LevelMap Module Documentation

Overview

The LevelMap Module is a ready-to-use Unity solution that provides a fully functional level map system for mobile and desktop games. The module is implemented using the MVC (Model-View-Controller) architectural pattern and ensures separation of game logic from presentation.

Features

- **Level Progression**: Automatic unlocking of the next level upon completion of the current one
- **Star Rating System**: Track level completion ratings (1-3 stars)
- **Progress Saving**: Automatic state saving to PlayerPrefs
- **Customizable UI**: Flexible appearance configuration through ScriptableObject
- **Event System**: Event system for integration with other game modules
- **Responsive Interface**: Support for various screen resolutions

Architecture

MVC Pattern

The module follows the MVC architecture:

- **Model** (`LevelMapModel`): Manages data and business logic
- **View** (`LevelMapView`): Handles UI display
- **Controller** (`LevelMapController`): Coordinates interaction between Model and View

Core Components

1. Data

LevelData - represents information about an individual level:

```csharp
public class LevelData
{
    public int levelId;           // Unique level ID
    public bool isUnlocked;       // Whether the level is unlocked
    public bool isCompleted;      // Whether the level is completed
    public int stars;             // Number of stars (0-3)
    public Vector2 mapPosition;   // Position on the map
    public string levelName;      // Level name
    public Sprite levelIcon;      // Level icon
}
```

LevelMapData - contains all level map information:

```csharp
public class LevelMapData
{
    public List<LevelData> levels;        // List of all levels
    public int currentLevel;              // Currently selected level
    public int maxUnlockedLevel;          // Maximum unlocked level
}
```

2. Interfaces

ILevelMapController - main interface for working with the module:

```csharp
public interface ILevelMapController
{
    void Initialize();                              // Initialize the module
    void ShowLevelMap();                           // Show level map
    void HideLevelMap();                           // Hide level map
    void CompleteLevel(int levelId, int stars);    // Complete level
    void UnlockLevel(int levelId);                 // Unlock level
    LevelData GetLevel(int levelId);               // Get level data
}
```

3. Events

The module provides the following events:

In Controller:

- `OnLevelSelected` - Triggered when a level is selected

- `OnBackPressed` - Triggered when the back button is pressed

In Model:

- `OnLevelUnlocked` - Triggered when a level is unlocked
- `OnLevelCompleted` - Triggered when a level is completed
- `OnCurrentLevelChanged` - Triggered when the current level changes

Installation and Setup

1. Import Module

1. Copy all module files to `Assets/Scripts/LevelMapModule/`
2. Ensure all scripts are in the `LevelMapModule` namespace

2. UI Setup

Creating Level Button Prefab

1. Create a GameObject with a `Button` component
2. Add child objects:
   - `Text` - for displaying level number
   - `Image` - for level icon
   - `Stars Container` - container with stars (1-3 objects)
   - `Lock Icon` - lock icon
   - `Completed Icon` - completion icon
3. Add `LevelButton` component and configure references to UI elements

Creating Main UI

1. Create a Canvas for the level map
2. Add a container for level buttons (e.g., with `GridLayoutGroup`)
3. Add a "Back" button
4. Add `LevelMapView` component to the main level map GameObject

3. Configure LevelMapViewSettings

Create and configure `LevelMapViewSettings`:

csharp

```csharp
[SerializeField] private LevelMapViewSettings settings = new LevelMapViewSettings
{
    levelButtonPrefab = prefabReference,      // Reference to button prefab
    levelButtonsContainer = containerTransform, // Container for buttons
    backButton = backButtonReference,         // Back button
    mapCanvas = canvasReference,              // Level map canvas
    buttonSpacing = 100f,                     // Spacing between buttons
    buttonsPerRow = 5                         // Number of buttons per row
};
```

Usage

Module Initialization

csharp

```csharp
// Create instances
var model = new LevelMapModel();
var view = FindObjectOfType<LevelMapView>();
var controller = new LevelMapController(model, view);

// Subscribe to events
controller.OnLevelSelected += OnLevelSelected;
controller.OnBackPressed += OnBackPressed;

// Initialize
controller.Initialize();
```

Basic Operations

Show/Hide Level Map

csharp

```csharp
controller.ShowLevelMap();  // Show
controller.HideLevelMap();  // Hide
```

Level Completion

```csharp
// Complete level with 3 stars
controller.CompleteLevel(levelId: 1, stars: 3);

// The next level is automatically unlocked upon completion
```

Level Unlocking

```csharp
// Unlock a specific level
controller.UnlockLevel(levelId: 5);
```

Getting Level Information

```csharp
LevelData levelData = controller.GetLevel(levelId: 1);
if (levelData != null)
{
    Debug.Log($"Level {levelData.levelId}: Unlocked={levelData.isUnlocked}, Stars={levelData.st
}
```

