CODE NINJAS®

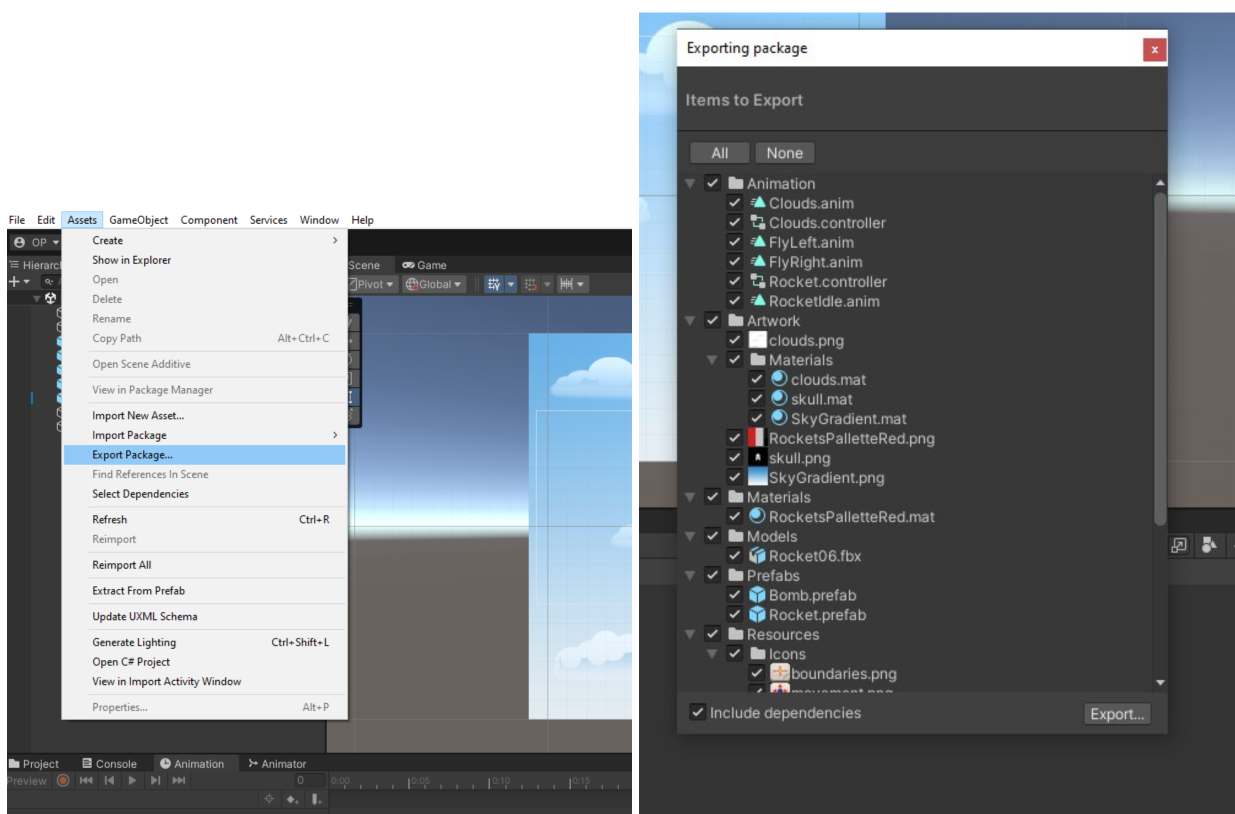
Bronze Belt Ninja Guide

Activity 09: Dropping Bombs Part 3

Activity 9: Dropping Bombs Part 3

Now that you've updated the objects in the game, you can add scoring and other elements of the User Interface. To do that, you'll need to make a few changes to the game itself so that it doesn't start until the player is ready for it to start.

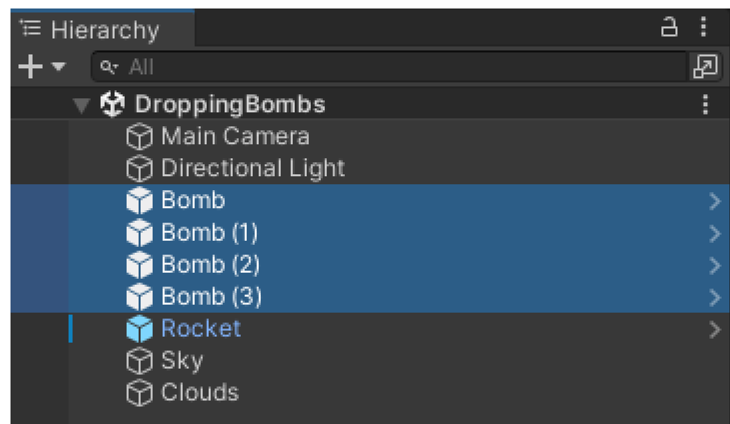
1 Before making additional changes to your project, let's make a backup of the entire game. In the **Projects** panel, make sure that the **Assets** folder is selected. Then click the **Assets** tab at the top of the screen and select **Export Package**. Make sure all assets are selected before clicking on the **Export** button. Give the package a name like **JS-DroppingBombsPart2**.



2

To set the game to start only when the player wants it to, we have to be able to control the objects in the game itself. That means having the game create and destroy the objects. Let's start with the bombs.

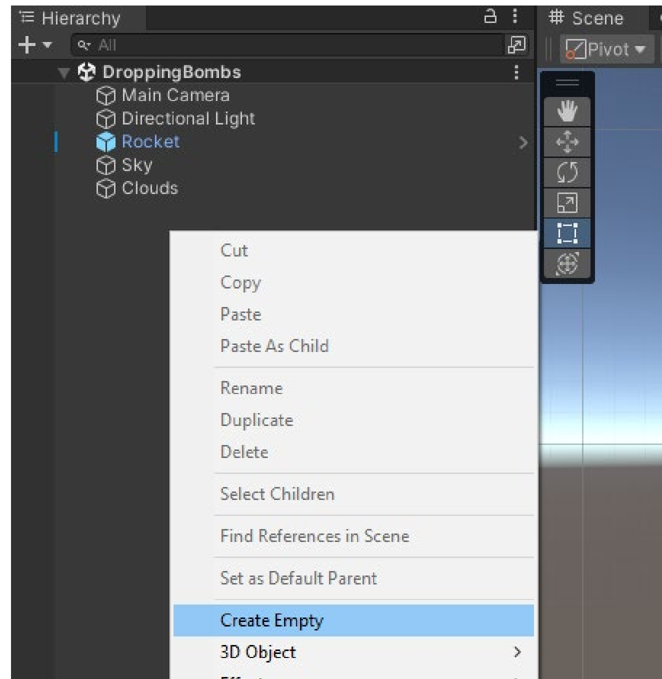
In the previous activity, the **Bomb GameObject** was made into a **Prefab**. We can delete any extra **Bomb** objects in the **Hierarchy** without affecting the object in the **Prefabs** folder. If you have any Bombs in the Hierarchy from testing, select all of the them and delete them.



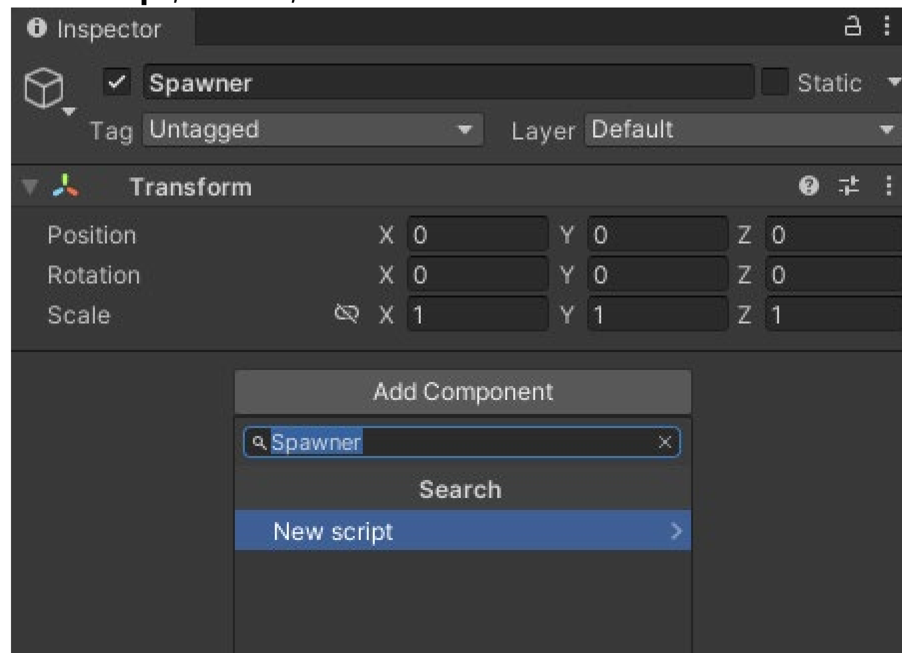
3

The dropping bombs no longer need to teleport back to the top of the scene. Open the **Prefabs** folder and select the **Bomb Prefab**. In the Inspector, find the Teleport script and remove it by clicking the three dots in the right corner and selecting **Remove Component**.

4 Now a **GameObject** can be created to automatically spawn the Bombs in the game. Right-click in the **Hierarchy** and select **Create Empty**. Rename this new empty object as ***Spawner***.



5 All the **Spawner GameObject** will do is hold the script that spawns the bombs. Click **Add Component** in the inspector panel and search for ***Spawner***. The script currently doesn't exist, so click **New Script**, **Create**, and **Add**.



Unity might place your new script in the **Assets** folder instead of the **Scripts** folder. If it does, just drag the script into the **Scripts** folder.

6 Double-click the **Spawner** script to open it up in the script editor. The next few steps will guide you through how to create this script and the image on step 10 will show you how it is meant to look. Try to follow the steps before looking at the image below.

7 First, we need a variable. Create a **Public** variable with the type **GameObject** and name it ***bombPrefab***.

8 The next variable is the default setting for how often the spawner makes a new object. This also needs to be **public**. The type is **float** and the name should be ***delay***. Set it to equal to **2.0f**.

9 We need a variable to let us know if the Spawner is active or not. Again, make this a **public** variable. The type is **bool** since it will only be true or false. Name it ***active*** and set it equal to **false**.

