Virtual Reality and extra features

In this chapter, we will cover:

1. Pausing the game
2. Implementing slow motion
3. Destroying objects at death a time
4. Gizmo to show selected object in Scene panel
5. Gizmo to display icon for object type
6. Gizmo to draw a grid in the Scene panel
7. Optimization with the Unity Profiler
8. Particles
9. Recording in Unity
10. Creating a simple VR scene
11. Deploying a simple VR scene to a standalone executable
12. Using a Windows batch file to force VR mode when running an executable
13. Balancing Sharpness and Performance by changing the Render Scale
14. User Interaction with the VREyeCaster
15. Gesture input with VRInput events
16. Object interaction with the VRInteractiveItem component
17. Gaze selection with a visible reticle
18. Displaying a clock to explore Spatial UI
19. Free anti-aliasing on text with a Canvas Scaler on a Worldspace Canvas
20. Fading when changing positions (to avoid VR Sickness nausea)
21. Optimizing for VR applications

Introduction

There are too many features in Unity 2018 to all be covered in an single book. In this chapter we present a set of recipes illustrating VR game development in Unity, plus a range of additional Unity features.

The Big Picture

Virtual Reality is about presenting to the player an immersive audio-visual experience, engaging enough for them to lose them selves in exploring and interacting with the game world that has been created.

From one point of view, VR simply requires 2 cameras, to generate the images for each eye, to give that realistic 3D effect.

xxxx

xxxx

Gizmos

Gizmos are another kinds of Unity editor customization. Gizmos are provide visual aids to game designers in the Scene panel. They can be useful as setup aids (to help us know what we are doing), or for debugging (understanding why objects aren't behaving as expected).

Gizmos are not drawn through Editor scripts, but as part of Monobehavious - i.e. they only work for GameObjects in the current scene. Gizmo drawing is usually performed in 2 methods:

- OnDrawGizmos(): This is executed every frame, for every GameObject in the hierarchy
- OnDrawGizmosSelect(): This is executed every frame, for just the/those GameObject(s) that are currently selected in the hierarchy

Gizmo graphical drawing makes it simple to draw lines, cubes, spheres. Also more complex shapes with meshes, and also displaying 2D image icons (located in the Assets/Gizmos folder).

Several recipes in this chapter illustrate how Gizmos can be useful. Often new GameObjects created from Editor Extensions will have helpful Gizmos assocaite with them.

Gizmos Unity manual entry:

```
- https://docs.unity3d.com/Manual/GizmosMenu.html
```

Introduction The first three recipes in this chapter provide some ideas for adding some extra features to your game (pausing, slow motion, and securing online games). The next two recipes then present ways to manage complexity in your games through managing states and their transitions. The rest of the recipes in this chapter provide examples of how to investigate and improve the efficiency and performance of your game. Each of these optimization recipes begins by stating an optimization principle that it embodies. The big picture Before getting on with the recipes, let's step back and think about the different parts of Unity games and how their construction and runtime behavior can impact on game performance. Games are made up of several different kinds of components: Audio assets 2D and 3D graphical assets Text and other file assets Scripts When a game is running, there are many competing processing requirements for your CPU and GPU, including: Audio processing Script processing 2D physics processing 3D physics processing Graphical rendering GPU processing One way to reduce the complexity of graphical computations and to improve frame rates is to use simpler models whenever possible—this is the reduction of the Level Of Detail (LOD). The general strategy is to identify situations where a simpler model will not degrade the user's experience. Typically, situations include where a model is only taking up a small part of the screen (so less detail in the model will not change what the user sees), when objects are moving very fast across the screen (so the user is unlikely to have time to notice less detail), or where we are sure the users' visual focus is elsewhere (for example, in a car racing game, the user is not looking at the quality of the trees but on the road ahead). We provide a LOD recipe, Improving performance with LOD groups, in this chapter. Unity's draw call batching may actually be more efficient than you or your team's 3D modelers are at reducing the triangle/vertex geometry. So, it may be that by manually simplifying a 3D model, you have removed Unity's opportunity to apply its highly effective vertex reduction algorithms; then, the geometric complexity may be larger for a small model than for a larger model, and so a smaller model may lead to a lower game performance! One recipe presents advice collected from several sources and the location of tools to assist in different strategies to try to reduce draw calls and improve graphical performance. We will present several recipes allowing you to analyze actual processing times and frame rates, so that you can collect data to confirm whether your design decisions are having the desired efficiency improvements. "You have a limited CPU budget and you have to live with it" Joachim Ante, Unite-07 At the end of the day, the best balance of heuristic strategies for your particular game project can only be discovered by an investment of time and hard work, and some form of profiling investigation. Certain strategies (such as caching to reduce component reflection lookups) should perhaps be standard practice in all projects, while other strategies may require tweaking for each unique game and level, to find which approaches work effectively to improve efficiency, frame rates, and, most importantly, the user experience when playing the game. "Premature Optimization is the root of all evil" Donald Knuth, "Structured Programming With Go To Statements". Computing Surveys, Vol 6, No 4, December 1974 Perhaps, the core strategy to take away from this chapter is that there are many parts of a game that are candidates for possible optimization, and you should drive the actual optimizations you finally implement for a particular game based on the evidence you gain by profiling its performance.

