17

Advanced Topics - Gizmos, Automated Testing, and More

This chapter will put together three sets of advanced recipes:

* Gizmos
* Automated testing
* An introduction to Unity Python

**Gizmos** facilitate Unity editor customization. Gizmos are visual aids for game designers that are provided in the Scene window. They can be useful as setup aids (to help us know what we are doing) or for debugging (understanding why objects aren't behaving as expected).

Gizmos are not drawn through Editor scripts, but as part of the Monobehaviour class, so they only work for GameObjects in the current scene. Gizmo drawing is usually performed in two methods:

* OnDrawGizmos(): This is executed every frame or editor window repaint, for every GameObject in the Hierarchy window.
* OnDrawGizmosSelect(): This is executed every frame, for just the/those GameObject(s) that are currently selected in the Hierarchy window.

Gizmo graphical drawing makes it simple to draw lines, cubes, and spheres. More complex shapes can also be drawn with meshes, and you can also display 2D image icons (located in the **Project** folder: Assets | Gizmos).

Several recipes in this chapter will illustrate how Gizmos can be useful. Often, new GameObjects created from editor extensions will have helpful Gizmos associated with them.

**Automated Testing** offering a way to formalise and structure how we test and develop our code. For a very simple computer program, we can write code, run it, enter a variety of valid and invalid data, and see whether the program behaves as we expect it to. This is known as a code-then-test approach. However, this approach has several significant weaknesses:

* Each time we change the code, as well as run new tests relating to the code we are improving, we have to run all the old tests to ensure that no unexpected modified behaviors have been introduced (in other words, our new code has not **broken** another part of our program)
* Running tests manually is time-consuming.
* We are relying on a human to rerun the test each time. However, this test may be run using different data, some data may be omitted, or different team members may take a different approach to run tests.

Therefore, even for simple programs (and most are not simple), some kind of fast, automated testing system makes a lot of sense.

There is an approach to software development called **test-driven development** (**TDD**), whereby code is **only** written until all tests pass. So, if we want to add or improve the behavior of our game program, we must specify what we want in terms of tests, and then the programmers write code to pass the tests. This avoids a situation whereby programmers write code and features that are not needed, spend time over-optimizing things that would have been fine, and so on. This means that the game development team directs its work toward agreed goals understood by all since they have been specified as tests.

The following diagram illustrates the basic TDD in that we only write code until all tests pass. Then, it's time to write more tests:

A purple arrow pointing to a rectangle

Description automatically generated with low confidence

Figure 13.1 – Test then code to the test

17\_10\_tdd\_diagram.png

Another way that TDD is often summarized is as red-green-refactor:

* **Red:** We write a new test for which code is needed, so initially, our test fails (in other words, we write a test for the new feature/improved behavior we wish to add to our system).
* **Green:** We write code that passes the new test (and all the existing ones).
* **Refactor:** We (may) choose to improve the code (and ensure that the improved code passes all the tests).

Two kinds of software tests exist, as follows:

* **Unit tests**
* **Integration tests**

A **unit test** tests a "unit" of code, which can be a single method, but which may include some other computer work being executed between the method being tested and the end result(s) being checked.

"A unit test is a piece of code that invokes a unit of work and checks one specific end result of that unit of work. If the assumptions on the end result turn out to be wrong, the unit test has failed*."  
                               — Roy Oshergrove (p. 5,*The Art of Unit Testing*(Second edition)*

Unit tests should be as follows:

* Automated (runnable at the "push of a button")
* Fast
* Easy to implement
* Easy to read
* Executed in isolation (tests should be independent of one another)
* Assessed as either having passed or failed
* Relevant tomorrow
* Consistent (the same results each time!)
* Able to easily pinpoint what was at fault for each test that fails

Most computer languages have a xUnit unit testing system available; for example:

* **C#**: NUnit
* **Java**: JUnit
* **PHP**: PHPUnit

Unity offers an easy way to write and execute NUnit tests in its editor (and at the command line).

Typically, each unit test will be written in three sections, like so:

* **Arrange**: Set any initial values needed (sometimes, we are just giving a value to a variable to improve code readability)
* **Act**: Invoke some code (and, if appropriate, store the results)
* **Assert**: Make assertions for what should be true about the code that's been invoked (and any stored results)

Observe that the naming of a unit test method (by convention) is quite verbose – it is made up of lots of words that describe what it does. For example, you might have a unit test method named TestHealthNotGoAboveOne(). The idea is that if a test fails, the name of the test should give a programmer a very good idea of what behavior is being tested and, therefore, how to quickly establish whether the test is correct, If so, it tells you where to look in your program code for what was being tested. Another part of the convention of naming unit tests is that numerals are not used, just words, so we write "one," "two," and so on, in the name of the test method.

An **integration test**(PlayMode tests in Unity) involves checking the behavior of interacting software components, for example, ones that are real time, or a real filesystem, or that communicate with the web or other applications running on the computer. Integration tests are usually not as fast as unit tests, and may not produce consistent results (since the components may interact in different ways at different times).

Both unit and integration yests are important, but they are different and should be treated differently.

Unity offers **PlayMode** testing, allowing integration testing as Unity scenes execute with testing code in them.

**Unity Python** is a package published by Unity Technologies. It allows Python code to be executed as part of a Unity project. The final recipe in this chapter will help you install and begin using this scripting tool in a Unity project.

In this chapter, we will cover the following recipes:

1. Using Gizmo to show the currently selected object in the Scene panel
2. Creating an editor snap-to-grid drawn by a Gizmo
3. Generating and running a default test script class
4. A simple unit test
5. Parameterizing tests with a data provider
6. Unit testing a simple health script class
7. Creating and executing a unit test in PlayMode
8. PlayMode testing a door animation
9. PlayMode and unit testing a player health bar with events, logging, and exceptions
10. Reporting Code Coverage testing
11. Running simple Python scripts inside Unity

# Technical requirements

To complete the recipes in this chapter, you will need Unity 2023.2 or later, plus one of the following:

* Microsoft Windows 10 (64-bit)/GPU: DX10, DX11, and DX12-capable
* macOS GPU Metal-capable Intel or AMD
  + Mojave 10.14+ / Intel x64 with SSE2 instruction set support
  + Big Sur 11.0 / Apple Silicon M1 or later
* Linux Ubuntu 20.04 or Ubuntu 18.04 / Gnome desktop running on X11 / GPU: OpenGL 3.2+ or Vulkan-capable Nvidia or AMD

For each chapter, there is a folder that contains the asset files you will need in this book's GitHub repository at: <https://github.com/PacktPublishing/Unity-2023-Cookbook> .

# Using Gizmo to show the currently selected object in the Scene panel

Gizmos are visual aids that are provided to game designers in the **Scene** panel. In this recipe, we'll highlight the GameObject that is currently selected in the **Hierarchy** window in the **Scene** panel.

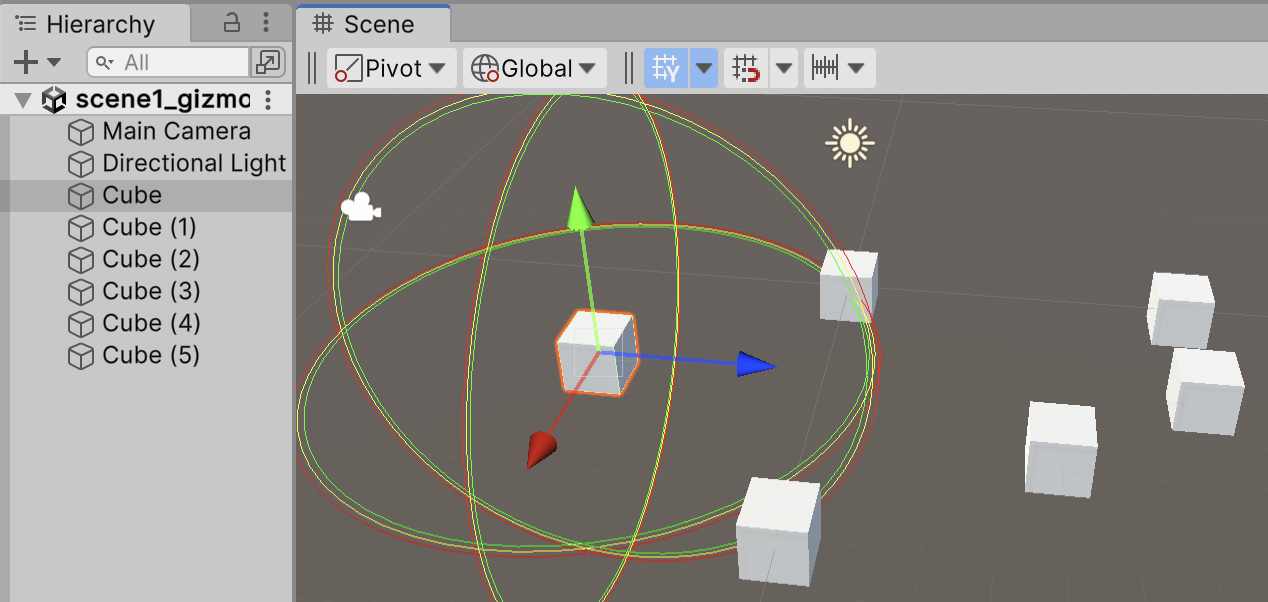


Figure 17.2 – Wireframe spheres around the selected GameObject

17\_02\_highlighed\_cube.png

## How to do it...

To create a Gizmo to show the selected object in the Scene panel, follow these steps:

1. Create a new Unity 3D project.
2. Create a 3D Cube by going to **Create | 3D Object | Cube**.
3. Create a C# script class called GizmoHighlightSelected and add an instance object as a component to the 3D Cube:

using UnityEngine;  
   
 public class GizmoHighlightSelected : MonoBehaviour {  
 public float radius = 5.0f;  
   
 void OnDrawGizmosSelected() {  
 Gizmos.color = Color.red;  
 Gizmos.DrawWireSphere(transform.position, radius);  
   
 Gizmos.color = Color.yellow;  
 Gizmos.DrawWireSphere(transform.position, radius - 0.1f);  
   
 Gizmos.color = Color.green;  
 Gizmos.DrawWireSphere(transform.position, radius - 0.2f);  
 }  
 }

1. Make several duplicates of the 3D Cube, distributing them randomly around the scene.
2. When you select one cube in the **Hierarchy** panel, you should see three colored wireframe spheres drawn around the selected GameObject in the **Scene** panel.

## How it works...

When an object is selected in a scene, if it contains a scripted component that includes the OnDrawGizmosSelected() method, then that method is invoked. Our method draws three concentric wireframe spheres in three different colors around the selected object. You can change the size of the wire spheres by changing the public radius property of the scripted component of a cube.

# Creating an editor snap-to-grid drawn by a Gizmo

If the positions of objects need to be restricted to specific increments, it is useful to have a grid drawn in the Scene window to help ensure that new objects are positioned based on those values, as well as code to snap objects to that grid.

