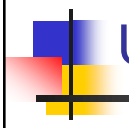


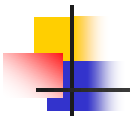
# Unity Game Engine

Introduction to Unity – Interactive Devices



## Topics

- How to create functional devices like doors
- Collecting items, which involves both interacting with objects in the level and tracking game (player's) state



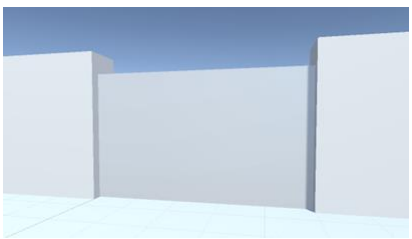
# Create Interactive Devices

- Case 1 : doors that open and close on a key press
- Case 2 : color changing monitor on a key press
- Case 3 : bumping into obstacles. (response to collisions)
  - 3.1 Push away and fall over (based on Physics)
  - 3.2 Trigger a device in the level
  - 3.3 Disappear on contact (for item pickups)



# Case 1 - Door

- Let's create a door that opens and closes by pressing a key.
- Add a door to the static walls.
  - e.g. Create a door where there is a gap between two walls using a cube object.
- Doors will open and close on a key press in our design.



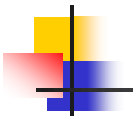
# DoorOpenDevice.cs

- Create and assign a script (**`DoorOpenDevice.cs`**) to the door object.



# DoorOpenDevice.cs

```
using UnityEngine;
using System.Collections;
public class DoorOpenDevice : MonoBehaviour {
  // The position to offset to when the door opens
  [SerializeField] private Vector3 dPos;
  // A Boolean to keep track of the open state of the door
  private bool _open;
  // Open or close the door depending on the open state.
  public void Operate() {
    if (_open) {
      Vector3 pos = transform.position - dPos;
      transform.position = pos;
    } else {
      Vector3 pos = transform.position + dPos;
      transform.position = pos;
    }
    _open = !_open;
  }
}
```



# DoorOpenDevice.cs Explained

- **`dPos`** variable defines the position to offset to when the door opens.
- **`_open`** variable keeps track of the open state of the door.
- **`Operate()`** method sets the object's transform to a new position by adding or subtracting the offset depending on whether the door is already open. Then **`_open`** is toggled on or off.



# Activate the Door

- Have the player activate the door.
- Create **`DeviceOperator.cs`** and attach it to the player.
  - The script will activate to open/close the door if the player is facing the door nearby and press the left "**`Cmd`**" (cloverleaf or Windows) key.

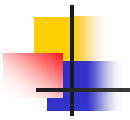
# DeviceOperator.cs

```
using UnityEngine;
using System.Collections;

public class DeviceOperator : MonoBehaviour {
  // How far away from the player to activate devices
  public float radius = 1.5f;
  void Update() {
    // Respond to the input button defined in Unity's input settings.
    if (Input.GetButtonDown("Fire3")) {
      Collider[] hitColliders =
        Physics.OverlapSphere(transform.position, radius);
```



# DeviceOperator.cs

```
      foreach (Collider hitCollider in hitColliders) {
        // Only send the  message when facing the right direction
        Vector3 direction =
           hitCollider.transform.position -
          transform.position;
        if (Vector3.Dot(transform.forward, direction)>.5f) {
            hitCollider.SendMessage("Operate",
              SendMessageOptions.DontRequireReceiver);
        } //if
      } // foreach

    } // if
  } // update
} //class
```



# DeviceOperator.cs Explained

- `Update()` will respond to `Fire3` (which is defined in the project's input settings as the left Command key).
- The code checks the distance and facing before opening the door.
  - `OverlapSphere()` returns an array of all objects that are within a given distance of a given position.
  - `if (Vector3.Dot(transform.forward, direction) > .5f)` checks if the player is facing the object.



# DeviceOperator.cs Explained (2)

- Send message to the object to invoke `Operate()`.
  - Most of the objects returned by `OverlapSphere()` won't have an `Operate()` method.
  - Normally `SendMessage()` prints an error message if nothing in the object received the message. In this case, by passing `DontRequireReceiver` to the method, make most objects ignore the message.

