7

External resource files and devices

In this chapter, we will cover:

- Loading external resource files – by Unity Default Resources
- Loading external resource files – by manually storing files in Unity Resources folder
- Loading external resource files – by downloading file from internet
- Saving and loading player data – using static properties
- Saving and loading player data – using PlayerPrefs
- Saving screenshots from the game
- Control characters in Unity with the Microsoft KINECT using the Zigfu samples
- Animating your own characters with the Microsoft KINECT controller
- Homemade mocap by storing movements from the Microsoft KINECT controller
- Setting up a leaderboard using PHP/MySQL

Introduction

External data can make your game better in many ways: it might add renewable content, help file organization and allow user preferences to be set. Also, it can make Unity talk with other peripherals. In every way, it can deliver a richer experience to players and developers alike. There are several different places from which external text files can be stored and read from.

In this chapter we present a wide range of uses for external data.

Loading external resource files – by Unity Default Resources

In this recipe, we will load an external image file and display it on the screen, using the Unity Default *Resources* file (a library created at the time the game is compiled).

NOTE: This method is perhaps the simplest way to store and read external resource files. However, it is only appropriate when the contents of the resource files **will not change after compilation**, since the contents of these text files are combined and compiled into the *resources.assets* file.

The *resources.assets* file can be found in the Data folder for a compiled game.

Getting ready

In folder *0423_07_01* we have provided the following for this recipe: an image file, a text file and an audio file in Ogg format:

- *externalTexture.jpg*
- *cities.txt*
- *soundtrack.ogg*

How to do it...

1 – In the *Project* window, create a new folder and rename it *Resources*.

1. Import the *externalTexture.jpg* file and place it into the *Resources* folder.

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

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

public class ReadDefaultResources : MonoBehaviour {
   public string fileName = "externalTexture";
   private Texture2D externalImage;

   private void Start () {
      externalImage = 
(Texture2D)Resources.Load(fileName);
   }

   private void OnGUI() {
        GUILayout.Label(externalImage);
    }
}
```

2. Play the scene. The texture will be loaded and displayed on the screen.
3. If you have another image file, put a copy into in the Resources folder, then in the Inspector 6 – change the public file name to the name of your image file and play the scene again. The new image should now be displayed.

How it works...

The *Resources.Load(fileName)* statement makes Unity look inside its compiled project data file *resources.assets* for the contents of file named 'externalTexture'. The contents are returned as a texture image, which is stored into variable *externalImage*. Our OnGU() method displays *externalImage* as a Label().

NOTE: The filename string passed to Resources.Load() does NOT include the file extension (such as .jpg or .txt).

There's more...

Some details you don't want to miss:

Loading text files with this method

You can load external text files using the same approach. The private variable needs to be a String (to store the text file contents). The *Start*() method uses a temporary *TextAsset* object to receive the text file contents, and the *text* property of this object contains the String contents to be stored in the private variable:

```
public class ReadDefaultResources : MonoBehaviour {
    public string fileName = "textFileName";
    private string textFileContents = "(file not found
yet)";

    private void Start () {
        TextAsset myObject  =
(TextAsset)Resources.Load(fileName);
        textFileContents = myObject.text;
    }

    private void OnGUI() {
        GUILayout.Label ( textFileContents );
    }
}
```

Loading and playing audio files with this method

You can load external audio files using the same approach. The private variable needs to be an AudioClip:

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

[RequireComponent (typeof (AudioSource))]

public class ReadDefaultResources : MonoBehaviour {
   public string fileName = "soundtrack";
   private AudioClip audioFile;

   void  Start (){
      audio.clip = (AudioClip)Resources.Load(fileName);
      if(!audio.isPlaying && audio.clip.isReadyToPlay)
         audio.Play();
   }

   private void OnGUI () {
      bool playButtonClicked = GUILayout.Button("Play");
      bool pauseButtonClicked =
GUILayout.Button("Pause");
      bool stopButtonClicked = GUILayout.Button("Stop");

      if(playButtonClicked){ audio.Play(); }
      if(pauseButtonClicked){ audio.Pause(); }
      if(stopButtonClicked){ audio.Stop(); }
   }
}
```

See also

- Loading external resource files – by manually storing files in Unity Resources folder
- Loading external resource files – by downloading file from internet

Loading external resource files – by manually storing files in Unity Resources folder

At times the contents of external resource files may need to be changed after game compilation. Hosting resource files on the web may not be an option. There is a method of manually storing and reading files from the *Resources* folder of the compiled game – which allows for those files to be changed after game compilation.

NOTE: This technique only works when you compile to a Windows or Mac *stand alone executable* – it will not work for Web Player builds for example.

Getting ready

Folder *0423_07_01* provides the texture image you can use for this recipe:

- *externalTexture.jpg*

How to do it...

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

```
// file: ReadManualResourceImageFile.cs
using UnityEngine;
using System.Collections;
using System.IO;

public class ReadManualResourceImageFile : MonoBehaviour
{
   private string fileName = "externalTexture.jpg";
   private string url;
   private Texture2D externalImage;

   private void Start () {
      url = "file:" + Application.dataPath;
      url = Path.Combine(url, "Resources");
      url = Path.Combine(url, fileName);

      StartCoroutine( LoadWWW() );
   }

   private void OnGUI() {
      GUILayout.Label ( "url = " + url );
      GUILayout.Label ( externalImage );
   }

   private IEnumerator LoadWWW(){
      yield return 0;
      WWW imageFile = new WWW (url);
      yield return imageFile;
        externalImage = imageFile.texture;
   }
}
```

2 – Build your (Windows or Mac) standalone executable.

3 – Copy image *externalTexture.jpg* into your standalone's *Resources* folder.

NOTE: You will need to place the files in the Resources folder manually after **every** compilation.

WINDOWS and LINUX: When you create a Windows or Linux standalone executable, there is also a _Data folder created with executable application file. The Resources folder can be found inside this data folder.

