Zombie Shooter Project 2b

Task 1. Make a Bullet Script

Do this

- In the **Project view**, create a new **C# Script** in the **Scripts Folder**
- Name the Script **Bullet**

Do this

- Type out this code into your script file
- Make sure your code is ***EXACTLY*** the same!

```
using UnityEngine;

public class Bullet : MonoBehaviour {
    public float moveSpeed = 100.0f;
    public int damage = 1;

    private void Start(){
        GetComponent<Rigidbody2D>().AddForce(transform.up * moveSpeed);
    }

    private void OnTriggerEnter2D(Collider2D other)
    {
        other.transform.SendMessage("TakeDamage", damage, SendMessageOptions.DontRequireReceiver);
        Die();
    }

    private void OnBecameInvisible()
    {
        Die();
    }

    private void Die(){
        Destroy(gameObject);
    }
}
```

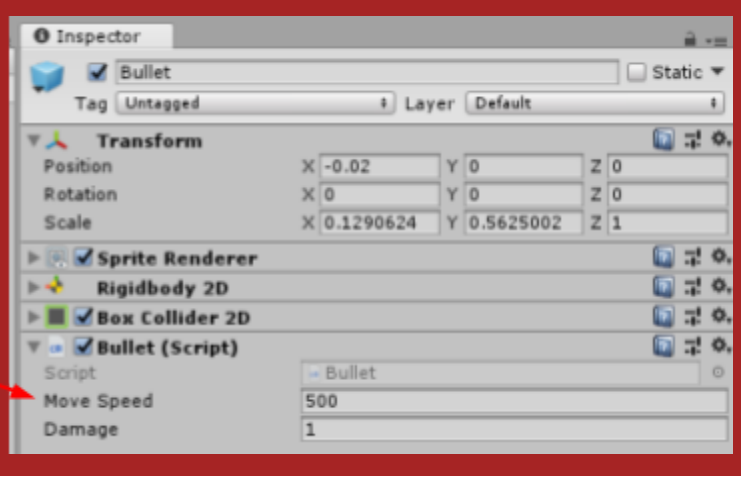
Do this

- Select the **Bullet Prefab** in the **Project view**
- **Drag** the **Bullet** script onto the **Bullet Prefab**

Check this

- Select the **Bullet Prefab** in the **Project view**
- In the Inspector, check the **Bullet Prefab** has the **Bullet** script attached

Check the Bullet script is attached



Explanation - moveSpeed property

- The **moveSpeed** the **Bullet** will travel at
- **moveSpeed** is a **public** property, so it is **editable** in the **Unity Editor**
- **moveSpeed** is a **float**, a **decimal number**
- **100.0f** is the **default speed** of the **Bullet**

```
public float moveSpeed = 100.0f;
```

Explanation - damage property

- The **damage** the **Bullet** does
- **damage** is **Editable** in the **Unity Editor**, because it is a **public property**
- **damage** is a type of **int** (a whole number like 1 or 39)

```
public int damage = 1;
```

Explanation - code breakdown

Set the Rigidbody2D to move when the bullet is spawned

```
using UnityEngine;

[RequireComponent(typeof(Rigidbody2D))]
public class Bullet : MonoBehaviour
{
    public float moveSpeed = 100;
    public int damage = 1;


    private void Start()
    {
        GetComponent<Rigidbody2D>().AddForce(transform.up * moveSpeed);
    }

    private void OnTriggerEnter2D(Collider2D other)
    {
        other.transform.SendMessage("TakeDamage", damage, SendMessageOptions.DontRequireReceiver);
        Die();
    }

    private void OnBecameInvisible()
    {
        Die();
    }

    private void Die()
    {
        Destroy(gameObject);
    }
}
```

Send a message to the GameObject we hit to take damage

Call the Die method

When the bullet is off the screen (invisible) call the Die method

Destroy the bullet in the scene

Explanation - Start method

- The **Start** method is a MonoBehaviour method
- **Start** is called **ONCE** when the **Bullet** is created or the game starts, whichever is first
  - We will spawn Bullets from a gun soon, so this will run when the Bullet spawns
- Start runs just **BEFORE** FixedUpdate and Update
- Because it is only called once, we can use it to set things up ready for any update methods

```
private void Start(){

}
```

Useful links

- Please refer to the **Event Functions help sheet** on the **DLE website** for more information on **Start**

Explanation - code breakdown

Get the Rigidbody2D component

```
private void Start()
{
    GetComponent<Rigidbody2D>().AddForce(transform.up * moveSpeed);
}
```

