12
Editor Extensions

In this chapter, we will cover:

- Editor extension to allow pickup type (and parameters) to be changed at design-time via a custom Inspector UI
- Editor extension to add 100 randomly located copies of a prefab with one menu click
- Displaying a progress bar to display Editor extensions progress completed
- Editor extension to have an object creator game object, with buttons to instantiate different pickups at cross-hair object location in scene

Introduction

One aspect of game development in general (and inventories as our particular examples in this chapter), is the distinction about WHEN we undertake an activity. **Run-time** is when the game is running (and when all our software and UI choices take affect). However, **Design-time** is the time when different members of our game design team work on constructing a wide range of game components, including the scripts, audio and visual assets, and the process of constructing each game level (or 'scene' in Unity-speak).

In this chapter we have introduced several recipes that make use of Unity's Editor extensions – these are scripting and multimedia components that enable a game software engineer to make design-time work easier, and less likely to introduce errors. Editor extensions allow workflow improvements, thus allowing designers to achieve their goals quicker and easier – e.g. removing the need for any scripting knowledge when generating many randomly located inventory pickups in a scene via a menu choice, or editing the type or properties of pickups being hand-placed in different locations in a level.

While the Editor extensions are quite an advanced topic, having someone on your team who can write custom editor components, such as those we illustrate, can greatly increase

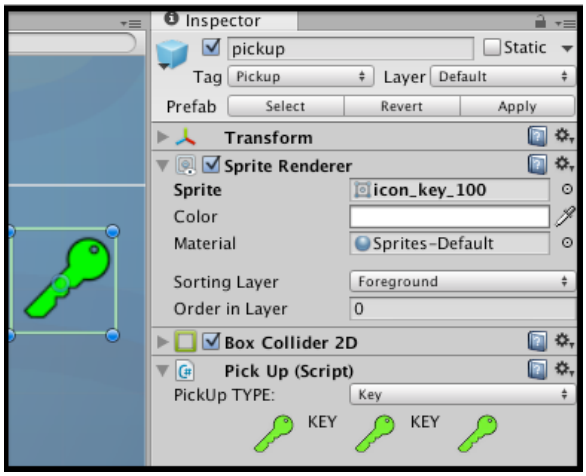
the productivity of a small team with only one or two members who are confident at scripting.

Editor extension to allow pickup type (and parameters) to be changed at design-time via a custom Inspector UI

The use of enums and corresponding drop-down menus in the Inspector panel to restrict changes to one of a limited set often works fine (for example pickup types for a pickup object). However, the trouble with this approach is when two or more properties are related and need to be changed together, so there is the danger of changing one property, e.g. pickup type from Heart to Key, but forgetting to change corresponding properties, e.g. leaving the Sprite Renderer component still showing a Heart sprite. Such mismatches cause problems both in terms of messing up intended level design, and of course the frustration of the player when they collide with something showing one pickup image, but a different kind of pickup type is added to the inventory!

If a class of gameObject has several related properties or components, that all need to be changed together, then a good strategy is to use Unity Editor extensions to do all the associated changes, each time a different choice is made from a drop-down menu showing the defined set of enumerated choices.

In this recipe we introduce an Editor extension for PickUp components of gameObjects.



Insert image 1362OT_12_41.png

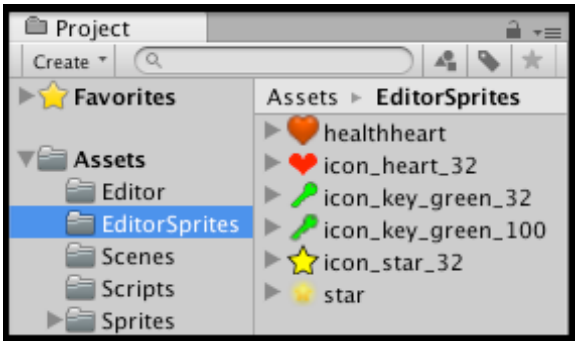
Getting ready

This recipe assumes you are starting with project `Simple2Dgame_SpaceGirl` setup from the first recipe in *Chapter 2 Inventory GUIs*. A copy of this Unity project is provided in a folder named `unityProject_spaceGirlMiniGame` in folder `1362_12_01`.

How to do it...

To create an editor extension to allow pickup type (and parameters) to be changed at design-time via a custom Inspector UI follow these steps:

1. Start with a new copy of mini-game `Simple2Dgame_SpaceGirl`.
2. In the **Project** panel create a new folder named `EditorSprites`. Move the following images from folder Sprites into this new folder: `star`, `healthheart`, `icon_key_green_100`, `icon_key_green_32`, `icon_star_32` and `icon_heart_32`.