10 The last variable will be to let us set the range for a random number for the spawn delay. We could use two different variables, but it's easier to use a **public Vector2** variable. Call it ***delayRange*** and set it equal to a new **Vector2(1,2)** (The x will be the first part of our range and the y will be the second part).

```
1   using System.Collections;
2   using System.Collections.Generic;
3   using UnityEngine;
4
    Unity Script | 0 references
5   public class Spawner : MonoBehaviour
6   {
7       public GameObject bombPrefab;
8       public float delay = 2.0f;
9       public bool active = true;
10      public Vector2 delayRange = new Vector2(1, 2);
11
12      // Start is called before the first frame update
        Unity Message | 0 references
13      void Start()
14      {
```

11 When the script is started, choose a random number for the delay and have it run the function that spawns the objects. For the first, we'll call a function named **ResetDelay()** (with no parameters).

Then, call the second function using **StartCoroutine**. A ***coroutine*** is like a function that can pause execution and return control to Unity and then continue where it left off. This is very useful since we'll be constantly pausing the spawn process until the random delay has expired.

The **StartCoroutine** calls the **EnemyGenerator** function.

```
11
12      // Start is called before the first frame update
        Unity Message | 0 references
13      void Start()
14      {
15          ResetDelay();
16          StartCoroutine(EnemyGenerator());
17      }
```

At this point, you will have errors in your code, this is because we haven't made the functions yet

12 Unlike other functions, a coroutine *must* start with the ***IEnumerator*** data type. So instead of void, our function is **IEnumerator EnemyGenerator()**

(See if you can try to figure out what code you need from the text before looking at the below image)

The first line in the function is **yield return new WaitForSeconds(delay);**. As you might have guessed, this is the part of the coroutine that lets Unity do what it needs to do while waiting for the required length of time.

The next line is a conditional that checks if the **Spawner** is active or not. Remember, we don't want to start the game until the player is ready and this keeps anything from spawning until it is active.

Inside the conditional, we do two things. We use **Instantiate** to spawn our **bombPrefab** at a position of 0,0,0 (right in the middle) with the current rotation of the prefab. Once that is done, we call the **ResetDelay** function to choose a new random number for the delay.

After all of that, regardless of whether anything was spawned, we start the coroutine all over again.

```
15          ResetDelay();
16          StartCoroutine(EnemyGenerator());
17      }
18
        2 references
19      IEnumerator EnemyGenerator()
20      {
21          yield return new WaitForSeconds(delay);
22          if(active) {
23
24              Instantiate(bombPrefab, new Vector3(randomX, spawnY, 0), bombPrefab.transform.rotation);
25              ResetDelay();
26          }
27
28          StartCoroutine(EnemyGenerator());
29      }
30  }
31
```

Again, at this time, you will have an error, because we still need to make the ResetDelay Function

CODE NINJAS®

13 The **ResetDelay** function is just a single line setting **delay** to equal a **Random.Range** between the values of **delayRange.x** and **delayRange.y**.

```
31    2 references
      void ResetDelay()
32    {
33        delay = Random.Range(delayRange.x, delayRange.y);
34    }
35  }
36
```

14 There are still some errors to fix, but before that, we are going to fix a problem that may occur. Currently, if the game were to be run, the bombs would always spawn at (0,0,0). We are going to fix this first by using a private variable.

This variable is a **Vector2** that will be called ***screenBounds*** and hold the dimension of the screen.

```
7     public GameObject bombPrefab;
8     public float delay = 2.0f;
9     public bool active = true;
10    public Vector2 delayRange = new Vector2(1, 2);
11
12    private Vector2 screenBounds;
13
14    // Start is called before the first frame update
```

15 When you start the script, you will get the value for this new variable. **screenBounds** is from the **Camera.main.ScreenToWorldPoint** as a new **Vector3** with x as the **Screen.width**, y as the **Screen.height** and z as the **Camera.main.transform.position.z**.

```
14    // Start is called before the first frame update
      Unity Message | 0 references
15    void Start()
16    {
17        ResetDelay();
18        StartCoroutine(EnemyGenerator());
19
20        screenBounds = Camera.main.ScreenToWorldPoint(new Vector3(Screen.width, Screen.height, Camera.main.transform.position.z));
21    }
22
```

16 The center of the screen is 0,0 so the left side is negative x (-x) and the right side is positive x (+x). We want to spawn the bomb somewhere between those two points.

To do this, before we Instantiate our bomb, make a **float** variable called **randomX** that is a **Random.Range** between **-screenBounds.x** and **screenBounds.x**. This will give us a random number between the left and right side of our screen.

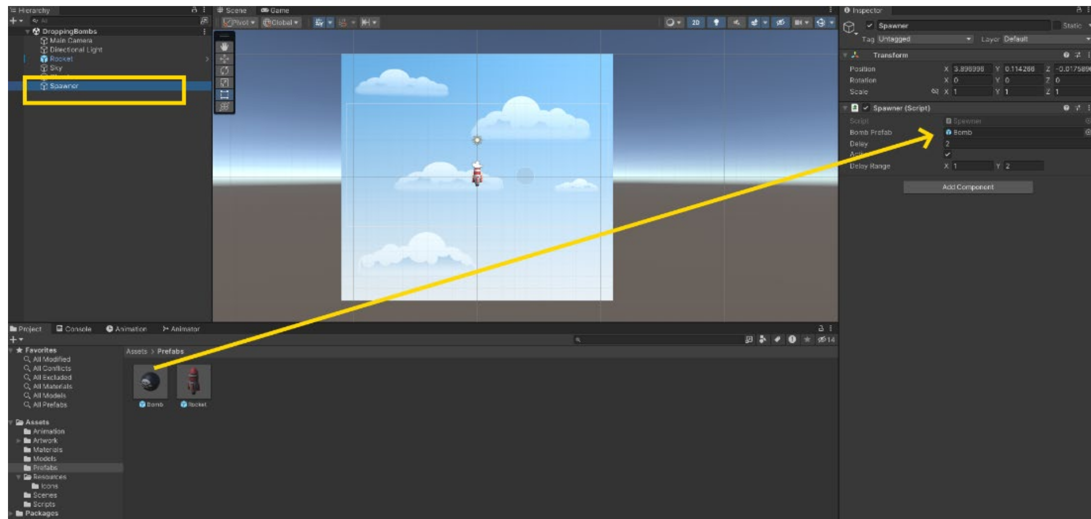
The Y position will always be the same, based on the **screenBounds.y + 1**

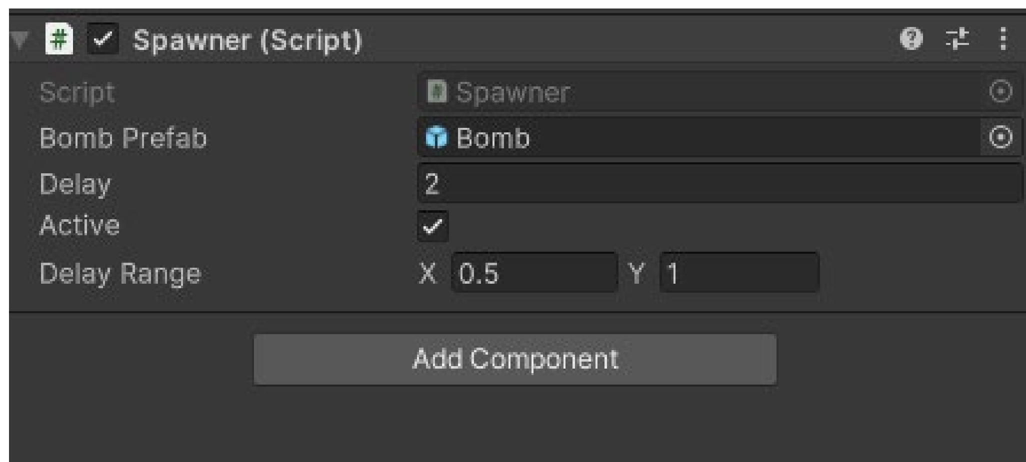
The **+1** is there to give a bit of space; you can try changing this number if you like.

These two numbers are passed to our **Instantiate** from earlier. If all is correct, the errors from earlier will be resolved.

```
   2 references
23 IEnumerator EnemyGenerator()
24 {
25     yield return new WaitForSeconds(delay);
26     if(active) {
27 
28         float randomX = Random.Range(-screenBounds.x, screenBounds.x);
29         float spawnY = screenBounds.y + 1;
30 
31         Instantiate(bombPrefab, new Vector3(randomX, spawnY, 0), bombPrefab.transform.rotation);
32         ResetDelay();
33     }
34 
35     StartCoroutine(EnemyGenerator());
36 }
37 
```

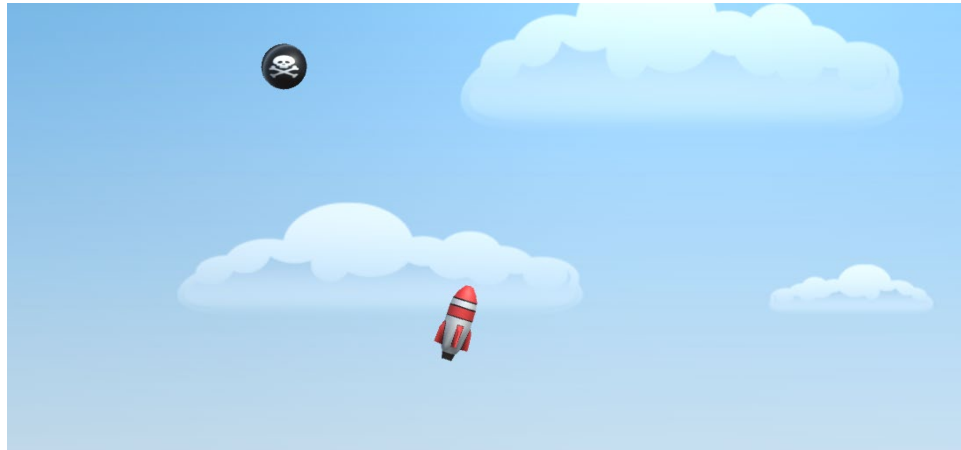
17 Save the **Spawner** script and return to Unity. Select the **Spawner Gameobject** in the **Hierarchy**. Search for the **Spawner** script in the inspector panel. The script needs to know what object to use for the **Bomb Prefab**. Open the **Prefabs** folder in the **Project** panel and drag the **Bomb** from the folder into the script as shown.





