1

Core UI - Messages, Menus, Scores, and Timers

In this chapter, we will cover:

- Displaying a “Hello World” UI text message
- Displaying a digital clock
- Displaying a digital countdown timer
- Creating a message that fades away
- Display a perspective 3D Text message
- Displaying an image
- Creating UI Buttons to move between scenes
- Organizing images inside panels and changing panel depths via buttons
- Displaying the value of an interactive UI Slider
- Displaying a countdown timer graphically with a UI Slider
- Displaying a radar to indicate relative locations of objects
- Creating UIs with the Fungus open source dialogue system
- Setting custom mouse cursor images
- User interaction Input Field for text entry
- User interaction Toggles and radio buttons via Toggle Groups

Introduction

A key element contributing to the entertainment and enjoyment of most games is the quality of the visual experience, and an important part of this is the UI (User Interface). UI elements involve ways for the user to interact with the game (such as buttons, cursors, text boxes, and so on), as well as ways for the game to present up-to-date information to

the user (such as the time remaining, current health, score, lives left, or location of enemies). This chapter is filled with UI recipes to give you a range of examples and ideas of creating game UIs.

The big picture

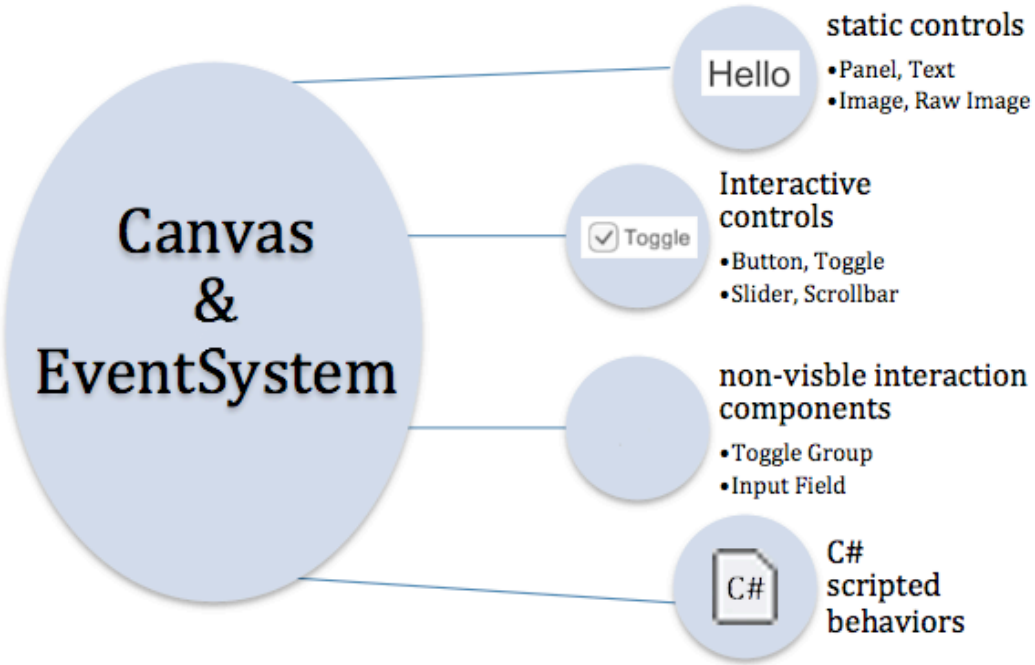
Every game is different, and so this chapter attempts to fulfill two key roles. The first aim is to provide step-by-step instructions on how to create a wide range of the Unity 5 UI elements and, where appropriate, associate them with game variables in code. The second aim is to provide a rich illustration of how UI elements can be used for a variety of purposes, so that you can get good ideas for how to make the Unity 5 UI set of controls deliver the particular visual experience and interactions for the game you are developing.

Basic UI elements can provide static images and text to just make the screen look more interesting. Using scripts we can change the content of these images and text objects, so that the players' numeric scores can be updated, or we show stickmen images to indicate how many lives the player has left, and so on. Other UI elements are interactive, allowing users to click buttons, choose options, and enter text, and so on. More sophisticated kinds of UI can involve collecting and calculating data about the game (such as percentage time remaining or enemy hit damage, or the positions and types of key gameObjects in the scene and their relationship to the location and orientation of the player) and then displaying these values in a natural, graphical way (such as progress bars or radar screens).

Core gameObjects, components and concepts relating to Unity UI development include:

- **Canvas** – every UI element is a child to a **Canvas**. There can be multiple **Canvas** gameObjects in a single scene. If no **Canvas** was already present then one will automatically be created when a new UI gameObject is created, with that UI object childed to the new **Canvas**.
- **EventSystem** – an **EventSystem** gameObject is required to manage the interaction events for UI controls. One will automatically be created with the first UI element.
- **Panel** – UI objects can be grouped together (logically and physically) with UI **Panels**. **Panels** can play several roles, including providing a gameObject parent in the **Hierarchy** for a related group of controls, they can provide a visual background image to graphically relate controls on screen, and they can also have scripted resize and drag interactions added if desired.
- **Visual UI** controls – the visible UI controls themselves including Button, Image, Text, Toggle, and so on.
- **Interaction UI** controls – these are non-visible components that are added to gameObjects, examples include Input Field and Toggle Group.

- The **Rect Transform** component – UI gameObjects can exist in a different space from that of the 2D and 3D scenes which cameras render, and therefore UI gameObjects all have the special **Rect Transform** component, which has some different properties to the scene gameObject Transform component (with its straightforward X/Y/Z position, rotation, and scale properties). Associated with **Rect Transforms** are pivot points (reference for scaling, resizing, and rotations) and anchor points. Read more about these core features below …
- **Sibling Depth** – the bottom to top display order (what appears on top of what) for UI element is determined initially by their sequence in the **Hierarchy**. At design-time, this can be manually set by dragging gameObjects into the desired sequence in the **Hierarchy**. At run-time, we can send messages to the **Rect Transforms** of gameObjects, to dynamically change their Hierarchy position (and therefore display order) as the game or user interaction demands. This is illustrated in recipe *Organizing images inside panels and changing panel depths via buttons*.

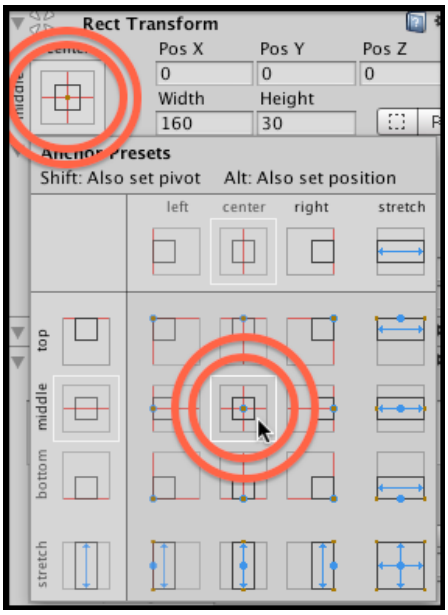


Insert image 1362OT_01_45.png

UI **Rect Transforms** represent a rectangular area, rather than a single point which is the case for scene gameObject **Transforms**. **Rect Transforms** describe how a UI element

should be positioned and sized relative to its parent. **Rect Transforms** have a width and height, which can be changed without affecting the local scale of the component. When the scale is changed for the **Rect Transforms** of a UI element, then this will also scale font sizes and borders on sliced images and so on. If all four anchors are at the same point, then resizing the Canvas will not stretch the **Rect Transform**, it will only affect the its position. In this case we'll see properties **Pos X** and **Pos Y**, and **Width** and **Height** for the rectangle. However, if the anchors are not all at the same point, then Canvas resizing will result in a stretching of the elements rectangle, and so instead of **Width** we'll see values for **Left** and **Right** – the position of the horizontal sides of the rectangle to the sides of the **Canvas**, where the **Width** will depend on the actual **Canvas** width (and the same for **Top/Bottom/Height**).

Unity provides a set of preset values for pivots and anchors, making the most common values very quick and easy to assign to an element's **Rect Transform**. A 3x3 grid allows quick choices about left/right/top/bottom/middle horizontal and vertical values, also the extra column on the right offers horizontal stretch presets, and the extra row at the bottom offers vertical stretch presets. Using the *SHIFT* and *ALT* keys sets the pivot and anchors when a preset is clicked.



Insert image 1362OT_01_03.png

The Unity manual provides a very good introduction to the **Rect Transform.** In additional Ray Wenderlich's two-part Unity UI web tutorial also presents a great

overview of the **Rect Transform**, pivots, and anchors. Both pages make great use of animated GIFs to illustrate the effect of different values for pivots and anchors:

- http://docs.unity3d.com/Manual/UIBasicLayout.html
- http://www.raywenderlich.com/78675/unity-new-gui-part-1

There are three Canvas render modes:

- **Screen Space – Overlay**: In this mode, UI elements are displayed without any reference to any camera (there doesn't even need to be any **Camera** in the scene). UI elements are presented in front of (overlaying) any camera display of scene contents.
- **Screen Space – Camera**: In this mode, the **Canvas** is treated as a flat plane in the frustum (viewing space) of a scene **Camera** – where this plane is always facing the camera. So any scene objects in front of this plane will be rendered in front of the UI elements on the canvas. The canvas is automatically resized if the screen size, resolution or camera settings are changed.
- **World Space**: In this mode, the **Canvas** acts as a flat plane in the frustum (viewing space) of a scene **Camera** – but the plane is not made to always face the **Camera**. How the **Canvas** appears is, just as with any other objects in the scene, relative to where (if anywhere) in the **Camera's** viewing frustum the **Canvas** plane is located and oriented.

In this chapter, we have focused on the **Screen Space – Overlay** mode. But all these recipes could equally be used with the other two modes.

Be creative! This chapter aims to act as a launching pad of ideas, techniques and reusable C# scripts for your own projects. Get to know the range of Unity UI elements, and try to work smart – often a UI element exists with most of the components you may need for something in your game, but you may need to adapt it somehow. An example of this can be seen in the recipe that makes a UI Slider non-interactive, using it instead to display a red-green progress bar for the status of a countdown timer – see this in recipe *Displaying a countdown timer graphically with a UI Slider*.

Displaying a "Hello World" UI text message

The traditional first problem to be solved with a new computing technology is often to display the message "Hello World". In this recipe you'll learn to create a simple UI Text object with this message, in large white text with a selected font, in the center of the screen.

Insert image 1362OT_01_01.png

Getting ready

For this recipe, we have prepared the font you need in a folder named `Fonts` in folder `1362_01_01`.

How to do it...

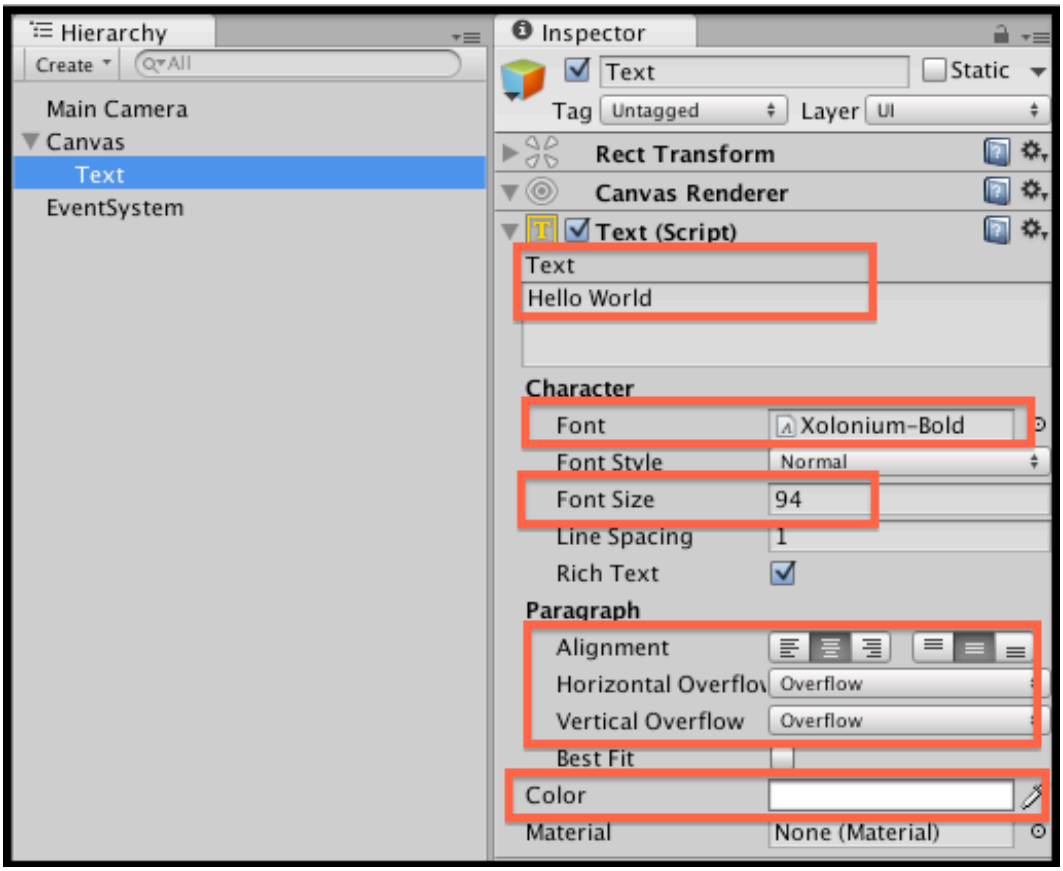
To display a **Hello World** text message, follow these steps:

1. Create a new Unity 2D project.
2. Import the provided folder `Fonts`.
3. In the **Hierarchy** panel add a **UI | Text** game object to the scene – choose menu: **GameObject | UI | Text**. Name this gameObject **Text-hello**.

> Alternatively use the **Create** menu immediately below the **Hierarchy** tab, choosing menu: **Create | UI | Text**.

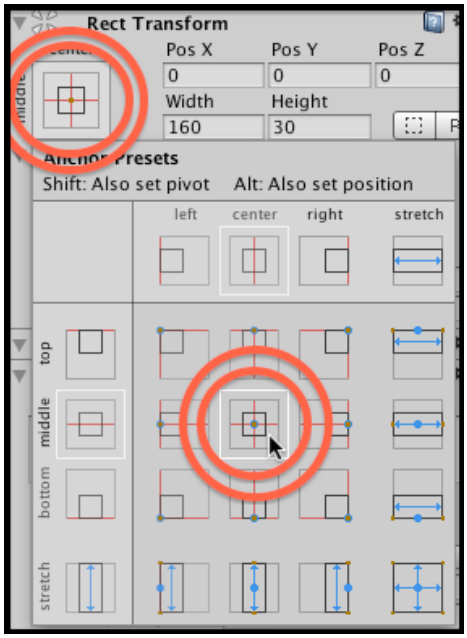
4. Ensure your new **Text-hello** gameObject is selected in the **Hierarchy** panel. Now in the **Inspector** ensure the follow properties are set:
    - Text = **Hello World**
    - Font = **Xolonium-Bold**
    - Font Size (large – this depends on your screen – try **50** or **100**)
    - Alignment = horizontal and vertical center
    - Horizontal and Vertical Overflow = **Overflow**

- Color = white



Insert image 1362OT_01_02.png

5. Now in the **Rect Transform** click the **Anchor Presets** square icon, which should result in several rows and columns of preset position squares appearing. Hold down *SHIFT* and *ALT* and click the center one (row 'middle' and column 'center').



