mincode Documentation

Contents

Introduction

The mincode package was created in order to save game development time and improve the product that we develop.

It's intended to improve clarity between different teams (also non tech teams) and to generalize important patterns that makes our apps run smoother and for us to develop with ease.

The motto is to work with this infrastructure as much as possible, and write new code as least as possible, as code is mostly the longer path to solve an issue. Code is also something that only a person that knows how to code can do. With this package – you can make simple logical operations non coders friendly.

In addition to the general variables and events, this package contains common game development use cases solutions (levels system, scene loading system, basic mobile UI, useful scripts). Completely generic and ready for you to focus on your content, rather than writing it on your own.

This document contains mentions of these Unity features:

- ScriptableObject: scripted data container
- UnityEvent: a set of actions. Looks like this in the editor:

For any bugs, comments, or anything – please contact reytangames@gmail.com

Check out our AppStore page - our games were quickly developed using this infrastructure and its concepts.

Inspired by Ryan Hipple's lecture, I cannot stress this enough – this guy changed my perspective on game dev completely, I strongly recommend you to watch it. Before developing MinCode, I watched it 5 times to make sure I understood every part of it.

We develop hyper casual games as side projects, in addition to a full time job, one dev one artist. We execute our games using this package – it usually takes us about 40 hours of work total (depending on the game's complexity).

Setup

To start, easily open the scenes provided in *MinCode/Scenes*. Learn the concepts that are in this guide and apply them to your game. Understand the connections between the moving parts. Hit play and play around to connect the dots of your learning curve.

It will change the way you develop games forever!

Events System

Turn your game into a smart event driven system. Rather than checking for variables values in Update, or using static C# events, or any other solution that wasn't generic enough to be shared among different scenes – you can now use this events system provided in the MinCode package.

Easily create, modify, extend and use. The event system is based on Unity's ScriptableObject and UnityEvent.

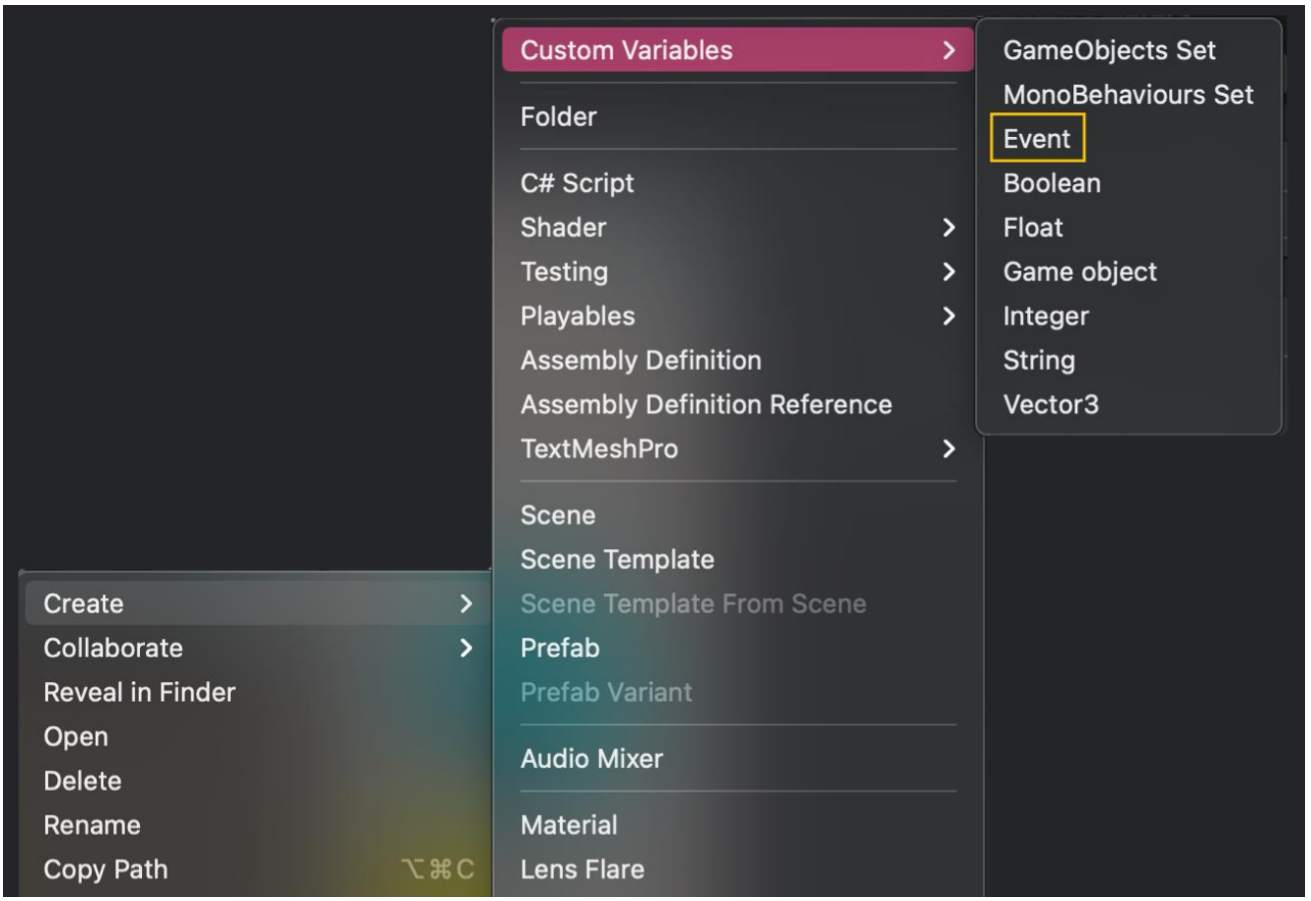
Components in this system:

- *GameEvent* : ScriptableObject
- *EventListener* : MonoBehavior
- *EventRaiser* : MonoBehavior

GameEvent

This is the event itself – the one that is raised, and lets all the listeners know that it happened.

