HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY

SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

GRADUATION RESEARCH I

Development of a Logic Puzzle Game on Android: Architecture and Implementation

Student Name: Nguyen Ha Tam
Student ID: 20215241
Class: Global ICT 02 K66
Supervisor: Dr. Nguyen Ba Ngoc

HANOI, February 2026

Abstract

This report documents the design and implementation of "Block Adventure," a logic puzzle game developed for the Android platform using the Unity Game Engine. The project aims to construct a robust, scalable software architecture that addresses the specific challenges of grid-based mobile games, moving beyond simple prototyping to professional engineering practices.

The research focuses on three key technical pillars: implementing a complex 9x9 grid evaluation system (checking rows, columns, and 3x3 sub-grids), utilizing Data-Driven Design (DDD) via ScriptableObjects for dynamic content management, and establishing an Event-Driven Architecture to decouple game logic from the user interface. Additionally, the project integrates mobile-specific features such as touch-based drag-and-drop mechanics and screen scaling. The result is a functional, optimized Android application that demonstrates a clear separation of concerns and maintainable code structure.

Contents

1 Introduction **2**
- 1.1 Context . . . 2
- 1.2 Problem Analysis . . . 2
- 1.3 Objectives . . . 2

2 Tools and Technology **3**
- 2.1 Unity Engine (2022 LTS) . . . 3
- 2.2 JetBrains Rider . . . 3

3 Gameplay Design and Specifications **4**
- 3.1 Core Mechanics . . . 4
- 3.2 Dynamic Difficulty and Progression . . . 4
- 3.3 Technical Specifications . . . 4
  - 3.3.1 Data-Driven Design (DDD) . . . 4
  - 3.3.2 Event-Driven Communication . . . 5

4 Detail Functionalities **6**
- 4.1 Application Flow and Menu System . . . 6
- 4.2 Core Gameplay Interface (HUD) . . . 8
- 4.3 Interaction and Visual Feedback . . . 9
  - 4.3.1 Snap Preview (Shadow System) . . . 9
  - 4.3.2 Invalid Move Feedback . . . 10
- 4.4 Progression and Dynamic Difficulty . . . 11
  - 4.4.1 Threshold Color Shifting . . . 11
  - 4.4.2 Combo and Bonus System . . . 12
- 4.5 Resource Management: The Request System . . . 13
  - 4.5.1 Active State . . . 13
  - 4.5.2 Consumption State . . . 14
  - 4.5.3 Depleted State . . . 15

5 Implementation of Complex Functionality **17**
- 5.1 Tooling and Data-Driven Workflow . . . 17
  - 5.1.1 ScriptableObject Architecture . . . 17
  - 5.1.2 Custom Inspector Implementation . . . 17
- 5.2 Algorithmic Optimization: Grid Evaluation . . . 18
- 5.3 System Architecture: The Observer Pattern . . . 19
- 5.4 Data Persistence: Binary Serialization . . . 19

6 Experimental Results **21**

7 Conclusion and Future Work **22**
- 7.1 Conclusion . . . 22
- 7.2 Future Work . . . 22

Introduction

1.1 Context

The mobile puzzle genre is characterized by simple mechanics that hide complex underlying logic. While the market is flooded with "Block Puzzle" clones, many suffer from poor performance and rigid codebases that make adding new features difficult. This project, "Block Adventure," takes inspiration from classics like *Sudoku* and *1010!*, but approaches the development through the lens of Software Engineering rather than just Game Design. By prioritizing architecture, the project aims to create a foundation that is easy to expand, test, and optimize for mobile devices.

1.2 Problem Analysis

Developing a logic puzzle game for Android presents several distinct technical challenges that this research aims to solve:

- **Logic-Visual Dissonance:** In grid-based games, the "Logical Grid" (the data array) and the "Visual Grid" (the GameObjects) often drift out of sync, leading to game-breaking bugs where valid moves are rejected.
- **Complex Evaluation Rules:** Unlike standard line-clearing games, this project requires evaluating multiple conditions simultaneously—rows, columns, and 3x3 sub-grids (similar to Sudoku)—which requires efficient algorithms to prevent frame drops on mobile devices.
- **Scalability of Content:** Hard-coding shape data (e.g., coordinates for T-shapes or L-shapes) inside C# classes leads to a rigid system. A method is needed to separate data from logic.

1.3 Objectives

The primary objectives of this research are:

1. To develop a complete 2D puzzle game prototype specifically optimized for the Android platform.
2. To implement a **Data-Driven** architecture that separates game configuration (Shapes, Levels) from runtime logic.
3. To engineer a decoupled **Event-Driven** system for handling game states (Scoring, Bonuses, Game Over).
4. To implement advanced gameplay features including sub-grid evaluation, dynamic difficulty scaling (color changes), and resource management (Request Button).

Tools and Technology

2.1 Unity Engine (2022 LTS)

The project utilizes Unity 2022 Long Term Support (LTS) as the primary development engine. Unity provides a robust component-based architecture (GameObject-Component pattern) that is ideal for modular development. Key advantages utilized in this project include:

- **2D Physics Canvas System:** Unity's built-in RectTransform and Raycast systems allow for precise handling of touch inputs and UI scaling across different Android screen resolutions.
- **ScriptableObject Architecture:** A native Unity feature that allows data to be stored as assets independent of class instances, which is crucial for the Data-Driven Design approach of this project.
- **Android Build Support:** Unity's seamless integration with the Android SDK/NDK allows for rapid iteration and testing on physical devices, ensuring the touch mechanics feel responsive.

2.2 JetBrains Rider

JetBrains Rider was selected as the Integrated Development Environment (IDE) over visual Studio due to its superior static code analysis. Rider offers specialized inspections for Unity API usage, identifying expensive operations (such as 'GetComponent' calls inside 'Update' loops) that could hinder performance on mobile devices. Its rigorous refactoring tools ensure the C# codebase remains clean and adheres to standard naming conventions.

