Effective Data Storage in the Unity3D engine

by Ruben Bergshoeff

Introduction

In the production of games, programmers are often focussed on the development of complex structures, patterns and interconnected systems. AI behaviour needs to be created, the economy of the game should depend on the spending behaviour of our player, the UI has to represent the state of the game and the player. Behind all these systems, one often overlooked element is king: data storage.
Ultimately, a game is just a collection of data, connected to a representation of that data that a player is experiencing. To that end, the various systems are communicating said data with each other, they are exchanging pointers or copying data from one another. But as games have become the work of teams rather than an individual programming the whole system, exchanging the right data has become a challenge, as well as testing individual systems without needing work of others. I decided to invest my research into this aspect that I tend to leave for last in projects and therefore is often left slightly untended, specifically in the Unity3D Engine.
The selection of the ideal method will be dependant on my specific needs: I need to record the state of the game. This comprises data on each player, information on rounds that have been played, and a few collections of questions and facts used in the game.
To conclude our introduction, the questions we want to see answered within this paper is the following:
What methods for data handling exist within the context of the Unity3D engine?
Which method would be most advantageous for our game?

Method

To research the various ways data storage is possible I am going to turn to desk research. By acquiring information from the Unity3D website and documentation, from the GDC vault on game development, and the Unite talks, I hope to find enough information to make a conclusive research paper.

Results and Analysis

Before we get into the various ways to handle data, I'll give a short introduction about the use of the Unity3D engine. It will be far from comprehensive, but some knowledge is relevant for understanding what follows.
Like most engines, the Unity3D engine has a project folder where all assets are stored. These assets can be mostly any filetype, often in game development we're talking 3D models, sprites, textures, fonts, sound files and scripts. 1

Ultimately, what the player sees when playing the game, is a scene. A scene is a collection of objects that form the gameplay. Unity has it's own unique system were anything that is put into the scene has to be a GameObject at base, one of Unity3D's own base classes. A GameObject can contain multiple other objects if they fit the requirements. Games more often than not exist of various scenes that switch whenever relevant. These scenes could represent gameplay levels, menu's or any other game element that could be displayed to the player disconnected from other elements. 2
Now that we've defined where we're coming from, let's dive into the data storage options within the Unity3D engine.

MonoBehaviour
MonoBehaviour is the base class from which every Unity script initially derives. This is the starting point when learning to work in the Unity3D engine. The MonoBehaviour base class provides a few methods that are automatically called at certain points in the game loop, for example at the start of a scene, every update frame, and on destruction. Scripts derived from this class are allowed to be put onto GameObject objects, which in turn allows them to exist within a scene.
Just like any other C# script, scripts derived from the MonoBehaviour class can contain properties and thus data, but what is unique for the MonoBehaviour derived classes is that their property values are displayed to be edited live whenever they are part of a GameObject. This allows for anyone working on the project to for example change a speed variable of any entity without changing the script itself, and to be able to see the values changing whenever underlying systems change the values. 3
The use of MonoBehaviour derived scripts as the base for handling data within a game has a lot of advantages. As said before, it allows for easy changing of values and displays it's values very clearly, which is great for ease of use. The values are part of the object the script is assigned to, so they are easily found. Another big advantage is that values can be changed even when the game is running within the editor, which allows for easy tweaking of settings where you can immediately see the results.
But there are some disadvantages as well. As a MonoBehaviour derived script has to be part of a GameObject, it exists only within the context of a scene. Whenever a scene is unloaded, it's data disappear and the handling of this needs to be done very carefully if that data is required at a later point. Often this will be accomplished by combining it with another form of data handling. Also, if one system needs a certain value from another system, that whole other system will be necessary when testing the first, which in turn could need another one, making simple unit testing a bigger chore than it needs to be.
Another disadvantage to this method is the fact that each MonoBehaviour derived script on a GameObject is it's own instance with it's own non-shared data. In some situations this makes sense, but a lot of times various elements in the game need to access the same data. For example, when working with multiple enemies, often their max health will be the same, so when you want to change this value you need to edit every instance of that script.

