Block-O-Mania Game Documentation

Table of Contents

Introduction

Block-O-Mania is an 2D board block puzzle game package that provides several modes such as Level, Classic, Time, Bomb, and Advanced. Depending on the mode, the game will present various objectives to complete, like reaching a certain score or clearing specific goals and blocks from the board.

Game Scenes

The game features two primary scenes:

- **Limited Moves**: In this scene, the player has a set number of moves to accomplish all the goals displayed on a dedicated panel called the goal panel/target panel. The player needs to complete the game within these moves.
- **Unlimited Moves**: In this scene, the player can take an unlimited number of moves to achieve the listed goals.

Android Build Creation

The game developer can select one of these scenes from GamingMonks > Block_Puzzle > Scenes, depending on the game's requirements and design. This division is beneficial as it allows for:

- **Flexibility**: Offers flexible workflow for developers.
- **Catering to different player skills**: Some players may prefer the challenge of limited moves, while others may enjoy a more relaxed unlimited moves mode.
- **Creating a variety of gameplay experiences**: This variation in gameplay can keep players engaged and interested in the game.

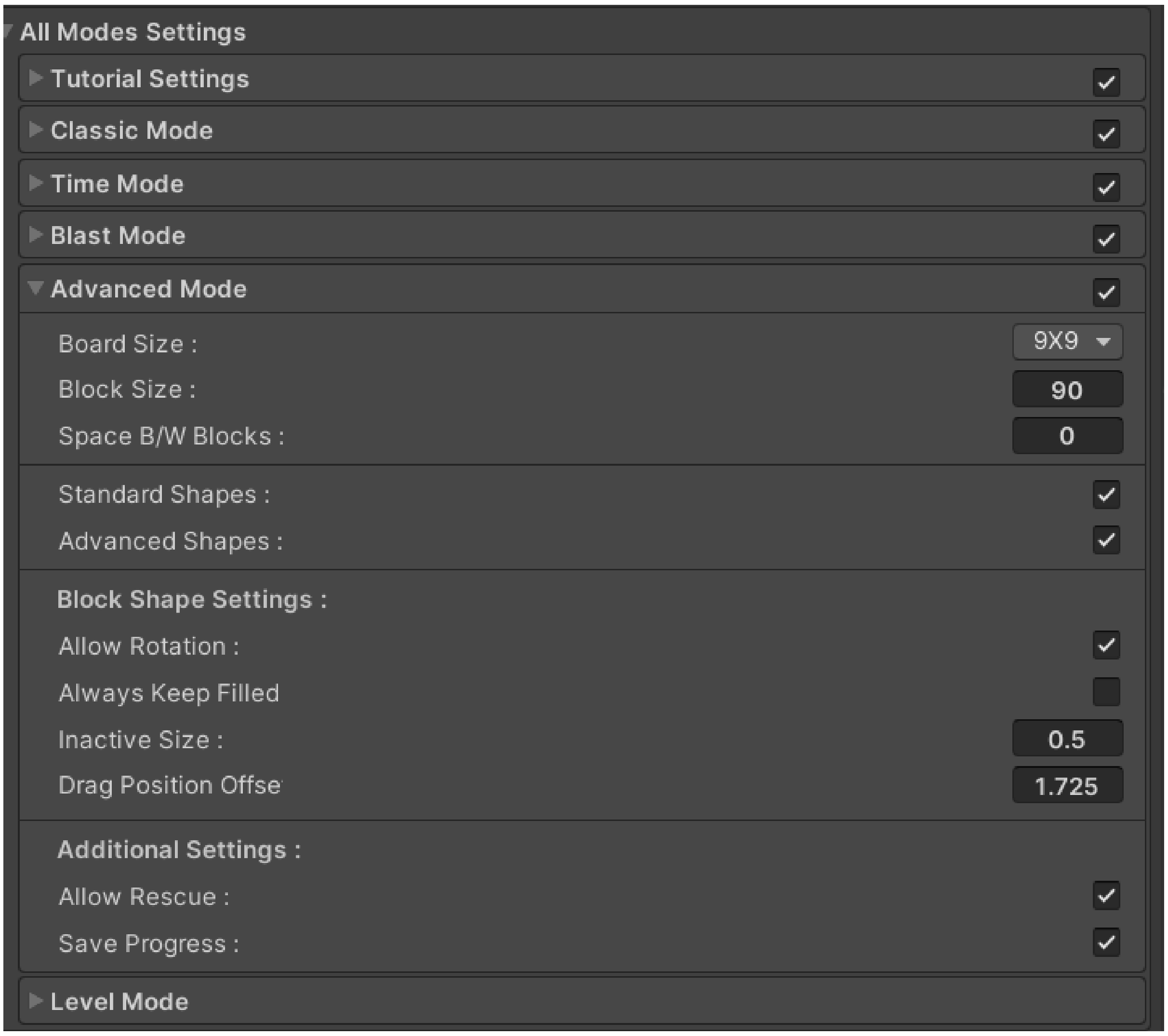
To create an android apk one must simply select either the unlimited moves or limited moves scenes from GamingMonks > Block_Puzzle > Scenes. Note that selecting both scenes might increase the final build size.

Gameplay Settings

This scriptable object defines the structure of all modes. Open the GameplaySettings from the GamingMonks > GameplaySettings within the unity editor

Other Modes Settings

These are configurations for tutorial, classic, bomb, timed and advanced modes.


All Modes Settings

Tutorial Settings ✓

Classic Mode ✓

Time Mode ✓

Blast Mode ✓

Advanced Mode ✓

Board Size : 9X9

Block Size : 90

Space B/W Blocks : 0

Standard Shapes : ✓

Advanced Shapes : ✓

Block Shape Settings :

Allow Rotation : ✓

Always Keep Filled

Inactive Size : 0.5

Drag Position Offse 1.725

Additional Settings :

Allow Rescue : ✓

Save Progress : ✓

Level Mode

- **Board Size** : Size of the board grid.
  Block Size : Size of each cell in the board grid.
- **Block Space** : Spacing between the blocks on the board grid.
- **Standard Shapes** : Boolean to allow standard shapes to be spawn.
  Advanced Shapes : Boolean to allow advanced shapes to be spawned.
- **Allow Rotation** : Boolean to see if rotation is allowed for block shape or not.
  Always Keep Filled : Block shape should be always filled or should add new after placing all current visible block shapes.
- **Inactive Size** : Size of block shape when its inside the shape container.
- **Drag Position Offset** : The distance between the shape and drag point.

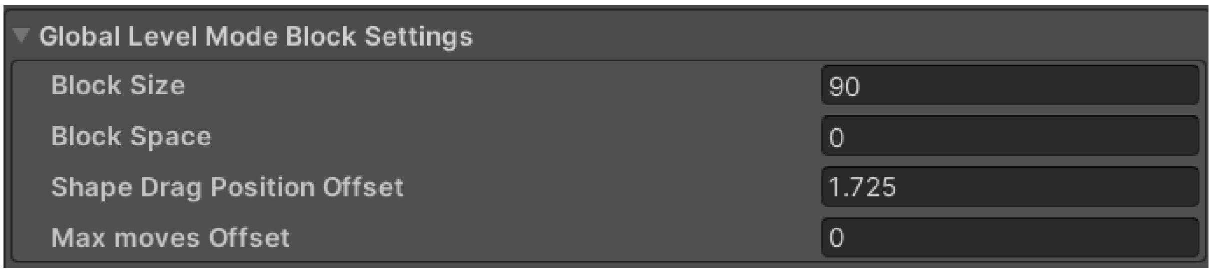
Based on one of the defined modes shown below we can set up the settings for that mode.

```
 public enum GameMode
{
    Tutorial,
    Classic,
    Timed,
    Blast,
    Advance,
    Level
}
```

Note:- Some of the settings such as blockSize and blockSpace can affect other settings as well as overall game UI.

Level Specific Global Mode Settings

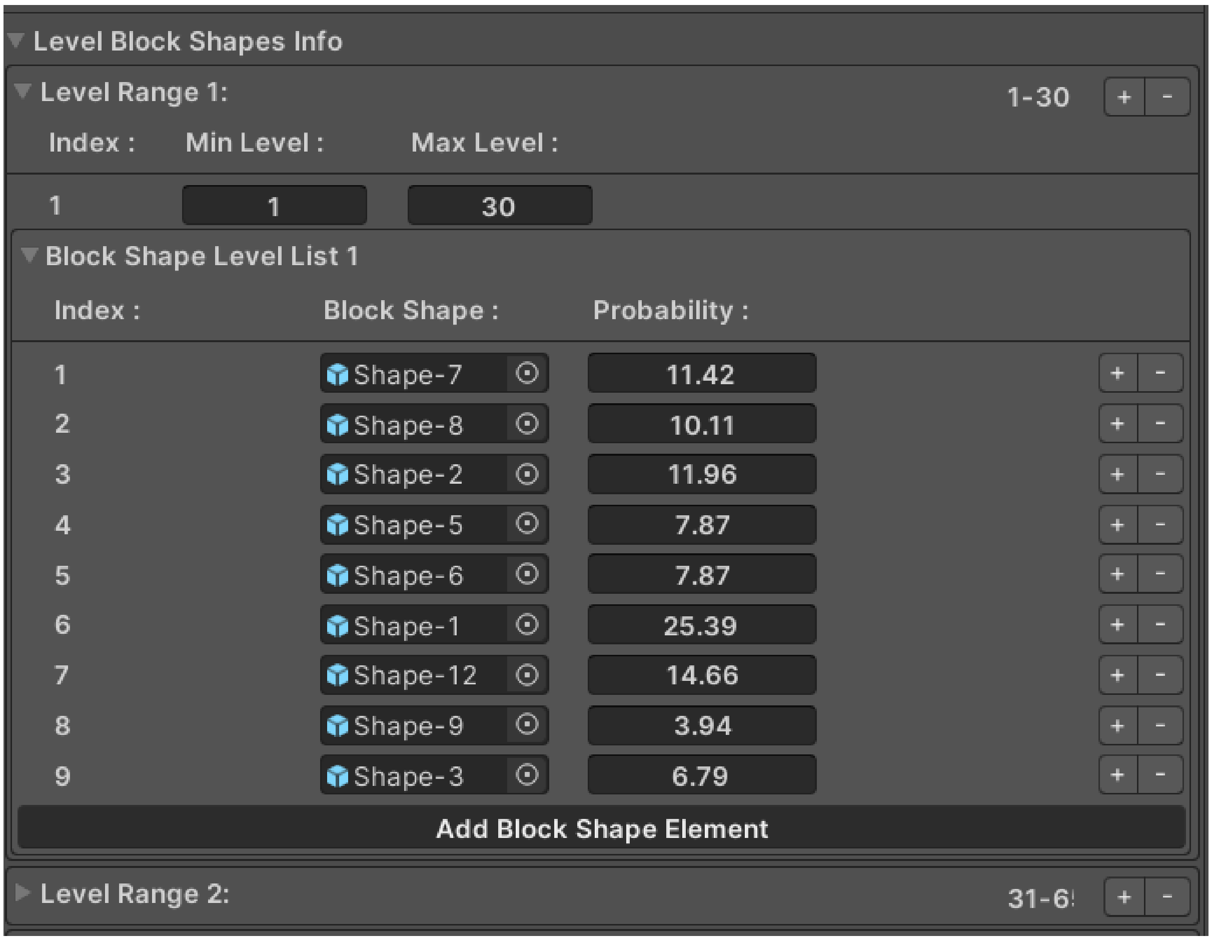
For level mode, these settings change drastically, so only global aspects are handled in this scriptable object while rest are handled in LevelSettings. These settings which override all the levels are as follows:-



Global Level Mode Block Settings

Block Size	90
Block Space	0
Shape Drag Position Offset	1.725
Max moves Offset	0

Block Size, Block Space and shape drag position offset are the same like for other modes but these values override all the levels for level mode.
Max Moves offset : The value of max moves will get incremented by the value given to this parameter for all levels.


Level Block Shapes Info

Level Range 1: 1-30

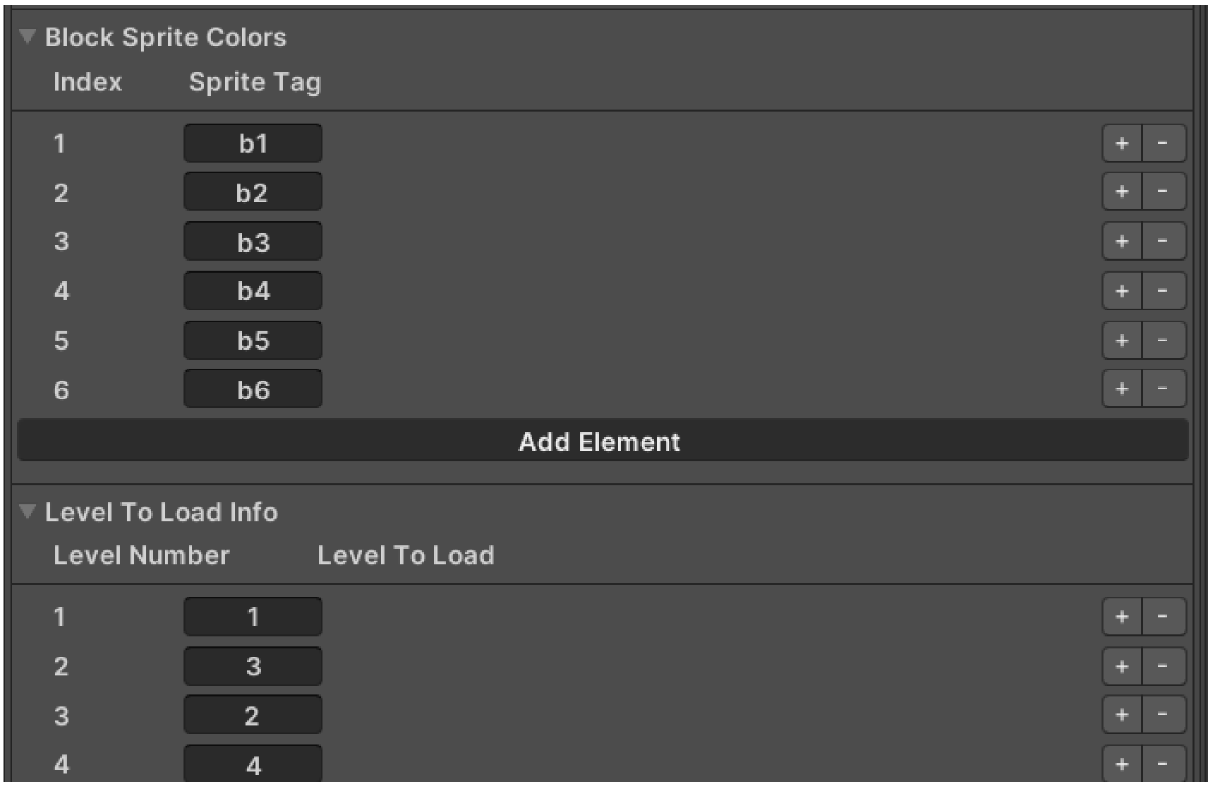
Index :	Min Level :	Max Level :
1	1	30

Block Shape Level List 1

Index :	Block Shape :	Probability :
1	Shape-7	11.42
2	Shape-8	10.11
3	Shape-2	11.96
4	Shape-5	7.87
5	Shape-6	7.87
6	Shape-1	25.39
7	Shape-12	14.66
8	Shape-9	3.94
9	Shape-3	6.79

The level block shapes info consists of a list of min level, max levels and a list of shapes with their probabilities .The Min Level and Max level is the range for the defined levels in which all levels belonging to this range will spawn shapes with the shapes and probabilities listed in the list below. These values will be overridden if “ Select shapes from this level” boolean is checked.

