Event System - Dispatcher

Foreword

Thank you for purchasing the Event System - Dispatcher!

I'm an independent developer and your feedback and support really means a lot to me. Please don't ever hesitate to contact me if you have a question, suggestion, or concern.

Tim
tim@ootii.com

Contents

Overview

Event and message dispatching is a critical part of any game. As games get more and more complex, so too are the interactions between objects. Message dispatching ensures that game objects are able to communicate in a consistent, reliable, and efficient way.

Dispatcher does this and makes it easy too! Simply tell the Dispatcher what an object wants to listen for. When another object sends that message to the Dispatcher, the Dispatcher will ensure all the "listeners" are notified.

With this dispatcher, you can create your own custom messages and take advantage of the message pool for super-fast performance.

Features

The Dispatcher supports the following features:

- Create and send custom message types
- Send messages containing custom data
- Send messages to everyone
- Send messages to objects based on their name
- Send messages to objects based on their tag
- Send messages to objects based on a string filter
- Send a message immediately
- Send a message at the next frame
- Schedule a message for the future
- Connect multiple listeners to a single message
- High performance message pooling

Setup

The Dispatcher is a code-based asset. That means to take advantage of its capabilities, you need to access it through C#. There are no inspectors or scripts to add to Game Objects (other than the game logic you need for your specific situation).

So, setup is pretty simple:

1. Create a new project or open an existing one

Follow the standard practice for creating or opening a scene. Nothing special here.

2. Import this package to your project

Import the package files from the Unity Asset Store.

How It Works

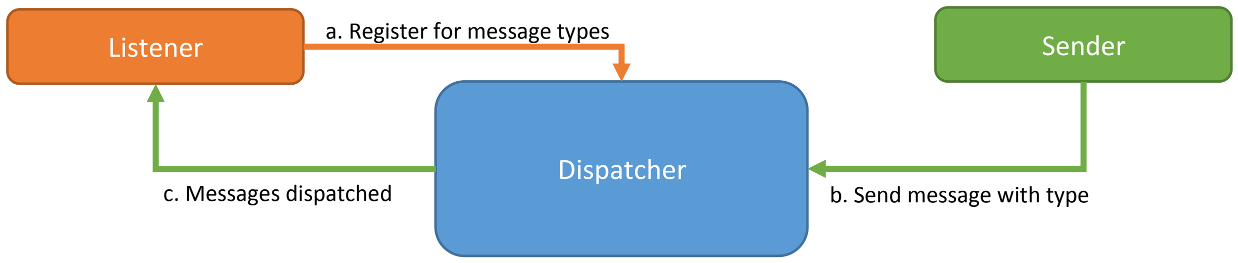
The event system has three main parts:

1. The Dispatcher
This is a static or 'Singleton' that actually takes messages and either sends them right away or holds them and sends them later. It's the core of the whole system.

2. Listeners
Listeners are Game Objects or classes that expect to get messages. They tell the dispatcher that they are interested in a specific type of message and then are notified when that message comes in.

3. Senders
Senders are Game Objects or classes that send messages to the dispatcher so listeners can get them. You can be a listener and a sender at the same time.

When coding, there's really only two things you need to do; register listeners and send messages. I've included several versions of each function in order to allow you to streamline code, but at their core they all do the same thing.



Here's a simple example of everything working together. It's a bit hokey since I'm sending a message to myself, but it shows all the pieces working together…

You'll see we include '**using com.ootii.Message;**' at the top. This way the code knows about the whole event system.

Then, we register a listener with '**AddListener'** and send a message with '**SendMessage'**.

The 'OnStarted' function that we registered with the listener is what gets called when the message type 'STARTED' comes in.

```
using UnityEngine;
using System.Collections;
using com.ootii.Messages;

public class Sample : MonoBehaviour
{
    // Use this for initialization
    void Start()
    {
        // Here an object registers a function to listen for the "STARTED" message.
        // When the "STARTED" message is sent (by any object), the OnStarted function
        // is called.
        //
        // Notice the 'true' as the last argument. This was added since the
        // SendMessage is called immediately after the add listener. Listeners are
        // added at the end of the frame by default.
        // That would be too late for the 'SendMessage' below. So, we ignore the delay
        // by telling the
        // listener to be added immediately. Normally, you'd leave it off.
        MessageDispatcher.AddListener("STARTED", OnStarted, true);

        // This is a goofy example since we don't typically need to send a message
        // to ourselves. However, imagine we were calling this from a totally different
        // object that didn't know about this one.
        //
        // Here we send the "STARTED" Message.
        MessageDispatcher.SendMessage(this, "STARTED", "Whoo Hoo!", 0);
    }

    /// Raised by the MessageDispatcher when the "STARTED" message is recieved.
    void OnStarted(IMessage rMessage)
    {
        Debug.Log((string)rMessage.Data);
    }
}
```

AddListener

Caching Listeners

When you add a listener to the dispatcher, the listener isn't typically added right away. Instead, we cache the request until the end of the frame. I do this so you can add listeners as a response to an incoming message.

