10

Extra Features and Performance Optimization

In this chapter, we will cover:

- Pausing the game
- Implementing slow motion
- 3D stereography with polarized projection
- Preventing your game from running on unknown servers
- Optimisation: identify performance 'bottlenecks' with code profiling
- Optimisation: destroy objects at a 'death' time
- Optimisation: disable objects whenever possible
- Optimisation: use delegates and events (and avoid SendMessage!)
- Optimisation: regularly executed methods independent of frame rate with coroutines
- Optimisation: spread long computations over several frames with coroutines
- Optimisation: caching rather than component lookups and 'reflection' over objects

Introduction

The first three recipes in this chapter provide some ideas for adding some extra features to your game (pausing, slow motion, 3D stereography, 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 code.

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 *BallShooterGame* containing a playable scene. The package is in the folder *0423_10_01*.

How to do it...

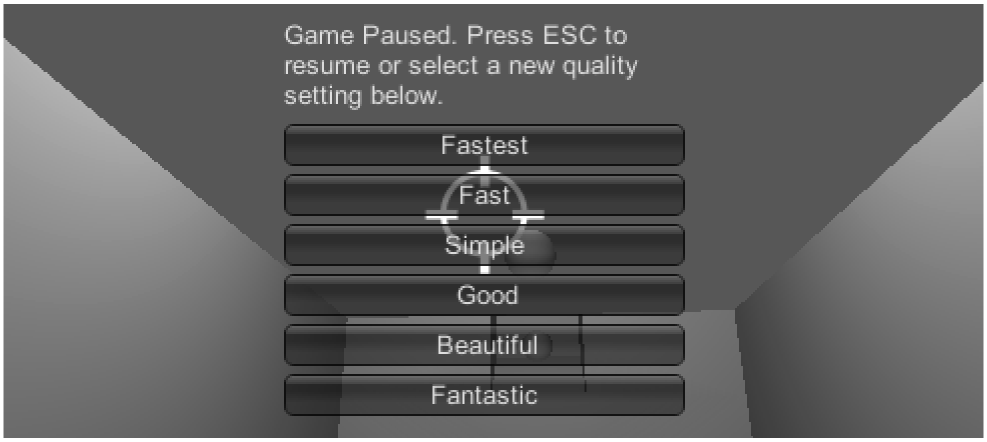
To pause your game upon pressing the ´Esc' key, follow these steps:

1. Import the package *BallShooterGame* into your project and open the level named *RoomLevel* (in folder *Scenes*).
2. Add the following C# Script *PauseGame* to *First Person Controller:*

```
// file: PauseGame.cs
using UnityEngine;
using System.Collections;

public class PauseGame : MonoBehaviour {
   public bool expensiveQualitySettings = true;
   private bool isPaused = false;

    private void Update() {
        if (Input.GetKeyDown(KeyCode.Escape)) {
        if (isPaused)
           ResumeGameMode();
        else
           PauseGameMode();
     }
   }

   private void ResumeGameMode() {
        Time.timeScale = 1.0f;
        isPaused = false;
        Screen.showCursor = false;
     GetComponent<MouseLook>().enabled = true;
    }

   private void PauseGameMode() {
        Time.timeScale = 0.0f;
        isPaused = true;
```

```
         Screen.showCursor = true;
      GetComponent<MouseLook>().enabled = false;
   }

   private void OnGUI() {
      if(isPaused)
         PauseGameGUI();
   }

   private void PauseGameGUI(){
      string[] names = QualitySettings.names;
      string message = "Game Paused. Press ESC to resume
or select a new quality setting below.";
      GUILayout.BeginArea(new Rect(0, 0, Screen.width,
Screen.height));
      GUILayout.FlexibleSpace();
      GUILayout.BeginHorizontal();
      GUILayout.FlexibleSpace();
      GUILayout.BeginVertical();
      GUILayout.Label(message, GUILayout.Width(200));

      for (int i = 0; i < names.Length; i++) {
         if
(GUILayout.Button(names[i],GUILayout.Width(200)))
            QualitySettings.SetQualityLevel(i,
expensiveQualitySettings);
      }

      GUILayout.EndVertical();
      GUILayout.FlexibleSpace();
      GUILayout.EndHorizontal();
      GUILayout.FlexibleSpace();
      GUILayout.EndArea();
   }
}
```

3. When you play the scene you should be able to pause/resume the game by pressing the *ESC* key.



Game Paused. Press ESC to resume or select a new quality setting below.

0423_10_01.png

How it works...

The value of Boolean flag variable *isPaused* is used to decide what to do when ESC is pressed. If the game is paused, the game resumes by calling *ResumeGameMode*(). If the game was not paused, then it will be paused by calling method *PauseGameMode*().

To pause the game we disable the *MouseLook* component, reveal the mouse cursor, and set variable *Time.timeScale* to zero – this sets the frame rate to zero, so if all logic is frame-based then effectively the game is paused. When the game is in pause mode (*isPaused* is true) *OnGUI*() will call *PauseGameGUI*() to use the opportunity to reveal the mouse cursor and display GUI buttons that activate different levels of quality settings.

To resume the game, we hide the mouse cursor, enable the *MouseLook* component, and set *Time.timeScale* back to 1.0.

There's more...

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

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

- Implementing slow motion

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 slow motion while the player presses down the right mouse button or the *Alt* key.

Getting ready

For this recipe, we will use the same prepared a package as the previous recipe: *BallShooterGame* in the folder *0423_10_01*. We will also need a texture for a time progress bar, which is named *barTexture.png* and can be found in the folder *0423_10_02*.

How to do it...

