Interface - User Manual

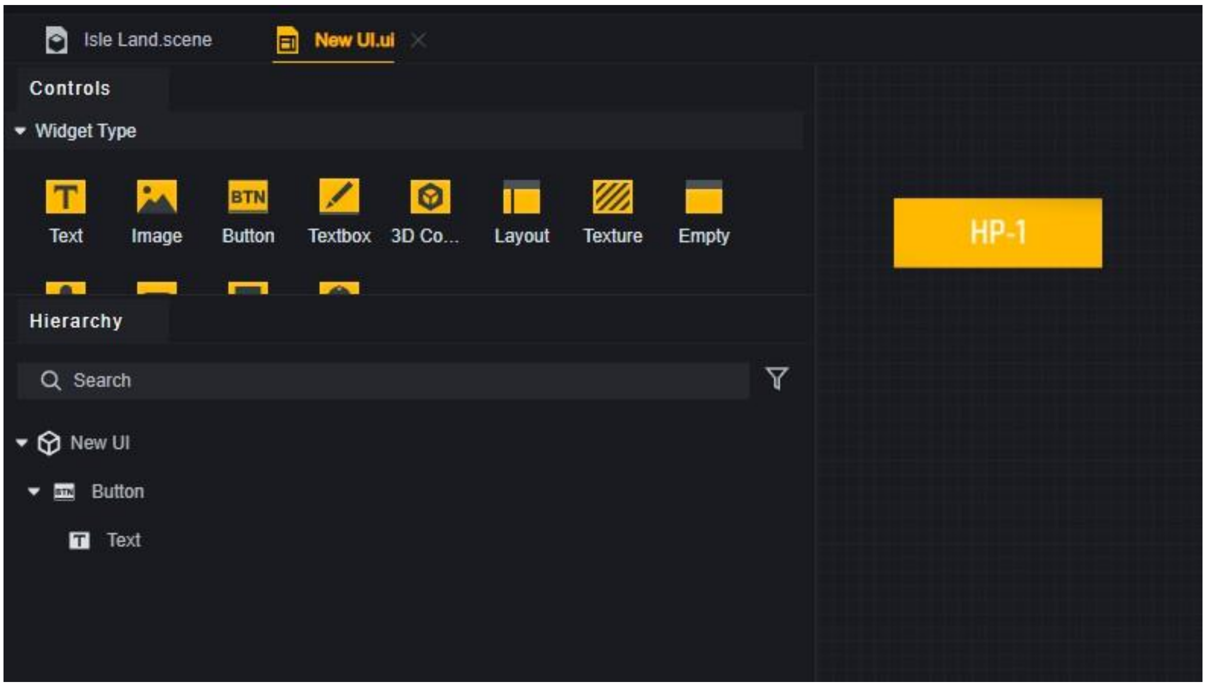
The interface is always displayed as a 2D screen at the forefront of the game window, providing information and receiving interaction from the player. This article will introduce some of the concepts involved, how to use the UI editor and how to script the logic of the UI.

Introduction to the concept

UI files

The UI file stores information about the customized UI, and the corresponding UI entity will be created based on the UI file when the game is run.

UI files can be created out of assets and edited. Within a single UI file, you can add multiple UI controls, which provide various functions such as text display, image display and receiving input.

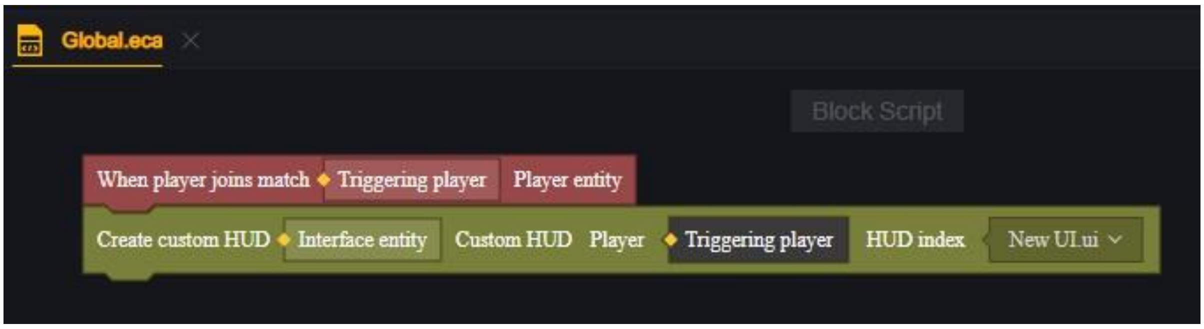


A single UI file can create multiple custom interface entities, such as a button created by each player through the same UI file, or plural buttons created multiple times on a player's interface using that UI file, with the UI entities to which the buttons belong being different entities created from the same UI file.

Customizing interface entities

When the game is run, corresponding custom interface entities, hereafter referred to as UI entities, are created based on the UI files.

The UI entity itself can mount scripts and has attributes. there are several controls under the UI entity, and the controls belonging to it can be fetched through the UI entity.

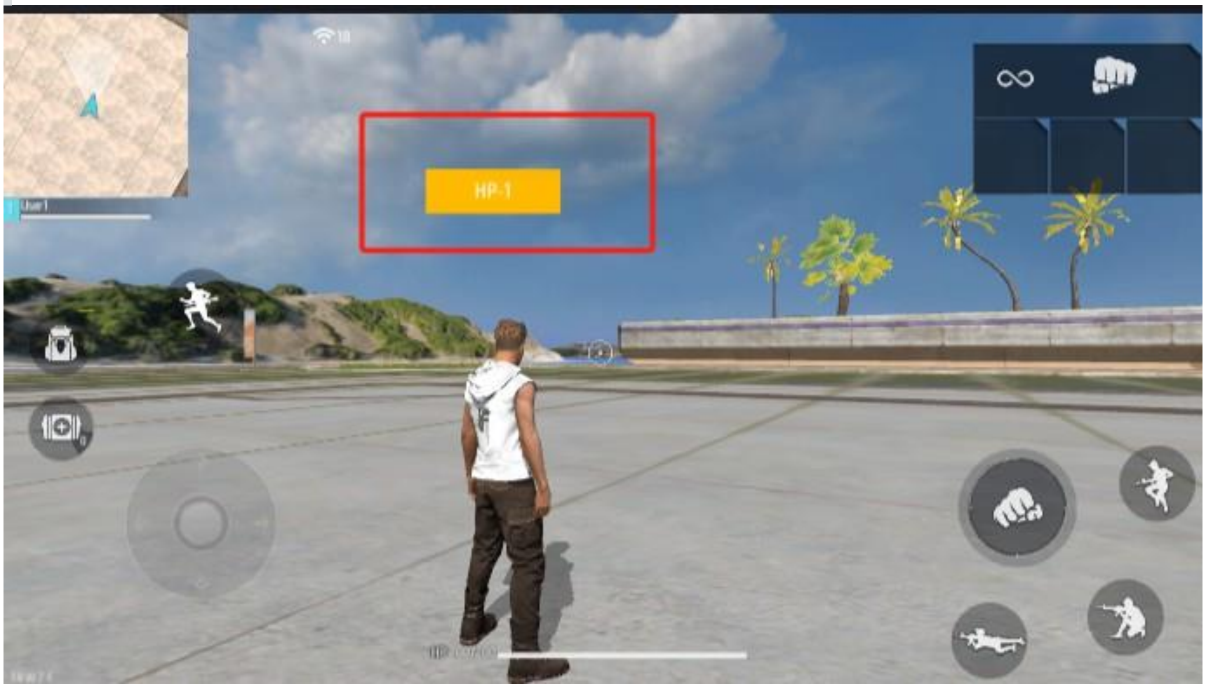

Global.eca

Block Script

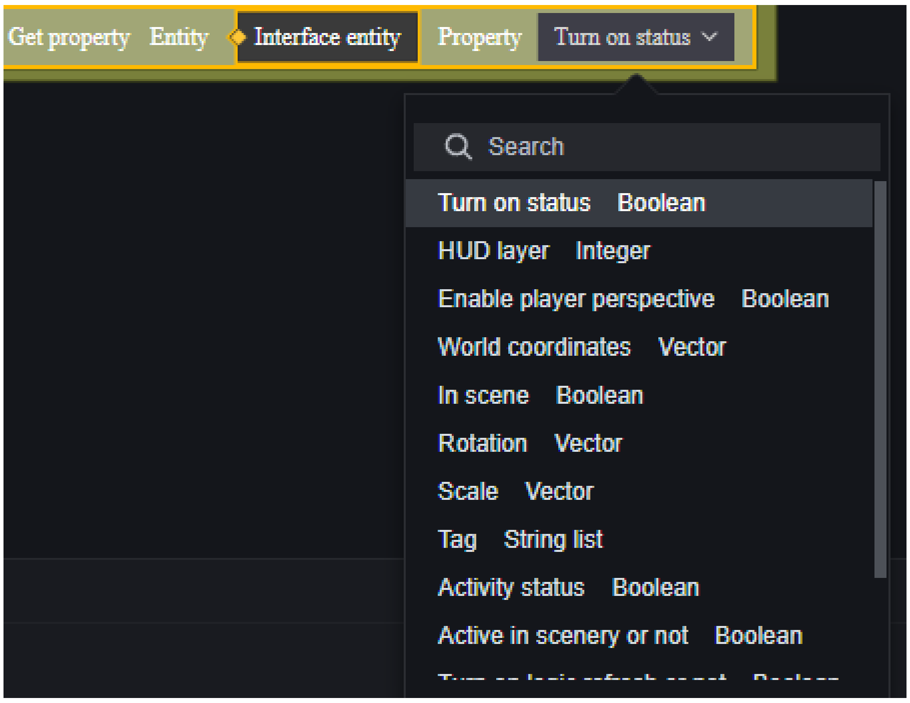
When player joins match ◆ Triggering player Player entity

Create custom HUD ◆ Interface entity Custom HUD Player ◆ Triggering player HUD index New UI.ui

Creating UI entities



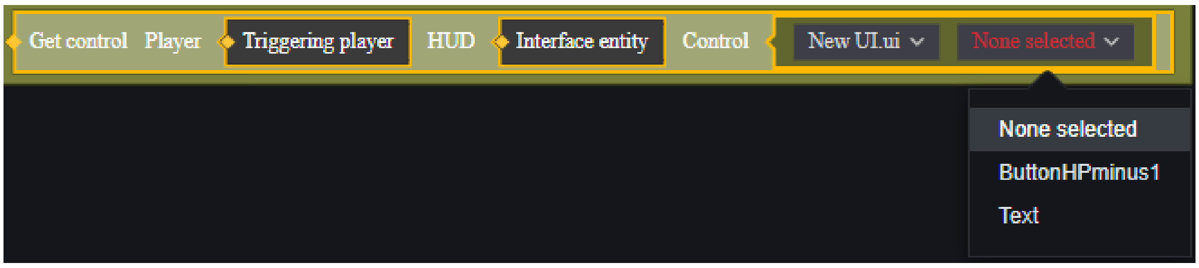
The only UI entity created on this player is a button control.


