FluentMotion - Gesture-based Interaction in Virtual Reality 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 gesture-based interaction in Virtual Reality using the LeapMotion controller. The *FluentMotion* API integrates a variety of modern technologies, such as C# LINQ (Language Integrated Query), Reactive Extensions and SteamVR. *FluentMotion* comes as an extension of the basic LeapMotion API, meant to facilitate the integration of gesture-based interaction in Virtual Reality projects. The API integrates with the Unity Game Engine, which provides the means of creating cross-platform Virtual Reality applications, allowing the definition of new, custom gestures using a human readable description (based on already existing ones) and creating interaction callbacks. *FluentMotion* works on HTC Vive and Oculus Rift for VR-enabled application and can also be used in desktop mode. This work improves the rudimentary API offered by LeapMotion and offers a more natural, 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: Human-Computer Interaction; Interaction Techniques ; Gestural Input

GENERAL TERMS
Virtual Reality; Gesture detection; API

INTRODUCTION

The concept of Virtual Reality (VR) 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 interaction through handheld controllers, which, to many, feels unnatural and unintuitive to use. To solve this issue, LeapMotion released Orion [13] 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, complex compositions and natural, human-readable description. Thus, the need for a more advanced API arised and *FluentMotion* was researched and developed with this need in mind.

FluentMotion is an extension of the LeapMotion API, based on Reactive Extensions [1], meant to facilitate the integration of gesture-based interaction in VR applications, that offers a more natural, powerful and flexible way of detecting and composing hand and finger gestures. Reactive Extensions is an implementation of the observer and iterator design patterns [5], 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. The API was partially inspired from *ReactiveUI* [3], a .NET framework for model-view-viewmodel (MVVM) applications based on Reactive Extensions. Some of the gesture detectors' syntax is based on the one of *ReactiveUI*'s *ReactiveObject*.

FluentMotion was built for the Unity Game Engine, a very popular game engine, used to empower a large number of VR projects. This provides increased usability, as it can be easily downloaded and integrated in any Untiy 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) to be detected for the hand the script is attached to. 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 is ready for usage.

A custom made application serves as a testbed for the *FluentMotion* API. The application features a set of icons representing one of the eight tested custom gestures the user has to perform. The icons dynamically change as the gestures are performed and correctly recognized.

The main limitations identified are mainly the ones that affect the LeapMotion controller, namely that hand gestures can only be recognized when performed in the user's field of view. However, one of the shortcomings of the API is the detection of complex moving gestures, which proves difficult. As of the current implementation, the only moving gestures supported are simple swipes.

RELATED WORKS

Ever since it was introduced to the public, LeapMotion promised to offer that which no other controller could do, namely provide natural, gesture-based interaction in VR environments. Even with studies that concluded that LeapMotion was not yet suited to compete with traditional input devices in desktop environments, for example as a contact-free pointing device [2], in VR it was quickly adopted as a much waited alternative to traditional, unnatural and obtrusive handheld controllers and thus LeapMotion enabled VR applications began to appear in the scientific world. Most attempts at using the LeapMotion controller in VR either implement the full gesture recognition logic based on a complex, low-level composition and detection API [8], or use the complex gesture composition mechanism developed for Unity [7], involving basic detectors and logic gates.

Kerefeyn and Maleshkov [9] used LeapMotion to control and manipulate objects in a VR scenario. Their solution splits the VR logic and the interaction logic and, while the overall interaction style proved to be a success, programming the interaction logic was cumbersome and strictly hard-coded for their application.

Vaitkevičius et al. [12] used LeapMotion to recognize the American Sign Language, building a system that is capable of learning gestures. The detection logic is implemented through the low-level API provided by the controller, extracting four types of hand features and manually composing either stationary or motion enabled gestures.

Khundam [10] use LeapMotion to control the movement of a first-person avatar in a VR scene. Movement is controlled by predefined gestures through LeapMotion's SDK for Unity, detected through complex composition of basic detectors using logic gates. The same gesture composition and detection approach is employed by Pop and Sabou in their work on dynamic data visualization and manipulation in VR [11].