Call the AddForce method

Give AddForce our Transforms facing direction (up)
Multiplied by moveSpeed

Explanation - Line 1

- We get hold of our **Rigidbody2D** component using **GetComponent**
- **Rigidbody2D** has a method called **AddForce** that will "push" the bullet in a direction
- **AddForce** needs a **Vector2** (X and Y) or **Vector3** (X,Y and Z) variable as a direction
- We want to move the bullet in the direction it's facing
- We can use **transform.up** to get our facing direction
- To move our bullet at the correct speed, we can multiply **transform.up** by our public variable, **moveSpeed**

```
private void Start(){
    GetComponent<Rigidbody2D>().AddForce(transform.up * moveSpeed);
}
```

Useful links

- More information about **Rigidbody2D** — Rigidbody2D - manual
- More information about **Rigidbody2D.AddForce** — Rigidbody2D.AddForce - Scripting Reference
- More information about **Transform** — Transform - Scripting Reference
- More information about **Transform.up** — Transform.up - Scripting Reference

Explanation - OnTriggerEnter2D method

- The **OnTriggerEnter2D** method is a MonoBehaviour method
- **OnTriggerEnter2D** is called every time a GameObject **overlaps** another GameObject
  - Both GameObjects require Collider2D components
  - At least one requires a Rigidbody2D component
- **OnTriggerEnter2D** provides the "other" object we overlapped in it's parameter, named "other"
  - The "other" object is the Collider2D component on the GameObject we just overlapped
  - We can get at any of the components on the "other" GameObject using this parameter!
  - For example, we can get its Transform component using "other.transform"
  - Or if it has a **Rigidbody2D**, we can get it using "other.transform.GetComponent<Rigidbody2D>()"

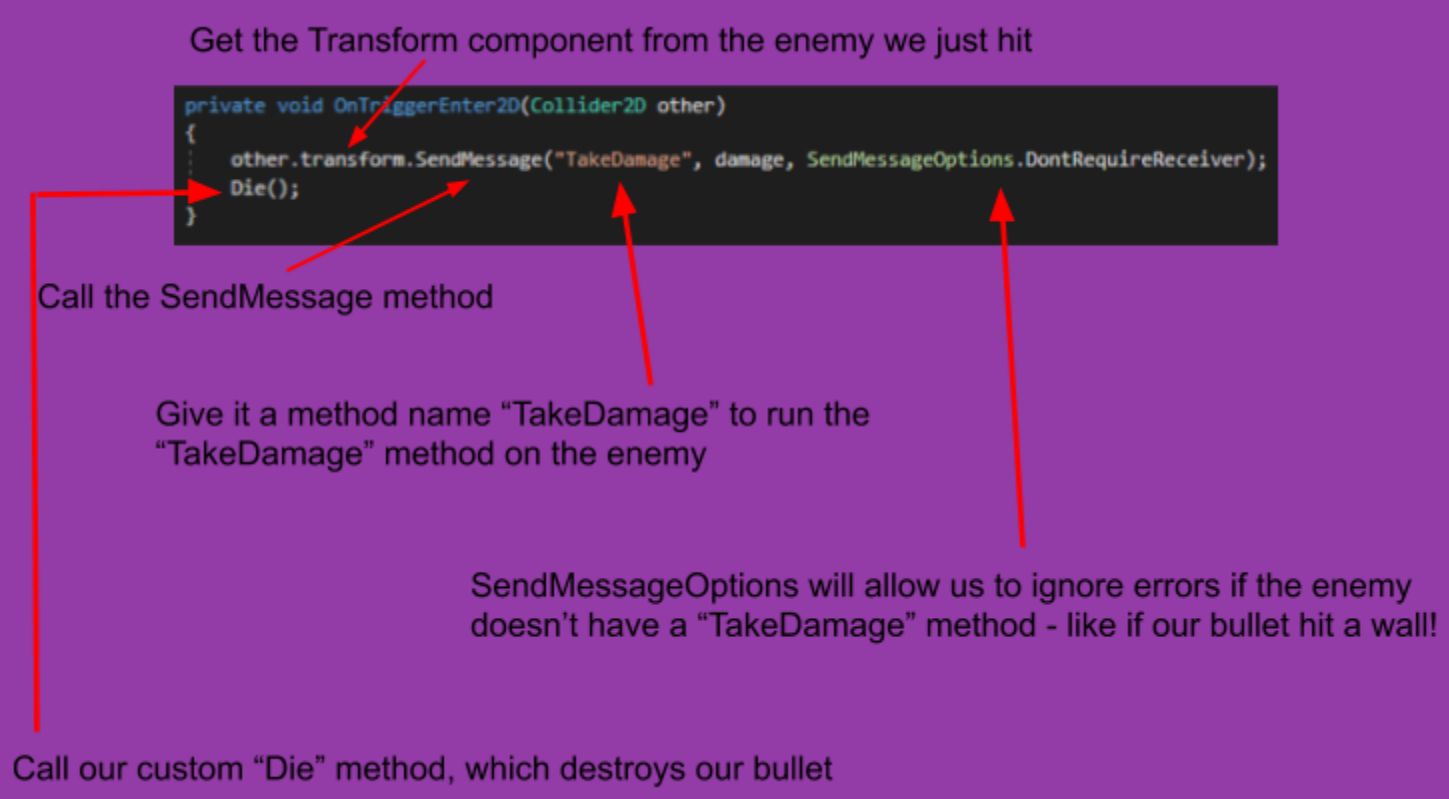
```
private void OnTriggerEnter2D(Collider2D other) {

}
```

