Оглавление

Eva

Introduction

EvaArchitecture or **Eva** is based on model–view–viewmodel (MVVM) pattern and is the implementation of the message broker or event bus pattern. It is designed to use SOLID (object-oriented design).

Simply saying it implements Observer or Publish/subscribe pattern.

One publisher creates an event and the subscriber listen and reacts on it:

Events can be empty or contain data. This data is called Model. Model can be of any type. Often there are not one, but many subscribers on the event. They all listen the event and do their own actions. They all receive the Model of the event.

Eva also allows the publisher to get results from subscribers. This results are in form of List<object>. Subscribers do their work and add data to those results. The publisher waits for all subscribers to finish their work and receive results.

Eva also allows you not to wait for results and subscribers. This is known as FireAndForget.
Eva also allows you to Publish events using **async await, coroutine or parallel (multi-threaded).**

Eva does not create objects in memory for each new instance of event. It works very smartly and doesn't waste heap memory. All events are lightweight. You also don't need to care about the size of the stack segment.

Especially created for Unity3d engine, **Eva** has excellent optimization for slow mobile phones and devices.

Created for use in large projects, Eva works great with a large number of publishers and subscribers:

Eva does not use code generation and is compatible with a lot of frameworks.
Eva itself does not increase the compilation time.

Note: Eva is reference based implementation and does not serialize events when they are published or received. This significantly increases execution speed and also minimizes overhead.

Important: Eva does not store event class names in string format in asset files. Instead, it uses Unity's built-in mechanics to store references. This way you can move events from one namespace to another or change the event name and it won't cause errors, unlike other frameworks that store class names in files as serialized strings.

Eva Features

Publish & Subscribe Parallel events

Async await events

Coroutine IEnumerator events

Usual events

All in one event type: publish event and subscibe in Usual, Coroutine, Async, Parallel at the same time

Publish events with any model type or without it

Wait and receive result in List<object> from subscribers

Liteweight events, no event instances creation, no extra use of heap or stack

UiService included with often used components EvaButton, EvaText, EvaImage

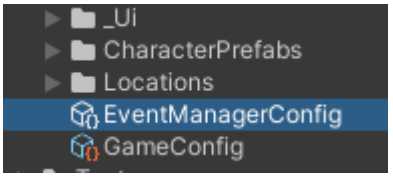
Fast Reference Finder included with additional mechanics to find events usages in the scenes, prefabs, components, and objects in event models.

Entity Component Service included and can use parallel feature of Eva events.

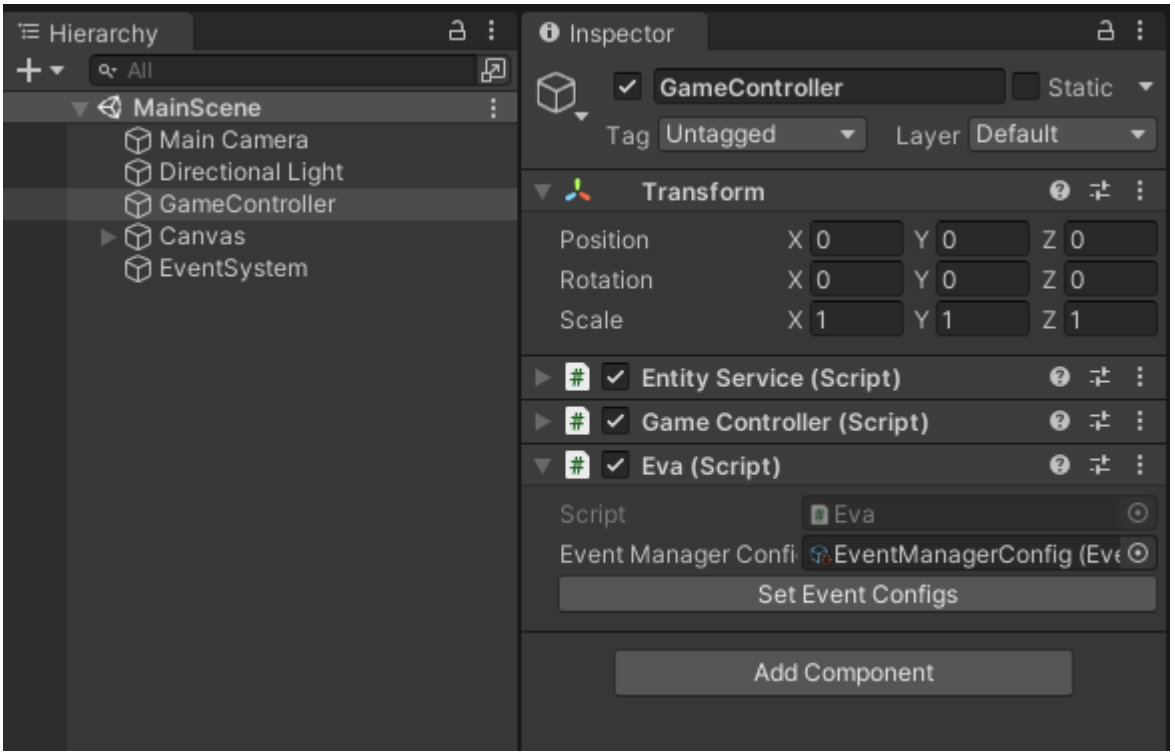
Log included which avoids overhead of string values calculations.

Adding Eva to the Project

First you need to create the EvaManagerConfig file using the context menu:



After that Add Eva component to the GameObject in a scene and set the reference to the config:



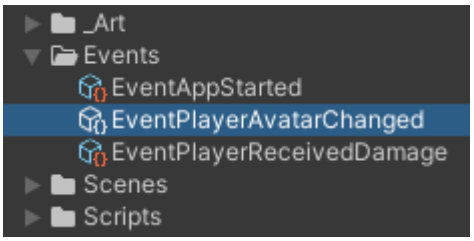
This config will store references to the events you create, but you can forget about it because there is a **Set Event Configs** button that will automatically add an existing event to this config. So just click it after creating the event. This config is simple List of references to the events. You can modify it as you want but there's no need to, since the button will do all the work of finding events and adding them for you.

