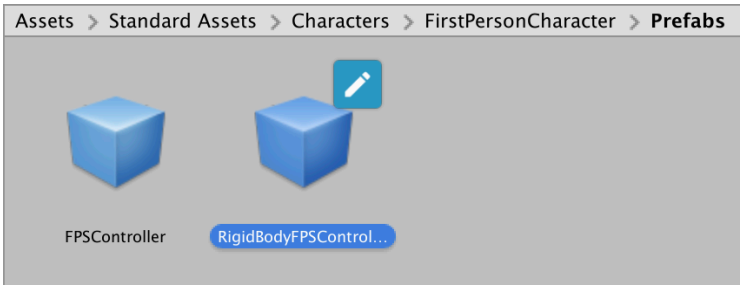
Quest 5 - Steps

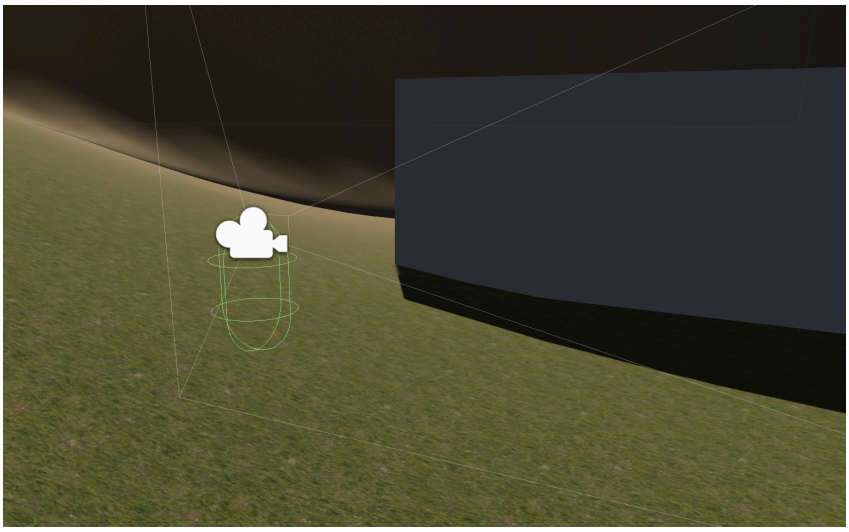
1) First Person Controller

If you would like to use the same terrain as your last quest, duplicate your Q4 scene and rename it to Q5. Make sure the edits for this quest are done in the Q5 file. Otherwise, ensure there is an active RigidBodyFPSController from the Standard Assets package in your scene for playtesting.

Delete any extra “Main Camera” objects in your scene that are not part of the RigidBodyFPSController or future steps might NOT work.



Place the character near your starting location. (You can use the CMD+Shift/CTRL+Shift key to help align the character to the landscape.) We’ll be setting up a few prototypical interactions in this immediate area. (See the requirements list.)



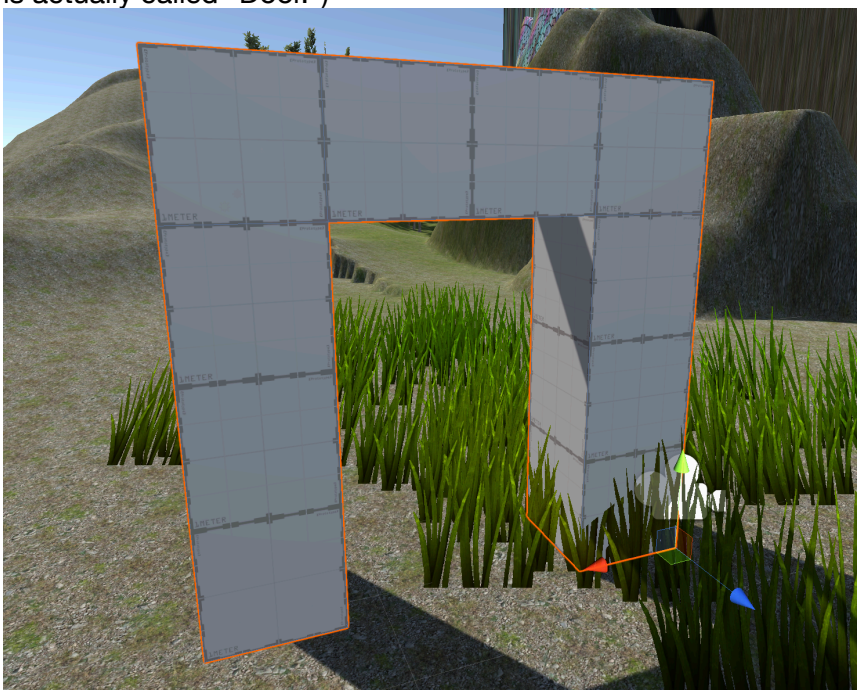
2) First Person Interactions

Create a PlayerController script and attach it to your RigidBodyFPSController. For first-person mode, we check for the E key and raycast only a short distance in front of the player to trigger interactions.