18 Now test the game by clicking the **Play** arrow above the Scene. You should see a new bomb between one and two seconds.

When you are done, stop the game by pressing the **Play** arrow again.

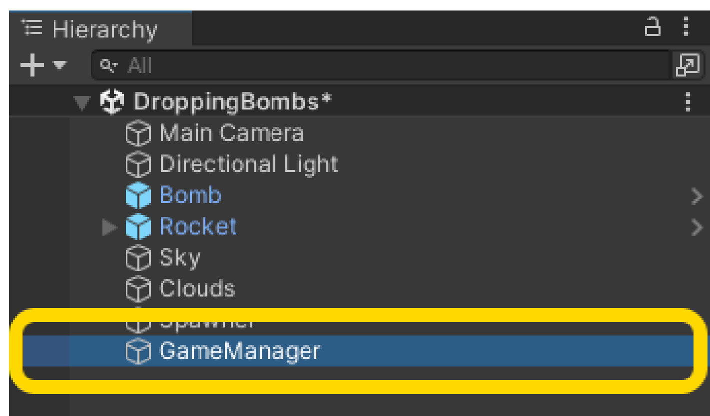


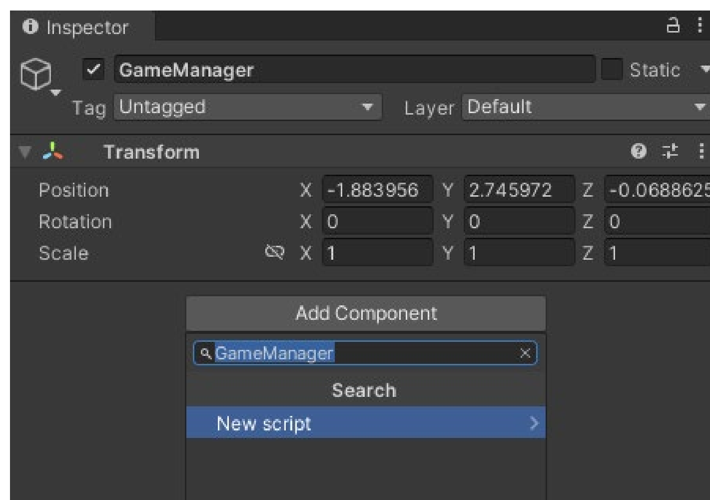
Pro Tip:

If you want to see more bombs in your scene, change the Delay Range in the Spawner script. Lower numbers mean a shorter delay, creating more bombs!

19 The **Spawner** allows you to make (and stop) bombs whenever you need to. Now there needs to be something to tell the **Spawner** when to turn it off and on. To do this, you will make a **GameManager**.

Just as you did with the **Spawner**, create an empty **Object** in the **Hierarchy** and name it ***GameManager***. In the **Scripts** folder, create a new **C#** script and also give it the name of ***GameManager***. Drag the new script onto the **GameManager GameObject**.





Why does the **GameManager** script look like it does? Unity has a special symbol reserved for these types of scripts. Other than the icon, it's exactly like all of the other scripts.

20 Open the **GameManager** script by double-clicking it. To communicate with the **Spawner**, you must first create a variable for it. Make it a **private** variable of type **Spawner** and give it the name ***spawner*** (all lower case).

```
Unity Script | 0 references
5   public class GameManager : MonoBehaviour
6   {
7
8       private Spawner spawner;
9
10      // Start is called before the first frame update
        Unity Message | 0 references
11      void Start()
12      {
```

21 We have yet to identify the spawner, which we can do when the script first awakes. Create a void function for **Awake**. Similar to **void Start**, **void Awake** is called when the game loads, but Awake is called before Start.

Inside the **Awake** function, we are going to find the **GameObject** called "**Spawner**" using the **GameObject.Find** function. However, since this finds a **GameObject** we will get an error since our variable is of the type **Spawner** and not GameObject. To fix this, we can add **GetComponent<Spawner>()** to the end of the line, to access the component on that object.

```
4
    Unity Script | 0 references
5   public class GameManager : MonoBehaviour
6   {
7       private Spawner spawner;
8
        Unity Message | 0 references
9       private void Awake()
10      {
11          spawner = GameObject.Find("Spawner").GetComponent<Spawner>();
12      }
13
```

22 Now that spawner is defined as the script component of the **Spawner GameObject**, you can access the functions and variables within. We want to turn the **Spawner** script off, so when the **GameManager** script starts, we can have the **active** variable of the **spawner** component set to false.

```
12      }
13
14      // Start is called before the first frame update
        Unity Message | 0 references
15      void Start()
16      {
17          spawner.active = false;
18      }
19
```

23 Lastly, there will need to be some way to turn the spawner script component back on. We need to listen for player input, which means we will use the **Update** function. Inside the function, you'll set up a conditional for input. If the user hits any key (or button), then you will set **spawner.active** to **true**. This can be accomplished with **Input.anyKeyDown**. So, when the user does anything, the spawner is activated.

```
19
20      // Update is called once per frame
        Unity Message | 0 references
21      void Update()
22      {
23          if (Input.anyKeyDown)
24          {
25              spawner.active = true;
26          }
27      }
28  }
29
```

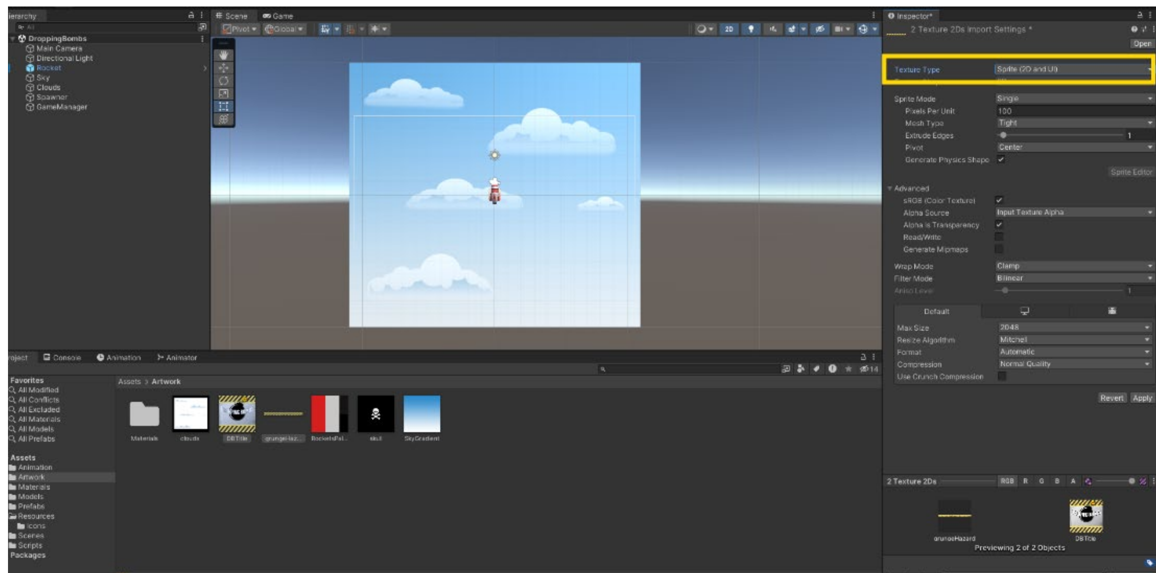
24 Save your script and return to Unity.

If you start the game now (go ahead, we'll wait), you'll see the rocket and the animations work, but there are no bombs until you hit any key (which means that the second that you use the arrow keys to move the rocket, the spawner becomes active). It would be useful to have a message to tell the player that they're supposed to press any key to start the game. This will be the first part of our **User Interface**.

25

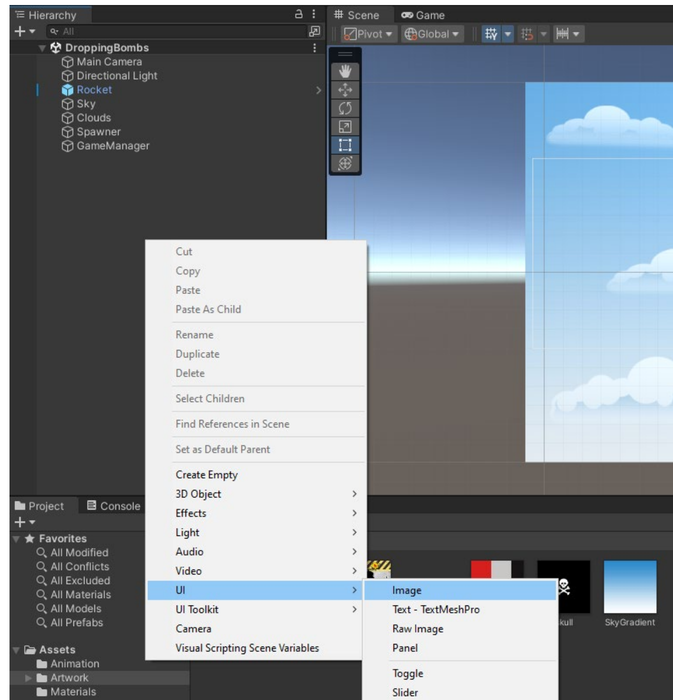
If you started the game, stop it before continuing to the next step.

Artwork for the User Interface has been prepared for you. Go to the **Artwork** folder and right-click inside the folder. Select **Import New Assets** and from the Bronze Belt assets folder, find and import **Activity 09 - DBTitle.png** and **Activity 09 - grungeHazard.png**. With both images selected, go to the **Inspector** panel and change **Texture Type** to **Sprite (2D and UI)**. Scroll down to the bottom of the Inspector and click **Apply**.



