



5.1 Clicky Mouse

Steps:

Step 1: Create project and switch to 2D view

Step 2: Create good and bad targets

Step 3: Toss objects randomly in the air

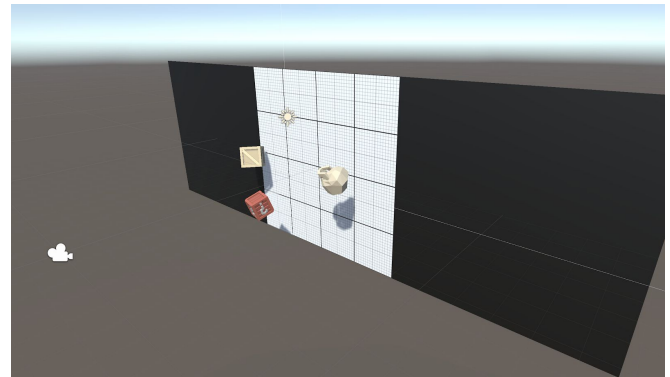
Step 4: Replace messy code with new methods

Step 5: Create object list in Game Manager

Step 6: Create a coroutine to spawn objects

Step 7: Destroy target with click and sensor

Example of project by end of lesson



Length: 60 minutes

Overview: It's time for the final unit! We will start off by creating a new project and importing the starter files, then switching the game's view to 2D. Next we will make a list of target objects for the player to click on: Three "good" objects and one "bad". The targets will launch spinning into the air after spawning at a random position at the bottom of the map. Lastly, we will allow the player to destroy them with a click!

Project Outcome: A list of three good target objects and one bad target object will spawn in a random position at the bottom of the screen, thrusting themselves into the air with random force and torque. These targets will be destroyed when the player clicks on them or they fall out of bounds.

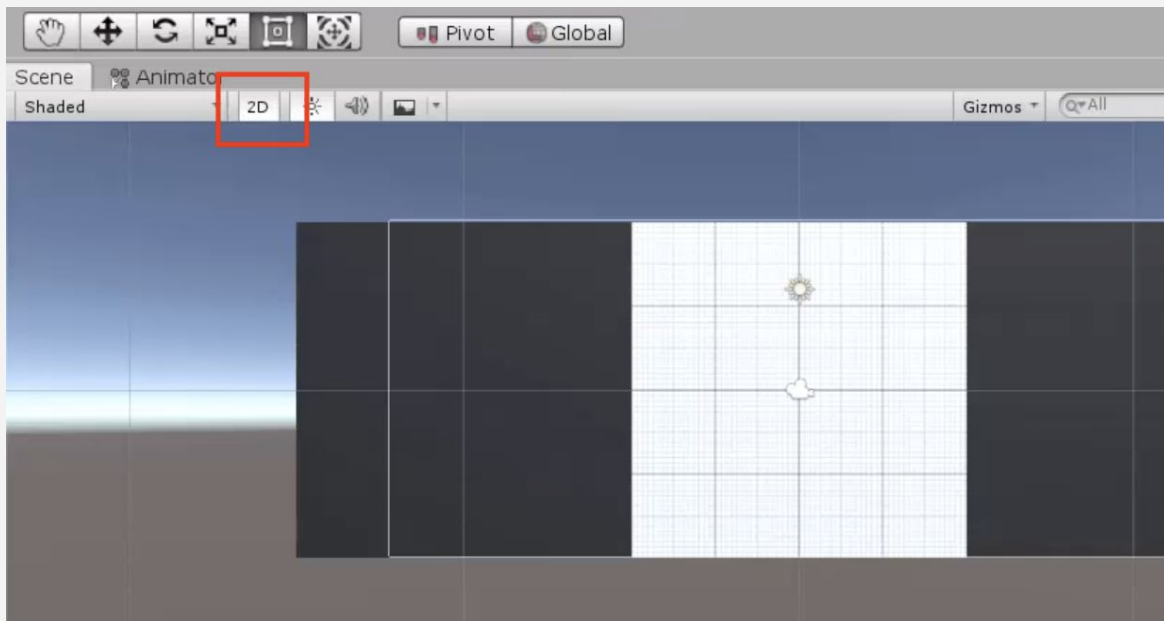
Learning Objectives: By the end of this lesson, you will be able to:

- Switch the game to 2D view for a different perspective
- Add torque to the force of an object
- Create a Game Manager object that controls game states as well as spawning
- Create a List of objects and return their length with Count
- Use While Loops to repeat code while something is true
- Use OnMouseDown to enable the player to click on things

Step 1: Create project and switch to 2D view

One last time... we need to create a new project and download the starter files to get things up and running.

