1. Project Purpose and General Overview

This project is a **grid-based block puzzle game** developed using the **Unity game engine** and built on a match-style gameplay logic.
The main objective of the game is to **clear adjacent blocks of the same color by clicking on them**, update their icon levels accordingly, and ensure that the game board **always contains at least one playable move**.

During development, **performance was a primary focus**. The project was optimized to minimize CPU, GPU, and memory usage by utilizing **object pooling**, **allocation-free operations**, and **reusable data structures**. As a result, the game is optimized to run with **stable FPS**, especially on mobile devices.

---

2. Core Gameplay Mechanics

The game board consists of a **two-dimensional grid structure**.
Each cell in the grid contains a **Block** object.

The gameplay follows these rules:

- Blocks of the same color that are **adjacent horizontally or vertically** (left, right, up, down) form a **group (match)**.
- The icon level of a block changes based on the **number of blocks in its group**.

Icon levels are defined as:

- Normal
- A
- B
- C

When the player clicks on a block:

- If the selected block belongs to a group of **at least two blocks**:
    - All blocks in the group are removed
    - Gravity is applied
    - The board is refilled
    - Block icons are updated
    - A potential **deadlock** state is checked

---

3. Script Structure and Responsibilities

3.1 BoardManager.cs

This script acts as the **central controller of the game**.
All board-related logic is managed through this component.

Main responsibilities include:

- Creating and managing the grid
- Performing Flood Fill (neighbor searching)
- Match detection
- Applying gravity
- Refilling the board
- Deadlock detection
- Shuffle and ForceTeleport mechanics
- Performance optimizations

Key performance techniques used:

- Use of Physics2D.RaycastNonAlloc (prevents Garbage Collector allocations)
- Avoiding Instantiate / Destroy calls through object pooling
- Non-recursive, stack-based Flood Fill implementation
- Fast cell tracking using a bool[] visited array
- Reuse of lists and collections to avoid allocations
- Preallocated data structures for shuffle operations

3.2 Block.cs

This script represents **each individual block cell** on the board.

Responsibilities include:

- Storing block color and icon data
- Updating sprites (icons) based on group size
- Keeping track of grid coordinates
- Synchronizing logical and visual positions

Important methods:

Init()
Called when the block is first created and initializes its default state.

UpdateIcon(int groupSize)
Determines the icon level based on the size of the connected block group.

SetCoordinates(x, y)
Updates both the logical grid coordinates and the visual position of the block.

3.3 GameSettings.cs (ScriptableObject)

This script serves as the **configuration file of the game**.
All values can be adjusted directly through the Unity Inspector.

Configurable parameters include:

- Grid width and height
- Icon level thresholds
- Number of available colors
- Sprite sets for each color

Advantages:

- Game balance can be adjusted without changing code
- Easy creation of different levels or themes
- Clear separation between game design and programming logic

3.4 Types.cs

This file contains enum definitions.

- ColorTypes → Block colors
- IconTypes → Block icon levels

This structure:

- Improves code readability
- Prevents invalid value usage
- Provides a safer and more maintainable codebase

---

4. Flood Fill (Neighbor Detection) Logic

The project uses a **non-recursive, stack-based Flood Fill algorithm**.

Reasons for choosing this approach:

- Recursive Flood Fill may cause **stack overflow** on large grids
- Iterative (stack-based) Flood Fill is faster and safer

How it works:

- The starting block is pushed onto a stack
- Visited cells are marked using a visited[] array
- Adjacent blocks of the same color are added to the stack
- The entire group is detected in a single pass

This Flood Fill logic is reused for:

- Match detection
- Icon updates
- Block removal after player input

---

5. Object Pooling System

Blocks are **never destroyed** during gameplay.

Workflow:

- Blocks are pre-created at game start using PrewarmPool()
- Removed blocks are returned to the pool
- New blocks are retrieved from the pool when needed

Benefits:

- No Garbage Collector spikes
- No FPS drops
- Stable performance on mobile devices

---

6. Deadlock Detection

The game automatically handles situations where **no valid moves remain**.

Process:

1. HasMatches() checks whether at least one valid move exists
2. If not, ShuffleBoard() is executed
3. If no matches still exist, ForceTeleport() is triggered

ForceTeleport() behavior:

- Two distant blocks of the same color are identified
- One block is swapped with a neighboring block
- The board is guaranteed to become playable

---

7. Performance Notes

- No Garbage Collection allocations
- No heavy operations inside the Update loop
- Redundant processing is avoided
- Mobile-friendly architecture
- Stable performance even on large grids

Github Repository: https://github.com/fevzibagriacik/Block_Blast_Game