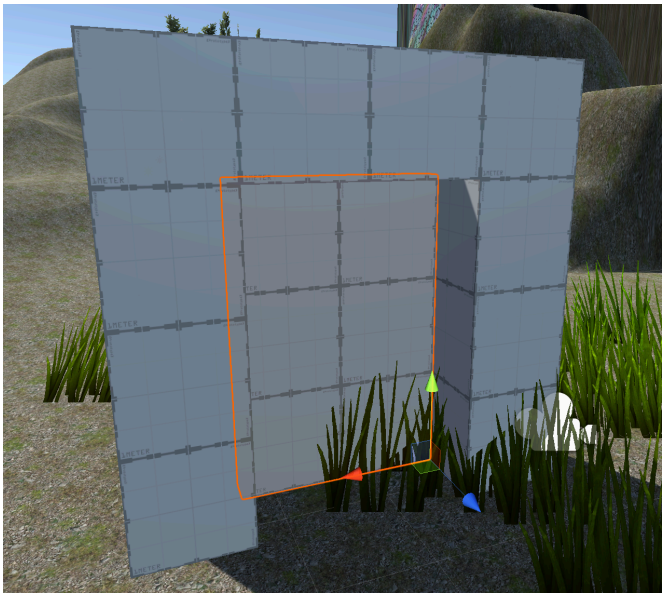
```
public class PlayerController : MonoBehaviour {
    public static PlayerController instance;

    // Methods
    void Awake() {
        instance = this;
    }

    void Update() {
        if(Input.GetKeyDown(KeyCode.E)) {
            Ray ray = Camera.main.ScreenPointToRay(Input.mousePosition);
            RaycastHit hit;
            if(Physics.Raycast(ray, out hit, 1.1f)) {
                // Handle First Person Interactions Here
                print("Interacted with " + hit.transform.name + " from " + hit.distance + "m.");
            }
        }
    }
}
```

3) Create a Door

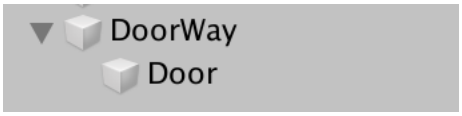
Using ProBuilder, I create an object I named “DoorWay.” (The ProBuilder preset for this object is actually called “Door.”)



Next I use the ProBuilder “Cube” preset to create the object I will actually call “Door.” Make sure the door is thin enough that you can distinguish it from the door way.



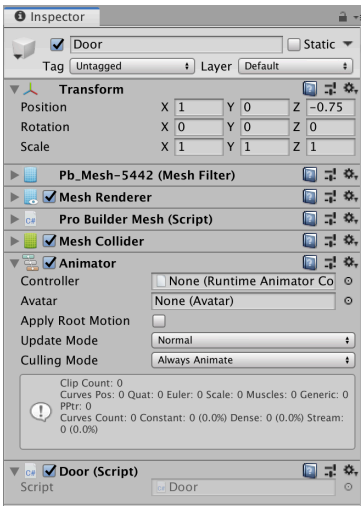
I nest “Door” within the “DoorWay” game object as a child, so that the two move together as a unit when placed on the map.



4) Door Animations

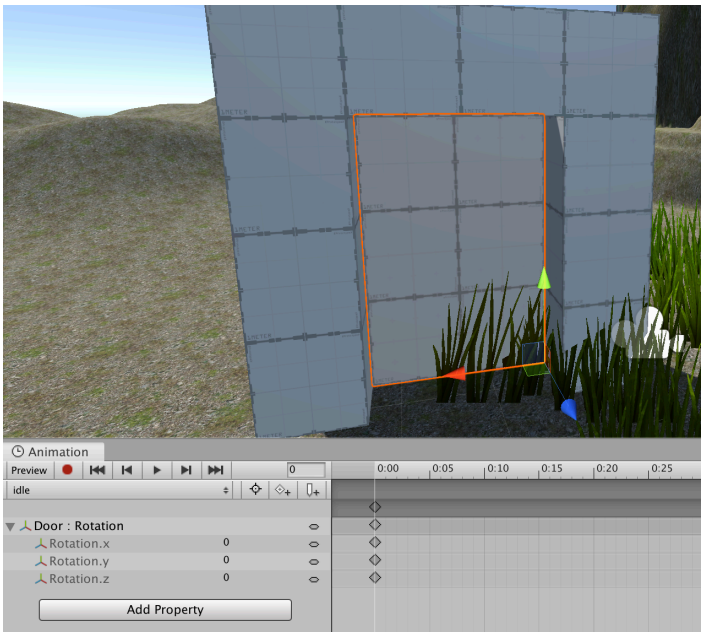
We will need three animations for our Door interactions:
- Idle
- Open
- Close

Start by adding an Animator to the Door. Do NOT add it to the DoorWay. Also create a Door.cs script and attach it to the Door. Do NOT add it to the DoorWay.