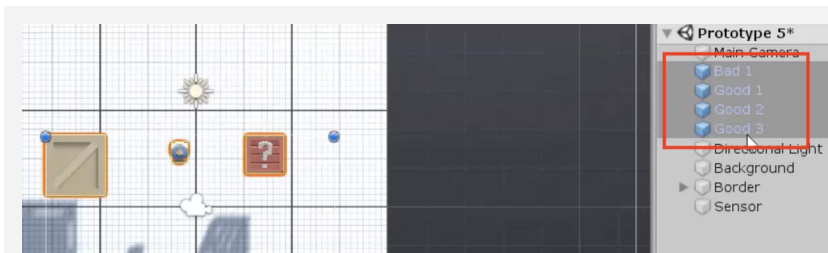
1. Open **Unity Hub** and create "Prototype 5" in your course directory on correct version in 3D
 2. Click on the link to access the Prototype 5 **starter files**, then **download and import** them into Unity
 3. Open the **Prototype 5** scene, then delete the **sample scene** without saving
 4. Click on the **2D icon** in Scene view to put Scene view in **2D**
 5. (optional) Change the texture and color of the **background** and the color of the **borders**
- **New Concept:** 2D View
 - **Demo:** Notice in 2D view: You can't rotate around objects or move them in the Z direction



Step 2: Create good and bad targets

The first thing we need in our game are three good objects to collect, and one bad object to avoid. It'll be up to you to decide what's good and what's bad.

1. From the **Library**, drag 3 "good" objects and 1 "bad" object into the Scene, rename them "Good 1", "Good 2", "Good 3", and "Bad 1"
 2. Add **Rigid Body** and **Box Collider** components, then make sure that Colliders surround objects properly
 3. Create a new Scripts folder, a new "Target.cs" script inside it, attach it to the **Target objects**
 4. Drag all 4 targets into the **Prefabs** folder to create "original prefabs", then **delete** them from the scene
- **Tip:** The bigger the collider boxes, the easier it will be to hit them
 - **Tip:** Try selecting multiple objects and applying scripts/components - very handy



Step 3: Toss objects randomly in the air

Now that we have 4 target prefabs with the same script, we need to toss them into the air with a random force, torque, and position.

1. In **Target.cs**, declare a new **private Rigidbody targetRb**; and initialize it in **Start()**
 2. In **Start()**, add an **upward force** multiplied by a **randomized speed**
 3. Add a **torque** with randomized **xyz values**
 4. Set the **position** with a randomized **X value**
- **New Function:** AddTorque
 - **Tip:** Test with different values by dragging them in during runtime
 - **Don't worry:** We're going to fix all these hard-coded values next

```
private Rigidbody targetRb;

void Start() {
    targetRb = GetComponent<Rigidbody>();
    targetRb.AddForce(Vector3.up * Random.Range(12, 16), ForceMode.Impulse);
    targetRb.AddTorque(Random.Range(-10, 10), Random.Range(-10, 10),
        Random.Range(-10, 10), ForceMode.Impulse);
    transform.position = new Vector3(Random.Range(-4, 4), -6); }
```

Step 4: Replace messy code with new methods

Instead of leaving the random force, torque, and position making our `Start()` function messy and unreadable, we're going to store each of them in brand new clearly named custom methods.