MAC: A Mac standalone application executable looks like single file, but it is actually a MacOS 'package' folder. Right-click the executable file and select Show Package Contents. You will then find the standalone's Resources folder inside the Contents folder.

3 – Run your standalone game application, and the image should be displayed.

How it works...

Note the need to use the *System.IO* package for this recipe.

When the executable runs the WWW object spots that the URL starts with the word *file* and so Unity attempts to find the external resource file in its *Resources* folder and then load its contents.

There's more...

Some details you don't want to miss:

Always use Path.Combine() rather than '/' or '\'

The filepath folder separator character is different for Windows and Mac file systems ('\' backslash for Windows, '/' forward slash for the Mac). However, Unity knows which kind of standalone you are compiling your project into, therefore the Path.Combine() method will insert the appropriate separator slash character to form the file URL that is required.

Use this approach to load a text file

To use this technique to load an external text file you'll need a private string variable:

```
private string textFileContents = "(still loading file
...)";
```

Replace your LoadWWW() method with one that extracts the text of the loaded resource and stores in your private string variable:

```
private IEnumerator LoadWWW(){
    yield return 0;
    WWW textFile = new WWW (url);
    yield return textFile;
      textFileContents = textFile.text;
}
```

Then change *OnGUI()* to display the string as a Label:

```
private void OnGUI() {
    GUILayout.Label ( "url = " + url );
    GUILayout.Label ( textFileContents );
}
```

WWW and resource contents

The WWW class defines several different properties and methods to allow downloaded media resource file data to be extracted into appropriate variables for use in the game. The most useful of these include:

- .text
    - a read-only property returning web data as a string
- .texture
    - a read-only property returning web data as a Texture2D image
- .GetAudioClip()
    - a method that returns web data as an AudioClip

NOTE: For more information about the Unity WWW class see:

`docs.unity3d.com/Documentation/ScriptReference/WWW.html`

See also

- Loading external resource files – by Unity Default Resources
- Loading external resource files – by downloading file from internet

Loading external resource files – by downloading file from internet

One way to store and read text file data is to store the text files on the web. In this recipe the contents of a text file for a given URL is downloaded and read and then displayed.

Getting ready

For this recipe you need to have access to files on a web server. If you run a local web server such as Apache, or have your own web hosting, then you could use the files in folder *0423_07_01* and the corresponding URL. Otherwise you may find the following URLs useful, since they are the web locations of an image file (a Packt logo) and a text file (an 'ASCII-art' badger picture):

`www.packtpub.com/sites/default/files/packt_logo.png`

`www.ascii-art.de/ascii/ab/badger.txt`

How to do it...

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

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

public class ReadWebImageTexture : MonoBehaviour {
   public string url =
"http://www.packtpub.com/sites/default/files/packt_logo.p
ng";
   private Texture2D externalImage;

   private void Start () {
      StartCoroutine( LoadWWW() );
   }

   private void OnGUI() {
      GUILayout.Label ( "url = " + url );
      GUILayout.Label ( externalImage );
   }

   private IEnumerator LoadWWW(){
      yield return 0;
      WWW imageFile = new WWW (url);
      yield return imageFile;
```

```
        externalImage = imageFile.texture;
    }
```

2 – Play the scene. Once downloaded, the contents of the image file will be displayed.

How it works...

When the game starts, our Start() method starts the co-routine method *LoadWWW()*. A co-routine is a method that can keep on running in the background without halting or slowing down the other parts of the game and the frame rate. The *yield* statement indicates that once a value can be returned for *imageFile* the remainder of the method can be executed – i.e. until the file has finished downloading no attempt should be made to extract the *texture* property of the WWW object variable.

Each frame our OnGUI() method displays the contents of string variable *url* and attempts to display the texture image *externalImage*

See also

- Loading external resource files – by Unity Default Resources
- Loading external resource files – by manually storing files in Unity Resources folder

Saving and loading player data – using static properties

Keeping track of the player's progress and user settings during a game is vital to give your game a greater feel of depth and content. In this recipe we will learn how to make our game 'remember' the player's name and score between different levels (scenes). Also, we will give the player the option of erasing his data.

Getting ready

We have included a complete project named *BlueSpheres* in the folder *0423_07_04*. In order to follow this recipe, make a copy of that folder as your starting point.

How to do it...

To save and load player data, follow these steps:

1. Open the *BlueSpheres* project and make yourself familiar with the game by playing it a few times and examining the contents of the scenes. The game starts on scene *menu_start*, inside folder *Scenes*.
2. Create a new C# script *Player* with the following code:

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

public class Player : MonoBehaviour {
   public static string username = null;
   public static int score = -1;

   public static void DeleteAll(){
      username = null;
      score = -1;
   }
}
```

3. Replace the contents of C# script *MenuStart* with the following code:

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

public class MenuStart : MonoBehaviour
{
   const string DEFAULT_PLAYER_NAME = "PLAYER_NAME";
   private string playerNameField = DEFAULT_PLAYER_NAME;

   private void OnGUI() {
      string rules = "Easiest Game Ever -- Click the blue
spheres to advance.";
      GUILayout.Label(rules);

      if(Player.username != null)
         WelcomeGUI();
      else
         CreatePlayerGUI();
   }

   private void WelcomeGUI() {
      string welcomeMessage = "Welcome, " +
Player.username + ". You currently have " + Player.score
+ " points.";
         GUILayout.Label(welcomeMessage);

      bool playButtonClicked = GUILayout.Button("Play");
      bool eraseButtonClicked = GUILayout.Button("Erase
Data");

      if( playButtonClicked )
         Application.LoadLevel(1);

      if( eraseButtonClicked )
         ResetGameData();
   }

   private void ResetGameData() {
      Player.DeleteAll();
      playerNameField = DEFAULT_PLAYER_NAME;
   }

   private void CreatePlayerGUI() {
       string createMessage = "Please, insert your
username below and click on Create User";
      GUILayout.Label(createMessage);

       playerNameField =
GUILayout.TextField(playerNameField, 25);

       bool createUserButtonClicked=
GUILayout.Button("Create User");

      if( createUserButtonClicked ){
         Player.username = playerNameField;
         Player.score = 0;
      }
   }
}
```

4. Replace the contents of C# script *SphereClick* with the following code:

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

public class SphereClick : MonoBehaviour
{
   private void OnGUI(){
      GUILayout.Label( "Score: " + Player.score );
   }

   private void OnMouseDown() {
      if( gameObject.CompareTag("blue") )
         Player.score += 50;

         Destroy(gameObject);
      GotoNextLevel();
   }

   private void GotoNextLevel() {
      int level = Application.loadedLevel + 1;
         Application.LoadLevel(level);
    }
}
```