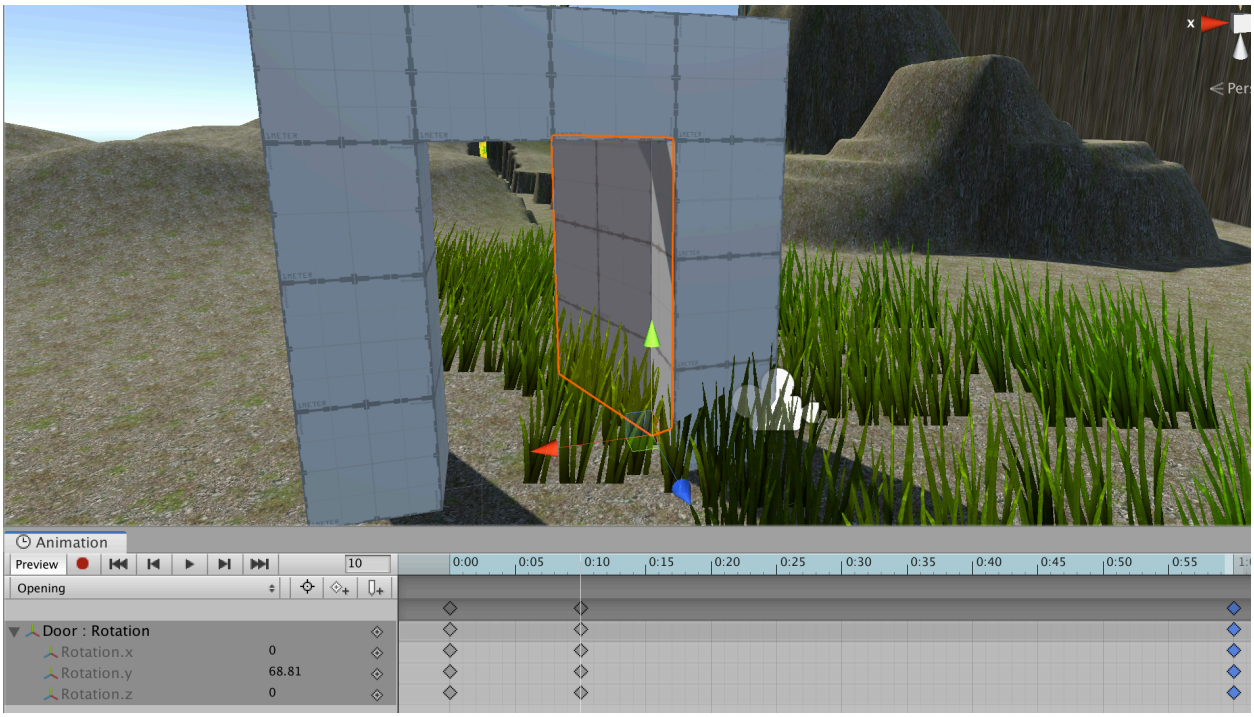


Create your three animations:

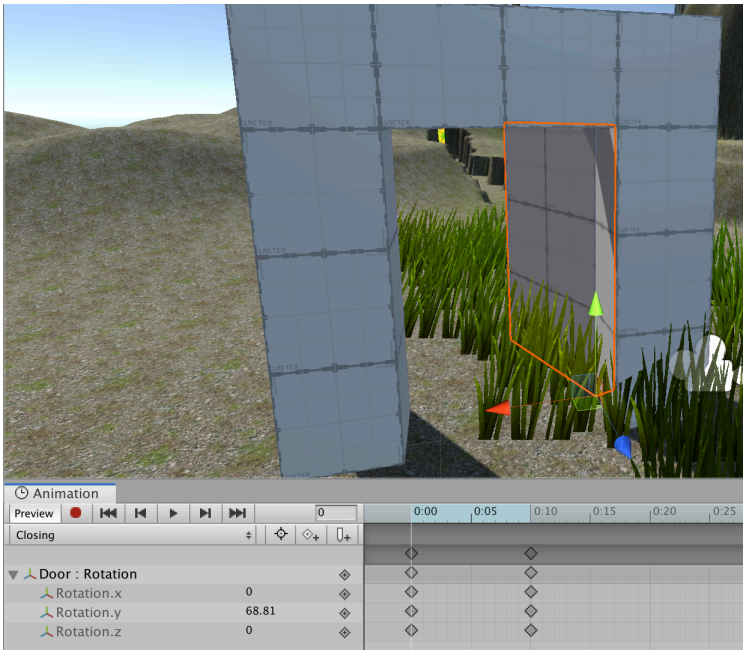
The IDLE animation is just the door sitting closed.



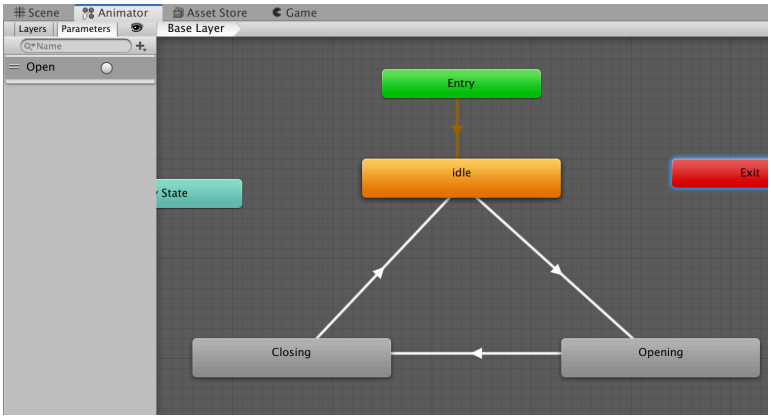
The OPENING animation is the door in the process of opening using a Y-rotation. These doors will auto-close, so make sure your animation includes time in which the door STAYS OPEN.



