**, select the **3-dot-icon** of the **Transform** (or any other) component, and select **Paste Component As New**.
5. In **Hierarchy**, select the **Balloon Controller** and delete or disable it (**right-click | Delete**).

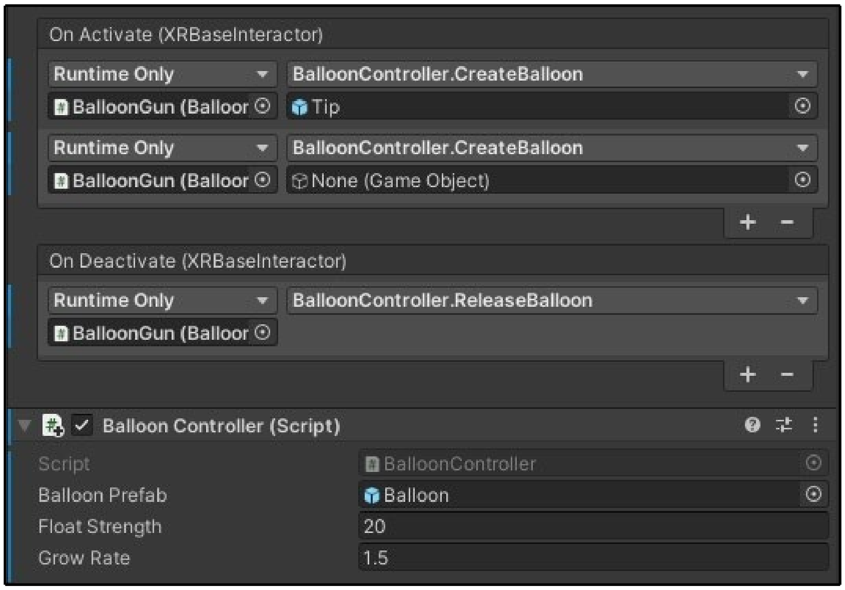
Define the **On Activate** event to call the CreateBalloon function in BalloonController

1. Select the **BalloonGun** in **Hierarchy**.
2. In **Inspector**, unfold the **Interactable Events** list on its **XR Grab Interactable** component.

3. For the **On Activate** event, click the **+** button to add one
4. Drag this **BalloonGun** object from **Hierarchy** onto the **Object** slot.
5. In its **Function** dropdown, select **BalloonController | CreateBalloon**.
6. From **Inspector**, drag the **Tip** child object onto the **Game Object** parameter slot.

When the user releases the trigger, the **On Deactivate** event is invoked, and we want to release the balloon. Let's set this up by going through the following steps:

1. For the **On Deactivate** event, click the **+** button.
2. Drag this **BalloonGun** object from **Hierarchy** onto the **Object** slot.
3. In its **Function** dropdown, select **BalloonController | ReleaseBalloon**.



On Activate (XRBaseInteractor)

Runtime Only	BalloonController.CreateBalloon
BalloonGun (Balloor	Tip
Runtime Only	BalloonController.CreateBalloon
BalloonGun (Balloor	None (Game Object)

On Deactivate (XRBaseInteractor)

Runtime Only	BalloonController.ReleaseBalloon
BalloonGun (Balloor	

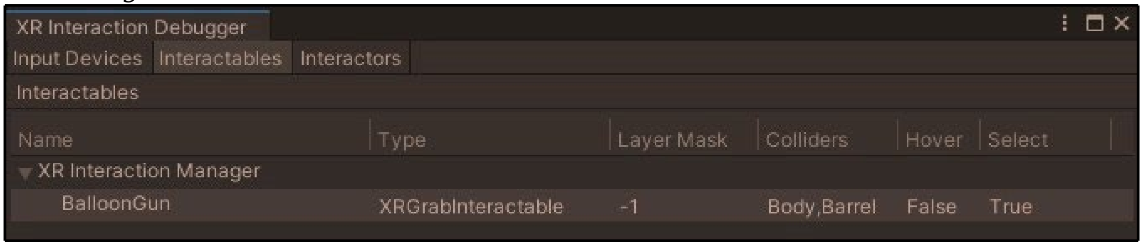
Balloon Controller (Script)

Script	BalloonController
Balloon Prefab	Balloon
Float Strength	20
Grow Rate	1.5

When you **Play** the scene, you should now be able to grab the gun with either hand and use the trigger button to create, inflate, and release balloons. Now our interactable gun is not just a gun, but a *balloon gun*!

Using the XR Interaction Debugger

To open the debugger, select **Window | Analysis | XR Interaction Debugger**


XR Interaction Debugger

Input Devices | Interactables | Interactors

Interactables

Name	Type	Layer Mask	Colliders	Hover	Select
XR Interaction Manager					
BalloonGun	XRGrabInteractable	-1	Body,Barrel	False	True

Popping Balloons

First, we'll make the balloons poppable with collision detection and an explosion. Then we'll add a ball to the scene that you can throw at a balloon to pop it.
And after the ball is thrown, we'll fetch it back by resetting it to its original position after a short delay.

