# Unit 06: States



## Introduction

In this next series of units, we'll take a look at **states** — the way a game keeps track of what everything is doing in the game. We'll also be taking a look at how to make things move using more sophisticated techniques. But to start with, we'll just make a door.

## Goal

The goal of this unit is to make a door that opens on contact from the player, and closes after a delay.

## Process

### Creating the door

1. Create an empty game object in your Hierarchy, and call it `Door`. Add as a child a cube. Position the `Door` object (*not* the Cube) somewhere useful in your scene.
2. Set the Transform of the Cube to be:

| | x | y | z |
|---|---|---|---|
| Position | 0 | 0.5 | 0 |
| Rotation | 0 | 0 | 0 |
| Scale | 2 | 1 | 0.5 |

1. Add a `Rigidbody` component to the `Door`. Set it to not use gravity, and to be Kinematic.

> Feel free to add materials, etc.

## Scripting the door, first pass

> This will be our first pass at making a door — aiming for the "Minimum Viable Product (MVP)", the least we can do to get the functionality of a door. Later, we will refactor the code to be better looking and more functional.

1. Select the `Door`, and add a new script to it called `Door`. Open the script in your editor.
2. Create two new class variables:

```
public class Door : MonoBehaviour
{
    public Vector3 closePosition;
    public Vector3 openPosition;
```

1. We'll set the `closePosition` variable in our `Start` method, just like we did for the Respawn functionality:

```
    void Start()
    {
        closePosition = transform.position;
    }
```

And we need to set the `openPosition` in the editor – set it to the same values as it's current position, but subtract 2 from the `x` value.

1. Next, let's make a method that opens our door:

```
    void OpenDoor()
    {
        Debug.Log("OpenDoor");
    }
```

1. And we'll call this method when something collides with the door:

```
    private void OnCollisionEnter(Collision other)
    {
        OpenDoor();
    }
```

> When you test it, you should get the Debug message when the player collides with the door.
>
> Now, let's make the door move. To make it move, we're going to use something called a **lerp**, which is shorthand for *linear interpolation*. It's a way to have a value – in this case, the location – change over time. Lerps get used a lot in games.
> In Unity, we have access to the `Vector3.Lerp` method, and we'll use that. At the minimum, it takes three parameters: the starting value, the ending value, and `t`, a number from 0 to 1 which is the

> percentage along the line to set the current value.
> So if we were lerping from A to B:
> `A---------B`
> If we set `t` = 0, the lerp would be at A:
> `A*--------B`
> And if we set `t` = 1, the lerp would be at B:
> `A--------*B`
> If `t` = 0.5, the lerp would be halfway:
> `A----*----B`

1. In your `OpenDoor` method, add the following line:

```
    void OpenDoor()
    {
        Debug.Log("OpenDoor");
        transform.position = Vector3.Lerp(closePosition, openPosition, 1.0f);
    }
```

> Now when you test, the door jumps from the closed to the open positions instantly.

## Scripting the door, second pass

> That's functional, but again – it's the least possible functionality for a door. For the next steps, we're going to make the door smoothly open. To do this, we're going to call the lerp over and over, moving the door in slight increments.
> We're going to use Coroutines, much like what we did with the projectiles.

1. Change your `OpenDoor` method to return an `IEnumerator` instead of `void`. This makes it a coroutine:

```
    IEnumerator OpenDoor()
    {
```

1. And in your `OnCollisionEnter` method, instead of calling `OpenDoor` we now need to start a coroutine:

```
    private void OnCollisionEnter(Collision other)
    {
        StartCoroutine("OpenDoor");
    }
```

> These two steps are the basic way to create a coroutine from existing code. Very useful!