Block Sprite Colors and Level To Load Info



Block Sprite Colors

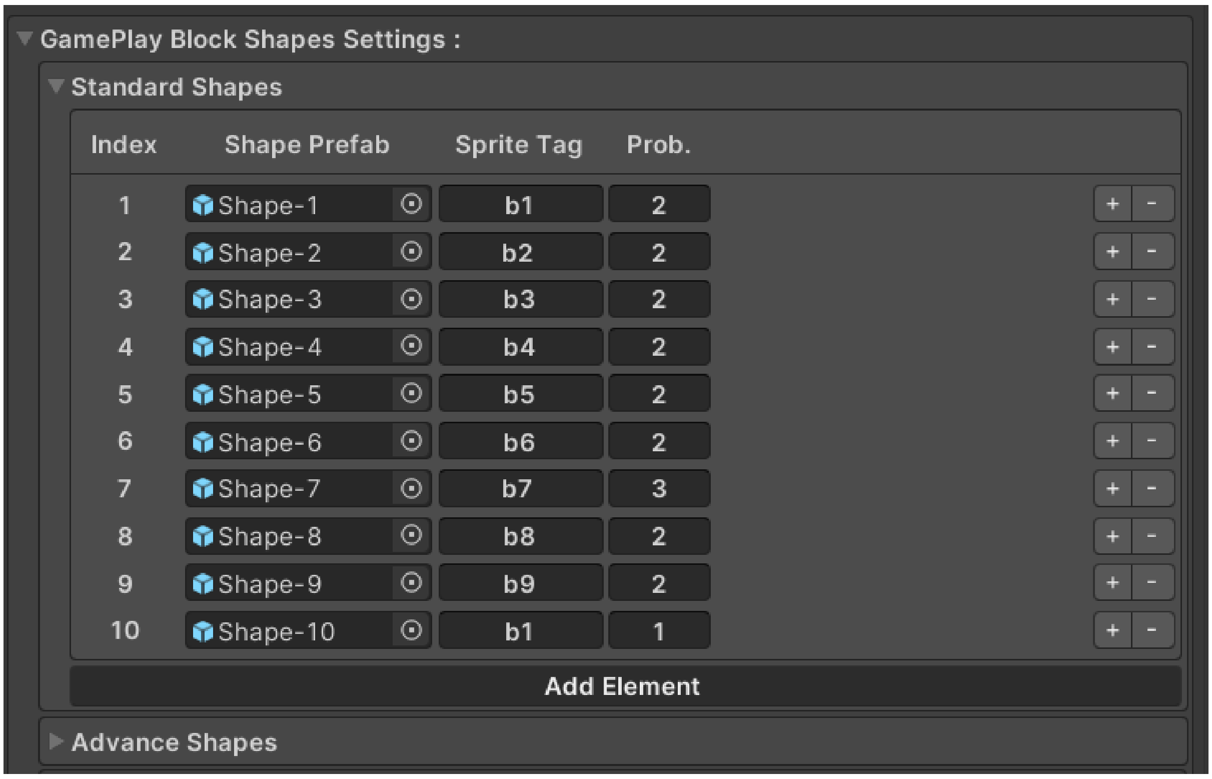
Index	Sprite Tag
1	b1
2	b2
3	b3
4	b4
5	b5
6	b6

Level To Load Info

Level Number	Level To Load
1	1
2	3
3	2
4	4

- **Block Sprite Colours** : The list stores the sprite tag which is used to retrieve the color for the blocks.
- **Level To Load Info** : This class will handle the relation between level number and which level to load from the level settings levels. In the level to load field we just write down the level we need to load and this will handle the ordering .

Gameplay Block shape Settings



GamePlay Block Shapes Settings :

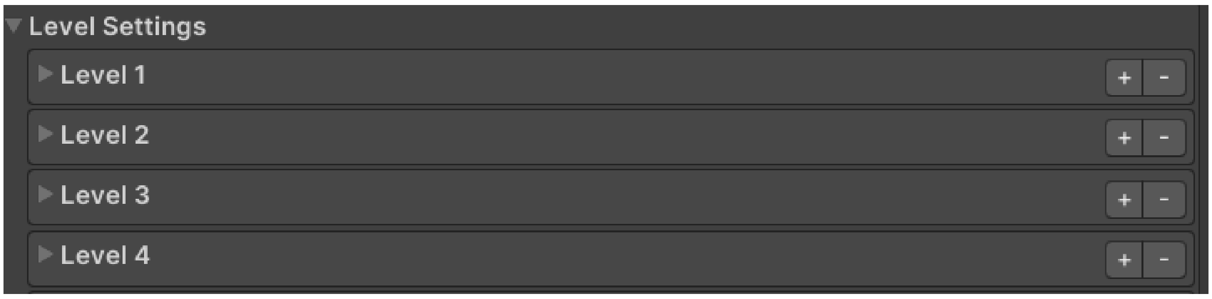
Standard Shapes

Index	Shape Prefab	Sprite Tag	Prob.
1	Shape-1	b1	2
2	Shape-2	b2	2
3	Shape-3	b3	2
4	Shape-4	b4	2
5	Shape-5	b5	2
6	Shape-6	b6	2
7	Shape-7	b7	3
8	Shape-8	b8	2
9	Shape-9	b9	2
10	Shape-10	b1	1

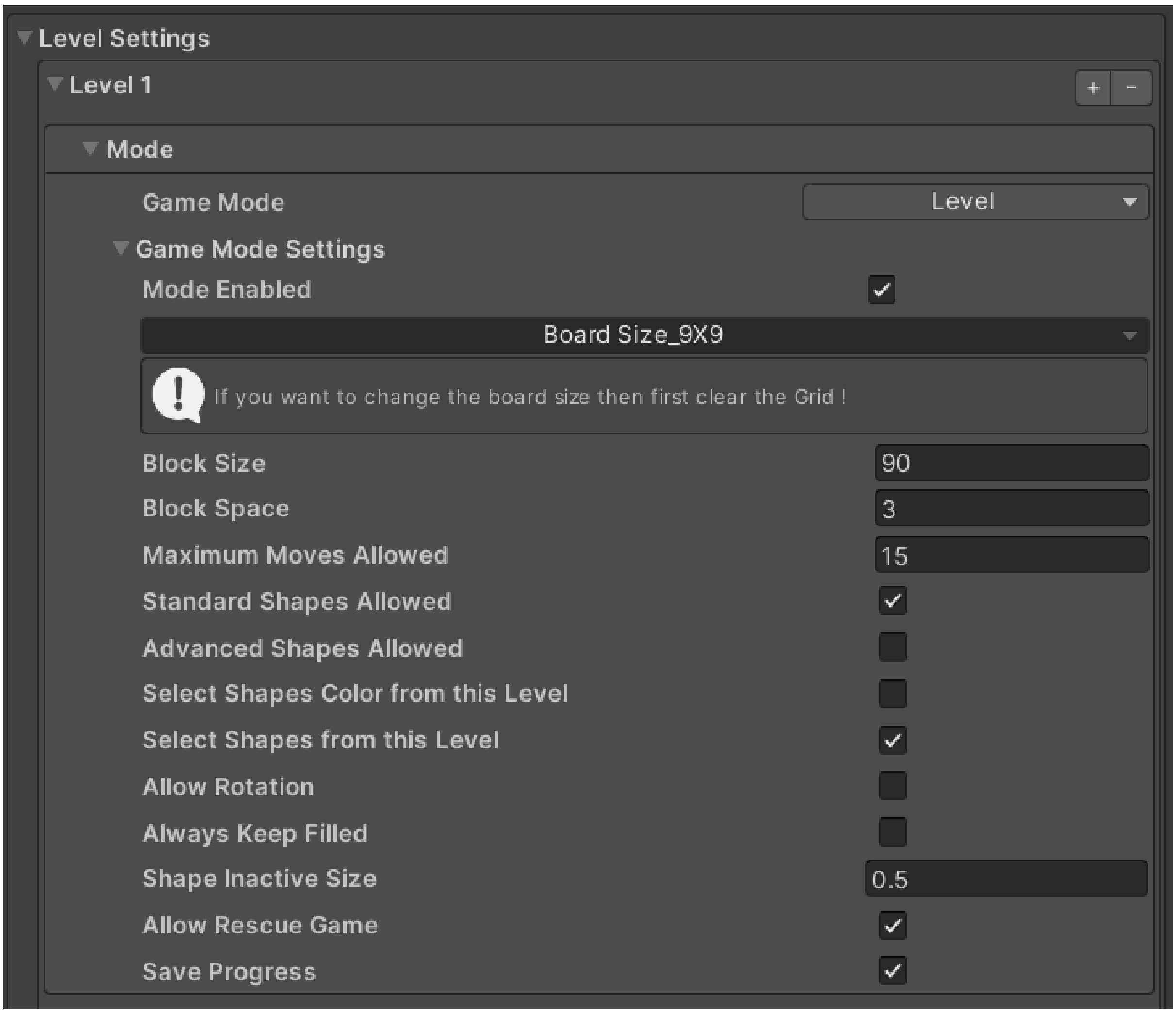
This will store the list of shapes which will be used in the game along with their probabilities and sprite tags. The shapes are divided into 2 types - Standard block shapes and Advanced block shapes. The standard shapes consist of all the simpler shapes used in classic mode. The advanced shapes are more complex shapes used in advanced mode. Level mode uses a combination of both.The sprite tag will store the color information of the shape and the probability will dictate after how long the particular shape will be spawned. For example a number 3 for probability means we will pick this shape 3 times out of the total number of the probabilities.

Level Settings

This scriptable object defines the structure of each level. A level contains data about its configuration, including game mode, goals, special block shapes, and more. The mode settings are similar to gameplay settings but with few more specific fields. Goals and board data and other level mode specific configurations are done in this scriptable object.



Mode Settings



Level Settings
Level 1
Mode

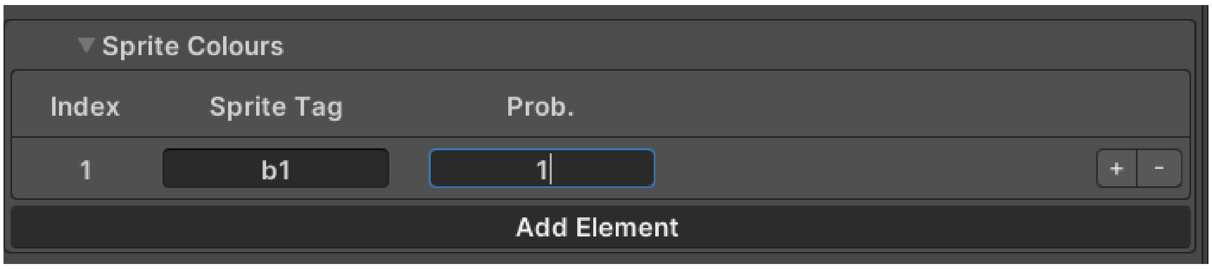
Game Mode	Level
Game Mode Settings	
Mode Enabled	✓
Board Size_9X9	
If you want to change the board size then first clear the Grid !	
Block Size	90
Block Space	3
Maximum Moves Allowed	15
Standard Shapes Allowed	✓
Advanced Shapes Allowed	
Select Shapes Color from this Level	
Select Shapes from this Level	✓
Allow Rotation	
Always Keep Filled	
Shape Inactive Size	0.5
Allow Rescue Game	✓
Save Progress	✓

- **Game Mode** : Level by default. But can be configured to other modes for special cases.
- **Maximum moves allowed** : maximum moves to complete the level within.
- **Select shapes color from this level** : Boolean to allow the color of shapes to be retrieved from this level configuration's sprite color's list.
- **Select Shapes from this level** : Boolean to allow shapes to be retrieved from this level configuration's standard and advanced shapes list.

The other settings are the same as gameplay settings Mode configuration.

Sprite Colours

List which contains sprite tag and probability for that color to appear in shapes.