Get property Entity ◆ Interface entity Property Turn on status

Search

Turn on status	Boolean
HUD layer	Integer
Enable player perspective	Boolean
World coordinates	Vector
In scene	Boolean
Rotation	Vector
Scale	Vector
Tag	String list
Activity status	Boolean
Active in scenery or not	Boolean

Some of the attributes of the UI entity

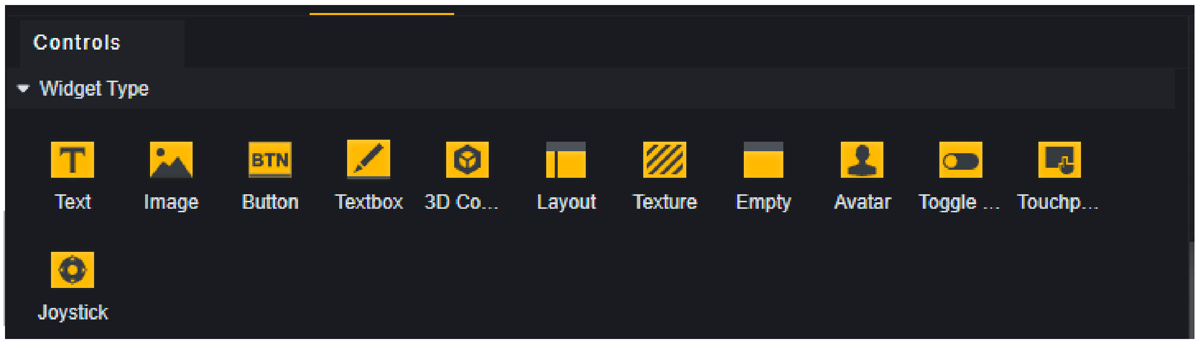


Controls for UI entities

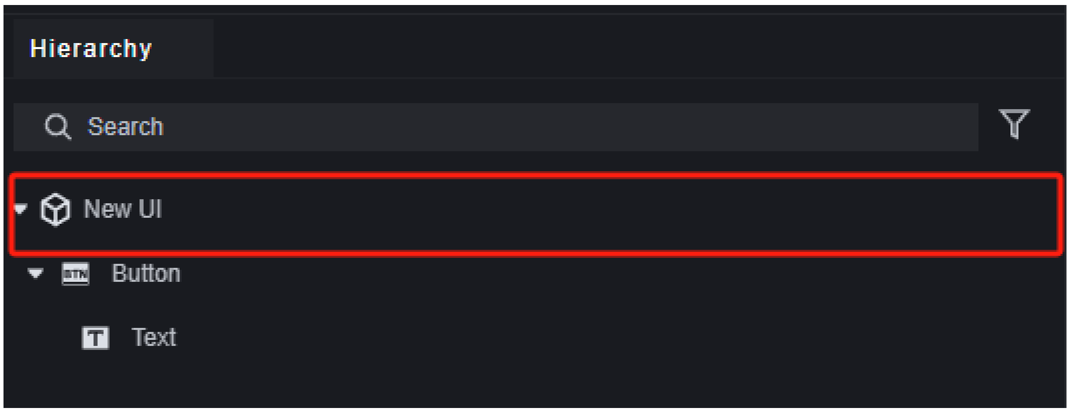
The UI file name is the custom interface entity ID. custom interface entity ID is an index, and an index is a different concept from an entity.

a control (e.g. button, text box etc) (computing)

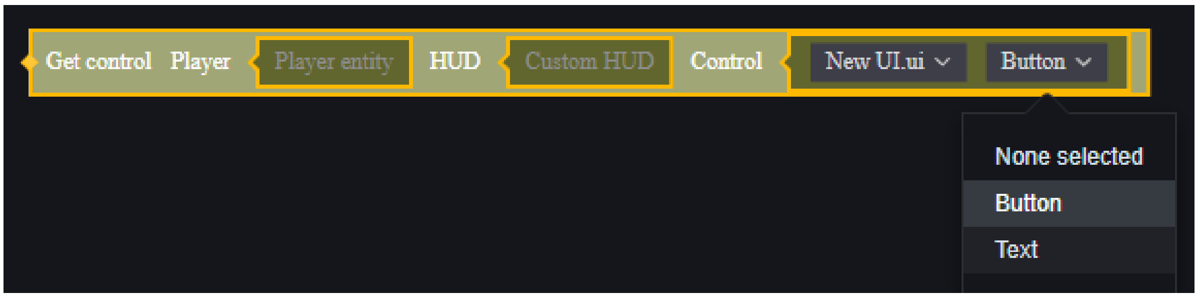
Controls are the basic building blocks of the UI, the components that actually produce functionality. In the UI file, you can select the controls to be used and configure the properties of each control.



Any UI file has a root node; UI controls are parented to the root node or to other UI controls. This root node corresponds to the UI entity created by the UI file, and has a structural role only, without providing actual control functionality.



In the script, you can get all of its controls from a particular custom interface entity and read and write its properties.



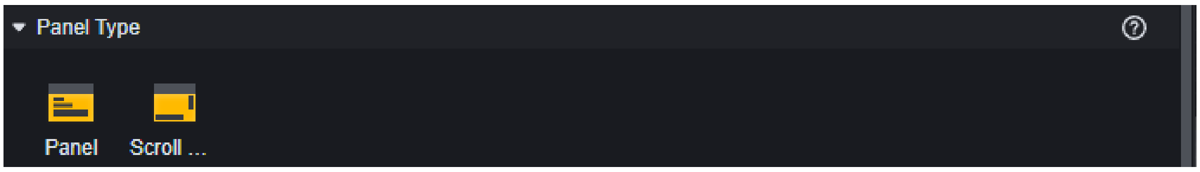
The UI in this script node is the interface entity created from the UI file, but a single UI file can create multiple entities, and you need to specify which controls on which interface entity for which player.

kneading board

Panels are containers for controls.

The panel itself has a certain visual representation and provides a role in the display hierarchy; controls under the same panel are affected by the panel hierarchy.

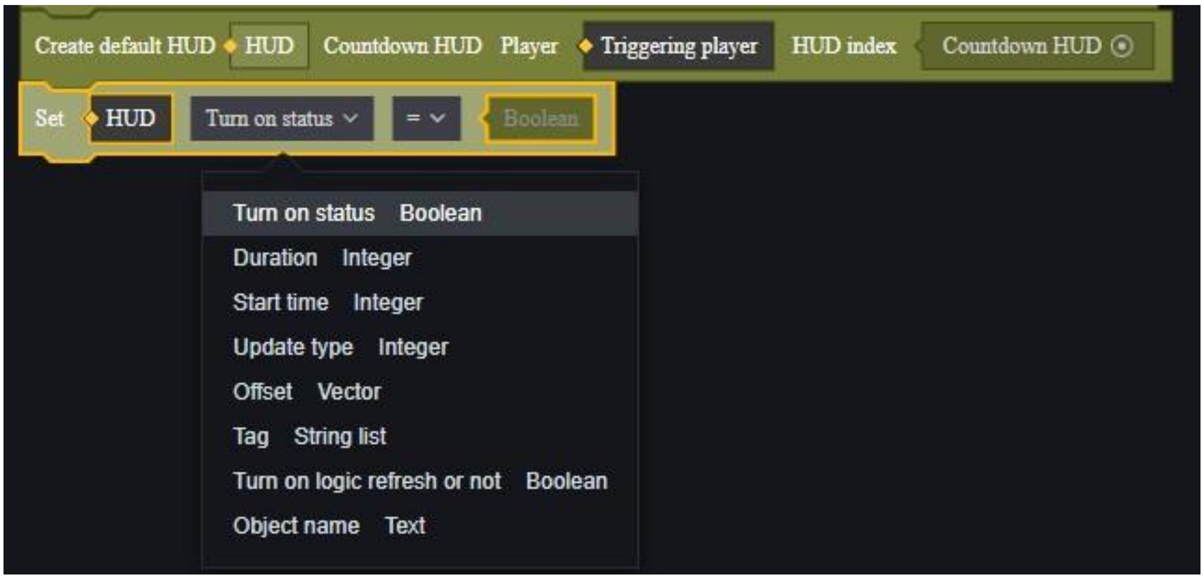
In addition to the basic panels, we also provide scrolling panels for scrolling through multiple controls when placed in a fixed area.



Built-in UI

The built-in UI is the official UI content that has already been created, some of which are loaded by default, such as the movement joystick and jump button; some of which are not loaded by default, such as the countdown timer and compass.

The built-in UI cannot modify the controls in it, but it is possible to dynamically modify the properties of the built-in UI entities open for editing.



Create default HUD HUD Countdown HUD Player Triggering player HUD index Countdown HUD

Set HUD Turn on status = Boolean

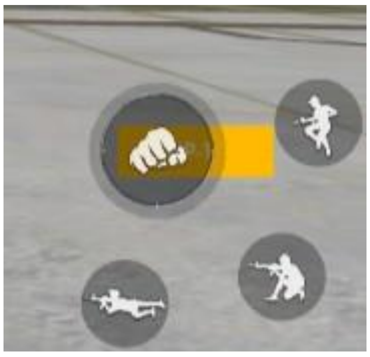
Turn on status	Boolean
Duration	Integer
Start time	Integer
Update type	Integer
Offset	Vector
Tag	String list
Turn on logic refresh or not	Boolean
Object name	Text

UI Hierarchy

When different interfaces overlap, the concept of UI hierarchy is introduced.

The UI level determines the drawing order of this custom UI, and conflicts with other rendering levels may cause the rendering order to be interspersed. The rendering order of controls within a custom UI is automatically set based on this property. It is recommended to leave a reasonable interval between rendering levels for custom UIs that may overlap.

Attempting to manipulate an overlapping region will always attempt to manipulate the control displayed at the top.

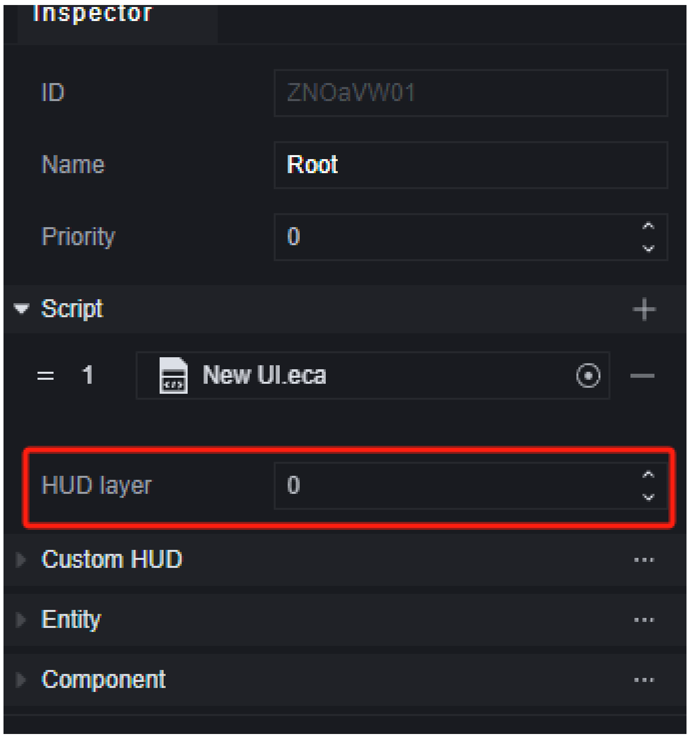


> Clicking on the part of the attack that overlaps the customization button at this point will only perform the attack. Clicking on the part of the custom button that does not overlap with the attack will trigger the custom button.

The custom UI hierarchy consists of the UIRoot hierarchy,

the panel hierarchy, and the control hierarchy. Among them:

The UIRoot level has the biggest impact, the UI entities created by different UI files will be sorted according to the UIRoot level, the bigger the UIRoot level the more the created UI entities will be displayed on the top.



