Universal Gesture Tracking Framework in OpenISS and ROS and its Applications

Jashanjot Singh

A Thesis

in

The Department

of

Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements

for the Degree of Master of Computer Science at

Concordia University

Montréal, Québec, Canada

January 2020

Concordia University

School of Graduate Studies

This is to certify that the thesis prepared

By: **Jashanjot Singh**

Entitled: **Universal Gesture Tracking Framework in OpenISS and ROS and its Applications**

and submitted in partial fulfillment of the requirements for the degree of

Master of Computer Science

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

_____ Chair
Dr. Aiman Hanna

_____ Examiner
Dr. Marta Kersten-Oertel

_____ Examiner
Dr. Sudhir P. Mudur

_____ Supervisor
Dr. Joey Paquet

_____ Supervisor
Dr. Serguei A. Mokhov

Approved by _____
Dr. Lata Narayanan, Chair of Department

February 6, 2020 _____
Dr. Amir Asif, Dean
Gina Cody School of Engineering and Computer Science

Abstract

Universal Gesture Tracking Framework in OpenISS and ROS and its Applications

Jashanjot Singh

In this research work, we present a common and extensible framework that abstracts different vision-based gesture recognition middleware and provides uniform gesture recognition data obtained from those via its simpler API to enable hand gesture interaction in different kinds of applications. We demonstrate various aspects of our framework via instrumentation and enable gesture interaction for our two specific yet different needs.

Firstly, we alleviate limited gesture tracking functionality in **ISSv2** aka. (Illimitable Space System v2), an interactive and configurable artists' toolbox that is used to create music visualizations, visual effects and interactive documentary film based on the inputs from users such as gestures, voice, motion, etc. *Secondly*, we provide a proof-of-concept solution to enhance and demonstrate limited language usability of the Forensic Lucid language's composition and compiler interactivity by enabling a forensic investigator to create partial Forensic Lucid encoded programs which require manipulation of preloaded digital evidence objects in a 3D warehouse-like application (**DigiEVISS**) via hand gesture interaction.

We also leverage Robot Operating System (ROS), an open source set of tools and libraries for its communication middleware to broadcast our framework data over the network. We provide this framework as a specialization of the **OpenISS** core framework and evaluate our framework on various aspects. We employ metrics such as effective frame rate and delay to evaluate our exemplified scenarios that represent our needs.

Acknowledgments

I would begin by expressing my sincere gratitude towards my supervisors Dr. Joey Paquet and Dr. Serguei Mokhov for supporting me to achieve this milestone in my academic career. This work would not have been feasible without their constant guidance, resources and encouragement. Next, I would like to thank my father, Raj Kumar and, mother, Jasbir Kaur, who always believed in me despite all odds and provided me with the resources that enabled me to reach this position. I am grateful for my sister Jasmine Kaur, who has supported me with great love and devotion.

Throughout my Masters, I had the privilege of befriending many people, to whom I am thankful for their endless support, and, I would like to mention their names explicitly. My dear friends Shaheen Manoucheher and Diego Pizarro Ribera who always supported me through difficult times. My friends from my part time work during my studies from whom I learned certain life skills, Alexander Gorbachev, Jayson Legault, Sohel Islam, Munna Miah and Felix Devost-Lamontagne for giving me free French language lessons. I am also thankful for the support of Haotao Lai and my peers Yiran Shen, Alexandre Simard and Jyotsana Gupta.

Finally, I would like to thank my beautiful girlfriend Marianne Venne, who is an incredible woman and the source of my happiness and inspiration. I greatly admire her personality and feel fortunate to have her as my partner. Our dog, Bali, whom we love unconditionally, and wakes us up every morning only to start our day with pure happiness. I yet again thank them all and I am grateful for everything that all of these amazing people have done for me.

Contents

List of Figures ix

List of Tables xii

1 Introduction 1

1.1 Research Domain . . . 1

1.2 Motivation . . . 4

1.2.1 Essential Background . . . 4

1.2.2 Motivational Scenarios . . . 6

1.2.3 Scenario I: Manipulation of digital evidence objects in a 3D warehouse-like application by a forensic investigator using hand gestures . . . 6

1.2.4 Scenario II: Real time visual effects for performance visualization on stage by an artist using hand gestures in ISSv2 . . . 8

1.2.5 Scenarios Summary . . . 9

1.2.6 Requirements . . . 12

1.3 Thesis Statement . . . 21

1.4 Goals and Objectives . . . 23

1.5 Research Questions . . . 24

1.6 Scope of the Thesis . . . 25

1.7 Contributions . . . 26

1.8 Structure of the Thesis . . . 27

2 Background **28**

2.1 Related Work . . . 29

2.1.1 Illimitable Space System . . . 29

2.1.2 GIPSY . . . 35

2.1.3 Forensic Lucid . . . 36

2.1.4 Toward Multimodal Interaction in Scalable Visual Digital Evidence Visualization Using Computer Vision Techniques and ISS . . . 39

2.1.5 OpenISS depth camera as a near-realtime broadcast service for performing arts and beyond . . . 42

2.1.6 Gesture Tracking . . . 42

2.1.7 Vision-Based Gesture Recognition . . . 43

2.1.8 Nielsen Usability Heuristics . . . 43

2.2 Libraries, Tools and Middleware . . . 44

2.2.1 OpenKinect . . . 44

2.2.2 PrimeSense . . . 45

2.2.3 Nuitrack . . . 48

2.2.4 Robot Operating System (ROS) . . . 50

2.2.5 Processing . . . 54

2.2.6 SWIG 4.0 . . . 55

2.2.7 Docker . . . 58

2.3 Summary . . . 58

3 Methodology **59**

3.1 Solution Overview . . . 59

3.2 Framework Design Approach . . . 62

3.2.1 OpenISS Core Framework . . . 64

3.2.2 OpenISS Gesture Framework . . . 66

3.3 Framework Design and Evaluation Methodology . . . 66

3.3.1 Top-Down and Bottom-Up Approaches . . . 67

3.3.2 Design and Implementation . . . 70
3.3.3 Evaluation . . . 73
3.4 Summary . . . 79

4 Framework Design and Instantiation **80**
4.1 Framework Design . . . 80
4.1.1 Data Acquisition . . . 81
4.1.2 Data Adaptation . . . 83
4.1.3 Data Delivery . . . 84
4.1.4 Design Structure and Layers . . . 84
4.2 Framework Instantiations . . . 94
4.2.1 NiTE2.0 . . . 95
4.2.2 Nuitrack . . . 98
4.2.3 ROS . . . 100
4.2.4 Framework Cross Language Module . . . 104
4.3 Framework Applications . . . 108
4.3.1 Integrated Sample Application . . . 108
4.3.2 **DigiEVISS** . . . 110
4.3.3 Scenario II Application . . . 118
4.4 Summary . . . 121

5 Evaluation and Results **122**
5.1 Evaluation Testbed Specifications . . . 122
5.2 Integrated Sample Application as Evaluation Testbed . . . 124
5.3 **DigiEVISS** . . . 130
5.4 ISSv2 . . . 132
5.5 Integrated Sample Application as ROS Client . . . 136
5.6 Test Game Application . . . 142
5.7 Summary . . . 146

6 Conclusion and Future Work **148**

6.1 Concluding Remarks 149

6.2 Limitations 151

6.3 Future Work 154

6.4 Summary 155

Bibliography **156**

Index **173**

Appendix **178**

A Docker File **178**

List of Figures

1 Vision-based hand gesture recognition [1] 2
2 Motivational scenarios (primary actors) 9
3 Motivation scenarios (secondary actors) 11
4 Coarse abstract overview of this work 23
5 Block diagram of illimitable space system (ISS) [2] 31
6 Conceptual pipeline of illimitable space system (ISS) [2] 32
7 ISSv2 various visual effects performances 34
8 Evidential statement [3] 37
9 Conceptual and actual nested visualization representation based on ISSv1 40
10 High level architecture of the proposed solution in [4] 41
11 OpenNI2.0 and NiTE2.0 stack 46
12 OpenNI and NiTE device and driver specific stack 48
13 Nuitrack architecture [5] 49
14 Nuitrack modules [5] 49
15 Nuitrack device and driver specific stack 50
16 ROS conceptual architecture levels 51
17 A typical ROS workspace and package structure [6] 52
18 Computation graph level 53
19 A Processing sketch 55
20 SWIG architecture 56
21 SWIG pipeline 56
22 SWIG Java demo 57
23 Solution overview 61

24 Framework development process [7] . . . 63
25 Gesture framework overview [7] . . . 64
26 OpenISS core and specialized gesture framework . . . 67
27 Domain model . . . 68
28 Data flow from device to application . . . 81
29 Adapter object adaptation via Composition . . . 83
30 Layered architecture . . . 86
31 `OIGestureTracker` UML class diagram . . . 88
32 OpenISS gesture framework kernel in UML . . . 88
33 OpenISS gesture framework enumerations . . . 91
34 OpenISS gesture framework class Hierarchy . . . 94
35 NiTE2.0 instantiation in UML for OpenISS gesture provider hot spot adapter . . . 96
36 Nuitrack instantiation in UML for OpenISS gesture provider hot spot adapter . . . 101
37 ROS adaptation on publisher/subscriber ends . . . 102
38 ROS OpenISS node and topics graph . . . 103
39 Publisher: **OpenISS** ROS NiTE2.0 and Nuitrack instantiations . . . 104
40 Subscriber: applications instantiating NiTE2.0 or Nuitrack . . . 105
41 Integrated sample application UML diagram . . . 108
42 Integrated sample application viewer . . . 109
43 3D application [8] (arranging planets with hands and gestures) . . . 111
44 **DigiEVISS** structure in UML . . . 112
45 Scenario I: **DigiEVISS** application . . . 113
46 **DigiEVISS** application first and second iterations . . . 114
47 **DigiEVISS** application (tracked hand pointers) . . . 114
48 **DigiEVISS** application (grabbing a sphere) . . . 115
49 **DigiEVISS** application (semantic link) . . . 115
50 ISSv2 backends `SimpleOpenNI` and **OpenISS** . . . 119
51 ISSv2 application scenario II visual effect 1 and 2 . . . 119