However, this means you can't immediately send a message unless you tell the dispatcher to ignore the cache. In fact, the example above does that exact thing. You'll see more about this in a second.

Message Handlers

When a message comes into the dispatcher, the dispatcher sends it off to the listeners. Under the hood, what the dispatcher is doing is calling a function that the listener said to call. In the example above, the "OnStarted" function is that function.

Notice, the function is based on a delegate and has a special signature:

```
public delegate void MessageHandler(IMessage rMessage);
```

The signature shows it has one argument "IMessage". This message object contains all the information about the message the listener is getting:

- Type
- Sender
- Delay
- Data

It also allows the listener to set a property:

- IsHandled (more on this later)

So, at its most basic form, AddListener simply says what type of message to listen for and what function will be called when that type of message comes in.

Filters

Most of the time, you probably won't need to worry about filters as the Message Type will determine what listeners get what messages. However, for advanced users, there is a filter value that can be set. Filters can be used in a couple of different ways:

1. As a basic string to add another level to the message type.
2. As a GameObject's name to specify a specific recipient.
3. As a GameObject's tag value to specify a specific recipient.

For the last two, the filter is based on the "RecipientType" property of the MessageDispatcher. This property says that when a GameObject filter is used, it's either treated as the recipient's "name" or "tag".

See below for how these filters are actually used.

AddListener Signatures

There are several different version of AddListener on the MessageDispatcher class. They all do the same thing, but I've added the function variations to allow you to write less code.

If you're using a modern IDE, the argument descriptions will pop-up so you can tell the difference between the different versions.

AddListener(string rMessageType, MessageHandler rHandler)
rMessageType = String representing the message type to look for. You can set this to anything you want.
rHandler = The function that will be called when the listener is being told about an incoming message

`AddListener(string rMessageType, MessageHandler rHandler, bool rImmediate)`
rMessageType = String representing the message type to look for. You can set this to anything you want.
rHandler = The function that will be called when the listener is being told about an incoming message.
rImmediate = Tells the dispatcher not to cache the add, but to do it immediately.

99% of the listeners will be added with one of the two functions defined above. However, there's some advanced features you may find useful.

AddListener(string rMessageType, string rFilter, MessageHandler rHandler)
rMessageType = String representing the message type to look for. You can set this to anything you want.
rFilter = String representing a second layer to determine if this listener will get the message.
rHandler = The function that will be called when the listener is being told about an incoming message.

`AddListener(string rMessageType, string rFilter, MessageHandler rHandler, bool rImmediate)`
rMessageType = String representing the message type to look for. You can set this to anything you want.
rFilter = String representing a second layer to determine if this listener will get the message.
rHandler = The function that will be called when the listener is being told about an incoming message.
rImmediate = Tells the dispatcher not to cache the add, but to do it immediately.

AddListener(UnityEngine.Object rOwner, string rMessageType, MessageHandler rHandler)
rOwner = The GameObject whose name/tag will be used to determine if this listener will get the message.
rMessageType = String representing the message type to look for. You can set this to anything you want.
rHandler = The function that will be called when the listener is being told about an incoming message

AddListener(UnityEngine.Object rOwner, string rMessageType, MessageHandler rHandler, bool rImmediate)
rOwner = The GameObject whose name/tag will be used to determine if this listener will get the message.
rMessageType = String representing the message type to look for. You can set this to anything you want.
rHandler = The function that will be called when the listener is being told about an incoming message.
rImmediate = Tells the dispatcher not to cache the add, but to do it immediately.

RemoveListener

Just as you can add a listener, you can remove a listener so it will no longer get messages.

The signatures for RemoveListener pair with those of AddListener. So, I'm not going to list them all over again. Needless to say, for each AddListener there is a RemoveListener function.

Like AddListener, RemoveListener is cached until the end of the frame. This way, you can remove a listener in response to a message and not cause an error. This can be ignored with the "rImmediate" argument.