To implement slow-motion, follow these steps:

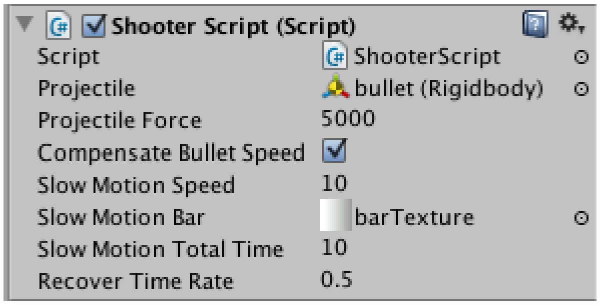
1. Start a new Unity project and Import texture image file *barTexture.png*.
2. Import the package named BallShooterGame into your project and open the level named *RoomLevel* (in folder *Scenes*).
3. We need to replace all of the exiting C# *ShooterScript* with new code to implement slow motion time effects. Open *ShooterScript* and replace everything with the following code:

```
// file: ShooterScript.cs
using UnityEngine;
using System.Collections;

public class ShooterScript : MonoBehaviour
{
   public Rigidbody projectile;
   public float projectileForce = 5000.0f;
   public bool compensateBulletSpeed = true;
   public float slowMotionSpeed = 10.0f;
   public Texture2D slowMotionBar;
   public float slowMotionTotalTime = 10.0f;
   public float recoverTimeRate = 0.5f;

   private bool isSlowMotionMode = false;
   private float remainingSloMoTime;
   private float startSloMoTimestamp;
   private float elapsedTime = 0.0f;
   private float actualForce;

   private void Start() {
      Screen.showCursor = false;
      remainingSloMoTime = slowMotionTotalTime;
      actualForce = projectileForce;
   }

   private void Update() {
      CheckUserInput();

      if( isSlowMotionMode )
         SlowMotionModeActions();
      else
         NormalTimeActions();
   }

   private void CheckUserInput() {
      if (Input.GetButtonDown("Fire1")){
         Rigidbody clone = (Rigidbody)
Instantiate(projectile, Camera.main.transform.position,
Camera.main.transform.rotation);
         clone.AddForce(clone.transform.forward *
actualForce);
      }
      if (Input.GetButtonDown("Fire2") &&
remainingSloMoTime > 0)
         ActivateSloMo();

      if (Input.GetButtonUp("Fire2"))
         DeactivateSloMo();
    }

   private void SlowMotionModeActions() {
      elapsedTime = (Time.time - startSloMoTimestamp) *
slowMotionSpeed;
      remainingSloMoTime = slowMotionTotalTime -
elapsedTime;

      if (elapsedTime >= slowMotionTotalTime)
         DeactivateSloMo();
   }

   private void NormalTimeActions() {
      if( remainingSloMoTime < slowMotionTotalTime )
```

```
         remainingSloMoTime += Time.deltaTime * 
recoverTimeRate;
      else
         remainingSloMoTime = slowMotionTotalTime;
   }

   private void ActivateSloMo() {
      Time.timeScale = 1.0f / slowMotionSpeed;
      Time.fixedDeltaTime = 0.02f / slowMotionSpeed;
      actualForce = projectileForce * slowMotionSpeed;
      isSlowMotionMode = true;

      float timeLeftToRefill = (slowMotionTotalTime - 
remainingSloMoTime)  / slowMotionSpeed;
      startSloMoTimestamp = (Time.time - 
timeLeftToRefill);
   }

   private void DeactivateSloMo() {
      Time.timeScale = 1.0f;
      Time.fixedDeltaTime = 0.02f;
      actualForce = projectileForce;
      isSlowMotionMode = false;
   }

   private void OnGUI() {
      float proportionRemaining = remainingSloMoTime / 
slowMotionTotalTime;
      float barDisplayWidth = slowMotionBar.width * 
proportionRemaining;

      int proportionRemaininInt = 
(int)(proportionRemaining * 100);
      string percentageString = proportionRemaininInt + 
"%";
      GUI.Label(new Rect(10, 15, 40, 20), 
percentageString);
      Rect newRect = new Rect(50, 10, barDisplayWidth, 
slowMotionBar.height);
      GUI.DrawTexture( newRect, slowMotionBar, 
ScaleMode.ScaleAndCrop, true, 10.0f);
   }
}
```

4. With the *First Person Controller* selected in the *Hierarchy*, drag texture image *barTexture* into the Inspector for the *Slow Motion Bar* public variable of the *ShooterScript* component.

5. Play your game. You should be able to activate slow motion by keeping the right mouse button or the *Alt* key pressed. A progress bar will slowly shrink to indicate how much slow motion time you have remaining.



☑ Shooter Script (Script)

Script	ShooterScript
Projectile	bullet (Rigidbody)
Projectile Force	5000
Compensate Bullet Speed	☑
Slow Motion Speed	10
Slow Motion Bar	barTexture
Slow Motion Total Time	10
Recover Time Rate	0.5

0423_10_02.png

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 dividing it by the *Slow Motion Speed* variable. We do the same to the variable *Time.fixedDeltaTime*, which updates the physics simulation of our game. Method ActivateSloMo() sets values for slow motion effects, and method DeactivateSloMo() returns values for normal speed time.

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 by the variable *Slow Motion Duration*). Whenever the player is not using bullet time, he has his quota filled according to the variable *Recover Rate*.

The GUI is implemented based upon finding the proportion of slow motion time remaining, and sizing the width of the bar texture accordingly.

There's more...

Some suggestions for you to improve your slow motion effect even further:

Adding Motion Blur (Unity Pro only)

Motion Blur is an image effect frequently identified with slow motion. Once attached to the camera , it could be enabled and disabled through the functions activateSloMo and deactivateSloMo, respectively. For more information on the Motion Blur image effect, access: `docs.unity3d.com/Documentation/Components/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 (provided you have Unity Pro).