Gameplay Design and Specifications

3.1 Core Mechanics

"Block Adventure" is a logic puzzle game played on a 9x9 grid. The gameplay combines the spatial management of *Tetris* with the sub-grid logic of *Sudoku*.

- **Drag and Drop:** Players drag complex shapes from a "Storage" area at the bottom of the screen onto the main grid.
- **Evaluation Rules:** The game engine checks for three distinct clear conditions:
  1. **Rows:** Filling any horizontal row of 9 blocks.
  2. **Columns:** Filling any vertical column of 9 blocks.
  3. **Sub-Grids:** Filling any of the nine 3x3 zones marked on the board.
- **Request System:** A strategic button allows players to request a new set of shapes if the current ones are unfavorable. This feature has a limited number of uses (default: 3), adding a layer of resource management.

3.2 Dynamic Difficulty and Progression

To maintain player engagement, the game features dynamic state changes:

- **Color Progression:** As the player's score reaches specific thresholds (e.g., 500 points, 1000 points), the spawning shapes change color palettes. This provides visual feedback of progress.
- **Color Bonus:** Clearing specific "Bonus Blocks" (identified by unique colors) triggers a specialized "Bonus Event," awarding extra points and visual effects.

3.3 Technical Specifications

The project is built as a 2D application targeting Android 10.0 and above. The architecture follows two strict design patterns:

3.3.1 Data-Driven Design (DDD)

We utilize Unity's **ScriptableObjects** to decouple data from logic. This is applied in three specific areas:

1. **Shape Configuration:** Each shape (Tetromino, Pentomino) is defined as a 'ShapeData' asset containing its matrix coordinates. This allows designers to create new shapes in the Inspector without coding.
2. **Level Configuration:** Difficulty parameters (spawn rates, color thresholds) are stored in config assets, allowing for easy balancing.

3. **State persistence:** High scores and player preferences are serialized separate from the game logic.

3.3.2 Event-Driven Communication

To prevent tight coupling between the Grid logic and the UI, an **Observer Pattern** is implemented via C# Actions (‘GameEvents.cs‘). This allows disparate systems to communicate without direct references. Key events include:

- `CheckIfShapeCanBePlaced`: Triggered when a player releases a shape.
- `OnLineCleared`: Triggered when the grid logic detects a full line or sub-square. The Audio Manager and Score UI listen to this to play sounds and update text.
- `RequestNewShapes`: Triggered by the UI button, instructing the ‘ShapeStorage‘ to regenerate.
- `OnBonusTriggered`: Broadcasts when a specific color clear occurs, triggering popup animations.

Detail Functionalities

This chapter documents the realized software artifact. It demonstrates the application's user interface (UI) flow, visual feedback systems, and specific gameplay features implemented for the Android platform.

4.1 Application Flow and Menu System

The application opens with a clean, high-contrast Main Menu designed for mobile readability. To maintain a polished user experience, UI transitions are animated rather than static.

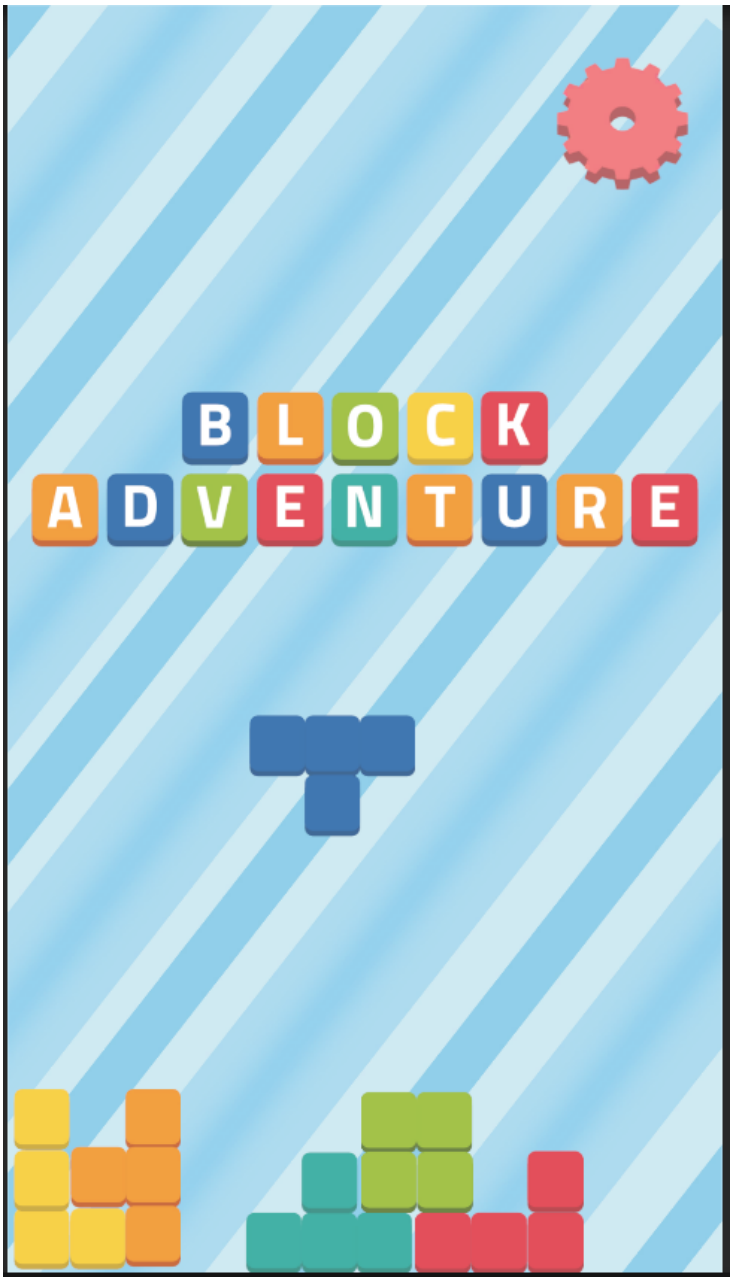

Figure 4.1: Main Menu Interface. The design uses a pastel color palette to reduce eye strain. The "Settings" cog (top right) triggers an animated sub-menu.