Inspector

ID: ZNOaVW01

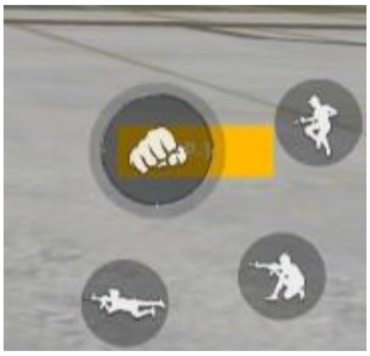
Name: Root

Priority: 0

Script

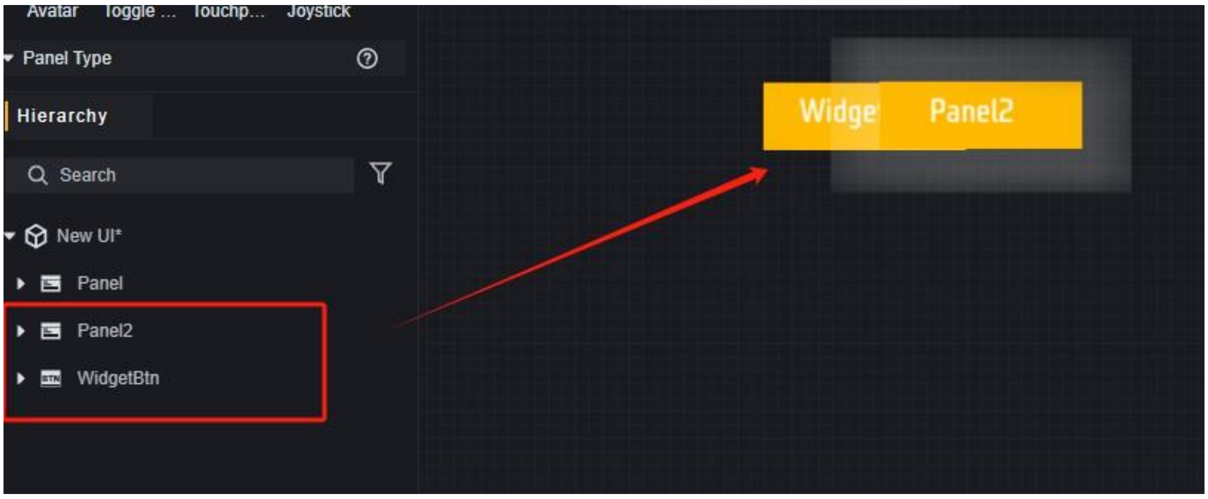
1 New UI.eca

HUD layer: 0

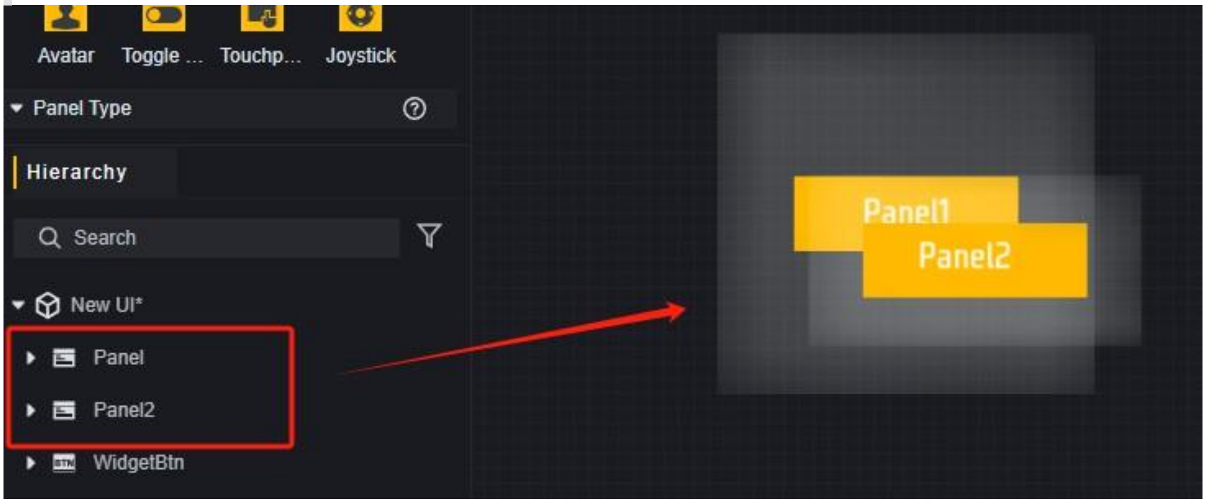
> When UIRoot levels are the same, there may be inconsistencies in operation detection and display, so don't set two overlapping UIs to the same level.
>
> 
>
> At this point, the UIRoot level where the custom button is located is -1, which is less than the level 0 of the default attack button
>
> The Root level of the built-in UI is all 0. If you need to place it below the built-in UI, you need to set the level to a negative number.

The **panel hierarchy** has a second impact, which affects the order of display within the same UI file. The panel hierarchy adheres to the following rules:

1. Panels and controls within panels are always displayed above controls under the same level.
2. Panels on the same level are determined by the order under the Hierachy menu between them, with the higher the level the further down the hierarchy, the more they are displayed on top.



Controls that are ordered under the panel are also obscured by the panel.

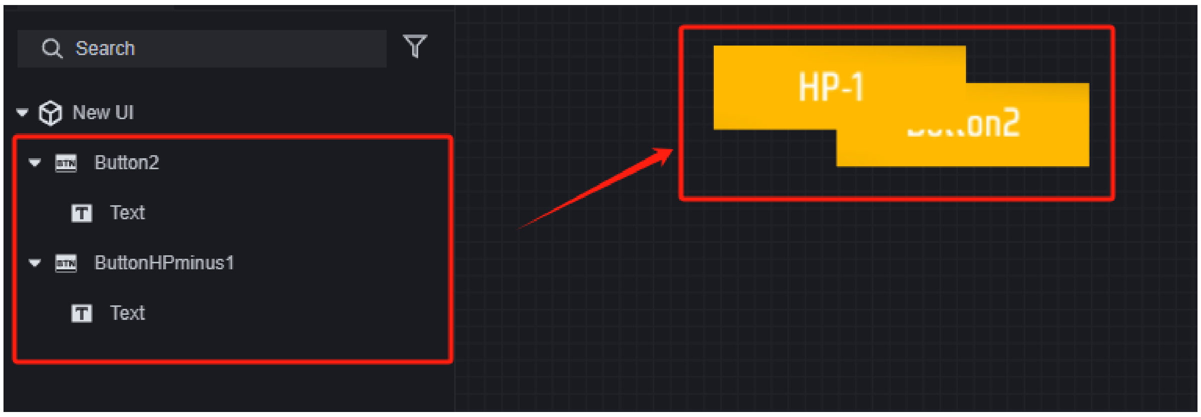


Two panels on the same level, the lower one will cover the upper one.

The panel hierarchy is hidden and cannot be set directly, you can only adjust the display order of the panels by modifying the hierarchy menu.

control hierarchy

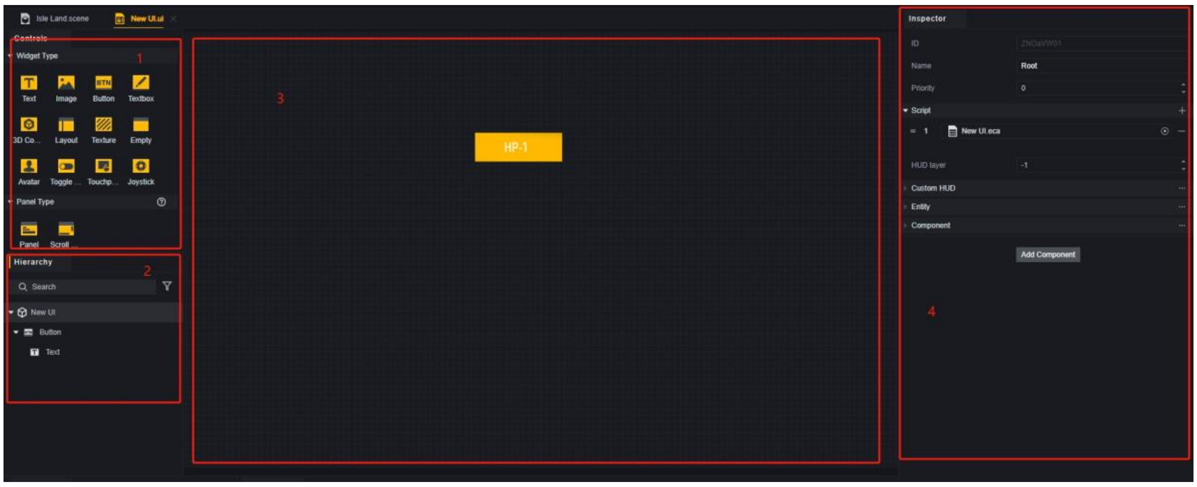
For controls under the same UIRoot, under the same panel, their hierarchy is determined by the order under the Hierachy menu, the further down the hierarchy the higher.



The control hierarchy is also hidden and cannot be set directly, you can only adjust the display order of the controls by modifying the hierarchy menu.

Customized Interface

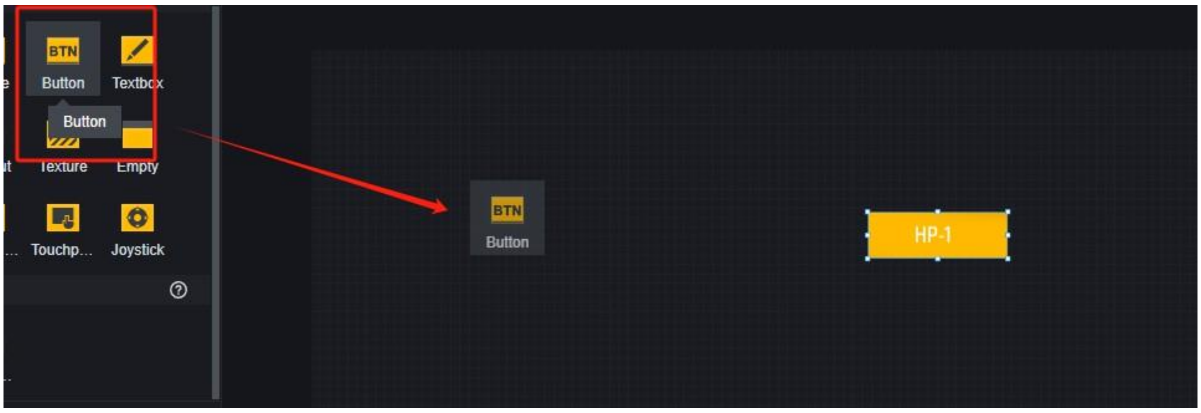
editorial interface



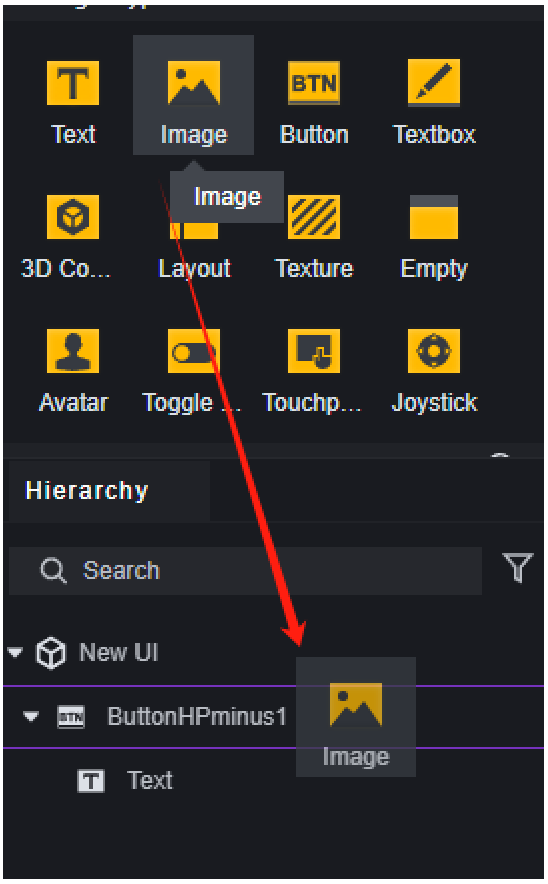
1. **Controls/Panels**: Here are the templates for all controls and panels.
2. **Hierarchy menu**: In the hierarchy menu, you can view the controls in the UI file. And you can check the hierarchy of the controls inside the UI file, the lower the controls are displayed at the top.
3. **Canvas**: In the canvas you can adjust the position and size of the control to achieve the desired effect.
4. **Properties Menu**: You can edit the properties of the control in the Properties menu.

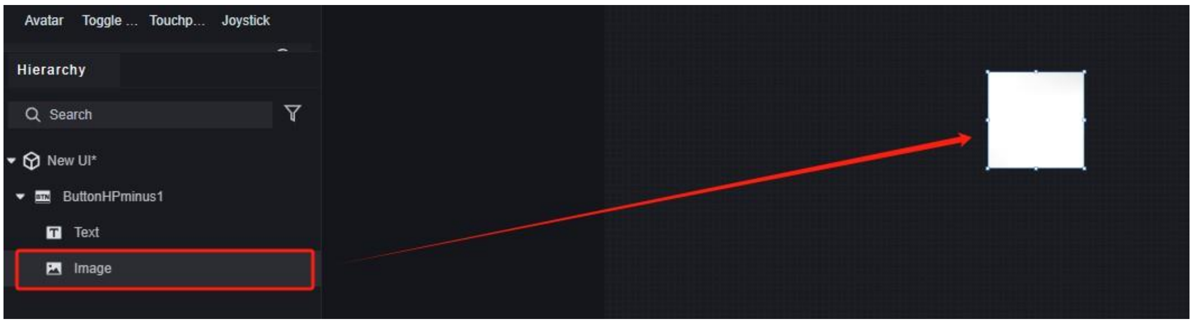
Using controls

Dragging a control from the control selection panel to the canvas creates a control at the corresponding position. Controls dragged onto the canvas are parented to the root node by default.



Dragging a control from the Control Selection Panel to any control or root node in the Hierarchy Menu creates a child control with that control or root node as the parent. It is created in the same location as the parent, but in the center of the canvas when the root node is the parent.