See also

- Pausing the game

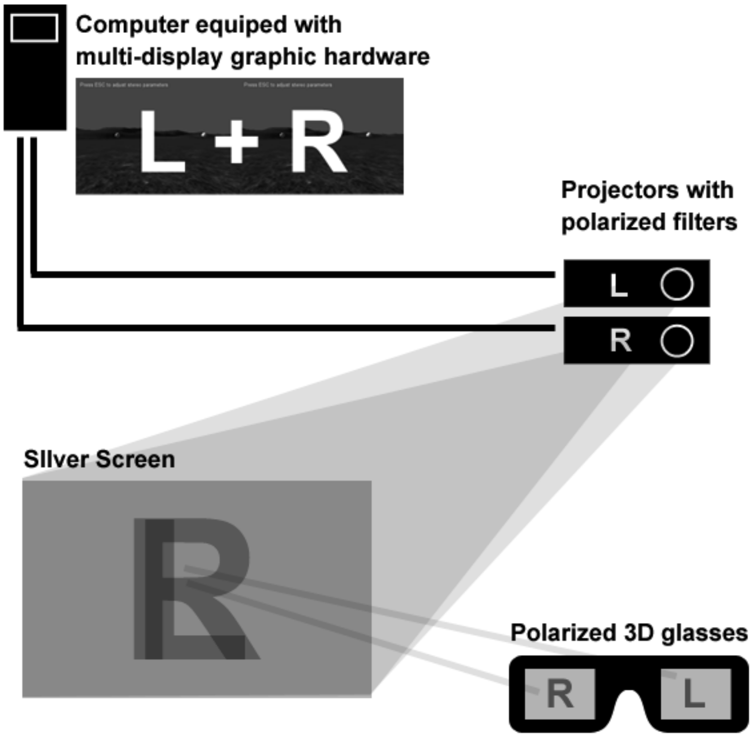
3D stereography with polarized projection

3D stereo is everywhere: from movie theaters to TV sets and portable gaming. While Unity doesn't natively support Quad Buffer technology in order to make active stereo implementation easy, you might achieve interesting results using passive stereoscopy through a polarized system. While this recipe might be of limited application, it might very useful for those working with public presentations.

Getting ready

For this recipe, we have prepared a package named *StereoScene* containing a playable scene including a *First Person Controller*, a terrain and some basic geometry. The package is in the folder *0423_10_03*.

Although you don't need any extra hardware to try this recipe, you would need, of course, an adequate projection setup to test it properly. That would include multi-display hardware such as Matrox *DualHead2Go*, a silver screen, two projectors with polarizing filters and polarized 3d glasses.



0423_10_03.png

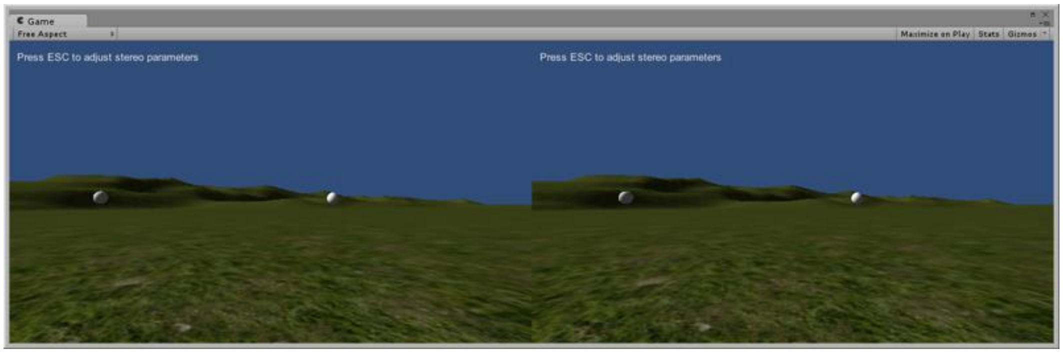
How to do it...

To crate side-by-side 3D stereo viewports, follow these steps:

1. Import the package *StereoScene* into your project and open the level named StereoCam.
2. Add the following C# Script *StereoCam* to the *Main Camera* that is a child of the *First Person Controller* game object:

```
// file: StereoCam.cs
using UnityEngine;
using System.Collections;

public class StereoCam : MonoBehaviour {
    public float parallaxDistance = 10.0f;
    public float eyeDistance = 0.05f;
    private bool showGUI = false;
    private GameObject lCamera;
    private GameObject rCamera;
    private GameObject parallax;

    void Start() {
      lCamera = new GameObject( "lCamera" );
      lCamera.AddComponent<Camera>();
      lCamera.AddComponent<GUILayer>();
      lCamera.AddComponent("FlareLayer");
      lCamera.camera.CopyFrom(Camera.main);
      rCamera = (GameObject) Instantiate(lCamera,
transform.position, transform.rotation);
      rCamera.name = "rCamera";
      lCamera.transform.parent = Camera.main.transform;
      rCamera.transform.parent = Camera.main.transform;
      parallax = new GameObject( "parallax" );
      parallax.transform.parent = Camera.main.transform;
      parallax.transform.localPosition = Vector3.zero;
      Vector3 parallaxPosition =
parallax.transform.localPosition;
      parallaxPosition.z = parallaxDistance;
      parallax.transform.localPosition =
parallaxPosition;
      camera.enabled = false;
      lCamera.camera.rect = new Rect(0, 0, 0.5f, 1);
      rCamera.camera.rect = new Rect(0.5f, 0, 0.5f, 1);
      lCamera.transform.localPosition = Vector3.zero;
      rCamera.transform.localPosition = Vector3.zero;
      Vector3 cameraPosition =
lCamera.transform.localPosition;
      cameraPosition.x = -eyeDistance;
      lCamera.transform.localPosition = cameraPosition;
      cameraPosition.x = eyeDistance;
      rCamera.transform.localPosition = cameraPosition;
   }

    void Update() {
        lCamera.transform.LookAt(parallax.transform);
        rCamera.transform.LookAt(parallax.transform);

        if (Input.GetKeyUp(KeyCode.Escape)){
            if (showGUI)
                showGUI = false;
            else
                showGUI = true;
        }

        if (showGUI){

            if (Input.GetKey(KeyCode.Alpha1))
                eyeDistance -= 0.001f;
            if (Input.GetKey(KeyCode.Alpha2))
                eyeDistance += 0.001f;

            if (Input.GetKey(KeyCode.Alpha3))
                parallaxDistance -= 0.01f;

            if (Input.GetKey(KeyCode.Alpha4))
                parallaxDistance += 0.01f;

            Vector3 cameraPosition =
lCamera.transform.localPosition;
            cameraPosition.x = -eyeDistance;
            lCamera.transform.localPosition =
cameraPosition;
            cameraPosition.x = eyeDistance;
            rCamera.transform.localPosition =
cameraPosition;
            Vector3 parallaxPosition =
parallax.transform.localPosition;
            parallaxPosition.z = parallaxDistance;
            parallax.transform.localPosition =
parallaxPosition;
        }
    }

    private void OnGUI() {
        for (int n = 0; n < 2; n++){
            if (showGUI){
                GUI.Label(new Rect((Screen.width * 0.5f *
n) + 10, 30, 500, 20), "Eye distance: " +
eyeDistance.ToString("F") + ". Press 1 to decrease and 2
to increase.");
                GUI.Label(new Rect((Screen.width * 0.5f *
n) + 10, 70, 500, 20), "parallax distance: " +
parallaxDistance.ToString("F") + ". Press 3 to decrease
and 4 to increase.");
            else
                GUI.Label(new Rect((Screen.width * 0.5f *
n) + 10, 10, 500, 20), "Press ESC to adjust stereo
parameters");
        }
    }
}
```

3. Play your scene. It should now be playing in two viewports: one for left eye and one for the right.



Press ESC to adjust stereo parameters

Press ESC to adjust stereo parameters

0423_10_04.png

4. Build the executable and run it in full screen mode on your dual projector system, making sure it behaves as a very wide single display, not two different displays side-by-side. Left and Right viewports should superimposed, and not side-by-side.

How it works...

Once attached to the Main Camera, our script creates three objects: two cameras (left and right) and an empty object named *parallax*. These objects are made children of the Main Camera, so they move and rotate accordingly. The stereographic effect is given by the distance between the left and right cameras, and also by how their focal points converge at the parallax. Also, we have used a *for* loop to make sure GUI elements are displayed on both viewports.

Since you might want to adjust the distance and parallax of your cameras, depending on your preferences and final result on your projection environment, we've made those variables adjustable (press the ESC key to read the instructions).

There's more...

Using Render to Texture (Unity Pro only)

As an alternative to having two viewports, you could get a similar result using Render to Texture, as proposed by Paul Bourke at

Using anaglyph resources

In case you prefer to develop an anaglyph stereoscopic effect, there are some products you should take a look at, such as:

- 3D Anaglyph System, by Azer2K. Available at `http://u3d.as/content/azert2k/3d-anaglyph-system/1AE.`
- Stereoskopix' *FOV2GO* at Unity's assets store, available at `http://u3d.as/content/stereoskopix/fov2go/2HA`.

Using NVidia 3D Vision

Alternatively, you could develop your game targeting NVidia's solution *3D Vision*. You can find, at the *Unity Asset Store*, a dedicated resource developed by Dembeta SL named *Active Sterescopic 3D for Unity*. Available at: `http://u3d.as/content/dembeta-sl/active-stereoscopic-3d-for-unity/2Vt`

Acquiring active stereo third-party software

In case you need a more robust solution for active stereo and VR, you could check out some third-party software:

- Mechdyne sells a solution named *getReal3D for Unity*, available at `http://www.mechdyne.com`.
- I'm in VR develops *MiddleVR for* Unity, available at: `http://www.imin-vr.com/middlevr-for-unity/`

Preventing your game from running on unknown servers

After all the hard work you've had to complete your web game project, it wouldn't be fair if it ended up generating traffic and income for 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:

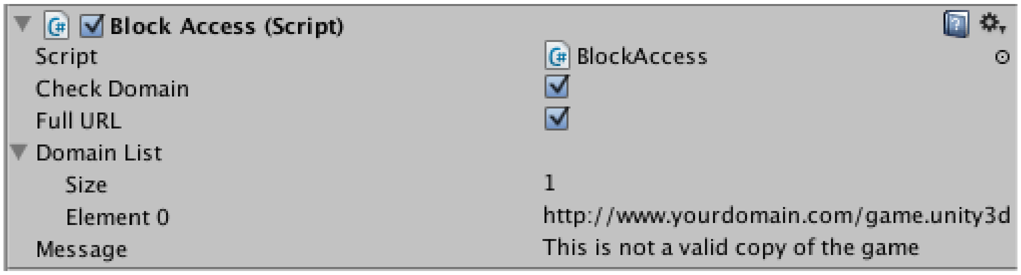
3 – Add the following *C# Script* to the Main Camera:

```
// file: BlockAccess
using UnityEngine;
using System.Collections;
```

```
public class BlockAccess : MonoBehaviour {
    public bool checkDomain = true;
    public bool fullURL = true;
    public string[] DomainList;
    public string message;
    private bool illegalCopy = true;

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

    private void OnGUI() {
        if (illegalCopy)
            GUI.Label(new Rect(Screen.width * 0.5f - 200,
Screen.height * 0.5f - 10, 400, 32), message);
        else
        Application.LoadLevel(Application.loadedLevel +
1);
    }
}
```

1. Now fill out its variables. 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.

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



☑ Block Access (Script)

Script	BlockAccess
Check Domain	☑
Full URL	☑
Domain List	
Size	1
Element 0	http://www.yourdomain.com/game.unity3d
Message	This is not a valid copy of the game

0423_10_05.png

2. Save your scene as *menu*.
3. Create a new scene and change its Main Camera background color to black. Save this scene as *nextLevel*.
4. 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 screen.