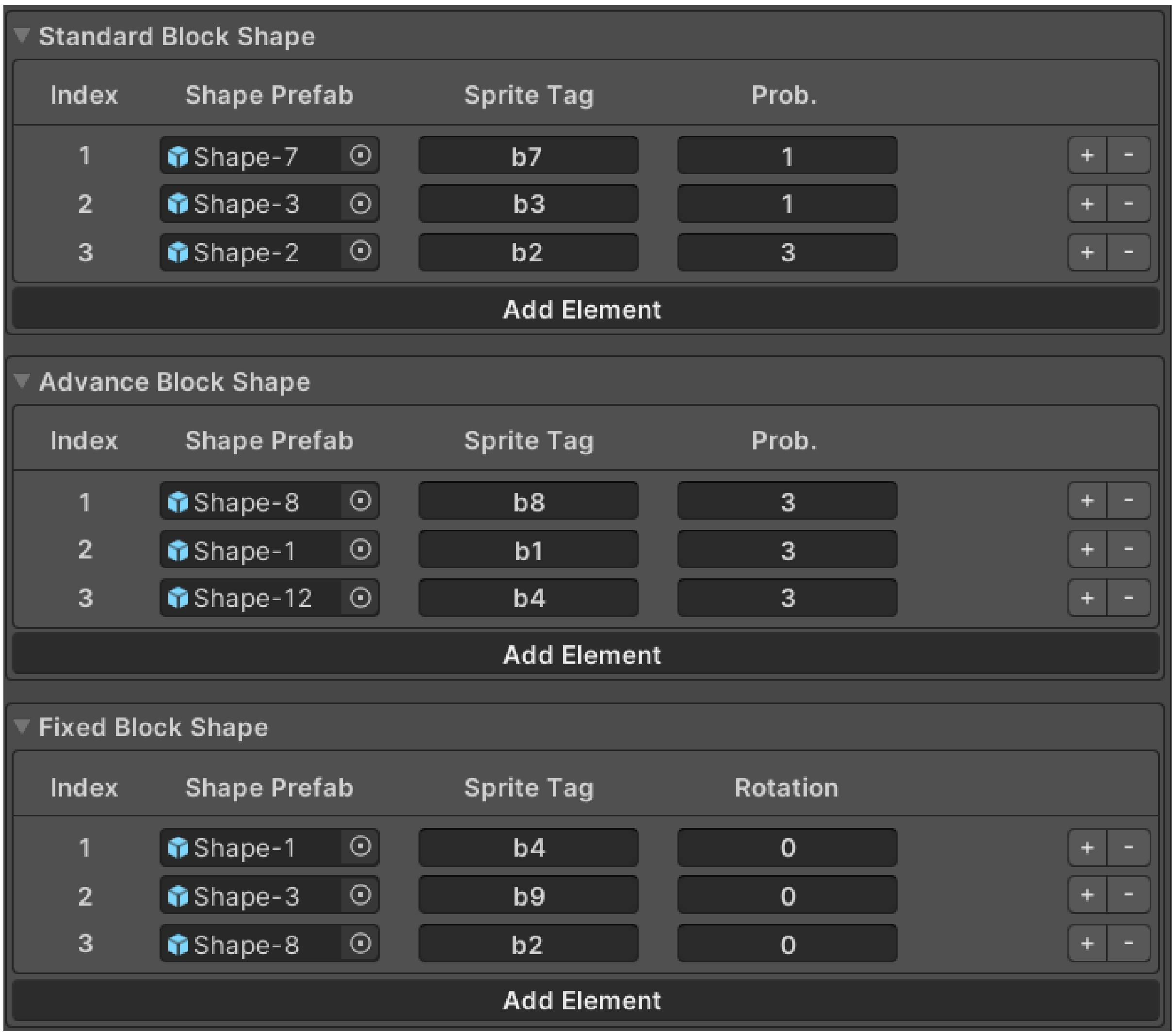


Sprite Colours

Index	Sprite Tag	Prob.
1	b1	1

Add Element

Level Mode Block Shape Lists



Standard Block Shape

Index	Shape Prefab	Sprite Tag	Prob.
1	Shape-7	b7	1
2	Shape-3	b3	1
3	Shape-2	b2	3

Add Element

Advance Block Shape

Index	Shape Prefab	Sprite Tag	Prob.
1	Shape-8	b8	3
2	Shape-1	b1	3
3	Shape-12	b4	3

Add Element

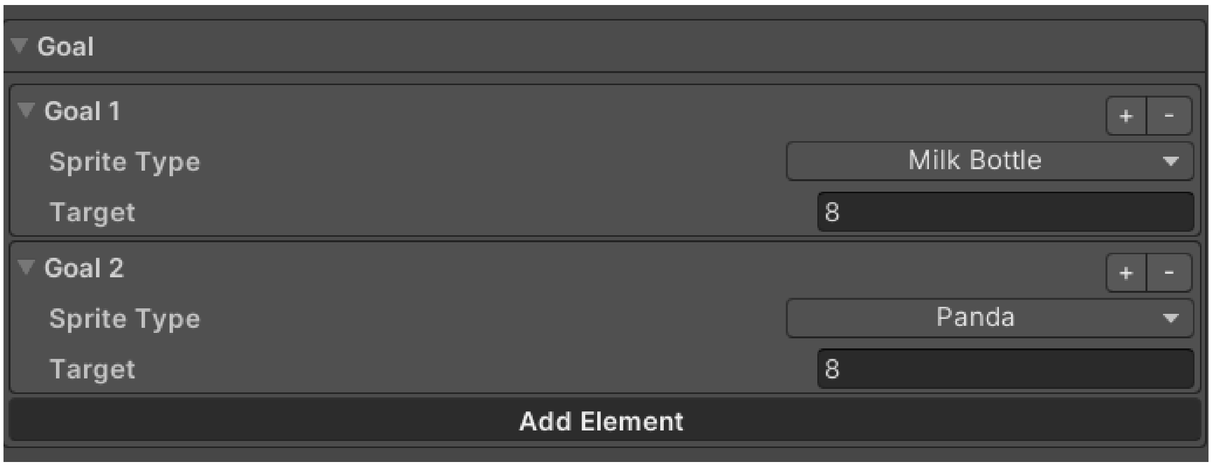
Fixed Block Shape

Index	Shape Prefab	Sprite Tag	Rotation
1	Shape-1	b4	0
2	Shape-3	b9	0
3	Shape-8	b2	0

Add Element

- **Standard Block Shape** : Level Specific standard block shape list.
- **Advanced Block Shape** : Level Specific advanced block shape list.
- **Fixed Block Shape** : First Block shapes which will appear at the start of the level. Having no value here will result in shapes getting picked from standard/advanced List based on their priority. Rotation can be changed anticlockwise based on the number provided. For example if 3 is the value then we rotate the shape 3 times anticlockwise.

Goal Configuration

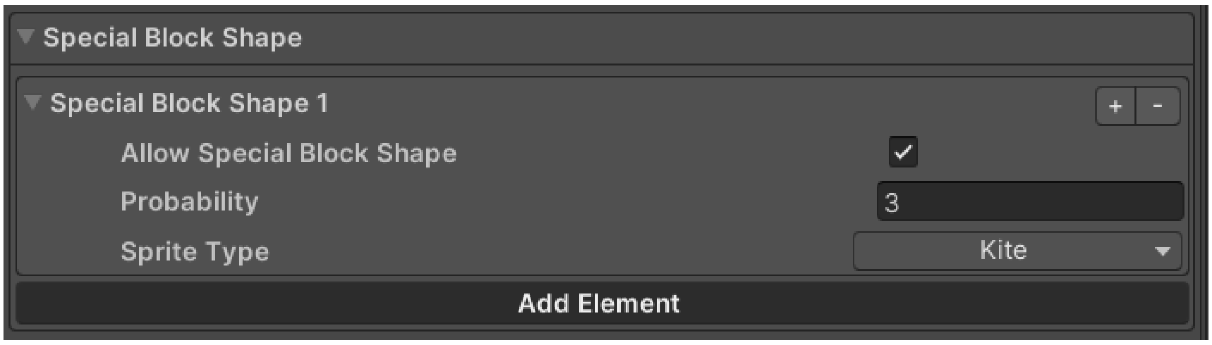

Goal
Goal 1
Sprite Type Milk Bottle
Target 8
Goal 2
Sprite Type Panda
Target 8
Add Element

- **Goal** : Displayed on the goal panel with a number which needs to decrease to 0 to win the game.
- **Sprite Type** : Enum used to define the different features in the game and for the goal panel to retrieve its sprite.
- **Target** : Number which needs to be decremented to 0 to win the game.

Some of the different sprite types which will be used in the goal panel and board are listed below :

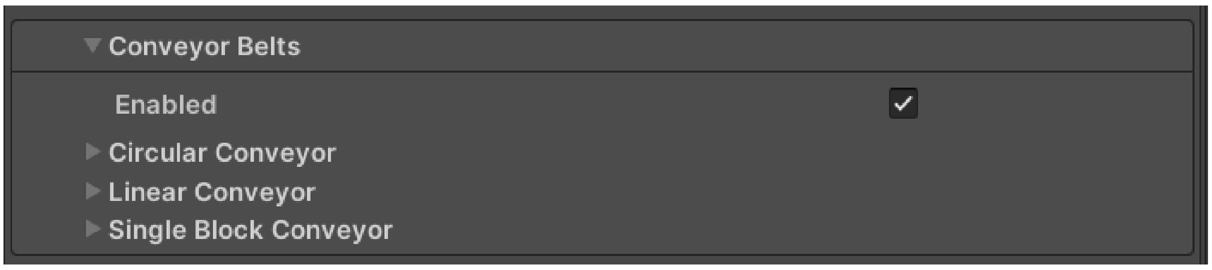
```
public enum SpriteType
{
    Empty,
    MilkBottle,
    MilkShop,
    Ice,
    Red,
    // ... other properties here ...
}
```

Special Block Shape


Special Block Shape
Special Block Shape 1
Allow Special Block Shape ✓
Probability 3
Sprite Type Kite
Add Element

- **Allow Special Block Shape** : The special block shape list handles the specific features needed for the level.
- If the level needs to have a specific sprite type feature in the shape then we use this list. This option allows whether to use the special block shape or not.
- **Probability** : after how many shapes the special shape should appear every time.
- **Sprite Type** : specifies the sprite type enum to be embedded into the block shape.

Conveyor configurations



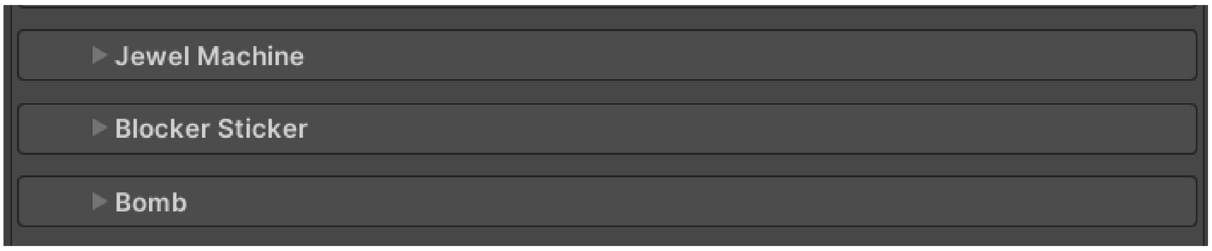
▼ Conveyor Belts
Enabled ✓
▶ Circular Conveyor
▶ Linear Conveyor
▶ Single Block Conveyor

This is a more complex section which deals with conveyors which will move grid data in a particular direction at the end of each block placed.

- **Circular Conveyor** : a closed loop conveyor in the form of a square.
- **Linear conveyor** : these typically stretch whole rows or whole columns.
- **Single Block Conveyo**r : Conveyors which occupy one block space.

Note :- Make sure to assign positions and other fields correctly, otherwise it will lead to unexpected results.

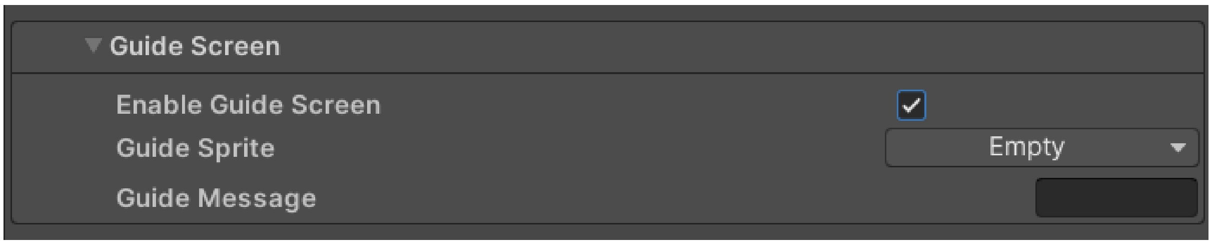
Other feature based configurations



▶ Jewel Machine
▶ Blocker Sticker
▶ Bomb

- **Jewel Machine** : Configurations for jewel machine which spawns blocks.
- **Blocker Sticker** : Configuration for a divider which occupies the space between blocks to restrict player shape placement. Can be cleared when a row/column with the stick is cleared.
- **Bomb** : Configurations for balloon and ice bombs which spawns and then explodes after a specific time.

Guide Screen Configurations


Guide Screen

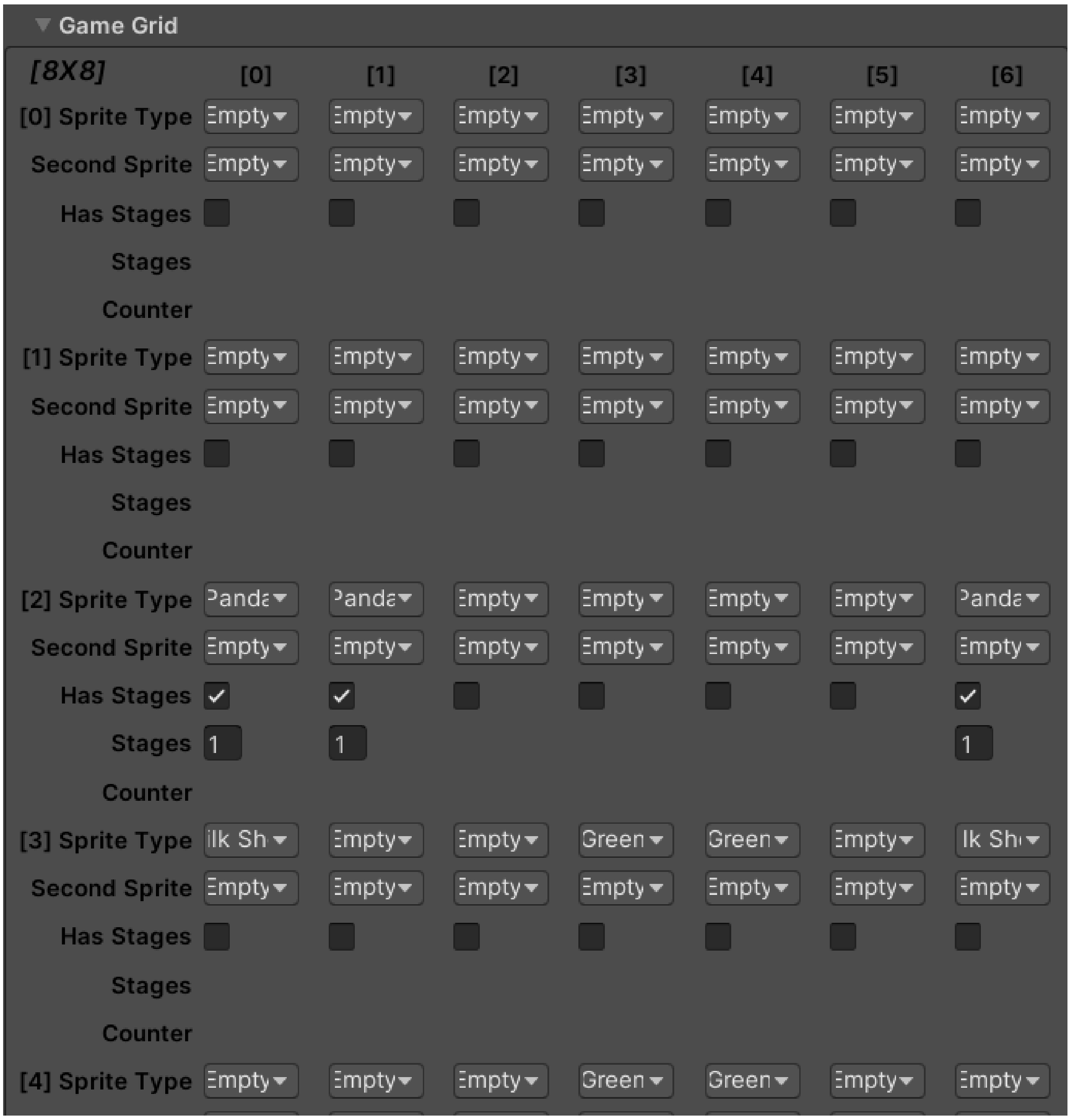
Enable Guide Screen ✓

Guide Sprite Empty

Guide Message

- **Enable Guide Screen :** boolean which decides whether the level has a guide screen which will pop up when level is loaded is true or false.
- **Guide Sprite** : sprite type enum used to select the display image in the guide panel.
- **Guide message** : Message to be displayed in the guide panel.