1. Declare and initialize new private float variables for ***minSpeed***, ***maxSpeed***, ***maxTorque***, ***xRange***, and ***ySpawnPos***;
2. Create a new function for ***Vector3 RandomForce()*** and call it in ***Start()***
3. Create a new function for ***float RandomTorque()***, and call it in ***Start()***
4. Create a new function for ***RandomSpawnPos()***, have it return a new ***Vector3*** and call it in ***Start()***

```
private float minSpeed = 12; private float maxSpeed = 16;
private float maxTorque = 10; private float xRange = 4;
private float ySpawnPos = -6;

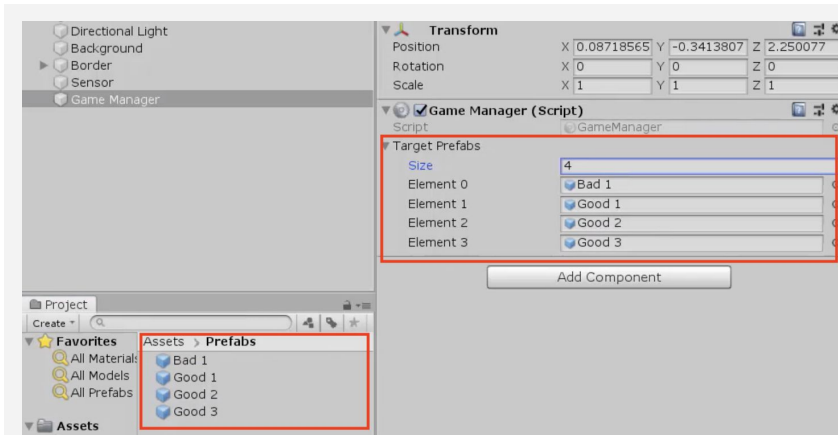
void Start() {
    ... targetRb.AddForce(... RandomForce(), ForceMode.Impulse);
    targetRb.AddTorque(... RandomTorque(), RandomTorque(),
    RandomTorque(), ForceMode.Impulse);
    transform.position = new Vector3(... RandomSpawnPos; }

Vector3 RandomForce() { return Vector3.up * Random.Range(minSpeed, maxSpeed);
}
float RandomTorque() { return Random.Range(-maxTorque, maxTorque); }
Vector3 RandomSpawnPos() { return new Vector3(Random.Range(-xRange, xRange),
ySpawnPos); }
```

Step 5: Create object list in Game Manager

The next thing we should do is create a list for these objects to spawn from. Instead of making a Spawn Manager for these spawn functions, we're going to make a Game Manager that will also control game states later on.

1. Create a new "Game Manager" **Empty object**, attach a new **GameManager.cs** script, then open it
 2. Declare a new **public List<GameObject> targets;**, then in the Game Manager inspector, change the list **Size** to 4 and assign your **prefabs**
- **New Concept:** Lists
 - **New Concept:** Game Manager
 - **Demo:** Feel free to reference old code: We used an array instead of a list to spawn the animals in Unit 2



Step 6: Create a coroutine to spawn objects

Now that we have a list of object prefabs, we should instantiate them in the game using coroutines and a new type of loop.

1. Declare and initialize a new **private float spawnRate** variable
 2. Create a new **IEnumerator SpawnTarget ()** method
 3. Inside the new method, **while(true)**, wait **1 second**, generate a **random index**, and spawn a random **target**
 4. In **Start()**, use the **StartCoroutine** method to begin spawning objects
- **Tip:** Feel free to reference old code: we used coroutines for the powerup cooldown in Unit 4
 - **Tip:** Arrays return an integer with `.Length`, while Lists return an integer with `.Count`
 - **New Concept:** While Loops

```
private float spawnRate = 1.0f;

void Start() { StartCoroutine(SpawnTarget()); }

IEnumerator SpawnTarget() {
    while (true) {
        yield return new WaitForSeconds(spawnRate);
        int index = Random.Range(0, targets.Count);
        Instantiate(targets[index]); } }
```

Step 7: Destroy target with click and sensor

Now that our targets are spawning and getting tossed into the air, we need a way for the player to destroy them with a click. We also need to destroy any targets that fall below the screen.

1. In **Target.cs**, add a new method for **private void OnMouseDown()** {}, and inside that method, destroy the gameObject
 2. Add a new method for **private void OnTriggerEnter(Collider other)** and inside that function, destroy the gameObject
- **New Function:** OnMouseDown
 - **Tip:** There is also OnMouseUp, and OnMouseEnter, but Down is definitely the one we want
 - **Tip:** You could use Update and check if target y position is lower than a certain value, but a sensor is better because it doesn't run all the time