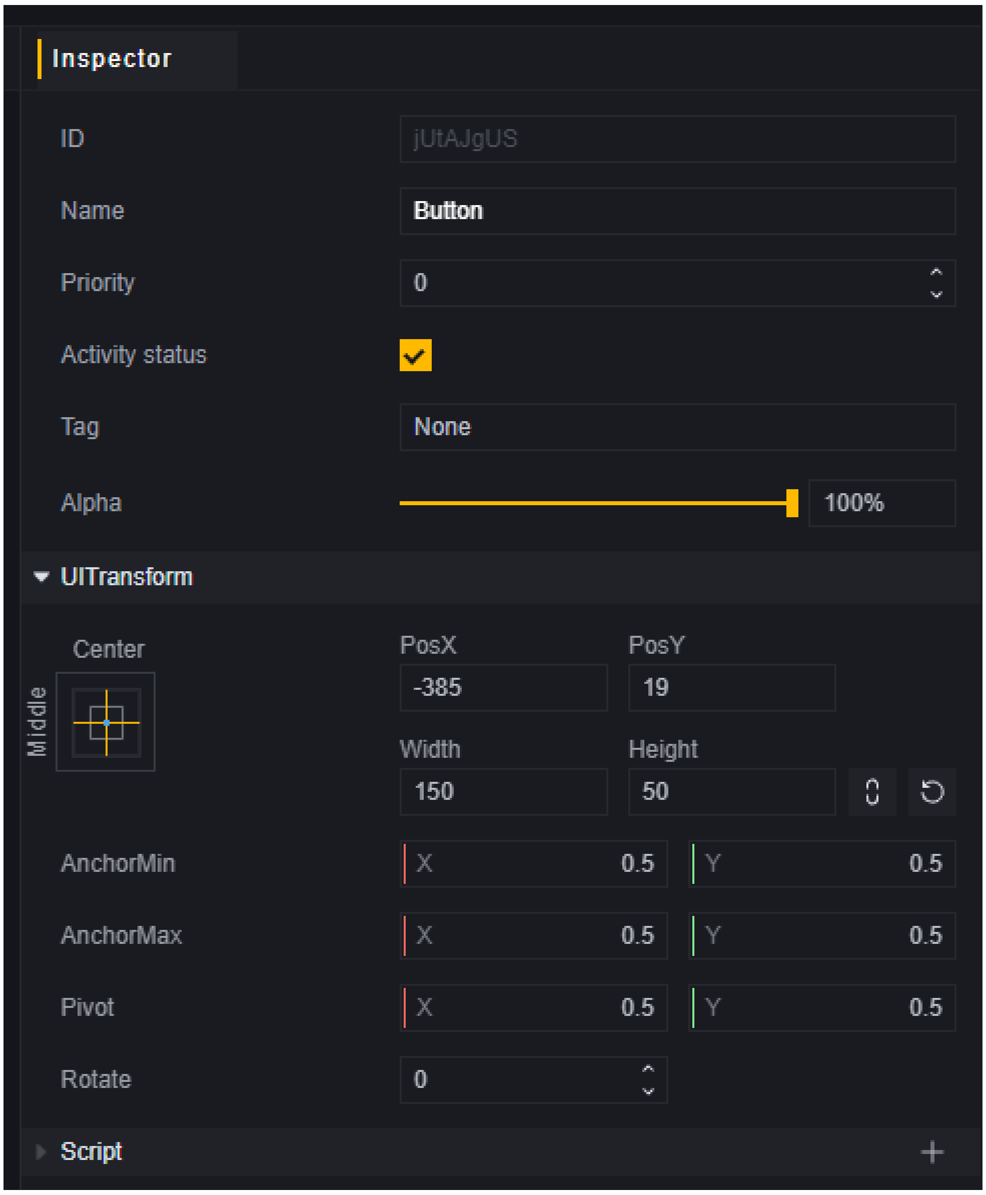




> Because the newly added image is the highest level in this UI, it overrides the previous button.

edit control

Modify the properties of the selected control in the Properties panel. Each control has generic properties and properties that are specific and related to the type of control.



Inspector

ID	jUtAJgUS
Name	Button
Priority	0
Activity status	✓
Tag	None
Alpha	100%

UITransform

Center / Middle

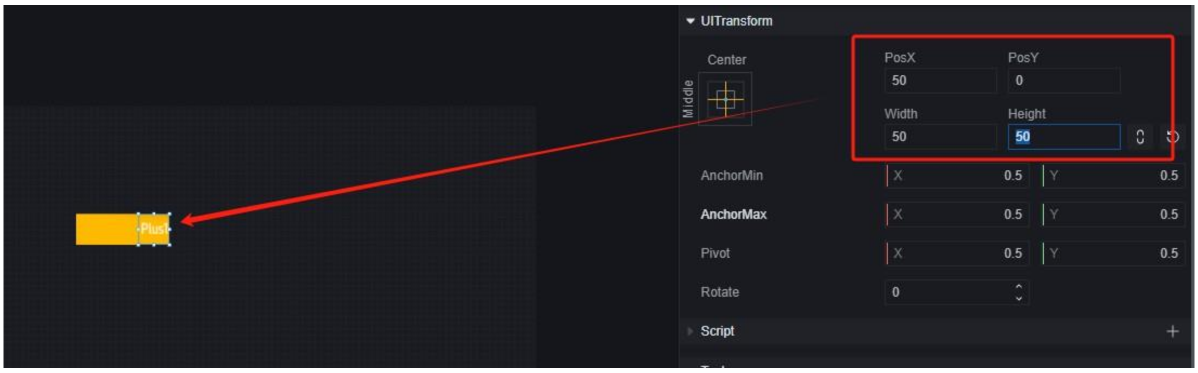
	PosX	PosY
	-385	19
	Width	Height
	150	50
AnchorMin	X 0.5	Y 0.5
AnchorMax	X 0.5	Y 0.5
Pivot	X 0.5	Y 0.5
Rotate	0	

Script

Common Properties

The UI control will adjust its size and position according to the anchor parameters and the parent UI size, while for UI controls without a parent, it will adapt to the user's device aspect ratio, thus achieving the effect of adaptive resolution

In most simple cases, you can use the anchor presets directly without having to manually modify the anchor parameters.



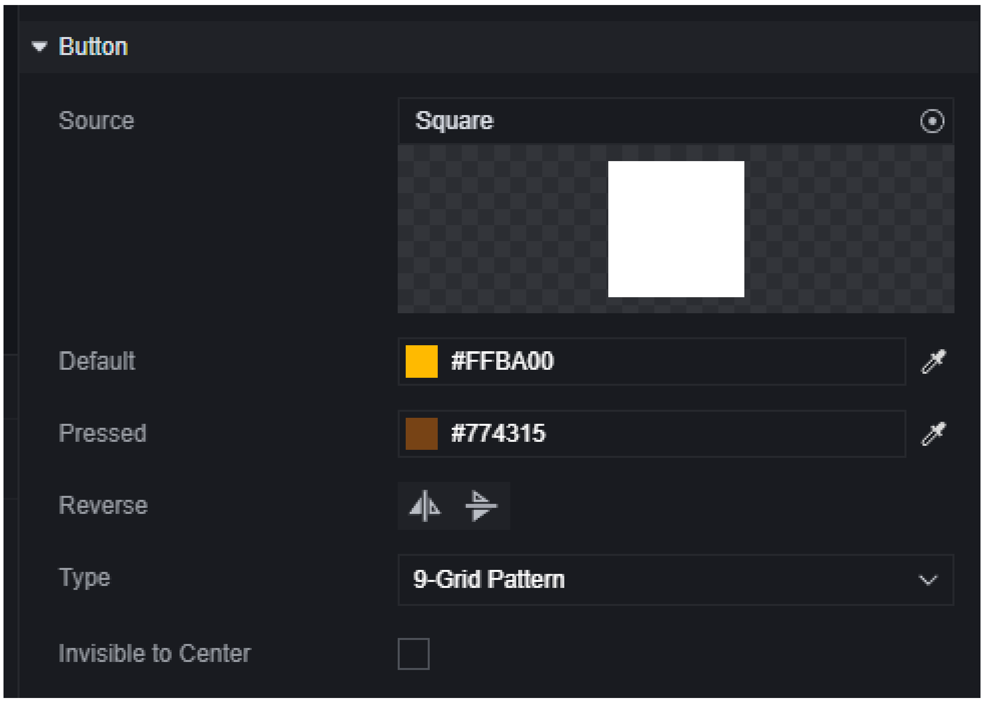
UITransform

Center / Middle

	PosX	PosY
	50	0
	Width	Height
	50	50
AnchorMin	X 0.5	Y 0.5
AnchorMax	X 0.5	Y 0.5
Pivot	X 0.5	Y 0.5
Rotate	0	

Script

Modify the UITransform property of a text control and it will be offset relative to its parent: the custom button. For each class of control, there are also properties specific to that class of control.


Button

Source	Square
Default	🟨 #FFBA00
Pressed	🟫 #774315
Reverse	
Type	9-Grid Pattern
Invisible to Center	☐

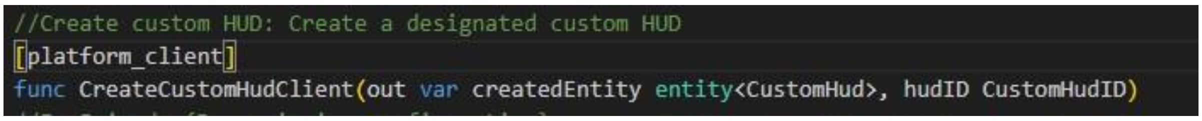
> Unique properties of buttons

scripts

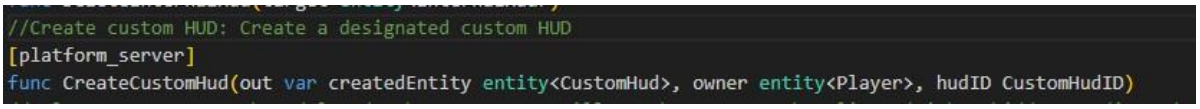
Client-server distinction

establish

Similar to the script itself, the UI needs to distinguish between server and client. The server UI is created by the server API and uses the server API and events internally. The client UI is the same. The UI created using the server meta-script is the server UI, and the UI created using the client meta-script is the client UI, because we have already restricted the platforms that can be used within the meta-script, and when using the code script, you need to make sure that the platform information of the API you are using is compatible with the platform on which the code script is running.



```
//Create custom HUD: Create a designated custom HUD
[platform_client]
func CreateCustomHudClient(out var createdEntity entity<CustomHud>, hudID CustomHudID)
```



```
//Create custom HUD: Create a designated custom HUD
[platform_server]
func CreateCustomHud(out var createdEntity entity<CustomHud>, owner entity<Player>, hudID CustomHudID)
```

Mounting Scripts

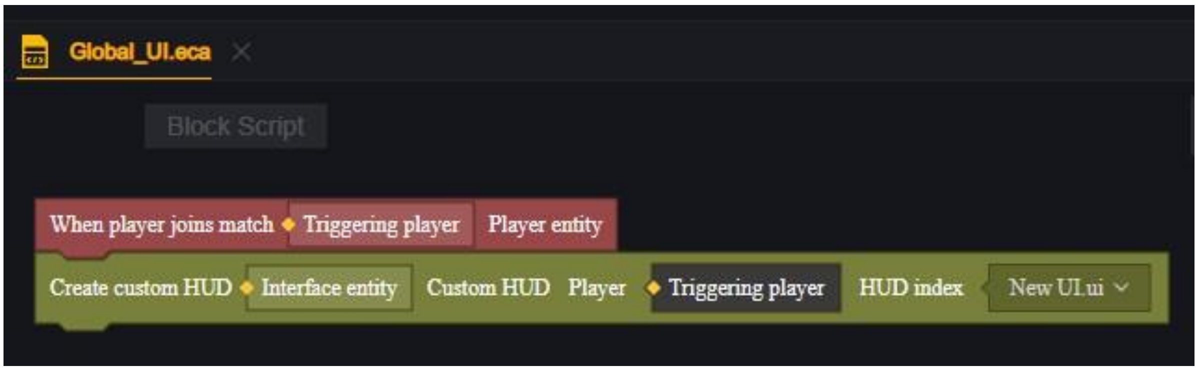
Scripts can be mounted on UI entities, it is highly recommended to mount only server scripts on the server UI

and only client scripts on the client UI. Server scripts on the client UI where server events are not triggered.

> It is not recommended to use scripts cross-platform.