52 Integrated sample application NiTE2.0 backend gesture recognition . 125
53 Integrated sample application NiTE2.0 backend hand tracking 126
54 Integrated sample application Nuitrack gesture adapter (gestures) . . 127
55 Integrated sample application Nuitrack gesture adapter (hands) . . . 128
56 **DigiEVISS** 132
57 Integrated sample application Processing (NiTE2.0) 133
58 Integrated sample application Processing (Nuitrack) 134
59 ISSv2 visual effects via gestures framework 135
60 Master node and **OpenISS** ROS node publishing data on various topics 137
61 Integrated sample application ROS (Nuitrack) gesture adapter 138
62 Integrated sample application ROS (NiTE2.0) gesture adapter 138
63 ROS server/publisher 141
64 ROS client/subscriber 142
65 Diff between existing vs updated controls 144
66 Gesture swipe left 145
67 Gesture swipe up 146
68 Gesture swipe down 146
69 Gesture swipe right 147

List of Tables

1 Scenarios vs. functional requirements 20
2 Scenarios vs. non-functional requirements 20
3 Requirements evaluation classification 77
4 OpenISS gesture framework API overview 84
5 OpenISS gesture framework concrete implementation structure 87
6 ROS OpenISS package 87
7 Environment hardware specifications 123
8 Middleware, libraries, and tools used 123
9 Depth sensors specifications 124
10 Integrated sample application (average rendering FPS vs. sensor FPS) 130
11 ISSv2 (average rendering FPS vs. sensor FPS) 136
12 Callbacks-per-second when message queue size is 0 140
13 Callbacks-per-second when message queue size is 1 140
14 Callbacks-per-second when message queue size is 100 140
15 Callbacks-per-second when message queue size is 1000 140
16 ROS overhead 141
17 Requirements evaluation summary 147

Chapter 1

Introduction

In this chapter we start by providing the reader with a brief overview of the research domain in Section 1.1. Subsequently, we describe the motivational scenarios that exemplify our needs in Section 1.2, accompanied by the essential preliminary background required in Section 1.2.1. Afterwards, we elicit specific functional and non-functional requirements from these motivational scenarios in Section 1.2.6. Then, we describe the research problem at hand in Section 1.3. Next, in Section 1.4 we present the high level objectives of this research. Then, we define our research questions in Section 1.5 that this work will answer, followed by the scope of this thesis in Section 1.6. Then, we enumerate the contributions made towards the completion of this research in Section 1.7. Finally, we outline the structure of the rest of the dissertation in Section 1.8.

1.1 Research Domain

Human-Computer Interaction (*HCI*) is a multidisciplinary field of study that focuses on various aspects of such interaction [9] and Software Engineering is one such field whose concepts play a vital role in aiding to realize such interaction [10, 11]. Traditionally, a keyboard or a mouse are well-known interaction devices for enabling humans to provide input. However, these devices have limitations in terms of the number of degrees of freedom they offer, especially in the context of interacting with

3D applications. For example, a mouse that merely provides two degrees of freedom is clearly not a good fit for 3D applications for proper emulation of three dimensions of space [12–14]. Interestingly, when it comes to humans interacting amongst themselves, we make extensive use of non-verbal forms of communication such as hand gestures along with verbal communication to express ourselves. Thus, it is quite apparent to desire interacting with the computers using natural communication modes such as hand gestures. At this point, an intersection of the *Computer Vision* and the *Human-Computer Interaction* domains enable the computers to understand human hand gestures, via a key approach known as *Vision-Based Hand Gesture Recognition and Hand Tracking* and fits better under the subdomain of HCI that is *Natural User Interfaces*. This approach requires image acquisition through a camera that can sense depth, such as the Microsoft Kinect, followed by pre-processing, segmentation, feature extraction and subsequently, classification of the object of interest that is the human hand as seen in Figure 1.

Figure 1: Vision-based hand gesture recognition [1]

However, irrespective of the source of the depth data, the key idea is to obtain the depth information along with its corresponding color image to enable specific applications in computer vision and image processing, including hand gesture recognition, hand tracking and even object recognition among others. In this context, depth, broadly scoped, is the distance from the camera to the surface of an object of interest where each pixel represents the distance from the camera to a seen object recorded within the depth image [15]. Moreover, authors have shown in [16] that using simply the depth images, it is possible to classify hand gestures much faster and with more precision than non-depth-based approaches, which is a crucial detail for the reader to remember. Thus, we can say that the depth information plays a key role in enabling hand gesture recognition and hand tracking for natural interaction in 3D computer applications.

Vision-Based Hand Gesture applications can be broadly divided into four main categories, namely [17]:

- *Assistive Technologies and Medical Systems* – gestures can be used to control the distribution of resources in hospitals, interact with medical instrumentation, control visualization displays, and even help handicapped users as part of their rehabilitation therapy. One such example is *Gestix*, a hand tracking device that allows surgeons to browse MRI images in an operating room [18, 19].

- *Entertainment* – in the video gaming industry, hand gestures enable natural interaction to enhance user experience. Additionally, various gestures help emulate real-life sports in video gaming. The *Kinect Sensor for Microsoft's Xbox 360* is a relevant example of gesture interaction application in the gaming industry which also triggered a lot of academic research about other related applications as well [20].

- *Crisis Management and Disaster Relief* – crisis management systems involve experts dealing with large swaths of data such as geospatial data, which, when visualized, require intuitive interaction modes to aid analysis and inferences. *DAVE_G*, is one such multimodal and multi-user geographical information system that supports analysis on geospatial data using gestures [21].

- *Human-Robot Interaction* – a very trivial example application requiring hand gestures for human-robot interaction would be simple instructions that resolve to navigation instructions such as pointing to a direction for mobile robots. Another concrete example is the work of Yin *et al.* [22] that uses gestures to control a hybrid service robot system called *HARO-1*.

Our work falls under the *entertainment* and *assistive technologies* for its specific use-case applications. In this research work we employ Software Engineering methodologies and concepts as tools and techniques to enable hand gesture interaction for different kinds of applications [10, 11]. Thus, the domain in this research work is an intersection of HCI Engineering and Software Engineering, that is employing

software engineering methodology in the HCI domain and its subdomain of Natural User Interfaces (NUI).

1.2 Motivation

1.2.1 Essential Background

Before we delve into the specifics of this research work, we will begin by introducing the reader with **two** individual bodies of work that will be referred to multiple times from this point onward throughout and are essential for the reader to be aware of before progressing further through this dissertation. Although, they are elaborated in detail in Chapter 2, the goal is to provide the reader with a necessary preliminary overview of the context.

Firstly, the Illimitable Space System (ISS), is a real-time interactive configurable toolbox especially for artists, used to create visual effects, musical visualizations and, interactive 3D documentary film based on multimodal user inputs such as voice or gestures along with the corresponding image mapping [23], that was first proposed by Dr. Miao Song in her doctoral dissertation [24]. Over several iterations, the **ISS-as-an-application** has evolved from a design perspective in the past few years, and is currently progressing towards having the **OpenISS** [25] (Chapter 2, Section 2.1.1.4) as its open source backend core to prospectively support its various functional and non-functional requirements [2], for instance, **FR2: Interact using Motion and Gesture** [2], one of the significant functional requirement for the **ISSv2** (the second iteration of the ISS) that we will address in this work from a gesture perspective only, and, address a few limitations of the **ISSv1** and **ISSv2** application instances mentioned later (Chapter 2, Section 2.1.1.1 and Section 2.1.1.2). Now, there were two primary motivations behind creating **OpenISS** as an open source core for any version of the **ISS**:

- Open Source code allows the open source community to participate in the development process crowdsourcing the contributions and wider adoption.

- A core allows parts of the system to be switched out at will without having to modify it all at once keeping the applications functioning.

The first iteration of the Illimitable Space System or as we call it, is the **ISSv1** (which is abandoned and no longer maintained presently), was built using **XNA** and C#, whereas the current version, the **ISSv2** is built using the **Processing** environment [26]. Now, the **ISSv1** and **ISSv2** were limited to the Microsoft Kinect sensor for depth and image data and, the **ISSv3** is specifically for AR/VR applications in Unity3D. However, further elaboration on the Illimitable Space System and its versions namely, **ISSv1**, **ISSv2**, and **ISSv3**, can be found in detail in Chapter 2, Section 2.1.1.1, Section 2.1.1.2, and, Section 2.1.1.3 respectively.

Secondly, is FORENSIC LUCID, an intensional programming/specification dialect of the LUCID declarative programming language [3, 27]. FORENSIC LUCID enables *cybercrime* investigators to reason about computer forensic cases by expressing in a program form the encoding of evidence, witness stories, and evidential statements, that can be tested against claims to see if there is a possible sequence or multiple sequences of events that explain a given *story*. Presently, a *forensic investigator* would require to write such a FORENSIC LUCID encoded program manually. A key interesting takeaway for the reader is that every FORENSIC LUCID program is a data-flow program that can be visually composed and illustrated as a **2D data-flow graph** [27] where the data flowing through the graph can be *multidimensional*. In our related work [4] we explore an idea of a scalable management, visualization, and evaluation of digital evidence in the context of cybercrime investigation with extensions to the interactive 3D documentary subsystem of the Illimitable Space System. We further elaborate on this related work in the next chapter in Section 2.1.4. However, this work aims towards the realization of the aforementioned idea but only from a gesture perspective.

Although, we further expand on FORENSIC LUCID in Section 2.1.3 in, but, at the same time we would like to make the reader aware of the fact that we will only expose FORENSIC LUCID to the extent that is strictly within the context and scope of this thesis.

1.2.2 Motivational Scenarios

We came up with the following primary scenarios that are based on actual and potential real-life use-cases and help us to elicit various functional and non-functional requirements for our solution to enable a common platform that can cater to these corresponding applications and enable real-time hand gesture interaction for specific yet diverse 3D applications.