Insert image 1362OT_12_55.png

3. In the **Hierarchy** panel rename gameObject `star` to be named `pickup`.
4. Edit the tags, changing tag **Star** to **Pickup**. Ensure the `pickup` gameObject now has the tag **Pickup**.
5. Add the following C# Script `PickUp` to gameObject `pickup` in the **Hierarchy**:

```
using UnityEngine;
using System;
using System.Collections;

public class PickUp : MonoBehaviour {
   public enum PickUpType {
      Star, Health, Key
   }

   [SerializeField]
```

```
    public PickUpType type;

    public void SetSprite(Sprite newSprite){
        SpriteRenderer spriteRenderer =
GetComponent<SpriteRenderer>();
        spriteRenderer.sprite = newSprite;
    }
}
```

6. In the **Project** panel, create a new folder named `Editor`. Inside this new folder create a new C# script class named `PickUpEditor`, with the following code:

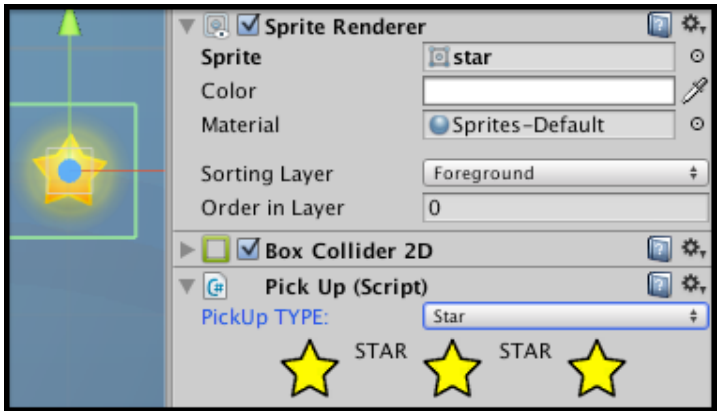
```
using UnityEngine;
using System.Collections;
using System;
using UnityEditor;
using System.Collections.Generic;

[CanEditMultipleObjects]
[CustomEditor(typeof(PickUp))]
public class PickUpEditor : Editor
{
    public Texture iconHealth;
    public Texture iconKey;
    public Texture iconStar;

    public Sprite spriteHealth100;
    public Sprite spriteKey100;
    public Sprite spriteStar100;

    UnityEditor.SerializedProperty pickUpType;

    private Sprite sprite;
    private PickUp pickupObject;

    void OnEnable () {
        iconHealth =
AssetDatabase.LoadAssetAtPath("Assets/EditorSprites/icon_he
art_32.png", typeof(Texture)) as Texture;
        iconKey =
AssetDatabase.LoadAssetAtPath("Assets/EditorSprites/icon_ke
y_32.png", typeof(Texture)) as Texture;
        iconStar =
AssetDatabase.LoadAssetAtPath("Assets/EditorSprites/icon_st
ar_32.png", typeof(Texture)) as Texture;
```

```
      spriteHealth100 =
AssetDatabase.LoadAssetAtPath("Assets/EditorSprites/healthh
eart.png", typeof(Sprite)) as Sprite;
      spriteKey100 =
AssetDatabase.LoadAssetAtPath("Assets/EditorSprites/icon_ke
y_100.png", typeof(Sprite)) as Sprite;
      spriteStar100 =
AssetDatabase.LoadAssetAtPath("Assets/EditorSprites/star.pn
g", typeof(Sprite)) as Sprite;

      pickupObject = (PickUp)target;
      pickUpType = serializedObject.FindProperty ("type");
   }


   public override void OnInspectorGUI()
   {
      serializedObject.Update ();

      string[] pickUpCategories = TypesToStringArray();
      pickUpType.enumValueIndex =
EditorGUILayout.Popup("PickUp TYPE: ",
pickUpType.enumValueIndex, pickUpCategories);

      PickUp.PickUpType type =
(PickUp.PickUpType)pickUpType.enumValueIndex;
      switch(type)
      {
      case PickUp.PickUpType.Health:
         InspectorGUI_HEALTH();
         break;

      case PickUp.PickUpType.Key:
         InspectorGUI_KEY();
         break;

      case PickUp.PickUpType.Star:
      default:
         InspectorGUI_STAR();
         break;
      }

      serializedObject.ApplyModifiedProperties ();
   }

   private void InspectorGUI_HEALTH()
   {
```

```
        GUILayout.BeginHorizontal();
        GUILayout.FlexibleSpace();
        GUILayout.Label(iconHealth);
        GUILayout.Label("HEALTH");
        GUILayout.Label(iconHealth);
        GUILayout.Label("HEALTH");
        GUILayout.Label(iconHealth);
        GUILayout.FlexibleSpace();
        GUILayout.EndHorizontal();

        pickupObject.SetSprite(spriteHealth100);
    }

    private void InspectorGUI_KEY()
    {
        GUILayout.BeginHorizontal();
        GUILayout.FlexibleSpace();
        GUILayout.Label(iconKey);
        GUILayout.Label("KEY");
        GUILayout.Label(iconKey);
        GUILayout.Label("KEY");
        GUILayout.Label(iconKey);
        GUILayout.FlexibleSpace();
        GUILayout.EndHorizontal();

        pickupObject.SetSprite(spriteKey100);
    }

    private void InspectorGUI_STAR()
    {
        GUILayout.BeginHorizontal();
        GUILayout.FlexibleSpace();
        GUILayout.Label(iconStar);
        GUILayout.Label("STAR");
        GUILayout.Label(iconStar);
        GUILayout.Label("STAR");
        GUILayout.Label(iconStar);
        GUILayout.FlexibleSpace();
        GUILayout.EndHorizontal();

        pickupObject.SetSprite(spriteStar100);
    }

    private string[] TypesToStringArray(){
```

```
        var pickupValues =
(PickUp.PickUpType[])Enum.GetValues(typeof(PickUp.PickUpTyp
e));

        List<string> stringList = new List<string>();

        foreach(PickUp.PickUpType pickupValue in
pickupValues){
            string stringName = pickupValue.ToString();
            stringList.Add(stringName);
        }

        return stringList.ToArray();
    }
}
```

7. In the **Inspector** select gameObject `pickup`, and choose different values of the drop-down menu **PickUp TYPE**. You should see corresponding changes in the image and icons in the **Inspector** for the **Pick Up (Script)** component (3 icons with the name of the type in between). The **Sprite** property of the **Sprite Renderer** component for this gameObject should change. Also, in the **Scene** panel you'll see the image in the scene change to the appropriate image for the pickup type you have chosen.



