11

Improving Games With Extra Features and Optimization

In this chapter, we will cover:

- Pausing the game
- Implementing slow motion
- Preventing your game from running on unknown servers
- State-driven behavior Do-It-Yourself states
- State-driven behavior with the State Design Pattern
- Reducing the number of objects by destroying objects at a “death” time
- Reducing the number of enabled objects by disabling objects whenever possible
- Reducing the number of active objects by making objects inactive whenever possible
- Improving efficiency with delegates and events (and avoiding SendMessage!)
- Executing methods regularly but independent of frame rate with coroutines
- Spreading long computations over several frames with coroutines
- Evaluating performance by measuring max and min frame rates (FPS)
- Identifying performance “bottlenecks” with the Performance Profiler
- Identifying performance “bottlenecks” with Do-It-Yourself Performance Profiling
- Cache GameObject and Component References to Avoid Expensive Lookups
- Improving performance with LOD Groups

- Improving performance through reduced draw-calls by designing for draw call "batching"

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 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), or 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 in this chapter: *Improving performance with LOD Groups*.

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, and then the geometric complexity, may be larger for a small model, than for a larger model – so a smaller model may lead to 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 present several recipes allowing you to analyze actual processing times and frame rates, so 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 that you should drive the actual optimizations you finally implement for a particular game based on the evidence you gain by profiling its performance.

Pausing the game

As compelling as your next game will be, you should always let players pause it for a short break. In this recipe, we will implement a simple and effective pause screen including controls for changing the display's quality settings.

Getting ready

For this recipe, we have prepared a package named `BallGame` containing a playable scene. The package is in the folder `1362_11_01`.

How to do it...

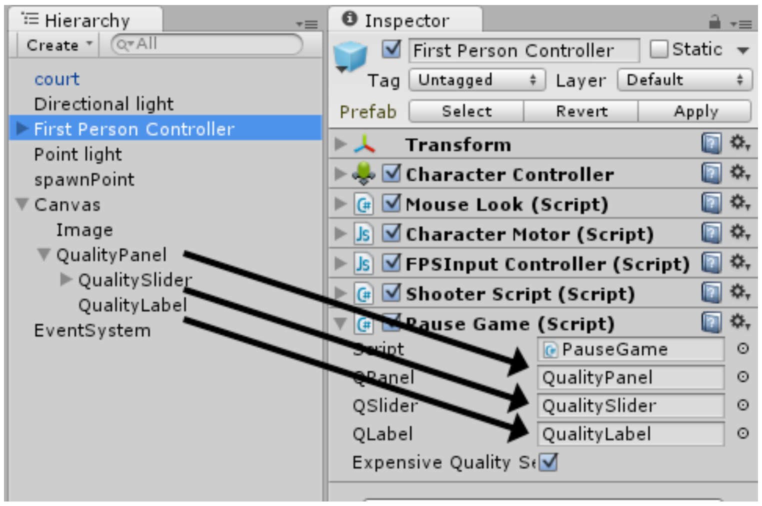
To pause your game upon pressing the *Esc* key, follow these steps:

1. Import the package `BallGame` into your project and, from the **Project** view, open the level named `BallGame_01`.
2. In the Inspector create a new tag '**Ball'**, apply this tag to prefab `ball` in folder `Prefabs`, and save the scene.
3. From the **Hierarchy** view, use the **Create** dropdown menu to add a **Panel** to the UI (**Create | UI | Panel**). Note that it will automatically add it to the current **Canvas** in the scene. Rename the panel `QualityPanel`.
4. Now use the **Create** dropdown menu to add a **Slider** to the UI (**Create | UI | Slider**). Rename it `QualitySlider`.
5. Finally, use the **Create** dropdown menu to add a **Text** to the UI (**Create | UI | Text**). Rename it `QualityLabel`. Also, from the **Inspector** view, **Rect Transform**, change its **Pos Y** to **-25**.
6. Add the following C# Script **PauseGame** to **First Person Controller**:

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

public class PauseGame : MonoBehaviour {
   public GameObject qPanel;
   public GameObject qSlider;
   public GameObject qLabel;
   public bool expensiveQualitySettings = true;
   private bool isPaused = false;

   void Start () {
      Cursor.visible = isPaused;
      Slider slider = qSlider.GetComponent<Slider> ();
```

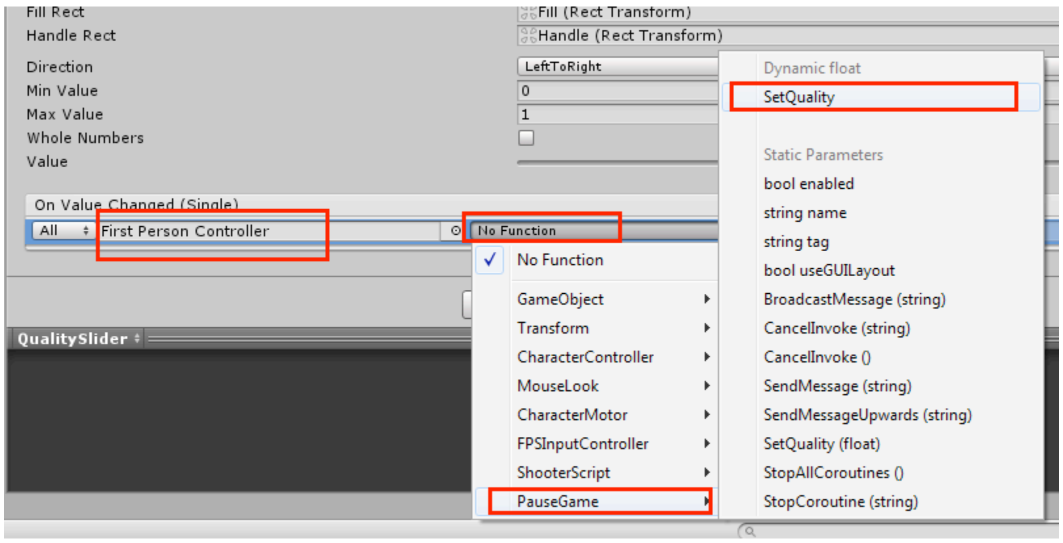
```
        slider.maxValue = QualitySettings.names.Length;
        slider.value = QualitySettings.GetQualityLevel ();
        qPanel.SetActive(false);
    }

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

    private void SetPause(){
        float timeScale = !isPaused ? 1f : 0f;
        Time.timeScale = timeScale;
        Cursor.visible = isPaused;
        GetComponent<MouseLook> ().enabled = !isPaused;
        qPanel.SetActive (isPaused);
    }

    public void SetQuality(float qs){
        int qsi = Mathf.RoundToInt (qs);
        QualitySettings.SetQualityLevel (qsi);
        Text label = qLabel.GetComponent<Text> ();
        label.text = QualitySettings.names [qsi];
    }
}
```

7. From the **Hierarchy** view, select the **First Person Controller**. Then, from the **Inspector**, access the **Pause Game** component and populate the **QPanel**, **QSlider**, and **QLabel** fields with the game objects **QualityPanel**, **QualitySlider**, and **QualityLabel**, respectively.



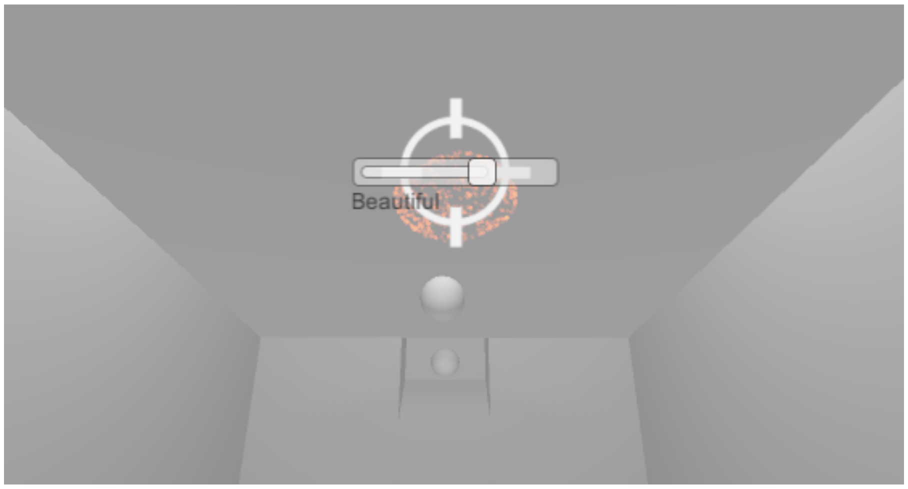
Insert image 1362OT_11_01.png

8. From the **Hierarchy** view, select **QualitySlider**. Then, from the **Inspector** view, **Slider** component, find the list named **On Value Changed (Single)** and click the **+** sign to add a command.
9. Drag the **First Person Controller** from the **Hierarchy** view 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**).



Insert image 1362OT_11_02.png

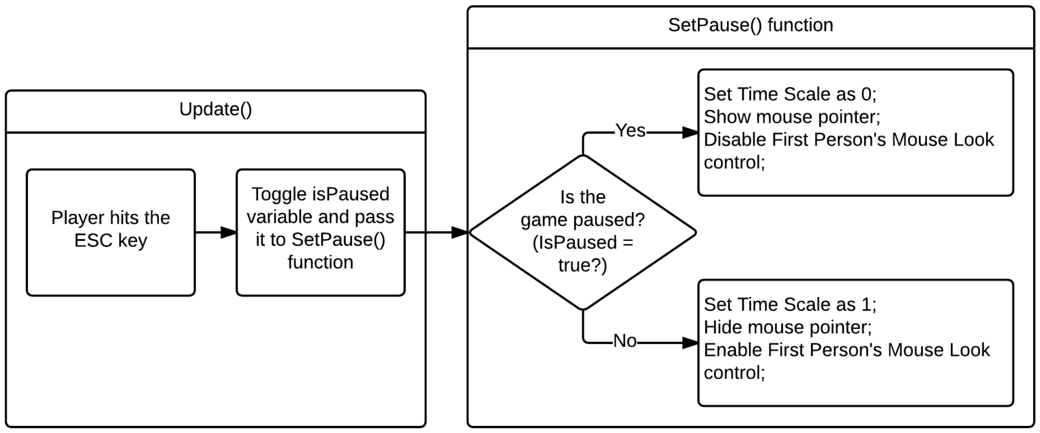
10. When you play the scene, you should be able to pause / resume the game by pressing the *Esc* key, also activating a slider that controls the game's quality settings.



Insert image 1362OT_11_03.png

How it works...

Pausing the game is actually an easy, straightforward task in Unity: all we need to do is set the game's **Time Scale** to 0 (and set it back to 1 to resume). In our code, we have included such command within the `SetPause()` function, which is called whenever the player presses the *Esc* key, also toggling the `isPaused` variable. To make things more functional, we have included a **GUI panel** featuring a **Quality Settings** slider that is activated whenever the game is paused.



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

Insert image 1362OT_11_04.png

Regarding the behavior for the Quality Settings slider and text, their parameters are adjusted at the start based on the game's variety of quality settings, their names and its current state. Then, changes in the slider's value redefine the quality settings, also updating the label text accordingly.

There's more...

You can always add more functionality to the Pause screen by displaying sound volume controls, save/load buttons, and so on.

Learning more about Quality Settings

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 it out at: `docs.unity3d.com/Documentation/ScriptReference/QualitySettings.html`.

See also

Refer to the next recipe in this chapter for more information.

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 recipe: `BallGame` in the folder `1362_11_02`.

How to do it...

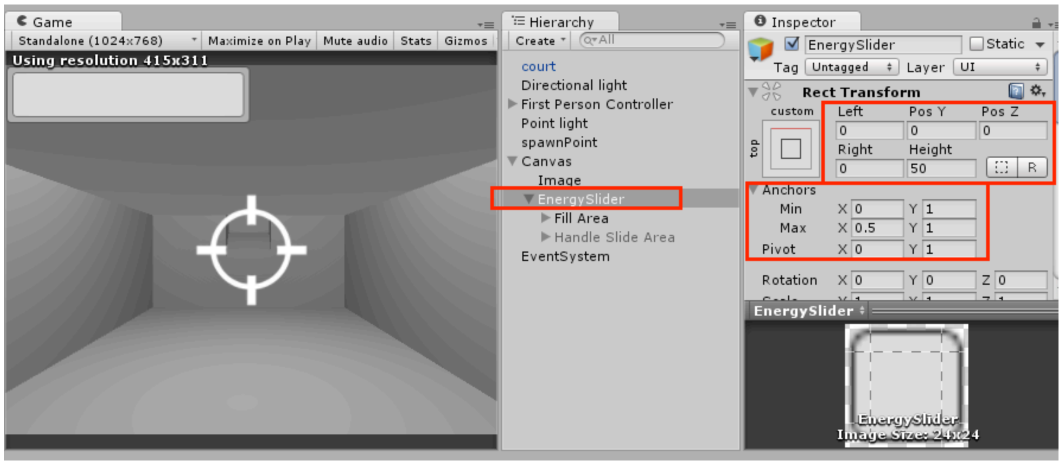
To implement slow-motion, follow these steps:

1. Import the package `BallGame` into your project and, from the **Project** view, open the level named `BallGame_01`.
2. In the **Inspector**, create a new tag **Ball**, apply this tag to prefab `ball` in folder `Prefabs`, and save the scene.
3. Add the following C# Script **BulletTime** to **First Person Controller**:

```
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);
        }
}
```

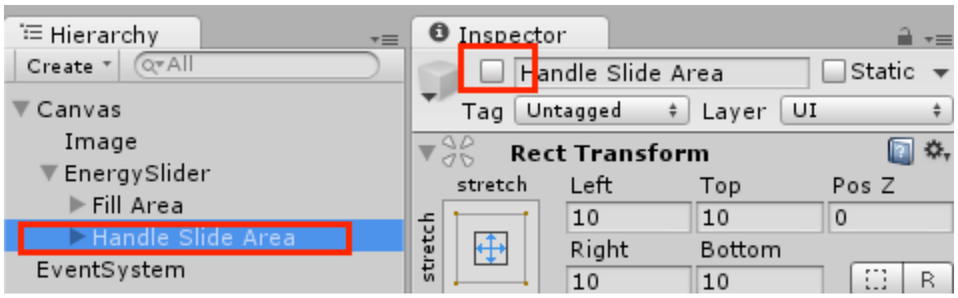
4. From the **Hierarchy** view, use the **Create** dropdown menu to add a **Slider** to the UI (**Create | UI | Slider**). Please note that it will be created as a child of the preexisting **Canvas** object. Rename it `EnergySlider`.
5. Select **EnergySlider** and, from the **Inspector** view, **Rect Transform** component, set its position as follows: **Left: 0**; **Pos Y: 0**; **Pos Z: 0**; **Right: 0**; **Height: 50**. Then, expand the **Anchors** settings and change it to: **Min X: 0**; **Y: 1**; **Max X: 0.5**; **Y: 1**; **Pivot X: 0**; **Y: 1**.



EnergySlider
Left 0 | Pos Y 0 | Pos Z 0
Right 0 | Height 50
Anchors
Min X 0 Y 1
Max X 0.5 Y 1
Pivot X 0 Y 1
Rotation X 0 Y 0 Z 0

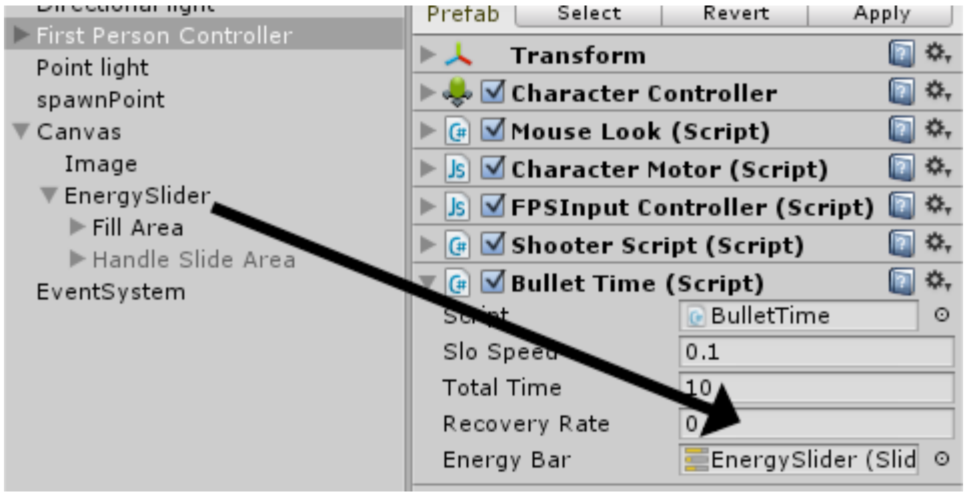
Insert image 1362OT_11_05.png

6. Also, select the **Handle Slide Area** child and disable it from the **Inspector** view.



Insert image 1362OT_11_06.png

7. Finally, select the **First Person Controller** from the **Hierarchy** view; find the **Bullet Time** component, and drag the **EnergySlider** from the **Hierarchy** view into its **Energy Bar** slot.



Insert image 1362OT_11_07.png

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 variable `Time.timeScale`. In our script, we do that by using the `sloSpeed` variable. Please note that we also need to adjust `Time.fixedDeltaTime`, 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 variable `totalTime`. Whenever the player is not using bullet time, he has his quota filled according to the `recoveryRate` variable.

Regarding the **GUI 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 more 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:

Customizing the Slider

Don't forget that you can personalize the slider's appearance by creating your own sprites, or even change 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 image effect, access: `docs.unity3d.com/Manual/script-MotionBlur.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.

See also

Refer to the recipe *Pausing the game* in this chapter for more information.

Preventing your game from running on unknown servers

After all the hard work you've had to go through to complete your web game project, it wouldn't be fair if it ended up generating traffic and income on someone else's website. In this recipe, we will create a script that prevents the main game menu from showing up unless it's hosted by an authorized server.

Getting ready

To test this recipe, you will need access to a provider where you can host the game.

How to do it...

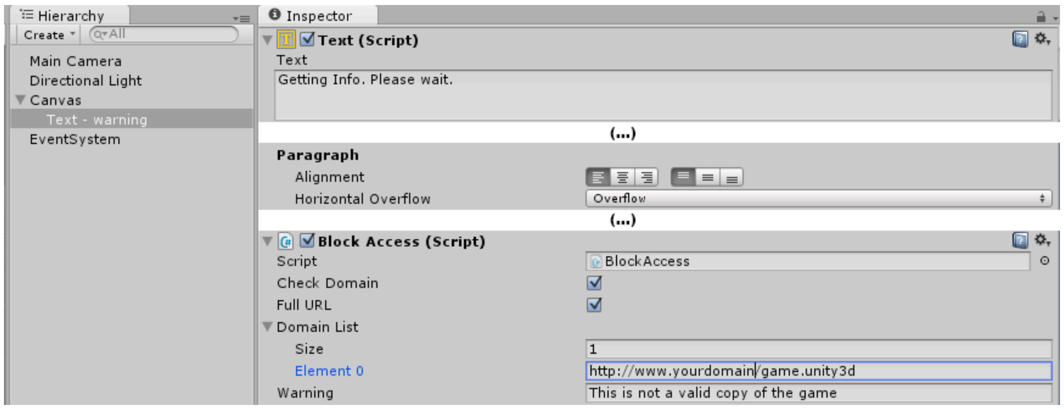
To prevent your web game from being pirated, follow these steps:

1. From the **Hierarchy** view, use the **Create** dropdown menu to create a **UI Text** GameObject (**Create | UI | Text**). Name it `Text – warning`. Then, from the **Text** component in the **Inspector**, change its **text** field to `Getting Info. Please wait`.
2. Add the following *C# Script* to the **Text – warning** Game Object:

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

public class BlockAccess : MonoBehaviour {
   public bool checkDomain = true;
   public bool fullURL = true;
   public string[] domainList;
   public string warning;


   private void Start(){
      Text scoreText = GetComponent<Text>();
      bool illegalCopy = true;

      if (Application.isEditor)
         illegalCopy = false;
      if (Application.isWebPlayer && checkDomain){
         for (int i = 0; i < domainList.Length; i++){
            if (Application.absoluteURL == domainList[i]){
               illegalCopy = false;
            }else if 
(Application.absoluteURL.Contains(domainList[i]) && 
!fullURL){
               illegalCopy = false;
            }
         }
      }

      if (illegalCopy)
         scoreText.text = warning;
      else
         Application.LoadLevel(Application.loadedLevel + 
1);
   }
}
```

3. From the **Inspector** view, leave the options **Check Domain** and **Full URL** checked; Increase **Size** of **Domain List** to 1 and fill out **Element 0** with the complete URL for your game. Type in the sentence `This is not a valid copy of the game` in the **Message** field. You might have to change the paragraph's **Horizontal Overflow** to **Overflow**.

> NOTE: Remember to include the Unity3d file name and extension in the URL - and not the HTML where it is embedded.



Main Camera
Directional Light
Canvas
Text - warning
EventSystem

Text (Script)
Text: Getting Info. Please wait.
(...)
Paragraph
Alignment
Horizontal Overflow: Overflow
(...)
Block Access (Script)
Script: BlockAccess
Check Domain: ✓
Full URL: ✓
Domain List
Size: 1
Element 0: http://www.yourdomain/game.unity3d
Warning: This is not a valid copy of the game

Insert image 1362OT_11_08.png

4. Save your scene as `menu`.
5. Create a new scene and change its **Main Camera** background color to black. Save this scene as `nextLevel`.
6. Let's build the game. Go to the menu **File | Build Settings…**, include the scenes **menu** and **nextLevel**, in that order, in the build list (**Scenes** in **Build**). Also, select **Web Player** as your platform and click **Build**.

How it works...

As soon as the scene starts, the script compares the actual URL of the `.unity3d` file to the ones listed in the `Block Access` component. If they don't match, the next level in the build is not loaded and a message appears on the screen. If they do match, the line of code `Application.LoadLevel(Application.loadedLevel + 1)` will load the next scene from the build list.

There's more...

Here is some information on how to fine tune and customize this recipe.

Improving security by using full URLs in your Domain List

Your game will be more secure if you fill out the Domain List with complete URLs (such as `http://www.myDomain.com/unitygame/game.unity3d`). In fact, it's recommended that you leave the option **Full URL** selected, so your game won't be stolen and published under a URL such as `www.stolenGames.com/yourgame.html?www.myDomain.com`.

Allowing redistribution with more domains

If you want your game to run from several different domains, increase **Size** and fill out more URLs. Also, you can leave your game completely free of protection by leaving the option **Check Domain** unchecked.

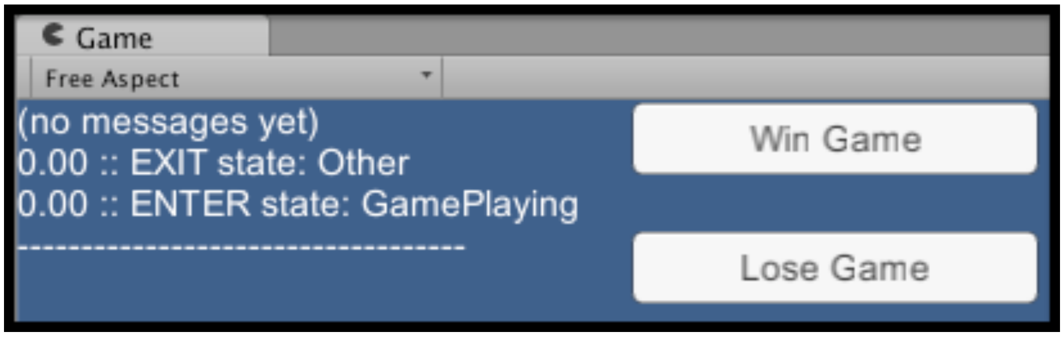
State-driven behavior Do-It-Yourself states

Games as a whole, and individual objects or characters, can often be thought of (or modeled as) passing through different *states* or *modes*. Modeling states and changes of state (due to *events* or game conditions) is a very common way to manage the complexity of games, and game components. In this recipe, we create a simple 3-state game (game playing / game won / game lost), using a single `GameManager` class.

How to do it...

To use states to manage object behavior follow these steps:

1. Create 2 UI Buttons at the top-middle of the screen. Name one **Button-win**, and edit its text to read **Win Game**. Name the second **Button-lose**, and edit its text to read **Lose Game**.
2. Create a UI Text object at the top-left of the screen. Name this **Text-state-messages,** and set its **Rect Transform** height property to **300**, and its **Text (Script) Paragraph Vertical Overflow** property to **Overflow**.



```
(no messages yet)
0.00 :: EXIT state: Other
0.00 :: ENTER state: GamePlaying
------------------------------------
```

Insert image 1362OT_08_38.png

3. Add the following C# script class GameManager to the Main Camera:

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

public class GameManager : MonoBehaviour {
   public Text textStateMessages;
   public Button buttonWinGame;
   public Button buttonLoseGame;

   private enum GameStateType {
      Other,
      GamePlaying,
      GameWon,
      GameLost,
   }

   private GameStateType currentState =
GameStateType.Other;
   private float timeGamePlayingStarted;
```

```
    private float timeToPressAButton = 5;

    void Start () {
        NewGameState( GameStateType.GamePlaying );
    }

    private void NewGameState(GameStateType newState) {
        // (1) state EXIT actions
        OnMyStateExit(currentState);

        // (2) change current state
        currentState = newState;

        // (3) state ENTER actions
        OnMyStateEnter(currentState);

        PostMessageDivider();
    }

    public void PostMessageDivider(){
        string newLine = "\n";
        string divider = "--------------------------------";
        textStateMessages.text += newLine + divider;
    }

    public void PostMessage(string message){
        string newLine = "\n";
        string timeTo2DecimalPlaces =
String.Format("{0:0.00}", Time.time);
        textStateMessages.text += newLine +
timeTo2DecimalPlaces + " :: " + message;
    }

    public void BUTTON_CLICK_ACTION_WIN_GAME(){
        string message = "Win Game BUTTON clicked";
        PostMessage(message);
        NewGameState( GameStateType.GameWon );
    }

    public void BUTTON_CLICK_ACTION_LOSE_GAME(){
        string message = "Lose Game BUTTON clicked";
        PostMessage(message);
        NewGameState( GameStateType.GameLost );
    }
```

```
   private void DestroyButtons(){
      Destroy (buttonWinGame.gameObject);
      Destroy (buttonLoseGame.gameObject);
   }

   //--------- OnMyStateEnter[ S ] - state specific actions
   private void OnMyStateEnter(GameStateType state){
      string enterMessage = "ENTER state: " +
state.ToString();
      PostMessage(enterMessage);

      switch (state){
      case GameStateType.GamePlaying:
         OnMyStateEnterGamePlaying();
         break;
      case GameStateType.GameWon:
         // do nothing
         break;
      case GameStateType.GameLost:
         // do nothing
         break;
      }
   }

   private void OnMyStateEnterGamePlaying(){
      // record time we enter state
      timeGamePlayingStarted = Time.time;
   }

   //--------- OnMyStateExit[ S ] - state specific actions
   private void OnMyStateExit(GameStateType state){
      string exitMessage = "EXIT state: " +
state.ToString();
      PostMessage(exitMessage);

      switch (state){
      case GameStateType.GamePlaying:
         OnMyStateExitGamePlaying();
         break;
      case GameStateType.GameWon:
         // do nothing
         break;
      case GameStateType.GameLost:
         // do nothing
         break;
```

```
        case GameStateType.Other:
            // cope with game starting in state 'Other'
            // do nothing
            break;
        }
    }

    private void OnMyStateExitGamePlaying(){
        // if leaving gamePlaying state then destroy the 2
buttons
        DestroyButtons();
    }

    //--------- Update[ S ] - state specific actions
    void Update () {
        switch (currentState){
        case GameStateType.GamePlaying:
            UpdateStateGamePlaying();
            break;
        case GameStateType.GameWon:
            // do nothing
            break;
        case GameStateType.GameLost:
            // do nothing
            break;
        }
    }

    private void UpdateStateGamePlaying(){
        float timeSinceGamePlayingStarted = Time.time -
timeGamePlayingStarted;
        if(timeSinceGamePlayingStarted > timeToPressAButton){
            string message = "User waited too long -
automatically going to Game LOST state";
            PostMessage(message);
            NewGameState(GameStateType.GameLost);
        }
    }
}
```

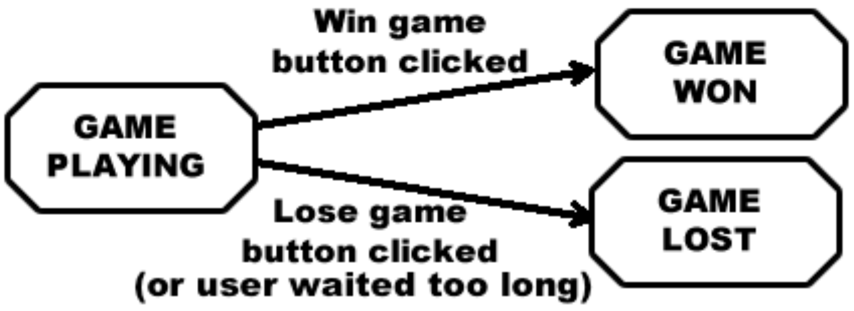
4. In the **Hierarchy** select button **Button-win**, and for its Button (Script) component add an OnClick action to call method `BUTTON_CLICK_ACTION_WIN_GAME()` from the **GameManager** component in the **Main Camera** gameObject.

5. In the **Hierarchy** select button **Button-lose**, and for its Button (Script) component add an OnClick action to call method `BUTTON_CLICK_ACTION_LOSE_GAME()` from the **GameManager** component in the **Main Camera** gameObject.
6. In the **Hierarchy** select gameObject **Main Camera**. Next drag into the **Inspector** ensure all 3 **GameManager (Script)** public variables **Text State Messages**, **Button Win Game and Button Lose Game**, have the corresponding Canvas gameObjects dragged into them (the 2 buttons and the UI Text gameObject).

How it works...

As can be seen in the state chart figure, this recipe models a simple game which starts in the **GAME PLAYING** state, then depending on the button clicked by the user, the game moves either into the **GAME WON** state or the **GAME LOST** state. Also, if the user waits too long to click a button, then the game moves into the **GAME LOST** state.

The possible states of the system are defined using the enumerated type `GameStateType`; and the current state of the system at any point in time is stored in variable `currentState`.



Insert image 1362OT_08_39.png

A fourth state is defined (`Other`), to allow us to explicitly set the desired `GamePlaying` state in our `Start()` method. When we wish the game state to be changed, we call method `NewGameState(…)` passing the new state the game is to change into. Method `NewGameState(…)` first calls method `OnMyStateExit(...)` with the current state, since there may be actions to be performed when a particular state is exited; for example when the `GamePlaying` state is exited, it Destroys the two buttons. Next method `NewGameState(…)`sets variable `currentState` to be assigned the new state. Next method `OnMyStateEnter(…)` is called, since there may be action to be performed immediately when a new state is entered. Finally, a message divider is posted to the UI Text box, with a call to method `PostMessageDivider()`.

When the `GameManager` object receives messages (for example, every frame for `Update()`) its behavior must be appropriate for the current state. So we see in this method a *switch* statement, which calls state-specific methods. For example, if the current state is `GamePlaying`, then when an `Update()` message is received, the method `UpdateStateGamePlaying()` will be called.

Methods `BUTTON_CLICK_ACTION_WIN_GAME()` and `BUTTON_CLICK_ACTION_LOSE_GAME()` are executed if their corresponding buttons have been clicked. They move the game into the corresponding WIN or LOSE state.

Logic has been written in method `UpdateStateGamePlaying()` so once the `GameManager` has been in the state `GamePlaying` for more than a certain time (defined in variable `timeToPressAButton`), the game will automatically change into the `GameLost` state.

So for each state we may need to write methods for state exit, state entry, and update events; and also a main method for each event with a `Switch` statement to determine which state method should be called (or not). As can be imagined, the size of our methods and the number of methods, in our `GameManager` class will grow significantly with as more states, and more complex game logic is needed for non-trivial games. The next recipe takes a more sophisticated approach to state-driven games, where each state has its own class.

See also

Refer to the next recipe in this chapter for more information on how to manage the complexity of states with class inheritance and the state Design Pattern.

State-driven behavior using the Sate Design Pattern

The previous pattern not only illustrated the usefulness of modeling game states, but also how a game manager class can grow in size and become unmanageable. To manage the complexity of many states and complex behaviors of states, the state pattern has been proposed in the software development community. Design patterns are general purpose software component architectures that have been tried and tested and found to be good solutions to commonly occurring software system features. The key features of the state pattern are that each state is modeled by its own class, and all states inherit (are sub-classed) from a single parent state class. The states need to know about each other in order to tell the game manager to change the current state. This is a small price to pay for the division of the complexity of the overall game behaviors into separate state classes.

Getting ready

This recipe builds upon the previous recipe. So make a copy of that project, open it and then follow the steps for this recipe.

How to do it...

To manage an object's behavior using the state pattern architecture, perform the following steps:

1. Replace the contents of C# script class `GameManager` with the following:

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

public class GameManager : MonoBehaviour {
   public Text textGameStateName;
   public Button buttonWinGame;
   public Button buttonLoseGame;

   public StateGamePlaying stateGamePlaying{get; set;}
   public StateGameWon stateGameWon{get; set;}
   public StateGameLost stateGameLost{get; set;}

   private GameState currentState;

   private void Awake () {
      stateGamePlaying = new StateGamePlaying(this);
      stateGameWon = new StateGameWon(this);
      stateGameLost = new StateGameLost(this);
   }

   private void Start () {
      NewGameState( stateGamePlaying );
   }

   private void Update () {
      if (currentState != null)
         currentState.StateUpdate();
   }

   public void NewGameState(GameState newState)
   {
      if( null != currentState)
         currentState.OnMyStateExit();
```

```
        currentState = newState;
        currentState.OnMyStateEntered();
    }

    public void DisplayStateEnteredMessage(string
stateEnteredMessage){
        textGameStateName.text = stateEnteredMessage;
    }

    public void BUTTON_CLICK_ACTION_WIN_GAME(){
        if( null != currentState){

    currentState.OnButtonClick(GameState.ButtonType.ButtonWi
nGame);
            DestroyButtons();
        }
    }

    public void BUTTON_CLICK_ACTION_LOSE_GAME(){
        if( null != currentState){

    currentState.OnButtonClick(GameState.ButtonType.ButtonLo
seGame);
            DestroyButtons();
        }
    }

    private void DestroyButtons(){
        Destroy (buttonWinGame.gameObject);
        Destroy (buttonLoseGame.gameObject);
    }
}
```