Create UI

You can only create the UI for the platform you're running on through the interface on the script. take the meta-script as an example:

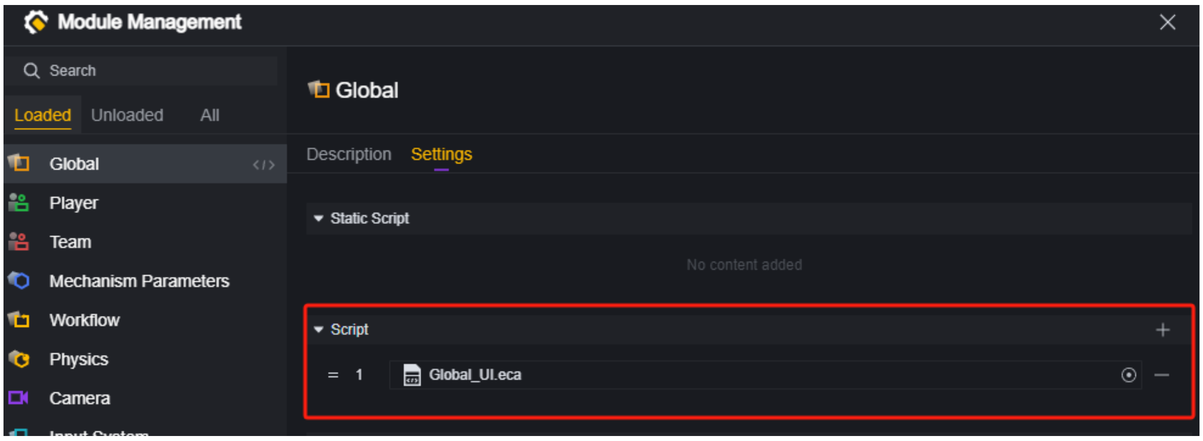

Global_UI.eca

Block Script

When player joins match ◆ Triggering player Player entity

Create custom HUD ◆ Interface entity Custom HUD Player ◆ Triggering player HUD index New UI.ui

It is recommended to add scripts dedicated to managing the UI in the Global module.


Module Management

Loaded Unloaded All

Global
Player
Team
Mechanism Parameters
Workflow
Physics
Camera

Global

Description Settings

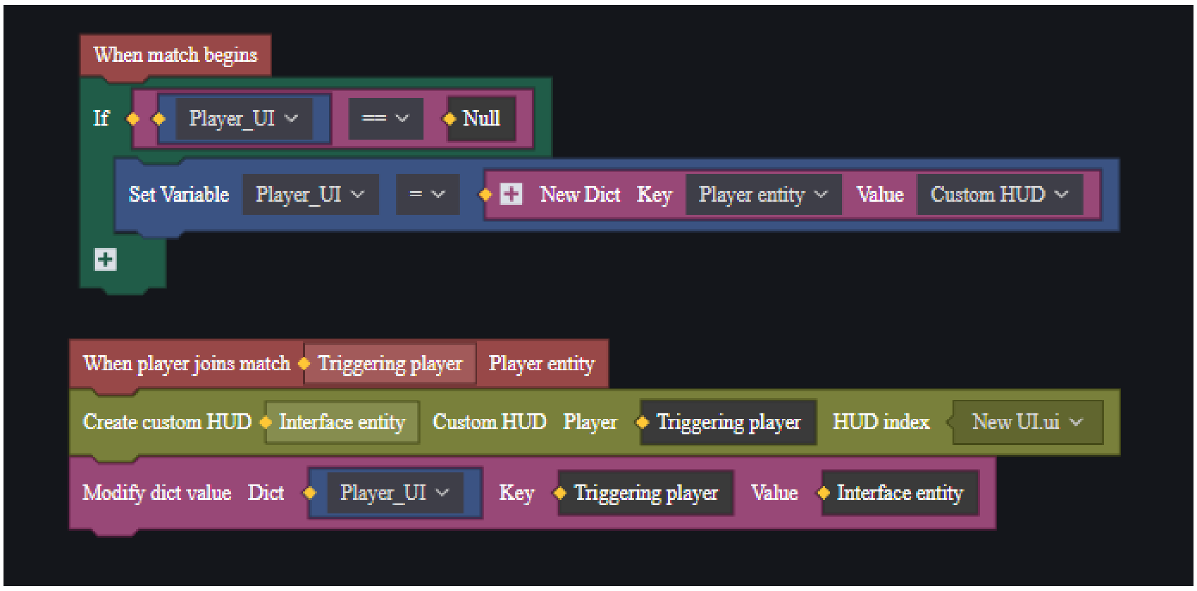
Static Script

No content added

Script

= 1 Global_UI.eca

When creating UI using the server API, you need to specify the player. Then it is very convenient to use a dictionary to manage the UI on the player:


When match begins

If ◆ ◆ Player_UI == ◆ Null

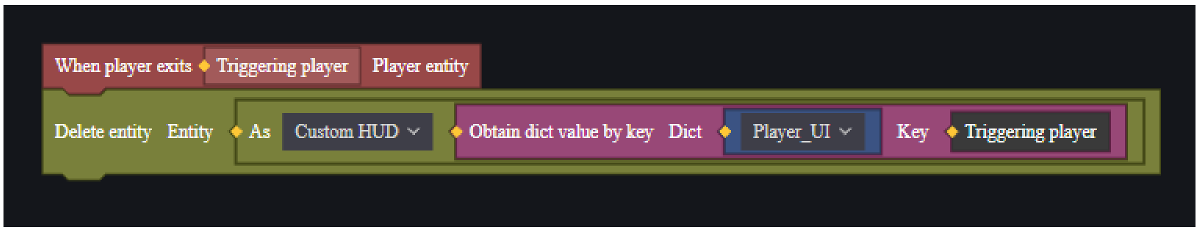
Set Variable Player_UI = ◆ New Dict Key Player entity Value Custom HUD

When player joins match ◆ Triggering player Player entity

Create custom HUD ◆ Interface entity Custom HUD Player ◆ Triggering player HUD index New UI.ui

Modify dict value Dict ◆ Player_UI Key ◆ Triggering player Value ◆ Interface entity

When you need to modify the UI on the corresponding player, use the dictionary to quickly fetch the UI entity:

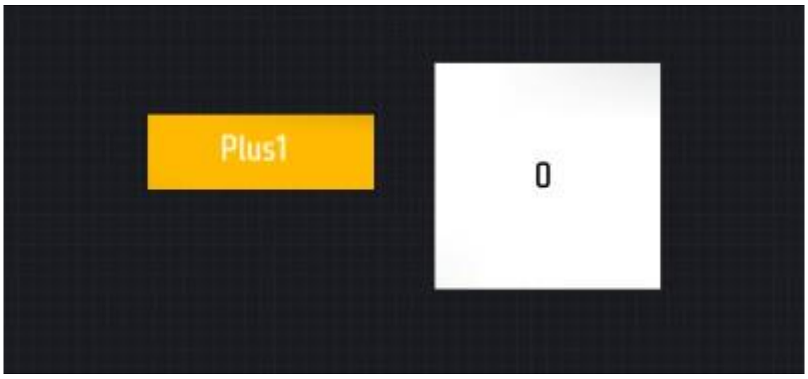

When player exits ◆ Triggering player Player entity

Delete entity Entity ◆ As Custom HUD ◆ Obtain dict value by key Dict ◆ Player_UI Key ◆ Triggering player

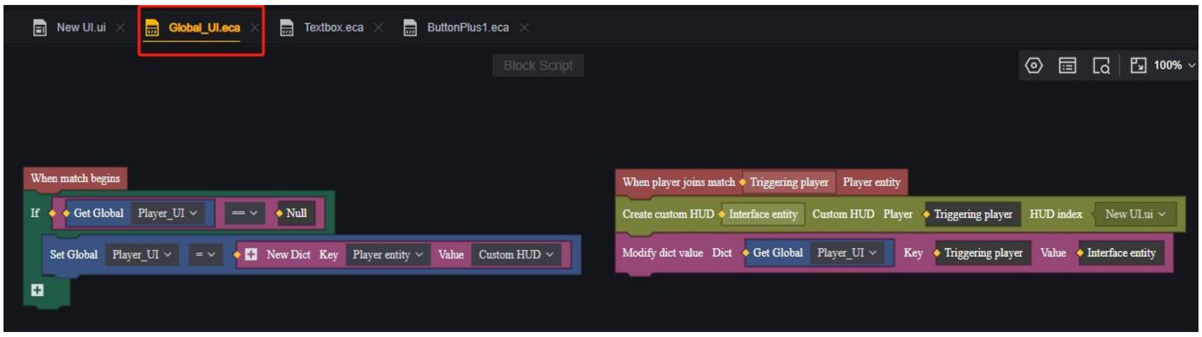
control logic

The player can make inputs via buttons, joysticks, toggles etc. Using the appropriate events the player's input can be acquired and the designed logic can be executed.

Suppose we have such a button and a text that each click adds 1 to the number on the text.



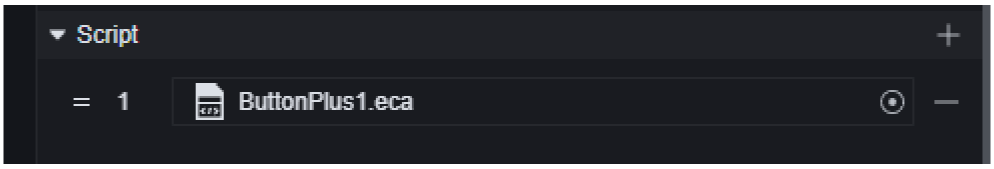
When that button is pressed, the text on another text control needs to be changed, so you need to fetch the text control for that UI entity. Using a dictionary, the UI entity is stored in a global variable when the UI is created:



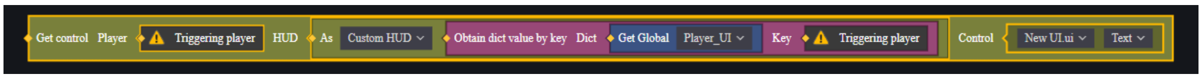
```
When match begins
If ◆ Get Global Player_UI == ◆ Null
    Set Global Player_UI = ◆ New Dict Key Player entity Value Custom HUD
```

```
When player joins match ◆ Triggering player  Player entity
Create custom HUD ◆ Interface entity  Custom HUD  Player ◆ Triggering player  HUD index  New UI.ui
Modify dict value  Dict ◆ Get Global Player_UI  Key ◆ Triggering player  Value ◆ Interface entity
```

This is the script used to create the UI in the global module Add a script for the button:



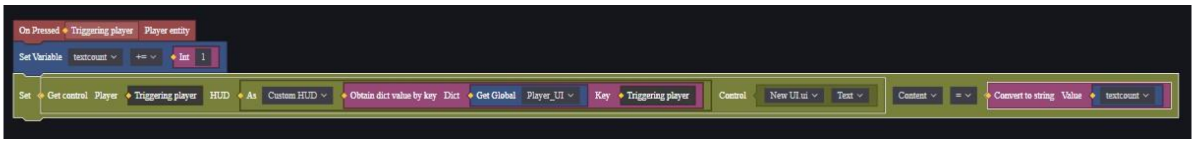
Get the text control from the UI entity stored in



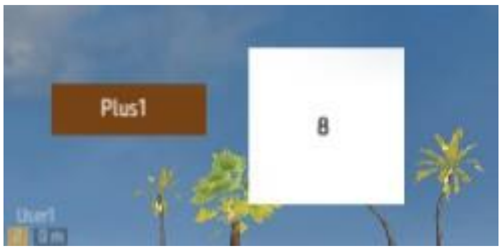
```
◆ Get control  Player ◆ Triggering player  HUD ◆ As  Custom HUD ◆ Obtain dict value by key  Dict ◆ Get Global  Player_UI  Key ◆ Triggering player  Control  New UI.ui  Text
```

the global variable: when the button is pressed,

set the value displayed by this control plus 1:



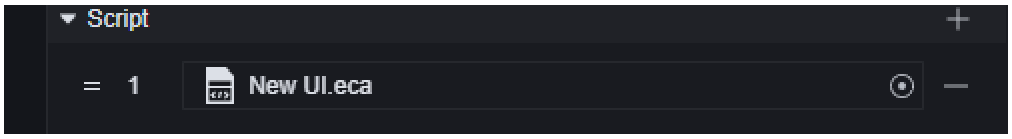
```
On Pressed ◆ Triggering player  Player entity
Set Variable  textcount  +=  ◆ Int 1
Set ◆ Get control  Player ◆ Triggering player  HUD ◆ As  Custom HUD ◆ Obtain dict value by key  Dict ◆ Get Global  Player_UI  Key ◆ Triggering player  Control  New UI.ui  Text  Content  =  ◆ Convert to string  Value ◆ textcount
```