Create the Event

First create the class for your new event, example:

```
[CreateAssetMenu(fileName = nameof(EventPlayerReceivedDamage),
    menuName = Menu.Events + nameof(EventPlayerReceivedDamage),
    order = Menu.EventsOrder)]
1 asset usage  10 usages  Roman Kuzmin  0+15 ext methods
public class EventPlayerReceivedDamage : EvaEvent<EventPlayerReceivedDamage>
{
}
```

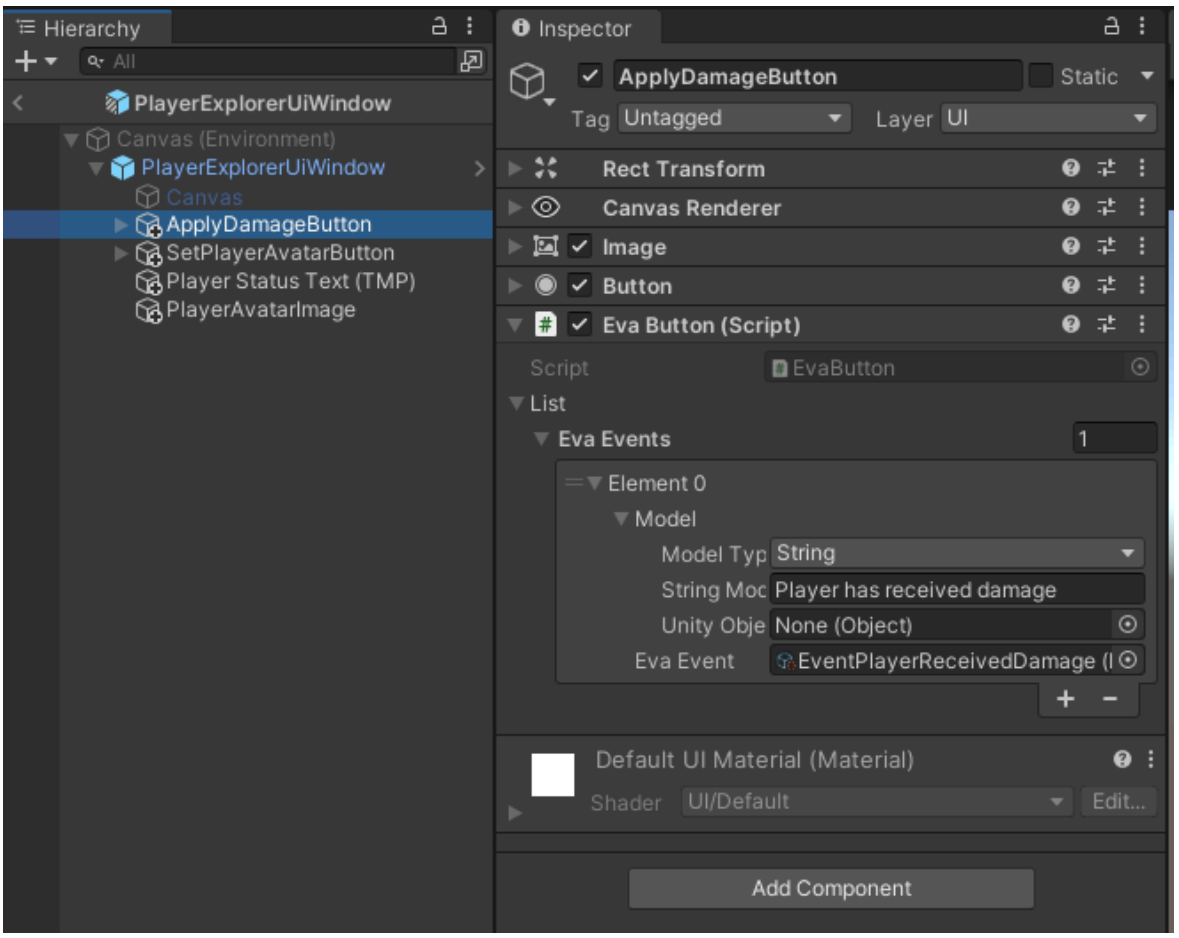
Open the context menu and create the asset for new event:



Now select the Eva component and press the **Set Event Configs.**
Congraits your event is now created!

Publish an Event

Create the Button and add the component EvaButton:



In the Eve Events field press the +plus icon. Drag and drop your event into the EvaEvent field.

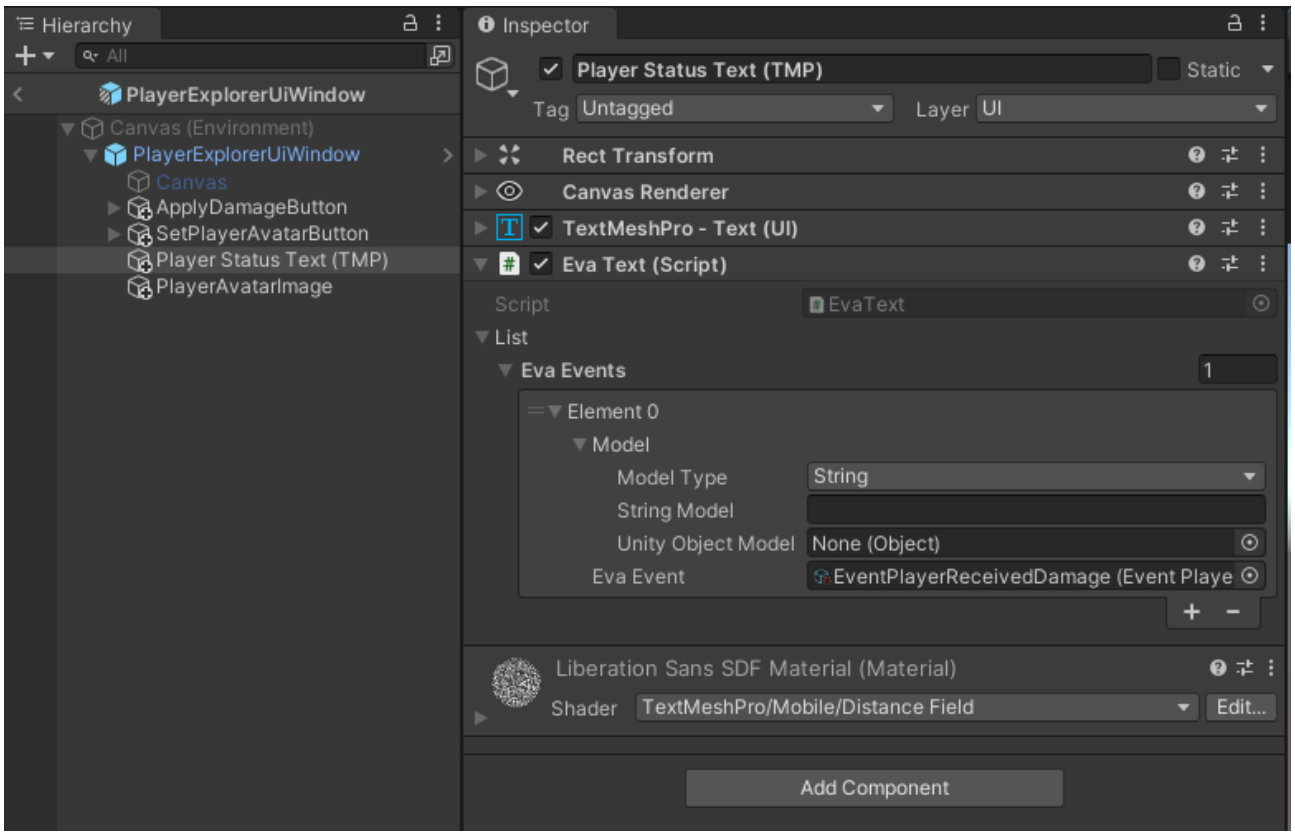
Select the event type, for example String and set the text in the string Model.