Insert image 1362OT_01_03.png

6. Your **Hello World** text should now appear nicely centered in the Game panel.

How it works...

You have added a new **Text-hello** gameObject to a scene. A parent **Canvas** and UI **EventSystem** will also have been automatically created.

You set text content and presentation properties, and use **Rect Transform** anchor presets to ensure that whatever way the screen is resized the text will stay horizontally and vertically centered.

There's more...

Some details you don't want to miss:

Styling substrings with Rich Text

Each separate UI **Text** component can have its own color, size, boldness styling, and so on. However, if you wish to quickly add some highlighting style to a part of a string to be displayed to the user, the following are examples of some of the HTML-style markups that are available without the need to create separate UI **Text** objects:

- Embolden text with the 'b' markup: `I am <b>bold</b>`

- Italicize text with the 'i' markup: `I am <i>italic</i>`
- Set text color with Hex values or a color name: `I am <color=green>green text</color>, but I am <color=#FF0000>red</color>`

Learn more from the Unity online manual Rich Text page at: http://docs.unity3d.com/Manual/StyledText.html.

Displaying a digital clock

Whether it is the real-world time, or perhaps an in-game countdown clock, many games are enhanced by some form of clock or timer display. The most straightforward type of clock to display is a string composed of the integers for hours-minutes-seconds, which is what we'll create in this recipe.

Insert image 1362OT_01_04.png

Getting ready

For this recipe, we have prepared the font you need in a folder named `Fonts` in folder `1362_01_01`.

How to do it...

To create a digital clock, follow these steps:

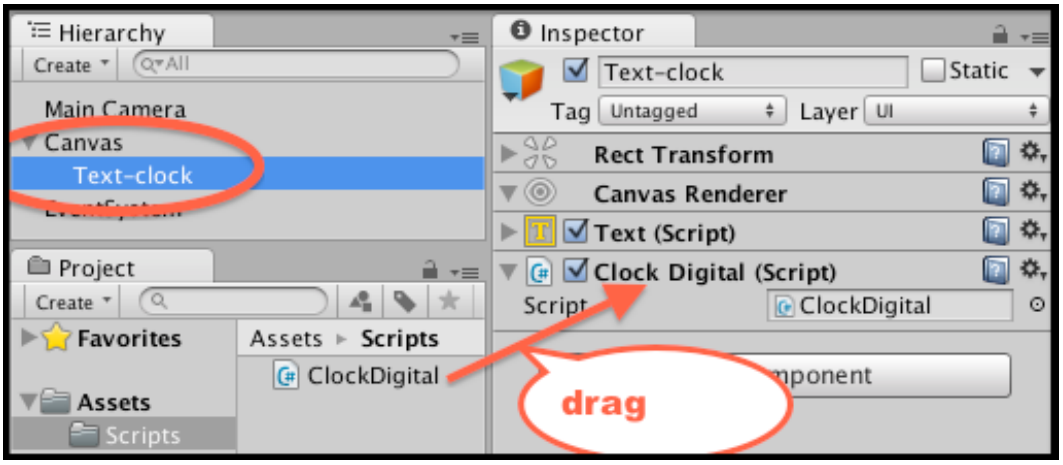
1. Create a new Unity 2D project.
2. Import the provided folder `Fonts`.
3. In the **Hierarchy** panel add a **UI | Text** game object to the scene named **Text-clock**.
4. Ensure gameObject **Text-clock** is selected in the **Hierarchy** panel. Now in the **Inspector** ensure the follow properties are set:
    - Text = **time goes here** (this placeholder text will be replaced by the time when the scene is running)
    - Font = **Xolonium Bold**
    - Font Size: **20**
    - Alignment = horizontal and vertical center

- Horizontal and Vertical Overflow = **Overflow**
- Color = white

5. Now in the **Rect Transform** click the **Anchor Presets** square icon, which should result in several rows and columns of preset position squares appearing. Hold down *SHIFT* and *ALT* and click row 'top' and column 'center'.
6. Create a folder named `Scripts` and create a C# script class `ClockDigital` in this new folder:

```
using UnityEngine;
using System.Collections;

using UnityEngine.UI;
using System;

public class ClockDigital : MonoBehaviour {
   private Text textClock;

   void Start (){
      textClock = GetComponent<Text>();
   }

   void Update (){
      DateTime time = DateTime.Now;
      string hour = LeadingZero( time.Hour );
      string minute = LeadingZero( time.Minute );
      string second = LeadingZero( time.Second );

      textClock.text = hour + ":" + minute + ":" +  second;
   }

   string LeadingZero (int n){
      return n.ToString().PadLeft(2, '0');
   }
}
```

7. With gameObject **Text-clock** selected in the **Hierarchy** panel drag onto it your script `ClockDigital`, to add an instance of this script class as a component to gameObject **Text-clock**.



Insert image 1362OT_01_05.png

8. When you run the scene you should now see a digital clock, showing hours, minutes and seconds, at the top center of the screen.

How it works...

You added a **Text** gameObject to a scene. You have added an instance of C# script class `ClockDigital` to that gameObject.

Notice that as well as the standard two C# packages (`UnityEngine` and `System.Collections`) that are written by default for every new script, you have added using statements for two more C# script packages `UnityEngine.UI` and `System`. The UI package is needed since our code uses UI `Text` object, and the `System` package is needed since it contains the `DateTime` class we need to access the clock on the computer where our game is running.

There is one variable, `textClock`, which will be a reference to the `Text` component, whose text content we wish to update each frame with the current time in hours, minutes, and seconds.

The `Start()` method (executed when the scene begins) sets variable `textClock` to find the `Text` component in the gameObject to which our scripted object has been added.

> NOTE: An alternative approach would be to make `textClock` a `public` variable. This would allow us to assign it via drag-and-drop in the **Inspector**.

Method `Update()` is executed every frame. The current time is stored in variable `time`, and strings are created by adding leading zeros to the number values for the hours, minutes and seconds properties of variable `time`.

This method finally updates the `text` property (that is, the letters and numbers the user sees) to be a string concatenating the hours, minutes, and seconds.

Method `LeadingZero(…)` takes as input an integer, and returns a string of this number with leading zeros added to the left if the value was less than 10.

There's more...

Some details you don't want to miss:

Unity tutorial for animating an analogue clock

Unity has published a nice tutorial on how to create 3D objects and animate them through C# script to display an analogue clock: https://unity3d.com/learn/tutorials/modules/beginner/scripting/simple-clock.

Displaying a digital countdown timer

This recipe shows how to display a digital countdown clock.

Insert image 1362OT_01_07.png

Getting ready

This recipe adapts the previous one, so make a copy of the project for the previous recipe, and work on this copy to follow this recipe.

For this recipe, we have prepared the script you need in a folder named `Scripts` in folder `1362_01_03`.

How to do it...

To create a digital countdown timer, follow these steps:

1. In the **Inspector** remove the scripted component `ClockDigital` from gameObject **Text-clock**.

2. Create a C# script class `DigitalCountdown` containing the following code, and add an instance as a scripted component to gameObject **Text-clock**:

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

public class DigitalCountdown : MonoBehaviour {
   private Text textClock;

   private float countdownTimerDuration
   private float countdownTimerStartTime;

   void Start (){
      textClock = GetComponent<Text>();
      CountdownTimerReset( 30 );
   }

   void Update (){
      // default - timer finished
      string timerMessage = "countdown has finished";
      int timeLeft = (int)CountdownTimerSecondsRemaining();

      if(timeLeft > 0)
         timerMessage = "Countdown seconds remaining = " +
LeadingZero( timeLeft );

      textClock.text = timerMessage;
   }

   private void CountdownTimerReset (float delayInSeconds){
      countdownTimerDuration = delayInSeconds;
      countdownTimerStartTime = Time.time;
   }

   private float CountdownTimerSecondsRemaining (){
      float elapsedSeconds = Time.time -
countdownTimerStartTime;
      float timeLeft = countdownTimerDuration -
elapsedSeconds;
      return timeLeft;
   }

   private string LeadingZero (int n){
      return n.ToString().PadLeft(2, '0');
```

```
        }
    }
```

3. When you run the scene, you should now see a digital clock counting down from 30. When the countdown reaches zero the message **countdown has finished** will be displayed.

How it works...

You added a **Text** gameObject to a scene. You have added an instance of C# script class `DigitalCountdown` to that gameObject.

There is one variable, `textClock`, which will be a reference to the `Text` component, whose text content we wish to update each frame with time remaining message (or timer complete message). Then a call is made to method `CountdownTimerReset(…)` passing an initial value of 30 seconds.

The `Start()` method (executed when the scene begins) sets variable `textClock` to find the `Text` component in the gameObject to which our scripted object has been added.

Method `Update()` is executed every frame. This method initially sets variable `timerMessage` to a message stating the timer has finished (the default message to display). But then the seconds remaining is tested to be greater than zero, and if so, then the message variable has its contents changed to display the integer (whole) number of seconds remaining in the countdown – retrieved from method `CountdownTimerSecondsRemaining()`. This method finally updates the `text` property (that is, the letters and numbers the user sees) to be a string with a message about the remaining seconds.

Method `CountdownTimerReset(…)` records the number of seconds provided, and the time the method was called.

Method `CountdownTimerSecondsRemaining()` returns an integer value of the number of seconds remaining.

Displaying a message that fades away

Sometimes we want a message to display just for a certain time, and then fade away and disappear.

Insert image 1362OT_01_20.png

Getting ready

This recipe adapts the first recipe *Displaying a "Hello World" text message*, so make a copy of that project to work on for this recipe.

For this recipe, we have prepared the script you need in a folder named `Scripts` in folder `1362_01_04`.

How to do it...

To display a text message that fades away, follow these steps:

1. Import the provided C# script class `CountdownTimer`.
2. Ensure gameObject **Text-hello** is selected in the **Hierarchy**. Then attach an instance of C# script class `CountdownTimer` as a component of this GameObject.
3. Create a C# script class `FadeAway` containing the following code, and add an instance as a scripted component to gameObject **Text-hello**:

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

public class FadeAway : MonoBehaviour {
   private CountdownTimer countdownTimer;
   private Text textUI;
   private int fadeDuration = 5;
   private bool fading = false;

   void Start (){
      textUI = GetComponent<Text>();
      countdownTimer = GetComponent<CountdownTimer>();

      StartFading(fadeDuration);
   }
```

```
    void Update () {
        if(fading){
            float alphaRemaining =
countdownTimer.GetProportionTimeRemaining();
            print (alphaRemaining);
            Color c = textUI.material.color;
            c.a = alphaRemaining;
            textUI.material.color = c;

            // stop fading when very small number
            if(alphaRemaining < 0.01)
                fading = false;
        }
    }

    public void StartFading (int timerTotal){
        countdownTimer.ResetTimer(timerTotal);
        fading = true;
    }
}
```

4. When you run the scene you should now see the message on the screen slowly fades away, disappearing after 5 seconds.

How it works...

An instance of the provided `CountdownTimer` script class was added as a component to gameObject **Text-hello**.

You have added to gameObject **Text-hello** an instance of the scripted class `FadeAway`. Method `Start()` caches references to the Text and CountdownTimer components in variables `countdownTimer` and `textUI`, and calls method `StartFading(…)` passing in the number 5, so the message will have faded invisible after 5 seconds.

Method `StartFading(…)` starts this timer scripted component to countdown to the given number of seconds, it also sets Boolean flag variable `fading` to true.

Method `Update()`, each frame, tests if variable `fading` is true. If it is true, then the alpha (transparency) component of the color of the **Text-hello** object is set to a value between 0.0 and 1.0, based on the proportion of time remaining in the `CountdownTimer`. Finally, if the proportion of timer remaining is less than a very small value (0.01) then variable `fading` is set to false (to save processing work since the text is now invisible).

Display a perspective 3D Text message

Unity provides an alternative way to display text in 3D via the Text Mesh component. While this is really most suitable for text-in-the-scene (such as billboards, road signs, and generally wording on the side of 3D objects that might be seen close up), it is quick to create and is another way of creating interesting menu or instructions scenes and the like.

In this recipe you'll learn to create scrolling 3D text, simulating the famous opening credits of the movie **Star Wars**.

Insert image 1362OT_01_27.png

Getting ready

For this recipe, we have prepared the fonts you need in a folder named `Fonts` and the text file you need in a folder named `Text`, in folder `1362_01_04`.

How to do it...

To display perspective 3D text, follow these steps:

1. Create a new Unity 3D project (this ensures we start off with a **Perspective** camera, suitable for the 3D effect we want to create).

> **Note.** If you need to mix 2D and 3D scenes in your project, you can always manually set any camera's **Camera Projection** property to **Perspective** or **Orthographic** via the **Inspector** panel.

2. In the **Hierarchy** panel select the **Main Camera**, and in the **Inspector** panel set its properties as follows: **Camera Clear Flags** to **solid color**, **Field of View** to **150**. Also set the **Background** color to black.
3. Import the provided folder `Fonts`.
4. In the **Hierarchy** panel add a **UI | Text** game object to the scene – choose menu: **GameObject | UI | Text**. Name this gameObject **Text-star-wars**. Set its Text Content to be Star Wars (each word on a new line), set its Font to **Xolonium Bold**, and its **Font Size** to 50. Use the anchor presets in the **Rect Transform** to position this UI **Text** object at the top center of the screen.
5. In the **Hierarchy** panel add a **3D Text** game object to the scene – choose menu: **GameObject | 3D Object | 3D Text**. Name this gameObject **Text-crawler**.
6. In the **Inspector** panel set the **Transform** properties for gameObject **Text-crawler** as follows: **Position** (0, -300, -20), **Rotation** (15, 0, 0).
7. In the **Inspector** panel set the **Text Mesh** properties for gameObject **Text-crawler** as follows:
    - **Text** (paste content of provided text file: `star_wars.txt`)
    - **Offset Z** = -20, **Line Spacing** = 0.8, **Anchor** = Middle center,
    - **Font Size** = 200, **Font** = SourceSansPro-BoldIt.
8. When the scene is run the Star Wars story text should now appear nicely squashed into 3D perspective on screen.