Player UI to changing the game's quality settings

In this recipe we show how the player can control the quality settings by providing UI Slider and reading the array of possible settings.

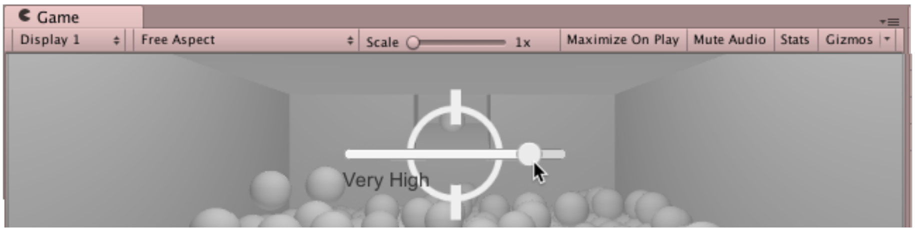


Figure 1: Insert Image B08775_16_02.png

Getting ready

For this recipe, we have prepared a package named BallGame containing two Scenes. The package is in the 16_01 folder.

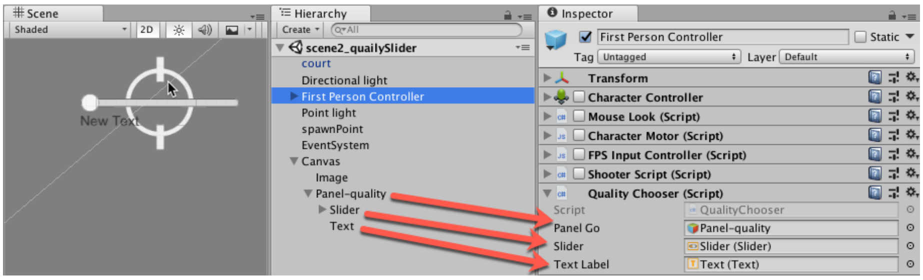
How to do it...

To create a player UI to changing the game's quality settings, do the following:

1. Create a new 3D project, and import the BallGame package.
2. Open the Scene named scene0_ballCourt.
3. In the Scene create a new UI Panel named Panel-quality, by choosing menu: Create | UI | Panel.
4. With GameObject Panel-quality selected in the Hierarchy, create a new UI Slider named Slider-quality, by choosing menu: Create | UI | Slider. This GameObject should be childed to GameObject Panel-quality.
5. With GameObject Panel-quality selected in the Hierarchy, create a new UI Text GameObject named Text-quality, by choosing menu: Create | UI | Text. This GameObject should be childed to GameObject Panel-quality. In the Inspector set its Transform Position Y value to -25.
6. Create a new C# script-class named QualityChooser, and attach an instance-object as a component to the First-Person Controller:

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

public class QualityChooser : MonoBehaviour {
    public GameObject panelGo;
    public Slider slider;
    public Text textLabel;

    void Awake () {
       slider.maxValue = QualitySettings.names.Length - 1;
        slider.value = QualitySettings.GetQualityLevel();
        SetQualitySliderActive(true);
    }

    public void SetQualitySliderActive(bool active) {
        Cursor.visible = active;
        panelGo.SetActive(active);
    }

    public void UpdateQuality(float sliderFloat) {
        int qualityInt = Mathf.RoundToInt (sliderFloat);
        QualitySettings.SetQualityLevel (qualityInt);
      textLabel.text = QualitySettings.names [qualityInt];
    }
}
```

7. In the Hierarchy select the First Person Controller. Then, from the Inspector, access the Quality Chooser component, populate the panelGo, Slide and Text Label public fields with UI ameObjects Panel-quality, Slider and Text.



☑ First Person Controller □ Static

Tag Untagged Layer Default

Script QualityChooser

Panel Go Panel-quality

Slider Slider (Slider)

Text Label Text (Text)

Figure 2: Insert Image B08775_16_03.png

8. From the Hierarchy panel, select QualitySlider. Then, from the Inspector panel, Slider component, find the list named On Value Changed (Single),

and click on the + sign to add a command.

9. Drag the First Person Controller from the Hierarchy into the Game Object field of the new command. Then, use the function selector to find the SetQuality function under Dynamic float (No Function > PauseGame | Dynamic float | SetQuality).

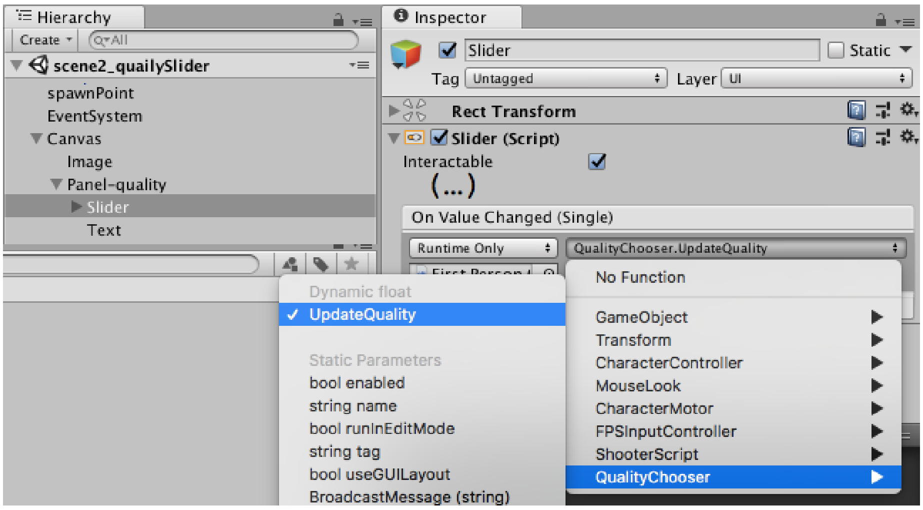


Figure 3: Insert Image B08775_16_04.png

10. When you play the Scene, you should be able to drag the quality slider to change the quality settings.

How it works...

You created a panel containing a UI Slider and a UI Text object.

Method SetQualitySliderActive(...) receives a true/false value, and uses this to show/hide the mouse cursor and the UI Panel.

Method UpdateQuality(...) receives a float value, from the Slider OnChange event. This value is converted to an integer, to then be the index of the selected Quality setting. This index is used both to select the project quality setting, and also to update the UI Text label with the name of the currently selected quality setting.

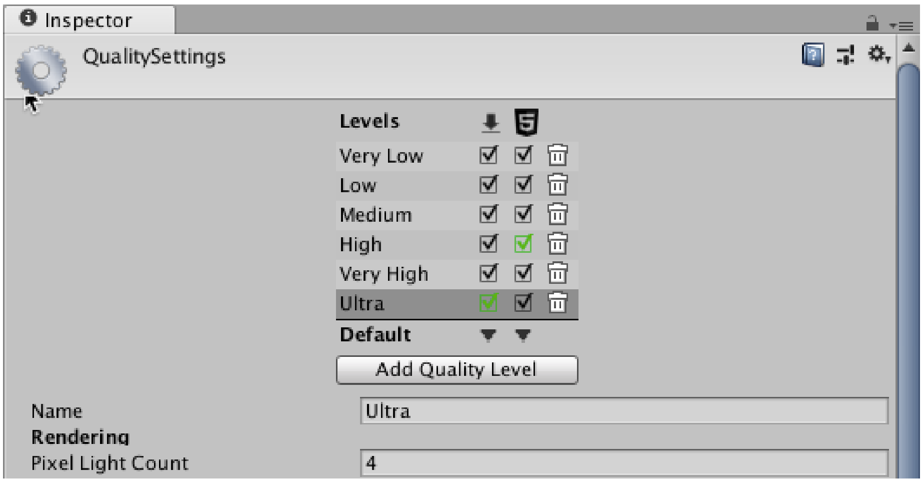
When the Scene begins, in method Awake(), the UI Slide has its maximum value set to 1 less then the number of project quality items (e.g. if 5 items, the slider will be from 0 .. 4). Also the slide is moved to the position corresponding to the current quality level, and method SetQualitySliderActive(...) is invokved with value true, to display both mouse pointer and the UI Panel showing the slide and text label. sdf