In this recipe, we'll use Gizmos to draw a grid with a customizable grid size, color, number of lines, and line length. The result of following this recipe will look as follows:

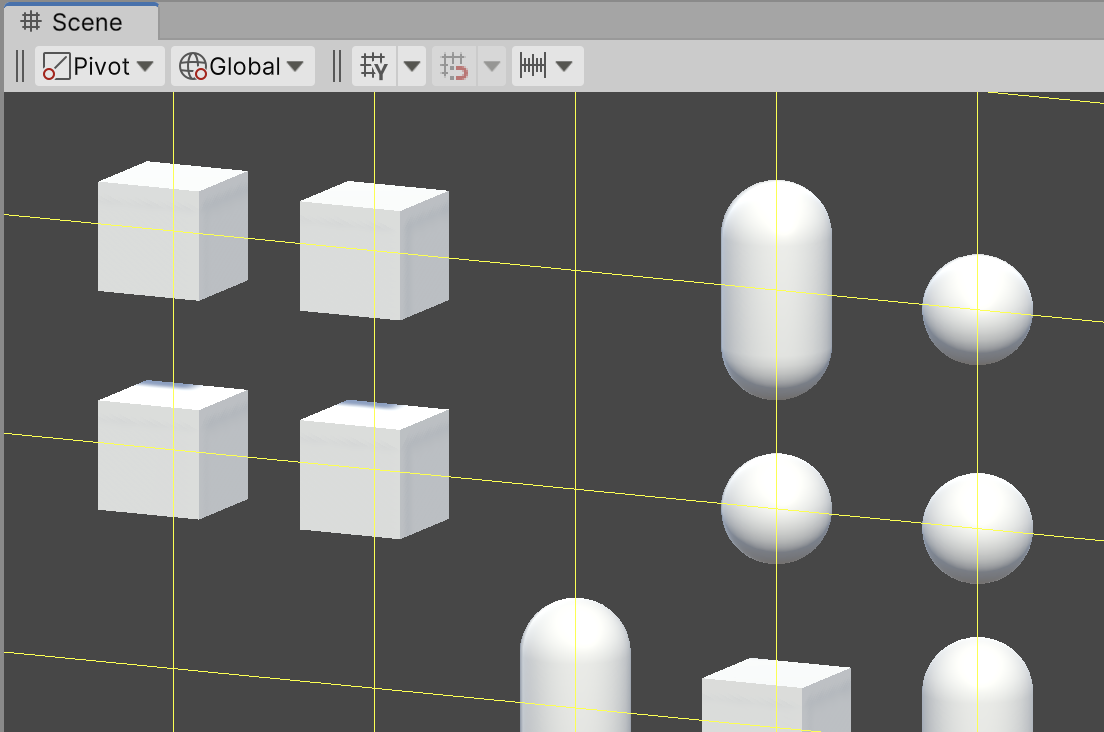


Figure 17.3 – Example of a visible grid to which objects have been snapped

17\_05\_objects\_snapped.png

## How to do it...

To create an editor snap-to-grid drawn by a Gizmo, follow these steps:

1. Create a new Unity 3D project.
2. In the **Scene** panel, turn off the Skybox view (or simply toggle off all the visual settings) so that you have a plain background for your grid:

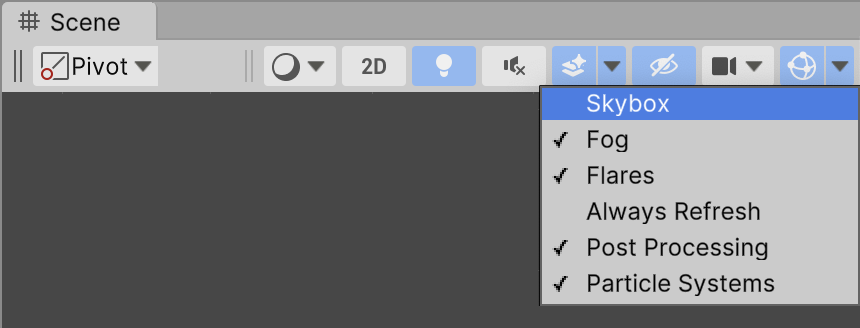


Figure 17.4 – Turning off the Skybox view in the Scene window

17\_01\_disable\_skybox\_preview.png

1. Updating the display and the child objects will be performed by a script class called GridGizmo. Create a new C# script class called GridGizmo that contains the following code:

using UnityEngine;  
   
 public class GridGizmo : MonoBehaviour {  
 public int grid = 2;  
   
 public void SetGrid(int grid) {  
 this.grid = grid;  
 SnapAllChildren();  
 }  
   
 public Color gridColor = Color.red;   
 public int numLines = 6;  
 public int lineLength = 50;  
   
 private void SnapAllChildren() {  
 foreach (Transform child in transform)  
 SnapPositionToGrid(child);  
 }  
   
 void OnDrawGizmos() {  
 Gizmos.color = gridColor;  
   
 int min = -lineLength;  
 int max = lineLength;  
   
 int n = -1 \* RoundForGrid(numLines / 2);  
 for (int i = 0; i < numLines; i++) {  
 Vector3 start = new Vector3(min, n, 0);  
 Vector3 end = new Vector3(max, n, 0);   
 Gizmos.DrawLine(start, end);  
   
 start = new Vector3(n, min, 0);  
 end = new Vector3(n, max, 0);   
 Gizmos.DrawLine(start, end);  
   
 n += grid;  
 }  
 }  
   
 public int RoundForGrid(int n) {  
 return (n/ grid) \* grid;  
 }  
   
 public int RoundForGrid(float n) {  
 int posInt = (int) (n / grid);  
 return posInt \* grid;  
 }  
   
 public void SnapPositionToGrid(Transform transform) {  
 transform.position = new Vector3 (  
 RoundForGrid(transform.position.x),  
 RoundForGrid(transform.position.y),  
 RoundForGrid(transform.position.z)  
 );  
 }  
 }

1. We can create an Editor script to add a new menu item to the **GameObject** menu. Create a folder named Editor and, in that folder, create a new C# script class called EditorGridGizmoMenuItem that contains the following code:

using UnityEngine;  
 using UnityEditor;  
   
 public class EditorGridGizmoMenuItem : Editor {  
 const string GRID\_GAME\_OBJECT\_NAME = "\_\_\_snap-to-grid\_\_\_";  
   
 [MenuItem("GameObject/Create New Snapgrid", false, 10000)]  
 static void CreateCustomEmptyGameObject(MenuCommand menuCommand) {  
 GameObject gameObject = new GameObject(GRID\_GAME\_OBJECT\_NAME);  
   
 gameObject.transform.parent = null;  
 gameObject.transform.position = Vector3.zero;  
 gameObject.AddComponent<GridGizmo>();   
 }  
 }

1. Now, let's add another Editor Script for a custom **Inspector** display (and updater) for the GridGizmo components. Also, in your Editor folder, create a new C# script class called EditorGridGizmo that contains the following code:

using UnityEngine;  
 using UnityEditor;  
  
   
 [CustomEditor(typeof(GridGizmo))]  
 public class EditorGridGizmo : Editor {  
 private GridGizmo gridGizmoObject;  
 private int grid;  
 private Color gridColor;  
 private int numLines;  
 private int lineLength;  
   
 private string[] gridSizes = {  
 "1", "2", "3", "4", "5"  
 };  
   
 void OnEnable() {  
 gridGizmoObject = (GridGizmo)target;  
 grid = serializedObject.FindProperty("grid").intValue;  
 gridColor = serializedObject.FindProperty("gridColor").colorValue;  
 numLines = serializedObject.FindProperty("numLines").intValue;  
 lineLength = serializedObject.FindProperty("lineLength").intValue;  
 }  
   
 public override void OnInspectorGUI() {  
 serializedObject.Update ();  
   
 int gridIndex = grid - 1;  
 gridIndex = EditorGUILayout.Popup("Grid size:", gridIndex, gridSizes);   
 gridColor = EditorGUILayout.ColorField("Color:", gridColor);  
 numLines = EditorGUILayout.IntField("Number of grid lines", numLines);  
 lineLength = EditorGUILayout.IntField("Length of grid lines", lineLength);  
   
 grid = gridIndex + 1;  
 gridGizmoObject.SetGrid(grid);   
 gridGizmoObject.gridColor = gridColor;  
 gridGizmoObject.numLines = numLines;   
 gridGizmoObject.lineLength = lineLength;   
 serializedObject.ApplyModifiedProperties ();  
 sceneView.RepaintAll();  
 }   
 }

1. Add a new GizmoGrid GameObject to the scene by choosing menu: **GameObject | Create New Snapgrid**. You should see a new GameObject named \_\_\_snap-to-grid\_\_\_  added to the **Hierarchy**.

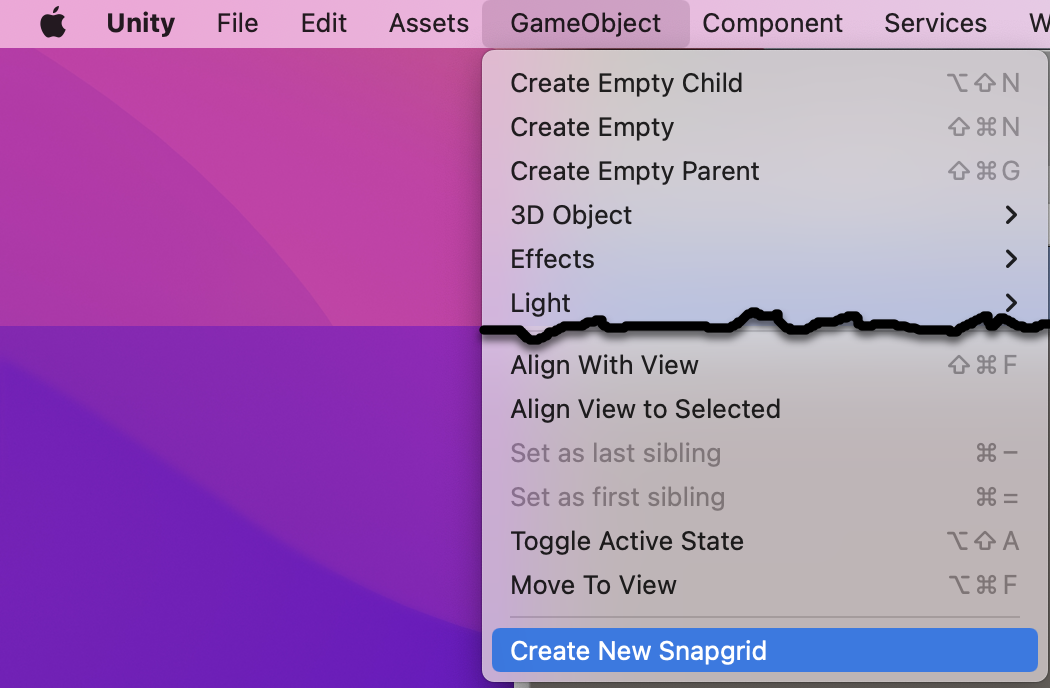


Figure 17.5 – Our new menu item at the bottom of the GameObject menu

17\_03\_new\_menu\_item.png

1. Select the \_\_\_snap-to-grid\_\_\_  GameObject and set its Position to (0, 0, 0). Now modify some of its properties in the **Inspector** panel. You can change the grid's size, the color of the grid's lines, the number of lines, and their length.

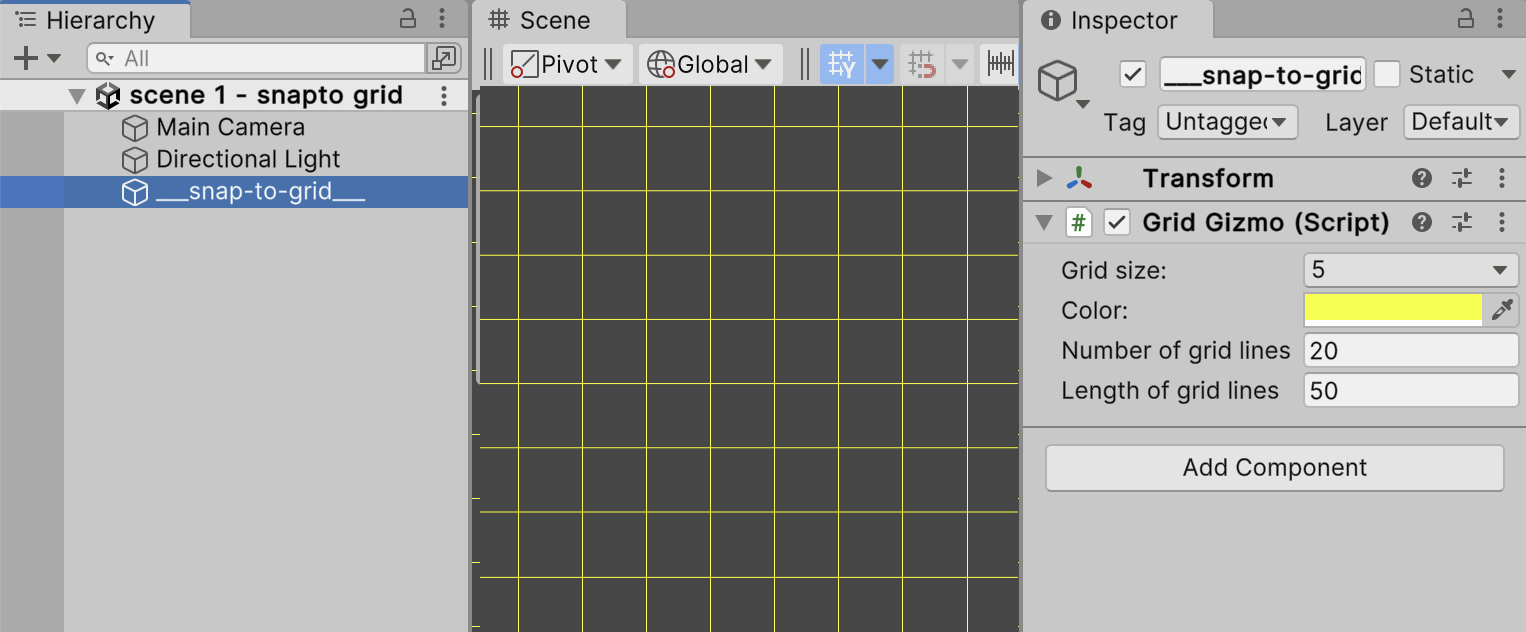


Figure 17.6 – Changing the properties of the snap grid

17\_04\_snap\_grid\_scene\_panel.png

1. Create a 3D Cube by choosing menu: **Create | 3D Object | Cube**. Set its Position to (0, 0, 0).
2. Snapping will work for all children of \_\_\_snap-to-grid\_\_\_ in the **Hierarchy** panel. So make this new Cube a child of \_\_\_snap-to-grid\_\_\_ by dragging it onto \_\_\_snap-to-grid\_\_\_.
3. We now need a small script class so that each time the GameObject is moved (in Editor mode), it asks for its position to be snapped by the parent scripted component. Create a C# script class called SnapToGizmoGrid containing the following:

using UnityEngine;  
   
 [ExecuteInEditMode]  
 public class SnapToGridGizmo : MonoBehaviour {  
 public void Update()  
 {  
 #if UNITY\_EDITOR  
 transform.parent.GetComponent<GridGizmo>().SnapPositionToGrid(transform);  
 #endif  
 }  
 }

1. Add add an instance of script class SnapToGizmoGrid  object as a component to the 3D Cube.
2. Make some duplicates of the 3D Cube, using the move tool in the Scene to distributte them randomly around the scene – you'll find that they snap to the grid.
3. Select  \_\_\_snap-to-grid\_\_\_ again and modify some of its properties in the **Inspector** panel. You'll see that the changes are instantly visible in the scene and that all the child objects that have a scripted component of SnapToGizmoGrid are snapped to any new grid size changes.