SendMessage

Once you have listeners setup, you're ready to send messages. Like with AddListener, there are several variations of the function in order to simplify code. However, they all work pretty much the same.

Message Type

When you send a message, the most important argument is typically the message type. This is just a string that you can set to any value you'd like and it acts as a basic filter.

In the example above, I created a listener to look for the message filter "STARTED" and then I send a message whose type was "STARTED". This is what allows the dispatcher to know which listener gets which message.

A word on string types

Some developers hate strings and prefer to use hashed values or simple int values. While it's true, there is a performance benefit to doing this, we're typically talking tiny fractions of a millisecond.

As long as we don't manipulate the strings, there's little to no impact on the garbage collector as well.

Since the majority of users on the Asset Store aren't hard-core programmers, I decided to go with strings for readability. Even with this approach, I'm finding that the Event System – Dispatcher is 30% to 100% faster than the standard Unity event system.

Delay

Messages can be sent with a delay. This allows you to send a message to the dispatcher immediately, but not have it delivered until the next frame or for a specific number of seconds.

To have the message delivered with a delay, use:

- "0" to send the message immediately
- "-1" to send the message next frame
- A value > 0 to send the message in that number of seconds

Data

Along with the message itself, you can add data to the message. Since this argument is an "object", you can add any value you want. The listener can then interrogate the message and use the data as needed.

Filter

As a mentioned in the AddListener section, advanced users can use filters to help determine which listeners get which messages. Filters can be used in a couple of different ways:

1. As a basic string to add another level to the message type.
2. As a GameObject's name to specify a specific recipient.
3. As a GameObject's tag value to specify a specific recipient.

Custom Messages

For advanced users, the Event System – Dispatcher allows you to create entirely new message structures.

I use a C# interface (IMessage) to define the basic message type, this is what allows us to set the type, delay, etc. However, you may have a situation where you want to create a whole new message structure. All you need to do is inherit from either the Message class or the IMessage interface. Once you do this, you can send messages using the basic “SendMessage(IMessage rMessage)” signature.

SendMessage Signatures

There are several different version of SendMessage on the MessageDispatcher class. They all do the same thing, but I’ve added the function variations to allow you to write less code.

SendMessage(string rType)
rType = String that determines which listener gets the message.

SendMessage(string rType, string rFilter)
rType = String that determines which listener gets the message.
rFilter = Additional string to help determine which listener gets the message.

SendMessage(string rType, float rDelay)
rType = String that determines which listener gets the message.
rDelay = Seconds to wait before sending the message (-1 means send next frame).

SendMessage(string rType, string rFilter, float rDelay)
rType = String that determines which listener gets the message.
rFilter = Additional string to help determine which listener gets the message.
rDelay = Seconds to wait before sending the message (-1 means send next frame).

SendMessage(object rSender, string rType, object rData, float rDelay)
rSender = Object who is sending the message.
rType = String that determines which listener gets the message.
rData = Additional data to send along with the message.
rDelay = Seconds to wait before sending the message (-1 means send next frame).

SendMessage(object rSender, object rRecipient, string rType, object rData, float rDelay)
rSender = Object who is sending the message.
rRecipient = GameObject whose name or tag will be used to determine who gets the message
rType = String that determines which listener gets the message.
rData = Additional data to send along with the message.
rDelay = Seconds to wait before sending the message (-1 means send next frame).

SendMessage(object rSender, string rRecipient, string rType, object rData, float rDelay)
rSender = Object who is sending the message.
rRecipient = string to use as an addition filter to determine who gets the message
rType = String that determines which listener gets the message.
rData = Additional data to send along with the message.
rDelay = Seconds to wait before sending the message (-1 means send next frame).

SendMessage(IMessage rMessage)
In the end, all the SendMessage signatures call this version.

You can use this if you have a custom message structure. Simply set your values on the IMessage object and then pass the message to the dispatcher using this function.

Cleaning Up Messages

Once the dispatcher sends off a message, it needs to clean that message up. This is especially true for delayed messages that the dispatcher held on to.

The way it does this is that it looks at two flags on the message: IsSent and IsHandled.

While processing, if the dispatcher sees the message flag IsSent is true or the flag IsHandled is true, the message will be deleted.

IsSent is set by the Dispatcher when it sends the message. IsHandled can be set by you as the listener is processing the message. I also find setting the IsHandled flag is helpful when you've got multiple listeners handling the same message. If one of them sets this flag, the other listener doesn't need to process the message.

This kind of "IsHandled" logic is for you to code as needed.

