GOOD JOB MATCH

2D Collapse/Blast Mechanic Game Case Study

Yiğit Durmuş

Technical Document: 2D Collapse/Blast Mechanic Game

1. Project Description

This project is a 2D puzzle game with collapse/explosion mechanics, developed from scratch without using any templates in 10 days for Good Job Games. The game is designed to be compatible with both mobile (Android/iOS) and PC platforms, ensuring a smooth and responsive user experience across different screen sizes and resolutions. Players interact with a dynamic game board by clicking or tapping groups of matching tiles to eliminate them and generate new tiles.

The primary objectives of this project are:

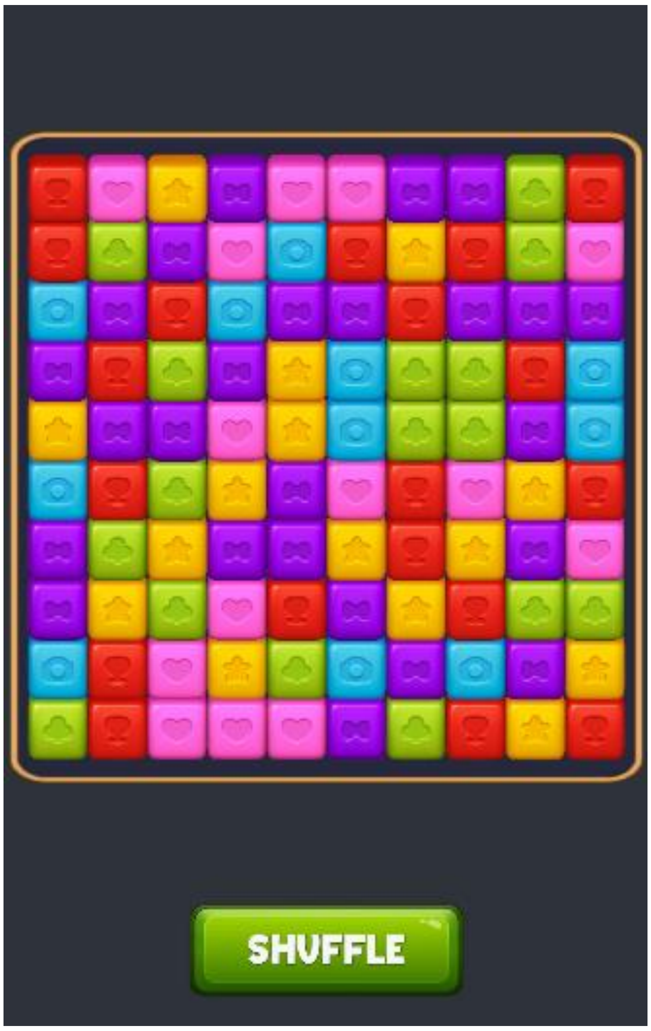
To create a **highly optimized, scalable,** and **performant** game board system.

To implement **non-deadlock shuffling**, ensuring that the board remains solvable at all times.

To provide a **customizable level editor**, allowing designers to create and modify levels efficiently.

To **support animation and asset customization** directly from Unity's Inspector.

To deliver **high-performance rendering and data management** through object pooling and efficient algorithms.