1. Now we need to create some code to make the door move. The first step is to make a new class variable to store a float about how long we want the animation to last:

```
public class Door : MonoBehaviour
```

```
{
    public Vector3 closePosition;
    public Vector3 openPosition;

    public float duration = 1.2f;
```

1. We can then use that duration in our animation:

```
    IEnumerator OpenDoor()
    {
        Debug.Log("OpenDoor");

        // this is the method variable we use to track the progress of the lerp
        float timeElapsed = 0f;

        while (timeElapsed < duration)
        {
            transform.position = Vector3.Lerp(closePosition, openPosition, timeElapsed / durat
ion);
            timeElapsed += Time.deltaTime;

            // This tells the coroutine to run the while loop again
            yield return null;
        }

        // once we're done with the loop, we force the final position just in case
        transform.position = openPosition;
    }
```

> We're going to do some somewhat tricky stuff here, but don't stress about it – what we're doing is setting up a `while` loop to loop until we reach the duration time. Inside the loop, we move the door a little bit, and increase the time.
>
> Lerps notoriously don't always reach the very end – they'll get to 2.999986 instead of 3, for example. So after we complete the duration loop, we just force the final value.

## Scripting the door, third pass

> When you test the door now, you should be able to open the door, and it was take as long as the duration you set in the editor. But when you collide with the door again, it'll start over! That's not right. We need to start tracking **state** – the fact that the door is open, closed, or even if it is currently opening or closing.
>
> To do this, we're going to introduce the programming type called an **enum**, which is short for "enumerator". It's a simple type that allows us to list words that we can use to name our states. With enums, you have to declare/define them before you can use them.

1. At the top of you class declaration, add the following:

```
public class Door : MonoBehaviour
{
    public Vector3 closePosition;
    public Vector3 openPosition;
```

```
    public float duration = 1.2f;

    public enum DoorState {
        Closed, Open, IsOpening, IsClosing
    };
    public DoorState doorState = DoorState.Closed;
```

> Note how we create the definition first, then create a new variable to hold the actual value. Also, check it out in the editor – we now have a cool drop-down menu that we can use to change the state in the editor (if we ever need to).

1. Now we can use those states in our method:

```
    IEnumerator OpenDoor()
    {
        // Set the state
        doorState = DoorState.IsOpening;

        // this is the method variable we use to track the progress of the lerp
        float timeElapsed = 0f;

        while (timeElapsed < duration)
        {
            transform.position = Vector3.Lerp(closePosition, openPosition, timeElapsed / durat
ion);
            timeElapsed += Time.deltaTime;

            // This tells the coroutine to run the while loop again
            yield return null;
        }

        // once we're done with the loop, we force the final position just in case
        transform.position = openPosition;
        // Set the state
        doorState = DoorState.Open;
    }
```

> Now when you test, you can see the state change in the editor. Now we can use this state to stop any other movement!

1. Change your `OnCollisionEnter` code:

```
    private void OnCollisionEnter(Collision other)
    {
        if (doorState == DoorState.Closed)
        {
            StartCoroutine("OpenDoor");
        }
    }
```

> The only time we want the door to open is when it's closed.

## Scripting the door, fourth pass

> We're getting there! Now let's make the door close when it's open and we run into it.

1. Create a `CloseDoor` method. It'll be much the same as the `OpenDoor` script:

```
    IEnumerator CloseDoor()
    {
        // Set the state
        doorState = DoorState.IsClosing;

        // this is the method variable we use to track the progress of the lerp
        float timeElapsed = 0f;

        while (timeElapsed < duration)
        {
            transform.position = Vector3.Lerp(openPosition, closePosition, timeElapsed / durat
ion);
            timeElapsed += Time.deltaTime;

            // This tells the coroutine to run the while loop again
            yield return null;
        }

        // once we're done with the loop, we force the final position just in case
        transform.position = closePosition;
        // Set the state
        doorState = DoorState.Closed;
    }
```

> Any time you copy-paste a chunk of code like this, it's a big flag that you can refactor the code later.

1. And then we can change the `OnCollisionEnter` code:

```
    private void OnCollisionEnter(Collision other)
    {
        if (doorState == DoorState.Closed)
        {
            StartCoroutine("OpenDoor");
        }

        if (doorState == DoorState.Open)
        {
            StartCoroutine("CloseDoor");
        }
    }
```

> Test it out, and make sure you can close the door once you've opened it.
>
> Speaking of refactoring, we're going to do a quick refactor right now. When you use enums, you can also use another structure called a **switch**. Switches allow you to collapse a big list of `if` statements together. If you imagine a more complicated state machine than a door, you'll end up with a lot of `if`s, which can get unweildy. Let's use a switch right now.