2. Create a new C# script class called `GameState`:

```
using UnityEngine;
using System.Collections;

public abstract class GameState {
    public enum ButtonType {
        ButtonWinGame,
        ButtonLoseGame
    }

    protected GameManager gameManager;
    public GameState(GameManager manager) {
        gameManager = manager;
    }

    public abstract void OnMyStateEntered();
    public abstract void OnMyStateExit();
    public abstract void StateUpdate();
    public abstract void OnButtonClick(ButtonType button);
}
```

3. Create a new C# script class called `StateGamePlaying`:

```
using UnityEngine;
using System.Collections;

public class StateGamePlaying : GameState {
    public StateGamePlaying(GameManager
manager):base(manager){}

    public override void OnMyStateEntered(){
        string stateEnteredMessage = "ENTER state:
StateGamePlaying";

    gameManager.DisplayStateEnteredMessage(stateEnteredMessa
ge);
        Debug.Log(stateEnteredMessage);
    }
    public override void OnMyStateExit(){}
    public override void StateUpdate() {}


    public override void OnButtonClick(ButtonType button){
        if( ButtonType.ButtonWinGame == button )

    gameManager.NewGameState(gameManager.stateGameWon);

        if( ButtonType.ButtonLoseGame == button )

    gameManager.NewGameState(gameManager.stateGameLost);
    }
}
```

4. Create a new C# script class called `StateGameWon`:

```
using UnityEngine;
using System.Collections;

public class StateGameWon : GameState {
    public StateGameWon(GameManager manager):base(manager){}
```

```
   public override void OnMyStateEntered(){
      string stateEnteredMessage = "ENTER state:
StateGameWon";

   gameManager.DisplayStateEnteredMessage(stateEnteredMessa
ge);
      Debug.Log(stateEnteredMessage);
   }
   public override void OnMyStateExit(){}
   public override void StateUpdate() {}
   public override void OnButtonClick(ButtonType button){}
}
```

5. Create a new C# script class called `StateGameLost`:

```
using UnityEngine;
using System.Collections;

public class StateGameLost : GameState {
   public StateGameLost(GameManager
manager):base(manager){}

   public override void OnMyStateEntered(){
      string stateEnteredMessage = "ENTER state:
StateGameLost";

   gameManager.DisplayStateEnteredMessage(stateEnteredMessa
ge);
      Debug.Log(stateEnteredMessage);
   }
   public override void OnMyStateExit(){}
   public override void StateUpdate() {}
   public override void OnButtonClick(ButtonType button){}
}
```

7. In the **Hierarchy** select button **Button-win**, and for its Button (Script) component add an OnClick action to call method `BUTTON_CLICK_ACTION_WIN_GAME()` from the **GameManager** component in the **Main Camera** gameObject.
8. In the **Hierarchy** select button **Button-lose**, and for its Button (Script) component add an OnClick action to call method `BUTTON_CLICK_ACTION_LOSE_GAME()` from the **GameManager** component in the **Main Camera** gameObject.
9. In the **Hierarchy** select gameObject **Main Camera**. Next drag into the **Inspector** ensure all 3 **GameManager (Script)** public variables **Text State Messages**, **Button Win Game and Button Lose Game**, have the corresponding

Canvas gameObjects dragged into them (the 2 buttons and the UI Text gameObject).

How it works...

The scene is very straightforward for this recipe. There is the single GameObject **Main Camera** which has the `GameManager` script object component attached to it.

A C# scripted class is defined for each state that the game needs to manage, for this example the three states `StateGamePlaying`, `StateGameWon`, and `StateGameLost`. Each of these state classes is a subclass of `GameState`. `GameState` defines properties and methods that all subclass states will possess:

- An enumerated type `ButtonType`, which defines the two possible button clicks that the game might generate: `ButtonWinGame` and `ButtonLoseGame`.
- The variable `gameManager`: so each state object has a link to the game manager.
- The constructor method that accepts a reference to the `GameManager`: that automatically makes variable `gameManager` refer to the passed in `GameManager` object.
- The four abstract methods `OnMyStateEntered()`, `OnMyStateExit()`, `OnButtonClick(…)`, and `StateUpdate()`. Note that abstract methods must have their own implementation for each subclass.

When the `GameManager` class' `Awake()` method is executed, three state objects are created, one for each of the playing/win/lose classes. These state objects are stored in their corresponding variables: `stateGamePlaying`, `stateGameWon`, and `stateGameLost`.

The `GameManager` class has a variable called `currentState`, which is a reference to the current state object at any time while the game runs (initially, it will be `null`). Since it is of the `GameState` class (the parent of all state classes), it can refer to any of the different state objects.

After `Awake()`, `GameManager` will receive a `Start()` message. This method initializes the `currentState` to be the `stateGamePlaying` object.

For each frame, the `GameManager` will receive `Update()`messages. Upon receiving these messages, `GameManager` sends a `StateUpdate()`messages to the `currentState` object. So for each frame the object for the current state of the game will execute those methods. For example, when the `currentState` is set to game playing, for each frame the `gamePlayingObject` will calls its (in this case, empty) `StateUpdate()` method.

Class StateGamePlaying implements statements in its `OnButtonClick()` method, so that when the user clicks on a button, the `gamePlayingObject` will call the `GameManager` instance's `NewState()` method, passing it the object corresponding to the new state. So if the user clicks on **Button-win**, the `NewState()` method is passed `gameManager.stateGameWon`.

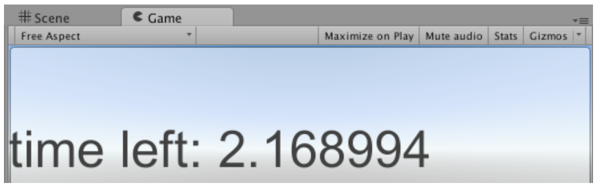
Reducing the number of objects by destroying objects at a "death" time

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 also processing resources, since Unity no longer needs to send the object `Update()` and `FixedUpdate()` messages and so on, or consider collisions or physics for such objects and so on.

However, there may be times when we wish to not destroy an object immediately, but at some known point in time in the future. Examples might include after a sound has finished playing (see recipe *Waiting for audio to finish before auto-destructing Object* in *Chapter 9*, *Playing and Manipulating Sounds*); or perhaps 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.



time left: 2.168994

Insert image 1362OT_11_14.png

How to do it...

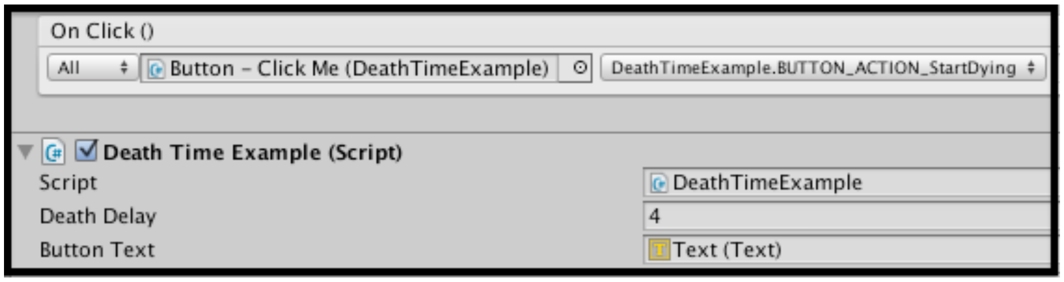
To destroy objects after a specified time, follow these steps:

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 pubic function `BUTTON_ACTION_StartDying()`.


On Click ()

All | Button - Click Me (DeathTimeExample) | DeathTimeExample.BUTTON_ACTION_StartDying

Death Time Example (Script)

Script	DeathTimeExample
Death Delay	4
Button Text	Text (Text)

Insert image 1362OT_11_15.png

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 set yet), or it is a non-negative value, which is the time we wish the object to destroy itself.

When the button is clicked, `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 method `Update()`will fire from this point onwards.

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

Method `UpdateTimeDisplay()` creates a string message stating how many seconds are left, and updates the **Button Text** to show this message.

Method `CheckDeath()` 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.

See also

Refer to the next recipe in this chapter for more information.

Reducing the number of enabled objects by disabling objects whenever possible

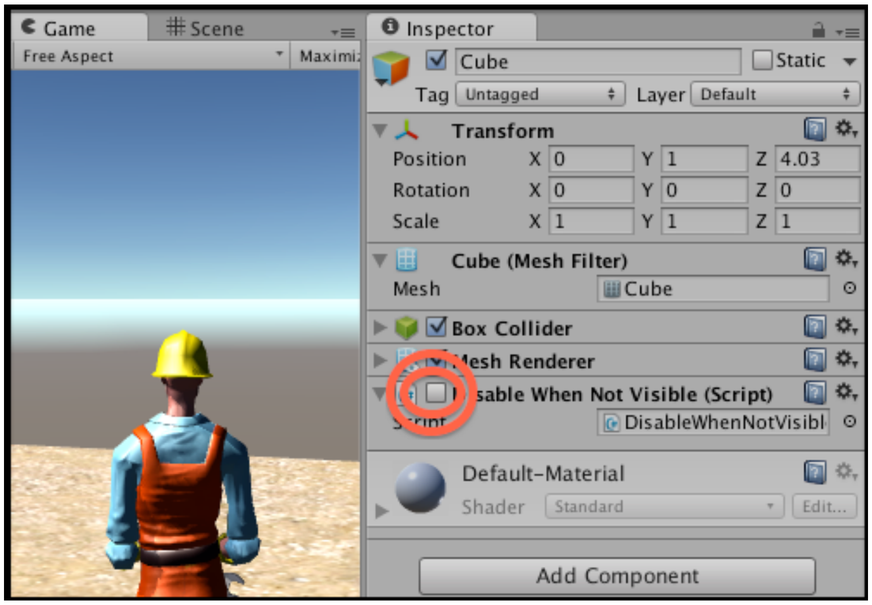
Optimization principal 1: Minimize the number of active and enabled objects in a scene.

Sometimes, we may not want to completely remove an object, but we can identify times when a scripted component of an object can be safely disabled. If a `MonoBehaviour` script is disabled, then Unity no longer needs to send the object messages such as `Update()`and `FixedUpdate()`each frame.

For example, if a **Non-Player Character** (**NPC**) should only demonstrate some behavior when the player can see that character, then we only need to be executing the behavior logic when the NPC is visible - the rest of the time we can safely disable the scripted component.

Unity provides the very useful events `OnBecameInvisible()` and `OnBecameVisible()`, which inform an object when it moves out of and into the visible area for one or more cameras in the scene.

This recipe illustrates the following rule of thumb: if an object has no reason to be doing actions when it cannot be seen, then we should disable that object while it cannot be seen.



Cube ☐ Static
Tag Untagged Layer Default

Transform

Position	X	0	Y	1	Z	4.03
Rotation	X	0	Y	0	Z	0
Scale	X	1	Y	1	Z	1

Cube (Mesh Filter)
Mesh Cube
✓ Box Collider
✓ Mesh Renderer
☐ Disable When Not Visible (Script)
Script DisableWhenNotVisibl
Default-Material
Shader Standard Edit...

Insert image 1362OT_11_16.png

How to do it...

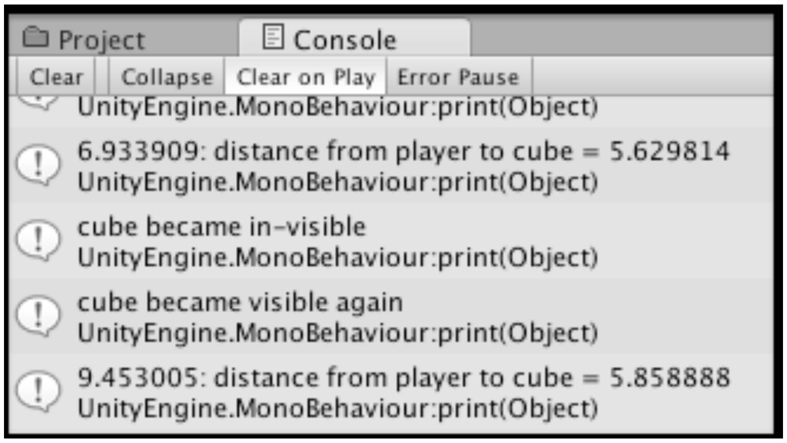
To disable objects to reduce computer processing workload requirements, follow these steps:

1. Create a new Unity project, importing the **Character Controller** and **Terrain Assets** packages.
2. Create a new **Terrain** (size **20 x 20**, located at **-10, 0, -10**) and texture-paint it with **GoodDirt** (which you'll find in folder **Standard Assets** from your import of the **Terrain Assets** package).
3. Add a **3rdPersonController** at (**0, 1, 0**).
4. Create a new **Cube** just in front of your **3rdPersonController** (so it is visible in the Game panel when you start running the game).
5. Add the following C# script class `DisableWhenNotVisible` to your **Cube**:

```
using UnityEngine;
using System.Collections;

public class DisableWhenNotVisible : MonoBehaviour {
   private GameObject player;

   void Start(){
      player = GameObject.FindGameObjectWithTag("Player");
   }

   void OnBecameVisible() {
      enabled = true;
      print ("cube became visible again");
   }

   void OnBecameInvisible() {
      enabled = false;
      print ("cube became invisible");
   }

   void Update(){
      //do something, so we know when this script is NOT
doing something!
      float d = Vector3.Distance( transform.position,
player.transform.position);
      print(Time.time + ": distance from player to cube = "
+ d);
   }
}
```

How it works...

When visible, the scripted `DisableWhenNotVisible` component of the Cube recalculates and displays the distance from itself to the **3rdPersonController** object's transform, via the variable `player` in method `Update()` for each frame. However, when this object receives the message `OnBecameInvisible()`, the object sets its `enabled` property to `false`. This results in Unity no longer sending `Update()`messages to the `GameObject`, so the distance calculation in `Update()` is no longer performed, thus reducing the game's processing workload. Upon receiving the message `OnBecameVisible()`, the `enabled` property is set back to `true`, and the object will then receive `Update()` messages each frame. Note that you can see the scripted component become disabled by seeing the blue *tick* in its **Inspector** checkbox disappear, if you have the **Cube** selected in the **Hierarchy** when running the game.



```
UnityEngine.MonoBehaviour:print(Object)
6.933909: distance from player to cube = 5.629814
UnityEngine.MonoBehaviour:print(Object)
cube became in-visible
UnityEngine.MonoBehaviour:print(Object)
cube became visible again
UnityEngine.MonoBehaviour:print(Object)
9.453005: distance from player to cube = 5.858888
UnityEngine.MonoBehaviour:print(Object)
```

Insert image 1362OT_11_17.png

The screenshot shows our **Console** text output, logging how the user must have turned away from the cube at 6.9 seconds after starting the game (and so the cube was no longer visible), and then at 9.4 seconds, the user turned so they could see the cube again, causing it to be re-enabled.

There's more...

Some details you don't want to miss:

Note - viewable in Scene panel still counts as visible!

