Unit 10: Game State

Introduction

Up until now, our variables and information have been object-specific, e.g. a health value for a given entity. In this unit, we'll be looking at game-wide states and information — which will allow for things like scores, gameplay timers, and player lives.

Goal

To add game-wide states, including scores and player lives.

Process

Creating the lives system

> There are many ways of having game state in Unity. We'll be using the most basic method – by simply attaching them to one GameObject, that can be accessed by any other object. This is sort of a **Singleton** design pattern, which has some drawbacks you might encounter in the later courses.

1. In your Hierarchy, create a new empty gameobject called `GameCore`. Add a new script, also called `GameCore`. It will start pretty basic:

```
public class GameCore : MonoBehaviour
{

}
```

1. Next, we're going to add a variable to keep track of how many lives the player has:

```
public class GameCore : MonoBehaviour
{
    public int playerLives = 3;
}
```

1. We can also create a method that will allow us to check the status of the lives count as we decrease them, in a similar way that we did with the Health script:

```
    public int playerLives = 3;

    public void decreasePlayerLives()
    {
        playerLives--;
        if (playerLives <= 0)
        {
            // Game over, dude!
            Debug.Log("GAME OVER");
        }
    }
```

> Hop over to your `PlayerDeath` script. We now need to connect it to the `GameCore` script, and decrease our lives count when the player dies. We'll need to create a new variable, connect the variable in the `Start` method, and call it in the `HandleDeath` method.

1. In `PlayerDeath`, add a new class variable:

```
public class PlayerDeath : Death
{
    public Vector3 spawnLocation;
    public Quaternion spawnRotation;

    public GameCore gameCore;
```

1. Connect the variable in the `Start` method:

```
    void Start()
    {
        spawnLocation = transform.position;
        spawnRotation = transform.rotation;

        gameCore = GameObject.FindObjectOfType<GameCore>();
    }
```

> `FindObjectOfType` will return the first instance of a type (or a script). It's super expensive to use, so use it rarely.

1. And make the call in `HandleDeath`:

```
    public override void HandleDeath()
    {
```

```
        // decrease the lives
        gameCore.decreasePlayerLives();

        GetComponent<Rigidbody>().position = spawnLocation;
```

> You can test the game now, and after three deaths, you should get the console output saying the game is over. We'll be handling what happens when the game is over in the next unit.

Creating the score system

> This next process will set up a scoring system, where destroying the enemies adds to a score. This process looks pretty similar to the lives system, except we'll be adding to the score when the enemy dies.

1. Firstly, add a new class variable to hold the score:

```
public class GameCore : MonoBehaviour
{
    public int playerLives = 3;
    public int playerScore = 0;
```

1. Then we add a new method to increase the score. Unlike the `playerDecreaseLives` method, this one takes a parameter to let us know by how much to increase the score, so different enemies can have different values.

```
    public void increasePlayerScore(int value)
    {
        playerScore += value;
    }
```

1. Now we're going to head over to the `TurretDeath` method to make additions. Start with the new class variable and assignment:

```
public class TurretDeath : Death
{
    public GameCore gameCore;

    void Start()
    {
        gameCore = GameObject.FindObjectOfType<GameCore>();
    }
```

> If you get super heebie-jeebies about this, that's not an incorrect feeling. What we're doing here is **repeating code**, which is especially bad as both `TurretDeath` and `PlayerDeath` both inherit from the same superclass, `Death`. There is a more clever way to do this, and I invite you to try to find it! For now, however, let's just repeast the code, and remember to refactor this later. Really!

1. Now we're going to make a new class variable for the value for destroying the enemy:

```
public class TurretDeath : Death
{
    public GameCore gameCore;
    public int value = 10;
```

1. Then send that value to the `GameCore` script in the `HandleDeath` method:

```
    public override void HandleDeath()
    {
        gameCore.IncreasePlayerScore(value);
        Destroy(gameObject);
    }
```

> Go ahead and test, and make sure when you destroy an enemy and get points for it.
>
> You can now also use the trigger system to add points, so the player can get points for reaching certain parts of the level.

Creating the timer system

> Lastly, we're going to take a look at how to have a timer system. There are a couple tricky parts to this – namely, whether you want it to count upwards or downwards, and what to do to "pause" the timer if we ever need to.
> For this first pass, we'll count down from an arbitrary number, and make the game end if the timer reaches zero.

1. Back in the `GameCore` script, we're going to add two new class variables:

```
public class GameCore : MonoBehaviour
{
    public int playerLives = 3;
    public int playerScore = 0;

    public float totalLevelSeconds = 20f; // we're going to set this low just for testing
    public float currentLevelSeconds;
```

1. Next, set the `currentLevelSeconds` in the `Start` method:

```
    void Start()
    {
        currentLevelSeconds = totalLevelSeconds;
    }
```

1. And in the `Update` method, we count down and check for the end:

```
    private void Update()
    {
```

```
        currentLevelSeconds -= Time.deltaTime;
        if (currentLevelSeconds <= 0f)
        {
            // Game over!
            Debug.Log("Time out!");
        }
    }
```

> Test your game, and see if you get the console message after twenty seconds.

Complete code

GameCore.cs

```
public class GameCore : MonoBehaviour
{
    public int playerLives = 3;
    public int playerScore = 0;

    public float totalLevelSeconds = 20f; // we're going to set this low just for testing
    public float currentLevelSeconds;

    void Start()
    {
        currentLevelSeconds = totalLevelSeconds;
    }

    private void Update()
    {
        currentLevelSeconds -= Time.deltaTime;
        if (currentLevelSeconds <= 0f)
        {
            // Game over!
            Debug.Log("Time out!");
        }
    }

    public void DecreasePlayerLives()
    {
        playerLives--;
        if (playerLives <= 0)
        {
            // Game over, dude!
            Debug.Log("Game over!");
        }
    }

    public void IncreasePlayerScore(int value)
    {
        playerScore += value;
    }
}
```

PlayerDeath.cs

```
public class PlayerDeath : Death
```

```
{
    public Vector3 spawnLocation;
    public Quaternion spawnRotation;

    public GameCore gameCore;

    void Start()
    {
        spawnLocation = transform.position;
        spawnRotation = transform.rotation;

        gameCore = GameObject.FindObjectOfType<GameCore>();
    }

    public override void HandleDeath()
    {
        // decrease the lives
        gameCore.DecreasePlayerLives();

        GetComponent<Rigidbody>().position = spawnLocation;
        GetComponent<Rigidbody>().rotation = spawnRotation;

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

        // Restore the player health to maximum
        if (TryGetComponent<Health>(out Health health))
        {
            health.RecoverFull();
        }
    }
}
```

TurretDeath.cs

```
public class TurretDeath : Death
{
    public GameCore gameCore;
    public int value = 10;
    void Start()
    {
        gameCore = GameObject.FindObjectOfType<GameCore>();
    }
    public override void HandleDeath()
    {
        gameCore.IncreasePlayerScore(value);
        Destroy(gameObject);
    }
}
```

Wrap-Up

In this unit we created a single spot for game-scoped variables to be stored and manipulated. In following units, we'll look at how we can show those variable on-screen in the user interface, and how to change scenes when the game is over.

Further Material

- The Singleton Design Pattern
- Unity Manual on FindObjectOfType