GUI and Game Management

The final things to add to our game are the GUI and *game management* that will make it feel like more of a real game. The GUI element we'll add is a score counter, and the game management elements we'll add are rounds and the ability for the game to restart when the player has run out of Baskets.

Score Counter

The score counter will help players get a sense of their level of achievement in the game.

Open _Scene_0 by double-clicking it in the Project pane. Then go to the menu bar and choose *GameObject > Create Other > GUI Text*. This will place a GUIText in the middle of the screen with the words *Gui Text* in it. Rename GUI Text to *ScoreCounter*. Try changing the x and y position of ScoreCounter. You'll notice that the coordinates for GUITexts differ completely from those for other GameObjects. This is because GUITexts are positioned relative to the screen rather than being positioned in world space. An x value of 0 is the far-left edge of the screen, and an x value of 1 is the right edge. A y value of 0 is the bottom of the screen, and a y value of 1 is the top. (Note that this also differs from the screen coordinates of Input.mousePosition, for which a y value of 0 is the top of the screen.)

Set the Transform and GUIText components of ScoreCounter as shown in the left half of Figure 28.10.

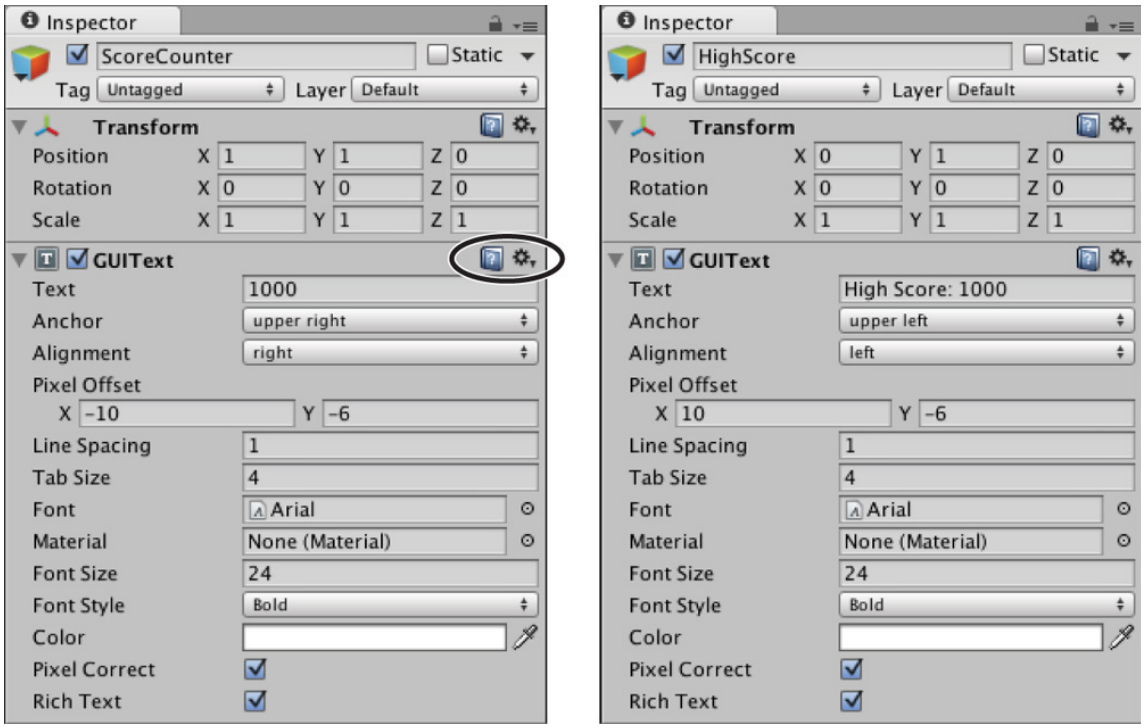

Figure 28.10 Transform and GUIText component settings for ScoreCounter and HighScore

For more information on the GUIText component, click the help icon in the top-right corner of the GUIText component (circled in black in Figure 28.10). You can use these help icons to learn more about any component.