Now, partially based on our proposed overall solution in our related work [4] (see Figure 10) that poses a much larger problem, which is further discussed in detail in Section 2.1.4, we describe our first scenario. The primary actor, a forensic investigator (hereafter referred to as F) wants to use FORENSIC LUCID to encode digital evidence in a FORENSIC LUCID program's context, e.g., Listing 2.1, creating observation sequences via semantic linkage between various digital evidences objects. Let's assume these objects are represented as 3D spherical models that can be arranged in a 3D application space using hand gestures. The actor, F will arrange these digital evidence objects in a desired observation sequence and eventually, an evidential statement to structure a 3D visual representation of a partial data-flow graph of digital evidence objects by natural interaction via hand gestures rather than manually encoding the digital evidence into a FORENSIC LUCID program using a keyboard. The digital evidence objects are assumed to have been preloaded into the serious game environment from external sources.

1.2.3 Scenario I: Manipulation of digital evidence objects in a 3D warehouse-like application by a forensic investigator using hand gestures

In this scenario, the forensic investigator F needs to interact with the preloaded digital evidence objects in an immersive 3D warehouse-like game application named **DigiEVISS** using hand gestures and manipulate them to create a 3D visual representation of the 2D data-flow graph illustrating a FORENSIC LUCID encoded program, in the following steps:

1. Select up to two digital evidence objects (known as "observations" in FORENSIC LUCID parlance), one at a time, via a gesture, and hold each one with each hand so that they can be naturally moved around with the hands that are holding these objects to be arranged in a desired sequence within the virtual 3D space.

2. Then create a *semantic link* (a time successor relationship) between these selected objects by bringing them closer and placing them in a desired order within **DigiEVISS**. This is required because FORENSIC LUCID observation sequence evidential context is a chronological ordered collection of observations.

3. Once the investigator is done creating an observation sequence with those semantically linked digital evidence objects, generate a FORENSIC LUCID encoded program including the evidential statement and compile it with the corresponding compiler within the General Intensional Program Compiler (GIPC) framework.

4. Repeat the procedure for as many observation sequences as necessary to compile the entire evidential statement.

Given that every FORENSIC LUCID program can be visually illustrated as a 2D data-flow graph, the idea is to leverage this attribute and extend it to a 3D visual representation of these data-flow graphs with digital evidence context objects, which further are a visual representation of the FORENSIC LUCID encoded context as if it were manually written. Therefore, the investigator is creating a 3D visual representation of a partial data-flow graph with these manipulations to digital evidence objects in **DigiEVISS**. Note that while an entire FORENSIC LUCID program can be represented as a data-flow graph, this work lays groundwork with the evidential contexts of *observation sequences* only and is a crucial detail to remember.

Next, we exemplify our another need in a scenario to alleviate the limitations of the **ISSv2**, and enable the gesture portion of the specific functional requirements, such as, **FR2: Interact using Motion and Gesture** in [2] (mentioned above in Section 1.2.1) for *performance visualization*, a mode in which once the performer

walks into the depth sensor viewing volume range, it captures gestures and postures of the performer. The Illimitable Space System then processes the motion and gesture data according to the depth data, gesture dictionary and/or posture dictionary and generates real-time visual feedback (where gesture and posture dictionaries are a set of pre-recognized gestures and postures with their corresponding feedback actions). In this scenario, the primary actor, an artist (hereafter referred to as A) interacts with the **ISSv2** for performance visualization in real-time using hand gestures and uses the existing viusal effects that are currently powered with the existing `SimpleOpenNI` backend, which is relatively old and is limited only to the Microsoft Kinect hardware, and exhibits quite a small gesture dictionary consisting of only *three* gestures out of which only two can be used effectively. Now A, wants to use a camera other than the Microsoft Kinect v1 or Kinect v2, simply, because these models have been discontinued by Microsoft itself and relatively newer devices have emrged that provide better accuracy, resolution, and response. As mentioned before, A has some existing visual effects that were created for the **ISSv2** earlier that should ideally work with the desired newer devices and their drivers without much modification to **ISSv2** itself. Additionally, A will have access to newer devices and gestures other than the Microsoft Kinect and `SimpleOpenNI` respectively, A will continuously need to create new visual effects that could take advantage of the newer devices' capabilities and wider gesture dictionaries to the full extent.

1.2.4 Scenario II: Real time visual effects for performance visualization on stage by an artist using hand gestures in ISSv2

In this scenario, the artist A requires real-time hand gesture recognition and hand tracking via a flexible solution that will enable A to use newer devices (for better quality and runtime performance) and assist in not only creating new visual effects, but also accommodate the existing ones with newer devices as well, and perform the following tasks:

1. First, select **OpenISS** as a backend instance "driver" to instantiate a desired middleware/library that provides real-time hand gesture recognition, hand tracking and necessary depth information functionalities via our solution, and, this is a configurable selection.

2. Then, execute and interact with the existing **ISSv2** performing arts applications for performance visualization using hand gesture interaction provided via our solution within the **OpenISS** backend as per usual with other backends such as the `SimpleOpenNI` backend mentioned before with limited gesture interaction.

3. Or, create new **ISSv2** performing arts application profiles for performance visualizations using completely new hand gestures previously not possible via our solution that must provide an easy to use Application Programming Interface (API) for the computation artists to do so.

1.2.5 Scenarios Summary

These scenarios, combined for convenience, are illustrated visually in Figure 2 since the work itself is about providing a common solution to both these scenarios. Thus,

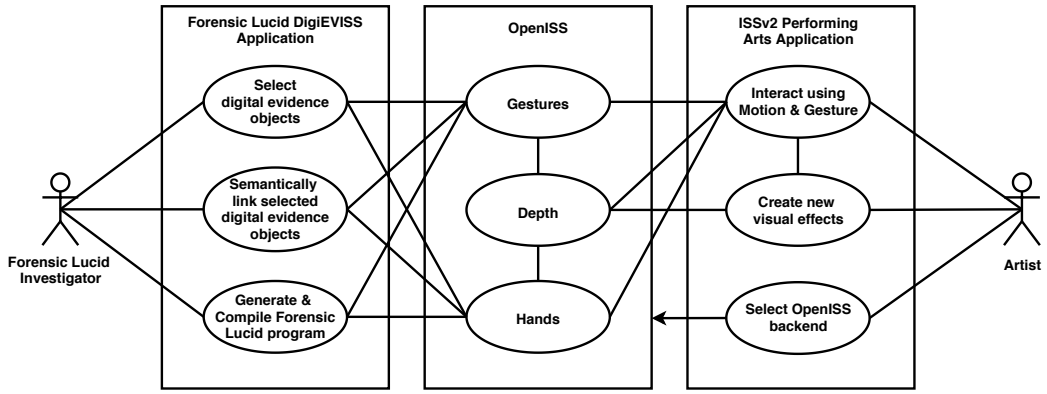


Figure 2: Motivational scenarios (primary actors)

so far, we described our scenarios from the perspective of our primary actors. Moving on, we further discuss a situation where either of our primary actors, the forensic investigator F or the artist A, may wish to extend their respective 3D applications for the following reasons:

1. The applications are using an *algorithm* X for real-time hand gesture recognition and hand tracking provided by some library/middleware. Where, X provides a total of *three* gestures in its gesture dictionary that it is capable of recognizing and the applications will have assigned those for desired actions.

2. Now, there is a newer *algorithm* Y for real-time hand gesture recognition and hand tracking, that is available by a newer library/middleware. However, Y provides a total of *five* gestures in its gesture dictionary.

3. Therefore, either of our actors wish to leverage this newer provider of gesture recognition capabilities, Y, with more gestures for enhancing their applications by adding more actions via these extra gestures.

4. Now, it is also possible that our actors might want a completely new application that uses Y for hand gesture recognition and hand tracking for reasons such as usability or simply the need to rewrite them based on their experience with the existing one.

Thus, our secondary actor, a developer (hereafter referred to as D), can help extend our solution to accommodate prospective solutions and/or create newer applications for our primary actors thus enabling comparisons among different middleware in order to make informed design decisions. As a result, our solution itself must enable a developer to extend it with a minimum effort as well as in a concise manner in order to maintain the program flow in these applications as unchanged and ensure that there is uniformity among the applications in terms of the API. This is a challenge that our work must address in order to enable easy extensibility and a ease-of-use of its API in an SDK-like manner.

Now that we have discussed the need of prospective solutions that our primary actors wish to leverage by asking a secondary actor to extend our solution and/or create new applications using it. At the same time, it will enable the primary actor to ask for a comparison among such prospective solutions so as to choose the most optimal one specific to their application needs.

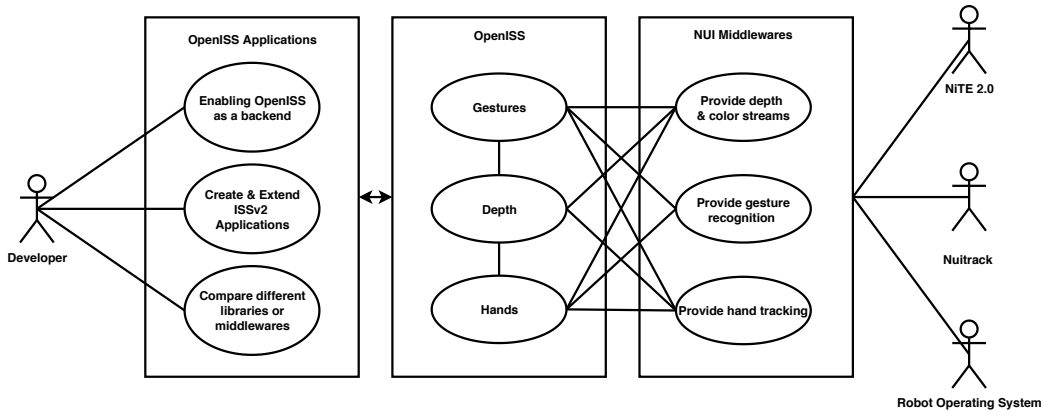


Figure 3: Motivation scenarios (secondary actors)

In short, our solution must enable comparisons among these different gesture providers, so that our secondary actor, the developer, can help the primary actor to analyze and make **recommendations** based on comparisons of various metrics/requirements (frame rate, resolutions, dictionary sizes, precision, etc.), and support the secondary actor to make informed design decisions when extending our solution or choosing the optimal gesture middleware/library for a specific application. The above situation can be described as follows:

1. Primary actor (A or F) finds two newer libraries/middleware P and Q, that provide gesture dictionaries of, e.g., 4 and 6 gestures.