There's more...

Here are some ways to go further with this recipe.

See/Edit List of Quality Settings

You can view, and modify the quality settings avaible for a project. Choose menu: Edit | Project Settings | Quality.



Inspector

QualitySettings

Levels	⬇	5	
Very Low	☑	☑	🗑
Low	☑	☑	🗑
Medium	☑	☑	🗑
High	☑	☑	🗑
Very High	☑	☑	🗑
Ultra	☑	☑	🗑
Default	▼	▼	

Add Quality Level

Name: Ultra

Rendering

Pixel Light Count: 4

Figure 4: Insert Image B08775_16_01.png

Pausing the game

As compelling as your next game will be, you should always let players pause it for a short break. Sometimes a game pause is to let the player rest, another reason might be to change some game setting such as volume or graphics quality.

Pausing the game usually involves a combination of freezing game action, and also hiding / revealing UI items, to display a message to the player and provide UI controls to change settings.

In this recipe, we will implement a simple and effective pause screen, that hides the previous recipe's quality settings slide when the game is being played, and reveals it when the game has been paused.

Getting ready

This recipe builds on the previous one, so make a copy of that and use that copy.

How to do it...

To pause your game upon pressing the Esc key, follow these steps:

1. Select the First Person Controller, and in the Inspector enable the following components:
    - Character Controller
    - Mouse Look (Script)
    - Character Motor (Script)
    - FPS Input Controller (Script)
    - Shooter (Script)
2. Add the following C# script PauseGame to First Person Controller:

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

public class PauseGame : MonoBehaviour {
    private bool isPaused = false;
    private QualityChooser qualityChooser;

    void Start () {
        qualityChooser = GetComponent<QualityChooser>();
    }

    void Update () {
        if (Input.GetKeyDown(KeyCode.Escape)) {
            isPaused = !isPaused;
            SetPause ();
        }
    }

    private void SetPause() {
        float timeScale = !isPaused ? 1f : 0f;
        Time.timeScale = timeScale;
        GetComponent<MouseLook> ().enabled = !isPaused;
        qualityChooser.SetQualitySliderActive(isPaused);
        }
    }
```

3. Edit the QualityController script-class, and in the Awake() method change the last line to pass false (not true) to method SetQualitySliderActive(...):

```
void Awake () {
   slider.maxValue = QualitySettings.names.Length - 1;
     slider.value = QualitySettings.GetQualityLevel();
     SetQualitySliderActive(false);
}
```

4. When you play the scene, you should be able to pause/resume the game by pressing the Escape key, also displaying/hiding the slider that controls the game's quality settings.

