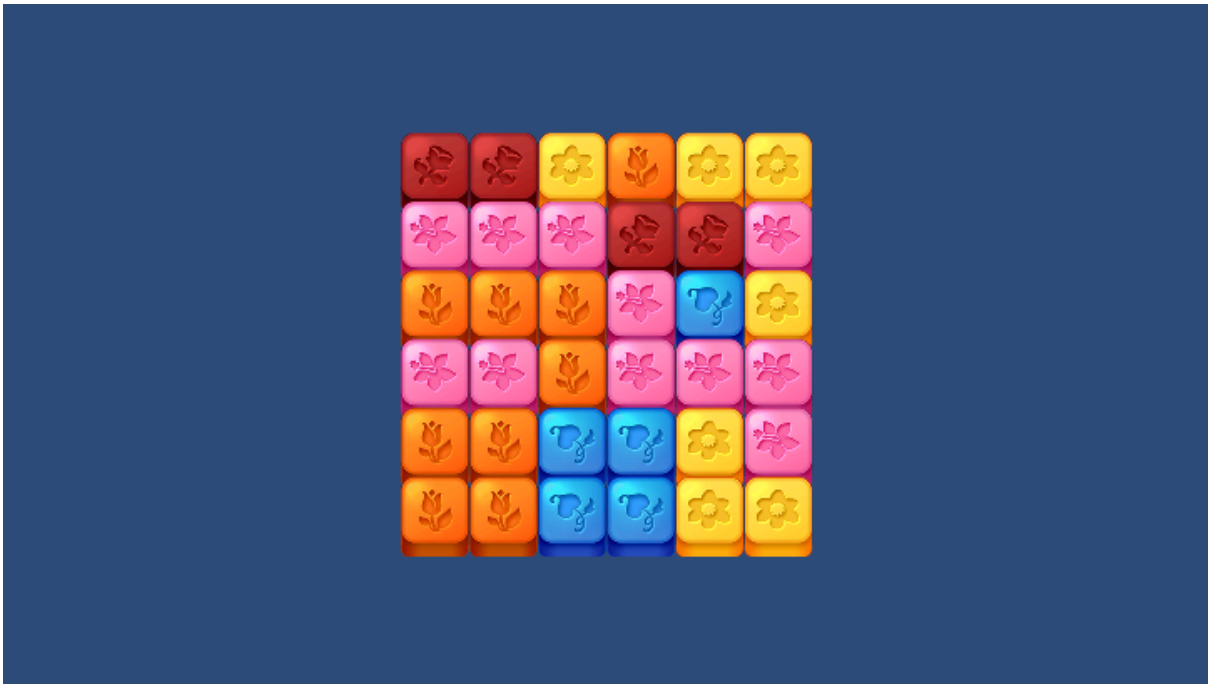
1º Assignment - Add The Yellow Piece

I began by setting up the Git repository, where you can view all the implemented changes here: [GitHub Repository].

To start, I reviewed the existing elements in the Scene and identified the `GameObject BoardRenderer`. I noticed that the yellow sprite was missing from the serializable list, so I added it. However, another script—`PieceSpawner`—was responsible for managing piece generation. By adjusting the array range from `0,4` to `0,5`, yellow pieces started appearing in the game.



2º Assignment - Fix a bug

To resolve this issue, I reviewed the `Board` script, which controls how pieces are processed. The solution was found in the `SearchForConnected` method: I modified the condition `neighbor.type == piece.type`. Previously, this condition attempted to resolve every piece above pink pieces, inadvertently affecting red and yellow pieces as well.

Before reaching this solution, I also examined `FindAndRemoveConnectedAt`, `Resolve`, `CreatePiecesAtTop`, and `MovePiecesOneDownIfPossible`. Since these functions are integral to the game's logic, I wanted to ensure they were working correctly—but after reviewing them, I confirmed they were functioning as intended.

