

MINISTRY OF NATIONAL EDUCATION



TECHNICAL UNIVERSITY
OF CLUJ-NAPOCA

**FACULTY OF AUTOMATION AND COMPUTER SCIENCE
COMPUTER SCIENCE DEPARTMENT**

**REACTIVE PROGRAMMING BASED GESTURE DETECTION IN
VIRTUAL REALITY USING LEAPMOTION**

LICENSE THESIS

**Graduate: Radu PETRIȘEL
Supervisor: Assist. Prof. Dr. Eng. Adrian SABOU**

2019

MINISTRY OF NATIONAL EDUCATION



TECHNICAL UNIVERSITY

OF CLUJ-NAPOCA

**FACULTY OF AUTOMATION AND COMPUTER SCIENCE
COMPUTER SCIENCE DEPARTMENT**

DEAN,
Prof. dr. eng. Liviu MICLEA

HEAD OF DEPARTMENT,
Prof. dr. eng. Rodica POTOLEA

Graduate: **Radu PETRIȘEL**

**REACTIVE PROGRAMMING BASED GESTURE DETECTION IN
VIRTUAL REALITY USING LEAPMOTION**

1. **Project proposal:** *A Reactive Programming oriented Unity asset for gesture detection using the LeapMotion controller*
2. **Project contents:** *(enumerate the main component parts) Presentation page, advisor's evaluation, title of chapter 1, title of chapter 2, ..., title of chapter n, bibliography, appendices.*
3. **Place of documentation:** *Technical University of Cluj-Napoca, Computer Science Department*
4. **Consultants:** Assist. Prof. Dr. Eng. Adrian SABOU
5. **Date of issue of the proposal:** November 1, 2018
6. **Date of delivery:** June 14, 2019

Graduate: _____

Supervisor: _____



FACULTY OF AUTOMATION AND COMPUTER SCIENCE
COMPUTER SCIENCE DEPARTMENT

**Declarație pe proprie răspundere privind
autenticitatea lucrării de licență**

Subsemnatul(a) **RADU PETRIȘEL** legitimat(ă) cu C.I. seria **CJ** nr. **315937** CNP **1960920125844**, autorul lucrării **REACTIVE PROGRAMMING BASED GESTURE DETECTION IN VIRTUAL REALITY USING LEAPMOTION** elaborată în vederea susținerii examenului de finalizare a studiilor de licență la Facultatea de Automatică și Calculatoare, Specializarea **CALCULATOARE, ENGLEZA** din cadrul Universității Tehnice din Cluj-Napoca, sesiunea **IULIE** a anului universitar **2018-2019**, declar pe proprie răspundere, că această lucrare este rezultatul propriei activități intelectuale, pe baza cercetărilor mele și pe baza informațiilor obținute din surse care au fost citate, în textul lucrării și în bibliografie.

Declar, că această lucrare nu conține porțiuni plagiate, iar sursele bibliografice au fost folosite cu respectarea legislației române și a convențiilor internaționale privind drepturile de autor.

Declar, de asemenea, că această lucrare nu a mai fost prezentată în fața unei alte comisii de examen de licență.

În cazul constatării ulterioare a unor declarații false, voi suporta sancțiunile administrative, respectiv, *anularea examenului de licență*.

Data

Nume, Prenume

RADU PETRIȘEL

Semnătura

Contents

Chapter 1	Introduction - Project Context	1
1.1	Virtual reality	1
1.1.1	History	1
1.1.2	Modern technology	2
1.2	Gesture recognition	2
Chapter 2	Project Objectives and Specifications	3
2.1	Introduction	3
2.2	Positioning	3
2.2.1	Problem statement	3
2.2.2	Product Position Statement	4
2.3	Stakeholder and User Descriptions	4
2.3.1	Stakeholder summary	4
2.3.2	User summary	4
2.3.3	User environment	5
2.3.4	Summary of key stakeholder or user needs	5
2.3.5	Alternatives and competetion	5
2.4	Product overview	5
2.4.1	Product perspective	6
2.4.2	Assumption and dependencies	6
2.5	Product features	6
2.6	Other product requirements	7
Chapter 3	Bibliographic research	9
3.1	Virtual reality	9
3.2	Gestures	11
3.2.1	Manual gestures	12
3.3	Gesture recognition	12
3.3.1	Algorithms	13
3.3.2	Input devices	14
3.4	Leap Motion	15
3.4.1	Hardware	16

3.4.2	Software	16
3.4.3	Gestures and detectors	17
3.5	Reactive programming	19
3.5.1	Reactive Extensions	19
Chapter 4	Analysis and Theoretical Foundation	22
4.1	Conceptual architecture	22
4.1.1	Leap hand model	22
4.1.2	Reactive hand model	23
4.1.3	MonoBehaviour	23
4.1.4	Reactive Gesture detector	24
4.2	Create Gesture detector	25
4.2.1	Brief description	25
4.2.2	Primary actor	25
4.2.3	Stakeholders and interests	25
4.2.4	Flow of events	25
4.2.5	Preconditions	26
4.2.6	Postconditions	26
4.3	Add Simple Gesture Detector	26
4.3.1	Brief description	26
4.3.2	Primary actor	26
4.3.3	Stakeholders and interests	27
4.3.4	Flow of events	27
4.3.5	Preconditions	28
4.3.6	Postconditions	28
Chapter 5	Detailed Design and Implementation	29
5.1	Fluent Motion simple gestures	29
5.1.1	Finger Gestures	29
5.1.2	Hand Gestures - single hand	29
5.1.3	Hand Gestures - both hands	30
5.2	When<TInput, TResult> method	30
5.2.1	Syntax	30
5.2.2	Type parameters	30
5.2.3	Parameters	31
5.2.4	Return Value	31
5.2.5	Remarks	31
5.2.6	Overloads	31
5.3	Fingers method	32
5.3.1	Syntax	32
5.3.2	Parameters	33
5.3.3	Return value	33

5.3.4	Remarks	33
5.3.5	Overloads	33
5.4	GetFinger method	34
5.4.1	Syntax	34
5.4.2	Parameters	34
5.4.3	Return value	34
5.4.4	Remarks	34
5.5	AreExtended method	35
5.5.1	Syntax	35
5.5.2	Parameters	35
5.5.3	Return value	35
5.5.4	Remarks	35
5.6	Thumb method (selector)	36
5.6.1	Syntax	36
5.6.2	Parameters	36
5.6.3	Return values	36
5.6.4	Remarks	36
5.7	Thumb method (filter)	37
5.7.1	Syntax	37
5.7.2	Parameters	37
5.7.3	Return values	37
5.7.4	Remarks	37
5.8	Index method (selector)	38
5.8.1	Syntax	38
5.8.2	Parameters	38
5.8.3	Return values	38
5.8.4	Remarks	38
5.9	Index method (filter)	39
5.9.1	Syntax	39
5.9.2	Parameters	39
5.9.3	Return values	39
5.9.4	Remarks	39
5.10	Middle method (selector)	40
5.10.1	Syntax	40
5.10.2	Parameters	40
5.10.3	Return values	40
5.10.4	Remarks	40
5.11	Middle method (filter)	41
5.11.1	Syntax	41
5.11.2	Parameters	41
5.11.3	Return values	41
5.11.4	Remarks	41

5.12	Ring method (selector)	42
5.12.1	Syntax	42
5.12.2	Parameters	42
5.12.3	Return values	42
5.12.4	Remarks	42
5.13	Ring method (filter)	43
5.13.1	Syntax	43
5.13.2	Parameters	43
5.13.3	Return values	43
5.13.4	Remarks	43
5.14	Pinky method (selector)	44
5.14.1	Syntax	44
5.14.2	Parameters	44
5.14.3	Return values	44
5.14.4	Remarks	44
5.15	Pinky method (filter)	45
5.15.1	Syntax	45
5.15.2	Parameters	45
5.15.3	Return values	45
5.15.4	Remarks	45
5.16	IsPinching method	46
5.16.1	Syntax	46
5.16.2	Parameters	46
5.16.3	Return values	46
5.16.4	Remarks	46
5.17	PalmIsFacing<TPlayer, TTarget> method	47
5.17.1	Syntax	47
5.17.2	Type parameters	47
5.17.3	Parameters	47
5.18	Return value	48
5.18.1	Remarks	48
5.18.2	Overloads	48
5.19	IsFist method	50
5.19.1	Syntax	50
5.19.2	Parameters	50
5.19.3	Return value	50
5.19.4	Remarks	50
5.20	IsMoving method	51
5.20.1	Syntax	51
5.20.2	Parameters	51
5.20.3	Return value	51
5.20.4	Remarks	51

5.21	IsMoving<TReference> method	52
5.21.1	Syntax	52
5.21.2	Type parameters	52
5.21.3	Parameters	52
5.21.4	Return value	53
5.21.5	Remarks	53
5.21.6	Overloads	53
Chapter 6	Testing and Validation	54
6.1	Performance	55
Chapter 7	User's manual	57
7.1	Unity integration	57
7.1.1	Creating Detectors	57
Chapter 8	Conclusions	61
	Bibliography	62
	Appendix A Relevant code	63
	Appendix B Other relevant information (demonstrations, etc.)	64
	Appendix C Published papers	65

Chapter 1

Introduction - Project Context

Virtual Reality is an experience that has gained huge popularity in the recent years. Because of this new means of interaction with this virtual world are needed and they should feel as natural as possible. Ergo, hand tracking and gesture detection is a "must have" for modern VR applications.

1.1 Virtual reality

The term "virtual" began its life in the late 1400s, meaning "being something in essence or effect, though not actually or in fact" [1], but, in the IT context, the word has the meaning "not physically existing but made to appear by software" [1]. The original use of the phrase "virtual reality" is found in French playwright' Antonin Artaud collection of essays *Le Théâtre et son double*, first published in 1938 [2].

1.1.1 History

The precise roots of virtual reality are challenged, partially because of how hard it was to formulate a definition of an alternate reality notion. In 1968, Ivan Sutherland created what was widely regarded as the first head-mounted display system for use in immersive simulation applications, with the help of his students. In the next two decades, VR devices were mainly used for medical, automobile industry design, military training and flight simulation purposes.

The 1990s saw the first commercially extensive release of consumer headsets, notably *Sega VR* (1991) and *Sega VR-1* (1994) launched by Sega, and *Nintendo's Virtual Boy* (1995). The 2000s were a period of comparative indifference from the public and investment towards VR techniques available on the market. Google launched *Street View* in 2007, a service that offers panoramic views of a growing amount of global locations such as highways, indoor houses and rural regions, which also integrates a stereoscopic 3D mode as of 2010.

The modern, consumer version of headsets started developing in the early 2010s. In 2013, Valve Corporation found and freely shared the breakthrough of low-persistence screens that make it possible today to show VR content lag-free and smear-free.

This discovery was quickly adopted by the other companies on the market, with Sony announcing *Project Morpheus* in 2014 and Google announcing *Cardboard* in 2015. In 2016, HTC released and shipped the first units of *Vive SteamVR*, the first major commercial headset for average users.

1.1.2 Modern technology

Present virtual reality headset displays rely on smartphone technologies including: gyroscopes and motion sensors for head, hand and body position monitoring, tiny high definition stereoscopic displays and small, lightweight and powerful computer processors.

Special input devices are required for interaction with the virtual world, such as hand controllers, haptic gloves, 3D mouse and optical tracking sensors. Both haptic gloves and hand controllers provide force feedback (in the form of vibration), with haptic gloves providing also feedback in the form of response force (like when picking a rubber duck).



Figure 1.1: Project Morpheus (PlayStation VR) at gamescom in 2015

1.2 Gesture recognition



Figure 1.2: A child being recognized by a simple gesture detection algorithm

Gesture recognition is an active research field with the objective of comprehending human gestures through mathematical models. Gestures can come from any posture or position of the body, but they typically come from the hand. Without actually touching them, users can use simple motions to command or communicate with machines.

Gesture recognition may be seen as a means for machines to commence to comprehend human body language, establishing a stronger link between computers and individuals than conventional text user interfaces or even GUIs (graphical user interfaces), which still restrict most inputs to the keyboard and/or mouse and communicate naturally with no mechanical instruments.