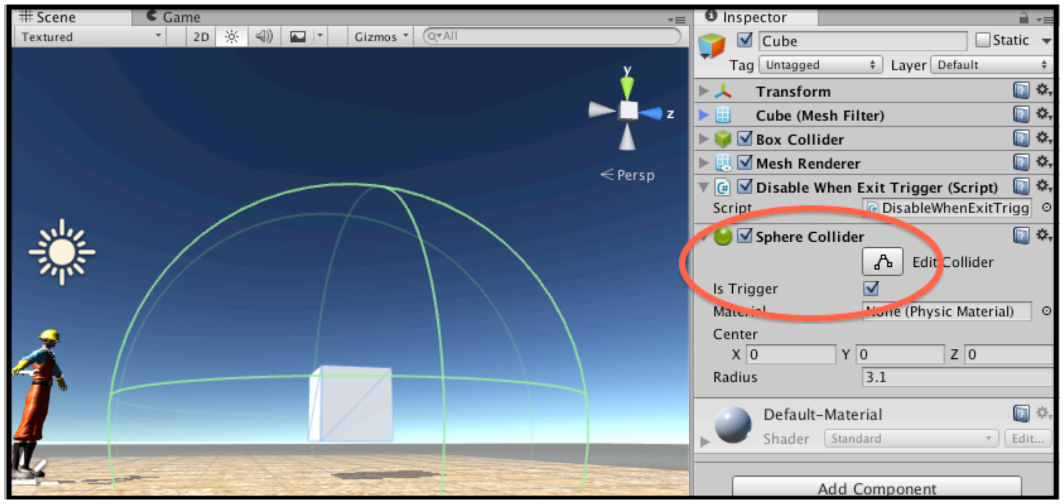
Note that even if the Game panel is not showing (rendering) an object, if the object is visible in a Scene panel, then it will still be considered visible. Therefore it is recommended that you hide / close the Scene panel when testing this recipe, otherwise it may be that the object does only become non-visible when the game stops running.

Another common case - only enable after OnTrigger()

Another common situation is that we only want a scripted component to be active if the player's character is nearby (within some minimum distance). In these situations, a sphere collider (with **Is Trigger** checked) can be setup on the object to be disabled/enabled (continuing our example, this would be on our **Cube**), and the scripted component can be enabled only when the player's character enters that sphere. This can be implemented by replacing the `OnBecameInvisible()` and `OnBecameVisible()` methods with `OnTriggerEnter()` and `OnTriggerExit()` methods as follows:

```
void OnTriggerEnter(Collider hitObjectCollider) {
   if (hitObjectCollider.CompareTag("Player")){
      print ("cube close to Player again");
      enabled = true;
   }
}

void OnTriggerExit(Collider hitObjectCollider) {
   if (hitObjectCollider.CompareTag("Player")){
      print ("cube away from Player");
      enabled = false;
   }
}
```

This screenshot illustrates a large sphere collider having been created around the cube, with its **Trigger** enabled:



Cube ✓ Static ☐
Tag Untagged Layer Default
Script DisableWhenExitTrigg
Is Trigger ✓
Material None (Physic Material)
Center X 0 Y 0 Z 0
Radius 3.1
Shader Standard

Insert image 1362OT_11_18.png

Many computer games (such as Half Life) use environmental design, such as corridors, to optimize memory usage by loading and unloading different parts of the environment. For example, when a player hits a corridor trigger, environment objects load and unload. See the following for more information about such techniques:

- `http://gamearchitect.net/Articles/StreamingBestiary.html`
- `http://cie.acm.org/articles/level-design-optimization-guidelines-for-game-artists-using-the-epic-games/`
- `http://gamedev.stackexchange.com/questions/33016/how-does-3d-games-work-so-fluent-provided-that-each-meshs-size-is-so-big`

See also

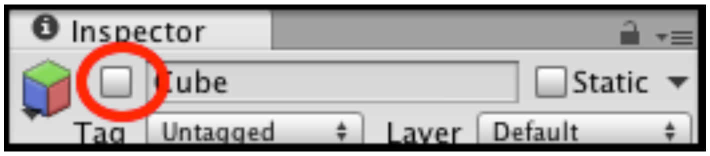
Refer to the following recipes in this chapter for more information:

- *Reducing the number of objects by destroying objects at a "death" time*
- *Reducing the number of active objects by making objects inactive whenever possible*

Reducing the number of active objects by making objects inactive whenever possible

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

Sometimes we may not want to completely remove an object, but it is possible to go one step further than disabling a scripted component, by making the parent `GameObject` that contains the scripted component `inactive`. This is just like deselecting the checkbox next to the `GameObject` in the **Inspector**.



Insert image 1362OT_11_19.png

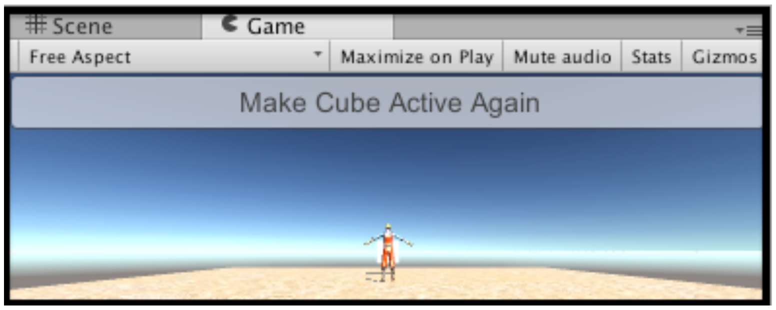
How to do it...

To reduce computer processing *workload* requirements by making an object inactive when it becomes invisible, follow these steps:

1. Copy the previous recipe.
2. Remove scripted component `DisableWhenNotVisible` from your **Cube,** and instead add the following C# script class `InactiveWhenNotVisible` to **Cube**:

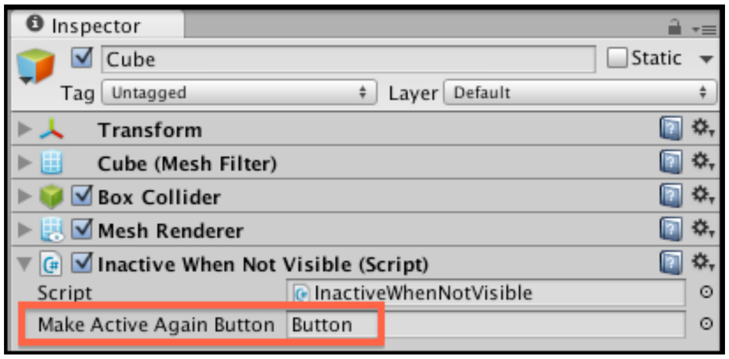
```
using UnityEngine;
using System.Collections;
using UnityEngine.UI;

public class InactiveWhenNotVisible : MonoBehaviour {
   // button action
   public void BUTTON_ACTION_MakeActive(){
      gameObject.SetActive(true);
      makeActiveAgainButton.SetActive(false);
   }

   public GameObject makeActiveAgainButton;

   private GameObject player;

   void Start(){
      player = GameObject.FindGameObjectWithTag("Player");
   }

   void OnBecameInvisible() {
      makeActiveAgainButton.SetActive(true);
      print ("cube became invisible");
      gameObject.SetActive(false);
   }

   void Update(){
      float d = Vector3.Distance( transform.position,
player.transform.position);
      print(Time.time + ": distance from player to cube = "
+ d);
   }
}
```

3. Create a new **Button**, containing the text `Make Cube Active Again`, and position the button so that it is at the top of the Game panel, and stretches the entire width of the Game panel.



Insert image 1362OT_11_20.png

4. With the **Button** selected in the **Hierarchy**, add a new **On Click()** event for this button, dragging the **Cube** as the target GameObject, and selecting pubic function `BUTTON_ACTION_makeCubeActiveAgain()`.
5. Uncheck the active checkbox next to the **Button** name in the **Inspector** (in other words, manually de-activate this **Button** - so we don't see the **Button** when the scene first runs).
6. Select the **Cube** in the **Inspector** and drag the **Button** into the `MakeActiveAgainButton` variable slot of its script class `InactiveWhenNotVisible` component.



Script: InactiveWhenNotVisible
Make Active Again Button: Button

Insert image 1362OT_11_21.png

How it works...

Initially the Cube is visible, and the Button is inactive (so not visible to the user). When the Cube receives an `OnBecameInvisible` event message, its `OnBecameInvisible()` method will execute. This method performs two actions:

- It first enables (and therefore makes visible) the `Button`

- It then makes inactive the script's parent `gameObject` (that is, the `Cube` GameObject)

When the Button is clicked, it makes the `Cube` object active again, and makes the `Button` inactive again. So at any one time only one of the objects `Cube` and `Button` are active, and each makes itself inactive when it makes the other one active.

Note that an inactive `GameObject` does not receive ANY messages, so it will not receive the `OnBecameVisible()` message; and this may not be appropriate for every object that is out of sight of the camera. However, when deactivating objects is appropriate, a larger performance saving is made, over simply disabling a single scripted `Monobehaviour` component of a `GameObject`.

The only way to re-activate an inactive object is for another object to set the `GameObject` component's active property back to `true`. In this recipe, it is the `Button` GameObject which when clicked runs method `BUTTON_ACTION_makeCubeActiveAgain()`, which allows our game to make the `Cube` active again.

See also

Refer to the following recipes in this chapter for more information:

- *Reducing the number of objects by destroying objects at a "death" time*
- *Reducing the number of enabled objects by disabling objects whenever possible*

Improving efficiency with Delegates and Events (and avoiding SendMessage!)

Optimization principal 2: Minimize actions requiring Unity to perform "reflection" over objects and searching of all current scene objects.

When events can be based on visibility, distance, or collisions, we can use such events as `OnTriggerExit` and `OnBecomeInvisible` as described in the previous recipes. When events can be based on time periods, we can use coroutines as described in other recipes in this chapter. However, some kinds of events are unique to each game situation, and C# offers several methods of broadcasting user-defined event messages to scripted objects. One approach is the `SendMessage(…)` method, which when sent to a GameObject will check every `Monobehaviour` scripted component and execute the named method if its parameters match. However, this involves an inefficient technique known as **reflection**. C# offers another event message approach known as **Delegates and Events**, which we describe and implement in this recipe. Delegates and events work in a similar way to `SendMessage(…)`, but are much more efficient, since Unity maintains a defined list of which objects are *listening* to the broadcast events. `SendMessage(…)` should be avoided if performance is important, since it means that Unity has to analyze each scripted object (*reflect over* the object) to see whether there is a public method corresponding to the message that has been sent - this is much slower than using delegates and events.

Delegates and events implement the **publish-subscribe (or pubsub) design pattern**. This is also known as the **Observer** design pattern. Objects can subscribe one of their methods to receive a particular type of event message from a particular publisher. In this recipe, we'll have a manager class which will publish new events when UI buttons are clicked. We'll create some UI objects, some of which **subscribe** to the color change events, so each time a color change event is published, subscribed UI objects receive the event message and change their color accordingly. C# publisher objects don't have to worry about how many objects subscribe to them at any point in time (it could be none, or 1000!), this is known as **loose coupling**, since it allows different code components to be written (and maintained) independently, and this is a desirable feature of object-oriented code.

How to do it...

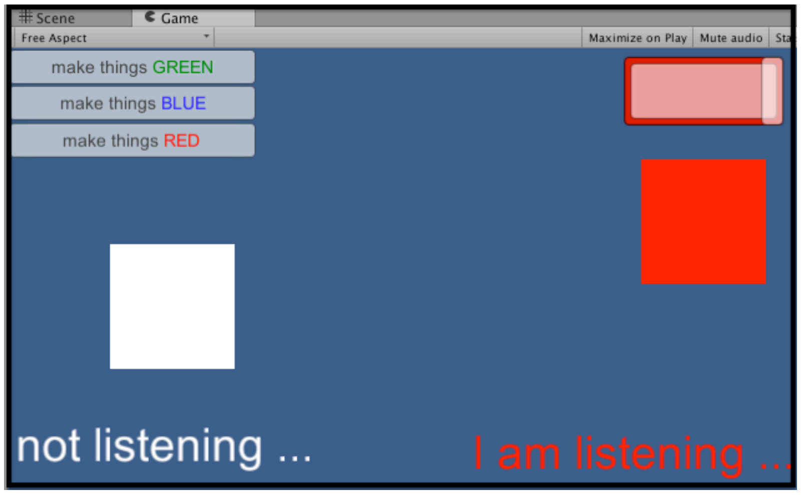
To implement delegates and events, follow these steps:

1. Create a new 2D project.
2. Add the following C# script class **ColorManager** to the **Main Camera**:

```
using UnityEngine;
using System.Collections;

public class ColorManager : MonoBehaviour {
   public void BUTTON_ACTION_make_green(){
      PublishColorEvent(Color.green);
   }

   public void BUTTON_ACTION_make_blue(){
      PublishColorEvent(Color.blue);
   }

   public void BUTTON_ACTION_make_red(){
      PublishColorEvent(Color.red);
   }

   public delegate void ColorChangeHandler(Color newColor);
   public static event ColorChangeHandler onChangeColor;

   private void PublishColorEvent(Color newColor){
      // if there is at least one listener to this delegate
      if(onChangeColor != null){
```

```
                // broadcast change color event
                onChangeColor(newColor);
            }
        }
    }
```

3. Create 2 UI **Image** objects, and 2 UI text objects. Position one **Image** and **Text** object to the lower left of the screen, and position the other to the lower right of the screen. Make the text on the lower left read **Not listening**, and make the text on the right of the screen read **I am listening**. For good measure, add a **Slider** UI object in the top right of the screen.
4. Create 3 UI Buttons in the top left of the screen, named **Button-GREEN**, **Button-BLUE**, and **Button-RED**, with corresponding text reading `make things <color=green>GREEN</color>`, `make things <color=blue>BLUE</color>`, and `make things <color=red>RED</color>`.



Insert image 1362OT_11_22.png

5. Attach the following C# script class `ColorChangeListenerImage` to both the lower right **Image**, and also the **Slider**:

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

    public class ColorChangeListenerImage : MonoBehaviour {
        void OnEnable() {
            ColorManager.onChangeColor += ChangeColorEvent;
        }
        private void OnDisable(){
            ColorManager.onChangeColor -= ChangeColorEvent;
        }

        void ChangeColorEvent(Color newColor){
                GetComponent<Image>().color = newColor;
        }
    }
```

6. Attach the following C# script class `ColorChangeListenerText` to the **I am listening Text** UI object:

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

    public class ColorChangeListenerText : MonoBehaviour {
        void OnEnable() {
            ColorManager.onChangeColor += ChangeColorEvent;
        }

        private void OnDisable(){
            ColorManager.onChangeColor -= ChangeColorEvent;
        }

        void ChangeColorEvent(Color newColor){
            GetComponent<Text>().color = newColor;
        }
    }
```

7. With **Button-GREEN** selected in the **Hierarchy**, add a new **On Click()** event for this button, dragging the **Main Camera** as the target GameObject, and selecting pubic function `BUTTON_ACTION_make_green()`. Do the same for the BLUE and RED buttons with functions `BUTTON_ACTION_make_blue()` and `BUTTON_ACTION_make_red()` respectively.
8. Run the game, when you click a change color button, the 3 UI objects on the right of the screen show all changes to the corresponding color, while the 2 UI objects at the bottom left of the screen remain in the default **White** color.

How it works...

First, let's consider what we want to happen - we want the right hand Image, Slider, and Text objects to change their color when they receive an event message `OnChangeColor()` with a new color argument.

This is achieved by each object having an instance of the appropriate `ColorChangeListener` class, that subscribes their `OnChangeColor()` methods to listen for color change events published from the `ColorManager` class. Since both the Image and Slider objects have an Image component whose color will change, they have scripted components of our C# class `ColorChangeListenerImage`, while the Text object needs a different class since it is the color of the Text component whose color is to be changed (so we add an instance of C# scripted component `ColorChangeListenerText` to the Text UI object). So as we can see, different objects may respond to receiving the same event messages in ways appropriate to each different object.

Since our scripted objects may be disabled and enabled at different times, each time a scripted `ColorChangeListener` object is enabled (such as when its GameObject parent is instantiated), its `OnChangeColor()` method is added (`+=`) to the list of those subscribed to listen for color change events; and each time `ColorChangeListenerImage/Text` objects are disabled, those methods are removed (`-=`) from the list of event subscribers.

When a `ColorChangeListenerImage/Text` object receives a color change message, its subscribed `OnChangeColor()` method is executed, and the color of the appropriate component is changed to the received `Color` value (green/red/blue).

The `ColorManager` has a public class (static) variable `changeColorEvent`, which defines an **event** to which Unity maintains a dynamic list of all the subscribed object methods. It is to this event that `ColorChangeListenerImage/Text` objects register or deregister their methods.

The `ColorManager` class displays three buttons to the user to make things green, red, and blue. When a button is clicked, the `changeColorEvent` is told to publish a new event, passing a corresponding `Color` argument to all subscribed object methods.

The `ColorManager` class declares a **delegate** named `ColorChangeHandler`. Delegates define the return type (in this case, `void`) and argument *signature* of methods that can be delegated (subscribed) to an event. In this case, methods must have the argument signature of a single parameter of type `Color`. Our `OnChangeColor()` method in classes `ColorChangeListenerImage/Text` match this argument signature, and so are permitted to subscribe to the `changeColorEvent` in class `ColorManager`.