2. Primary actor asks the secondary actor the developer (D) to demonstrate the comparison between P and Q.

3. Based on application specific metrics, the primary actor decides to go with either P or Q depending upon the comparison results.

Thus, it is yet another challenge to enable such comparisons via our solution to support the users to make informed decisions for specific use cases within the context of this research work.

1.2.6 Requirements

Now, the motivational scenarios mentioned in Section 1.2.2 enable us to elicit specific functional and non-functional requirements that we have listed further below in this section. These requirements will necessarily describe the solution itself, its functionality and the capacity in which it must provide aforementioned functionality. The majority of these requirements are derived from the related work referenced in [2, 4] and the corresponding applications.

1.2.6.1 Functional Requirements

Definition. *Functional Requirements are the ones that describe the behavior of the software system; thus, a set of functionalities that a software system can provide.*

The motivational scenarios described earlier have one thing in common, they both require real-time hand gesture recognition and hand tracking for their specific use-cases described as scenarios (thus, forming the basis for the requirements).

1. **FR1: The system shall provide real-time hand gesture recognition.**

   Our very first functional requirement is to provide real-time hand gesture recognition functionality. In our first scenario, the hand gestures represent various actions that enable the investigator to interact with the digital evidence objects in a 3D application and perform manipulations, whereas in our second scenario, an artist interacts with the application for real-time visual effects using hand gestures for a live on stage performance. For instance, a specific gesture may trigger a pick or grab action to enable the investigator to manipulate a visual digital evidence object representation a 3D immersive application. Similarly, for the artist, certain gestures can trigger visual effects that enhance on stage performances.

2. **FR2: The system shall provide real-time hand tracking.**

   As gestures merely trigger an action, to continue the action we need hand tracking data in terms of hand positions in 3D coordinates for their use in 3D

applications. For example, the investigator interacts with any two visual digital evidence objects with their hands and, then, in order to create a semantic link among them, they must bring the objects closer to one another. Likewise, an artist might need hand tracking to showcase a visual effect in which a trail of stars follow the motion of the hand.

3. **FR3: The system shall provide real-time depth information of user's hands.**

   As mentioned before in Section 1.1, depth data is adequate for detection and recognition of hand gestures and is relatively faster than non-depth approaches for gesture detection and recognition. Therefore, a real-time stream of depth frames containing essential depth data as captured by the depth sensing device must be made available so that relevant information can be extracted from these frames by middleware like NiTE2.0 or Nuitrack. Normally, depth frames would be provided by their respective device drivers provided by the device manufacturers or SDK such as **Microsoft Kinect for Windows SDK** [28]. However, some open source drivers exist as well for commonly used depth sensors such as the Microsoft Kinect, explained later in the next chapter in Section 2.2.

4. **FR4: The system shall serve as an easy-to-switch-to backend with the existing ISSv2 pipeline.**

   As mentioned previously our solution must be sustainable with the current **ISSv2**, which essentially means that although **ISSv2** currently has limited hand gesture recognition specific to the Microsoft Kinect hardware, it still has diverse performing arts applications, which, ideally, should behave the same way with minimal modifications if necessary, once our solution addresses such limitations by replacing existing gesture providing backend with its own. For instance, if a newer device such as the Intel RealSense D435 is being used instead of the Microsoft Kinect, the behavior of the application must at a minimum remain unchanged.

In other words, we must have a solution to serve as a backend instance that serves as a gesture data provider for gesture-recognition algorithms, tracking algorithms, multiple device encapsulation and sensor related data drivers without affecting the front-end application that is the application side of the Illimitable Space System that puts the show on the stage with visualizations and effects for performing arts.

All these requirements represent a subset of the objectives of this research work along with the non-functional requirements that are discussed in the next section.

1.2.6.2 Non-Functional Requirements

Definition. *Non-functional requirements are the ones that describe the quality standards expected to be met by the software, thus, how adequately the software can provide the functionalities desired to the user.*

1. **NFR1: Real-time Response**

   Definition. *The system should be able to process the frames at a speed that is at least 10 frames-per-second (FPS).*

   Our very first non-functional requirement, inarguably, is *Real-time Response* with a known, bearable latency. However, to precisely define this bearable latency, the minimum threshold must be at least **10 FPS** (which was an acceptable output frame rate for some **ISSv2** visual effects in shows), as the human vision can minimally perceive it as motion. However, it is also plausible that a sensitive response may make the application too prone to errors and often missing targets. Now, broadly, there are four major stages before output frames can be seen visualized via a desired mode for which we require real-time response in terms of frame rate (also, see Figure 6), namely:

   - Motion or Gesture Capture
   - Motion or Gesture Interpretation

- Visual Effects Generation
- Visual Effects Rendering

This essentially means that irrespective of the input frame rate, the output frame rate after all these stages must be minimally **10 FPS** for a minimal motion perception. However we could say, roughly, our scenario in Section 1.2.3 strictly requires real-time response from the proposed system whereas our scenario in Section 1.2.4 can accommodate the above mentioned latency. This is simply because of the nature of both applications are relatively different but it is still desirable to have real-time response for both these scenarios. Where, non-real-time scenarios include processing of pre-recorded video footage to extract and record gesture frames and hand position for possible later use in other applications as well, but this side of the project is not the focus of this thesis presently.

2. **NFR2: API Usability**

   Definition. *The ability of an Application Programming Interface provided by a system to be easy to use and learn by developers that wish to implement it [29, 30].*

   Now, one of the design goals for **ISSv2** is to enable real artists to be able to develop performance visualization applications with a minimal technical background. Additionally, our motivational scenarios clearly depict the need for an extensible solution, so it is strongly desirable to make that process as easy as possible. Therefore, keeping in mind this constraint our solution must provide a simpler abstraction as compared to existing solutions such as NiTE2.0 and Nuitrack. All our actors, whether primary or secondary come from completely different domains, but we can broadly classify them as programmers and non-programmers. As a result, both these factors significantly affect our solution's API. Moreover, designing APIs with their users in mind can result in fewer errors, along with greater efficiency, effectiveness, and security [29].

An *Application Programming Interface* (API) is a logical interface to a software component that hides the internal details required to implement it. The notion of a good abstraction is relatively subjective and there can be multiple ways of abstracting a logical component [31]. This is yet another challenge that we must address in this work. However, we do emphasize that although this usability is relatively different from end-user interface usability, which requires extensive user testing, it can still be evaluated with Nielsen's usability heuristics [29, 32]. Therefore, by following a human-centric design approach API designers can make APIs easier to use by developers [29]. Thus, it is more of a usability criteria for developers who wish to use our solution as an SDK (software development kit) and extend it for their own specific use-cases.

In this research work, we design our API on preexisting applications and APIs provided by middleware such as **ISSv2** backends and middleware such as NiTE2.0. Moreover, the latter is well received by the community and relatively small and simple. So, we will demonstrate a simpler abstraction of a typical gesture recognition API via our solution.

3. **NFR3: Extensibility**

   Definition. *The ability of a system to accommodate prospective solutions easily.*

   We have already mentioned before that there will always be room for better solutions as technology or academic research progresses and recent integration of depth cameras into mobile devices will certainly push efforts into improving upon existing solutions even further. This means, that there can be prospective technology solutions, which will outperform current solutions or even provide wider range of gestures (gesture dictionary) that can be leveraged in both our use cases and beyond. Normally, a larger gesture dictionary may improve interaction possibilities, but it still needs to be constrained for specific application types and intuitive enough for the users to be able to recall and use the gestures easily that matches their mental models. Henceforth, the solution

requirement to be extensible and accommodate such potential additions from the emerging technology and middleware APIs easily.

4. **NFR4: Interoperability**

   Definition. *The ability of a system to inter-operate with other software systems and share meaningful data.*

   In Section 1.3, we mention that we will leverage the existing solutions available in the form of middleware, libraries or even meta-operating systems that provide hand gesture recognition capabilities or prerequisite computer vision pipelines. One such meta-operating system is the *Robot Operating System*, abbreviated as ROS [33]. In our recent work titled *OpenISS depth camera as a near-real-time broadcast service for performing arts and beyond* [34] we exposed the Microsoft Kinect depth camera streams as REST[1] and SOAP[2] web services. There, one of our original future work items mentioned ROS as one of the potential candidates for exposing depth cameras as a service that various clients can subscribe to because ROS has a diverse array of community-contributed packages that can interface with various motion capture systems including state-of the art technology providers like *Vicon Motion Capture Systems* [35].

   ROS is originally a robotics computer vision and sensor middleware, it is a complex yet mature system that provides ready to use tools and stacks for writing intercommunicating robotics software. It has a peer-to-peer architecture to pass messages between different processes called nodes that are typically distributed across two or more machines. We talk about its architecture in detail in Section 2.2.4, Chapter 2. It will suffice for now to say, the ROS community provides a few ROS packages, which can interface with different depth devices, but none of them provides real-time hand gesture and hand recognition functionality. However, the point is that since as ROS expands our reach over to more devices, we certainly want to leverage its platform.

[1] Representation State Transfer
[2] Simple Object Access Protocol

More importantly, our solution requires depth information and ROS can provide it to us from the same depth cameras we use; however, it is another problem to demonstrate such interoperability with an external system such as ROS by acting as a client to it. It is crucial to our solution as we want to be able to use different kinds of depth data providers based on specific needs that may arise in the larger scale deployments of **ISSv2** and if ROS can be such a source, we need it to be a part of our solution. However, we do expect some performance overhead using ROS due to (de)marshalling and network delays, on which we elaborate in Chapter 3 and later measure it in Chapter 5.