How it works...

You have simulated the opening screen of the movie Star Wars, with a flat UI **Text** object title at the top of the screen, and 3D **Text Mesh** with settings that appear to be disappearing into the horizon with 3D perspective 'squashing'.

There's more...

Some details you don't want to miss:

We have to make this text crawl like in the movie …

With a few lines of code, we can make this text scroll into the horizon just like in the movie. Add the following C# script class `ScrollZ` as a component to gameObject **Text-crawler**:

```
using UnityEngine;
using System.Collections;

public class ScrollZ : MonoBehaviour {
    public float scrollSpeed = 20;
```

```
    void Update () {
        Vector3 pos = transform.position;
        Vector3 localVectorUp =
transform.TransformDirection(0,1,0);
        pos += localVectorUp * scrollSpeed * Time.deltaTime;
        transform.position = pos;
    }
}
```

Each frame, via method `Update()`, the position of the 3D text object is moved in the direction of this gameObject's local up direction.

Where to learn more

Learn more about 3D Text and Text Meshes in the Unity online manual at: http://docs.unity3d.com/Manual/class-TextMesh.html.

Displaying an image

There are many cases where we wish to display an image on screen, including logos, maps, icons, splash graphics, and so on. In this recipe we will display an image at the top of the screen, and make it stretch to fit to whatever width the screen is resized.

Insert image 1362OT_01_06.png

Getting ready

For this recipe, we have prepared the image you need in a folder named `Images` in folder `1362_01_06`.

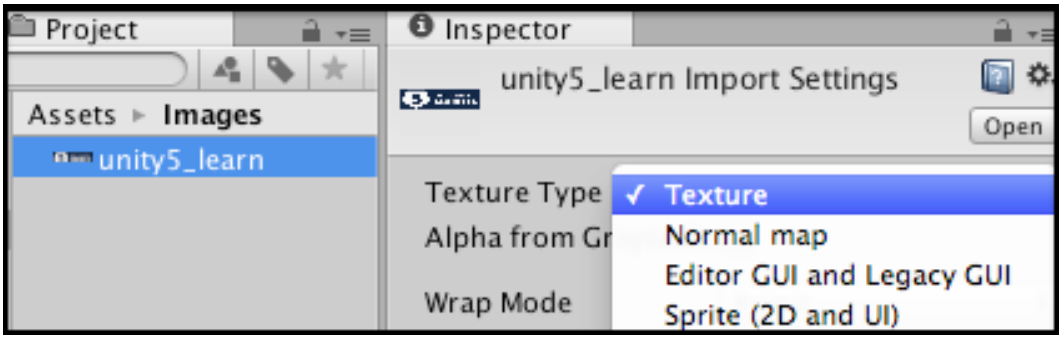
How to do it...

To display a stretched image, follow these steps:

1. Create a new Unity 3D project.

> NOTE: 3D projects will by default import images as **Texture**, and 2D projects will import images as **Sprite (2D and UI)**. Since we're going to use a **RawImage** UI component, we need our images to be imported as Textures.

2. Set the Game panel to size 400 x 300. Do this via menu: **Edit | Project Settings | Player**. Ensure **Resolution | Default is Full Screen** is unchecked, and width/height to 400 x 300. Then in the **Game** panel select **Stand Alone (400 x 300)**. This will allow us to test the stretching of our image to width 400 pixels.
3. Import the provided folder `Images`. In the **Inspector** ensure image `unity5_learn` has **Texture Type: Texture**. If it does not, then choose **Texture** from the dropdown list and click the button **Apply**.

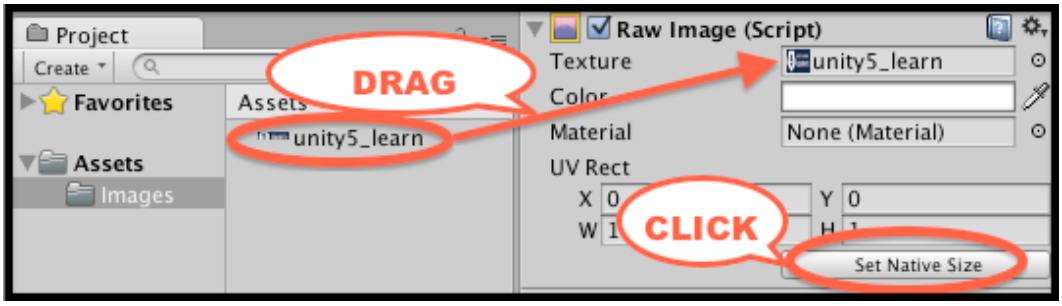


Insert image 1362OT_01_13.png

4. In the **Hierarchy** panel add a **UI | RawImage** gameObject to the scene named **RawImage-unity5**.

NOTE: If you wish to PREVENT distortion and stretching of an image, then use the UI **Sprite** gameObject instead, and ensure you check option **Preserve Aspect** in its **Image (Script)** component in the **Inspector**.

5. Ensure gameObject **RawImage-unity5** is selected in the **Hierarchy** panel. From your **Project** `Images` folder drag image `unity5_learn` into the **Raw Image (Script)** public property **Texture**. Click button **Set Native Size** to preview the image before it gets stretched…



Insert image 1362OT_01_14.png

6. Now in the **Rect Transform** click the **Anchor Presets** square icon, which should result in several rows and columns of preset position squares appearing. Hold down *SHIFT* and *ALT* and click row 'top' and column 'stretch'.
7. The image should now be positioned neatly at the top of the **Game** panel, and stretched to the full width of 400 pixels.

How it works...

You have ensured that an image has **Texture Type: Texture**. You added a **UI RawImage** control to the scene. The **RawImage** control has been made to display image file `unity5_learn`.

The image has been positioned at the top of the **Game** panel, and using anchor and pivot presets made to stretch to fill the whole width, which we set to 400 pixels via the **Player** settings.

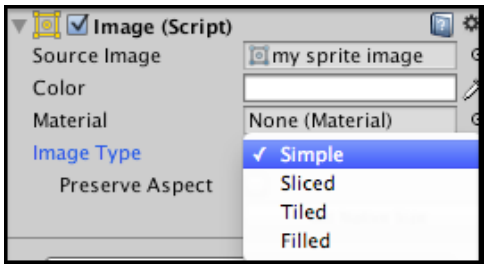
There's more...

Some details you don't want to miss:

Working with Sprites and UI Image components

If you simply wish to display non-animated images, then **Texture** images and the UI **RawImage** controls are the way to go. However, if you want more options about how an image is displayed (such as tiling, and animation), then the UI Sprite control should be used instead. This control needs image files to be imported as type Sprite (2D and UI).

Once an image file has been dragged into the UI Image control's Sprite property, additional properties will be available, such Image Type and an option to preserve aspect ratio and so on.



Insert image 1362OT_01_15.png

See also

An example of tiling a sprite image can be found in recipe *Revealing icons for multiple object pickups by changing the size of a tiled image* in *Chapter 3*, *Inventory GUIs*.

Creating UI Buttons to move between scenes

As well as the scenes where the player plays the game, most games will have menu screens, which display to the user messages about instructions, high scores, the level they have reached so far, and so on. Unity provides UI **Buttons** to make it easy to offer users a simple way to indicate their choice of action on such screens.

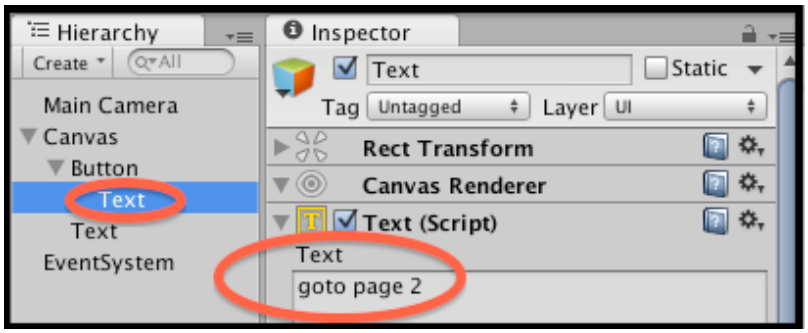
In this recipe, we create a very simple game consisting of two screens, each with a button to load the other one.

Insert image 1362OT_01_29.png

How to do it...

To create a button-navigable multi-scene game, follow these steps:

1. Create a new Unity 2D project.
2. Save the current (empty) scene naming it **page1**.
3. Add a UI **Text** object, positioned at the top center of the scene, containing text **Main Menu / (page 1)** in a large font size.
4. Add a UI **Button** to the scene, positioned in the middle center of the screen. In the **Hierarchy** panel click the show children triangle to display the UI **Text** child of this button gameObject. Select the **Text** button-child gameObject and in the **Inspector** for the **Text** property of the **Text (Script)** component, enter the button text **goto page 2**.



Insert image 1362OT_01_30.png

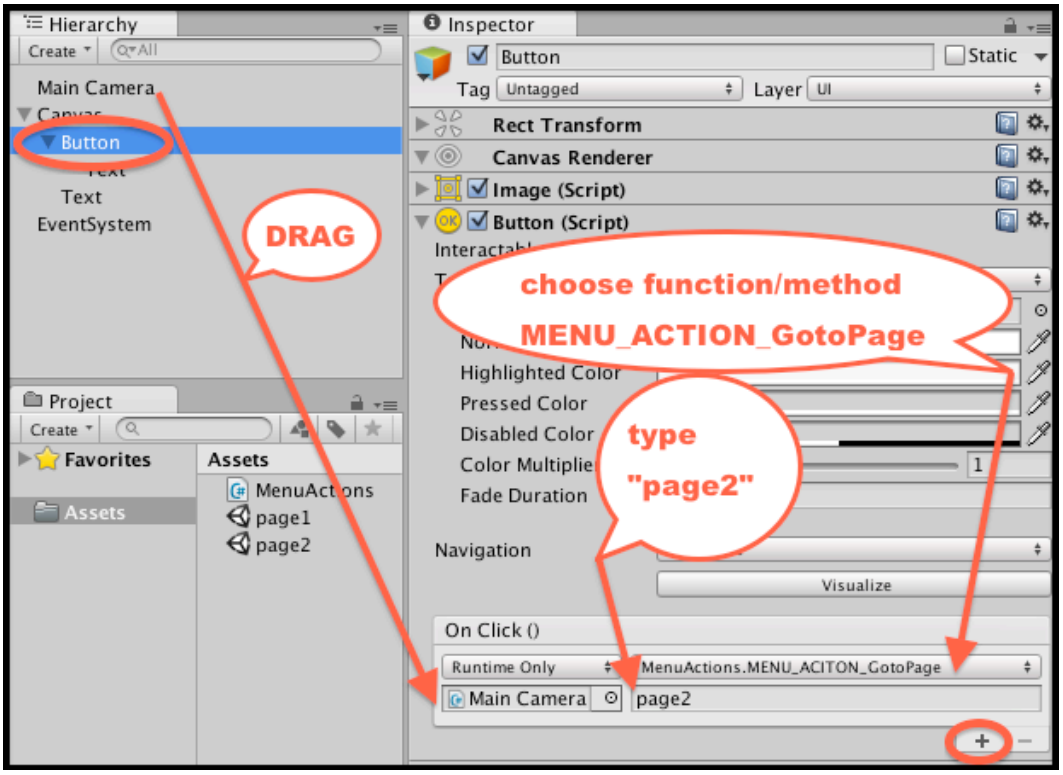
5. Add the current scene to the build, choosing menu: **File | Build Settings…** and then clicking the **Add Current** button so that scene **page1** becomes the first scene in the list of **Scenes in the Build**.

> NOTE: We cannot tell Unity to load a scene that has not been added to the list of scenes in the build. We use the code `Application.LoadLevel(…)` to tell Unity to load the scene name (or numeric index) provided.

6. Create a C# script class `MenuActions` containing the following code, and add an instance as a scripted component to the **Main Camera**:

```
using UnityEngine;
using System.Collections;

public class MenuActions : MonoBehaviour {
    public void MENU_ACTION_GotoPage(string sceneName){
        Application.LoadLevel(sceneName);
    }
}
```

7. Ensure the **Button** is selected in the **Hierarchy**, and click the plus sign "+" button at the bottom of the **Button (Script)** component in the Inspector to create a new **OnClick** event handler for this button.
8. Drag the **Main Camera** from the **Hierarchy** over the **Object** slot – immediately below the menu saying **Runtime Only** – this means that when the **Button** receives an **OnClick** event we can call a public method from a scripted object inside the **Main Camera**.
9. Now select method **MENU_ACTION_GotoPage() from** the MenuActions dropdown list (initially showing **No Function**). Type **page2** (the name of the scene we want to be loaded when this button is clicked) in the text box below the method dropdown menu. This string **page2** will be passed to the method when the button receives an **OnClick** event message.



Insert image 1362OT_01_31.png

10. Save the current scene, and then create a new empty scene, and save this new scene as **page2**.
11. Follow similar steps for this scene: Add a UI Text gameObject displaying text **Instructions / (page 2)** in a large font size. Add a UI Button showing text **goto page 1.**
12. Add the current scene to the build (so now both **page1** and **page2** should be listed in the build).
13. Add an instance as a script class `MenuActions` to the **Main Camera**.
14. Select the **Button** in the **Hierarchy**, and add an **On Click** event handler, which will pass method **MENU_ACTION_GotoPage()** the string **page2** (the name of the scene we want to be loaded when this button is clicked).
15. Save the scene.
16. When you run scene **page1** you should be presented with your **Main Menu** text, and the button, which when clicked, makes the game load scene **page2**. On scene **page2** you have a button to take you back to **page1**.

How it works...

You have created 2 scenes, and added both to the game build. Each scene has a button, which when clicked (when game is playing) makes Unity load the (named) other scene. This is made possible because when each button is clicked it runs method `MENU_ACTION_GotoPage(…)` from the scripted `MenuActions` component inside the **Main Camera**. This method inputs a text string of the name of the scene to be loaded, so the button in scene **page1** gives the string name of **page2** as the scene to be loaded, and vice versa.

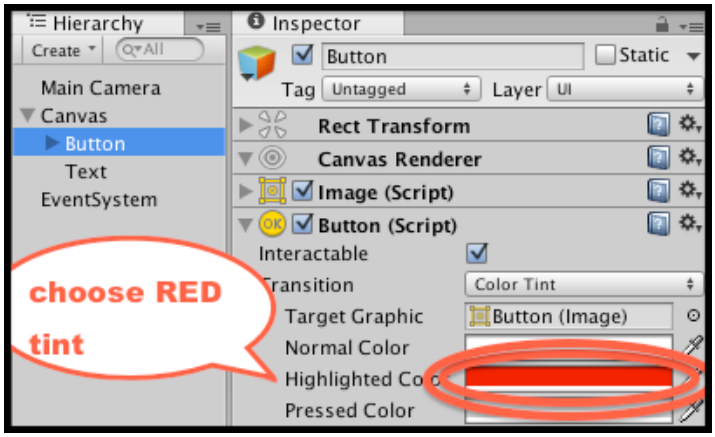
When a UI **Button** is added to the Hierarchy, automatically a child UI **Text** object is also created and the content of the **Text** property of this UI **Text** child is the text the user sees on the button.