Game Grid Configurations


Game Grid

[8X8]	[0]	[1]	[2]	[3]	[4]	[5]	[6]
[0] Sprite Type	Empty	Empty	Empty	Empty	Empty	Empty	Empty
Second Sprite	Empty	Empty	Empty	Empty	Empty	Empty	Empty
Has Stages	☐	☐	☐	☐	☐	☐	☐
Stages							
Counter							
[1] Sprite Type	Empty	Empty	Empty	Empty	Empty	Empty	Empty
Second Sprite	Empty	Empty	Empty	Empty	Empty	Empty	Empty
Has Stages	☐	☐	☐	☐	☐	☐	☐
Stages							
Counter							
[2] Sprite Type	Panda	Panda	Empty	Empty	Empty	Empty	Panda
Second Sprite	Empty	Empty	Empty	Empty	Empty	Empty	Empty
Has Stages	✓	✓	☐	☐	☐	☐	✓
Stages	1	1					1
Counter							
[3] Sprite Type	ilk Sh	Empty	Empty	Green	Green	Empty	lk Sh
Second Sprite	Empty	Empty	Empty	Empty	Empty	Empty	Empty
Has Stages	☐	☐	☐	☐	☐	☐	☐
Stages							
Counter							
[4] Sprite Type	Empty	Empty	Empty	Green	Green	Empty	Empty

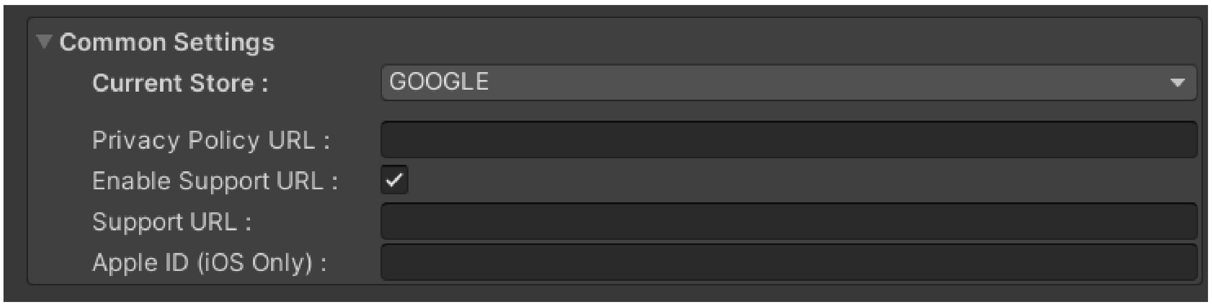
Game Grid : stores board data which will decide what will be shown on the board at the start of each level.
Sprite Type : Enum which decides the feature to be placed on the board. Empty means no feature will be placed.
Secondary Type : Enum which decides which will be the second layer feature which will be placed on the grid. The secondary type will be on top of the primary sprite type.
Has Stages : Used in features which have more than one stage. For example, pandas.
Stages : Defines how many stages the feature has before it can be reduced and cleared.
Counter : Counter needed for specific features on the board.

App Settings

App Settings constitutes the primary configuration platform that includes common settings such as URLs, privacy policy, and support URL. Additionally, it features settings related to reviews, daily rewards, and matters pertaining to currency and rewards. To access App

Settings, navigate to the GamingMonks > App Settings menu item within the Unity editor. Even though the settings are designed to be self-explanatory, here's a closer look at the details of the App Settings.

Common Settings

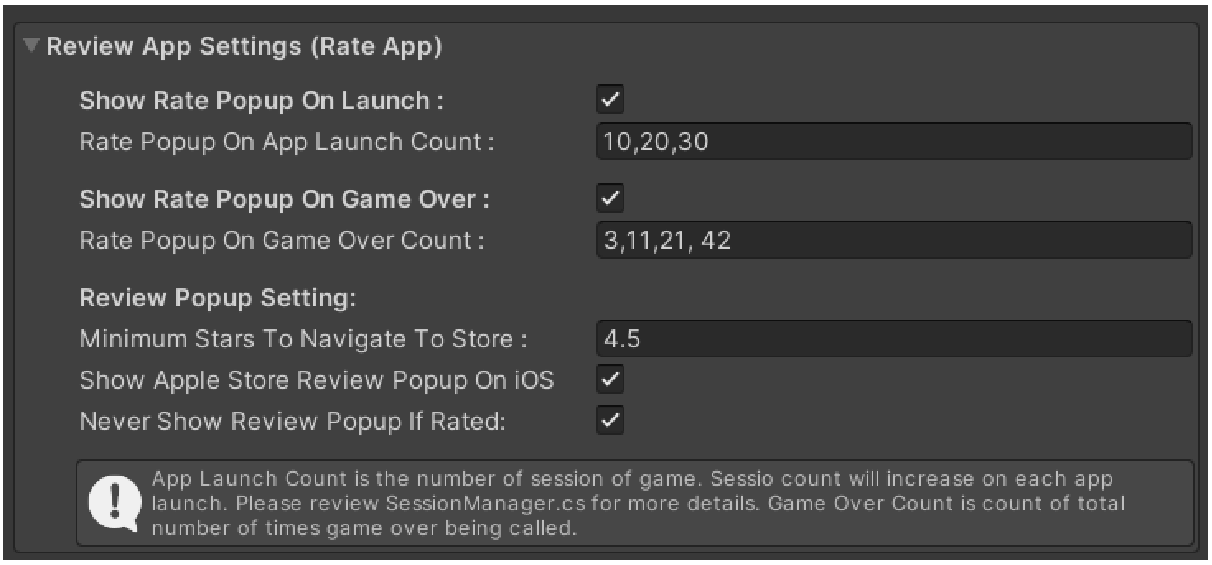


Common Settings

Current Store :	GOOGLE
Privacy Policy URL :	
Enable Support URL :	✓
Support URL :	
Apple ID (iOS Only) :	

- **Common Settings** : This segment contains various details pertaining to privacy, assistance, and store options.
- **Current Store**: This allows you to designate the active Android Store, choices include Google, Amazon, and Samsung.
- **Privacy Policy URL**: This is the URL that leads to the privacy policy.
- **Enable Support URL**: When activated, this option grants the user access to the support page via the Settings.
- **Apple ID**: This is exclusively for iOS, and it's used for navigating to the review page and the In-Game Review page.

Review App Settings



Review App Settings (Rate App)

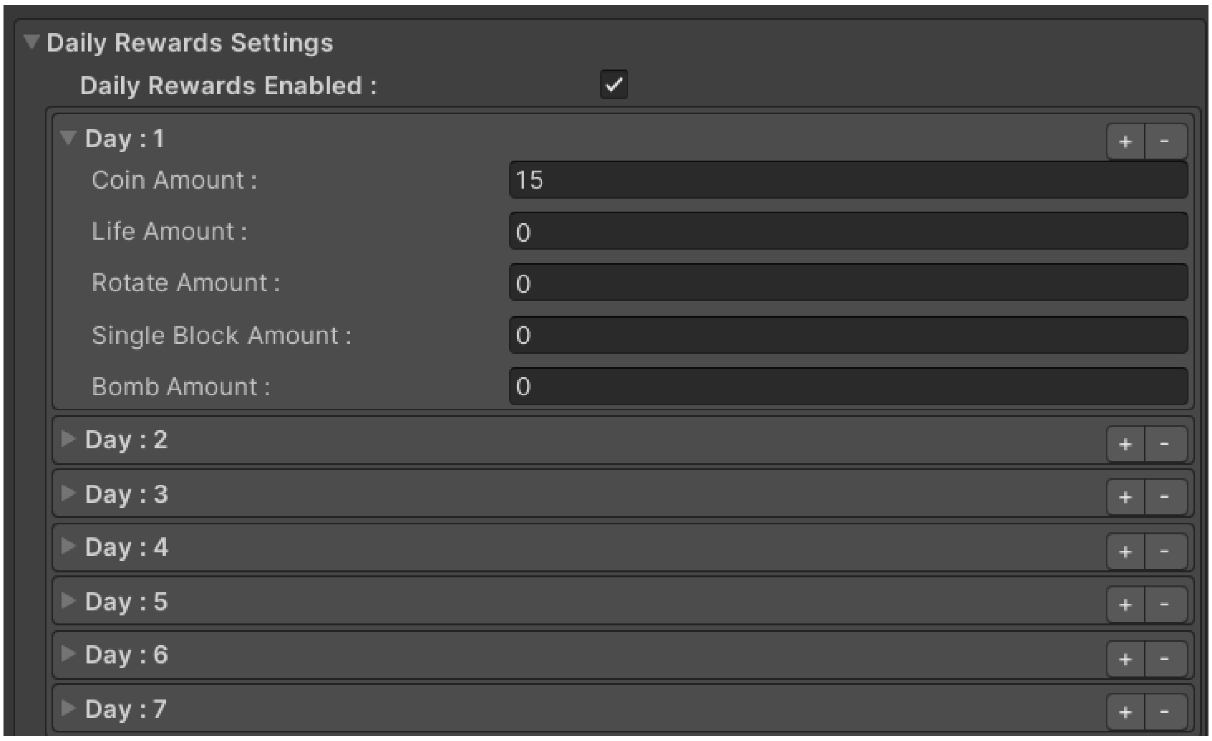
Show Rate Popup On Launch :	✓
Rate Popup On App Launch Count :	10,20,30
Show Rate Popup On Game Over :	✓
Rate Popup On Game Over Count :	3,11,21, 42
Review Popup Setting:	
Minimum Stars To Navigate To Store :	4.5
Show Apple Store Review Popup On iOS	✓
Never Show Review Popup If Rated:	✓

App Launch Count is the number of session of game. Sessio count will increase on each app launch. Please review SessionManager.cs for more details. Game Over Count is count of total number of times game over being called.

- **Review App Settings** : These configurations determine the frequency and timing of review prompts for the user. For iOS, an in-built review system is utilized, whereas for Unity editor and Android, a custom review popup is deployed. Users will select stars, which will then redirect them to the respective store.

- **Show Rate Popup On Launch** : If it's desired to have a review prompt appear upon app launch, this toggle can be activated. The specific sessions in which the user should be prompted to review the app should also be indicated. For further code modifications, refer to the SessionManager.cs script component.
- **Show Rate Popup On Game Over** : If it's intended to present a review prompt when the game concludes, this toggle can be enabled. Again, the specific app sessions when the user should be prompted to review the app should be noted.

Daily Rewards Settings


Daily Rewards Settings

Daily Rewards Enabled : ✓

Day : 1

Coin Amount :	15
Life Amount :	0
Rotate Amount :	0
Single Block Amount :	0
Bomb Amount :	0

Day : 2

Day : 3

Day : 4

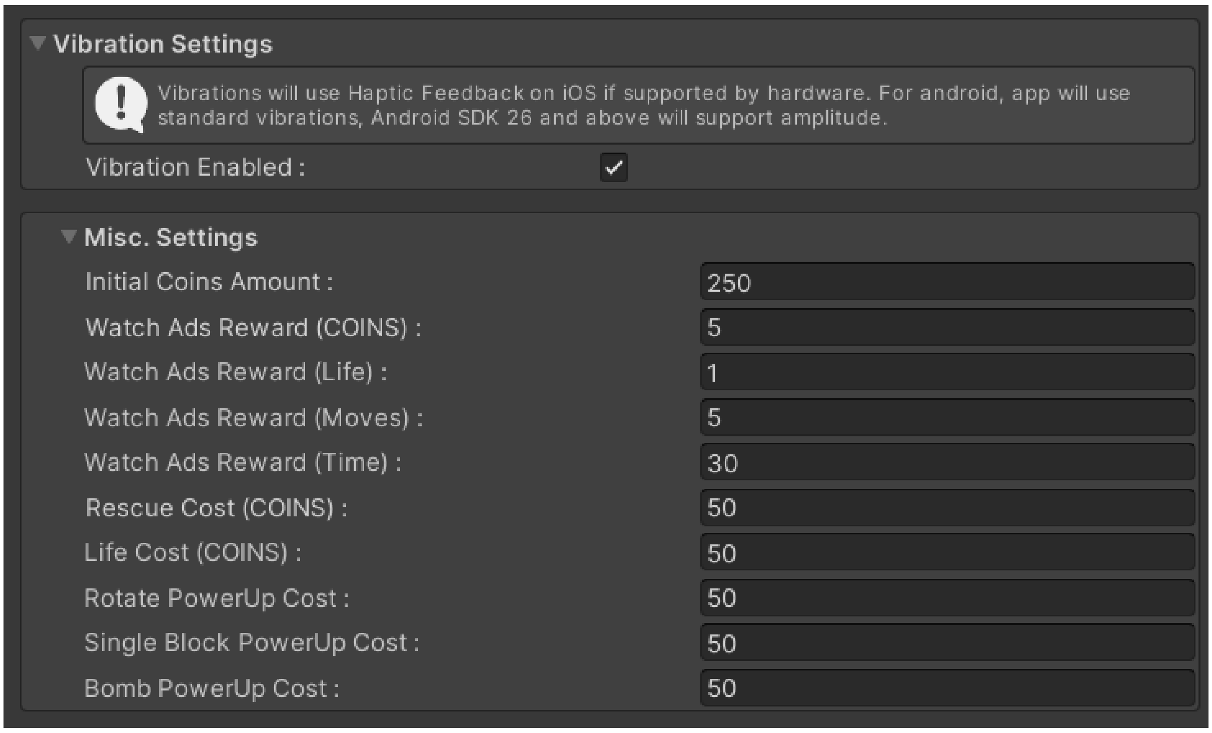
Day : 5

Day : 6

Day : 7

Users will receive specified rewards on a daily basis according to the established settings. These rewards will reset on the 8th day. So on the 8th day you will get day 1 rewards and so on. In the configurations the coin amount is the currency used in the game, Life amount is the stamina system of the game and the other 3 are power ups.