File Saving
There are various ways to save data into files, which will work like a database to reference to. More often than not you will get a text file within a known format, like XML or Json. The great thing about a text file is that it has little to do with weather or not we are in a specific scene, or even if the game is running at all. Data can be easily read when needed and

written back when it is no longer necessary. Anyone working on the project can also easily open said file and change values in it. An added value of using a file like this is that multiple instances can reference this file, so the max health example issue of the MonoBehaviour class could be solved this way. Often this solution is used to save the game state when the player exits, or to keep a record of text lines used in the game, or a database of object settings.
Working with direct file saving and reading has it's disadvantages though. It's not possible to access the values within like one would with a script instance. You have to read the whole file and change a value in the desired position and than save it back to it's directory. When this is done just at the loading or unloading of a scene or game, this is perfectly fine, but when a value needs to be constantly altered and accessed by various elements, this becomes an issue both in performance and safety. 4 Another issue is ease of use, as one cannot monitor the changing of values live, and the values cannot be manually altered while the game is running. Lastly, text parsing has become more and more supported, but is not yet viable on all devices.

ScriptableObject
This base class within the Unity3D engine is often overlooked, because the starting point of working with Unity are the MonoBehaviour components, as you need these to add behaviour to any element within the game, and they already allow for data handling. But a very important difference between the ScriptableObject base class and the MonoBehaviour base class is that the ScriptableObject derived classes can exist outside of scenes. This means that the internal state of a ScriptableObject instance is not discarded at any point, but just keeps on existing while the game exists. 5
Working with ScriptableObjects has quite a few advantages; as they exist detached from scenes, their instances are easily organised within the project folder structure. To return to our example, just like with the file saving method editing the max health of enemies doesn't even require anyone to know where the enemies are, you just need to find the instance containing this variable within the project. But where it differs from the file saving method is that it's values are easily changed within the editor, there is no file that has to be opened, and these values are displayed and editable live whenever the game is running. Because ScriptableObject derived classes cannot be added to the scene, there will always be a MonoBehaviour derived class in the actual game referencing the ScriptableObject to actually display the data in the relevant way. This way it may sound like this would add overhead, but the opposite is often the case. The data and the behaviour parts are separated within this method, which makes it easier to test any system without needing additional systems, just the reference to the data. Systems do not even need to know about the existence of each other, just of the data. 6
As with each method, there are disadvantages. ScriptableObjects cannot exist within the scene as instances, so any communication to the player will have to be done with additional scripts, which at times can make the connection between data and behaviour unclear of no clear conventions have been made regarding this, or if project folder management is a mess. Also, because they do not get reset whenever a scene is loaded or unloaded, their values are always persistent, which is not always desired. For example, player health will often be reset to full health in a new level, something that won't happen automatically because the old value is not discarded.

Conclusion

As I've stated at the beginning of this paper, the selection of the ideal method will be dependant on my specific needs: I need to record the state of the game. This comprises data on each player, information on rounds that have been played, and a few collections of questions and facts used in the game.
From the information I have collected and displayed above, I conclude that for our situation ScriptableObject derived data saving would be the ideal situation. We have a lot of scenes in our app that need to access the information on the individual players throughout the game, information that has to persist throughout the game. Each of these scenes needs to handle this information differently, and sometimes change it's behaviour if the values are different, so having this data accessible without having other systems exist makes testing and improving the individual behaviour a lot easier.
For keeping record of the collections of questions and facts both the file saving and ScriptableObject methods would be viable. The decision on this is a matter of preference as changing values while testing isn't very relevant. Text parsing on mobile platforms is viable, but using ScriptableObjects might make it easier as support for this is very clear.

Sources

1. https://docs.unity3d.com/Manual/AssetWorkflow.html
2. https://docs.unity3d.com/Manual/CreatingScenes.html
3. https://docs.unity3d.com/Manual/BestPracticeUnderstandingPerformanceInUnity5.html
4. https://docs.unity3d.com/Manual/EditingValueProperties.html
5. Overthrowing the MonoBehaviour Tyranny in a Glorious Scriptable Object Revolution - Richard Fine - https://youtu.be/6vmRwLYWNRo
6. Game Architecture with Scriptable Objects - Ryan Hipple - https://youtu.be/raQ3iHhE_Kk