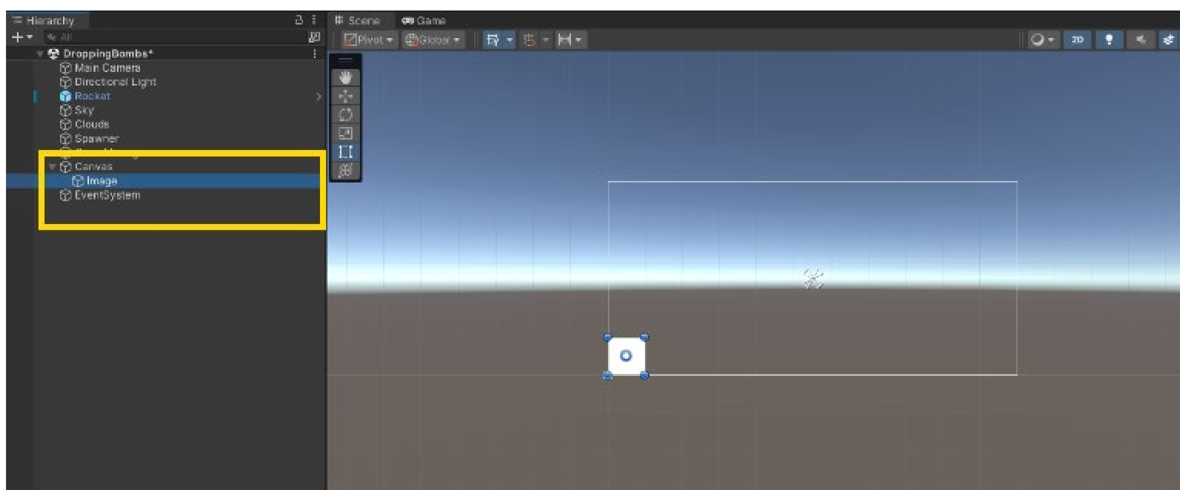
If at any time when you are making edits to the texture component and you forget to hit "Apply" a reminder pops up. Either apply the changes or revert them.

26 Right-click in the **Hierarchy** panel to bring up the **Create** menu. Select **UI**, then select **Image**.

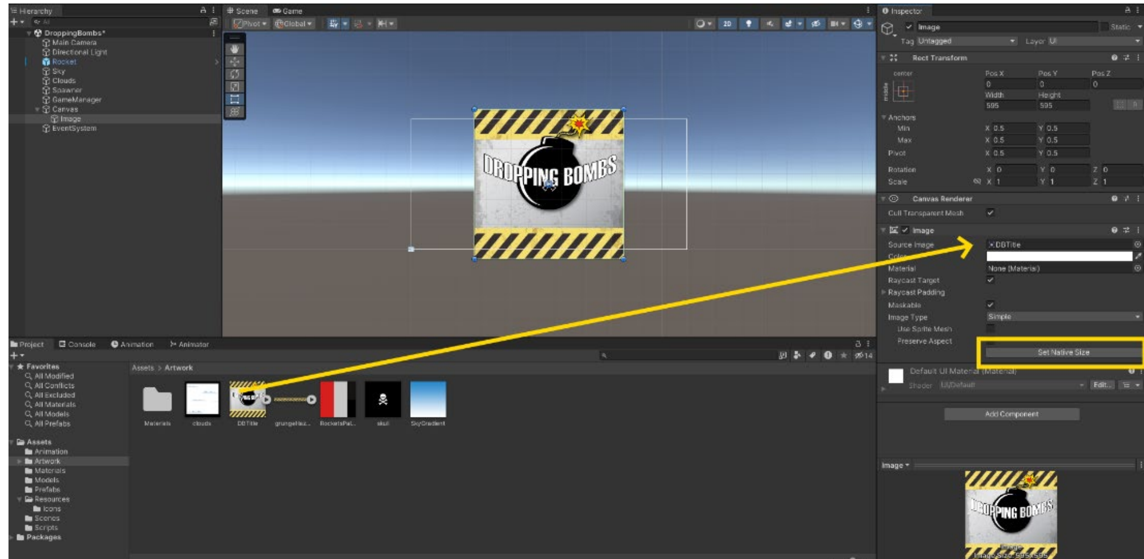


27 When you do this, a Canvas object is created to hold all the **UI** elements. Additionally, an **EventSystem** is created to support the **Canvas**. Without an **EventSystem**, you won't be able to interact with the **UI**.

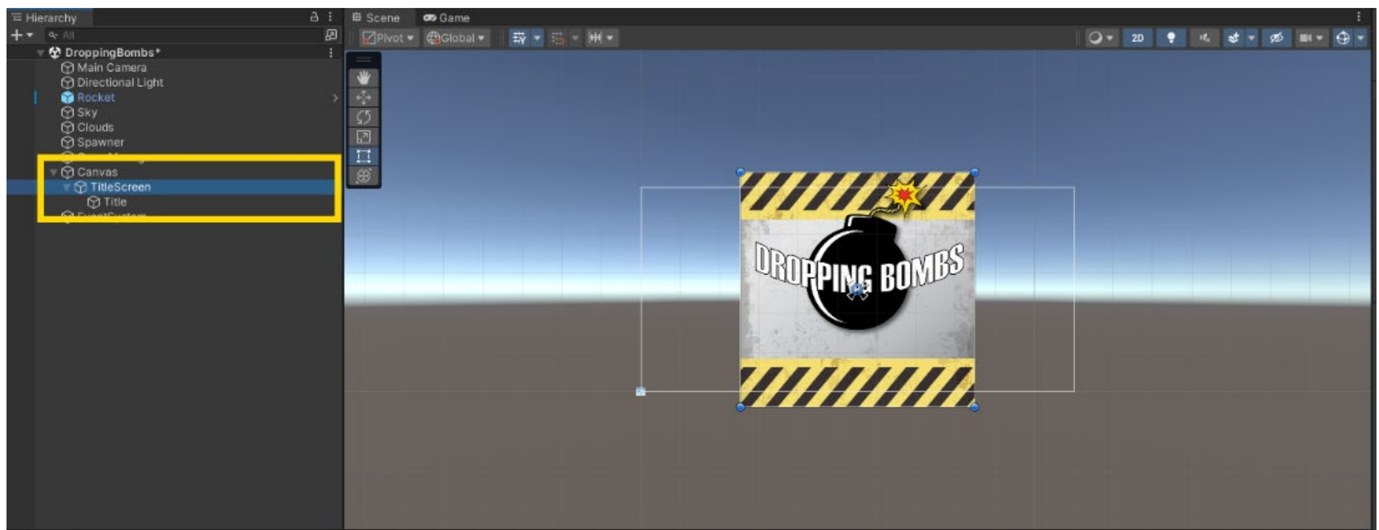
The **Canvas** is much larger than the camera view, so double click the newly created image in the Hierarchy to focus onto it.



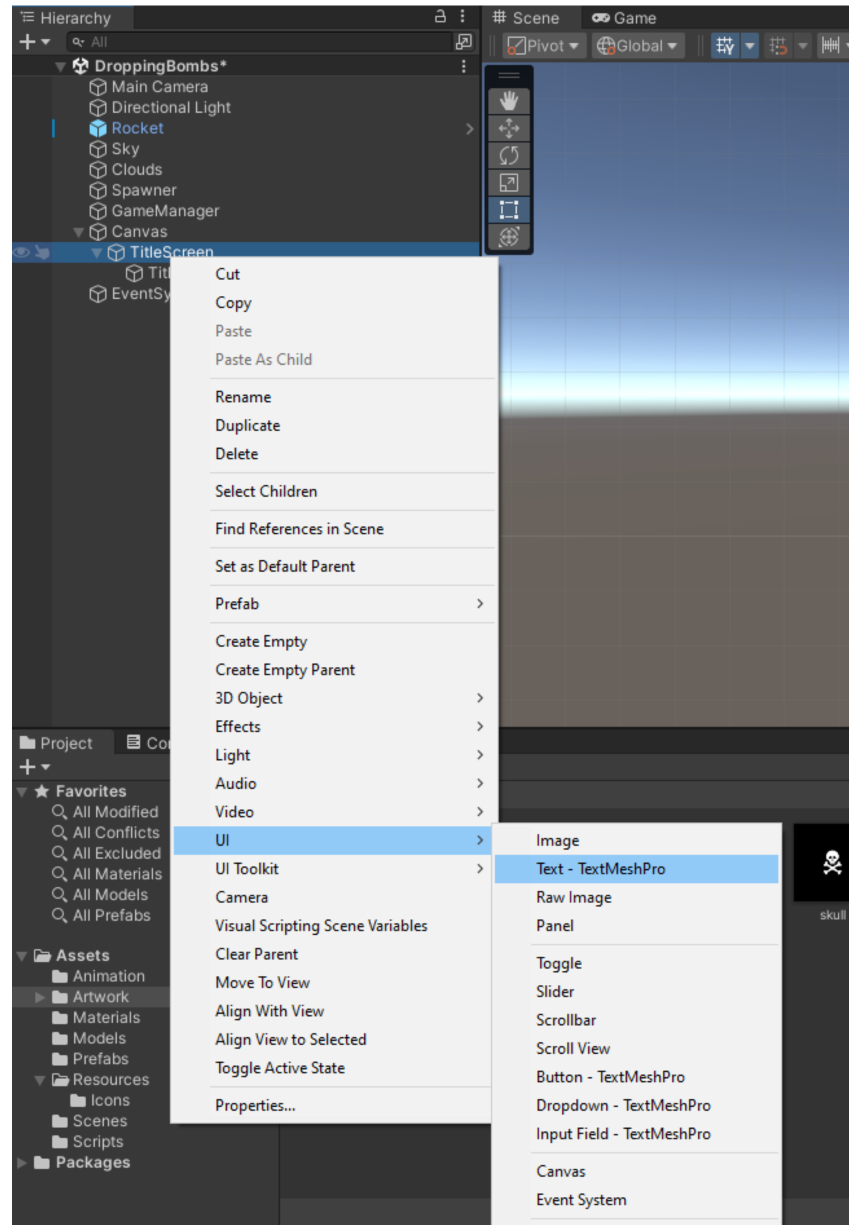
28 Change the name of the **Image** to ***Title,*** then click on the gear and select **Reset** to put the image to the center of the **Canvas**. Drag **DBTitle** into the slot for the **Source Image** and click **Set Native Size** to enlarge the image to the size of the graphic.



29 The title screen will have several objects, so let's create an **Empty Object** inside of the Canvas to hold everything and call it ***TitleScreen***. Drag the **Title** object so that it is inside of the **TitleScreen** object.

