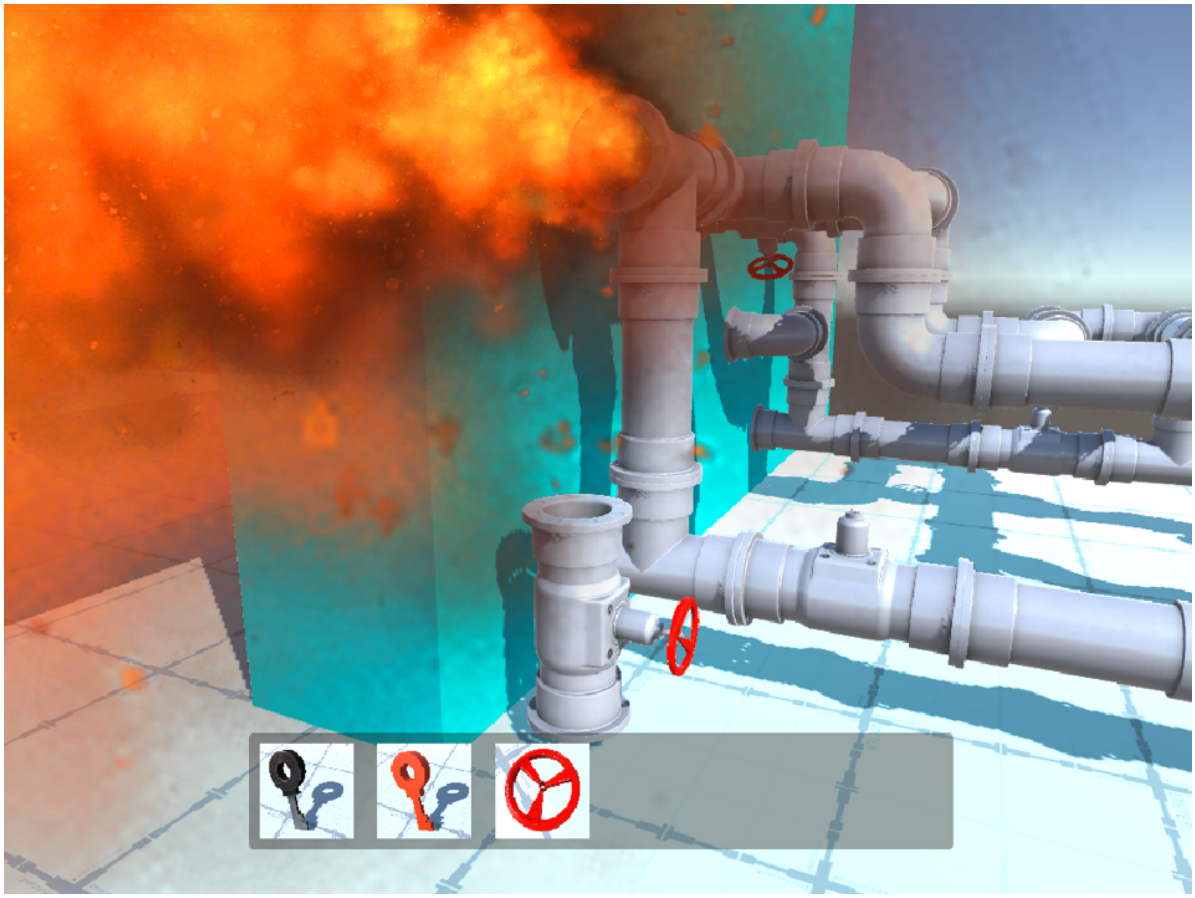
COMP4002/G54GAM Lab Exercise 06 – Inventory and Items

12/03/19

This lab exercise involves creating another common and highly re-usable game mechanic - objects that the player can pick up and carry in their inventory, and subsequently use to overcome obstacles in the form of “locked door” challenges.



The game will have the following features:

- A first or third person character
- Items that the player can pick up and store in an inventory
- The ability to use an individual item from the inventory
- Obstacles that require the selection of a specific item to pass

Start by creating a new project with some terrain, or copy and extend one of the previous two exercises - it doesn’t matter if you use a first or third person character perspective.

