Dream Games
Software Engineering Study - Implementation Documentation

Yusuf Anıl Yazıcı

27 March 2025

1 Introduction

This document serves as a comprehensive record of the implementation process for the Dream Games Software Engineering Study. It details the approach taken to develop a level-based mobile puzzle game in Unity with C#, following the requirements outlined in the case study. The document covers the iterative implementation process, project structure, key components, challenges encountered, and solutions applied.

2 Project Overview

The game consists of:

- A main menu with a level button showing the player's current level
- Multiple levels with different configurations of cubes and obstacles
- Match-3 style gameplay where the player taps groups of adjacent same-colored cubes
- Special items (rockets) created by matching 4+ cubes
- Different obstacles with unique behaviors
- Level progression system with locally persisted progress

3 Implementation Iterations

3.1 Iteration 1: Basic Setup and Scene Navigation ✓

3.1.1 Goals

- Set up project structure
- Create basic MainScene and LevelScene
- Implement scene navigation
- Add level persistence

3.1.2 Implementation Steps

1. Create folder structure in the project (Scripts, Scenes, Prefabs, Materials, etc.)
2. Create MainScene with a simple LevelButton
3. Create a basic LevelScene
4. Implement level persistence using PlayerPrefs
5. Add scene navigation logic

3.1.3 Key Scripts

- `GameManager.cs`: Controls game state and scene transitions

3.2 Iteration 2: Grid System and Basic Cubes ✓

3.2.1 Goals

- Create grid system
- Implement basic colored cubes
- Add simple matching detection
- Enable player interaction

3.2.2 Implementation Steps

1. Build grid generation system based on width/height
2. Create colored cube prefabs
3. Implement basic cube matching logic (2+ adjacent same color)
4. Add tap detection for cubes
5. Add simple cube destruction

3.2.3 Key Scripts

- `GridManager.cs`: Handles grid creation and management
- `Cube.cs`: Basic cube behavior and properties
- `GridCell.cs`: Individual cell in the grid
- `MatchFinder.cs`: Finds groups of matching cubes

3.3 Iteration 3: Cube Falling and Grid Refilling ✓

3.3.1 Goals

- Implement gravity for cubes
- Add grid refilling mechanics
- Create move counter
- Add basic win/lose conditions

3.3.2 Implementation Steps

1. Implement falling logic for cubes
2. Create system to spawn new cubes at the top
3. Add move counter and decrement on valid moves
4. Implement win condition (clear level) and lose condition (out of moves)

3.3.3 Key Scripts

- `FallingController.cs`: Manages cube falling behavior
- `GridFiller.cs`: Fills empty grid spaces with new cubes
- `MoveCounter.cs`: Tracks remaining moves
- `LevelController.cs`: Manages level state and win/lose conditions

3.4 Iteration 3.5: UI Refinement and Visual Improvements ✓

3.4.1 Goals

- Ensure UI responsiveness for different 9:16 screen sizes
- Fix layering issues between game elements and UI

3.4.2 Implementation Steps

1. Implement layer-based rendering system for proper visual hierarchy
2. Configure sorting layers for game elements, overlays and UI elements

3.4.3 Key Scripts and Components

- Sorting Layers configuration in Project Settings
- Canvas rendering and sorting order settings
- Canvas Scaler component settings for UI responsiveness
- Proper Canvas hierarchy and element organization

3.5 Iteration 4: Basic Obstacle Implementation ✓

3.5.1 Goals

- Add one simple obstacle type (Box)
- Implement obstacle clearing mechanics
- Update win condition based on obstacles

3.5.2 Implementation Steps

1. Create box obstacle prefab
2. Implement obstacle damage logic
3. Update win condition to check for remaining obstacles
4. Add basic UI for showing remaining obstacles

3.5.3 Key Scripts

- `Obstacle.cs`: Base class for all obstacles
- `BoxObstacle.cs`: Box obstacle implementation

3.6 Iteration 5: Level Loading from JSON ✓

3.6.1 Goals

- Implement level loading from JSON files
- Make LevelScene adapt to level data
- Set up level progression

3.6.2 Implementation Steps

1. Create level data parser
2. Create level data structure to hold grid configuration
3. Implement level loading logic in LevelController
4. Debug and fix cube position synchronization issues

3.6.3 Key Scripts

- `LevelParser.cs`: Parses level data from JSON files
- `LevelData.cs`: Data structure for level configuration

3.7 Iteration 6: UI Enhancement ✓

3.7.1 Goals

- Add game UI elements
- Implement win/lose popups
- Add basic visual feedback

3.7.2 Implementation Steps

1. Create UI for move counter
2. Add basic win celebration
3. Create fail popup with retry and main menu buttons

3.7.3 Key Scripts

- `UIManager.cs`: Manages all UI elements
- `PopupController.cs`: Controls popup behavior

3.8 Iteration 7: Rocket Implementation ✓

3.8.1 Goals

- Add rocket creation (when matching 4+ cubes)
- Implement basic rocket explosions
- Handle rocket falling