How it works...

To pause a Unity game with a script, we need to do is set the game's Time Scale to 0 (and set it back to 1 to resume) . Method SetPause() does actions according to the value of the isPaused variable:

- isPause true: Time Scale 0 (pause the game), disable MouseLook component, and activate the quality slider and mouse cursor
- isPause false: Time Scale 1 (resume the game), enable MouseLook component, and de-activate the quality slider and mouse cursor

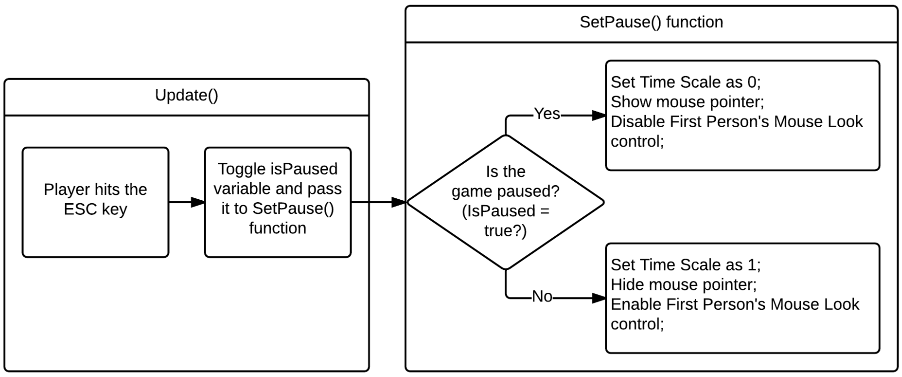

SetPause() function
Update()
Player hits the ESC key
Toggle isPaused variable and pass it to SetPause() function
Is the game paused? (IsPaused = true?)
Yes
No
Set Time Scale as 0;
Show mouse pointer;
Disable First Person's Mouse Look control;
Set Time Scale as 1;
Hide mouse pointer;
Enable First Person's Mouse Look control;

Figure 5: Insert Image B08775_16_06.png

In method Update(), each frame a test is made for whether the Escape key has been pressed. If pressed, the value of isPaused is toggled, and method SetPause() is invoked.

There's more...

Here are some ways to go further with this recipe.

Learning more about QualitySettings

Our code for changing quality settings is a slight modification of the example given by Unity's documentation. If you want to learn more about the subject, check out

- http://docs.unity3d.com/ScriptReference/QualitySettings.html.

Offer user futher game settings

You could add more UI panels to be activated when the game is paused, for sound volume controls, save/load buttons, and so on.

Implementing slow motion

Since Remedy Entertainment's Max Payne, slow motion, or bullet time, became a popular feature in games. For example, Criterion's Burnout series has successfully explored the slow motion effect in the racing genre. In this recipe, we will implement a slow motion effect triggered by the pressing of the mouse's right button.

Getting ready

For this recipe, we will use the same package as the previous recipes, BallGame in the 16_03 folder.

How to do it...

To implement slow motion, follow these steps:

1. Import the BallGame package into your project and, from the Project panel, open the level named scene1_ballGame.
2. Create C# script-class BulletTime, and add an instance-object as a component to the First Person Controller GameObject:

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

public class BulletTime : MonoBehaviour {
    public float sloSpeed = 0.1f;
    public float totalTime = 10f;
    public float recoveryRate = 0.5f;
    public Slider EnergyBar;
    private float elapsed = 0f;
    private bool isSlow = false;

    void Update (){
    if (Input.GetButtonDown ("Fire2") && elapsed < totalTime)
            SetSpeed (sloSpeed);

        if (Input.GetButtonUp ("Fire2"))
            SetSpeed (1f);

        if (isSlow) {
            elapsed += Time.deltaTime / sloSpeed;

