The Drowned Team – Coding Guide
by *Maurizio Fischetti*

Introduction

Hello Drowned Team! The reason of existence of this document is to let you understand the core systems of the game and the things that are behind the core gameplay loop. This document will provide you a quick guide through the main functions of the core system to let you feel free to prototype and experimenting using the functions and the methods that can come handy when you need!

Of course, if something does not work as expected or you don't understand something, feel free to ask! I hope that you find this useful, I'm trying my best to make things as simple to understand as I can. I will update this document every time something will be add or modify as coding goes on.

The Drowned Team – Coding Guide
by *Maurizio Fischetti*

GameManager : MonoBehaviour

The main class of the game, it's the refer class for every other sub-manager in the game.
You can access it using **GameManager**.*Instance*, Instance is a Static GameManager type.

States

The game manager can have different States:

1. *States.**None***: actually, never use this.
2. *States. **Initializing***: is in initialization right after the creation, it goes in *starting* when fully initialized.
3. *States. **Starting***: goes on starting right after initialization success. The game consider itself as "Started" with this state. (Should be in this state only at the first start of the game or when player reset the game from the inside), *Raise the event GameStart.*
4. *States. **Playing***: if not in "pause" (pause menu), should always be considered in "Playing" state even in the main menu. *Raise the event GameUnpause.*
5. *States. **Paused***: should be considered in "pause" only when the player wants to stop the action of the current scene/level (pause menu). *Raise the event GamePause.*
6. *States. **Quitting***: this will quit the application after some operation. Raise the event GameQuit. Quit the application.

Publics

The following are the public entities and vars that could be accessed from other scripts.

- **Properties**:
  - Static **GameManager *Instance*:** the static instance of the GameManager object, use this for access the Game Manager's methods and other managers.
  - **EventManager *EventManager***: (get) the instance of the Event Manager.
  - **UIManager *UIManager***: (get) the instance of the User Interface Manager.
  - **AudioManager *AudioManager***: (get) the instance of the Audio Manager.
  - **DataManager**: (get) the instance of the Data Manager.
  - **LevelManager**: (get) the instance of the LevelManager.
  - **States GameState**: (get) the current state of the game.
- **Methods**:
  - ***SetState***(**State** *newState*): use this to set the state of the game. It's not possible to set ***States***.*None* and ***States***.*Initializing* manually, because there are thought to be automatic.

Use Cases

You can use it every time you need access to Managers vars and Methods...Example:

```
Private GameManager _GM =  GameManager.Instance;
Private void Start()
{
        //setting a new Game State.
        _GM.SetState(_GM.States.Paused);
}
```

AudioManager : MonoBehaviour

Audio Manager role is to manage audio clips and the audio of the game.

You can Access it like this:

```
//wherever you need access in your script
…
Private AudioManager _AM = GameManager.Instance.AudioManager;
```

Publics

- **Properties**
  - AudioSource ***AudioSourceMusic***: (get) it's located in the child object of audiomanager, audio source for music.
  - AudioSource ***AudioSourceSFX***: (get) it's located in the child object of audiomanager, audio source for SFXs.
- **Collections**
  - List<**Music**> ***Musics***: collection of Music classes.
  - List<**SFX**> ***SFXs***: collection of SFXs classes.
- **Methods**
  - ***PlayMusic***(...): Play the music with the manager's audiosource. Overloaded.
  - ***PlaySFX***(...): Play the SFX with the manager's audiosource. Overloaded.
  - ***PlayMusicLocal***(...): Play the music with a given audiosource. Overloaded.
  - ***PlaySFXLocal***(...): Play the SFX with a given audiosource. Overloaded.
- **Classes**
  - **Music**: custom class for music, made for designer to easily create a music object in the editor (combining an audioclip and a name).
  - **SFX**: custom class for SFX, made for designer to easily create SFXs object in the editor (combining an audioclip and a name).

Warning AudioManager cannot reproduce Musics or SFXs outside its collections.

Use Case

A little example:

```
…

//wherever you need access to it
AudioManager _AM = GameManager.Instance.AudioManager;
Private void Start()
{
//PlaySFX with a given name
_AM.PlaySFX("StartingSound");
}
```

EventManager : MonoBehaviour

Event Manager is a class which is responsible of the events in the game.
Every Event related to the gameplay should be put and called from here.

You can Access it like this:

```
//wherever you need access in your script
…
Private EventManager _EM = GameManager.Instance.EventManager;
```

Events Convention

When naming events, please be sure to use the correct term. An Event must be called before or after something just happened.

