

# FluentMotion - a Unity Asset for Gesture Detection using LeapMotion and Reactive Programming

**Radu Petrisel**

Technical University of Cluj-Napoca  
28, G. Baritiu Street, 400027  
Cluj-Napoca, Romania  
radupetrisel@gmail.com

**Adrian Sabou**

Technical University of Cluj-Napoca  
28, G. Baritiu Street, 400027  
Cluj-Napoca, Romania  
adrian.sabou@cs.utcluj.ro

## ABSTRACT

This paper presents an approach to creating a human readable, flexible and extendable API for the LeapMotion controller. The API (*FluentMotion*) is built using a variety of modern technologies, such as C# LINQ (Language Integrated Query), Reactive Extensions [1] and SteamVR [4]. *FluentMotion* comes as an extension of the basic LeapMotion API, making it easy to integrate in projects using LeapMotion. The API works on the Unity Game Engine, which provides the means of creating Virtual Reality applications, defining new gestures (based on the already existing basic ones) and creating callbacks for the gestures used in the application. *FluentMotion* works on both HTC Vive and Oculus Rift for VR situations, but can also be used in desktop mode. The issue this API addresses is the rudimentary API offered by LeapMotion and the need for a more powerful and flexible way of detecting and composing hand and finger gestures.

## Author Keywords

Gesture detection; Leap Motion; Virtual Reality; Unity; Reactive Extensions;

## ACM Classification Keywords

H.5.m. Information Interfaces and Presentation (e.g. HCI): Miscellaneous

## GENERAL TERMS

Virtual Reality; Gesture detection; API

## INTRODUCTION

The concept of virtual reality has been around since the late 20th century, with many prototypes being developed as early as the 1960s. In 2016, the first two consumer headsets were released by HTC and Oculus. Since then, the Virtual Reality industry grew exponentially, being used in a variety of applications ranging from entertainment to space exploration.

One of the main shortcomings of the current virtual reality setups is the handheld controllers, which, to many, feel unnatural and unintuitive to use. To solve this issue, LeapMotion released Orion [6] in 2016 for its already existing LeapMotion Controller. The controller is a USB device meant to capture hand and finger motions without actually touching it.

LeapMotion's basic API for Unity is straightforward, extensible and very well integrated with the game engine, but it lacks flexibility and complex compositions. Thus, the need for a more advanced API arised. *FluentMotion* appeared as a response to this need.

By using *FluentMotion*, a virtual reality developer can easily define gestures in a human-readable way that is both composable and flexible. These traits of the API are due to it being based on Reactive Extensions.

Reactive Extensions is an implementation of the observer [5] and iterator design patterns. They are available in many popular languages, such as C#, Java, C++ and Swift. Reactive Extensions use functional programming in order to reduce the amount of boilerplate code one has to write.

*FluentMotion* was built for Unity Game Engine, a very popular game engine in our days. This increases the usability of the API, as it can be easily downloaded and integrated in any Unity project straight from the built in Unity Package Manager.

The user simply adds the LeapMotion "prefab" to his project, then builds the *FluentMotion* hands rig on the LeapMotion one. On the new rig, the user can add any gestures (as Unity scripts) he/she wishes to detect for the hand the script is attached to.

The base gestures are defined in abstract classes that must be extended. Only one method needs to be implemented, namely the *OnDetect* callback. Once this is done, the detector ready for usage.

For testing purposes, an application using *FluentMotion* exists. The application features a cube with an icon on it. The icon represents one of the eight possible gestures the user has to do in order for the cube to disappear: fist, "L" (index & thumb extended), thumbs-up, pinch or swipe (either left, right, up or down). Once the player performs the correct gesture, the icon changes to a new one representing a different gesture the cube expects.

The API was partially inspired from *ReactiveUI* [2], a .NET framework for MVVM applications based on Reactive Extensions. Some of the gesture detectors' syntax is based on the one of *ReactiveUI*'s *ReactiveObject*.

Like the LeapMotion controller, *FluentMotion* is intended only for hand gestures that can be performed in the user's field of

view. This is the main limitation of LeapMotion in terms of usability.