Insert image 1362OT_12_42.png

How it works...

Our script class `PickUp` has the enum `PickUpType` with the 3 values: `Star`, `Health` and `Key`. Also there is the variable type, storing the type for the gameObject an instance of this script appears as a component within. Finally there is a method `SetSprite(…)`

which sets the **Sprite Renderer** component of the parent gameObject to be set to the provided `Sprite` parameter. Is it this method that is called from the editor script each time the pickup type is changed from the drop-down menu (with the corresponding Sprite for the new type being passed).

The vast majority of the work for this recipe is the responsibility of the script class `PickUpEditor`. While there is a lot in this script, its work is relatively straightforward – each frame – via method `OnInspectorGUI()` – a dropdown list of `PickUpType` values is presented to the user. Based on the value selected from this drop-down list, one of three methods is executed: `InspectorGUI_HEALTH(), InspectorGUI_KEY(), InspectorGUI_STAR()`. Each of these methods displays 3 icons and the name of the type in the Inspector beneath the drop-down menu, and ends by calling the `SetSprite(…)` method of the gameObject being edited in the Inspector, to update the **Sprite Renderer** component of the parent gameObject with the appropriate sprite.

The C# attribute `[CustomEditor(typeof(PickUp))]` appearing before our class is declared, tells Unity to use this special Editor script to display component properties in the **Inspector** panel for Pickup (Script) components of gameObjects, rather than the Unity's default **Inspector** displays for public variables of such scripted components.

Before and after its main work, method `OnInspectorGUI()` – first ensures any variables relating to the object being edited in the Inspector have been updated – `serializedObject.Update()`. And the last statement of this method correspondingly ensures that any changes to varaibles in the Editor script have been copied back to the gameObject being edited – `serializedObject.ApplyModifiedProperties()`.

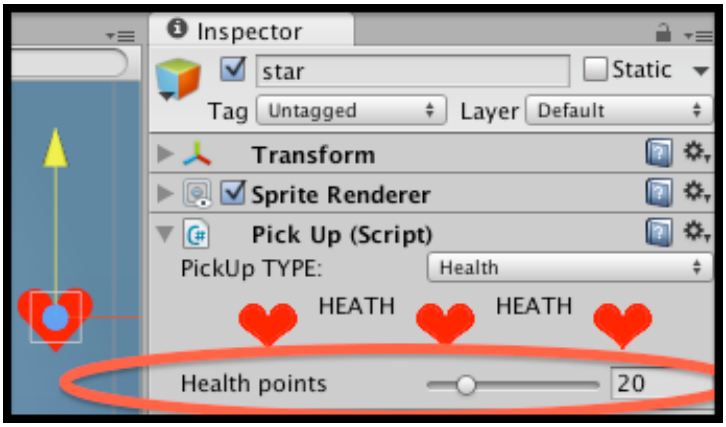
Method `OnEnable()` of script class `PickUpEditor` loads the 3 small icons (for display in the **Inspector**), and the 3 larger Sprite images (to update the **Sprite Renderer** for display in the **Scene/Game** panels). Variable `pickupObject` is set to be a reference to the scripted component `PickUp`, allowing us to call the `SetSprite(…)` method. Variable `pickUpType` is set to be linked to the type variable of the `PickUp` scripted component whose special **Inspector** Editor view this script makes possible – `serializedObject.FindProperty ("type")`.

There's more...

Some details you don't want to miss:

Offer custom editing of pickup parameters via Inspector

Many pickups have some properties, rather than simply being an item being carried. For example, a health pickup may add health 'points' to the player's character, and a coin pickup may add money 'points' to the characters bank balance etc. So let's add an integer `points` variable to our PickUp class, and offer the user the ability to easily editor this points value via a UI slider in our customer Inspector editor.



Insert image 1362OT_12_49.png

To add editable points property to our PickUp objects follow these steps:

1. Add the following extra line into C# Script `PickUp` to create our new integer `points` variable:

```
public int points;
```

2. Add the following extra line into C# Script `PickUpEditor` to work with our new integer `points` variable:

```
UnityEditor.SerializedProperty points;
```

3. Add the following extra line into method `OnEnable()` in C# Script `PickUpEditor` to associate our new `points` variable with its corresponding value in the PickUp scripted component of the gameObject being edited:

```
void OnEnable () {
    points = serializedObject.FindProperty ("points");
    pickUpType = serializedObject.FindProperty ("type");
    // rest of method as before …
```

4. Now we can add an extra line into each GUI method for the different PickUp types. For example we can add a statement to display an **IntSlider** to the user to be able to see and modify the points value for a Health PickUp object. We add a new statement at the end of method `InspectorGUI_HEALTH()` in C# Script `PickUpEditor` to display a modifiable **IntSlider** representing our new `points` variable as follows:

```
private void InspectorGUI_HEALTH(){
    // beginning of method just as before …

    pickupObject.SetSprite(spriteHealth100);

    // now display Int Slider for points
```

```
    points.intValue = EditorGUILayout.IntSlider ("Health
points", points.intValue, 0, 100);
}
```