Useful links

- More information about **OnTriggerEnter2D** — OnTriggerEnter2D - Scripting Reference
- More information about **Collider2D** — Collider2D - Scripting Reference

Explanation - code breakdown



Explanation - Line 1

- We want to tell the Enemy it has been damaged and give it our bullet's **damage**
- The enemy will have a **TakeDamage** method that takes an int variable for **damage**
- We will use the SendMessage method (part of the Monobehaviour class) to run the **TakeDamage** method on the enemy and give it our damage variable
- We can access the enemy transform using the "other" parameter in the **OnTriggerEnter** method
- The enemy transform will give us access to it's SendMessage method
- NOTE: SendMessage can give an error if it can't run the method - like if we don't have a **TakeDamage** method on the enemy!
- NOTE: we can "ignore" the error using **SendMessageOptions** - it has a setting called "DontRequireReceiver" for this!

- SendMessage requires 3 parameters - a method name (as a string), an optional parameter to send and an optional "options"
- Our method name is "TakeDamage"
- Our optional parameter is damage (our public variable **damage** on the bullet)
- Our "options" is DontRequireReciever, which ignores errors if the "TakeDamage" method is not found on the enemy we hit

```
private void OnTriggerEnter2D(Collider2D other) {
    other.transform.SendMessage("TakeDamage", damage, SendMessageOptions.DontRequireReceiver);
    Die();
}
```

Useful links

- More information about **SendMessage** — SendMessage - Scripting Reference
- More information about **SendMessageOptions** — SendMessageOptions - Scripting Reference

Explanation - Line 2

- We call the custom method **Die** on our bullet
- **Die** will destroy the bullet in the scene
- The **Die** method is explained later in this document!

```
private void OnTriggerEnter2D(Collider2D other) {
    other.transform.SendMessage("TakeDamage", damage, SendMessageOptions.DontRequireReceiver);
    Die();
}
```

Explanation - OnBecameInvisible method

- The **OnBecameInvisible** method is a MonoBehaviour method
- **OnBecameInvisible** is called when a GameObject can no longer be seen by our Camera (it is off the screen)
    - The GameObject is "invisible" to the camera

```
private void OnBecameInvisible() {

}
```

Useful links

- More information about **OnBecameInvisible** — OnBecameInvisible - Scripting Reference

Explanation - Line 1

- We call the custom method **Die** on our bullet
- **Die** will destroy the bullet in the scene
- The **Die** method is explained later in this document!

```
private void OnBecameInvisible() {
    Die();
}
```

Explanation - Die method

- Die is a Custom method, meaning we made it up for our Bullet!
- All it will do is Destroy our Bullet using the **Destroy** method, removing the bullet from the scene

```
private void Die() {

}
```

Explanation - method signatures

- All methods have a **signature**
- A **signature** is a methods parts OUTSIDE of the curly brackets (with some exceptions)
- Our custom Die method has 3 parts to its signature
    - The **return type** "void"
        - A return type of "void" returns nothing!
        - Note: A method's return type isn't "technically" part of the signature in official C# documentation
    - The **method name** "Die"
    - The **parameter list** "()"
        - The parameter list can have values which will be used inside the method - we will see that later!
- NOTE: the "private" part of the signature denotes if other classes can access it (they can't for this method)

```
private void Die() {

}
```

Useful links

- More information about **C# Methods** — C# Methods

Explanation - code breakdown

Destroy this GameObject

```
void Die(){
    Destroy( gameObject );
}
```

A reference to the Bullet GameObject

Explanation - Line 1

- In our custom method, we call the Destroy method, MonoBehaviours way of destroying GameObjects
- Note we give the parameter “gameObject”
- gameObject refers to the Bullet itself
- The same as saying “I want to destroy me”
- Suicidal GameObjects? Sad times.

```
private void Die() {
    Destroy( gameObject );
}
```

Useful links

- More information about **Destroy** Destroy
