2
Inventory GUIs

In this chapter, we will cover:

- Creating the Simple2DGame_SpaceGirl mini-game for this chapter
- Displaying single object pickups with carrying and not-carrying text
- Displaying single object pickups with carrying and not-carrying icons
- Displaying multiple pickups of same object with text totals
- Displaying multiple pickups of same object with multiple status icons
- Revealing icons for multiple object pickups by changing the size of a tiled image
- Displaying multiple pickups of different objects as a list of text via a dynamic List<> of PickUp objects
- Displaying multiple pickups of different objects as text totals via a dynamic Dictionary<> of PickUp objects and 'enum' pickup types
- Generalizing multiple icon displays using UI Grid Layout Groups (with scrollbars!)

Introduction

Many games involve the player collecting items, or choosing from a selection of items. Examples could be collecting keys to open doors, collecting ammo for weapons, choosing from a collection of spells to cast, and so on.

The recipes in this chapter offer a range of solutions to displaying to the player whether they are carrying an item or not, or if they are allowed more than one of an item, and how many they have.

The big picture

The two parts software design for implementing inventories relate to how we choose to **represent** the data about inventory items (i.e. the **data types and structures** to store the data), and secondly how we choose to **display information** about inventory items to the player (the **UI: User Interface**).

Also, while not strictly inventory items, player properties such as lives left, or health or time remaining can also be designed around the same concepts we present in this chapter.

We need to first think about the nature of different inventory items for any particular game:

- Single items
    - Example(s): the only key for a level, our suit of magic armor
    - Data type: bool (true / false)
    - UI: nothing (if not carried), or text/image to show being carried
        - Or perhaps text saying ‘no key’ / ‘key’, or 2 images, one showing empty key outline, and the second showing full color key
        - If we wish to highlight to player that there is an option to be carrying this item
- Continuous item:
    - Example(s): time left, health, shield strength
    - Data type: float (e.g. 0.00 – 1.00) or integer scale (e.g. 0%Q .. 100%)
    - UI: text number or image progress bar / pie chart
- 2 or more of same item
    - Example(s): lives left, or number of arrows or bullets left
    - Data type: int (whole numbers)
    - UI: text count or images
- Collection of related items
    - Example(s): keys of different colors to open correspondingly colored doors, potions of different strength with different titles
    - Data structure: a struct or class for the general item type (e.g. class Key(color / cost / doorOpenTagString), stored as an array of List
    - UI: text list, or list/grid arrangement of icons
- Collection of different items

- Example(s): keys, potions, weapons, tools – all in same inventory system
- Data structure: List<> or Dictionary<> or array of Objects, which can be instances of different class for each item type

Each of the above representations and UI display methods are illustrated by the recipes in this chapter.

Creating the Simple2DGame_SpaceGirl mini-game for this chapter

This recipe presents the steps to create the 2DSpaceGirl mini-game, on which all the recipes of this chapter are based.

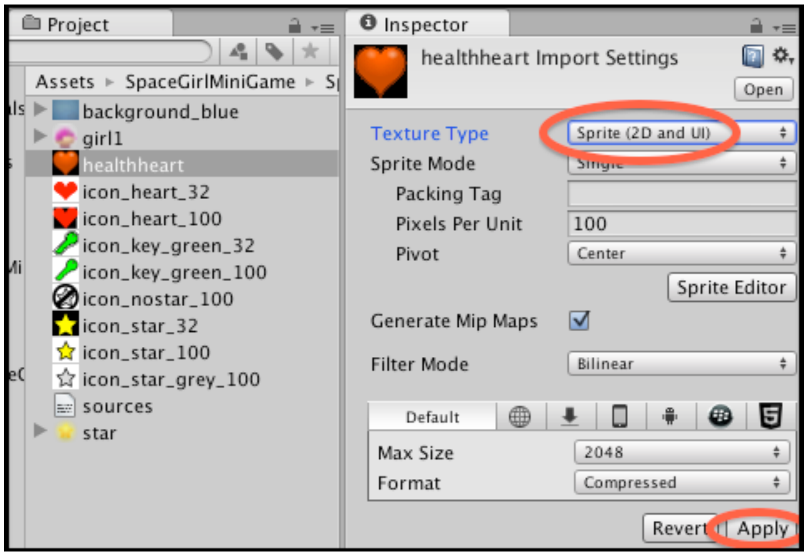
Getting ready

For this recipe, we have prepared the images you need in a folder named `Sprites` in folder `1362_02_01`. We have also provided the completed game as a Unity package in this folder named `Simple2DGame_SpaceGirl`.

How to do it...

To create the Simple 2D Game Space Girl mini-game follow these steps:

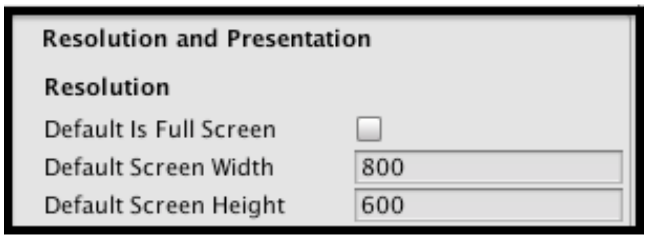
1. Create a new, empty 2D project.
2. Import supplied folder `Sprites` into your project.
3. Convert each sprite image to be of type **Sprite (2D and UI)**. To do this select the sprite in the **Project** panel, then in the **Inspector** change choose **Sprite (2D and UI)** from drop-down menu **Texture Type**, and click the **Apply** button.



Texture Type: Sprite (2D and UI)
Sprite Mode: Single
Packing Tag:
Pixels Per Unit: 100
Pivot: Center
Generate Mip Maps: ✓
Filter Mode: Bilinear
Max Size: 2048
Format: Compressed

Insert image 1362OT_02_54.png

4. Set the Unity Player screen size to 800x600: choose menu **Edit | Project Settings | Player**, then for option **Resolution and Presentation** uncheck '`Default is Full Screen`' and set the width to 800 and height to 600.



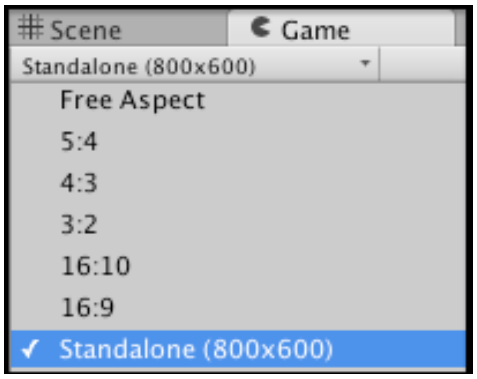
Resolution and Presentation

Resolution

Default Is Full Screen: ☐
Default Screen Width: 800
Default Screen Height: 600

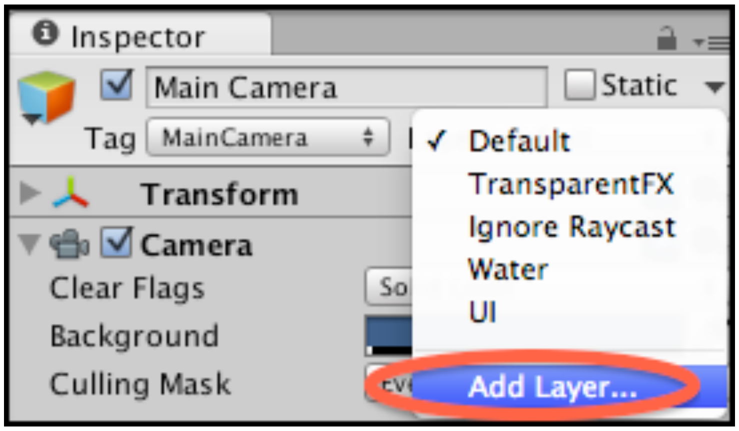
Insert image 1362OT_02_03.png

5. Select the **Game** panel, if not already chosen then choose **Standalone (800x600)** from the dropdown menu.



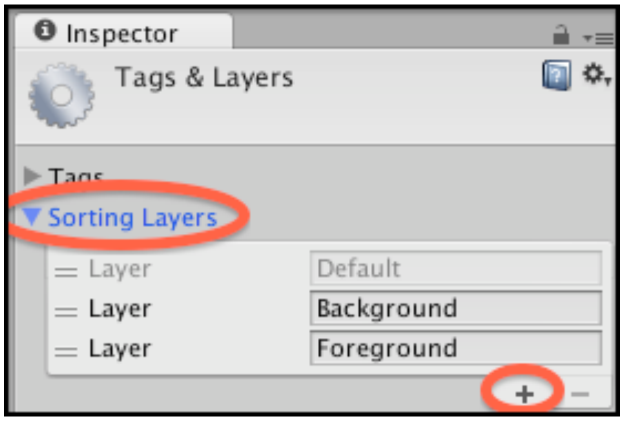
Insert image 1362OT_02_02.png

6. Display the **Tags & Layers** properties for the current Unity project. Either choose menu **Edit | Project Settings | Tags and Layers**. Alterntatively, if you are already editing a gameObject, then you can select menu **Add Layer…** from the **Layer** drop-down menu at the top of the **Inspector** panel.



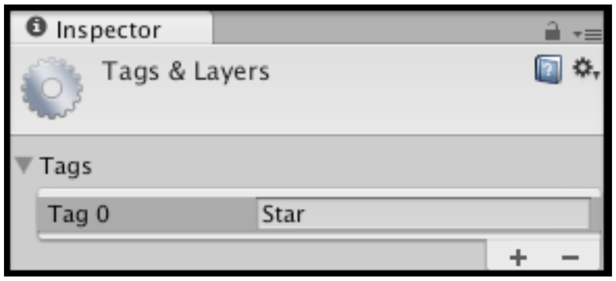
Insert image 1362OT_02_51.png

7. The **Inspector** should now being displaying the **Tags & Layers** properties for the current Unity project. Use the expand/contract triangle tools to contract **Tags** and **Layers**, and to expand **Sorting Layers**.
8. Use the plus sign '+' button to add 2 new sorting layers: first add one named **Background**, next add one named **Foreground**. The sequence is important, since unity will draw items in layers further down this list on top of items earlier in the list.



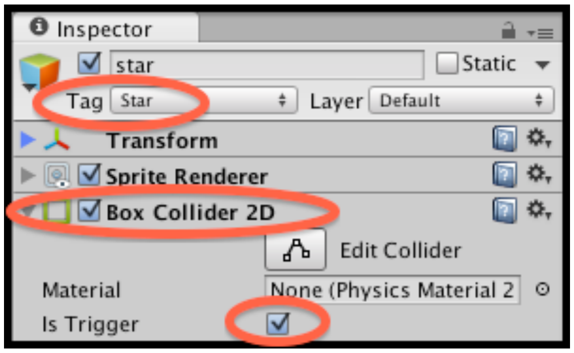
Insert image 1362OT_02_52.png

9. Drag sprite `background-blue` from the **Project** panel (folder `Sprites`) into either the **Game** or **Hierarchy** panel to create a gameObject for the current scene.
10. Set the **Sorting Layer** of gameObject `background-blue` to **Background** (in the **Sprite Renderer** component).
11. Drag sprite `star` from the **Project** panel (folder `Sprites`) into either the **Game** or **Hierarchy** panel to create a gameObject for the current scene.
12. In the Inspector add a new tag **'Star'**, by selecting option **Add Tag…** from the **Tag** drop-down menu at the top of the **Inspector** panel.



Insert image 1362OT_02_03.png

13. Apply tag **'Star'** to gameObject `star` in the **Hierarchy** scene.
14. Set the **Sorting Layer** of gameObject `star` to **Foreground** (in the **Sprite Renderer** component).
15. Add to gameObject `star` a Box Collider 2D (**Add Component | Physics 2D | Box Collider 2D**) and check its '`Is Trigger`'.


Inspector

✓ star ☐ Static

Tag Star Layer Default

Transform

✓ Sprite Renderer

✓ Box Collider 2D

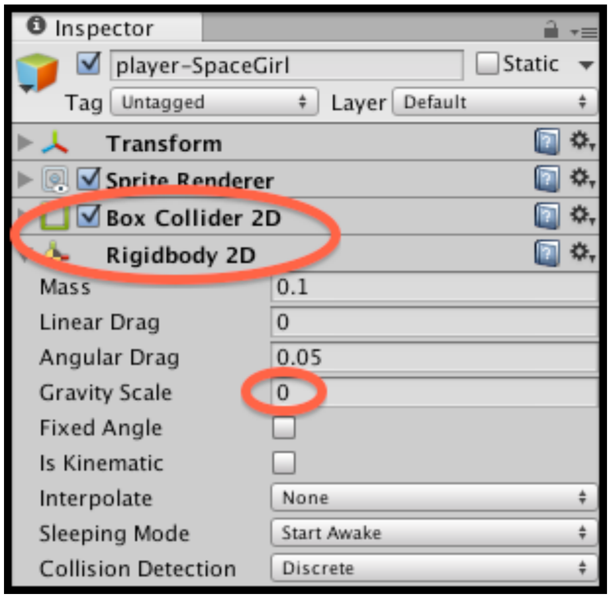
Edit Collider

Material None (Physics Material 2

Is Trigger ✓

Insert image 1362OT_02_08.png

16. Drag sprite `girl1` from the **Project** panel (folder `Sprites`) into either the **Game** or **Hierarchy** panel to create a gameObject for the player's character in the current scene. Rename this gameObject `player-SpaceGirl`.
17. Set the **Sorting Layer** of gameObject `player-SpaceGirl` to **Foreground** (in the **Sprite Renderer** component).
18. Add to gameObject `player-SpaceGirl` a Box Collider 2D (**Add Component | Physics 2D | Box Collider 2D**).
19. Add to gameObject `player-SpaceGirl` a RigidBody 2D (**Add Component | Physics 2D | Rigid Body 2D**). Set its **Gravity Scale** to zero (so it isn't falling down the screen due to simulated gravity).