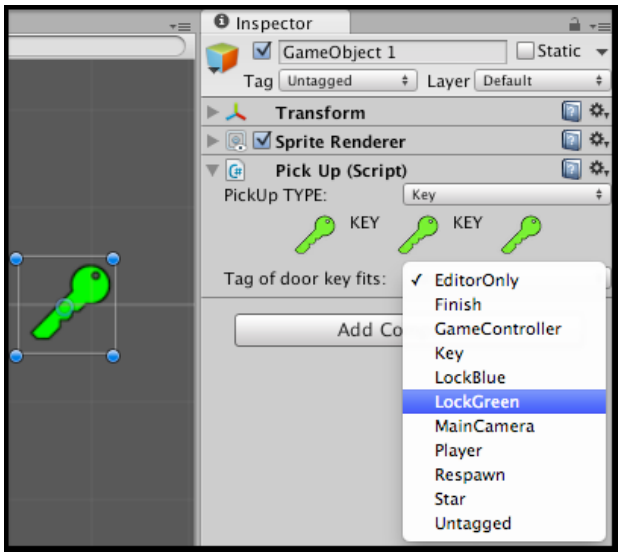
We provide 4 parameters to method `IntSlider(…)`. The first is the text label the user will see next to the slider. The second is the initial value the slider show display. The last two are the maximum and minimum values. In our example we are permitting values from 0 to 100, but if health pickups only offer 1, 2 or 3 healthpoints, then we'd just call with: `EditorGUILayout.IntSlider ("Health points", points.intValue, 1, 5)`. This method returns a new integer value based on where the slider has been positioned, and this new value is stored back into the integer value part of our `SerializedProperty` variable `points`.

Note, the loading and saving of values from the scripted component in the gameObject and our Editor script is all part of the work undertaken by our calls to `Update()` and `ApplyModifiedProperties()` on the serialized object in method `OnInspectorGUI()`.

Note, since points may not have any meaning for some pickups, e.g. keys, then we simply would not display any slider for the GUI Inspector editor when the user is editing PickUp objects of that type.

Offer drop-down list of tags for key-pickup to fit via Inspector

While the concept of ‘points’ may have no meaning for a key pickup, the concept of the type of lock that a given key fits is certainly something we may wish to implement in a game. Since Unity offers us defined (and editable) list of string tags for any gameObject, often it is sufficient, and straightforward, to represent the type of a lock or door corresponding to a key via its tag. For example, a green key might fit all objects tagged LockGreen, and so on.



Insert image 1362OT_12_50.png

Therefore, it is very useful to be able to offer a custom Inspector editor for a string property of key pickups, that stores the tag of the lock(s) the key can open. This task combines several actions, including using C# to retrieve an array of tags from the Unity editor, then the building and offering of a drop-down list of these tags to the user, with the current value already selected in this list.

To add an selectable list of strings for the tag for lock(s) that a key fits follow these steps:

1. Add the following extra line into C# Script `PickUp` to create our new integer `fitsLockTag` variable:

```
public string fitsLockTag;
```

2. Add the following extra line into C# Script `PickUpEditor` to work with our new integer `fitsLockTag` variable:

```
UnityEditor.SerializedProperty fitsLockTag;
```

3. Add the following extra line into method `OnEnable()` in C# Script `PickUpEditor` to associate our new `fitsLockTag` variable with its corresponding value in the PickUp scripted component of the gameObject being edited:

```
void OnEnable () {
   fitsLockTag = serializedObject.FindProperty
("fitsLockTag");
   points = serializedObject.FindProperty ("points");
   pickUpType = serializedObject.FindProperty ("type");
   // rest of method as before …
```

4. Now we need to add some extra lines of code into they GUI method for key PickUps. We need to add several statements to the end of method `InspectorGUI_KEY()` in C# Script `PickUpEditor` to setup and display a selectable **Popup** drop-down list representing our new `fitsLockTag` variable as follows. Replace method `InspectorGUI_KEY()` with the following code:

```
private void InspectorGUI_KEY() {
   GUILayout.BeginHorizontal();
   GUILayout.FlexibleSpace();
   GUILayout.Label(iconKey);
   GUILayout.Label("KEY");
   GUILayout.Label(iconKey);
   GUILayout.Label("KEY");
   GUILayout.Label(iconKey);
   GUILayout.FlexibleSpace();
   GUILayout.EndHorizontal();

   pickupObject.SetSprite(spriteKey100);

   string[] tags =
UnityEditorInternal.InternalEditorUtility.tags;
   Array.Sort(tags);
   int selectedTagIndex = Array.BinarySearch(tags,
fitsLockTag.stringValue);
   if(selectedTagIndex < 0) selectedTagIndex = 0;
   selectedTagIndex = EditorGUILayout.Popup("Tag of door
key fits: ", selectedTagIndex, tags);

   fitsLockTag.stringValue = tags[selectedTagIndex];
}
```

We've added several statements to the end of this method. First `tags`, an array of strings is created (and sorted), containing the list of tags currently available in the Unity editor for the current game. We then attempt to find the location in this array of the current value of `fitsLockTag` – we can use the `BinarySearch(…)` method of built-in script class Array because we have alphabetically sorted our Array (which also makes it easier for the user to navigate). If the string in `fitsLockTag` cannot be found in array `tags` then the first item will be selected by default (index 0).