The *GameEvent* can be create from *menu > Custom Variables > Event*



After creating, we can:

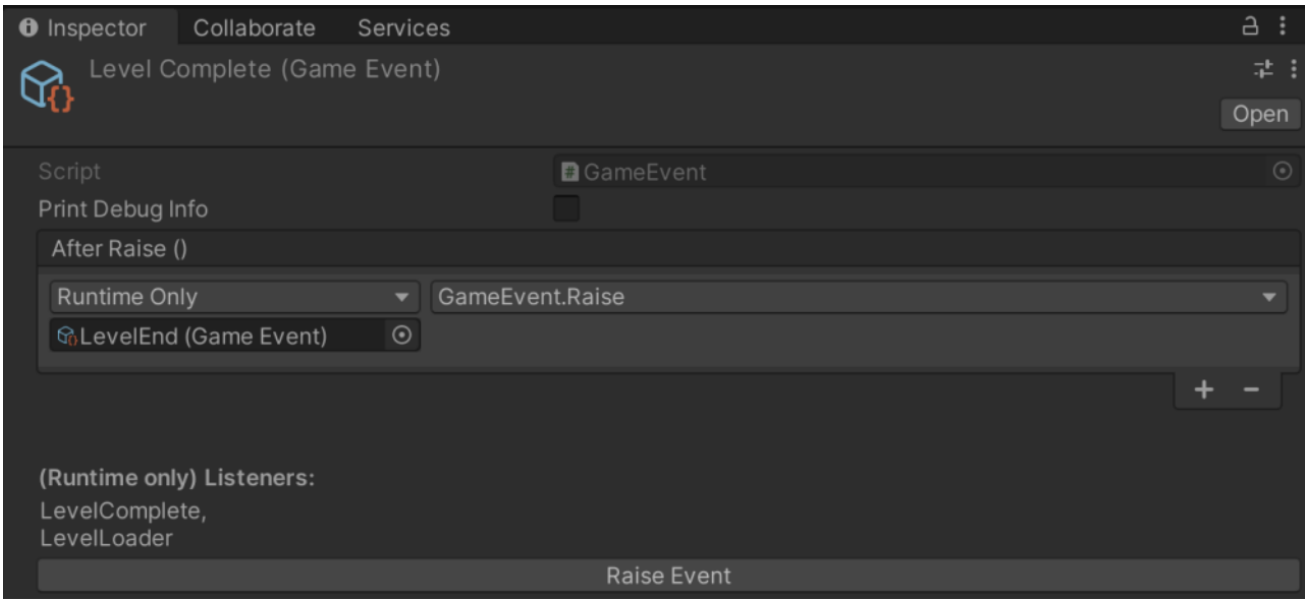
- *Print debug info* – will print to console when raised, listener added, listener removed
- Add a pipeline of actions to happen after raised - using UnityEvent, we can execute actions listed in *AfterRaise ()* section
- See listeners (runtime only) - the object names of all the listeners that will be notified once this event is raised
- Manually raise it (runtime only) - a button that allows you to demo that event's raising. This can

save you a lot of time – rather than reproducing a situation over and over again, simply raise the event and see how its listeners behave

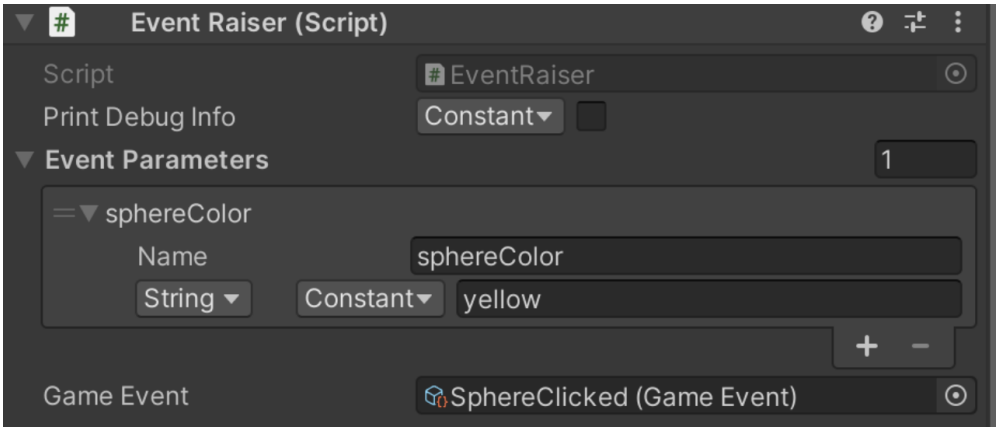
- Pass event parameters (optional)

In this example, we used the LevelComplete event (included in the mincode package, at *Metadata/Events/Level/LevelState*).
This event also raises the *LevelEnd* event, as once a level completes we want to behave differently than if a level failed, but the steps when the level ends are common to both of them. For this reason, both *LevelFail* and *LevelComplete* events raise *LevelEnd* after it is raised.

Its listeners are *LevelComplete* – which plays a sound once this event is raised, and *LevelLoader* – that increases the player's level number (see in this doc – *Level system*).



To raise an event with parameters, you can raise it from the editor using *EventRaiser:*



or you can raise it from code, using any of these methods:

- *GameEvent.RaiseWithParameter(string name, object value)* - for simple cases where there is only one parameter. For example *gameEvent.Raise("color", "yellow");*

- *GameEvent.Raise(IEnumerable<(string name, object value)> eventParameters)* - for cases

where you want to pass multiple parameters.
For example:

```
var parameters = new List<(string name, object value)>
{
   ("color", "yellow"),
   ("scale", 1)
};
gameEvent.Raise(parameters);
```

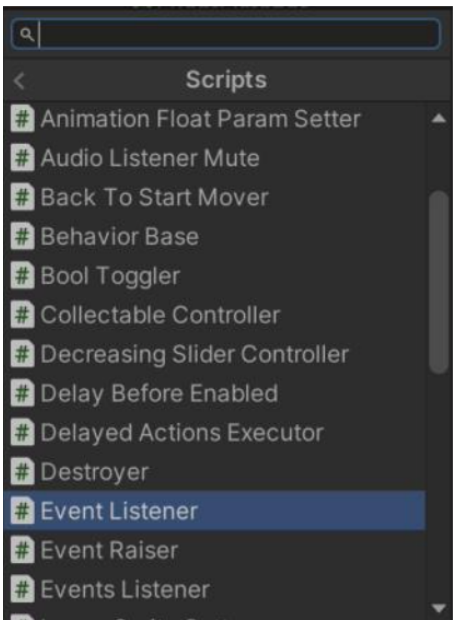
- *GameEvent.Raise(IDictionary<string, object> eventParameters)* - for cases where you want to pass multiple parameters, and can set the dictionary by yourself. It is the same as *GameEvent.Raise(IEnumerable<(string name, object value)> eventParameters)*, but less convenient to use for non-coders.
  For example:

```
var parameters = new Dictionary<string, object>
{
   { "color", "yellow" },
  { "scale", 1 }
};
gameEvent.Raise(parameters);
```