## How it works...

Scripts that we place in a folder named Editor are known as Editor scripts, and in such scripts, we can customize and extend the features and look-and-feel of the Unity Editor. In this recipe, we've created Editor scripts to display and limit the movement of scene objects in a grid.

The EditorGridGizmoMenuItem script class adds a new item to the GameObject menu. When selected, a new GameObject is added to the Hierarchy window named \_\_\_snap-to-grid\_\_\_, positioned at (0, 0, 0), and containing an instance object component of the GridGizmo script class.

GridGizmo draws a 2D grid based on public properties for grid size, color, number of lines, and line length. Regarding the SetGrid(...) method, as well as updating the integer grid size variable grid, it also invokes the SnapAllChildren() method so that each time the grid size is changed, all child GameObjects are snapped into the new grid positions.

The SnapToGridGizmo script class includes an Editor attribute called [ExecuteInEditMode] so that it will receive Update() messages when its properties are changed at design time in the Editor. Each time Update() is invoked, it calls the SnapPositionToGrid(...) method in its parent GridGizmo instance object so that its position is snapped based on the current settings of the grid. To ensure this logic and code is not compiled into any final build of the game, the contents of Update() are wrapped in an #if UNITY\_EDITOR compiler test. Such content is removed before a build is compiled for the final game.

The EditorGridGizmo script class is a custom Editor Inspector component. This allows us to both control which properties are displayed in the Inspector window and how they are displayed, and it allows actions to be performed when any values are changed. So, for example, after changes have been saved, the sceneView.RepaintAll() statement ensures that the grid is redisplayed since it results in an OnDrawGizmos() message being sent.

## There's more...

The preceding implementation works fine when the only objects we want to tie to the grid are children of our \_\_\_snap-to-grid\_\_\_ GameObject. However, if we don't want to require affected objects to be such children, then we can use the singleton pattern to allow a GameObject anywhere in the **Hierarchy** panel to get a reference to the **GridGizmo** component. Do the following to adapt this recipe to use this approach:

1. Update the contents of the GridGizmo.cs #C script class so that it matches the following:

using System.Collections;  
using System.Collections.Generic;  
using UnityEngine;  
  
public class GridGizmo : MonoBehaviour  
{  
 private static GridGizmo \_instance = null;  
  
 public static void SetInstance(GridGizmo instance) {  
 \_instance = instance;  
 }  
  
 public static GridGizmo GetInstance() {  
 if(\_instance == null){  
 throw new System.Exception("error - no GameObject has GridGizmo component to snap to ...");  
 }  
  
 return \_instance;  
 }  
  
 public int grid = 2;  
  
 public void SetGrid(int grid) {  
 this.grid = grid;  
 SnapAllChildren();  
 }  
 ... the rest of the script is unchanged ...

1. Update the contents of the Update() method of the SnapToGizmoGrid.cs C# script class so that it matches the following:

using System.Collections;  
using System.Collections.Generic;  
using UnityEngine;  
  
[ExecuteInEditMode]  
public class SnapToGridGizmo : MonoBehaviour  
{  
 /\*\*  
 \* we've moved!  
 \* snap position to its grid  
 \*/  
 public void Update()  
 {  
#if UNITY\_EDITOR  
 GridGizmo.GetInstance().SnapPositionToGrid(transform);  
#endif  
 }  
   
}

1. Update the contents of the EditorGridGizmoMenuItem.cs C# script class to match the following:

using UnityEngine;  
using UnityEditor;  
using System.Collections;  
  
public class EditorGridGizmoMenuItem : Editor  
{  
 const string GRID\_GAME\_OBJECT\_NAME = "\_\_\_snap-to-grid\_\_\_";  
  
 /\*\*  
 \* menu item to create a GridSnapper  
 \*/  
 [MenuItem("GameObject/Create New Snapgrid", false, 10000)]  
 static void CreateCustomEmptyGameObject(MenuCommand menuCommand)  
 {  
 GameObject gameObject = new GameObject(GRID\_GAME\_OBJECT\_NAME);  
   
 // ensure not a child of any other object  
 gameObject.transform.parent = null;  
   
 // zero position  
 gameObject.transform.position = Vector3.zero;  
  
 // add Scripted component  
 GridGizmo newInstance = gameObject.AddComponent<GridGizmo>();  
 GridGizmo.SetInstance(newInstance);  
 }  
}

When the menu item is chosen, the method in the EditorGridGizmoMenuItem C# script class creates the game grid GameObject, adds a GridGizmo scripted component, and also uses the new SetInstance(...) method of the GridGizmo script class to store the reference to the GridGizmo scripted component in its static variable, called \_instance. This means that when we add GameObjects such as cubes or cylinders anywhere in the **Hierarchy** panel, we can add to them a SnapToGizmoGrid scripted component that can call the GridGizmo.GetInstance() public state method. To summarize what we have done, by using the singleton pattern, we have allowed any GameObject, anywhere in the **Hierarchy** panel, to access the GridGizmo component.

# 3 - Generating and running a default test script class

Unity can create a default C# test script for you, thereby enabling you to quickly start creating and executing tests on your project. In this recipe, we will add the Unity Test Framework to a project and use it to automatically generate a default test script for us. This will be the basis for several of the following recipes:

Graphical user interface, text, application, chat or text message

Description automatically generated

Figure 17.16 – Passing the test (indicated by green ticks)

17\_11\_passingTests.png

## How to do it...

To generate a default test script class, follow these steps:

1. Create a new 3D Unity project.
2. Display the **Test Runner** panel by going to **Window | General | Test Runner**.

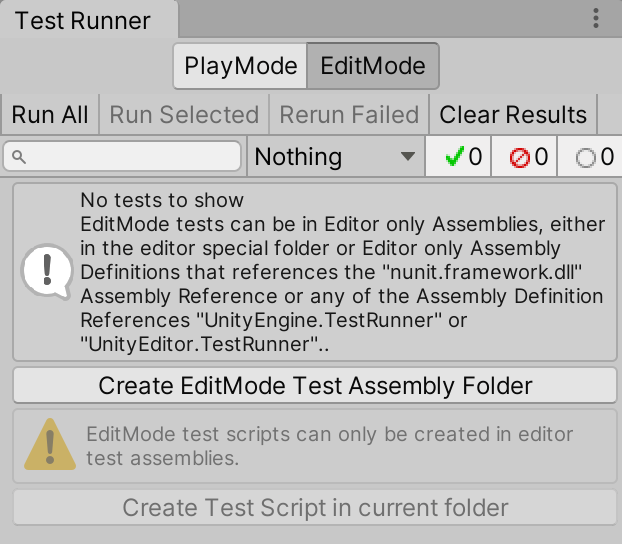


Figure 17.17 – The Test Runner panel

17\_12\_test\_runner\_panel.png

1. Ensure that the **EditMode** button is selected in the **Test Runner** panel.
2. In the **Test Runner** panel, click the **Create EditMode** **Test Assembly** **Folder** button. You'll now see a folder called Tests that's been created in the **Project** panel.
3. Select the Tests folder.
4. In the **Test Runner** panel, click the **Create Test** Script in Current Folder button.
5. You should now have a new C# script named NewTestScript inside your **Tests** folder.
6. To run the tests in your script class, click the **Run All** button in the **Test Runner** panel.
7. You should now see all green ticks (check marks) in the panel, as shown in Figure 13.16.

## How it works...

Unity checks that you have a folder named Editor selected in the Project window, and then creates a NewTestScript C# script class for you containing the following:

using System.Collections;  
using System.Collections.Generic;  
using NUnit.Framework;  
using UnityEngine;  
using UnityEngine.TestTools;  
  
  
public class NewTestScript {  
 // A Test behaves as an ordinary method  
 [Test]  
 public void NewTestScript1SimplePasses() {  
 // Use the Assert class to test conditions  
 }  
  
 // A UnityTest behaves like a coroutine in Play Mode. In Edit Mode you can use  
 // `yield return null;` to skip a frame.  
 [UnityTest]  
 public IEnumerator NewTestScriptWithEnumeratorPasses() {  
 // Use the Assert class to test conditions.  
 // Use yield to skip a frame.  
 yield return null;  
 }  
 }

In the Test Runner window, you should see the script class and its two methods listed. Note that the first line in the Test Runner window is the Unity project name; the second line will say <projectName>.dll, followed by your script class's namespace, which is Tests, followed by the script class called NewTestScript, and, finally, each of the test methods. The generated default script creates two methods:

* NewTestScriptSimplePasses(): This is a simple, empty test method (since it's empty, it will always pass).
* NewTestScriptWithEnumeratorPasses(): This is a coroutine test method that always yields a null return so, again, it will not create a failed event and always pass.

The generated script and two methods are a basic skeleton that we can populate with simple methods and coroutine methods as appropriate for each project's testing requirements.

There are three symbols to indicate the status of each test/class:

* **Empty circle**: The test hasn't been executed since the script class was last changed.
* **Green tick** (checkmark): The test passed successfully.
* **Red cross**: The test failed.

## There's more...

Here are some details that you won't want to miss.

### Creating a default test script from the Project window's Create menu

Another way of creating a default Unit Test script is by going to the Project window and going to **Create | Testing | C# Test Script**.

### Edit mode minimum skeleton unit test script

Be aware that if you are only going to use this script class for testing in **EditMode**, you can delete the second method using statements such as the following to give you a minimal skeleton to work from:

using NUnit.Framework;  
   
 public class UnitTestSkeleton  
 {  
 [Test]  
 public void NewTestScriptSimplePasses()  
 {  
 // write your assertion(s) here  
 }  
 }

This simpler skeleton testing class is for when we are writing code-only unit tests.

# 4 - A simple unit test

In the same way as printing "hello world" is most programmers' first program statement, asserting that 1 + 1 = 2 is perhaps the most common first test that's executed for those learning unit testing. That's what we'll create in this recipe:

Graphical user interface, text, application, chat or text message

Description automatically generated

Figure 17.18 – Our simple numeric test method passing with a green tick

17\_13\_equals.png

## How to do it...

To create and execute a simple unit test, follow these steps:

1. Create a new 3D Unity project.
2. Display the **Test Runner** panel by going to **Window | General | Test Runner**.
3. Ensure that the **EditMode** button is selected in the **Test Runner** panel.
4. In the **Test Runner** panel, click the **Create EditMode** **Test Assembly** **Folder** button. You'll now see a folder called Tests that's been created in the **Project** panel.
5. Select the Tests folder.
6. In the **Test Runner** panel, click the **Create Test** Script in Current Folder button.
7. You should now have a new C# script named NewTestScript inside your **Tests** folder.
8. Rename the script class from NewTestScript  to  SimpleTester and replace its contents with the following:

using NUnit.Framework;  
   
 class SimpleTester  
 {  
 [Test]  
 public void TestOnePlusOneEqualsTwo()  
 {  
 // Arrange  
 int n1 = 1;  
 int n2 = 1;  
 int expectedResult = 2;  
   
 // Act  
 int result = n1 + n2;  
   
 // Assert  
 Assert.AreEqual(expectedResult, result);  
 }  
 }

1. Click the **Run All** button in the **Test Runner** panel.
2. You should see the results of your unit test being executed – if the test concluded successfully, it should have a green "tick" next to it.

## How it works...

In this recipe, you declared that the TestOnePlusOneEqualsTwo() method in the SimpleTester.cs C# script class is a test method. When executing this test method, Unity **Test Runner** executes each statement in sequence, so variables n1, n2, and expectedResult are set, then the calculation of 1 + 1 is stored in the variable result, and finally (the most important bit), we make an assertion of what should be true after executing that code. Our assertion states that the value of the expectedResult variable should be equal to the value of the variable result.