5. Replace the OnGUI() method of C# script *MenuEnd* with the following code:

```
private void OnGUI() {
   GUILayout.Label("Congratulations " + Player.username);
   GUILayout.Label("You have finished the game, score = "
+ Player.score);

   bool mainMenuButtonClicked = GUILayout.Button("Main
Menu");
   if( mainMenuButtonClicked )
            Application.LoadLevel(0);
  }
```

6. Save your scripts and play the game. As you progress from level (scene) to level, you should find the score and player's name are 'remembered', until you quit the application.

How it works...

The *Player* class uses static (class) properties *username* and *score* to store the current player's name and score. Since these are public static properties, any object from any scene can access these values – static properties are 'remembered' from scene to scene. This class also provides the public static method *DeleteAll*() that resets *username* to null and *score* to -1;

Class *MenuStart* tests the value of *Player.username*, and if it is null then a textbox and button are provided to allow the user to enter a new username; which is then stored in *Player.username*. Once the value in *Player.username* is no longer null, the user is offered a button to start playing the game, or to erase the username and any score current stored in class Player.

The 3 game playing scenes now include a GUI method to display the current value in the *Player.score* property, which has 50 added to it each time a sphere tagged 'blue' is clicked.

Class *MenuEnd* congratulates the player by name (retrieving the name from *Player.username*), and tells them their score (from *Player.score*).

See also

- Saving and loading player data – using PlayerPrefs

Saving and loading player data – using PlayerPrefs

While the previous recipe illustrates how static properties allow a game to 'remember' values between difference scenes, those values are 'forgotten' once the game application has quit. Unity provides the *PlayerPrefs* feature to allow a game to store and retrieve data between different game playing sessions.

Getting ready

This recipe builds upon the previous recipe. In case you haven't completed the previous recipe, we have included a complete project named *BlueSpheres-static* in the folder *0423_07_05*. In order to follow this recipe, make a copy of that folder as your starting point.

How to do it...

To save and load player data, follow these steps:

1. Open the *BlueSpheres-static* project and delete the C# script *Player.*
2. Edit the C# script *MenuStart* by replacing method OnGUI() with the following code (the 'if' statement is to be changed):

```
private void OnGUI() {
   string rules = "Easiest Game Ever -- Click the blue
spheres to add points and advance levels.";
   GUILayout.Label(rules);

   if(PlayerPrefs.HasKey("username"))
      WelcomeGUI();
   else
      CreatePlayerGUI();
}
```

3. Edit C# script *MenuStart* by replacing method *ResetGameData*() with the following code:

```
private void ResetGameData() {
   PlayerPrefs.DeleteAll();
   playerNameField = Player.defaultName;
}
```

4. Edit C# script *MenuStart* by replacing the first line of method *WelcomeGUI*() with the following:

```
string welcomeMessage = "Welcome, " +
PlayerPrefs.GetString("username") + ". You currently have
" + PlayerPrefs.GetInt("score") + " points.";
```

5. Now edit C# script *SphereClick* by replacing methods *ONGUI*() and *OnMouseDown*() with the following code:

```
private void OnGUI(){
   GUILayout.Label( "Score: " +
PlayerPrefs.GetInt("score") );
}

private void OnMouseDown() {
   if( gameObject.CompareTag("blue") ){
      int newScore = 50 + PlayerPrefs.GetInt("score");
      PlayerPrefs.SetInt("score", newScore);
   }

   Destroy(gameObject);
   GotoNextLevel();
}
```

6. Edit C# script *MenuEnd* by replacing the first two lines of method *OnGUI*() with the following:

```
GUILayout.Label("Congratulations " +
PlayerPrefs.GetString("username"));

GUILayout.Label("You have finished the game, score = " +
PlayerPrefs.GetInt("score"));
```

7. Save your scripts and play the game. Quit from Unity and then restart the application. You should find that the player's name, level and score are now kept between game sessions.

How it works...

All we needed to do was to change our code to take advantage of Unity's *PlayerPrefs* Runtime Class. This Class is capable of storing and accessing information (*String*, *Int* and *Float* variables) in the user's machine. Values are stored in a 'plist' file (Mac) or the registry (Windows), in a similar way to web browser 'cookies', and therefore 'remembered' between game application sessions.

NOTE: For more information on *PlayerPrefs*, see Unity's online documentation:

docs.unity3d.com/Documentation/ScriptReference/PlayerPrefs.html

See also

- Saving and loading player data – using static properties

Saving screenshots from the game

In this recipe, we will learn how to take in-game snapshots and save it in an external file. Better yet, we will make it possible to choose between three different methods.

Getting ready

In order to follow this recipe, please import the package *screenshots*, available in the folder *0423_07_06*, into your project. The package includes a basic terrain and a camera that can be rotated via mouse.

How to do it...

To save screenshots from your game, follow these steps:

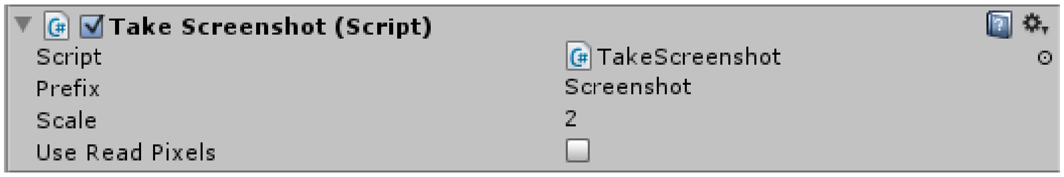
1. Import the *screenshots* package and open the *screenshotLevel* scene.