EventListener

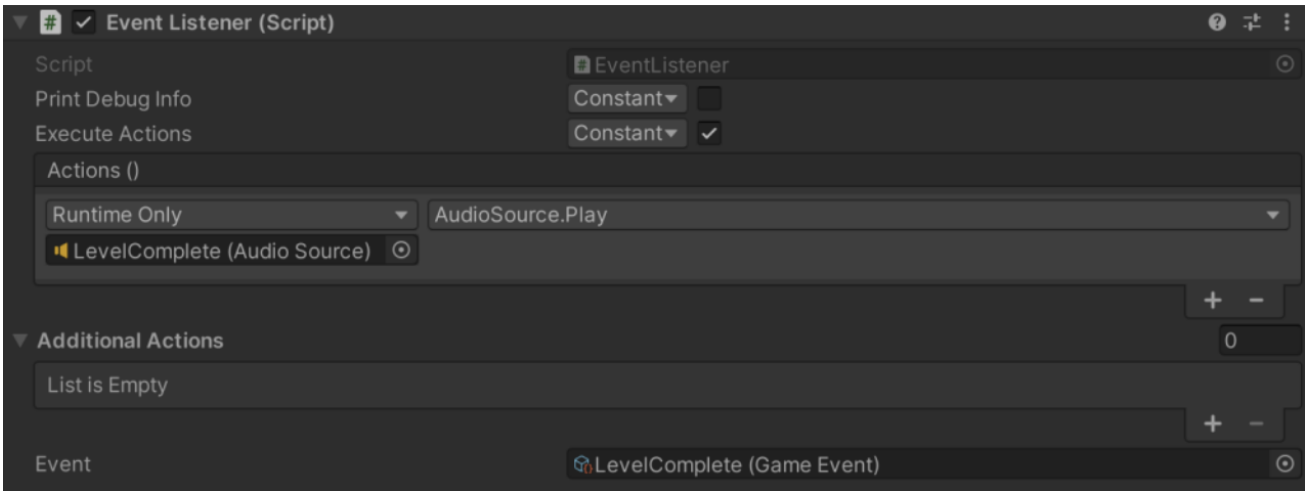
A simple script that registers to an event (add itself as a listener). After the event is raised, it executes a set of actions using UnityEvent.

In order to add an *EventListener* to a game object, simply *Add Component > Event Listener*:



After adding, fill in the Actions () that should be executed once the event is raised, and fill the Event that you want to listen to (the event that we created as a ScriptableObject).

Notice that one event may have many listeners – you don't have to execute all the actions all in one place. If an event affects components that are not logically tight, you can listen to those events in various objects located in different scenes.



In this example, we see the other side of the example we saw for the LevelComplete event. This is the listener, that once the *LevelComplete* event is raised, will play a sound, by calling *AudioSource.Play* (given from Unity's AudioSource. As you can see, we can call methods that are on the component's script using UnityEvent). This example is in MinCode's *Audio scene* > *LevelComplete* game object.

You can access the raised event parameters from your code, using: *EventListener.GetEventParameter<T>(string parameterName, IDictionary<string, object> parametersByName)* . check out the *SphereColorText* script to see an example of a script that changes a text component to print the clicked sphere's color.

```
public void OnEventRaised(IDictionary<string, object> parametersByName)
{
    const string SphereColorParameterName = "sphereColor";
    var sphereColorValue = clickedEventListener.GetEventParameter<string>(SphereColorParameterName, parametersByName);

    TextUi.text = $"{sphereColorValue.ToString().ToUpper()} colored sphere was clicked!";
}
```

Variables and References

Variables

mincode uses a completely generic implementation for scriptable object variables.

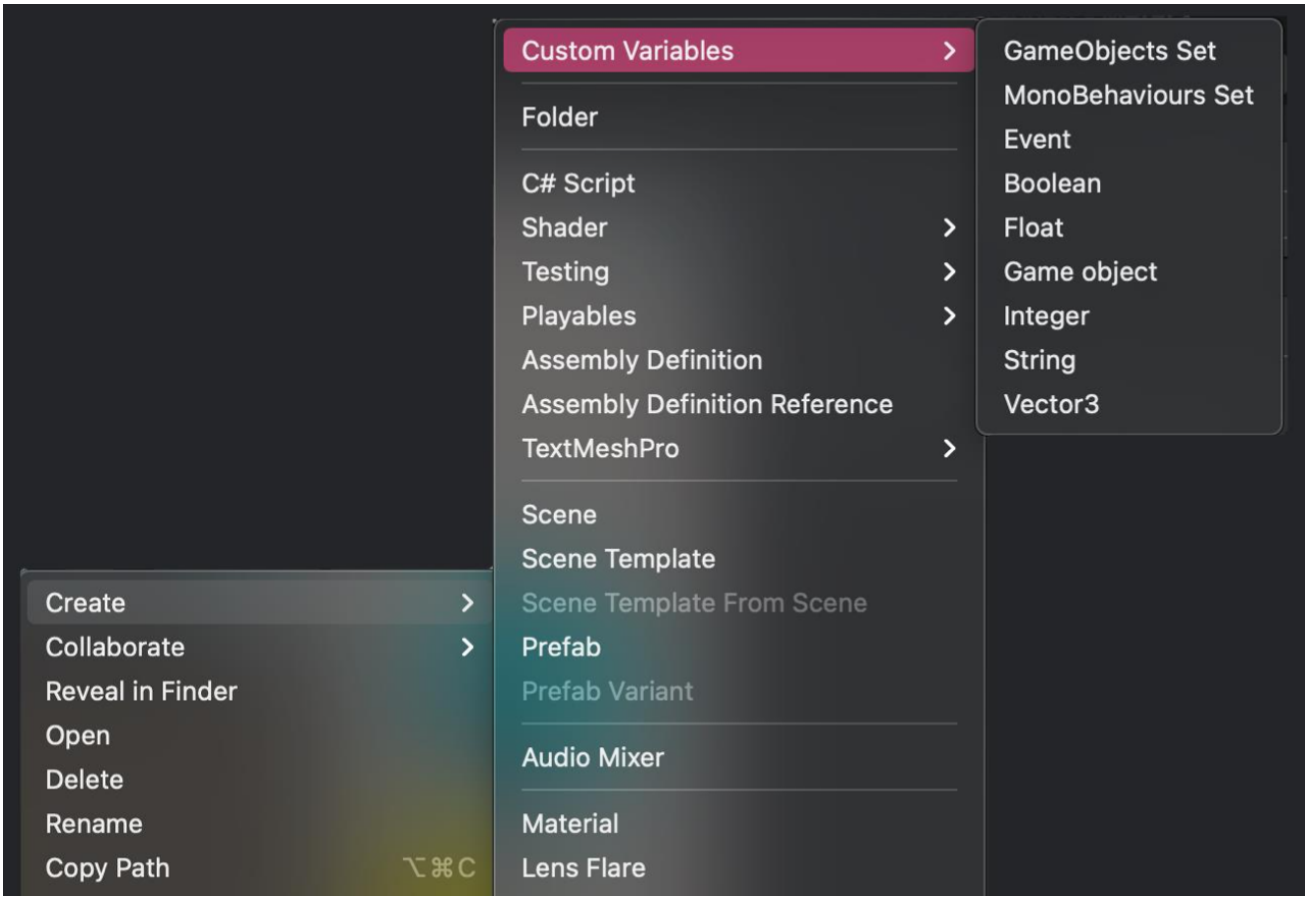
This feature is a game changer – you can now share data across multiple scripts and components. No need for that huge singleton class that holds data for you. No need for a distributed cache. No need for dependencies that are just wrong – from now on, you can have a loosely coupled architecture game, and work with variables that are small pieces of data inside Unity.

