Persistent auto-saved progress

By Ryan Clarke

The Problem

Originally each time you start up the game all levels would be unlocked, and any medals you collected throughout your previous playthroughs would be lost! To drive the player experience as designed we must have persistent data on each start up of the game. This data will be tied into unlocking levels and showing the players previous accomplishments in each level.

Why This Is An Issue

The game is designed to be played in short bursts, without the ability for the game to remember the players progress for each level, they would need to play from the start each time. If they lost their achievements this would be very disheartening for the player. We want to provide the best experience for the player that we can so this problem needs a good solution.

Pseudo Code of LevelData Class

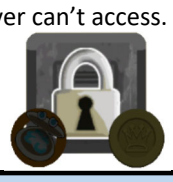
```
class LevelData
        LevelData leveldata
        int numberOfLevels
        List<Level> myLevels

        OnStart(){
                if levelData == null
                {
                        levelData = this
                        ReadList()
                }else{
                        Destroy(this)
                }
        }
        SaveList(){
                ConnectToFile('save.dat')
                Serialize(myLevels)
                CloseConnection()
        }
        ReadList(){
                if save.dat exists
                {
                        myLevels = Deserialize("save.dat")
                }else{
                        PopulateList()
                }
        }
        PopulateList(){
                for numberOfLevels
                {
                        myLevels.Add(newLevel)
                }
                UnlockFirstLevel()
                SaveList()
        }
```

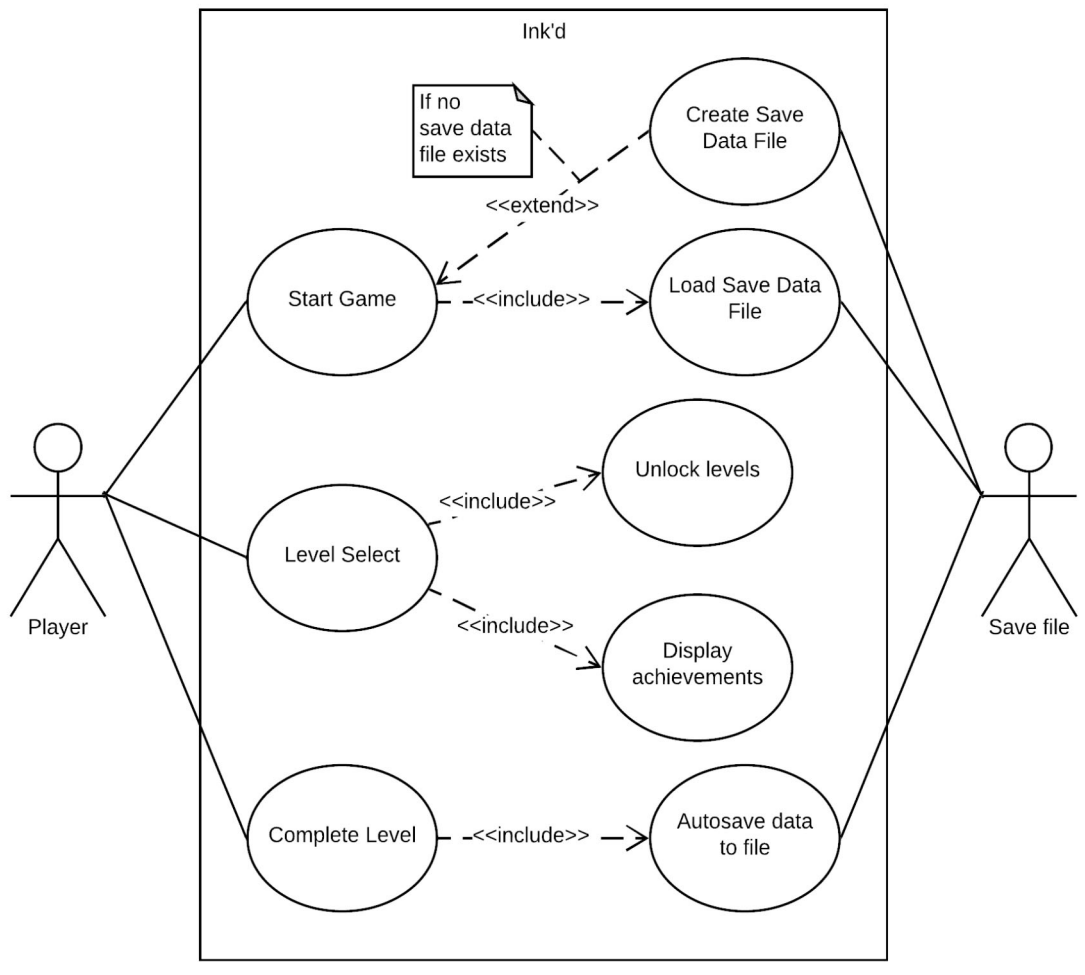
Player View

An unlocked level with a gold medal no clam!