Tip: You can also select the UnityEngine.Object type and drag and drop any object into the Model. This way, when you publish a model, this object will be sent to all subscribers. Below we will see how to send the **texture** and set the player avatar **image** in this way.

Subscribe on Event

Create ui Text. It can be usual text or TextMeshPRO.

Now add component EvaText:

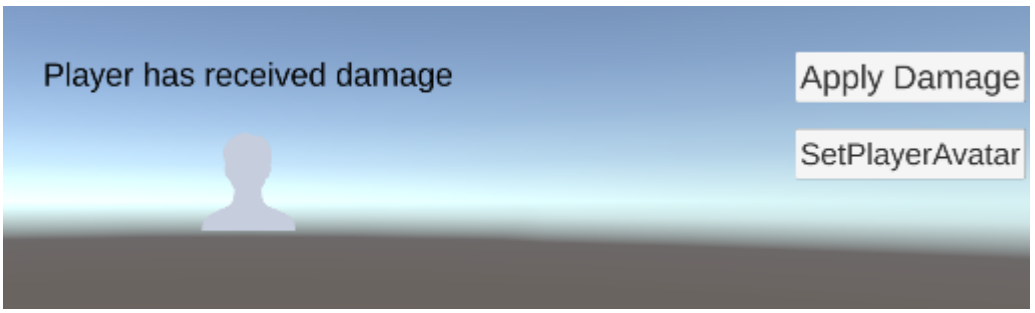


In the Eve Events field press the +plus icon. Drag and drop your event into the EvaEvent field.

Select the event type, for example **String**. There is no need to specify a model here because you will get the model in this event that you listen to in EvaText. When an event arrives, its model will contain text and its string will be automatically assigned to the text.

Test the Event

Press Play in Unity. Now select and click the button on Ui:



After you pressed the button the event will be published. There can be many subsribers on this event but we've created only one right now. So the text is listening on the event. Click

the button and the text will receive the event, its model and automatically change the text value.

Important: The user interface updates itself. It does not reference any scripts, methods, events, or callbacks in the code. This way you will avoid errors when changing a script, method or generic class. Moreover, if there is no publisher, subscribers will still be able to listen to events and this will not result in errors. An extra benefit is that when the publisher occurs, there is no need to set up callbacks in the UI since it automatically listens for events.

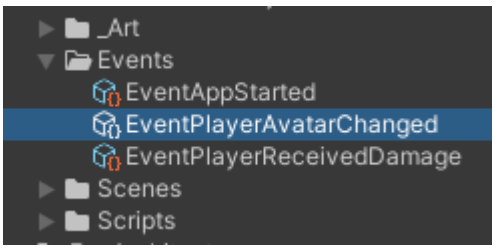
Important: Moreover, you can replace one publisher with another, for example, one service with another, without changing subscribers. And this can be done in rnuntime also! Cool, yeah.

Create a Texture Event to change the player's avatar image

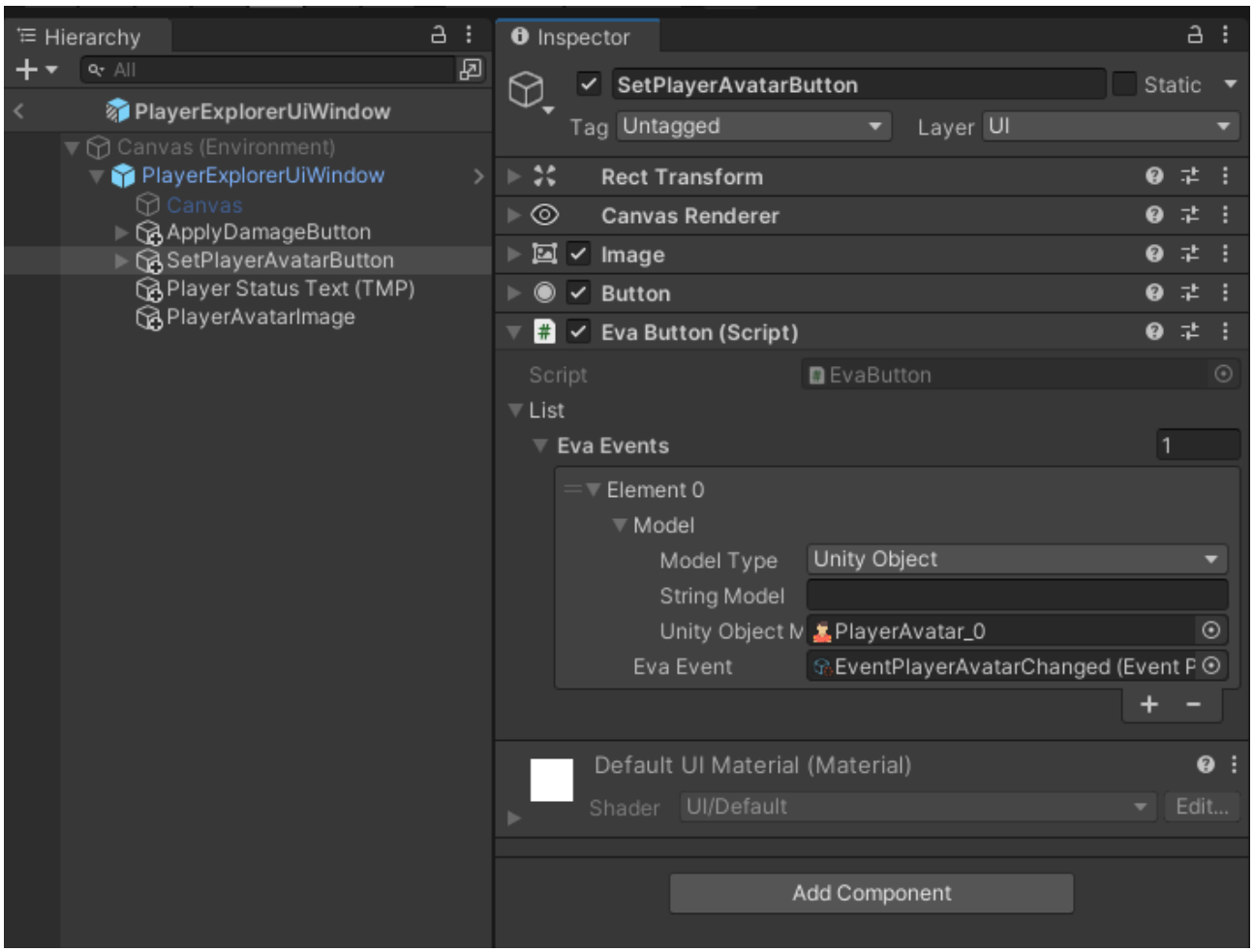
First create the class for event:

```
[CreateAssetMenu(fileName = nameof(EventPlayerAvatarChanged)
    , menuName = Menu.Events + nameof(EventPlayerAvatarChanged),
    order = Menu.EventsOrder)]
1 asset usage  3 usages  Roman Kuzmin  0+15 ext methods
public class EventPlayerAvatarChanged : EvaEvent<EventPlayerAvatarChanged>
{
}
```