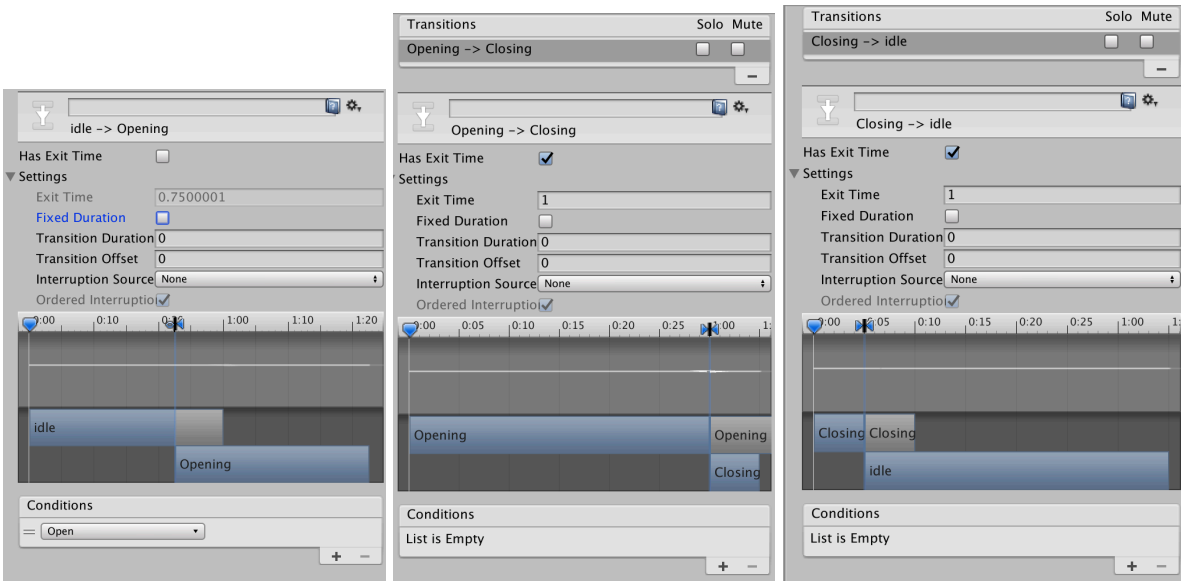
The CLOSING animation is the door in the process of returning to a zero-rotation from the same open rotation as the OPENING animation. The animation transition from OPENING to CLOSING should be seamless.



The Animator flow goes like this:
- The door starts looping in the Idle state
- When the “Open” Trigger is set, the door immediately transitions to “Opening”
- When “Opening” has completed, the door immediately transitions to “Closing”
- When “Closing” has completed, the door immediately transitions to “Idle” which loops.

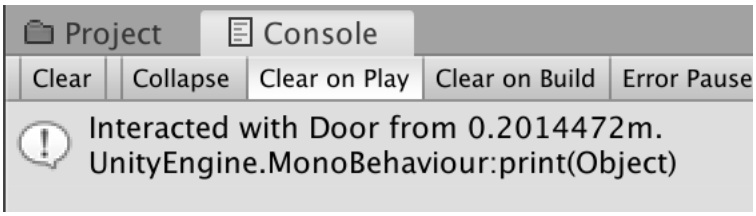


Settings for the three transitions:



5) Basic Door Interaction

If we try to interact with our door right now, we will only get a debug message in the console.



The Door script will have a public function that triggers the necessary animations to open.

```
public class Door : MonoBehaviour
{
    // Outlets
    Animator animator;

    // Methods
    void Awake() {
        animator = GetComponent<Animator>();
    }

    public void Interact() {
        animator.SetTrigger("Open");
    }
}
```

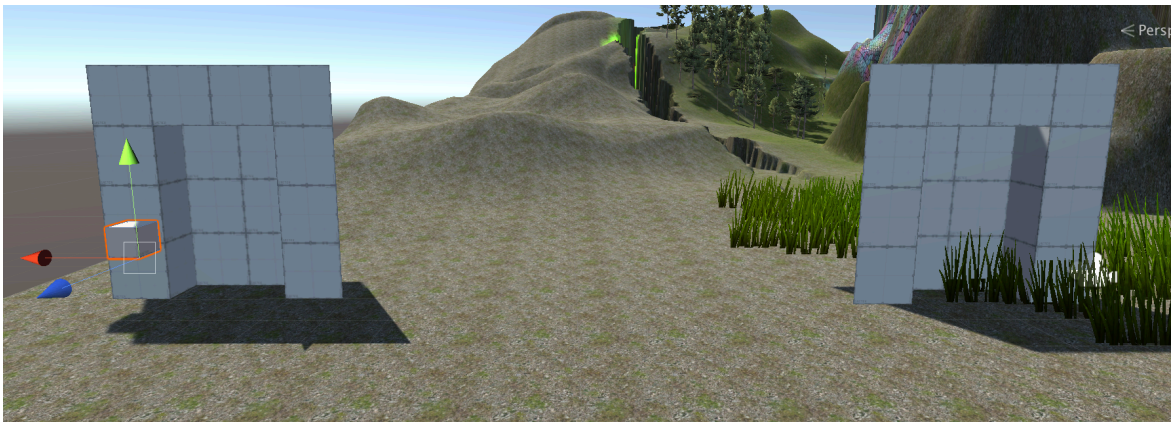
The PlayerController will check if the interacted object has a Door script, and will trigger it accordingly.

```
void Update() {
    if(Input.GetKeyDown(KeyCode.E)) {
        Ray ray = Camera.main.ScreenPointToRay(Input.mousePosition);
        RaycastHit hit;
        if(Physics.Raycast(ray, out hit, 1.1f)) {
            // Handle First Person Interactions Here
            print("Interacted with " + hit.transform.name + " from " +

            // Doors
            Door targetDoor = hit.transform.GetComponent<Door>();
            if(targetDoor != null) {
                targetDoor.Interact();
            }
        }
    }
}
```

6) Create a Button

Create a Button object and attach an InteractButton.cs script to it. (I'm just going to use a ProBulder cube.)