Add Points for Each Caught Apple

There are two scripts that are notified when a collision occurs between an Apple and a Basket: the Apple and Basket scripts. In this game, there is already an `OnCollisionEnter()` method on the Basket C# script, so we'll modify this to give the player points for each Apple that is caught. 100 points per Apple seems like a reasonable number (though I've personally always thought it was a little ridiculous to have those extra zeroes at the end of scores). Open the Basket script in MonoDevelop and add the bolded lines shown here:

Click here to view code image

```
using UnityEngine;
```

```
using System.Collections;

public class Basket : MonoBehaviour {

    public GUIText        scoreGT;                                          // 1

    void Update () {
        ...
    }

    void Start() {
        // Find a reference to the ScoreCounter GameObject
        GameObject scoreGO = GameObject.Find("ScoreCounter");               // 2
        // Get the GUIText Component of that GameObject
        scoreGT = scoreGO.GetComponent<GUIText>();                          // 3
        // Set the starting number of points to 0
        scoreGT.text = "0";
    }

    void OnCollisionEnter( Collision coll ) {
        // Find out what hit this Basket
        GameObject collidedWith = coll.gameObject;
        if ( collidedWith.tag == "Apple" ) {
            Destroy( collidedWith );
        }

        // Parse the text of the scoreGT into an int
        int score = int.Parse( scoreGT.text );                              // 4
        // Add points for catching the apple
        score += 100;
        // Convert the score back to a string and display it
        scoreGT.text = score.ToString();
    }
}
```

1. Be sure you don't neglect to enter this line. It's in an earlier part of the code than the others.
2. `GameObject.Find("ScoreCounter")` searches through all the GameObjects in the scene for one named `"ScoreCounter"` and assigns it to the local variable `scoreGO` .
3. `scoreGO.Getcomponent<GUIText>()` searches for a GUIText component on the `scoreGO` GameObject, and this is assigned to the public field `scoreGT` . The starting score is then set to zero on the next line.
4. `int.Parse( scoreGT.text )` takes the text shown in ScoreCounter and converts it to an integer. 100 points are added to the int `score` , and it is then assigned back to the text of `scoreGT` after being parsed from an int to a string by `score.ToString()` .

Notifying Apple Picker That an Apple Was Dropped

Another aspect of making Apple Picker feel more like a game is ending the round and deleting a Basket if an Apple is dropped. At this point, Apples manage their own destruction, which is fine, but the Apple needs to somehow notify the ApplePicker script of this event so that Apple Picker can end the round and destroy the rest of the Apples. This will involve one script calling a function on another. Start by making these modifications to the Apple C# script:

Click here to view code image

```
using UnityEngine;
using System.Collections;

public class Apple : MonoBehaviour {
    public static float     bottomY = -20f;

    void Update () {
        if ( transform.position.y < bottomY ) {
            Destroy( this.gameObject );

            // Get a reference to the ApplePicker component of Main Camera
            ApplePicker apScript = Camera.main.GetComponent<ApplePicker>(); // 1
```

```
            // Call the public AppleDestroyed() method of apScript
            apScript.AppleDestroyed();                                  // 2
        }
    }
}
```

1. Grabs a reference to the ApplePicker script component on the Main Camera. Because the `Camera` class has a built-in static variable `Camera.main` that references the Main Camera, it is not necessary to use `GameObject.Find("Main Camera")` to obtain a reference to Main Camera. `Getcomponent<ApplePicker>()` is then used to grab a reference to the *ApplePicker (Script)* component on Main Camera and assign it to `apScript`. After this is done, it is possible to access public variables and methods of the `ApplePicker` instance that is attached to Main Camera.
2. This calls a non-existant `AppleDestroyed()` method of the `ApplePicker` instance.