Vibration and Misc. Settings



Vibration Settings

Vibrations will use Haptic Feedback on iOS if supported by hardware. For android, app will use standard vibrations, Android SDK 26 and above will support amplitude.

Vibration Enabled :	✓

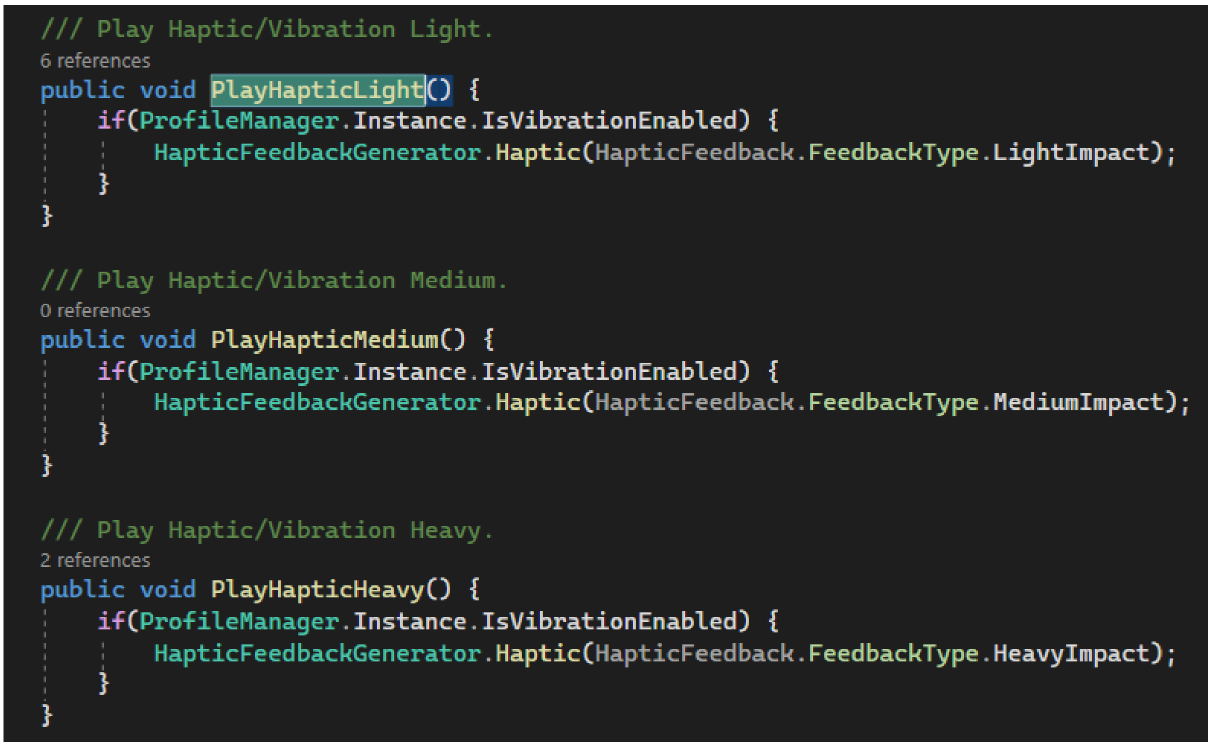
Misc. Settings

Initial Coins Amount :	250
Watch Ads Reward (COINS) :	5
Watch Ads Reward (Life) :	1
Watch Ads Reward (Moves) :	5
Watch Ads Reward (Time) :	30
Rescue Cost (COINS) :	50
Life Cost (COINS) :	50
Rotate PowerUp Cost :	50
Single Block PowerUp Cost :	50
Bomb PowerUp Cost :	50

Vibration Settings : This decides whether or not vibrations should be utilized. On iOS, Haptic Feedback will be used if the hardware supports it. For Android, standard vibrations will be employed, and amplitude support is available for Android SDK 26 and above.

Misc Settings : These are miscellaneous settings related to rewards, the starting balance of coins, the reward provided for watching the rewarded video, and the quantity of coins needed to resume a game.

Haptic Feedback



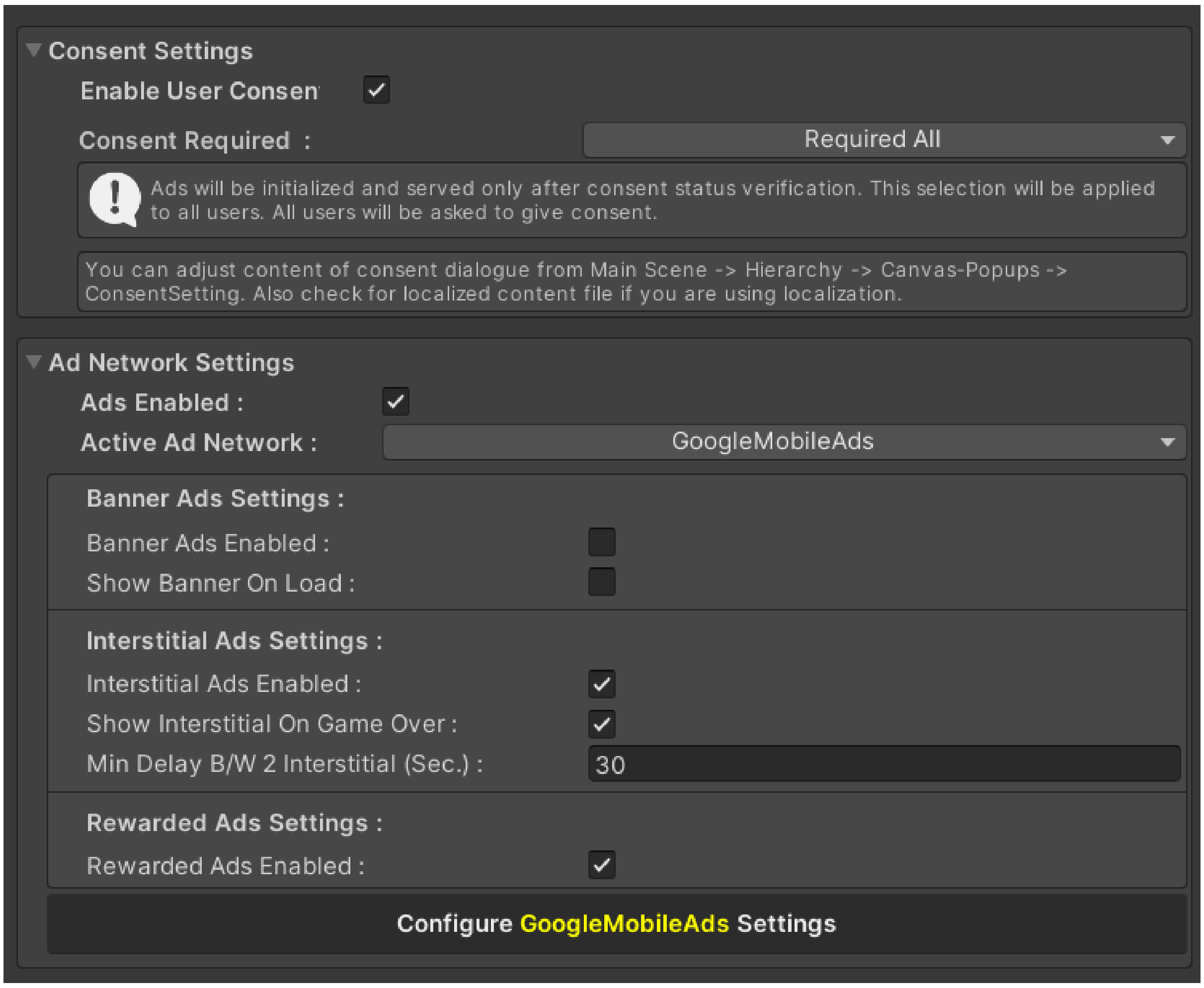
```
/// Play Haptic/Vibration Light.
6 references
public void PlayHapticLight() {
    if(ProfileManager.Instance.IsVibrationEnabled) {
        HapticFeedbackGenerator.Haptic(HapticFeedback.FeedbackType.LightImpact);
    }
}

/// Play Haptic/Vibration Medium.
0 references
public void PlayHapticMedium() {
    if(ProfileManager.Instance.IsVibrationEnabled) {
        HapticFeedbackGenerator.Haptic(HapticFeedback.FeedbackType.MediumImpact);
    }
}

/// Play Haptic/Vibration Heavy.
2 references
public void PlayHapticHeavy() {
    if(ProfileManager.Instance.IsVibrationEnabled) {
        HapticFeedbackGenerator.Haptic(HapticFeedback.FeedbackType.HeavyImpact);
    }
}
```

This game template is equipped with inherent support for iOS haptic feedback. iOS utilizes dedicated hardware for haptic feedback, but some older devices might not possess this feature. In such cases, vibrations will be implemented instead. For Android, haptic vibrations using amplitude are supported for API 26 and above. If the device API is below 26 or does not support amplitude, vibrations will be utilized instead. The Android platform requires the "android.permission.VIBRATE" permission, which will be automatically appended to the AndroidManifest.xml. Refer to the provided image for instructions on utilizing haptic feedback.

AD Settings


Consent Settings

Enable User Consen ✓

Consent Required : Required All

Ads will be initialized and served only after consent status verification. This selection will be applied to all users. All users will be asked to give consent.

You can adjust content of consent dialogue from Main Scene -> Hierarchy -> Canvas-Popups -> ConsentSetting. Also check for localized content file if you are using localization.

Ad Network Settings

Ads Enabled : ✓

Active Ad Network : GoogleMobileAds

Banner Ads Settings :

Banner Ads Enabled : ☐

Show Banner On Load : ☐

Interstitial Ads Settings :

Interstitial Ads Enabled : ✓

Show Interstitial On Game Over : ✓

Min Delay B/W 2 Interstitial (Sec.) : 30

Rewarded Ads Settings :

Rewarded Ads Enabled : ✓

Configure GoogleMobileAds Settings

The GamingMonks game template is provisioned with pre-existing ad network integration. This ad network can be easily configured by following a few simple steps outlined in this guide. The aim is to facilitate the setting up of ad monetization with this game template.

Configuring the Ad Network

This game template provides support for four distinct ad networks: Unity Ads, AdMob, AppLovin Max, and IronSource. The selection of any of these networks for monetization is permissible. In the event that a different ad network is needed, there's an option to configure a custom ad network, allowing for tailored coding for the ad network based on the app's needs.

To initiate the configuration process, open the Ad Settings from the GamingMonks > Ad Settings menu item within the Unity editor. Refer to the attached image for what the settings should look like. This setting comprises two different sections.

Consent Setting

In the event that the app necessitates user consent for serving personalized ads, the 'User consent' option should be enabled and the 'Consent Required' status should be set to 'Required All'. If consent is not needed, this can be set to 'Not Required'.

Ad Network Settings

The ad network of preference and ad configuration can be chosen based on requirements. Detailed information about all files within ad settings are provided below:
Ads Enabled: This field determines if ads should be served within the app. If unchecked, all subsequent settings will be disabled.
Active Ad Network: Select a preferred ad network for monetizing the app. Currently, support is extended to Unity Ads, Google Mobile Ads, AppLovin Max, and IronSource.
- Banner Ads Settings: This setting is to decide if banner ads should be served within the app. If enabled, ads will be loaded and displayed automatically upon ad network initialization.
- Interstitial Ads Settings: Determines if interstitial ads should be served within the app. Additional settings dictate when these ads are shown, for example, upon game over or when a user navigates from the game to the home screen.
- Rewarded Ads Settings: Specifies whether rewarded ads should be loaded within the app. With this game template, the rewarded ad is automatically configured for game rescue.
- Configure "Active Ad Network" Settings: This button leads to further configuration of the selected ad network, including Ad Keys, banner positioning, and other specifics.

NOTE: For customization of the ad serving experience and understanding the ad network workflow, refer to the AdManager.cs script component. The code contains comprehensive comments, facilitating ease of understanding.

Setting Up Unity Ads

The GamingMonks game template comes with support for Unity Monetization SDK, offering a simple step-by-step process to set up your project for unity ads. Initially, activate Unity Ads via the Services window. This can be accessed through Window > General > Services, or by selecting the Cloud icon in the toolbar. If your project isn't yet linked to the services, ensure to establish this connection first. You can refer to this guide to align your project with Unity Services.

*** CRITICAL *** Please ensure that the "Enable built-in Ads Extension" option is deactivated to prevent any potential conflicts with Unity Ads' built-in extension, which could cause errors. This option can be turned off by accessing the "Advanced" section and deselecting it, as shown in the attached image.

Integration of Unity Monetization SDK
For this, you need to download and incorporate the Unity Monetization SDK from the Unity asset store. Upon import, the app will automatically recognize the SDK and add the HB_UNITYADS scripting symbol to the player settings. Note that this scripting symbol is crucial for the utilization of the Unity Monetization SDK, so please verify its addition.

Configuration of Ad Keys
Post importing the Unity Monetization SDK, navigate to GamingMonks > Ad Settings in the Unity editor to open Ad Settings. Here, choose Unity Ads from the Active Ad Network dropdown and click on the "Configure Unity Ads Settings" button. This action will launch a new window with settings specific to Unity Monetization.

In this section, fill in the Game Id and designate placements for banner, interstitial, and rewarded ads for both Android and iOS platforms. You also have the option to adjust the Test mode and banner placement as per your requirements. With these steps, your app is ready for monetization through Unity Ads.

NOTE: It is advised to review the official guide for the implementation of Unity Monitization SDK regularly to stay updated with any new modifications or updates. You can modify the Unity Monetization SDK implementation code from the UnityAdsManager.cs script component.