3.8.2 Implementation Steps

1. Create rocket prefabs (horizontal and vertical)
2. Implement rocket creation logic
3. Add rocket explosion mechanics
4. Handle rocket interaction with obstacles

3.8.3 Key Scripts

- `Rocket.cs`: Base class for rockets
- `RocketCreator.cs`: Logic for creating rockets from matches

3.9 Iteration 8: Advanced Obstacles and Mechanics ✓

3.9.1 Goals

- Add remaining obstacle types (Stone, Vase)
- Implement specific damage rules for each
- Add falling mechanics for Vase obstacles

3.9.2 Implementation Steps

1. Create Stone and Vase prefabs
2. Implement specific damage behaviors
3. Handle obstacle falling logic for Vase
4. Update obstacle clearing detection

3.9.3 Key Scripts

- `StoneObstacle.cs`: Stone obstacle implementation
- `VaseObstacle.cs`: Vase obstacle implementation

3.10 Iteration 9: Animations ✓

3.10.1 Goals

- Implement basic animations
- Add particle effects
- Polish UI elements

3.10.2 Implementation Steps

1. Add animation for cube matching/explosion
2. Implement falling animations
3. Add obstacle clearing effects
4. Enhance UI transitions
5. Create animation sequencing system
6. Implement particle effect pooling

3.10.3 Key Scripts

- `AnimationManager.cs`: Central manager for all animations
- `ParticleManager.cs`: Manages particle effect pools and spawning
- `ParticlePool.cs`: Object pooling system for particle effects
- `GeneralAnimations.cs`: Handles falling and filling animations
- `CubeAnimations.cs`: Handles cube-specific animations
- `RocketAnimations.cs`: Manages rocket creation and explosion animations
- `ObstacleAnimations.cs`: Controls obstacle damage and destruction animations
- `PopupAnimations.cs`: Handles UI animations for popups and buttons

3.11 Iteration 10: Final Touches ✓

3.11.1 Goals

- Add rocket-rocket combinations
- Fine-tune gameplay
- Test and optimize

3.11.2 Implementation Steps

1. Implement rocket combination mechanics
2. Adjust difficulty and balance
3. Optimize performance
4. Complete testing across all levels
5. Optimize animation timing issues

4 Project Structure and Key Components

4.1 Architecture Overview

The project follows a modular architecture with clear separation of concerns. At its core, the system is built around a grid-based gameplay environment where different types of grid items (cubes, obstacles, and rockets) interact according to specific rules. The architecture emphasizes object-oriented principles including inheritance, encapsulation, and polymorphism.

4.2 Core Components and Class Hierarchy

- **Game Management**
  - `GameManager`: Singleton that manages game state, scene transitions, and level persistence. It handles saving/loading the current level using PlayerPrefs and provides methods for navigating between scenes.
  - `LevelController`: Orchestrates gameplay flow, including win/loss conditions, obstacle tracking, and move management. It loads level data, initializes the grid, and coordinates between different systems during gameplay. It also monitors remaining obstacles to determine when a level is completed.
  - `GridManager`: Creates and maintains the game grid based on level specifications. It provides methods for accessing cells, converting between grid and world coordinates, and facilitates interaction between grid items.
  - `MoveCounter`: Tracks remaining moves and determines when the player runs out of moves, triggering game over conditions.
- **Grid Items Hierarchy**
  - `GridItem` (abstract): Base class for all grid-based objects with common properties like grid position and interaction methods.
    - `Cube`: Colored cubes that players match. They have specific colors and can be matched in groups of 2+ adjacent same-colored cubes. When matched in groups of 4+, they can create rockets.
    - `Rocket`: Special items created from matching 4+ cubes. They can be horizontal or vertical, clearing entire rows or columns when activated. Rockets can also combine with other rockets to create special explosion patterns.
    - `Obstacle` (abstract): Base class for destructible level objectives with health/damage system.
      - `BoxObstacle`: Takes damage from adjacent matches or rockets. Requires one damage point to be destroyed.
      - `StoneObstacle`: Takes damage only from rockets. Immune to normal match damage. Requires one damage point to be destroyed.
      - `VaseObstacle`: Takes two damage points to be destroyed and can fall like cubes. Only takes one damage from each match group.
- **Gameplay Systems**
  - `MatchFinder`: Detects groups of matching cubes when the player taps on them. It handles match validation, rocket creation conditions, and triggers match processing.
  - `FallingController`: Manages the gravity system that makes cubes and vase obstacles fall into empty spaces below them. It calculates falling paths and triggers appropriate animations.
  - `GridFiller`: Fills empty spaces at the top of the grid with new randomly colored cubes. It works in conjunction with the FallingController to ensure the grid is always filled.

  - `MatchGroup`: Represents a group of matched cubes with the same color. It stores information about the match size, color, and handles the destruction process.
  - `RocketCreator`: Utility class that handles the creation of rockets when players match 4+ cubes. It determines rocket direction and initializes the rocket properly.