NOTE: An easy to understand video about Unity delegates and events can be found at: `http://www.youtube.com/watch?v=N2zdwKIsXJs`.

See also

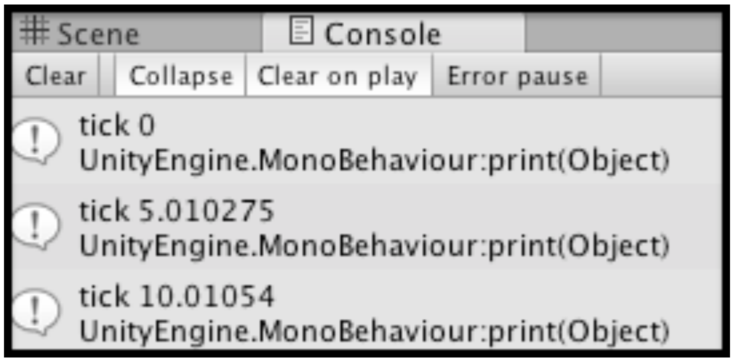
Refer to the following recipe in this chapter for more information:

- *Caching rather than component lookups and 'reflection' over objects*

Executing methods regularly but independent of frame rate with coroutines

Optimization principal 3: Call methods as few times as possible. While it is very simple to put logic into `Update()` and have it regularly executed each frame, we can improve game performance by executing logic as occasionally as possible. So if we can get away with only checking for some situation every 5 seconds, then great performance savings can be made to move that logic out of `Update()`.

A coroutine is a function that can suspend its execution until a `yield` action has completed. One kind of yield action simply waits for a given number of seconds. In this recipe, we use coroutines and yield to show how a method can be only executed every 5 seconds - this could be useful for non-player characters to decide whether they should randomly *wake up*, or perhaps choose a new location to start moving towards.



```
tick 0
UnityEngine.MonoBehaviour:print(Object)
tick 5.010275
UnityEngine.MonoBehaviour:print(Object)
tick 10.01054
UnityEngine.MonoBehaviour:print(Object)
```

Insert image 1362OT_11_23.png

How to do it...

To implement methods at regular intervals independent of the frame rate, follow these steps:

1. Add the following C# script class `TimedMethod` to the **Main Camera**:

```
using UnityEngine;
using System.Collections;

public class TimedMethod : MonoBehaviour {
   private void Start() {
      StartCoroutine(Tick());
```

```
        }

        private IEnumerator Tick() {
            float delaySeconds = 5.0F;
            while (true) {
                print("tick " + Time.time);
                yield return new WaitForSeconds(delaySeconds);
            }
        }
    }
```

How it works...

When the `Start()` message is received, the `Tick()` method is started as a coroutine. Method `Tick()` sets the delay between executions (variable `delaySeconds`) to 5 seconds. An infinite loop is then started, where the method does its actions (in this case, just printing out the time); finally, a `yield` instruction is executed, which causes the method to suspend execution for the given delay of 5 seconds. After the yield instruction has completed, the loop will continue executing once again, and so on. What is important to understand when working with coroutines is that the method will **resume executing** from the same state it yielded.

You may have noticed: **there are no `Update()` or `FixedUpdate()` methods at all.** So although our game has logic being regularly executed, in this example there is no logic that has to be executed every frame - fantastic!

There's more...

Some details you don't want to miss:

Have different actions happening at different intervals

Coroutines can be used to have different kinds of logic being executed at different regular intervals. So logic that needs frame-by-frame execution goes into `Update()` and logic that works fine once or twice a second might go into a coroutine with a 0.5 second delay; logic that can get away with less occasional updating can go into another coroutine with a 2 or 5 second delay and so on. Effective and noticeable performance improvements can be found by carefully analyzing (and testing) different game logic to identify the **least frequent execution** that is still acceptable.

See also

Refer to the next recipe in this chapter for more information.

Spreading long computations over several frames with coroutines

Optimization principal 3: Call methods as few times as possible.

Coroutines allow us to write asynchronous code - we can ask a method to go off and calculate something, but the rest of the game can keep on running without having to wait for that calculation to end. Or we can call a coroutine method each frame from `Update()`, and organize the method to complete part of a complex calculation each time it is called.

Note that coroutines are not **threads**, but they are very handy in that each can progress further each frame and it allows us to write code that does not have to wait for certain methods to complete before some other can begin.

When games start requiring complex computations, such as for artificial intelligence reasoning, it may not be possible to maintain acceptable game performance when trying to complete all calculations in a single frame - this is where coroutines can be an excellent solution.

This recipe illustrates how a complex calculation can be structured into several pieces, each to be completed one frame at a time.

> NOTE: An excellent description of coroutines (and other Unity topics) can be found on Ray Pendergraph's wikidot website:
>
> `http://raypendergraph.wikidot.com/unity-developer-s-notes#toc6`

How to do it...

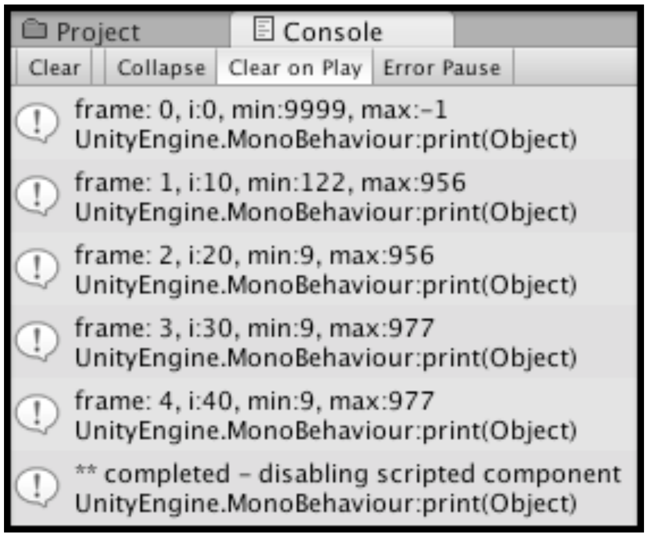
To spread computations over several frames, follow these steps:

1. Add the following script class `SegmentedCalculation.cs` to the **Main Camera**:

```
using UnityEngine;
using System.Collections;

public class SegmentedCalculation : MonoBehaviour {
   private const int ARRAY_SIZE = 50;
   private const int SEGMENT_SIZE = 10;
   private int[] randomNumbers;

   private void Awake(){
      randomNumbers = new int[ARRAY_SIZE];
```

```
        for(int i=0; i<ARRAY_SIZE; i++){
            randomNumbers[i] = Random.Range(0, 1000);
        }

        StartCoroutine( FindMinMax() );
    }

    private IEnumerator FindMinMax() {
        int min = int.MaxValue;
        int max = int.MinValue

        for(int i=0; i<ARRAY_SIZE; i++){
            if(i % SEGMENT_SIZE == 0){
                print("frame: " + Time.frameCount + ", i:" + i
+ ", min:" + min + ", max:" + max);

                // suspend for 1 frame since we've completed
another segment
                yield return null;
            }

            if(randomNumbers[i] > max){
                max = randomNumbers[i];
            } else if(randomNumbers[i] < min){
                min = randomNumbers[i];
            }
        }

        // disable this scripted component
        print("** completed - disabling scripted component");
        enabled = false;
    }
}
```

2. Run the game and you'll see how the search for highest and lowest values in the array progresses in steps, avoiding undesirable delays between each new frame.



```
frame: 0, i:0, min:9999, max:-1
UnityEngine.MonoBehaviour:print(Object)
frame: 1, i:10, min:122, max:956
UnityEngine.MonoBehaviour:print(Object)
frame: 2, i:20, min:9, max:956
UnityEngine.MonoBehaviour:print(Object)
frame: 3, i:30, min:9, max:977
UnityEngine.MonoBehaviour:print(Object)
frame: 4, i:40, min:9, max:977
UnityEngine.MonoBehaviour:print(Object)
** completed - disabling scripted component
UnityEngine.MonoBehaviour:print(Object)
```

Insert image 1362OT_11_24.png

How it works...

Array `randomNumbers` of random integers is created in `Awake()`. Then, method `FindMinMax()` is started as a coroutine. The size of the array is defined by constant `ARRAY_SIZE`, and the number of elements to process each frame by `SEGMENT_SIZE`.

Method `FindMinMax()` sets initial values for *min* and *max*, and begins to loop through the array. If the current index is divisible by the `SEGMENT_SIZE` (remainder 0), then we make the method display the current frame number and variable values, and suspend execution for one frame with a `yield null` statement. For every loop, the value for the current array index is compared with `min` and `max`, and those values are updated if a new minimum or maximum has been found. When the loop is completed, the scripted component disables itself.

There's more...

Some details you don't want to miss:

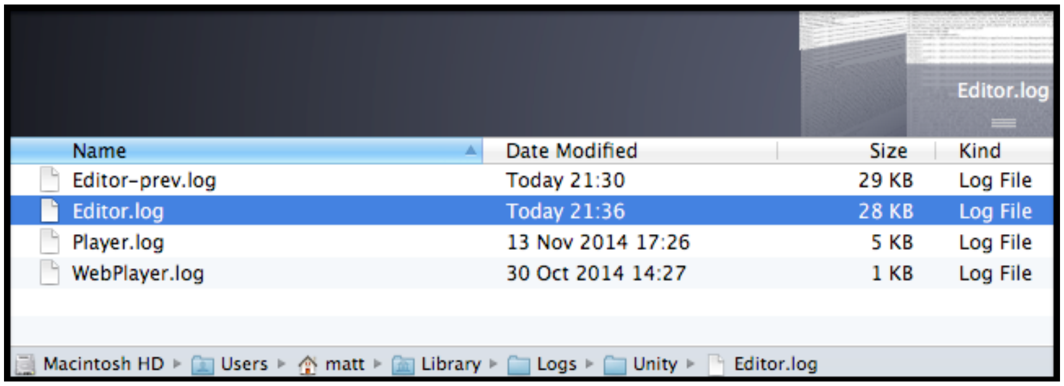
Retrieving the complete Unity log text files from your system

As well as seeing log texts in the Console panel, you can also access the Unity editor log text file as follows:

- Mac:
  - `~/Library/Logs/Unity/Editor.log`

- and access through the standard Console app

- Windows: `C:\Users\username\AppData\Local\Unity\Editor\Editor.log`
- Mobile devices (see the Unity documentation for accessing device log data)



Name	Date Modified	Size	Kind
Editor-prev.log	Today 21:30	29 KB	Log File
Editor.log	Today 21:36	28 KB	Log File
Player.log	13 Nov 2014 17:26	5 KB	Log File
WebPlayer.log	30 Oct 2014 14:27	1 KB	Log File

Editor.log
Macintosh HD ▸ Users ▸ matt ▸ Library ▸ Logs ▸ Unity ▸ Editor.log

Insert image 1362OT_11_25.png

For more information about Unity logs files, see the online Manual:

`http://docs.unity3d.com/Manual/LogFiles.html`

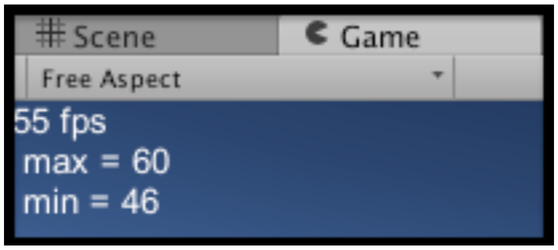
See also

Refer to the following recipe in this chapter for more information:

- *Executing methods regularly but independent of frame rate with coroutines*

Evaluating performance by measuring max and min frame rates (FPS)

Optimization principal 4: Use performance data to drive design and coding decisions. A useful raw measurement of game performance is the maximum and minimum frame rate for a section of a game. In this recipe, we make use of a Creative Commons **FPS** (**Frames Per Second**) calculation script to record the maximum and minimum frame rates for a game performing some mathematics calculations each frame.



55 fps
max = 60
min = 46

Insert image 1362OT_11_26.png

Getting ready

For this recipe, we have provided C# script `FPSCounter.cs` in the folder `1362_11_12`. This file is the one we have modified to include the maximum and minimum values, based on the do-it-yourself frame rate calculation script from Annop "Nargus" Prapasapong, kindly published under Creative Commons on the Unify Wiki at:

`http://wiki.unity3d.com/index.php?title=FramesPerSecond`

How to do it...

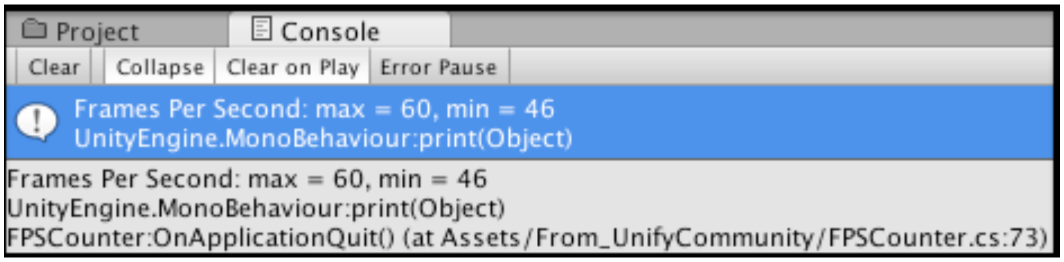
To calculate and record the maximum and minimum frames per second, follow these steps:

1. Start a new project, and import the script `FPSCounter.cs`
2. Add script class `FPSCounter` to the **Main Camera**.
3. Add the following C# script class `SomeCalculations` to the **Main Camera**:

```
using UnityEngine;
using System.Collections;

public class SomeCalculations : MonoBehaviour {
   public int outerLoopIterations = 20;
   public int innerLoopMaxIterations = 100;

   void Update(){
      for(int i = 0; i < outerLoopIterations; i++){
         int innerLoopIterations =
Random.Range(2,innerLoopMaxIterations);
         for(int j = 0; j < innerLoopIterations; j++){
            float n = Random.Range(-1000f, 1000f);
         }
      }
   }
```

```
        }
```

4. Run the game for 20 to 30 seconds. On the screen, you should see the current average, and the maximum and minimum frame rates displayed.
5. Stop the game running. You should now see in the **Console** a summary message stating the max and min frames per second.



```
Frames Per Second: max = 60, min = 46
UnityEngine.MonoBehaviour:print(Object)

Frames Per Second: max = 60, min = 46
UnityEngine.MonoBehaviour:print(Object)
FPSCounter:OnApplicationQuit() (at Assets/From_UnifyCommunity/FPSCounter.cs:73)
```

Insert image 1362OT_11_27.png

How it works...

The `SomeCalculations` script ensures we make Unity do something each frame, in that it performs lots of calculations when method `Update()` is called each frame. There is an outer loop (loop counter `i`) of public variable `outerLoopIterations` iterations (which we set to 20), and an inner loop (loop counter `j`), which is a random number of iterations between 2 and the value of public variable `innerLoopMaxIterations` (which we set to 100).

The work for the calculations of average **Frames Per Second** (**FPS**) is performed by the `FPSCounter` script, which runs coroutine method `FPS()` at the chosen frequency (which we can change in the **Inspector**). Each time the method `FPS()` executes, it recalculates the average frames per second, updates the max and minimum values if appropriate, and if the **Display While Running** checkbox was ticked, then a **GUIText** object on screen is updated with a message of the average, max, and min FPS.

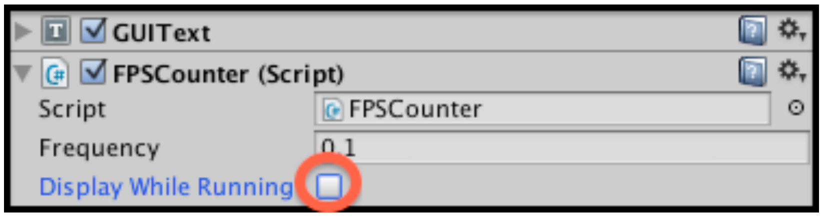
Finally, method `OnApplicationQuit()` in script class `FPSCounter` is executed when the game is terminated, and prints to the console the summary max/min FPS message.

There's more...

Some details you don't want to miss:

Turn off runtime display to reduce FPS processing

We have added an option so that you can turn off the runtime display, which will reduce the processing required for the FPS calculations. You just have to un-check the **Display While Running** checkbox in the **Inspector**.



