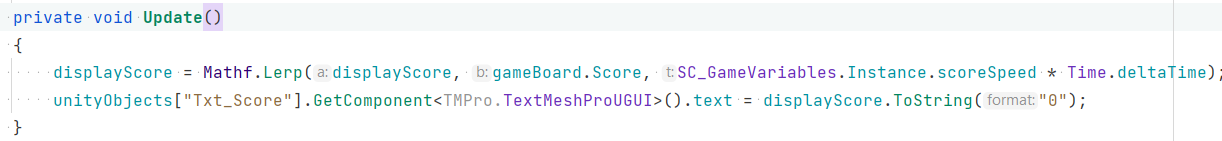
First, let's look at the example code and say if there's a need for refactoring. We can see that the overall code size is small, but the code itself is very poorly organized, which is a good indication that it needs to be fully refactored. Let's look for some examples of poor style/composition:

1. There is no separation between view and busyness logic. Gems GameObjects, which are clearly views, alter their and their neighbor's coordinates in the controller.  
   otherGem.posIndex.y++;
2. There is no model. The board is represented in the controller as a simple two-dimensional array, while it is certainly much better to represent it as a serializable model. This will allow us to better test our game by starting it from a certain state saved in the model (we can provide an editor for that). We also can incapsulate the board logic, like selecting a gem, into the model.
3. Match finding logic is repeated two times in GameBoard.FindAllMatches() and GameBoard.MatchesAt(), which turns into a problem when the match rules become more complex. All match logic should be located in one place to avoid code duplication problems.
4. The score indicator is not a view component. It is accessed in a controller through the GetComponent method in an Update loop, which is highly suboptimal.  
   
5. It is better to separate the BoardController from the GameController. The GameController can have its own model in the future.
6. Animations are not event-based and their end is awaited using delays, requiring manual synchronization if animation speed changes.

Also, there is a need for many other code style improvements.

Let's start by creating a Git/LFS repository in our project to document our changes and make them easily revertable if needed.

We will also put the old code into the \_LegacyCode folder to quickly reference it if needed.

First, we create GameController and BoardController classes, which are the singletons accessed through their static Instance properties. I would prefer to use dependency injection like Zenject or VContainer in a large project, though.

**Task 1.**

Let's isolate the gems matching logic in a couple of special classes HorizontalMatchDetector and VerticalMatchDetector, which both inherit from the common abstract class MatchDetector. Having match logic separated this way allows us to achieve two goals:

1. We can reuse the logic when filling the board, refilling the board, and doing matches in a game. There is no code duplication.
2. If some rules for more complex matches appear (like T-matches), we can easily add them as a separate class.

Now, we can easily implement these match detectors in the RefillBoard method of the BoardController so that the new gems cascading into place do not create unnecessary matches.

**Task 2.**

The gem pooling system works on classes located in the Unility/Pooling directory and is based on implementing the IPooledPrefab<T> interface by the prefab's class. It's really easy to use and requires adding only a few lines to the class. It's better to initialize the pools first like it is done with the IniPrefabPool() method in the game, but the pools will init themselves automatically otherwise. As usual, care must be taken not to use initialization in the Start() method in a prefab's class. Also, the prefab's state must be reset entirely after returning from the pool.

**Task 3.**

As the gems matching logic is not separated in MatchDetector class, it is very easy to add the rule for matching the bombs now. We also have to add colorful bombs to the project and the required methods to BoardController class to support bomb explosion and destruction. The time settings for the bomb are in the GameSettings class. Using the special Match class for detected matches allows us to easily place the logic for the new bomb detection there (when 4+ gems are matched together).

Task 5

It's not really easy to implement gem drop animation exactly like it is done in a Royal Match. What can we say about the visual reference:

1. Apparently, some easing is used when dropping the gems. It looks like the exponential easing works the best, but it's not certain.
2. Gems drop in a sequence, each row falling synchronously and every next row falling after a short delay.
3. After gems drop, they do a small jump, starting from the bottom gem, which looks like a shake from the fall.

To imitate this behavior, I created a special MoveSequence within the BoardController class, which allows us to define a sequence of gems that need to be dropped and then play it twice for the actual dropdown and the jump reaction.

The animation parameters are listed in a GameSetting object:

delayStep: the time before dropping the next row

shakeTime: the time to shake the gem after the drop

jumpPower: the height of the jump