This exercise will make use of the *Delegate* software pattern, in particular using *Events* to allow the different components to communicate.

Inventory Architecture

There are multiple components that will make up the inventory functionality. It is important that you implement this in stages, creating stub methods that do little more than print out debug text to indicate that one element is working as expected as you build this up.

Functionality is comprised of:

- Items
  - That implement a common interface
    - Enabling them to be "picked up"
    - Require them to have a *Sprite* so they can be displayed in the inventory

- A Game Object for the Inventory itself
  - Maintains a list of items
  - Broadcasts an event when a new item has been added, and when an item is used

- A head-up display (HUD)
  - Listens for new items being added to the inventory
  - Displays items currently in the inventory
  - Has a button for each item in the inventory allowing it be used

- Doors
  - That become active when the player is in range
  - That listen for an event to say that a specific item (i.e. the key) has been used
  - That open if both conditions are met

To begin with, we will create a simple *key* item that, once collected, can be used to open a locked *door*.

Inventory Items

- Create a new script *IInventoryItem* to serve as the common interface for items that can be kept in the inventory.

Interfaces cannot contain property fields as they are merely interfaces rather than abstract classes, but by including the *get* accessor we can require the implementing class to provide this value, essentially adding a property field to the interface.

```
public interface IInventoryItem
{
    string itemName { get; }

    Sprite itemImage { get; }

    void onPickup();
}
```

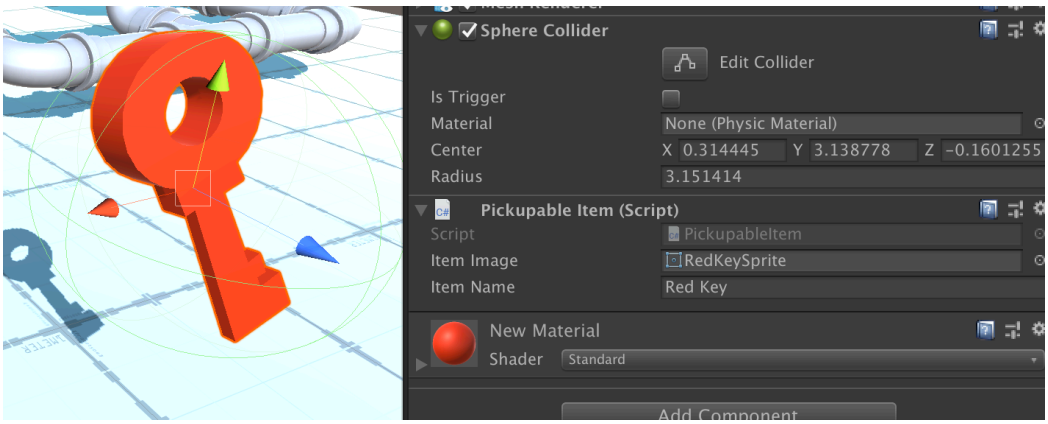
https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/properties

Any object that implements this interface will be able to be stored in the inventory.

- Create a game object to act as a key, and a script that implements the IInventoryItem interface to attach to it. The key could be a simple mesh primitive (cube, sphere etc.), but in the screenshot a mesh from the asset store has been used.