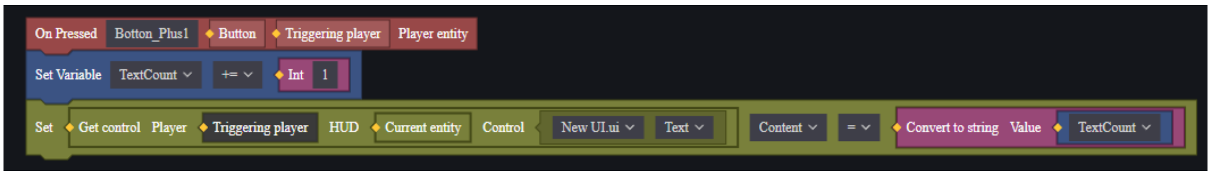
Pressed the button eight times.

pull back (of a key (in music)

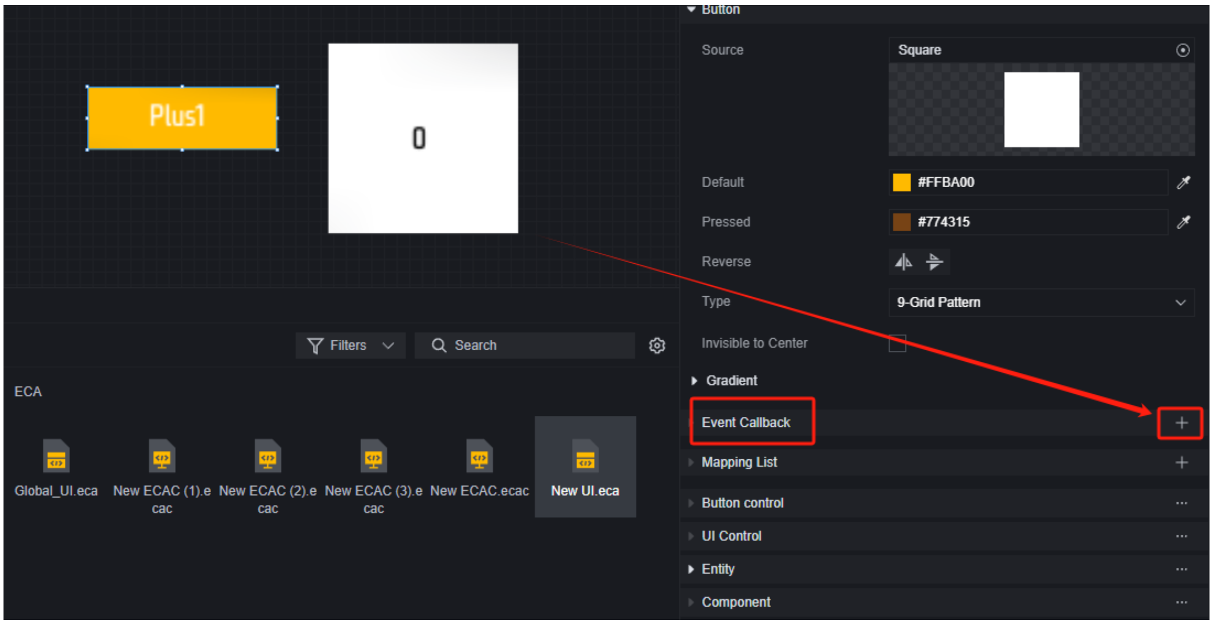
In the above example, it is easy to realize that the operation of modifying the text is not complicated, but fetching the entity where the text control is located is more cumbersome. Using callbacks, you can avoid this tedious operation and quickly realize the interaction between different controls under the same UI entity.
In fact, it's faster to use callbacks in most cases. In the
same example, create a script for a UI entity:


Script
= 1 New UI.eca

Create a callback function for a button whose logic is to add 1 to the number displayed in the text control each time it is called:

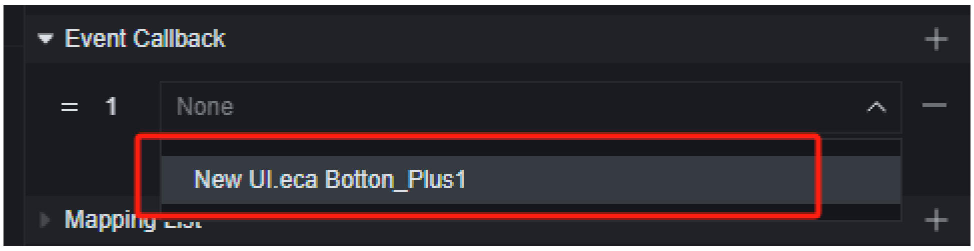

```
On Pressed  Botton_Plus1  Button  Triggering player  Player entity
Set Variable  TextCount  +=  Int  1
Set  Get control  Player  Triggering player  HUD  Current entity  Control  New UI.ui  Text  Content  =  Convert to string  Value  TextCount
```

> The UI callback functions have special nodes located
under the event category. Add callbacks to buttons:

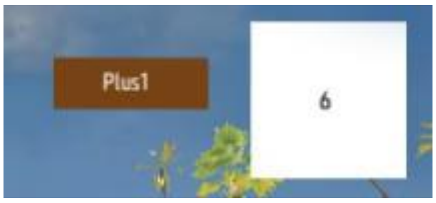


Source: Square
Default: #FFBA00
Pressed: #774315
Reverse
Type: 9-Grid Pattern
Invisible to Center
Gradient
Event Callback
Mapping List
Button control
UI Control
Entity
Component

Select the callback function you just created:


Event Callback
= 1 None
New UI.eca Botton_Plus1
Mapping List

This completes it:



> Pressed the button six times.

Because the callback function sits on the UI entity, it is much easier to get to the controls under the UI entity than when they are on the controls. It is also easier to maintain when the number of controls and logic in the UI entity is large.

suggestion

For server UI, operating UI with high frequency may generate large traffic consumption, if your UI becomes less sensitive, you can try to limit the frequency of some information synchronization operations. For example, using a progress bar to show the player's experience value, when the player acquires experience with high frequency, it may generate larger traffic, it is recommended to separate the experience value data and experience value UI performance, and synchronize the data and performance every once in a while (this trick can be used elsewhere as well).

At the same time, creating and destroying UIs with high frequency will cause large performance overhead and traffic consumption. If you need to show and hide a UI intermittently, consider changing the properties related to the UI display instead of prioritizing an implementation that creates and destroys it repeatedly. This is more server and low-performance device friendly.

typical example

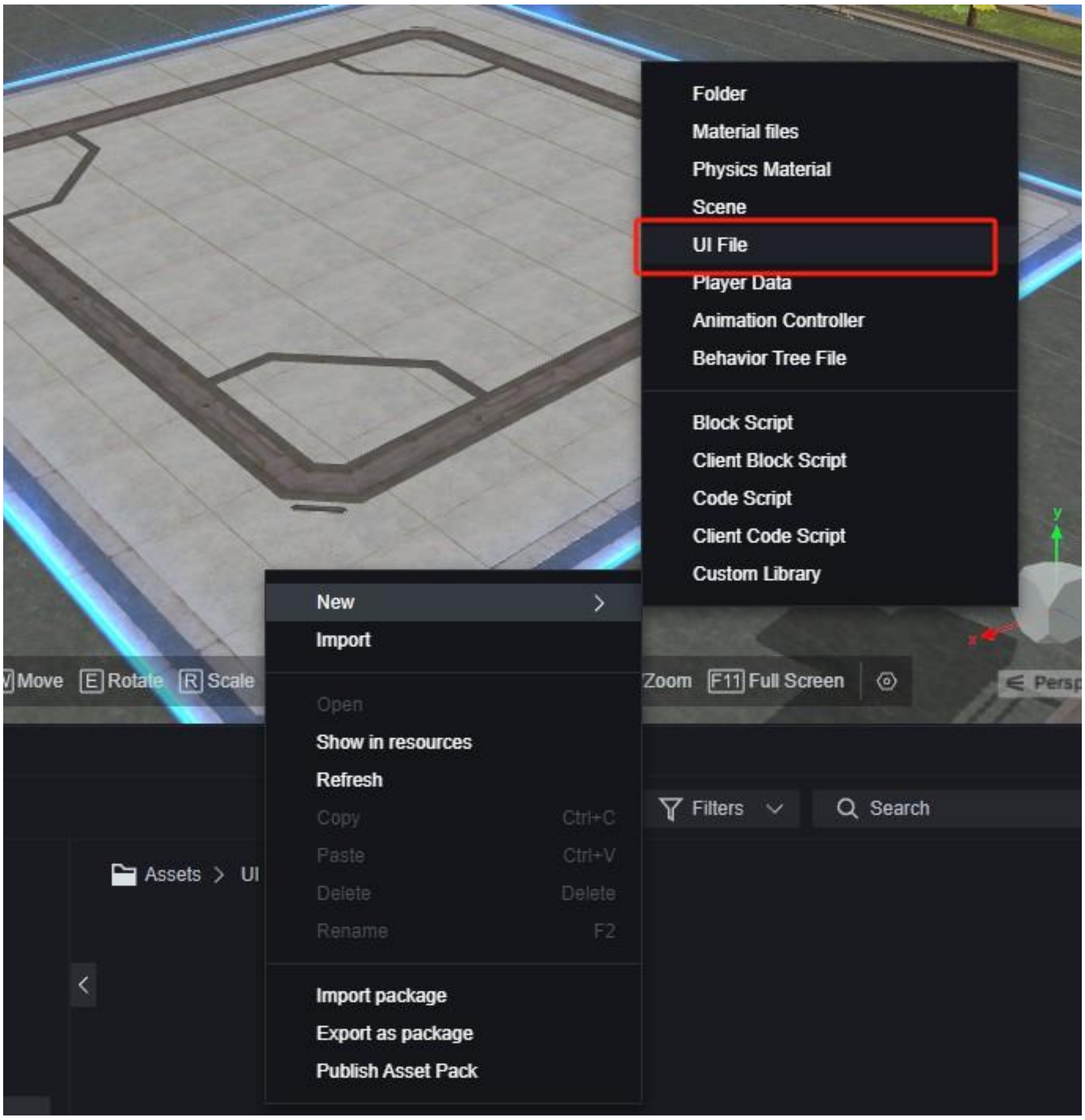
The following assumes a simple custom interface requirement and demonstrates how to create a functional custom interface and destroy it.

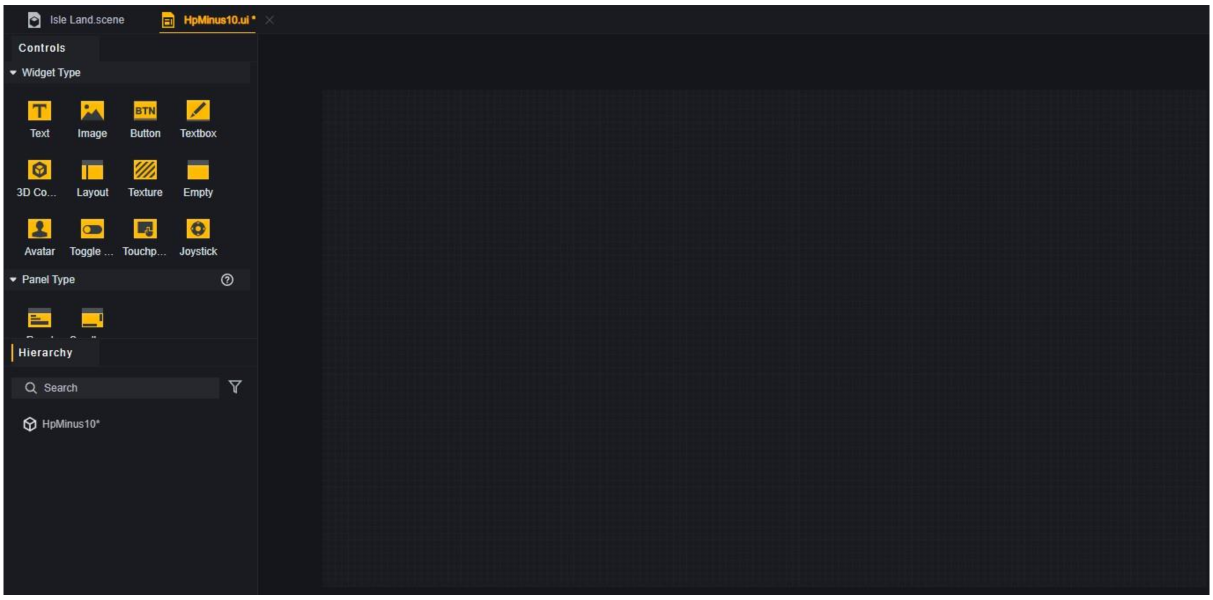
Hypothetical requirement: create a button for the center of each player's screen that causes HP -10 each time it is clicked, and after clicking the button and reducing blood HP is less than or equal to

Destroy the button at 100.