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

```
// file: TakeScreenshot.cs
using UnityEngine;
using System.Collections;
using System;
using System.IO;
public class TakeScreenshot : MonoBehaviour {
public string prefix = "Screenshot";
public int scale = 1;
public bool useReadPixels = false;
private Texture2D texture;

void  Update (){
 if (Input.GetKeyDown (KeyCode.P))
        StartCoroutine(TakeShot());
}

IEnumerator  TakeShot (){
   string date = System.DateTime.Now.ToString("_d-MMM-
yyyy-HH-mm-ss-f");
   int sw = Screen.width;
   int sh = Screen.height;
   Rect sRect = new Rect(0,0,sw,sh);
   if (useReadPixels){
      yield return new WaitForEndOfFrame();
      texture = new Texture2D
(sw,sh,TextureFormat.RGB24,false);
      texture.ReadPixels(sRect,0,0);
      texture.Apply();
      byte[] bytes= texture.EncodeToPNG();
      Destroy (texture);
      File.WriteAllBytes(Application.dataPath +
"/../"+prefix + date + ".png", bytes);

   } else {
      Application.CaptureScreenshot(prefix + date +
".png", scale);
   }
}
}
```

2. Save your script and attach it to the *Main Camera* Game Object by dragging it from the *Project* view to the *Main Camera* game object in the *Hierarchy* view.
3. Access the *Take Screenshot* component. Change *Scale* to 2 and leave *Use Read Pixels* unchecked.

NOTE: If you want your image files name to start with something different than *Screenshot*, change it in the field *Prefix*.


Take Screenshot (Script)

Script	TakeScreenshot
Prefix	Screenshot
Scale	2
Use Read Pixels	☐

0423_07_01.png

4. Play the scene. A new screenshot, with twice the original size, will be saved in your project folder every time you press "P".

How it works...

Once the script has detected that the P key was pressed, the screen is captured and stored as an image file into the same folder where the executable is. Unless *Use Read Pixels* is selected, the script will call a built-in Unity function called *CaptureScreenshot()*, capable of scaling up the original screen size (in our case, based on the *Scale* variable of our script)

There's more...

We have included the option *Use Read Pixel* as a demonstration of how to save your images to disk without using Unity's *CaptureScreenshot()* function. One advantage of this method is that it could be adapted to capture and save only a portion of the screen. The *Scale* variable from our script will not affect it, though.

Control characters in Unity with the Microsoft KINECT using the Zigfu samples

The Microsoft Kinect human motion controller offers an exciting way for players to interact with games and control their game characters. Some of the original developers of the software used in the Kinect started a company called Zigfu, and they now offer a Unity-Kinect development kit, which is a straightforward way to make Unity games that can use the Microsoft Kinect motion controller.

At the time of writing an unlimited time free trial is available to download. The Zigfu free trial is like Unity free – it can be used for non-commercial projects and includes a 'watermark' in projects.

> NOTE: Be patient!
>
> When you run a Kinect-device application you will have to wait 10-20 seconds for your computer to establish a link with the Kinect, and for the Kinect to detect a person standing in front of the camera.
>
> Learn more about the Microsoft Kinect at `www.xbox.com/KINECT`

Getting ready

Go to the Zigfu website downloads sections, and locate the Unity ZDK package (Zigfu Development Kit). At the time of writing the ZDK Unity package is named *ZDK_Unity35_.99f_trial.unitypackage*, and can be found at:

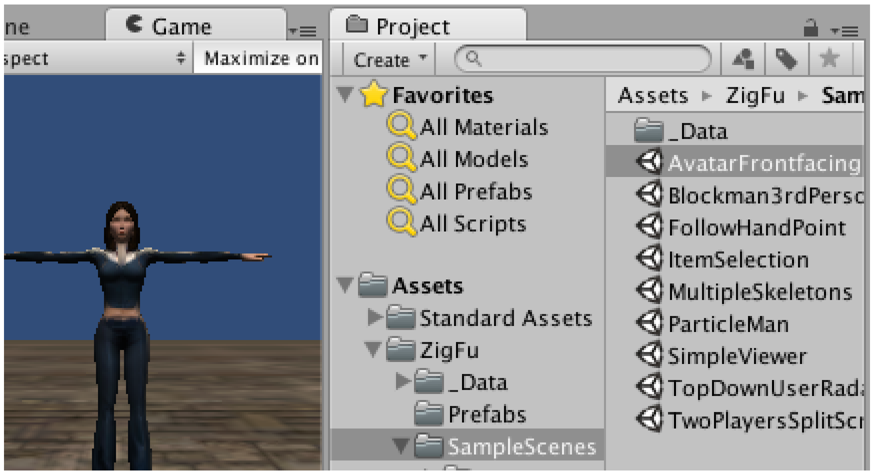
- `http://zigfu.com/en/zdk/unity3d/`

How to do it...

1. Create a new Unity project and Import the ZDK package.
2. You should now see two folders appear in your *Project* panel: *Standard Assets* and *Zigfu*