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

private void OnTriggerEnter(Collider other) {
    Destroy(gameObject); }
```

Lesson Recap

New Functionality

- Random objects are tossed into the air on intervals
- Objects are given random speed, position, and torque
- If you click on an object, it is destroyed

New Concepts and Skills

- 2D View
- AddTorque
- Game Manager
- Lists
- While Loops
- Mouse Events

Next Lesson

- We'll add some effects and keep track of score!



5.2 Keeping Score

Steps:

Step 1: Add Score text position it on screen

Step 2: Edit the Score Text's properties

Step 3: Initialize score text and variable

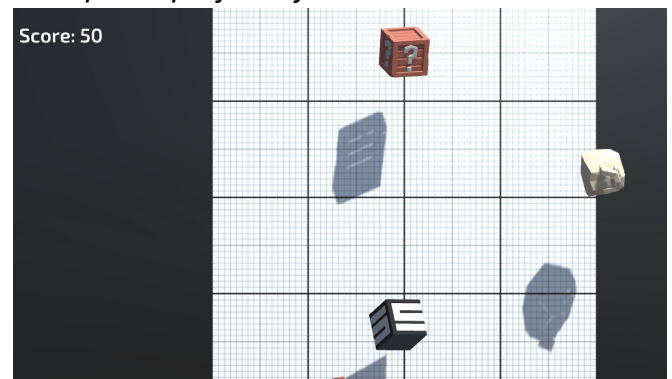
Step 4: Create a new UpdateScore method

Step 5: Add score when targets are destroyed

Step 6: Assign a point value to each target

Step 7: Add a Particle explosion

Example of project by end of lesson



Length: 60 minutes

Overview: Objects fly into the scene and the player can click to destroy them, but nothing happens. In this lesson, we will display a score in the user interface that tracks and displays the player's points. We will give each target object a different point value, adding or subtracting points on click. Lastly, we will add cool explosions when each target is destroyed.

Project Outcome: A "Score: " section will display in the UI, starting at zero. When the player clicks a target, the score will update and particles will explode as the target is destroyed. Each "Good" target adds a different point value to the score, while the "Bad" target subtracts from the score.

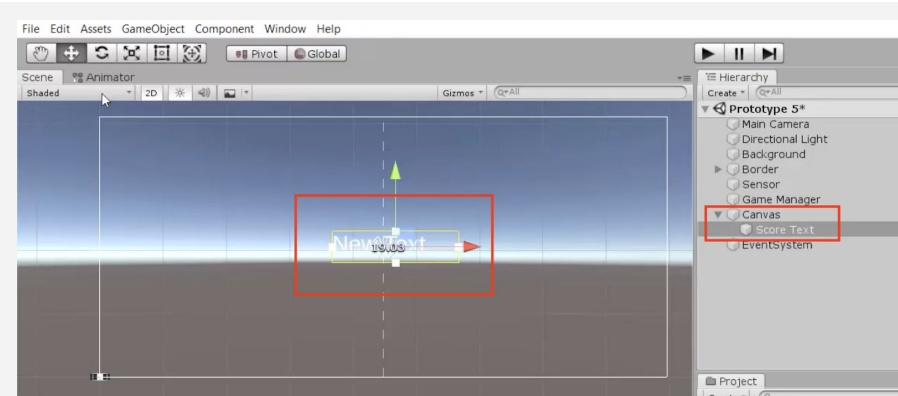
Learning Objectives: By the end of this lesson, you will be able to:

- Create UI Elements in the Canvas
- Lock elements and objects into place with Anchors
- Use variables and script communication to update elements in the UI

Step 1: Add Score text position it on screen

In order to display the score on-screen, we need to add our very first UI element.

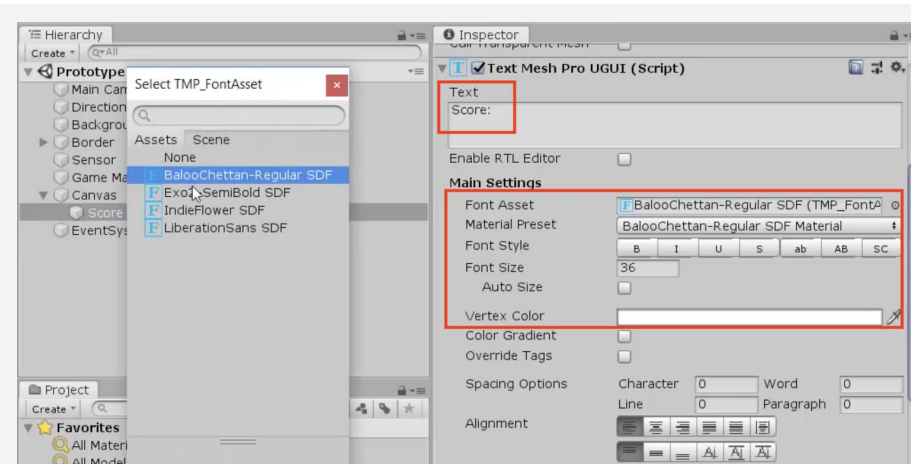
1. Create > UI > **TextMeshPro text**, then if prompted click the button to **Import TMP Essentials**
 2. Rename the new object "Score Text", then **zoom out** to see the **canvas** in Scene view
 3. Change the **Anchor Point** so that it is anchored from the **top-left corner**
 4. In the inspector, change its **Pos X** and **Pos Y** so that it is in the top-left corner
- **New Concept:** Text Mesh Pro / TMPPro
 - **New Concept:** Canvas
 - **New Concept:** Anchor Points
 - **Tip:** Look at how it displays in scene vs game view. It may be hard to see white text depending on the background



Step 2: Edit the Score Text's properties

Now that the basic text is in the scene and positioned properly, we should edit its properties so that it looks nice and has the correct text.

1. Change its text to "Score:"
2. Choose a **Font Asset**, **Style**, **Size**, and **Vertex color** to look good with your background



Step 3: Initialize score text and variable

We have a great place to display score in the UI, but nothing is displaying there! We need the UI to display a score variable, so the player can keep track of their points.

1. At the top of **GameManager.cs**, add `"using TMPro;"`
2. Declare a new **public TextMeshProUGUI scoreText**, then assign that variable in the inspector
3. Create a new **private int score** variable and initialize it in **Start()** as `score = 0;`
4. Also in **Start()**, set `scoreText.text = "Score: " + score;`

- **New Concept:**
Importing Libraries