There's more...

Some details you don't want to miss:

Visual animation for button mouse over

There are several ways we can visually inform the user that the button is interactive when they move their mouse cursor over it. The simplest is to add a color tint that will appear when the mouse is over the button – this is the default Transition. With the **Button** selected in the **Hierarchy**, choose a tint color (for example, red) for the **Highlighted Color** property of the **Button (Script)** component in the **Inspector**.



Insert image 1362OT_01_33.png

Another form of visual Transition to inform the user of an active button is Sprite Swap. In this case, properties for different images for **Targeted / Highlighted / Pressed /**

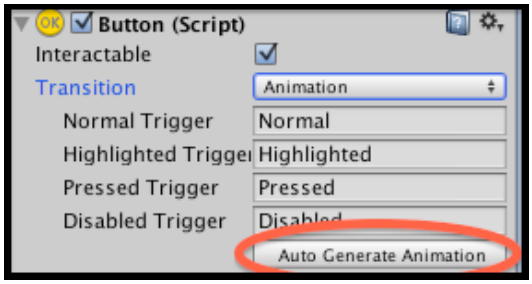
Disabled are available in the **Inspector**. The default **Targeted Graphic** is the built-in Unity **Button (image)** – this is the grey rounded rectangle default when buttons gameObjects are created. Dragging in a very different looking image for the **Highlighted Sprite** is an effective alternative to setting a color hint. We have provided an image `rainbow.png` with the project for this recipe that can be used for the **Button** mouse over **Highlighted Sprite**.

Insert image 1362OT_01_34.png

Animating button properties on mouse over

Finally, animations can be created for dynamically highlighting a button to the user, for example a button might get larger when the mouse is over it, and then shrink back to its original size when the mouse pointer is moved away. These effects are achieved by choosing the **Animation** option for the **Transition** property, and creating an animation controller with triggers for **Normal, Highlighted, Pressed and Disabled**. To animate a button to get larger when the mouse is over it (the Highlighted state) do the following:

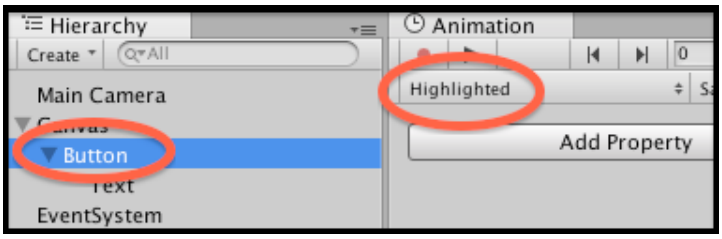
1. Create a new Unity 2D project.
2. Create a button.
3. In the **Inspector Button (Script)** component set the **Transition** property to **Animation**.
4. Click the **Auto Generate Animation** button (just below the **Disabled Trigger** property) for component **Button (Script)**.



Insert image 1362OT_01_41.png

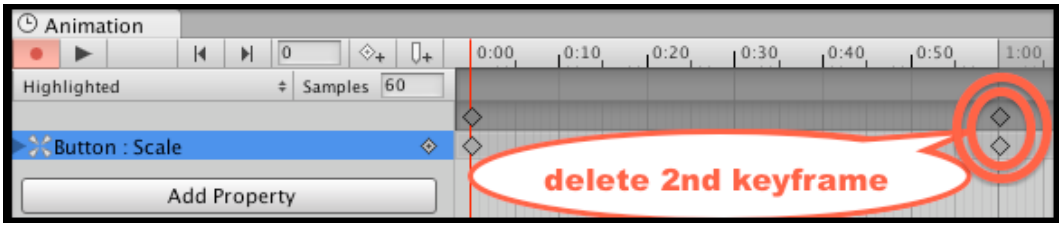
5. Save the new controller naming it **button-animation-controller**.

6. Ensure the **Button** gameObject is selected in the **Hierarchy**, and then in the **Animation** panel select the **Highlighted** clip from the drop down menu.



Insert image 1362OT_01_42.png

7. In the **Animation** panel click the red **record** circle button, and then click the **Add Property** button, choosing to record changes to the **Rect Transform | Scale** property.
8. Two keyframes will have been created, delete the second one at time 1:00 (since we don't want a 'bouncing' button).



Insert image 1362OT_01_43.png

9. Select the first keyframe at time 0:00 (the only one now!), and then in the **Inspector** set the X and Y scale properties of the **Rect Transform** component to (1.2, 1.2).
10. Finally click the red **record** circle button a second time to end recording of animation changes.
11. Save and run your scene and you should see the button smoothly animate to get larger when the mouse is over it, and then smoothly return to its original size when the mouse is moved away.

The following web pages offer video and web-based tutorials on UI animations:

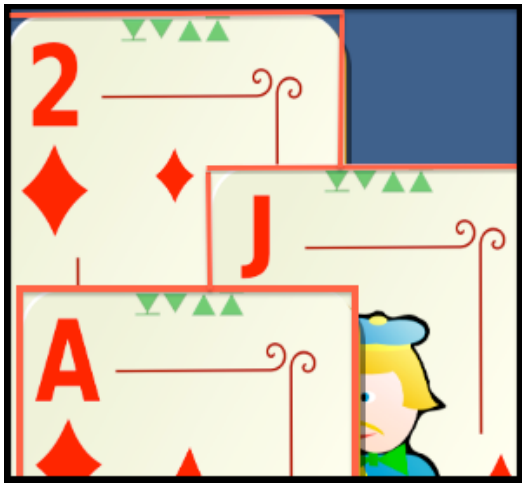
- The Unity Button transitions tutorial:
  http://unity3d.com/learn/tutorials/modules/beginner/ui/ui-transitions
- Ray Wenderlich's tutorial (part 2) including button animations:

http://www.raywenderlich.com/79031/unity-new-gui-tutorial-part-2

Organizing images inside panels and changing panel depths via buttons

UI **Panels** are provided by Unity to allow UI controls to be grouped and moved together, and also to visually group elements with an **Image** background (if desired). The **sibling depth** is what determines which UI elements appear above or below others. We can see the sibling depth explicitly in the **Hierarchy**, since the top-to-bottom sequence of UI gameObjects in the **Hierarchy** sets sibling depth. So the first item has depth 1, the second depth 2, and so on. UI gameObjects with larger sibling depths (further down the **Hierarchy**) appear above UI gameObjects with lower sibling depths.

In this recipe we'll create 3 UI panels, each showing a different playing card image. We'll also add four triangle arrangement buttons to change the display order (move to bottom, move to top, move up one, and move down one).



Insert image 1362OT_01_28.png

Getting ready

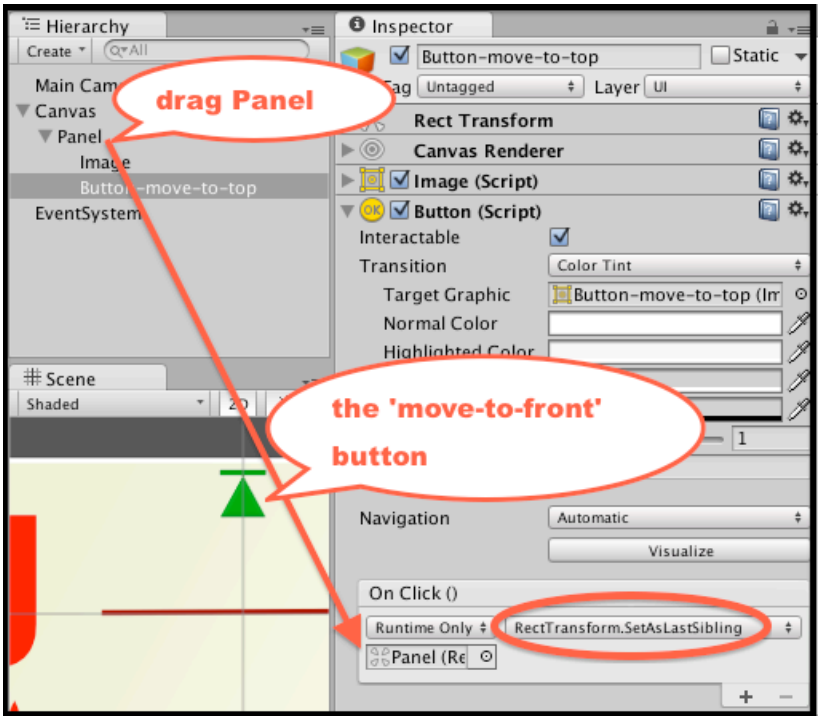
For this recipe, we have prepared the images you need in a folder named `Images` in folder `1362_01_08`.

How to do it...

To create UI **Panels** whose layering can be changed by the user clicking buttons, follow these steps:

1. Create a new Unity 2D project.
2. Create a new UI **Panel**, named **Panel-jack-diamonds**, positioned in the middle center of the screen, and sized 200 wide by 300 high. Uncheck the Image (Script) component for this panel (since we don't want to see the default semi-transparent rectangular grey background image of a panel).
3. Create a new UI **Image** and child this image to **Panel-jack-diamonds**.
4. Position image **Panel-jack-diamonds** center-middle, and size it to 200 x 300. Drag the **Jack-of-diamonds** playing card image into the **Source Image** property for the **Image (Script)** component in the **Inspector**.
5. Create a UI **Button**, named **Button-move-to-front**, child this button to **Panel-jack-diamonds**. Delete the **Text** child gameObject of this button (since we'll use an icon to indicate what this button does).
6. Size button **Button-move-to-front** to 16x16, and position it top-center, so it can be seen at the top of the playing card. Drag the `icon_move_to_front` arrangement triangle icon image into the **Source Image** property for the **Image (Script)** component in the **Inspector**.
7. Ensure button **Button-move-to-front** is selected in the **Hierarchy**. Then click the plus sign "+" button at the bottom of the **Button (Script)** component in the **Inspector** to create a new **OnClick** event handler for this button.
8. Drag the **Panel-jack-diamonds** from the **Hierarchy** over the **Object** slot (immediately below the menu saying **Runtime Only**).
9. Now select method **RectTransform.SetAsLastSibling** from the dropdown function list (initially showing **No Function**).

> This means that when the **Button** receives an **OnClick** event, the **RectTransform** of the **Panel** will be sent the message **SetAsLastSibling** – this will move the **Panel** to the bottom of the gameObjects in the **Canvas**, and therefore make this Panel in front of all other gameObjects in the **Canvas**.



Insert image 1362OT_01_14.png

10. Repeat from Step 2, creating a second **Panel** with a move-to-front button, name this second Panel **Panel-2-diamonds**, then move, and position it slightly to the right of **Panel-jack-diamonds**, allowing both move-to-front buttons to be seen.
11. Save your scene and run the game. You should be able to click the move-to-front button on either card to move that card's panel to the front. If you run the game with the Game panel not maximized, you'll actually see the panels changing order in the list of children of the **Canvas** in the **Hierarchy**.

How it works...

You've created 2 UI **Panels**, each panel containing an image of a playing card and a button whose action will make its parent panel move to the front. The button's action illustrates how the **OnClick** function does not have to be the calling of a public method of a scripted component of an object, but it can be sending a message to one of the components of the targeted gameObject – in this instance we send message **SetAsLastSibling** to the **RectTransform** of the Panel in which the Button is located.

There's more...

Some details you don't want to miss:

Moving up or down by just position, using scripted methods

While the Rect Transform offers the useful **SetAsLastSibling** (move to front) and **SetAsFirstSibling** (move to back), and even **SetSiblingIndex** (if we knew exactly what position in the sequence to type in), there isn't a built-in way to make an element move up or down just a single position in the sequence of gameObjects in the Hierarchy. However, we can write two straightforward methods in C# to do this, and we can add buttons to call these methods, providing full control of the top-to-bottom arrangement of UI controls on screen. To implement 4 buttons (move-to-front / move-to-back / up one / down one), do the following:

1. Create a C# script class `ArrangeActions` containing the following code, and add instance as a scripted components to each of your **Panels**:

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

public class ArrangeActions : MonoBehaviour {
   private RectTransform panelRectTransform;

   void Start(){
      panelRectTransform = GetComponent<RectTransform>();
   }

   public void MoveDownOne(){
      print ("(before change) " + gameObject.name +  "
sibling index = " + panelRectTransform.GetSiblingIndex());

      int currentSiblingIndex =
panelRectTransform.GetSiblingIndex();
      panelRectTransform.SetSiblingIndex(
currentSiblingIndex - 1 );

      print ("(after change) " + gameObject.name +  "
sibling index = " + panelRectTransform.GetSiblingIndex());
   }

   public void MoveUpOne(){
      print ("(before change) " + gameObject.name +  "
sibling index = " + panelRectTransform.GetSiblingIndex());
```