mincode offers these types of variables:

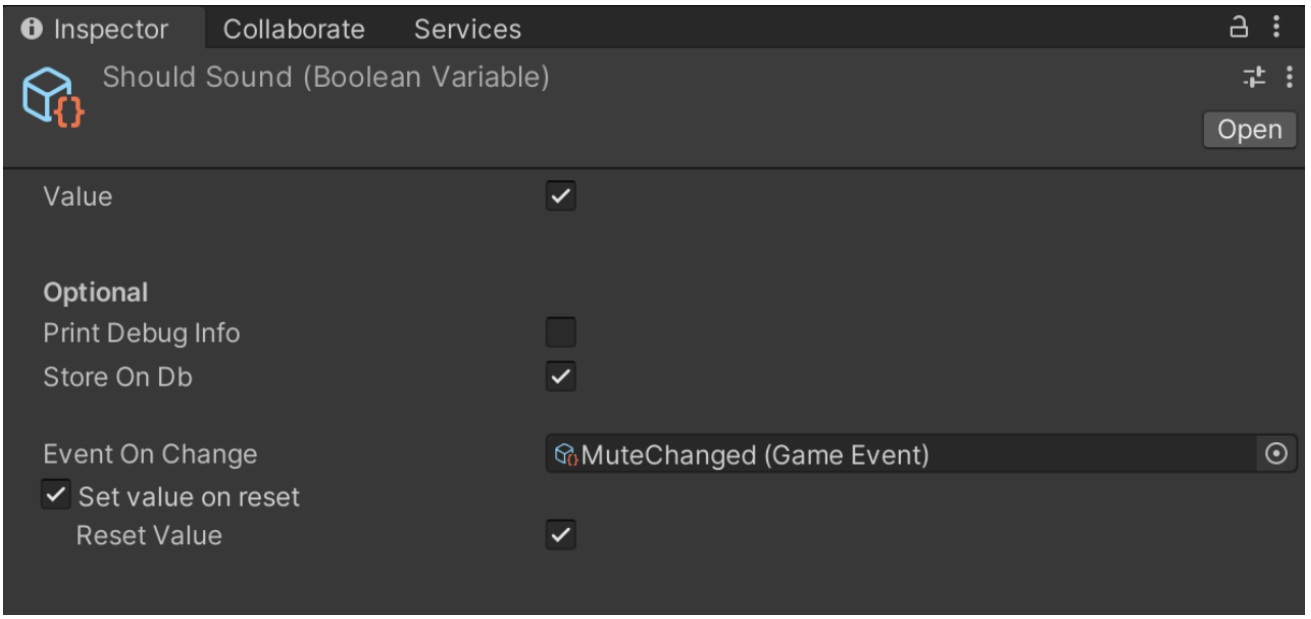
- Integer
- Float
- String
- Vector3
- Boolean

- GameObject
- GameObject runtime set
- MonoBehavior runtime set
- An abstract implementation that you can easily inherit and have whatever variable of type that you need! You can find it in *Logic/Scripts/ScriptableObjects/Abstract/VariableScriptableObject.cs*

Variables are easily created via the menu, at the same place that creates events:

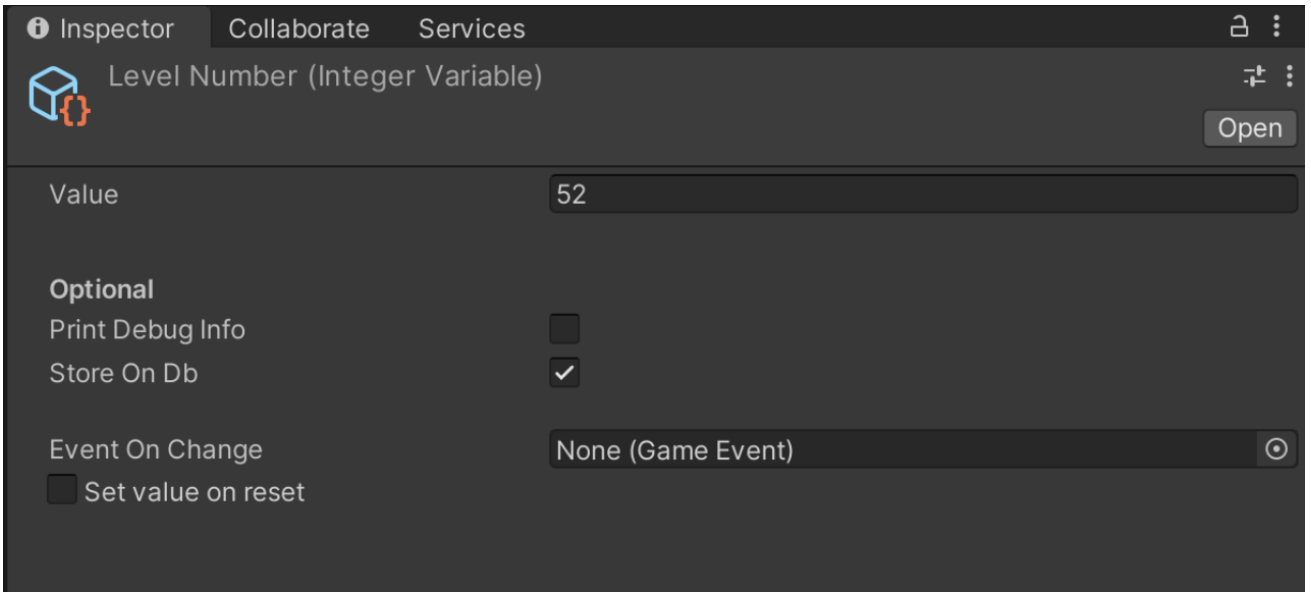


mincode offers custom editors (completely generic! Once you extend a variable, you can easily inherit from the generic editor to have one of your own). You can find them in *Logic/Scripts/Editor*. Here's an example for a variable, in this case we will use the ShouldSound variable, that saves for us a variable that indicates if we should play sounds, or that our player wants to mute them.
This variable is also saved to *PlayerPrefs*, so that the next time the player opens our game, he won't need to turn off the sound again. And also – it has a reset value of true, so that by default, sounds will be played, unless the player turned it off (all provided by mincode – also generic!)