```
private int score;
public TextMeshProUGUI scoreText;

void Start() {
    StartCoroutine(SpawnTarget());
    score = 0;
    scoreText.text = "Score: " + score; }
```

Step 4: Create a new UpdateScore method

The score text displays the score variable perfectly, but it never gets updated. We need to write a new function that racks up points to display in the UI.

1. Create a new **private void UpdateScore()** method
2. Cut and paste `scoreText.text = "Score: " + score;` into the new method, then call **UpdateScore()** in **Start()**
3. Add the parameter **int scoreToAdd** to the **UpdateScore** method, then fix the error in **Start()** by passing it a value of **zero**
4. In **UpdateScore()**, increase the score by setting `score += scoreToAdd;`
5. Call **UpdateScore(5)** in the **spawnTarget()** function

- **Don't worry:** It doesn't make sense to add to score when spawned, this is just temporary

```
void Start() {
    ... score = 0;
    scoreText.text = "Score: " + score; UpdateScore(0); }

IEnumerator SpawnTarget() {
    while (true) { ... UpdateScore(5); }

    private void UpdateScore(int scoreToAdd) {
        score += scoreToAdd;
        scoreText.text = "Score: " + score; }
```

Step 5: Add score when targets are destroyed

Now that we have a method to update the score, we should call it in the target script whenever a target is destroyed.

1. In Target.cs, create a reference to **private** **GameManager gameManager;**
2. Initialize GameManager in **Start()** using the **Find()** method
3. In GameManager.cs, make the **UpdateScore** method **public**
4. When a target is **destroyed**, call **UpdateScore(5);**, then **delete** the method call from SpawnTarget()

- **Tip:** Feel free to reference old code: We used script communication in Unit 3 to stop the game on GameOver
- **Warning:** If you try to call UpdateScore while it's private, it won't work

```
private GameManager gameManager;

void Start() {
    ... gameManager = GameObject.Find("Game
Manager").GetComponent<GameManager>();}

private void OnMouseDown() {
    Destroy(gameObject); gameManager.UpdateScore(5); }

private-public void UpdateScore(int scoreToAdd) { ... }
```

Step 6: Assign a point value to each target

The score gets updated when targets are clicked, but we want to give each of the targets a different value. The good objects should vary in point value, and the bad object should subtract points.

1. In Target.cs, create a new **public int pointValue** variable
2. In each of the **Target prefab's** inspectors, set the **Point Value** to whatever they're worth, including the bad target's **negative value**
3. Add the new variable to **UpdateScore(pointValue);**

- **Tip:** Here's the beauty of variables at work. Each target can have their own unique pointValue!

```
public int pointValue;

private void OnMouseDown() {
    Destroy(gameObject);
    gameManager.UpdateScore(5 pointValue); }
```

Step 7: Add a Particle explosion

The score is totally functional, but clicking targets is sort of... unsatisfying. To spice things up, let's add some explosive particles whenever a target gets clicked!

1. In Target.cs, add a new **public ParticleSystem explosionParticle** variable
2. For each of your target prefabs, assign a **particle prefab** from *Course Library > Particles* to the **Explosion Particle** variable
3. In the **OnMouseDown()** function, **instantiate** a new explosion prefab