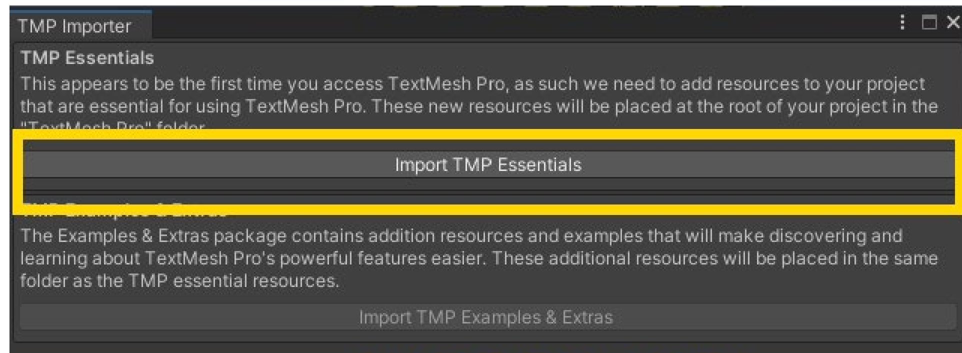


30 Next, let's add some text that tells the player how to start the game. Right-click on the **TitleScreen** object and go back to **UI**. We are going to add **Text – Text Mesh Pro**.



31

You may get a box that tells you to import Text Mesh Pro. If so, click **Import TMP Essentials**. This will install all the required TMP assets. This may take a few minutes.



32

With that installed, our text should appear in the middle. Let's stretch it and move it somewhere sensible. Inside the text, we can write our message. You can create something similar to the image below.

You may wish to tinker with the size and the color to make it look even better.

33 Now that the player knows that they need to press any key to start the game, we need to add code to hide the **TitleScreen**. Switch back to the **GameManager** script in the script editor.

Let's start by adding a **public** variable to hold the **TitleScreen Gameobject**.

```
     Unity Script | – references
5    public class GameManager : MonoBehaviour
6    {
7        private Spawner spawner;
8        public GameObject title;
         Unity Message | – references
9        private void Awake()
10       {
11           spawner = GameObject.Find("Spawner").GetComponent<Spawner>();
12       }
13
```

34 When the **spawner** is off, the **title** should be on. When the **spawner** is on, the **title** should be off. In the **Start** function, add a line for the **GameObject title.SetActive(true)**. In the conditional for **Input** in the **Update** function, add a line for the **GameObject title.SetActive(false)**. Save the script.

```
14       // Start is called before the first frame update
         Unity Message | 0 references
15       void Start()
16       {
17           spawner.active = false;
18           title.SetActive(true);
19       }
20
21       // Update is called once per frame
         Unity Message | 0 references
22       void Update()
23       {
24           if (Input.anyKeyDown)
25           {
26               spawner.active = true;
27               title.SetActive(false);
28           }
29       }
30   }
```

Pro Tip:

You might be wondering why the boolean for the **spawner** is set to equal true or false while the boolean for **title** uses **SetActive** with a parameter of true or false. In **Unity**, a **GameObject** (like **title**) can use **SetActive**. However, **spawner** is *not* a **GameObject**, it is a component inside of a **GameObject**. With a component, we have to work with the functions and variables inside the component.

35 Switch back to **Unity** and select **GameManger** in the **Hierarchy**. Drag **TitleScreen** from the **Hierarchy** into the slot for **Title** in the **GameManager** script component.

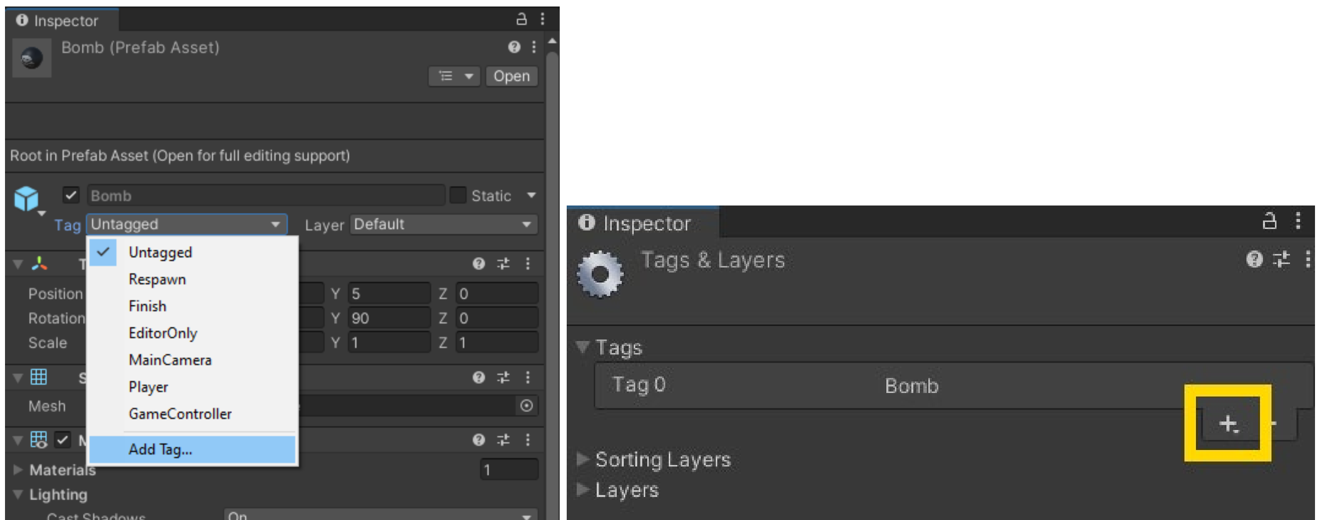
36 Now start the game by clicking the Play arrow above the Scene. The interface appears with a friendly message about how to start. When you press any key, the interface is hidden and the Bombs start spawning. So far, so good.

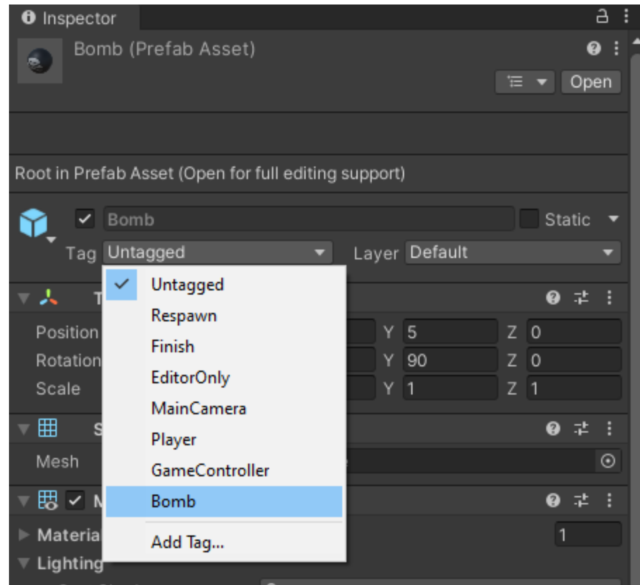
Stop the game by clicking the Play arrow a second time.

37

When the bombs spawn, they stay in the program's memory, with more bombs being added every second! We don't need the bombs after they've passed off the bottom of the screen, so we can get rid of them and use that for scoring, too.

Every time a new bomb is spawned, it gets a new name. We need another way to identify the spawned bombs. Select the **Bomb** prefab in the **Prefabs** folder. Click the **Tag** menu and select **Add Tag**. In the **Tags & Layers** menu, click the plus (**+**) symbol and name the new tag "**Bomb**". Select the **Bomb** prefab again and set the **Tag** to **Bomb**.





CODE NINJAS

38

Switch back to the **GameManager** in the script editor. To know when the Bomb has gone below the screen, you can use the same **screenBounds** that we used when spawning the Bomb.

Add a **private Vector2** variable with the name of "**screenBounds**". In the **Awake** function, add the same line that we used for the Spawner, setting **screenBounds** to equal **Camera.main.ScreenToWorldPoint** as shown below.

```
private Spawner spawner;
public GameObject title;
private Vector2 screenBounds;
Unity Message | 0 references
private void Awake()
{
    spawner = GameObject.Find("Spawner").GetComponent<Spawner>();
    screenBounds = Camera.main.ScreenToWorldPoint(new Vector3(Screen.width, Screen.height, Camera.main.transform.position.z));
}
```

39

Switch back to the **GameManager** in the script editor. Next, we'll need to find any object with the **Bomb** tag and remove it when it goes below a certain point. In the **Update** function, add a line at the bottom that sets
var nextBomb = GameObject.FindGameObjectsWithTag("Bomb").

```
22
23      // Update is called once per frame
        Unity Message | 0 references
24      void Update()
25      {
26          if (Input.anyKeyDown)
27          {
28              spawner.active = true;
29              title.SetActive(false);
30          }
31
32          var nextBomb = GameObject.FindGameObjectsWithTag("Bomb");
33
34          foreach (GameObject bombObject in nextBomb)
35          {
36
37
38          }
39      }
40  }
41
```

40

Now that we have all bombs, we need to loop through them. You may be familiar with a **for loop**, but instead we will use a **foreach loop** instead.
A **foreach loop** is a way to loop through every element in an array or a list. Specify what type it is, then what name you'll use to access that element (Similar to using '**i**' in a **for** loop), and what array to loop through. In our example, we are looking through all **GameObjects** (called ***bombObject***) in the **array nextBomb**.