If the assertion is true, the test passed; otherwise, it failed. Generally, as programmers, we expect our code to pass, so we inspect each fail very carefully, first to see whether we have an obvious error, then perhaps to check whether the test itself is correct (especially if it's a new test), and then to begin to debug and understand why our code behaved in such a way that it did not yield the anticipated result.

## There's more...

Here are some details that you won't want to miss.

### Shorter tests with values in the assertion

For simple calculations, some programmers prefer to write less test code by putting the values directly into the assertion. So, as shown here, our 1 + 1 = 2 test could be expressed in a single assertion, where the expected value of 2 and the expression 1 + 1 are entered directly into an AreEqual(...) method's invocation:

using NUnit.Framework;  
   
 class SimpleTester  
 {  
 [Test]  
 public void TestOnePlusOneEqualsTwo()  
 {  
 // Assert  
 Assert.AreEqual(2, 1 + 1);  
 }  
 }

However, if you are new to testing, you may prefer the previous approach, whereby the way you prepare, execute, and store the results, as well as the property assertions about those results, are structured clearly in a sequence of **Arrange/Act/Assert**. By storing values in meaningfully named variables, what we are asserting is very clear.

### Expected value followed by the actual value

When comparing values with assertions, it is customary for the expected (correct) value to be given first, followed by the actual value:

Assert.AreEqual( <expectedValue>, <actualValue> );

While it makes no difference to the true or false nature of equality, it can make a difference to messages when tests fail with some testing frameworks (for example, "got 2 but expected 3" has a very different meaning to "got 3 but expected 2"). Hence, the following assertion would output a message that would be confusing, since 2 was our expected result:

public void TestTwoEqualsThreeShouldFail() {  
 // Arrange  
 int expectedResult = 2;  
   
 // Act  
 int result = 1 + 2; // 3 !!!  
   
 // Assert  
 Assert.AreEqual(result, expectedResult);  
 }

The following screenshot illustrates how we will get a misleading message when the arguments are the wrong way around in our assertion method:

Text

Description automatically generated

Figure 17.19 – Confusing message due to an incorrect argument sequence in the assertion

17\_14\_failedTestDetails.png

# 5 - Parameterizing tests with a data provider

If we are testing our code using a range of test data, then sometimes, there is little difference between each test apart from the values. Rather than duplicating our Arrange/Act/Assert statements, we can reuse a single method, and Unity Test Runner will loop through a collection of test data, running the test method for each set of test data. The special method that provides multiple sets of test data to a test method is known as a **DataProvider**, and we'll create one in this recipe:

Graphical user interface, text, application

Description automatically generated

Figure 17.20 – Running the test method with many sets of values with a DataProvider method

17\_14\_failedTestDetails.png

## How to do it...

To parameterize tests with a data provider method, follow these steps:

1. Create a new 3D Unity project.
2. Display the **Test Runner** panel by going to **Window | General | Test Runner**.
3. Ensure that the **EditMode** button is selected in the **Test Runner** panel.
4. In the **Test Runner** panel, click the **Create EditMode** **Test Assembly** **Folder** button. You'll now see a folder called Tests that's been created in the **Project** panel.
5. Select the Tests folder.
6. In the **Test Runner** panel, click the **Create Test** Script in Current Folder button.
7. You should now have a new C# script named NewTestScript inside your **Tests** folder.
8. Rename the script class from NewTestScript  to  DataProviderTester and replace its contents with the following:

using NUnit.Framework;  
   
 class DataProviderTester  
 {  
 [Test, TestCaseSource("AdditionProvider")]  
 public void TestAdd(int num1, int num2, int expectedResult)  
 {  
 // Arrange  
 // (not needed - since values coming as arguments)  
   
 // Act  
 int result = num1 + num2;  
   
 // Assert  
 Assert.AreEqual(expectedResult, result);  
 }  
   
 // the data provider  
 static object[] AdditionProvider =  
 {  
 new object[] { 0, 0, 0 },  
 new object[] { 1, 0, 1 },  
 new object[] { 0, 1, 1 },  
 new object[] { 1, 1, 2 }  
 };  
 }

1. Display the **Test Runner** window by going to **Window | General | Test Runner**.
2. Ensure that the **EditMode** button is selected in the Test Runner window.
3. Click **Run All**.
4. You should see the results of your unit test being executed. You should see four sets of results for the TestAdd(...) test method, one for each of the datasets provided by the AdditionProvider method.

## How it works...

We have indicated that the TestAdd(...) method is a test method with a compiler attribute called [Test]. However, in this case, we have added additional information to state that the [TestCaseSource(...)] data source for this method is the AdditionProvider method.

This means that Unity Test Runner will retrieve the data objects from the additional provider and create multiple tests for the TestAdd(...) method, one for each set of data from the AdditionProvider() method.

In the **Test Runner**panel, we can see a line for each of these tests:

TestAdd(0, 0, 0)  
TestAdd(1, 0, 1)  
TestAdd(0, 1, 1)  
TestAdd(1, 1, 2)

# 6 - Unit testing a simple health script class

Let's create something that might be used in a game and that can easily be unit tested. Classes that do **not** subclass from MonoBehaviour are much easier to unit test since instance objects can be created using the new keyword. If the class is carefully designed with private data and public methods with clearly declared dependencies as parameters, it becomes easy to write a set of tests to make us confident that objects of this class will behave as expected in terms of default values, as well as valid and invalid data.

In this recipe, we will create a health script class and a set of tests for this class. This kind of class can be reused for both the health of human players and also **artificial intelligence** (**AI**)-controlled enemies in a game:

Graphical user interface, text, application

Description automatically generated

Figure 17.21 – Passing tests for our Health script class

17\_14\_failedTestDetails.png

## How to do it...

To unit test a health script class, follow these steps:

1. Create a new 3D Unity project.
2. Create a new folder named \_Scripts.
3. Inside your \_Scripts folder, create a new Health.cs C# script class containing the following:

using UnityEngine;  
 using System.Collections;  
   
 public class Health  
 {  
 private float \_health = 1;  
   
 public float GetHealth()  
 {  
 return \_health;  
 }  
   
 public bool AddHealth(float heathPlus)  
 {  
 if(heathPlus > 0){  
 \_health += heathPlus;  
  
 // ensure never more than 1  
 if(\_health > 1) \_health = 1;  
 return true;  
 } else {  
 return false;  
 }  
 }  
   
 public bool KillCharacter()  
 {  
 \_health = 0;  
 return true;  
 }  
 }

1. Since we want to test scripts in this folder, we need to add an **Assembly Definition** here. From the **Project** panel choose menu: **Create |** **Assembly Definition** , and rename the new **Assembly Definition** HealthScriptAssembly.

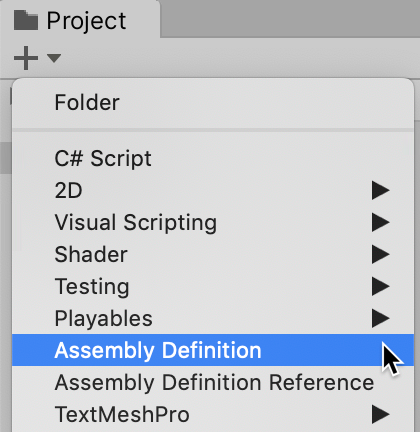


Figure 17.22 – Creating an Assembly Definition in our \_Scripts folder

17\_14\_failedTestDetails.png

1. Select HealthScriptAssembly  in the **Project** panel. In the **Inspector** ensure the **Name** property is HealthScriptAssembly, and in the list of **Platforms** ensure that only the **Editor** is checked; and then click the **Apply** button at the bottom of the **Inspector** panel.

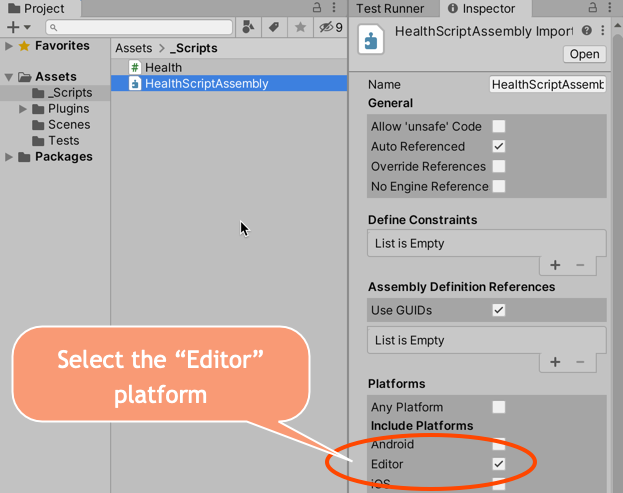


Figure 17.23 – Setting the Editor platform and new name for HealthScriptAssembly

17\_14\_failedTestDetails.png

1. Display the **Test Runner** window by going to **Window | General | Test Runner**.
2. Ensure that the **EditMode** button is selected in the Test Runner window.
3. In the **Test Runner** window, click the **Create EditMode Test Assembly** **Folder** button. You'll now see a folder called **Tests** in the **Project** window.
4. Select the **Tests** folder. This contains an **Assembly Reference** named **Tests**. Select this asset file in the Project window and look at its properties in the Inspector window. This **Assembly Reference** already has references to two **Assembly Definition References**: UnityEngine.TestRunner and UnityEditor.TestRunner.
5. Now, we need to add our HealthScriptAssembly to the **Tests** assembly so that we can test our Health script class. In the **Inspector** window, click the plus (+) button to add a new **Assembly Definition Reference** to the slot that was created to locate and drag the **HealthScript Assembly Definition**, and then click the **Apply** button at the bottom of the **Inspector** panel.

Graphical user interface, application

Description automatically generated

Figure 17.24 – Adding HealthScriptAssembly to the Tests Assembly Definition

17\_14\_failedTestDetails.png

1. In the Test Runner window, click the Create Test Script in Current Folder button. Rename the script class TestHealth.cs and ensure it contains the following code:

using NUnit.Framework;  
  
class TestHealth {  
 [Test]  
 public void TestReturnsOneWhenCreated() {  
 // Arrange  
 Health h = new Health ();  
 float expectedResult = 1;  
  
 // Act  
 float result = h.GetHealth ();  
  
 // Assert  
 Assert.AreEqual (expectedResult, result);  
 }   
  
 [Test]  
 public void TestPointTwoAfterAddPointOneTwiceAfterKill() {  
 // Arrange  
 Health h = new Health();  
 float healthToAdd = 0.1f;  
 float expectedResult = 0.2f;  
  
 // Act  
 h.KillCharacter();  
 h.AddHealth(healthToAdd);  
 h.AddHealth(healthToAdd);  
 float result = h.GetHealth();  
  
  
 // Assert  
 Assert.AreEqual(expectedResult, result);  
 }  
  
 [Test]  
 public void TestNoChangeAndReturnsFalseWhenAddNegativeValue() {  
 // Arrange  
 Health h = new Health();  
 float healthToAdd = -1;  
 bool expectedResultBool = false;  
 float expectedResultFloat = 1;  
  
 // Act  
 bool resultBool = h.AddHealth(healthToAdd);  
 float resultFloat = h.GetHealth();  
  
 // Assert  
 Assert.AreEqual(expectedResultBool, resultBool);  
 Assert.AreEqual(expectedResultFloat, resultFloat);  
 }  
  
 [Test]  
 public void TestReturnsZeroWhenKilled() {  
 // Arrange  
 Health h = new Health();  
 float expectedResult = 0;  
  
 // Act  
 h.KillCharacter();  
 float result = h.GetHealth();  
  
 // Assert  
 Assert.AreEqual(expectedResult, result);  
 }  
  
  
 [Test]  
 public void TestHealthNotGoAboveOne() {  
 // Arrange  
 Health h = new Health();  
 float expectedResult = 1;  
  
 // Act  
 h.AddHealth(0.1f);  
 h.AddHealth(0.5f);  
 h.AddHealth(1);  
 h.AddHealth(5);  
 float result = h.GetHealth();  
  
 // Assert  
 Assert.AreEqual(expectedResult, result);  
 }  
}

1. Display the **Test Runner** window by going to **Window | Debug | Test Runner**.

***Note****: At the time of writing, using a Unity 2023 beta system, there is an issue with the* TestHealth script *class not being able to find the* Heath *class. A quick fix is to move the* Health *script class into the Tests folder – hopefully this will have been fixed for the release candidate of Unity 2023.*

1. Ensure that the **EditMode** button is selected in the **Test Runner** window.
2. Click **Run All**.
3. You should see the results of your unit tests being executed.

## How it works...

Each of the C# script classes is described here.

### Health.cs

This script class has one private property. Since it is private, it can only be changed by methods. Its initial value is 1.0 – in other words, 100% health:

* health (float): The valid range is from 0 (dead!) to 1.0 (100% health).