Chapter 2

Project Objectives and Specifications

2.1 Introduction

The purpose of this chapter is to collect, analyze and define high-level needs and features of the Unity asset named Fluent Motion. It focuses on the capabilities needed by the stakeholders and the target users, and why these needs exist.

2.2 Positioning

2.2.1 Problem statement

As Virtual Reality (VR) is becoming more accessible to the average person, more problems arise with the means of interacting with the VR world. A solution to this issue is the Leap Motion hand tracking device, which offers a natural means of human-VR interaction. The problem with Leap Motion is its non-friendly Application Programmer Interface (API).

The problem of	Leap Motion's unfriendly API
affects	developers in the VR field who use Leap Motion
the impact of which is	a limited number of applications using Leap Motion
a successful solution would be	easy to use fluent (in terms of code readability) adhere to the reactive programming paradigm available on Unity's asset store

2.2.2 Product Position Statement

Fluent Motion comes as a union between three technologies – Virtual Reality, Leap Motion and ReactiveX.

So far, Virtual Reality and Leap Motion already are integrated (by means of Leap Motion’s API), but ReactiveX can offer a more fluent way of expressing what an application using the first two mentioned technologies together.

For	Virtual Reality developers
who	use Leap Motion
Fluent Motion	is an extension of Leap Motion using ReactiveX
that	offers a fluent API for Leap Motion
unlike	the default API
Fluent Motion will	be easy to use be fluent (in terms of code readability) adhere to the reactive programming paradigm

2.3 Stakeholder and User Descriptions

2.3.1 Stakeholder summary

Name	Description	Responsibilities
Developer (VR)	Person who wants to create Virtual Reality applications	Use Fluent Motion
Developer (Fluent Motion)	Person who creates and maintains Fluent Motion	Create, improve and offer technical support for Fluent Motion

2.3.2 User summary

Name	Description	Responsibilities	Stakeholder
Developer (VR)	Person who wants to create VR applications	Use Fluent Motion	<i>Developer (VR)</i>

2.3.3 User environment

2.3.3.1 Users

The API will be used by developer teams of any size.

2.3.3.2 Infrastructure

The infrastructure needed by Fluent Leap is an aggregation of the hardware requirements of the combined systems and technologies, i.e.:

Operating system	Windows 7 SP1, Windows 8.1 or later, Windows 10
Middleware	SteamVR platform
Additional hardware	Leap Motion hand tracking device a VR headset (at the time of writing, Oculus Rift, HTC Vive or Valve Index)
Miscellaneous	.NET Framework 4.6 or newer Unity 5.6 or later

2.3.4 Summary of key stakeholder or user needs

Need	Priority	Concerns	Current solution	Proposed solution
VR API	0	Developer	Leap Motion default API, using ANSI C language imperative style	Fluent Motion, using C# language and ReactiveX
Desktop API	1	Developer	Leap Motion default API, using ANSI C language impertive style	Fluent Motion, using C# language and ReactiveX
Usability	0	Developer	Leap Motion default API, using ANSI C language imperative style	Fluent Motion, using C# language and ReactiveX

2.3.5 Alternatives and competetion

External competition is represented by the current API offered by LeapMotion.

2.4 Product overview

The API should provide all the functionality already provided by Leap Motion’s default API, but in a higher-level language.

2.4.1 Product perspective

This product will extend existing features from Leap Motion, making them more readable and developer friendly.

2.4.2 Assumption and dependencies

For developers:

Operating system	Windows 7 SP1, Windows 8.1 or later, Windows 10
Middleware	SteamVR platform
Additional hardware	Leap Motion hand tracking device a VR headset (at the time of writing, Oculus Rift, HTC Vive or Valve Index)
Miscellaneous	.NET Framework 4.6 or newer Unity 5.6 or later

For end products that reference Fluent Motion:

	Minimum	Recommended
CPU	Intel Core i3-8100	Intel i5-4590 or AMD FX 8350 equivalent
GPU	Nvidia GeForce GTX 1060 3GB or AMD Radeon RX 570	Nvidia GeForce GTX 970 or AMD Radeon R9 290 equivalent
Memory	8GB	16GB
Output	HDMI 1.4, DisplayPort 1.2	DisplayPort 1.2
Input	2x USB 3.1 gen 1 (Type-A)	2x USB 3.1 gen 2 (Type-A)

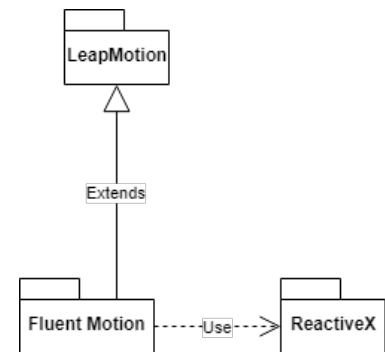


Figure 2.1: Fluent Motion architectural diagram

2.5 Product features

1. Hands module

The module that will allow the developer to use the two hands objects from the scene in order to detect gestures or motions, and assign callbacks when certain criteria regarding the hands are met.

This module is available in both Virtual Reality and Desktop modes.

2. Fingers module

The module that will allow the developer to use the fingers (individually or in groups) in the scene for detecting gestures or motions, and assign callbacks when certain criteria regarding the fingers are met.

This module is available in both Virtual Reality and Desktop modes.

3. Virtual Reality module

The module that will allow the user to develop Virtual Reality applications. This module will act as a dependency to other modules.

It isn't mutually exclusive with the Desktop Module, BUT at least one of the two must be present.

4. Desktop module

The module that will allow the user to develop applications using Leap Motion in desktop mode.

This module will act as dependency to other modules. It isn't mutually exclusive with the Virtual Reality Module, BUT at least one of the two must be present.

5. Gesture definition module

The module that allows the user to define new gestures besides already existing ones. This module depends on either Virtual Reality module or Desktop module, and on either Hands module, Fingers module or both.

6. Gestures module

This module will provide definitions to some basic gestures (like finger pointing to some object, hand swipe, etc.). This module depends on either Virtual Reality module or Desktop module, and on either Hands module and Fingers module.

7. Demo

This module will provide some demonstrative code for new users to acquaint themselves with the basic flows and code syntax of Fluent Motion.

2.6 Other product requirements

1. High readability

The main purpose of the API is to be fluently read, i.e. the code should sound almost like natural language when read by other developers.

2. Open source

The project will be open source and anyone will be able to contribute to it.

3. Performance

Virtual Reality applications shouldn't fall below 80 frames per second (FPS), and, as such, the API shouldn't introduce a high processing time per frame to fall below that threshold.

4. Scalability

The API should support detecting up to 10 distinct gestures per application and interacting with at least 20 objects (excluding hand to hand interactions).

5. Maintainability

The VR world is still young and technologies evolve fast, so the API should be highly maintainable to keep its edge.

6. Extendibility

The API should be easily extendable by any backer on Git.

Chapter 3

Bibliographic research

3.1 Virtual reality

Virtual Reality (VR) is the usage of computer technology to produce a world that is simulated. Unlike conventional user interfaces, VR positions the user in an environment in which they are immersed and able to interact with 3D worlds rather than watching a screen in front of them. The computer is converted into an arbiter for this alternative reality by simulating as many senses as applicable, such as sight, hearing, touch or even scent.

The head-mounted display (HMD) is by far the most instantly recognizable element of Virtual Reality. Human creatures have always been visual animals, and display technology is often the greatest gap between immersive systems of Virtual Reality and conventional GUIs.

The future of wearable devices is unraveling but still uncertain with a multitude of evolving wired (tethered) and wireless (untethered) options. Concepts like the HTC Vive Pro Eye, Oculus Quest and Playstation VR are leading the way, but competitors such as Google, Apple, Samsung and others might also shock the sector with fresh amounts of immersion and usability.

Virtual Reality has a variety of uses in a multitude of domains, such as:

1. **Education and training**

VR is used to provide a virtual world for trainees to improve their abilities without the risk of failure in the real world. Applications such as flight simulators, surgery training and spacewalk training have been used for decades in strongly technologised countries.

2. **Engineering and robotics**

The use of 3D computer-aided design (CAD) data was limited by 2D monitors and paper printouts until the mid-to-late 1990s, when video projectors, 3D tracking, and computer technology enabled a renaissance in the use of 3D CAD data in virtual reality environments. Innovative VR engineering systems allow engineers to visualize virtual prototypes before any physical models are available.

3. Entertainment

During the early to mid 1990s, several vanilla commercial virtual reality headsets have been released for gaming, such as the *Virtual Boy* developed by Nintendo or *VFX1 Headgear* developed by Forte Technologies. Films produced for VR permit the audience to view a 360-degree environment. This can involve the use of VR cameras to produce films and series that are interactive in VR. VR may help people attend concerts without effectively being there.

4. Healthcare and clinical therapies

A 2017 report by Goldman Sachs investigated healthcare applications of VR and AR. [3]. Some companies are adapting VR for fitness by using gamification concepts to encourage exercise. Since the 2000s, virtual reality has been used for rehabilitation. Despite countless research, there is a lack of excellent quality proof of its effectiveness compared to other techniques of rehabilitation without advanced and costly facilities for Parkinson's disease therapy. [4]

5. Heritage and archaeology

Virtual reality allows highly accurate recreation of historic locations so that artistic renderings can be published in different media. The initial sites are very often unavailable to the public or difficult to portray due mainly to the bad condition of their conservation. Using this technology, virtual replicas of grottoes, environment, ancient cities and monuments, sculptures and archaeological components can be developed.

6. Occupational safety

VR simulates real workplaces for occupational safety and health purposes. Perspective, viewing angle, and acoustic and tactile characteristics alter depending on where the individual is standing and how they move relative to the setting. VR enables all phases of a product life cycle, from design, through use, up to disposal, to be simulated, analyzed and optimized.

All of the above mentioned domains are part of the day-to-day life of some people already. For them, interacting with the virtual reality they are put in should feel free and natural. In a real life situation, they would use their hands to perform the actions they are performing in virtual reality during those "training" periods. So the question arises: how do they perform their everyday tasks in virtual reality?

If somebody were to ask that question to the two big players in the VR game (namely, HTC and Oculus), the answer will be the same from them all - handheld controllers (depicted in figure 3.1). They are very versatile and offer various means of interaction, but they lack one simple feature - feeling natural. Take, for instance, a surgeon training in VR to perform a new kind of surgery that he hadn't performed before. By using the controllers, he has a handicap that would not be present in a real life situation. As such, his training does not quite copy the real thing.

So, how would one improve the issue of VR interaction? As of now, there are two possibilities - *haptic gloves* or *hand-tracking devices*. Haptic gloves are special devices that users put on their hands, which track their hands' motions and send them to the computer to process. Haptic gloves offer the advantage of sending back haptic feedback (pressure and force). Hand tracking devices use cameras and complex algorithms to track the users' hands. The advantage of hand-tracking devices is that the hands can move unrestricted and without needing additional hardware. Both hand tracking devices and haptic gloves have a common denominator - **gestures**.

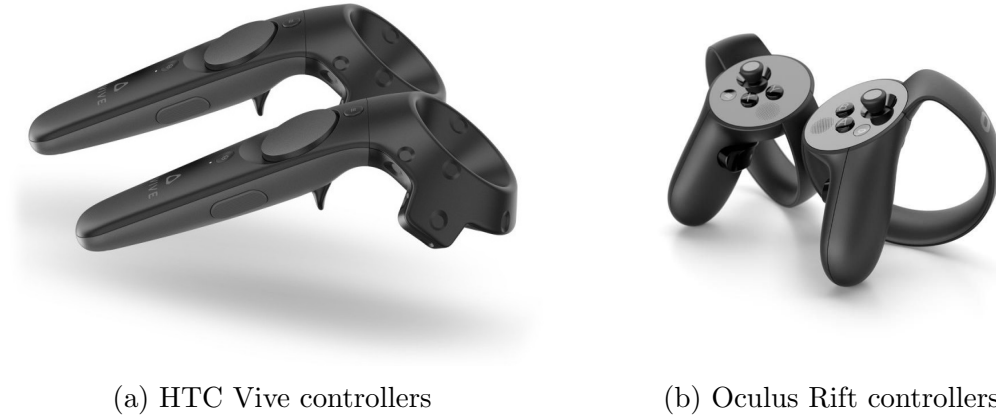


Figure 3.1: The two types of VR controllers available on the market

3.2 Gestures

A gesture is a type of non-verbal communication in which real physical movements transmit specific messages, both in place or in combination with speech. Gestures include movement of the hands, face, or other parts of the body. Gestures differ from physical non-verbal communication that does not communicate specific messages, such as purely expressive displays, proxemics, or displays of joint attention. [5]

Gestures can be of two types: *informative (passive)* or *communicative (active)*. *Informative gestures* are passive gestures that provide details about the speaker as a person and not about what the speaker is attempting to communicate, while *communicative gestures* are deliberately and meaningfully made by a individual as a means of height-



Figure 3.2: Military air marshallers use hand and body gestures to direct flight operations aboard aircraft carriers

ening or altering speech generated in the larynx (or with hands in the situation of sign languages) although he or she may not be actively conscious of the fact that they produce communicative gestures.

