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.

The project involves creating a game where players match colored cubes to clear obstacles and progress through levels. The implementation follows object-oriented programming principles, with a focus on modularity, extensibility, and proper visual feedback through animations and effects.

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
- Implement proper overlay for panels
- Refine visual hierarchy of UI elements
- Fix layering issues between game elements and UI

3.4.2 Implementation Steps

1. Implement layer-based rendering system for proper visual hierarchy
2. Create overlay system that darkens game elements but not UI panels
3. Configure sorting layers for game elements, overlays and UI elements
4. Ensure UI elements scale and position correctly on different aspect ratios

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

4 Project Structure

4.1 Key Classes and Inheritance Hierarchy

- **Core Management Classes**
  - `GameManager`: Singleton that manages game state, scene transitions, and level persistence
  - `LevelController`: Manages level state, win/loss conditions, and gameplay progression
  - `GridManager`: Handles grid creation, cell access, and coordinate transformations
  - `AnimationManager`: Central manager for all animations with unified interface
  - `ParticleManager`: Manages particle effect pools and instantiation

- **Item Hierarchy**
  - `GridItem` (abstract): Base class for all grid-based objects
    - `Cube`: Colored cubes that players match
    - `Rocket`: Special items created from matching 4+ cubes
    - `Obstacle` (abstract): Base class for destructible level objectives
      - `BoxObstacle`: Takes damage from adjacent matches and rockets
      - `StoneObstacle`: Takes damage only from rockets
      - `VaseObstacle`: Takes 2 damage points, can fall with gravity
- **Animation System**
  - `AnimationManager`: Singleton that provides centralized animation control
  - Animation Modules:
    - `CubeAnimations`: Handles cube-specific animations (destruction, combine, etc.)
    - `RocketAnimations`: Manages rocket animations (creation, explosion, etc.)
    - `ObstacleAnimations`: Controls obstacle animations (damage, destruction)
    - `PopupAnimations`: Handles UI transitions and effects

4.2 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
      MatchFinder.cs
      MatchGroup.cs
      RocketCreator.cs
   Items/
      Cube.cs
      GridItem.cs
      Rocket.cs
      Obstacles/
          Obstacle.cs
          BoxObstacle.cs
          StoneObstacle.cs
          VaseObstacle.cs
   UI/
      ObstacleCounter.cs
      PopupController.cs
   Utils/
       LevelData.cs
       LevelParser.cs
Prefabs/
   Obstacles/
   Particles/
Materials/
Resources/
   Levels/
   Prefabs/
CaseStudyAssets2025/
```

5 Key Components

5.1 GameManager

The central controller that manages game state, scene transitions, and level persistence. It handles level loading/saving and ensures all systems communicate properly.

5.2 GridManager

Responsible for creating and maintaining the game grid. It provides methods for accessing cells, converting between grid and world coordinates, and handling grid initialization.

5.3 LevelController

Manages the gameplay flow for each level, including win/lose conditions, obstacle tracking, and player move management. It coordinates with other systems to ensure proper game state transitions.

5.4 AnimationManager

A singleton that provides a unified interface for all game animations. It centralizes animation configuration and ensures consistent visual feedback throughout the game.

5.5 MatchFinder

Detects groups of matching cubes when the player taps on them. It handles match validation, rocket creation conditions, and triggers match processing.

5.6 Rocket Implementation

Rockets are created when players match 4+ cubes. They can be horizontal or vertical and clear entire rows or columns when activated. Rocket-rocket combinations create special explosion patterns.

5.7 Obstacle System

The game features three obstacle types with unique behaviors:

- Box: Takes damage from adjacent matches or rockets
- Stone: Takes damage only from rockets
- Vase: Takes two damage points and can fall like cubes

6 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.

7 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. **Popup Celebration Particles**: The win celebration could be enhanced with more elaborate particle systems to create a more rewarding visual experience. Adding confetti, fireworks, or star burst effects when completing levels would increase player satisfaction.

3. **Scene Transition Animations**: Implementing smooth transitions between the main menu and level scenes would improve the overall game flow and provide a more polished feel. Cross-fades, wipes, or custom transition effects could be added to create a seamless experience.

4. **Grid Masking for Spawned Cubes**: Adding a mask to the grid area would prevent newly spawned cubes from being visible outside the grid boundaries. This would create a cleaner visual experience by hiding cubes until they enter the play area properly.

5. **Screen Orientation and Responsiveness**: Currently, the game only supports a 9:16 portrait orientation, which limits its compatibility across different devices and screen sizes. Implementing a responsive design that supports various aspect ratios and orientations would improve accessibility and user experience.

6. **Level Data Prefetching**: The current implementation loads level data when needed, which could cause minor delays when transitioning between levels. Implementing a prefetching system that loads the next level's data in the background would create faster level transitions and improve player experience.

7. **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.

8. **Audio System**: Implementing a comprehensive audio system with sound effects for matches, obstacles breaking, and ambient music would significantly improve the overall game experience.

8 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.