Signaler Documentation

Elegant decoupling solution for Unity projects.

Introduction

What is Signaler?

Signaler is a system that will allow you to decouple your code, isolating each part so that it does not have dependencies on others. Why is this good? When your code has many dependencies (couplings), this can make it difficult to change your project, especially as your code base grows ever larger. When you have coupled code, it is more difficult to work in teams of people that need to focus on specific chunks of the project without worrying about the changes other members are performing. Coupled code can lead to spaghetti code, which is a term that means unstructured and difficult to maintain code.

Once you have structured your code on a centralized signal system, you only have a single dependency for all your game objects: the signals themselves. Signals allow you to work on a section of code without any reliance on what else is happening in the game.

Example

Let's say you use a tranditional coupling method in Unity, linking a spaceship in your game to the FX system, the UI, the sound system, and a game manager. These connections are fairly straightforward and you can do most of them directly from Unity's inspector windows once you have your scripts set up correctly. But now your spaceship needs to know about every system that will be affected by changes in its state. This might seem trivial, but as more systems come into play, you will have to continually grow the number of dependencies the ship will need to know about. Further, if you are working in a team, they will need to constantly adjust their code as other parts change.

Now imagine all the other objects and systems in your game besides this one ship. They all need to be linked together, creating more and more couplings, and reducing the ease of making changes later.

Enter Signaler.

Instead of linking your spaceship to all the other systems in the game, you instead just broadcast signals from the ship. It no longer cares who hears what it has to say, it just knows that it needs to say it. When the ship is destroyed, for instance, instead of telling the FX manager to make an explosion, the audio manager to make a sound, the UI to update the interface, and the game manager to change the state, it only has to broadcast a signal that it was destroyed. The FX, audio, UI, and game manager will all subscribe to this signal, performing their updates as needed. Any one of them can drop out of this subscription later, if needed, without affecting the ship's dependencies. The signal is the only point of shared dependency in the new structure. Teams can work independently on each system, knowing that all they need to respond to, or broadcast out, are signals.

Another Example

Let's say you are making an Asteroid clone game. You have dozens of asteroids in the scene and an asteroid manager script that controls creating and destroying them. In a traditional Unity manner, you would probably instantiate the asteroids, then link them to their manager. When the asteroid is destroyed, you would then notify its manager to clean up the scene and remove the asteroid directly from the asteroid script. This works well, until you decide that you actually want to have sub-managers to control different asteroid sizes. You then have to adjust the asteroid to link to these sub-managers, which in turn now report to a main asteroid manager script. This might seem easy to do, but now you have broken other dependencies that were expecting the main asteroid manager to handle the cleanup and instantiation, further requiring you to make more changes to your code.

Using Signaler in this scenario, the asteroids would not need to know who manages them. All they need to do is send out a signal that they are destroyed when hit. Either the main asteroid manager or the sub-manager, or any other system (FX, sound, UI, etc) can subscribe to this signal and act accordingly. The asteroid is complete and does not need to change its dependencies as the game evolves. It has become a modular and loosely-coupled entity.

Signaler parts

There are four main parts to the Signaler system:

1. **Signaler** singleton. This system routes broadcasts to all subscribers in the order the subscribers have chosen.
2. **Signals**: These are classes or structs that convey information from a broadcaster to subscriber(s). Signals can be either a message or a request. Messages are one way information from broadcaster to subscriber(s). Requests are two-way, with a signal sent from the broadcaster to subscriber(s) and then a separate response sent back from the subscriber(s) to the broadcaster. Signals are usually immutable (they do not change after they are created), so structs provide the better alternative to classes. Structs are not allocated and will not be garbage collected, which is especially good for mobile environments.
3. **Broadcaster**: A class that sends out a signal. Information about who received the broadcast is sent back back to the broadcaster, whether it was a message or request signal.
4. **Subscriber**: A class that listens to and handles a particular signal broadcast.

Classes can be both broadcasters and subscribers.

Broadcasters

Broadcasters send out signals, either to let other parts of the project know that something happened, or to request information that it needs. Broadcasters inherit from the IBroadcaster interface to allow the Signaler manager to keep track of who sent out a signal.

Example of a broadcast:

```
var subscriberCount = Signaler.Instance.Broadcast<MySignal>(this, signal:
signalData, group: 11);
```

where the class calling the broadcast inherits from IBroadcaster, passing itself as the first argument. The signal type (struct or class) is of MySignal type in this example. The signal variable is an object created from the MySignal class or struct. The signalData variable is optional and is null by default (used often in triggers that need no further information). The group 11 is used to filter the signal to only subscribers of this group number. Group number is optional and null by default.

The Broadcast, whether it is a message or request will return the number of subscribers that heard the signal.

Subscribers

Subscribers listen to signals, handling them in their own methods. Subscribers inherit from the ISubscriber interface to allow the Signaler manager to notify the broadcaster who received the signal.