Inspector

✓ player-SpaceGirl ☐ Static

Tag Untagged Layer Default

Transform

✓ Sprite Renderer

✓ Box Collider 2D

Rigidbody 2D

Mass	0.1
Linear Drag	0
Angular Drag	0.05
Gravity Scale	0
Fixed Angle	☐
Is Kinematic	☐
Interpolate	None
Sleeping Mode	Start Awake
Collision Detection	Discrete

Insert image 1362OT_02_30.png

20. Create a new folder for your scripts named `Scripts`.
21. Create the following C# Script `PlayerMove` (in folder `Scripts`) and add it to to gameObject `player-SpaceGirl` in the **Hierarchy**:

```
using UnityEngine;
using System.Collections;

public class PlayerMove : MonoBehaviour {
   public float speed = 10;
   private Rigidbody2D rigidBody2D;

   void Awake(){
      rigidBody2D = GetComponent<Rigidbody2D>();
   }

   void FixedUpdate(){
      float xMove = Input.GetAxis("Horizontal");
      float yMove = Input.GetAxis("Vertical");

      float xSpeed = xMove * speed;
      float ySpeed = yMove * speed;

      Vector2 newVelocity = new Vector2(xSpeed, ySpeed);

      rigidBody2D.velocity = newVelocity;
   }
}
```

22. Save the scene (name it **Main Scene** and save it into a new folder named `Scenes`).

How it works...

You have created a player character in the scene, with its movement scripted component `PlayerMove`. You have also created a star gameObject (a pickup), tagged 'Star' and with 2D box collider that will trigger a collision when the player's character hits it. When you run the game the `player-SpaceGirl` character should move around using the WASD or arrow keys or joystick. Currently nothing will happen if the `player-SpaceGirl` character hits a star, since that has yet to be scripted…

You have added a background (gameObject `background-blue`) to the scene, which will be behind everything since it is in the rearmost sorting layer **Background**. Items you want to appear in front of this background (the player's character and the star so far) are placed on sorting layer **Foreground**. Learn more about Unity tags and layers at the following location:

`http://docs.unity3d.com/Manual/class-TagManager.html`

Displaying single object pickups with carrying and not-carrying text

Often the simplest inventory situation is to display text to tell the user if they are carrying a single item (or not).

Getting ready

This recipe assumes you are starting with project `Simple2Dgame_SpaceGirl` setup from the first recipe in this chapter. So either make a copy of that project, or do the following:

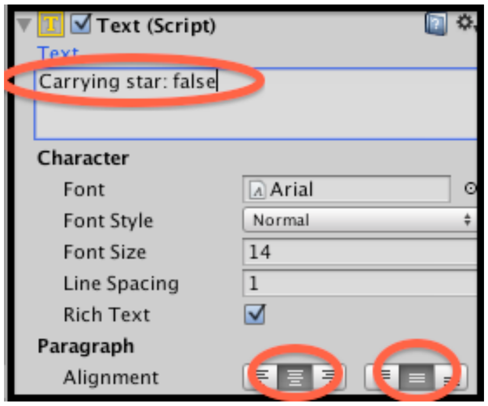
- create a new, empty 2D project
- import package `Simple2Dgame_SpaceGirl`
- open scene **Scene1** (in folder `Scenes`)
- set the Unity Player screen size to 800x600 (see previous recipe for how to do this), and select this resolution in the **Game** panel the dropdown menu
- convert each sprite image to be of type **Sprite (2D and UI)**. In the **Inspector** change choose **Sprite (2D and UI)** from drop-down menu **Texture Type**, and click the **Apply** button

For this recipe, we have prepared the font you need in a folder named `Fonts` in folder `1362_02_02`.

How to do it...

To display text to inform the user about the status of carrying a single object pickup follow these steps:

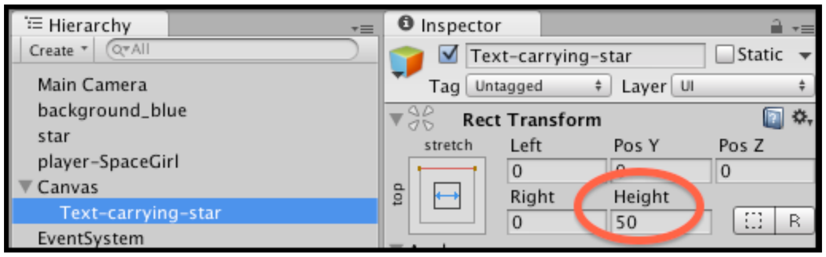
1. Start with a new copy of mini-game `Simple2Dgame_SpaceGirl`.
2. Add a UI **Text** object (**Create | UI | Text**). Rename it `Text-carrying-star`. Change its tet to "**no star :-(**".
3. Import the provided `Fonts` folder into your project.
4. In the **Inspector** panel set the font of `Text-carrying-star` to '**Xolonium-Bold**' (folder `Fonts`), and set its color to yellow. Center the text horizontally and vertically, and set its height to 50, and set the **Font Size** to 32.



Text: Carrying star: false
Font: Arial
Font Style: Normal
Font Size: 14
Line Spacing: 1
Rich Text: ✓

Insert image 1362OT_02_05.png

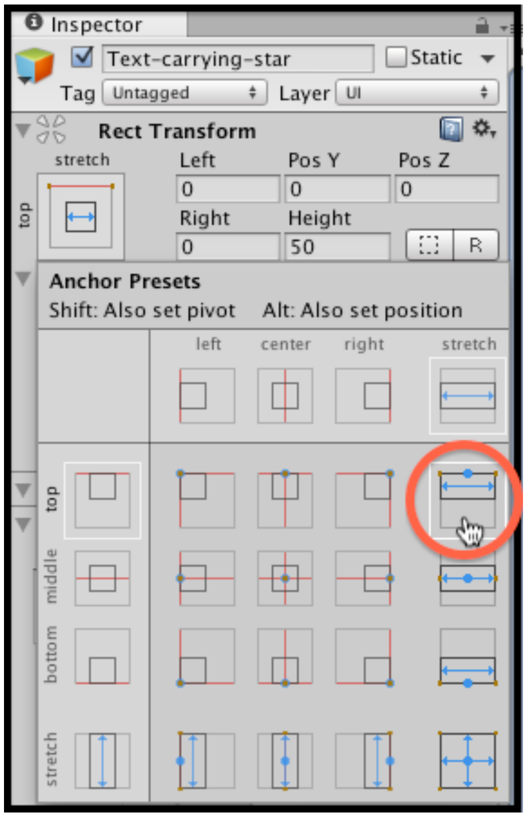
5. In its **Rect Transfrom** component, set its height to 50.



Text-carrying-star
Tag: Untagged
Layer: UI
Left: 0
Pos Y: 0
Pos Z: 0
Right: 0
Height: 50

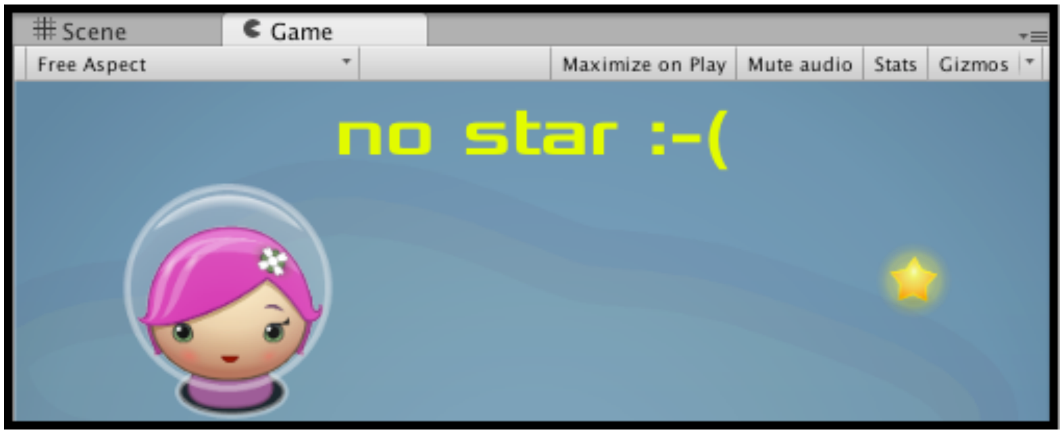
Insert image 1362OT_02_53.png

6. Edit its **Rect Transform**, while holding down *SHIFT* and *ALT* (to set pivot and position) choose the top-stretch box.



Insert image 1362OT_02_06.png

7. Your text should now be positioned at the middle top of the **Game** panel, and its width should stretch to match that of the whole panel.

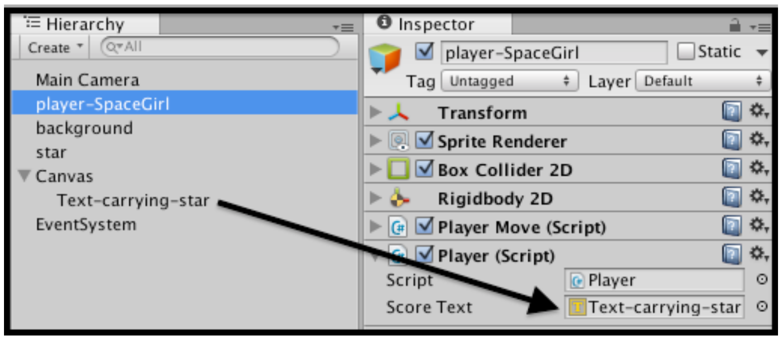


Insert image 1362OT_02_07.png

8. Add the following C# Script `Player` to gameObject `player-SpaceGirl` in the **Hierarchy**:

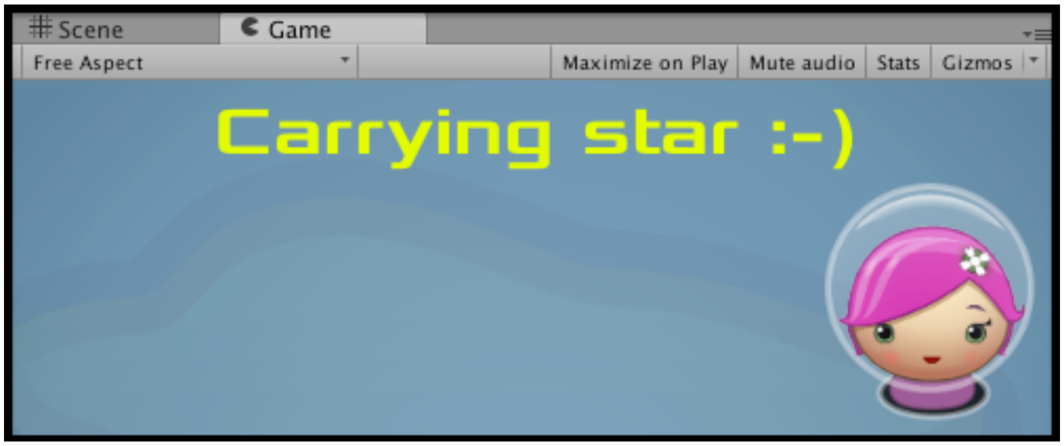
```
using UnityEngine;
using System.Collections;
using UnityEngine.UI;

public class Player : MonoBehaviour {
   public Text starText;
   private bool carryingStar = false;

   void Start(){
      UpdateStarText();
   }

   void OnTriggerEnter2D(Collider2D hit){
      if(hit.CompareTag("Star")){
         carryingStar = true;
         UpdateStarText();
         Destroy(hit.gameObject);
      }
   }

   private void UpdateStarText(){
      string starMessage = "no star :-(";
      if(carryingStar) starMessage = "Carrying star :-)";
      starText.text = starMessage;
   }
}
```

9. From the **Hierarchy** view, select gameObject `player-SpaceGirl`. Then, from the **Inspector**, access the **Player (Script)** component and populate the **Star Text** public field with UI **Text** object `Text-carrying-star`.



Insert image 1362OT_11_10.png

10. When you play the scene, after moving the character into the star, the the star should dissapear and the on-screen UI **Text** message should change to "**Carrying star :-)**".



Insert image 1362OT_11_09.png

How it works...

The `Text` variable `starText` is a reference to the UI **Text** object `Text-carrying-star`. The `bool` variable `carryingStar` represents whether or not the player is carrying the star at any point in time; it is initialized to false.

The `UpdateStarText()` method copies the contents of string `starMessage` to the text property of `starText`. The default value of this string tells the user that the player is not carrying the star, but an IF-statement tests the value of `carryingKey`, and if that is true then the message is changed to inform the player that they are carrying the star.

Each time the player's character collides with any object that has its '**Is Trigger'** set to true an `OnTriggerEnter2D()` event message is sent to both objects involved in the collision. The `OnTriggerEnter2D()` message is passed a parameter that is the Collider2D component inside the object just collided with.

Our player's `OnTriggerEnter2D()` method tests the 'tag' string of the object collided with to see if it has the value '**Star**. Since the gameObject **star** we created has its trigger set, and has the tag '**Star**', then the IF-statement inside this method will detect a collision with **star** and does three actions: it sets the Boolean variable `carryingStar` to true, it calls method `UpdateStarText()` and it destroys the game object it has just collided with (in this case **star**).

NOTE: Boolean variables are often referred to as '**flags'**

The use of a bool (true/false) variable to represent whether some feature of the game state is true or false is very common. Programmers often refer to these variables as 'flags'. So programmers might refer to variable `carryingStar` as the star-carrying flag.

When the scene begins, via method `Start()`, we call method `UpdateStarText()`, this ensures we are not relying on text typed into the UI **Text** object `Text-carrying-star` at design-time, but that the UI seen by the user is always that set by our run-time methods. This avoids problems where the words to be displayed to the user are changed in code, but not in the Inspector – which leads to a mis-match between the on-screen text when the scene first runs, and after it has been updated from a script.

A golden rule in Unity game design is to **avoid duplicating content in more than one place**, and therefore we avoid having to maintain two or more copies of the same content. Each duplicate is an opportunity for maintenance issues when some, but not all, copies of a value are changed.

Maximizing use of prefabs is anther example of this principle in action.

There's more...

Some details you don't want to miss:

Separation of 'View' logic

A game design pattern (best practice approach) called the Model-View-Controller Pattern (MVC) is to separate the code that updates the UI from the code that changes player and game variables, such as score and inventory item lists etc. Although this recipe has only one variable and one method to update the UI, well structured game architectures 'scale up' to cope with more complex games, so it is often worth the effort of a little more code and an extra script class, even at this game beginning stage, if we want our final game architecture to be well structured and maintainable.