```
        int currentSiblingIndex =
panelRectTransform.GetSiblingIndex();
        panelRectTransform.SetSiblingIndex(
currentSiblingIndex + 1 );

        print ("(after change) " + gameObject.name +  "
sibling index = " + panelRectTransform.GetSiblingIndex());
    }
}
```

2. Add a second button to each card panel, this time using arrangement triangle icon image `icon_move_to_front`, and set the **OnClick** event function for these buttons to **SetAsFirstSibling**.
3. Add 2 further buttons to each card panel, with the up and down triangle icons images `icon_down_one` and `icon_up_one`. Set the **OnClick** event handler function for the down-one buttons to call method `MoveDownOne()`, and set the functions for the up-one buttons to call method `MoveUpOne()`.
4. Copy one of the panels to create a third card (this time showing the Ace of diamonds). Arrange the 3 cards so you can see all 4 buttons for at least 2 of the cards, even when those cards are at the bottom (see screenshot).
5. Save the scene and run your game. You should now have full control of the layering of the three card panels.

Displaying the value of an interactive UI Slider

This recipe illustrates how to create an interactive UI **Slider**, and execute a C# method each time the user changes the slider value.

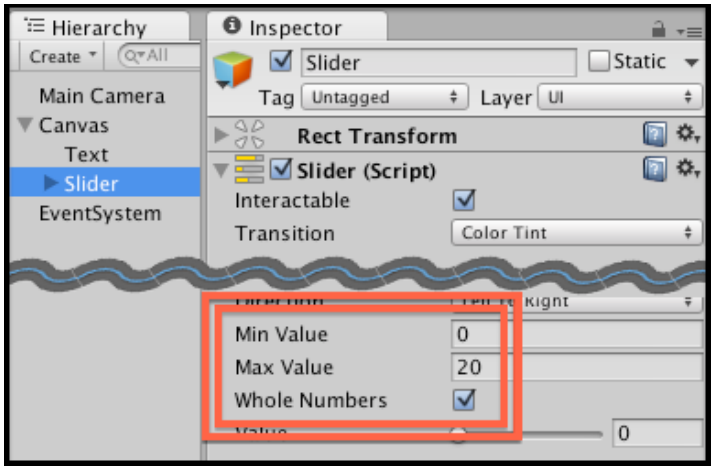
Insert image 1362OT_01_08.png

How to do it...

To create a UI Slider and displays its value on screen, follow these steps:

1. Create a new 2D project.
2. Add a UI Text gameObject to the scene, with Font size 30 and placeholder text such as "**slider value here**" (this text will be replaced with the slider value when the scene starts).

3. In the **Hierarchy** panel add a **UI | Slider** game object to the scene – choose menu: **GameObject | UI | Slider**.
4. In the **Inspector** modify settings for the **Rect Transform** to position the slider at the top middle of the screen, and the text just below it.
5. In the **Inspector** set the **Min Value** of the slider to 0, the **Max Value** to 20, and tick the **Whole Numbers** checkbox.



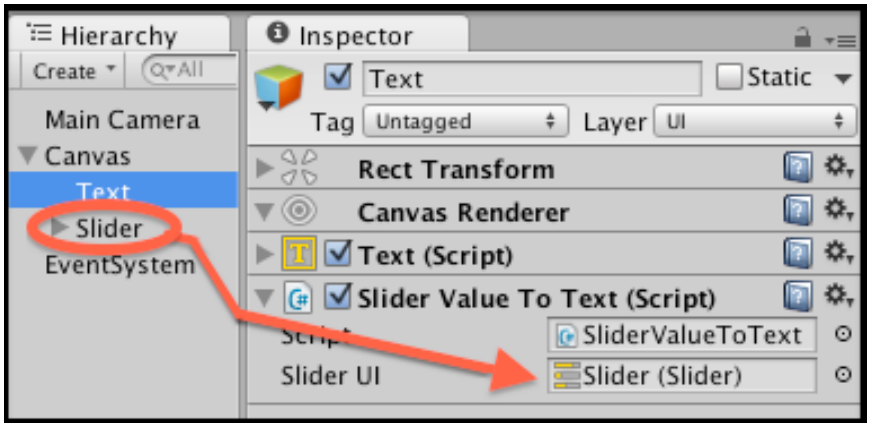
Insert image 1362OT_01_09.png

6. Create a C# script class `SliderValueToText` containing the following code, and add an instance as a scripted component to gameObject **Text**:

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

public class SliderValueToText : MonoBehaviour {
   public Slider sliderUI;
   private Text textSliderValue;

   void Start (){
      textSliderValue = GetComponent<Text>();
      ShowSliderValue();
   }

   public void ShowSliderValue () {
      string sliderMessage = "Slider value = " +
sliderUI.value;
      textSliderValue.text = sliderMessage;
```

```
        }
    }
```

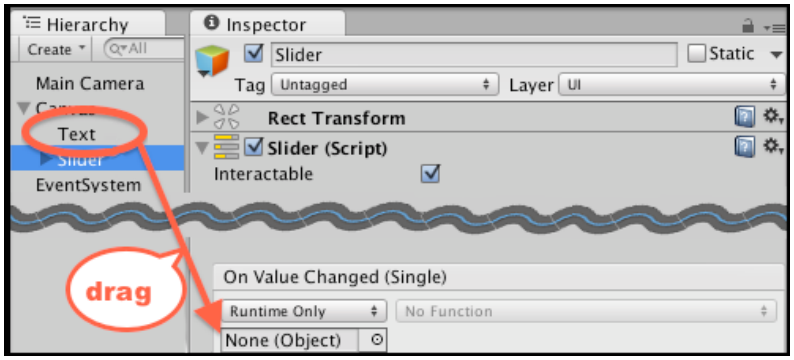
7. Ensure gameObject **Text** is selected in the **Hierarchy**. Then, in the **Inspector**, drag gameObject **Slider** into the public **Slider UI** variable slot for scripted component `Slider Value To Text (Script)`.



Insert image 1362OT_01_12.png

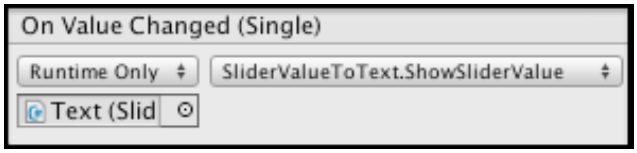
8. Ensure gameObject **Slider** is selected in the **Hierarchy**. Then, in the **Inspector**, drag gameObject **Text** into the public **None (Object)** slot for scripted component **Slider (Script)**, in the section for **On Value Changed (Single)**.

> You have now told Unity to which object a message should be sent each time the slider is changed.



Insert image 1362OT_01_11.png

9. From the dropdown menu now select `SliderValueToText` and method `ShowSliderValue()`. This means each time the slider is updated, method `ShowSliderValue()` in the scripted object in gameObject **Text** will be executed.



Insert image 1362OT_01_10.png

10. When you run the scene you should now see a slider, and below it a text message in the form `Slider value = <n>`.
11. Each time the slider is moved the text value show be (almost) instantly updated. The values should range from 0 (slider leftmost) to 20 (slider rightmost).

> NOTE: The update of the text value on screen probably won't be instantaneous, as in happening the same frame as the slider value is moved, since there is some computation involved in the slider deciding an On Value Changed event message needing to be triggered, and then looking up any methods of objects that are registered as event handlers for such an event. Then the statements in the object's method need to be executed in sequence. However, this should all happen within a few millisconds, and so be sufficieintly fast to offer the user a satisifyingly responsive UI for interface actions like changing moving this slider.

How it works...

You have added to gameObject **Text** a scripted instance of class `SliderValueToText`.

Method `Start()`, which is executed when the scene first runs, sets variable to be a reference to the **Text** component inside the **Slider**. Next method `ShowSliderValue()` is called, so that the display is correct when the scene begins (the initial slider value is displayed).

This contains method `ShowSliderValue()`, which gets the value of the slider and updates the text displayed to be a message in the form `Slider value = <n>`.

You created a UI Slider gameObject, and set it to be whole numbers in the range 0 to 20.

You added to the UI Slider gameObject's list of On Value Changed event listeners method `ShowSliderValue()`of scripted component `SliderValueToText`. So each time

the slider value changes, it sends a message to call method `ShowSliderValue()`, and so the new value is updated on screen.

Displaying a countdown timer graphically with a UI Slider

There are many cases where we wish to inform the player of the proportion remaining or complete of some value at a point in time, for example a loading progress bar, the time or health remaining compared to the starting maximum, how much the player has filled up their water bottle from the fountain of youth, and so on. In this recipe, we illustrate how to remove the interactive 'handle' of a UI Slider, and change the size and color of its components to provide us with an easy to use, general purpose progress/proportion bar. In this recipe, we use our modified slider to graphically present to the user how much time remains for a countdown timer.

Insert image 1362OT_01_18.png

Getting ready

This recipe adapts the previous one, so make a copy of the project for the previous recipe, and work on this copy to follow this recipe.

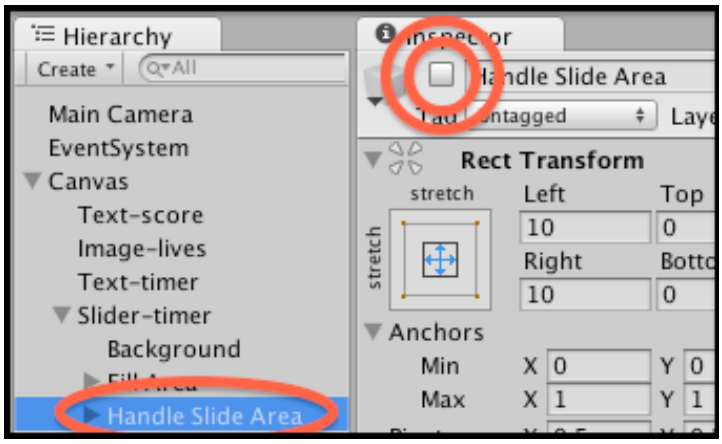
For this recipe, we have prepared the script and images you need in folders named `Scripts` and `Images` folder in `1362_01_10`.

How to do it...

To create a digital countdown timer with a graphical display, follow these steps:

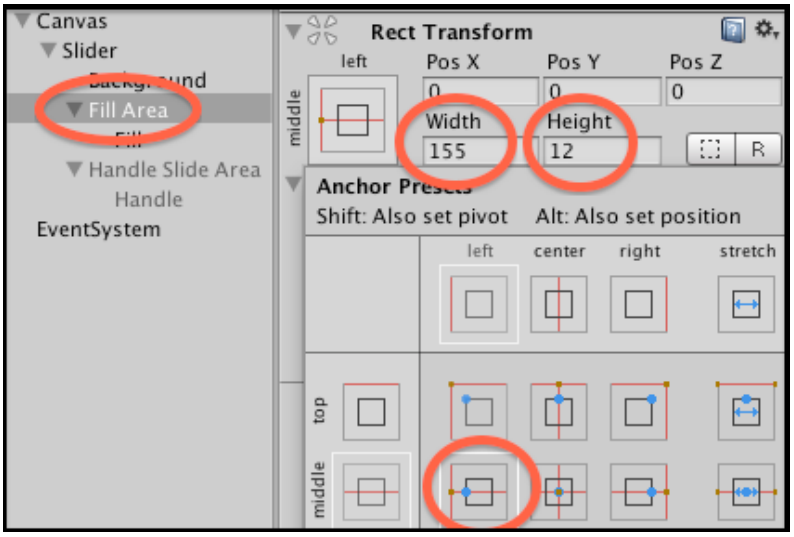
1. Delete gameObject **Text**.

2. Import script `CountdownTimer` and images `red_square` and `green_square` to this project.
3. Ensure gameObject **Slider** is selected in the **Hierarchy**.
4. Deactivate the **Handle Slide Area** child gameObject (by unchecking it).
    - You'll see the 'drag circle' disappear in the **Game** panel (the user will not be dragging the slider, since we want this slider to be display-only).



Insert image 1362OT_01_17.png

5. Select the **Background** child:
    - Drag image `red_square` into the **Source Image** property of the **Image (Script)** component in the **Inspector**.
6. Select the **Fill** child:
    - Drag image `green_square` into the **Source Image** property of the **Image (Script)** component in the **Inspector**.
7. Select the **Fill Area** child:
    - In the **Rect Transform** component use the **Anchors** preset position of **left-middle**.
    - Set **Width** to 155 and **Height** to 12.



Insert image 1362OT_01_19.png

8. Ensure gameObject **Slider** is selected in the **Hierarchy**. Then attach an instance of C# script class `CountdownTimer` as a component of this gameObject.
9. Create a C# script class `SliderTimerDisplay` containing the following code, and add an instance as a scripted component to gameObject **Text**:

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

public class SliderTimerDisplay : MonoBehaviour {
   private CountdownTimer countdownTimer;
   private Slider sliderUI;
   private int startSeconds = 30;

   void Start (){
      SetupSlider();
      SetupTimer();
   }

   void Update () {
      sliderUI.value =
countdownTimer.GetProportionTimeRemaining();
      print (countdownTimer.GetProportionTimeRemaining());
   }

   private void SetupSlider (){
```

```
            sliderUI = GetComponent<Slider>();
            sliderUI.minValue = 0;
            sliderUI.maxValue = 1;
            sliderUI.wholeNumbers = false;
        }