Let's explain each part in this editor window:

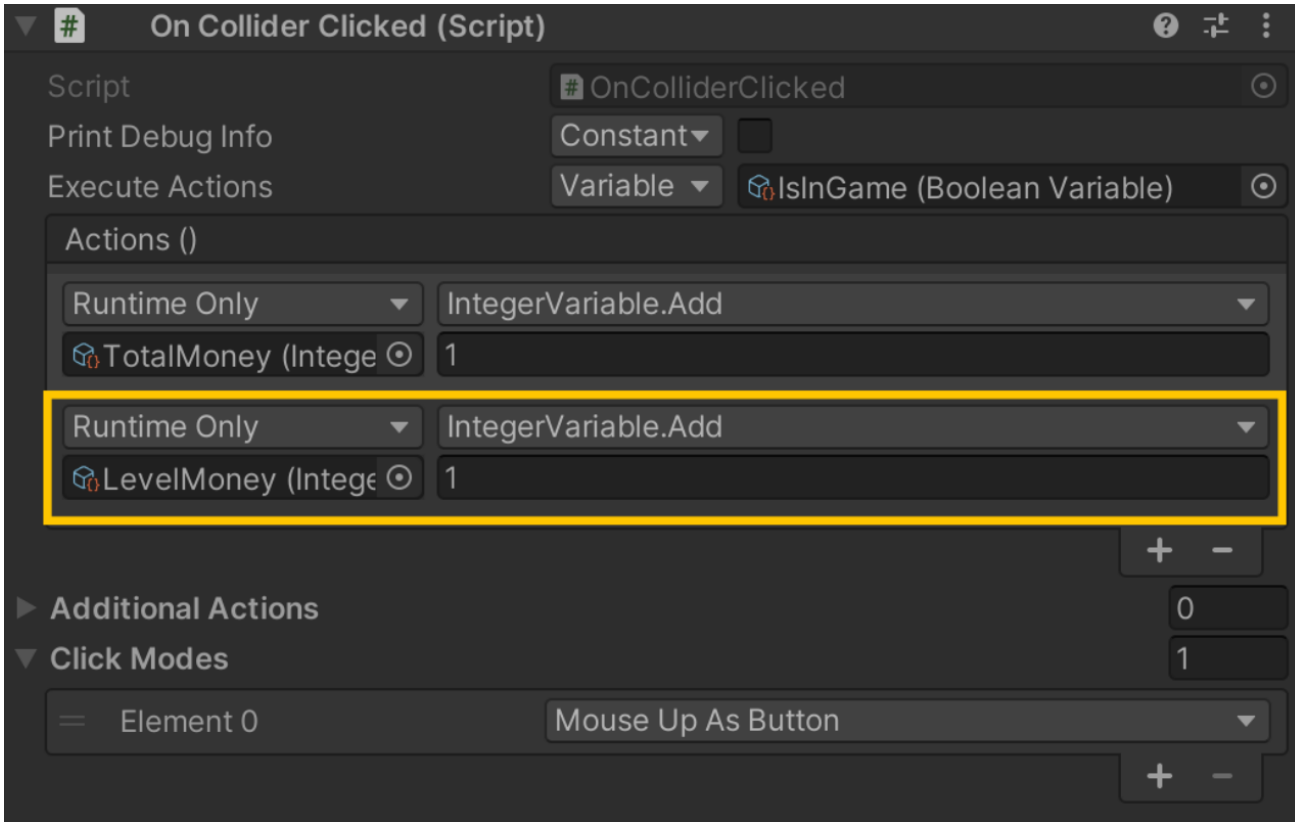
- **Value:** the current value of this variable. Every component that reads this variable will get this value. It won't always be a checkbox, and will change according to the variable's type. For example, for integers it will hold an integer value. *LevelNumber* integer variable inspector:



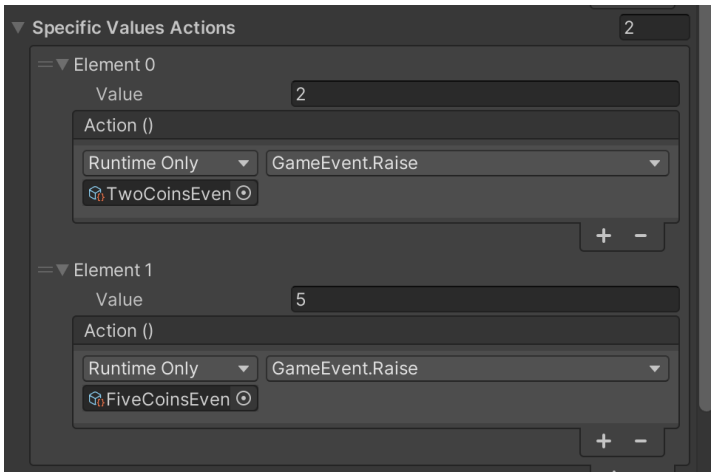
- **Print Debug Info:** integrated debug for the ease of dev – when true, it will print to the console once updated and when reset
- **Store On Db:** Save this variable to Unity's *PlayerPref*. Notice that Unity supports Integers, Floats and Strings, so if you want to save more complex things to the *PlayerPref* then it needs to be manually coded
- **Event On Change:** a *GameEvent* that is raised once the variable is changed. You can easily make a script respond to a variable change by using the extensions method *CreateChangeEventListenerIfThereIs*. For example, see MinCode's *BoolToggler* script. Each time the boolean *toToggle* variable is changed, the script will execute the *Refresh* method:

```
toToggle.CreateChangeEventListenerIfThereIs(gameObject, BooleanReference.Create(false), Refresh);
```