There is currently no public `AppleDestroyed()` method in the ApplePicker script, so you will need to open the ApplePicker C# script in MonoDevelop and make the following bolded changes:

Click here to view code image

```
using UnityEngine;
using System.Collections;

public class ApplePicker : MonoBehaviour {

    public GameObject       basketPrefab;
    ...                                                             // 1
    public float            basketSpacingY = 2f;

    void Start () {
        ...
    }

    public void AppleDestroyed() {                                  // 2
        // Destroy all of the falling Apples
        GameObject[] tAppleArray=GameObject.FindGameObjectsWithTag("Apple");// 3
        foreach ( GameObject tGO in tAppleArray ) {
            Destroy( tGO );
        }
    }
}
```

1. This is another way that ellipses (`...`) are used to shorten code listings. Here, lines have been omitted between the lines above and below the ellipses. Again, this is an indication that you don't need to modify any code between the lines.
2. The `AppleDestroyed()` method must be declared `public` for other classes (like `Apple`) to be able to call it. By default, methods are all *private* and unable to be called (or even seen) by other classes.
3. `GameObject.FindGameObjectsWithTag("Apple")` will return an array of all existing Apple GameObjects. The subsequent `foreach` loop iterates through each of these and destroys them.

Destroying a Basket When an Apple Is Dropped

The final bit of code for this scene will manage the deletion of one of the Baskets each time an Apple is dropped and stop the game when all the Baskets have been destroyed. Make the following changes to the ApplePicker C# script:

Click here to view code image

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;                                   // 1

public class ApplePicker : MonoBehaviour {

    ...                                                             // 2
    public float            basketSpacingY = 2f;
    public List<GameObject> basketList;
```

```
    void Start () {
        basketList = new List<GameObject>();
        for (int i=0; i<numBaskets; i++) {
            GameObject tBasketGO = Instantiate( basketPrefab ) as GameObject;
            Vector3 pos = Vector3.zero;
            pos.y = basketBottomY + ( basketSpacingY * i );
            tBasketGO.transform.position = pos;
            basketList.Add( tBasketGO );                                    // 3
        }
    }

    public void AppleDestroyed() {
        // Destroy all of the falling Apples
        GameObject[] tAppleArray = GameObject.FindGameObjectsWithTag( "Apple" );
        foreach ( GameObject tGO in tAppleArray ) {
            Destroy( tGO );
        }

        //// Destroy one of the Baskets
        // Get the index of the last Basket in basketList
        int basketIndex = basketList.Count-1;
        // Get a reference to that Basket GameObject
        GameObject tBasketGO = basketList[basketIndex];
        // Remove the Basket from the List and destroy the GameObject
        basketList.RemoveAt( basketIndex );
        Destroy( tBasketGO );
    }

}
```

1. We will be storing the Basket GameObjects in a List, so it is necessary to use the `System.Collections.Generic` code library. (For more information about Lists, see Chapter 22, "Lists and Arrays.") The `public List<GameObject> basketList` is declared at the beginning of the class, and it is defined and initialized in the first line of `Start()`.
2. Here, the ellipses omit all the lines before `public float basketSpacingY = 2f;`.
3. A new line is added to the end of the for loop that `Adds` the baskets to `basketList`. The baskets are added in the order they are created, which means that they are added bottom to top.

In the method `AppleDestroyed()` a new section has been added to destroy one of the Baskets. Because the Baskets are added from bottom to top, it's important that the last Basket in the List is destroyed first (to destroy the Baskets top to bottom).

Adding a High Score

Create a new GUIText in the scene just as you did for the ScoreCounter and name it *HighScore*. Give its Transform and GUIText components the settings shown in the right side of Figure 28.10.

Next, create a new C# script named *HighScore*, attach it to the HighScore GameObject in the Hierarchy pane, and give it the following code:

Click here to view code image