The user is then shown the drop-down list via `GUILayout` method `EditorGUILayout.Popup(…)`, and this method returns the index of whichever item is selected. The selected index is stored into `selectedTagIndex`, and the last statement in the method extracts the corresponding string and stores that string into variable `fitsLockTag`.

Logic to open doors with keys based on `fitsLockTag`

In our Player collision logic, we can now search through our inventory to see if any key items fit the lock we have collided with. For example, if a green door were collided with, and the player was carrying a key that could open such doors, then that item should be removed from the inventory `List<>` and the door be opened.

To implement this you would need to add an `IF`-test inside `OnTriggerEnter()` to detected collision with the item tagged `Door`, and then logic to attempt to open the door, and if unsuccessful do the appropriate action (e.g. play sound) to inform the player they cannot open the door yet (we'll assume we have written a door animation controller that plays the appropriate animation and sounds etc. when a door is to be opened).

```
if("Door" == hitCollider.tag){
   if(!OpenDoor(hitCollider.gameObject))
      DoorNotOpenedAction();
}
```

Method `OpenDoor()` would need to identify which item (if any) in the inventory can open such a door, and if found, then that item should be removed from the `List<>` and the door be opened by the appropriate method.

```
private bool OpenDoor(GameObject doorGO){
   // search for key to open the tag of doorGO
   int colorKeyIndex = FindItemIndex(doorGO.tag);
   if( colorKeyIndex > -1 ){
      // remove key item from inventory List<>
      inventory.RemoveAt( colorKeyIndex );

      // now open the door ...
      DoorAnimationController doorAnimationController =
doorGO.GetComponent<>(DoorAnimationController);
      doorAnimationController.OpenDoor();

      return true;
   }

   return false;
}
```

Here is code for a method to find the inventory list key item fitting a door tag:

```
private int FindItemIndex(string doorTag){
   for (int i = 0; i < inventory.Count; i++){
      PickUp item = inventory[i];
      if( (PickUp.PickUpType.Key == item.type) &&
 (item.fitsLockTag == doorTag))
         return i;
```

```
        }

        // not found
        return -1;
    }
```

Need to add [SerializeField] for private properties

Note, if we wished to create editor extensions to work with private variables, then we'd need to explicitly add `[SerializeField]` in the line immediately before the variable to be changed by the editor script. Public variables are serialized by default in Unity, so this was not required for our public `type` variable in script class `PickUp`.

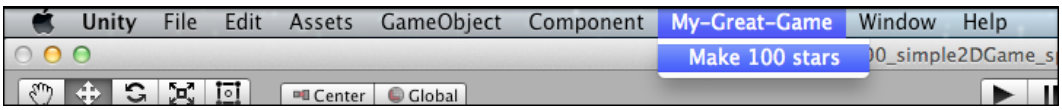
Learn more from the Unity documentation

Unity provides documentation pages about Editor scripts at the following location:

`http://docs.unity3d.com/ScriptReference/Editor.html`

Editor extension to add 100 randomly located copies of a prefab with one menu click

Sometimes we want to create LOTS of pickups, randomly in our scene. Rather than doing this by hand it is possible to add a custom menu and item to the Unity editor, which when selected will execute a script. In this recipe we create a menu item that calls a script to create 100 randomly positioned star pickup prefabs in the Scene.



Insert image 1362OT_12_43.png

Getting ready

This recipe assumes you are starting with project `Simple2Dgame_SpaceGirl` setup from the first recipe in this chapter.

How to do it...

To create an editor extension to add 100 randomly located copies of a prefab with one menu click follow these steps:

1. Start with a new copy of mini-game `Simple2Dgame_SpaceGirl`.

2. In the **Project** panel, create a new folder named `Prefabs`. Inside this new folder create a new empty prefab named `prefab_star`. Populate this prefab by dragging gameObject `star` from the **Hierarchy** panel over `prefab_star` in the **Project** panel. The prefab should now turn blue and have a copy of all of gameObject star's properties and components.
3. Delete gameObject `star` from the **Hiearchy**.
4. In the **Project** panel, create a new folder named `Editor`. Inside this new folder create a new C# script class named `MyGreatGameEditor`, with the following code:

```
using UnityEngine;
using UnityEditor;
using System.Collections;
using System;

public class MyGreatGameEditor : MonoBehaviour {
   const float X_MAX = 10f;
   const float Y_MAX = 10f;

   static GameObject starPrefab;

   [MenuItem("My-Great-Game/Make 100 stars")]
   static void PlacePrefabs(){
      string assetPath =
"Assets/Prefabs/prefab_star.prefab";
      starPrefab =
(GameObject)AssetDatabase.LoadMainAssetAtPath(assetPath);

      int total = 100;
      for(int i = 0; i < total; i++){
         CreateRandomInstance();
      }
   }

   static void CreateRandomInstance(){
      float x = UnityEngine.Random.Range(-X_MAX, X_MAX);
      float y = UnityEngine.Random.Range(-Y_MAX, Y_MAX);
      float z = 0;
      Vector3 randomPosition = new Vector3(x,y,z);

      Instantiate(starPrefab, randomPosition,
Quaternion.identity);
   }
}
```

5. After 20-30 seconds you should now see a new menu appear **My Great Game**, with a single menu item, **Make 100 stars**. Chose this menu item and as if by magic you should now see 100 new **prefab_star(Clone)** gameObjects appear in the scene!

Insert image 1362OT_12_44.png

How it works...