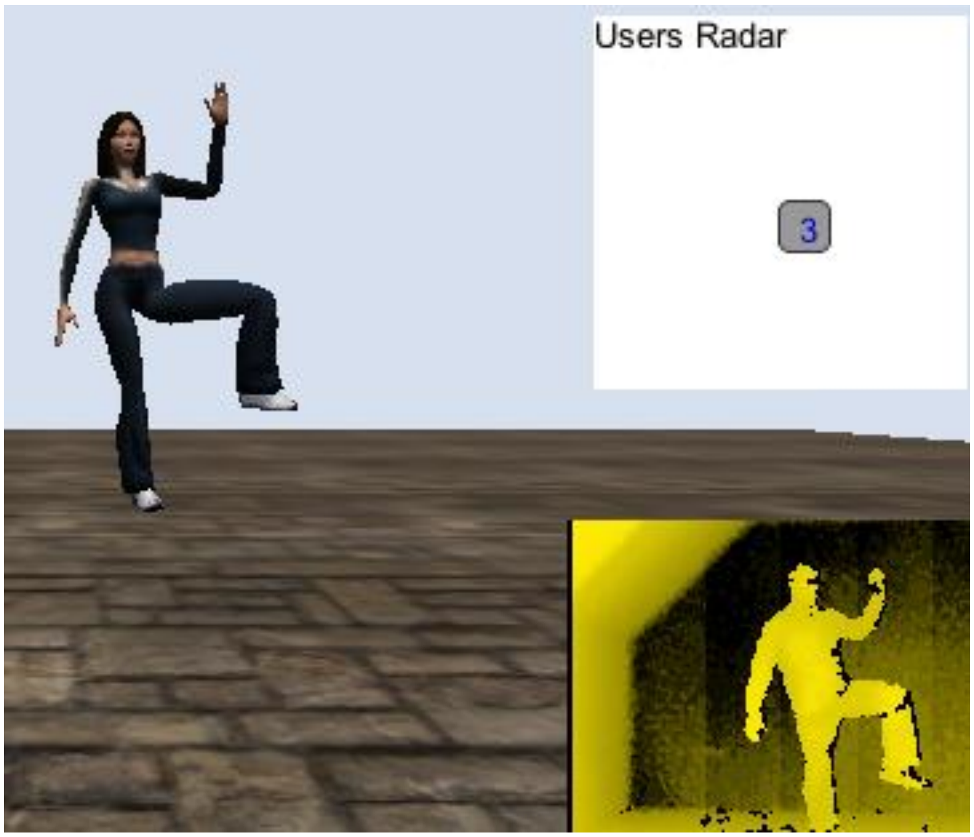
NOTE: You can ignore any warning messages about inconsistent line endings

3. Open scene *AvatarFrontfacing* – You'll find it in the *SampleScenes* folder inside the *Zigfu* folder. You should now see an avatar in the *Scene* and *Game* panels



0423_07_02.png

4. Plug in your Kinect camera, and ensure the Kinect power supply is connected and switched on
5. Run the scene …
6. After taking a few seconds to open communication between the sensor and Unity you should then be able to control the avatar through your own body movements.



0423_07_03.png

7. You'll also see the 'radar' display (top right, white square), indicating the position of each user relative to the camera, and the raw point cloud display (bottom-right, yellow-black square)

How it works...

The sample scene contains a character, and also all the scripted Zigfu objects connected to this character, so that when the Zigfu objects receive movement event messages from the Kinect camera, these movements are passed on to move and rotate the appropriate parts of the visible character.

See also

- Animating your own characters with the Microsoft KINECT controller

- Homemade mocap by storing movements from the Microsoft KINECT controller

Animating your own characters with the Microsoft KINECT controller

Perhaps the best way to understand how the Zigfu objects and event messages can be used in your own Unity projects is to build a basic figure out of cubes, and associate particular cubes with particular skeleton objects. In this recipe you'll build part of a block character from cubes (a head, left shoulder, elbow and hand), and use Zigfu scripts to be able to control this arm when the game runs with the Kinect controller attached.

Getting ready

NOTE: The scripts you'll be using in this recipe can be found in the following folders of the Project panel once you have Import the ZDK Unity package:

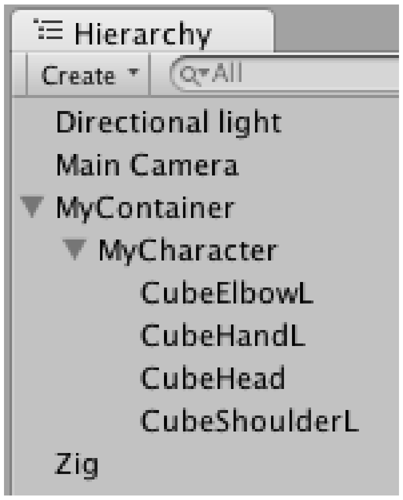
- Zigfu/Scripts/UserEngagers/
- Zigfu/Scripts/UserControls/
- Zigfu/SampleScenes/Scripts/
- Standard Assets/Scripts/CameraScripts/

How to do it...

1. Create a new Unity project and Import the ZDK package.
2. Add directional light
3. Create an empty GameObject name it *MyContainer*
4. Create an empty GameObject named *MyCharacter*, and in the Hierarchy drag this object to become a child of *MyContainer*
5. Create the following cubes, and in the *Hierarchy* drag these objects to become a children of *MyCharacter*:
   - a cube named *CubeHead* at (0,0,0) sized (0.2, 0.2, 0.2)
   - a cube named *CubeShoulderL* at (0.2, -0.2, 0) sized (0.2, 0.2, 0.2)
   - a cube named *CubeElbowL* at (0.5, -0.2, 0) sized (0.2, 0.2, 0.2)
   - a cube named *CubeHandL* at (0.8, -0.2, 0) sized (0.2, 0.2, 0.2)

NOTE: The positions of these cubes don't matter, since their positions in the game will be controlled by the head/arm positions detected by the Kinect sensor. However, it can be handy to lay them out in an approximation of a block-person to avoid getting confused when adding new cubes.

6. Create an empty GameObject name it *Zig*



0423_07_04.png

7. Add to Main Camera the scripted component *SmoothFollow*, set its Distance variable to 4, and with *Main Camera* selected, in the Inspector drag *CubeHead* into the public variable slot for Target for the *SmoothFollow* scripted component
8. Add to *Zig* the scripted component *Zig*
9. Add to *Zig* the scripted component *ZigEngageSingleUser*
10. Ensuring *Zig* is selected, in the *Inspector* for the *ZigEngageSingleUser* component drag *MyCubeCharacter* over public variable *Engaged Users* (its size should increase from 0 to 1, and *MyCharacter* should be listed as Element 0)
11. Add to *MyCharacter* the scripted component *ZigSkeleton*, and in the *Inspector* ensure the following options are ticked (checked):
    - Update Joint Positions
    - Update Root Positions
    - Update Orientation
12. With *MyCharacter* selected, drag the following in the *Inspector*

- CubeHead into the public variable slot for Head
- CubeShoulderL into the public variable slot for Left Shoulder
- CubeElbowL into the public variable slot for Left Elbow
- CubeHandL into the public variable slot for Left Hand

13. Run the scene – you should now be controlling the cubes on screen as you move your head and left arm!

How it works...