Duplicate your DoorWay from the previous steps and move it close to the Button.



7) Button-Door Interaction

The goal of this setup is to have a door that opens ONLY through the button, and not from the player interacting with the door. The other door should be unaffected.

First, we will revise the Door to be more intelligent in checking the source of interactions. We will add an option to configure that Doors can only be opened by a specific sender (a given player, a specific button, etc.)

```
public class Door : MonoBehaviour
{
    // Outlets
    Animator animator;

    // Configuration
    public GameObject requiredSender;

    // Methods
    void Awake() {
        animator = GetComponent<Animator>();
    }

    public void Interact(GameObject sender = null) {
        bool shouldOpen = false;

        // Is this a valid interaction?
        if(!requiredSender) {
            shouldOpen = true;
        } else if(requiredSender == sender) {
            shouldOpen = true;
        }

        if(shouldOpen) {
            animator.SetTrigger("Open");
        }
    }
}
```

Notice that by specifying a default argument value of “GameObject sender = null”, we are able to avoid breaking existing functionality for the normal door.

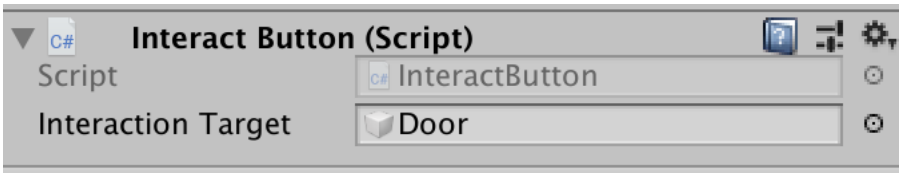
Next, we will write the InteractButton.cs to forward any interactions to its target door.

```
public class InteractButton : MonoBehaviour
{
    // Configuration
    public GameObject interactionTarget;

    // Methods
    public void Interact() {
        if(interactionTarget != null) {
            // Doors
            Door targetDoor = interactionTarget.GetComponent<Door>();
            if(targetDoor != null) {
                targetDoor.Interact(gameObject); // Pass ourselves as the sender
            }
        }
    }
}
```

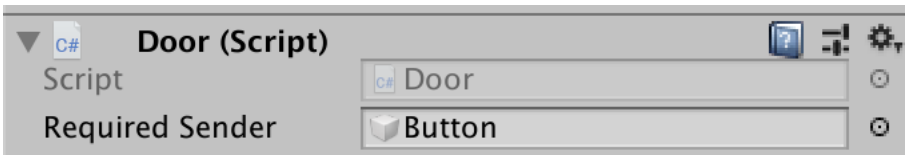
Finally, we will expand the available PlayerController interactions to include the new InteractButton.

```
// Handle First Person Interactions Here
print("Interacted with " + hit.transform.name + " from " + hit.distance + "m.");

// Doors
Door targetDoor = hit.transform.GetComponent<Door>();
if(targetDoor != null) {
    targetDoor.Interact();
}

// Buttons
InteractButton targetButton = hit.transform.GetComponent<InteractButton>();
if(targetButton != null) {
    targetButton.Interact();
}
```

You must specify the Door as the target of the Button.



You must specify the Button as a valid sender for the Door.

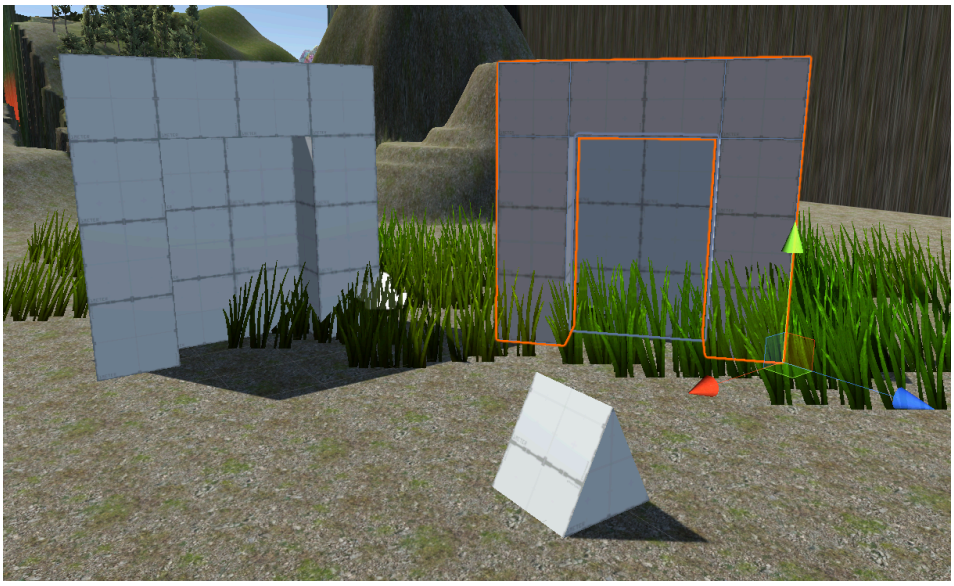


Playtest and ensure that:
- The first door still works through direct interaction
- The second door does NOT work through direct interaction
- The second door works through button interaction

8) Create a Key

Create a Key object and attach an KeyItem.cs script to it. (I'm just going to use a ProBulder prism.)