There are three public methods:

* GetHealth(): This returns the current value of the health float number (which should be between 0 and 1.0).
* AddHealth(float): This takes a float as input (the amount to add to the health) and returns a Boolean true/false regarding whether the value was valid. Note that the logic of this method is that it accepts values of 0 or more (and will return true), but it will ensure that the value of health is never more than 1.
* KillCharacter(): This method sets health to zero and returns true since it is always successful in this action.

### TestHealth.cs

This script class has five methods:

* TestReturnsOneWhenCreated(): This tests that the initial value of health is 1 when a new Health object is created.
* TestPointTwoAfterAddPointOneTwiceAfterKill(): This tests that after a kill (health set to zero), and then adding 0.1 on two occasions, that the health is 0.2.
* TestReturnsZeroWhenKilled(): This tests that the health value is set to zero immediately after the KillCharacter() method has been called.
* TestNoChangeAndReturnsFalseWhenAddNegativeValue(): This tests that attempting to add a negative value to health should return false and that the value of health should not have changed. This method is an example of a test with more than one assertion (but both are related to the actions).
* TestHealthNotGoAboveOne(): This test verifies that even when lots of values are added to health that total more than 1.0, the value that's returned from GetHealth() is 1.

Hopefully, all the tests pass when you run them, giving you some confidence that the logic that's been implemented in the Health.cs script class behaves as intended.

# 7 - Creating and executing a unit test in PlayMode

It's a good idea to write as much of the logic for a game as isolated, non-MonoBehaviour classes that are easy to unit test in Edit Mode as possible. However, some of the logic in a game relates to things that happen when the game is running. Examples include physics, collisions, and timing-based events. We test these parts of our games in PlayMode.

In this recipe, we'll create one very simple PlayMode test to check that the physics affect a Rigidbody (based on an example from the Unity documentation):

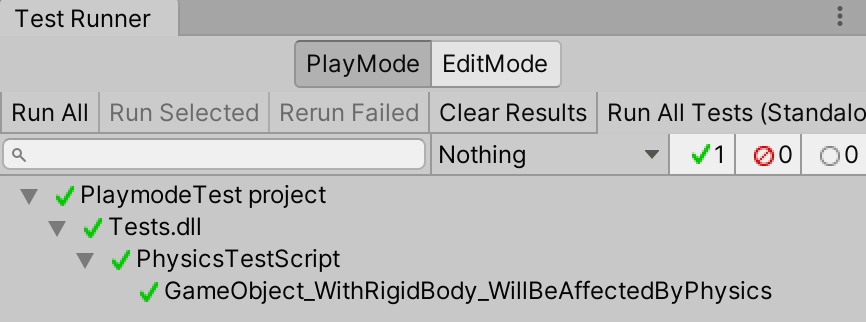


Figure 17.25 – Running a physics PlayMode test

17\_14\_failedTestDetails.png

## How to do it...

To create and execute a unit test in PlayMode, follow these steps:

1. Create a new 3D Unity project.
2. Display the **Test Runner** window by going to **Window | General | Test Runner**.
3. Ensure that the **PlayMode** button is selected in the **Test Runner** window.
4. In the **Test Runner** window, click the **Create PlayMode Test Assembly** **Folder** button. You'll now see a folder called **Tests** in the **Project** window.
5. In the **Test Runner** window, click the **Create Test Script** in current folder button. Rename the script class PhysicsTestScript.cs and ensure it contains the following code:

using UnityEngine;  
 using UnityEngine.TestTools;  
 using NUnit.Framework;  
 using System.Collections;  
   
 public class PhysicsTestScript  
 {  
 [UnityTest]  
 public IEnumerator GameObject\_WithRigidBody\_WillBeAffectedByPhysics()  
 {  
 // Arrange  
 var go = new GameObject();  
 go.AddComponent<Rigidbody>();  
 var originalPosition = go.transform.position.y;  
   
 // Act  
 yield return new WaitForFixedUpdate();  
   
 // Assert  
 Assert.AreNotEqual(originalPosition, go.transform.position.y);  
 }  
 }

1. Click **Run All**.
2. In the **Hierarchy** window, you'll see that a temporary scene is created (named something along the lines of **InitTestScene6623462364**) and that a GameObject named **Code Based Test Runner** is created.
3. In the **Game** window, you will briefly see the message **Display 1 No Cameras** **Rendering**:

Graphical user interface, application, PowerPoint

Description automatically generated

Figure 17.26 – Temporary scene and GameObject for the runtime test

17\_14\_failedTestDetails.png

1. You should see the results of your unit test being executed – if the test is concluded successfully, it should have a green tick next to it.

## How it works...

Methods marked with the [UnityTest] attribute are run as coroutines. A coroutine has the ability to pause execution (when it meets a yield statement) and return control to Unity, but then to continue where it left off when called again (for example, the next frame, second frame, and so on). The yield statement indicates the statement after which, and for how long, execution of the method is to be paused. Examples of different types of yield include the following:

* Waiting until the next frame: null
* Waiting for a given length of time: WaitForSeconds(<seconds>)
* Waiting until the next fixed-update time period (physics is not applied to each frame (since the framerate varies) but after a fixed period of time): WaitForFixedUpdate()

The GameObject\_WithRigidBody\_WillBeAffectedByPhysics() method creates a new GameObject and attaches to it a Rigidbody. It also stores the original Y position. The yield statement makes PlayMode Test Runner wait until physics has begun at the next fixed update period. Finally, an assertion is made that the original Y position is not equal to the new Y position (after the physics fixed update). Since the defaults for a Rigidbody are that gravity will be applied, this is a good test that physics is being applied to the new object (in other words, it should have started falling down once physics had been applied).

# 8 - PlayMode testing a door animation

Having learned the basics of PlayMode testing in the previous recipe, let's test something non-trivial that we might find in a game. In this recipe, we'll create a PlayMode test to ensure that a door opening animation plays when the player's sphere object enters a collider.

A scene has been provided with the player's sphere initialized to roll toward a red door. When the sphere hits the collider (the OnTriggerEnter event), some code sets the door's Animator Controller Opening variable to true, which transitions the door from its closed state to its open state, as shown in the following screenshot:



Figure 17.27 – The door will open (upward) when hit by the sphere

17\_22\_ball\_scene.png

Thanks to the creator of the ground texture; it was designed by Starline and published at [Freepik.com](http://Freepik.com).

## Getting ready

For this recipe, a Unity package has been provided (doorScene.unitypackage) in the 17\_07 folder.

## How to do it...

To PlayMode test a door animation, follow these steps:

1. Create a new 3D project and delete the default folder, called Scenes.
2. Import the Unity package provided (doorScene.unitypackage).
3. Add the doorScene and menuScene scenes to the project build (the sequence doesn't matter), by choosing menu: **File | Build Settings…**, opening each scene and clicking the **Add Open Scene** button in the **Build Settings** panel.
4. Ensure that the scene that's currently open is menuScene .
5. Display the **Test Runner** window by going to **Window | General | Test Runner**.
6. Ensure that the **PlayMode** button is selected in the **Test Runner** window.
7. In the **Project** window, select the top-level folder called Assets.
8. In the **Test Runner** window, click the **Create PlayMode Test Assembly** **Folder** button. A new folder, named Tests, should have been created.
9. In the **Project** window, open the Tests folder.
10. In the **Test Runner** window, click the **Create Test Script** in the current folder button. Rename this script class DoorTest.
11. Edit the DoorTest.cs script class by replacing its content with the following:

using System.Collections;  
 using NUnit.Framework;  
 using UnityEngine;  
 using UnityEngine.SceneManagement;  
 using UnityEngine.TestTools;  
   
 public class DoorTest  
 {  
 const int BASE\_LAYER = 0;  
 private string initialScenePath;  
 private Animator doorAnimator;  
 private Scene tempTestScene;  
   
 // name of scene being tested by this class  
 private string sceneToTest = "doorScene";  
   
 [SetUp]  
 public void Setup()  
 {  
 // setup - load the scene  
 tempTestScene = SceneManager.GetActiveScene();  
 }  
 }

1. Add the following test method to DoorTest.cs:

[UnityTest]  
 public IEnumerator TestDoorAnimationStateStartsClosed()  
 {  
 // load scene to be tested  
 yield return SceneManager.LoadSceneAsync(sceneToTest, LoadSceneMode.Additive);  
 SceneManager.SetActiveScene(SceneManager.GetSceneByName(sceneToTest));  
   
 // Arrange  
 doorAnimator = GameObject.FindWithTag("Door").GetComponent<Animator>();  
 string expectedDoorAnimationState = "DoorClosed";  
   
 // immediate next frame  
 yield return null;  
   
 // Act  
 AnimatorClipInfo[] currentClipInfo = doorAnimator.GetCurrentAnimatorClipInfo(BASE\_LAYER);  
 string doorAnimationState = currentClipInfo[0].clip.name;  
   
 // Assert  
 Assert.AreEqual(expectedDoorAnimationState, doorAnimationState);  
   
 // teardown - reload original temp test scene  
 SceneManager.SetActiveScene(tempTestScene);  
 yield return SceneManager.UnloadSceneAsync(sceneToTest);  
 }

1. Add the following test method to DoorTest.cs:

[UnityTest]  
 public IEnumerator TestIsOpeningStartsFalse()  
 {  
 // load scene to be tested  
 yield return SceneManager.LoadSceneAsync(sceneToTest, LoadSceneMode.Additive);  
 SceneManager.SetActiveScene(SceneManager.GetSceneByName(sceneToTest));  
   
 // Arrange  
 doorAnimator = GameObject.FindWithTag("Door").GetComponent<Animator>();  
   
 // immediate next frame  
 yield return null;  
   
 // Act  
 bool isOpening = doorAnimator.GetBool("Opening");  
   
 // Assert  
 Assert.IsFalse(isOpening);  
   
 // teardown - reload original temp test scene  
 SceneManager.SetActiveScene(tempTestScene);  
 yield return SceneManager.UnloadSceneAsync(sceneToTest);  
 }

1. Add the following test method to DoorTest.cs:

[UnityTest]  
 public IEnumerator TestDoorAnimationStateOpenAfterAFewSeconds()  
 {  
 // load scene to be tested  
 yield return SceneManager.LoadSceneAsync(sceneToTest, LoadSceneMode.Additive);  
 SceneManager.SetActiveScene(SceneManager.GetSceneByName(sceneToTest));  
   
 // wait a few seconds  
 int secondsToWait = 3;  
 yield return new WaitForSeconds(secondsToWait);  
   
 // Arrange  
 doorAnimator = GameObject.FindWithTag("Door").GetComponent<Animator>();  
 string expectedDoorAnimationState = "DoorOpen";  
   
   
 // Act  
 AnimatorClipInfo[] currentClipInfo = doorAnimator.GetCurrentAnimatorClipInfo(BASE\_LAYER);  
 string doorAnimationState = currentClipInfo[0].clip.name;  
 bool isOpening = doorAnimator.GetBool("Opening");  
   
 // Assert  
 Assert.AreEqual(expectedDoorAnimationState, doorAnimationState);  
 Assert.IsTrue(isOpening);  
   
 // teardown - reload original temp test scene  
 SceneManager.SetActiveScene(tempTestScene);  
 yield return SceneManager.UnloadSceneAsync(sceneToTest);  
 }

1. Click **Run All**.
2. As the tests run, you will see that, in the **Hierarchy**, **Game**, and **Scene** windows, a temporary scene is created, then **doorScene** running, with the sphere rolling toward the red door.
3. You should see the results of your unit test being executed – if all the tests conclude successfully, there should be green ticks (check marks) next to each test.

## How it works...

In this recipe, you added two scenes to the build so that they can be selected in our scripts using SceneManager during PlayMode testing.

We opened menuScene so that we can clearly see when Unity runs different scenes during our PlayMode testing – and we'll see the menu scene reopened after testing takes place.

There is a SetUp() method that is executed before each test. The SetUp() and TearDown() methods are very useful for preparing things before each test and resetting things back to how they were before the test took place. Unfortunately, aspects such as loading our door scene before running each test, and then reloading the menu after each test, involve waiting until the scene load process has completed. We can't place yield statements in our SetUp() and TearDown() methods, so you'll see that each test has repeated scene loading at the beginning and end of each test:

// load scene to be tested  
 yield return SceneManager.LoadSceneAsync(sceneToTest, LoadSceneMode.Additive);  
 SceneManager.SetActiveScene(SceneManager.GetSceneByName(sceneToTest));  
   
 // Arrange-Act-Assert goes here  
   
 // teardown - reload original temp test scene  
 SceneManager.SetActiveScene(tempTestScene);  
 yield return SceneManager.UnloadSceneAsync(sceneToTest);

For each test, we wait, either for a single frame (yield null) or for a few seconds (yield return new WaitForSeconds(...)). This ensures that all objects have been created and physics has started before our test starts running. The first two tests check the initial conditions – in other words, that the door begins in the DoorClosed animation state and that the animation controller's isOpening variable is false.