Use Case

Example: I want an event that informs me when a player has died.

```
//in the event manager I will name the Action and the raise event method.

Public Action PlayerDead;
Public void RaiseOnPlayerDead()
{
PlayerDead?.Invoke();
}

//in the player script I will rise the events if the life goes <= 0
…
Private void CheckHealth()
{
            If (Player.Health <=0)
            { EM.RaiseOnPlayerDead();}
}
…

//in a third class that needs to be informed of the player death.
private void OnEnable()
{
_EM.PlayerDead += OnPlayerDead() //subscribe to the event
}
Private void OnDisable()
{
_EM.PlayerDead -= OnPlayerDead() //unsubscribe from even
}

//what happen if the events get called? (The player is dead)
Private void OnPlayerDead()
{
//do something useful here
//points -= 5;
}
```

UIManager : MonoBehaviour

User Interface manager, basically it's role is to manage and manipulate the user interface in the game.
You can Access it like this:

```
//wherever you need access in your script.
…
Private UIManager_UIM = GameManager.Instance.UIManager;
```

Publics

- **Properties**
  - GameObject **UIContainer**: (get) the parent of all the UI in the game.
- **Collections**
  - List<Canvas> **UICanvasList**: this is the list of all the canvases that are child of the UIManager and UIContainer, every time you add a Canvas inside the UIContainer it will automatically add the new entry on Initialization(runtime), you can also set the list in the editor.
- **Methods**
  - ***HideUICanvas***(string *canvasName*): deactivate the canvas with the given name. The canvas object MUST be a child of the *UIManager* or *UIContainer* GameObject.
  - ***ShowUICanvas***(string *canvasName*): activate the canvas with the given name. The canvas object MUST be a child of the *UIManager* or *UIController* GameObject.

Use Case

A little example:

```
//inside another script.
Private void Start()
{
_UIM.ShowUICanvas("MainMenu");
//just be sure that a gameobject called 'MainMenu' with a canvas component exist in the
        scene! Or is at least inside the UICanvasList.
}
```

Warning: the naming is **Case Sensitive**, 'Mainmenu' will be different from 'MainMenu' or 'mainMenu' etc...

DataManager : MonoBehaviour

DataManager is the class responsible for game data saving and loading system.
In the inspector of the editor you can set some properties as encryption and filenames.

You can Access it like this:

```
//wherever you need access in your script
…
Private DataManager _DM = GameManager.Instance.DataManager;
```

Publics

- GameData GameData: it's a reference to a Serializable class, basically the data class to save. Note: there is no need to initialize a new GameData object just use this reference.
- **Methods**
    - ***Save(GameData gd):*** save into a file the given GameData class.
    - ***GameData Load( ):*** returns from a file a GameData class type.

Use Case

The GameManager Is already set to Load the game at the start and save it when the application is quitting. If you want to save/load in a specific case, you can use:

```
//you can use save and load in a specific situation as shown below
_DM.Save(_DM.GameData); //save the game
(Overwrite the original file or create a new one if file not exists)
_DM.GameData = _DM.Load() //game data now will contain the data inside the save file.
```

GameData

GameData is the class that will store all the data that needs to be saved or loaded.

You can access it like this:

```
//wherever you need access in your script
…
GameManager.Instance.DataManager.GameData;
```

Publics

- Basically, all the vars inside here should have the access public, cause you need to get and set them for the saving system.
- **Methods:** this class should not have methods!

Use Case

Just write inside the class the vars that need to be saved / loaded and give them a base value for the new game. If you want to set a var you can access it via:

GameManager.Instance.DataManager.GameData.*VariableName* = *variabeValue*;

LevelManager : MonoBehaviour

LevelManager is the class that will Manage the levels and send information to the current level loaded.

You can access it like this:

```
//wherever you need access in your script
…
GameManager.Instance.LevelManager;
```

Publics

- Contains collection of the levels prefab, the current level and the player class / prefab.
- **Methods:**
  - **LoadLevel(GameObject** *levelObj***): load the scene with the given level prefab (overrides: int *id*, string *levelName*).**
  - **LevelSelectionPrevious():** works with UI and other script, select the previous level.
  - **LevelSelectionNext():** works with UI and other script, select the next level.
  - **LoadNextLevel():** directly load the level scene with the next level (id+1) if possible
  - **LoadPreviousLevel():** directly load the level scene with the previous level (id-1) if possible

Use Case

Example:

```
//wherever you need access to it
LevelManager _LM = GameManager.Instance.LevelManager;

Private void Start()
{
//At the start select the next level
_LM.LevelSelectionNext();

//play the selected Level
_LM.LoadLevel(_LM.CurrentLevel.LevelData.LevelID); //as int or
_LM.LoadLevel(_LM.CurrentLevel.LevelName); //as string or
_LM.LoadLevel(_LM.CurrentLevelPrefab); //as GameObject (the level prefab)

}
```

LevelData

LevelData is the class that will store all the level data that needs to be saved or loaded. Each instance of the level data class should refer only on it's level instance. For each istance of the class "level" must exist only one istance of LevelData.

You can access it like this:

```
//wherever you need access in your script
…
GameManager.Instance.LevelManager.CurrentLevel.LevelData;
```

Publics

- As the GameData, all the vars inside here should have the access public, cause you need to get and set them for the saving system.
- **Methods:** this calss should not have methods!

Use Case

Just write inside the class the vars that need to be saved / loaded and give them a base value for the new game. If you want to set a var you can access it via:

GameManager.Instance.LevelManager.CurrentLevel.*VariableName* = *variabeValue*;

You can also set it through the DataManager, but its better doing it via the Level you need to change the values.