While excellent results were obtained in all cases from integrating LeapMotion in various VR research, the process of complex gesture composition and detection is encumbering, both through the low-level API and the LeapMotion SDK for Unity, the resulting programming logic being restricted to specific use-cases, difficult to extend and, most importantly, difficult to understand by other programmers. As far as we are aware, no other high-level libraries exist that implement the LeapMotion gesture recognition logic in a natural, powerful and flexible way, although various hints at possible such ways for structuring such APIs have been identified [6].

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) [4].

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 [4].

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 Realityor for desktop mode. First, the service removes background objects and compensates for ambient lightning, 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

However, the core of the LeapMotion API remains the same and, even though it is straightforward, extensible and very well integrated with the Unity game engine, it still lacks flexibility, complex compositions and natural, human-readable description [8].

LEAPMOTION GESTURES

The LeapMotion API [8] defines mappings for four human body parts:[8]

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

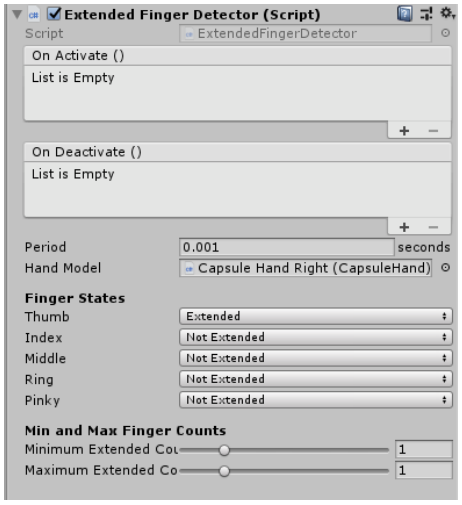
Figure 1. The leapmotion controller

- **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:

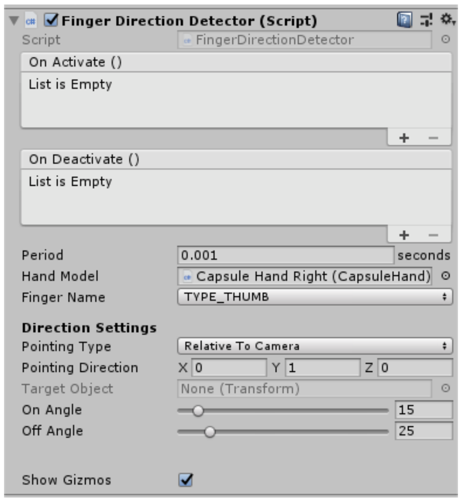
- Finger Extended Detector - configured to detect a thumb extended and other fingers not extended



Extended Finger Detector (Script)
Script ExtendedFingerDetector
On Activate ()
List is Empty
On Deactivate ()
List is Empty
Period 0.001 seconds
Hand Model Capsule Hand Right (CapsuleHand)
Finger States
Thumb Extended
Index Not Extended
Middle Not Extended
Ring Not Extended
Pinky Not Extended
Min and Max Finger Counts
Minimum Extended Cou 1
Maximum Extended Co 1

Figure 2. Leap Thumb extended detector

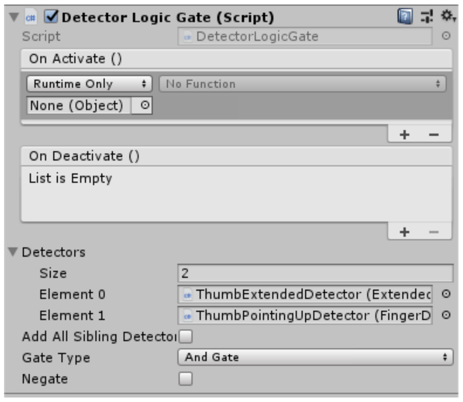
- Finger Pointing Detector - configured to detect that the thumb is pointing up (*Vector3(0, 0, 1)*) relative to the horizon