3º Assignment - Add Animation To The Game Board

First, in the `Board` script, I created a reference to `BoardRenderer` so that whenever a piece is created or moved, an event is triggered:

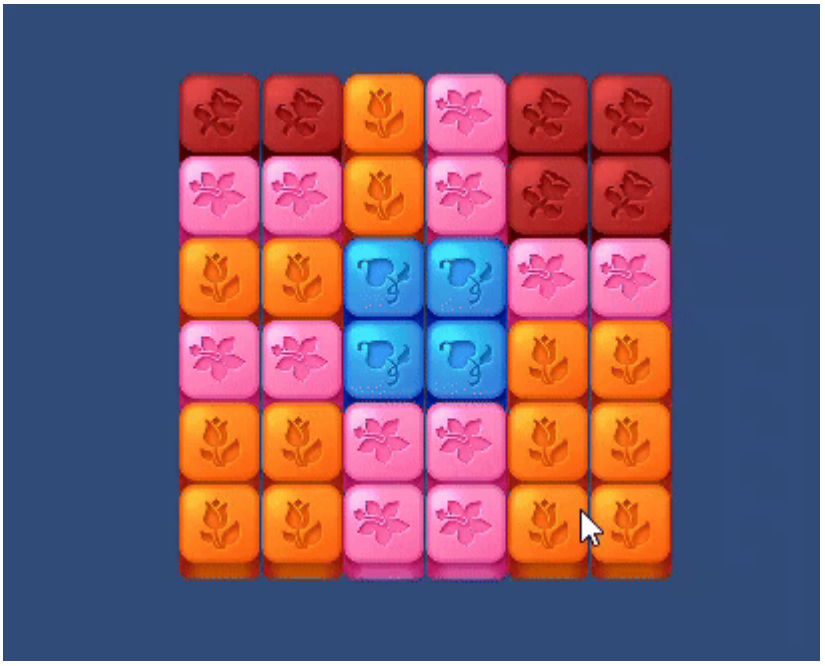
```
boardRenderer?.AddCreatedPiece(pieceToMove);
```

This adds the affected pieces to a list for further processing.

The `BoardRenderer` script required more significant changes. I added logic to `CreateVisualPiecesFromBoardState`, which now handles all pieces while applying special treatment to those in the newly created list. This behavior is implemented through two new methods:

- `AnimatePiece`
- `AnimateCoroutine`

These methods register the positions of each piece and the one immediately above it, making them fall gradually. This creates a distinction between newly created pieces and those that were simply moved. For smoother animations, I would typically use an easing plugin.

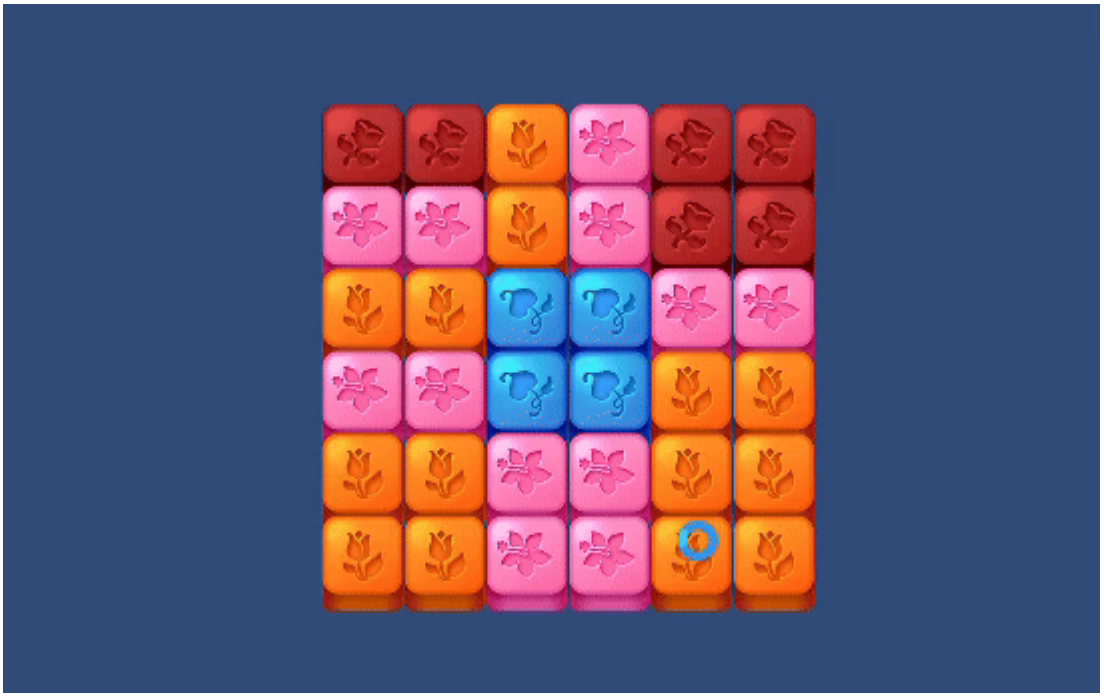


4º Assignment - Add Power Piece

For this exercise, I made several changes to the `Board` script, along with some minor modifications to `IBoard`, which I will discuss shortly.