To implement the separation of view pattern for this recipe we need to do the following:

1. Add the following C# Script `PlayerInventoryDisplay` to gameObject `player-SpaceGirl` in the **Hierarchy:**

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

public class PlayerInventoryDisplay : MonoBehaviour
{
   public Text starText;

   public void OnChangeCarryingStar(bool carryingStar){
      string starMessage = "no star :-(";
      if(carryingStar) starMessage = "Carrying star :-)";
      starText.text = starMessage;
   }
}
```

2. From the **Hierarchy** view, select gameObject `player-SpaceGirl`. Then, from the **Inspector**, access the `PlayerInventoryDisplay` **(Script)** component and populate the **Score Text** public field with UI **Text** object `Text-carrying-star`.
3. Remove existing C# Script component `Player` and replace with this C# Script `PlayerInventory` containing the following (simplified) code:

```
using UnityEngine;
using System.Collections;

public class PlayerInventory : MonoBehaviour {
   private PlayerInventoryDisplay playerInventoryDisplay;
   private bool carryingStar = false;

   void Start(){
      playerInventoryDisplay =
GetComponent<PlayerInventoryDisplay>();

   playerInventoryDisplay.OnChangeCarryingStar(carryingStar
);
   }
```

```
            void OnTriggerEnter2D(Collider2D hit){
                if(hit.CompareTag("Star")){
                    carryingStar = true;

            playerInventoryDisplay.OnChangeCarryingStar(carryingStar
        );
                    Destroy(hit.gameObject);
                }
            }
        }
```

As can be seen, the `PlayerInventory` script class no longer has to maintain a link to the UI **Text**, or worry about changing the text property of that UI component – all that work is now the responsibility of the `PlayerInventoryDisplay` script. When the Player instance component detects a collision with the star, after changing the `carryingStar` bool flag's value to true, it just calls method `OnChangeCarryingStar()` of the `PlayerInventoryDisplay` component.

The result is that the code for script class `PlayerInventory` concentrates on the player collision and status variables, while the code for script class `PlayerInventoryDisplay` handles the communication to the user. Another advantage of this design pattern is that the method in which information is communicated to the user via the UI can be changed (e.g. from text to an icon), without any change to the code in script class Player.

Note – there is no difference in the experience of the player, all the changes are to improve the architectural structure of our game code.

Displaying single object pickups with carrying and not-carrying icons

Graphic icons are an effective way to inform the player that they are carrying an item. In this recipe if no star is being carried a grey-filled icon in a blocked-off circle is displayed, then after the star has been picked up, a yellow-filled icon is displayed.

In many cases icons are clearer (they don't require reading and thinking about), and also can be smaller on screen than text messages for indicating player status and inventory items.

Insert image 1362OT_02_14.png

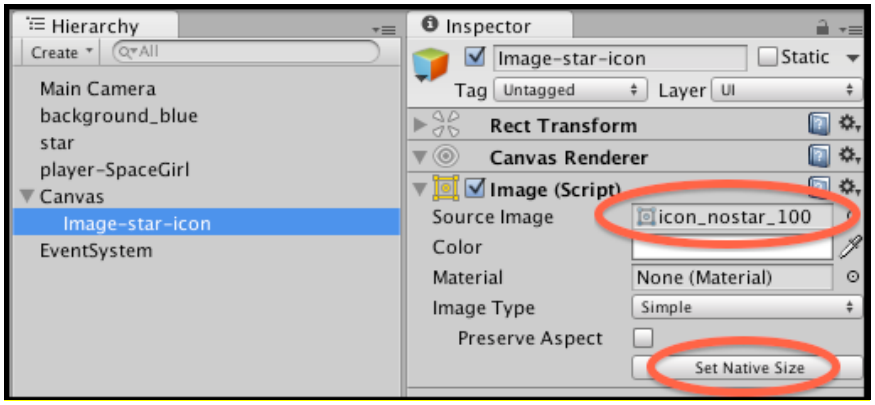
Getting ready

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

How to do it...

To toggle carrying and not-carrying icons for a single object pickup follow these steps:

1. Start with a new copy of mini-game `Simple2Dgame_SpaceGirl`.
2. In the **Hierachy** panel add a new UI **Image** object (**Create | UI | Image**). Rename it `Image-star-icon`.
3. Select `Image-star-icon` in the **Hierachy** panel.
4. From the **Project** panel drag sprite **icon_star_100** (folder `Sprites`) into the **Source Image** field in the **Inspector** (in the **Image (Script)** component).
5. Click the **Set Native Size** button for this for the **Image** component. This will resize the UI **Image** to fit the physical pixel width and height of sprite file **sicon_nostar_100.**



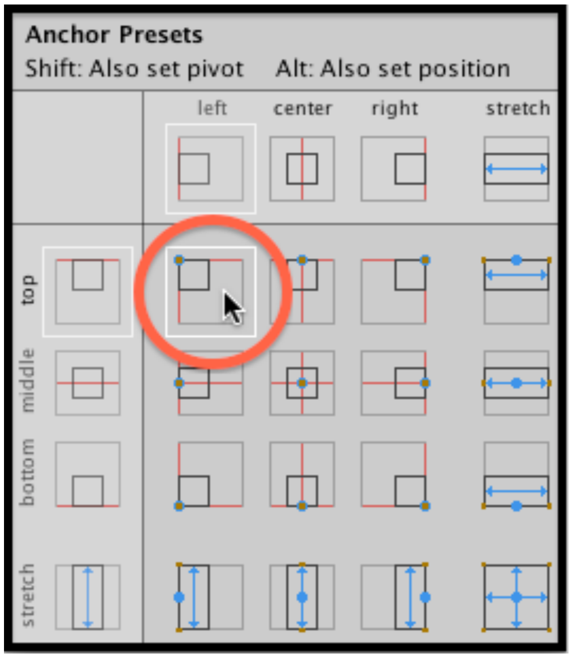
Main Camera
background_blue
star
player-SpaceGirl
Canvas
Image-star-icon
EventSystem

Image-star-icon
Tag Untagged
Layer UI

Source Image icon_nostar_100
Color
Material None (Material)
Image Type Simple
Preserve Aspect

Insert image 1362OT_11_11.png

6. Now we will position our icon at the **top** and **left** of the **Game** panel. Edit the UI **Image's Rect Transform** component, while holding down *SHIFT* and *ALT* (to set pivot and position) choose the top-left box. The UI **Image** should now be positioned at the top-left of the **Game** panel.

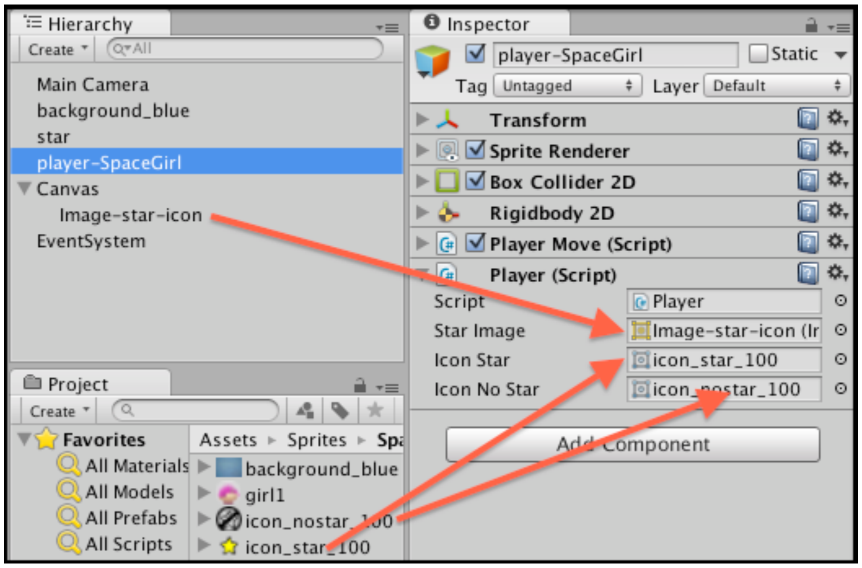


Insert image 1362OT_02_12.png

7. Add the following C# Script `Player` to gameObject `player-SpaceGirl` in the **Hierarchy**:

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

public class Player : MonoBehaviour {
   public Image starImage;
   public Sprite iconStar;
   public Sprite iconNoStar;
   private bool carryingStar = false;

   void OnTriggerEnter2D(Collider2D hit){
      if(hit.CompareTag("Star")){
         carryingStar = true;
         UpdateStarImage();
         Destroy(hit.gameObject);
      }
   }

   private void UpdateStarImage(){
      if(carryingStar)
         starImage.sprite = iconStar;
      else
         starImage.sprite = iconNoStar;
   }
}
```

8. From the **Hierarchy** view, select gameObject `player-SpaceGirl`. Then, from the **Inspector**, access the **Player (Script)** component and populate the **Star Image** public field with UI **Image** object `Image-star-icon`.
9. Now populate the **Icon Star** public field from the **Project** panel with sprite `icon_star_100`, and populate the **Icon No Star** public field from the **Project** panel with sprite `icon_nostar_100`.



player-SpaceGirl Static
Tag Untagged Layer Default

Script: Player
Star Image: Image-star-icon (Ir
Icon Star: icon_star_100
Icon No Star: icon_nostar_100

Insert image 1362OT_02_13.png

10. Now when you play the scene, you should see the no star icon (grey-filled icon in a blocked-off circle) at the top left until you pickup the star, at which time it will change to show the carrying star icon (yellow-filled star).

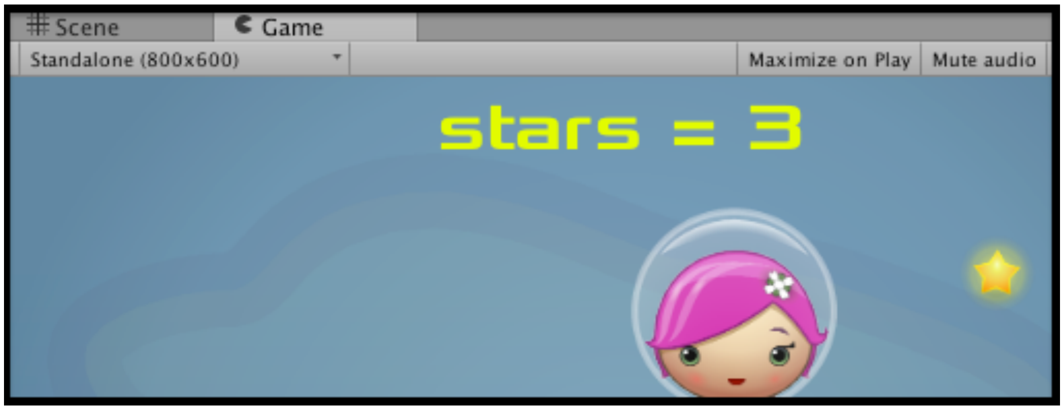
How it works...

The `Image` variable `starImage` is a reference to the UI **Image** object `Image-star-icon`. `Sprite` variables `iconStar` and `iconNoStar` are references to `Sprite` files in the Project panel – the sprites to display to tell the player whether or not a star is being carried. The `bool` variable `carryingStar` represents whether or not the player is carrying the star at any point in time; it is initialized to false.

Much of the logic for this recipe is the same as the previous one. Each time method `UpdateStarImage()` is called it sets the UI **Image** to the sprite that corresponds to the value of bool variable `carryingsStar`.

Displaying multiple pickups of same object with text totals

When several items of the same type have been picked-up, often the simplest way to convey what is being carried to the user is to display a text message, showing the numeric total of each item type being carried. In this recipe the total number of stars collected is displayed using a UI **Text** object.



Insert image 1362OT_02_15.png

Getting ready

This recipe assumes you are starting with project `Simple2Dgame_SpaceGirl` setup from the first recipe in this chapter. The font you need can be found in folder `1362_02_02`.

How to do it...

To displaying inventory total text for multiple pickups of same type of object follow these steps:

1. Start with a new copy of mini-game `Simple2Dgame_SpaceGirl`.
2. Add a UI **Text** object (**Create | UI | Text**). Rename it `Text-carrying-star`. Change its tet to "**stars = 0**".
3. Import the provided `Fonts` folder into your project.
4. In the **Inspector** panel set the font of `Text-carrying-star` to '**Xolonium-Bold**' (folder `Fonts`), and set its color to yellow. Center the text horizontally and vertically, and set its height to 50, and set the **Font Size** to 32.
5. In its **Rect Transfrom** component, set its height to 50. Edit its **Rect Transform**, while holding down *SHIFT* and *ALT* (to set pivot and position) choose the top-stetch box. Your text should now be positioned at the middle top of the **Game** panel, and its width should stretch to match that of the whole panel.
6. Add the following C# Script `Player` to gameObject `player-SpaceGirl` in the **Hierarchy**:

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

public class Player : MonoBehaviour {
    public Text starText;
    private int totalStars = 0;

    void Start(){
        UpdateStarText();
    }

    void OnTriggerEnter2D(Collider2D hit){
        if(hit.CompareTag("Star")){
            totalStars++;
            UpdateStarText();
            Destroy(hit.gameObject);
        }
    }

    private void UpdateStarText(){
        string starMessage = "stars = " + totalStars;
        starText.text = starMessage;
    }
}
```

7. From the **Hierarchy** view, select gameObject `player-SpaceGirl`. Then, from the **Inspector**, access the **Player (Script)** component and populate the **Star Text** public field with UI **Text** object `Text-carrying-star`.
8. Select gameObject `star` in the **Hierarchy** panel, and make 3 more copies of this gameObject.

> **NOTE**
>
> Use keyboard shortcut *CTRL-D* (Windows) or *CMD-D* (Mac) to quickly duplicate gameObjects.

9. Move these new gameObject to different parts of the screen.
10. Play the game – each time you pickup a star, the total should be displayed in the form “**stars = 2**”.

How it works...

The `Text` variable `starText` is a reference to the UI **Text** object `Text-carrying-star`. The `int` variable `totalStars` represents how many stars have been collected so far; it is initialized to zero.

In method `OnTriggerEnter2D()` the `totalStars` counter is incremented by 1 each time the player’s character hits an object tagged “**Star**”. The collided star gameObject is destroyed, and a call made to method `UpdateStarText()`.

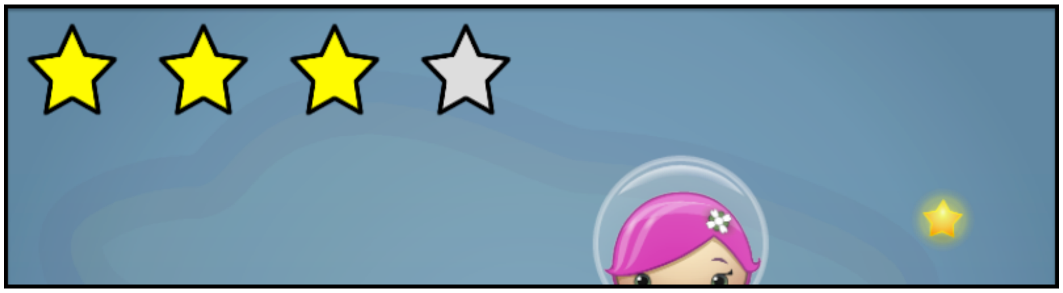
Method `UpdateStarText()` updates the text content of UI **Text** object `Text-carrying-star` with text string “`stars = `” concatenated with the integer value inside variable `totalStars`, to display the updated total number of stars to the user.

Displaying multiple pickups of same object with multiple status icons

If there is a small, fixed total number of an item to be collected rather than text totals, an alternative effective UI approach is to display ‘placeholder’ icons (empty or greyed out pictures) to show the user how many of the item remain to be collected; and each time an items is picked-up a placeholder icon is replaced by a full color ‘collected’ icon.

In this recipe we use grey-filled star icons as the placeholders, and yellow-filled star icons to indicated each collected star.

Since our UI code is getting a little more complicated, this recipe will implement the **model-view-controller** design pattern to separate the ‘view’ code from the core player logic (as introduced at the end of recipe: *Displaying single object pickups with carrying and not-carrying text*).



Insert image 1362OT_02_16.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 display multiple inventory icons for multiple pickups of same type of object follow these steps:

1. Start with a new copy of mini-game `Simple2Dgame_SpaceGirl`.
2. Add the following C# Script **`Player`** to gameObject `player-SpaceGirl` in the **Hierarchy**:

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

public class Player : MonoBehaviour {
   private PlayerInventoryDisplay playerInventoryDisplay;
   private int totalStars = 0;

   void Start(){
      playerInventoryDisplay = 
GetComponent<PlayerInventoryDisplay>();
   }

   void OnTriggerEnter2D(Collider2D hit){
      if(hit.CompareTag("Star")){
         totalStars++;
```

```
        playerInventoryDisplay.OnChangeStarTotal(totalStars);
                Destroy(hit.gameObject);
            }
        }
    }
```

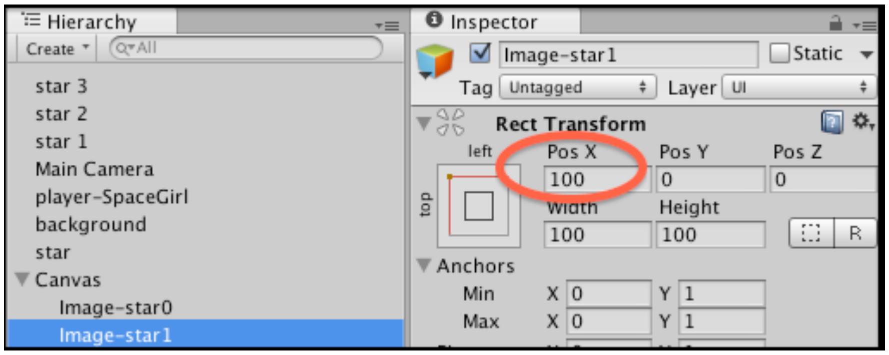
4. Select gameObject `star` in the **Hierarchy** panel, and make 3 more copies of this gameObject (Windows *CTRL-D* / Mac *CMD-D*).
5. Move these new gameObject to different parts of the screen.
6. Add the following C# Script `PlayerInventoryDisplay` to gameObject `player-SpaceGirl` in the **Hierarchy:**

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

public class PlayerInventoryDisplay : MonoBehaviour
{
    public Image[] starPlaceholders;

    public Sprite iconStarYellow;
    public Sprite iconStarGrey;

    public void OnChangeStarTotal(int starTotal){
        for (int i = 0;i < starPlaceholders.Length; ++i){
            if (i < starTotal)
                starPlaceholders[i].sprite = iconStarYellow;
            else
                starPlaceholders[i].sprite = iconStarGrey;
        }
    }
}
```

7. Select the Canvas in the **Hierachy** panel, and add a new UI Image object (**Create | UI | Image**). Rename it `Image-star0.`
8. Select `Image-star0` in the **Hierachy** panel.
9. From the **Project** panel drag sprite `icon_star_grey_100` (folder `Sprites`) into the **Source Image** field in the **Inspector** for the **Image** component.
10. Click the **Set Native Size** button for this for the **Image** component. This will resize the UI **Image** to fit the physical pixel width and height of sprite file `icon_star_grey_100`**.**
11. Now we will position our icon at the **top** and **left** of the **Game** panel. Edit the UI **Image's Rect Transform** component, while holding down *SHIFT* and *ALT* (to

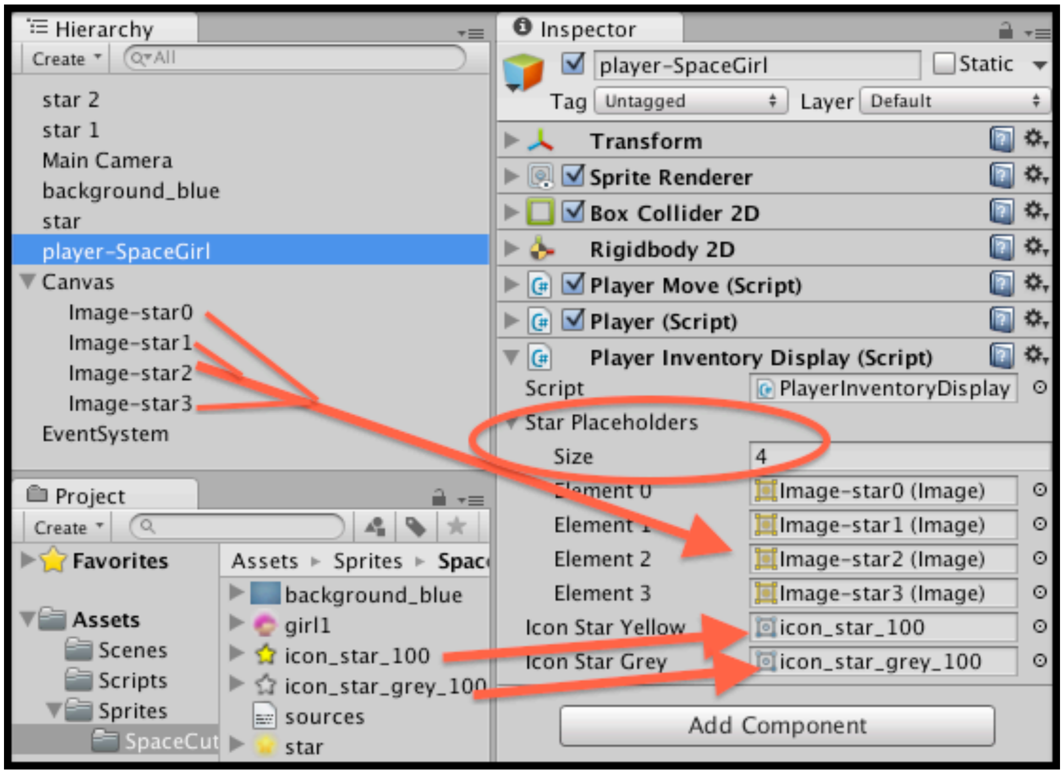
set pivot and position) choose the top-left box. The UI **Image** should now be positioned at the top-left of the **Game** panel.

12. Make 3 more copies of `Image-star0` in the **Hierachy** panel, naming them `Image-star1`, `Image-star2`, `Image-star3`.
13. In the **Inspector** panel change the **Pos X** position (in the **Rect Transform** component) of `Image-star1` to 100, of `Image-star2` to 200, and of `Image-star3` to 100.



Insert image 1362OT_02_17.png

14. From the **Hierarchy** view, select gameObject `player-SpaceGirl`. Then, from the **Inspector**, access the **Player Inventory Display (Script)** component and set the Size property of public field **Star Playholders** to `4`.
15. Next populate the **Element 0/1/2/3** array values of public field **Star Playholders** with UI **Image** objects `Image-star0/1/2/3`.
16. Now populate the **Icon Star Yellow** and **Icon Star Grey** public fields from the **Project** panel with sprite `icon_star_100` and `icon_star_grey_100`.



player-SpaceGirl   Static
Tag Untagged   Layer Default

Script: PlayerInventoryDisplay
Star Placeholders
Size: 4
Element 0: Image-star0 (Image)
Element 1: Image-star1 (Image)
Element 2: Image-star2 (Image)
Element 3: Image-star3 (Image)
Icon Star Yellow: icon_star_100
Icon Star Grey: icon_star_grey_100

Insert image 1362OT_02_18.png

17. Now when you play the scene, you should see the sequence of 4 grey placeholder star icons initally, and each time you collide with a star, the next icon at the top should turn yellow.

How it works...

Four UI **Image** objects `Image-star0/1/2/3` have been created at the top of the screen, initialized with the grey placeholder icon. The grey and yellow icon Sprite files have been rezised to be 100 x 100 pixels, making their arrangement horizontal positioning at design-time easier, since their positions are (0,0), (100, 0), (200, 0) and (300,0). In a more complicated game screen, or one where 'real estate' is precious the actual size of the icons would probably be smaller and whatever the game graphic designer decides.

The `int` variable `totalStars` represents how many stars have been collected so far; it is initialized to zero. The `PlayerInventory` variable `playerInventory` is a reference to the scripted component that manages our inventory display – this variable is set when the scene begins to run in method `Start()`.

In method `OnTriggerEnter2D()` the `totalStars` counter is incremented by 1 each time the player's character hits an object tagged "**Star**". As well as destroying the hit gameObject, method `OnChangeStarTotal(…)` of the `PlayerInventory` component is called, passing the new star total integer.

Method `OnChangeStarTotal(…)` of script class `PlayerInventory` has references to the four UI **Images**, and loops through each item in the array of Image references, setting the given number of Images to yellow, and the remaining to grey. This method is public, allowing it to be called from an instance of script class `Player`.

As can be seen, the code in script class `Player` is still quite straightforward, since we have moved all of the inventory UI logic to its own class, `PlayerInventory`.

Revealing icons for multiple object pickups by changing the size of a tiled image

Another approach that could be taken to show increasing numbers of images is to make use of tiled images. The same visual effect as in the previous recipe can also be achieved by making use of a tiled grey star image of width 400 (showing 4 copies of the grey star icon), **behind** a tiled yellow star image, whose width is 100 times the number of stars collected. We'll adapt the previous recipe to illustrate this technique.

Getting ready

This recipe follows on from the previous recipe in this chapter.

How to do it...

To display grey and yellow star icons for multiple object pickups using tiled images follow these steps:

1. Make a copy of your work for the previous recipe.
2. In the Hierarchy panel remove the 4 key UI Images in the Canvas.
3. Select the Canvas in the **Hierachy** panel, and add a new UI **Image** object (**Create | UI | Image**). Rename it `Image-stars-grey.`
4. Select `Image-stars-grey` in the **Hierachy** panel.
5. From the **Project** panel drag sprite `icon_star_grey_100` (folder `Sprites`) into the **Source Image** field in the **Inspector** (in the **Image (Script)** component).
6. Click the **Set Native Size** button for this for the **Image** component. This will resize the UI **Image** to fit the physical pixel width and height of sprite file **star_empty_icon.**

7. Now we will position our icon at the **top** and **left** of the **Game** panel. Edit the UI **Image's Rect Transform** component, while holding down *SHIFT* and *ALT* (to set pivot and position) choose the top-left box. The UI **Image** should now be positioned at the top-left of the **Game** panel.
8. In the **Inspector** panel change the **Width** (in the **Rect Transform** component) of `Image-stars-grey` to 400. Also set the Image Type (in the **Image (Script)** component) to **Tiled**.

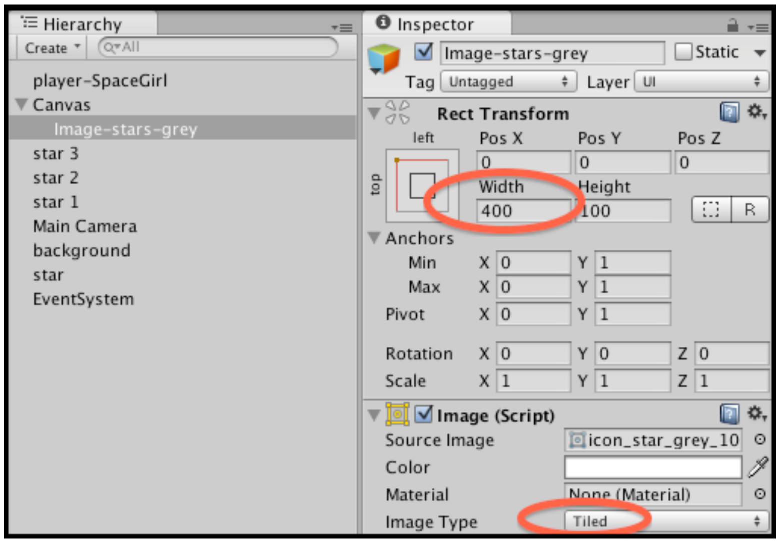


Image-stars-grey
Tag Untagged
Layer UI
Pos X 0, Pos Y 0, Pos Z 0
Width 400, Height 100
Anchors Min X 0, Y 1
Max X 0, Y 1
Pivot X 0, Y 1
Rotation X 0, Y 0, Z 0
Scale X 1, Y 1, Z 1
Source Image icon_star_grey_10
Color
Material None (Material)
Image Type Tiled

Insert image 1362OT_02_19.png

9. Make a copy of `Image-stars-grey` in the **Hierachy** panel, naming the copy `Image-stars-yellow`.
10. With `Image-stars-yellow` selected in the Hierarchy, from the **Project** panel drag sprite `icon_star_100` (folder `Sprites`) into the **Source Image** field in the **Inspector** (in the **Image (Script)** component).
11. Set the Width of `Image-stars-yellow` to 0 (in the **Rect Transform** component). So now we have the yellow stars tiled image above the grey tiled image, but since its width is zero, we don't see any of the yellow stars yet.
12. Replace the existing C# Script `PlayerInventoryDisplay` with the following code:

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

public class PlayerInventoryDisplay : MonoBehaviour
```

```
{
   public Image iconStarsYellow;

   public void OnChangeStarTotal(int starTotal){
      float newWidth = 100 * starTotal;

   iconStarsYellow.rectTransform.SetSizeWithCurrentAnchors(
RectTransform.Axis.Horizontal, newWidth);
   }
}
```

13. From the **Hierarchy** view, select gameObject `player-SpaceGirl`. Then, from the **Inspector**, access the **Player Inventory Display (Script)** component and populate the **Icons Stars Yellow** public field with UI **Image** object `Image-stars-yellow`.