Finger Direction Detector (Script)
Script FingerDirectionDetector
On Activate ()
List is Empty
On Deactivate ()
List is Empty
Period 0.001 seconds
Hand Model Capsule Hand Right (CapsuleHand)
Finger Name TYPE_THUMB
Direction Settings
Pointing Type Relative To Camera
Pointing Direction X 0 Y 1 Z 0
Target Object None (Transform)
On Angle 15
Off Angle 25
Show Gizmos ✓

Figure 3. Leap Thumb pointing up detector

- And Logic Gate - to combine the other two detectors and have callbacks (C# scripts) attached to it



Detector Logic Gate (Script)
Script DetectorLogicGate
On Activate ()
Runtime Only No Function
None (Object)
On Deactivate ()
List is Empty
Detectors
Size 2
Element 0 ThumbExtendedDetector (Extendec
Element 1 ThumbPointingUpDetector (FingerD
Add All Sibling Detecto
Gate Type And Gate
Negate

Figure 4. Leap Thumb pointing up detector

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.

REACTIVE EXTENSIONS

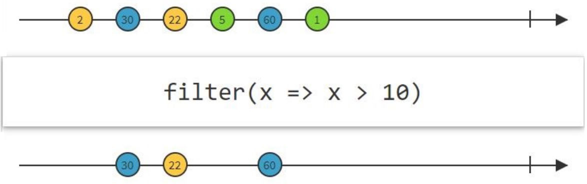


Figure 5. Example of a RX operator

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 concerts (such as mutlithreading 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 elemnts 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.

Algorithm 1 Iterable

```
GETDATAFROMLOCALMEMORY()
.SKIP(10)
.TAKE(5)
.MAP($s \rightarrow s + "transformed"$)
.FOREACH($println("next \rightarrow " + it)$)
```

Algorithm 2 Iterable

```
GETDATAFROMLOCALMEMORY()
.SKIP(10)
.TAKE(5)
.MAP($s \rightarrow s + "transformed"$)
.SUBSCRIBE($println("onNext \rightarrow " + it)$)
```

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.

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.

Unity integration

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 3rd (as in figure 6)

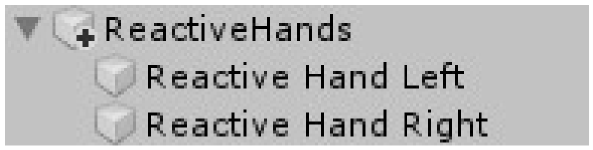


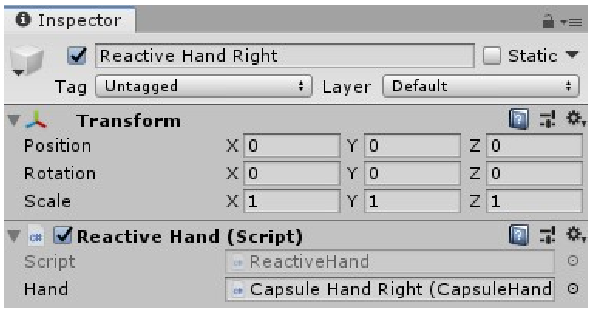
Figure 6. Setting up *FluentMotion* game objects

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 3rd, referencing the other two *ReactiveHand*s

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 shows in figure 8.

The code in figure 8 is one of the many ways for expressing this detector. Another option is presented in figure 9.