Key Changes:

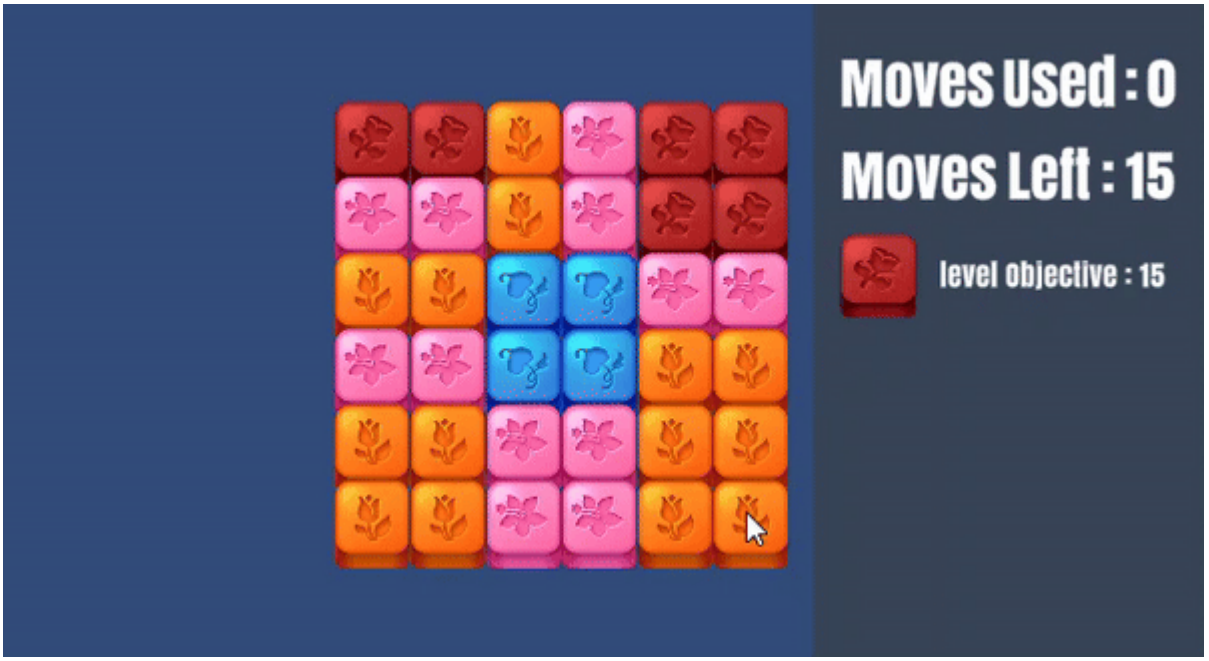
1. **New Enum for Connection Types**
   I introduced a public enum called `TypeOfConnection`, which determines the type of connection that occurs during piece resolution.
2. **Modifications to `FindAndRemoveConnectedA`**
   I updated this method to check for connections of five or more pieces. When this condition is met, a random piece from the new array—Bombs—appears. These Bombs can be either horizontal or vertical.
3. **Changes to `Resolve`**
   I added a new method, `ResolveSpecialPieces`, which modifies the connection type enum. It adjusts the enum value based on whether the piece is vertical or horizontal.
4. **Enhancements to `SearchForConnected`**
   This method is responsible for identifying neighboring pieces. I added an `else if` condition related to the new enum, altering the criteria for what qualifies as a neighbor. In this case, all pieces can be considered neighbors under specific conditions.
5. **Updating `GetNeighbors`**
   I modified this method so that, depending on the enum value, it only searches for neighbors in a specific direction.