One of the shortcomings of the API is the detection of complex moving gestures. As of the current implementation, the only moving gestures that can be detected are simple swipes.

## THE LEAPMOTION CONTROLLER



Figure 1. The leapmotion controller

The LeapMotion controller is a device that consists of two stereo cameras which track infrared light with a wavelength of 850 nanometers (allowing it to work even in dark rooms) [3].

The device has a large interaction space, about  $0.37 m^3$ . Its range is limited by LED light propagation through space, which is roughly 60cm from the sensor [3].

After the hardware does its job of recording the images, the software starts doing some heavy mathematical lifting. Despite what most users think, the LeapMotion controller uses raw sensor data for tracking, not depth maps.

The LeapMotion service is responsible for processing this sensor data. Every application that uses LeapMotion has a reference to an implementation of this service, either for Virtual Reality or for desktop mode. First, the service removes background objects and compensates for ambient lighting, and then reconstructs a 3D representation of the raw device data.

The tracking layer then extracts information from the 3D representation and feeds these results as frames to a transport protocol. From thereon, each application uses this frames as input.

On June 11, 2018, LeapMotion released the latest generation of Orion - version 4. It has been in beta since, but it came with major improvements over the past iterations of LeapMotion's tracking software. These include:

- increased range of the sensor from 60 to 80cm
- faster hand initialization
- better hand pose stability and reliability
- more accurate shape and scale for hands

## REACTIVE EXTENSIONS

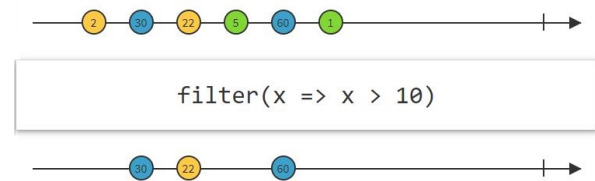


Figure 2. Example of a RX operator

ReactiveX is a powerful library for asynchronous and event-based programming. It is an implementation of the observer pattern meant for event-driven programming. It also extends the observer pattern with operators that allow the user to compose sequences declaratively without worrying about low-level concerns (such as multithreading and the problems that come with it).

Figure 2 shows how an operator works on an observable. In the example, the operator is *filter*. *Filter* takes as input a predicate, a function that maps a value to a boolean (true or false). So, from the source observable [2, 30, 22, 5, 60, 1], by filtering the elements greater than 10, we are left with only [30, 22, 60]. Note that the elements are emitted in the same order that they were in the source, almost instantly. The vertical line at the end represents the end of the observable stream. One can attach a callback to that, called *OnComplete*.

The main data structure used by ReactiveX is *Observables*. As stated on their intro page:

You can think of the Observable class as a “push” equivalent to Iterable, which is a “pull.” With an Iterable, the consumer pulls values from the producer and the thread blocks until those values arrive. By contrast, with an Observable the producer pushes values to the consumer whenever values are available. This approach is more flexible, because values can arrive synchronously or asynchronously. (ReactiveX intro)

The following figures illustrate the resemblance between the iterable and observable.

### Iterable

```
getDataFromLocalMemory()
    .skip(10)
    .take(5)
    .map({ s -> return s + " transformed"
    })
    .forEach({ println "next => " + it })
```

Figure 3. Iterable

## Observable

```
getDataFromNetwork()
    .skip(10)
    .take(5)
    .map({ s -> return s + " transformed"
    })
    .subscribe({ println "onNext => " + it
    })
```

Figure 4. Observable

One might say that the only difference is the call to *subscribe* instead of *forEach*. While, indeed, both of the code snippets produce the same result, the real difference is the data flow.

In the *forEach* example, the thread is blocked until 15 elements arrive from the *getDataFromNetwork* call (first 10 are skipped, then only 5 are processed by the *map*).

In the *subscribe* example, the only delay in the thread's execution is the creation of the observable stream, after which other instructions are executed. When data arrives from the *getDataFromNetwork*, the thread which created the observable is interrupted and data is processed.