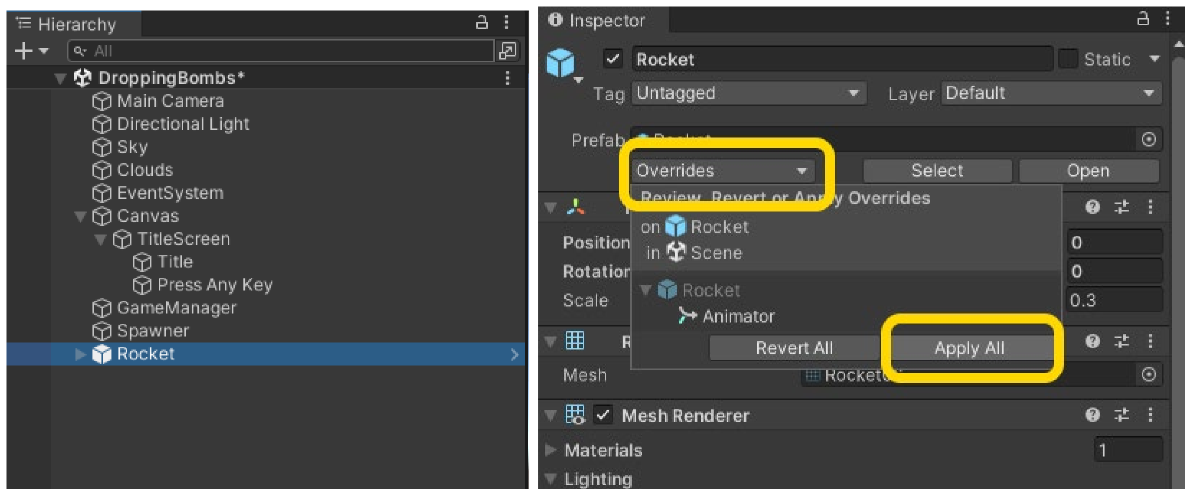
41 Inside the loop, we need a conditional that compares the Y value of the **bombObject** to see if it is off our screen. Any object that falls below our screen will be destroyed. Later, we'll use this for scoring as well.

```
22
23      // Update is called once per frame
        Unity Message | 0 references
24      void Update()
25      {
26          if (Input.anyKeyDown)
27          {
28              spawner.active = true;
29              title.SetActive(false);
30          }
31
32          var nextBomb = GameObject.FindGameObjectsWithTag("Bomb");
33
34          foreach (GameObject bombObject in nextBomb)
35          {
36              if (bombObject.transform.position.y < (-screenBounds.y - 12))
37              {
38                  Destroy(bombObject);
39              }
40
41          }
42      }
43  }
44
```

Don't forget to save your script.

42

In addition to the bombs, the rocket should be spawned and destroyed as well when the game starts and stops. However, the **Rocket** prefab was changed in the previous activity by adding the **Rocket Animate** script. When changes are made to a prefab in the **Hierarchy**, the prefab itself does not automatically get those changes. To make sure that our **Rocket** prefab is up to date, select the **Rocket** in the **Hierarchy**, and in the **Inspector** panel, click **Overrides** and select **Apply All** to make sure that the prefab matches the **GameObject** in the **Hierarchy**. After that, it will be safe to delete the **Rocket** from the **Hierarchy**.



43 Let's have the **GameManager** spawn the **Rocket** from the **Prefabs**. Switch back to the **GameManager** script in the script editor and add a **public** variable with the type of **GameObject** and name it ***playerPrefab***. You also need a **boolean** to know if the game has started or not. Add a **private** variable of the bool type and give it the name of "***gameStarted***" and set it to **false**.

```
7       private Spawner spawner;
8       public GameObject title;
9       private Vector2 screenBounds;
10
11      [Header("Player")]
12      public GameObject playerPrefab;
13      private GameObject player;
14      private bool gameStarted = false;
15      private void Awake()
16      {
17          spawner = GameObject.Find("Spawner").
18          screenBounds = Camera.main.ScreenToWo
19      }
20
```

44 Our **Update** function is getting a bit long and complicated. We are going to create a new function below Update and move some of the code inside to make it easier to read. This new function will be called ***ResetGame***.

45 Below the Update Function, add the **void ResetGame** function. Inside the newly created function, put the **spawner.active** and **title.SetActive** lines that used to be in the **Input.anyKeyDown** conditional above. Then **Instantiate playerPrefab** in the middle of the scene using the current **playerPrefab** rotation. Then set the **gameStarted** boolean to true so that new players don't keep spawning while the game is running. Save the script.

```
48
49
        0 references
50      void ResetGame()
51      {
52          spawner.active = true;
53          title.SetActive(false);
54
55          player = Instantiate(playerPrefab,new Vector3(0,0,0), playerPrefab.transform.rotation);
56          gameStarted = true;
57      }
58  }
```

46 Once we have created the function, we need to call it. We are going to call it when we press any key, but only if the game hasn't started.

```
28
29      // Update is called once per frame
        Unity Message | 0 references
30      void Update()
31      {
32          if (!gameStarted)
33          {
34              if (Input.anyKeyDown)
35              {
36                  ResetGame();
37              }
38          }
39
```

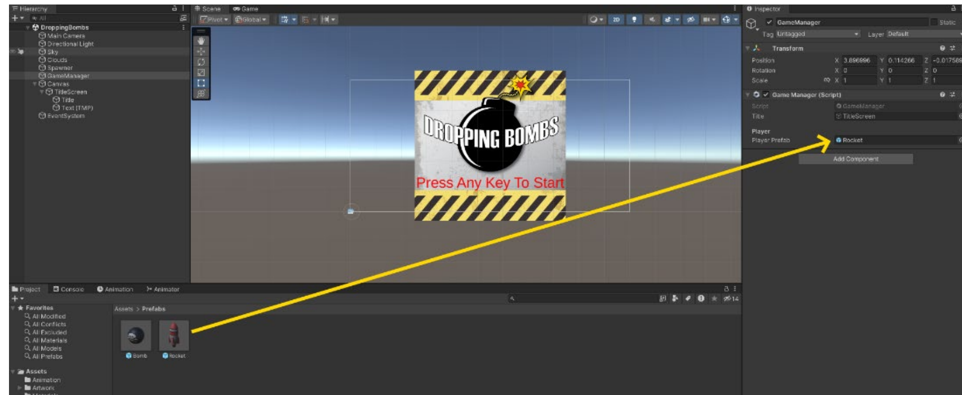
47 Currently, the **Rocket** has a **Reset** script that restarts everything when the **Rocket** collides with anything. In **Unity**, go into the **Scripts** folder and double-click **Reset** to edit it. There's only one line of code inside a single **OnCollision** function that has the **SceneManager.LoadScene(0)**. Replace that line with **Destroy(gameObject)** and save the script.

```
1   using System.Collections;
2   using System.Collections.Generic;
3   using UnityEngine;
4   using UnityEngine.SceneManagement;
5
    Unity Script (1 asset reference) | 0 references
6   public class Reset : MonoBehaviour
7   {
        Unity Message | 0 references
8       private void OnCollisionEnter(Collision collision)
9       {
10          Destroy(gameObject);
11      }
12  }
```

In **C#**, **gameObject** always refers to the **GameObject** that the script is attached to. It saves a bit of time.

48

Back in Unity, we need to assign the **Rocket** Prefab in the **GameManager**. We will do this now, so we don't forget later. The game won't work until we write some more code, so let's get back to coding!



49

While still in the script editor, switch back to the **GameManager** script. While the game is running, check to see if the **Rocket (player GameObject)** has been destroyed. We're already checking if **gameStarted** is false, so just check to see if the game is running by adding an **else** to the conditional. Inside the **else**, you'll add a conditional to check if the player object has been destroyed and if so, run the **OnPlayerKilled** function.

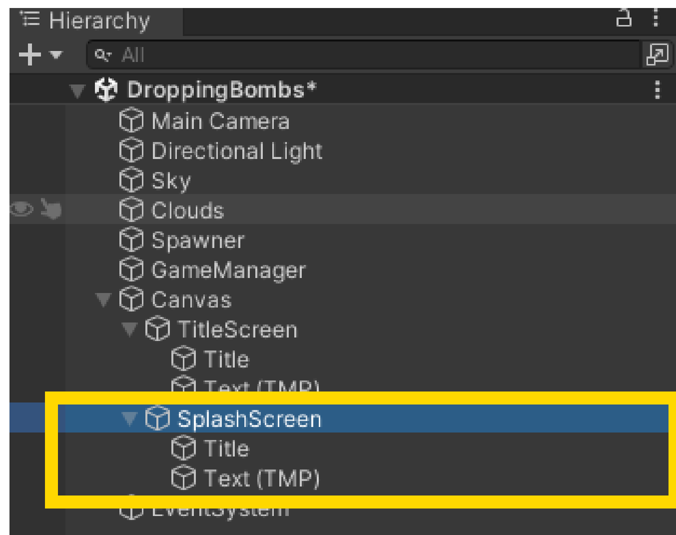
At this point, we will get an error, but we will fix that.

```
28
29      // Update is called once per frame
        Unity Message | 0 references
30      void Update()
31      {
32          if (!gameStarted)
33          {
34              if (Input.anyKeyDown)
35              {
36                  ResetGame();
37              }
38          }
39          else
40          {
41              if (!player)
42              {
43                  OnPlayerKilled();
44              }
45          }
46
```

50 At the end of the script, let's add the **OnPlayerKilled** function. When the player is killed, the **spawner** should stop. Set **gameStarted** to **false**. Finally, you'll want to show the **splash** screen (which we haven't made yet, so you will get an error).

```
61
    1 reference
62  void OnPlayerKilled()
63  {
64      spawner.active = false;
65      gameStarted = false;
66
67      splash.SetActive(true);
68  }
69
```

51 Switch back to Unity to make a splash screen. To save time, use **Ctrl+D** to make a copy of the title screen and rename it **SplashScreen**. Then, adjust the elements within it as follows:



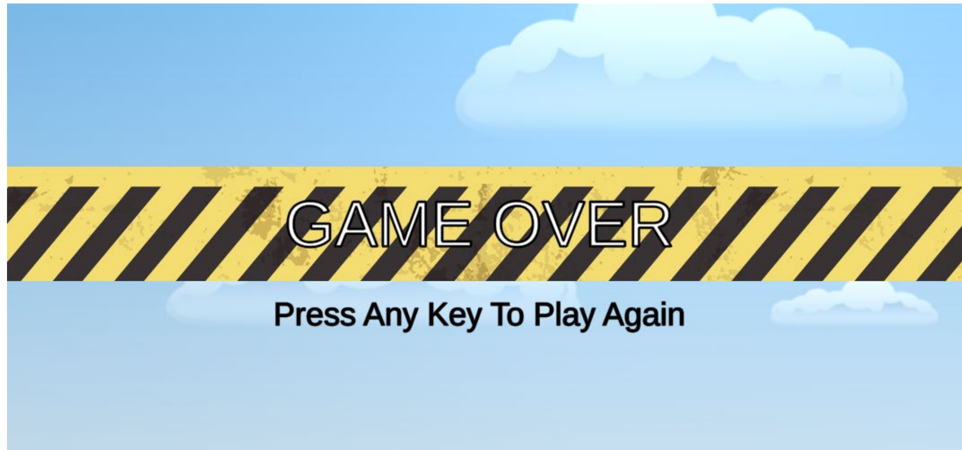
52 Change the **Title** image to **Background** and replace the **DBTitle** image with **grungeHazard**. Click the **Set Native Size** button.