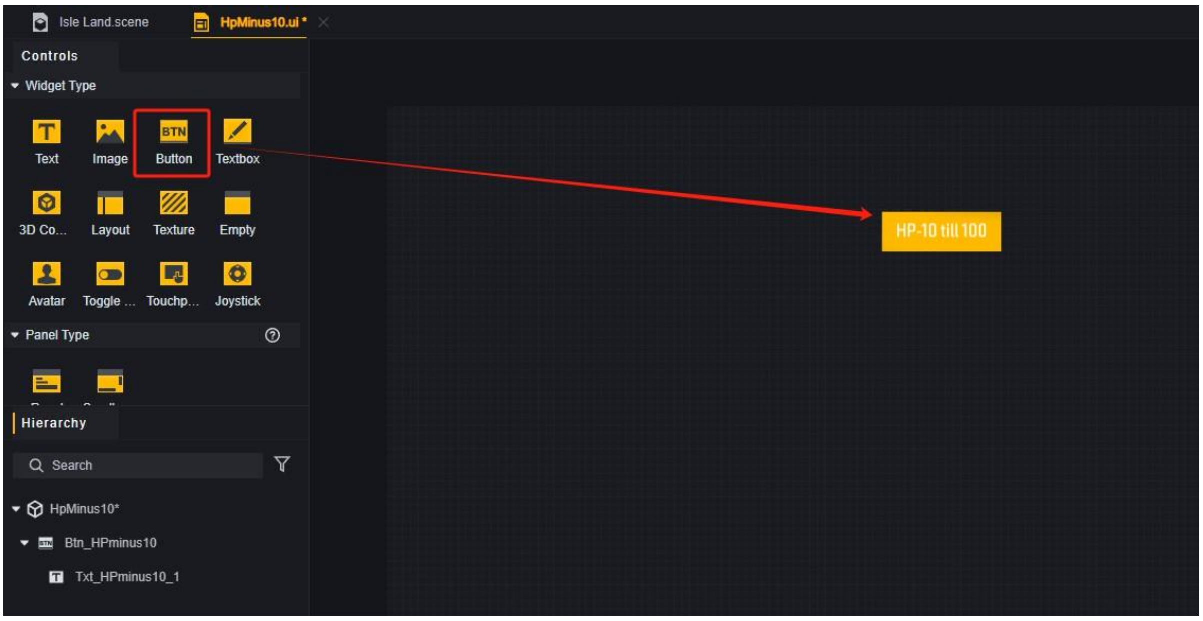
Creating the UI

Create a new UI file and open it for editing

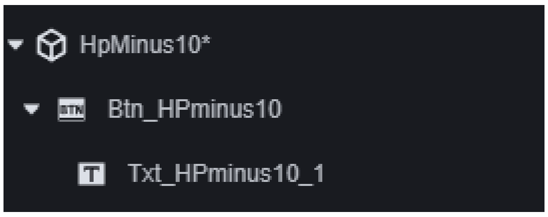




The canvas corresponds to the player's screen and creates a button at the top center. Also, add a text control as a child control for the button with the purpose of explaining the button's function to the player.

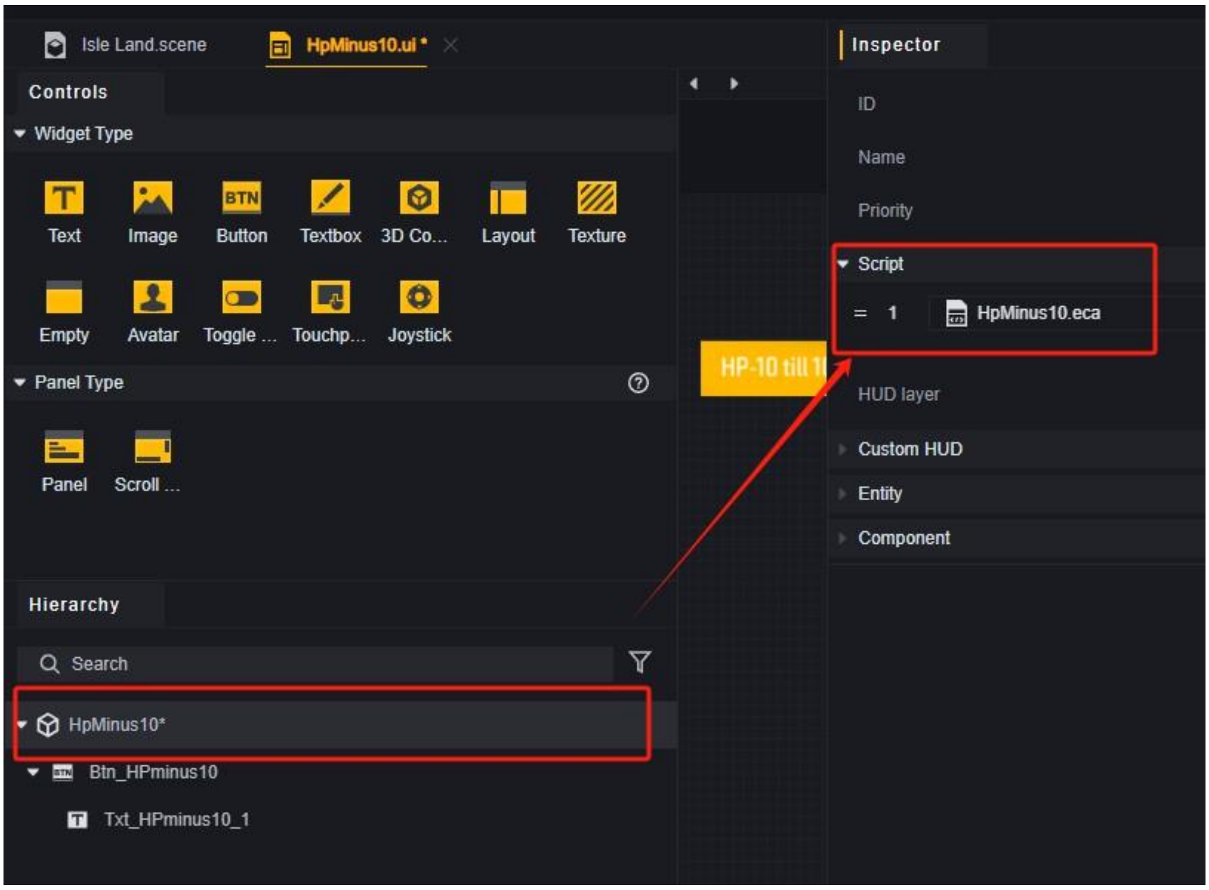


The control name can be modified in the hierarchical menu for ease of administration.

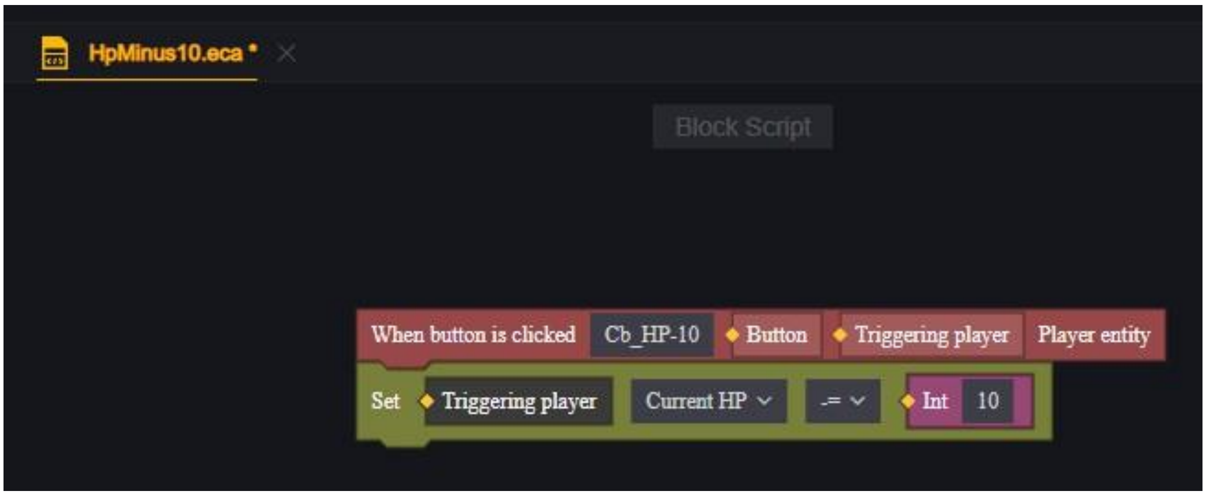


button logic

We decided to use callbacks to handle reducing player blood by mounting a script for the root node.

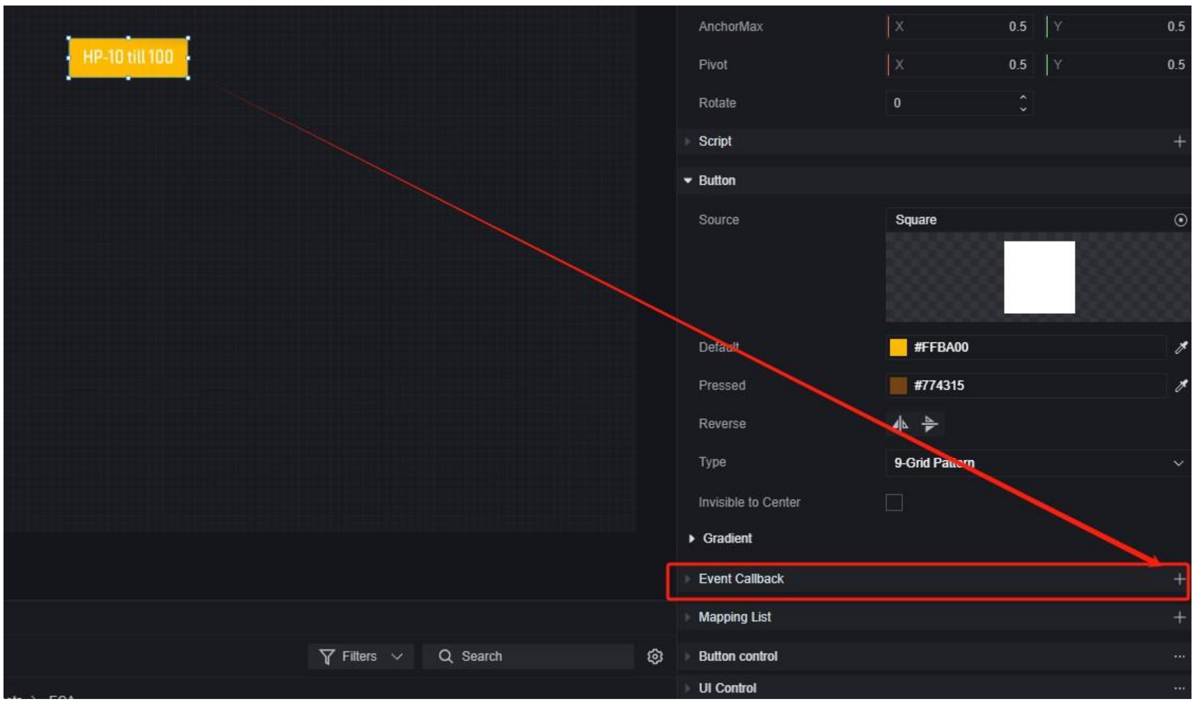


Write logic for the root node script.