5. **NFR5: Structural Scalability**

   Definition. *The ability of a system to expand in a chosen dimension without major modifications to its architecture.*

   With respect to both our scenarios we can easily imagine a situation where a need may arise to decouple the sensor device and the system and the actors. For example, if the depth sensing cameras can be mounted on drones overlooking artists that are performing on stage and using the **ISSv2** applications then the data must be available over a network. Similarly, an expert FORENSIC LUCID forensic investigator may be called upon remotely, to look over sensitive data that must not leave its premises so as to bring the application to the digital evidence storage rather than the other way around. Thus, we want our solution to enable the investigator to still send gesture-interaction data over a network, although we do not perform any setup to test such a scenario including drones or a forensic investigator rather we will focus on first enabling it to test its feasibility.

   Therefore, by using ROS the sensor device can be physically distant from the system, in othere words, by leveraging ROS's network stack we can scale the distance between the source and destination of the motion capture aka mocap

data. At the same time, a system has in general a physical limit of the depth sensing devices it can connect to, whereas using the mature networking stack of ROS itself we can potentially scale the system in terms of an array of devices that can be connected over the ROS network.

Moreover, we merely consider this as a positive side effect of our previous requirement, that is, interoperability with ROS (see NFR4). Although, we do not evaluate these scenarios of multiple devices over ROS rather one in a simple two machines client-server or publisher-subscriber fashion but we feel confident enough to mention them for the reader to be aware of the functionalities that can be leveraged in the near future in case the needs arise to do so. Henceforth, our structural requirement to decouple the sensor and the end user application and allow it to select a desired/available device without affecting the application.

6. **NFR6: Device Adaptability**

   Definition. *The ability of a system to switch between devices only minimally affecting the end user application.*

   We also mentioned earlier that we need our solution to support as many depth sensing devices as possible, so we need our solution to enable such device adaptability via a minimal effort from a design point of view [36] with of course affecting the end-user application minimally by exhibiting a plug-and-play feel.

7. **NFR7: Cross-Platform**

   Definition. *The ability of a system to function on more than one platform.*

   In Section 1.2.2, although both scenarios are specific applications, but one cannot rule out the possibility of the end users with different platforms. Therefore, we need our solution to minimally support the *Linux* distributions and Unix based MacOS operating systems, where majority of our target applications run. Some applications, namely games, and others, also require Windows support. Thus, our proposed solution must be cross-platform.

8. **NFR8: Cross-Language**

   Definition. *The ability of a system to share data from its implementation language to a different programming language application.*

   For real-time constraints and existing middleware platforms, we will use the C++ programming language as our implementation language by default. Whereas, as mentioned earlier, **ISSv2** is primarily JAVA-based and so are its existing visual effects. Thus, we need our solution to bridge the gap between with a wrapper and demonstrate data sharing from C++ to JAVA and at the same time making sure that the real-time aspect remains minimally affected.

We can summarize the above functional and non-functional requirements along with the scenarios in Section 1.2.3 and Section 1.2.4 from which they were elicited in a tabular form as seen in Table 1 and Table 2 respectively.

Table 1: Scenarios vs. functional requirements

Scenarios	**Functional Requirements**
I	FR1, FR2
II	FR1, FR2, FR3, FR4

Table 2: Scenarios vs. non-functional requirements

Scenarios	**Non-Functional Requirements**
I	NFR2, NFR3, NFR5, NFR6, NFR7
II	NFR1,NFR2,NFR3, NFR4, NFR5, NFR6, NFR7, NFR8

1.3 Thesis Statement

We research and develop a common, flexible and extensible solution that exhibits *real-time hand gesture recognition and hand tracking* data provider functionality, housed within the **OpenISS** core platform, to enable specific, yet, diverse applications from a natural user interaction perspective for the Illimitable Space System itself. These applications were illustrated before in the form of *scenarios* in Section 1.2.2. Now, the reader must wonder that we just mentioned in the above section that the Illimitable Space System was already used to create real-time visual effects and musical visualizations based on inputs such as voice or gestures on stage.

So, why do we still want to provide our solution for real-time hand gesture interaction and hand tracking?

When the Illimitable Space System evolved from **ISSv1** to **ISSv2**, from a design perspective (although it indeed was an improvement over its previous versions) by changing technologies it sacrificed or degraded some of its functionalities such as *Hand Gesture Recognition* and *Green Screen* to name a few [2]. At the same time, as the *Vision-Based Hand Gesture Recognition and Hand Tracking* domain has evolved due to its numerous applications in various fields, newer depth sensing devices and different open source libraries have started appearing, which raised the need for a flexible and extensible solution that can accommodate all these prospective additions for the **ISSv2** itself depending on the setup and sustainable evolution of the system and its versions.

Currently, the **ISSv2** provides very limited hand gesture recognition, only specific to the Microsoft Kinect devices and their drivers, but, there are other recent depth sensing devices available in the market that we certainly need to leverage, such as the Intel RealSense D435 camera, that provides better accuracy and is still in production unlike the Microsoft Kinect v1 and Kinect v2. However, it is crucial and yet another challenge for these adoptions to be sustainable, that is, keeping the compatibility with current devices and gesture-providing algorithms intact while enabling newer devices and gesture providing algorithms.

This is where **OpenISS** plays its role as a flexible and extensible core component for the Illimitable Space System, by allowing itself to act as a *backend*, just like the first `SimpleOpenNI` [37] backend for the **ISSv2**, which was tightly coupled with main pipeline, among others and is currently providing the aforementioned limited hand gesture recognition and hand tracking functionality specific to the Microsoft Kinect in the **ISSv2** using outdated drivers and a limited gestures dictionary. In this context, *backend* means the underlying components of the **ISSv2** that are responsible for providing certain computer vision pipeline functionality that the **ISSv2** leverages for real-time performance visualization on stage. Henceforth, through an **OpenISS** instance, we will support the **ISSv2**, as a more flexible, modern, and sustainable backend.

So, this research work is not about providing a novel *Vision-Based Hand Gesture Recognition and Hand Tracking* algorithm, or an improvement, or an aggregation of existing ones. Rather, we will leverage the existing solutions available in the form of middleware such as NiTE2.0 and Nuitrack, or even meta-operating systems such as ROS to provide real-time hand gesture recognition and hand tracking functionalities, and consume the essential data structures that these middleware provide and adapt them in a uniform manner via our common platform that is flexible and extensible to enable real-time natural interaction from a gesture point of view in diverse 2D or 3D applications. Thus, this work provides and abstract software model to enable such applications. We demonstrate these capacities by developing a solution usable for the implementation of **ISSv2**'s backend requirements, as well as enabling a usable platform for a forensic investigator to use hand gestures to interactively create observation sequences and introduce semantic links between digital evidence objects and encode them in a FORENSIC LUCID program as elaborated further. The high-level abstraction from a hand gesture interaction perspective of our work is illustrated in Figure 4. The two specific motivational scenarios exemplify our needs, elaborated in detail in Section 1.2.3 and Section 1.2.4, along with the reasoning for their relationship with the needs created by our previous works mentioned in Section 2.1 fit into this architecture. These scenarios directly fall into the *Vision-Based Hand Gesture*

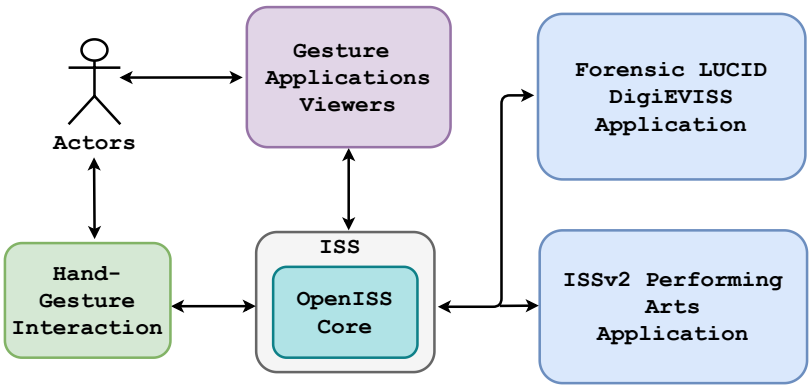


Figure 4: Coarse abstract overview of this work

application categories *Assistive Technologies* and *Entertainment* as mentioned in Section 1.1. We then evaluate our solution to demonstrate that it can enable hand gesture interaction for our scenarios and requirements with real-time response.

1.4 Goals and Objectives

Now that we have elicited the functional and non-functional requirements in Section 1.2.6, from the motivational scenarios in Section 1.2.2, we will proceed to define our end goal and the objectives that we must meet to achieve our end goal. Simply put, we want to provide a common, flexible and extensible solution that enables real-time hand gesture interaction for two different **ISSv2** and forensic evidence management in a 3D warehouse-like application:

- To enable a forensic investigator to manipulate digital evidence objects using hand gestures to create observation sequences and semantic links using hand gestures in **DigiEVISS**.
- To enable hand gesture interaction functionality for **ISSv2** via a flexible, extensible and device adaptable solution that is real-time, cross-language and cross-platform, minimally affecting existing infrastructure of the **ISSv2** itself through the **OpenISS** core for performance visualization and performing arts.

Our objectives are essentially to fulfill and evaluate our functional and non-functional requirements in Section 1.2.6 and the end goal is to enable both our scenarios mentioned in Section 1.2 via a common, flexible and extensible solution.

1.5 Research Questions

We pose the following research questions that we must answer in this research work to corroborate our objective and requirements. These research questions drive our evaluation scenarios in Chapter 5 as a proof of concept to demonstrate the work done in order to realize our requirements.