Making the Balloons Poppable

Set up on the balloon prefab:

1. In the **Project** Prefabs/ folder, open your Balloon prefab for editing by double- clicking it.
2. Select **Component | Physics | Sphere Collider**.
3. To scale and center the Collider into position, click the **Edit Collider** icon in its component.
4. In the **Scene** window, the green Collider outline has small anchor points that you can click to edit. Note that the *Alt* key pins the center position and *Shift* locks the scale ratio.
5. Or you can edit the **Center** and **Radius** values directly. I like **Radius** 0.22 and **Center** (0, 0.36, 0) on my balloon.

Add a script to handle the collision events:

1. Create a new C# script on the **Balloon**, named Poppable.
2. Open it for editing.

The Poppable script will provide a callback function for OnCollisionEnter events. When another object with a Collider enters this object's Collider, our function will get called. At that point, we'll call PopBalloon, which instantiates the explosion and destroys the balloon:

```
public class Poppable : MonoBehaviour
{ public GameObject popEffectPrefab;

    void OnCollisionEnter(Collision collision)
    { if (transform.parent == null &&
            collision.gameObject.GetComponent<Poppable>() == null)
        {
            PopBalloon();
        }
    }

    private void PopBalloon()
    { if (popEffectPrefab != null)
        {
            GameObject effect = Instantiate(popEffectPrefab, transform.position,
                transform.rotation);
            Destroy(effect, 1f);
        }
        Destroy(gameObject);
    }
}
```

Save the script. Try it in Unity. Press **Play**. When you create a balloon and release it, whack it with your hand. It disappears.

Adding a Popping Explosion

If you recall, in Lab 3 *Using Gaze-Based Control*, we used an explosion particle-effect prefab from Standard Assets. Let's reuse that here, with some modifications to make it smaller (and less like a hand grenade!):

1. In the **Project** window, navigate to the StandardAssets/ParticleSystems /Prefabs/ folder.
2. Select and duplicate the **Explosion** prefab (**Edit | Duplicate**).
3. Rename the copy as BalloonPopEffect and move it into your own Prefabs/ folder.

Edit the **BalloonPopEffect** prefab by double-clicking it and going through the following steps:

1. In the root **BalloonPopEffect**, set its **Transform Scale** to (0.01, 0.01, 0.01).
2. Remove or disable the **Explosion Physics Force** component (**right-click | Remove Component**).
3. Remove or disable the **Light** component.
4. Change the **Particle System Multiplier's Multiplier** value to 0.01.
5. In **Hierarchy**, delete or disable the child **Fireball** game object.
6. Delete or disable the child **Shockwave** game object.
7. Save the prefab and exit back to the scene hierarchy.

Now assign the effect to the Balloon:

1. Open the **Balloon** prefab again by double-clicking on it.
2. Drag the **BalloonPopEffect** prefab onto the **Poppable**'s **Pop Effect Prefab** slot.
3. Save and exit to the scene.

Press **Play**, create and release a balloon, and hit it with your hand.

Disabling Rigid Physics While in Hand

To implement this, we just need to add a few things. First, at the top of the class, add a variable to hold the current balloon's Rigidbody, as follows:

```
private Rigidbody rb;
```

In the CreateBalloon function, add the following lines to get the balloon's Rigidbody and make
it kinematic:

```
public void CreateBalloon(GameObject parentHand)
{     balloon     =     Instantiate(balloonPrefab,     parentHand.transform);
      balloon.transform.localScale = new Vector3(0.1f, 0.1f, 0.1f); rb =
      balloon.GetComponent<Rigidbody>(); rb.isKinematic = true;
}
```

Then, restore the physics in ReleaseBalloon, as follows:

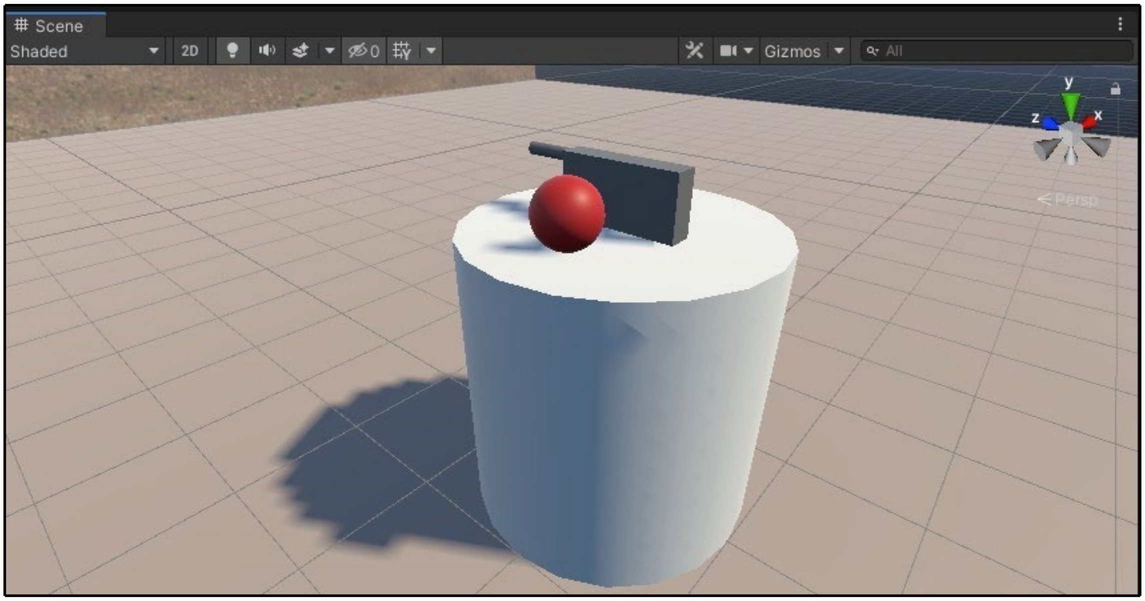
```
public void ReleaseBalloon()
{        rb.isKinematic        =        false;
      balloon.transform.parent          =          null;
      rb.AddForce(Vector3.up     *     floatStrength);
      GameObject.Destroy(balloon,  10f);  balloon  =
      null;
}
```

