Colton Young- Software Engineering & Computer Science

Sunday, April 10, 2016

Persistent Data in Unity

Introduction

In Unity, and indeed, many other game engines, data persistence is an important function to employ to hold values for attributes that last not only between scenes, but also between executions of a game. For example, a health bar's value would be important to keep as you load into a new scene (level) and player preferences (such as the game's music's volume) would want to be maintained across instances of the game. To save this kind of data in Unity, there are a multitude of different ways to accomplish these tasks, but some of these methods are more suitable in certain situations than others. I have found and compiled a short introduction to three different ways to store important values so they persist between scenes and/or executions.

Background

For me, there were three different types of data values that I wanted to store for different purposes:

Data values that were in one scene that I wanted to carry over into another scene, like my background music. (These data values were stored in scripts derived from the Monobehaviour base class that most user-created, offline scripts are derived from- it gives access to standard function like Start(), Update(), StartCoroutine() etc. that form the basis for how a Unity Game works.)

Data values that can be changed by the user and should persist between scenes and executions, such as a game volume variable that would store a number from 0 to 1 indicating how loud the player wanted the game.

Data values that persist between scenes and executions **but** should not be able to be changed by a user (in my case, it was storing variables that tracked achievement progress.)

Each of these types of data values can be stored in many different ways, but there are certain choices that are better for the different kinds of data types.

Method 1. Carrying values over through scripts that invoke the method DontDestroyOnLoad() and a pseudo-Singleton

Sometimes you may want to store values that persist across scenes in a game, but are not retained over multiple executions- this is best for things you want to begin fresh each time you run the game, such as your game's background music or a variable that is recording the time played for the current game session only. One way to do this is to create a script that exclusively holds these kind of data values and create a static instance of this script (making sure to use a Singleton pattern so that a new instance of the static instance is not instantiated on scene changes) such as like this:

```
public static BackGroundMenuAndMusic BGMusic;

    void OnAwake ()
    {
        //Setup a Singleton Like Object
        if (BGMusic == null) {
            DontDestroyOnLoad (gameObject);
            BGMusic = this;
        }
        //and make sure it persists
        //if another is found.. destroy so only one exists.
        else if (BGMusic != this) {
```

Blog Archive

▼ 2016 (1)
▼ April (1)
Persistent Data in Unity

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

The reason I put this in the onAwake() function instead of a Start() function is so that this is the very first thing that happens when we boot up the game. Otherwise, if we put it in a start function, the instantiation of our variables will happen during the same time frame that any of our other scripts' start functions are happening. In this case, nothing in my first scene relies on the instantiation of this variable, so there would be no drawback to using the Start() function. However, if one of our other scripts in this scene relies on this variable, we can't guarantee that our variables will be instantiated before that script needs them because their start functions are running simultaneously, and we don't know which will complete first on any given execution.

You will also want to invoke the DontDestroyOnLoad() method in our script so that the gameObject that this script is attached to is carried over from scene to scene. With this setup, not only is there only one instance of your background music (or whatever variable you're storing) that carries over from scene to scene, it is also accessible from any other script in any other scene. So while we've now allowed the music to persist across scenes, we've also allowed ourselves a way to modify the music variable with the available methods and this modification persists across scenes. So if we want to use a slider in some scene to change the volume of the music, it will affect the volume across all scenes. All that's left to do is create an arbitrary gameObject to attach this script to and put this gameObject in all of our scenes so that our script instantiates our variables regardless of what scene the game is started from (this part is mostly useful as a developer, since the player will, in most cases, always begin the game on a certain scene.)

The second chunk of code that is necessary to use this object from another script in a separate scene is to bring it into the scene with variable assignments. In other words, we bring the object from another script into the scene by making it a new variable and giving it a new name. You can see in the following code how I did this to bring the object that my audio source is attached to into the scene where my slider that adjusts the volume is:

```
//instantiating an instance of a gameObject (which will be where I put the
//audio object from my other scene) and a slider object
private GameObject theAudioObject;
private Slider theSlider;

void Start () {
    //assign the value of theAudioObject that I've just created a reference
    //to the audio object in my other scene- its name is AudioHoldingObject
    theAudioObject = GameObject.Find ("AudioHoldingObject");
    //make this instance of a Slider be a reference to the slider in this scene
    theSlider = Slider.FindObjectOfType<Slider> ();



    theSlider.value = theAudioObject.GetComponent<AudioSource>().volume;
```

We can now manipulate the variables of the gameObject's components (in this case, its audio source component) as if it was in this scene. Therefore, we now have access to this object within this scene and as long as we have the DontDestroyOnLoad command, the original object created in the first scene is the gameObject that we are modifying—persistent across scenes.