Event Handling

```csharp
private void OnLevelSelected(int levelId)
{
    Debug.Log($"Player selected level {levelId}");
    // Load selected level
    SceneManager.LoadScene($"Level_{levelId}");
}

private void OnBackPressed()
{
    Debug.Log("Back button pressed");
    // Return to main menu
    SceneManager.LoadScene("MainMenu");
}
```

Extending Functionality

Custom Level Data

You can extend `LevelData` with additional fields:

```csharp
[Serializable]
public class ExtendedLevelData : LevelData
{
    public string description;
    public float bestTime;
    public int coinsCollected;

    public ExtendedLevelData(int id) : base(id) { }
}
```

Custom Unlock Conditions

Override logic in `LevelMapModel`:

```csharp
public class CustomLevelMapModel : LevelMapModel
{
    public override void CompleteLevel(int levelId, int stars = 0)
    {
        base.CompleteLevel(levelId, stars);

        // Custom unlock logic
        if (stars >= 2 && levelId % 5 == 0) // Every 5th level requires minimum 2 stars
        {
            UnlockLevel(levelId + 1);
        }
    }
}
```

Animations and Effects

Add animations to `LevelMapView`:

csharp

```csharp
public void UpdateLevel(LevelData levelData)
{
    if (levelButtons.TryGetValue(levelData.levelId, out var button))
    {
        button.UpdateData(levelData);

        // Add unlock animation
        if (levelData.isUnlocked)
        {
            PlayUnlockAnimation(button);
        }
    }
}
```

Best Practices

1. Memory Management

Always call `Dispose()` when destroying the controller:

csharp

```csharp
private void OnDestroy()
{
    controller?.Dispose();
}
```

2. Error Handling

The module includes basic error handling for saving/loading data. For production code, it's recommended to add additional checks.

3. Performance

- Use object pools for level buttons when dealing with large numbers of levels (>100)
- Consider lazy loading of UI elements to optimize loading times

4. Testing

Create test scenarios to verify:

- Correct saving/loading of progress
- Proper event handling
- Edge case handling (non-existent levels, corrupted data)

Integration Examples

Simple Integration

```csharp
public class GameManager : MonoBehaviour
{
    [SerializeField] private LevelMapView levelMapView;
    private ILevelMapController levelMapController;

    private void Start()
    {
        var model = new LevelMapModel();
        levelMapController = new LevelMapController(model, levelMapView);

        levelMapController.OnLevelSelected += LoadLevel;
        levelMapController.OnBackPressed += ReturnToMainMenu;

        levelMapController.Initialize();
        levelMapController.ShowLevelMap();
    }

    private void LoadLevel(int levelId)
    {
        SceneManager.LoadScene($"Level_{levelId}");
    }

    private void ReturnToMainMenu()
    {
        SceneManager.LoadScene("MainMenu");
    }

    private void OnDestroy()
    {
        levelMapController?.Dispose();
    }
}
```

Advanced Integration with Dependency Injection

csharp

```csharp
public class LevelMapInstaller : MonoBehaviour, ILevelMapInstaller
{
    [SerializeField] private LevelMapView view;

    public ILevelMapController Install()
    {
        var model = new LevelMapModel();
        var controller = new LevelMapController(model, view);

        controller.Initialize();
        return controller;
    }
}
```

Troubleshooting

Common Issues

1. **Buttons not responding**: Check that `LevelButton` components are properly configured with button references
2. **Progress not saving**: Ensure PlayerPrefs write permissions are available
3. **UI layout issues**: Verify `LevelMapViewSettings` spacing and container setup
4. **Events not firing**: Check event subscriptions and ensure proper initialization order

Debug Tips

Enable debug logging by adding this to your controller:

csharp

```csharp
private void HandleLevelButtonClicked(int levelId)
{
    Debug.Log($"Level button clicked: {levelId}");
    model.SetCurrentLevel(levelId);
    OnLevelSelected?.Invoke(levelId);
}
```

API Reference

LevelMapController Methods

Method	Description	Parameters
`Initialize()`	Initializes the level map system	None
`ShowLevelMap()`	Makes the level map visible	None
`HideLevelMap()`	Hides the level map	None
`CompleteLevel(int, int)`	Marks a level as completed	`levelId`, `stars` (0-3)
`UnlockLevel(int)`	Unlocks a specific level	`levelId`
`GetLevel(int)`	Retrieves level data	`levelId`

Events

Event	Parameters	Description
`OnLevelSelected`	`int levelId`	Fired when player selects a level
`OnBackPressed`	None	Fired when back button is pressed

Conclusion

The LevelMap Module provides a flexible and extensible solution for creating level maps in Unity games. The module follows SOLID principles and allows easy customization of both appearance and game logic to meet specific project requirements. Its MVC architecture ensures maintainable code and clear separation of concerns, making it suitable for projects of any scale.