The Settings menu utilizes a Unity Animator Controller to slide in and out of view, preserving screen real estate on smaller devices.

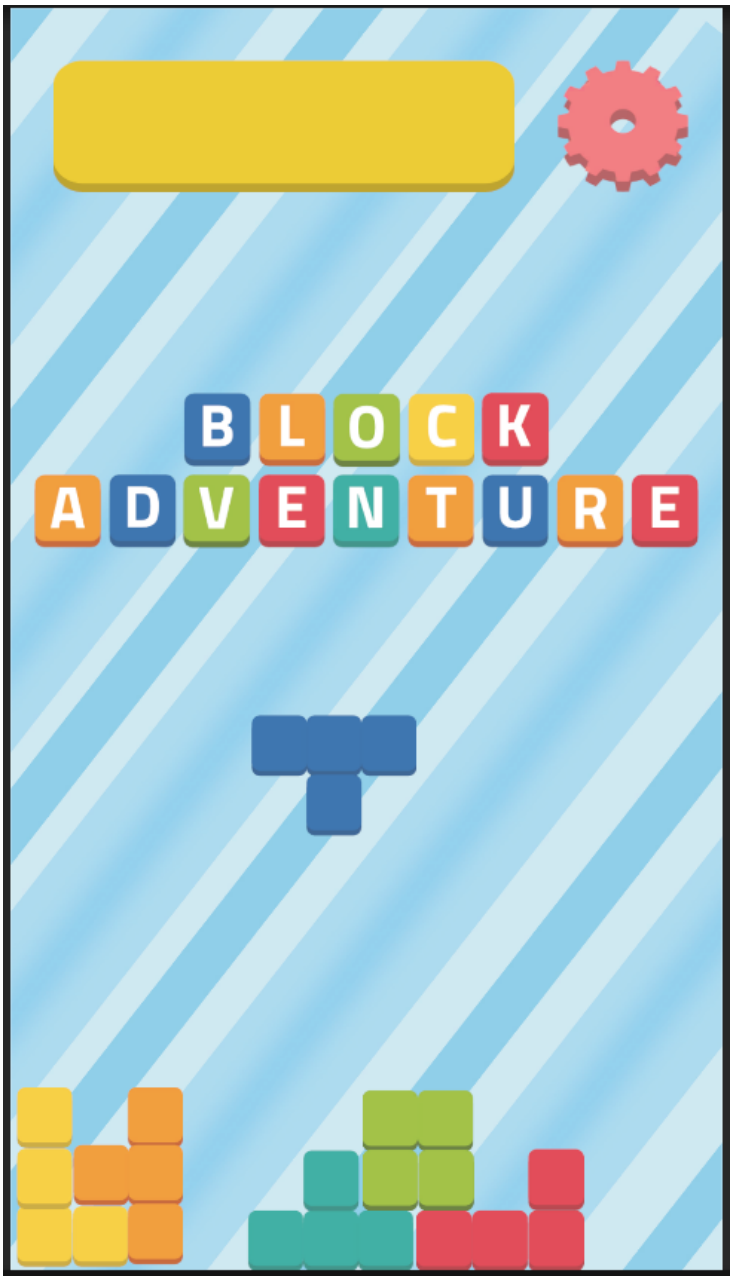

Figure 4.2: Settings Animation State. The settings panel slides into view without loading a new scene, ensuring a seamless experience.

4.2 Core Gameplay Interface (HUD)

The In-Game HUD is divided into three functional zones optimized for one-handed mobile play:

1. **Status Bar (Top):** Displays current score, best score (persisted data), and the Pause/Home buttons.
2. **Play Area (Center):** The 9x9 interactive grid.
3. **Shape Storage (Bottom):** A dynamic layout group that centers the random shapes.