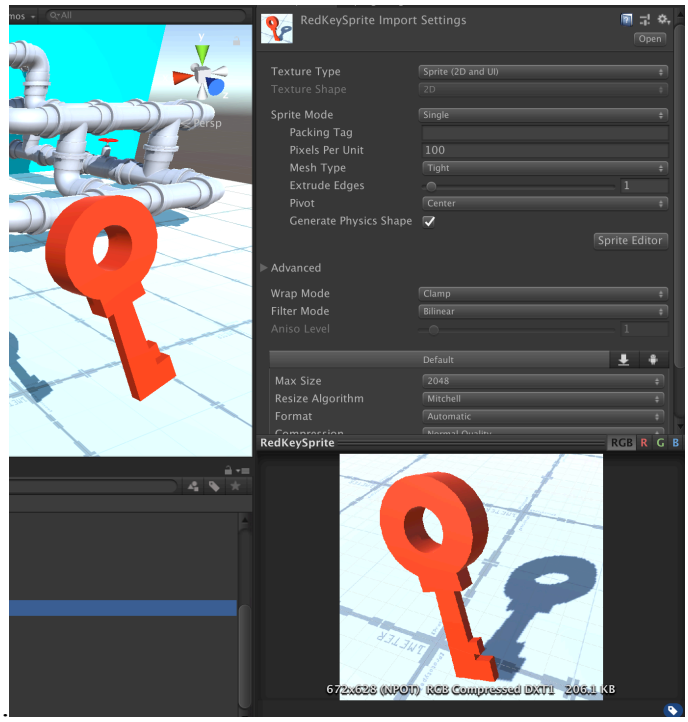
- Add an appropriate *Collider* to the key object. When the player character intersects with the collider the object will be automatically picked up.

https://assetstore.unity.com/packages/3d/handpainted-keys-42044




```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class PickupableItem : MonoBehaviour, IInventoryItem
{
    public Sprite _itemImage;
    public string _itemName;

    public string itemName
    {
        get
        {
            return _itemName;
        }
    }

    public Sprite itemImage
    {
        get
        {
            return _itemImage;
        }
    }

    public void onPickup()
    {
        gameObject.SetActive(false);
    }
}
```

- Create a *Sprite* resource so the key can be displayed in the inventory. This can be any appropriate sprite, however the simplest way to create this is to take a screenshot of the key object in the scene, and then import this as a sprite. This sprite should then be assigned to the itemImage property of the key object.



- Create an empty game object to serve as the inventory, and add a new script to it, also called *Inventory*. When an item is added to the inventory it should be added to the list of items in the inventory, the items *onPickup* method is called, which simply sets the item as inactive – removing it from being visible and interacting with the scene.

```
using System.Collections.Generic;

...

    List<IInventoryItem> items = new List<IInventoryItem>();

    public void addItem(IInventoryItem item)
    {
        items.Add(item);
        item.onPickup();

        // broadcast event to the hud
    }
```

Return to your Player controller. Here you can either add code to your own controller script, or create a new script to add to the existing FPSController object. Either way, when the controller's collider hits another collider (for example the collider for the key), check whether the other object has an IInventoryItem component – i.e. is there a script that implements the inventory item interface attached to the object – and if there is we should instruct the inventory to "pick up" the object. To enable this the player controller will need a reference to the inventory object, set via the inspector.

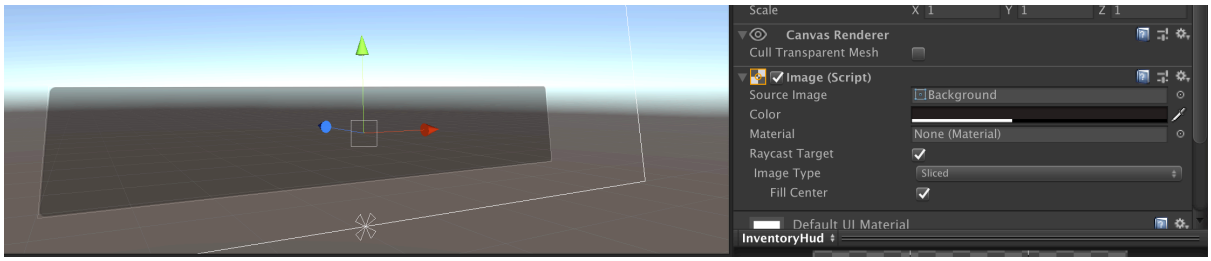
```
private void OnControllerColliderHit(ControllerColliderHit hit)
{
    IInventoryItem item = hit.gameObject.GetComponent<IInventoryItem>();

    if(item!=null)
    {
        inventory.addItem(item);
    }
}
```

Test this to ensure that when the key item is approached it correctly disappears and is added to the list held by the inventory.