Now create the event asset using the context menu:



Add the Button to the ui and add component EvaButton:

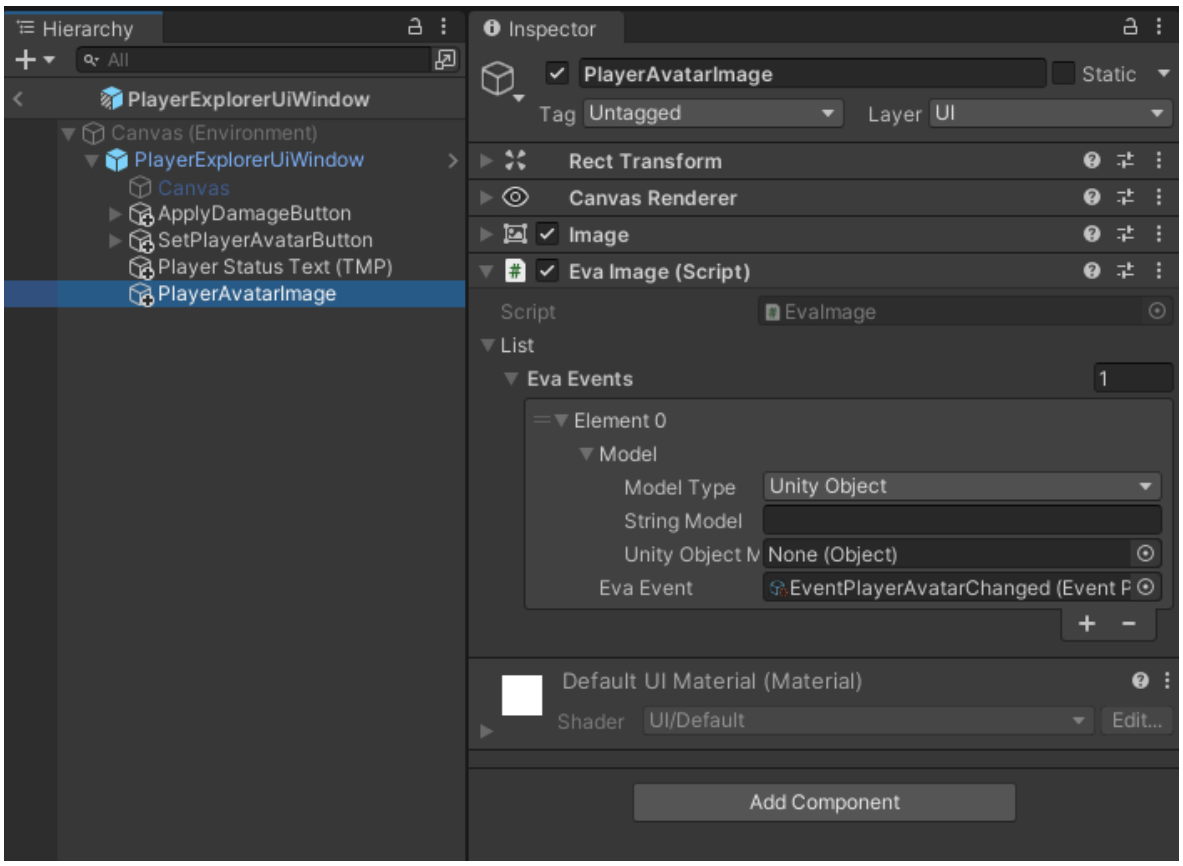


In the Eve Events field press the +plus icon. Drag and drop your event into the EvaEvent field.

Set the Model Type = Unity Object.

Now drag and drop texture into the field Unity Object Model.

Let’s create the Ui Image and add the component **EvaImage**:



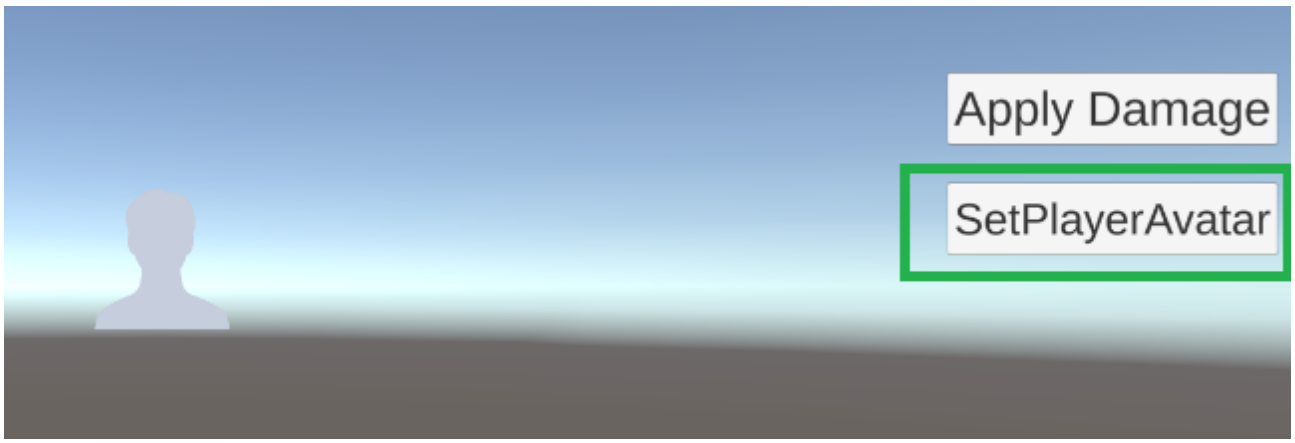
In the Eve Events field press the +plus icon. Drag and drop your event into the EvaEvent field.