Within the realm of communicative gestures, the first distinction to be made is between gestures made with the hands and arms, and gestures made with other parts of the body, such as head or shoulders. From now on, we shall focus only on manual gestures.

3.2.1 Manual gestures

Manual gestures are split into four categories:

1. **Symbolic (emblematic)**

These are standard, culture-specific gestures which can be used as a substitute for words (like waving your hand to say "hello" or "goodbye"). In distinct cultural contexts, a single emblematic gesture can have a very distinct meaning, varying from complimentary to extremely offensive. Symbolic gestures are iconic gestures that are widely recognized, fixed, and have conventionalized meanings. [6]

2. **Deictic (indexical)**

Deictic gestures may happen concurrently or in place of vocal expression. Deictic gestures are gestures consisting of indicative motions or pointing movements. They often replace words and pronouns like "this", "there" or "that".

3. **Motor (beat)**

In verbal speech, motor or beat gestures typically consist of brief, rhythmic, repetitive motions strongly linked to sentence construction. Beat gestures do not happen separately of verbal expression, unlike symbolic and deictic gestures. Some individuals wave their hands, for instance, as they talk to highlight a word or sentence. These gestures are closely coordinated with speech.

4. **Lexical (iconic)**

Other spontaneous gestures used in speech known as iconic gestures are more content-filled, and the significance of the co-occurring voice may echo, or be elaborated. They portray elements of pictures, behavior, individuals, or items in space. For instance, a gesture depicting the throwing act may be synchronous with the saying, "He threw the ball right into the window."

3.3 Gesture recognition

Gesture recognition is an active research field with the objective of comprehending human gestures through mathematical models. Gestures can come from any posture or position of the body, but they typically come from the hands. Without actually touching them, users can use simple motions to command or communicate with machines.

Gesture recognition may be seen as a means for machines to commence to comprehend human body language, establishing a stronger link between computers and individuals than conventional text user interfaces or even GUIs (graphical user interfaces), which still restrict most inputs to the keyboard and/or mouse and communicate naturally with no mechanical instruments.

There are two gesture types in the human-computer interaction context:

1. **Offline gestures**

These gestures are processed after the object is interacted with by the user (e.g. activate a menu gesture).

2. **Online gesture**

Direct manipulation gestures, like scaling or moving an object.

3.3.1 Algorithms

The strategy to translating a gesture could be performed in distinct ways based on the category of input data. Most of the methods, however, are based on important pointers in a 3D reference system. The gesture can be identified with considerable precision depending on the relative movement of all these, depending on the quality of both the input and the strategy of the algorithm.

Gesture detection algorithms can be split in three major categories, based on the hand models they are using:

1. **3D model-based algorithms**

The 3D model approach can use volumetric or skeletal models, or even a combination of the two. Volumetric methods were used extensively in the field of computer animation and computer vision. The advantage of this strategy is the simplicity of the parameters for all these objects. This technique's disadvantage is that it is very computational-intensive, and technologies still need to be developed for real-time interpretation.

2. **Skeletal-based algorithms**

Instead of using intensive 3D model processing and working with several parameters, one could just make use of a simplified version of the parameters, like the joint angle along with the dimensions of the section. The advantage of this method is that the algorithms are faster (thus requiring less resources) and the templated patterns can be persisted in a database and used for later matching. The drawback is that, because of the reduced parameters, the algorithms do not perform as accurately as 3D model-based algorithms.

3. **Appearance-based models**

These algorithms no longer use a body's 3D representation since they obtain the

parameters from the videos and images directly using a template database. One of the approaches of gesture detection using appearance-based models uses image sequences as gesture templates (either using the images directly or features extracted from the images).

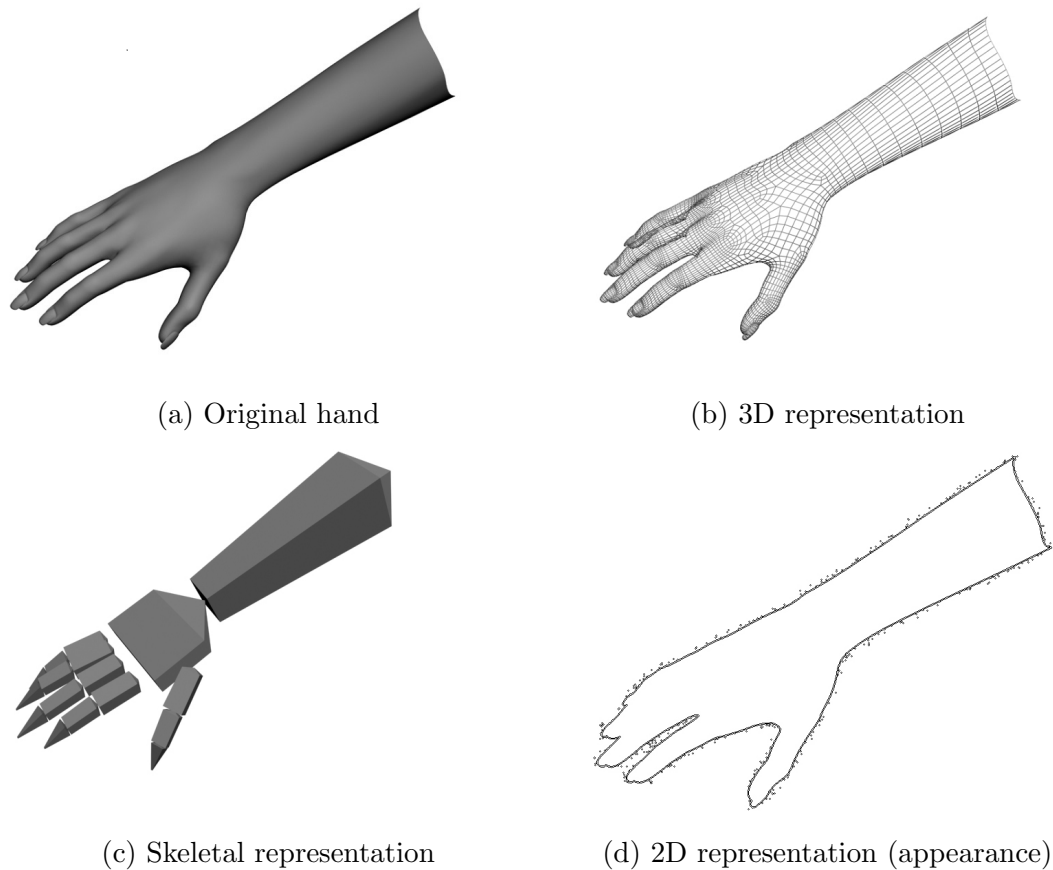


Figure 3.3: The different digital representations of a hand

3.3.2 Input devices

While there is a big quantity of studies conducted in gesture recognition focusing on image/video, there is a certain variation among applications within the instruments and implementations used.

1. Wired gloves

Using magnetic or inertial monitoring devices, wired gloves can provide information to the computer about the hands' position and rotation. In addition, some gloves can sense the curling of fingers with a high degree of precision (5-10 degrees) or even provide the user with haptic feedback, which is a model of touch sensation.

2. Depth-aware cameras

Using dedicated cameras like structured light or time-of-flight cameras, one may create a depth map of what one sees through the camera in a limited range and use this information to approximate a 3D model of what one sees. Due to their limited range capacities, these can be efficient in detecting hand gestures.

3. Stereo cameras

Using two cameras whose positions are known to each other, the cameras' output can calculate an approximate 3D representation. A positioning reference such as a lexian-stripe or an infrared emitter can be used to get the distance between the cameras.

4. Gesture based controllers

These controllers function as an extension of the body so that most of their movement can be easily recorded by software while gestures are made. An instance of developing gesture-based movement recording is skeletal hand tracking, created for apps of virtual reality and augmented reality.

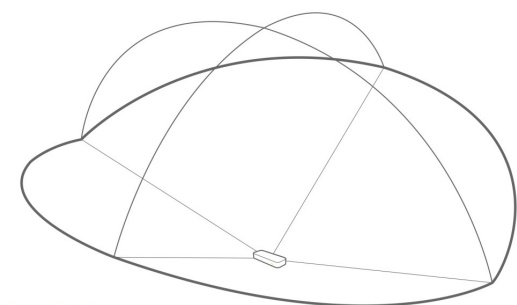
One of the more popular and accessible gesture detection devices is the Leap Motion controller, which is presented in the following section.

3.4 Leap Motion

The *Leap Motion Controller* is a tiny peripheral USB device intended to be facing upwards on a physical desktop, but can also be mounted on a VR headset.



(a) The LeapMotion controller



Interaction Area
2 feet above the controller, by 2 feet wide on each side
(150° angle), by 2 feet deep on each side (120° angle)

(b) The interaction area

Figure 3.4: The LeapMotion system

3.4.1 Hardware

The *Leap Motion Controller* is really quite straightforward from a hardware view. Two cameras and three infrared LEDs are the core of the device. These track infrared light with a wavelength of 850 nanometers, which is outside the visible light spectrum. [7]

The unit has a big interaction room of 0.22 m³ thanks to its wide-angle glasses, which takes the form of an inverted pyramid – the intersection of the areas of perspective of the binocular cameras (see figure 3.4b). The viewing range of the device is 60cm to 80cm, depending on the version of the firmware used.

This raw data is then stored in the device’s local memory and then sent via USB to the *Leap Motion tracking software*. As the cameras work with near-infrared light, the data is in the form of grayscale stereo images, as shown in figure 3.5.

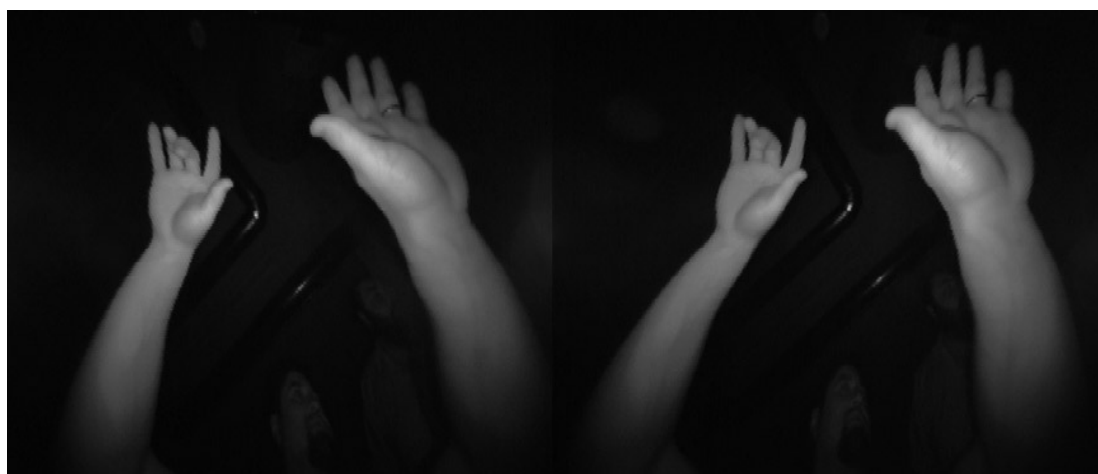


Figure 3.5: Leap Motion raw data

3.4.2 Software

It’s time for some heavy mathematical lifting once the picture information is streamed to the computer. The *Leap Motion Controller* does not produce depth maps despite common misconceptions - instead it applies sophisticated algorithms to the raw sensor information.