How it works...

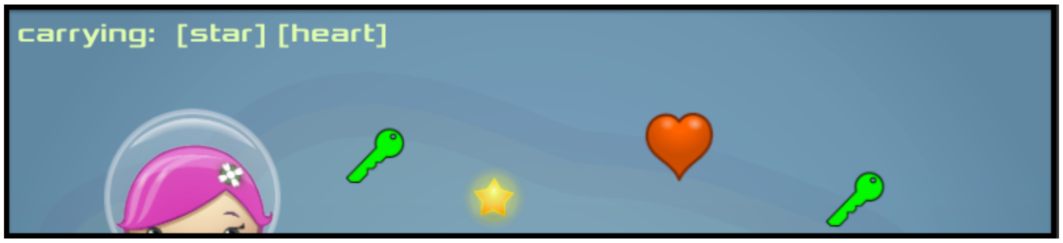
UI **Image** `Image-stars-grey` is a tiled image, width enough (400px) for it to be shown 4 times. UI **Image** `Image-stars-yellow` is a tiled image, above the grey one, initially with width set to zero, so no yellow stars can be seen.

Each time a star is picked-up a call is made to method `OnChangeStarTotal(…)` of script class `PlayerInventoryDisplay` ; passing the new integer number of stars collected. By multiplying this by the width of the yellow sprite image (100px) we get the correct width to set for UI **Image** `Image-stars-yellow` so that the corresponding number of yellow stars will now be seen by the user. Any stars remaining to be collected will still be seen as the grey stars that are not yet covered up.

The actual task of changing the width of UI **Image** `Image-stars-yellow` is completed by calling method `SetSizeWithCurrentAnchors(…)`. The first parameter is the 'axis', so we pass constant `RectTransform.Axis.Horizontal` so that it will be the width that is changed. The second parameter is the new size for that axis – so we pass a value that is 100 times the number of stars collected so far (variable `newWidth`).

Displaying multiple pickups of different objects as a list of text via a dynamic List<> of PickUp objects

When working with different kinds of pickups, one approach is to use a C# **List** to maintain a flexible-length data structure of the items currently in the inventory. In this recipe we show how each time an item is picked up a new object is added to such a **List** collection, and iteration through the **List** is how the text display of items is generated each time the inventory changes. We introduce a very simple PickUp script class, demonstrating how information about a pickup can be stored in a scripted component, and extracted upon collision and stored in our List.



Insert image 1362OT_02_36.png

Getting ready

This recipe assumes you are starting with project `Simple2Dgame_SpaceGirl` setup from the first recipe in this chapter. The font you need can be found in folder `1362_02_02`.

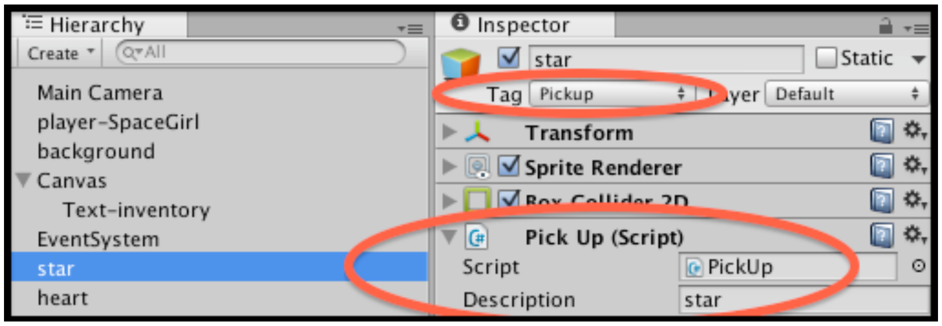
How to do it...

To display inventory total text for multiple pickups of different object types follow these steps:

1. Start with a new copy of mini-game `Simple2Dgame_SpaceGirl`.
2. Edit the tags, changing tag **Star** to **Pickup**. Ensure the `star` gameObject now has the tag **Pickup**.
3. Add the following C# Script `PickUp` to gameObject `star` in the **Hierarchy**:

```
using UnityEngine;
using System.Collections;

public class PickUp : MonoBehaviour {
   public string description;
}
```

4. In the **Inspector** change the description property of component **Pick Up (Script)** of gameObject `star` to the text `star.`



Insert image 1362OT_02_37.png

5. Select gameObject `star` in the **Hierarchy** panel, and make a copy of this gameObject, renaming the copy `heart`.
6. In the **Inspector** change the description property of component **Pick Up (Script)** of gameObect `heart` to the text `heart`. Also drag from the **Project** panel (folder Sprites) image **healthheart** into the Sprite property of gameObject `heart`. The player should now see the heart image on screen for this pickup item.
7. Select gameObject `star` in the **Hierarchy** panel, and make a copy of this gameObject, renaming the copy `key`.
8. In the **Inspector** change the description property of component Pick Up (Script) of gameObect `key` to the text `key.` Also drag from the **Project** panel (folder Sprites) image **icon_key_green_100** into the Sprite property of gameObject `key`. The player should now see the key image on screen for this pickup item.
9. Make another one or two copies of each pickup gameObject, and arrange them around the screen, so there are 2 or 3 each of star, heart and key pickup gameObjects.
10. Add the following C# Script `Player` to gameObject `player-SpaceGirl` in the **Hierarchy**:

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

public class Player : MonoBehaviour {
    private PlayerInventoryDisplay playerInventoryDisplay;
    private List<PickUp> inventory = new List<PickUp>();

    void Start(){
        playerInventoryDisplay =
GetComponent<PlayerInventoryDisplay>();
```

```
        playerInventoryDisplay.OnChangeInventory(inventory);
    }

    void OnTriggerEnter2D(Collider2D hit){
        if(hit.CompareTag("Pickup")){
            PickUp item = hit.GetComponent<PickUp>();
            inventory.Add( item );

    playerInventoryDisplay.OnChangeInventory(inventory);
            Destroy(hit.gameObject);
        }
    }
}
```

11. Add a UI **Text** object (**Create | UI | Text**). Rename it `Text-inventory-list`. Change its text to "**the quick brown fox jumped over the lazy dog the quick brown fox jumped over the lazy dog**" or some other long list of nonsense words, to test the overflow settings you change in the next step.
12. In the **Text (Script)** component ensure **Horitzonal Overflow** is set to **Wrap**, and set **Vertical Overflow** to **Overflow** – this will ensure text will wrap onto a second or third line (if needed) and not be hidden if there are lots of pickups.
13. In the **Inspector** panel also set its font to '**Xolonium-Bold**' (folder `Fonts`), and set its color to yellow. For the **Alignment** propety center the text horizontally, and ensure the text is top-aligned vertically, and set the **Font Size** to 28 and choose a yellow text **Color**.
14. Edit its **Rect Transform**, set its height to 50. Then, while holding down *SHIFT* and *ALT* (to set pivot and position) choose the top-stetch box. The text should now be positioned at the middle top of the **Game** panel, and its width should stretch to match that of the whole panel.
15. Your text should now appear at the top of the game panel.
16. Add the following C# Script `PlayerInventoryDisplay` to gameObject `player-SpaceGirl` in the **Hierarchy**:

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

public class PlayerInventoryDisplay : MonoBehaviour
{
    public Text inventoryText;

    public void OnChangeInventory(List<PickUp> inventory){
        // (1) clear existing display
```

```
            inventoryText.text = "";

            // (2) build up new set of items
            string newInventoryText = "carrying: ";
            int numItems = inventory.Count;
            for(int i = 0; i < numItems; i++){
               string description = inventory[i].description;
               newInventoryText += " [" + description+ "]";
            }

            if(numItems < 1) newInventoryText = "(empty
        inventory)";

            // (3) update screen display
            inventoryText.text = newInventoryText;
         }
      }
```

17. From the **Hierarchy** view, select gameObject `player-SpaceGirl`. Then, from the **Inspector**, access the **Player Inventory Display (Script)** component and populate the **Inventory Text** public field with UI **Text** object `Text-inventory-list`.
18. Play the game – each time you pickup a star or key or heart, the updated list of what you are carrying should be displayed in the form "**carrying: [key] [heart]**".

How it works...

In script class `Player` variable `inventory` is a C# List. This is a flexible data structure, which can be sorted, searched, and dynamically (at run time, when the game is being played) have items added to and removed from. The `<PickUp>` in pointy brackets means that variable `inventory` will contain a List of `PickUp` objects. For this recipe our `PickUp` class just has a single field, a string description, but we'll add more sophisticated data items in `PickUp` classes in later recipes.

When the scene starts, the `Start()` method of script class `Player` gets a reference to the `PlayerInventoryDisplay` scripted component, and also initializes variable `inventory` to be a new, empty C# List of `PickUp` objects. When the `OnColliderEnter2D(…)` method detects collisions with items tagged Pickup, the PickUp object component of the item hit is added to our `inventory` List. A call is also made to method `OnChangeInventory(…)` of `playerInventoryDisplay` to update out inventory display to the player, passing the updated `inventory` List as a parameter.

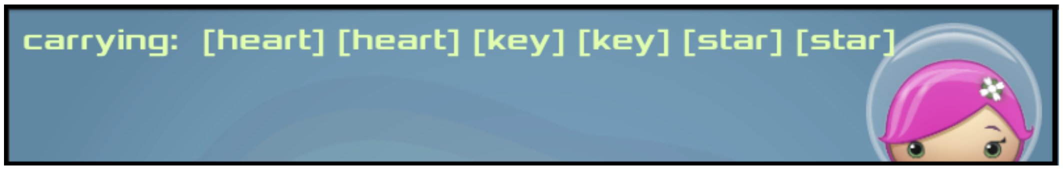
Script class `playerInventoryDisplay` has a public variable, linked to the UI **Text** object `Text-inventory-list`. Method `OnChangeInventory(…)` first sets the UI text to empty, then loops through the inventory list, building up a string of each items description in square brackets ([key] [heart] etc.). If there were no items in the list then the string is set to the text `(empty inventory)`. Finally the text property of UI **Text** object `Text-inventory-list` is set to the value of this string representation of what is inside variable `inventory`.

There's more...

Some details you don't want to miss:

Order items in inventory list alphabetically

It would be nice to alphabetically sort the words in the `inventory` list – both for neatness and consistency (so in a game if we pick up a key and a heart it will look the same regardless of which order), but also so items of the same type will be listed together, so we can easily see how many of each item we are carrying.



Insert image 1362OT_02_38.png

To implement alphabetic sorting of the items in the `inventory` list we need to do the following:

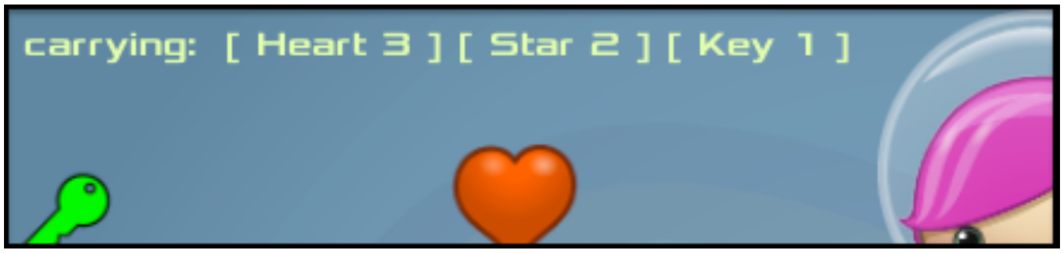
1. Add the following C# code to the beginning of method `OnChangeInventory(...)` in script class `PlayerInventoryDisplay`:

```
public void OnChangeInventory(List<PickUp> inventory){
    inventory.Sort(
        delegate(PickUp p1, PickUp p2){
            return p1.description.CompareTo(p2.description);
        }
    );

    // rest of the method as before …
}
```

You should now see all the items listed in alphabetic sequence. This C# code takes advantage of the `List.Sort(…)` method, a feature of collections whereby each item can be compared to the next, and they are swapped if in the wrong order (if the `CompareTo(…)` methods returns false).

Displaying multiple pickups of different objects as text totals via a dynamic Dictionary<> of PickUp objects and 'enum' pickup types

While the previous recipe worked fine, any old text might have been typed into the description for a pickup, or perhaps mistyped '**star**' and '**Sstar**' and '**starr**' etc. A much better way of restricting game properties to one of a pre-defined (enumerated) list of possible values is to use C# 'enums'. As well as removing the chance of mistyping a string, it also means we can write code to appropriately deal with the pre-defined set of possible values. In this recipe we improve our general purpose PickUp class by introducing 3 possible pickup types (Star, Heart and Key), and write inventory display code that counts the number of each type of pickup being carried, and displays these totals via a UI **Text** object on screen. We also switch from using a List to using a Dictionary, since this data structure is designed specifically for key-value pairs, perfect for associating a numeric total with an enumerated pickup 'type'.



Insert image 1362OT_02_39.png

Getting ready

This recipe follows on from the previous recipe in this chapter.

How to do it...