Method 2. Using the Unity PlayerPrefs class to store variables across executions

The Unity engine has a great tool for storing values across execution of your game. This is the PlayerPrefs class and it works like this: On the first execution of your game on a device, the executable will create a file within the game folder that stores a text file that, depending on the device, can be a .txt, .dat, .xml, .plist, etc. This file is the file that holds the PlayerPrefs data and its specific file path can be found in the Unity Manual here: http://docs.unity3d.com/ScriptReference/PlayerPrefs.html. One of the dangers here, however, is that these files are easily readable and editable. With a simple text editor, a person could come in and easily read and change the values stored in this file. This is why this solution is good for storing settings that are individually chosen preferences, hence the name PlayerPrefs. In my case, the only value I'm storing is my volume information, so this method of storing data makes sense because I'm not worried about the user going into the file and changing the value. It doesn't disrupt the gameplay experience to have this value changed externally (compare this to the next method I'll talk about, saving with serialization.) The PlayerPrefs file works something like a dynamically-allocated, String-indexed array. The Set functions require two parameters, the key which is a string, and the value you wish to store. To retrieve your value, all you have to do is make a call to the appropriate get function and supply the index name.

Saving an attribute to the PlayerPrefs file looks like this,

```
    //PlayerPrefs.SetInt() and PlayerPrefs.SetString() are the other set
methods
    PlayerPrefs.SetFloat ("Game Volume", VolToChangeTo);
    PlayerPrefs.Save ();
```

Make sure to invoke the appropriate method for your variable type. Once you have made a change, call the save function and the values are now stored in the file for future use. All that's necessary to bring a variable out of this file is to call the corresponding Get methods which you can do from any scene, as PlayerPrefs is a part of the UnityEngine library, one of the two default libraries when writing scripts. This solution is clearly simple, but vulnerable because of its ability to externally be edited, which is why there's a better third option for storing valuable data.

Method 3. Saving attributes to a file with binary serialization

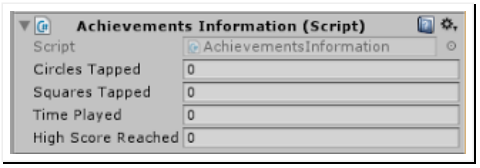
Of the three methods for saving data presented here, the serialization to a file is the most secure because files formatted this way are very difficult to look at and modify. This is because they are arranged in a very specific way that would allow a program to read and use them, but to the human eye, are unintelligible. This method is best used for storing values that the user should never have control over, so, for example, storing achievement progress. I would not want the user to be able to manually change the values of their achievement progress to "game the system," so I wouldn't want to store these values in the PlayerPrefs file. Instead, I convert them into an unreadable, un-editable (by the user) file. This method is organized, essentially, into two storage locations for your important data. The first is a local set of variables attached to a game object that's in all of your scenes and is present in your execution of the game. The other storage location for your set of the same variables is in an external file (in my case, a .dat file.) When you load your data in, you read the dat file to find the values you want, and then put them into your local set of variables to use. When you want to take your local set and save them for future use, you convert them into a serializable object which is then written to the file. To start using this method, you must first set up a script that has a serializable data structure, a save method, and a load method. Additionally, it is typically most useful to make a static instance of this script so that you only ever have one local copy of the variable set. As you can see here in my example, I have made a static instance of my script (called staticAchieveInfo) and specified four values that I want to track.

```
public static AchievementsInformation staticAchieveInfo;

    public int circlesTapped;
    public int squaresTapped;
    public float timePlayed;
    public float highScoreReached;
```

I also created a Singleton in this script so that if this script is in multiple scenes, I don't have multiple occurrences of my instance.

Notice the public keyword on my static instance. This allows me to call the Load and Save functions from any other script, saving me the hassle of having to bring this whole script into every other script where I need the load and save functions. This has set up my local set of variables which you can see here in the Unity inspector:



You must then create your save and load functions so that you can write these variables to the file and bring them back. To do this, import these two libraries:

```
using System.Runtime.Serialization.Formatters.Binary;
using System.IO;
```

We need these because we will be performing some input/output functions as well as using binary serialization.