Duplicate your DoorWay from the previous steps and move it close to the Button.



9) Key Interactions

When we collect a key, it will add its ID to the players list of collected keys. This allows us to specify that certain key items are relevant to certain interactions.

```
lic class PlayerController : MonoBehaviour {
 public static PlayerController instance;

 // State Tracking
 public List<int> keyIdsObtained;

 // Methods
 void Awake() {
     instance = this;
     keyIdsObtained = new List<int>();
 }
```

The key's ID is added to the player's collection on collision.

```
public class KeyItem : MonoBehaviour
{
    // Configuration
    public int id;

    // Methods
    void OnCollisionEnter(Collision other) {
        PlayerController targetPlayer = other.gameObject.GetComponent<PlayerController>();
        if(targetPlayer != null) {
            targetPlayer.keyIdsObtained.Add(id);
            Destroy(gameObject);
        }
    }
}
```

10) Key-Door Interactions

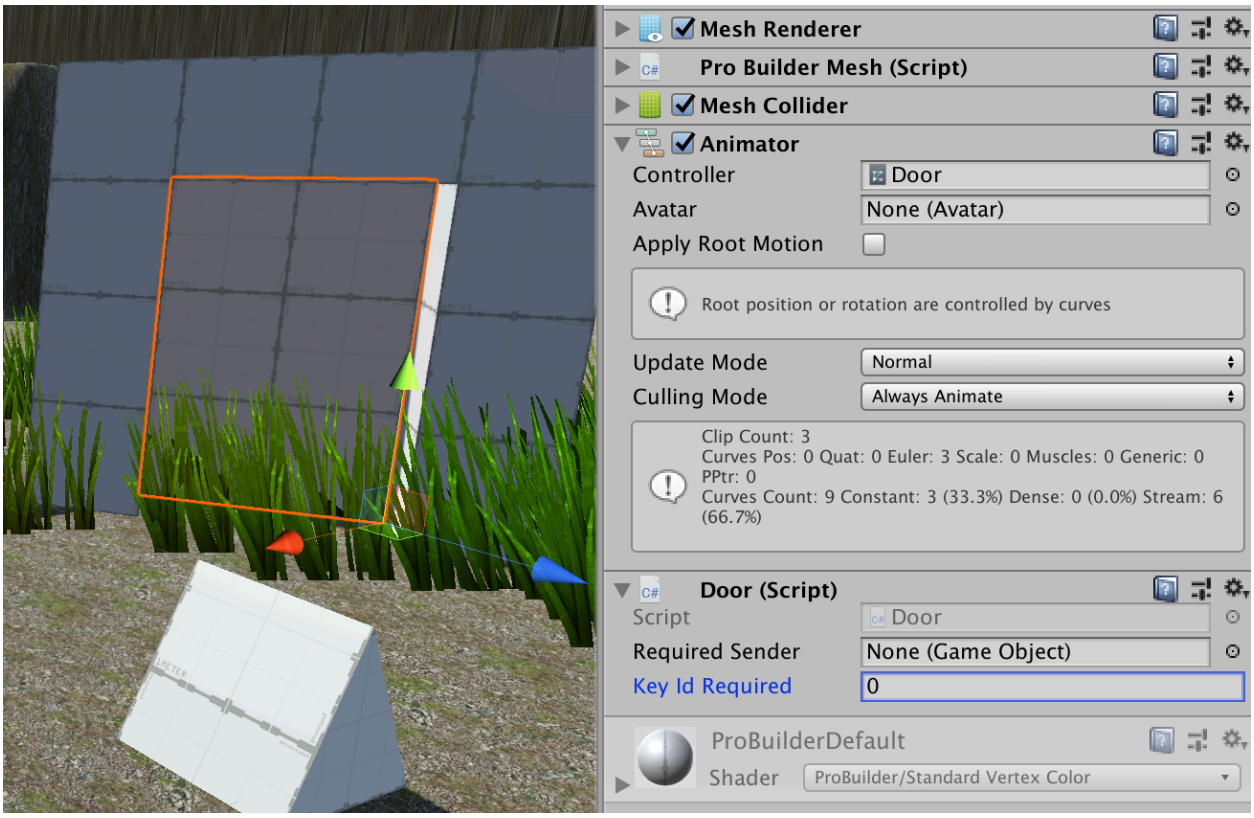
We will add another configuration value to the Door, so that it checks not only for senders, but also for collected keys.

```
lic class Door : MonoBehaviour

// Outlets
Animator animator;

// Configuration
public GameObject requiredSender;
public int keyIdRequired = -1; // Default -1 means none required
```