The final test waits a few seconds (which is enough time for the sphere to roll up to the door and trigger the opening animation) and tests that the door is entering/has entered the DoorOpen animation state and that the animation controller's isOpening variable is true.

As can be seen, there is quite a bit more to PlayMode testing than unit testing, but this means that we have a way to test actual GameObject interactions when features such as timers and physics are running. As this recipe demonstrates, we can also load our own scenes for PlayMode testing, be they special scenes that have been created just to test interactions or actual scenes that are to be included in our final game build.

## There's more...

There seem to be some changes in how to enable PlayMode tests for all assemblies. The issue is that enabling PlayMode tests can increase the size of build projects – the default setting is to disable PlayMode tests for all assemblies. If your version of the Unity Editor Test Runner window does not offer a menu option to Enable playmode tests for all assemblies, then you can enable these by setting the playModeTestRunnerEnabled setting to 1 in the ProjectSettings/ProjectSetting.asset file. You can learn more about this in the Unity Test Framework documentation: <https://docs.unity3d.com/Packages/com.unity.test-framework@1.1/manual/workflow-create-playmode-test.html>.

# 8 - PlayMode and unit testing a player health bar with events, logging, and exceptions

In this recipe, we will combine many different kinds of tests for a feature that's included in many games – a visual health bar representing the player's numeric health value (in this case, a float number from 0.0 to 1.0). Although far from comprehensively testing all the aspects of the health bar, this recipe will provide a good example of how we can go about testing many different parts of a game using the Unity Testing tools.

A Unity package has been provided that contains the following:

* Player.cs: A player script class for managing values for player health that uses delegates and events to publish health changes to any listening View classes.
* Two View classes that register to listen for player health change events:
  + HealthBarDisplay.cs: This updates fillAmount for a UI Image for each new player health value that's received.
  + HealthChangeLogger.cs: This prints messages about the new player health value that's received by the Debug.Log file.
* PlayerManager.cs: A manager script that initializes player and HealthChangeLogger objects, and also allows the user to change the health of the player by pressing the Up and Down arrow keys (simulating healing/damage during a game).
* A scene that has two UI Images – one is a health bar outline (red heart and a black outline), while the second is the filler image, showing dark blue to light blue to green, for weak to strong health values.

This recipe allows several different kinds of testing to be demonstrated:

* PlayMode testing, to check that the actual fillAmount of the UI Image displayed matches the 0.0 to 1.0 range of the player's health.
* Unit testing, to check that the player's health starts with the correct default value and correctly increases and decreases after calls to the AddHealth(...) and ReduceHealth(...) methods are made.
* Unit testing, to check that health change events are published by the player object.
* Unit testing, to check that expected messages are logged in Debug.Log.
* Unit testing, to check that argument out-of-range exceptions are thrown if negative values are passed to the player's AddHealth(...) or ReduceHealth(...) methods. This is demonstrated in the following screenshot:

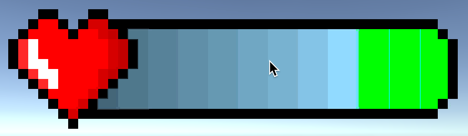


Figure 17.28 – The graphical heath bar we'll be testing in PlayMode

17\_23\_heathbar\_UI.png

Thanks to Pixel Art Maker for the health bar image: <http://pixelartmaker.com/art/49e2498a414f221>.

## Getting ready

For this recipe, a Unity package has been provided (healthBarScene.unitypackage) in the 17\_09 folder.

## How to do it...

To PlayMode and unit test a player health bar, follow these steps:

1. Create a new 3D Unity project and delete the default Project panel folder  Scenes.
2. Import the Unity package provided (healthBarScene.unitypackage).
3. Open the HealthBar scene and add this scene to the project's Build, by choosing menu: **File | Build Settings...**.
4. Display the **Test Runner** window by going to **Window | General | Test Runne**r.
5. We need to enable **PlayMode** tests for all assemblies. Do this by displaying the drop-down menu in the top-right corner of the **Test Runner** window and selecting **EnablePlayMode** tests for all assemblies (click **OK** for any message concerned with restarting the editor).
6. Now, we have to restart the Unity Editor (just close the application and then reopen your project).
7. Since our PlayMode tests make use of the Player script class in the **Project** folder (**Assets | HealthBarScene | \_Scripts**), we need to add an **Assembly Definition** there. Select this folder in the Project panel, then from the **Create** menu, create a new Assembly Definition, naming it PlayerAssembly.

Graphical user interface, application

Description automatically generated

Figure 17.29 – PlayerAssembly created in the \_Scripts folder

17\_23\_heathbar\_UI.png

1. Now, select **PlayMode** in the **Test Runner** window.
2. In the **Project** window, select the top-level folder, called Assets.
3. In the **Test Runner** window, click the **Create PlayMode Test Assembly** **Folder** button. A new folder, named Tests, should have been created.
4. Select the Tests folder. This contains an **Assembly Reference** named Tests. Select this asset file in the Project window and look at its properties in the Inspector window. This Assembly Reference already has references to two Assembly Definition References: UnityEngine.TestRunner and UnityEditor.TestRunner.
5. We now need to add our PlayerAssembly  asset to the Tests assembly so that we can perform our PlayMode tests that make use of the Player script class. In the Inspector window, click the plus (+) button to add a new Assembly Definition Reference. In the slot that's been created, locate and drag the PlayerAssembly  Assembly Definition asset file, then click the **Apply** button at the bottom of the **Inspector** panel.
6. In the **Test Runner** window, click the **Create Test Script in Current Folder** button. Rename this script class HealthBarPlayModeTests and replace its content with the following code:

using UnityEngine;  
 using UnityEngine.UI;  
 using UnityEngine.TestTools;  
 using NUnit.Framework;  
 using System.Collections;  
 using UnityEngine.SceneManagement;  
   
   
 [TestFixture]  
 public class HealthBarPlayModeTests  
 {  
 private Scene tempTestScene;  
   
 // name of scene being tested by this class  
 private string sceneToTest = "HealthBar";  
   
 [SetUp]  
 public void Setup()  
 {  
 // setup - load the scene  
 tempTestScene = SceneManager.GetActiveScene();  
 }  
 }

1. Add the following test to HealthBarPlayModeTests.cs:

[UnityTest]  
 public IEnumerator TestHealthBarImageMatchesPlayerHealth()  
 {  
 // load scene to be tested  
 yield return SceneManager.LoadSceneAsync(sceneToTest, LoadSceneMode.Additive);  
 SceneManager.SetActiveScene(SceneManager.GetSceneByName(sceneToTest));  
   
 // wait for one frame  
 yield return null;  
   
 // Arrange  
 Image healthBarFiller = GameObject.Find("image-health-bar-filler").GetComponent<Image>();  
 PlayerManager playerManager = GameObject.FindWithTag("PlayerManager").GetComponent<PlayerManager>();  
 float expectedResult = 0.9f;  
   
 // Act  
 playerManager.ReduceHealth();  
   
 // Assert  
 Assert.AreEqual(expectedResult, healthBarFiller.fillAmount);  
   
 // teardown - reload original temp test scene  
 SceneManager.SetActiveScene(tempTestScene);  
 yield return SceneManager.UnloadSceneAsync(sceneToTest);  
 }

1. Click **Run All**.
2. As the tests run, you will see that, in the **Hierarchy**, **Game**, and **Scene** windows, a temporary scene is created, then **HealthBarScene** running, with the visual health bar.
3. You should see the results of your **PlayMode** test being executed – if the test concludes successfully, there should be a green tick (checkmark).
4. Now, let's add some unit tests to our player health feature.
5. Select **EditMode** in the **Test Runner** window.
6. In the **Project** window, select the top-level folder, called Assets.
7. In the **Test Runner** window, click the **Create EditMode Test Assembly** **Folder** button. A new folder, named Tests 1, should have been created; rename this TestsEditMode.
8. Select the TestsEditMode folder. This contains an **Assembly Reference** named Tests. Select this asset file in the Project window and look at its properties in the Inspector window. This Assembly Reference already has references to two Assembly Definition References: UnityEngine.TestRunner and UnityEditor.TestRunner.
9. As before, we need to add our PlayerAssembly  asset to the Tests assembly so that we can perform our PlayMode tests that make use of the Player script class. In the **Inspector** window, click the plus (+) button to add a new Assembly Definition Reference. In the slot that's been created, locate and drag the PlayerAssembly  Assembly Definition asset file, then click the **Apply** button at the bottom of the **Inspector** panel.
10. In the **Test Runner** window, click the **Create Test Script in the current folder**button. Rename this script class EditModeUnitTests.
11. Edit the EditModeUnitTests.cs script class, replacing its content with the following code:

using System;  
 using UnityEngine.TestTools;  
 using NUnit.Framework;  
 using UnityEngine;  
   
 public class EditModeUnitTests  
 {  
   
 // inner unit test classes go here  
   
 }

1. Add the following class and basic tests to the EditModeUnitTests class in EditModeUnitTests.cs:

public class TestCorrectValues  
 {  
 [Test]  
 public void DefaultHealthOne()  
 {  
 // Arrange  
 Player player = new Player();  
 float expectedResult = 1;  
   
 // Act  
 float result = player.GetHealth();  
   
 // Assert  
 Assert.AreEqual(expectedResult, result);  
 }  
   
 [Test]  
 public void HealthCorrectAfterReducedByPointOne()  
 {  
 // Arrange  
 Player player = new Player();  
 float expectedResult = 0.9f;  
   
 // Act  
 player.ReduceHealth(0.1f);  
 float result = player.GetHealth();  
   
 // Assert  
 Assert.AreEqual(expectedResult, result);  
 }  
   
 [Test]  
 public void HealthCorrectAfterReducedByHalf()  
 {  
 // Arrange  
 Player player = new Player();  
 float expectedResult = 0.5f;  
   
 // Act  
 player.ReduceHealth(0.5f);  
 float result = player.GetHealth();  
   
 // Assert  
 Assert.AreEqual(expectedResult, result);  
 }  
 }

1. Add the following class and limit test to the EditModeUnitTests class in EditModeUnitTests.cs:

public class TestLimitNotExceeded  
 {  
 [Test]  
 public void HealthNotExceedMaximumOfOne()  
 {  
 // Arrange  
 Player player = new Player();  
 float expectedResult = 1;  
   
 // Act  
 player.AddHealth(1);  
 player.AddHealth(1);  
 player.AddHealth(0.5f);  
 player.AddHealth(0.1f);  
 float result = player.GetHealth();  
   
 // Assert  
 Assert.AreEqual(expectedResult, result);  
 }  
 }

1. Add the following class and event tests to the EditModeUnitTests class in EditModeUnitTests.cs:

public class TestEvents  
 {  
 [Test]  
 public void CheckEventFiredWhenAddHealth()  
 {  
 // Arrange  
 Player player = new Player();  
 bool eventFired = false;  
   
 Player.OnHealthChange += delegate  
 {  
 eventFired = true;  
 };  
   
 // Act  
 player.AddHealth(0.1f);  
   
 // Assert  
 Assert.IsTrue(eventFired);  
 }  
   
 [Test]  
 public void CheckEventFiredWhenReduceHealth()  
 {  
 // Arrange  
 Player player = new Player();  
 bool eventFired = false;  
   
 Player.OnHealthChange += delegate  
 {  
 eventFired = true;  
 };  
   
 // Act  
 player.ReduceHealth(0.1f);  
   
 // Assert  
 Assert.IsTrue(eventFired);  
 }  
 }

1. Add the following class and exception tests to the EditModeUnitTests class in EditModeUnitTests.cs:

public class TestExceptions  
 {  
 [Test]  
 public void Throws\_Exception\_When\_Add\_Health\_Passed\_Less\_Than\_Zero()  
 {  
 // Arrange  
 Player player = new Player();  
   
 // Act  
   
 // Assert  
 Assert.Throws<ArgumentOutOfRangeException>(  
 delegate  
 {  
 player.AddHealth(-1);  
 }  
 );  
 }  
   
 [Test]  
 public void Throws\_Exception\_When\_Reduce\_Health\_Passed\_Less\_Than\_Zero()  
 {  
 // Arrange  
 Player player = new Player();  
   
 // Act  
   
 // Assert  
 Assert.Throws<ArgumentOutOfRangeException>(  
 () => player.ReduceHealth(-1)  
 );  
 }  
 }

1. Add the following class and logging tests to the EditModeUnitTests class in EditModeUnitTests.cs:

public class TestLogging  
 {  
 [Test]  
 public void Throws\_Exception\_When\_Add\_Health\_Passed\_Less\_Than\_Zero()  
 {  
 Debug.unityLogger.logEnabled = true;  
   