## LEAPMOTION GESTURES

The LeapMotion API defines mappings for four human body parts:

- **arm** (from elbow to wrist) - has one hand attached
- **hand** - has five fingers attached
- **finger** - has three joints (for attaching objects) and four bones
- **bone**

LeapMotion offers a variety of gesture detectors already implemented, which can also be combined by the use of a Logic Gate. The logic gate is a higher level detector, combining two or more basic detectors.

As an example, a "thumbs up" gesture would be detected as combination of the following detectors:

- Finger Extended Detector - configured to detect a thumb extended and other fingers not extended
- Finger Pointing Detector - configured to detect that the thumb is pointing up (*Vector3(0, 0, 1)*) relative to the horizon
- And Logic Gate - to combine the other two detectors and have callbacks (C# scripts) attached to it

This approach requires adding three components to a game object and referencing the first two detectors (Finger Extended Detector and Finger Pointing Detector) from the Logic Gate.

This can quickly get out of hand when requiring a high number of combined gestures.

## FLUENTMOTION GESTURES

From LeapMotion's human body parts, *FluentMotion* makes use only of **hands** and **fingers**, and defines the following basic gestures.

### Finger Gestures

- *IsExtended* - selected finger is extended
- *IsPointingTo* - selected finger is pointing to a given target (Unity GameObject or a hand)
- *AreExtended* - selected fingers are extended (the others are marked as *don't care*, so they could be extended or not)

### Hand Gestures - single hand

- *IsPinching* - hand is pinching (as of Orion 4.4, i.e. *when PinchStrenght > 0.8*)
- *PalmIsFacing* - palm is facing a given target (can be any object that has a mapping to a Unity *Vector3*) with a given angle tolerance
- *IsFist* - hand is making a fist (i.e. *FistStrenght > 0.8*)
- *IsMoving* - hand is moving in a given direction (expressed as a Unity *Vector3*) with a given speed (in millimeters per second) and angle tolerance (for the direction)

### Hand Gestures - both hands

- *PalmsAreFacing* - both palms are facing a target object or, if no object is given, facing each other with a given angle tolerance
- *AreMakingFists* - both hands are making fists
- *AreMoving* - both hands are moving in a given direction (*Vector3*) and with a given angle tolerance

*FluentMotion* also supports selecting only some fingers from a hand for extra processing, like checking which is extended and which is not or more complex predicates like finger pointing in a dynamically changing direction.

Besides the already defined gestures, users can create their own gesture detectors by implementing the *IReactiveDetector* or by inheriting from one of its three base implementations: *ReactiveFingerDetector*, *ReactiveHandDetector* or *ReactiveHandsDetector*.

The main advantage of *FluentMotion* is that all gestures - basic and user defined - can be chained indefinitely. Through chaining, more complex gestures can be defined, like swiping right with your left hand while your thumb is up and your index is pointing towards some game object or towards the sky.

## USING FLUENTMOTION

### Setup

After having the LeapMotion and SteamVR set up, adding *FluentMotion* to your application is done in two simple steps:

1. Add three new empty game objects anywhere in the scene. It is recommended, but not mandatory, to make two of the objects children to the 3<sup>rd</sup> (as in figure 5)



Figure 5. Setting up *FluentMotion* game objects

2. Attach a *ReactiveHand* script to two of the objects, referencing either the left or the right hand from the LeapRig, and attach a *ReactiveHands* script to the 3<sup>rd</sup>, referencing the other two *ReactiveHands*

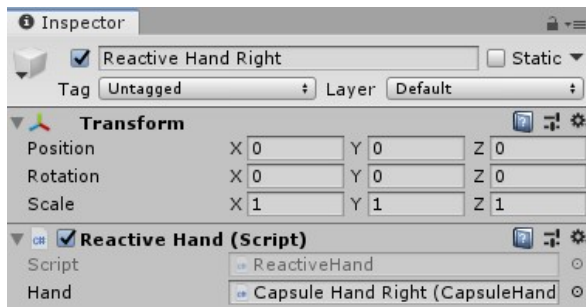


Figure 6. The *ReactiveHand* script attached to the game object

### Creating Detectors

To create a detector, extend the *ReactiveHandDetector* class in your own script. As an example, a possible implementation for the "L" gesture - thumb and index are extended, and the others are not - is shown in figure 7.