The *ZigEngageSingleUser* component applies observed (KINECT input) positions and rotations for skeleton objects to associated GameObjects in the scene. In step 12 above, you linked the Transforms of the cubes you created with particular skeleton elements that the *ZigEngageSingleUser* component is tracking form the KINECT. Each frame this component applies the position and transform changes to your cubes, and we see our block character move as we move ourselves in front of the camera.

There's more...

Some details you don't want to miss:

Completing your block character

To complete your block character you might use a capsule GameObject for the Torso, and cubes for the other parts of the arms and legs. You might wish to refer to the Zigfu sample scene *Blockman3rdPerson* since it is essentially the same as this recipe when it has been completed.

See also

- Control characters in Unity with the Microsoft KINECT using the Zigfu samples
- Homemade mocap by storing movements from the Microsoft KINECT controller

Homemade mocap by storing movements from the Microsoft KINECT controller

The positions and rotations of each skeleton component are available to access at any frame, therefore, if we create a data structure to store them, then a recording (and playback) of motions controlled via the KINECT can be implemented. This is demonstrated in the following recipe.

Getting ready

This recipe assumes you already have a project that uses Zigfu to control a character. You could either start with one of the scenes from the ZDK, or you could also build on the project from the previous recipe, where you created your own block character (that's what we did).

How to do it...

1. Open your existing *Zigfu* project
2. Create a sphere named *GhostHandL*, ensuring it is similar in size to the hand of the character in your scene
   (so this would be (0.2,0.2,0.2) if you are building on the previous recipe)
3. Add a red material to *GhostHandL*

Note: It doesn't matter what position GhostHandL starts at, since it is going to follow the path of the Left hand of your model once you start play back

4. Create a new C# script *ObjectAtFrame* with the following code:

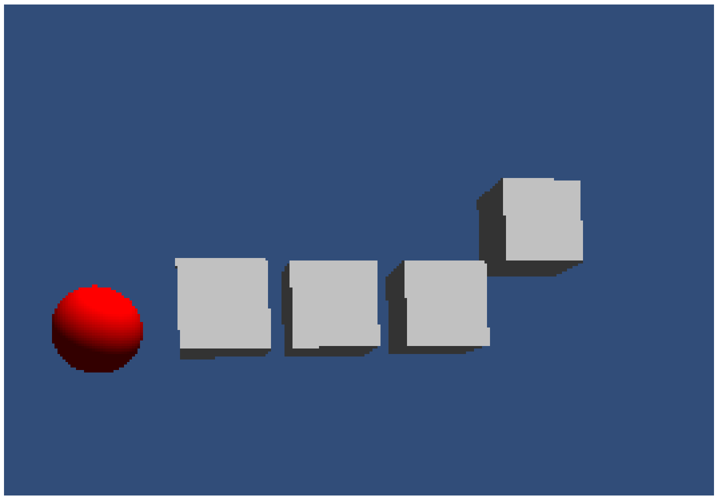
```
// file: ObjectAtFrame.cs
using UnityEngine;
using System.Collections;

public class ObjectAtFrame {
   public Vector3 position;
   public Quaternion rotation;

   public ObjectAtFrame(Vector3 newPosition, Quaternion
newRotation) {
      position = newPosition;
      rotation = newRotation;
   }
}
```

5. Add the following script class to the *Main Camera*:

```
// file: RecordMovements.cs
using UnityEngine;
using System.Collections;
using System.Collections.Generic;
```

```
public class RecordMovements : MonoBehaviour {
   public Transform leftHand;
   public Transform redSphere;
   private bool isRecording = false;
   private List<ObjectAtFrame> movementList = new
List<ObjectAtFrame>();
   private int currentFrame = 0;

   private void OnGUI() {

      if( !isRecording ) {
         bool startRecordingButtonClicked =
GUILayout.Button ("START recording");
         if( startRecordingButtonClicked ) {
            isRecording = true;
            movementList.Clear();
         }
      } else {
         GUILayout.Label ( "RECORDING ------------------
-" );
         GUILayout.Label("number of frames record = " +
movementList.Count);
         bool startRecordingButtonClicked =
GUILayout.Button ("STOP recording");
         if( startRecordingButtonClicked ){
            isRecording = false;
         }
      }
   }

   private void Update () {
      if( isRecording )
         StoreFrame();
      else
         if( movementList.Count > 0)
            PlayBackFrame();
   }

   private void StoreFrame() {
      ObjectAtFrame leftHandAtFrame = new
ObjectAtFrame(leftHand.position, leftHand.rotation);
      movementList.Add (leftHandAtFrame);
   }

   private void PlayBackFrame(){
      currentFrame++;
      if( currentFrame > (movementList.Count -1))
         currentFrame = 0;
      ObjectAtFrame objectNow =
movementList[currentFrame] as ObjectAtFrame;
      redSphere.position = objectNow.position;
      redSphere.rotation = objectNow.rotation;
   }
}
```

6. With the *Main Camera* selected in the *Hierarchy*, associate the following objects in the *Hierarchy* to their corresponding public variables in the *Inspector*:
   - Drag *redSphere* into variable *Red Sphere*
   - Drag the left hand of your character into variable *Left Hand*



0423_07_05.png

7. Run the project. Once the KINECT is controlling your character, click the "START recording" button, move your left hand in a big circle, and then click the "STOP recording" button.
8. You should then see the red sphere following your recorded left hand movement of a big circle – it will cycle through this recorded movement repeatedly.

How it works...

Important data structures for this recipe:

- Class *ObjectAtFrame* provides a data structure in which to store the position and rotation of one part of the character's body for a given frame.