 // Arrange  
 Player player = new Player();  
 HealthChangeLogger healthChangeLogger = new HealthChangeLogger();  
 string expectedResult = "health = 0.9";  
   
 // Act  
 player.ReduceHealth(0.1f);  
   
 // Assert  
 LogAssert.Expect(LogType.Log, expectedResult);  
 }  
 }

*You can see that the inner classes allow us to group the unit tests visually in the Test Runner window:*

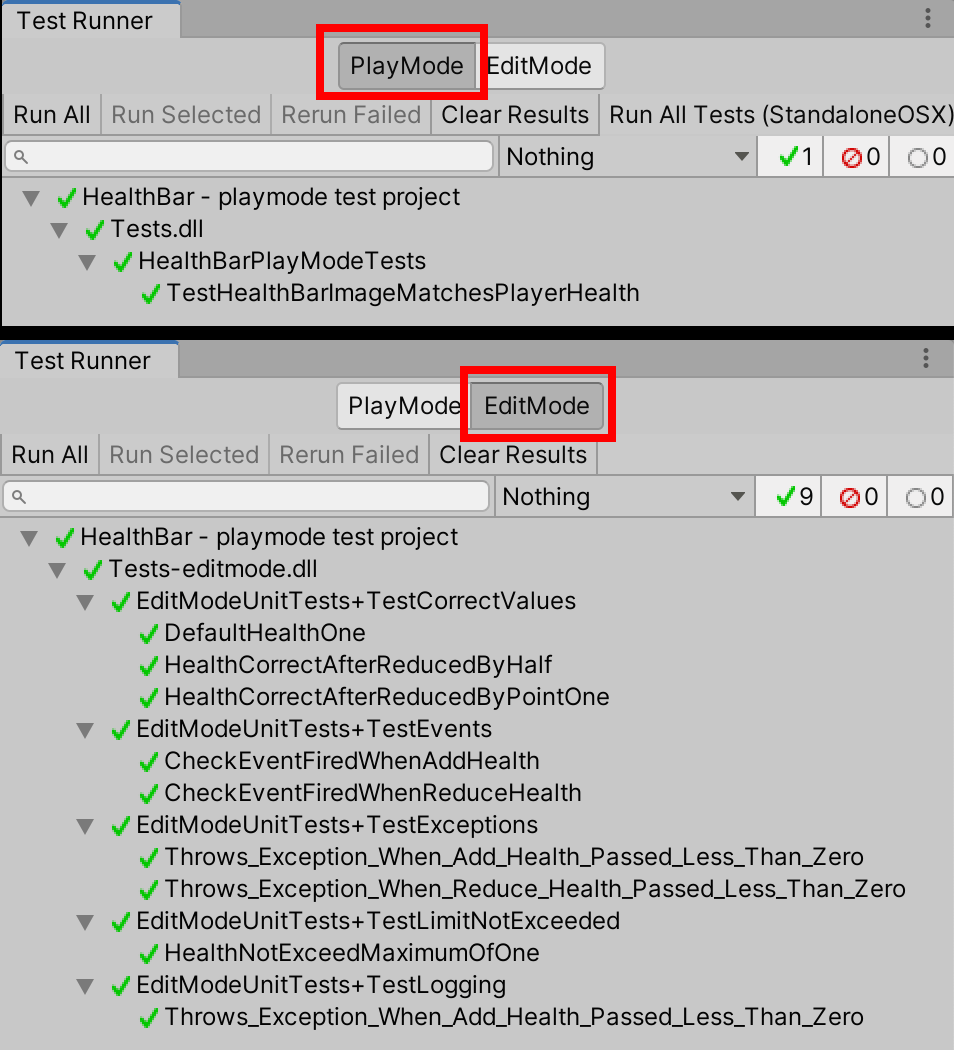


Figure 17.30 – PlayMode and EditMode test results - all of which passed (green ticks)

17\_23\_heathbar\_UI.png

## How it works...

Let's take a look at how this recipe works in detail.

### PlayMode testing

The PlayMode test called TestHealthBarImageMatchesPlayerHealth() loads the HealthBar scene, gets a reference to the instance object of PlayerManager, which is a component of the PlayerManager GameObject, and invokes the ReduceHealth() method. This method reduces the player's health by 0.1. So, from its starting value of 1.0, it becomes 0.9.

The PlayerManager GameObject also has as a component that's an instance object of the C# HealthBarDisplay script class. This object registers to listen to published events from the player class. It also has a public UI Image variable that has been linked to the UI Image of the health bar filler image in the scene.

When the player's health is reduced to 0.9, it publishes the OnChangeHealth(0.9) event. This event is received by the HealthBarDisplay object instance, which then sets the fillAmount property of the linked health bar filler image in the scene.

The TestHealthBarImageMatchesPlayerHealth() PlayMode test gets a reference to the object instance named image-health-bar-filler, storing this reference in the healthBarFiller variable. The test assertion that's made is that the expectedResult value of 0.9 matches the actual fillAmount property of the UI Image in the scene:

Assert.AreEqual(expectedResult, healthBarFiller.fillAmount);

### Unit tests

There are several unit tests that can be grouped by placing them inside their own classes, inside the EditModeUnitTests script class.

* TestCorrectValues:
  + DefaultHealthOne(): This tests that the default (initial value) of the player's health is 1.
  + HealthCorrectAfterReducedByPointOne(): This tests that when the player's health is reduced by 0.1, it becomes 0.9.
  + HealthCorrectAfterReducedByHalf(): This tests that when the player's health is reduced by 0.5, it becomes 0.5.
* TestLimitNotExceeded:
  + HealthNotExceedMaximumOfOne(): This tests that the value of the player's health does not exceed 1, even after attempts to add 1, 0.5, and 0.1 to its initial value of 1.
* TestEvents:
  + CheckEventFiredWhenAddHealth(): This tests that an OnChangeHealth() event is published when the player's health is increased.
  + CheckEventFiredWhenReduceHealth(): This tests that an OnChangeHealth() event is published when the player's health is decreased.
* TestLogging:
  + CorrectDebugLogMessageAfterHealthReduced(): This tests that a Debug.Log message is correctly logged after the player's health is reduced by 0.1 to 0.9.
* TestExceptions:
  + Throws\_Exception\_When\_Add\_Health\_Passed\_Less\_Than\_Zero(): This tests that an ArgumentOutOfRangeException is thrown when a negative value is passed to the AddHealth(...) player method.
  + Throws\_Exception\_When\_Reduce\_Health\_Passed\_Less\_Than\_Zero(): This tests that an ArgumentOutOfRangeException is thrown when a negative value is passed to the ReduceHealth(...) player method.

*These two tests illustrate one convention of naming tests that adds an underscore (\_) character between each word in the method name in order to improve readability.*

## See also

You can learn more about the LogAssert Unity Script reference in the Unity documentation: <https://docs.unity3d.com/ScriptReference/TestTools.LogAssert.html>.

The method for unit testing C# events has been adapted from a post on philosophicalgeek.com: <http://www.philosophicalgeek.com/2007/12/27/easily-unit-testing-event-handlers/>.

The delegate-event publishing of health change events in this health bar feature is an example of the Publisher-Subscriber design pattern.

# 10 - Reporting Code Coverage testing

A useful tool in projects with code testing is to be able to analyze how much of a C# script class is being tested. For example, is every method being tested with at least 1 set of test data? Unity now offers a Code Coverage feature, which we'll explore in this final code testing recipe. As shown in the following screenshot, Unity allows us to create a set of HTML pages for documenting the Code Coverage of tests against C# code. With this, we can see what percentage of our code is covered by tests, and even which lines of code are, and are not, covered by our tests:

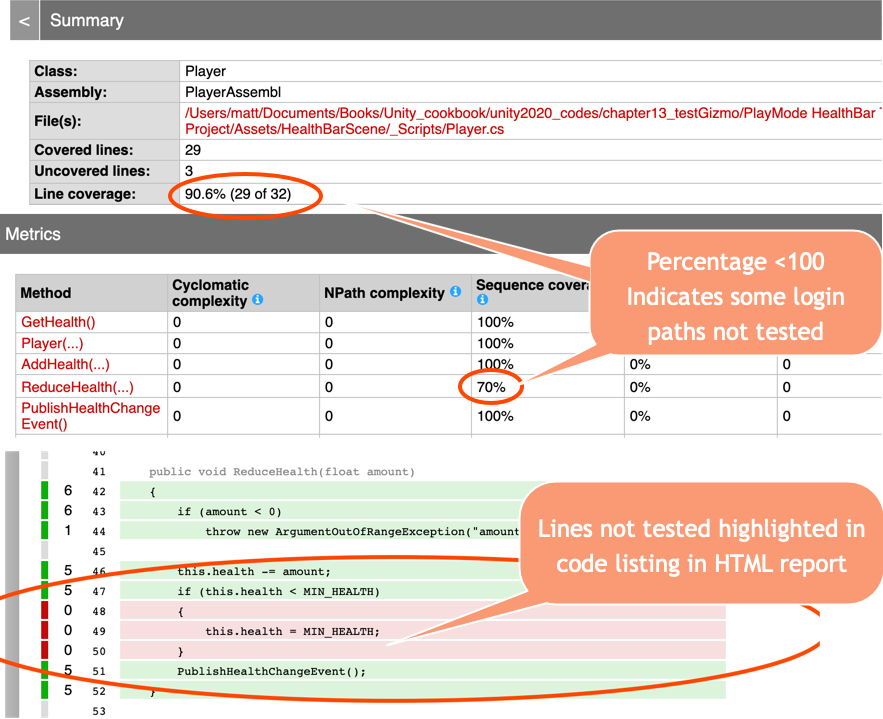


Figure 17.31 – Code Coverage HTML report for the Player script class

17\_26\_coverageReport.png

## Getting ready

This project builds on the previous one, so make a copy of that and work on the copy.

## How to do it...

To add Code Coverage reporting to a project with unit tests, follow these steps:

1. Open the **Code Coverag**e window by going to **Window | Analysis | Code Coverage**.
2. In the **Code Coverage** window, click the Switch to Debug Mode button.
3. Next then check the **Enable Code Coverage** option. Then, ensure the **HTML Report** and **Auto Generate Report** options are checked in the **`** section.

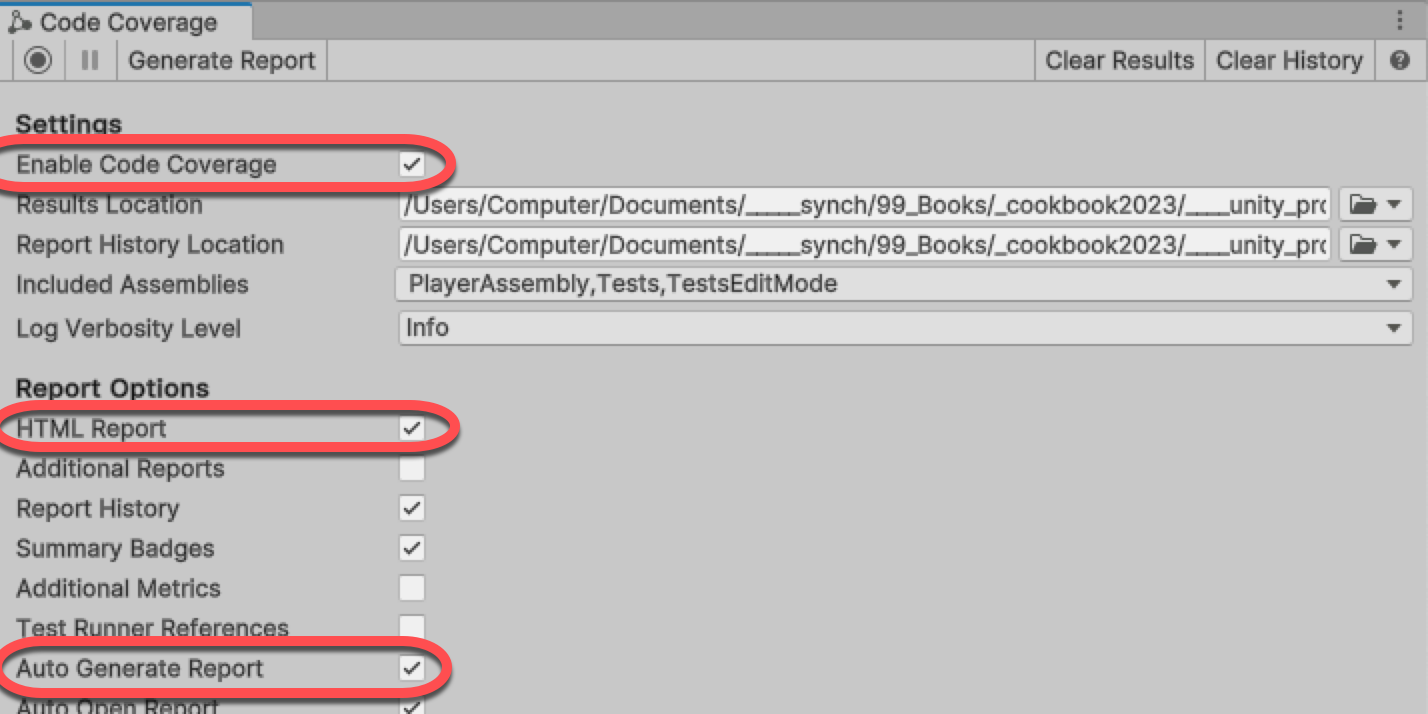


Figure 17.33 – Code Coverage window options

13\_09\_code\_coverageHTML.png

1. In the **Test Runner**, choose **PlayMode** and click the **Run All Tests** button.
2. Once the testing is finished, an HTML code coverage report should automatically be created and opened in your computer's default web browser application. The report will be created in a new folder named Report inside a folder named Code Coverage in your Unity project folder.