# GameObject.SendMessage()

public void **SendMessage**(string **methodName**,
object **value** = null,
SendMessageOptions **options** =
SendMessageOptions.RequireReceiver);

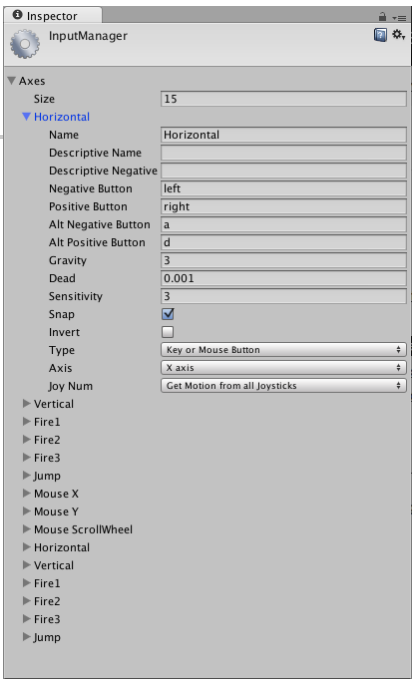
This calls the method named `methodName` on every MonoBehaviour in this game object.

- `methodName` – The name of the method to call.
- `value` - An optional parameter value to pass to the called method.
- `options` – Should an error be raised if the method doesn't exist on the target object?
  - `SendMessageOptions.DontRequireReceiver`
  - `SendMessageOptions.RequireReceiver`



# Unity Input Manager

- Unity supports keyboard, joystick and gamepad input.
- **Edit > Project Settings > Input** for current input mappings.
- You can setup joysticks, gamepads, keyboard, and mouse, then access them all through one simple scripting interface.



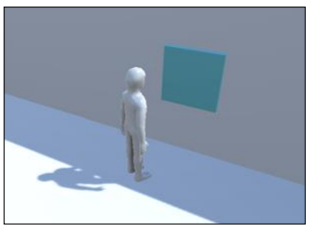
# Unity Input Manager – Virtual Axes

- From scripts, all virtual axes are accessed by their name.
- Every project has the following **default input axes** when it's created:
  - **Horizontal** is mapped to a,d and the left and right arrow keys.
  - **Vertical** is mapped to w, s and the up/down arrow keys.
  - **Fire1**, **Fire2**, **Fire3** are mapped to Control (Ctrl), Option (Alt), and Command, respectively.
  - **Mouse X** and **Mouse Y** are mapped to the delta of mouse movement.
  - **Window Shake X** and **Window Shake Y** is mapped to the movement of the window.



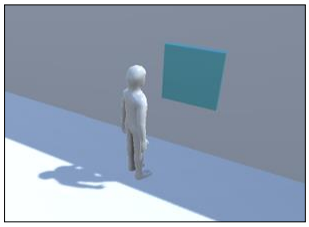
# Case 2 - Monitor

- Let's create a color changing wall display.
  - Create a new cube (wall display) and place it so that one side is barely sticking out of the wall.



# Case 2 - Monitor

- Create a new script `ColorChangeDevice.cs` and attach that script to the wall display.





# ColorChangeDevice.cs

```
using UnityEngine;
using System.Collections;
public class ColorChangeDevice : MonoBehaviour {
  // Declare a method with the same name as the door script.
  public void Operate() {
    // The numbers are RGB values that range from 0 to 1.
    Color random = new Color(Random.Range(0f,1f),
        Random.Range(0f,1f), Random.Range(0f,1f));
    //The color is set in the material attached to the object.
    GetComponent<Renderer>().material.color = random;
  }
}
```

The wall display will react to the same "operate" key as used with the door. `DeviceOperator.cs` works for both doors and the display. The code assigns a random color to the object's material.

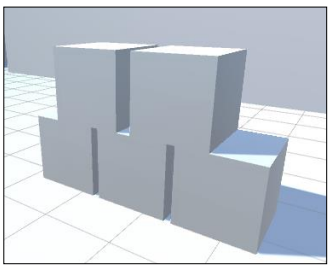
# Case 3 - Bumping into Objects

- 3.1 Push away and fall over (based on Physics)
- 3.2 Trigger a device in the level
- 3.3 Disappear on contact (for item pickups)



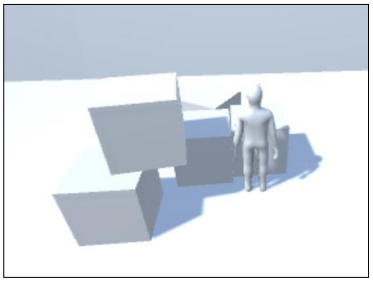
# Case 3.1 – Physics-enabled Obstacles

- Create **cube** objects and position them in a neat stack.



# Physics-enabled Obstacles

- Add a **Rigidbody** component to each cube.
- Unity's physics system will act only on objects that have a Rigidbody component.
- The boxes are now ready to react to physics forces.