There's more...

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

Be specific!

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`.

Allow redistribution

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 Server* unchecked.

Optimisation: identify performance 'bottlenecks' with code profiling

Optimisation principal 1: Work smart not hard. The best performance improvements are achieved when effort is focused to optimize those parts of the game that take up the most computer resources – the 'bottlenecks'. Code profiling involves identifying resource bottlenecks by analyzing how long different actions take to complete their tasks. For graphics-intensive operations, the 'Stats' panel of the game window can be used, to record frame rates and 'dropped calls'. For memory and processor intensive code Unity-Pro offers detailed, frame by frame, code profiling. Finally, for those who do not have Unity-Pro, there is a freely available code profiling class from the 'Unify' community wiki, which we demonstrate in this recipe. Code profiling involves timing how long it takes between starting to do something, and ending it. Individual statements or loops may be profiled within a method, or the duration of a complete method may be analysises, or the time between one event and some other (e.g. the time for Unity to be asked to load a new scene, and for that scene to then load and start running).

> The Unity C# code profiler demonstrated in this recipe is by Michael Garforth, and can be found at: `http://wiki.unity3d.com/index.php/Profiler`

Getting ready

The C# script required for this recipe can be found in folder 0423_10_05:

- Profile.cs

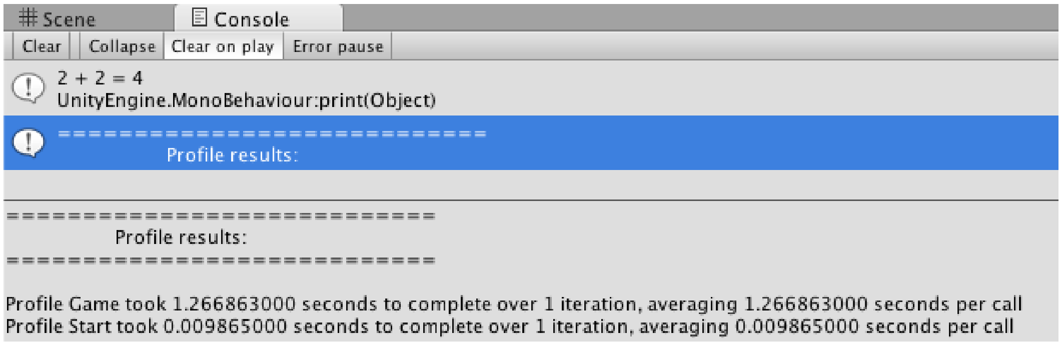
How to do it...

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

```
// file: ProfileScript.cs
using UnityEngine;

public class ProfileScript : MonoBehaviour
{
   private void Awake() {
      Profile.StartProfile("Game");
   }

   private void Start(){
     Profile.StartProfile("Start");
     int answer = 2 + 2;
     print("2 + 2 = " + answer);
     Profile.EndProfile("Start");
   }

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

How it works...

The Profile class defines 3 important static methods that can be called to record times for profiling: *StartProfile*(<taskName>), *EndProfile*(< taskName>), and *PrintResults*(). Different tasks can be timed by choosing an appropriate string (such as ‘Game’), and calling *StartProfile*() when they begin, and *EndProfile*() after they have completed. Finally, *PrintResults*() tells the Profile class to display the results of all recordings in the console window.



```
2 + 2 = 4
UnityEngine.MonoBehaviour:print(Object)

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

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

Profile Game took 1.266863000 seconds to complete over 1 iteration, averaging 1.266863000 seconds per call
Profile Start took 0.009865000 seconds to complete over 1 iteration, averaging 0.009865000 seconds per call
```

0423_10_06.png

Our *ProfileScript* object added to the Main Camera requests two named tasks be profiled: the simple 2 + 2 calculation in the *Start*() method (tagged ‘Start’); and that the time between the *Awake*() method being called, and the application quitting because the user stopped game execution (tagged ‘Game’). As can be seen in the figure, the profile for ‘Start; shows that the time taken was 0.009865000 seconds; and for ‘Game’ it was 1.266863000 seconds.

This recipe illustrates a straightforward way to measure the time taken between different points in your game. If the same string ‘tags’ are used for *StartProfile*() and *EndProfile*() many times (e.g. in a loop), then the Profile class will output the average time the profiled task took to complete.

There's more...

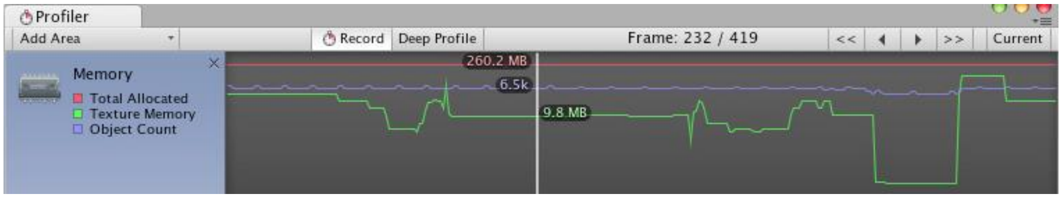
Some details you don't want to miss:

Unity Pro only – code profiling

If you have Unity Pro then the Unity code Profiler can provide much more detailed timing analysis of your game, on a frame-by-frame basis. For exampled (see screenshot) the number of objects in each frame can be analysed (the middle line in the figure).

For more information see Unity's documentation pages at the following URL:

`http://docs.unity3d.com/Documentation/Manual/Profiler.html`



0423_10_07.png

Optimisation: destroy objects at a 'death' time

Optimisation principal 2: Minimise the number of objects in a scene. 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 OnGUI() messages, or consider collisions or physics for such objects etc.

However, there may be times when we wish not to destroy an object immediately, but at some known future point in time. Examples might include after a sound has finished playing (see Chapter 6 for just such a recipe); or perhaps the player only has a certain time to collect an bonus object before it disappears; or perhaps an object displaying a message to the player should disappear after a certain time (See Chapter 4 for such a recipe).

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

How to do it...

1. Create a Cube named *Cube*
2. Add the following script class DeathTimeExample.cs to *Cube*:

```
// file: DeathTimeExample.cs
using UnityEngine;
using System.Collections;

public class DeathTimeExample : MonoBehaviour {
   public float deathDelay = 4f;
   private float deathTime = -1;

   private bool menuMode = true;

   private void OnGUI(){
      if( menuMode )
         GUIMenu();
      else
         GUITimeDisplay();
   }

   private void GUITimeDisplay(){
      float timeLeft = Time.time - deathTime;
      GUILayout.Label("time left before death = " +
timeLeft);

   }

   private void GUIMenu(){
      bool startDyingButtonClicked =
GUILayout.Button("Start dying");
      if( startDyingButtonClicked ){
         StartDying();
         menuMode = false;
      }
   }

   private void Update() {
      if( deathTime > 0 && Time.time > deathTime )
         Destroy( gameObject );
   }

   private void StartDying() {
      deathTime = Time.time + deathDelay;
   }
}
```

How it works...

The *OnGUI*() method initially presents a button to the user, which when clicked causes the *StartDying*() method to be called. After the button is clicked the time left before the object destroys itself is displayed. Which of the two GUI methods is executed depends on the value of Boolean variable *menuMode*.

The float variable *deathDelay* stored the number of seconds the object is wait 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 positive value, which is the time we wish the object to destroy itself. When method *StartDying*() is called, it sets this *deathTime* variable to the current time plus whatever value is set in *deathDelay*.

Every frame method *Update*() calls the *CheckDeath*() method. *CheckDeath*() tests if *deathTime* greater than zero (i.e. a death time has been set), and then 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 Cube GameObject removed from the Hierarchy once its death time has been reached.

See also

- Optimisation: disable objects whenever possible

Optimisation: disable objects whenever possible

Optimisation principal 3: Minimise the number of 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 Monobehavior script is disabled, then Unity no longer needs to send the object Update() and OnGUI() messages 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 / into the visible area for one or more cameras in the scene.

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

How to do it...

1. Create a new Unity project, importing the *Character Controller* package
2. Create a new terrain
3. Remove the *Main Camera*
4. Add a *3rdPersonController* in the center of the terrain
5. Create a new *Cube* just in front of your *3rdPersonController*
6. Add the following C# script class to your *Cube*:

```
// file: DisableWhenInvisible
using UnityEngine;
using System.Collections;

public class DisableWhenInvisible : MonoBehaviour {
   public Transform player;

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

    void OnBecameInvisible() {
        enabled = false;
     print ("cube became in-visible");
   }

   private void OnGUI() {
     float d = Vector3.Distance( transform.position, 
player.position);
     GUILayout.Label ("distance from player to cube = " 
+ d);
   }
}
```

7. With the Cube selected in the Hierarchy drag your *3rdPersonController* into the Inspector for the public variable *player*.

How it works...

When visible the scripted *DisableWhenInvisible* component of the Cube recalculates and displays the distance from itself to the *3rdPersonController's* transform, via the variable *player* in method OnGUI() 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() and OnGUI() messages to the object, so the distance calculation in OnGUI() 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() and OnGUI() messages each frame. Note 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.

There's more...

Some details you don't want to miss:

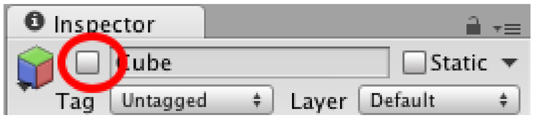
Another common case – only enable after OnTrigger()

Another common situation is that we only want an scripted component to be active if the player's character is nearby (within some minimum distance). In these situations a sphere collider can be setup, and the scripted component can be enabled only when the player's character enters that sphere. This can be implemented using the OnTriggerEnter() and OnTriggerExit() events; for example:

```
private void OnTriggerEnter(Collider hitObjectCollider) {
   if (hitObjectCollider.gameObject.CompareTag("Player"))
      enabled = true;
}

private void OnTriggerExit(Collider hitObjectCollider) {
   if (hitObjectCollider.gameObject.CompareTag("Player"))
      enabled = false;
}
```

Go one step further – make the parent GameObject inactive

In some cases you can go one step further, and make the parent GameObject that contains the scripted component 'inactive'. This is just like deselecting the checkbox next to the GameObject in the Inspector



□ Cube □ Static

Tag Untagged Layer Default

0423_10_08.png

To deactivate the parent GameObject you set its *active* property to false; e.g.:

```
void OnBecameInvisible() {
   gameObject.active = false;
    print ("cube became in-visible");
}
```

Note, however, 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's *active* property back to true. The following script, when added to an active GameObject (such as terrain or controller), will display a button to demonstrate how an inactive object can be reactivated. For this script to work, the public *cubeGO* variable must be set by dragging the cube over the variable in the Inspector:

```
// file: MakeCubeActive.cs
using UnityEngine;
using System.Collections;

public class MakeCubeActive : MonoBehaviour {
   public GameObject cubeGO;

   private void OnGUI(){
      bool makeCubeActiveButtonClicked =
GUILayout.Button("make cube active");
      if( makeCubeActiveButtonClicked )
         cubeGO.active = true;
   }
}
```

See also

- Optimisation: destroy objects at a 'death' time

Optimisation: use delegates and events (and avoid SendMessage!)

When events can be based on distance or collisions, we can use OnTrigger methods as described in the previous recipe. When events can be based on time periods we can use coroutines as described in a later recipe in this chapter. When there are other kinds of events, we can use C# delegates and events, as described in this recipe. These work in a similar way to *SendMessage*(), but are much more efficient, since Unity has a defined list of which objects are 'listening' to the broadcast events. *SendMessage*() should be avoided, since it means 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. 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 displaying some color change buttons, and publishing a new event for each button clicked; we’ll also have a cube and a sphere that subscribe to the color change events, so each time a color change event is published, both the cube and sphere should receive the event message and change their color accordingly. Publishers 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...

1. Add the following C# script class *ColorManager* the *Main Camera*:

```
// file: ColorManager.cs
using UnityEngine;
using System.Collections;

public class ColorManager : MonoBehaviour {
   public delegate void ColorChangeHandler(Color
newColor);
   public static event ColorChangeHandler
changeColorEvent;

   void OnGUI(){
      bool makeGreenButtonClicked =
GUILayout.Button("make things GREEN");
      bool makeBlueButtonClicked = GUILayout.Button("make
things BLUE");
      bool makeRedButtonClicked = GUILayout.Button("make
things RED");

      if(makeGreenButtonClicked)
         PublishColorChangeEvent( Color.green );

      if(makeBlueButtonClicked)
         PublishColorChangeEvent( Color.blue );

      if(makeRedButtonClicked)
         PublishColorChangeEvent( Color.red );
   }
   private void PublishColorChangeEvent(Color newColor){
      // if there is at least one listener to this event
      if(changeColorEvent != null){
         // broadcast change color event
         changeColorEvent( newColor );
      }
   }
}
```

2. Create a Cube at (0,0,0), and attach the following C# script class ColorChangeListener to it:

```
// file: ColorChangeListener.cs
using UnityEngine;
using System.Collections;

public class ColorChangeListener : MonoBehaviour {
   void OnEnable() {
      ColorManager.changeColorEvent += OnChangeColor;
   }

   private void OnDisable(){
      ColorManager.changeColorEvent -= OnChangeColor;
   }

   void OnChangeColor(Color newColor){
      renderer.sharedMaterial.color = newColor;
   }
}
```

3. Create a sphere at (-5, 0, 0), and attach a *ColorChangeListener* component to it
4. Create a directional light, so we can see the cube and sphere colors easily.

How it works...

First let’s consider what we want to happen – we want the cube and sphere to both change their color when they receive an event message OnChangeColor() with a new color argument.

This is achieved by each object instance of the ColorChangeListener class (there is one as a component of the cube and one for the sphere), subscribing their OnChangeColor() methods to listen for color change events published from the ColorManager class. 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 ColorChangeListener objects are disabled, those methods are removed (-=) from the list of event subscribers.

When a ColorChangeListener object receives a color change message, its subscribed OnChangeColor() method is executed, and the color of the shared materials of the renderer 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 ColorChangeListener 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 (green/red/blue) to all subscribed object methods (I.e. the cube and sphere).

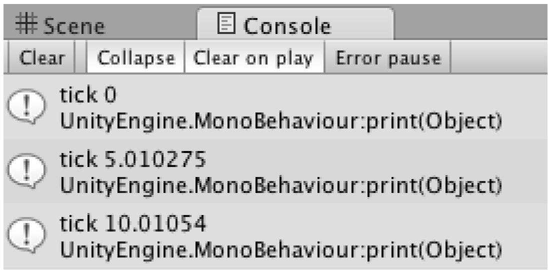
The ColorManager class declares a ‘delegate’ named ColorChangeHandler. Delegates define the return type (in this case void) and argument ‘signature’ of methods that can be delegated (subscribed) to an event. In this case methods must have the argument signature of a single parameter of type *Color*. Our OnChangeColor() method in class ColorChangeListener matches this argument signature, and so it is 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`

Optimisation: regularly executed methods independent of frame-rate with coroutines

Optimisation principal 4: 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)
```

0423_10_09.png

How to do it...

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

```
// file: TimedMethod.cs
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 given, which causes the method to suspend execution for the given delay of 5 seconds. After the yield instruction has completed, the loop will execute once again, and so on.

You'll may have noticed: **there is no Update() method at all.** So although our game has logic being regularly executed, in this exampled 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

Coroutine 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*(); logic that works find 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 analysing (and testing) different game logic to identify the least frequent execution that is still acceptable

See also

- Optimisation: spread long computations over several frames with coroutines

Optimisation: spread long computations over several frames with coroutines

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. 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...

1. Add the following script class ColorManager.cs the *Main Camera*:

```
// file: SegmentedCalculation.cs
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 = 9999;
      int max = -1;

      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. When you run the scene you should see something like the following:

```
frame:  1, i:0, min:9999, max:-1
frame:  2, i:10, min:30, max:793
frame:  3, i:20, min:30, max:892
frame:  4, i:30, min:4, max:892
frame:  5, i:40, min:4, max:905
** completed - disabling scripted component
```

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. Each 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:

Retrieve 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
- Windows: C:\Users\username\AppData\Local\Unity\Editor\Editor.log

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

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

See also

- Optimisation: regularly executed methods independent of frame rate with coroutines

Optimisation: caching rather than component lookups and 'reflection' over objects

Optimisation principal 5: Minimise actions requiring Unity to perform 'reflection' over objects. Reflection is when at run-time, Unity has to analyze objects to see whether they contain a given methods or components, an example would be the simple and useful, but slow, *FindObjectsByTag*(). Another action that slows Unity down is each time we make it lookup an object's component, either explicitly using *GetComponent*(), or implicitly using a component accessor variable such as 'transform' or 'renderer'. 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 the half the distance from the Main Camera (to which the script is attached) to another GameObject in the scene tagged 'Player'.