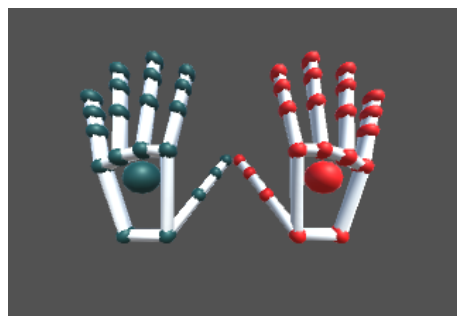


Figure 3.6: Capsule hands

The Leap Motion Service first compensates background objects (e.g. head) and lightning, and then extracts from the data the relevant information - arms, hands and fingers.

Though a transport layer, the results(frames) are fed to the *Leap Motion Control Panel* or to native and web clients. These organize the data into an object-oriented structure.

Although the device itself has been the same since it's launch in 2013, the software has undergone several do-overs and upgrades. Their SDK started with desktop-only tracking capabilities for its first two major versions. In 2014, they added a VR tracking mode to it and in 2016 they released Orion, Leap Motion's VR-dedicated SDK.

3.4.3 Gestures and detectors

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

1. **Arm**

The name of this data structure is a bit misleading because it actually represents the forearm - the lower part of the arm, from elbow down to wrist. There are a maximum of two arms in the scene, each with only one **Hand** attached to it.

2. **Hand**

There can be a maximum of two hands in the scene at any given time. Hands have a special attribute called *handedness*, which determines if the hand is *left* or *right*. Each hand has exactly five fingers attached to it. If the Leap Motion controller cannot see one of the fingers (because it is obscured by the hand or by another object), it will try to determine its pose based on past data.

3. **Finger**

Each finger has three joints that can be used to attach new visual models to the hands (e.g. different colour capsule hands, natural looking hands). Fingers only have data about their tip position and direction, and the four bones inside them. Fingers can be of one of five types - *thumb*, *index*, *middle*, *ring* or *pinky*.

4. **Bone**

Bones can be of one of four type - *metacarpal*, *proximal phalange*, *intermediate phalange* or *distal phalange*. Even though the thumb does not have a *metacarpal* bone in it in the real world, Leap Motion decided to add a *zero length metacarpal bone* to the thumb, so that the fingers are kept consistent regardless of their type (see figure 3.7).

Leap Motion 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 (figure 3.8a)

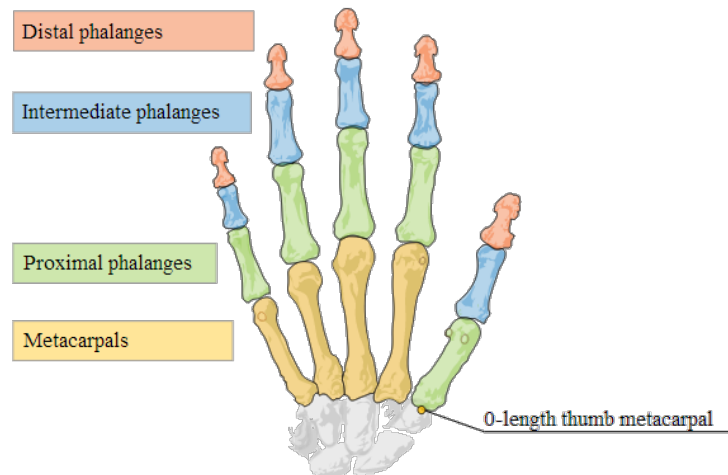
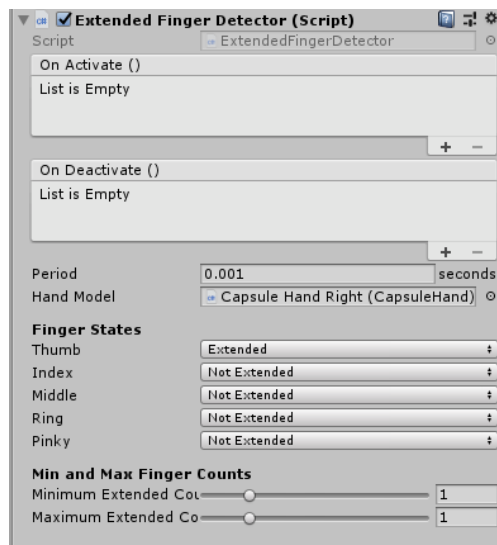
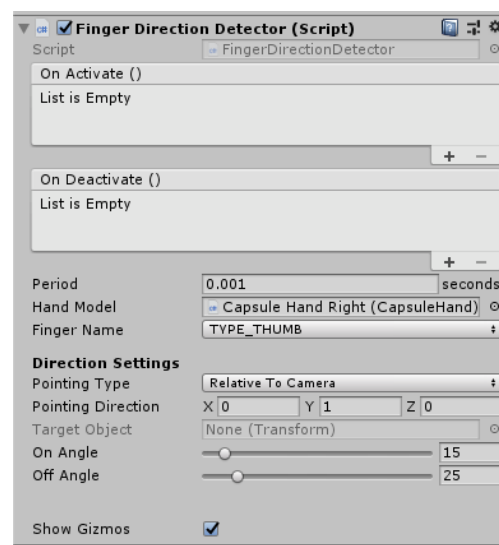


Figure 3.7: Leap Motion bones

- **Finger Pointing Detector** - configured to detect that the thumb is pointing up ($Vector3(0, 0, 1)$) relative to the horizon (figure 3.8b)



(a) Leap Thumb extended detector



(b) Leap Thumb pointing up detector

Figure 3.8: Leap Motion basic detectors for the "thumbs-up" gesture

- **And Logic Gate** - to combine the other two detectors and have callbacks (C# scripts) attached to it (figure ref 3.9)

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

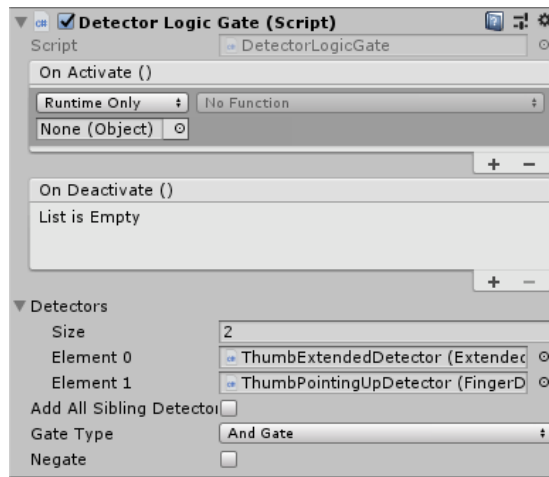


Figure 3.9: Leap Thumb pointing up detector by combining the detectors 3.8a and 3.8b

Logic Gate. This can quickly get out of hand when requiring a high number of combined gestures. The logic gate detector is also highly coupled to the other two detectors, and any change in the base detectors (conditions, renaming or, the worse, moving) will come with a change in the logic gate detector. Even though, with this approach, the basic detectors are reusable, the logic gates usually are not.

This issues fuel the need for a higher level API that is flexible and easily extendable, and which has high reusability. The API should adhere to a programming model which reacts to change rather than query components for changes so that it gives high performance.

3.5 Reactive programming

Reactive programming is a declarative programming paradigm concerned with asynchronous data streams and the propagation of change. With this paradigm it is feasible to easily express static (e.g. lists) or dynamic (e.g. events) information streams and to also indicate that an implied dependency remains within the related implementation model, which promotes the automatic propagation of the altered information stream.

Examples of Reactive Programming include hardware description languages (HDLs), such as VHDL or Verilog, in which changes are modeled as they propagate through a circuit. As a manner to optimize the development of dynamic user interfaces and virtually-real-time system animation, reactive programming has been suggested.

3.5.1 Reactive Extensions

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

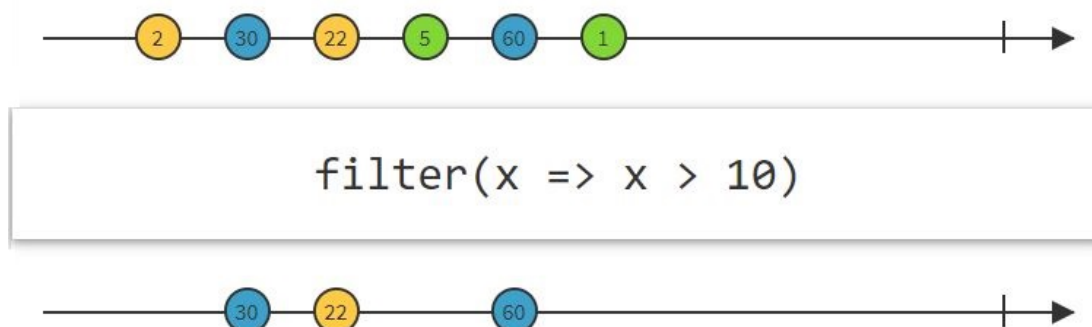


Figure 3.10: Example of a RX operator

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 3.10 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)

Code snippets 3.1 and 3.2 show the resemblance between the iterable and 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.

```

GetDataFromLocalMemory()
    .Skip(10)
    .Take(5)
    .Select(s -> $"{s} transformed")
    .ForEach(s -> Console.WriteLine($"next -> {s}"));

```

Listing 3.1: Iterable

```

GetDataFromNetwork()
    .Skip(10)
    .Take(5)
    .Select(s -> $"{s} transformed")
    .ForEach(s -> Console.WriteLine($"onNext -> {s}"));

```

Listing 3.2: Observable

ReactiveX observables are intended to be ***composable, flexible*** and ***less opinionated***. These provide a huge advantage over structures like Java *Futures* or C# *Awaitables*, because it removes the need for ambiguous nesting of callbacks.

RX Observables also offer three methods for of flow control - *OnNext*, *OnError* and *OnCompleted* - which give the programmer a high degree liberty. Table 3.1 shows how observables integrate in the programming world, at the crossroads of asynchronous multiple items data streams.

	single items	multiple items
synchronous	T GetData	$IEnumerable<T>$ GetData
asynchronous	$Awaitable<T>$ GetData	$Observable<T>$ GetData

Table 3.1: Observable position in multiple items and asynchronous world

Chapter 4

Analysis and Theoretical Foundation

4.1 Conceptual architecture

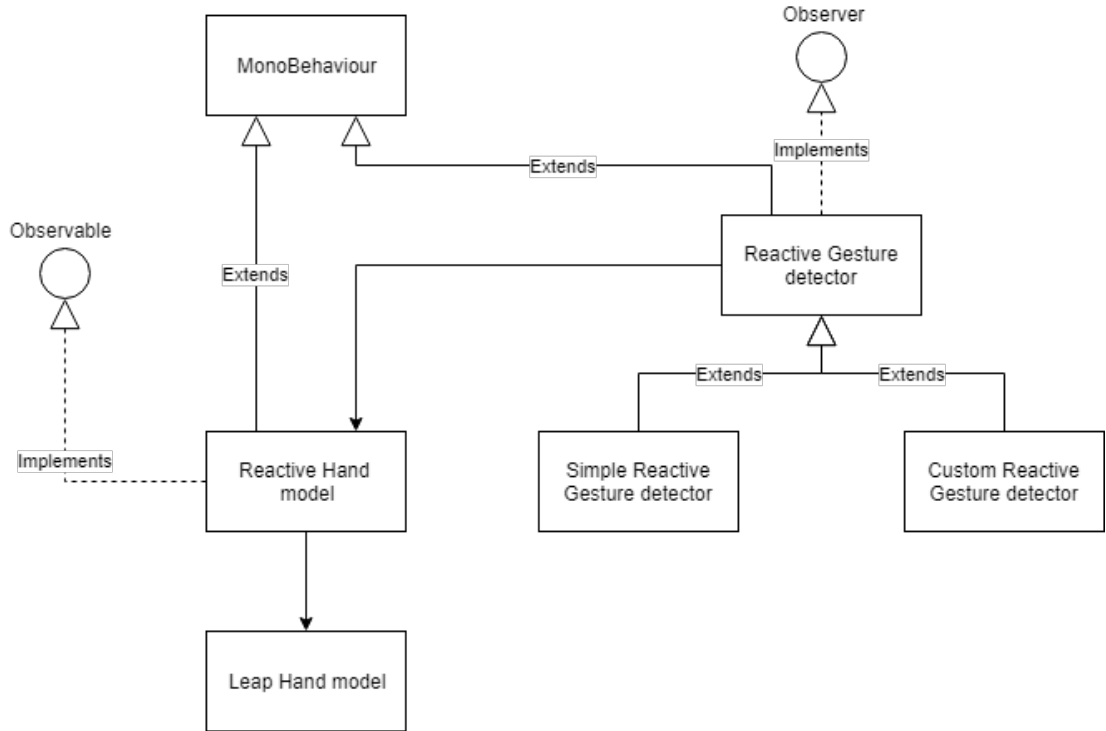
This section describes in detail the proposed conceptual architecture of the system presented in the previous chapters.