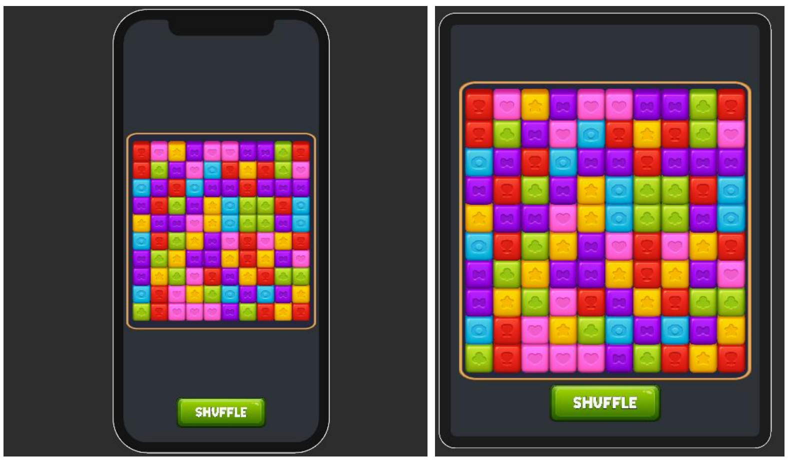
2. Key Features

2.1. **Dynamic Board Generation**

The board adapts dynamically to different screen sizes and resolutions. Implemented in BoardUI.cs, the CreateBoard() method calculates the optimal tile size based on:

1. The camera's aspect ratio.
2. The desired number of rows and columns.

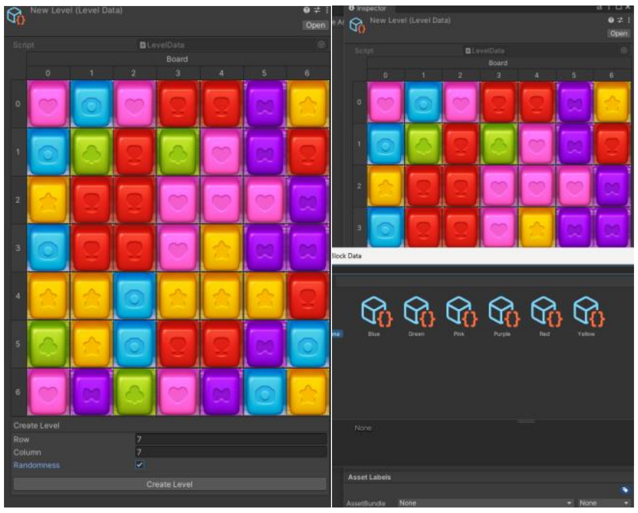
Ensures consistent gameplay on any device resolution by adjusting spacing and margins accordingly.



2.2. **Level Editor**

Levels can be created and edited using a JSON-based data structure (LevelData). Easily create automatic or manual levels with editor. Developers and designers can define:

1. Board dimensions (number of rows/columns).
2. Initial block placements and colors.
3. Special conditions (e.g., obstacles, power-ups).



Script: LevelData

Create Level
Row: 7
Column: 7
Randomness: ✓
Create Level

None

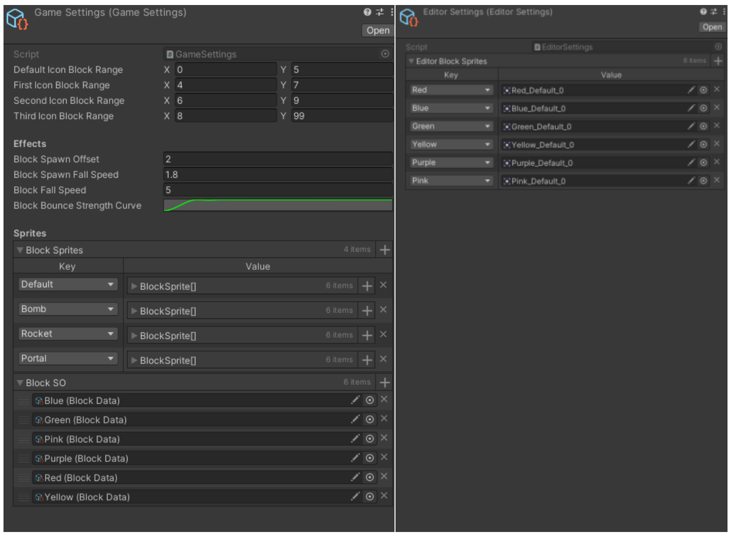
AssetBundle: None | None

2.3. **Adjustable Animations & Assets from Inspector**

All animations and assets (e.g., tile sprites, effects) can be configured directly from the Unity Inspector. Customizable parameters:

1. Block movement animations
2. Explosion effects
3. Shuffle animations

Allows designers to iterate on the visual style without modifying the code.