Getting ready

For this recipe we have provided C# script Profile.cs in folder 0423_10_11.

How to do it...

1. Start a new project, and import the script *Profile.cs*
2. Create a Cube named *Cube-Player* at (1,1,1), and give it the tag 'Player'
3. Add the following C# script class *SimpleMath* the *Main Camera*:

```
// file: SimpleMath.cs
using UnityEngine;
using System.Collections;

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

4. Add the following C# script class *ProfileScript1* the *Main Camera*:

```
// file: ProfileScript1.cs
using UnityEngine;
using System.Collections;

public class ProfileScript1 : MonoBehaviour
{
   private int ITERATIONS = 2000;
   private void Start(){
      for(int i=0; i < ITERATIONS; i++){
         FindDistanceMethod();
      }

      Profile.PrintResults();
   }

   private void FindDistanceMethod(){
      Profile.StartProfile("Method-1");
      Vector3 pos1 = transform.position;
      Transform playerTransform =
GameObject.FindGameObjectWithTag("Player").transform;
      Vector3 pos2 = playerTransform.position;
      float distance = Vector3.Distance(pos1, pos2);
      SimpleMath mathObject = GetComponent<SimpleMath>();
      float halfDistance = mathObject.Halve(distance);
      Profile.EndProfile("Method-1");
   }
}
```

5. When you run the scene you should see something like the following:

```
Profile Method-1 took 0.012487000 seconds to complete over 2000 iterations,
averaging 0.000006244 seconds per call
= 6.243500000 micro-seconds per call
```

6. FindGameObjectsWithTag() is slow, so let’s fix that by adding a public Transform variable named playerTransformCache, and dragging *Cube-Player* over this variable in the Inspector when the Main Camera is selected:

```
public Transform playerTransformCache;
```

7. Now replace your FindDistanceMethod() with the following, which makes used of this cached reference to Cube-Player’s transform, avoiding the slow object-tag reflection lookup altogether:

```
   private void FindDistanceMethod(){
      Profile.StartProfile("Method-2");
      Vector3 pos1 = transform.position;
      Vector3 pos2 = playerTransformCache.position;
      float distance = Vector3.Distance(pos1, pos2);
      SimpleMath mathObject = GetComponent<SimpleMath>();
      float halfDistance = mathObject.Halve(distance);
      Profile.EndProfile("Method-2");
   }
```

8. That should run faster (around 25-35% faster). But wait! Let’s improve things some more. At the moment to find *pos1* we are making Unity find the transform

component of the *Main Camera* every time the method is called. Since the camera is not moving, lets cache this *Vector3* position as follows:

```
  private Vector3 pos1Cache = new Vector3();

  private void Awake(){
    pos1Cache = transform.position;
  }
```

9. Now replace your FindDistanceMethod() with the following, which makes used of this cached Main Camera position:

```
  private void FindDistanceMethod(){
    Profile.StartProfile("Method-3");
    Vector3 pos2 = playerTransformCache.position;
    float distance = Vector3.Distance(pos1Cache, pos2);
    SimpleMath mathObject = GetComponent<SimpleMath>();
    float halfDistance = mathObject.Halve(distance);
    Profile.EndProfile("Method-3");
  }
```

10. That should improve things around 10%. But we can still improve things – you'll notice an explicit GetComponent() call to get a reference to our *mathObject*. Let's cache this scripted component reference as well, to save GetComponent() for each iteration. We'll declare a variable *mathObjectCache* and in Awake() we will set to refer to our SimpleMath scripted component.

```
  private SimpleMath mathObjectCache;

  private void Awake(){
    pos1Cache = transform.position;
    mathObjectCache = GetComponent<SimpleMath>();
  }
```

11. Now replace your FindDistanceMethod() with the following, which makes used of this cached component reference

```
  private void FindDistanceMethod(){
    Profile.StartProfile("Method-4");
    Vector3 pos2 = playerTransformCache.position;
    float distance = Vector3.Distance(pos1Cache, pos2);
    float halfDistance =
mathObjectCache.Halve(distance);
    Profile.EndProfile("Method-4");
  }
```

12. After running the scene after each improved method you should have seen something like the following, which gives us quantitative evidence of how the

performance of our method is improving after each step in our refactoring process:

```
Profile Method-2 took 0.006818000 seconds to complete over 2000 iterations, averaging 0.000003409 seconds per call = 3.409000000 micro-seconds per call

Profile Method-3 took 0.006009000 seconds to complete over 2000 iterations, averaging 0.000003005 seconds per call = 3.004500000 micro-seconds per call

Profile Method-4 took 0.003555000 seconds to complete over 2000 iterations, averaging 0.000001778 seconds per call = 1.777500000 micro-seconds per call
```

How it works...

This recipe illustrates how we try to cache references once, before any iterations, for variables whose value will not change – such as references to GameObjects and their components, and in this example the Vector3 position of the *Main Camera*. Through this removing of implicit and explicit component and object lookups form a method to be performed many times, the performance improvement is clear to see.

NOTE: Two good places to learn more about Unity performance optimisation techniques are from the Performance Optimisation web page in the Unity Script Reference; and Unity's Joachim Ante's Unite07-presentation by "Performance Optimization". Some of the recipes in this chapter had their origins from suggestions in these sources:

`docs.unity3d.com/Documentation/ScriptReference/index.Performance_Optimization.html`

`unity3d.com/support/old-resources/unite-presentations/performance-optimization`