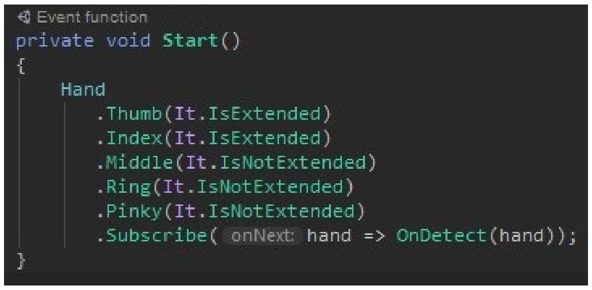


Inspector
Reactive Hand Right — Static
Tag: Untagged — Layer: Default

Transform
	X	Y	Z
Position	0	0	0
Rotation	0	0	0
Scale	1	1	1

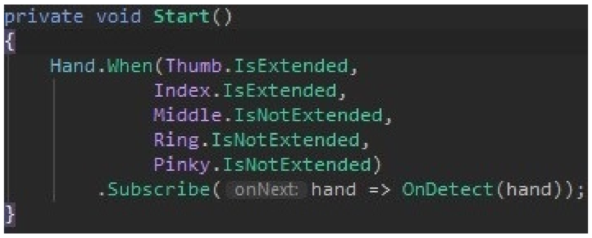
Reactive Hand (Script)
Script: ReactiveHand
Hand: Capsule Hand Right (CapsuleHand

Figure 7. The *ReactiveHand* script attached to the game object



```
Event function
private void Start()
{
    Hand
        .Thumb(It.IsExtended)
        .Index(It.IsExtended)
        .Middle(It.IsNotExtended)
        .Ring(It.IsNotExtended)
        .Pinky(It.IsNotExtended)
        .Subscribe( onNext: hand => OnDetect(hand));
}
```

Figure 8. The *ReactiveHand* script start function



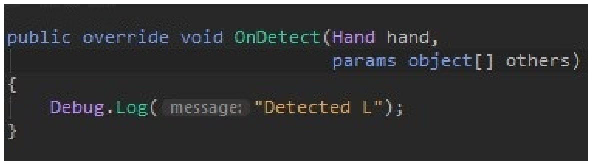
```
private void Start()
{
    Hand.When(Thumb.IsExtended,
              Index.IsExtended,
              Middle.IsNotExtended,
              Ring.IsNotExtended,
              Pinky.IsNotExtended)
        .Subscribe( onNext: hand => OnDetect(hand));
}
```

Figure 9. The *ReactiveHand* script start function, using the *When* operator

The *When* operator is a very powerful construct available in *FluentMotion*. The first argument to this operator is an optional reduction (combination) lambda function which takes two boolean values as input and returns one boolean value. The default value for this reduction function is the AND operator (the returned boolean value is the logical "and" between the two inputs). Then comes a variable number of predicates, lambda expressions that take as input one value (of the same type as the underlying data stream) and return a boolean value. The combination lambda is applied to each of the predicates' outputs, left to right. If one of the values changes the result of the reduction from *true* to *false*, the reduction process stops (due to the *shortcircuit* property of boolean operators) and the whole data stream cancels.

Of course, a mix of the two methods described in figures 8 and 9 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 10.

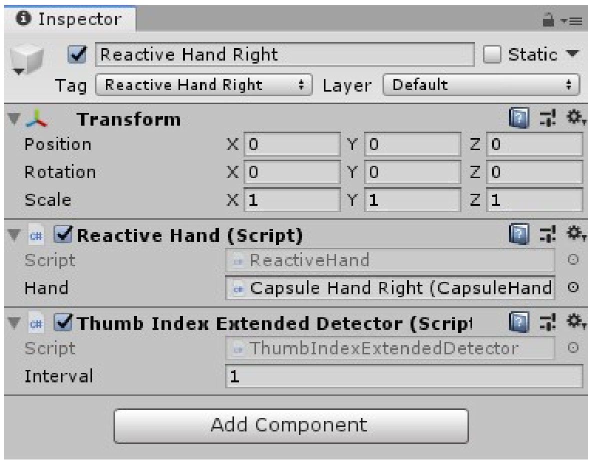


```
public override void OnDetect(Hand hand,
                              params object[] others)
{
    Debug.Log( message: "Detected L");
}
```

Figure 10. *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 *ReactiveHand*s in the scene - for example, the right hand - as shown in figure 11.