- **Set value on reset:** Each time that this variable gets a reset, we will set it to the given value (when loaded for the first time in a session or when manually resetting (notice that if *Store On Db* is checked, it will only get a reset if it's not saved in the DB already)).
- Some of the variables also support built in arithmetic methods. For example, when a coin is collected and we want to increase player's money counter, we can easily do something like this:



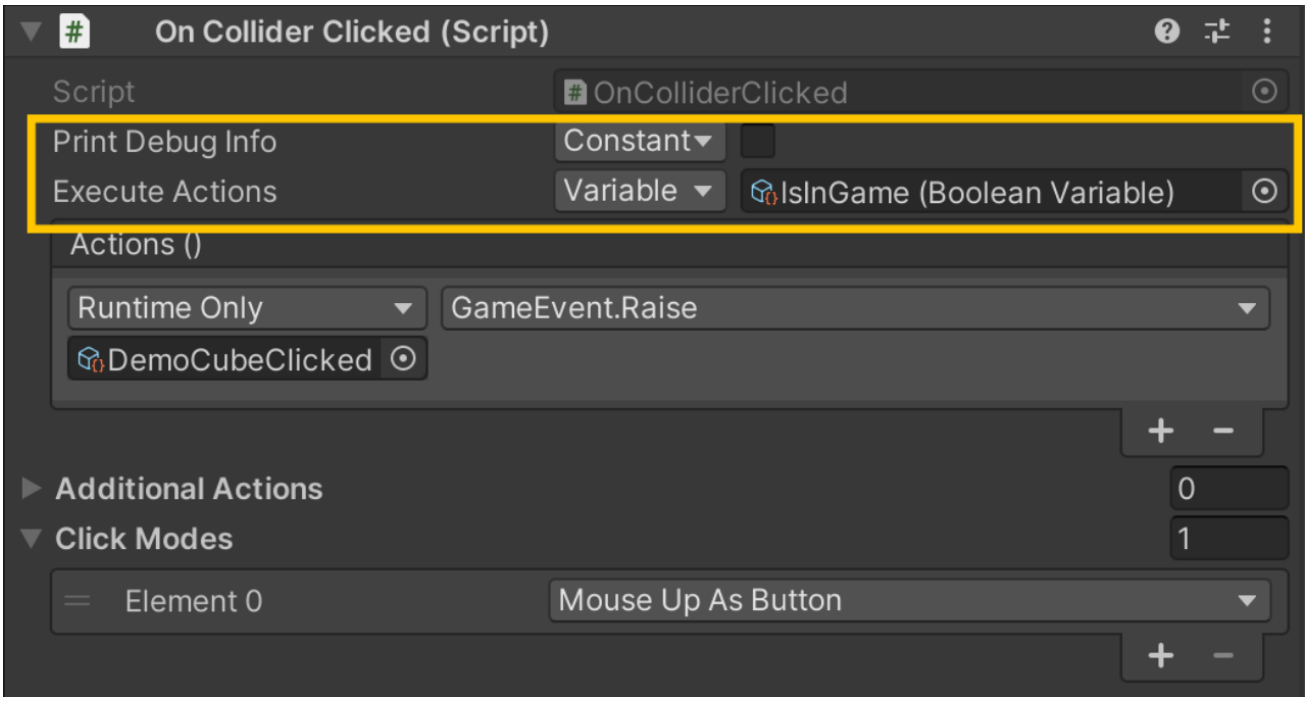
- **Specific Values Actions:** You can set specific actions for specific values. For example in this case, when the value is 2, we will fire an event. When the value is 5, we will fire another event:



References

MinCode's reference uses its variables, and lets the user choose to use a constant or a variable. It has a custom property drawer, so each time you use a reference, you can easily select if you use a MinCode variable or a constant. This one is also completely generic for you to extend.

Let's look at an example, this time a script that executes actions once a collider was clicked (also provided in this package, this example is from *Level/LevelHolder/Level1/Cube*):



As you can see, *Print Debug Info* is marked as a **constant** – that means that it doesn't need a variable to be set, we can control it directly from the editor. It doesn't depend on a variable.

On the other hand, see *Execute Actions* which is treated as a **variable**, and will read IsInGame boolean variable's value to get the value. This allows this script to easily ignore clicks on game objects if we are not in game.

In our games, we will mostly use references in our scripts – as it's easier to control. Many times when starting development it's easier to work with a constant, and then when development progresses we want to work with a variable. Once changed – our code remains the same. The reference does the hard work of resolving the value for us.

References are members in your script's class. To use them, simply declare them as any other member. For example:

public BooleanReference executeActions;

You can easily set a default value for those variables as well, like this:

public BooleanReference executeActions = BooleanReference.Create(true);

Debugging System

mincode offers some base classes that you can inherit from and use. The two most basic are

- *BehaviorBase* : MonoBehavior
- *ActionsExecutor* : BehaviorBase

Both provide a *PrintDebugInfo(string)* method, that you can call to in your scripts. This method will check if the *printDebugInfo* variable is true / false, and will print the given debug string or will ignore it.

This way, you can easily turn on and off debugging of a component.

ActionsExecutor is also handed with a built-in UnityEvent that has actions that should be executed once *ExecuteActions* is called. It also provides AdditionalActions, that is a dictionary, that has a string key (identifier for the action) with a value of UnityEvent.

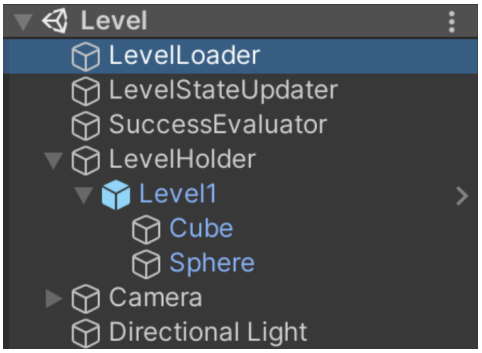
Using the *AdditionalActions* you can easily execute more action sets of UnityEvents without any additional code! Simply define them and invoke them by calling the *ExecuteAdditionalActions(string key)* method from UnityEvent or from your script.