```
public ParticleSystem explosionParticle;

private void OnMouseDown() {
    Destroy(gameObject);
    Instantiate(explosionParticle, transform.position,
        explosionParticle.transform.rotation);
    GameManager.UpdateScore(5 pointValue); }
```

Lesson Recap

New Functionality

- There is a UI element for score on the screen
- The player's score is tracked and displayed by the score text when hit a target
- There are particle explosions when the player gets an object

New Concepts and Skills

- TextMeshPro
- Canvas
- Anchor Points
- Import Libraries
- Custom methods with parameters
- Calling methods from other scripts

Next Lesson

- We'll use some UI elements again - this time to tell the player the game is over and reset our game!



5.3 Game Over

Steps:

Step 1: Create a Game Over text object

Step 2: Make GameOver text appear

Step 3: Create GameOver function

Step 4: Stop spawning and score on GameOver

Step 5: Add a Restart button

Step 6: Make the restart button work

Step 7: Show restart button on game over

Example of project by end of lesson



Length: 60 minutes

Overview: We added a great score counter to the game, but there are plenty of other game-changing UI elements that we could add. In this lesson, we will create some “Game Over” text that displays when a “good” target object drops below the sensor. During game over, targets will cease to spawn and the score will be reset. Lastly, we will add a “Restart Game” button that allows the player to restart the game after they have lost.

Project Outcome: When a “good” target drops below the sensor at the bottom of the screen, the targets will stop spawning and a “Game Over” message will display across the screen. Just underneath the “Game Over” message will be a “Reset Game” button that reboots the game and resets the score, so the player can enjoy it all over again.

Learning Objectives: By the end of this lesson, you will be able to:

- Make UI elements appear and disappear with .SetActive
- Use Script Communication and Game states to have a working “Game Over” screen
- Restart the game using a UI button and Scene Management

Step 1: Create a Game Over text object

If we want some “Game Over” text to appear when the game ends, the first thing we’ll do is create and customize a new UI text element that says “Game Over”.

1. Right-click on the **Canvas**, create a new UI > **TextMeshPro - Text** object, and rename it “Game Over Text”
2. In the inspector, edit its **Text**, **Pos X**, **Pos Y**, **Font Asset**, **Size**, **Style**, **Color**, and **Alignment**
3. Set the “Wrapping” setting to “Disabled”

- **Tip:** The center of the screen is the best place for this Game Over message - it grabs the player’s attention



Step 2: Make GameOver text appear

We’ve got some beautiful Game Over text on the screen, but it’s just sitting and blocking our view right now. We should deactivate it, so it can reappear when the game ends.

1. In GameManager.cs, create a new **public TextMeshProUGUI gameOverText**; and assign the **Game Over** object to it in the inspector
2. **Uncheck** the Active checkbox to **deactivate** the Game Over text by default
3. In **Start()**, activate the Game Over text

- **Don’t worry:** We’re just doing this temporarily to make sure it works

```
public TextMeshProUGUI gameOverText;

void Start() {
    ...
    gameOverText.gameObject.SetActive(true); }
```

Step 3: Create GameOver function

We've temporarily made the "Game Over" text appear at the start of the game, but we actually want to trigger it when one of the "Good" objects is missed and falls.

1. Create a new **public void GameOver()** function, and **move** the code that activates the game over text inside it
2. In Target.cs, call **gameManager.GameOver()** if a target collides with the **sensor**
3. Add a new "Bad" tag to the **Bad object**, add a condition that will only trigger game over if it's *not* a bad object

```
void Start() {
    ... gameOverText.gameObject.SetActive(true); }

public void GameOver() {
    gameOverText.gameObject.SetActive(true); }

<----->
private void OnTriggerEnter(Collider other) {
    Destroy(gameObject);
    if (!gameObject.CompareTag("Bad")) { gameManager.GameOver(); } }
```

Step 4: Stop spawning and score on GameOver

The "Game Over" message appears exactly when we want it to, but the game itself continues to play. In order to truly halt the game and call this a "Game Over", we need to stop spawning targets and stop generating score for the player.

1. Create a new **public bool isGameActive**;
2. As the **first line** in **Start()**, set **isGameActive = true**; and in **GameOver()**, set **isGameActive = false**;
3. To prevent spawning, in the **SpawnTarget()** coroutine, change **while (true)** to **while (isGameActive)**
4. To prevent scoring, in Target.cs, in the **OnMouseDown()** function, add the condition **if (gameManager.isGameActive)** {