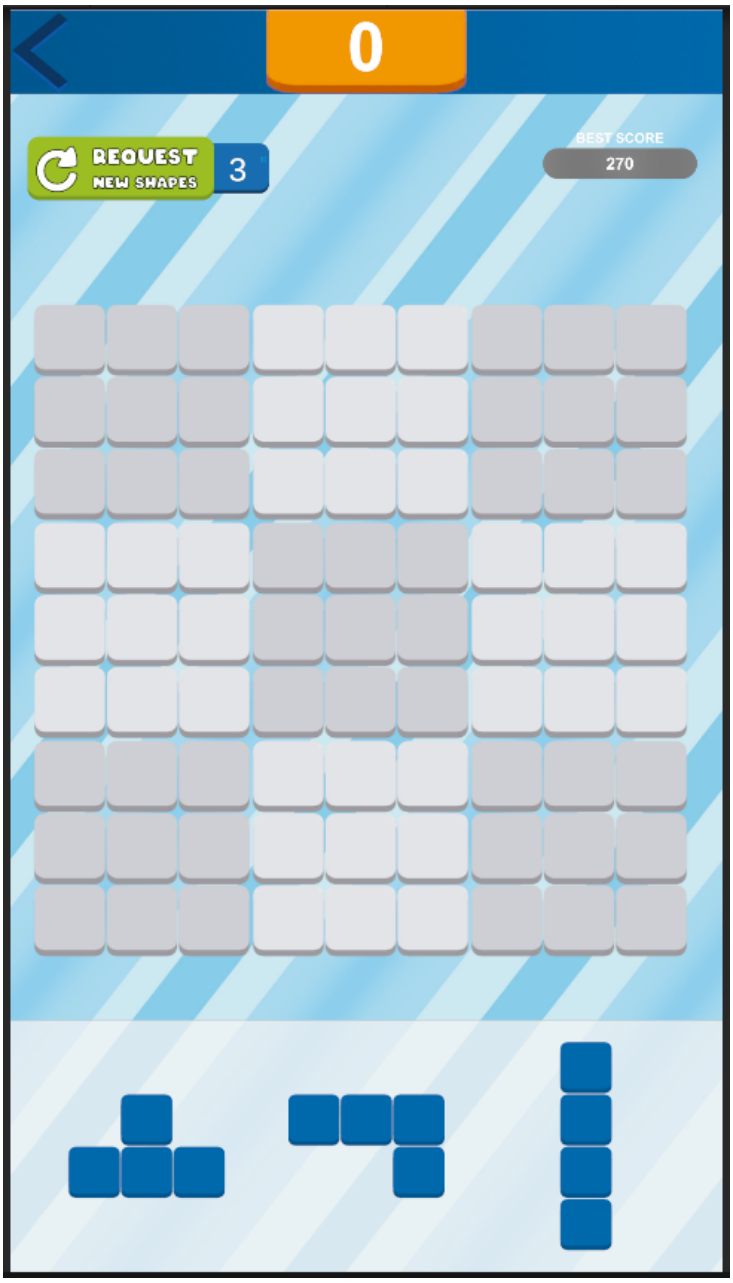

Figure 4.3: Standard Gameplay Layout. The grid acts as the central anchor for all interactions.

4.3 Interaction and Visual Feedback

A critical challenge in mobile puzzle games is the "Fat Finger" problem, where the user's hand obscures the placement area. To solve this, the application implements a **Snap Preview** system.

4.3.1 Snap Preview (Shadow System)

When a player drags a shape over a valid position, a dark "shadow" appears on the grid exactly where the shape will land. This allows players to see the destination even if their finger is blocking the actual shape.

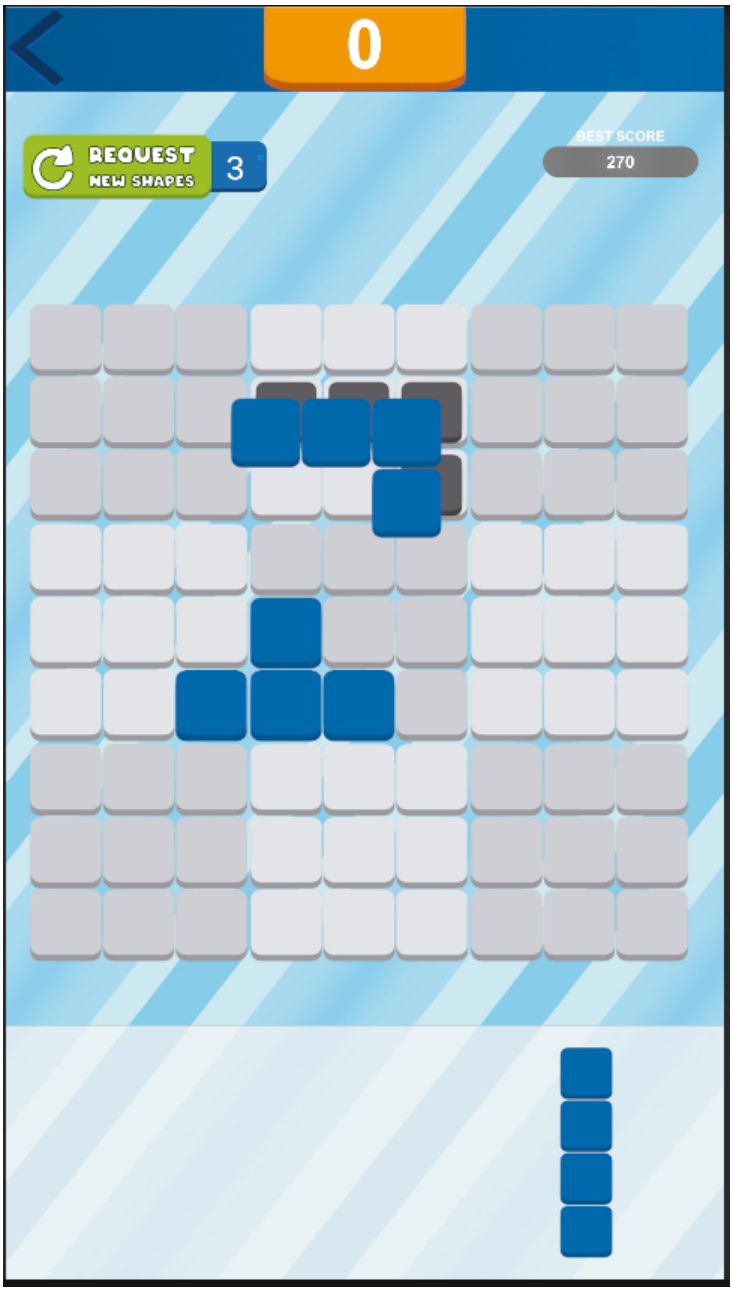

Figure 4.4: Snap Preview System. The dark grey shadow indicates a valid drop position, providing immediate confidence to the player.

4.3.2 Invalid Move Feedback

If a player attempts to drop a shape in an occupied or out-of-bounds area, the shadow does not appear, and the shape snaps back to the storage area upon release.