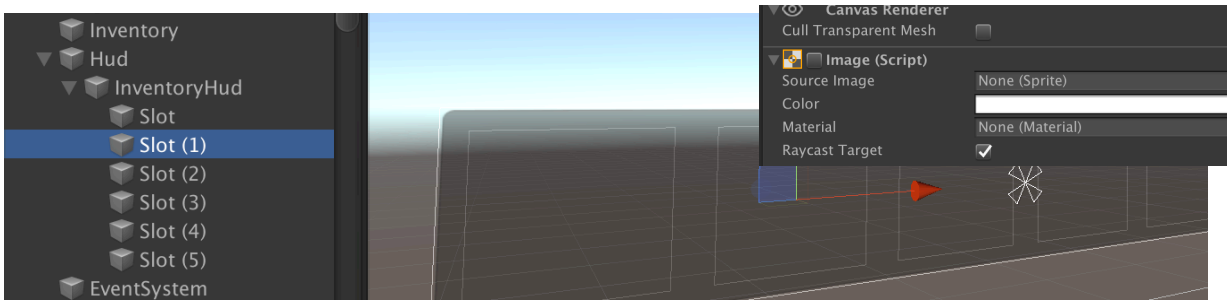
Inventory HUD

Next, create a head-up display (HUD) to display the contents of the inventory and allow the player to use the items within it.

- Create a *Canvas* object (UI->Canvas) and add a *Panel* (UI->Panel) as a child. This panel will form the inventory bar that will host the sprites of the items in the inventory. In the screenshot, it has been renamed “InventoryHud”, it has been scaled to 600x100 pixels and its background colour made darker so it is clearly visible.



Each item sprite will take up a single “slot” in the inventory bar. Each slot is implemented as an *Image* object (UI->Image) that is a child of the background panel and these are then spaced evenly across it. Note that each of these slot images should initially be *disabled* in the inspector (i.e. un-ticked) to indicate that the slot is empty and available to display an item.



The next step is to have the HUD listen for events broadcast by the Inventory to tell it to display the new sprite as the new item is added. This is enabled by introducing a *Delegate* to the Inventory. A *Delegate* is a variable that can be used to store a reference to a *method* to be called, rather than just some data (equivalent to a function pointer). In this exercise, we broadcast *Events* from one class to any other classes that have registered an interest in

these events, and the delegate is used to maintain a list of subscribing class methods that should receive the event.

https://unity3d.com/learn/tutorials/topics/scripting/delegates

https://unity3d.com/learn/tutorials/topics/scripting/events

We will use a *Delegate* to allow the HUD class to register a method with the Inventory that will be called when the new item is added. To be able to pass data via an event we must extend the base event class *EventArgs*, and send the new item along with it. Add this to the script containing your IInventoryItem interface definition.

```
using System;

...

public class InventoryEventArgs: EventArgs
{
    public InventoryEventArgs(IInventoryItem item)
    {
        this.item = item;
    }

    public IInventoryItem item;
}
```

https://docs.microsoft.com/en-us/dotnet/standard/events/

- Using this new object we can now create an *EventHandler* in the Inventory script. Other classes can subscribe to this event handler and, when it is *invoked*, will be delivered the extended *EventArgs* object including the item being added.

```
using System;

...

    public event EventHandler<InventoryEventArgs> ItemAdded;

    public void addItem(IInventoryItem item)
    {
        items.Add(item);
        item.onPickup();

        // broadcast event to hud
        if (ItemAdded != null)
        {
            ItemAdded.Invoke(this, new InventoryEventArgs(item));
        }

    }
```

- Add a script to the HUD canvas object. When this object is started, it should register with the event handler of the inventory, and then it can handle the subsequent events. Again, this will need to be assigned a reference for the inventory via the inspector.