Level System

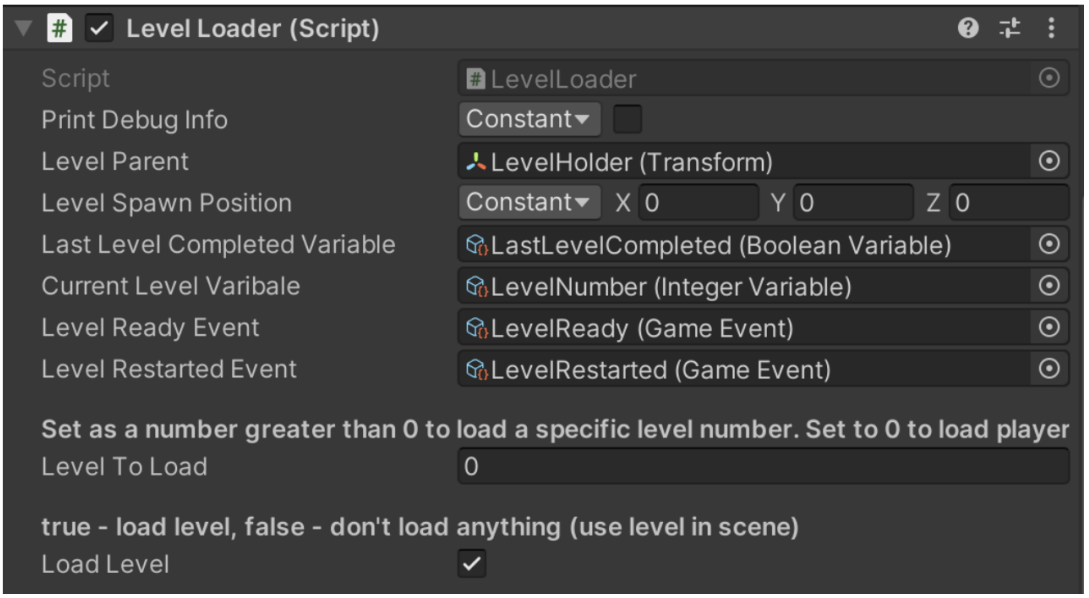
MinCode provides a solution for dealing with levels. It has an easy to implement a solution that loads levels infinitely for you, and offers built in progression and level restart.

The levels in this system are prefabs. The magic happens in a script called *LevelLoader*.
Let's see an example from this package's sample project in the Level scene.

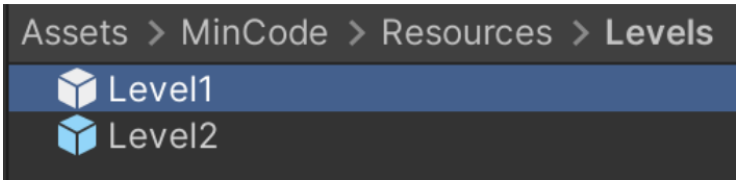
- Hierarchy:



- Inspector:



- Path for level prefabs:



The *LevelLoader* script will read the *LevelNumber* variable, and will load the player's current level. Once the player will finish all the levels available, it will go back to the prefab of Level1, yet the *LevelNumber* will keep increasing.

This is valuable in case you develop a demo that only has a few levels, but can still let you see the player's retention, as he still sees progression.

LevelLoader also gives you an option to easily play a specific level, using the Level To Load property. Simply give it the level number to play and it will play it for you.

You can also create levels that you want to test that are outside of the levels sequences, and mark LoadLevel as false – so it will skip the level loading. This is helpful when testing a movement mechanic, with a tailored level that can provide easy testing when developing.

Levels as prefab are extremely comfortable, as you can easily create and modify them. It also allows you to separate between logics that will happen only on a specific level, and logics that are common and should always run in each level. You can also set the camera differently on each level using this mechanism. It's very versatile for you to choose what will serve you best!

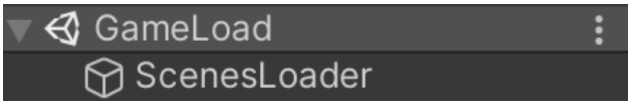
Another practice that helped us a lot is to use Unity's prefab variants. This way you can have common practices in the parent prefab, and level specification in the variants.

Scene Loading System

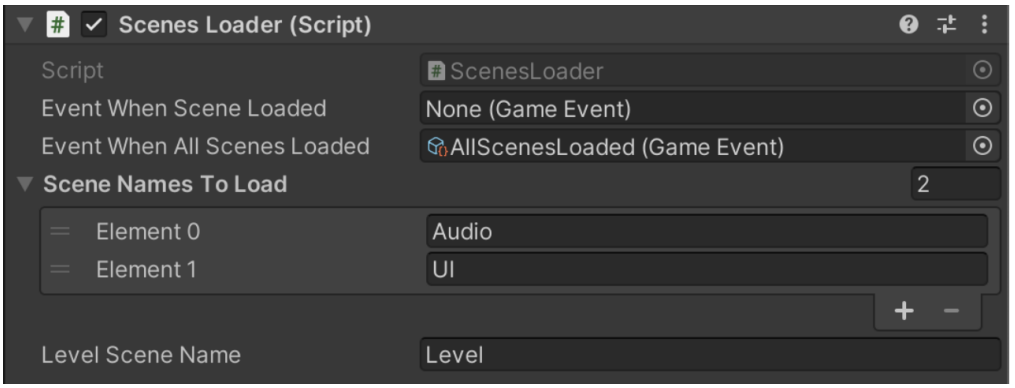
MinCode provides a solution for scene loading. It is very important to separate different parts of your game into different scenes. This way you prevent A LOT of GIT / collab conflicts, and reduce “noise” from logics that you don’t always care about. For example, when working on levels, we don’t always care about the audio, and not always want it to distract us.

Let’s see how it works.

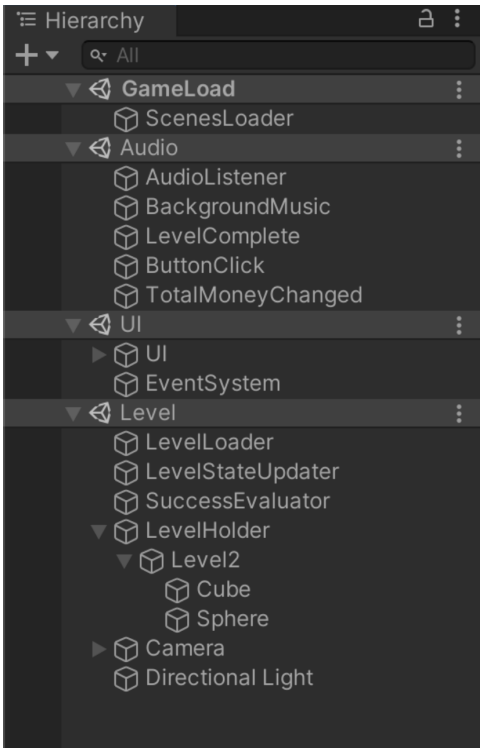
- Hierarchy:



- Inspector:



Scene Names To Load is the list of scenes that you want to load, and it will load them in the given order. The Level scene name is special, as once it’s loaded it means that we are ready to go and let our player start. It will also be the active scene in our game.
Notice that all scenes provided to the SceneLoader should be added to the build in Build Settings. In mincode’s provided demo, we use these scenes when the game runs:



This is an important concept – we try to separate the moving parts of our game as much as possible. We try to decouple them as much as we can. In our games, we also have scenes for vibration (listens to events and vibrates accordingly), metadata update (updates variables in response to events), integration (to integrate with 3rd party SDKs) and many more.