The key check happens after all the other requirement checks, as the requirement for a sender can overlap with the requirement for a key.

```
// Check required keys if other requirements met
if(keyIdRequired >= 0 && !PlayerController.instance.keyIdsObtained.Contains(keyIdRequired)) {
    shouldOpen = false;
}

if(shouldOpen) {
    animator.SetTrigger("Open");
}
```

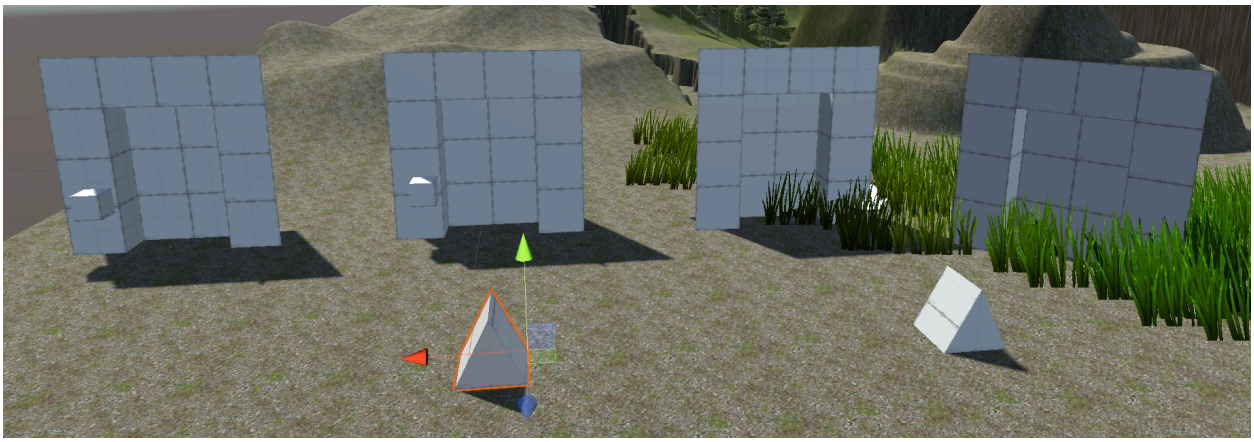
An ID must be assigned to the door. A value of -1 means it won't check for a key.



Playtest. Ensure that all the previous doors still work as expected. This new door should only open if the player has collected the correct key.

11) Create a Second Key

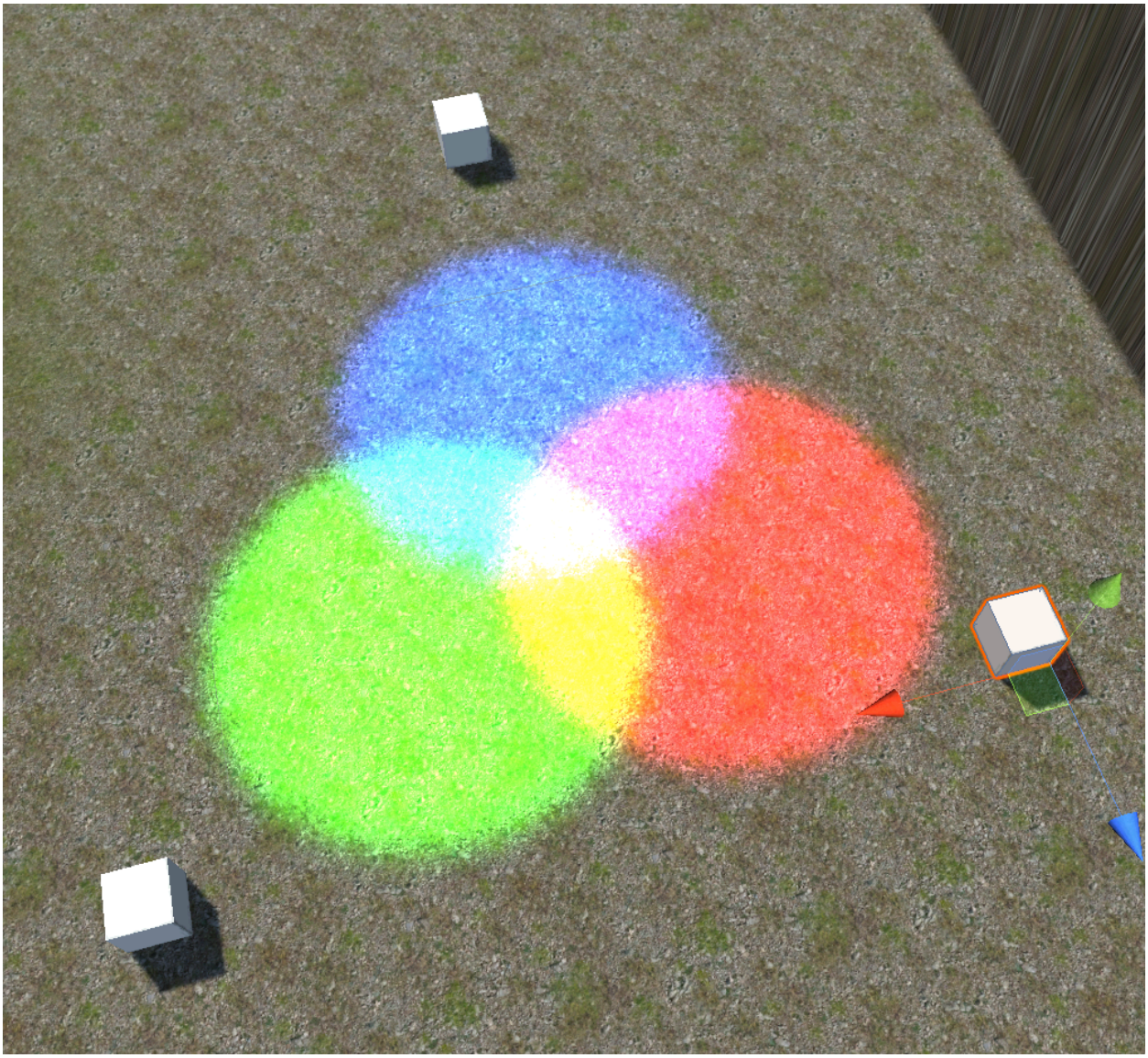
Repeat and combine prior steps to create a NEW Door that opens if the player collects a DIFFERENT key and responds ONLY to a new door BUTTON. The first key should not be valid on this new door.