## How it works...

The Unity debugger monitors which lines of code are being executed as the tests are run. It uses this data to compile a report on how much of each method of each class has been executed for the assemblies involved in the testing. By adding the Code Coverage package and enabling it, debugging data will be collected and reported upon when you run Unity tests.

As shown in the figure above the Unity **Code Coverage** tool generates a set of web pages to inform us about how much and which lines of our code are being examined with our unit tests. While even 100% coverage does not guarantee the code is "correct," a high percentage of code coverage does indicate that the behavior of most of our code is being tested to some extent.

# Running simple Python scripts inside Unity

Unity Python is a package that allows Python code to be executed as part of a Unity project. In this recipe, we'll install the package, test the Python Script Editor window with a traditional Hello World Python print statement, and create C# scripts to run Python based on the examples provided by Unity at <https://docs.unity3d.com/Packages/com.unity.scripting.python@4.0/manual/inProcessAPI.html>.

## How to do it...

To run simple Python scripts inside Unity, follow these steps:

1. Create a new 2D project.
2. Open the **Package Manager** by choosing menu: **Window | Package Manager**.
3. Set the list of packages to those in the **Unity Registry**, and search for **Python Scripting**. Then click **Install**.

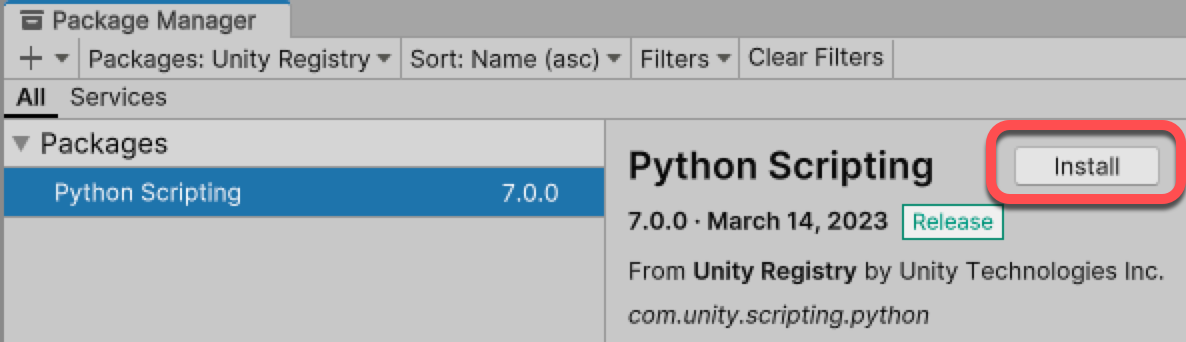


Figure 17.34 – Adding the Python Scripting package to a project

13\_06\_package\_python.png

1. Open the Python Script Editor panel by choosing menu: **Window | General | Python Console**.
2. Enter print ('Hello World from Python') in the editor section (lower half) of the Python Script Editor window and click the Execute button. You should see a message stating Hello World appear in the output (top section) of the window:

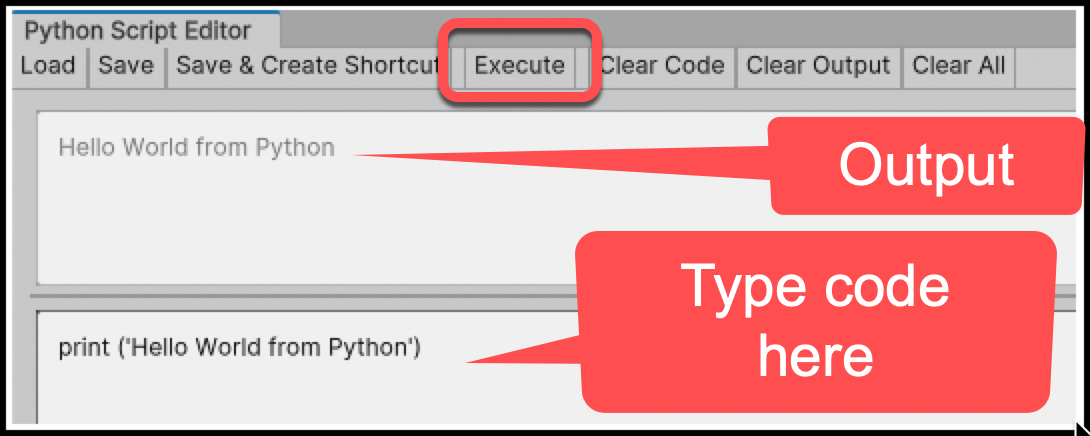


Figure 17.35 – Executing a Hello World statement in the Python Script Editor window

13\_07\_python\_hello\_world.png

1. Now, let's create a C# script that can execute Python in a string using the Python API. Create a \_Scripts folder in the **Project** panel. Inside it, a new C# script class file named HelloConsole.cs that contains the following code:

using UnityEditor.Scripting.Python;  
using UnityEditor;  
  
public class HelloConsole  
{  
 [MenuItem("My Python/Hello Console")]  
 static void PrintHelloWorldFromPython()  
 {  
 PythonRunner.RunString(@"  
 import UnityEngine;  
 UnityEngine.Debug.Log('hello console')  
 ");  
 }  
}

1. You should now see a new menu named **My Python** with an item called **Hello Console**. When you select this menu item, a message stating hello console should be output to the Console window.
2. Now, let's create a text file in our \_Scripts folder containing pure Python – you may need to create this text file outside the Unity Editor by navigating to the \_Scripts folder and using a text editor application to create and save this new file. Create a new text file named renamer.py that contains the following code:

import UnityEngine  
  
all\_objects = UnityEngine.Object.FindObjectsOfType(UnityEngine.GameObject)  
for go in all\_objects:  
 if go.name[-1] != '\_':  
 go.name = go.name + '\_'

1. Now, let's create a C# script that offers a menu item that will execute our Python script file. Create a new C# script class file called InvokeRenamer.cs that contains the following code:

using UnityEditor.Scripting.Python;  
using UnityEditor;  
using UnityEngine;  
  
public class InvokeRenamer  
{  
 [MenuItem("My Python/Underscore Renamer")]  
 static void RunEnsureNaming()  
 {  
 PythonRunner.RunFile($"{Application.dataPath}/\_Scripts/renamer.py");  
 }  
}

1. On the **My Python** menu item, there should now be a second menu item named **Underscore Renamer**. When you select this menu item, you should see that all the GameObjects in the **Hierarchy** panel now end with underscore characters.

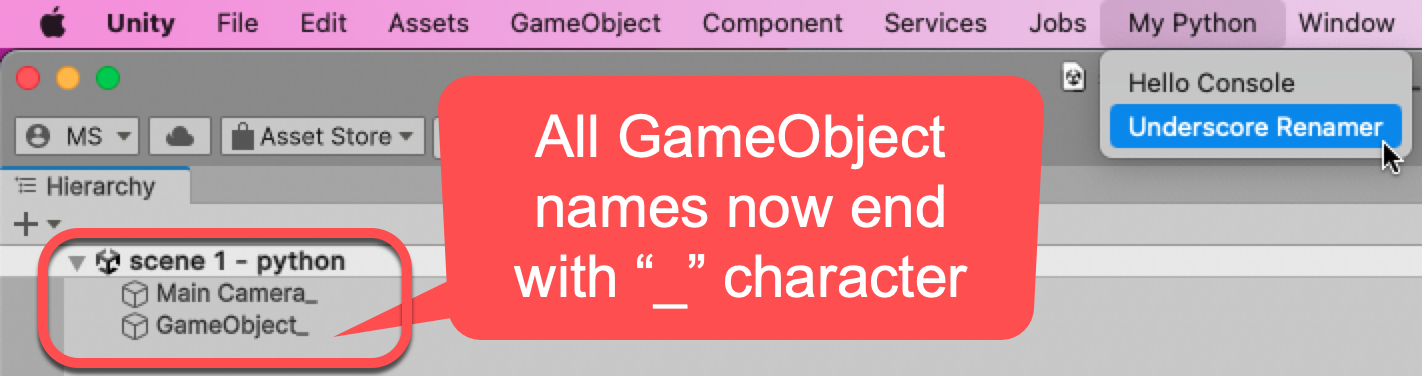


Figure 17.36 – The My Python menu items and the renamed Hierarchy window GameObjects

13\_08\_python\_underscore\_menu.png

## How it works...

The Python Scripting package adds an API (code library) that we can access through our C# code, as well as the **Python Script Editor** window for testing and running simple Python statements. We tested **Python Script Editor** with a traditional Hello World from Python print statement.

Our C# script class HelloConsole demonstrated how we can use the PythonRunner.RunString($...) method in our C# scripts to execute a string containing Python code. Note that we must add a using statement in out Python code to import the UnityEditor.Scripting.Python library for this work.

***Note****: The @-sign in our* HelloConsole *C# script class allowed us to write several lines of Python code in a single string variable declaration. This sign means that the string contents are treated as a* ***verbatim string literal****, meaning any special characters such as new lines or slashes etc. are considered part of the string inside the two double quotation marks.*

In both our scripts, we used the [MenuItem("<menu/item>")] Editor instruction to enable us to easily test out code from a menu item.

Our InvokeRenamer C# script class demonstrated how we can use the PythonRunner.RunFile($...) method in our C# scripts to execute a text file containing Python code. We created a file called renamer.py containing pure Python code in order to loop through all the GameObjects. We added an underscore suffix if they didn't already have such a suffix. We followed the convention of using the .py file extension for files containing just Python code.

Notice how we indicated the location of the renamer.py file by writing {Application.dataPath}/\_Scripts/ before the file name. We could change this to indicate a different location for the Python file we wish to execute.

In both our Python string and file, we were able to make use of the **UnityEngine** library (assembly) by writing import UnityEngine before our Python statements – this works just like the C# using statement.

# Further reading

You can learn more about Gizmos at the following links:

* the Unity Gizmos manual entry at <https://docs.unity3d.com/ScriptReference/Gizmos.html>.
* Unity Gizmos tutorial: <https://learn.unity.com/tutorial/creating-custom-gizmos-for-development-2019-2>

You can learn more about Unity testing at the following links:

* Unity Test Framework Manual documentation: <https://docs.unity3d.com/Manual/testing-editortestsrunner.html>
* Unity Test Framework package documentation: <https://docs.unity3d.com/Packages/com.unity.test-framework@1.1/manual/index.html>
* Unity Test Framework how-to pages: <https://unity.com/how-to/unity-test-framework-video-game-development>
* A website for the book The Art of Unit Testing (and lots of other learning resources associated with testing): <http://artofunittesting.com/>
* A great dual article tutorial about Unity testing by Tomek Paszek from Unity (talking about the old Unity test tools, but most of the content is still very relevant): <https://blogs.unity3d.com/2014/06/03/unit-testing-part-2-unit-testing-monobehaviours/>
* YouTube, where you can learn lots about Unity testing (and other topics) from Infalliblecode: <https://www.youtube.com/infalliblecode>
* CodeProject.com's introduction to TDD and NUnit: <https://www.codeproject.com/Articles/162041/Introduction-to-NUnit-and-TDD>
* A great tutorial about unit testing by Anthony Uccello on Kodeco (formally Ray Wenderlich): <https://www.raywenderlich.com/9454-introduction-to-unity-unit-testing>
* The Code Coverage features of the Unity Test tools: <https://docs.unity3d.com/Packages/com.unity.testtools.codecoverage@1.0/manual/>

Testing in Unity usually involves **Unity Assemblies**. This is an approach to separating the components of a game into separate modules. A great introduction to Unity Assemblies by Erdiizgi can be found at <https://erdiizgi.com/why-modular-game-development-and-how-to-do-it-with-unity/>.

You can learn more about Unity Python by reading the official package documentation: <https://docs.unity3d.com/Packages/com.unity.scripting.python@4.0/manual/inProcessAPI.html>.