Setting Up Google Ads
This guide elaborates on the process of integrating and implementing ads in the application using Google Mobile Ads SDK. Here is a detailed breakdown of the steps:

Importing Google Mobile Ads SDK
Begin by downloading and importing the Google Mobile Ads SDK from its official site. Once imported, the application will automatically detect the SDK and add the HB_ADMOB scripting define symbol to the player settings. This scripting symbol is a prerequisite for using the Google Mobile Ads SDK, so please verify its addition.

Configuring your AdMob App ID
In the Unity editor, navigate to Assets > Google Mobile Ads > Settings. Enable AdMob by checking the 'Enabled' box under the Google AdMob section. Then, input your AdMob App ID for both Android and iOS in the respective fields.

Utilizing Unity Play Services Resolver
Navigate to Assets > Play Services Resolver > Android Resolver > Resolve in the Unity editor. The Unity Play Services Resolver library will subsequently transfer the declared dependencies into the Assets/Plugins/Android directory of your Unity application.
NOTE: Dependencies for the Google Mobile Ads Unity plugin can be found in Assets/GoogleMobileAds/Editor/GoogleMobileAdsDependencies.xml

Setting up Ad Keys
With the Google Mobile Ads SDK imported, open Ad Settings via GamingMonks > Ad Settings in the Unity editor. Now, select 'Google Mobile Ads' from the 'Active Ad Network' dropdown menu and click on the 'Configure Google Mobile Ads Settings' button. This will bring up a new window with Google Mobile Ads settings.

In this window, configure the App ID and Ad Unit Keys for banner, interstitial, and rewarded ads for both Android and iOS. Adjust the 'Tag For Child Treatment' and banner placement as required. With these steps, your application is ready to be monetized using the Google Mobile Ads SDK.

NOTE: Be sure to periodically check the official guide for implementing the Google Mobile Ads SDK in case there are any recent updates or changes. You can modify the Google Mobile Ads SDK implementation code via the GoogleMobileAdsManager.cs script component.

Settings Up Applovin Max Ads
This instruction manual is intended for the purpose of app monetization using the AppLovin Max SDK. It provides a comprehensive guide on setting up and displaying ads using AppLovin Max SDK.

Importing the AppLovin Max SDK
Visit the official website to download and import the AppLovin Max SDK. It might be necessary to sign up to access this page. Once imported, the app will automatically identify the SDK and add the HB_APPLOVINMAX scripting define symbol to the player settings. This symbol is crucial for using the AppLovin Max SDK, so ensure that it is added.

Configuring Ad Keys
After importing the AppLovin Max SDK, navigate to GamingMonks > Ad Settings from the menu item within the unity editor. Here, select AppLovin Max as the Active Ad Network from the dropdown menu and click on the "Configure AppLovin Max Settings" Button. This action opens a new window with settings specific to AppLovin Max.

You can then set up the SDK Key and Ad Unit Ids for banner, interstitial, and rewarded ads for both Android and iOS platforms respectively. And that's it! Your app is now prepared to monetize through the AppLovin Max SDK.

NOTE: Stay updated with the official guide on implementing the AppLovin Max SDK for any recent updates or modifications. You can customize the AppLovin Max SDK implementation code from the AppLovinAdManager.cs script component.

Setting Up IronSource Ads
This manual provides a comprehensive guide for monetizing your app using the IronSource SDK. It offers detailed steps for setting up and displaying ads with the IronSource SDK.

Importing the IronSource SDK
You can download and import the IronSource SDK from its official website. Following the import, the app will automatically recognize the SDK and integrate the HB_IRONSOURCE scripting define symbol into the player settings. It's crucial to note that this scripting symbol is required to utilize the IronSource SDK, so ensure its presence. Special settings pertain to the IronSource SDK, notably concerning custom Gradle settings and iOS info.plist; make sure to properly understand and implement these parts from the official implementation guide.

Configuring Ad Keys
Once the IronSource SDK has been imported, proceed to open Ad Settings from GamingMonks > Ad Settings in the menu item inside the Unity editor. At this point, select IronSource as the Active Ad Network from the drop-down menu and click on the "Configure IronSource Max Settings" Button. This action opens a new window featuring IronSource settings.

Now, you can set up App IDs for both Android and iOS respectively. And that's it! Your app is now equipped to monetize through the IronSource SDK.

NOTE: Please refer to the official guide for implementing the IronSource SDK to stay informed about any recent updates or available changes. You can alter the IronSource SDK implementation code via the IronSourceAdManager.cs script component.

Setting Up Custom Ad Network
If you wish to monetize your app with an ad network that's not included in the pre-existing options, this guide will assist you in integrating such a network into your app.

Download and Incorporate the Ad Network
First, download and import your chosen Ad Network from its official site and go through the basic integration guide. Next, access the CustomAdManager.cs script component from the project.

Modify SDK Code
The app will automatically call upon certain ad-specific methods for initialization, ad display, ad hiding, and so on. Below is a detailed explanation of all the pre-coded methods within the CustomAdManager script:

- **InitializeAdNetwork**: The app automatically invokes this method. Add your ad network initialization code here. Also, remember to call the "StartLoadingAds" and its subsequent method to load and prepare ads for display.
- **ShowBanner**: This method is automatically called based on your ad configuration to display banner ads within the app. Insert code specific to banner ad display in this method, if supported by the ad network.
- **HideBanner**: This method is automatically invoked based on your ad configuration to hide ads. Include code specific to hiding banner ads in this method if your ad network supports it.
- **IsInterstitialAvailable**: This method is invoked to determine if an Interstitial ad is ready for display. Return a boolean 'true' if an interstitial ad is available and prepared for display, and 'false' otherwise.
- **ShowInterstitial**: This method is automatically invoked based on your ad configuration to display interstitial ads within the app. Add code specific to interstitial ad display in this method if supported by the ad network.
- **IsRewardedAvailable**: This method is invoked to check if a rewarded ad is ready for display. Return a boolean 'true' if a rewarded ad is available and prepared for display, and 'false' otherwise.
- **ShowRewarded**: This method is automatically called upon based on your ad setup to display rewarded ads within the app. Insert code specific to rewarded ad display in this method, if supported by the ad network.

Managing Callbacks
We strongly suggest reviewing the AdManager.cs to understand how ad callbacks function. Invoke the relevant callback to AdManager to handle it forward. We've provided example snippets within the CustomAdManager script for easier comprehension.

Setting up AdMob Ad Network

Setting up Google AdMob for Unity involves a series of steps, which include creating an AdMob account, setting up your ad units on AdMob, and then integrating the Google Mobile Ads Unity plugin into your Unity project. Here's a basic guide:

Create an AdMob Account:
If you don't already have an AdMob account, you'll need to create one. Visit the Google AdMob website (https://admob.google.com) and sign up.

Create Ad Units:
Once you have an AdMob account, sign in and create a new Ad unit. You can choose from different ad formats like Banner, Interstitial, or Rewarded Ads depending on your need. When you create an ad unit, you will receive an Ad Unit ID, which you'll need later.

Download and Import Google Mobile Ads Unity Plugin:
Next, you need to import the Google Mobile Ads Unity Plugin into your Unity project. You can download it from the Unity Asset Store or GitHub.

UnityAssetStore:
https://assetstore.unity.com/packages/tools/integration/google-mobile-ads-sdk-160890
GitHub: https://github.com/googleads/googleads-mobile-unity
After downloading the plugin, go to Assets > Import Package > Custom Package in Unity and select the Google Mobile Ads Unity Plugin package file to import it into your project.

Configure Google Mobile Ads SDK:
Open Assets > Google Mobile Ads > Settings from the Unity Editor. Input your AdMob App ID into the field in the Inspector and make sure that Google AdMob is enabled.

Implement Ads in your Game:
Now you're ready to implement ads in your game. The Google Mobile Ads Unity Plugin includes C# scripts which you can customize to control when and how ads are shown.

Here's a simple example of how you might show a banner ad with AdMob:

```csharp
using GoogleMobileAds.Api;

private void Start()
{
    // Initialize the Google Mobile Ads SDK.
    MobileAds.Initialize(initStatus => { });

    // Create a banner ad request.
    AdRequest request = new AdRequest.Builder().Build();

    // Define your Ad Unit ID.
    string adUnitId = "YOUR_AD_UNIT_ID";
    // Create a BannerView.
       BannerView  bannerView  =  new  BannerView(adUnitId,  AdSize.Banner,
AdPosition.Bottom);

    // Load the banner with the request.
    bannerView.LoadAd(request);
}
```

Test Your Ads:
Always make sure to test your ads before publishing your game. AdMob provides test ad units that you can use during development to ensure that ads are integrating correctly without accruing invalid impressions.
Remember to replace YOUR_AD_UNIT_ID with the actual Ad Unit ID you received when you created your AdMob ad unit.

Build Your Project:
Finally, you can build your project. You should now be able to see the ads displayed according to the logic you implemented in your game.
This is a simplified guide, and the actual implementation may depend on your game's specific needs. Always refer to the official Google AdMob and Google Mobile Ads SDK documentation for the most accurate and up-to-date information.

Note:- For interstitial ads, rewarded ads, etc you can view the official admob documentation for more details.

Showing Ads
The application is preconfigured to display Banner ads and Interstitial ads when the game concludes or when the player exits the game. Rewarded ads are configured to appear during the rescue game. If you need to modify the timing or locations of ads, refer to the following instructions.

Displaying a Banner Ad
To present a banner ad, invoke the ShowBanner method with the instance of AdManager like so:
AdManager.Instance.ShowBanner();

If you wish to hide the banner ad, invoke the HideBanner method in this manner:
AdManager.Instance.HideBanner();

Displaying an Interstitial Ad
For displaying an interstitial ad, invoke the ShowInterstitial method with the instance of AdManager, as shown here:
AdManager.Instance.ShowInterstitial();

It's advisable to verify if the interstitial ad is loaded and ready to be displayed before presenting it. This can be accomplished as follows:
bool isAvailable = AdManager.Instance.IsInterstitialAvailable();

Displaying a Rewarded Ad
To present a rewarded ad, invoke the ShowRewardedWithTag method with the instance of AdManager, like so:
AdManager.Instance.ShowRewardedWithTag(“placementTag”);

Similar to interstitial ads, it's recommended to check if the rewarded ad is loaded and ready to be shown. This can be done as follows:
bool isAvailable = AdManager.Instance.IsRewardedAvailable();

Handling Rewarded Ads Reward Callback:
To manage rewards upon completion of rewarded ads, please register the AdManager.OnRewardedAdRewardedEvent callback.

IAP Settings

GamingMonks game templates utilize the Unity IAP SDK for incorporating and utilizing in-app purchasing within the game. This manual will assist you in setting up and using Unity's in-app purchasing SDK within this game template.

Activating In-App Purchasing

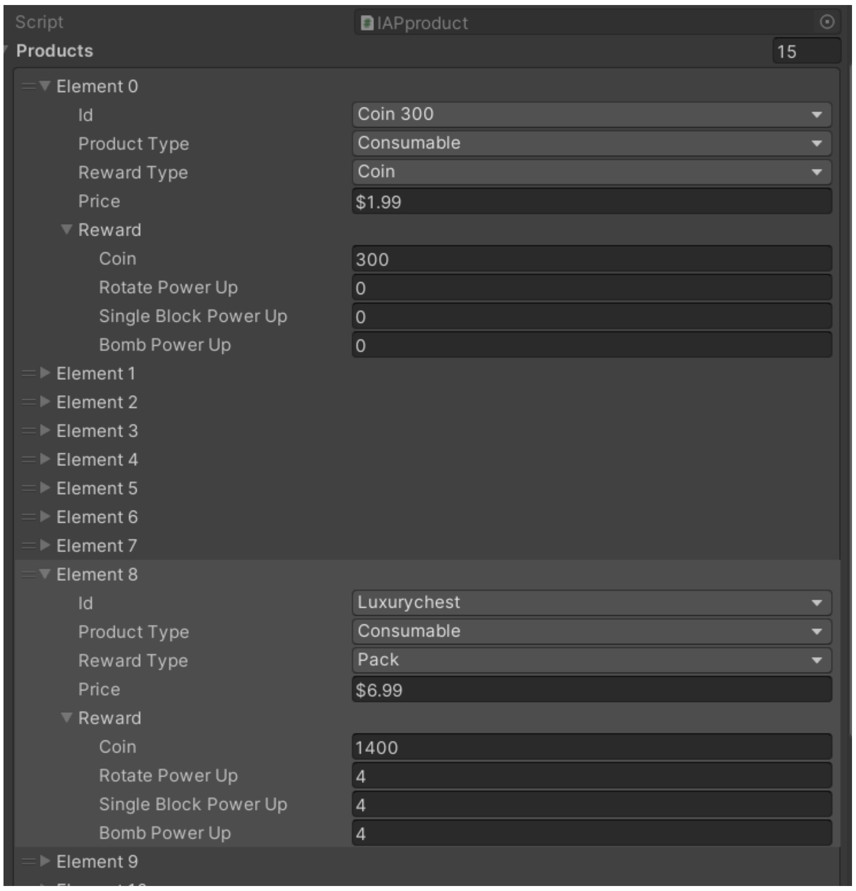
Unity IAP is a component of Unity Services and can be activated from the Services Window of the Unity editor. To access the Services Window, navigate to Window > General > Services, or select the cloud icon in the toolbar. If your project has not been linked to a Services project yet, you'll need to do that first. Refer to this guide for help with linking your project to Unity Services.

After your project is linked, select In-App Purchasing and click the Enable button in the Service Window to activate In-App Purchasing.

Upon importing the In-App purchase SDK, please open Unity IAP Settings from the GamingMonks > Unity IAP Settings menu item within the Unity editor. This action will open the in-app setup page for configuring IAP products.

Configuring InApp Products
This game template is pre-configured with products. You can effortlessly modify, add, or delete products here. Below is a detailed description of each field to configure:
- **Product ID**: The SKU of the product.
- **Product Type**: Choose the product type from consumable, non-consumable, and subscription options.
- **Reward Type**: The reward type can be selected from COINS, REMOVE Ads, or Other.

- **Price**: It defines the purchase price for the IAP.
- **Reward** : Class which contains the currency(COINS) and different power ups.