53 The Text object gets renamed as ***Play Again*** and the text is altered to ***Press Any Key to Play Again***. Change the color to black and move it up so that it's underneath the banner.

54

Use **Ctrl+D** to copy the Text object and make this into a "**Game Over**" object. Change the text to ***GAME OVER***, increase the font size to **72** and make sure that this is white with a black outline. Use the **Rect Tool** to resize and adjust the text box. Feel free to change any of the fonts.

At this point, you can change how the Game Over screen looks. Below is an example of what it could look like.



55

Switch back to the **GameManager** script in the Script editor. Make a copy of **public GameObject title** and change the variable name to ***splash***.

```
Unity Script (1 asset reference) | 0 references
6   public class GameManager : MonoBehaviour
7   {
8       private Spawner spawner;
9       public GameObject title;
10      private Vector2 screenBounds;
11
12      public GameObject splash;
13
14      [Header("Player")]
15      public GameObject playerPrefab;
16      private GameObject player;
17      private bool gameStarted = false;
```

56

In the **Start** function, copy **title.SetActive(true)** and change it to ***splash.SetActive(false)***.

```
24
25      // Start is called before the first frame update
        Unity Message | 0 references
26      void Start()
27      {
28          spawner.active = false;
29          title.SetActive(true);
30          splash.SetActive(false);
31      }
32
33      // Update is called once per frame
        Unity Message | 0 references
34      void Update()
35      {
```

57

In the **ResetGame** function, copy **title.SetActive(false)** and rename it to ***splash.SetActive(false)***. Save your script.

```
72
        1 reference
73      void ResetGame()
74      {
75          spawner.active = true;
76          title.SetActive(false);
77
78          splash.SetActive(false);
79
80          player = Instantiate(playerPrefab,new Vector3(0,0,0), playerPrefab.transform.rotation);
81          gameStarted = true;
82      }
83  }
84
```

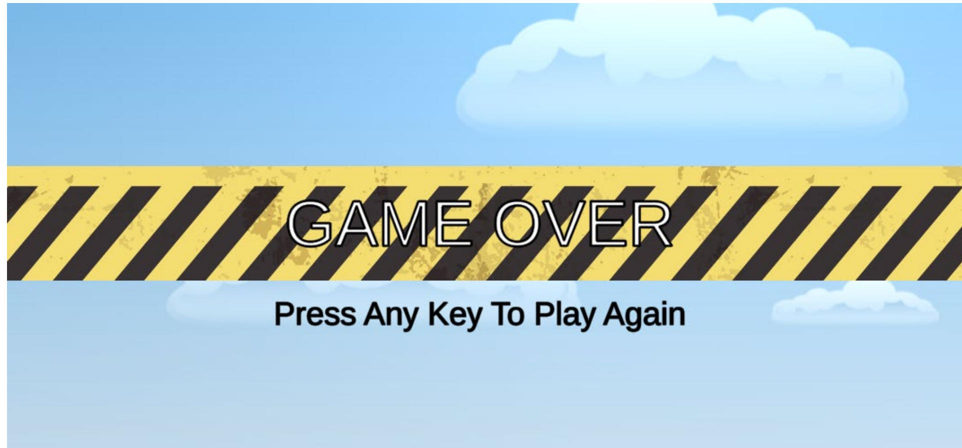
58

Switch back to Unity. With the **GameManager** object selected, make sure that the objects for **SplashScreen** and **Rocket** are in the appropriate slots in the **GameManager** script in the **Inspector** panel. Save your project.

59

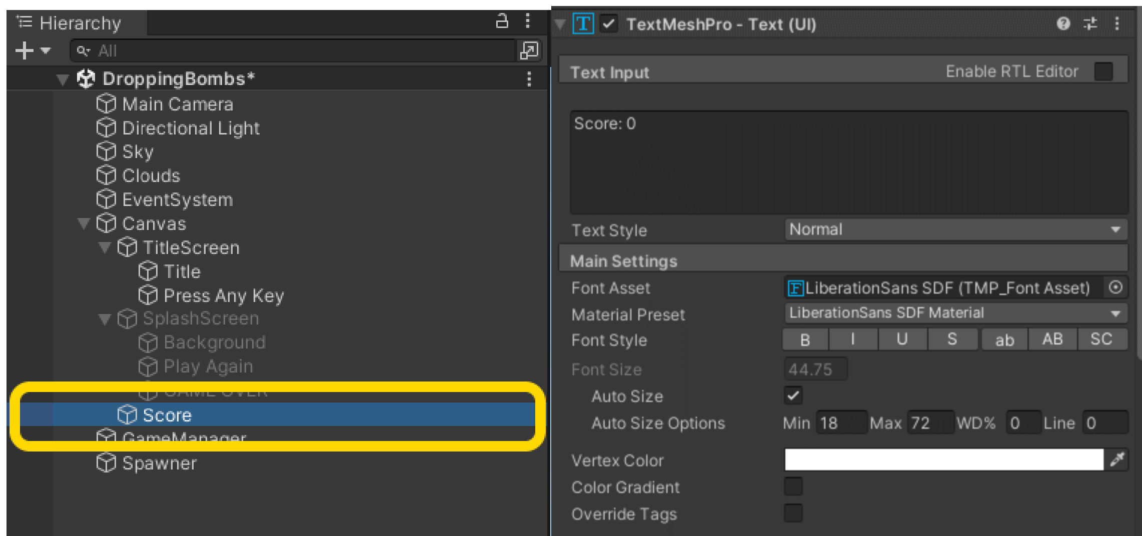
Play your game by clicking the **Play** arrow above the **Scene**. When you lose, the splash screen should appear and give you a chance to play again. All that's left to do is the score.

Stop your game by clicking the **Play** arrow again.



60

With the **Canvas** Selected, right click on it and add **Text Mesh Pro Text**. Use the **Move Tool** to move the object to the upper left of the canvas. Change the text to the upper left of the Canvas and change it to say ***Score: 0***. You can resize and change the font style if needed.



61

We are going to update our score text to display the number of bombs that have been deleted. When a bomb gets removed, we get a point.

First, let's go back to our **GameManager** script and add the needed libraries at the top for **Text Mesh Pro**.

```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4  using TMPro;
5  
   Unity Script (1 asset reference) | 0 references
6  public class GameManager : MonoBehaviour
7  {
8      private Spawner spawner;
9      public GameObject title;
```

62

Next, let's add some variables for our score. We want a **public TMP_Text** that is the score text component.

Add a **public** integer that says how many points we gain each time, and a **private** integer that holds our score.

```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4  using TMPro;
5  
   Unity Script (1 asset reference) | 0 references
6  public class GameManager : MonoBehaviour
7  {
8      private Spawner spawner;
9      public GameObject title;
10     private Vector2 screenBounds;
11 
12     public GameObject splash;
13 
14     [Header("Player")]
15     public GameObject playerPrefab;
16     private GameObject player;
17     private bool gameStarted = false;
18 
19     [Header("Score")]
20     public TMP_Text scoreText;
21     public int pointsWorth = 1;
22     private int score;
23 
       Unity Message | 0 references
```

63 Before our game starts, we want to **disable** the **TMP_Text** so it only appears when needed. Inside the **Awake** function, add a line to set **scoreText.enabled** to **false**.

```
22      private int score;
23
        Unity Message | 0 references
24      private void Awake()
25      {
26          spawner = GameObject.Find("Spawner").GetComponent<Spawner>();
27          screenBounds = Camera.main.ScreenToWorldPoint(new Vector3(Screen.width, Screen.height, Camera.main.transform.position.z));
28          scoreText.enabled = false;
29      }
30
31      // Start is called before the first frame update
```

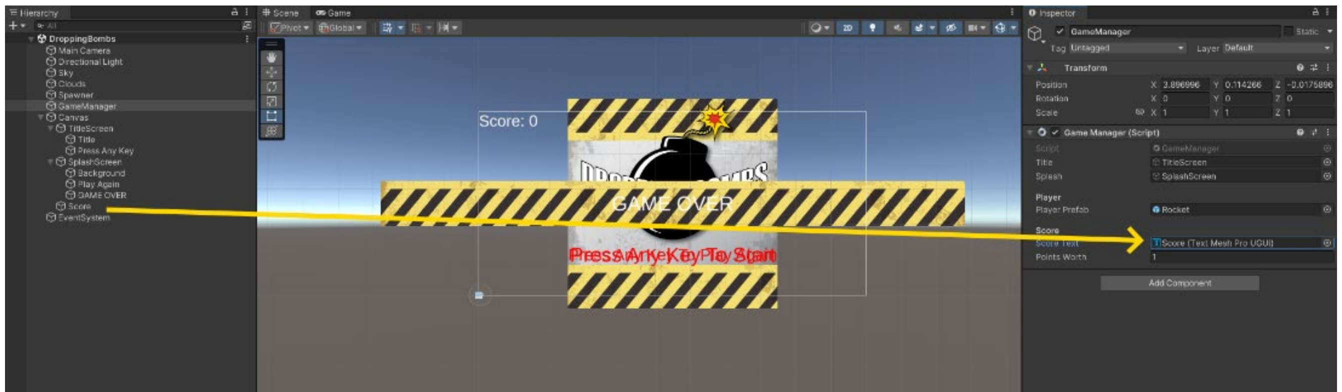
64 Inside the **restart** function, we are going to add a line that **resets** the score when the game restarts. Set **scoreText.enabled = true** and reset the score integer to **0**.

```
        1 reference
114     void ResetGame()
115     {
116         spawner.active = true;
117         title.SetActive(false);
118
119         splash.SetActive(false);
120
121         scoreText.enabled = true;
122         score = 0;
123
124         scoreText.text = "Score: " + score.ToString();
125
126
127
128
129         player = Instantiate(playerPrefab,new Vector3(0,0,0), playerPrefab.transform.rotation);
130         gameStarted = true;
131     }
132 }
133
```