- The class *RecordMovements* attached to the *Main Camera*, uses a generic List data structure named *movementList*, to create a dynamic ordered collection of *ObjectAtFrame* objects.

When the "START recording" button is clicked, the *isRecording* Boolean flag is set to true, and *movementList* is emptied with the Clear() method. Method Update() is executed each frame, and if *isRecording* is true, then method StoreFrame() is called.

Method StoreFrame() retrieves the position (Vector3) and rotation (Quaternion) of the Transform component of the left hand of the character (this was associated before running the project, by dragging that left hand object of the character to the public variable *Left Hand (leftHand)*). A new *ObjectAtFrame* object *leftHandAtFrame* is created with the position and rotation values, and added to the end of *movementList*.

When the "STOP recording" button is clicked, the *isRecording* Boolean flag is set to false. Method Update() is executed each frame, and if *isRecording* is false, and we have recorded 1 or more frames (i.e. the Count() of *movementList* is greater than zero), then method PlayBackFrame() is called.

Method PlayBackFrame() increments the *currentFame* integer, and reset this back to zero if its value is greater than the index of the last element in *movementList*. The position and rotation of the element for *currentFame* in *movementList* is retrieved, and applied to the Transform component of our *redSphere* object. Thus, the red sphere is made to follow the movement and rotation we recorded for that frame.

There's more...

Some details you don’t want to miss:

Recording movements of multiple body parts

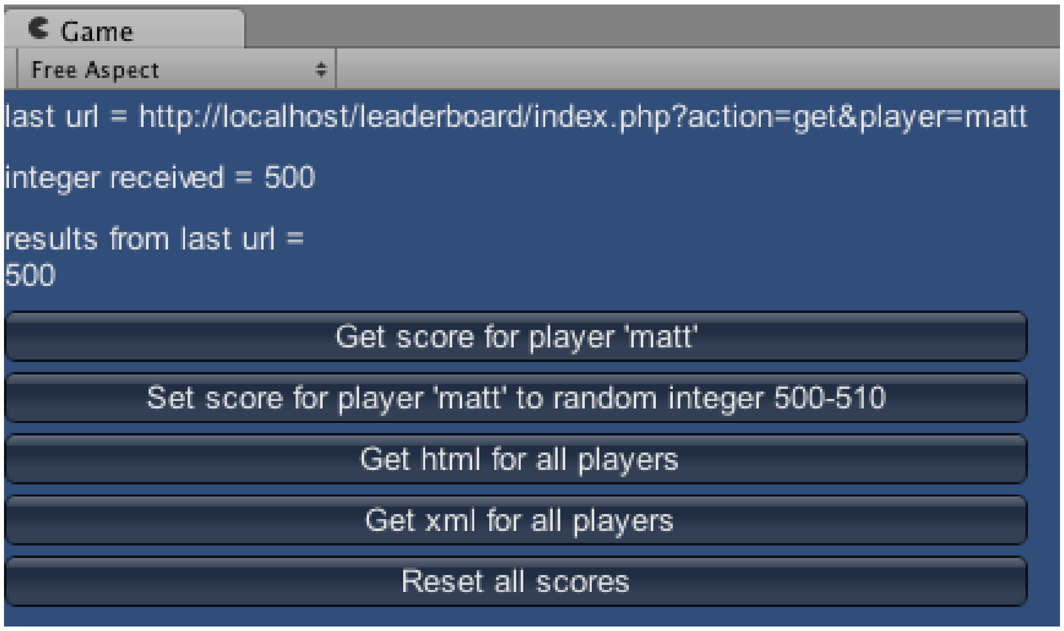
This recipe has illustrated the ‘proof of concept’ of how to record motion movements of a single skeleton object in Unity with the KINECT device. However, it is more likely that we would wish to record the position and rotation of EVERY object in the skeleton. This would require a slightly more sophisticated data structure. If performance was not an issue, perhaps a *Dictionary* data structure could be used, with identifiers for each part of the skeleton. If efficiency were important, then perhaps parallel arrays, two for each skeleton object, could be implemented (one for position Vector3s, and one for rotation Quaternions). Using a fixed number of array elements (i.e. a maximum number of frames) would be the trade off of speed against flexibility for this approach.

See also

- Control characters in Unity with the Microsoft KINECT using the Zigfu samples
- Animating your own characters with the Microsoft KINECT controller

Setting up a leaderboard using PHP/MySQL

Games are more fun when there is a leaderboard of high scores that players have achieved. Even single player games can communicate to a shared web-based leaderboard. This recipe includes both the client side (Unity) code, as well as the web-server side PHP scripts to set and get player scores from a MySQL database.



```
last url = http://localhost/leaderboard/index.php?action=get&player=matt

integer received = 500

results from last url =
500
```

Game
Free Aspect
Get score for player 'matt'
Set score for player 'matt' to random integer 500-510
Get html for all players
Get xml for all players
Reset all scores

0423_07_06.png

Getting ready

This recipe assumes you either have your own web hosting, or are running a local webserver and database server such as XAMPP or MAMP. Your web server needs to support PHP, and you also need to be able to create MySQL databases.

All the SQL, PHP and C# scripts for this recipe can be found in folder 0423_07_10.

NOTE: If you are hosting your leaderboard on a public website, you should change the names of the database, database user and password for reasons of security. You should also implement some for of secret game code, as described in the "There's more..." section.

How to do it...