        private void SetupTimer (){
            countdownTimer = GetComponent<CountdownTimer>();
            countdownTimer.ResetTimer(startSeconds);
        }
    }
```

10. Run your game, you should see the slider move with each second, revealing more and more of the red background to indicate the time remaining.

How it works...

You hid the **Handle Slide Area** child so **Slider** is for display only, and cannot be interacted with by the user. The **Background** color of **Slider** was set to red, so that as the counter goes down, more and more red is revealed – warning the user that time is running out. The **Fill** of the **Slider** was set to green, so that the proportion remaining is displayed in green (more green, larger the value of the slider/timer).

An instance of the provided `CountdownTimer` script class was added as a component to the Slider. Method `ResetTimer(…)` records the number of seconds provided, and the time the method was called. Method `GetProportionRemaining()` returns a value from 0.0 to 1.0 representing the proportion of the seconds reaming (1.0 being all seconds, 0.5 half the seconds, 0.0 no seconds left).

You have added to gameObject **Slider** an instance of the scripted class `SliderTimerDisplay`. Method `Start()` calls methods `SetupSlider()` and `SetupTimer()`.

Method `SetupSlider()` sets variable `sliderUI` to be a reference to the **Slider** component, and sets up this slider to map to float (decimal) values between 0.0 and 1.0.

Method `SetupTimer()` sets variable `countdownTimer` to be a reference to the **CountdownTimer** component, and starts this timer scripted component to countdown from 30 seconds.

Each frame method `Update()` sets the slider value to the float returned by calling method `GetProportionRemaining()` from the running timer.

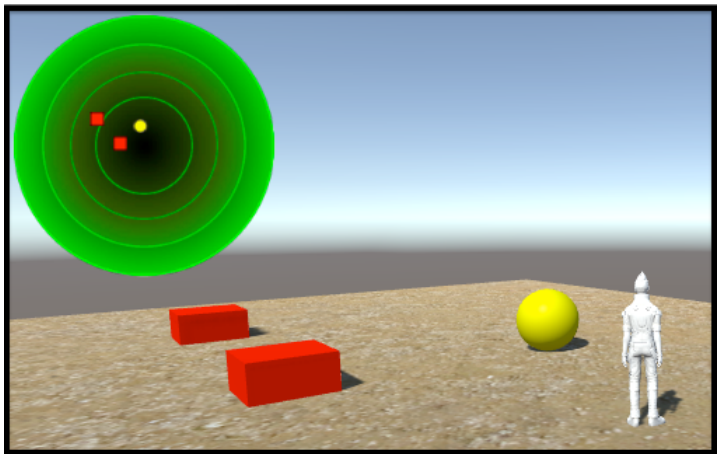
> **NOTE: Try to work with floats between 0.0 and 1.0 whenever possible**

Integers could have been used, setting the Slider min to 0 and max to 30 (for 30 seconds). However, changing the total number of seconds would then also require the Slider settings to be changed. In most cases working with a float proportion between 0.0 and 1.0 is the more general-purpose and reusable approach to adopt.

Displaying a radar to indicate relative locations of objects

A radar displays the locations of other objects relative to the player, usually based on a circular display, where the center represents the player, and each graphical 'blip' indicates how far away, and what relative direction objects are to the player. Sophisticated radar displays will display different categories of objects with different colored or shaped 'blip' icons.

In the screenshot we can see 2 red square 'blips', indicating the relative position of the 2 red cube gameObjects tagged 'Cube' near the player, and a yellow circle 'blip' indicating the relative position of the yellow sphere gameObject tagged 'Sphere'. The green circle radar background image gives the impression of an aircraft control tower radar or something similar.



Insert image 1362OT_01_21.png

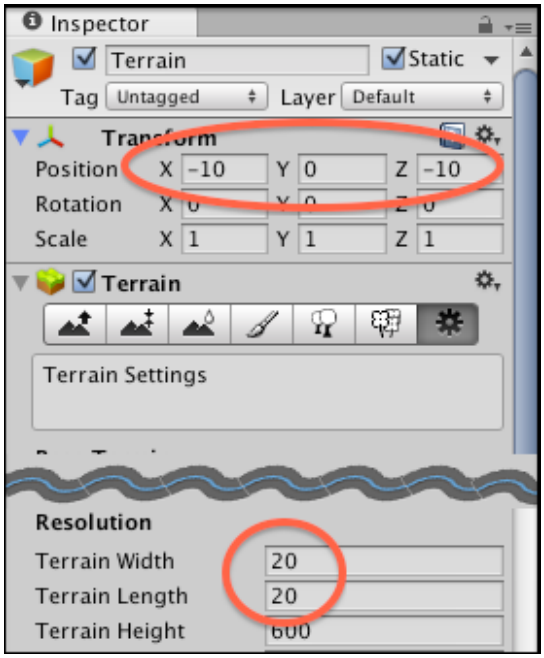
Getting ready

For this recipe, we have prepared the images you need in a folder named `Images` in `1362_01_11`.

How to do it...

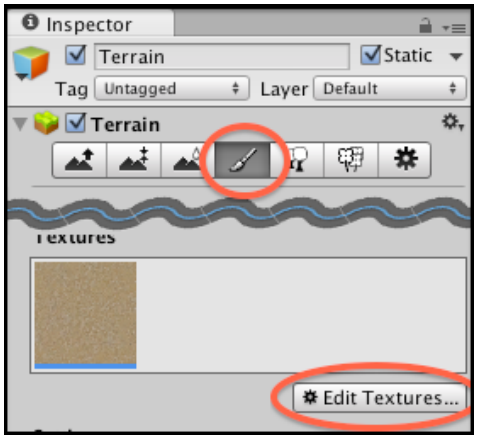
To create a radar to show relative positions of objects, follow these steps:

1. Create a new 3D project, importing the following standard assets:
   - **Environment,**
   - **Characters,**
   - **Cameras.**
2. Create a terrain, by choosing menu: **Create | 3D Object | Terrain**.
3. Size the terrain 20 x 20, positioned at (-10, 0, -10) – so its center is at (0,0,0).



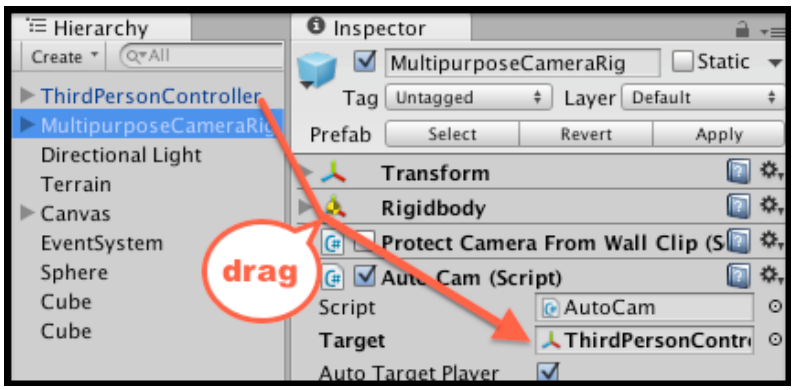
Insert image 1362OT_01_47.png

4. Texture paint your terrain with **SandAlbedo**.



Insert image 1362OT_01_48.png

5. From the **Standard Assets** folder in the **Project** panel, drag the prefab **ThirdPersonController** into the scene, and position it at (0, 1, 0).
6. Tag this **ThirdPersonController** gameObject **Player**.
7. Remove the **Main Camera** gameObject.
8. From the **Standard Assets** folder in the **Project** panel drag prefab **Multi-PurposeCameraRig** into the scene.
9. With **Multi-PurposeCameraRig** selected in the **Hierarchy**, drag gameObject **ThirdPersonController** gameObject into the **Target** property of the **Auto Cam (Script)** public variable in the **Inspector**.



Insert image 1362OT_01_22.png

10. Import the provided folder `Images`.

11. In the **Hierarchy** panel add a **UI | RawImage** game object to the scene named **RawImage-radar**.
12. Ensure gameObject **RawImage-radar** is selected in the **Hierarchy** panel. From your **Project** `Images` folder drag image `radarBackground` into the **Raw Image (Script)** public property **Texture**.
13. Now in the **Rect Transform** position **RawImage-radar** top-left using the **Anchor Presets**. Then set the width and height to 200 pixels.
14. Create another new UI **RawImage** named **RawImage-blip**. Assign this texture `yellowCircleBlackBorder`. Tag this gameObject **Blip**.
15. In the Project panel create a new empty prefab named **blip-sphere**, and drag gameObject **RawImage-blip** into this prefab to store all its properties.
16. Now change the texture of **RawImage-blip** to be `redSquareBlackBorder`.
17. In the Project panel create a new empty prefab named **blip-cube**, and drag gameObject **RawImage-blip** into this prefab to store all its properties.
18. Delete gameObject **RawImage-blip** from the **Hierarchy** panel.
19. Create a C# script class `Radar` containing the following code, and add an instance as a scripted component to gameObject **RawImage-radar**:

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

public class Radar : MonoBehaviour{
   public float insideRadarDistance = 20;
   public float blipSizePercentage = 5;

   public GameObject rawImageBlipCube;
   public GameObject rawImageBlipSphere;

   private RawImage rawImageRadarBackground;
   private Transform playerTransform;
   private float radarWidth;
   private float radarHeight;
   private float blipHeight;
   private float blipWidth;

   void Start (){
      playerTransform =
GameObject.FindGameObjectWithTag("Player").transform;
      rawImageRadarBackground = GetComponent<RawImage>();

      radarWidth =
rawImageRadarBackground.rectTransform.rect.width;
```

```
      radarHeight =
rawImageRadarBackground.rectTransform.rect.height;

      blipHeight = radarHeight * blipSizePercentage/100;
      blipWidth = radarWidth * blipSizePercentage/100;
   }

   void Update (){
      RemoveAllBlips();
      FindAndDisplayBlipsForTag("Cube", rawImageBlipCube);
      FindAndDisplayBlipsForTag("Sphere",
rawImageBlipSphere);
   }

   private void FindAndDisplayBlipsForTag(string tag,
GameObject prefabBlip){
      Vector3 playerPos = playerTransform.position;
      GameObject[] targets =
GameObject.FindGameObjectsWithTag(tag);

      foreach (GameObject target in targets) {
         Vector3 targetPos = target.transform.position;
         float distanceToTarget =
Vector3.Distance(targetPos, playerPos);
         if( (distanceToTarget <= insideRadarDistance) ){
            Vector3 normalisedTargetPosiiton =
NormalisedPosition(playerPos, targetPos);
            Vector2 blipPosition =
CalculateBlipPosition(normalisedTargetPosiiton);
            DrawBlip(blipPosition, prefabBlip);
         }
      }
   }

   private void RemoveAllBlips(){
      GameObject[] blips =
GameObject.FindGameObjectsWithTag("Blip");
      foreach (GameObject blip in blips)
         Destroy(blip);
   }

   private Vector3 NormalisedPosition(Vector3 playerPos,
Vector3 targetPos){
      float normalisedyTargetX = (targetPos.x -
playerPos.x)/insideRadarDistance;
      float normalisedyTargetZ = (targetPos.z -
playerPos.z)/insideRadarDistance;
```

```
      return new Vector3(normalisedyTargetX, 0,
normalisedyTargetZ);
   }

   private Vector2 CalculateBlipPosition(Vector3
targetPos){
      // find angle from player to target
      float angleToTarget = Mathf.Atan2(targetPos.x,
targetPos.z) * Mathf.Rad2Deg;

      // direction player facing
      float anglePlayer = playerTransform.eulerAngles.y;

      // subtract player angle, to get relative angle to
object
      // subtract 90
      // (so 0 degrees (same direction as player) is UP)
      float angleRadarDegrees =  angleToTarget -
anglePlayer - 90;

      // calculate (x,y) position given angle and distance
      float normalisedDistanceToTarget =
targetPos.magnitude;
      float angleRadians = angleRadarDegrees *
Mathf.Deg2Rad;
      float blipX = normalisedDistanceToTarget *
Mathf.Cos(angleRadians);
      float blipY = normalisedDistanceToTarget *
Mathf.Sin(angleRadians);

      // scale blip position according to radar size
      blipX *= radarWidth/2;
      blipY *= radarHeight/2;

      // offset blip position relative to radar center
      blipX += radarWidth/2;
      blipY += radarHeight/2;

      return new Vector2(blipX, blipY);
   }

   private void DrawBlip(Vector2 pos, GameObject
blipPrefab){
      GameObject blipGO =
(GameObject)Instantiate(blipPrefab);
      blipGO.transform.SetParent(transform.parent);
```

```
        RectTransform rt =
blipGO.GetComponent<RectTransform>();

    rt.SetInsetAndSizeFromParentEdge(RectTransform.Edge.Left
, pos.x, blipWidth);

    rt.SetInsetAndSizeFromParentEdge(RectTransform.Edge.Top,
pos.y, blipHeight);
    }
}
```

20. Create 2 cubes, tagged **Cube**, textured with red image **icon32_square_red**. Position each away from the player's character.
21. Create a sphere, tagged **Sphere**, textured with red image **icon32_square_yellow**. Position this away from the cubes and the player's character.
22. Run your game, you should see 2 red squares and one yellow circle on the radar, showing the relative positions of the red cubes and yellow sphere. If you move too far away, then the blips will disappear.

> NOTE: This radar script scans 360 degrees all around the player, and only considers straight line distances in the X-Z plane. So distances in this radar are not effected by any height difference between Player and target gameObject. The script could be adapted to ignore targets whose height is more than some threshold different to the Player's. Also, as presented, this recipes radar 'sees' through everything, even if there are obstacles between the Player and target. The recipe could be extended to not show obscured targets through the user of ray-casting techniques. See the Unity scriping reference for more details about ray-casting: `docs.unity3d.com/ScriptReference/Physics.Raycast.html`.

How it works...

A radar background is displayed on screen. The center of this circular image represents the position of the player's character. You have created two prefabs, one for red square images to represent each red cube found within the radar distance, and one for yellow circles to represent yellow sphere gameObjects.

The `Radar` C# script class has been added to the radar UI Image gameObject. This class defines 4 public variables:

- `insideRadarDistance` – this value defines the maximum distance an object may be from the player to still be included on the radar (objects further than this distance will not be displayed on the radar).
- `blipSizePercentage` – this public variable allows the developer to decide how large each 'blip' will be as a proportion of the radar's image.

- `rawImageBlipCube` and `rawImageBlipSphere` – these are references to the prefab UI RawImages to be used to visually indicate the relative distance and position of cubes and spheres on the radar.

Since there is a lot happening in the code for this recipe, each method will be described below in its own section.

Method Start()

The `Start()` method caches a reference to the **Transform** component of the player's character (tagged "Player"). This allows this scripted object to know about the position of the Player's character each frame. Next the width and height of the radar image are cached – so relative positions for 'blips' can be calculated based on the size of this background radar image. Finally, the size of each blip (width and height) is calculated, using the public `blipSizePercentage` variable.

Method Update()

The `Update()` method calls method `RemoveAllBlips()`, which removes any old **RawImage** UI gameObjects of cubes and spheres that might currently be displayed.

Next the method `FindAndDisplayBlipsForTag(…)` is called twice, first for objects tagged **Cube**, to be represented on the radar with prefab `rawImageBlipCube` and then again for objects tagged **Sphere**, to be represented on the radar with prefab `rawImageBlipSphere`. As you might expect, most of the hard work for the radar is to be performed by method `FindAndDisplayBlipsForTag(…)`.

Method FindAndDisplayBlipsForTag(…)

This method inputs two parameters, the string tag for objects to the searched for, and a reference to the RawImage prefab to be displayed on the radar for any such tagged objects within range.

First the current position of the player's character is retrieved, from the cached player transform variable. Next an array is constructed, referring to all gameObjects in the scene that have the provided tag. This array of gameObjects is looped through, and for each gameObject the following actions are performed:

- the position of the target gameObject is retrieved
- the distance from this target position to the player's position is calculated, and if this distance is within range (less than or equal to `insideRadarDistance`) then three steps are now required to get the blip for this object to appear on the radar:
    - the normalized position of the target is calculated by calling `NormalisedPosition(…)`

- the position of the blip on the radar is then calculated from this normalized position, by calling `CalculateBlipPosition(…)`
- finally the **RawImage** blip is displayed by calling `DrawBlip(…)` and passing the blip position and the reference to the **RawImage** prefab to be created there

Method NormalisedPosition(…)

Method `NormalisedPosition(…)` inputs the player's character position and the target gameObject position, and has the goal of outputting the relative position of the target to the player, returning a Vector3 object with a triplet of X, Y, and Z values. Note that since the radar is only 2D, we ignore the Y-value of target gameObjects, so the Y-value of the Vector3 returned by this method will always be 0. So, for example, if a target was at exactly the same location as the player, the returned X, Y, Z Vector3 object would be (0,0,0).

Since we know that target gameObject is no further from the player's character than `insideRadarDistance` then we can calculate a value in the range -1 … 0 … +1 for the X and Z axis by finding the distance on each axis from the target to the player and then dividing by `insideRadarDistance`. An X value of -1 means the target is fully to the left of the player (at distance equal to `insideRadarDistance`), and +1 means fully to the right. A value of 0 means the target has the same X position as the player's character. Likewise for -1 … 0 … +1 values in the Z-axis (this axis represents how far in front or behind us an object is located, which will be mapped to the vertical axis in our radar).

Finally this method constructs and returns a new Vector3 object, with the calculated X and Z normalized values, and a Y value of zero.

> **NOTE: Normalized position**
>
> A 'normalized' value is one that has been simplified in some way, so the context has been abstracted away. In this recipe, we are interested is where an object is relative to the player. So our normal form is to get a value of the X and Z position of a target in the range -1 to +1 for each axis. Since we are only considering gameObject within out `insideRadarDistance` value, we can then map these normalized target positions directly onto the location of the radar image in our UI.

