NOTE

This documentation is for the Keiser M3i Multi-Bike Receiver Unity Wrapper. If you aren't working with the Multi-Bike Receivers (little bluetooth-enabled boxes that send M3i data over a wired or wireless LAN) then this isn't the library for you. Perhaps you were after a library for connecting the bike to your phone? I'm making that too, but it's not done yet!

Introduction

This is a fairly short documentation, as the actual wrapper itself is fairly easy to use. The best way to explain how to use this wrapper is to explain my BikeDemo scene supplied with the library. If you didn't import that demo with the package, I recommend doing so!

Using the Wrapper

Setting the Scene

In order to use the wrapper, you need only a couple of objects in the scene:

A Game Object with the *KeiserManager* MonoBehaviour added to it. This is your main manager script and the wrapper can't work without it. It tells the wrapper how to configure itself and also handles the listeners that handle bike additions, removals and updates:

Bike Listeners: This is a List of BikeListener implementations, more on these below.

Start On Awake: Tells the manager whether to start listening for packets on awake, or whether to wait for your code to tell it to start (or you can press the buttons "Start Listening" and "Stop Listening" in the editor).

Listening IP and ***Port:*** This is the multicast IP/Port to listen on for packets from the receivers. By default the M3i Receivers multicast on 239.10.10.10, port 35680. Multicast addresses should match the configuration of the receivers and must be in the range of 235.0.0.0 to 239.255.255.255 to work. The port isn't as important, but try to use a unique one. Make sure these are configurable if you're shipping an application, most places probably don't leave these as the defaults!

Bike Timeout: This is the amount of time (in milliseconds) to wait before assuming a bike has stopped completely. If no packets with the specific UUID are seen within this period of time then the bike will be removed from the list and you should handle this removal appropriately in your BikeListener implementation(s).

The other Game Object you need is an object with a listener on. This can be the same object as your KeiserManager or a completely different one. To get a better idea of how the listener works, check out the SimpleBikeListener supplied with the demo. In a nutshell though, a listener must do the following:

1.) Implement the *BikeListenerInterface* interface and also extend *MonoBehaviour*
2.) Implement *gotNewBike(KeiserBike)*, *lostBike(KeiserBike)* and *updatedBike(KeiserBike)*
3.) Be added to the *KeiserManager* instance under the *Bike Listeners* List.

KeiserBike is pointer-based within the listener. When a bike is added, a new instance is created. When a bike is lost or updated, the same instance of *KeiserBike* will be passed to the method. This is done through the UUID of the bike (**not the bike ID**) to uniquely identify it from all other bikes no matter what Bike ID. The Bike ID is in the range of 1 to 150, with the potential for multiple bikes within the same facility to have the same ID, hence Bike ID should be user-facing only and not used to identify bikes internally.

KeiserBike

The *KeiserBike* class is a data store for an individual bike, and stores both the current data (under nested class instance bikeData) and also the delta data (under the nested class instance bikeDeltas) so you can monitor the change in values too. For a full run-down of what the class contains, take a look at the KeiserBike.cs file.

The Bike Demo

BikeDemo uses a very simple Bike Listener called *SimpleBikeListener*, which takes a prefab, the *KeiserManager* instance and contains an empty list of the *BikeCube* script, which is specific to this demo and not part of the wrapper itself. It simply creates instances of the provided prefab (KeiserCube), with the *BikeCube* script, which contains an instance of KeiserBike when created (as they are created on *gotNewBike*) and has a *BikeDidUpdate* method. An important thing to note is that *BikeDidUpdate* does not pass the bike instance through. As stated above, the same instance is used on each update for that bike's UUID, so when the bike instance updates it also updates within *BikeCube.* You can of course do as you please with passing through data, this just demonstrates that you don't need to!

BikeDidUpdate demonstrates a very basic movement mapping, translating the rpm into a velocity. This isn't a very accurate representation of RPM into motion of course, but demonstrates what can be done.

That's it!

I did say it was easy. If you have any questions, you can contact me via Twitter: @IrregularExpr, or post in the support thread for this asset. Alternatively, drop me an email: assets@benwoodford.co.uk

Troubleshooting & FAQ

I don't have a Keiser M3i, how can I develop for it?

Keiser have helpfully developed a set of tools for working with the Multi-Bike system, including a debugger for viewing the data, a simulator for simulator a receiver sending data to a certain IP/Port, and a finder for viewing receivers on the network. You can find these tools here.