```
public bool isGameActive;

void Start() { ... isGameActive = true; }

public void GameOver() { ... isGameActive = false; }

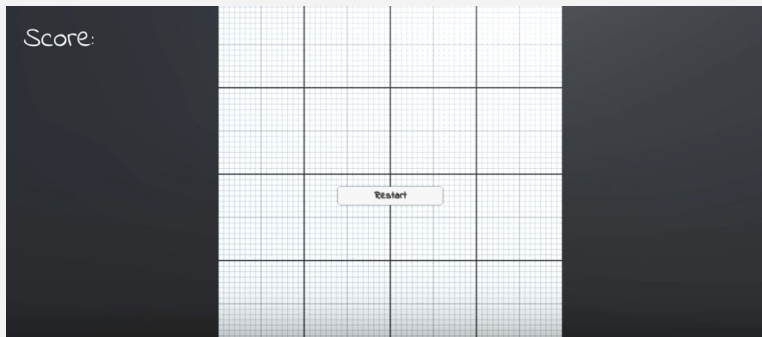
IEnumerator SpawnTarget() { while (true isGameActive) { ... }
<----->
private void OnMouseDown() {
    if (gameManager.isGameActive) { ... [all function code moved inside] }}
```

Step 5: Add a Restart button

Our Game Over mechanics are working like a charm, but there's no way to replay the game. In order to let the player restart the game, we will create our first UI button

1. Right-click on the **Canvas** and *Create > UI > Button*
2. Rename the button "Restart Button"
3. Temporarily **reactivate** the Game Over text in order to reposition the Restart Button nicely with the text, then **deactivate** it again
4. Select the Text child object, then edit its **Text** to say "Restart", its **Font**, **Style**, and **Size**

- **New Concept:**
Buttons



Step 6: Make the restart button work

We've added the Restart button to the scene and it LOOKS good, but now we need to make it actually work and restart the game.

1. In GameManager.cs, add **using UnityEngine.SceneManagement;**
2. Create a new **void RestartGame()** function that reloads the current scene
3. In the **Button's** inspector, click **+** to add a new **On Click event**, drag it in the **Game Manager** object and select the **GameManager.RestartGame** function

- **New Concept:** Scene Management
- **New Concept:** On Click Event
- **Don't worry:** The restart button is just sitting there for now, but we will fix it later

```
using UnityEngine.SceneManagement;

void RestartGame() {
    SceneManager.LoadScene(SceneManager.GetActiveScene().name); }
```

Step 7: Show restart button on game over

The Restart Button looks great, but we don't want it in our faces throughout the entire game. Similar to the "Game Over" message, we will turn off the Restart Button while the game is active.

1. At the top of GameManager.cs add **using UnityEngine.UI;**
2. Declare a new **public Button restartButton;** and assign the **Restart Button** to it in the inspector
3. **Uncheck** the "Active" checkbox for the **Restart Button** in the inspector
4. In the **GameOver** function, activate the **Restart Button**

- **Tip:** Adding "using UnityEngine.UI" allows you to access the Button class

```
using UnityEngine.UI;

public Button restartButton;

public void GameOver() { ...
    restartButton.gameObject.SetActive(true); }
```

Lesson Recap

New Functionality

- A functional Game Over screen with a Restart button
- When the Restart button is clicked, the game resets

New Concepts and Skills

- Game states
- Buttons
- On Click events
- Scene management Library
- UI Library
- Booleans to control game states

Next Lesson

- In our next lesson, we'll use buttons to really add some difficulty to our game



5.4 What's the Difficulty?

Steps:

Step 1: Create Title text and menu buttons

Step 2: Add a DifficultyButton script

Step 3: Call SetDifficulty on button click

Step 4: Make your buttons start the game

Step 5: Deactivate Title Screen on StartGame

Step 6: Use a parameter to change difficulty

Example of project by end of lesson



Length: 60 minutes

Overview: It's time for the final lesson! To finish our game, we will add a Menu and Title Screen of sorts. You will create your own title, and style the text to make it look nice. You will create three new buttons that set the difficulty of the game. The higher the difficulty, the faster the targets spawn!

Project Outcome: Starting the game will open to a beautiful menu, with the title displayed prominently and three difficulty buttons resting at the bottom of the screen. Each difficulty will affect the spawn rate of the targets, increasing the skill required to stop "good" targets from falling.

Learning Objectives: By the end of this lesson, you will be able to:

- Store UI elements in a parent object to create Menus, UI, or HUD
- Add listeners to detect when a UI Button has been clicked
- Set difficulty by passing parameters into game functions like SpawnRate

Step 1: Create Title text and menu buttons

The first thing we should do is create all of the UI elements we're going to need. This includes a big title, as well as three difficulty buttons.

1. Duplicate your **Game Over text** to create your **Title Text**, editing its name, text and all of its attributes
2. Duplicate your **Restart Button** and edit its attributes to create an "Easy Button" button
3. Edit and duplicate the new Easy **button** to create a "Medium Button" and a "Hard Button"

- **Tip:** You can position the title and buttons however you want, but you should try to keep them central and visible to the player



Step 2: Add a DifficultyButton script

Our difficulty buttons look great, but they don't actually do anything. If they're going to have custom functionality, we first need to give them a new script.

1. For all 3 new buttons, in the Button component, in the **On Click ()** section, click the **minus (-)** button to remove the RestartGame functionality
2. Create a new **DifficultyButton.cs** script and attach it to **all 3 buttons**
3. Add **using UnityEngine.UI** to your imports
4. Create a new **private Button button;** variable and initialize it in **Start()**