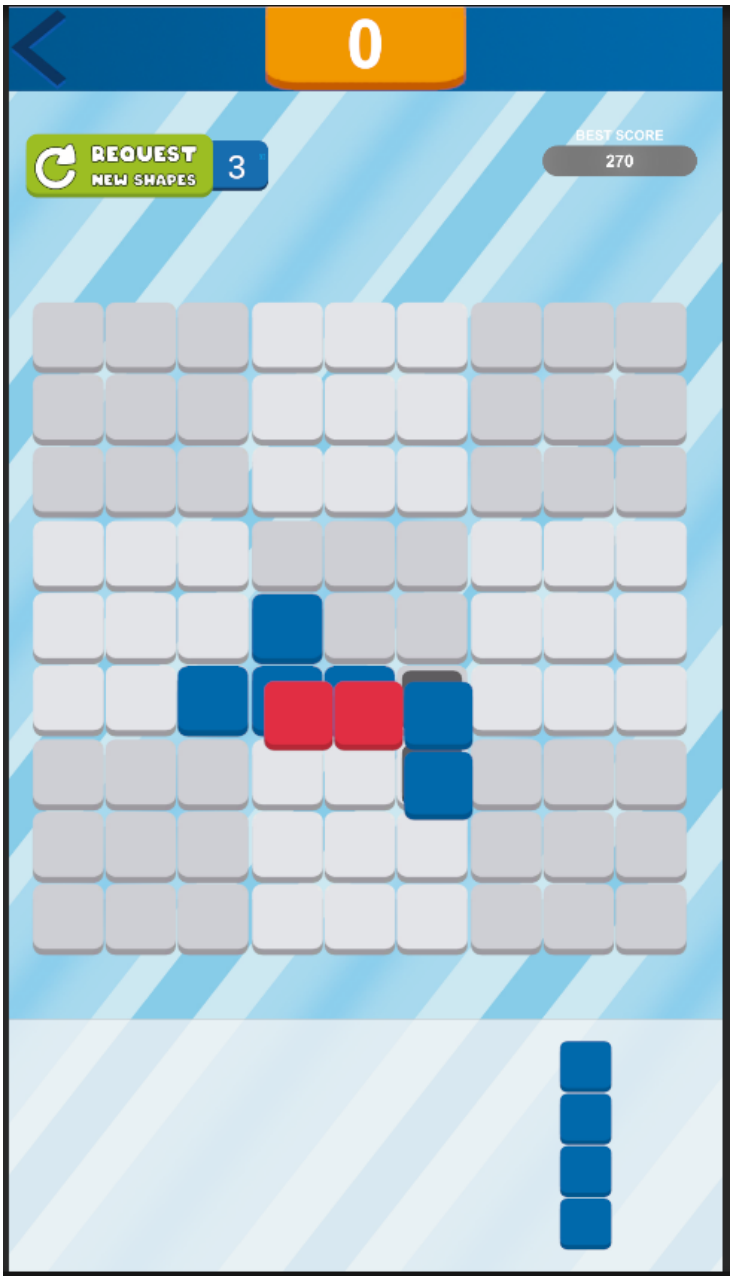

Figure 4.5: Invalid Move State. No shadow appears when hovering over occupied blocks (blue/red), indicating the move is forbidden.

4.4 Progression and Dynamic Difficulty

To prevent visual fatigue and provide a sense of progression, the game environment changes dynamically based on the player's performance.

4.4.1 Threshold Color Shifting

When the player crosses specific score thresholds (e.g., scoring 10 points in the early game), the game triggers a global event that shifts the color palette of the newly spawned shapes. This acts as a subtle "Level Up" indicator without interrupting gameplay.

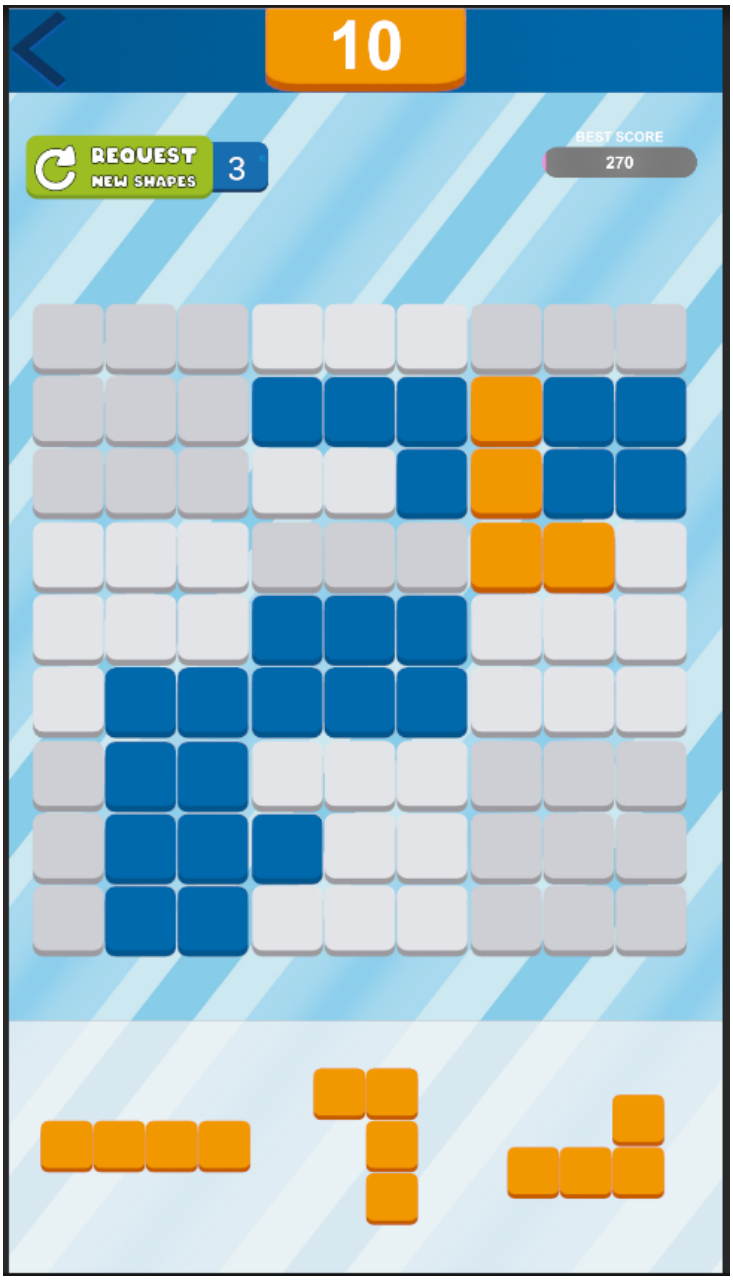

Figure 4.6: Dynamic Color Shift. Upon reaching a score of 10, the spawning algorithm shifts from the default palette to a new Orange/Blue theme.