# Physics-enabled Obstacles

- Make the player apply a force to move the box upon collision.
- Create/Modify **`RelativeMovement.cs`** script of the player.

## RelativeMovement.cs (Modified)

```
...

public float pushForce = 3.0f;
...
void OnControllerColliderHit(ControllerColliderHit hit) {
  _contact = hit;
  Rigidbody body = hit.collider.attachedRigidbody;
  if (body != null && !body.isKinematic) {
    body.velocity = hit.moveDirection * pushForce;
  }
}
...
```

Whenever the player collides with something, check if the collided object has a `Rigidbody` component. If so, apply a velocity to that `Rigidbody`.



## RelativeMovement.cs (explained)

- **`OnControllerColliderHit()`** is called when the controller hits a collider while performing a Move.
  - This can be used to push objects when they collide with the character.
- **`ControllerColliderHit`** is used by OnControllerColliderHit to give detailed information about the collision and how to deal with it.

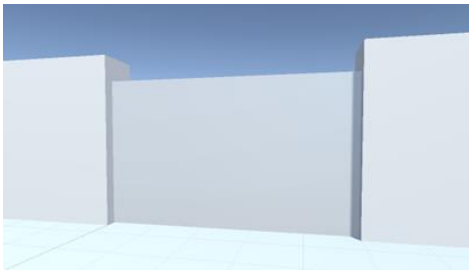
# ControllerColliderHit

| | |
|---|---|
| collider | The collider that was hit by the controller. |
| controller | The controller that hit the collider. |
| gameObject | The game object that was hit by the controller. |
| **moveDirection** | **The direction the CharacterController was moving in when the collision occured.** |
| moveLength | How far the character has travelled until it hit the collider. |
| normal | The normal of the surface we collided with in world space. |
| point | The impact point in world space. |
| rigidbody | The rigidbody that was hit by the controller. |
| transform | The transform that was hit by the controller. |



# Case 3.2 – Trigger the device

- Let's create a door that will open/close in response to the character colliding with a trigger object.
- Create a door and place it in a wall gap.



# DoorOpenDevice.cs

- Create and assign a script (**`DoorOpenDevice.cs`**) to the door object.



# DoorOpenDevice.cs

```
using UnityEngine;
using System.Collections;
public class DoorOpenDevice : MonoBehaviour {

  // The position to offset to when the door opens
  [SerializeField] private Vector3 dPos;

  // A Boolean to keep track of
  // the open state of the door
  private bool _open;
```

- **`dPos`** variable defines the position to offset to when the door opens.
- **`_open`** variable keeps track of the open state of the door.

# DoorOpenDevice.cs

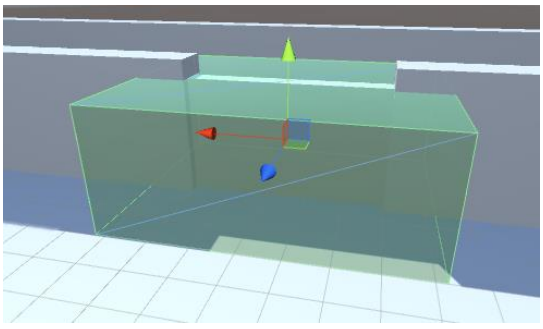
```
  public void Activate() {
    if (!_open) {
      Vector3 pos = transform.position + dPos;
      transform.position = pos;
      _open = true;
    }
  }
  public void Deactivate() {
    if (_open) {
      Vector3 pos = transform.position - dPos;
      transform.position = pos;
      _open = false;
    }
  }
...
```

Activate() and Deactivate() methods are separate functions to open and close the door.



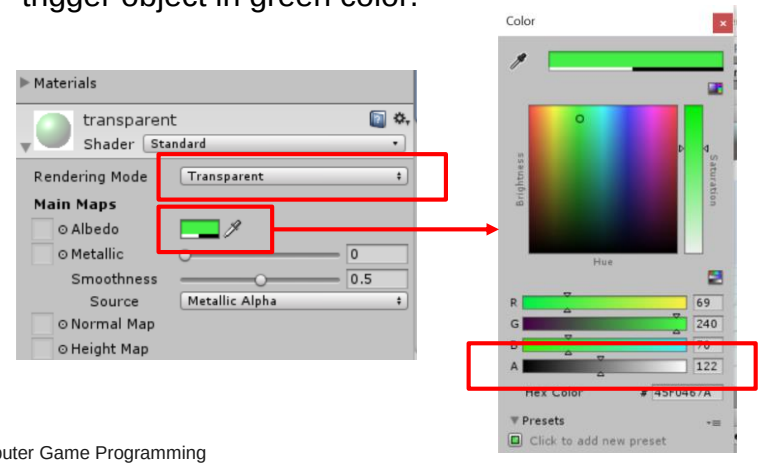
# Trigger Object

- Create a new cube to use for the trigger object.
- Position and scale the cube to encompasses the door and surrounds an area around the door.