Advanced Example: Delay

I'm using code similar to the first example. Again, it's sort of hokey since I'm sending a message to myself. However, it keeps the code really short.

In this example, I want to use a delay of 5 seconds to send messages to listeners. So, I do the following:

1. Use AddListener.

2. Use SendMessage with the delay.

Results

When the message is sent, the dispatcher will wait 5 seconds and then send the message.

Code

```
using UnityEngine;
using System.Collections;
using com.ootii.Messages;

namespace com.ootii.Demos.MD
{
    public class Sample1 : MonoBehaviour
    {
        // Use this for initialization
        void Start()
        {
            // Add a listener
            MessageDispatcher.AddListener("WON", OnStarted, true);

            // Here we send the "WON" Message.
            MessageDispatcher.SendMessage("WON", 5f);
        }

        /// <summary>
        /// Raised by the MessageDispatcher when the "STARTED" message is recieved.
        /// </summary>
        void OnStarted(IMessage rMessage)
        {
            Debug.Log((string)rMessage.Data);
        }
    }
}
```

Advanced Example: Filter

I'm using code similar to the first example. Again, it's sort of hokey since I'm sending a message to myself. However, it keeps the code really short.

In this example, I want to use a filter as an additional condition to send messages to listeners. So, I do the following:

1. Use AddListener with a filter.

2. Use SendMessage with the filter.

Results

When the message is sent, there could be lots of listeners looking for the message type of "WON". However, only the listeners who were registered with the filter will get the message.

Code

```
using UnityEngine;
using System.Collections;
using com.ootii.Messages;

namespace com.ootii.Demos.MD
{
    public class Sample1 : MonoBehaviour
    {
        // Use this for initialization
        void Start()
        {
            // Add a listener
            MessageDispatcher.AddListener("WON", "Filter1", OnStarted, true);

            // Here we send the "WON" Message.
            MessageDispatcher.SendMessage("WON", "Filter1");
        }

        /// <summary>
        /// Raised by the MessageDispatcher when the "STARTED" message is recieved.
        /// </summary>
        void OnStarted(IMessage rMessage)
        {
            Debug.Log((string)rMessage.Data);
        }
    }
}
```

Advanced Example: Name Filter

I'm using code similar to the first example. Again, it's sort of hokey since I'm sending a message to myself. However, it keeps the code really short.

In this example, I want to use the filter to send messages to objects with a specific name. So, I do the following:

1. Use AddListener with a GameObject to register listeners based on the GameObject they belong to. This store's the game object's name as the filter.

2. Use SendMessage with the GameObject we want to send the message to.

Results

When the message is sent, there could be lots of listeners looking for the message type of "WON". However, only the listeners who were registered with the Game Object's name will get the message.

Code

```
using UnityEngine;
using System.Collections;
using com.ootii.Messages;

namespace com.ootii.Demos.MD
{
    public class Sample1 : MonoBehaviour
    {
        // Use this for initialization
        void Start()
        {
            // Add a listener
            MessageDispatcher.AddListener(gameObject, "WON", OnStarted, true);

            // Here we send the "WON" Message.
            MessageDispatcher.SendMessage(this, gameObject, "WON", "Whoo Hoo!", 0);
        }

        /// <summary>
        /// Raised by the MessageDispatcher when the "STARTED" message is recieved.
        /// </summary>
        void OnStarted(IMessage rMessage)
        {
            Debug.Log((string)rMessage.Data);
        }
    }
}
```

Advanced Example: Custom Message

In the “Assets\MessageDispatcher\Demos” folder, you’ll find the “MyCustomMessage.cs” file. This is an example of how to create your own message class.

You’ll notice that it inherits from “Message”, but includes new fields: MaxHealth and CurrentHealth.

In the “UI.cs” file (line 141), you’ll see how we instantiate the custom message instance and send it. Below is that code:

```
MyCustomMessage lMessage = MyCustomMessage.Allocate();
lMessage.Type = "CUSTOM";
lMessage.MaxHealth = 100;
lMessage.CurrentHealth = 50;
lMessage.Sender = this;
lMessage.Data = Color.blue;
lMessage.Delay = 1.0f;
MessageDispatcher.SendMessage(lMessage);
```

Nothing too complicated… we simply create the message using the object pool, set the values we want, and use MessageDispatcher.SendMessage to send it.

In this example, the listener who gets the message would need to release the message so we don’t suck-up resources. The UI.cs file does that on line 286.

Support

If you have any comments, questions, or issues, please don't hesitate to email me at support@ootii.com. I'll help any way I can.

Thanks!

Tim