```
using UnityEngine;
using System.Collections;

public class HighScore : MonoBehaviour {
    static public int    score = 1000;

    void Update () {
        GUIText gt = this.GetComponent<GUIText>();
        gt.text = "High Score: "+score;
    }
}
```

The lines in `Update()` simply display the value of `score` in the GUIText component. It is not necessary to call `ToString()` on the `score` in this instance because when the `+` operator is used to concatenate a string with another data type (the int `score` in this case), `ToString()` is called implicitly (that is, automatically).

Making the int `score` not only public but also static gives us the ability to access it from any other script by simply typing `HighScore.score`. This is one of the powers of static variables that we will use throughout the prototypes in this book. Open the Basket C# script and add the following lines to see how this is used:

Click here to view code image

```
void OnCollisionEnter( Collision coll ) {
    ...
    // Convert the score back to a string and display it
    scoreGT.text = score.ToString();

    // Track the high score
    if (score > HighScore.score) {
        HighScore.score = score;
    }
}
```

Now `HighScore.score` is set any time the current score exceeds it.

Finally, open the ApplePicker C# script and add the following lines to reset the game whenever a player runs out of Baskets:

Click here to view code image

```
public void AppleDestroyed() {
    ...

    //// Destroy one of the Baskets
    ...
    basketList.RemoveAt( basketIndex );
    Destroy( tBasketGO );

    // Restart the game, which doesn't affect HighScore.Score
    if ( basketList.Count == 0 ) {
        Application.LoadLevel( "_Scene_0" );
    }
}
```

`Application.LoadLevel( "_Scene_0" )` will reload _Scene_0. This effectively resets the game to its beginning state. However, because `HighScore.score` is a static variable, it is not reset along with the rest of the game. This means that high scores will remain from one round to the next. However, whenever you press the Play button again to stop the game, `HighScore.score` will reset. It is possible to fix this through the use of Unity's PlayerPrefs. PlayerPrefs store information from Unity scripts on the computer so that the information can be recalled later and isn't destroyed when playback stops. Add the following bolded changes to the HighScore C# script:

Click here to view code image

```
using UnityEngine;
using System.Collections;

public class HighScore : MonoBehaviour {
    static public int    score = 1000;

    void Awake() {                                                  // 1
        // If the ApplePickerHighScore already exists, read it
        if (PlayerPrefs.HasKey("ApplePickerHighScore")) {           // 2
            score = PlayerPrefs.GetInt("ApplePickerHighScore");
        }
        // Assign the high score to ApplePickerHighScore
        PlayerPrefs.SetInt("ApplePickerHighScore", score);          // 3
    }

    void Update () {
        GUIText gt = this.GetComponent<GUIText>();
        gt.text = "High Score: "+score;
        // Update ApplePickerHighScore in PlayerPrefs if necessary
        if (score > PlayerPrefs.GetInt("ApplePickerHighScore")) {   // 4
            PlayerPrefs.SetInt("ApplePickerHighScore", score);
        }
```

```
    }
}
```

1. `Awake()` is a built-in Unity method (like `Start()` or `Update()`) that happens when the instance of HighScore is first created (so `Awake()` always occurs before `Start()`).
2. PlayerPrefs is a dictionary of values that are referenced through keys (that is, unique strings). In this case, we're referencing the key *ApplePickerHighScore*. Here, the first line checks to see whether an *ApplePickerHighScore* already exists in PlayerPrefs and reads it in if it does exist.
3. The last line of `Awake()` assigns the current value of `score` to the ApplePickerHighScore PlayerPrefs key. If an ApplePickerHighScore already exists, this will rewrite the value back to PlayerPrefs; if the key does not already exist, however, this ensures that an ApplePickerHighScore key is created.
4. With the added lines, `Update()` now checks every frame to see whether the current `HighScore.score` is higher than the one stored in PlayerPrefs and updates PlayerPrefs if that is the case.

This usage of PlayerPrefs enables the Apple Picker high score to be remembered on this machine, and the high score will survive stopping playback, quitting Unity, and even restarting your computer.