Inspector
Reactive Hand Right — Static
Tag: Reactive Hand Right — Layer: Default

Transform
	X	Y	Z
Position	0	0	0
Rotation	0	0	0
Scale	1	1	1

Reactive Hand (Script)
Script: ReactiveHand
Hand: Capsule Hand Right (CapsuleHand

Thumb Index Extended Detector (Script
Script: ThumbIndexExtendedDetector
Interval: 1

Add Component

Figure 11. 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 the 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.

All the gestures are expected to be done by either the left or the right hand. In figure 12, a pinch gesture is made with the right hand, but the cube expects a "L" gesture, so it does nothing.

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*.

Gesture	Icon
Thumbs-up	👍
"L"	☝
Fist	✊
Pinch	👌
Swipe-up	⏫
Swipe-down	⏬
Swipe-left	⏪
Swipe-right	⏩

Table 1. Gesture to icon mappings

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.ThereadPool)* operator to schedule a detection chain on a different thread from the RX Pool.

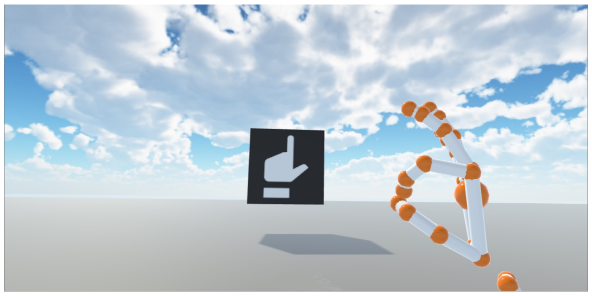


Figure 12. Cube not changing when an incorrect gesture is made

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 i5Intel 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. ReactiveX. (2019). http://reactivex.io
2. Daniel Bachmann, Frank Weichert, and Gerhard Rinkenauer. 2015. Evaluation of the Leap Motion Controller as a New Contact-Free Pointing Device. *Sensors* 15 (2015), 214–233. DOI: http://dx.doi.org/10.3390/s150100214 Publikation.
3. K. Boogaart. 2018. *You, I, and ReactiveUI*. Blurb, Incorporated. https://kent-boogaart.com/you-i-and-reactiveui/
4. 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/
5. Ralph Johnson Erich Gamma, Richard Helm and John Vlissides. 1994. *Design Patterns: Elements of Reusable Object-Oriented Software*. 293.
6. Jewgeni Horn. 2014. Gesture recognition with Leap Motion controller and Reactive Extensions. (9 04 2014). https://www.zuehlke.com/blog/en/gesture-recognition-with-leap-motion-controller-and-reactive-extensions/
7. Leap Motion Inc. 2019a. LeapMotion Developer - Unity. (2019). https://developer.leapmotion.com/unity/
8. Leap Motion Inc. 2019b. LeapMotion Developer - Using the Tracking API. (2019). https://developer-archive.leapmotion.com/documentation/python/devguide/Leap_Guides2.html
9. Stoyan Kerefeyn and Stoyan Maleshkov. 2015. Manipulation of virtual objects through a LeapMotion optical sensor. *International Journal of Computer Science Issues* 12 (10 2015), 52–57.
10. C. Khundam. 2015. First person movement control with palm normal and hand gesture interaction in virtual reality. In *2015 12th International Joint Conference on Computer Science and Software Engineering (JCSSE)*. 325–330. DOI: http://dx.doi.org/10.1109/JCSSE.2015.7219818
11. Mihai Pop and Adrian Sabou. 2017. Gesture-based Visual Analytics in Virtual Reality. *Revista Romana de Interactiune Om-Calculator* 10 (2017), 216–230. Issue 3.
12. Aurelijus Vaitkevičius, Mantas Taroza, Tomas Blažauskas, Robertas Damaševičius, Rytis Maskeliūnas, and Marcin Woźniak. 2019. Recognition of American Sign Language Gestures in a Virtual Reality Using Leap Motion. *Applied Sciences* 9, 3 (2019). DOI: http://dx.doi.org/10.3390/app9030445
13. VRFocus. 2016. (18 04 2016). https://www.vrfocus.com/2016/02/leap-motion-announces-orion-for-faster-more-accurate-vr-hand-trac