To display multiple pickups of different objects as text totals via a dynamic `Dictionary<>` follow these steps:

1. Make a copy of your work for the previous recipe.
2. Replace the content of script class `PickUp` with the following code:

```
using UnityEngine;
using System.Collections;

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

   public PickUpType type;
}
```

3. Replace the content of script class `Player` with the following code:

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

public class Player : MonoBehaviour {
   private InventoryManager inventoryManager;

   void Start(){
      inventoryManager = GetComponent<InventoryManager>();
   }

   void OnTriggerEnter2D(Collider2D hit){
      if(hit.CompareTag("Pickup")){
         PickUp item = hit.GetComponent<PickUp>();
         inventoryManager.Add( item );
         Destroy(hit.gameObject);
      }
   }
}
```

4. Replace the content of script class `PlayerInventoryDisplay` with the following code:

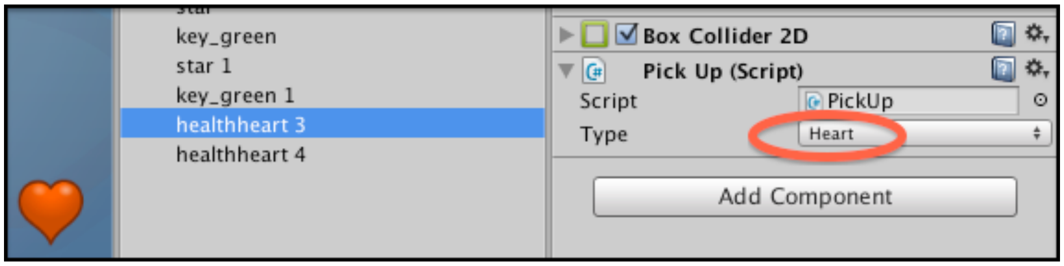
```
using UnityEngine;
using System.Collections;
using UnityEngine.UI;
using System.Collections.Generic;

public class PlayerInventoryDisplay : MonoBehaviour {
   public Text inventoryText;
   private string newInventoryText;

   public void
OnChangeInventory(Dictionary<PickUp.PickUpType, int>
inventory){
      inventoryText.text = "";

      newInventoryText = "carrying: ";
```

```
        int numItems = inventory.Count;

        foreach(var item in inventory){
           int itemTotal = item.Value;
           string description = item.Key.ToString();
           newInventoryText += " [ " + description + " " +
itemTotal +  " ]";
        }

        if(numItems < 1) newInventoryText = "(empty
inventory)";

        inventoryText.text = newInventoryText;
     }
  }
```

5. Add the following C# Script `InventoryManager` to gameObject `player-SpaceGirl` in the **Hierarchy**:

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

public class InventoryManager : MonoBehaviour {
   private PlayerInventoryDisplay playerInventoryDisplay;
   private Dictionary<PickUp.PickUpType, int> items = new
Dictionary<PickUp.PickUpType, int>();

   void Start(){
      playerInventoryDisplay =
GetComponent<PlayerInventoryDisplay>();
      playerInventoryDisplay.OnChangeInventory(items);
   }

   public void Add(PickUp pickup){
      PickUp.PickUpType type = pickup.type;
      int oldTotal = 0;
      if(items.TryGetValue(type, out oldTotal))
         items[type] = oldTotal + 1;
      else
         items.Add (type, 1);

      playerInventoryDisplay.OnChangeInventory(items);
   }
}
```

6. In the **Hierarchy** (or **Scene**) panel, select each pickup **gameObject** in turn, and choose from the dropdown menu its corresponding **Type** in the **Inspector**. As you can see, public varibles that are of an `enum` type are automatically restricted to the set of possible values as a combo-box drop down menu in the **Inspector**.



Script: PickUp
Type: Heart

Insert image 1362OT_02_40.png

7. Play the game. First you should see a message on screen stating the inventory is empty, then as you pick up one or more items of each pickup type you'll see text totals of each type you have collected.

How it works...

Each pickup gameObject in the scene has a scripted component of class `PickUp`. The `PickUp` object for each pickup gameObject has a single property, a pickup type, which has to be one of the enumerated set of `Star, Key, Heart`. The `Player` script class gets a reference to the `InventoryManager` component via its `Start()` method, and each time the player's character collides with a pickup gameObject it calls the `Add(…)` method of the inventory manager passing the `PickUp` object of the object collided with.

In this recipe the inventory being carried by the player is being represented by a C# `Dictionary<>`. In this case we have in script class `InventoryManager` a Dictionary of **key-value** pairs, where the **key** is one of the possible `PickUp.PickUpType` enumerated values, and the **value** is an integer total of how many of that type of pickup is being carried. Each `InventoryItemTotal` object has just 2 properties: a `PickUp` type and an integer total. This extra 'layer' of the `InventoryManager` has been added between script class `Player` and `PlayerInventoryDisplay` to both separate the `Player` behavior from how the inventory is internally stored, and also to prevent the `Player` script class from becoming too large and attempting to handle too many different responsibilities.

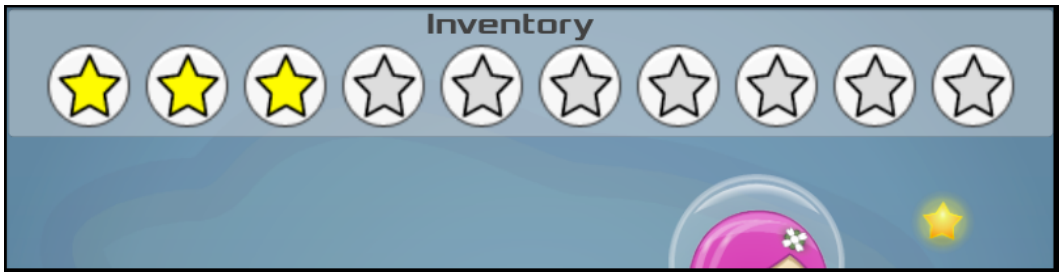
C# Dictionaries provide a method `TryGetValue(…)`, which receives parameters of a key, and is passed a reference to a variable the same data type as the value for the `Dictionary`. When the `Add(…)` method of the inventory manager is called, the type of the `PickUp` object is tested, to see if a total for this type is already in `Dictionary<> items`. If an item total is found inside the `Dictionary` for the given type, then the value for this item in the `Dictionary` is incremented. If no entry is found for the given type, then a new element is added to the `Dictionary` with a total of 1.

The last action of method `Add(…)` is to call method `OnChangeInventory(…)` of the `PlayerInventoryDisplay` scripted component of the player's gameObject, to update the text totals displayed on screen. This method in `PlayerInventoryDisplay` iterates through the `Dictionary`, building up a string of the type names and totals, and then updates the text property of the UI Text object with the string showing the inventory totals to the player.

Learn more about using C# Lists and Dictionaries in Unity in this Unity Technologies tutorial:
https://unity3d.com/learn/tutorials/modules/intermediate/scripting/lists-and-dictionaries

Generalizing multiple icon displays using UI Grid Layout Groups (with scrollbars!)

The recipes in this chapter up to this point have been 'hand-crafted' for each situation. While this is fine, sometimes more general and 'automated' approaches to inventory UIs can save time and effort, and still achieve visual and usability results of equal quality. In this next recipe we begin to explore a more 'engineered' approach to inventory UIs, by exploiting the automated sizing and layouts offered by Unity 5's Grid Layout Group component.



Insert image 1362OT_02_20.png

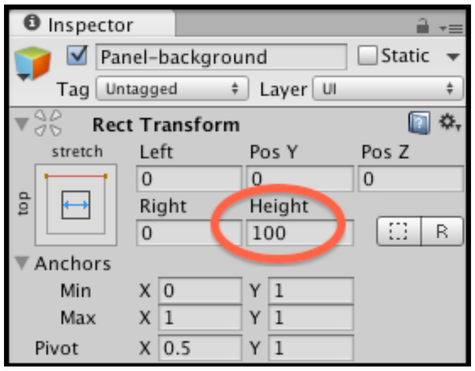
Getting ready

This recipe assumes you are starting with project `Simple2Dgame_SpaceGirl` setup from the first recipe in this chapter. The font you need can be found in folder `1362_02_02`.

How to do it...

To display grey and yellow star icons for multiple object pickups using UI Grid Layout Groups follow these steps:

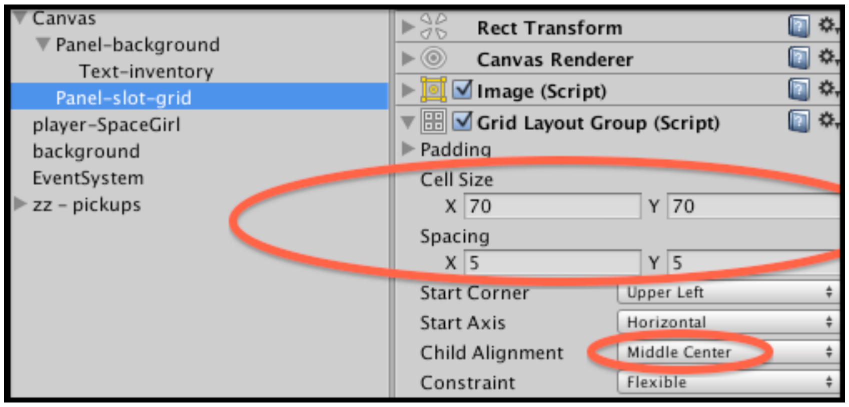
1. Start with a new copy of mini-game `Simple2Dgame_SpaceGirl`.
2. In the **Hierachy** panel create a UI Panel `Panel-background` (**Create | UI | Panel**).
3. Let's now position `Panel-background` at the **top** of the **Game** panel, stretched the horizontal width of the canvas. Edit the UI **Image's Rect Transform** component, while holding down *SHIFT* and *ALT* (to set pivot and position) choose the top-stretch box. Now choose the top-stetch box.
4. The panel will still be taking up the whole game window, so now in the **Inspector** panel change the **Height** (in the **Rect Transform** component) of `Panel-background` to 100.


Inspector
Panel-background · Static
Tag: Untagged · Layer: UI
Rect Transform
stretch / top
Left: 0 · Pos Y: 0 · Pos Z: 0
Right: 0 · Height: 100
Anchors
Min: X 0 · Y 1
Max: X 1 · Y 1
Pivot: X 0.5 · Y 1

Insert image 1362OT_02_21.png

5. Add a UI **Text** object (**Create | UI | Text**). Rename it `Text-inventory`. Change its tet to "**Inventory**".
6. In the **Hierachy** panel child this UI Text object to panel `Panel-background`.
7. In the **Inspector** panel also set the font of `Text-inventory` to '**Xolonium-Bold**' (folder `Fonts`). Center the text horizontally, and top align the text vertically, and set its height to 50, and set the **Font Size** to 23.
8. Edit the **Rect Transform** of `Text-inventory` while holding down *SHIFT* and *ALT* (to set pivot and position) choose the top-stretch box. The text should now be positioned at the middle top of the **UI** Panel `Panel-background`, and its width should stretch to match that of the whole panel.
9. Select the Canvas in the **Hierachy** panel, and add a new UI Panel object (**Create | UI | Image**). Rename it `Panel-slot-grid`.

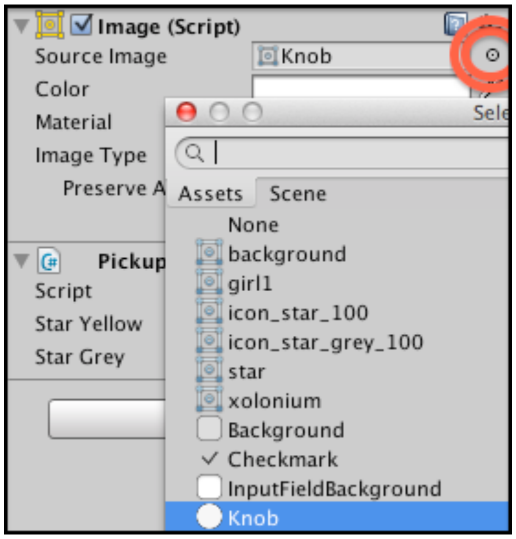
10. Position `Panel-slot-grid` at the **top** of the **Game** panel, stretched the horizontal width of the canvas. Edit the UI **Image's Rect Transform** component, while holding down *SHIFT* and *ALT* (to set pivot and position) choose the top-stretch box. Now choose the top-stetch box.
11. In the **Inspector** panel change the **Height** (in the **Rect Transform** component) of `Panel-slot-grid` to 80, and set its **Top** to -20(so it is below UI **Text** gameObject `Text-inventory`).
12. With Panel `Panel-slot-grid` selected in the **Hierachy** panel, add a Grid Layout Group component (**Add Component | Layout | Grid Layout Group**). Set cell size to 70x70, and Spacing to 5x5. Also set the Child Alignment to Middle Center (so our icons will have even spacing at the far left and right).



Cell Size
X 70 Y 70
Spacing
X 5 Y 5
Start Corner: Upper Left
Start Axis: Horizontal
Child Alignment: Middle Center
Constraint: Flexible

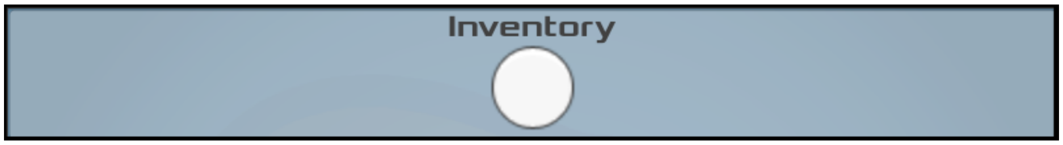
Insert image 1362OT_02_22.png

13. With Panel `Panel-slot-grid` selected in the **Hierachy** panel, add a Mask (script) component (**Add Component | UI | Mask**). Un-check option **Show Mask Graphic**. Having this mask component means that any overflow of our grid will NOT be seen by the user – only content within the Image area of Panel `Panel-slot-grid` will ever be visible.
14. Add to your Canvas a UI **Image** object (**Create | UI | Image**). Rename it `Image-slot`.
15. In the **Hierachy** panel child UI **Image** object `Image-slot` to panel `Panel-slot-grid`.
16. Set the **Source Image** of `Image-slot` to the Unity provided **Knob** (circle) image.



Insert image 1362OT_02_23.png

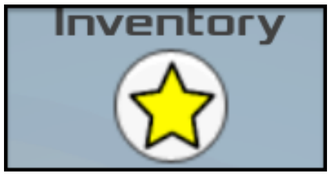
17. Since `Image-slot` is the only UI object inside `Panel-slot-grid` then it will be displayed (sized 70x70) centered in that panel.



Insert image 1362OT_02_24.png

18. Each image slot will have a yellow star child image, and a grey star child image. Let's create those now.
19. Add to your Canvas a UI **Image** object (**Create | UI | Image**). Rename it `Image-star-yellow`.
20. In the **Hierachy** panel child UI **Image** object `Image-star-yellow` to image `Image-slot`.
21. Set the **Source Image** of `Image-star-yellow` to the `icon_star_100` image (in folder `Sprites`).
22. Now we will set our yellow star icon image to fully fill its parent `Image-slot` by stretching horizotnally and vertically. Edit the UI **Image's Rect Transform** component, while holding down *SHIFT* and *ALT* (to set pivot and position) choose bottom right option, to fully **stretch** horizontally and vertically. UI **Image**

`Image-star-yellow` should now be visible in the middle of the `Image-slot` circular **Knob** image.