Throwing a Ball Projectile

Our final step is to add a ball to the scene that also has a Collider so that you can throw it at a balloon to pop it.

1. In the scene **Hierarchy** root, create a new sphere (by going to **+ | 3D Object | Sphere**) and name it Ball.
2. **Scale** the ball to (0.1, 0.1, 0.1) and place it on the **Pedestal** at **Position** (-0.125, 0.55, 0).
3. Make it red by adding new material or using the **Red Material** in your Materials/ folder (drag the material onto the **Ball**).
4. Add a **Grab Interactable** component (C**omponent | XR | XR Grab Interactable**).

5. On its **Rigidbody** component, change the **Collision Detection** to **Continuous**.
6. On its **XR Grab Interactable** component, change the **Throw Velocity Scale** to 2.



Press **Play**. Grab the gun with one hand and inflate a balloon. Grab the ball with the other hand and throw it at the balloon.

Resetting the Ball Position

To reset the ball position back onto the pedestal after you've thrown it, we'll create one more script. Create a new C# script named ResetAfterDelay, attach it to the **Ball**, and edit it as follows:

```
      public class ResetAfterDelay : MonoBehaviour
      { public float delaySeconds = 5f;
        private Vector3 startPosition;
private void Start()
        { startPosition = transform.position;
        }


        public void Reset()
        {
        StartCoroutine(AfterDelay());
        }


        IEnumerator AfterDelay()
```

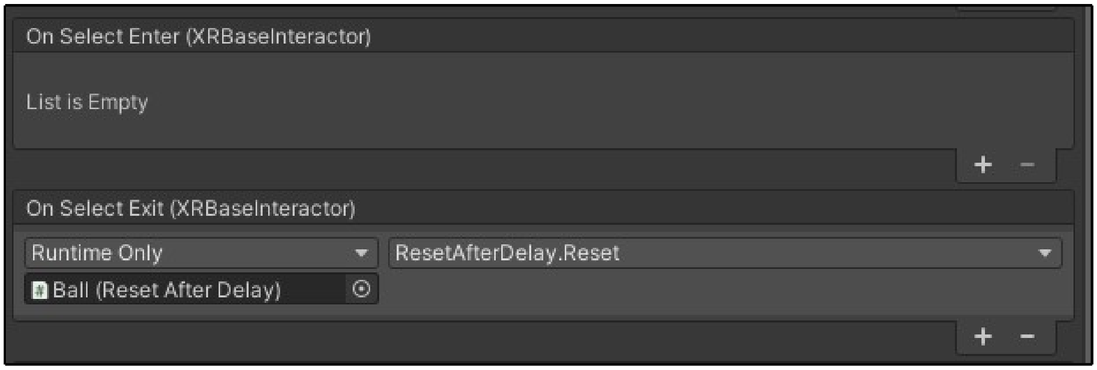
```
    {
    yield return new WaitForSeconds(delaySeconds);
    transform.position = startPosition;

    Rigidbody rb = GetComponent<Rigidbody>();
    rb.velocity = Vector3.zero; rb.angularVelocity =
    Vector3.zero;
    }
   }
```

Save your script, and back in Unity, we can wire up the Reset function by calling the function when the player throws (or otherwise lets go of) the Ball. This is when the **XR Grab Interactable**'s **On Select Exit** event is invoked:

1. With the **Ball** selected in **Inspector**, unfold the **Interactable Events** list on its **XR Grab Interactable** component.
2. At the **On Select Exit** event, add a new response by pressing its **+** button (in the lower-right corner).
3. Drag the **Ball** from **Hierarchy** onto the event's **Object** slot.
4. In the **Function** drop down, select **ResetAfterDelay | Reset**.

The Inspector is shown in the following screenshot:


On Select Enter (XRBaseInteractor)
List is Empty
On Select Exit (XRBaseInteractor)
Runtime Only
ResetAfterDelay.Reset
Ball (Reset After Delay)

When you press **Play**, grab the ball and throw it. Five seconds later, the ball will return to its original position.

END OF LAB 4