```
using UnityEngine.UI;

private Button button;

void Start() {
    button = GetComponent<Button>(); }
}
```

Step 3: Call SetDifficulty on button click

Now that we have a script for our buttons, we can create a *SetDifficulty* method and tie that method to the click of those buttons

1. Create a new **void SetDifficulty** function, and inside it, **Debug.Log(gameObject.name + " was clicked")**;
 2. Add the **button listener** to call the **SetDifficulty** function
- **New Function:** AddListener
 - **Don't worry:** onClick.AddListener is similar what we did in the inspector with the Restart button
 - **Don't worry:** We're just using Debug for testing, to make sure the buttons are working

```
void Start() {
    button = GetComponent<Button>();
    button.onClick.AddListener(SetDifficulty); }

void SetDifficulty() {
    Debug.Log(gameObject.name + " was clicked"); }
```

Step 4: Make your buttons start the game

The Title Screen looks great if you ignore the target objects bouncing around, but we have no way of actually starting the game. We need a *StartGame* function that can communicate with *SetDifficulty*.

1. In GameManager.cs, create a new **public void StartGame()** function and move everything from **Start()** into it
 2. In DifficultyButton.cs, create a new **private GameManager gameManager**; and initialize it in **Start()**
 3. In the **SetDifficulty()** function, call **gameManager.startGame()**;
- **Don't worry:** Title objects don't disappear yet - we'll do that next

```
void Start() { ... }

public void StartGame() {
    isActive = true;
    score = 0;
    StartCoroutine(SpawnTarget());
    UpdateScore(0); }

<----->
private GameManager gameManager;

void Start () { ...
    gameManager = GameObject.Find("Game Manager").GetComponent<GameManager>(); }

SetDifficulty() { ... gameManager.startGame(); }
```

Step 5: Deactivate Title Screen on StartGame

If we want the title screen to disappear when the game starts, we should store them in an empty object rather than turning them off individually. Simply deactivating the single empty parent object makes for a lot less work.

1. Right-click on the Canvas and *Create > Empty Object*, rename it "Title Screen", and drag the **3 buttons** and **title** onto it
2. In GameManager.cs, create a new **public GameObject titleScreen;** and assign it in the inspector
3. In **StartGame()**, deactivate the title screen object

```
public GameObject titleScreen;

StartGame() {
    ... titleScreen.gameObject.SetActive(false); }
```

Step 6: Use a parameter to change difficulty

The difficulty buttons start the game, but they still don't change the game's difficulty. The last thing we have to do is actually make the difficulty buttons affect the rate that target objects spawn.

1. In DifficultyButton.cs, create a new **public int difficulty** variable, then in the Inspector, assign the **Easy** difficulty as 1, **Medium** as 2, and **Hard** as 3
2. Add an **int difficulty** parameter to the **StartGame()** function
3. In **StartGame()**, set **spawnRate /= difficulty;**
4. Fix the error in DifficultyButton.cs by passing the difficulty parameter to **StartGame(int difficulty)**

- **New Concept:**
/= operator

```
public int difficulty;

void SetDifficulty() {
    ... gameManager.startGame(difficulty); }

<----->
public void StartGame(int difficulty) {
    spawnRate /= difficulty; }
```

Lesson Recap

New Functionality

- Title screen that lets the user start the game
- Difficulty selection that affects spawn rate

New Concepts and Skills

- AddListener()
- Passing parameters between scripts
- Divide/Assign (/=) operator
- Grouping child objects

Next Lesson

- In our next lesson, we'll use buttons to really add some difficulty to our game