Note how using the *Delegate* pattern the HUD can "add" a method to the EventHandler. When the event fires, the script finds the child *Panel* object, and then iterates through each of its child *Image* objects. When the next empty slot is found, we enable the image and set the image sprite data to be the sprite of the new item, retrieved from the InventoryEventArgs object.

https://docs.unity3d.com/ScriptReference/Transform.Find.html

```
void Start()
{
    inventory.ItemAdded += InventoryItemAdded;
}

private void InventoryItemAdded(object sender, InventoryEventArgs e)
{
    Transform panel = transform.Find("InventoryHud");

    foreach(Transform slot in panel)
    {
        Image image = slot.GetComponent<Image>();

        if (!image.enabled)
        {
            image.enabled = true;
            image.sprite = e.item.itemImage;

            break;
        }
    }
}
```

- Ensure that the appropriate sprite appears on the HUD inventory bar when the item is collected.

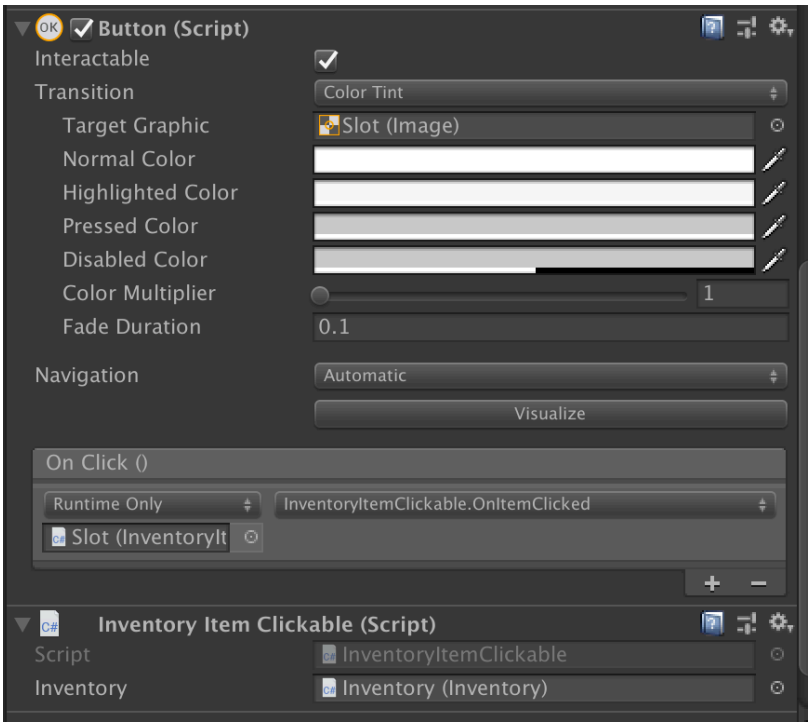
Using Items

There are several ways in which you could allow the player to make use of the items in the inventory, for example listening for key presses corresponding to each slot ("1" for slot 1, "2" for slot 2 and so on). In this case we will turn each slot into a button that can be clicked with the mouse to "use" the object.

This will work as follows:

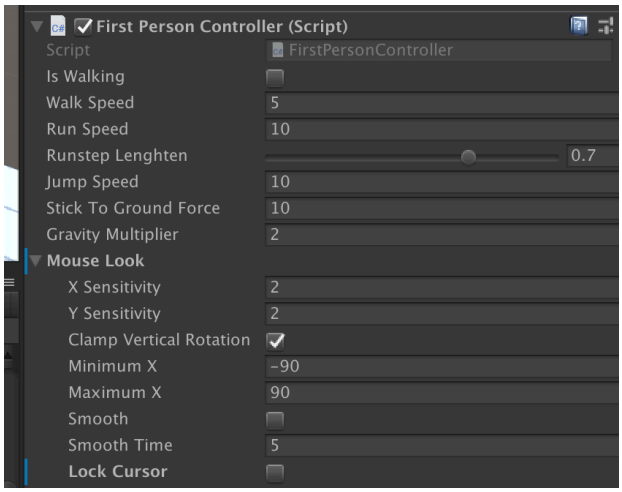
- Click on an item in the inventory
- The item subsequently calls a method in the Inventory script
- The Inventory script broadcasts an event to say that the item has been used
- Objects react the event as appropriate

- Add a *Button* component and a new script to one of the slots (you will need to add these components to all of the slots, or duplicate this slot again). The button *OnClick()* method should call into the Inventory to trigger "using" the item. To set the OnClick method you will need to select the slot object and the function to call.