The core aim of this recipe is to add a new menu, containing a single menu item that will execute the action we desire. C# attribute `[MenuItem("<menuName>/<menuItemName>")]` declares the menu name, the menu item name, and Unity will execute the static method that follows in the code listing, each time the menu item is selected by the user.

In this recipe the statement `[MenuItem("My-Great-Game/Make 100 stars")]` declares the menu name as `My-Great-Game` and the menu item as `Make 100 stars`. The method immediately following this attribute is method `PlacePrefabs()`. When this method is executed it makes variable `starPrefab` become a reference to the prefab found via the path "`Assets/Prefabs/prefab_star.prefab`". Then a FOR-loop is executed 100 times, each time calling method `CreateRandomInstance()`.

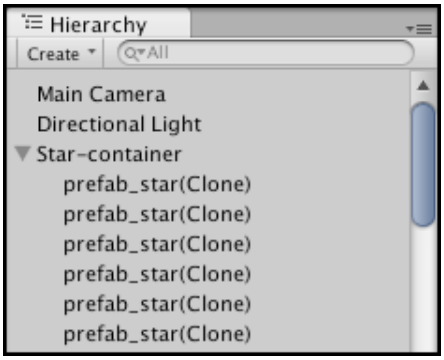
Method `CreateRandomInstance()` creates a Vector3 position `randomPosition` variable, making use of constants `X_MAX` and `Y_MAX`. The `Instantiate(...)` built-in method is then used to create a new gameObject in the scene, making a clone of the prefab and locating it at the position defined by `randomPosition`.

There's more...

Some details you don't want to miss:

Child each new gameObject to a single parent, to avoid filling up the Hierarchy with 100s of new objects

Rather than have 100s of new object clones fill up our Hierarchy panel, a good way to keep things tidy is to have an empty 'parent' gameObject, and child a collection of related gameObjects to it. Let's have a gameObject in the **Hierarchy** named **star-container**, and child all the new stars to this object.



Insert image 1362OT_12_46.png

We need a variable which will be a reference to our container object, `starContainerGO`. We also need a new method, `CreateStarContainerGO()` , which will find a reference to gameObject **star-container**, if such an object already exist, or create a new empty gameObject and give it this name, if one does not already exist. Add the following variable and method to our script class:

```
static GameObject starContainerGO;

static void CreateStarContainerGO() {
   string containerName = "Star-container";
   starContainerGO = GameObject.Find(containerName);
   if (null != starContainerGO)
DestroyImmediate(starContainerGO);
   starContainerGO = new GameObject(containerName);
}
```

Before we create the prefab clones, we need to first ensure we have created out star container gameObject. So we need to call our new method as the first thing we do when method PlacePrefabs() is executed – so add a statement to call this method at the beginning of method PlacePrefabs():

```
static void PlacePrefabs(){
   CreateStarContainerGO();
```

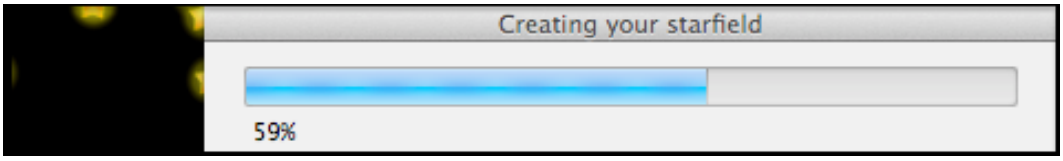
```
        // rest of method as before ...
    }
```

Now we need to modify method `CreateRandomInstance()`, so that it gets a reference to the new gameObject it has just created, so that it can then child this new object to our star-container gameObject variable `starContainerGO`. Modify method `CreateRandomInstance()` so that it looks as follows:

```
    static void CreateRandomInstance() {
        float x = UnityEngine.Random.Range(-X_MAX, X_MAX);
        float y = UnityEngine.Random.Range(-Y_MAX, Y_MAX);
        float z = 0;
        Vector3 randomPosition = new Vector3(x,y,z);

        GameObject newStarGO = (GameObject)Instantiate(starPrefab,
    randomPosition, Quaternion.identity);
        newStarGO.transform.parent = starContainerGO.transform;
    }
```

Displaying a progress bar to display Editor extensions progress completed

If an Editor task is going to take more than half a second or so, then we should indicate progress complete/remaining to the user via a progress bar, so they understand that something is actually happening and the application has not crashed and frozen.



Insert image 1362OT_12_45.png

Getting ready

This recipe adds to the previous one, so make a copy of that project folder, and do your work for this recipe with that copy.

How to do it...

To add a progress bar during the loop (and then remove it after the loop is complete) do the following:

1. Replace method `PlacePrefabs()` with the following code:

```
static void PlacePrefabs(){
   string assetPath = "Assets/Prefabs/prefab_star.prefab";
   starPrefab =
(GameObject)AssetDatabase.LoadMainAssetAtPath(assetPath);

   int total = 100;
   for(int i = 0; i < total; i++){
      CreateRandomInstance();
      EditorUtility.DisplayProgressBar("Creating your starfield",
i + "%", i/100f);
   }

   EditorUtility.ClearProgressBar();
}
```

How it works...