- **Animation System**
  - `AnimationManager`: Singleton that provides a unified interface for all game animations. It centralizes animation configuration and ensures consistent visual feedback throughout the game.
  - Animation Modules:
    * `GeneralAnimations`: Handles common animations like falling and spawning.
    * `CubeAnimations`: Handles cube-specific animations (destruction, combine, etc.).
    * `RocketAnimations`: Manages rocket animations (creation, explosion, projectiles).
    * `ObstacleAnimations`: Controls obstacle animations (damage, destruction).
    * `PopupAnimations`: Handles UI transitions and effects.
  - `ParticleManager`: Manages particle effect pools and handles particle instantiation, using object pooling for performance optimization.
  - `ParticlePool`: Implements object pooling for particle effects to reduce garbage collection and improve performance.

- **UI System**
  - `PopupController`: Manages game popups including win, lose, and level completion screens.
  - `ObstacleCounterUI`: Displays the count of remaining obstacles in the level.

- **Level Management**
  - `LevelParser`: Parses level data from JSON files stored in Resources.
  - `LevelData`: Data structure that holds level configuration, including grid dimensions, move count, and initial grid layout.

4.3 Folder Structure

The project follows a logical folder structure that organizes scripts by their functionality. The main scripts structure is shown below:

```
Assets/
 Scenes/
    MainScene.unity
```

```
   LevelScene.unity
Scripts/
   Animations/
      Core/
         AnimationManager.cs
      Effects/
         ParticleManager.cs
         ParticlePool.cs
      Types/
          CubeAnimations.cs
          GeneralAnimations.cs
          ObstacleAnimations.cs
          PopupAnimations.cs
          RocketAnimations.cs
   Core/
      GameManager.cs
      GridManager.cs
      LevelController.cs
      MoveCounter.cs
   Grid/
      FallingController.cs
      GridCell.cs
      GridFiller.cs
      GridMask.cs
      MatchFinder.cs
      MatchGroup.cs
      RocketCreator.cs
   Items/
      Cube.cs
      GridItem.cs
      Rocket.cs
      Obstacles/
          Obstacle.cs
          BoxObstacle.cs
          StoneObstacle.cs
          VaseObstacle.cs
   UI/
      Header.cs
      ObstacleCounter.cs
      PopupController.cs
   Utils/
       LevelData.cs
       LevelParser.cs
Prefabs/
   Obstacles/
   Particles/
Materials/
Resources/
```

```
Levels/
Prefabs/
```

5 Challenges Encountered

While implementing this project, several challenges were encountered:

1. **Unity Learning Curve**: As I had limited prior experience with Unity, adapting to the engine's component-based architecture and lifecycle methods required significant learning.
2. **Animation System**: Implementing a robust animation system that could handle various game elements while maintaining performance was particularly challenging. The solution involved creating a centralized AnimationManager with specialized animation modules.
3. **Grid Coordination**: Ensuring proper synchronization between grid data and visual representations was difficult, especially when handling falling mechanics and refilling.
4. **UI Integration**: Integrating responsive UI elements that work across different screen sizes required careful canvas setup and proper event handling.
5. **Performance Optimization**: Managing particle effects and animations efficiently to maintain smooth gameplay required implementing object pooling and optimizing animation sequences.

6 Potential Improvements

While the current implementation meets all the requirements outlined in the case study, several potential improvements could enhance the game further:

1. **Animation Optimization**: The current animation system could be further optimized to reduce performance overhead, especially during complex sequences like rocket explosions or large matches. Implementing a more efficient animation pooling system and reducing unnecessary calculations during tweening would improve frame rates on lower-end devices.
2. **Screen Orientation and Responsiveness**: Currently, the game only designed to support a 9:16 portrait orientation. Making sure to have a responsive design that supports various aspect ratios and orientations would improve accessibility and user experience.
3. **Advanced Visual Effects**: Adding more sophisticated visual effects such as dynamic lighting, screen shake on large matches, and color blending could enhance the game's visual appeal and increase player engagement.
4. **Audio System**: Implementing a comprehensive audio system with sound effects for matches, obstacles breaking, and ambient music would significantly improve the overall game experience.

7 Conclusion

This implementation of the Dream Games Software Engineering Study demonstrates a comprehensive understanding of object-oriented programming principles and their application in game development. The project structure emphasizes modularity, extensibility, and separation of concerns, making future additions and changes straightforward.

The implementation successfully meets all the requirements outlined in the case study, providing a playable level-based puzzle game with match mechanics, special items, obstacles, and appropriate visual feedback through animations and effects.

Through this project, I've gained valuable experience in game development with Unity, animation systems, UI design, and performance optimization techniques. The challenges encountered provided opportunities for growth and problem-solving that will prove beneficial in future projects.

The potential improvements outlined in the previous section provide a roadmap for further enhancing the game, indicating an understanding of both the current implementation's strengths and opportunities for advancement.