            if (elapsed >= totalTime)
                SetSpeed (1f);
        } else {
            elapsed -= Time.deltaTime * recoveryRate;
           elapsed = Mathf.Clamp (elapsed, 0, totalTime);
        }

    float remainingTime = (totalTime - elapsed) / totalTime;
        EnergyBar.value = remainingTime;
    }

    private void SetSpeed (float speed) {
        Time.timeScale = speed;
        Time.fixedDeltaTime = 0.02f * speed;
        isSlow = !(speed >= 1.0f);
    }
}
```

3. Add a UI Slider to the Scene named Slider-energy, choosing menu: Create | UI | Slider. Please note that it will be created as a child of the existing Canvas object.

4. Select Slider-energy and, from the Rect Transform component in the Inspector, set its Anchors as follows:

- Min X: 0, Y: 1
- Max X: 0.5, Y: 1
- Pivot X: 0, Y: 1

5. Select Slider-energy and, from the Rect Transform component in the Inspector, set its position as follows:

   - Left: 0
   - Pos Y: 0
   - Pos Z: 0
   - Right: 0
   - Height: 50

6. In the Inspector, set as inactive the sliders child GameObject Handle Slide Area.

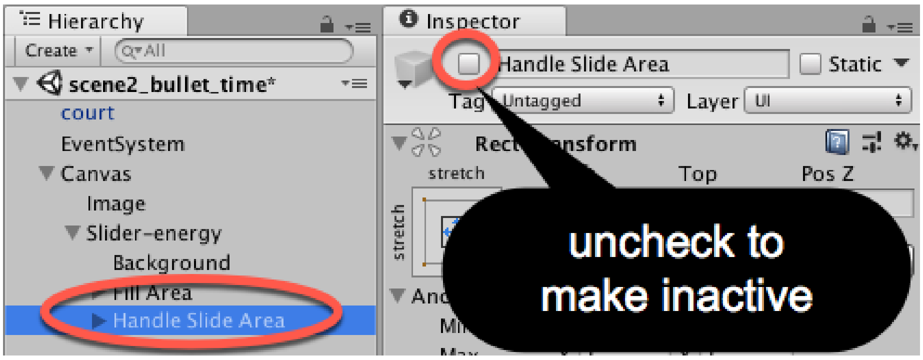


Figure 6: Insert Image B08775_16_07.png

7. Finally, select the First Person Controller from the Hierarchy, find the Bullet Time component, and drag GameObject Slider-energy from the Hierarchy into its Energy Bar slot.

8. Play your game. You should be able to activate slow motion by holding down the right mouse button (or whatever alternative you have set for Input axis Fire2). The slider will act as a progress bar that slowly shrinks, indicating the remaining bullet time you have.

How it works...

Basically, all we need to do to have the slow motion effect is decrease the Time.timeScale variable. In our script, we do that by using the sloSpeed variable. Please note that we also need to adjust the Time.fixedDeltaTime variable, updating the physics simulation of our game.

In order to make the experience more challenging, we have also implemented a sort of energy bar to indicate how much bullet time the player has left (the initial value is given, in seconds, by the totalTime variable). Whenever the player is not using bullet time, he has his quota filled according to the recoveryRate variable.

Regarding the UI slider, we have used the Rect Transform settings to place it on the top-left corner and set its dimensions to half of the screen's width and 50 pixels tall. Also, we have hidden the handle slide area to make it similar to a traditional energy bar. Finally, instead of allowing direct interaction from the player with the slider, we have used the BulletTime script to change the slider's value.

There's more...

Some suggestions for you to improve your slow motion effect even further are as follows.

Customizing the slider

Don't forget that you can personalize the slider's appearance by creating your own sprites, or even by changing the slider's fill color based on the slider's value.

Try adding the following lines of code to the end of the Update function:

```
 GameObject fill = GameObject.Find("Fill").gameObject;
Color sliderColor = Color.Lerp(Color.red, Color.green, remainingTime);
 fill.GetComponent<Image> ().color = sliderColor;
```

Adding Motion Blur

Motion Blur is an image effect frequently identified with slow motion. Once attached to the camera, it could be enabled or disabled depending on the speed float value. For more information on the Motion Blur post-processing image effect, refer to:

- https://github.com/Unity-Technologies/PostProcessing/wiki/Motion-Blur
- https://docs.unity3d.com/Packages/com.unity.postprocessing@2.0/manual/Motion-Blur.html
- https://docs.unity3d.com/Packages/com.unity.postprocessing@2.0/manual/Manipulating-the-Stack.html

Creating sonic ambience

Max Payne famously used a strong, heavy heartbeat sound as sonic ambience. You could also try lowering the sound effects volume to convey the character focus when in slow motion. Plus, using audio filters on the camera could be an interesting option.

Destroying objects at death a time

Optimization principal 1: Minimize the number of active and enabled objects in a scene.

One way to reduce the number of active objects is to destroy objects when they are no longer needed. As soon as an object is no longer needed, we should destroy it; this saves both memory and processing resources since Unity no longer needs to send the object such messages as Update() and FixedUpdate(), or consider object collisions or physics and so on.

However, there may be times when we wish not to destroy an object immediately, but at some known point in the future. Examples might include after a sound has finished playing (see that recipe Waiting for audio to finish before auto-destructing object in Chapter 9, Playing and Manipulating Sounds), the player only has a certain time to collect a bonus object before it disappears, or perhaps an object displaying a message to the player should disappear after a certain time.

This recipe demonstrates how objects can be told to start dying, and then to automatically destroy them after a given delay has passed.

How to do it...

To destroy objects after a specified time, follow these steps:

1. Create a new 2D project.
2. Create a UI Button named Click Me, and make it stretch to fill the entire window.
3. In the Inspector, set the Button’s Text child to have left-aligned and large text.
4. Add the following script class DeathTimeExample.cs to Button Click Me:

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

public class DeathTimeExample : MonoBehaviour {
  public void BUTTON_ACTION_StartDying() {
    deathTime = Time.time + deathDelay;
  }

  public float deathDelay = 4f;
  private float deathTime = -1;

  public Text buttonText;

  void Update(){
    if(deathTime > 0){
      UpdateTimeDisplay();
      CheckDeath();
    }
  }

  private void UpdateTimeDisplay(){
    float timeLeft = deathTime - Time.time;
    string timeMessage = "time left: " + timeLeft;
    buttonText.text = timeMessage;
  }

  private void CheckDeath(){
    if(Time.time > deathTime) Destroy( gameObject );
  }
}
```

5. Drag the Text child of Button Click Me into the script's public variable Button Text, so this script is able to change the button text to show the countdown.

6. With Button Click Me selected in the Hierarchy, add a new On Click() event for this button, dragging the button itself as the target GameObject and selecting public function BUTTON_ACTION_StartDying(),as shown in the following screenshot:

7. Now, run the scene; once the button is clicked, the button's text should show the countdown. Once the countdown gets to zero, Button Click Me will be destroyed (including all its children, in this case, just the GameObject Text).

How it works...

The float variable deathDelay stores the number of seconds the object waits before destroying itself once the decision has been made for the object to start dying. The float variable deathTime either has a value of -1 (no death time yet set) or it is a non-negative value, which is the time we wish the object to destroy itself.

When the button is clicked, the BUTTON_ACTION_StartDying() method is called. This method sets this deathTime variable to the current time plus whatever value is set in deathDelay. This new value for deathTime will be a positive number, meaning the IF-statement in the Update() method will fire from this point onward.

Every frame method Update() checks if deathTime is greater than zero (that is, a death time has been set), and, if so, it then calls, the UpdateTimeDisplay() and CheckDeath() methods.

The UpdateTimeDisplay() methods creates a string message stating how many seconds are left and updates the Button Text to show this message.

The CheckDeath() method tests whether the current time has passed the deathTime. If the death time has passed, then the parent gameObject is immediately destroyed.

When you run the scene, you'll see the Button removed from the Hierarchy once its death time has been reached.