# Trigger Object (2)

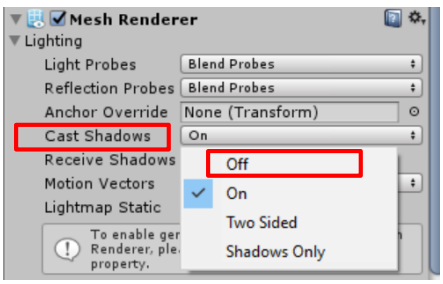
- Create and assign a **semitransparent** material to the trigger object in green color.





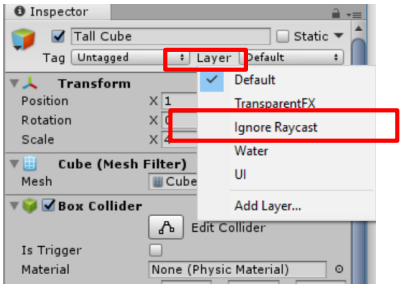

# Trigger Object (3)

- Select the "**Is Trigger**" check box for the collider.
- **Turn off shadow** casting from this object if necessary.



# Trigger Object (4)

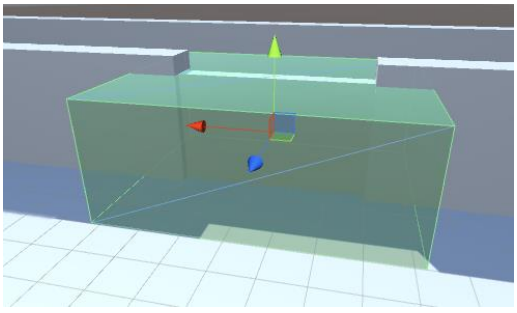
- Set the object to the "**Ignore Raycast**" Layer.
  - Physics.Raycast will ignore colliders in this layer.






# Trigger the device

- Create `DeviceTrigger.cs` script and attach it to the trigger object.
- The player's CharacterController will generate a trigger event when colliding with a trigger object.



# DeviceTrigger.cs

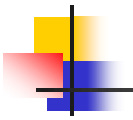
```
using UnityEngine;
using System.Collections;
public class DeviceTrigger : MonoBehaviour {
  // List of target objects that this trigger will activate
  [SerializeField] private GameObject[] targets;
  // OnTriggerEnter() is called when an object enters the trigger volume
  void OnTriggerEnter(Collider other) {
    foreach (GameObject target in targets) {
      target.SendMessage("Activate");
    }
  }
  //OnTriggerExit() is called when an object leaves the trigger volume.
  void OnTriggerExit(Collider other) {
    foreach (GameObject target in targets) {
      target.SendMessage("Deactivate");
    }
  }
}
```

These functions are called once when another object first enters and exits the trigger (as opposed to being called over and over while the object is inside the trigger volume).



# DeviceTrigger.cs Explained

- This script defines an array of target objects for the trigger. In this scenario, only one target object (the door). It's possible to have multiple devices controlled by a single trigger.
- When a Trigger happens, loop through the array of targets to send a message to all the targets.
  - The messages being sent is `Activate()` or `Deactivate()`.



# 3.3. Collect Items

- Many games include items that can be picked up by the player such as equipment, health packs, and power-ups.
- Create collectable items
  - Select "**Is Trigger**" setting in the collider.
  - Set the object to the "**Ignore Raycast**" layer.
  - Give the object a distinct material and name.



# CollectibleItem.cs

- Create a script called `CollectableItem` and attach it to the items.
- Make prefabs of the items so that you can clone them throughout the level.

# CollectibleItem.cs

```
using UnityEngine;
using System.Collections;
public class CollectibleItem : MonoBehaviour {
    // Type the name of this item in the Inspector.
   [SerializeField] private string itemName;

   void OnTriggerEnter(Collider other) {
     //Debug.Log("Item collected: " + itemName);
     //Other functions to do upon collection
     Destroy(this.gameObject);
   }
}
```



# Q & A