Script	IAPproduct
Products	15
Element 0	
Id	Coin 300
Product Type	Consumable
Reward Type	Coin
Price	$1.99
Reward	
Coin	300
Rotate Power Up	0
Single Block Power Up	0
Bomb Power Up	0
Element 1	
Element 2	
Element 3	
Element 4	
Element 5	
Element 6	
Element 7	
Element 8	
Id	Luxurychest
Product Type	Consumable
Reward Type	Pack
Price	$6.99
Reward	
Coin	1400
Rotate Power Up	4
Single Block Power Up	4
Bomb Power Up	4
Element 9	

Buying a Product
To initiate a product purchase, the UnityIAPButton script is necessary. Add the UnityIAPButton script component to any UI Button that will function as an IAP Button. After adding it, please choose the Button Type from the purchase and restore options. Depending on the Button Type selected - purchase or restore - this button will automatically invoke the PurchaseProduct method or RestoreAllProducts method from IAPManager.

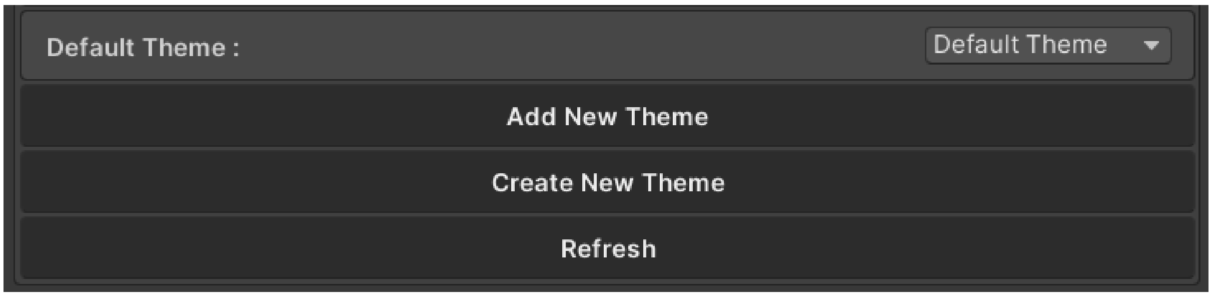
Button Type: This determines whether the Unity IAP Button will serve as a purchasing button or will act to restore non-consumable products.
Product Name: If the button type is set as Purchase, you can select the product to be requested for purchase.
Title Text: This is the product's title; this field can be omitted if it's not needed.
Description Text: This is the product's description; this field can also be skipped if not needed.
Price Text: The price of the product. Assign this field if the Localised price of IAP needs to be displayed. This field can also be omitted if not needed.

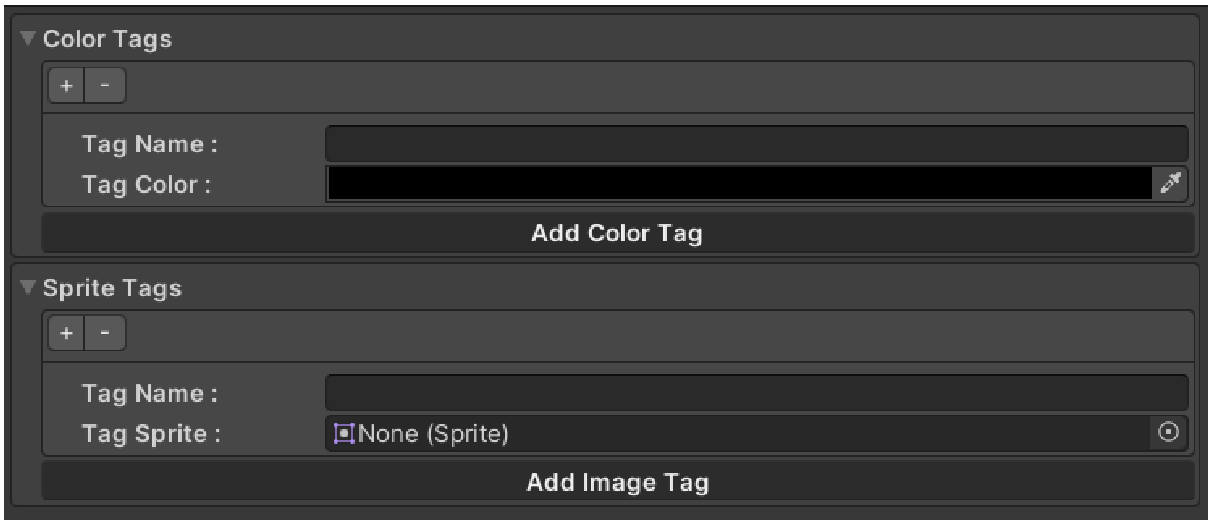
UI Theme Settings

The GamingMonks game template includes a pre-integrated UI theme feature that can be set up using a few straightforward steps. To set up the UI themes, there are two main parts: Creating a UI theme and Configuring a UI theme.

Building a Theme
Start the setup by accessing UI Theme Settings under GamingMonks > UI Theme Settings in the Unity editor. The interface should match the attached image. Enable the Use UI Theme Toggle Button and press the Create New Theme button.

When you press the Create New Theme button, a new Scriptable object editor will appear. UI Theme comprises two aspects, which will be displayed as in the attached image.

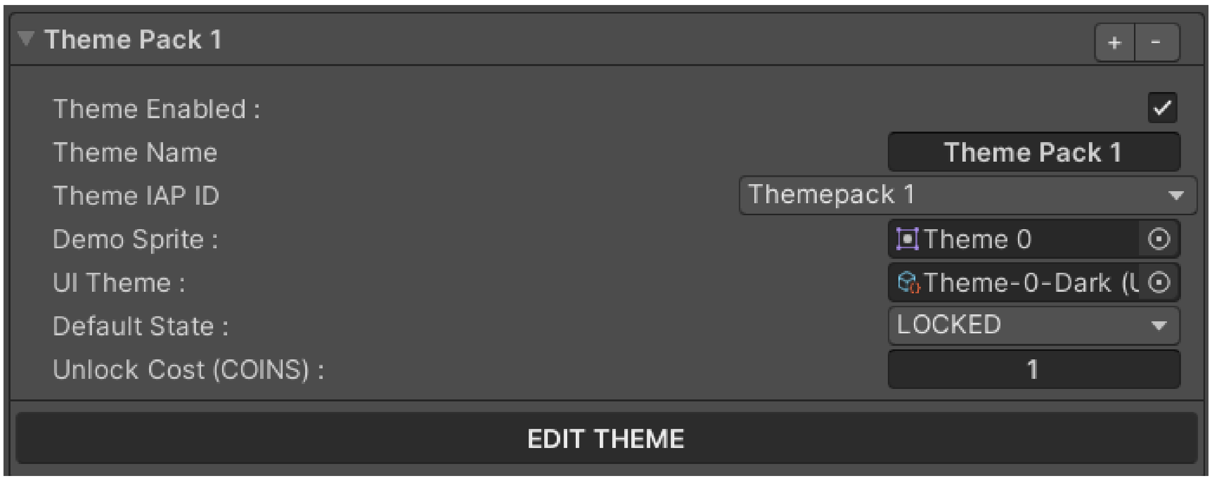




- **Color Tags**: These are simple color configurations operating on a key-value basis, where the tag is the key, and the associated color is the value. Please set all necessary color tags and their colors.
- **Sprite Tags**: These contain a sprite image tagged to them. Configure as many as required.

Remember, to avoid errors, ensure all UI Themes carry the same color tags and sprite tags.

Configuring the UI Theme


Theme Pack 1

Theme Enabled : ✓
Theme Name: Theme Pack 1
Theme IAP ID: Themepack 1
Demo Sprite : Theme 0
UI Theme : Theme-0-Dark (L
Default State : LOCKED
Unlock Cost (COINS) : 1

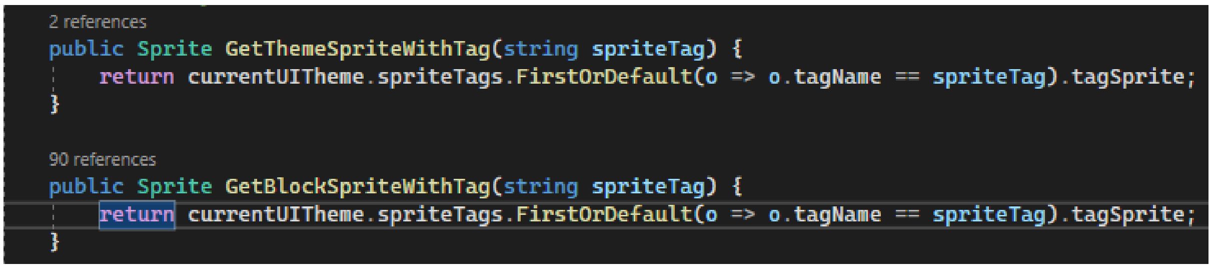
After creating the UI themes, configure these themes in the UI Theme Setting. For every UI theme created, add the corresponding configuration. These configurations encompass:

- **Theme Enabled**: A checkbox to toggle theme activation.
- **Theme Name**: The name of the theme configuration.
- **Theme IAP ID**: Theme ID for playstore.
- **Demo Sprite**: Assign a demo sprite here. The demo sprite will be visible in the theme selection screen. You can further configure the theme selection screen from the MainScene -> Canvas-Popups -> SelectTheme popup.
- **UI Theme**: Here, assign the UI Theme you've created with Color Tags and Sprite Tags.
- **Default State**: Defines whether a theme is locked or unlocked.
- **Unlock Cost** : defines how much is the cost to unlock the theme if coins are used.

Remember, the essence of this process is to maintain the important terminologies unchanged, specifically the names preceding the colon symbol.

Using UI Themes

The following methods of the ThemeManager.cs script is used to retrieve the necessary sprite by passing the needed tag.

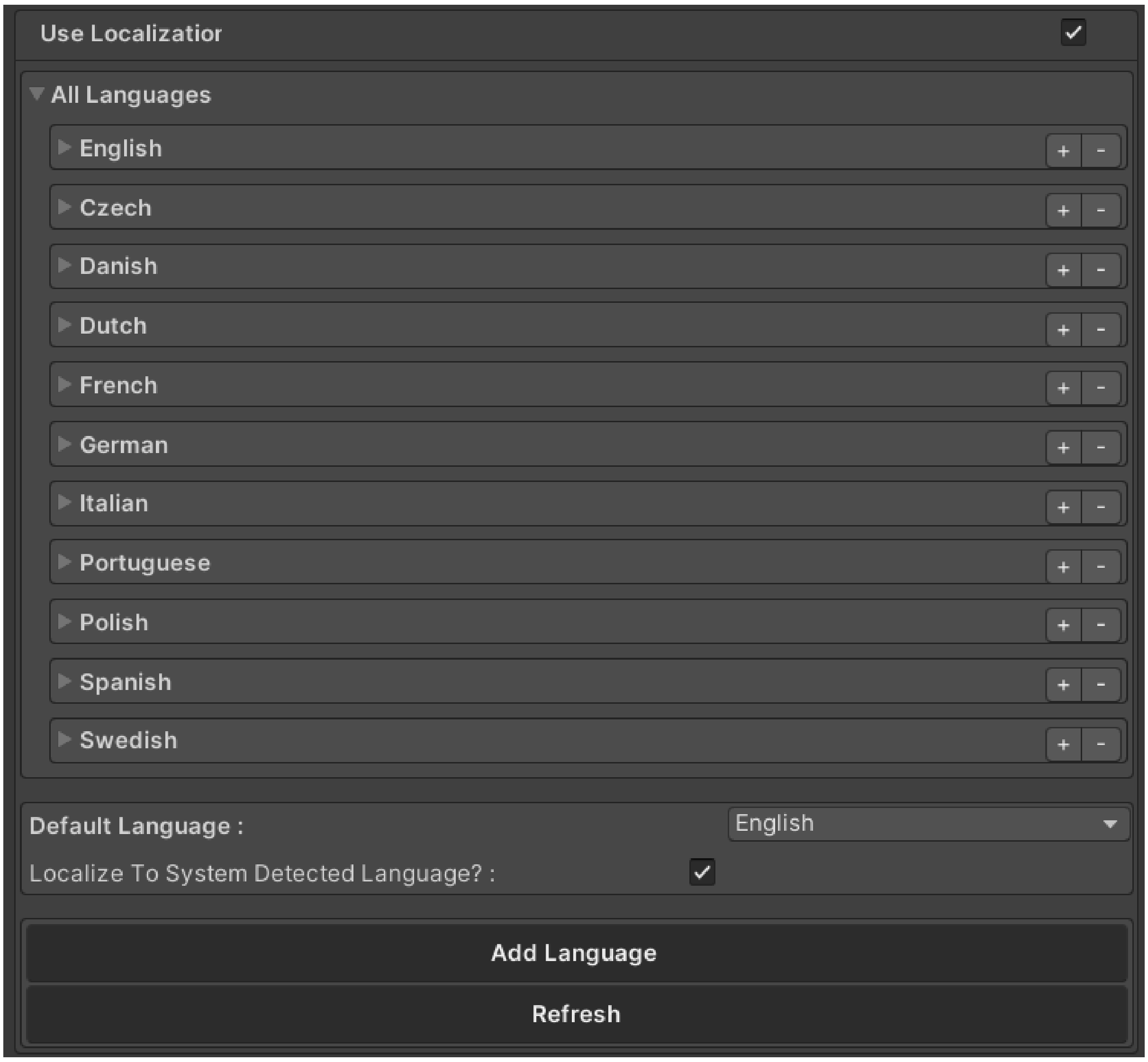


```
2 references
public Sprite GetThemeSpriteWithTag(string spriteTag) {
    return currentUITheme.spriteTags.FirstOrDefault(o => o.tagName == spriteTag).tagSprite;
}

90 references
public Sprite GetBlockSpriteWithTag(string spriteTag) {
    return currentUITheme.spriteTags.FirstOrDefault(o => o.tagName == spriteTag).tagSprite;
}
```

Localization Settings

The GamingMonks Game template comes with pre-installed localization integration. Currently, it supports 11 languages, but you can add as many languages as you want. Here's a guide on how to configure localization for your game.



Use Localizatior ✓

All Languages

English + -

Czech + -

Danish + -

Dutch + -

French + -

German + -

Italian + -

Portuguese + -

Polish + -

Spanish + -

Swedish + -

Default Language : English

Localize To System Detected Language? : ✓

Setting Localization
Open Localization Settings: Go to the GamingMonks > Localization Settings menu item inside the Unity editor. The interface should resemble the attached image.

Use Localization Toggle: If your app needs localization, turn this toggle on. If localization isn't required, turn it off, and there's no need to adjust additional settings.