4.4.2 Combo and Bonus System

The scoring system rewards efficiency. Clearing multiple lines or specific color groups triggers "Bonus Events."

- **Multi-Line Clear:** Triggering a "Superb" popup when clearing 2+ lines simultaneously.
- **Color Bonus:** Clearing a specific set of colored blocks triggers a themed bonus (e.g., "Mint Bonus").

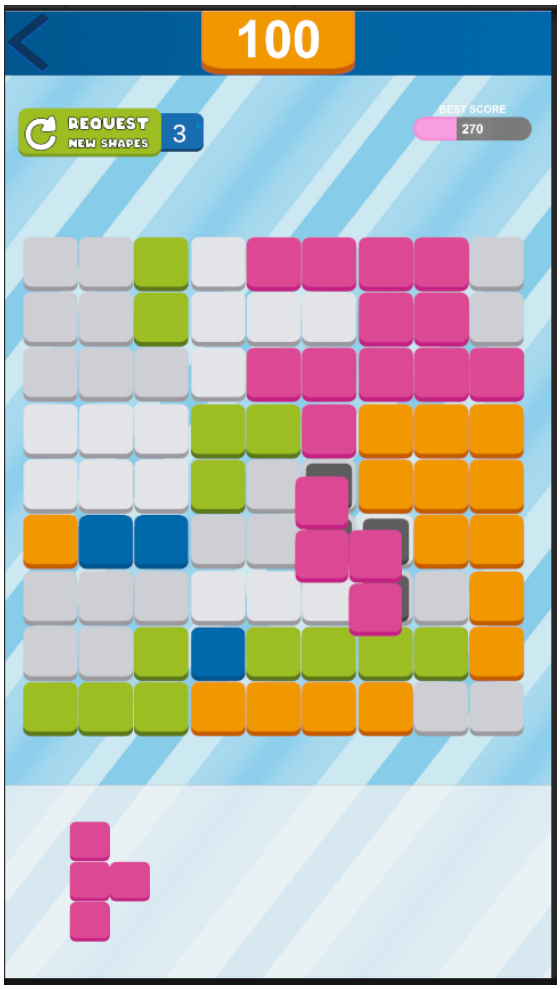

Figure 4.7: Multi-Line "Superb" Clear.

Figure 4.8: Color-Specific Bonus Event.

4.5 Resource Management: The Request System

To mitigate the randomness (RNG) inherent in puzzle games, a "Request System" was implemented. This allows players to refresh their available shapes. This feature is limited to 3 uses per game, introducing a resource management layer.

4.5.1 Active State

The button is interactable and displays the remaining count (e.g., 3).

Figure 4.9: Request Button (Active). The player has 3 requests remaining.

4.5.2 Consumption State

Upon use, the shapes are instantly regenerated, and the counter decrements.