Next, because we don't want to just serialize and send four arbitrarily named variables and their values to a dat file directly, we must create a serializable container- a class that we can give our local variables to, which will then be transformed by the binary formatter into its final form to be written to the dat file. To do this, we must first use the Serializable keyword on our class and then give it an arbitrary name with arbitrary variable names (although its best to give your variables names that correspond to the ones you used for your local set, so that you can easily recognized which variables in the class to give your local variables to.) See as follows:

```
[Serializable]
class AchievementData
{
    public int circlesTapped;
    public int squaresTapped;
    public float timePlayed;
    public float highScoreReached;
}
```

The Serializable keyword tells the compiler that this object is able to be serialized and you must have this keyword in brackets, capitalized, and right before the “class” keyword.
Notice I used the variable names as my local set so that I know which variable goes with each of my local set. What follows is the code for the save and load functions with comments above each line, explaining their meaning:

```
public void Save()
{
    //We create a BinaryFormatter instance with this line
    BinaryFormatter bf = new BinaryFormatter ();
    //We then create our FileStream instnace here and use the create method which we're giving the filepath
    //the variable Application.persistentDataPath is the data path specified by Unity for the type of machine running the game
    //it will be different for different devices- this line allows you to have it generate the filepath on its own
    //"/AchievementInfo.dat" is what I named my final file
    FileStream file = File.Create (Application.persistentDataPath + "/AchievementInfo.dat");
    //We now create an instance of our serializable class and give its variables the values of our local variables
    AchievementData data = new AchievementData ();
    data.circlesTapped = circlesTapped;
    data.squaresTapped = squaresTapped;
    data.highScoreReached = highScoreReached;
    data.timePlayed = timePlayed;

    //this is the binary formatter's serialize method which requires us to specify what file to write to, and what data we're writing
    //in this case, I give it the names of my variable that holds the filepath, and I give it my serializable instance
    //this method doesn't exactly "encode" the information, but you can think of it like that- it's converting it to a form that
    //is understandable by a computer, but not by a user
    bf.Serialize (file, data);
    //at this point, all of the data that we wanted to save has been formatted and written to the file, all that's left is to close the file
    file.Close ();
}
```

```
public void Load()
{
    //When we load, we first make sure that the file we will read from exits
    if (File.Exists (Application.persistentDataPath + "/AchievementInfo.dat")) {
        //We then create a binary formatter like we did in the save function
        BinaryFormatter bf = new BinaryFormatter ();
        //we now call the open method for our filestream, instead of the create method and give it the file location and the parameter Open
        FileStream file = File.Open (Application.persistentDataPath + "/AchievementInfo.dat", FileMode.Open);
        //We then use our binary formatter to undo the "encoding", giving us our serializable instance back and we give that instance the
        //name data
        AchievementData data = (AchievementData)bf.Deserialize (file);
        //and finally, we close the file because we've succesfully pulled the data object out
        file.Close ();

        //Here is where we assign the values of the serialized object to our local copy, thus completing the load for these variables
        circlesTapped = data.circlesTapped;
        squaresTapped = data.squaresTapped;
        timePlayed = data.timePlayed;
        highScoreReached = data.highScoreReached;
    }
}
```

Our local data can now be transformed into a serializable object, formatted into a .dat file, and then pulled out and reassigned to our local data at a later time, all while being relatively unusable to the user. To use these functions, all we have to do is use the script name, the static instance name, and the method name as such, because we made a public static instance of the script at the beginning:

```
AchievementsInformation.staticAchieveInfo.Save ();
AchievementsInformation.staticAchieveInfo.Load ();
```

Conclusion

As you can see, there are many ways to accommodate data persistence, each with their own use. Whereas keeping a music file across scenes could use the serialization method, it would be a bit overkill for its uses. Similarly, saving the achievement information into the PlayerPrefs file would be dangerous, as someone could update the file to show them with a high score of 1,000,000,000,000,000 or some such unearned value that would be cheating the system. Evaluate the type of information you need to persist and how long you want it to persist for, and you can decide which of these methods to use. As a rule of thumb, use the 1st method for data that should reset at the beginning of each execution but persist through the execution, use the 2nd method for data that wouldn’t be disruptive if the user had access to it, and use the 3rd method for data that should not be editable by a user, but is important enough to save across executions.

This post written with the Unity Game Engine in mind by Colton Young, 4/10/16

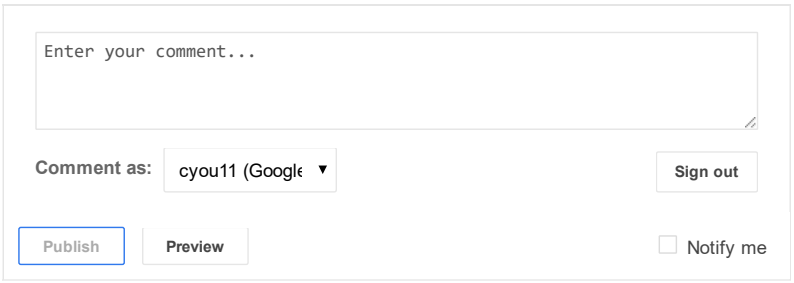
Posted by cyou11 at 5:41 PM

G+1 Recommend this on Google

Labels: computer, design, engine, engineering, game, science, software, technology, unity

No comments:

Post a Comment



Home

Subscribe to: Post Comments (Atom)