Set the Model Type = Unity Object.

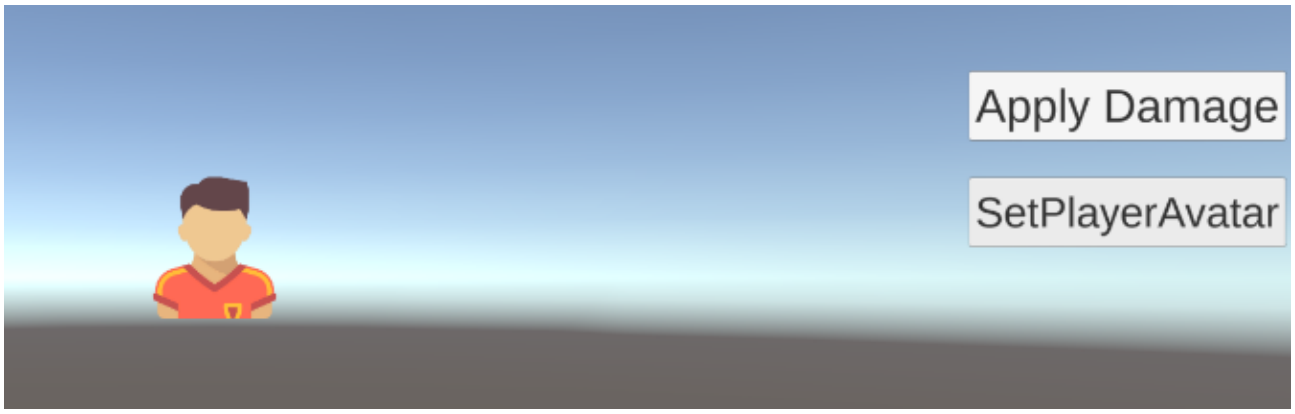
There is no need to specify a model here (in the field Unity Object Model) because you will get the model in this event that you listen to in EvaText. So leave the field empty: None (Object).

Press Ctrl + S to save your work.

Press Play in Unity. Now select and click the button on Ui:



The EvaImage will receive the model of the event and will automatically set the texture:



Publish an Event in the Code

Examples:

```
var results :List<object> = Eva.GetEvent<EventPlayerAvatarChanged>().Publish(_textureAvatar);
// or
results = Eva.GetEvent<EventPlayerAvatarChanged>().Publish(_spriteAvatar);
// or
results = Eva.GetEvent<EventPlayerAvatarChanged>().Publish(_imageAvatar);

Log.Info( message: () => $"results={results.Count}");
```

So we ask Eva to GetEvent and then simply Publish it with model. That's all we need to do.

You can send the unity Texture2D, Sprite or Image.

Defenition example:

using **UnityEngine.UI**;

```
[SerializeField] private Texture2D _textureAvatar;  Serializable
[SerializeField] private Sprite _spriteAvatar;  Serializable
[SerializeField] private Image _imageAvatar;  Unchanged
```

Here we use **[SerializeField] private** to restrict access to these fields from other classes. Of course we can use public:

```
public Texture2D _textureAvatar;  Serializa
public Sprite _spriteAvatar;  Serializable
public Image _imageAvatar;  Unchanged
```

But all other classes can change values in these fields. And that is why using the **[SerializeField] private** is much better. If you want, you can create properties to access them from other classes:

```
new * More...
public Texture2D TextureAvatar => _textureAvatar;
new *
public Sprite SpriteAvatar
{
    get => _spriteAvatar;
    set => _spriteAvatar = value;
}
new *
public Image ImageAvatar => _imageAvatar;
```

But in this example, there is no need to use these properties to publish events.

Tips: private fields has suffix underscore "_", for example, **_textureAvatar** to avoid extra using of word **this:**

```
[SerializeField] private Texture2D _textureAvatar;  Serializable

new *
public void MyMethod(Texture2D textureAvatar)
{
    // simplifying: this._textureAvatar = textureAvatar
    _textureAvatar = textureAvatar;
}
```

Create Courotine Event and Async Event

All is the same as for usual Event here. Simply add the word Coroutine or Async at the end to distinguish them. Example:

```
[CreateAssetMenu(fileName = nameof(EventPlayerCreateCoroutine)
    , menuName = Menu.Events + nameof(EventPlayerCreateCoroutine),
    order = Menu.EventsOrder)]
No asset usages  5 usages  0+15 ext methods
public class EventPlayerCreateCoroutine : EvaEvent<EventPlayerCreateCoroutine>
{
}
```

```
[CreateAssetMenu(fileName = nameof(EventPlayerCreateAsync)
    , menuName = Menu.Events + nameof(EventPlayerCreateAsync),
    order = Menu.EventsOrder)]
No asset usages  5 usages  0+15 ext methods
public class EventPlayerCreateAsync : EvaEvent<EventPlayerCreateAsync>
{
}
```

Publish Usual, Coroutine or Async

Usual:

```
var results:List<object> = Eva.GetEvent<EventAppStarted>().Publish((DateTime.Now, true));
```

Coroutine:

```
new *
private IEnumerator MyMethodCoroutine()
{
    var results = new List<object>();
    yield return Eva.GetEvent<EventAppStarted>().PublishCoroutine((DateTime.Now, true), results);

    Log.Info(message:() => $"results={results.Count}");
}
```

Async:

```
new *
private async void MyMethodAsync()
{
    var results:List<object> = await Eva.GetEvent<EventAppStarted>().PublishAsync((DateTime.Now, true));

    Log.Info(message: () => $"results={results.Count}");
}
```

Subscribe on the Event in the Code

Example:

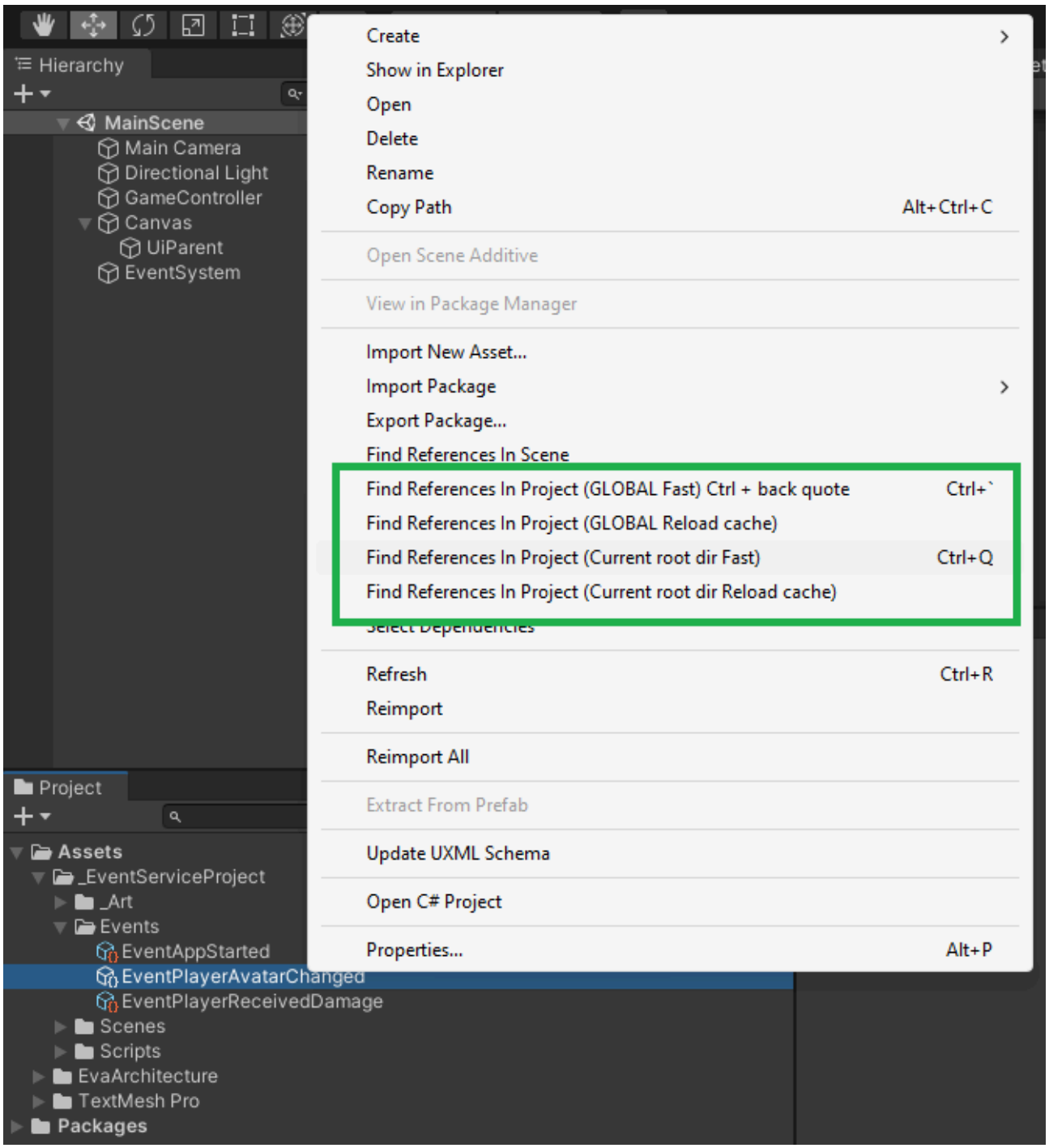
```
0+6 usages  Roman Kuzmin *
public override void Subscribe(bool mustSubscribe)
{
    base.Subscribe(mustSubscribe);

    Eva.GetEvent<EventAppStarted>().Subscribe(mustSubscribe, OnEvent);
    Eva.GetEvent<EventPlayerCreateCoroutine>().Subscribe(mustSubscribe, OnEventCoroutine);
    Eva.GetEvent<EventPlayerCreateAsync>().Subscribe(mustSubscribe, OnEventAsync);
}
```

Create methods to implement the logic:

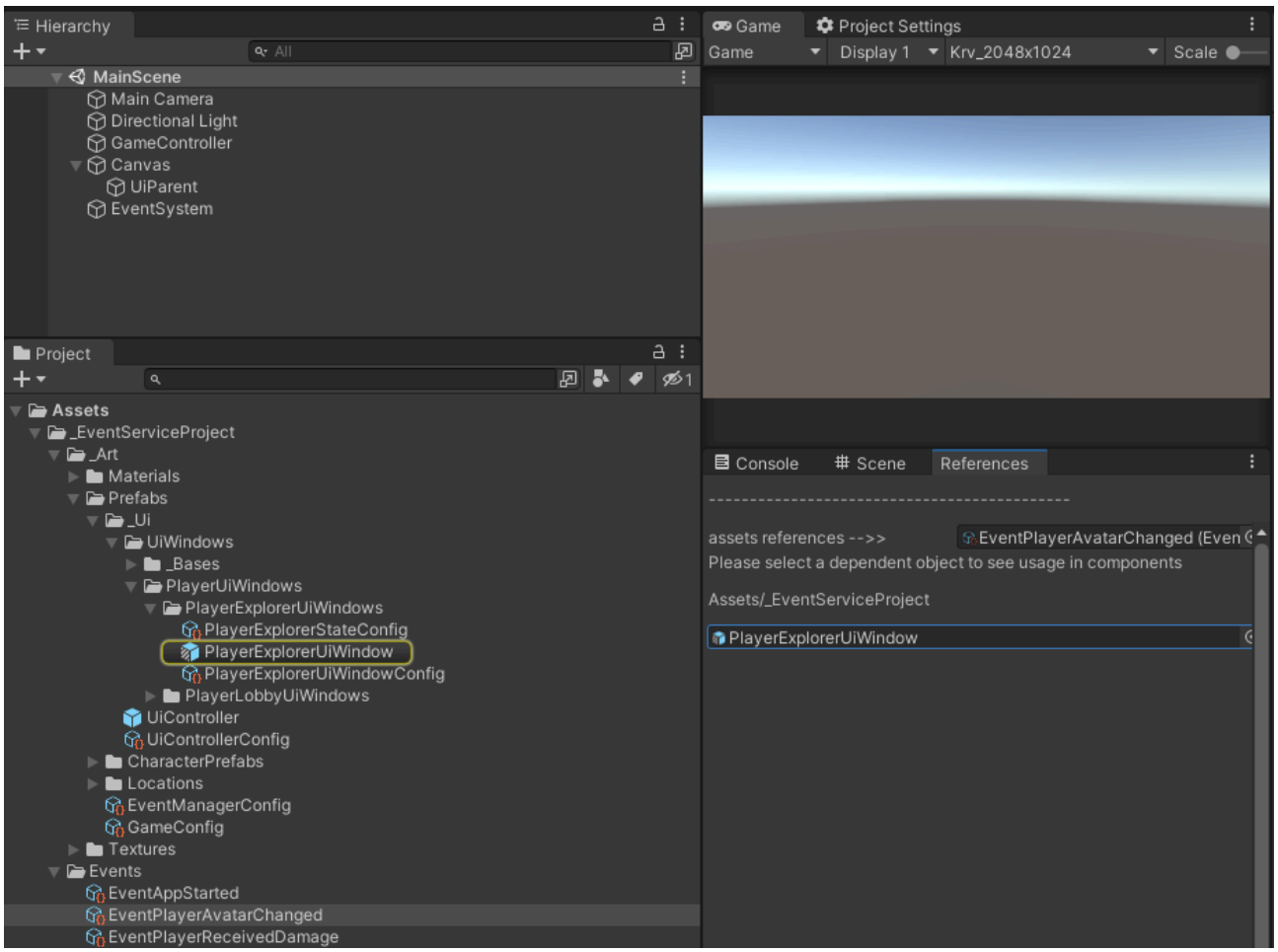
Here **Do(model, results)** is any method which does some logic.

```
// usual event Subscriber
Frequently called  1 usage  new *
private void OnEvent(EventAppStarted raiseEvent, (DateTime dt, bool b) model,
    List<object> results)
{
    Log.Info( message: () => $"PlayerController, GameEvents.App.StartedType, model={model}");
    Do(model, results);
    results.Add( item: $"Hi, I'm subscriber OnEvent(EventAppStarted), done my work");
}

// Coroutine event Subscriber
1 usage  new *
private IEnumerator OnEventCoroutine(EventPlayerCreateCoroutine raiseEvent, object model,
    List<object> results)
{
    yield return new WaitForSeconds(0.1f);
    Do(model, results);
    results.Add( item: $"Hi, I'm subscriber OnEventCoroutine(EventAppStarted), done my work");
}

// Async event Subscriber
Frequently called  1 usage  Roman Kuzmin *
private async Task OnEventAsync(EventPlayerCreateAsync raiseEvent, object model,
    List<object> results)
{
    await Task.Delay(TimeSpan.FromSeconds(0.1));
    Do(model, results);
    results.Add( item: $"Hi, I'm subscriber OnEventAsync(EventAppStarted), done my work");
}
```