Game Settings (Game Settings)

Script	GameSettings	
Default Icon Block Range	X 0	Y 5
First Icon Block Range	X 4	Y 7
Second Icon Block Range	X 6	Y 9
Third Icon Block Range	X 8	Y 99

Effects

Block Spawn Offset	2
Block Spawn Fall Speed	1.8
Block Fall Speed	5
Block Bounce Strength Curve	

Sprites

Block Sprites — 4 items

Key	Value
Default	BlockSprite[] — 6 items
Bomb	BlockSprite[] — 6 items
Rocket	BlockSprite[] — 6 items
Portal	BlockSprite[] — 6 items

Block SO — 6 items

- Blue (Block Data)
- Green (Block Data)
- Pink (Block Data)
- Purple (Block Data)
- Red (Block Data)
- Yellow (Block Data)

Editor Settings (Editor Settings)

Script: EditorSettings

Editor Block Sprites — 6 items

Key	Value
Red	Red_Default_0
Blue	Blue_Default_0
Green	Green_Default_0
Yellow	Yellow_Default_0
Purple	Purple_Default_0
Pink	Pink_Default_0

2.4. **Non-Deadlock Shuffle System**

The game prevents situations where the board has no valid moves by implementing an intelligent shuffle algorithm.

1. Detects deadlock conditions using IsBoardPlayable().
2. Rearranges blocks randomly in ShuffleBoard().
3. Ensures that a valid move is always present using HandleDeadlock().

2.5. **Seamless Gameplay**

Players can execute back-to-back moves without any cooldown or waiting time.The game allows continuous interaction, ensuring a fluid and engaging experience.No forced delays between moves, making gameplay feel fast and responsive.

2.6 **Flexible Project Architecture With Scriptable Object**

This architecture allows for extensive customization and scalability, enabling the creation and modification of various elements such as blocks, levels, settings, and more without significant constraints. By leveraging Scriptable Objects, dependencies are minimized, ensuring a highly modular and maintainable structure. This design not only simplifies the addition of new features but also makes the project adaptable to future changes, providing a robust foundation for both development and iteration.

3. Algorithms Used in the Project

3.1. **Flood Fill Algorithm for Matching Groups**

Implemented in GetMatchingTiles(Tile startTile) in Board.cs. Used to find and clear connected tiles of the same color. Process:

1. Start from the clicked tile.
2. Check neighboring tiles recursively (up, down, left, right).
3. Mark visited tiles using a boolean array (_visitedCells).
4. Store matched tiles in a list (_matchedTiles).

If the group is large enough, remove the blocks; otherwise, shake them to indicate an invalid move. Optimizations:

1. Uses a Stack-based implementation instead of recursion to prevent stack overflows.
2. SetFloodFillCache() prepares memory beforehand for visited cells, reducing allocation overhead.

3.2. **Shuffle Algorithm (Ensuring No Deadlocks)**

Implemented in ShuffleBoard() in Board.cs. Ensures that the board remains solvable at all times. Process:

1. Extract all blocks into a temporary list.
2. Randomly shuffle the list using the Fisher-Yates Shuffle.
3. Reassign blocks to tiles.

Verify board playability using HandleDeadlock().

3.3. **Column Reordering & Filling System**

Implemented in OrderColumn(int columnNum) in Board.cs. Ensures that blocks fall into empty spaces after matches are cleared. Calls BlockManager.Instance.SpawnBlock() to generate new blocks only when needed.

3.4. **Deadlock Prevention (HandleDeadlock() Method)**

Used to prevent situations where no valid moves are left. Process:

1. Generates random flood fill values using Helpers.GenerateRandomDivisors() by board size.
2. Selects random tiles to be filled using Helpers.SelectRandomElements().
3. Calls FillTilesWithAmount() to ensure there is always at least one valid move and it strategically places tiles in a unique manner to create a more natural and randomized arrangement.

4. Performance Considerations

4.1. **Object Pooling (BoardPool.cs)**

Instead of creating/destroying objects dynamically, the game reuses pre-allocated objects. Reduces CPU overhead and garbage collection.