Method CalculateBlipPosition(…)

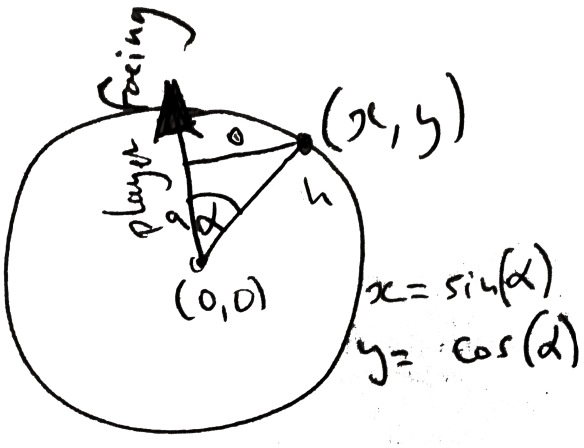
First we calculate `angleToTarget`, the angle from (0,0,0) to our normalized target position.

Next we calculate `anglePlayer`, the angle the player's character is facing. This recipe makes use of the 'yaw' angle of rotation, which is rotation about the Y-axis – that is, the

direction a character controller is facing. This can be found in the 'y' component of a GameObject's eulerAngles component of its transform. You can imagine looking from above, down at the character controller, and seeing what direction they are facing – this is just what we are trying to display graphically with the compass.

Our desired radar angle (variable `angleRadarDegrees`) is calculated by subtracting the player's direction angle from the angle between target and player, since a radar displays the relative angle from the direction the player is facing, to the target object. In mathematics, an angle of zero indicates an 'east' direction, to correct that we need to also subtract 90 degrees from the angle.

The angle is then converted into radians, since that is required for the Unity trigonometry methods. We then multiply these `Sin()` and `Cos()` results by our normalized distances, to calculate the X and Y values respectively (see figure).



Insert image 1362OT_01_46.png

Our final position values need to be expressed as pixel lengths relative to the center of the radar. So we multiple our blipX and blipY values by half the width and the height of the radar; note, only half, since these values are relative to the center of the radar.

We then add half the width and height of the radar image to the blipX/Y values, so these values are now positioned relative to the center.

Finally a new Vector2 object is created and returned, passing back these final calculated X and Y pixel values for the position of our blip icon.

Method DrawBlip()

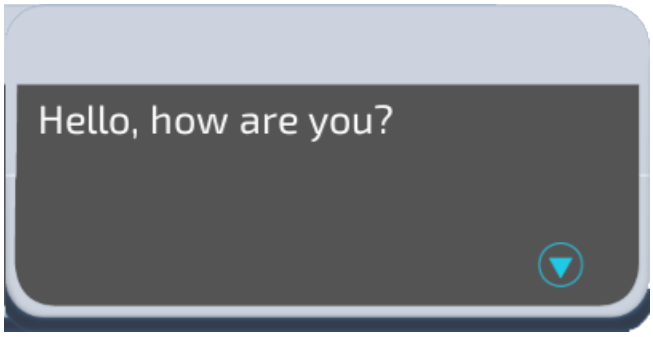
Method `DrawBlip()` takes input parameters of the position of the blip (as a Vector2 X,Y pair) and the reference to the RawImage prefab to be created at that location on the radar.

A new gameObject is created from the prefab, and parented to the radar gameObject (of which the scripted object is also a component). A reference is retrieved to the **Rect Transform** of the new **RawImage** gameObject that has been created for the 'blip'. Calls to the Unity **RectTransform** method `SetInsetAndSizeFromParentEdge(…)` result in the blip gameObject being positioned at the provided horizontal and vertical locations over the radar image, regardless of where in the **Game** panel the background radar image has been located.

Creating UIs with the Fungus open source dialog system

Rather than constructing your own UI and interactions from scratch each time, there are plenty of UI and dialogue systems available for Unity. One powerful, free and open source dialog system is called Fungus, which uses a visual flowcharting approach to dialog design.

In this recipe, we'll create a very simple, two-sentence dialogue, to illustrate the basics of Fungus.



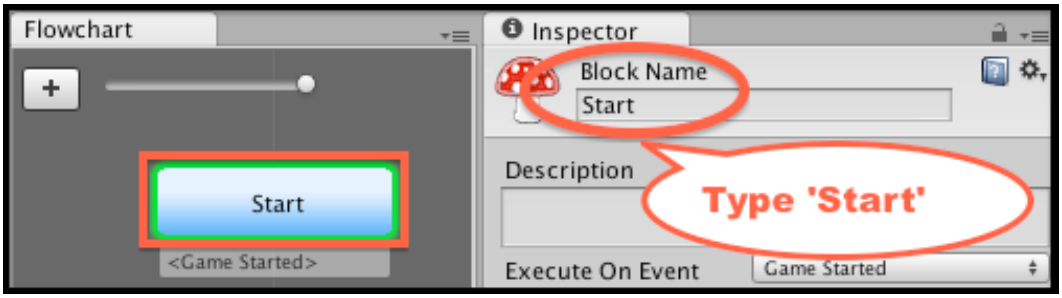
Insert image 1362OT_01_23.png

How to do it...

To create a two-sentence dialogue using Fungus, follow these steps:

1. Download the latest version of the Fungus **unitypackage** from the FungusGames website: `http://fungusgames.com/`.

2. Create a new Unity 2D project.
3. Import the Fungus **unitypackage** by choosing menu: **Assets | Import Package | Custom Package...**, and then navigating to your downloaded file location.
4. Create a new Fungus **Flowchart** gameObject by choosing menu: **Tools | Fungus | Create | Flowchart**.
5. Display and dock the Fungus **Flowchart** window panel, by choosing menu: **Tools | Fungus | Flowchart Window**.
6. There will be one block in the **Flowchart** Window, click this block to select it (a green border appears around the block to indicate it is selected), and then in the **Inspector** panel change the name of this block to **Start**.

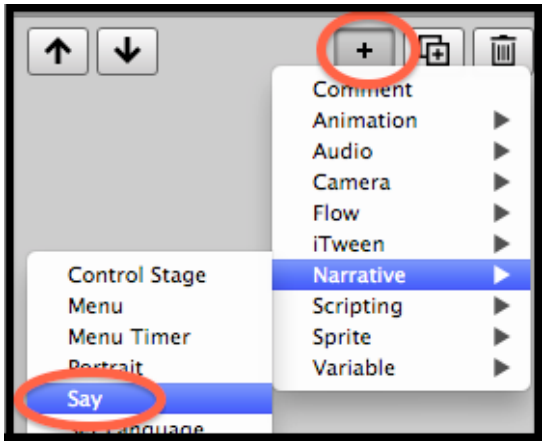


Insert image 1362OT_01_24.png

7. Each Block in a Flowchart follows a sequence of commands, so we are now going to create a sequence of commands to display 2 sentences to the user when the game runs.

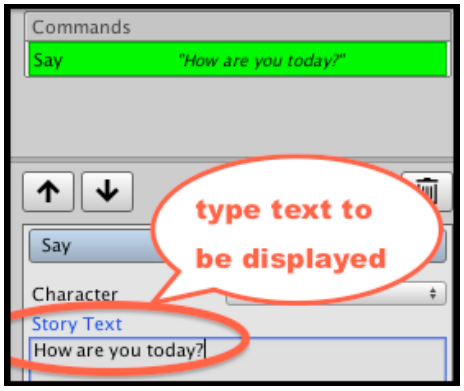
> Sequence of **Commands** in a **Block**
>
> Each **Block** in a **Flowchart** follows a sequence of **Commands**, so to display 2 sentences to the user when the game runs we need to create a sequence of 2 **Say** commands in the **Inspector** properties for our block.

8. Ensuring the **Start** block is still selected in the **Flowchart** panel, now click the plus '+' button at the bottom section of the **Inspector** panel to display the menu of **Commands**, and select the **Narrative | Say** command.



Insert image 1362OT_01_25.png

9. Since we only have one command for this block, that command is automatically selected (highlighted green) in the top half of the **Inspector**. The bottom half of the **Inspector** presents the properties for the currently selected **Command**. In the bottom half of the **Inspector** for the **Story Text** property, enter the text of the question we wish to be presented to the user: **How are you today?**.



Insert image 1362OT_01_26.png

10. Now create another **Say Command**, and type the following for its **Story Text** property: **Very well thank you.**
11. When you run the game, the user should first be presented with the text **How are you today?** (hearing a clicking noise as each letter is 'typed' on screen). After the user clicks the 'continue' triangle button (bottom right of the dialogue

window), they will then be presented with the second sentence: **Very well thank you.**

How it works...

You have created a new Unity project, and imported the Fungus asset package, containing the Fungus Unity menus, windows and commands, and also example projects.

You have added a **Fungus Flowchart** to your scene, with a single **Block** that you have named **Start**. Your block starts to execute when the game begins (since the default for the first block is to execute upon receiving the event **Game Started**).

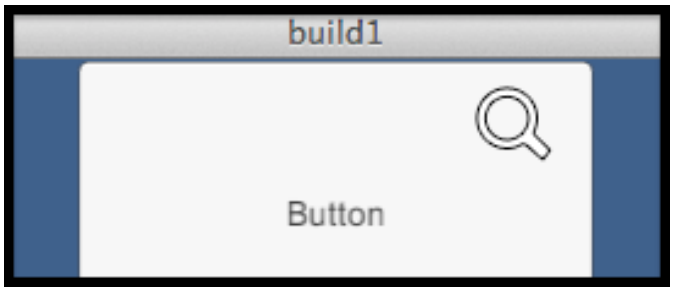
In block **Start** you added a sequence of two **Say Commands**. Each command presents a sentence to text to the user, and then waits for the continue button to be clicked before proceeding to the next **Command**.

As can be seen, the Fungus system handles the work of creating a nicely presented panel to the user, displaying the desired text and continue button. Fungus offers many more features, including menus, animations, control of sounds and music, and so on, details of which can be found by exploring their provided example projects, and their websites:

- `http://fungusgames.com/`
- `https://github.com/FungusGames/Fungus`

Setting custom mouse cursor images

Cursor icons are often used to indicate the nature of the interaction that can be done with the mouse. Zooming, for instance, might be illustrated by a magnifying glass. Shooting, on the other hand, is usually represented by a stylized target. In this recipe, we will learn how to implement custom mouse cursor icons to better illustrate your gameplay – or just to escape the Windows, OSX, and Linux default GUI.



Insert image 1362OT_01_32.png

Getting ready

For this recipe, we have prepared the images you need in a folder named `IconsCursors` in folder `1362_01_13`.

How to do it...

To make a custom cursor appear when the mouse is over a gameObject, follow these steps:

1. Create a new Unity 2D project.
2. Add a **Directional Light** to the scene, by choosing menu: **Create | Light | Directional light**.
3. Add a 3D **Cube** to the scene, scaled to (5,5,5) – because this was created as a 2D project the cube will appear as a grey square in the **Game** panel (2D projects have an orthographic camera, so we won't see perspective effects).
4. Import the provided folder `IconsCursors`.

> Ensure each image in this folder has been imported as Texture Type **Cursor** – if they are not, then select this type for each image and click the **Apply** button in the **Inspector**.

5. Create a C# script class `CustomCursorPointer` containing the following code, and add an instance as a scripted component to gameObject **Cube**:

```
using UnityEngine;
using System.Collections;

public class CustomCursorPointer : MonoBehaviour {
   public Texture2D cursorTexture2D;

   private CursorMode cursorMode = CursorMode.Auto;
   private Vector2 hotSpot = Vector2.zero;

   public void OnMouseEnter() {
      SetCustomCursor(cursorTexture2D);
   }

   public void OnMouseExit() {
      SetCustomCursor(null);
   }

   private void SetCustomCursor(Texture2D curText){
      Cursor.SetCursor(curText, hotSpot, cursorMode);
```

```
    }
}
```

> NOTE: Event methods `OnMouseEnter()` and `OnMouseExit()` have been purposely declared as public. This will allow these methods to also be called from UI gameObjects when they receive OnPointerEnterExit events.

6. With **Cube** selected in the **Hierarchy** drag the `CursorTarget` image into the public **Cursor Texture 2D** variable slot in the **Inspector** for component **Customer Cursor Pointer (Script)**.
7. Save the current scene, and add it to the Build.

> NOTE: You will not be able to see the custom cursors in the Unity Editor. You must build your game application, and you'll see the custom cursors when you run the build app.

8. Build your project. Now run your built application and when the mouse pointer moves over the grey square of the **Cube** it should change to the custom `CursorTarget` image you chose.

How it works...

You have added a scripted object to a cube that will tell Unity to change the mouse pointer when an **OnMouseEnter** message is received – that is, when the user's mouse point moves over the part of the screen where the cube is being rendered. When an **OnMouseExit** event is received (the users mouse pointer is no longer over the cube part of the screen), the system is told to revert to the operating system default cursor. This event should be received within a few milliseconds of the user's mouse exiting from the collider.

There's more...

Some details you don't want to miss:

Custom cursors for mouse over UI controls

Unity 5 UI controls do not receive **OnMouseEnter** and **OnMouseExit** events. They can respond to **PointerEnter/Exit** events, but this requires adding **Event Trigger** components. To change the mouse pointer when the mouse moves over a UI element, do the following:

1. Add a UI **Button** to the scene.

2. Add an instance of C# script class `CustomCursorPointer` to the button.
3. With **Button** selected in the **Hierarchy**, drag the `CursorZoom` image into the public **Cursor Texture 2D** variable slot in the **Inspector** for component **Customer Cursor Pointer (Script)**.
4. In the **Inspector** add an **Event Triggers** component to the **Button**. Choose menu: **Add Component | Event | Event Trigger**.
5. Add a **Pointer Enter** event to your **Event Trigger** component, click the plus "+" button to add an event handler slot, and drag the Button gameObejct into the Object slot.
6. From the Function dropdown menu choose **CustomCursorPointer** and then choose method **OnMouseEnter**.

> We have added an Event Handler so that when the **Button** receives a **Pointer Enter** (mouse over) event, it will execute method the **OnMouseEnter()** method of the **CustomCursorPointer** scripted object inside the Button.

7. Add a **Pointer Exit** event to your **Event Trigger** component, and make it call method `OnMouseExit()` from **CustomCursorPointer** when this event is received.
8. Save the current scene.
9. Build your project. Now run your built application and when the mouse pointer moves over the **Button** it should change to the custom `CursorZoom` image you chose.

User interaction Input Field for text entry

While many times we just wish to display non-interactive text messages to the user, there are times (such as name entry for high scores) where we wish the user to be able to enter text or numbers into our game. Unity provides the Input Field UI component for this purpose. In this recipe we create a simple text input UI making use of the default Button image and text gameObjects, and we add a script to respond to each new value of the input field.