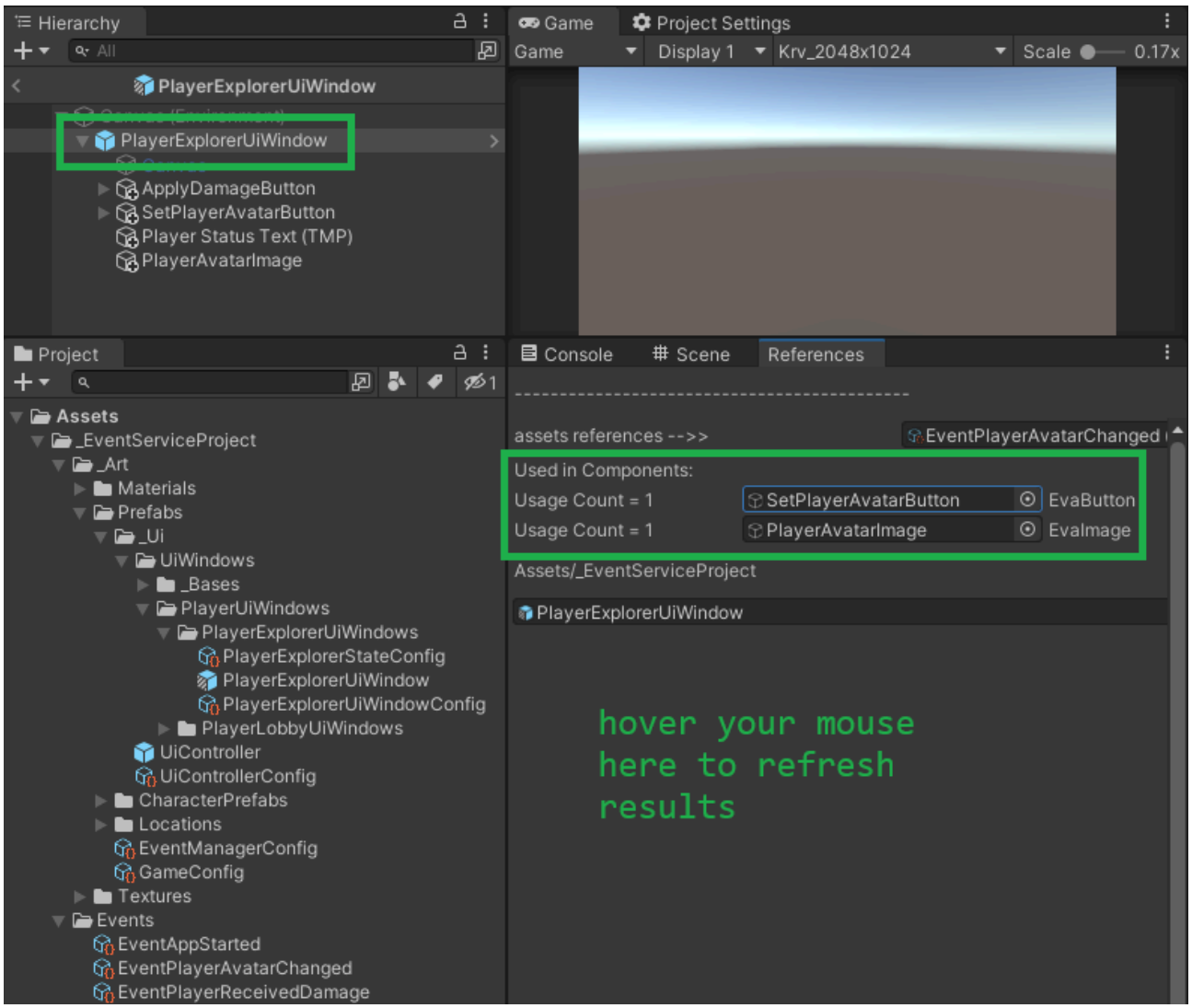
Reference and Event usages finder



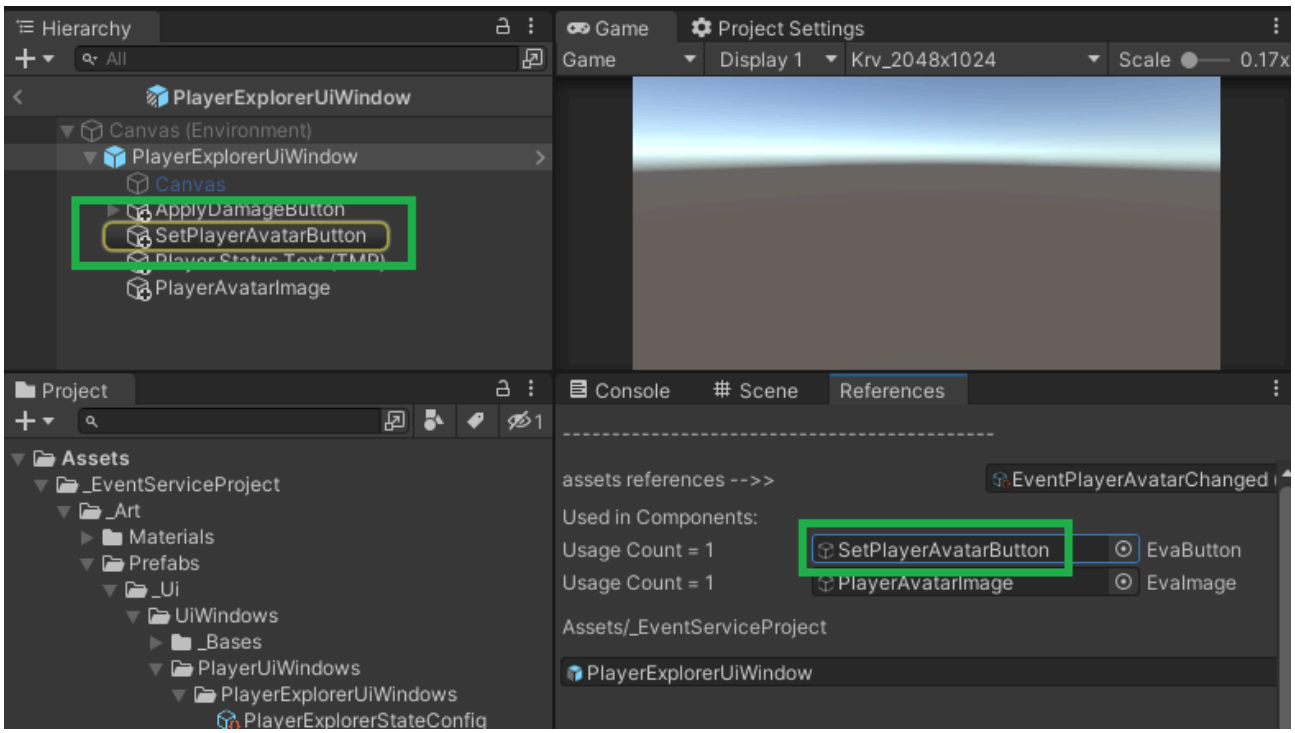
Searching results:



Select and enter into prefab to automatically find components where the event is used:



Select the found item to automatically navigate to it:

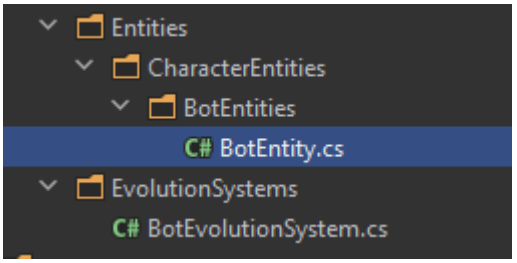


EntityComponentService usage example

Eva contains EntityComponentService which can be used to process a lot of objects at the same time. Use method **SafeForEach** or **SafeParallelForEach** (if you want to use process them in parallel threads).

Please note that you of course cannot access UnityEngine in a parallel model as Unity does not allow this. So the solution is to perform the calculation in parallel, then return the result and process it on the main thread, applying the changes to the unity objects.

Below is an example Main thread (not parallel case). Please contact me if you need an example for a parallel case.



```
[Serializable]
2 usages  Roman Kuzmin
public class BotEntity : BaseEntity<BotModel, BaseView>
{
    Roman Kuzmin
    public BotEntity() { }
    1 usage  Roman Kuzmin
    public BotEntity(BotModel model, BaseView view) : base(model, view) { }
}
```

```
2 usages  Roman Kuzmin
public class BotEvolutionSystem : BaseEvolutionSystem
{
    private const float MOVE_SPEED = 10f;

    Frequently called  0+2 usages  Roman Kuzmin
    public override void Update()
    {
        base.Update();
        if (EntitiesContainer.IsNull())
            return;

        var stopwatch = new Stopwatch();
        stopwatch.Start();
        var objectsCount = 0;

        EntitiesContainer.SafeForEach( action: it :BaseEntity =>
        {
            if (it.IsNull()
                || !(it is BotEntity botEntity))
                return;

            if (!botEntity.Enabled)
                return;

            var view = botEntity.View;
            if (view.IsNull())
                return;

            if (view is MonoBehaviour monoBehaviour
                && !monoBehaviour.gameObject.IsNull())
            {
                var tr :Transform = monoBehaviour.gameObject.transform;
                var pos :Vector3 = tr.localPosition;
                tr.localPosition = new Vector3(
                    x: pos.x + (UnityEngine.Random.Range(-MOVE_SPEED, MOVE_SPEED) * Time.deltaTime),
                    pos.y,
                    z: pos.z + (UnityEngine.Random.Range(-MOVE_SPEED, MOVE_SPEED) * Time.deltaTime));

                objectsCount++;
            }
        }, onError: exception => Log.Error( message: () => $"BotEvolutionSystem, Update", exception: ()=> exception));

        stopwatch.Stop();
        Log.Info( message: () => $"BotEvolutionSystem, Update, processed objectsCount={objectsCount}"
                    + $" in elapsedMilliseconds={stopwatch.ElapsedMilliseconds}");
    }
}
```

Contacts and license

This is commercial, non-free software.

If you want to use it for free or commercial purposes (or for any other purpose), you need to buy it.

This software may not be resold or redistributed in any way without the consent of the author.

You may not decompile, disassemble or modify this software.

Author: Roman Kuzmin, jetacore@gmail.com

Licence: https://unity.com/legal/as-terms