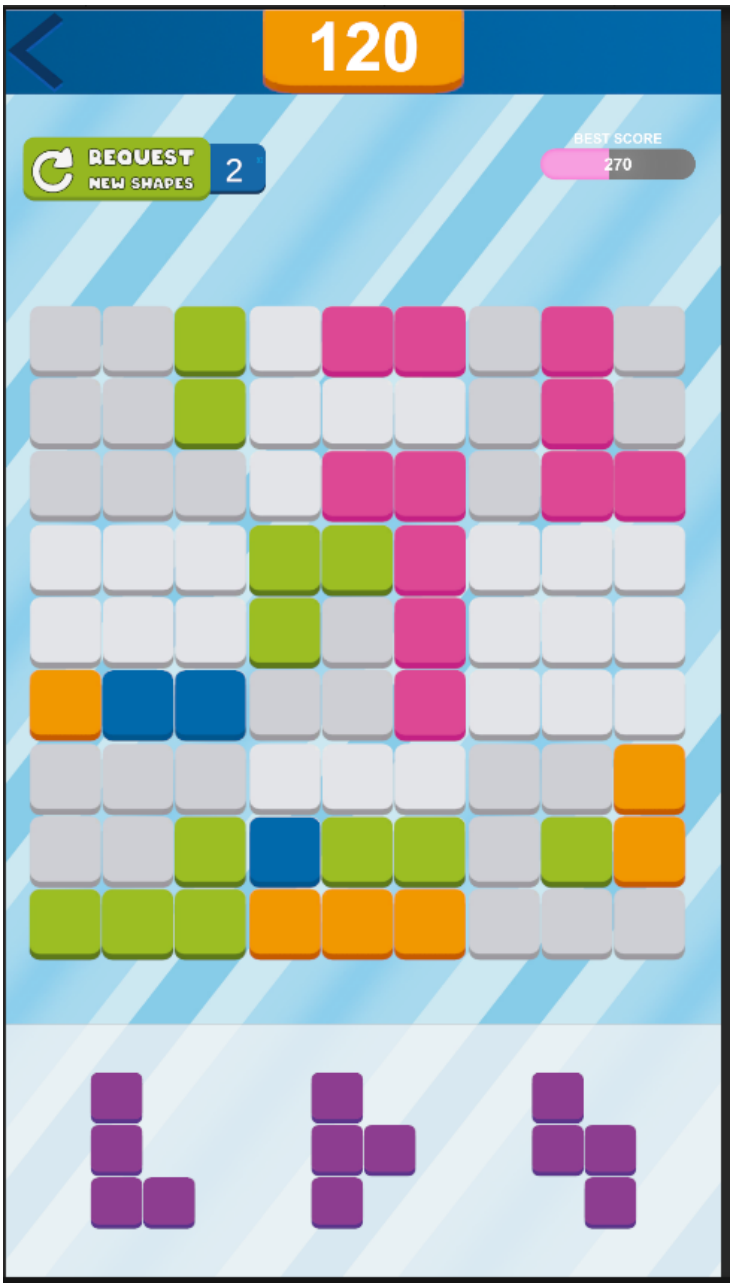

Figure 4.10: Request Button (Used). The counter drops to 2 after a successful refresh.

4.5.3 Depleted State

When the counter reaches 0, the button enters a "Disabled" state (greyed out), preventing further interaction and forcing the player to adapt to the given shapes.

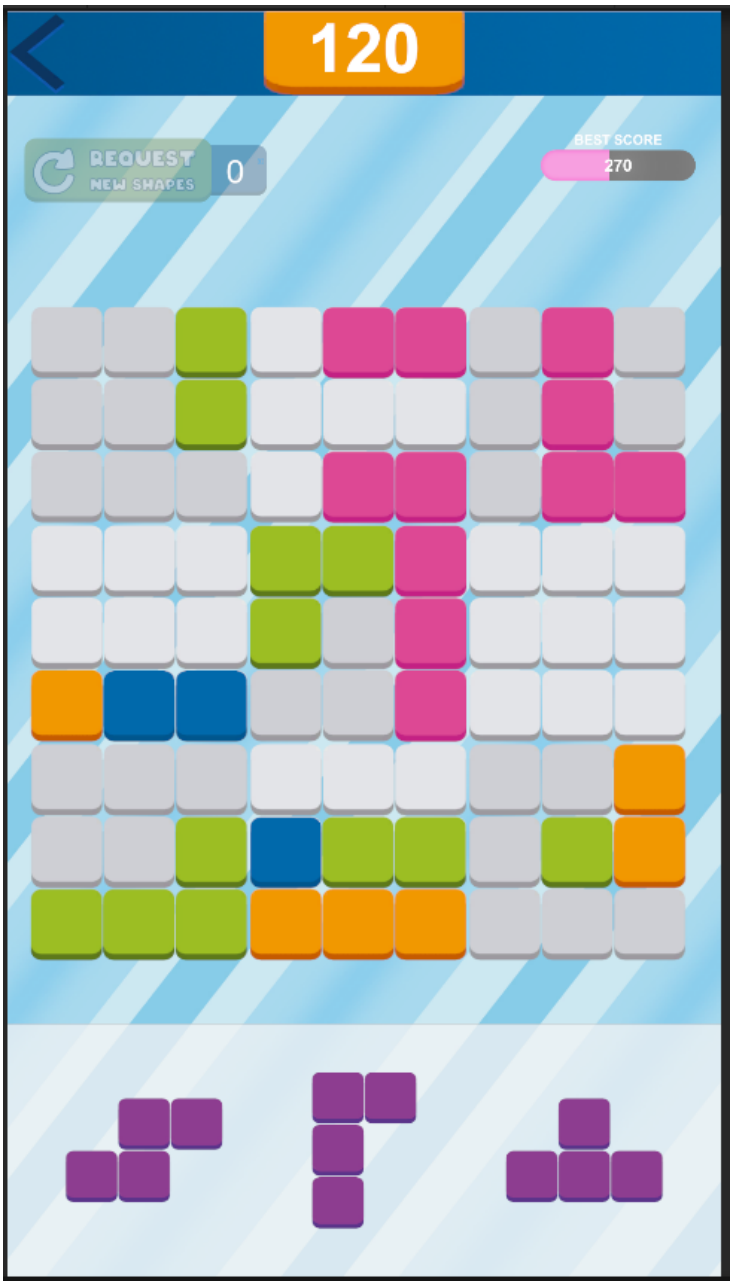

Figure 4.11: Request Button (Depleted). The button becomes non-interactable when resources are exhausted.

Implementation of Complex Functionality

This chapter analyzes the specific software engineering challenges addressed in the project. It focuses on the "Industrialized" aspects of the architecture: Tooling, Algorithmic Optimization, Decoupling, and Data Persistence.

5.1 Tooling and Data-Driven Workflow

A primary goal of this research was to eliminate "Magic Numbers" and hard-coded configurations.

5.1.1 ScriptableObject Architecture

The `ShapeData` class serves as the data container for all block configurations. By inheriting from `ScriptableObject`, these assets exist independently of the scene, allowing for memory-efficient references (Flyweight Pattern).

```
[CreateAssetMenu]
[System.Serializable]
public class ShapeData : ScriptableObject
{
    [System.Serializable]
    public class Row {
        public bool[] column;
        public void ClearRow() {
            for (int i = 0; i < column.Length; i++) column[i] = false;
        }
    }
    public int columns = 0;
    public int rows = 0;
    public Row[] board;

    public void CreateNewBoard() {
        board = new Row[rows];
        for (var i = 0; i < rows; i++) {
            board[i] = new Row(columns);
        }
    }
}
```