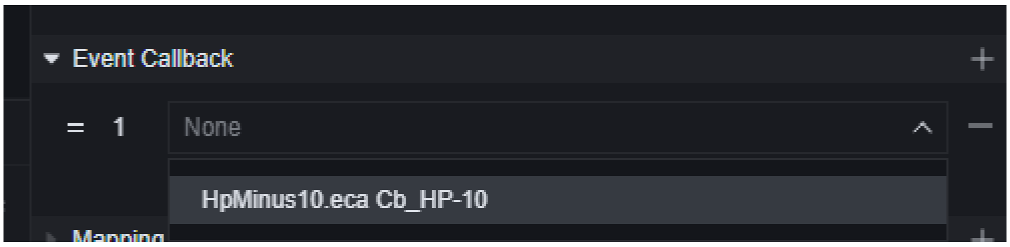


```
When button is clicked | Cb_HP-10 | Button | Triggering player | Player entity
Set | Triggering player | Current HP | -= | Int | 10
```

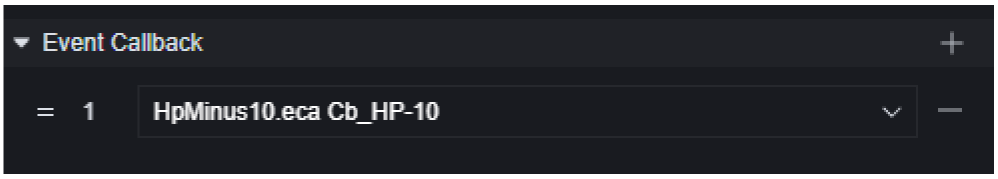
Here we have created a callback function for button clicks Cb_HP-10 return to UI file edit, select the button control and add a callback for it.



AnchorMax X 0.5 Y 0.5
Pivot X 0.5 Y 0.5
Rotate 0
Source Square
Default #FFBA00
Pressed #774315
Reverse
Type 9-Grid Pattern
Invisible to Center



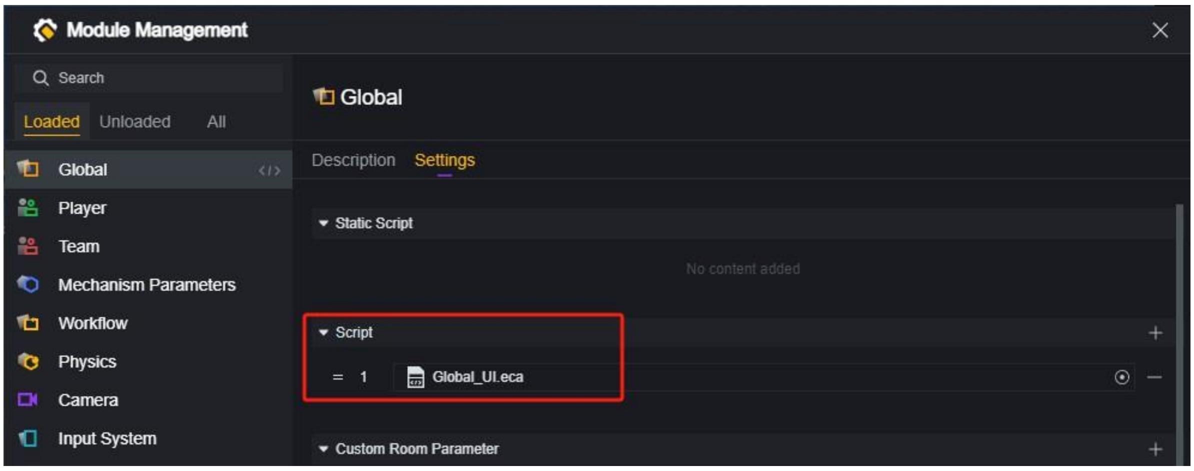
= 1 None
HpMinus10.eca Cb_HP-10



= 1 HpMinus10.eca Cb_HP-10

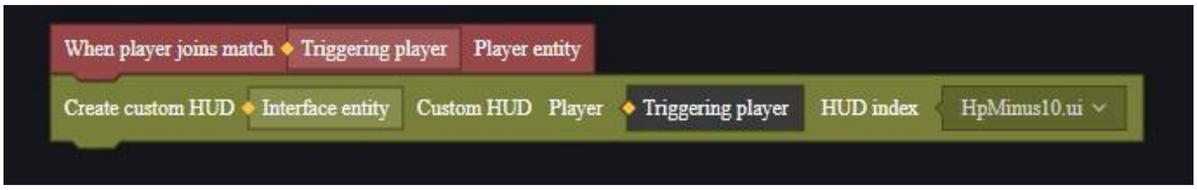
Global UI Management Script

The editing of the UI file then comes to an end, and you need to go and edit the creation and destruction conditions of that UI file. Create a script in the global module that will be used to manage the UI. When it comes to blood, choose to use a server script.



```
Script
= 1    Global_UI.eca
```

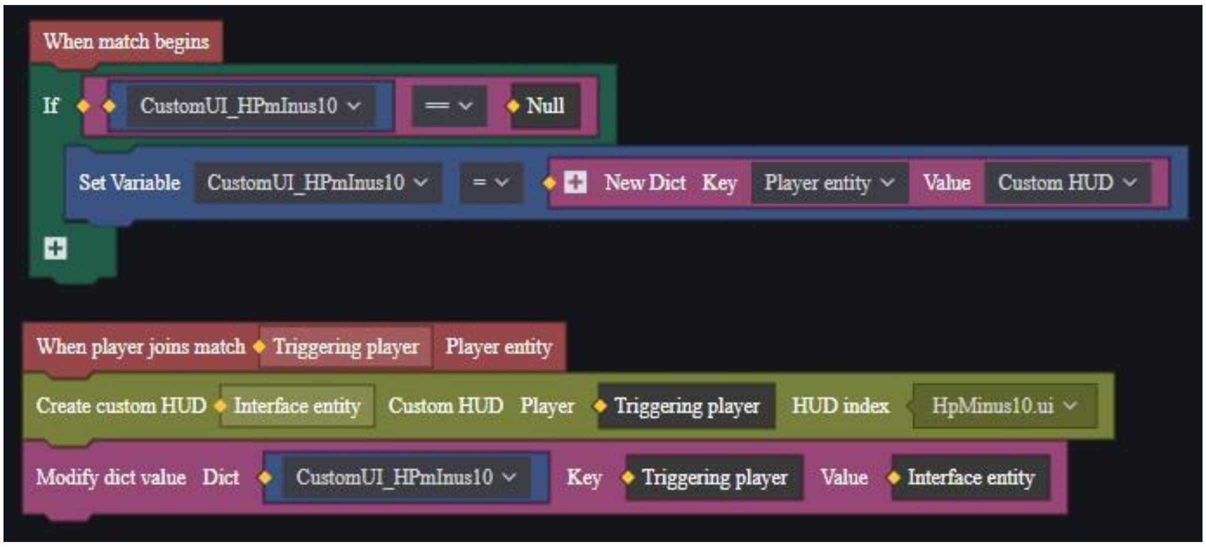
Create this UI file for each player when they join.



```
When player joins match ◆ Triggering player   Player entity
Create custom HUD ◆ Interface entity   Custom HUD   Player ◆ Triggering player   HUD index   HpMinus10.ui
```

You can also choose other points in time for creation, such as at the start of the game, or at the start of a phase. Choosing when the player joins will simply make it easier to obtain the player entity.

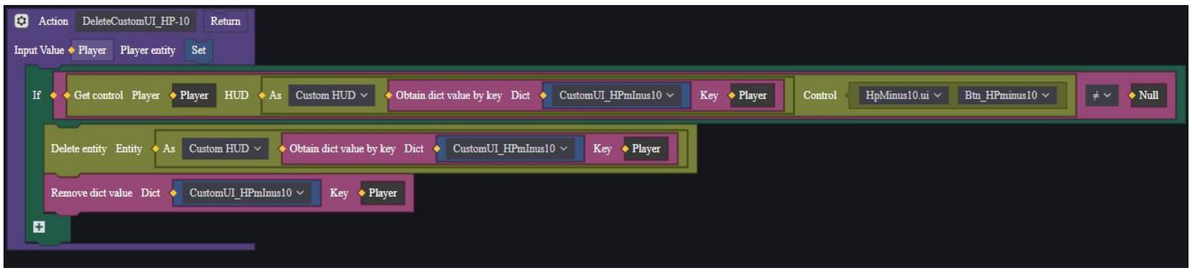
Create a dictionary to manage each player's blood reduction buttons



```
When match begins
If ◆ ◆ CustomUI_HPmInus10   ==   ◆ Null
    Set Variable   CustomUI_HPmInus10   =   ◆ New Dict   Key   Player entity   Value   Custom HUD

When player joins match ◆ Triggering player   Player entity
Create custom HUD ◆ Interface entity   Custom HUD   Player ◆ Triggering player   HUD index   HpMinus10.ui
Modify dict value   Dict ◆ CustomUI_HPmInus10   Key ◆ Triggering player   Value ◆ Interface entity
```

The destruction condition in the requirement is when the button is clicked, so you need to make a function that destroys the UI in the global UI management, which is triggered by an external call to the button.

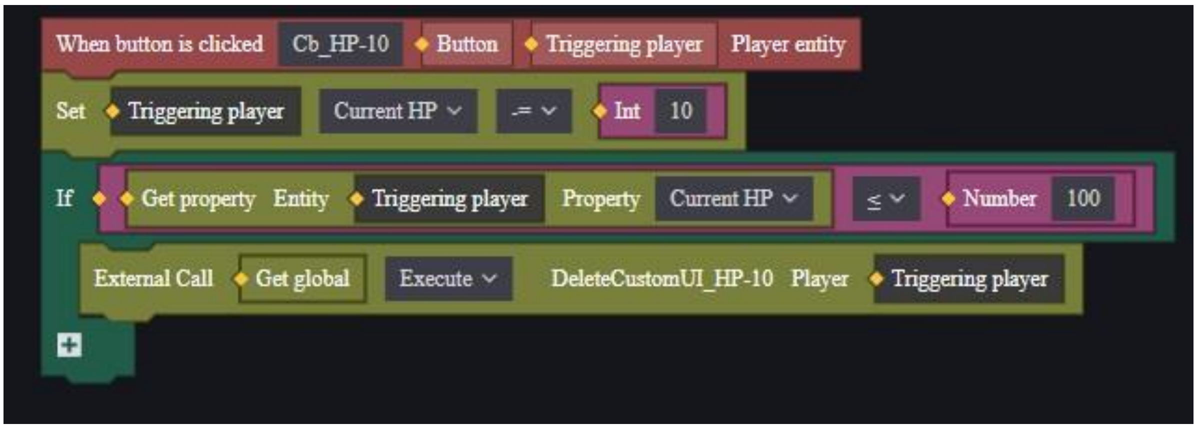


```
Action   DeleteCustomUI_HP-10   Return
Input Value ◆ Player   Player entity   Set
  If ◆ ◆ Get control   Player ◆ Player   HUD ◆ As   Custom HUD   ◆ Obtain dict value by key   Dict ◆ CustomUI_HPmInus10   Key ◆ Player   Control   HpMinus10.ui   Btn_HPminus10   ≠   ◆ Null
    Delete entity   Entity ◆ As   Custom HUD   ◆ Obtain dict value by key   Dict ◆ CustomUI_HPmInus10   Key ◆ Player
    Remove dict value   Dict ◆ CustomUI_HPmInus10   Key ◆ Player
```

1. Set the player as the input parameter.
2. Confirm that the custom UI has been removed by confirming that the player has a specific component.
3. If not, delete the custom UI.
4. Also clears the custom UI entities stored in the dictionary.

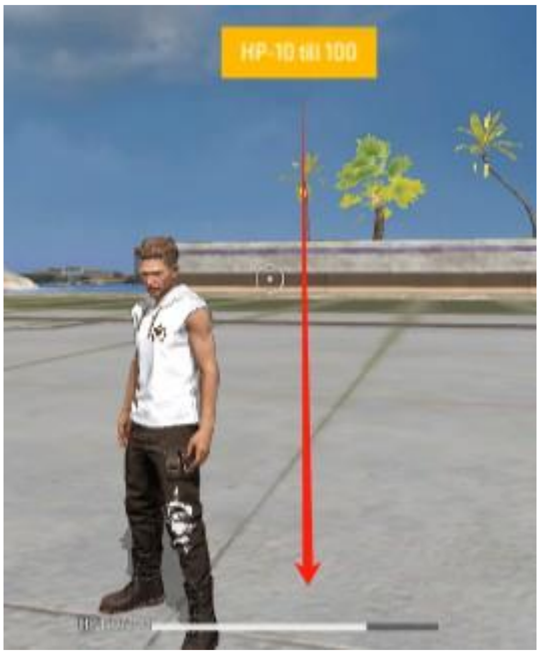
Add delete UI logic

Returns the UI root node script for editing, adding deletion conditions, and externally calling the delete custom UI function when satisfied.



```
When button is clicked  Cb_HP-10  Button  Triggering player  Player entity
Set  Triggering player  Current HP  -=  Int 10
If  Get property  Entity  Triggering player  Property  Current HP  ≤  Number 100
    External Call  Get global  Execute  DeleteCustomUI_HP-10  Player  Triggering player
+
```

on-the-spot survey


HP-10 till 100