12) Create Three Lights and Three Buttons

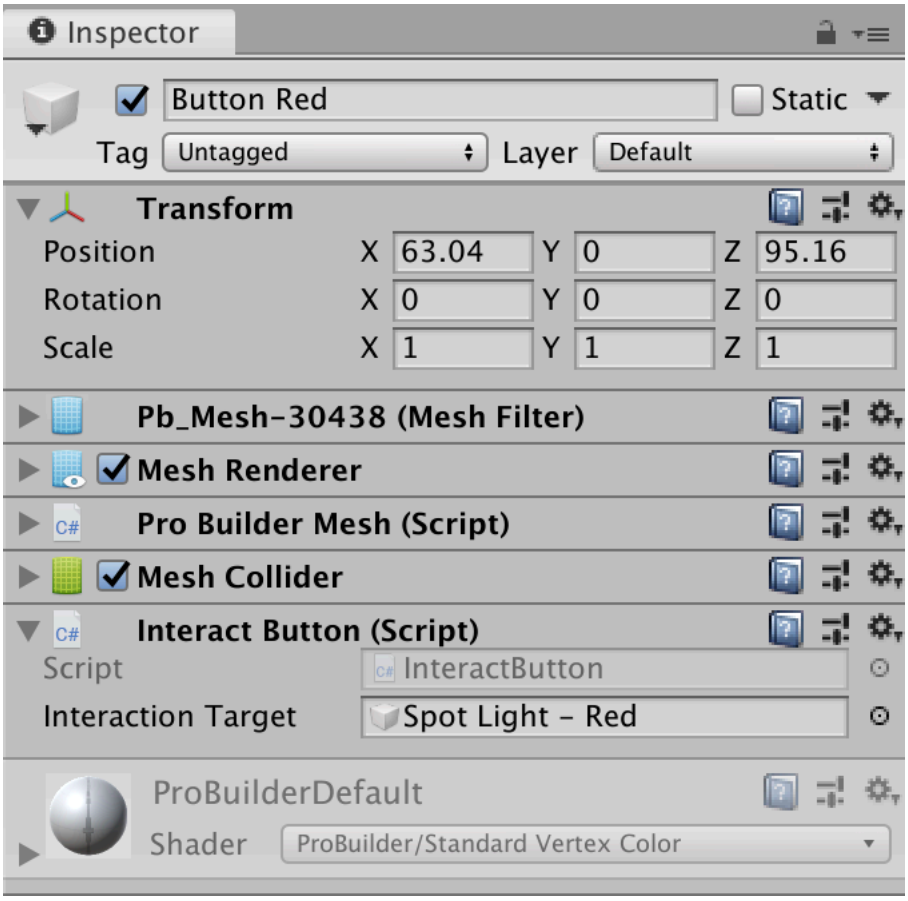
Create three overlapping spotlights using 3 different colors. To each spotlight, attach an InteractLight.cs script.

Create three buttons, one for each light.



13) Button-Light Interactions

For each button, specify its Interaction Target as the light that is closest to it.



We will program an interaction event for our lights that will toggle their current active state in the scene.

```
public class InteractLight : MonoBehaviour
{
    public void Interact() {
        // Flip/toggle our current active state
        gameObject.SetActive(!gameObject.activeInHierarchy);
    }
}
```

Finally, revise the InteractButton functionality to cover InteractLights in addition to Doors.

```
// Doors
Door targetDoor = interactionTarget.GetComponent<Door>();
if(targetDoor != null) {
    targetDoor.Interact(gameObject); // Pass ourselves as the sender
}

// Lights
InteractLight targetLight = interactionTarget.GetComponent<InteractLight>();
if(targetLight != null) {
    targetLight.Interact();
}
```

14) Playtest

Playtest to ensure all interactions work as expected and that the addition of any new features hasn't broken any earlier interactions.

Interactable Content

- A door you can open directly
- A door that is opened using a nearby button
- A door that opens if a specific key has been collected
- A door that is opened using a nearby button if a different key has been collected
- Three light switches that control 3 overlapping lights of different colors