Script: FPSCounter
Frequency: 0.1
Display While Running: ☐

Insert image 1362OT_11_28.png

See also

Refer to the following recipes in this chapter for more information:

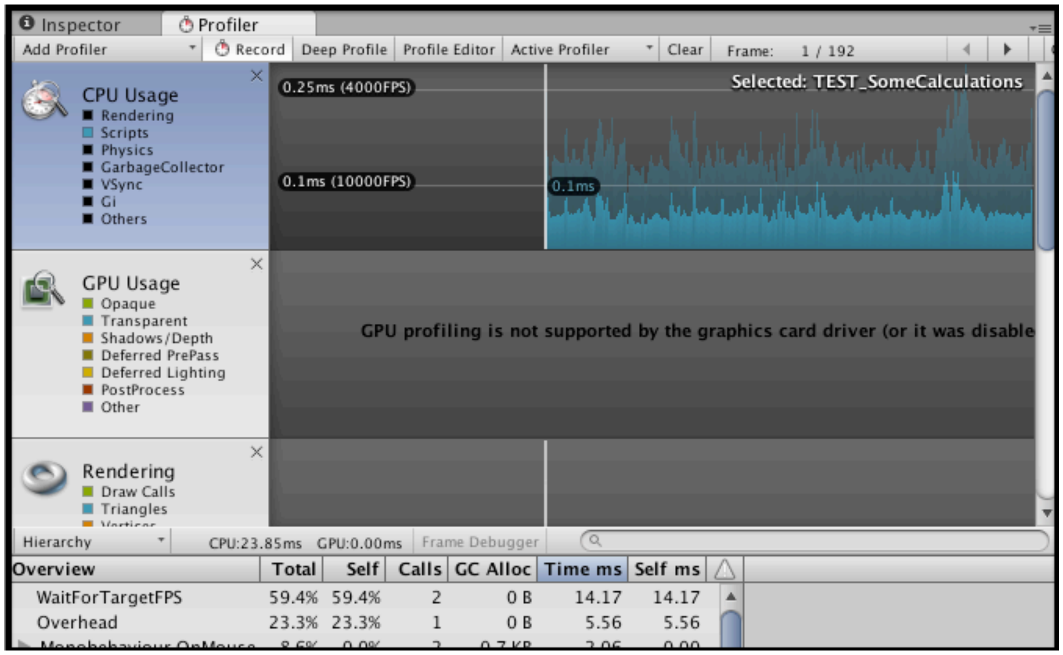
- Identifying performance "bottlenecks" with the Performance Profiler
- Identifying performance "bottlenecks" with Do-It-Yourself. Performance Profiling

Identifying performance "bottlenecks" with the Performance Profiler

Optimization principal 4: Use performance data to drive design and coding decisions.

As well as following general asset and code design principals, which we know ought to lead to improved performance, each game is different, and in reality the only way to know which design decisions affect performance the most is to collect and analyze runtime performance data. While a raw **Frames Per Second** (**FPS**) measurement is useful, to choose between different decisions having detailed information about the processing requirements for rendering and code execution each frame is invaluable.

Unity 5s Profiler offers detailed breakdown of code and rendering processing requirements, as well as processing required by GPU, audio, and both 2D and 3D physics. Perhaps the most useful, it allows programmers to explicitly record data for named code segments. We will name our profile `MATT_SomeCalculations` and record and examine frame-by-frame processing requirements for our calculations.



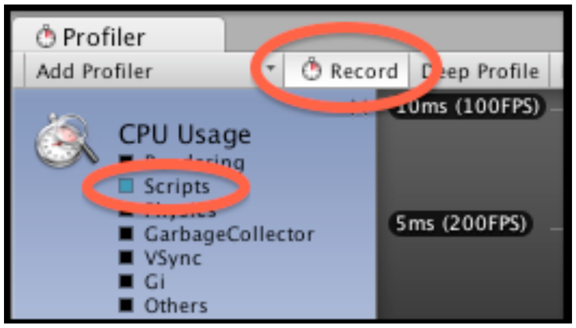
Overview	Total	Self	Calls	GC Alloc	Time ms	Self ms
WaitForTargetFPS	59.4%	59.4%	2	0 B	14.17	14.17
Overhead	23.3%	23.3%	1	0 B	5.56	5.56

Insert image 1362OT_11_29.png

How to do it...

To record processing requirements using the Unity Profiler, follow these steps:

1. Start a new 2D project.
2. Open the **Profiler** window from menu **Window**, and ensure the **Record** option is selected, and that the **Scripts** performance data is being collected.

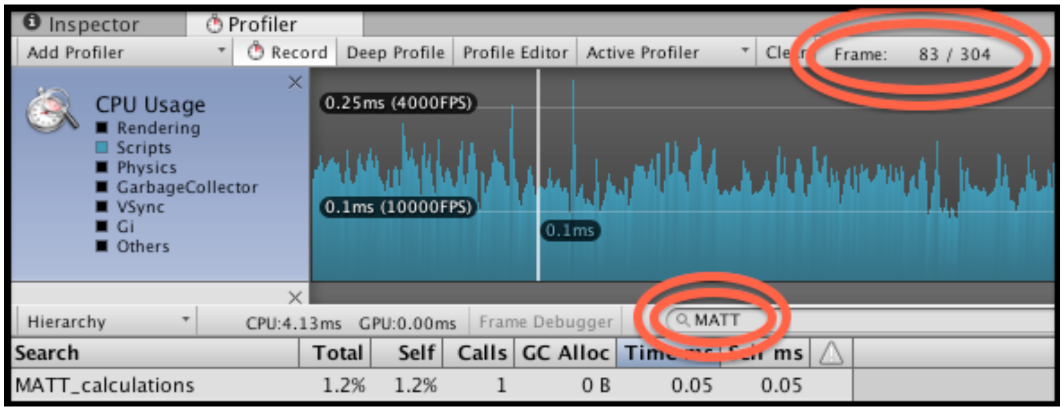


Insert image 1362OT_11_30.png

3. Add the following C# script class `ProfileCalculations` to the **Main Camera**:

```
using UnityEngine;
using System.Collections;

public class ProfileCalculations : MonoBehaviour {
   public int outerLoopIterations = 20;
   public int innerLoopMaxIterations = 100;

   void Update(){
      Profiler.BeginSample("MATT_calculations");

      for(int i = 0; i < outerLoopIterations; i++){
         int innerLoopIterations =
Random.Range(2,innerLoopMaxIterations);
         for(int j = 0; j < innerLoopIterations; j++){
            float n = Random.Range(-1000f, 1000f);
         }
      }

      Profiler.EndSample();
   }
}
```

4. Run the game for 20 to 30 seconds.
5. Stop the game running. You should now see in the **Profiler** panel details of the breakdown of processing required for the selected frame - each of the jagged lines in the top right of the **Profiler** panel represents the collected data for a frame.
6. View data for different frames by dragging the white line to a different horizontal position - the current frame, and total number of frames is shown at the top right in the form **Frame: frame / totalFrames**.
7. Since we have named a code profile sample, prefixed with **MATT**, we can limit the display of data to only samples containing that word. In the search text box (next to the little magnifying glass) type `MATT`, and you should now see just a single row of profile data for our sample **MATT_calculations**. We can see that for frame 83, our code took up 1.2% of the processing for that frame.



Search	Total	Self	Calls	GC Alloc	Time ms	Self ms
MATT_calculations	1.2%	1.2%	1	0 B	0.05	0.05

Insert image 1362OT_11_31.png

How it works...

The `ProfileCalculations` script ensures we make Unity do something each frame; it does lots of calculations with an inner and outer loop just like in the previous FPS recipe.

The important 2 statements are those that mark the beginning and ending of a named code sample to be recorded and presented in the **Profiler**. The statement `Profiler.BeginSample("MATT_calculations")` starts our named profile, and it is ended with the `EndSample` statement.

Using an eye-catching prefix allows us to easily isolate our named code profile for analysis, using the search text box in the **Profiler** panel.

See also

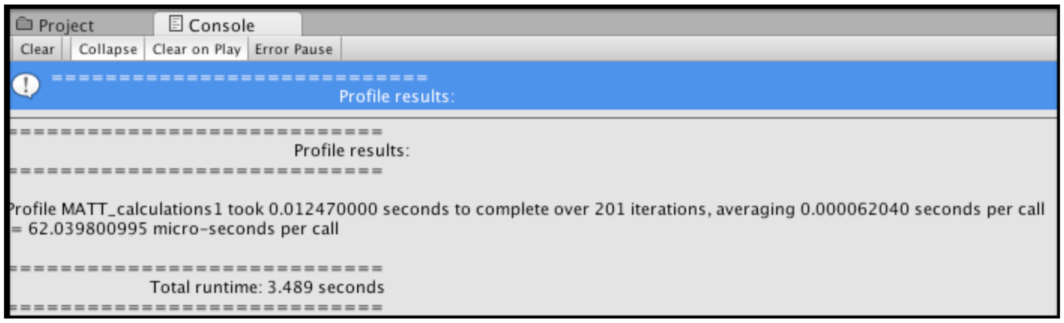
Refer to the following recipes in this chapter for more information:

- *Evaluating performance by measuring max and min frame rates (FPS)*
- *Identifying performance "bottlenecks" with Do-It-Yourself Performance Profiling*

Identifying performance "bottlenecks" with Do-It-Yourself Performance Profiling

Optimization principal 4: Use performance data to drive design and coding decisions.

The Unity 5 performance profiler is great - but there may be times where we wish to have completed control over the code we are running and how it displays or logs data. In this recipe, we explore how to use a freely available script for **Do-It-Yourself** (**DIY**) performance profiling that works with the free version of Unity. While it's not quite as fancy as the graphical and detailed profiling of the performance profiler from Unity, it still provides low level data about the time required each frame by named parts of scripts, which is sufficient for making code design decisions to improve game performance.



```
==========================
Profile results:

==========================
Profile results:
==========================

Profile MATT_calculations1 took 0.012470000 seconds to complete over 201 iterations, averaging 0.000062040 seconds per call
= 62.039800995 micro-seconds per call

==========================
Total runtime: 3.489 seconds
==========================
```

Insert image 1362OT_11_32.png

Getting ready

For this recipe, we have provided C# script `Profile.cs` in folder `1362_11_14`. This is the do-it-yourself profiling script from Michael Garforth, kindly published under Creative Commons on the Unify Wiki at: `http://wiki.unity3d.com/index.php/Profiler`.

How to do it...

To record processing requirements using Do-It-Yourself code profiling, follow these steps:

1. Start a new project, and import the script `Profile.cs`.
2. Add the following C# script class `DIYProfiling` to the **Main Camera**:

```
using UnityEngine;
using System.Collections;

public class DIYProfiling : MonoBehaviour {
   public int outerLoopIterations = 20;
   public int innerLoopMaxIterations = 100;

   void Update(){
      string profileName = "MATT_calculations";
      Profile.StartProfile(profileName);
```

```
        for (int i = 0; i < outerLoopIterations; i++){
            int innerLoopIterations =
Random.Range(2,innerLoopMaxIterations);
            for (int j = 0; j < innerLoopIterations; j++){
                float n = Random.Range(-1000f, 1000f);
            }
        }

        Profile.EndProfile(profileName);
    }


    private void OnApplicationQuit() {
        Profile.PrintResults();
    }
}
```

6. Run the game for a few seconds.
7. Stop the game running. You should now see in the **Console** a summary message stating total processing time for our named Profile, average time, and number of iterations, and also the total time for which the game was run.

How it works...

As you can see, the script is almost identical to that used with the Unity Profiling in the previous recipe. Rather than calling the Unity Profiler, we call static (class) methods of Michael Garforth's `Profile` class.

We call `Profile` class methods `StartProfile(…)` and `EndProfile(…)` with the string name for what is to be analyzed (in this example, `MATT_calculations`).

Finally, method `OnApplicationQuit()` is executed when the game is terminated, calling the `PrintResuls()` method of class `Profile`, which prints to the console the summary performance information.

The `Profile` class records how many times, and how long between Start and End each named profile is called, and outputs summary information about these executions when `PrintResuls()` is called.

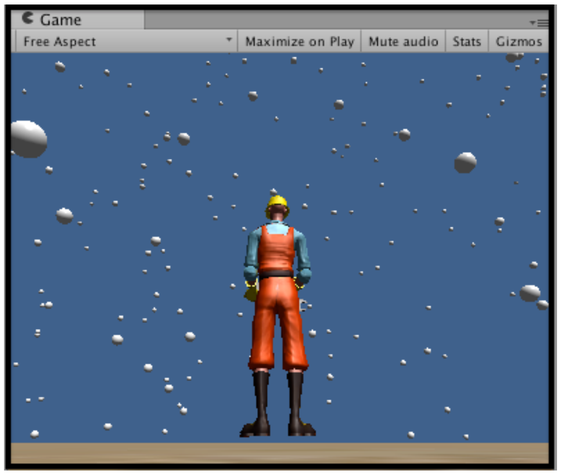
See also

Refer to the following recipes in this chapter for more information:

- *Evaluating performance by measuring max and min frame rates (FPS)*
- *Identifying performance "bottlenecks" with the Performance Profiler*

Cache GameObject and Component References to Avoid Expensive Lookups

Optimization principal 2: Minimize actions requiring Unity to perform "reflection" over objects and searching of all current scene objects. Reflection is when at run-time, Unity has to analyze objects to see whether they contain a method corresponding to a 'message' the object has received, an example would be `SendMessage()`. An example of making Unity perform a search over all active objects in a scene would be the simple and useful, but slow, `FindObjectsByTag()`. Another action that slows Unity down is each time we make it look up an object's component using `GetComponent()`.



Insert image 1362OT_11_33.png

In the olden days for many components, Unity offered **quick component property getters** such as `.audio` to reference the `AudioSource` component of a script's parent **GameObject**, and `rigidbody` to reference the RigidBody component and so on. However, this wasn't a consistent rule and in other cases, you had to use `GetComponent()`. With Unity 5, all these **quick component property getters** have been removed (with the exception of `.transform`, which is automatically cached, so has no performance cost to use). To help game developers update their scripts to work with Unity 5, they introduced **Automatic Script Updating**, whereby (after a suitable warning to have backed up files before going ahead!) Unity will go through scripts replacing **quick component property getters** code with the standardized `GetComponent<ComponentTyle>()` code pattern, such as `GetComponent<Rigidbody>()` and `GetComponent<AudioSource>()`. However, while script updating makes things consistent, and also makes explicit all these

GetComponent() reflection statements, each GetComponent() execution eats up valuable processing resources.

> You can read more about Unity's reasons for this, (and the alternative **Extension Methods** approach they rejected; a shame - I think we'll see them appear in a later version of Unity since it's an elegant way to solve this coding situation) in this June 2014 blog post and manual page:
>
> http://blogs.unity3d.com/2014/06/23/unity5-api-changes-automatic-script-updating/
> http://unity3d.com/learn/tutorials/modules/intermediate/scripting/extension-methods

In this recipe, we'll incrementally refactor a method, making it more efficient at each step by removing reflection and component lookup actions. The method we'll improve is to find half the distance from the **GameObject** in the scene tagged Player (a **3rd Person Controller**), and 1,000 other **GameObjects** in the scene tagged Respawn.

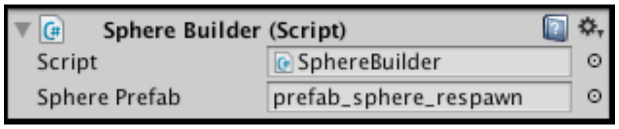
How to do it...

To improve code performance by caching component lookups, follow these steps:

1. Create a new 3D project, importing the **Character Controller** and **Terrain Assets** packages.
2. Create a new **Terrain** (size **200 x 200**, located at **-100, 0, -100**) and texture-paint it with **GoodDirt**.
3. Add a **3rdPersonController** at the center of the terrain (that is, **0, 1, 0**). Note that this will already be tagged **Player**.
4. Create a new **Sphere**, and give it the tag **Respawn**.
5. In the **Project** panel, create a new empty prefab named **prefab_sphere**, and drag the **Sphere** from the **Hierarchy** panel into your prefab in the **Project** panel.
6. Now delete the **Sphere** from the **Hierarchy** panel (since all its properties have been copied into our prefab).
7. Add the following C# script class SphereBuilder to the **Main Camera**:

```
using UnityEngine;
using System.Collections;

public class SphereBuilder : MonoBehaviour
{
   public const int NUM_SPHERES = 1000;
   public GameObject spherePrefab;

   void Awake(){
        List<Vector3> randomPositions =
BuildVector3Collection(NUM_SPHERES);
        for(int i=0; i < NUM_SPHERES; i++){
           Vector3 pos = randomPositions[i];
           Instantiate(spherePrefab, pos,
Quaternion.identity);
        }
    }

    public List<Vector3> BuildVector3Collection(int
numPositions){
        List<Vector3> positionArrayList = new
List<Vector3>();
        for(int i=0; i < numPositions; i++) {
           float x = Random.Range(-100, 100);
           float y = Random.Range(1, 100);
           float z = Random.Range(-100, 100);
           Vector3 pos = new Vector3(x,y,z);
           positionArrayList.Add (pos);
        }

        return positionArrayList;
    }
}
```

8. With the **Main Camera** selected in the **Hierarchy**, drag **prefab_sphere** from the Project panel in **Inspector** pubic variable `Sphere Prefab`, for script component **SphereBuilder**.