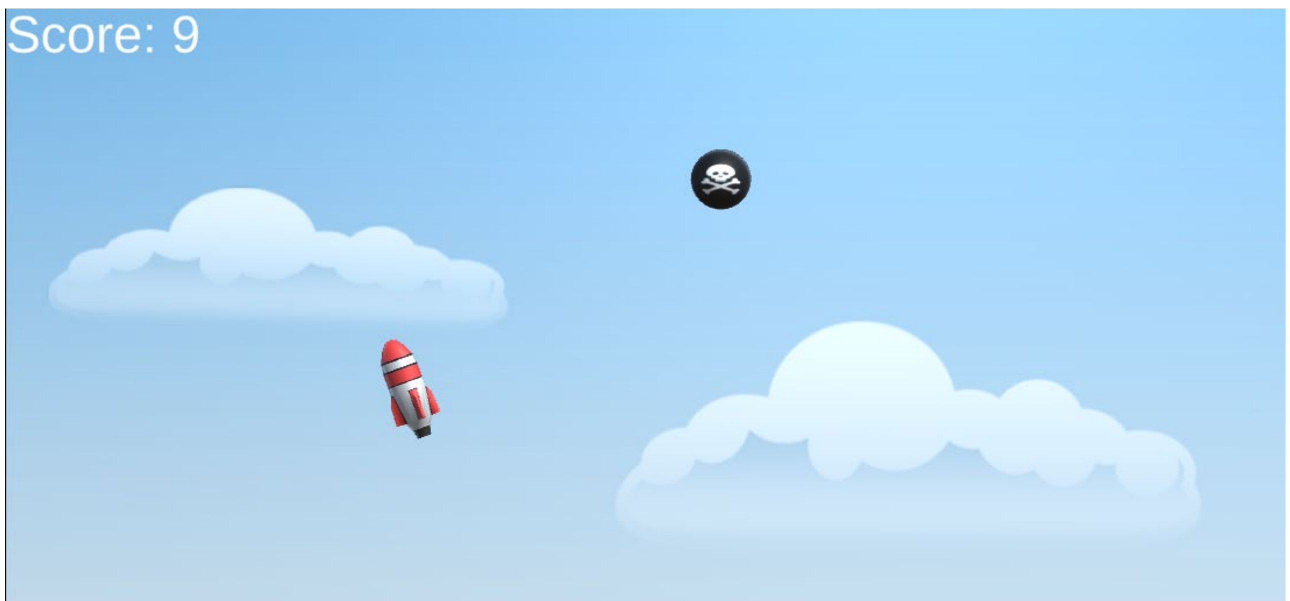
65 Finally, when our bomb gets removed for going too low, we want to gain a point. However, this will cause a problem if we lose, and the few remaining bombs get removed. To fix this, we can add a simple **if** statement to make sure that the game has **started**, and we also need to update our text to display the new value.

```
57
58
59          var nextBomb = GameObject.FindGameObjectsWithTag("Bomb");
60
61          foreach (GameObject bombObject in nextBomb)
62          {
63              if (bombObject.transform.position.y < (-screenBounds.y - 12))
64              {
65                  if (gameStarted)
66                  {
67                      score += pointsWorth;
68                      scoreText.text = "Score: " + score.ToString();
69                  }
70
71                  Destroy(bombObject);
72              }
73
74          }
75      }
76
```

66 Almost done, let's head back into **Unity** and link our text to the **TMP** slot in the script. **Save** your project.



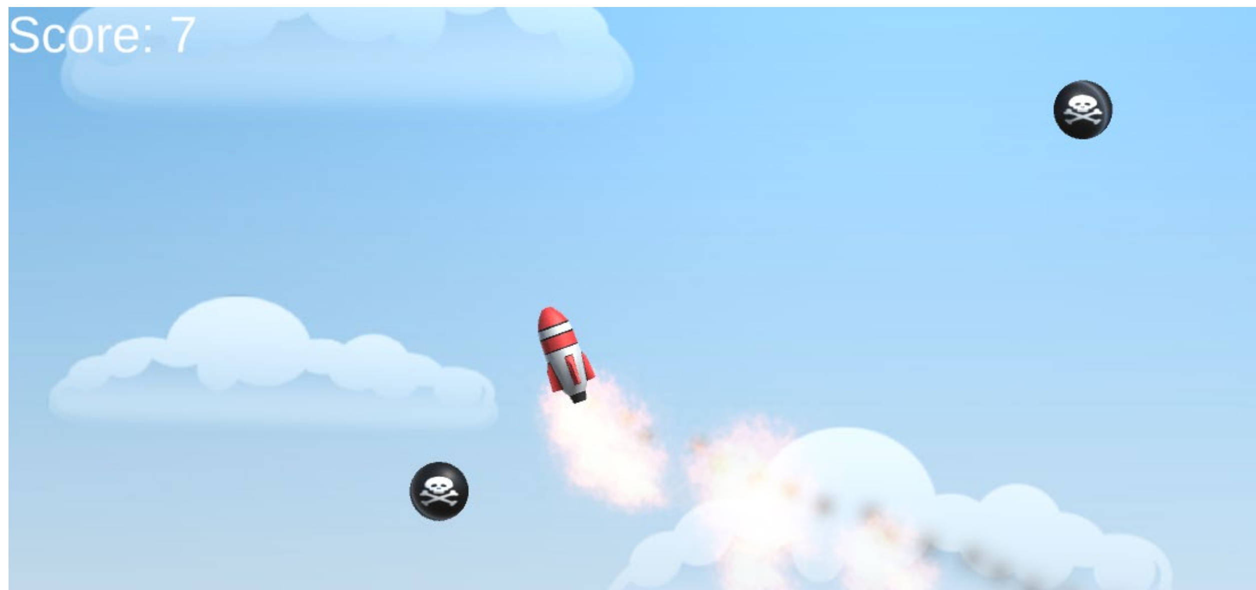
67 Now play your game. In the next section, there will be steps adding some more polish to finish the game!



Particle Systems and Player Preferences

Particle systems are great ways to simulate fire, smoke, water and other "fluid" visual effects. As the name implies, the system is making something out of a collection of smaller pieces and is ideal for explosions, fireworks, and even feedback for picking up power-ups in a game.

As with the other sections in this book, we're going to cover some of the essential information that you need to know to use these systems. The systems will be covered in more detail in future books.



The Joy of Particle Systems

In Unity, most objects are represented as "solids," rigid and well-defined. However, a solid is only one possible state of matter - liquids and gases are also commonly encountered in the world. Since liquids and gases are less rigid and well-defined than solids, a different system is required to simulate them in Unity. This section introduces the Unity particle system and some of the ways that it can be used.

What is a Particle System?

A particle system creates and displays a collection of simple images or objects as part of a whole. Each particle has its own behavior, but when viewed with other particles, it creates the illusion of smoke, fire, or many other possible effects.

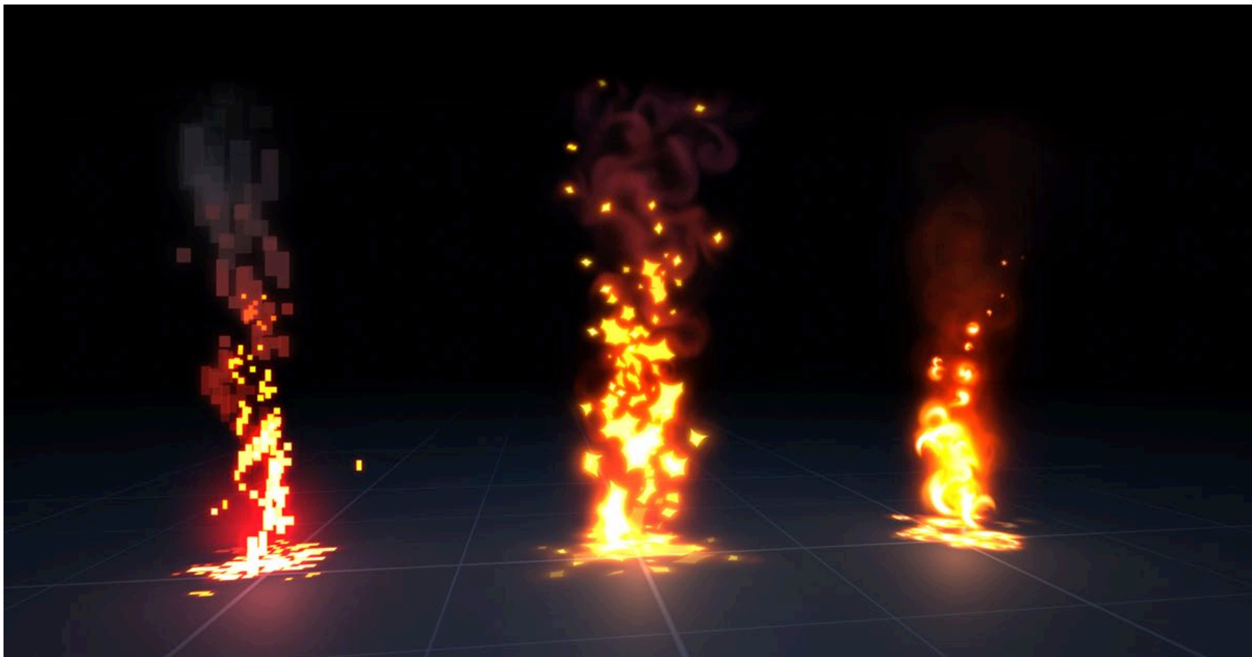


Image Source: https://assetstore.unity.com/packages/vfx/particles/fire-explosions/stylized-fire-fx-i-67379

Particles and the Environment

Image Source: https://80.lv/articles/creating-projectiles-in-unity/

Particles can be set to interact with (or being interacted by) the world in which they are set, or they can ignore it entirely. Particles can bounce off solids, be affected by wind and/or gravity, and can even act as triggers for other events.

Using Particle Systems

Since particle systems can be used to simulate so many things, both real and imagined, they have a lot of settings and sometimes you will find yourself making subtle changes to get the look just right. Even so, a particle system is just like any other GameObject in Unity and can exist on their own or be attached to other objects and can even be controlled by scripts. This section will introduce you to some of the most essential aspects of using particle systems.