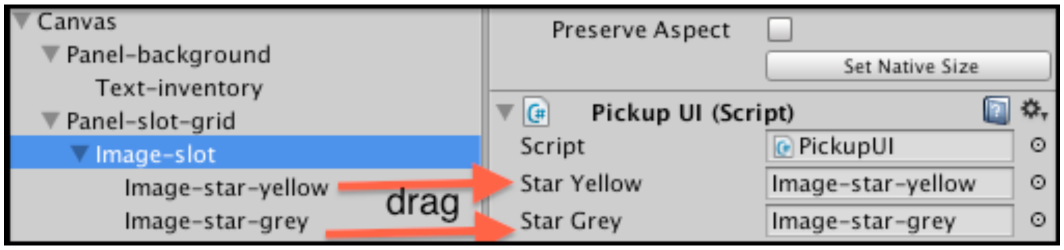
Insert image 1362OT_02_25.png

23. Duplicate `Image-star-yellow` in the **Hierachy** panel, naming the copy `Image-star-grey`. This new gameObject should also be a child of `Image-slot`.
24. Change the **Source Image** of `Image-star-grey` to the `icon_star_grey_100` image (in folder `Sprites`). At any time, our inventory slot can now display nothing, a yellow star icon, or a grey star icon, depending on whether `Image-star-yellow` and `Image-star-grey` are enabled or not: we'll control this through the inventory display code later in this recipe.
25. In the **Hierachy** panel ensure `Image-slot` is selected, and add C# Script `PickupUI` with the following code:

```
using UnityEngine;
using System.Collections;

public class PickupUI : MonoBehaviour {
   public GameObject starYellow;
   public GameObject starGrey;

   void Awake(){
      DisplayEmpty();
   }

   public void DisplayYellow(){
      starYellow.SetActive(true);
      starGrey.SetActive(false);
   }

   public void DisplayGrey(){
      starYellow.SetActive(false);
      starGrey.SetActive(true);
   }

   public void DisplayEmpty(){
      starYellow.SetActive(false);
```

```
            starGrey.SetActive(false);
        }
    }
```

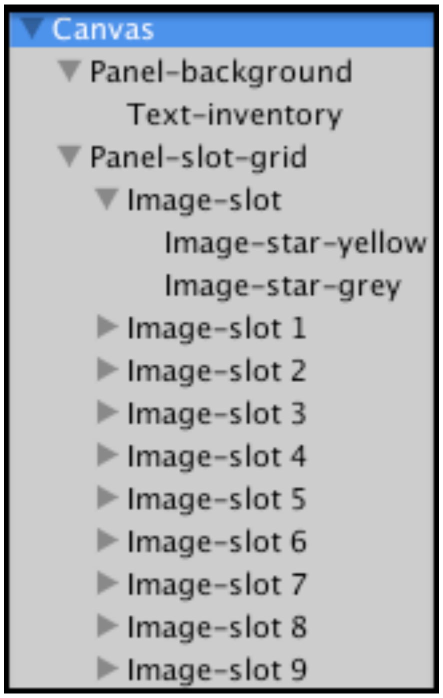
26. With gameObject `Image-slot` selected in the **Hierarchy** panel, drag each of its two children `Image-star-yellow` and `Image-star-grey` into their corresponding **Inspector** panel **Pickup UI** slots **Star Yellow** and **Star Grey**.



Script: PickupUI
Star Yellow: Image-star-yellow
Star Grey: Image-star-grey

Insert image 1362OT_02_27.png

27. In the **Hierachy** panel make 9 duplicates of `Image-slot` in the **Hierachy** panel, they should automatically be named `Image-slot 1 .. 9`. See the screenshot to ensure the Hierarchy of your Canvas is correct – the parenting of `Image-slot` as a child of `Image-slot-grid`, and the parenting of `Image-star-yellow` and `Image-star-grey` as children of each `Image-slot` is very important.



Insert image 1362OT_02_26.png

28. In the **Hierachy** panel ensure `player-SpaceGirl` is selected, and add C# Script `Player` with the following code:

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

public class Player : MonoBehaviour {
   private PlayerInventoryModel playerInventoryModel;

   void Start(){
      playerInventoryModel =
GetComponent<PlayerInventoryModel>();
   }

   void OnTriggerEnter2D(Collider2D hit){
      if(hit.CompareTag("Star")){
         playerInventoryModel.AddStar();
         Destroy(hit.gameObject);
      }
   }
}
```

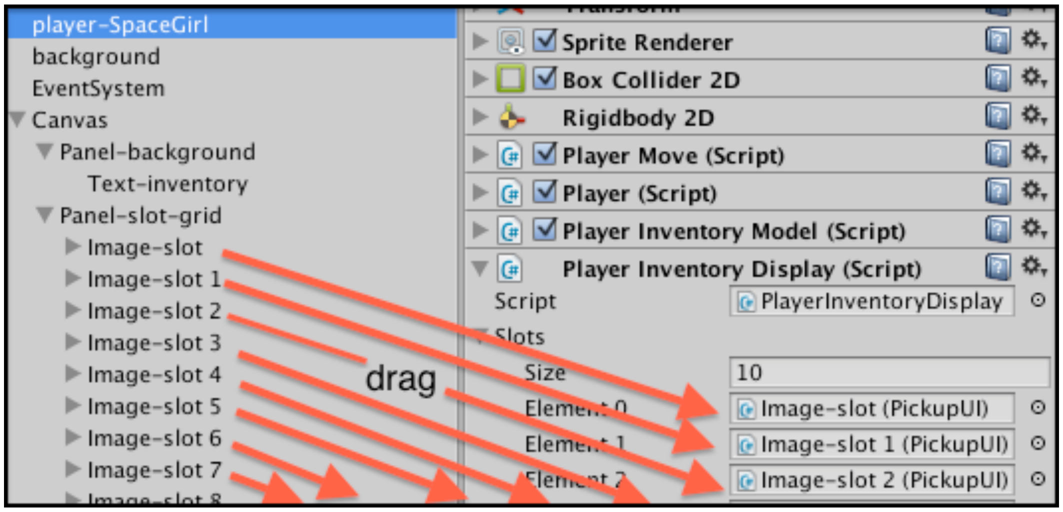
29. In the **Hierachy** panel ensure `player-SpaceGirl` is selected, and add C# Script `PlayerInventoryModel` with the following code:

```
using UnityEngine;
using System.Collections;

public class PlayerInventoryModel : MonoBehaviour {
   private int starTotal = 0;
   private PlayerInventoryDisplay playerInventoryDisplay;

   void Start(){
      playerInventoryDisplay =
GetComponent<PlayerInventoryDisplay>();
      playerInventoryDisplay.OnChangeStarTotal(starTotal);
   }

   public void AddStar(){
      starTotal++;
      playerInventoryDisplay.OnChangeStarTotal(starTotal);
   }
}
```

30. In the **Hierachy** panel ensure `player-SpaceGirl` is selected, and add C# Script `PlayerInventoryDisplay` with the following code:

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

public class PlayerInventoryDisplay : MonoBehaviour
{
   const int NUM_INVENTORY_SLOTS = 10;
   public PickupUI[] slots = new
PickupUI[NUM_INVENTORY_SLOTS];

   public void OnChangeStarTotal(int starTotal){
      for(int i = 0; i < NUM_INVENTORY_SLOTS; i++){
         PickupUI slot = slots[i];
         if(i < starTotal)
            slot.DisplayYellow();
         else
            slot.DisplayGrey();
      }
   }
}
```

31. With gameObject `player-SpaceGirl` selected in the **Hierarchy** panel, drag the ten `Image-slot` gameObjects into their corresponding locations in the **Player Inventory Display (Script)** component array **Slots**, in the **Inspector** panel.



Insert image 1362OT_02_28.png

32. Save the scene and play the game. As you pickup stars you should see more of the grey stars change to yellow in the inventory display.

How it works...

We have created a simple panel (`Panel-background`) and text at the top of the game canvas – showing a greyish background rectangle and text "Inventory". We created a small panel inside this area (`Panel-slot-grid`), with a Grid Layout Group component, which automatically sizes and lays out the 10 `Image-slot` gameObjects we created with the Knob (circle) Source Image. By adding a Mask component to `Panel-slot-grid` we ensure no content will 'overflow' outside of the rectangle of the Source Image for this panel.

Each of the 10 `Image-slot` gameObjects that are children of `Panel-slot-grid` contains a yellow star image and a grey-star image. Also each `Image-slot` gameObjects has a script component `PickupUI`. The `PickupUI` script offers 3 public methods, which will show just the yellow star image, just the grey star image, or neither (so an empty Knob circle image will be seen).

Our player's character gameObject `player-SpaceGirl` has a very simple basic Player script – this just detected collisions with objects tagged 'Star', and when this happens it removes the star gameObject collided with, and calls method `AddStar()` to its `playerInventoryModel` scripted component. The `PlayerInventoryModel` C# script class maintains a running integer total of the number of stars added to the inventory. Each time method `AddStar()` is called it increments (adds 1) to this total, and then calls the `OnChangeStarTotal(…)` method of scripted component `playerInventoryDisplay`. Also when the scene starts, an initial call is made to method `OnChangeStarTotal(…)`, so that the UI display for the inventory is setup to show that we are initially carrying no stars.

C# script class `PlayerInventoryDisplay` has 2 variables, one is a constant integer defining the number of 'slots' in our inventory, which for this game we set to 10. The other variable is an array of references to PickupUI scripted components – each of these is a reference to the scripted component in each of the 10 `Image-slot` gameObjects in our `Panel-slot-grid`. When method `OnChangeStarTotal(…)` being passed the number of stars we are carrying, it loops through each of the 10 'slots'. While the current slot is less than our star total, a yellow star is displayed, by the calling of method `DisplayYellow()` of the current slot (`PickupUI` scripted component). Once the loop counter is equal to or larger than our star total, then all remaining slots are made to display a grey star, via the calling of method `DisplayGrey()`.

This recipe is an example of the **low coupling** of the MVC (**Model-View-Controller**) design pattern. We have designed our code to not rely or make too many assumptions about other parts of the game, so that the chances of a change in some other part of our game 'breaking' our inventory display code is much smaller. The display (View) is separated from the logical representation of what we are carrying (Model), and changes to the Model are made by public methods called from the Player (Controller).

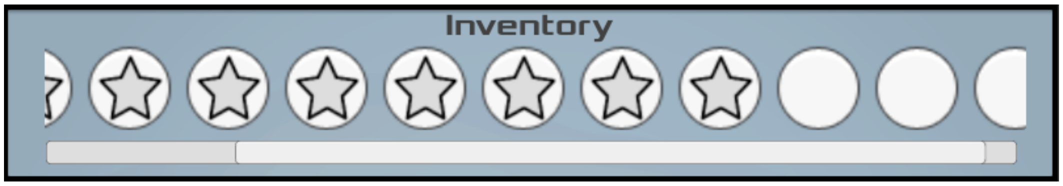
NOTE
It might seem that we could make our code simpler by assuming that slots are always displaying grey (no star), and just changing one slot to yellow each time a yellow star is picked-up. But this would lead to problems if something happens in the game (e.g. hitting a black hole, or being shot by an alien) that makes us drop one or more stars. C# script class `PlayerInventoryDisplay` makes no assumptions about which slots may or may not have been displayed grey or yellow or empty previously – each time it is called it ensures an appropriate number of yellow stars are displayed, and all other slots are displayed with grey stars.

There's more...

Some details you don't want to miss:

Add a horizontal scrollbar to the inventory slot display

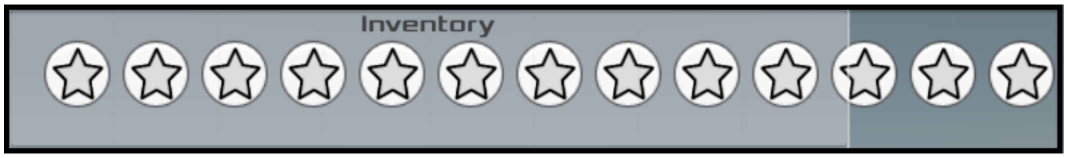
We can see 10 inventory slots now – but what if there are many more? One solution is to add a scroll bar, so that the user can scroll left and right, viewing 10 at a time. Let's add a horizontal scroll bar to our game. This can be achieved without any C# code changes, all through the Unity 5 UI system.



Insert image 1362OT_02_35.png

To implement a horizontal scrollbar for our inventory display we need to do the following:

1. Increase the Height of `Panel-background` to 130 pixels.
2. In the Inspector set the **Child Alignment** property of component **Grid Layout Group (Script)** of `Panel-slot-grid` to Upper Left. Then move this panel to the right a little, so that the 10 inventory icons are centered on screen.
3. In the Hierarchy duplicate Image-slot 9 three more times, so that there are now 13 inventory icons in `Panel-slot-grid`.
4. In the Scene panel drag the right hand edge of panel `Panel-slot-grid` to make it wide enough so that all 13 inventory icons fit horizontally – of course the last 3 will be off screen.



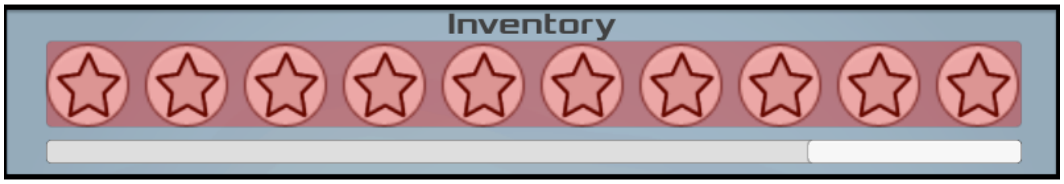
Insert image 1362OT_02_33.png

5. Add a UI **Panel** to the **Canvas**, and name it `Panel-scroll-container`, and tint it red, by seting the **Color** property of its **Image (Script)** component to red.
6. Size and position `Panel-scroll-container` so that it is just behind our `Panel-slot-grid`. So you should now see a red rectangle behind the 10 inventory circle slots.
7. In the **Hierarchy** drag `Panel-slot-grid` so that it is now childed to `Panel-scroll-container`.