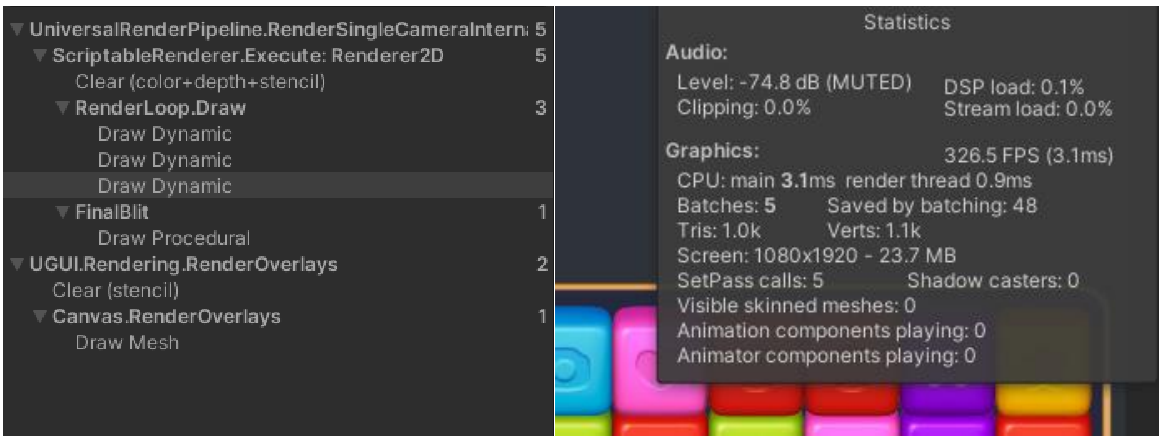
4.2. **Asynchronous Operations (UniTask)**

All algorithms, initializations and animations run asynchronously using UniTask.

Ensures smooth animations and prevents game lag.

4.3. **Optimized Rendering**

By utilizing dynamic batching, along with optimized renderer settings and sprite shaders, maximum performance has been achieved. The game minimizes draw calls by combining similar elements where possible.



```
▼ UniversalRenderPipeline.RenderSingleCameraIntern 5
  ▼ ScriptableRenderer.Execute: Renderer2D        5
      Clear (color+depth+stencil)
    ▼ RenderLoop.Draw                              3
        Draw Dynamic
        Draw Dynamic
        Draw Dynamic
    ▼ FinalBlit                                    1
        Draw Procedural
▼ UGUI.Rendering.RenderOverlays                   2
    Clear (stencil)
  ▼ Canvas.RenderOverlays                          1
      Draw Mesh
```

```
Statistics
Audio:
  Level: -74.8 dB (MUTED)      DSP load: 0.1%
  Clipping: 0.0%               Stream load: 0.0%
Graphics:                      326.5 FPS (3.1ms)
  CPU: main 3.1ms  render thread 0.9ms
  Batches: 5       Saved by batching: 48
  Tris: 1.0k       Verts: 1.1k
  Screen: 1080x1920 - 23.7 MB
  SetPass calls: 5          Shadow casters: 0
  Visible skinned meshes: 0
  Animation components playing: 0
  Animator components playing: 0
```

4.4. Optimized Animation & Effect System

This aligns well with the concept of GC-free, performance-efficient animation**s** using specialized libraries.

4.5. Efficient Data Structures:

Stacks and lists are used to manage tile groupings efficiently, avoiding unnecessary garbage collection (GC) events.

Memory allocations are minimized by reusing lists instead of creating new instances.

RAM usage is optimized by avoiding redundant object references and clearing unused data promptly.

4.6. Optimized Algorithms:

Flood Fill Algorithm is optimized for fast tile matching and efficient memory usage.

Fisher-Yates Shuffle is used to efficiently randomize board elements, ensuring fairness and preventing bias.

Custom algorithms are implemented to handle deadlocks and ensure smooth game progression without performance drops.

5. Technical Details

Unity Version: 2022.3f.13f1

3rd Packages: Prime Tween, Odin, UniTask

Github: https://github.com/hidro0x/CaseStudyGJG