5º Assignment - Add Winning/Losing Conditions

To solve this issue, I created four new scripts:

1. **`LevelDataReferencer`**
   A static script responsible for managing level-related data using getters and setters, allowing modifications as needed.
2. **`LevelDataUI`**
   Handles UI updates based on the data from `LevelDataReferencer`. It contains `SerializeField` references for each field that needs to be displayed and includes a method to update all UI elements accordingly.
3. **`VictoryConditionManager`**
   This script checks whether victory or loss conditions are met. It references `GameObjects` responsible for resetting the game and includes a method, `CheckVictoryConditions`, which updates an internal boolean to track whether the game has ended.
4. **`LoadingScript`**
   Normally, I take a static approach to loading, but in this version of Unity, invoking an enum via a button click in the editor isn't allowed. To work around this, I created a general loading script that can exist multiple times in the scene. It uses a `SerializableField` reference to modify the enum in the editor, allowing the scene to be restarted when needed.



6º Assignment - Propose Improvements To The Program

The existing **Boot script** is useful for programmers, but it can be challenging for designers to implement changes dynamically. To make it more accessible, I made significant adjustments, allowing for:

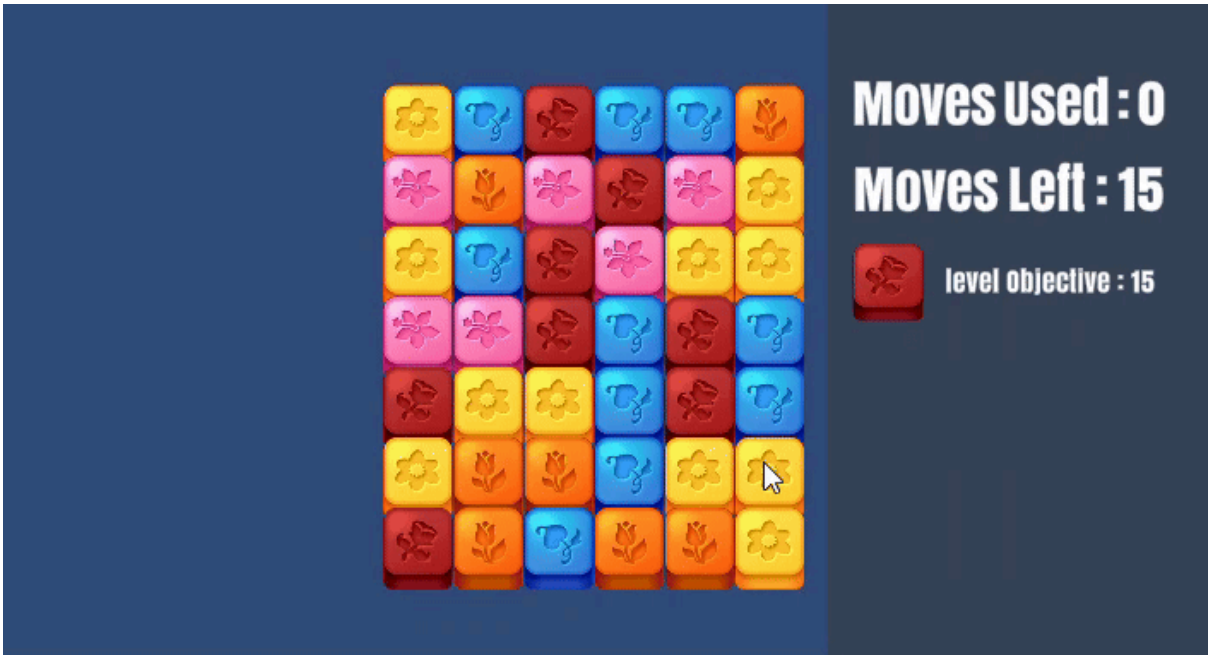
- **Modifying board width and height** dynamically.
- **Introducing a `RowData` list**, which stores collections of numbers, enabling individual row modifications to fine-tune level design.
- **A more flexible level creation process**, making it easier to tweak levels without manually designing each one.

Since handcrafted level design isn't always feasible, I also explored an additional approach:

Randomization & Data Persistence

With the extensive modifications to the Boot script, I expanded its functionality by:

- **Adding randomizer logic**, generating procedural levels dynamically.
- **Enabling data persistence**, allowing random level configurations to be saved and modified later.
- **Working with a random canvas**, so levels can be adjusted even after being procedurally generated.


Moves Used : 0
Moves Left : 15
level objective : 15