Example of a subscription:

```
Signaler.Instance.Subscribe<MySignal>(this, OnMySignal, group: 11, order: 234);
```

where the class subscribing to the broadcast inherits from the ISubscriber interface, passing itself as the first argument. The signal type (struct or class) is of MySignal type in this example. The method that will handle the signal is OnMySignal. This example uses the group number 11 and has order 234. The group and order are optional, using null and zero as defaults, respectively.

Groups allow subscribers to listen to signals only when their particular group is broadcast. This can be useful in separating large numbers of object, such as game entities.

Ordering allows you to control who hears the signal broadcast first. Ordering goes from negative numbers to positive numbers (example: -300, -134, 0, 5, 12, 200, 5430).

Handling:

When a subscriber handles a signal, it passes back a boolean value that indicates that it handled the message. If true is passed back, then the broadcaster will know this subscriber heard the signal. If false is passed back, the broadcaster will not know the subsciber heard the signal.

Example of handling:

```
private bool OnMySignal(MySignal signal)
{
        Debug.Log(name + " received signal. Message = '" + signal.message + "'");
        return true;
}
```

This example handles signals of type MySignal. The signal has a message member that is used in debugging to the console. The method returns true to let the broadcaster know the signal was received.

Message Signals

Message signals are one-way signals that are sent from broadcaster to subscriber(s). They can be a class or struct and contain any number of members, methods, and sub-classes / sub-structs.

Example 1

```
public struct MySignal
{
    public string message;
}
```

This simple signal only contains a string message. Example:

```
Signaler.Instance.Broadcast<MySignal>(this, signal: new MySignal { message = "Hello
World!" });
```

Example 2

```
public struct SimpleTrigger { }
```

This signal has no members, so it would be used like a trigger. In these special cases, the broadcast does not need to specify any data for the signal. Example:

```
Signaler.Instance.Broadcast<SimpleTrigger>(this);
```

Request Signals

Request signals are two-way from broadcaster to subscriber(s), back to broadcaster. They consist of two parts, the signal going out and the response coming back. Both the outgoing signal and the response can be either a class or a struct. All subscribers hear the signal before returning responses back to the broadcaster.

Example 1

This example will use a trigger struct (no data) to ask for color information, returning a simple struct as a response from the subscriber(s).

```
public struct GetColorSignal { }

public struct GetColorResponse
{
    public Color theColor;
}
```

Example broadcaster usage:

```
var responses = new List<SignalResponse<GetColorResponse>>();
if (Signaler.Instance.Broadcast<GetColorSignal, GetColorResponse>(this, out
responses) > 0)
{
        Debug.Log(responses[0].response.theColor);
}
```

This broadcast sends a trigger signal and returns a list of response. If there is at least one response, then it will output the first one to the console.

Example subscriber usage:

```
void Awake()
{
    Signaler.Instance.Subscribe<GetColorSignal, GetColorResponse>(this, OnGetColor);
}

private bool OnGetColor(GetColorSignal signal, out GetColorResponse response)
{
    response = new GetColorResponse() { theColor = this.color; };
    return true;
}
```

Example 2

This example will ask for a specific entity's position, if it meets the criteria.

```
public struct GetEntityPositionSignal
{
    public bool isMoving;
}

public struct GetEntityPositionResponse
{
    public Vector3 position;
}
```

Example broadcaster usage:

```
var responses = new List<SignalResponse<GetEntityPositionResponse>>();
Signaler.Instance.Broadcast<GetEntityPositionSignal,
GetEntityPositionResponse>(this, out responses, signal: new GetEntityPosition {
isMoving = true; }, group: 1234);
foreach (var r in responses)
{
    Debug.Log(r.subscriber + " position = " + r.response.position);
}
```

This will broadcast out a request to get an entity's position, sending the isMoving criteria to be true. Only entities in group 1234 will receive this request. It will then iterate through all the responses, debugging the subscribers and their positions back to the console.

Example subscriber usage:

```
private bool isMoving;

void Awake()
{
    Signaler.Instance.Subscribe<GetEntityPositionSignal,
GetEntityPositionResponse>(this, OnGetEntityPosition, group: 1234);
}

private bool OnGetEntityPosition(GetEntityPositionSignal signal, out
GetEntityPositionResponse response)
{
    if (signal.isMoving == this.isMoving)
    {
        response = new GetEntityPositionResponse() { position = transform.position;
};
        return true;
    }
    else
    {
        response = null;
        return false;
    }
}
```

This will subscribe to the GetEntityPosition signal only for group 1234. It handles the signal, returning a position response only if the isMoving member of the signal matches the subscriber's isMoving member (in this case only if the entity is moving).

Notes:

In this particular example, the signal is filtered twice, once by the group and once by the internal handling method's isMoving check. The group filter takes place inside the Signal manager and is far more efficient for large groups. It uses a dictionary lookup to filter the signal, where filtering inside the handler means that each subscriber needs to be called and then checked. For thousands of subscribers, this could be slow, so if you only need a small group (or even one) subscriber to see a message, try to use groups for speed. This can be particularly useful if you are just trying to reach one entity or system where there are many subscribers to the signal.