1. Replace your `OnCollisionEnter` code with this:

```
    private void OnCollisionEnter(Collision other)
    {
        switch (doorState)
```

```
        {
            case DoorState.Closed:
                StartCoroutine("OpenDoor");
                break;
            case DoorState.Open:
                StartCoroutine("CloseDoor");
                break;
        }
    }
```

> Note how at the end of every case, we use `break;`. This is because with switch statements, you can allow it to continue to check states that follow afterwards; you won't do this often though. So remember your `break;` statements.

## Closing after a time delay

> Alright, we've got a basic state machine set up for the door. Now let's set a timer to close the door. We'll be using more coroutines, in a structure very similar to the projectile delay.

1. Firstly, let's make a new class variable:

```
public class Door : MonoBehaviour
{
    public Vector3 closePosition;
    public Vector3 openPosition;

    public float duration = 1.2f;
    public float closeDelay = 0.8f;
```

1. Next, let's make a new coroutine to handle the waiting:

```
    IEnumerator CloseDoorAfterDelay()
    {
        yield return new WaitForSeconds(closeDelay);
        StartCoroutine("CloseDoor");
    }
```

> Note how all we're doing is waiting for the delay, then starting a new `CloseDoor` coroutine. Simple!

1. Lastly, we can start the coroutine at the end of the `DoorOpen` method:

```
        // once we're done with the loop, we force the final position just in case
        transform.position = openPosition;
        // Set the state
        doorState = DoorState.Open;

        // Close the door after a delay
        StartCoroutine("CloseDoorAfterDelay");
```

> Test it, and make sure it's all working.

## Complete Code

```
public class Door : MonoBehaviour
{
    // Positions
    public Vector3 closePosition;
    public Vector3 openPosition;

    // Timing
    public float duration = 1.2f;
    public float closeDelay = 0.8f;

    // The state machine
    public enum DoorState {
        Closed, Open, IsOpening, IsClosing
    };
    public DoorState doorState = DoorState.Closed;

    // Start is called before the first frame update
    void Start()
    {
        closePosition = transform.position;
    }

    private void OnCollisionEnter(Collision other)
    {
        switch (doorState)
        {
            case DoorState.Closed:
                StartCoroutine("OpenDoor");
                break;
            case DoorState.Open:
                StartCoroutine("CloseDoor");
                break;
        }
    }

    IEnumerator OpenDoor()
    {
        // Set the state
        doorState = DoorState.IsOpening;

        // this is the method variable we use to track the progress of the lerp
        float timeElapsed = 0f;

        while (timeElapsed < duration)
        {
            transform.position = Vector3.Lerp(closePosition, openPosition, timeElapsed / durat
ion);
            timeElapsed += Time.deltaTime;

            // This tells the coroutine to run the while loop again
            yield return null;
        }

        // once we're done with the loop, we force the final position just in case
        transform.position = openPosition;
        // Set the state
        doorState = DoorState.Open;

        // Close the door after a delay
        StartCoroutine("CloseDoorAfterDelay");
```

```
    }

    IEnumerator CloseDoor()
    {
        // Set the state
        doorState = DoorState.IsClosing;

        // this is the method variable we use to track the progress of the lerp
        float timeElapsed = 0f;

        while (timeElapsed < duration)
        {
            transform.position = Vector3.Lerp(openPosition, closePosition, timeElapsed / durat
ion);
            timeElapsed += Time.deltaTime;

            // This tells the coroutine to run the while loop again
            yield return null;
        }

        // once we're done with the loop, we force the final position just in case
        transform.position = closePosition;
        // Set the state
        doorState = DoorState.Closed;
    }

    IEnumerator CloseDoorAfterDelay()
    {
        yield return new WaitForSeconds(closeDelay);
        StartCoroutine("CloseDoor");
    }
}
```

## Wrap-Up

In this unit we created a very simple **state machine** to keep track of the state of a door. Using this state machine, we can open and close the door, and make sure it won't open or close at the wrong time. We also added a timer to close the door after a delay, using a pattern we've used earlier.

Hopefully you can see how you might use state machines elsewhere in your game!

> You might also want to check out other ways of moving things, including SmoothDamp.

## Further Material

- Nice intro to Coroutines
- Unity manual on Lerps