8. Add a UI **Mask** to `Panel-scroll-container` so now you should only be able to see the 10 inventory icons that fit within the rectangle of this red-tinted panel.

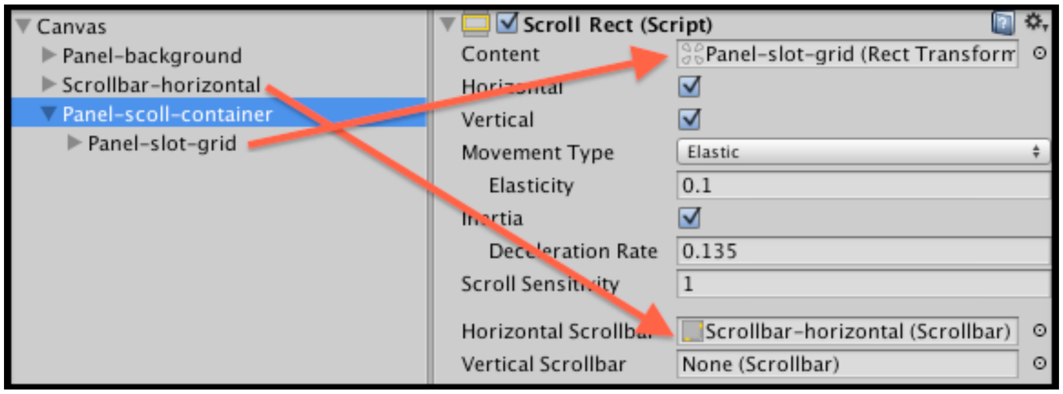
> NOTE
>
> You may wish to temporarily set as inactive this Mask component, so you can see and work on the unseen parts of `Panel-slot-grid` if required.

9. Add a UI **Scrollbar** to the **Canvas**, and name it `Scrollbar-horizontal`, and move it to be just below the 10 inventory icons, and the same width as the red-tinted `Panel-scroll-container`.



Insert image 1362OT_02_32.png

10. Add to `Panel-scroll-container` a UI **Scroll Rect** component.
11. In the **Inspector** drag `Scrollbar-horizontal` to the Horizontal Scrollbar property of the **Scroll Rect** component of `Panel-scroll-container`.
12. In the **Inspector** drag `Panel-slot-grid` to the Content property of the **Scroll Rect** component of `Panel-scroll-container`.



Scroll Rect (Script)
Content: Panel-slot-grid (Rect Transform)
Horizontal: ✓
Vertical: ✓
Movement Type: Elastic
Elasticity: 0.1
Inertia: ✓
Deceleration Rate: 0.135
Scroll Sensitivity: 1
Horizontal Scrollbar: Scrollbar-horizontal (Scrollbar)
Vertical Scrollbar: None (Scrollbar)

Insert image 1362OT_02_34.png

13. Now ensure the **Mask** component of of `Panel-scroll-container` is set as active, so we don't see the 'overflow' of `Panel-slot-grid`, and uncheck this **Mask** components option to Show Mask Graphic, (so we don't see the red rectangle any more).

You should now have a working scrollable inventory system. Note that the last three new icons will just be empty circles, since the inventory display script does not have references to or attempt to make any changes to these extra 3 'slots'; so the script code would need to be changed to reflect every additional slot we add to `Panel-slot-grid`.

Automation of PlayerInventoryDisplay getting references to all the slots…

There was a lot of dragging slots from the Hierarchy panel into the array for scripted component `PlayerInventoryDisplay`. This takes a bit of work (and mistakes might be made dragging items in the wrong order, or the same item twice). Also if we change the number of slots, then we may have to do this all over again, or try to remember to drag more slots if we increase the number and so on. A better way of doing things is to make the first task of script class `PlayerInventoryDisplay` when the scene begins to create each of these `Image-slot` gameObjects as a child of `Panel-slot-grid`, and populate the array at the same time.

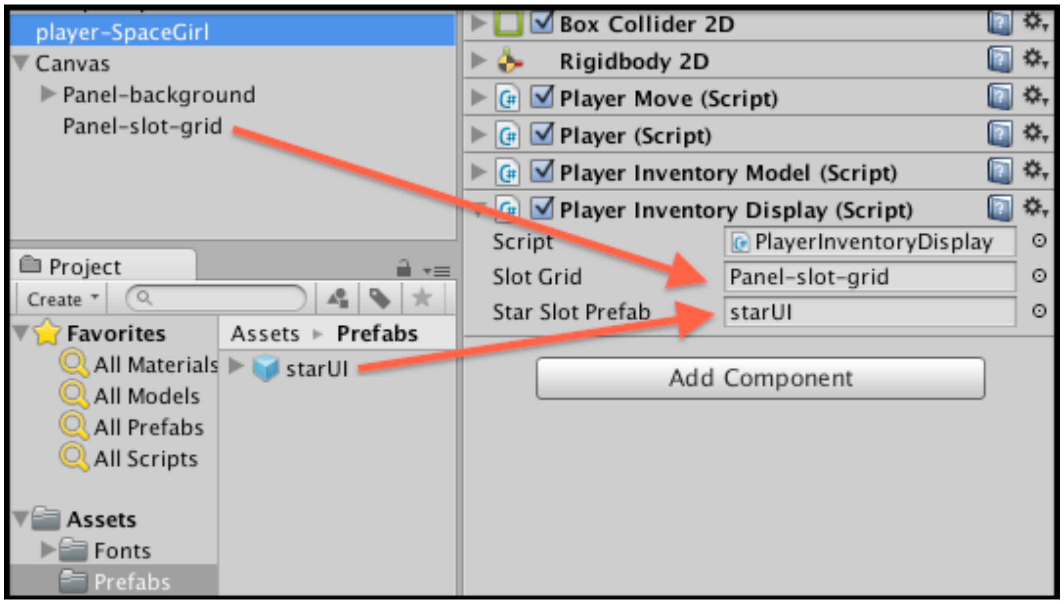
To implement the automated population of our scripted array of PickupUI objects for this recipe we need to do the following:

1. Create a new folder named `Prefabs`. In this folder create a new empty prefab named `starUI`.
2. From the **Hiearchy** drag gameObject `Image-slot` into your new new empty prefab named `starUI`. This prefab should now turn blue, showing it is populated.
3. In the **Hiearchy** delete gameObject `Image-slot`, and all its copies `Image-slot 1 - 9`.
4. Replace C# Script `PlayerInventoryDisplay` in gameObject `player-SpaceGirl` with the following code:

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

public class PlayerInventoryDisplay : MonoBehaviour
```

```
{
   const int NUM_INVENTORY_SLOTS = 10;
   private PickupUI[] slots = new
PickupUI[NUM_INVENTORY_SLOTS];
   public GameObject slotGrid;
   public GameObject starSlotPrefab;

   void Awake(){
      for(int i=0; i < NUM_INVENTORY_SLOTS; i++){
         GameObject starSlotGO =
   (GameObject)Instantiate(starSlotPrefab);

      starSlotGO.transform.SetParent(slotGrid.transform);
         starSlotGO.transform.localScale = new
   Vector3(1,1,1);

         slots[i] = starSlotGO.GetComponent<PickupUI>();
      }
   }

   public void OnChangeStarTotal(int starTotal){
      for(int i = 0; i < NUM_INVENTORY_SLOTS; i++){
         PickupUI slot = slots[i];
         if(i < starTotal)
            slot.DisplayYellow();
         else
            slot.DisplayGrey();
      }
   }
}
```

5. With gameObject `player-SpaceGirl` selected in the **Hierarchy** panel, drag the gameObject `Panel-slot-grid` into **Player Inventory Display (Script)** variable **Slot grid**, in the **Inspector** panel.
6. With gameObject `player-SpaceGirl` selected in the **Hierarchy** panel, drag from the **Project** panel prefab `starUI` into **Player Inventory Display (Script)** variable **Star Slot Prefab**, in the **Inspector** panel.



Insert image 1362OT_02_29.png

The public array has been made private, and no longer needs to be populated through manual drag-and-drop. When you run the game it will play just the same as before, with the population of the array of images in our inventory grid panel has now been automated. Method `Awake()` creates new instances of the prefab (as many as defined by constant `NUM_INVENTORY_SLOTS`), and immediately childed them to `Panel-slot-grid`. Since we have a Grid Layout Group component, their placement is automatically neat and tidy in our panel.

Note: The scale property of the transform component of gameObjects is reset when a gameObject changes its parent (to maintain relative child-size-to-parent-size). So it is a good idea to always reset the local scale of gameObjects to (1,1,1) immediately after they have been childed to another gameObject. We do this in the FOR-loop to `starSlotGO` immedately following the `SetParent(…)` statement.

Note, we use method `Awake()` for creating the instances of the prefab in `PlayerInventoryDispay`, so that we know this will be executed before the `Start()` method in `PlayerInventoryModel` – since no `Start()` method is every executed until all `Awake()` methods for all gameObjects in the scene have been completed.

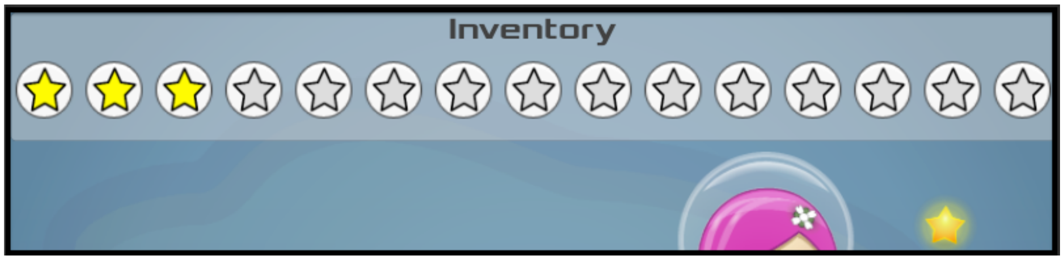
Automatically changing grid cell size based on number of 'slots' in inventory

Consider a situation where we wish to change the number of 'slots'. Another alternative to using scrollbars is to change the cell size in the **Grid Layout Group** component. We can automate this through code, so that the cellsize is changed to ensure that `NUM_INVENTORY_SLOTS` automate will fit along the width of our panel at the top of the canvas.

To implement the automated resizing of the **Grid Layout Group** cellsize for this recipe we need to do the following:

1. Add the following method `Start()` to C# Script `PlayerInventoryDisplay` in gameObject `player-SpaceGirl` with the following code:

```
void Start(){
   float panelWidth =
slotGrid.GetComponent<RectTransform>().rect.width;
   print ("slotGrid.GetComponent<RectTransform>().rect = "
+ slotGrid.GetComponent<RectTransform>().rect);

   GridLayoutGroup gridLayoutGroup =
slotGrid.GetComponent<GridLayoutGroup>();
   float xCellSize = panelWidth / NUM_INVENTORY_SLOTS;
   xCellSize -= gridLayoutGroup.spacing.x;
   gridLayoutGroup.cellSize = new Vector2(xCellSize,
xCellSize);
}
```



Insert image 1362OT_02_30.png

We write our code in method `Start()`, rather than adding to code in method `Awake()`, to ensure that the **RectTransform** of gameObject `Panel-slot-grid` has finished sizing (in this recipe it stretches based on the width of the **Game** panel). While we can't know the sequence that **Hierarchy** gameObjects are created when a scene begins, we can rely on the Unity behavior that every gameObject is sent the message `Awake()`, and only after all corresponding `Awake()` methods have finished executing are all objects then sent message `Start()`. So code in any `Start()` method can safely assume every gameObject has been initialized.

The screenshot shows the value of `NUM_INVENTORY_SLOTS` having been changed to 15, and the cellsize having been corresponding changed, so that all 15 now fit horizontally in our panel. Note the spacing between cells is subtracted from the calculated available with divided by the number of slots (`xCellSize -= gridLayoutGroup.spacing.x`) – since that spacing is needed between each item displayed as well.

Add some ‘help’ methods to the Rect Transform script class

If we wish to further change, say, the `RectTransform` properties using code, we can add Extension Methods, by created a file containing special static methods and using the special ‘this’ keyword. See the following code, which adds `SetWidth(…)`, `SetHeight(…)` and `SetSize(…)` methods to the `RectTransform` scripted component:

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

public static class RectTransformExtensions
{
   public static void SetSize(this RectTransform trans,
Vector2 newSize) {
      Vector2 oldSize = trans.rect.size;
      Vector2 deltaSize = newSize - oldSize;
      trans.offsetMin = trans.offsetMin - new
Vector2(deltaSize.x * trans.pivot.x, deltaSize.y *
trans.pivot.y);
      trans.offsetMax = trans.offsetMax + new
Vector2(deltaSize.x * (1f - trans.pivot.x), deltaSize.y *
(1f - trans.pivot.y));
   }

   public static void SetWidth(this RectTransform trans,
float newSize) {
      SetSize(trans, new Vector2(newSize,
trans.rect.size.y));
   }

   public static void SetHeight(this RectTransform trans,
float newSize) {
      SetSize(trans, new Vector2(trans.rect.size.x,
newSize));
   }
}
```

Unity C# allows us to add these extensions methods by declaring ‘static void’ methods whose first argument is in the form ‘this <ClassName> <var>’. The method can then be called as a built-in method defined in the original class.

All we would need to do is create a new C# script class file `RectTransformExtensions` in folder Scripts in the **Project** panel, containing the code above. In fact you can find a whole set of useful extra `RectTransform` methods (on which the above is an extract) created by OrbcreationBV and available online at: `http://orbcreation.com/orbcreation/page.orb?1099`.

Conclusion

In this chapter, we have introduced recipes demonstrating a range of C# data representations for inventory items, a range of Unity UI interface components to display the status and contents of player inventories at run-time.

Inventory UI needs good quality graphical assets for a high quality result, some sources of assets you might wish to explore include the following sites:

- The graphics for our SpaceGirl mini game are from the "Space Cute" art by Daniel Cook, he generously publishes lost of 2D art for game developers to use
  `http://www.lostgarden.com/`
  `http://www.lostgarden.com/search?q=planet+cute`
- Sethbyrd - lots of fun 2D graphics
  `http://www.sethbyrd.com/`
- Royalty-free art for 2D games:
  `http://www.gameart2d.com/freebies.html`