1. The system should be an extension of the Leap Motion Unity Asset, so it uses Leap hand models as input.
2. The application should adhere to the Reactive Programming paradigm, which implies using the *Observer* and *Observable* interfaces.
3. The system should be available in the Untiy Game Engine requires that some (or all) its components extend *MonoBehaviour* abstract class.
4. The system must allow the developer using it to define custom gestures, but it must also present some basic gestures that can be used out of the box.

4.1.1 Leap hand model

The Leap hand model components represents the actual Leap Hands given in Leap Motion's Unity SDK. With the current model used by Leap Motion, only gestures that imply moving hands and fingers can be detected (not arms and bones). Finger gestures cannot be defined independent of the hand the finger is attached to, so there is no value added by extending the finger models with reactive models.

This component refers to three concepts from Leap Motion's Unity SDK - left hand, right hand and hands (as a pair). All finger related gestures should be defined against hands so as to add more value to the system and increase usability and flexibility.

Figure 4.1: Conceptual architecture of *Fluent Motion*

4.1.2 Reactive hand model

The Reactive hand model is the bottommost layer of the system. It can be seen as a data access layer from classical layered architectures or even as a services layer. This component's purpose is to map the Leap Hand models to a ReactiveX publish-subscribe model. It is trivial to deduce, together with requirement 2 that this component should implement the *Observable* (*publisher*) interface. In order to achieve high cohesion, the component should also be responsible of notifying *observers* (*subscribers*) when the state of the underlying Leap hand model changes.

This component should also be integrated with Unity (as of requirement 3). In order to make it usable in the game editor interface, it must extend the *MonoBehaviour* abstract class from Unity.

4.1.3 MonoBehaviour

MonoBehaviour is the base class from which every Unity script derives [9]. It provides seven overridable functions for creating applications.

1. Start

Start is called on the frame when a script is enabled just before any of the Update methods are called the first time. Start is called exactly once in the script's lifetime.

2. **Update**

If the *MonoBehaviour* is enabled, *Update* is called every frame. It is the most commonly used function for game scripting.

3. **FixedUpdate**

FixedUpdate is a framerate independent function used for physics calculations (collisions, object gravity). The rate at which it is called is dependent on the Physics system.

4. **LateUpdate** *LateUpdate* is called after all *Update* functions in order to order script execution.

5. **OnGUI**

OnGUI is called when a new event is received from the GUI or when a new render of UI elements is required. This function can be called multiple times per frame.

6. **OnDisable**

This function is called when the *MonoBehaviour* is disabled or destroyed.

7. **OnEnable** This function is called when the *MonoBehaviour* is enabled and becomes active.

4.1.4 Reactive Gesture detector

This component should contain most of the logic needed for creating detectors, without reducing extendability. As per requirement 3, it should be usable in the game editor and, as such, should extend the *MonoBehaviour* abstract class. Trivially, it is easy to observe that the component should be the *observer* which subscribes to *Reactive hand model's publish* method.

4.1.4.1 Simple Reactive Gesture detector

This component will be present in the *Fluent Motion* asset and provide implementations of basic gestures, such as swiping, finger pointing to some object, etc. It must present the same interface as any other detector, simple or custom.

4.1.4.2 Custom Reactive Gesture detector

While not actually present in the *Fluent Motion* asset at the time of releasing, this component represents the custom gestures that developers can add when using the asset. It must present the same interface as any other detector, simple or custom.

4.2 Create Gesture detector

4.2.1 Brief description

The purpose of this section is to capture the flow of events that an actor must follow in order to manage the creation of a gesture detector in a custom scenario.

4.2.2 Primary actor

Developer using FluentMotion

4.2.3 Stakeholders and interests

Developer using FluentMotion – interested in detecting a new gesture in order to do an action based on that gesture.

4.2.4 Flow of events

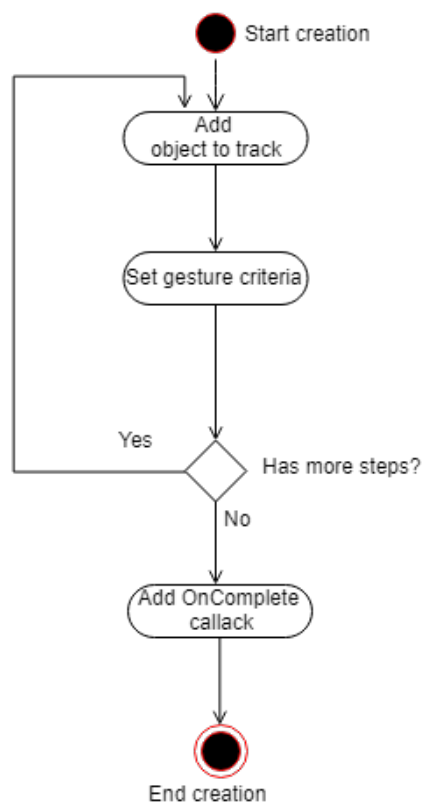


Figure 4.2: Create gesture detector flow

1. Basic flow

- (a) **Use case start** - this use case starts when the actor wants to add a new gesture to his/her application.
- (b) The actor sets the object to be tracked for this gesture (Hand(s)/Finger(s)).
- (c) The actor sets the criteria for detecting this part of the gesture.
- (d) The actor adds a flow to be called when the gesture was detected successfully.
- (e) **Use-Case End** - the actor ends the creation operation at step 1e

2. Alternate flows

This flow can occur at step 1c i.e. the actor wants to add another step to the gesture. When this happens, go to step 1b.

4.2.5 Preconditions

- 1. The actor has a Unity project with the FluentMotion asset referenced.
- 2. The new gesture conforms to the Reactive Gesture Detector component's interface.

4.2.6 Postconditions

- 1. The new gesture should not influence other gestures' behavior.
- 2. The new gesture should not add too much overhead such that the rendering process and detection can happen in real time.
- 3. When the gesture is successfully detected, its associated callback is correctly invoked as described in its own flow.

4.3 Add Simple Gesture Detector

4.3.1 Brief description

The purpose of this section is to capture the flow of events that an actor must follow in order to manage adding a simple gesture detector in a custom scenario.

4.3.2 Primary actor

Developer using Fluent Motion.

4.3.3 Stakeholders and interests

Developer using Fluent Motion - interested in performing an action when a simple gesture is detected.

4.3.4 Flow of events

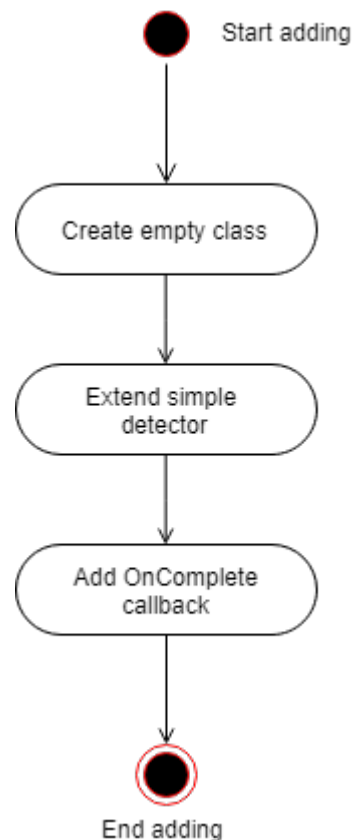


Figure 4.3: Add Simple Gesture Detector flow

1. Basic flow

- (a) **Use case start** - this use case starts when the actor wants to add a simple gesture detector to his/her application.
- (b) The actor creates an empty class.
- (c) The actor extends the Simple Detector class he/she needs.
- (d) The actor overrides the OnComplete method in the new class.
- (e) **Use-Case End** - the actor ends the addition operation 1e.

2. Alternate flows

There are no alternative flows.

4.3.5 Preconditions

1. The actor has a Unity project with the FluentMotion asset referenced.

4.3.6 Postconditions

1. When the gesture is successfully detected, its associated callback is correctly invoked as described in its own flow.

Chapter 5

Detailed Design and Implementation

5.1 Fluent Motion simple gestures

Fluent Motion defines a total of ten simple gestures that can be detected by the Leap Motion controller. The methods presented below each have multiple implementations, so as not to put too much pressure on the user regarding the syntax he or she should use.

5.1.1 Finger Gestures

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

5.1.2 Hand Gestures - single hand

1. **IsPinching** - hand is pinching (as of Orion 4.4, i.e. *when PinchStrenght > 0.8*)
2. **PalmIsFacing** - palm is facing a given target (can be any object that has a mapping to a Unity *Vector3*) with a given angle tolerance
3. **IsFist** - hand is making a fist (i.e. *FistStrenght > 0.8*)
4. **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)

5.1.3 Hand Gestures - both hands

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

Besides the enumerated gestures, several simple detectors are present in the form of *abstract* classes, which only need the *OnDetect* callback to be implemented in order to make use of the detector.

5.2 When<TInput, TResult> method

Applies a filter which is the result of multiple combined conditions to an observable.

5.2.1 Syntax

```
public static IObservable<TInput> When<TInput, TResult>(this
    IObservable<TInput> observable,
    Func<bool, bool, bool> reducer,
    Func<TInput, TResult> selector,
    params Func<TResult, bool>[] conditions)
```

Listing 5.1: Declaration

```
Hand.When((a, b) => a && b, //reducer
    hand => hand.GetThumb(), //selector
    finger => finger.IsExtended,
    finger => IsPointingTo(finger, someTarget))
    .Subscribe(x => DoSomething(x));
```

Listing 5.2: Usage example

5.2.2 Type parameters

1. **TInput**
The type of the input and output observables.
2. **TResult**
The type of the predicates' input and selector output.

5.2.3 Parameters

- **observable**

Type: *System.IObservable<TInput>*

An observable sequence whose elements to filter.

- **reducer**

Type: *System.Func<bool, bool, bool>*

A reduction function that combines the predicates of *conditions* to a single value.

- **selector**

Type: *System.Func<TInput, TResult>*

A function that is applied to each element of the source observable before being tested against the *conditions*.

- **conditions**

Type: *Func<TResult, bool>[]*

One or more functions to test each source element for conditions.

5.2.4 Return Value

Type: *System.IObservable<TInput>*

An observable sequence that contains elements from the input sequence that satisfy the combined conditions.

5.2.5 Remarks

You can call this method as an instance method on any object of type *IObservable<TInput>*. When you use instance method syntax to call this method, omit the first parameter.

5.2.6 Overloads

1. **When<TInput, TResult>(IObservable<TInput>, Func<TInput, TResult>, params Func<TResult, bool>[])**

```
public static IObservable<TInput> When<TInput, TResult>(
    this IObservable<TInput> observable,
    Func<TInput, TResult> selector,
    params Func<TResult, bool>[] conditions
)
```

Listing 5.3: Declaration

This method replaces the *Func<bool, bool, bool> reducer* parameter from the original *When* with the AND boolean operator.

2. **When<T>(IObservable<T>, params Func<T, bool>[])**

```
public static IObservable<T> When<T>(
    this IObservable<T> observable,
    params Func<T, bool>[] conditions
)
```

Listing 5.4: Declaration

This method replaces the *Func<bool, bool, bool> reducer* parameter from the original *When* with the AND boolean operator and the *Func<T, T> selector* with the identity mapping. (see code snippet 5.5)

```
public static T Identity<T>(T t){
    return t;
}
```

Listing 5.5: Identity mapping

5.3 Fingers method

Projects each hand element from an observable to the list of specified fingers.