- *Can we design a general extensible solution that meets all our stated requirements and enable us to realize both of our motivation scenarios described in Section 1.2.2?*
- *Can we enable comparisons among different gesture libraries or middleware via our solution to make recommendations regarding which gesture provider suits which application type?*
- *Specifically, which gesture library or middleware is better suited for Motivation Scenario 1, in Section 1.2.3?*
- *Specifically, which gesture library or middleware is better suited for Motivation Scenario 2, in Section 1.2.4?*
- *How much overhead does our solution and ROS introduce to the system?*

1.6 Scope of the Thesis

In this section, we will describe the boundaries of this research work. We primarily focus on providing a common solution for our two motivation scenarios mentioned in Section 1.2.2. Although, we provide qualitative and quantitative analysis from a functional and non-functional point of view, we do not perform any meaningful formal end-user usability evaluation specifically with an actual forensic investigator or a dance show in a controlled environment, which we defer to the future work. Now, specifically for API Usability, see NFR2, we also do not perform any formal user evaluation with a developer rather we design our API keeping in mind a subset of Nielsen's heuristics [32] applicable to our work to guide our design to inherently cater to these usability heuristics but for APIs [29].

Despite the fact that there are other open source and proprietary middleware, libraries etc. available that provide rich features like gesture recognition, hand tracking and, even individual fingers tracking such as `Leap Motion` and `OpenPose`, we compare only NiTE2.0, Nuitrack, and then, both of these over the ROS middleware as depth and gesture information providers for our solution.

Similarly, there is a wide array of depth sensors or cameras that provide depth data crucial for gesture recognition, hand tracking, skeleton tracking and other such related applications. We have performed limited testing with available devices such as the Microsoft Kinect v1, Kinect v2 and Intel RealSense D435. However, this definitely must not affect the adaptability of our solution.

Finally, we will perform limited testing of the service side of our solution via ROS between two machines only, where the first is acting as a server or data publisher while the second acting as a client or data subscriber to experiment and calculate the overhead introduced by ROS communication infrastructure. However, we will not test multiple devices or drones simultaneously with the ROS for a single application.

1.7 Contributions

In this section we list the major contributions made in this research work. These contributions directly cater to the requirements and objectives defined previously in Section 1.2.6 and Section 1.4 and related supporting work.

1. We provide a common solution that provides functionalities such as real-time hand gesture recognition, detection and hand tracking for specific, yet, diverse applications that require natural interaction via hand gestures.

2. We provide a common solution that can be easily extended to accommodate prospective hand gesture recognition algorithms/libraries/middleware that may outperform the current ones in terms of advanced gestures, diverse gesture dictionaries or faster and efficient vision based recognition and detection of gestures.

3. We provide a common solution that provides a high-level abstraction for complex tasks and facilitates an easy understanding for creating diverse applications specifically for natural interactions via hand gestures.

4. We provide a common solution that enables interoperability with a mature and powerful meta-operating system ROS or REST services and enables decoupling of the depth sensing devices from the end-user application, if required.

5. We provide an integrated sample application that demonstrates various functional and non-functional aspects of our solution and at the same time acting as a significant resource on how to use our solution to create applications that require natural interaction with hand gestures.

6. We provide a ROS package of our solution itself to the open source community to use it to test and create hand gesture interaction applications.

7. This work is the first to enable Nuitrack in the **Processing** environment.

8. Related publications:

   - *Toward Multimodal Interaction in Scalable Visual Digital Evidence Visualization Using Computer Vision Techniques and ISS* [4].
   - *ISSv2 and OpenISS distributed system for real-time interaction for performing arts.* [38].
   - *OpenISS depth camera as a near-real-time broadcast service for performing arts and beyond* [34].

9. We provide a Docker image to free the end user from the hassle of deployment of our solution and associated preliminary deployment setup.

10. *Demand-Driven SOA Simulation Platform Based on GIPSY for Context-Based Brokerage* [39].

1.8 Structure of the Thesis

The forthcoming Chapter 2 provides the required background in detail in the context of this research work. Then, in Chapter 3 we describe the approaches applied to realize and evaluate our solution to the problems stated in the problem statement and fulfill our various functional and non-functional requirements. Subsequently, the architecture and design details along with the implementation details are presented throughout Chapter 4. Lastly, in Chapter 5 and Chapter 6 we evaluate the work done, discuss the results and limitations of our design and implementation and conclude.

Chapter 2

Background

In this chapter we will discuss essential foundational resources such as research works, texts, tools, technologies and concepts that helped this work take shape. It is necessary to mention the works that jointly create the need for this follow-up research work [46].

Firstly, Serguei Mokhov's Ph.D. thesis titled *Intensional Cyberforensics* [3], that presents FORENSIC LUCID, a dialect of the LUCID programming language that uses the intensional programming paradigm to formally model and implement a cyberforensics investigation process including digital evidence with backtracing of event reconstruction and so forth explained further in Section 2.1.3. Now, on top of this, we explore a scalable management, visualization, and evaluation of digital evidence in the context for cybercrime investigations to enhance FORENSIC LUCID language composition and interaction from a usability perspective in our work presented in [4] (see Section 2.1.4).

Secondly, to fulfill the functional requirement (**FR2: Interact using Motion and Gesture**) of the current version of Illimitable Space System that is the **ISSv2** [2] that we will address in this work from a gesture perspective only as mentioned briefly in the previous chapter in Section 1.2.1. At the same time, a partial requirement of this research work to interface with ROS and leverage its functionality originally stems from our other work presented in [34] (see Section 2.1.5). Nevertheless, recent preceding **OpenISS** works, on **person re-identification** [47] by Haotao Lai and

the **facial recognition framework** [48] by Yiran Shen play an important role in shaping up this work as well and will eventually sit adjacent to these preceding works in a symbiotic configurable environment. Both these works laid some groundwork such as **device abstraction module** in parallel to which we will build our solution. However, the reader must note that all of these related works have already been introduced except the last two, or atleast mentioned in the previous chapter in sections such as Section 1.2.1, Section 1.2.2 and Section 1.2.6 but, elaborated in detail in this chapter further in their respective subsections under Section 2.1.

Last but not the least, works that provided primary insights for writing style strictly include Yi Ji's work titled **Scalability Evaluation of the GIPSY Runtime System** [49] and Alexander Simard's thesis titled **A Framework for Interoperability Across Heterogeneous Service Description Models** [50].

2.1 Related Work

In this part of this chapter, we elaborate all the related works that directly influence this research work including the ones briefly mentioned in the above paragraph. We successively discuss various previously mentioned works in depth in the upcoming sections. The forthcoming sections will provide a relatively complete picture of how these different works fit together however strictly in context to this research work.

2.1.1 Illimitable Space System

The Illimitable Space System, as mentioned briefly in the Chapter 1, page 1, is the main source of our functional and non-functional requirements for this work. It is designed to be a flexible, multi-modal, real-time, interactive, and configurable artists' toolbox used to create music visualizations, visual effects and interactive documentary film based on the inputs from users such as gestures, voice, motion, etc. The goal of the Illimitable Space System is to enhance the interaction between actors and graphics so that it is all projected as one integrated piece. It is a multidisciplinary project with contributors from computer science, software engineering, computation

and media arts, and design. Moreover, Illimitable Space System and now its open source backend core **OpenISS** [25] rely on computer vision techniques and machine learning provided by **OpenCV** [51] and **MARF** [52], motion capture libraries for the Microsoft Kinect depth cameras and others. It additionally includes sound control, input from voice and music, and augmented and virtual reality components to co-create either augmented performance or have an installation or film, or use as an education tool for artists or children [23, 53, 54].

The functionality of the system generally includes the actors who are performing, light, sound, computer graphics, animation, and resulting interactive visual effects. The Microsoft Kinect sensors (depth cameras) were primarily used to capture motion data for visual effects generation, lighting, projection, audio and other basic elements that focus on actors' performance. Gestures, motion and voice are captured and are taken as input. It is processed using motion and speech recognition subsystems. After combining the resulting data with computer graphics and animation, the output is projected on the screen and/or on the performer in real-time. All these are captured via the *Data Capture System* to generate a meaningful response to the performers and/or the audience. Depending on the setup, participants, including actors, audience, and other users may freely move in the designated physical space. The captured data from the theatrical space are transferred to the Illimitable Space System for processing and feedback. A high level block diagram of the system can be seen in Figure 5 and the conceptual pipeline can be seen in Figure 6 [24].

As we briefly mentioned the evolution of Illimitable Space System before in the Section 1.2.1, page 4 in Chapter 1, we will further elaborate on it here starting with **ISSv1** [24].

2.1.1.1 ISSv1

The very first iteration of the Illimitable Space System was **ISSv1**, an experimental foundation of which was laid out in Miao Song's Ph.D. thesis titled *Computer-Assisted Interactive Documentary and Performance Arts in Illimitable Space* [24]. Some **ISSv1**-based works made it to public spaces and stage [55–59]. **ISSv1** is

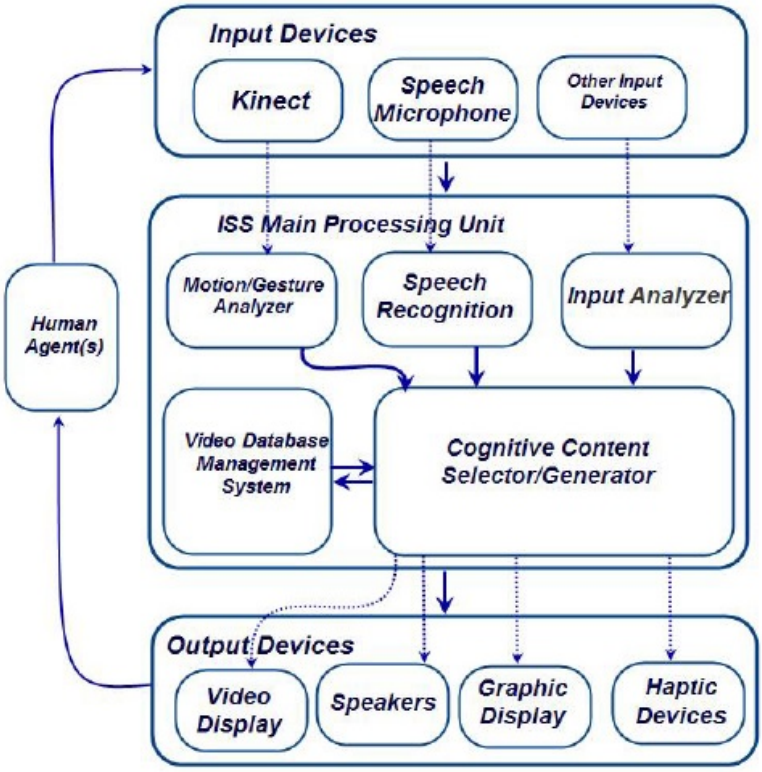


Figure 5: Block diagram of illimitable space system (ISS) [2]

no longer maintained however, and its development stopped mainly because of its inflexible architecture and its complete reliance on the **Microsoft XNA**, which was discontinued. Moreover, **ISSv1** was not portable since it used Microsoft's C# and **Kinect SDK for Windows**. Additionally, it wasn't computation artist friendly because usually, artists use **Max** [60, 61], **PureData** [62] and **Processing** [26] and even **Unity**, but didn't use generally C# with XNA. However, in this prototype iteration the gestures and voice processing was provided by the **Microsoft Kinect SDK for Windows** [63–66] and was also too dependent on a single Windows platform [2, 53]. However, during the design, development and experimentation, the general architecture for ISS-like systems was established [67,68]. The interactive documentary film component of **ISSv1** has inspired the Forensic Lucid visualization application **DigiEVISS** for digital evidence representation and interaction elaborated further in Section 2.1.4 [3, 4].