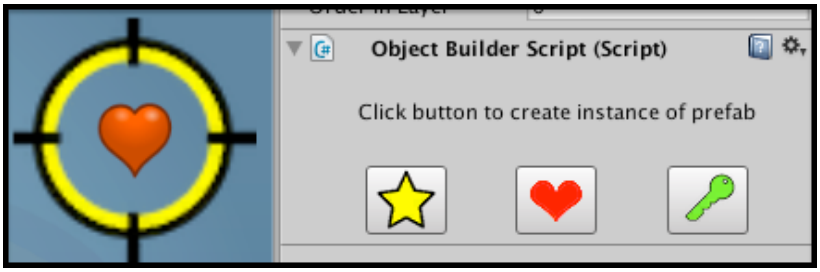
As can be seen, inside the FOR-loop we call `EditorUtility.DisplayProgressBar(...)` passing 3 parameters. The first is a string title for the progress bar dialog window, the second is a string to show below the bar itself (usually a percentage is sufficient), and the final parameter is a value between 0.0 and 1.0, indicating the percentage complete to be displayed.

Since we have loop variable `i` that is a number from 1 to 100, we can display this integer followed by a percentage sign for our second parameter, and just divide this number by 100 to get the decimal value needed to specify how much of the progress bar should be shown as completed. If the loop were running for some other number, we'd just divide the loop counter by the loop total to get our decimal progress value.

Editor extension to have an object creator game object, with buttons to instantiate different pickups at cross-hair object location in scene

If a level designer wishes to place each pickup carefully 'by hand', we can still make this easier than having them have to drag copies of prefabs manually from the **Projects** panel. In this recipe we provide a 'cross-hairs' gameObject, which buttons in the Inspector allowing the game designer to create instances of 3 different kinds of prefab at precise locations by clicking the appropriate button when the center of the cross-hairs is at the desired location.

A Unity Editor extension is at the heart of this recipe, and illustrates how such extensions can allow less technical members of a game development team take an active role in level creation within the Unity Editor.



Insert image 1362OT_12_47.png

Getting ready

This recipe assumes you are starting with project `Simple2Dgame_SpaceGirl` setup from the first recipe in this chapter.

For this recipe, we have prepared the cross-hairs image you need in a folder named `Sprites` in folder `1362_12_03`.

How to do it...

To create an object-creator gameObject follow these steps:

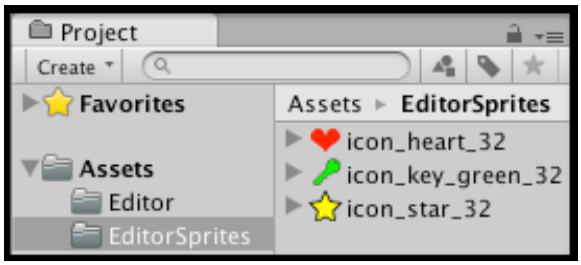
1. Start with a new copy of mini-game `Simple2Dgame_SpaceGirl`.
2. In the **Project** panel, rename gameObject `star` as `pickup`.
3. In the **Project** panel, create a new folder named `Prefabs`. Inside this new folder create three new empty prefabs named `star`, `heart` and `key`.
4. Populate the `star` prefab by dragging gameObject `pickup` from the **Hierarchy** panel over `star` in the **Project** panel. The prefab should now turn blue and have a copy of all of gameObject `star's` properties and components.
5. Add a new tag `Heart` in the Inspector. Select gameObject `pickup` in the **Hierarchy** panel, and assign it this tag `Heart`. Also drag from the **Project** panel (folder `Sprites`) image **healthheart** into the Sprite property of gameObject `pickup` – so the player sees the heart image on screen for this pickup item.
6. Populate the `heart` prefab by dragging gameObject `pickup` from the **Hierarchy** panel over `heart` in folder `Prefabs` in the **Project** panel. The prefab should now turn blue and have a copy of all of gameObject `pickup's` properties and components.

7. Add a new tag `Key` in the Inspector. Select gameObject `pickup` in the **Hierarchy** panel, and assign it this tag `Key`. Also drag from the **Project** panel (folder `Sprites`) image **icon_key_green_100** into the Sprite property of gameObject `pickup` – so the player sees the key image on screen for this pickup item.
8. Populate the `key` prefab by dragging gameObject `pickup` from the **Hierarchy** panel over `key` in folder `Prefabs` in the **Project** panel. The prefab should now turn blue and have a copy of all of gameObject `pickup's` properties and components.
9. Delete gameObject `pickup` from the **Hiearchy**.
10. In the **Project** panel, create a new folder named `Editor`. Inside this new folder create a new C# script class named `ObjectBuilderEditor`, with the following code:

```
using UnityEngine;
using System.Collections;
using UnityEditor;

[CustomEditor(typeof(ObjectBuilderScript))]
public class ObjectBuilderEditor : Editor{
   private Texture iconStar;
   private Texture iconHeart;
   private Texture iconKey;

   private GameObject prefabHeart;
   private GameObject prefabStar;
   private GameObject prefabKey;

   void OnEnable () {
      iconStar =
Resources.LoadAssetAtPath("Assets/EditorSprites/icon_star_3
2.png", typeof(Texture)) as Texture;
      iconHeart =
Resources.LoadAssetAtPath("Assets/EditorSprites/icon_heart_
32.png", typeof(Texture)) as Texture;
      iconKey =
Resources.LoadAssetAtPath("Assets/EditorSprites/
icon_key_green_32.png", typeof(Texture)) as Texture;

      prefabStar =
Resources.LoadAssetAtPath("Assets/Prefabs/star.prefab",
typeof(GameObject)) as GameObject;
      prefabHeart =
Resources.LoadAssetAtPath("Assets/Prefabs/heart.prefab",
typeof(GameObject)) as GameObject;
```

```
        prefabKey =
Resources.LoadAssetAtPath("Assets/Prefabs/key.prefab",
typeof(GameObject)) as GameObject;
    }

    public override void OnInspectorGUI(){
        ObjectBuilderScript myScript =
(ObjectBuilderScript)target;

        GUILayout.Label("");
        GUILayout.BeginHorizontal();
        GUILayout.FlexibleSpace();
        GUILayout.Label("Click button to create instance of
prefab");
        GUILayout.FlexibleSpace();
        GUILayout.EndHorizontal();
        GUILayout.Label("");

        GUILayout.BeginHorizontal();
        GUILayout.FlexibleSpace();
        if(GUILayout.Button(iconStar))
myScript.AddObjectToScene(prefabStar);
        GUILayout.FlexibleSpace();
        if(GUILayout.Button(iconHeart))
myScript.AddObjectToScene(prefabHeart);
        GUILayout.FlexibleSpace();
        if(GUILayout.Button(iconKey))
myScript.AddObjectToScene(prefabKey);
        GUILayout.FlexibleSpace();
        GUILayout.EndHorizontal();

    }
}
```