5.3.1 Syntax

```
public static IObservable<IList<Finger>> Fingers(
    this IObservable<Hand> observable,
    params Finger.FingerType[] fingerTypes
)
```

Listing 5.6: Declaration

```
Hand.Fingers(
    Finger.FingerType.TYPE_THUMB,
    Finger.FingerType.TYPE_INDEX)
    .Subscribe(list => DoSomething(list));
```

Listing 5.7: Usage example

5.3.2 Parameters

- **observable**
Type: *System.IObservable<Leap.Hand>*
An observable sequence of hand states.
- **fingerTypes**
Type: *Leap.Finger.FingerType[]*
One or more finger types to select.

5.3.3 Return value

Type: *System.IObservable<IList<Leap.Finger>>*
An observable sequence containing fingers' states.

5.3.4 Remarks

You can call this method as an instance method on any object of type *IObservable<Hand>*. When you use instance method syntax to call this method, omit the first parameter.

5.3.5 Overloads

1. Fingers(IObservable<Hand>)

```
public static IObservable<IList<Finger>> Fingers(  
    this IObservable<Hand> observable  
)
```

Listing 5.8: Declaration

This method returns an observable containing the states of **ALL** the fingers on the hand.

5.4 GetFinger method

Projects each hand element from an observable sequence to one specified finger.

5.4.1 Syntax

```
public static IObservable<Finger> Finger(  
    this IObservable<Hand> observable,  
    Finger.FingerType fingerType  
)
```

Listing 5.9: Declaration

```
Hand.Finger(Finger.FingerType.TYPE_THUMB)  
    .Subscribe(finger => DoSomething(finger));
```

Listing 5.10: Usage example

5.4.2 Parameters

- **observable**
Type: *System.IObservable<Leap.Hand>*
An observable sequence of hand states.
- **fingerType**
Type: *Leap.Finger.FingerType*
The finger type of the required finger.

5.4.3 Return value

Type: *System.IObservable<Leap.Finger>*
An observable sequence containing a finger's states.

5.4.4 Remarks

You can call this method as an instance method on any object of type *IObservable<Hand>*. When you use instance method syntax to call this method, omit the first parameter.

5.5 AreExtended method

Filters the hand states in which the specified fingers are extended.

5.5.1 Syntax

```
public static IObservable<Hand> AreExtended(  
    this IObservable<Hand> observable,  
    params Finger.FingerType[] fingerTypes  
)
```

Listing 5.11: Declaration

```
Hand.AreExtended(  
    Finger.FingerType.TYPE_THUMB,  
    Finger.FingerType.TYPE_RING)  
    .Subscribe(hand => DoSomething(hand));
```

Listing 5.12: Usage example

5.5.2 Parameters

- **observable**
Type: *System.IObservable<Leap.Hand>*
An observable sequence of hand states.
- **fingerTypes**
Type: *Leap.Finger.FingerType[]*
The finger types of the tested fingers.

5.5.3 Return value

Type: *System.IObservable<Leap.Hand>*

An observable sequence containing the hand's states when the specified fingers are extended.

5.5.4 Remarks

You can call this method as an instance method on any object of type *IObservable<Hand>*. When you use instance method syntax to call this method, omit the first parameter.

5.6 Thumb method (selector)

Projects a hand state observable sequence to a thumb state observable sequence.

5.6.1 Syntax

```
public static IObservable<Finger> Thumb(  
    this IObservable<Hand> observable  
)
```

Listing 5.13: Declaration

```
Hand.Thumb()  
    .Subscribe(thumb => DoSomething(thumb));
```

Listing 5.14: Usage example

5.6.2 Parameters

- **observable**
Type: *System.IObservable<Leap.Hand>*
An observable sequence of hand states.

5.6.3 Return values

Type: *System.IObservable<Leap.Finger>*
An observable sequence containing the thumb's states.

5.6.4 Remarks

You can call this method as an instance method on any object of type *IObservable<Hand>*. When you use instance method syntax to call this method, omit the first parameter.

5.7 Thumb method (filter)

Filters a hand state observable by conditions on the thumb.

5.7.1 Syntax

```
public static IObservable<Hand> Thumb(  
    this IObservable<Hand> observable,  
    params Func<Finger, bool>[] conditions  
)
```

Listing 5.15: Declaration

```
Hand.Thumb(  
    thumb => thumb.IsExtended)  
    .Subscribe(thumb => DoSomething(thumb));
```

Listing 5.16: Usage example

5.7.2 Parameters

- **observable**
Type: *System.IObservable<Leap.Hand>*
An observable sequence of hand states.
- **conditions**
Type: *System.Func<Leap.Finger, bool>[]*
One or more conditions to test the thumb against.

5.7.3 Return values

Type: *System.IObservable<Leap.Hand>*

An observable sequence containing the hand states which satisfy the conditions on the thumb.

5.7.4 Remarks

You can call this method as an instance method on any object of type *IObservable<Hand>*. When you use instance method syntax to call this method, omit the first parameter.

5.8 Index method (selector)

Projects a hand state observable sequence to a thumb state observable sequence.

5.8.1 Syntax

```
public static IObservable<Hand> Index(  
    this IObservable<Hand> observable  
)
```

Listing 5.17: Declaration

```
Hand.Index()  
    .Subscribe(index => DoSomething(index));
```

Listing 5.18: Usage example

5.8.2 Parameters

- **observable**

Type: *System.IObservable<Leap.Hand>*

An observable sequence of hand states.

5.8.3 Return values

Type: *System.IObservable<Leap.Finger>*

An observable sequence containing the index's states.

5.8.4 Remarks

You can call this method as an instance method on any object of type *IObservable<Hand>*. When you use instance method syntax to call this method, omit the first parameter.

5.9 Index method (filter)

Filters a hand state observable by conditions on the index.

5.9.1 Syntax

```
public static IObservable<Hand> Index(
    this IObservable<Hand> observable,
    params Func<Finger, bool>[] conditions
)
```

Listing 5.19: Declaration

```
Hand.Index(
    index => index.IsExtended)
    .Subscribe(index => DoSomething(index));
```

Listing 5.20: Usage example

5.9.2 Parameters

- **observable**
Type: *System.IObservable<Leap.Hand>*
An observable sequence of hand states.
- **conditions**
Type: *System.Func<Leap.Finger, bool>[]*
One or more conditions to test the index against.

5.9.3 Return values

Type: *System.IObservable<Leap.Hand>*
An observable sequence containing the hand states which satisfy the conditions on the index.

5.9.4 Remarks

You can call this method as an instance method on any object of type *IObservable<Hand>*. When you use instance method syntax to call this method, omit the first parameter.

5.10 Middle method (selector)

Projects a hand state observable sequence to a middle finger state observable sequence.

5.10.1 Syntax

```
public static IObservable<Hand> Middle(  
    this IObservable<Hand> observable  
)
```

Listing 5.21: Declaration

```
Hand.Middleware()  
    .Subscribe(middle => DoSomething(middle));
```

Listing 5.22: Usage example

5.10.2 Parameters

- **observable**

Type: *System.IObservable<Leap.Hand>*

An observable sequence of hand states.

5.10.3 Return values

Type: *System.IObservable<Leap.Finger>*

An observable sequence containing the middle finger's states.

5.10.4 Remarks

You can call this method as an instance method on any object of type *IObservable<Hand>*. When you use instance method syntax to call this method, omit the first parameter.

5.11 Middle method (filter)

Filters a hand state observable by conditions on the middle finger.

5.11.1 Syntax

```
public static IObservable<Hand> Middle(  
    this IObservable<Hand> observable,  
    params Func<Finger, bool>[] conditions  
)
```

Listing 5.23: Declaration

```
Hand.Middleware(  
    middle => middle.IsExtended)  
    .Subscribe(middle => DoSomething(middle));
```

Listing 5.24: Usage example

5.11.2 Parameters

- **observable**
Type: *System.IObservable<Leap.Hand>*
An observable sequence of hand states.
- **conditions**
Type: *System.Func<Leap.Finger, bool>[]*
One or more conditions to test the middle finger against.

5.11.3 Return values

Type: *System.IObservable<Leap.Hand>*

An observable sequence containing the hand states which satisfy the conditions on the middle finger.

5.11.4 Remarks

You can call this method as an instance method on any object of type *IObservable<Hand>*. When you use instance method syntax to call this method, omit the first parameter.

5.12 Ring method (selector)

Projects a hand state observable sequence to a ring finger state observable sequence.

5.12.1 Syntax

```
public static IObservable<Hand> Ring(  
    this IObservable<Hand> observable  
)
```

Listing 5.25: Declaration

```
Hand.Ring()  
    .Subscribe(ring => DoSomething(ring));
```

Listing 5.26: Usage example

5.12.2 Parameters

- **observable**
Type: *System.IObservable<Leap.Hand>>*
An observable sequence of hand states.

5.12.3 Return values

Type: *System.IObservable<Leap.Finger>>*

An observable sequence containing the ring finger's states.

5.12.4 Remarks

You can call this method as an instance method on any object of type *IObservable<Hand>*. When you use instance method syntax to call this method, omit the first parameter.

5.13 Ring method (filter)

Filters a hand state observable by conditions on the ring finger.

5.13.1 Syntax

```
public static IObservable<Hand> Ring(
    this IObservable<Hand> observable,
    params Func<Finger, bool>[] conditions
)
```

Listing 5.27: Declaration

```
Hand.Ring(
    ring => ring.IsExtended)
    .Subscribe(ring => DoSomething(ring));
```

Listing 5.28: Usage example

5.13.2 Parameters

- **observable**
Type: *System.IObservable<Leap.Hand>*
An observable sequence of hand states.
- **conditions**
Type: *System.Func<Leap.Finger, bool>[]*
One or more conditions to test the ring finger against.

5.13.3 Return values

Type: *System.IObservable<Leap.Hand>*

An observable sequence containing the hand states which satisfy the conditions on the ring finger.

5.13.4 Remarks

You can call this method as an instance method on any object of type *IObservable<Hand>*. When you use instance method syntax to call this method, omit the first parameter.

5.14 Pinky method (selector)

Projects a hand state observable sequence to a pinky finger state observable sequence.

5.14.1 Syntax

```
public static IObservable<Hand> Pinky(  
    this IObservable<Hand> observable  
)
```

Listing 5.29: Declaration

```
Hand.Pinky()  
    .Subscribe(pinky => DoSomething(pinky));
```

Listing 5.30: Usage example

5.14.2 Parameters

- **observable**

Type: *System.IObservable<Leap.Hand>*

An observable sequence of hand states.

5.14.3 Return values

Type: *System.IObservable<Leap.Finger>* An observable sequence containing the pinky finger's states.

5.14.4 Remarks

You can call this method as an instance method on any object of type *IObservable<Hand>*. When you use instance method syntax to call this method, omit the first parameter.

5.15 Pinky method (filter)

Filters a hand state observable by conditions on the pinky finger.

5.15.1 Syntax

```
public static IObservable<Hand> Pinky(  
    this IObservable<Hand> observable,  
    params Func<Finger, bool>[] conditions  
)
```

Listing 5.31: Declaration

```
Hand.Pinky(  
    pinky => pinky.IsExtended)  
    .Subscribe(pinky => DoSomething(pinky));
```

Listing 5.32: Usage example

5.15.2 Parameters

- **observable**
Type: *System.IObservable<Leap.Hand>*
An observable sequence of hand states.
- **conditions**
Type: *System.Func<Leap.Finger, bool>[]*
One or more conditions to test the pinky finger against.

5.15.3 Return values

Type: *System.IObservable<Leap.Hand>*
An observable sequence containing the hand states which satisfy the conditions on the pinky finger.

5.15.4 Remarks

You can call this method as an instance method on any object of type *IObservable<Hand>*. When you use instance method syntax to call this method, omit the first parameter.

5.16 IsPinching method

Filters an observable hand state sequence based on the *pinching* gesture.

5.16.1 Syntax

```
public static IObservable<Hand> IsPinching(  
    this IObservable<Hand> observable  
)
```

Listing 5.33: Declaration

```
Hand.IsPinching()  
    .Subscribe(hand => DoSomething(hand));
```

Listing 5.34: Usage example

5.16.2 Parameters

- **observable**

Type: *System.IObservable<Leap.Hand>*

An observable sequence of hand states.

5.16.3 Return values

Type: *System.IObservable<Leap.Hand>*

An observable sequence containing hand states which are pinching.