All Languages Setup: After enabling the localization toggle, add as many languages as you want. Set up each language's Name, Display Name, Language Code, Flag, and localized string files. The string file consists of tags with text that is localized. Configure this for all languages. See the attached image for a small sample of string files. Note that the current localized content has been translated using Google Translate.

Default Language: Choose which language should be used as the default if the user hasn't selected a language and the system language isn't detected.

Refresh: Remember to press the refresh button after configuring each language to ensure it's applied to the app. Failing to refresh may lead to the unavailability of the language during gameplay.

Localize To System Detected Language: If you want the app content to be automatically localized to the device's default language (if it's supported and configured within the app), turn this setting on.

Configuring Localized Text File
In the GamingMonks game template, each language configuration is associated with a string-"LANG-CODE".xml file. This XML file should include a name attribute for the text tag and a text attribute which contains the text associated with the provided tag.

Here's an example of the XML structure:

```xml
<strings>
    <string name="welcome_message" text="Hello, player!" />
</strings>
```

Important: Do not change the format of the string file as the string file parsing code in the LocalizationManager.cs script depends on this format.

Consistency Across Languages
If there's any change in text or tag, be sure to apply that change to all language files. Likewise, if you add a new text to a string file, also add it to all string files for the configured languages.

Formatted Text
You can use formatted text. For example, if there's text that needs to insert a number or special text within the localized content, it can be easily done. It works in the same way as using String.Format("Level {0}").

```xml
<strings>
    <string name="level_message" text="You are at level {0}" />
</strings>
```

How to use Localization
The use of localization in GamingMonks is straightforward and requires minimal programming. Most controls are drag and drop, and there are several extension methods which are simple to use.

Note: Our localization tool supports Unity UI text only, as this game template runs only on Unity UI.

- **Localize UI Text**
  If any text needs to be localized automatically, please add LocalizedText.cs script component to it. Once the component is added, insert the associated tag with the

targeted text and it's done.For text that contains a number within localized text, please use LocalizedTextFormatted.cs script component.

- **Technical Info**
  The localization tool contains all methods and classes necessary to handle and use localization. Here are some useful methods in the LocalizationManager:

```
// Returns the current localized language configuration
LocalizationManager.Instance.GetCurrentLanguage();

// Sets the given language as active in-game language
LocalizationManager.Instance.SetLocalizedLanguage(currentButtonLan
guage);

// Returns localized text for the given tag
string localizedText =
LocalizationManager.Instance.GetTextWithTag("txtSuccess");
```

- **Using Extensions**
  The localization tool comes with some built-in extensions for localization. If there's any need to add or modify any extension, please refer to the LocalizationExtensions.cs script.

  Here are examples of how to use these extensions:

```
// Sets localized text with the given tag to the Text component
txtTitle.SetTextWithTag("txtGameOver_gridfull");

// Sets the localized text for the given tag to the text component
with formatting given as the argument
thisText.SetFormattedTextWithTag(txtTag, formattedValue1);
```

- **Callbacks**
  Callbacks are important to detect language initialization and changes, in order to handle code based on these changes. Here is a code snippet showing how to implement and use localization related event callbacks:

```
LocalizationManager.Instance.onLanguageChanged += (language) =>
{
    // handle the language change
};
```

  In this example, when the language changes, the specified code block will be executed. This is a powerful way to handle UI or game logic updates based on the current language setting.
- **Localization Supports Non-UI Texts?**

GamingMonks localization tool primarily supports Unity UI text. Direct support for TextMesh or TextMeshPro isn't built-in, but you can use `LocalizationManager` class to show localized content explicitly. If you often handle non-UI texts, consider adding more extensions and classes catering to your needs.

- **How to Change Language Selection Screen?**
  To customize the language selection screen, go to `Canvas-Popups -> SelectLanguage` GameObject in the Unity scene hierarchy. You can make modifications as per your needs. Additionally, review the `LanguageSelection` and `LanguageButton` script components for further customization. They control the logic for language selection in your game.

Game Initialization

At the start of the game, the Awake() function initializes the GamePlaySettings, levelSettings and AppSettings ScriptableObjects by loading them from the resources folder. These then can be used to get all the data and use them in the required script.

```
private void Awake() {
    if (gamePlaySettings == null)  {
                                                  gamePlaySettings      =
(GamePlaySettings)Resources.Load("GamePlaySettings");
    }
    if (levelSO == null) {
        levelSO = (LevelSO)Resources.Load("LevelSettings");
    }
    if (appSettings == null) {
        appSettings = (AppSettings)Resources.Load("AppSettings");
    }
    if (adSettings == null) {
        adSettings= (AdSettings)Resources.Load("AdSettings");
    }
    if (uiThemeSettings == null) {
        uiThemeSettings=
              (UIThemeSettings)Resources.Load("UIThemeSettings");
    }
}
```

Gameplay Code Flow

The following section depicts the entire code flow from entering the gameplay to the elements loading up in the gameplay scene.

Transitioning from Level Selection to Gameplay

```
public   void   LoadGamePlayFromLevelSelection(GameMode   gameMode,   int
levelNumber)
{
    // Deactivates UI components related to level selection
    // ... statements here ...

    // Activates UI components related to gameplay
    // ... statements here ...

     // Sets game-specific variables, such as the current level and game
mode
    // ... statements here ...

    // Restarts the game in preparation for new gameplay
    GamePlayUI.Instance.RestartGame();

    // Code for other logic setup...
}
```

The LoadGamePlayFromLevelSelection function is used to transition from the level selection screen to the actual gameplay screen. It deactivates the UI components related to the level selection and activates those related to gameplay. It sets up the required game variables, such as the current level and game mode, and then calls the RestartGame method from GamePlayUI to prepare for the new gameplay.

Restarting/Loading the Game

```
public void RestartGame()
{
    // Clears progress data
    GameProgressTracker.Instance.ClearProgressData();

    // Resets game data including board and shape data
    ResetGame();

    // Clears goals (targets) of the goal panel
    TargetController.Instance.DestroyTargetsOnReloadLevel();

    // Resets ad preferences
    AdmobManager.Instance.ResetAdPreferences();

    // Starts the gameplay
    StartGamePlay(currentGameMode);
}
```

The RestartGame method from GamePlayUI is called when starting new gameplay. This function takes care of clearing the old game data, resetting the game state including the game board and shapes data, and clearing the goals or targets. It also resets ad preferences using AdmobManager. Finally, it calls the StartGamePlay function to commence the gameplay with the current game mode.

The UIController script contains other functions as well to load gameplay from the tutorial or home screen, handling different situations as per the game's requirements.

Starting Gameplay

```
public void StartGamePlay(GameMode gameMode)
{
    // Preparatory actions for different game modes
    if (gameMode == GameMode.Level)
    {
              // Level-specific preparations: setting up moves count,
initializing targets, and more...
    }
    else
    {
         // Actions for other game modes: setting UI layering, resetting
game data if necessary...
    }

    // Specific actions for Time mode
    if (currentGameMode == GameMode.Timed)
    {
          // Enabling and starting the timer, optionally from previous
session's remaining time
    }

    // Checks if the there is user progress from the previous session
                            bool       hasPreviosSessionProgress       =
GameProgressTracker.Instance.HasGameProgress(currentGameMode);

    // Enables gameplay screen if not active
    if (!gamePlay.gameObject.activeSelf)
    {
        gamePlay.gameObject.SetActive(true);
    }

    // Generated gameplay grid
    gamePlay.boardGenerator.GenerateBoard(progressData);
    // Rest of the code...
}
```

The StartGamePlay function initiates the game according to the selected GameMode. For GameMode.Level, it sets up the game according to the level's settings and goals. For other game modes, it sets UI layering and resets the game data if necessary.

A specific block of code handles the GameMode.Timed, where it enables and starts a timer, and the timer can start from the remaining time of a previous session if any.

This function also checks for the existence of a previous game session, and retrieves the progress data if available. The gameplay screen is activated if not already active.

The gamePlay.boardGenerator.GenerateBoard(progressData); line is crucial as it generates the game board according to the progressData. It constructs the game grid based on previous session data if available, or generates a new one if starting a new game.

After initializing the gameplay, the StartGamePlay function calls some callbacks and analytics tracking, and shows initial game tips to the player.

Board Generation

The GenerateBoard function initializes a game board for the match-3 game. It takes in a ProgressData object, which is used to preserve the game state between sessions.

The board is created based on the current game mode and level settings. It uses a double loop to iterate over each cell in the grid, defined by the rowSize and columnSize variables. These loops create an instance of a Block in each cell, setting up its logical position, appearance, and other properties.

Within each iteration, the function first configures each block's default properties:

```
Block block = blockElement.GetComponent<Block>();
block.gameObject.SetActive(true);
block.SetBlockLocation(row, column);
blockRow.Add(block);
block.assignedSpriteTag = block.defaultSpriteTag;
```

When the game mode is Level, the function additionally modifies each block according to the properties specified in the level configuration:

```
if (GamePlayUI.Instance.currentGameMode == GameMode.Level)
{
    //... Block modification code
}
```

The modification code reads data from activeLevel.rows[row].coloum[column], where activeLevel is the current level, and adjusts each block's properties accordingly. This can include the block's sprite type, stage, remaining counter, and secondary sprite.

Once all blocks have been configured and added to the game grid, the function calls GamePlay.Instance.OnBoardGridReady(). This function completes the setup of the game board:

```
public void OnBoardGridReady()
{
    int totalRows = allRows.Count;
    for (int rowId = 0; rowId < allRows[0].Count; rowId++) {
         List<Block> thisColumn = new List<Block>();
         for (int columnId = 0; columnId < totalRows; columnId++) {
             thisColumn.Add(allRows[columnId][rowId]);
         }
         allColumns.Add(thisColumn);
    }
}
```

This method is responsible for creating an organized list of columns, making it easier for the game logic to process each column in the grid. This function does so by iterating over each row and column in the grid, creating a list of blocks for each column, and adding each column list to allColumns.

After this, depending on the current game mode and level, GenerateBoard may enable additional features, such as a conveyor belt or jewel machine. It may also place bombs and blockers on the board.

Finally, if the ProgressData object provided at the beginning is not null, the function restores the game board to the previous state. This involves setting the status of each block and placing any bombs that were present in the previous session. This allows players to continue a game they previously left.

In summary, GenerateBoard is a function that generates a game board based on various settings and optionally a previous game state. It iterates over each cell in the grid, configuring each block individually, and sets up additional game features if necessary. The game board is ready to use once GamePlay.Instance.OnBoardGridReady() is called.

Block Shape Generation

The board generator also calls the PrepareShapeContainer(ProgressData progressData) function.This function prepares the block shape containers. Each container is a UI panel that holds a shape to be used in the gameplay.

```
public void PrepareShapeContainer(ProgressData progressData)
{
      //Preparing Shape Pool
      //Preparing Color Pool
      //Preparing Upcoming Shapes
}
```

Function Variables:
ProgressData progressData represents the progress data of the game. If progress data is not null and the game mode isn't Level mode, the function fills all the shape containers with the current shapes from the progress data.

Function Execution:
Preparing Shape Pool: This function calls PrepareShapePool(), which prepares a block shape pool with a given probability amount as logical blocks shape references based on gameplay settings. It takes into account various gameplay settings such as the current game mode, whether to pick block shapes from this level, and whether standard or advanced shapes are allowed.

```
void PrepareShapePool()
{
    // Logic for pool preparation based on different conditions
}
```

Preparing Color Pool: Next, PrepareColourPool() is called. It prepares a pool of colors to be used for the shapes. If the game settings allow picking colors from this level, it also prepares an upcoming color pool.

```
private void PrepareColourPool()
{
    // Logic for color pool preparation based on different conditions
}
```

Preparing Upcoming Shapes: The PrepareUpcomingShapes() function prepares a pool of upcoming shapes which will be next in line to fill in the shape containers after the current ones are used.

```
void PrepareUpcomingShapes()
{
     // Logic for upcoming shapes pool preparation based on different
conditions
}
```

Filling Shape Containers: Lastly, the function checks if there is any progress data. If there is, and the game mode isn't Level mode, it fills all the shape containers with the shapes from the progress data. If there isn't any progress data, it just fills all the shape containers.

```
void FillAllShapeContainers()
{
    // Logic for filling shape containers
}
```

```
void FillAllShapeContainers(ShapeInfo[] currentShapesInfo)
{
    // Logic for filling shape containers based on current shapes info
}
```

By the end of this function execution, the shape containers will be ready for gameplay with the appropriate shapes based on the progress data and gameplay settings.

Level Mode Goals Setup

The Goals Setup in the goals panel at the top of the board is handled by the targetController.cs Script.

Initializing Targets (InitializeTargets(LevelGoal[] goals)): This function creates and sets up the game's targets based on an array of goals passed to it. Each goal results in a new target instantiated from a predefined targetPrefab. If any goal's sprite type matches AllColourBlock, the canCollectAnyEmptyBlock flag is set to true. The instantiated targets are then added to the targets list.

```
// Instantiating each LevelGoal as a Target object and adding to the targets list.
Target target = Instantiate<Target>(targetPrefab, gameObject.transform);
...
targets.Add(target);
```

Updating Targets (UpdateTarget(Transform block, SpriteType spriteType)): This function checks for and updates the appropriate target when a block of a certain spriteType is acted upon. If the block's sprite type matches any target's sprite type, that target's count is updated. If all targets have been completed (checked by verifying if the targets' count is less than or equal to 0), touch inputs are disabled to prevent further interactions until necessary. It also calls SpawnDestroyingBlock function, which likely creates an effect or action when the block corresponding to the target is destroyed.

```
// Checking if the sprite type of the block matches with any target's sprite type.
if(spriteType == target.spriteType)
...
// Updating the target count.
target.UpdateTargetCount();
...
// If all targets have been achieved, touch is disabled to prevent unwanted actions.
InputManager.Instance.DisableTouch();
```

Destroying Targets (DestroyTargetsOnReloadLevel()): This function destroys all target objects when the level is reloaded. Both active and completed targets are destroyed and their respective lists are cleared. It also sets the isAllTargetsCollected flag to false, indicating that the targets are yet to be collected for the reloaded level.

```
// Destroying each Target object in the targets list.
Destroy(target.gameObject);
...
// Clearing the targets list and resetting the isAllTargetsCollected flag.
targets.Clear();
...
isAllTargetsCollected = false;
```

Overall, these functions ensure the dynamic creation, updating, and destruction of targets based on player actions and level changes. There are additional functions not shown here that handle animations related to these targets, providing visual feedback to the player. These could include animations for target completion, destruction of the corresponding blocks, and perhaps an animation or effect when all targets are collected.