11. Our Editor script is expecting to find the 3 icons in a folder named EditorSprites, so let's do this. First created a new folder named `EditorSprites`. Next drag the three 32x32 pixel icons from the Sprites folder into this new `EditorSprites` folder. Our Editor script should now be able load these icons for image-based buttons that it will be drawing in the Inspector, from which the user chooses which pickup prefab object to clone into the scene.



Insert image 1362OT_12_56.png

12. From the **Project** panel, drag sprite **cross_hairs.fw** into the **Scene**. Rename this gameObject `object-creator-cross-hairs` and in its **Sprite Renderer** component in the **Inspector** set is **Sorting Layer** to **Foreground**.
13. Attatch the following C# script to gameObject `object-creator-cross-hairs`:

```
using UnityEngine;
using System.Collections;

public class ObjectBuilderScript : MonoBehaviour {
   void Awake(){
      gameObject.SetActive(false);
   }

   public void AddObjectToScene(GameObject
prefabToCreateInScene){
      GameObject newGO =
(GameObject)Instantiate(prefabToCreateInScene,
transform.position, Quaternion.identity);
      newGO.name = prefabToCreateInScene.name;
   }
}
```

14. Select the **Rect Tool** (shortcut key T), and as you drag gameObject `object-creator-cross-hairs` and click the desired icon in the **Inspector**, new pickup gameObjects will be added to the scene's **Hierarchy**.

How it works...

The script class `ObjectBuilderScript` has just 2 methods, one of which has just one statement – the `Awake()` method simply makes this gameObject become inactive when the game is running (since we don't want the user to see our cross-hairs created tool during gameplay). Method `AddObjectToScene(…)` receives a reference to a prefab as a parameter, and instantiates a new clone of the prefab in the scene at the location of gameObject `object-creator-cross-hairs` at that point in time.

Script class `ObjectBuilderEditor` has C# attribute `[CustomEditor(typeof(ObjectBuilderScript))]` immediately before the class is declared, telling Unity to use this class to control how `ObjectBuilderScript` gameObject properties and components are shown to the user in the **Inspector**.

There are 6 variables, 3 Textures for the icons to form the buttons in the Inspector, and 3 GameObject references to the prefabs of which instances will be created. Method `OnEnable()` assigns values to these 6 variables using built-in method `Resources.LoadAssetAtPath()`, retrieving the icons from **Project** folder `EditorSprites`, and getting references to the prefabs in **Project** folder `Prefabs`.

Method `OnInspectorGUI()` has a variable `myScript`, which is set to be a reference to the instance of scripted component `ObjectBuilderScript` in gameObject **`object-creator-cross-hairs`** (so we can call its method when a prefab has been chosen…). The method then displays a mixture of empty text **`Labels`** (to get some vertical space), and **`FlexibleSpace`** (to get some horizontal spacing and centering), and displays 3 buttons to the user, with icons of star, heart and key. The scripted GUI technique for Unity custom **Inspector** GUIs wraps an **`IF`**-statement around each button, and on the frame the user clicks the button, the statement block of the **`IF`**-statement will be executed. When any of the 3 buttons is clicked, a call is made to `AddObjectToScene(…)` of scripted component `ObjectBuilderScript`, passing the prefab corresponding to the button that was clicked.

Conclusion

In this chapter, we have introduced recipes demonstrating some Unity Editor extension scripts, illustrating how we can make things easier and less script-based, and less prone to errors, by limiting and controlling the properties of objects and how they are selected or changed via the **Inspector**.

The concept of serialization was raised in the Editor extension recipes, whereby we need to remember that when we are editing item properties in the Inspector each change needs to be saved to disk, so that the updated property is correct when we next use or edit that item. This is achieved in method `OnInspectorGUI()` by first calling `serializedObject.Update()`, and after all changes have been made in the Inspector, finally calling `serializedObject.ApplyModifiedProperties()`. Some sources of more information and examples about custom Editor extensions include:

- learn more about custom Unity editors in Ryan Meier's blog `http://www.ryan-meier.com/blog/?p=72`
- more custom Unity editor scripts/tutorials, including grids and color pickers:

http://code.tutsplus.com/tutorials/how-to-add-your-own-tools-to-unitys-editor--active-10047