5.16.4 Remarks

You can call this method as an instance method on any object of type *IObservable<Hand>*. When you use instance method syntax to call this method, omit the first parameter.

5.17 PalmIsFacing<TPlayer, TTarget> method

Filters the hand state observable based on the palm normal direction.

5.17.1 Syntax

```
public static IObservable<Hand> PalmIsFacing<TPlayer,
    TTarget>(
    this IObservable<Hand> observable,
    TPlayer player,
    Func<TPlayer, Vector4> positionSelector,
    TTarget target,
    Func<TTarget, Vector4> targetPositionSelector,
    Func<float> tolerance
)
```

Listing 5.35: Declaration

```
Hand.PalmIsFacing(
    Player.instance,
    player => player.hmdTransform.position,
    gameObject,
    target => target.transform.position
)
.Subscribe(hand => DoSomething(hand));
```

Listing 5.36: Usage example

5.17.2 Type parameters

1. TPlayer

The type of the player reference object. For Virtual Reality applications, this is generally *Valve.VR.InteractionSystem.Player*, and for desktop applications *UnityEngine.Camera*.

2. TTarget

The type of the target game object.

5.17.3 Parameters

- **observable**

Type: *System.IObservable<Leap.Hand>*
An observable sequence of hand states.

- **player**
Type: *TPlayer*
The player instance.
- **positionSelector**
Type: *System.Func<TPlayer, UnityEngine.Vector4>*
A function mapping the player instance to a position in the scene.
- **target**
Type: *TTarget*
The target object.
- **targetPositionSelector**
Type: *System.Func<TTarget, UnityEngine.Vector4>*
A function mapping the target object to a position in the scene.
- **tolerance**
Type: *System.Func<float>*
A function supplying the value of the tolerance between the palm direction and the target direction. The tolerance can thus change over time.

5.18 Return value

Type: *System.IObservable<Leap.Hand>*

An observable sequence of hand states where the palm faces the target object with the given tolerance.

5.18.1 Remarks

You can call this method as an instance method on any object of type *IObservable<Hand>*. When you use instance method syntax to call this method, omit the first parameter.

5.18.2 Overloads

1. **PalmIsFacing<TPlayer, TTarget>(IObservable<Hand>, TPlayer, Func<TPlayer, Vector4>, TTarget, Func<TTarget, Vector4>, float)**

```
public static IObservable<Hand> PalmIsFacing<TPlayer,
    TTarget>(
    this IObservable<Hand> observable,
    TPlayer player,
    Func<TPlayer, Vector4> positionSelector,
    TTarget target,
```

```

    Func<TTarget, Vector4> targetPositionSelector,
    float tolerance = 0.075f
)

```

Listing 5.37: Declaration

This method takes a *float* for the **tolerance** parameter instead of a function returning a floating point value. Useful for cases when the tolerance should be constant.

2. **PalmIsFacing<TTarget>(IObservable<Hand>, TTarget, Func<TTarget, Vector4>, Func<float>)**

```

public static IObservable<Hand> PalmIsFacing<TTarget>(
    this IObservable<Hand> observable,
    TTarget target,
    Func<TTarget, Vector4> selector,
    Func<float> tolerance
)

```

Listing 5.38: Declaration

This method uses the *Valve.VR.InteractionEngine.Player* instance for the **player** parameter and selects the head-mounted display position for the **positionSelector** parameter. Useful for VR applications.

3. **PalmIsFacing<TTarget>(IObservable<Hand>, TTarget, Func<TTarget, Vector4>, float)**

```

public static IObservable<Hand> PalmIsFacing<TTarget>(
    this IObservable<Hand> observable,
    TTarget target,
    Func<TTarget, Vector4> selector,
    float tolerance = 0.075f)

```

Listing 5.39: Declaration

This method uses the *Valve.VR.InteractionEngine.Player* instance for the **player** parameter, selects the head-mounted display position for the **positionSelector** parameter and accepts a *float* parameter for the tolerance. Useful for VR applications in which the tolerance is constant.

5.19 IsFist method

Filters the hand state observable based on the *fist* gesture.

5.19.1 Syntax

```
public static IObservable<Hand> IsFist(  
    this IObservable<Hand> observable  
)
```

Listing 5.40: Declaration

```
Hand.IsFist()  
    .Subscribe(hand => DoSomething(hand));
```

Listing 5.41: Usage example

5.19.2 Parameters

- **observable**
Type: *System.IObservable<Leap.Hand>*
An observable sequence of hand states.

5.19.3 Return value

Type: *System.IObservable<Leap.Hand>*
An observable sequence of hand states where the hand is making a fist gesture.

5.19.4 Remarks

You can call this method as an instance method on any object of type *IObservable<Hand>*. When you use instance method syntax to call this method, omit the first parameter.

5.20 IsMoving method

Filters the hand state observable based on the moving speed. By default, in order to detect the hand as moving, it must have a speed greater than 1 millimeter per second (mm/s).

5.20.1 Syntax

```
public static IObservable<Hand> IsMoving(  
    this IObservable<Hand> observable,  
    float speed = 0.1f  
)
```

Listing 5.42: Declaration

```
Hand.IsMoving(1.0f) //optional parameter  
    .Subscribe(hand => DoSomething(hand));
```

Listing 5.43: Usage example

5.20.2 Parameters

- **observable**

Type: *System.IObservable<Leap.Hand>*

An observable sequence of hand states.

- **speed**

Type: *float*

A floating point value representing the minimum speed required to detect a movement. Defaults to *0.1*.

5.20.3 Return value

Type: *System.IObservable<Leap.Hand>* An observable sequence of hand states where the hand is moving.

5.20.4 Remarks

You can call this method as an instance method on any object of type *IObservable<Hand>*. When you use instance method syntax to call this method, omit the first parameter.

5.21 IsMoving<TReference> method

Filters the hand state observable based on the moving direction and speed. By default, in order to detect the hand as moving, it must have a speed greater than 1 millimeter per second (mm/s).

5.21.1 Syntax

```
public static IObservable<Hand> IsMoving<TReference>(
    this IObservable<Hand> observable,
    TReference reference,
    Func<TReference, Vector4> to,
    float speed = 0.1f,
    float tolerance = 0.0.75f
)
```

Listing 5.44: Declaration

```
Hand.IsMoving(
    Player.instance,
    player => player.hmdTransform.right)
.Subscribe(hand => DoSomething(hand));
```

Listing 5.45: Usage example

5.21.2 Type parameters

1. TReference

The type of the movement direction reference object (e.g. main camera, player, HMD, etc).

5.21.3 Parameters

- **observable**

Type: *System.IObservable<Leap.Hand>*

An observable sequence of hand states.

- **reference**

Type: *TReference*

The movement direction reference coordinate axes.

- **to**

Type: *System.Func<TReference, UnityEngine.Vector4>*

A function which maps the *reference* to a position in the scene.

- **speed**

Type: *float*

A floating point value representing the minimum speed required to detect a movement. Defaults to *0.1*.

- **tolerance**

Type: *float* A floating point value representing the tolerance between the actual movement direction and the target movement direction. Defaults to *0.075*.

5.21.4 Return value

Type: *System.IObservable<Leap.Hand>>*

An observable sequence of hand states where the hand is moving in the target direction.

5.21.5 Remarks

You can call this method as an instance method on any object of type *IObservable<Hand>*. When you use instance method syntax to call this method, omit the first parameter.

5.21.6 Overloads

1. **IsMoving<TReference>(IObservable<Hand>, TReference, SwipeDirection, float, float)** where **TReference : Component**

```
public static IObservable<Hand> IsMoving<TReference>(
    this IObservable<Hand> observable,
    TReference reference,
    SwipeDirection direction,
    float speed = 0.1f,
    float tolerance = 0.075f
)
where TReference : Component
```

Listing 5.46: Declaration

This method accepts as parameter a *SwipeDirection* to select the target direction of the movement.

Chapter 6

Testing and Validation

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 6.1.

All the gestures are expected to be done by either the left or the right hand. In figure 6.1, a pinch gesture is made with the right hand, but the cube expects a "L" gesture, so it does nothing.

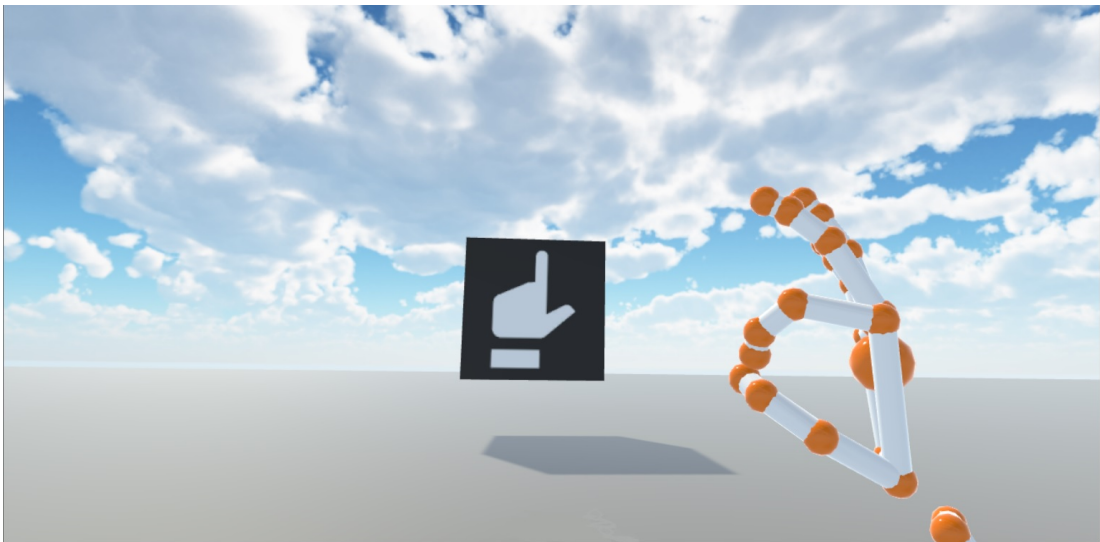


Figure 6.1: Cube not changing when an incorrect gesture is made






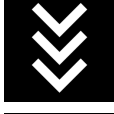


Gesture	Icon
Thumbs-up	
"L"	
Fist	
Pinch	
Swipe-up	
Swipe-down	
Swipe-left	
Swipe-right	

Table 6.1: Gesture to icon mappings

6.1 Performance

ReactiveX 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 ReactiveX 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 ReactiveX's *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.

Chapter 7

User's manual

7.1 Unity integration

After having the Leap Motion and SteamVR assets set up, adding Fluent Motion 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 7.1)

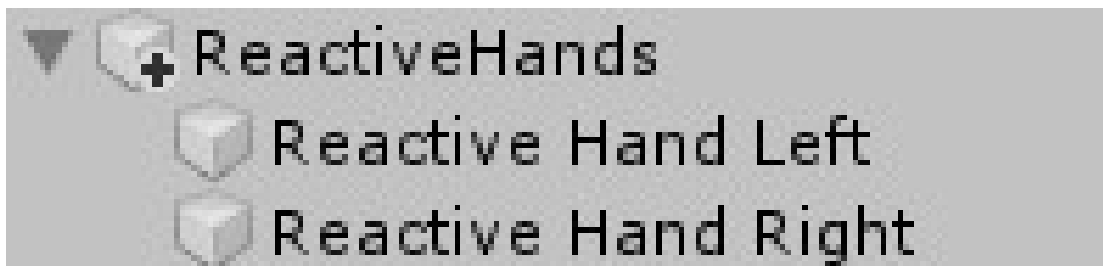


Figure 7.1: Setting up Fluent Motion game objects

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 *ReactiveHands*

7.1.1 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 7.3.