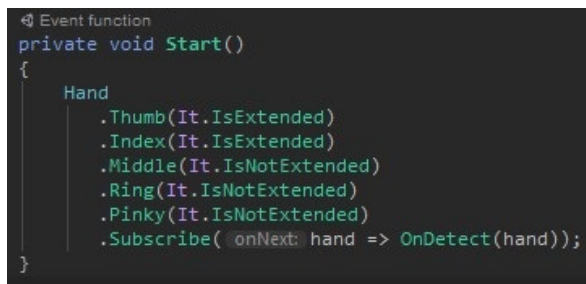


Figure 7. The *ReactiveHand* script start function

The code in figure 7 is one of the many ways for expressing this detector. Another option is presented in figure 8.

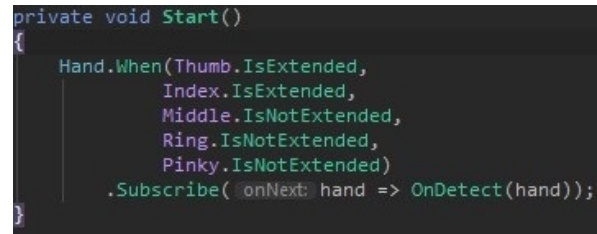


Figure 8. The *ReactiveHand* script start function, using the *When* operator

The *When* operator is a very powerful construct available in *FluentMotion*. It accepts a variable number of predicates, and a combination function for those predicates. If no combination function is given, the default AND is used.

Of course, a mix of the two methods described in figures 7 and 8 can be used - say, for when you also want the thumb to point upwards.

The next step is to implement the *OnDetect* method. A possible implementation is shown in figure 9.

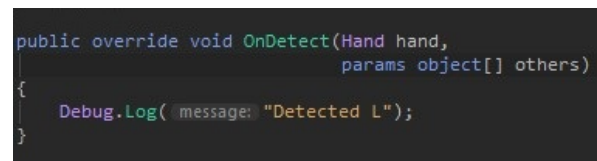


Figure 9. *OnDetect* function

The *params object[] others* are variable arguments - you can pass any extra parameters to this function.

Last, add the script to one of the *ReactiveHands* in the scene - for example, the right hand - as shown in figure 10.

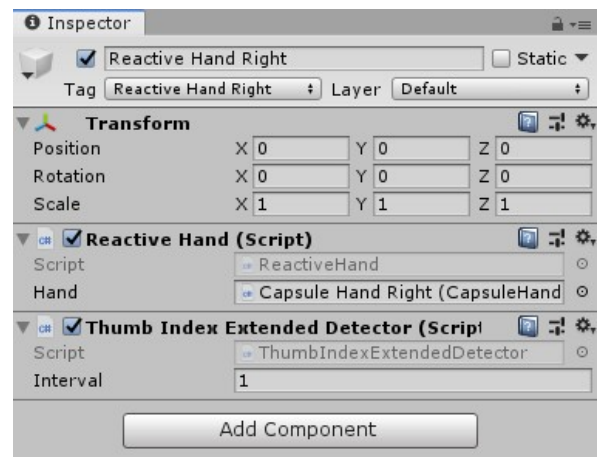


Figure 10. Script attached to the right *ReactiveHand*

The *Interval* field represents the sampling rate expressed in seconds. The sampling rate means how often the same gesture should be detected. The field is of type **double**, so values less

than 1 second can be set. The default value is 500ms (0.5 seconds).

## TESTING

For testing purposes, a simple application with 8 possible gestures was created. In the application there is a cube with an icon on it. The icon represents the gesture the cube expects from the player. Once that gesture is detected, the icon (and expected gesture) changes to another random gesture from the pool. The 8 possible gestures and their icon representation are shown in table 1.









Gesture	Icon
Thumbs-up	
"L"	
Fist	
Pinch	
Swipe-up	
Swipe-down	
Swipe-left	
Swipe-right	

Table 1. Gesture to icon mappings

All the gestures are expected to be done by either the left or the right hand. In figure 11, a pinch gesture is made with the right hand, but the cube expects a "L" gesture, so it does nothing.