Sphere Builder (Script)
Script: SphereBuilder
Sphere Prefab: prefab_sphere_respawn

Insert image 1362OT_11_34.png

9. Add the following C# script class `SimpleMath` to the **Main Camera**:

```
using UnityEngine;
using System.Collections;

public class SimpleMath : MonoBehaviour {
    public float Halve(float n){
        return n / 2;
    }
}
```

Method 1 – AverageDistance calculation

Follow these steps:

1. Add the following C# script class `AverageDistance` to the **Main Camera**:

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

public class AverageDistance : MonoBehaviour
{
   void Update(){
      // method1 - basic
      Profiler.BeginSample("TESTING_method1");
      GameObject[] sphereArray = 
GameObject.FindGameObjectsWithTag("Respawn");
      for (int i=0; i < SphereBuilder.NUM_SPHERES; i++){
         HalfDistanceBasic(sphereArray[i].transform);
      }
      Profiler.EndSample();
   }

   // basic
   private void HalfDistanceBasic(Transform 
sphereGOTransform){
      Transform playerTransform = 
GameObject.FindGameObjectWithTag("Player").transform;
      Vector3 pos1 = playerTransform.position;
      Vector3 pos2 = sphereGOTransform.position;

      float distance = Vector3.Distance(pos1, pos2);

      SimpleMath mathObject = GetComponent<SimpleMath>();
      float halfDistance = mathObject.Halve(distance);
   }
}
```

2. Open the **Profiler** panel, and ensure record is selected and Script processing load is being recorded.
3. Run the game for 10-20 seconds.
4. In the **Profiler** panel, restrict the listed results to only samples starting with `TEST`. For whichever frame you select, you should see the percentage CPU load and milliseconds required for **TESTING_method1**.

Method 2 – Cache array of Respawn object transforms

Follow these steps:

1. `FindGameObjectWithTag()` is slow, so let's fix that for the search for objects tagged `Respawn`. First in C# script class `AverageDistance`, add a private Transform array variable named `sphereTransformArrayCache`:

   ```
   private Transform[] sphereTransformArrayCache;
   ```

2. Now add method `Start()`, the statement that stores in this array references to the **Transform** component of all our `Respawn` tagged objects:

   ```
   private void Start(){
      GameObject[] sphereGOArray =
   GameObject.FindGameObjectsWithTag("Respawn");
      sphereTransformArrayCache = new
   Transform[SphereBuilder.NUM_SPHERES];
      for (int i=0; i < SphereBuilder.NUM_SPHERES; i++){
         sphereTransformArrayCache[i] =
   sphereGOArray[i].transform;
      }
   }
   ```

3. Now in method `Update()`, start a new **Profiler** sample named **TESTING_method2**, that uses our cached array of games objects tagged with `Respawn`:

   ```
   // method2 - use cached sphere ('Respawn' array)
   Profiler.BeginSample("TESTING_method2");
   for (int i=0; i < SphereBuilder.NUM_SPHERES; i++){
      HalfDistanceBasic(sphereTransformArrayCache[i]);
   }
   Profiler.EndSample();
   ```

4. Once again, run the game for 10-20 seconds, and set the **Profiler** panel to restrict the listed results to only samples starting with `TEST`. For whichever frame you select, you should see the percentage CPU load and milliseconds required for **TESTING_method1** and **TESTING_method2**.

Method 3 – Cache reference to Player Transform

That should run faster. But wait! Let's improve things some more. Let's make use of a cached reference to **Cube-Player** component's transform, avoiding the slow object-tag reflection lookup altogether. Follow these steps:

1. First add a new private variable, and a statement in method `Start()` to assign the `Player` object's transform in this variable `playerTransformCache`:

   ```
   private Transform playerTransformCache;
   private Transform[] sphereTransformArrayCache;
   ```

```
private void Start(){
   GameObject[] sphereGOArray =
GameObject.FindGameObjectsWithTag("Respawn");
   sphereTransformArrayCache = new
Transform[SphereBuilder.NUM_SPHERES];
   for (int i=0; i < SphereBuilder.NUM_SPHERES; i++){
      sphereTransformArrayCache[i] =
sphereGOArray[i].transform;
   }

   playerTransformCache =
GameObject.FindGameObjectWithTag("Player").transform;
}
```

2. Now in `Update()`, add code to start a new **Profiler** sample named **TESTING_method3**:

```
// method3 - use cached playerTransform
Profiler.BeginSample("TESTING_method3");
for (int i=0; i < SphereBuilder.NUM_SPHERES; i++){
   HalfDistanceCachePlayerTransform(sphereTransformArrayCac
he[i]);
}
Profiler.EndSample();
```

3. Finally, we need to write a new method that calculates the half-distance making use of the cached player transform variable we have setup. So add this new method: `HalfDistanceCachePlayerTransform( sphereTransformArrayCache[i] )`:

```
// playerTransform cached
private void HalfDistanceCachePlayerTransform(Transform
sphereGOTransform){
   Vector3 pos1 = playerTransformCache.position;
   Vector3 pos2 = sphereGOTransform.position;
   float distance = Vector3.Distance(pos1, pos2);
   SimpleMath mathObject = GetComponent<SimpleMath>();
   float halfDistance = mathObject.Halve(distance);
}
```

Method 4 – Cache Player's Vector3 position

Let's improve things some more. If for our particular game we can make the assumption that the player character does not move, we have an opportunity to cache the player's position once, rather than retrieving it each frame.

Follow these steps:

1. At the moment to find pos1, we are making Unity find the position Vector3 value inside the playerTransform *every time method* Update() *is called.* Let's cache this `Vector3` position with a variable and statement in `Start()` as follows:

```
private Vector3 pos1Cache;

private void Start(){
    ...
    pos1Cache = playerTransformCache.position;
}
```

2. Now write a new half-distance method that makes use of this cached position:

```
// player position cached
private void HalfDistanceCachePlayer1Position(Transform
sphereGOTransform){
    Vector3 pos1 = pos1Cache;
    Vector3 pos2 = sphereGOTransform.position;
    float distance = Vector3.Distance(pos1, pos2);
    SimpleMath mathObject = GetComponent<SimpleMath>();
    float halfDistance = mathObject.Halve(distance);
}
```

3. Now in method `Update()`, add the following code so we create a new sample for our Method 4, and call our new half-distance method:

```
// method4 - use cached playerTransform.position
Profiler.BeginSample("TESTING_method4");
for (int i=0; i < SphereBuilder.NUM_SPHERES; i++){
    HalfDistanceCachePlayer1Position(sphereTransformArrayCac
he[i]);
}
Profiler.EndSample();
```

Method 5 – Cache reference to SimpleMath component

That should improve things again. But we can still improve things - you'll notice in our latest half-distance method, we have an explicit `GetComponent()` call to get a reference to our `mathObject` - this will be executed *every time the method is called.* Follow these steps:

1. Let's cache this scripted component reference as well, to save a `GetComponent()` reflection for each iteration. We'll declare a variable `mathObjectCache` and in `Awake()`, we will set it to refer to our `SimpleMath` scripted component:

```
private SimpleMath mathObjectCache;

private void Awake(){
        mathObjectCache = GetComponent<SimpleMath>();
    }
```

2. Let's write a new half-distance method that uses this cached reference to the Math component `HalfDistanceCacheMathComponent(i)`:

```
    // math Component cache
    private void HalfDistanceCacheMathComponent(Transform
    sphereGOTransform){
        Vector3 pos1 = pos1Cache;
        Vector3 pos2 = sphereGOTransform.position;
        float distance = Vector3.Distance(pos1, pos2);
        SimpleMath mathObject = mathObjectCache;
        float halfDistance = mathObject.Halve(distance);
    }
```

3. Now in method `Update()`, add the following code so we create a new sample for our Method 5, and call our new half-distance method:

```
    // method5 - use cached math component
    Profiler.BeginSample("TESTING_method5");
    for (int i=0; i < SphereBuilder.NUM_SPHERES; i++){
        HalfDistanceCacheMathComponent(sphereTransformArrayCache
    [i]);
    }
    Profiler.EndSample();
```

Method 6 – Cache array of sphere Vector3 positions

We've improved things quite a bit, but there is still a glaring opportunity to use caching to improve our code – if we can assume that the spheres do not move, which seems reasonable in this example. At present, every frame and for every sphere, in our half-distance calculation method we are asking Unity to retrieve the value of the Vector3 position property in the transform of current sphere (this is our variable `pos2`), this position is used to calculate the distance of the current sphere from `Player`. Let's create an array of all those `Vector3` positions, so we can pass the current one to our half-distance calculation method and save the work of retrieving it so many times.

Follow these steps:

1. First add a new private variable, and a statement inside our existing loop in method `Start()` to assign each sphere's Vector3 transform position in array `spherePositionArrayCache`:

```
    private Vector3[] spherePositionArrayCache = new
    Vector3[SphereBuilder.NUM_SPHERES];

    private void Start(){
```

```
    GameObject[] sphereGOArray =
GameObject.FindGameObjectsWithTag("Respawn");
    sphereTransformArrayCache = new
Transform[SphereBuilder.NUM_SPHERES];
    for (int i=0; i < SphereBuilder.NUM_SPHERES; i++){
        sphereTransformArrayCache[i] =
sphereGOArray[i].transform;
        spherePositionArrayCache[i] =
sphereGOArray[i].transform.position;
    }

    playerTransformCache =
GameObject.FindGameObjectWithTag("Player").transform;
    pos1Cache = playerTransformCache.position;
}
```

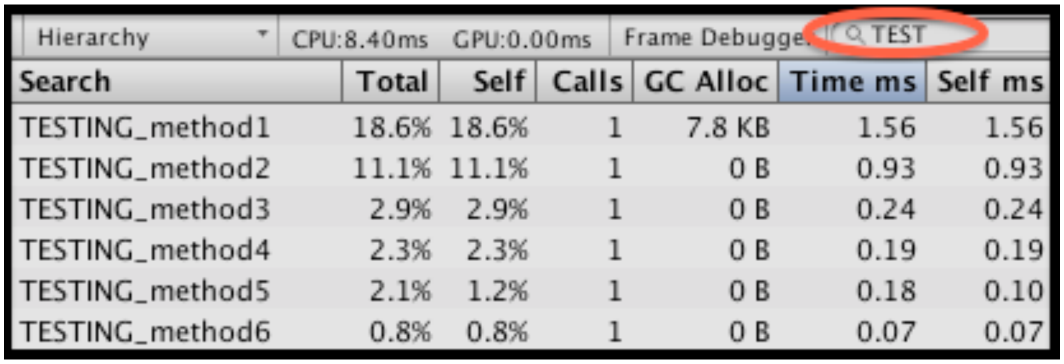
2. Let's write a new half-distance method that uses this array of cached positions:

```
// sphere position cache
private void HalfDistanceCacheSpherePositions(Transform
sphereGOTransform, Vector3 pos2){
    Vector3 pos1 = pos1Cache;
    float distance = Vector3.Distance(pos1, pos2);
    SimpleMath mathObject = mathObjectCache;
    float halfDistance = mathObject.Halve(distance);
}
```

3. Now in method `Update()`, add the following code so we create a new sample for our Method 6 and call our new half-distance method:

```
// method6 - use cached array of sphere positions
Profiler.BeginSample("TESTING_method6");
for (int i=0; i < SphereBuilder.NUM_SPHERES; i++){
    HalfDistanceCacheSpherePositions(sphereTransformArrayCac
he[i], spherePositionArrayCache[i]);
}
Profiler.EndSample();
```

4. Open the **Profiler** panel, and ensure record is selected and Script processing load is being recorded.
5. Run the game for 10-20 seconds.
6. In the **Profiler** panel, restrict the listed results to only samples starting with `TEST`. For whichever frame you select, you should see the percentage CPU load and milliseconds required for each method (lower is better for both these values!). For almost every frame, you should see how/if each method refined by caching has reduced the CPU load.



Hierarchy | CPU:8.40ms GPU:0.00ms | Frame Debugger | TEST

Search	Total	Self	Calls	GC Alloc	Time ms	Self ms
TESTING_method1	18.6%	18.6%	1	7.8 KB	1.56	1.56
TESTING_method2	11.1%	11.1%	1	0 B	0.93	0.93
TESTING_method3	2.9%	2.9%	1	0 B	0.24	0.24
TESTING_method4	2.3%	2.3%	1	0 B	0.19	0.19
TESTING_method5	2.1%	1.2%	1	0 B	0.18	0.10
TESTING_method6	0.8%	0.8%	1	0 B	0.07	0.07

Insert image 1362OT_11_35.png

How it works...

This recipe illustrates how we try to cache references once, before any iteration, for variables whose value will not change – such as references to `GameObjects` and their components, and in this example the `Transform` components and `Vector3` positions of objects tagged `Player` and `Respawn`. Of course, as with everything, there is a 'cost' associated with caching, and that cost is the memory requirements to store all those references. This is known as the **Space-Time Tradeoff**. You can learn more about this classic computer science speed vs. memory tradeoff at the following web page: `https://en.wikipedia.org/wiki/Space%E2%80%93time_tradeoff`.

In methods that need to be performed many times, this removing of implicit and explicit component and object lookups may offer a measurable performance improvement.

NOTE: Two good places to learn more about Unity performance optimization techniques are from the Performance Optimization web page in the Unity Script Reference; and Unity's Jonas Echterhoff & Kim Steen Riber Unite2012-presentation *Performance Optimization Tips and Tricks for Unity*. Many recipes in this chapter had their origins from suggestions in these sources:

```
http://docs.unity3d.com/410/Documentation/ScriptReferenc
   e/index.Performance_Optimization.html
http://unity3d.com/unite/archive/2012
```

See also

Refer to the following recipes in this chapter for more information:

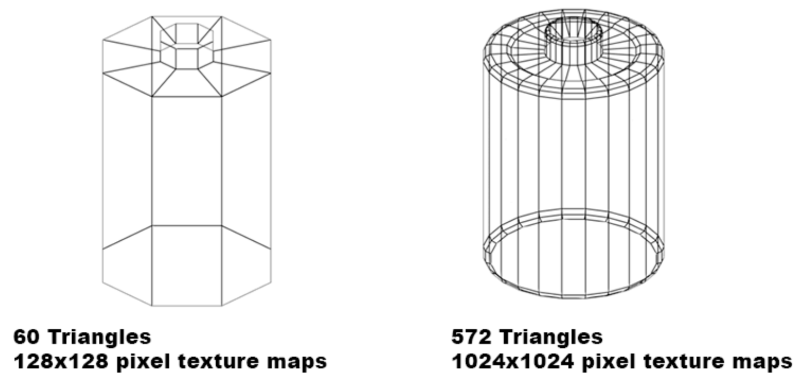
- *Improving efficiency with delegates and events (and avoiding SendMessage!)*
- *Identifying performance "bottlenecks" with the Performance Profiler*
- *Identifying performance "bottlenecks" with Do-It-Yourself Performance Profiling*

Improving performance with LOD Groups

Optimization principal 5: Minimize the number of draw calls.

Detailed geometry and high resolution texture maps can be a double-edged sword: they can deliver a better visual experience, but they can impact negatively on the game's performance. **LOD groups** address this issue by replacing high-quality objects by simplified versions whenever that object takes up a smaller portion of the screen than necessary for a high-quality version to make a significant difference.

In this recipe, we will use a **LOD group** to create a game object featuring two different levels of detail: a high-quality version for whenever the object takes up more than 50% of the screen; and a low-quality version for the times it takes up less than that amount.



Insert image 1362OT_11_09.png

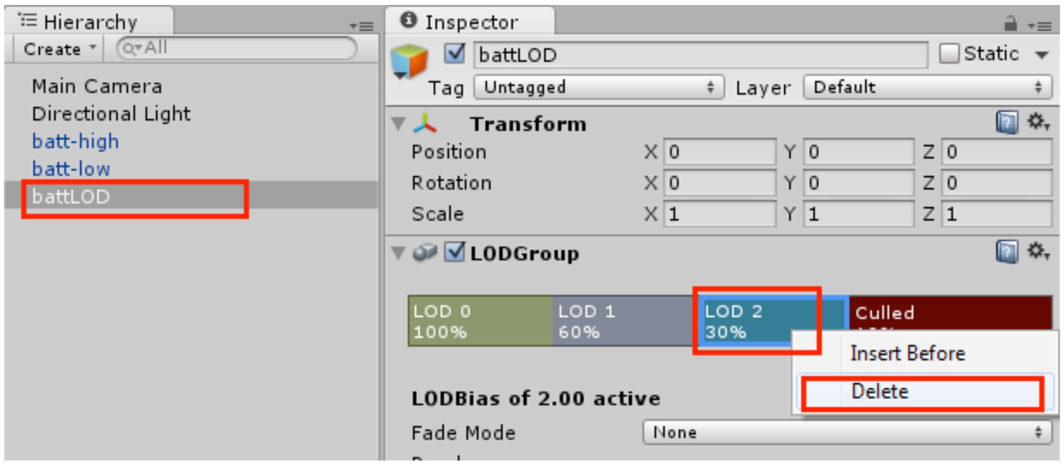
Getting ready

For this recipe, we have prepared two prefabs for the high and low quality versions of the game object. They share the same dimensions and transform settings (position, rotation, and scale), so they can replace each other seamlessly. Both prefabs are contained within the package named `LODGroup`, available in the folder `1362_11_16`.

How to do it...

To create a LOD Group, follow these steps:

1. Import the package **LODGroup** into your project.
2. From the **Project** view, inside the **LOD** folder, drag the **batt-high** prefab into the **Hierarchy** view. Then, do the same for the **batt-low** prefab. Make sure they are placed at the same **Position** (**X**: 0; **Y**: 0; **Z**: 0).
3. From the **Create** dropdown menu in the **Hierarchy** view, create a new empty game object (**Create | Create Empty**). Rename it **battLOD**.
4. Add the **LODGroup** component to **battLOD** (menu **Component | Rendering | LODGroup**).
5. Select the **battLOD** object and, from the **Inspector** view, **LODGroup** component, right-click **LOD 2** and delete it (since we'll have only two different LODs: **LOD 0** and **LOD 1**).



battLOD

Tag: Untagged, Layer: Default

Transform

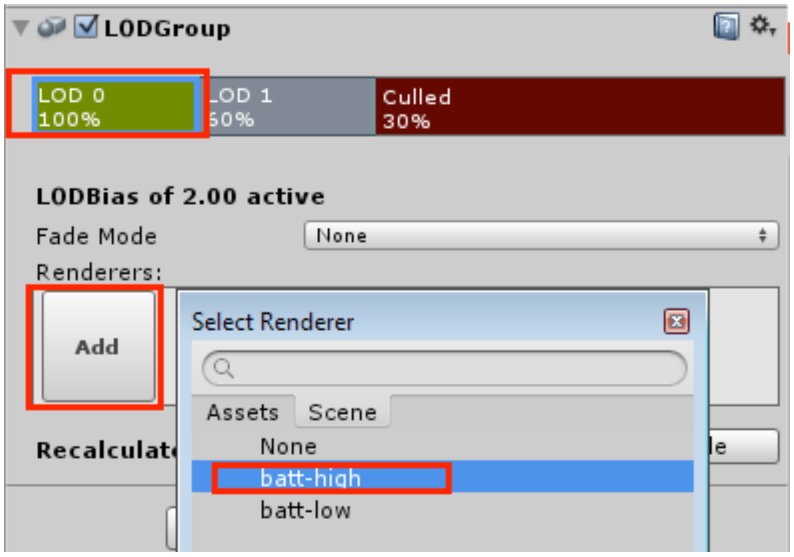
Position	X	0	Y	0	Z	0
Rotation	X	0	Y	0	Z	0
Scale	X	1	Y	1	Z	1

LODBias of 2.00 active

Fade Mode: None

Insert image 1362OT_11_10.png

6. Select the **LOD 0** area, click the **Add** button, and select the **batt-high** game object from the list. A message about re-parenting objects will appear. Select **Yes, Reparent**.

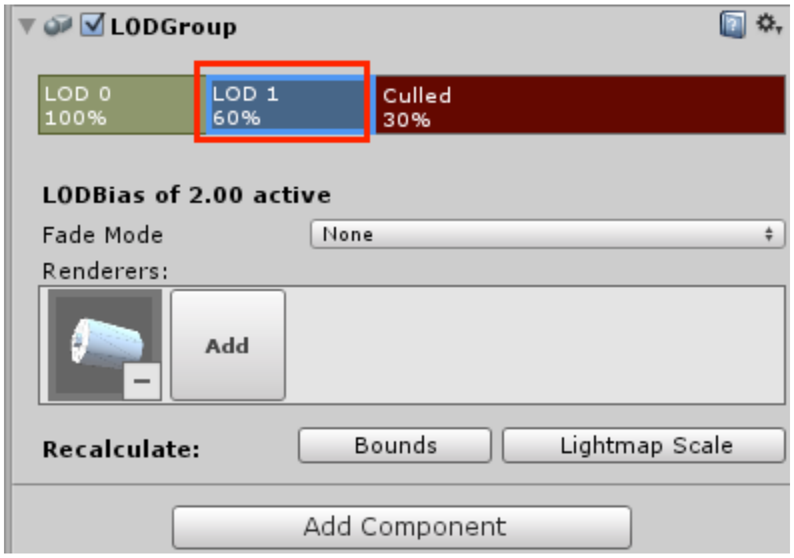


LODBias of 2.00 active

Fade Mode: None

Insert image 1362OT_11_11.png

7. Select the **LOD 1** section, click **Add**, and select the **batt-low** object. Again, chose **Yes, Reparent** when prompted.

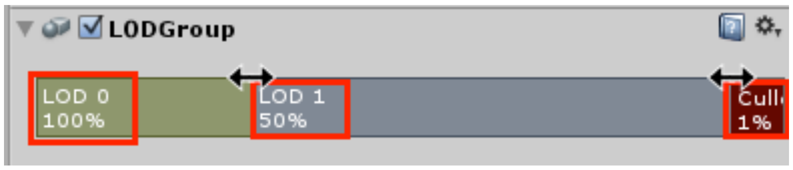


LODBias of 2.00 active

Fade Mode: None

Insert image 1362OT_11_12.png

8. Drag the limits of the LOD renderers to set them as: **LOD 0: 100%**, **LOD 1**: **50%**, **Culled: 1%.** That will make Unity render **bat-high** whenever it occupies 61%-100% of the screen space, **batt-low** when 2%-50%, and do not render anything if 1% or less.



Insert image 1362OT_11_13.png

9. Move the scene's camera towards the **battLOD** object and back. You will notice how Unity swaps between the high-definition and low-definition LOD renderer as it occupies more or less than 50% of the screen's space.

How it works...

Once we have populated the LOD renderers with the appropriate models, the **LODGroup** component will select and display the right renderer based on how much of the screen's percentage the object takes up - or even display nothing at all.

There's more...

Some details you don't want to miss:

Adding more LOD renderers

You can add more LOD renderers by right-clicking an existing LOD renderer and selecting **Insert Before** from the context menu.

Fading LOD transitions

In case you want to minimize the *popping* that occurs when renderers are swapped, you can try changing the parameter **Fade Mode** from **None** to **Percentage** or **Crossfade**.

See also

Refer to the following recipe in this chapter for more information:

- *Improving performance through reduced draw-calls by designing for draw call "batching"*

Improving performance through reduced draw-calls by designing for draw call "batching"

Optimization principal 5: Minimize the number of draw calls. One way to minimize draw-calls is by prioritizing design decisions to qualify objects for Unity's **Static** and **Dynamic draw call batching**.

The more CPU efficient batching method is Unity's **static batching**. It allows reduction of draw calls for any sized geometry. If that is not possible, then the next best thing is **dynamic batching**, which again allows Unity to process together several moving objects in a single draw call.

Note, that there is a cost – batching uses memory, and static batching uses more memory than dynamic memory. So you can improve performance with batching, but you'll be increasing the scene's memory 'footprint'. As always, use memory and performance profiling to evaluate which use of techniques is best for your game and its intended deployment device…

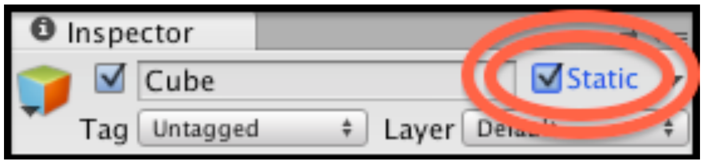
How to do it...

In this section, we will learn how to make possible **Static Batching** and **Dynamic Batching**.

Static Batching

To make possible Unity **Static Batching**, you need to do the following:

1. Ensure models share the same material.
2. Mark models as **Static**.



Insert image 1362OT_11_37.png

Objects that can be safely marked as **Static** include environment objects that won't move or be scaled.

Many techniques can be used to ensure models share the same material including:

- Avoid using textures by directly painting vertices of the model (useful links for this are provided in the *There's more…* section)

- Increasing the number of objects textured with exactly the same texture.
- Artificially enabling objects to share the same texture by combining multiple-textures into a single one (texture atlassing)
- Maximizing script use of `Renderer.sharedMaterial` rather than `Renderer.material` (since use of `Render.material` involves making a copy of the material, and therefore de-qualifies that GameObejct for batching)

In fact, both static and dynamic batching only work with objects that use the same material - so all methods above apply equally for making batching possible as well.

Dynamic Batching

To make possible Unity **Dynamic Batching**, you need to do the following:

1. Ensure models share the same material.
2. Keep the number of **vertex attributes** below 900 for each mesh.
3. Have the group of objects to quality for dynamic batching to use the same transform scale (although non-uniform scaled models can still be batched).
4. If possible, have dynamic lightmapped objects point to the same lightmap location, to facilitate dynamic batching.
5. Avoid the use of multi-pass shaders and real-time shadows if possible, since both these prevent dynamic batching.

To calculate the number of **vertex attributes**, you need to multiply the number of vertices by the number of attributes used by the **Shader**. For example, for a **Shader** using 3 attributes (vertex position, normal, and UV), it would mean that a model must have less than 300 vertices to keep the total number of attributes below 900, to qualify for dynamic batching.

There's more...

Some details you don't want to miss:

Reduce the need for textures by vertex painting

For more information about this topic, see the following:

- Blender:

`http://wiki.blender.org/index.php/Doc:2.6/Manual/Materials/Special_Effects/Vertex_Paint`

- 3D Studio Max:

`http://www.3dmax-tutorials.com/Vertex_Paint_Modifier.html`

- Maya - free Vertex Chameleon plugin
  `http://www.renderheads.com/portfolio/VertexChameleon/`

Information sources about reducing textures and materials

For more information about this topic, see the following:

- Unity manual page for Draw Call Batching:
  `http://docs.unity3d.com/Manual/DrawCallBatching.html`
- Paladin Studios:
  `http://www.paladinstudios.com/2012/07/30/4-ways-to-increase-performance-of-your-unity-game/`
- Nvidia white paper on texture atlassing to increase draw call batching opportunities:
  `http://http.download.nvidia.com/developer/NVTextureSuite/Atlas_Tools/Texture_Atlas_Whitepaper.pdf`
- Nvidia free texture tools and Photoshop plug-in:
  `http://www.nvidia.com/object/texture_atlas_tools.html`

See also

Refer to the following recipe in this chapter for more information:

- *Improving performance with LOD Groups*

Conclusion

In this chapter, we have introduced some extra features, and a range of approaches to improve game performance and collect performance data for analysis.

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 rest of the recipes in this chapter provide examples of how to investigate, and improve, the efficiency and performance of your game.

There's more...

Just as there as many components of a game, there are many parts of a game where processing *bottlenecks* may be found and needs to be addressed to improve overall game performance. Some additional suggestions and further reference sources are now provided to provide a launching pad for your further exploration of the issues of optimization and performance, since such topics could take up a whole book rather than just one chapter…

Optimization of game audio

Mobile devices have considerably less memory and processing resources than consoles, desktops, or even laptops, and often raise the biggest challenges when it comes to game audio. For example, the iPhone can only decompress 1 audio clip at a time, so a game may suffer processing spikes (that is, slow down game frame rate) due to audio decompression issues.

Paladin Studios recommend the following audio file compression strategies for mobile games:

- **Short Clips** - Native (no compression)
- **Longer clips** (or ones that loop) - compressed in memory
- **Music** - Stream from disc
- **Files which consistently cause CPU spikes** - Decompress on load

For more information about this topic, see the following:

- Unity manual audio: `http://docs.unity3d.com/Manual/AudioFiles.html`
- Paladin Studios: `http://www.paladinstudios.com/2012/07/30/4-ways-to-increase-performance-of-your-unity-game/`
- Apple developers audio page: `https://developer.apple.com/library/ios/documentation/AudioVideo/Conceptual/MultimediaPG/UsingAudio/UsingAudio.html`

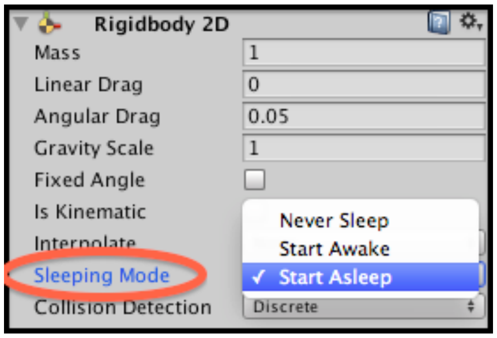
Physics engine optimization

For some strategies relating to physics, you might consider to improve performance the following:

- If possible, use **geometric primitive colliders** (2D box / 2D circle / 3D box / 3D sphere / 3D cylinder)
    - You can have multiple primitive colliders
- You can also have primitive colliders on child objects:
    - As long as you have a rigidbody on the root object in the object hierarchy
- Avoid **2D polygon** and **3D mesh** colliders:
    - These are much more processor intensive
- Try increasing the delay between each `FixedUpdate()` method call to reduce physics:

    - Although not to the point where user experience or game behavior is below acceptable quality!

- Wherever possible, start off rigid bodies in Sleep mode (so they don't require physics processing until woken up by code or a collision). See the Unity script reference pages for making objects go to sleep and wake up:
    - `docs.unity3d.com/ScriptReference/Rigidbody.Sleep.html`
    - `docs.unity3d.com/ScriptReference/Rigidbody.WakeUp.html`



Rigidbody 2D

Mass	1
Linear Drag	0
Angular Drag	0.05
Gravity Scale	1
Fixed Angle	☐
Is Kinematic	
Interpolate	
Sleeping Mode	Never Sleep / Start Awake / [✓ Start Asleep]
Collision Detection	Discrete

Insert image 1362OT_11_36.png

More tips for improving script efficiency

Some code strategies you might consider to improve performance include:

- Use **Structs** rather than **Classes** to improve speed up processing.
- Wherever possible, use simple arrays of primitive types rather than **ArrayLists**, **Dictionaries,** or more complex collection classes. A good article about choosing the most appropriate collection in Unity can be found at this URL: `http://wiki.unity3d.com/index.php/Choosing_the_right_collection_type`
- Raycasting is slow, so avoid performing it every frame, for example, use coroutines to only raycast every 3rd or 10th frame ...
- Finding objects is slow, so avoid finding objects in `Update()` or inner loops, and you can have objects set up a `public static` variable to allow quick instance retrieval rather than using a `Find(…)` method at all
- Avoid using `OnGUI()`, since it is called every frame just like `Update()` – this is much easier with the new Unity 5 UI system

Sources of more wisdom about optimization

Here are several other sources you might want to explore to learn more about game optimization topics:

- Unity general mobile optimization page:
  `http://docs.unity3d.com/Manual/MobileOptimisation.html`
- X-team Unity best practices:
  `http://x-team.com/2014/03/unity-3d-optimisation-and-best-practices-part-1/`
- Code Project:
  `http://www.codeproject.com/Articles/804021/Unity-and-Csharp-Performance-Optimisation-tips`
- General graphics optimization:
  `http://docs.unity3d.com/Manual/OptimizingGraphicsPerformance.html`
- University Twente - occlusion culling - and lots of other tips…:
  `http://www.unitymagic.com/shop/en/occlussion-culling-tutorial/`
- Learn more about mobile physics at Unity's iPhone optimization physics page:
  `http://docs.unity3d.com/Manual/iphone-Optimizing-Physics.html`

Published articles discussion on premature optimization

Here are several articles discussing Donald Knuth's famous quotation about premature optimization being 'evil':

- Joe Duffy's Blog:
  `http://joeduffyblog.com/2010/09/06/the-premature-optimization-is-evil-myth/`
- "When is optimisation premature?" Stack Overflow:
  `http://stackoverflow.com/questions/385506/when-is-optimisation-premature`
- "The Fallacy of Premature Optimization". Randall Hyde, published by ACM, source: Ubiquity Volume 10, Issue 3, 2009:
  `http://ubiquity.acm.org/article.cfm?id=1513451`

Sources of more about Game Managers and the State Pattern

Learn more about implementing the State Pattern and Game Managers in Unity from these sites:

- http://rusticode.com/2013/12/11/creating-game-manager-using-state-machine-and-singleton-pattern-in-unity3d/
- https://github.com/thefuntastic/Unity3d-Finite-State-Machine