> NOTE: You could, of course, create a working text input quicker than this recipe's method by choosing menu: **Create | UI | Input Field**, which creates a gameObject containing an Input Field component, and child text and placeholder gameObjects. However, by following the steps in this recipe you'll learn the interrelationships between the different interface elements, because

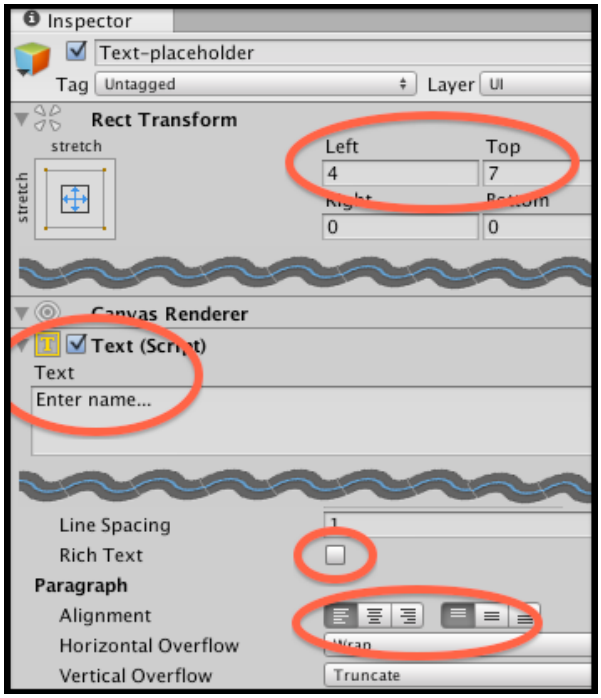
you'll be creating those connections manually from the deconstructed parts of the UI Button gameObject.

Insert image 1362OT_01_37.png

How to do it...

To create a promoted text input box to the user, with faint placeholder text, follow these steps:

1. Create a new Unity 2D project.
2. In the **Inspector** change the background of the **Main Camera** to solid white.
3. Add a **UI Button** to the scene. Delete the **Button (Script)** component of this **Button** gameObject (since it won't be a button, it will be an interactive text input by the time we are finished with it!).
4. Rename the **Text** child gameObject of the **Button** to **Text-placeholder**. Uncheck the **Rich Text** option, change the text to **Enter name…**, change the **Alignment** to Left and Top, and in the **Rect Transform** set **Left** to 4 and **Top** to 7.



Insert image 1362OT_01_38.png

5. Duplicate **Text-placeholder** naming the copy **Text-prompt**. Change the **Text** of this gameObject to **Name:**, and set its **Left** position to -50.
6. Duplicate **Text-placeholder** again, naming this new copy **Text-input**. Delete all of the content of the Text property of this new gameObject.
7. Select **Text-placeholder** in the **Hierarchy** and we will now make the placeholder text mostly transparent. Set to about a quarter (64) the **A** (alpha) **Color** value of the **Text (Script)** component of this gameObject.
8. Select **Text-input** in the **Hierarchy** and add an **Input Field** component, by choosing menu: **Add Component | UI | Input Field**.
9. Drag gameObject **Text-input** into the **Text Component** property of this **Input Field**, and drag gameObject **Text-placeholder** into the **Placeholder** property.
10. Save and run your scene. You now have a working text input UI for your user. When there is no text content, the faint placeholder text will be displayed. As soon as any characters have been typed, the placeholder will be hidden and the characters typed will appear in black text. Then, if all the characters are deleted, the placeholder would appear again.

How it works...

The core to interactive text input in Unity is the responsibility of the **Input Field** component. This needs a reference to a UI **Text** gameObject. To make it easier to see where the text can be typed, we have made use of the default rounded rectangle image that Unity provides when a **Button** gameObject is created. **Buttons** have both an Image component and a **Text** child gameObject – so two of the items we need we can get very easily by creating a new **Button** and simply removing the **Button (Script)** component.

There are usually 3 **Text** gameObjects involved with user text input: the static prompt text (in our recipe example the text **Name:**); then the faint placeholder text, reminding users where and what they should type; and finally the text object (with font and color settings and so on) that is actually displayed to the user showing the characters as they type.

At runtime a **Text-Input Input Caret** gameObject is created – displaying the blinking vertical line to inform the user of where their next letter will be typed. Note, the **Content Type** of the **Input Field (Script)** in the **Inspector** can be set to several specific types of text input, including email addresses, integer or decimal numbers only, password text (where an asterisk is displayed for each entered character).

There's more...

Some details you don't want to miss:

Execute a C# method to respond to each time the user changes the input text content

Having interactive text on screen isn't much use unless we can retrieve the text entered to use in our game logic, and we may need to know each time the user changes the text content and act accordingly.

To add code and events to respond to each time the text content has been changed by the user, do the following:

1. Add an instance of C# script class `DisplayChangedTextContent` to gameObject **Text-input**:

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

public class DisplayChangedTextContent : MonoBehaviour {
   private InputField inputField;

   void Start(){
      inputField = GetComponent<InputField>();
```

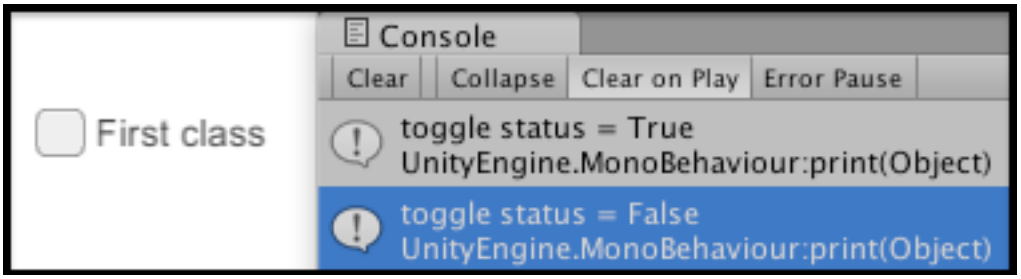
```
    }

    public void PrintNewValue (){
        string msg = "new content = '" + inputField.text +
"'";
        print (msg);
    }
}
```

2. Add an **End Edit (String)** event to the list of event handlers for the **Input Field (Script)** component, click the plus "+" button to add an event handler slot, and drag gameObject **Text-input** into the Object slot.
3. From the Function dropdown menu choose **DisplayChangedTextContent** and then choose method **PrintNewValue**.
4. Save and run the scene. Each time the user types new text and then presses *Tab* or *Enter* the **End Edit** event will fire, and you'll see a new content text message printed into the Console window by our script.

Insert image 1362OT_01_36.png

User interaction Toggles and radio buttons via Toggle Groups

Users make choices, and often these choices are whether to have one of the two available options (for example, sound on or off), or sometimes to choose one of several possibilities (for example, difficulty level easy / medium / hard). Unity UI **Toggles** allow users to turn options on and off, and when combined with Toggle Groups they restrict choices to one of a group of items. In this recipe, we'll first explore the basic **Toggle** and a script to respond to change in values. Then in the *There's More* section, we'll extend the example to illustrate **Toggle Groups** and styling these with round images to make the look more like traditional radio buttons.



Insert image 1362OT_01_39.png

Getting ready

For this recipe, we have prepared the images you need in a folder named `UI Demo Textures` in folder `1362_01_15`.

How to do it...

To display an on/off UI Toggle to the user, follow these steps:

1. Create a new Unity 2D project.
2. In the **Inspector** change the **Background** color of the **Main Camera** to white.
3. Add a **UI Toggle** to the scene.
4. Enter **First Class** as the **Text** for the **Label** child gameObject of the **Toggle**.
5. Add an instance of C# script class `ToggleChangeManager` to the **Toggle** gameObject:

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

public class ToggleChangeManager : MonoBehaviour {
   private Toggle toggle;

   void Start () {
      toggle = GetComponent<Toggle>();
   }

   public void PrintNewToggleValue(){
      bool status = toggle.isOn;
      print ("toggle status = " + status);
   }

}
```

6. With the **Toggle** gameObject selected add an **On Value Changed** event to the list of event handlers for the **Toggle (Script)** component, click the plus "+" button to add an event handler slot, and drag gameObject **Toggle** into the Object slot.
7. From the Function dropdown menu, choose **ToggleChangeManager** and then choose method **PrintNewToggleValue**.
8. Save and run the scene. Each time you check or uncheck the **Toggle** then the **On Value Changed** event will fire, and you'll see a new text message printed into the Console window by our script stating the new Boolean true/false value of the **Toggle**.

How it works...

When you created a Unity UI **Toggle** gameObject it comes with several child gameObjects automatically – **Background, Checkmark**, and text **Label**. Unless we need to style the look of a **Toggle** in a special way, all that is needed usually is simply to edit the text **Label** so the user knows what option or feature this **Toggle** is going to turn on/off.

The C# scripted class `ToggleChangeManager` method `Start()` gets a reference to the Toggle component in the gameObject to which the script instance is located. When the game is running, each time the user clicks the Toggle to change its value an **On Value Changed** event is fired, and we registered method `PrintNewToggleValue()` to be executed when such an event occurs. This method retrieves and then prints out to the **Console** the new Boolean true/false value of the **Toggle**.

There's more...

Some details you don't want to miss:

Adding more toggles and a Toggle Group to implement mutually-exclusive radio buttons

Unity UI **Toggles** are also the base component if we wish to implement a group of mutually-exclusive options in the style of radio buttons. To create such a group of related choices, do the following:

1. Import folder `UI Demo Textures` into the project.
2. Remove the C# script class `ToggleChangeManager` component from the **Toggle** gameObject.
3. Rename gameObject **Toggle** as **Toggle-easy**.
4. Change the Label text to **Easy**, and tag this gameObject with new tag **Easy**.

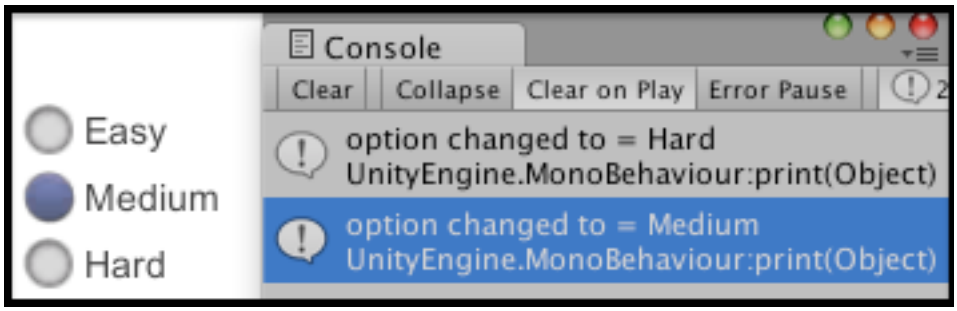
5. Select the **Background** child gameObject of **Toggle-easy**, and in the **Image (Script)** component drag image `UIToggleBG` into the **Source Image** property.
6. Ensure the **Is On** property of the **Toggle (Script)** component is checked, and then select the **Checkmark** child gameObject of **Toggle-easy**, and in the **Image (Script)** component, drag image `UIToggleButton` into the **Source Image** property.

> Of the 3 choices (easy, medium, hard) we'll offer the user, we'll set the easy option to be the one initially selected. Therefore we need its "**Is On**" property to be checked, which will lead to its Checkmark image being displayed.
>
> To make these Toggles look more like radio buttons, the background of each is set to the circle image of `UIToggleBG`, and the checkmark (which displays for Toggles that are On) is filled circle image `UIToggleButton`.

7. Duplicate gameObject **Toggle-easy**, naming the copy **Toggle-medium**. Set its **Rect Transform** property Pos Y to -25 (so this copy is positioned below the easy option), and uncheck the **Is On** property of the **Toggle (Script)** component. Tag this copy with a new tag **Medium**.
8. Duplicate gameObject **Toggle- medium**, naming the copy **Toggle-hard**. Set its **Rect Transform** property Pos Y to -50 (so this copy is positioned below the medium option). Tag this copy with a new tag **Hard**.
9. Add an instance of C# script class `RadioButtonManager` to gameObject **Canvas**:

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

public class RadioButtonManager : MonoBehaviour {
   private string currentDifficulty = "easy";

   public void PrintNewGroupValue(Toggle sender){
      // only take notice from Toggle just swtiched to On
      if(sender.isOn){
         currentDifficulty = sender.tag;
         print ("option changed to = " +
currentDifficulty);
      }
   }
}
```

10. With the **Toggle-easy** gameObject selected add an **On Value Changed** event to the list of event handlers for the **Toggle (Script)** component, click the plus "+" button to add an event handler slot, and drag gameObject **Canvas** into the Object slot.
11. From the Function dropdown menu choose **RadioButtonManager** and then choose method **PrintNewGroupValue**. In the Toggle parameter slot, which is initially None (Toggle), drag gameObject **Toggle-easy**.
12. Do the same for gameObjects **Toggle-medium** and **Toggle-hard** – so each Toggle object calls method `PrintNewGroupValue(…)` of C# scripted component `RadioButtonManager` in the Canvas gameObject, passing itself as a parameter.
13. Save and run the scene. Each time you check one of the three radio buttons the **On Value Changed** event will fire, and you'll see a new text message printed into the **Console** window by our script, stating the tag of whichever **Toggle** (radio button) was just set to True (**Is On**).



Insert image 1362OT_01_40.png

Conclusion

In this chapter, we have introduced recipes demonstrating a range of Unity 5 UI components, and illustrated how the same components can be used in different ways (such as an interactive slider being used to display the status of a countdown timer). One set of UI components in many games are those that communicate to the user what they are carrying (or yet to pickup), we have dedicated another chapter in this book to inventory UIs in *Chapter 3*, *Inventory GUIs*, which provides many recipes and additional UI controls such as adding interactive scroll bars.

Here are some suggestions for further reading, tutorials and resources to help you continue your learning of UI development in Unity.

- Learn more about the Unity UI on their manual pages: `http://docs.unity3d.com/Manual/UISystem.html`.

- Work through the Unity UI tutorial videos:
  https://unity3d.com/learn/tutorials/modules/beginner/ui.
- Ray Wenderlich's great tutorial on Unity UI development:
  http://www.raywenderlich.com/78675/unity-new-gui-part-1.

  Unit's documentation pages about designing UI for multiple resolutions:
  http://docs.unity3d.com/Manual/HOWTO-UIMultiResolution.html.

Games need fonts in a style to match the gameplay and theme – here are some sources of free personal/commercial fonts suitable for many games:

- FontSquirrel – all fonts 100% free for commercial use:
  http://www.fontsquirrel.com/

- DaFont website – see each font for individual license, many ask for donation if used for commercial purposes:
  http://www.dafont.com/xolonium.font

- Naldz Graphics blog - see each font for individual license:
  http://naldzgraphics.net/textures/

- 1001 Free Fonts (for personal use):
  http://www.1001freefonts.com/index.php