A locked level that the player can't access.



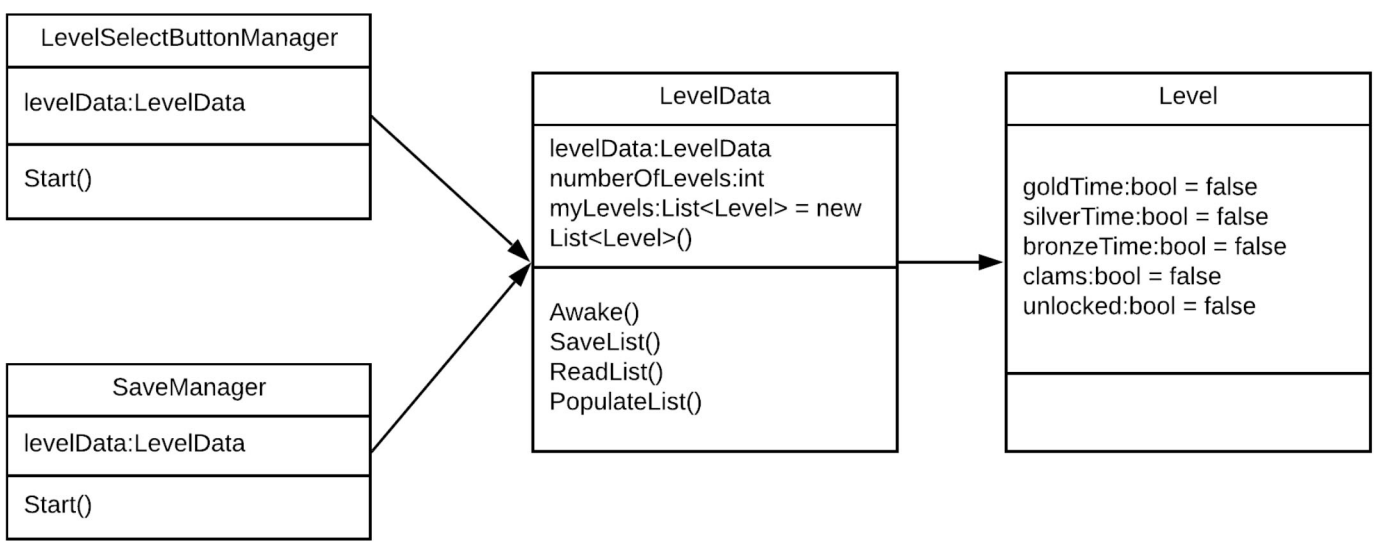
Use Case Of Save Data



My Solution

To solve the problem I started by making a Level class that has bools to represent if a level is unlocked and the medals in that level that have been collected. Then to manage these levels and the save data file they will be stored in I have made a LevelData manager, a pseudo code version of this manager can be seen on the right. The manager makes sure only one manager exists and if it's the first initialisation reads the save.dat file into a list. If the file doesn't exist yet PopulateList() is used to create a fresh list and save.dat file. Level data in the list is updated at the end of each level. The save manager seen in the class diagram below calls SaveList on the start of each new scene providing the auto save feature required to avoid data lose. When the player opens the main menu, the LevelSelectButtonManager checks the list of levels and shows the player their medals for each level and unlocks access to any levels the player has.

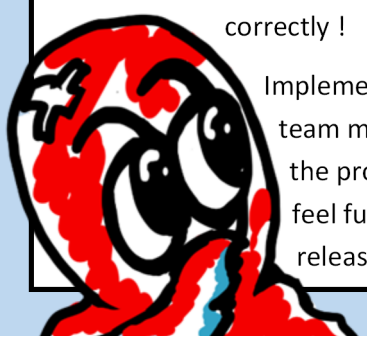
Class Diagram



Impact

The system plays a key role in making the prototype feel like a real game and allows us to provide progression through the levels instead of them all being accessible. This progression lets the story be told correctly !

Implementing this system raised team moral and excitement for the project by making the game feel fuller and much closer to a releasable product.



Improvements

Having added the system it's given us the ability to have additional unlockables that are tied to the collection of items or the speed at which players managed to complete levels. There's already talk of hidden unlockable levels!

As for improvements to the system there are actually a lot I would like to make given the time before release.
I would like to add additional encryption to the save.dat to give more protection from tampering.
There is a quick change in swapping from a list of classes to a list of structs, this won't make a huge amount of difference as we don't have many levels but should cut down on the storage size slightly.
Public variables have been used far too liberally throughout this system, I need to go back through and privatise any unnecessary public variables.
In a couple of the classes the Start function handles code that could be broken down nicely into separate functions, giving easier maintainability and more clarity.