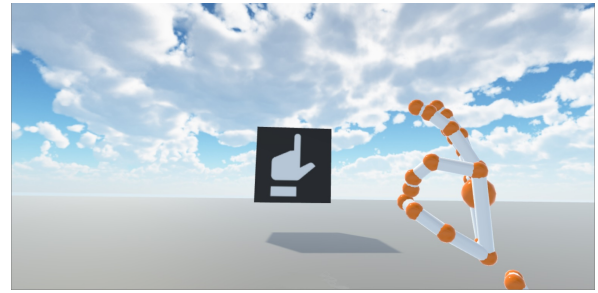


Figure 11. Cube not changing when an incorrect gesture is made

## PERFORMANCE

Reactive Extensions operators have varying performance, depending not only on the used operator, but also on the mapping or condition given to that operator. *FluentMotion* uses only the simpler operators from the RX environment, like *Select*, *Where*, *Subscribe* and *Sample*.

Chained operators do not add too big a performance penalty over the simpler ones. This is due to short circuiting in Reactive Extensions operators. That is, if the first operator in a chain fails (the gesture was not detected), all the operators after it aren't hit.

The short circuiting also means that chaining should be done in a careful way. The *IsMoving* operator has a much higher computational cost than the *IsPinching* operator. This means that moving the latter operator higher up the chain greatly impacts the overall performance of the application.

For better performance gains, more than one thread can be used. The user can take advantage of the Reactive Extensions' *ObserveOn(Scheduler.ThreadPool)* operator to schedule a detection chain on a different thread from the RX Pool.

One minor drawback of the *ObserveOn* is that the thread must be switched back to the main thread before doing any operations on the scene by calling the *ObserveOnMainThread* operator before the *Subscribe* call.

## API REQUIREMENTS

In order to use *FluentMotion*, the following software requirements exist:

- **Unity Game Engine** 2018.3.7f1 (or newer)
- **UniRX** 6.2.2
- **LeapMotion Orion** 4.0.0
- **LeapMotion Unity Core** 4.4.0
- **LeapMotion Interaction Engine** 1.2.0
- **SteamVR** 2.2.0

Hardware requirements are set by **SteamVR**, as it is the most demanding of the above mentioned software requirements. Those are:

- **CPU** Intel Core i5/Intel Core i5-4590/AMD FX 8350 equivalent or better

- **GPU** NVIDIA GeForce GTX 970, AMD Radeon R9 290 equivalent or better
- **RAM** 4GB
- **VRAM** 4GB
- **OS** Windows 7 SP1, Windows 8.1 or later, Windows 10

The hardware requirements apply only to the applications developed using *FluentMotion*, not to first hand API users (developers).

## CONCLUSIONS

Virtual Reality, even though a new, is a galloping technology whose tendencies are to become closer and closer to the actual reality. This tendency has fueled companies like LeapMotion to invent new and more natural means of interacting with the Virtual Reality world.

Their basic API, though powerful on its own, does not offer much flexibility and readability. This missing features created the need for a new, modern API.

In an attempt to answer this call, *FluentMotion* could be the needed replacement.

This paper presented the features of this new API, which features a flexible way of defining new gestures, Unity Game Engine integration and increased readability.

Future improvements include continuous gestures (like detecting letters drawn in the air) and the detection of "negated gestures" (*detect when \*this\* does not occur*).

## REFERENCES

1. 2019. (2019). <http://reactivex.io>.
2. Kent Boogaart. 2018. *You, I and ReactiveUI*.
3. Alex Colgan. 2014. How does the Leap Motion Controller Work? (2014). <http://blog.leapmotion.com/hardware-to-software-how-does-the-leap-motion-controller-work/>
4. Valve Corporation. 2019. (2019). <https://store.steampowered.com/steamvr>
5. Ralph Johnson Erich Gamma, Richard Helm and John Vlissides. 1994. *Design Patterns: Elements of Reusable Object-Oriented Software*. 293.
6. VRFocus. 2016. (18 04 2016). <https://www.vrfocus.com/2016/02/leap-motion-announces-orion-for-faster-more-accurate-vr-hand-tra>