If you are using the FPSController or the RigidBodyController from the Standard Assets package it is likely that when running the game the mouse cursor is captured – i.e. you can't see it. This presents an issue when clicking on the buttons in the inventory, as pressing escape to release the cursor also loses focus from the game window, and clicking again regains focus but does not correctly click the inventory button.

The solution to this is to disable "Lock Cursor" in the character controller as appropriate.




```
public class InventoryItemClickable : MonoBehaviour
{
    public IInventoryItem item;

    public Inventory inventory;

    public void OnItemClicked()
    {
        if (item != null)
```

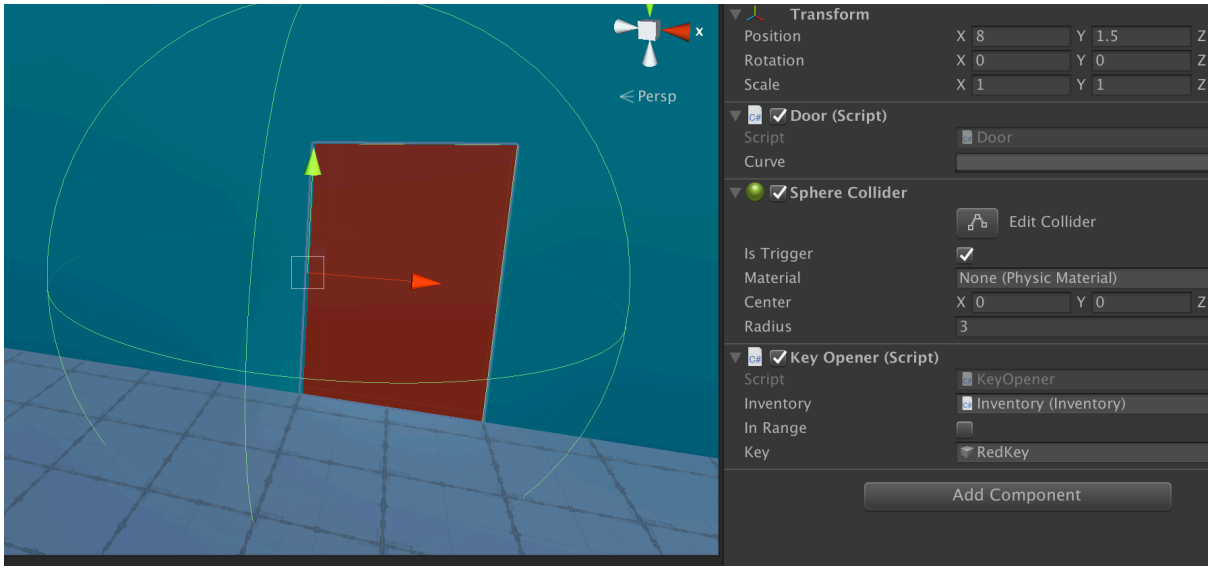
```
        {
            Debug.Log("Using: " + item.itemName);
            inventory.useItem(item);
        }
    }
}
```

As above, the inventory button will need a reference to both the inventory (set via the inspector) and the item to be used. This can be allocated when the slot is first filled in the HUD script:

```
    Image image = slot.GetComponent<Image>();
    // new
    InventoryItemClickable button = slot.GetComponent<InventoryItemClickable>();

    if (!image.enabled)
    {
        image.enabled = true;
        image.sprite = e.item.itemImage;
        button.item = e.item; // new

        break;
    }
```

Finally, create a door object to be opened by this item when it is used. You could make use of the door that opens via the weighted switch in lab 04, or start with an object that simply log debug output when it is opening, or failing to open.