In this example, we have these scenes:

- **Level:** holds the level and the components that are related to it.
- **Audio:** plays sounds in response to events. For example, when a level is complete
- **GameLoad:** This scene is in index 0 in the build and it's the first to load. It will load for us the other scenes, using *ScenesLoader*. If you have objects that have the DontDestroyOnLoad property, you can locate them here.
- **UI:** responsible for UI only. When working on levels, we don't always want the UI on. It's • very comfortable that we can load this scene only if we need to work on our UI.

Extra scripts in MinCode

MinCode also provides lots of scripts that are commonly used in games. Feel free to use them, and get inspiration on MinCode's concept from them – have many reusable scripts, so you can develop with ease, writing as little code as possible. We try to write generic scripts that will provide solutions for other future challenges that we might have.

The provided scripts are for the most common use cases – resolving world touch position (*TouchInput*), execute actions when a collider was clicked (*OnColliderClicked*), execute actions when a mouse action happens (*OnMouseInputAnywhere*), execute actions when colliding (*OnCollisionTriggerAction*) and many more!

Provided scripts:

- Entities/
  - **ActionOnCollisionTrigger:** Executes an action once a collision happens.
  - **ActionOnDisable:** Executes an action once a game object is disabled (using OnDisable).
  - **ActionOnEnable:** Executes an action once a game object is enabled (using OnEnable).
  - **ActionOnStart:** Executes an action once a game object's Start.
  - **AnimationFloatParamSetter:** Sets an animator float parameter. For example, if you want to change the speed of an animation at runtime.
  - **BackToStartMover:** Moves an object back to where it spawned, and back to its starting rotation, from when it was awaked. It uses Lerp to do that, so your game will feel smooth.
  - **BoolToggler:** Toggles a boolean variable, and executes actions according to its value. For example, if we want to respond to the *ShouldPlaySounds* variable. When true, we want to set the game's volume to 1, and when false, to 0.
  - **CollectableController:** A controller for objects that can be collected (a coin for example). It also provides an option to play particles that will be played once collected, and lets you execute custom actions after collection (like playing an animation).
  - **DelayedActionsExecutor:** Executes actions after a given amount of time, using Unity's coroutines.
  - **Destroyer:** Destroys a game object and executes actions afterwards.
  - **EventRaiser:** Raises a game event (we use it mostly on animations, to raise an event once an animation is done).
  - **LookAt:** Look at a target game object, by changing the transform's forward. The target game object can be changed at runtime, and can even be a GameObject variable.
  - **ParticlesPlayer:** Plays particles from a prefab.
  - **RigidbodyController:** Adds force or torque to a rigidbody

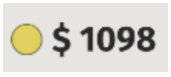
- Extensions/
  - **EnumerableExtensions:** Use the provided methods to better work with your IEnumerables.
  - **FloatExtensions, VectorExtensions:** Implements *GetString()* method that returns a string with 5 digits precision.
  - **ParticlesExtensions.PlayIndependent:** Play particles without depending on a game object. It's a good solution for cases where you play particles after an object is destroyed

- Interactors/
  - **OnClickOutsideCollider:** Executes an action when there is a mouse interaction outside of the game object's collider (mouse up / down / up as button).

    - **OnColliderClicked:** Executes an action when a game object collider interacts with the mouse (mouse up / down / up as button).
    - **OnMouseInputAnywhere:** Executes an action when there is a mouse interaction anywhere.
    - **ScreenEdgesResolver:** Returns world's position screen edges.
    - **TouchInput:** A static class that provides methods for resolving touch interaction
        - *GetInputRay():* Creates a Ray from the input's position. Useful for touch interaction with objects.
        - *GetWorldInputPosoition(float z):* Gets world position of current touch.
        - *GetScreenInputPosoition(TouchPhase? phase = null)*: Gets input position of the player, and handles multiple touch positions, sticking with the first one.

- Level/
    - **SuccessEvaluator:** Evaluates level money and stars, and saves it to variables.

- UI/
    - **DecreasingSliderController:** Implements a slider that its value decreases as time goes by, and increses when *Increase()* is called. Can be easily called by *OnMouseInputAnywhere* script to increase in each player's touch.
    - **DelayBeforeEnabled:** Disables a game object on enable, and after a given amount of time activates it. Useful if you have a button that should appear after a few seconds.
    - **ProgressionUpdater:** Updates a progression slider. Can use variables to resolve its values, so you can update a variable on the *Level* scene and easily reflect it in the *UI* Scene.
    - **RandomTextSetter:** Gets a list of strings, randomly picks one, and changes the TextMeshPro's text. Useful when you want to compliment the player with various texts, or when you want to display a random emoji. See an example in MinCode's *UI/GameState/PostGame/ScreenComponents/Background/RandomCompliment.*
    - **TakeScreenshots:** Captures the editor's game image.
    - **TextWrapper:** Wraps a variable in a given prefix and suffix. Updates once the variable is updated if this variable has a change event, without you manually updating it. See an example in mincode's demo UI scene at: *UI/StaticTop/MoneyHolder/MoneyGrid/MoneyAmount* - a text that displays the player's total money, and is automatically updated once *TotalMoney* variable is updated, and adds a "$ " prefix to display it



Basic Mobile UI

mincode provides a standard UI for you to use. You can easily change it to fit your needs, and use mincode's features to best suit your needs. It is found in the *UI* scene.

- Hierarchy:

GameState holds objects that will activate and deactivate according to the game's flow:

- PreGame: Displayed before the game (level) starts. Deactivated when the LevelStart event is raised.
- InGame: Displayed when in game. Deactivates when the LevelEnd event is raised.
- PostGame: Displayed when level is done. Deactivates when LevelStart is raised.
- StaticTop holds the player's money counter, level number counter, and a restart button All its components are updated according to events, using the TextWrapper script.

The restart button is active only when in game. It uses the *BoolToggler* script to do that. It works with the *IsInGame* variable, which indicates if there is a level playing currently.

I hope you will find this package useful! Enjoy.