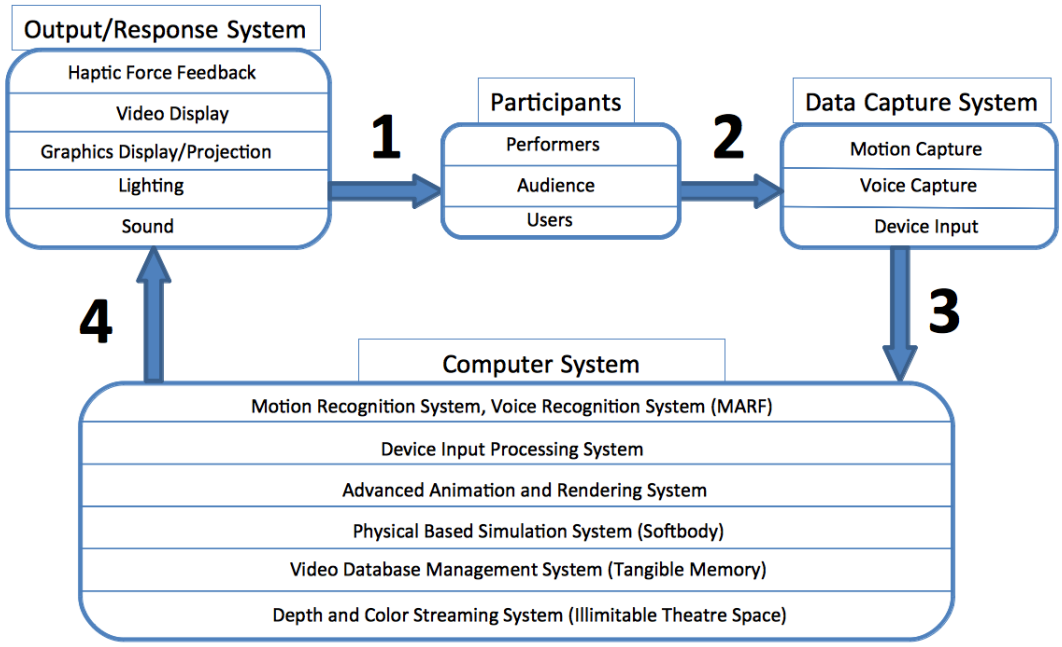


Figure 6: Conceptual pipeline of illimitable space system (ISS) [2]

2.1.1.2 ISSv2

ISSv2 was built using mixing of **Max** and **Processing** originally to make it cross-platform and artist friendly to program and for software engineers and computation/media artists to better collaboration [2, 69]. However, this resulted in the Illimitable Space System losing some of the **ISSv1** functionality such as partial gesture recognition, voice processing, and green screen. Furthermore, newer devices such as Microsoft Kinect v2, ZED, Intel RealSense D435, and, Structure Sensor with better tracking hardware and accuracy, at the same time, different libraries started showing up with better tracking than `SimpleOpenNI` [37] built on top of OpenNI2.0 and NiTE2.0. Thus, there was a strong need to regain the lost functionality while keeping Illimitable Space System flexible and portable to accept, accommodate and support newer devices, libraries or frameworks. We started with the backend driver wrappers for devices, but there were also interesting libraries for creative coding and

communications like **OpenFrameworks** and **Open Sound Control** that we needed to integrate with the Illimitable Space System. Therefore, **OpenISS** core came into existence at this point to support **ISSv2** needs as well as other applications within the Illimitable Space System and beyond.

Now, the Illimitable Space System mainly consists of three logical core components namely, *recognition, computing, and media output and control*. The motion capture and recognition, voice recognition, and device processing systems are filtering, parsing, and sorting motion, audio, and device/sensor input streams captured into **ISSv2** by various capture devices. The Illimitable Space System then computes, simulates, and produces as needed computer graphics, projection mapping, and dynamic audio output based on the mathematical and physically based algorithms. The system then re-renders the newly computed visual effects onto projector(s), if/as needed plays generated audio output, and, if required, may controls dynamic lighting (interacted with originally as a result of participants actions) [2]. Some of the real-life performances using the Illimitable Space System can be seen in Figure 7 [2,54,70–72].

2.1.1.3 ISSv3

ISSv3, is a small research-creation interactive documentary prototype to enhance user engagement and create immersion for interactive documentaries by combining Augmented Reality and Virtual Reality in order to create virtual worlds where users can enjoy watching, hearing, and reading content [73, 74]. **ISSv3** follows the same architectural patterns as **ISSv2**, but its application side is written primarily in Unity. Our longer-term goal to make the **OpenISS** backend available in Unity for **ISSv3**, similarly to the Zigfu Development Kit [75] and Nuitrack [5] (cf. Section 2.2.3, page 48) for cross-platform mobile and desktop AR/VR application development, but this is currently outside of the scope of this thesis.

Figure 7: ISSv2 various visual effects performances

2.1.1.4 OpenISS

Open Illimitable Space System is a suite of configurable tools and the new open-source core [25] for the **ISSv2**and other similar applications for motion-based data VFX production and beyond. Inspired by the development of the different version of the **ISS**, it exhibits multi-modal interaction and provides a platform for artists to enhance their performance by leveraging modern day technology and being available and open platform for education, learning, and contributions. **OpenISS** initially designed and led by Serguei Mokhov began as an experiment with its C API in 2016–2017 as a teaching medium in C and systems programming and shell scripting [76]. Subsequently, C++, Java, and JavaScript APIs started to appear in order to expose the motion capture data as-a-service for wider creative-broadcast illimitable [34,38,77] dissemination and interaction inspired by the notion of *illimitable space* as defined in Miao Song's thesis [24]. As a result, **OpenISS** rapidly became

an experimental C/C++ framework with wrappers and API for motion capture and other interactions and research subjects. This also allowed device abstraction as a basic default requirement.

Moreover, **ISSv1** and **ISSv2** identified issues with person tracking and re-identification. Likewise, additional requirements for another application were produced for facial expression detection and landmark detection for character animation VFX. This is where the works by Lai and Shen extended the **OpenISS**'s framework design to incorporate instances employing deep learning, green-screening, and facial processing [47, 48].

This core is designed to allow the **ISSv2** platform to be run as a distributed system. Video and depth capture are done from a computer acting as a server with a client component for displaying the applied effects and video from a web browser. This has the added benefit of allowing the artist to broadcast their performance live and opens the way for audience interaction. More information regarding this web service aspect of **OpenISS** is further elaborated in our previously mentioned work in Section 2.1.5, page 42. Experimental performances were done as well in District 3 and CHI 2018 [38]. An open Docker-container is being produced as well that contains all necessary dependencies for **OpenISS** development (cf. Section 2.2.7).

OpenISS has its viewer modules being extended to interactive browser based art from Chao Wang [78] as well as Max [60] and PureData [62] data-flow frontends in-progress by Jonathan Llewellyn and team [53, 54].

2.1.2 GIPSY

The *General Intensional Programming System* is a programming environment designed for the compilation/execution of all dialects of the LUCID family of programming languages. It consists in three modular sub-systems: GIPC, the *General Intensional Programming Language Compiler*, GEE, the *General Eduction Engine*, and RIPE, the *Intensional Run-time Programming Environment* [79]. However, just like the FORENSIC LUCID, we will limit the extent to which GIPSY is presented in this research work. For now, the reader must know that GIPSY houses the

development of the GIPC's FORENSIC LUCID compiler instance [3]. GIPSY is an open-source platform implemented primarily in JAVA to investigate properties of the LUCID family of intensional programming languages, including FORENSIC LUCID. Moreover, GIPSY is a great body of work, developed and maintained by the *GIPSY Research and Development Group* at Concordia University.

Now, using the GIPSY platform, programs written in different dialects of LUCID can be compiled and executed in a distributed processing environment. Therefore, we will quickly converge here to the fact that GIPSY serves as a compilation and evaluation platform for the FORENSIC LUCID language. GIPSY has a collection of compilers under the GIPC framework and a corresponding runtime environment under the GEE. These two modules are the primary components for compilation and execution of intensional programs such as FORENSIC LUCID programs in the context of cybercrime investigations. Within the context of this research we merely leverage the GIPC in the GIPSY to simply validate the semantic correctness of FORENSIC LUCID encoding program generated corresponding to the observation sequence of digital evidence in a 3D application where one uses our solution to semantically link digital evidence objects in a sequence.

It is worth mentioning our related work [39] was investigating a service-oriented use of GIPSY's GEE's tiers in a simulation, which inspired in part the OpenISS-as-a-Service project [38].