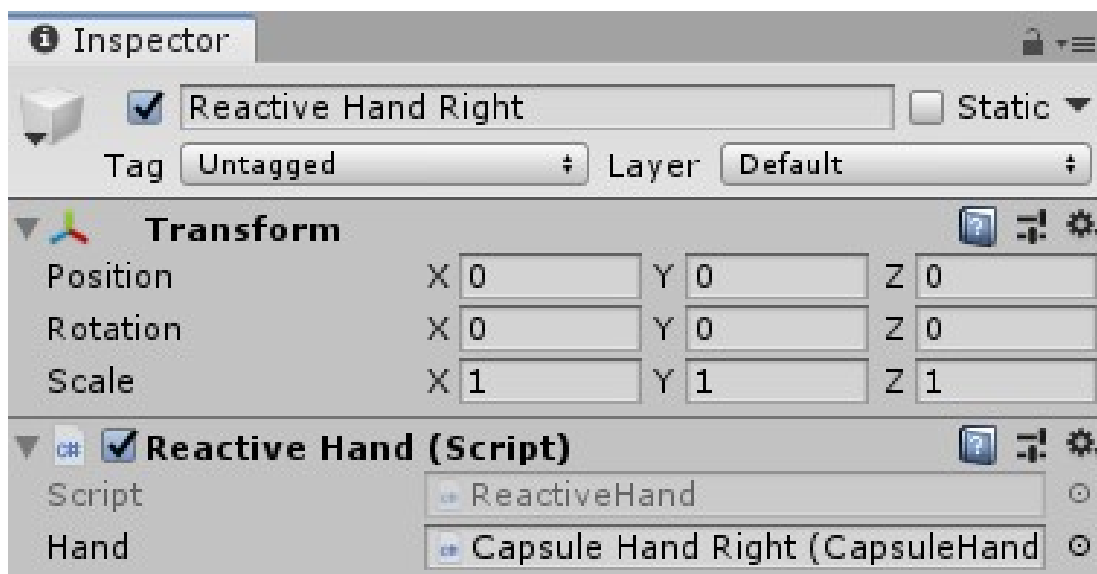


Figure 7.2: The *ReactiveHand* script attached to the game object

The code in figure 7.3 is one of the many ways for expressing this detector. Another option is presented in figure 7.4.

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.

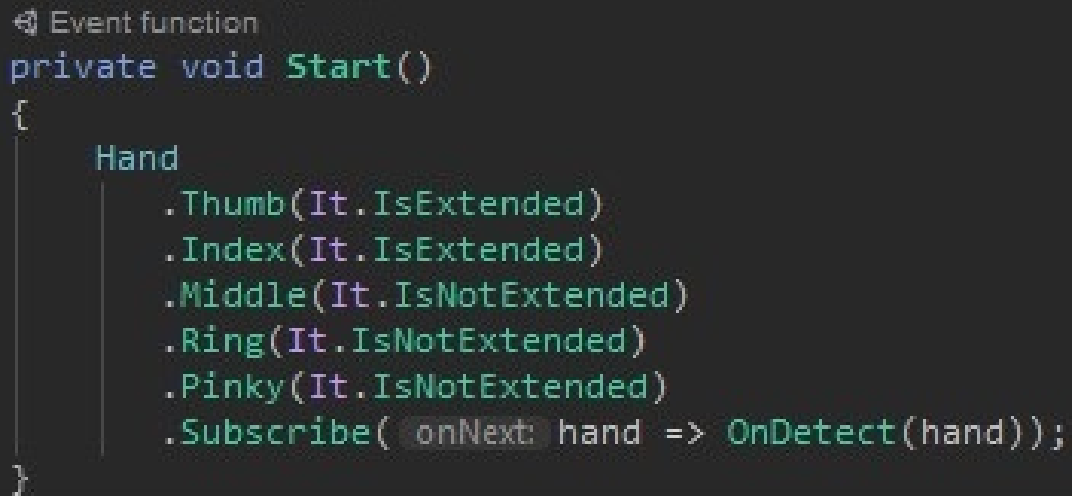
Of course, a mix of the two methods described in figures 7.3 and 7.4 can be used - say, for when you also want the thumb to point upwards.

The next step is to implement the *OnDetect* method. A possible implementation is shown in figure 7.5.

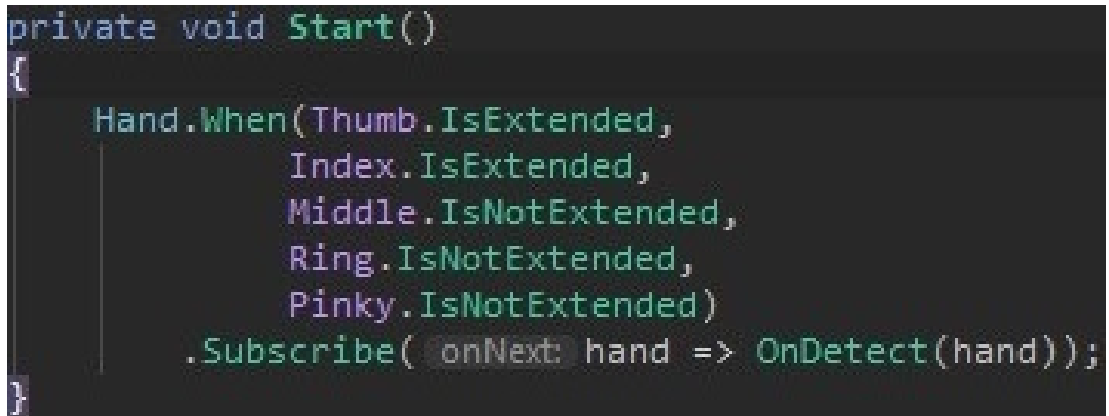
The *params object[] others* are variable arguments - you can pass any extra parameters to this function.

Last, add the script to one of the *ReactiveHands* in the scene - for example, the right hand - as shown in figure 7.6.

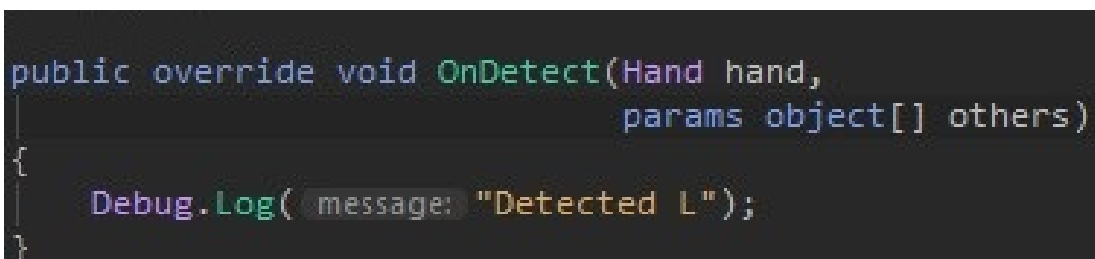
The *Interval* field represents the sampling rate expressed in seconds. The sampling rate means how often the same gesture should be detected. The field is of type **double**, so values less than 1 second can be set. The default value is 500ms (0.5 seconds).

A screenshot of the Unity Inspector window showing the Start function of the ReactiveHand script. The function is marked as an 'Event function' with a lightning bolt icon. The code is as follows:

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

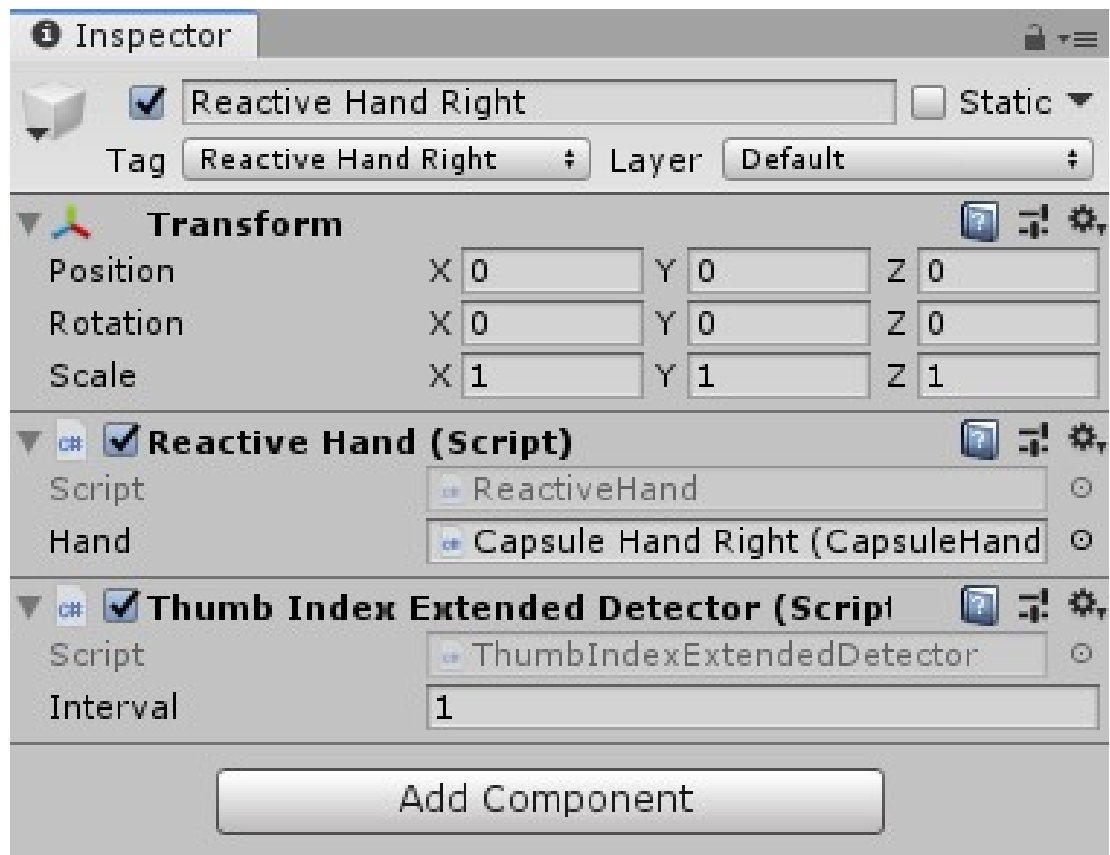
Figure 7.3: The *ReactiveHand* script start functionA screenshot of the Unity Inspector window showing the Start function of the ReactiveHand script. The function is marked as an 'Event function' with a lightning bolt icon. The code is as follows:

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

Figure 7.4: The *ReactiveHand* script start function, using the *When* operatorA screenshot of the Unity Inspector window showing the OnDetect function of the ReactiveHand script. The function is marked as an 'Event function' with a lightning bolt icon. The code is as follows:

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

Figure 7.5: *OnDetect* function

Figure 7.6: Script attached to the right *ReactiveHand*

Chapter 8

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 Leap Motion to invent new and more natural means of interacting with the VR 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, Fluent Motion could be the needed replacement. This document 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*).

Bibliography

- [1] 2019. [Online]. Available: <https://www.etymonline.com/search?q=virtual>
- [2] A. Artaud, *The Theatre and its Double*, 1938.
- [3] “Virtual and augmented reality: The next big computing platform?” *Goldman Sachs*.
- [4] K. D. et al., “Virtual reality for rehabilitation in parkinson’s disease,” *Cochrane Database of SYstematic Reviews*, vol. 2016, pp. 1–50, 2016.
- [5] K. Adam, *Gesture: Visible Action as Utterance*. Cambridge University Press, 2004.
- [6] R. M. K. et al., *Lexical Gestures and Lexical Access: A Process Model*. Cambridge University Press, 2001, pp. 261–283.
- [7] A. Colgan, “How does the leap motion controller work?” 2014. [Online]. Available: <http://blog.leapmotion.com/hardware-to-software-how-does-the-leap-motion-controller-work/>
- [8] L. M. Inc., “Leapmotion developer - using the tracking api,” 2019. [Online]. Available: https://developer-archive.leapmotion.com/documentation/python/devguide/Leap_Guides2.html
- [9] Untiy, “Monobehaviour script.” [Online]. Available: <https://docs.unity3d.com/ScriptReference/MonoBehaviour.html>

Appendix A

Relevant code

```
/** Maps are easy to use in Scala. */
object Maps {
  val colors = Map("red" -> 0xFF0000,
                   "turquoise" -> 0x00FFFF,
                   "black" -> 0x000000,
                   "orange" -> 0xFF8040,
                   "brown" -> 0x804000)

  def main(args: Array[String]) {
    for (name <- args) println(
      colors.get(name) match {
        case Some(code) =>
          name + " has code: " + code
        case None =>
          "Unknown color: " + name
      }
    )
  }
}
```


Appendix B

Other relevant information
(demonstrations, etc.)

Appendix C

Published papers