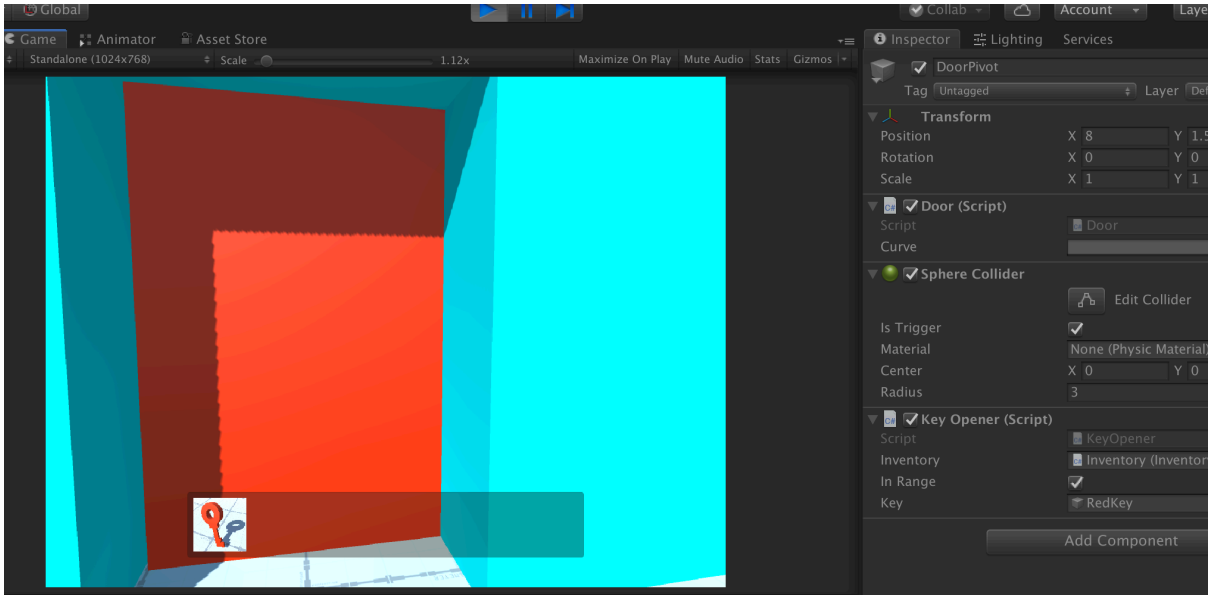
Either way, the door object should have a *Sphere Collider* trigger component that, when the player enters or exits it, flags whether they are within or outside an appropriate range (that is, how far away you require them to be to use the key). As the HUD registered with the Inventory, the door should register with the Inventory to receive use events. You will need to create an appropriate ItemUsed event handler in the inventory.



When the door receives the event and the associated IInventoryItem, the script should check specifically *which* item has been used, not just that *any* item has been used. Here the

item is cast to a gameObject and checked against a variable set via the inspector. This allows us to specify a particular item to open this particular door.

```
public bool inRange = false;

public GameObject key;

void Start()
{
    // register with the event handler
    inventory.ItemUsed += Inventory_ItemUsed;
}

void Inventory_ItemUsed(object sender, InventoryEventArgs e)
{
    // check if the correct item is in use
    if ((e.item as MonoBehaviour).gameObject == key)
    {
        // check if in range
        if (inRange)
        {
            gameObject.GetComponent<Door>().Open();
        }
    }
}

private void OnTriggerEnter(Collider other)
{
    inRange = true;
}

private void OnTriggerExit(Collider other)
{
    inRange = false;
}
```



Exercises

Obstacle Variations

There are many variations of the locked door challenge. For example, as in the screenshot, the “obstacle” could be a fire particle effect that is “unlocked” (i.e. disabled) when the

player uses the “key” – a valve – to turn off a gas supply. This could also include scripting the various objects to make the valve component appear in the right place, and to be animated rotating. The elements in the first screenshot were implemented using free assets from the asset store.

Implement your own variation of an obstacle that requires a “key” to be found to open it, but develop a key and door pairing where there is a novel premise for using the key as above.

Multi-Key Obstacle
Obstacle logic can be made more complex – as multiple items be handled by the same obstacle, we could require the player to find a number of different keys as above. This is enabled by introducing simple variables to the obstacles, where the use of each particular key toggles the variable. When all variables are *true,* the unlock event for the object is triggered.

Implement an obstacle that requires two or more “keys”.