Listing 5.1: ShapeData.cs: Data Definition

5.1.2 Custom Inspector Implementation

To facilitate rapid level design, a Custom Editor was implemented using Unity's `Editor` API. This overrides the default inspector to render the `board` array as a visual grid of toggle buttons.

```
[CustomEditor(typeof(ShapeData), false)]
[CanEditMultipleObjects]
[System.Serializable]
public class ShapeDataDrawer : Editor
{
    public override void OnInspectorGUI()
    {
        serializedObject.Update();
        // ... (Drawing input fields)

        if (ShapeDataInstance.board != null && ShapeDataInstance.board.Length > 0)
        {
            DrawBoardTable(); // Renders the visual grid
        }

        serializedObject.ApplyModifiedProperties();
        if (GUI.changed) EditorUtility.SetDirty(ShapeDataInstance);
    }
}
```

Listing 5.2: ShapeDataDrawer.cs (Snippet): Editor GUI Logic

5.2 Algorithmic Optimization: Grid Evaluation

Evaluating a 9x9 grid for multiple win conditions (Rows, Columns, and 3x3 Sub-grids) every frame can be computationally expensive on mobile devices. To optimize this, the project utilizes a **Lookup Table** approach encapsulated in the `LineIndicator` class.

Instead of calculating which blocks belong to "Square 0" at runtime, the associations are pre-defined in static arrays.

```
public class LineIndicator : MonoBehaviour
{
    // Maps Grid Index (0-80) to specific Rows/Columns
    public int[,] line_data = new int[9, 9]
    {
        { 0, 1, 2, 3, 4, 5, 6, 7, 8 },      // Row 0
        { 9, 10, 11, 12, 13, 14, 15, 16, 17 }, // Row 1
        // ... (remaining rows)
    };

    // Maps Grid Index to 3x3 Sub-Squares (Sudoku Logic)
    public int[,] square_data = new int[9, 9]
    {
        { 0, 1, 2, 9, 10, 11, 18, 19, 20 },    // Top-Left Square
        { 3, 4, 5, 12, 13, 14, 21, 22, 23 },   // Top-Middle Square
        // ... (remaining squares)
    };

    public int GetGridSquareIndex(int square)
    {
        // Optimized search to identify which sub-grid a block belongs to
        for (int row = 0; row < 9; row++) {
            for (int col = 0; col < 9; col++) {
                if (square_data[row, col] == square) return row;
            }
        }
```

```
27          return -1;
28      }
29  }
```

Listing 5.3: LineIndicator.cs: Lookup Tables

This approach reduces the complexity of the check from $O(N^2)$ in a naive implementation to a series of $O(1)$ array lookups during the evaluation phase.

5.3 System Architecture: The Observer Pattern

To decouple the Game Logic from the UI, an Event-Driven Architecture was established. The ‘GameEvents‘ class acts as a static bridge.

```
1  public class GameEvents : MonoBehaviour
2  {
3      public static Action<int> UpdateBestScore;
4      public static Action<Config.SquareColor> ShowBonusScreen;
5      // ...
6  }
```

Listing 5.4: GameEvents.cs: Event Definitions

The UI components, such as ‘BestScoreBar‘, act as observers. They subscribe to events when enabled and unsubscribe when disabled, preventing memory leaks. This ensures the UI is reactive rather than polled.

```
1  public class BestScoreBar : MonoBehaviour
2  {
3      public TextMeshProUGUI bestScoreText;
4
5      private void OnEnable() {
6          GameEvents.UpdateBestScore += UpdateBestScore;
7      }
8
9      private void OnDisable() {
10         GameEvents.UpdateBestScore -= UpdateBestScore;
11     }
12
13     private void UpdateBestScore(int bestScore) {
14         bestScoreText.text = bestScore.ToString();
15     }
16 }
```

Listing 5.5: BestScoreBar.cs: The Observer

5.4 Data Persistence: Binary Serialization

For saving player progress (High Scores), the project implements a robust binary serialization system rather than relying on the insecure ‘PlayerPrefs‘. The ‘BinaryDataStream‘ class handles the File I/O operations, ensuring data is written securely to the persistent data path.

```
1  public class BinaryDataStream
2  {
3      public static void Save<T>(T serializableObject, string fileName)
```

```
    {
        string path = Application.persistentDataPath + "/saves/";
        Directory.CreateDirectory(path);

        BinaryFormatter formatter = new BinaryFormatter();
        FileStream fileStream = new FileStream(path + fileName + ".tam", FileMode.Cr

        try {
            formatter.Serialize(fileStream, serializableObject);
        }
        catch (SerializationException e) {
            Debug.LogError("Save Error: " + e.Message);
        }
        finally {
            fileStream.Close();
        }
    }
}
```

Listing 5.6: BinaryDataStream.cs: Generic File Handling

Experimental Results

1. **Grid Logic:** Successfully verified that rows, columns, and 3x3 sub-grids clear correctly.
2. **Persistence:** Confirmed that "Best Score" remains saved even after the application is force-closed and restarted.
3. **Responsiveness:** The "Request New Shapes" button correctly refreshes the hand and updates the UI counter.

Conclusion and Future Work

7.1 Conclusion

This Graduation Research successfully implemented "Block Adventure," a feature-complete puzzle game for Android. Moving beyond simple prototyping, the project demonstrated the value of "Industrialized" Unity development. By employing Design Patterns such as the **Observer Pattern** (Events), **Flyweight Pattern** (ScriptableObjects), and **Singleton/Static Service** (Binary Persistence), the resulting codebase is modular, testable, and scalable.

The development process highlighted the importance of separating Data from Logic. The Custom Editor tools created during this project proved that investing time in "Developer Experience" (DX) significantly speeds up content creation in the long run.

7.2 Future Work

To further enhance the project for a commercial release, future work will focus on:

- **Audio System:** Implementing a dedicated Audio Manager using the existing Event Bus.
- **Hint System:** Developing an A* (A-Star) search algorithm to suggest moves to the player when they are stuck.
- **Cloud Save:** replacing the local BinaryFormatter with a cloud-based solution (e.g., Firebase) for cross-device persistence.