1. (on your server) Create a new MySQL database named *cookbook_highscores*
2. (on your server) Create a new database user (username=cookbook. password=cookbook), with full rights to the database you just created
3. (on your server) Execute the following SQL to create database table *score_list*

```
CREATE TABLE `score_list` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `player` varchar(25) NOT NULL,
  `score` int(11) NOT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB  DEFAULT CHARSET=latin1 AUTO_INCREMENT=1 ;
```

4. (on your server) Copy the provided PHP script files to your web server:
   - index.php
   - scoreFunctions.php
   - htmlDefault.php
5. Add the following script class to the Main Camera:

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

public class WebLeaderBoard : MonoBehaviour {
   private string url;
   private string action;
   private string parameters;
   private string textFileContents = "(still loading file
...)";

   private void OnGUI() {
      // hide closing tag
      string prettyText = textFileContents.Replace("</",
"?@?");
```

```
      // prefix opening tag with newline
      prettyText = prettyText.Replace("<", "\n<");

      // return closing tag
      prettyText = prettyText.Replace("?@?", "</");

      GUILayout.Label ( "last url = " + url );
      GUILayout.Label ( StringToInt(textFileContents) );
      GUILayout.Label ( "results from last url = " +
prettyText );

      WebButtons();
   }

   private void WebButtons() {
      bool getButtonWasClicked = GUILayout.Button("Get
score for player 'matt'");
      bool setButtonWasClicked = GUILayout.Button("Set
score for player 'matt' to random integer 500-510");
      bool htmlButtonWasClicked = GUILayout.Button("Get
html for all players");
      bool xmlButtonWasClicked = GUILayout.Button("Get
xml for all players");
      bool resetButtonWasClicked =
GUILayout.Button("Reset all scores");

      if( getButtonWasClicked )
         GetAction();
      if( setButtonWasClicked )
         SetAction();
      if( htmlButtonWasClicked )
         HTMLAction();
      if( xmlButtonWasClicked )
         XMLAction();
      if( resetButtonWasClicked )
         ResetAction();
   }

   private string StringToInt(string s) {
      string intMessage = "integer received = ";
      try{
         int integerReturned = Int32.Parse(s);
         intMessage += integerReturned;
      }
      catch(System.Exception e){
         intMessage += "(not an integer) ";
         print (e);
      }
      return intMessage;
   }

   private void GetAction() {
      action = "get";
      parameters = "&player=matt";
      StartCoroutine( LoadWWW() );
   }

   private void SetAction() {
      int randomScore = UnityEngine.Random.Range(500,
510);
      parameters = "&player=matt&score=" + randomScore;
      action = "set";
      StartCoroutine( LoadWWW() );
   }

   private void HTMLAction() {
      action = "html";
      parameters = "";
      StartCoroutine( LoadWWW() );
   }

   private void XMLAction() {
      action = "xml";
      parameters = "";
      StartCoroutine( LoadWWW() );
   }

   private void ResetAction() {
      action = "reset";
      parameters = "";
      StartCoroutine( LoadWWW() );
   }

   private IEnumerator LoadWWW(){
      string baseUrl =
"http://localhost/leaderboard/index.php?action=";
      url = baseUrl + action + parameters;
//    yield return 0;
      WWW textFile = new WWW (url);
      yield return textFile;
       textFileContents = textFile.text;
   }
}
```

How it works...

The player’s scores are stored in a MySQL database. Access to the basis is facilitated through the PHP scripts provided. In our example all the PHP scripts were placed into a folder named *leaderboard* in the web server root folder. So the scripts are accessed via the URL “http://localhost/leaderboard/”. All access is through the PHP file *index.php*. There are 5 actions implemented, and each is indicated by adding the action name at the end of the URL (this is the GET HTTP method, sometimes used for web forms – take a look at the address bar of your browser next time you search Google for example …). The actions, and their parameters (if any) are as follows:

- action = get, parameters: player = matt
    - this action asks for the integer score of the named player to be found
    - returns: score integer
- action = set, parameters: player = matt, score = 101
    - this action asks for the provide score of the named player to be stored in the database (but only if this new score is greater than the currently stored score)
    - returns: score integer (if database update was successful), otherwise a negative value (to indicate no update took place)
- action = html, no parameters
    - this action asks for HTML text listing all player scores to be returned
    - returns: HTML text
- action = xml, no parameters
    - this action asks for XML text listing all player scores to be returned
    - returns: XML text
- action = reset, no parameters
    - this action asks for a set of default player name and score values to replace the current contents of the database table
    - returns: the string “reset”

The OnGUI() method first displays the current URL string, any integer value that was extracted from the response message received from the server, and the full text string received from the server. Also 5 buttons are offered to the user, which set up the corresponding action and parameters to be added to the URL for the next call to the web server via method LoadWWW().

There's more...

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

Extract the full leaderboard data as XML for display within Unity

The XML text that can be retrieved from the PHP web server provides a useful method of allowing a Unity game to retrieve the full set of leaderboard data from the database, and then the leaderboard can be displayed to the user in the Unity game (perhaps in some nice 3D fashion, or through a game-consistent GUI). See Chapter 8 for several recipes that illustrate ways to work with XML data in Unity.

Using secret game codes to secure your leaderboard scripts

The Unity and PHP code presented above illustrates a simple, unsecured web-based leaderboard. To prevent players 'hacking' into the board with false scores, it is usual to encode some form of secret game code (or 'key') into the communications. Only update requests that include the correct code will actually cause a change to the database.

The Unity code would combine the secret key (in this example the string "harrypotter"), with something related to the communication – for example the same MySQL/PHP leader board may have different database records for different games, identified with a game Id.

```
// Unity Csharp code
string key = "harrypotter"
string gameId = 21;
string gameCode = Utility.Md5Sum(key + gameId);
```

The server-side PHP code would receive both the encrypted game code, and also the piece of game data used to create that encrypted code (in this example, the game Id, and the MD5 hashing function, which is available both in Unity and in PHP). The secret key ("harrypotter") is used with the game Id to create an encrypted code that can be compared with the code received from the Unity game (or whatever user agent or browser is attempting to communicate with the leaderboard server scripts). Database actions will only be executed if the game code created on the server matches that send along with the request for a database action.

```
// PHP – security code
$key = "harrypotter"
$game_id =  $_GET['game_id'];
$provided_game_code =  $_GET['game_code'];
$server_game_code = md5($key.$game_id);

if( $server_game_code == $provided_game_code ) {
    // codes match - do processing here
}
```

See also

- How to download and work with text files and XML files in Chapter 8
- Preventing your game from running on unknown servers in Chapter 10