2.1.3 Forensic Lucid

We will start by reiterating our previous statement in the Section 1.2, page 4 in Chapter 1 that we will only expose those aspects of FORENSIC LUCID [3] that are essential for the reader to be aware of within the context of this research work. Now, moving on with a gentle introduction to the FORENSIC LUCID language. So far, the reader is loosely aware of a few things such as, a FORENSIC LUCID program is a data-flow program that can be visually represented as a data-flow graph [27] and, with FORENSIC LUCID, a cybercrime investigator can reason about cyberforensic cases by expressing in a program form the encoding of the evidence, witness stories,

and evidential statements as FORENSIC LUCID contexts, that can be tested against claims to see if there is a possible sequence or multiple sequences of events that explain a given story. At this point, we introduce the reader with two terms strictly in the domain of the FORENSIC LUCID language.

Definition. *The **observation sequence** context, **os** represents a partial description of an incident told by evidence or a witness account (electronic or human).*

It is formally a chronologically ordered collection of observations representing a story witnessed by someone or something (e.g., a human witness, a sensor, or a logger). It may also encode a description of any digital or physical evidence found. All these *'stories'* (observation sequences) all together represent an *evidential statement* context about an incident (its knowledge base).

Definition. *The **evidential statement**, **es** is an unordered collection of observation sequences.*

Where, a simplified visual representation of the evidential statement can be seen in Figure 8.

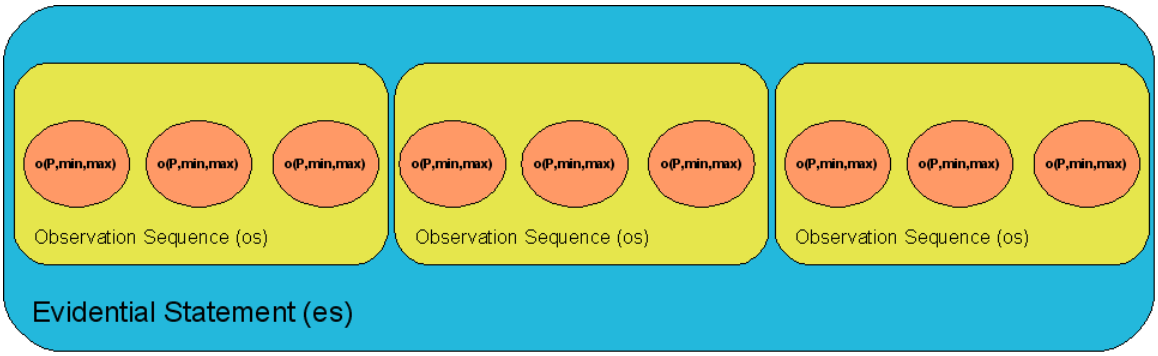


Figure 8: Evidential statement [3]

Definition. FORENSIC LUCID*, is a dialect of the* LUCID *programming language that uses the intensional programming paradigm to formally model and implement a cyberforensics investigation process with backtracing of event reconstruction, formalizing and modeling the evidence as multi-dimensional hierarchical contexts and proving or disproving the claims in an intensional manner of expression and eductive evaluation [3].*

Even though the above definition might seem daunting for a reader without adequate background in LUCID, intensional programming and cybercrime investigation and reasoning in general, we will no more discuss the breadth of capabilities of the FORENSIC LUCID language. For the curious readers we recommend the corresponding work [3] that constitutes the entirety of the body that is FORENSIC LUCID itself. Rather, we will focus on the role and significance of FORENSIC LUCID within the context of this research. For cyberforensic analysis, FORENSIC LUCID can express the encoding of the evidence, witness stories and evidential statements in program form that can be tested against claims to see if there is a possible sequence or multiple sequences of events that explain a given story [46]. However, that includes writing a FORENSIC LUCID program manually. An example of a simple FORENSIC LUCID program taken for a collection of compilable examples from **GitHub** [80] is given in Listing 2.1. We use this program to build our proof-of-concept visualization interaction application **DigiEVISS** to reproduce the observation sequences dynamically from their visually constructed ordering into a written textual file that later can be sent to GIPC described in Section 2.1.2 for syntax and semantics checking.

```
os
where
  observation o0 = ("A printed" => "very well", 1, 0, 0.85, 123);
  observation o1 = ("B printed" => "not very well", 1, 0, 0.15, 231);
  observation sequence os1 = {o0, o1};
  observation sequence os2 = o0 fby o1;
  evidential statement es1 = os2;
  os = os2;
end
```

Listing 2.1: Simple FORENSIC LUCID observation sequence context

Now, the long term goal of the FORENSIC LUCID project is to enhance the usability of the FORENSIC LUCID language composition and interaction via providing a scalable management, visualization and evaluation of digital evidence in the context of cybercrime investigations. This may be achieved by enabling investigators to represent and create semantic links among digital evidence within an easy to use

interface powered by multi-modal interactions including but not limited to eye-gaze, gestures and navigational hardware [4].

However, reiterating our previously mentioned statement that in this work we only focus on enabling gesture interaction via our solution to quickly visualize, analyze and sequence digital evidence via gestures by enabling a baseline platform as a first building block to build upon for an eventual use by an investigator. At this time, we will not provide a scalable management or evaluation of digital evidence. Within this work we represent our digital evidence with pre-defined data rather than a real digital evidence. The goal is to provide a proof-of-concept that we can enable such a subset of functionality that emulates case specification via observation sequencing and evaluating the validity of the underlying FORENSIC LUCID program encoding via GIPSY.

Therefore, the intersection of this research work and the FORENSIC LUCID language is merely *enabling* the interaction aspect of the language along with its visualization in 2D/3D only to enhance the limited usability of the language itself in forensic case specification and evaluation by GIPSY. However, the previous statement is quite broad and must be interpreted very cautiously in accordance with the context of this research work.

2.1.4 Toward Multimodal Interaction in Scalable Visual Digital Evidence Visualization Using Computer Vision Techniques and ISS

In our research work [4] we explored and positioned an idea of a scalable management, visualization, and evaluation of digital evidence in the context for cybercrime investigations with inspiration from the interactive 3D documentary subsystem of the **ISSv1**.

The proposed modifications would enable investigators to represent and create semantic links among digital evidence objects within an easy to use interface powered by multi-modal interactions including but not limited to eye-gaze, gestures and

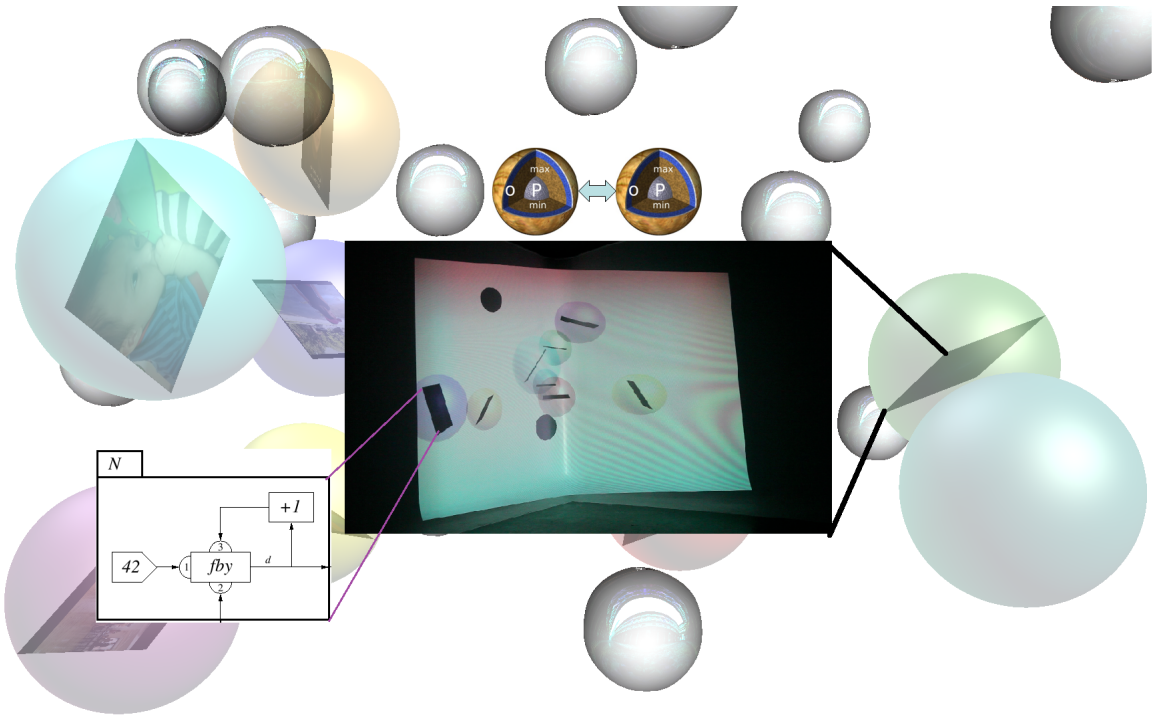


Figure 9: Conceptual and actual nested visualization representation based on ISSv1

navigational hardware as mentioned prior. That work may scale when properly re-engineered and enhanced to act as an interactive '3D window' into the evidential knowledge base grouped into the semantically linked 'bubbles' visually representing the documented evidence. By moving such a contextual window, or rather, navigating within the theoretically illimitable space an investigator can sort out and reorganize the knowledge items as needed prior launching the reasoning computation. This notion of the sort in literature is referred to as "serious games" [44]. The most recent significant work of that type appeared this year only in the SIGGRAPH Asia Real-Time Live in Brisbane, Australia, called *A Clever Label* [81], which is an investigative documentary experience that combines a personal story with a interactive room-scale VR and gesture-based interaction experience to examine multiple sides of a topic from big data and to enable something similar is our long-term vision in the open source world. We have plans to possibly collaborate or interoperate with that project. However, strictly in context to this work it is still a rather novel proposal that will demonstrate feasibility of limited language usability of FORENSIC LUCID that and can also be scaled to the entire language.

The interaction design aspect would be of a particular usefulness to open up the documented case knowledge and link the relevant witness accounts and group the related knowledge together. This is a proposed solution to the large-scale visualization problem of large volumes of 'scrollable' evidence that does not need to be all visualized at once, but behave like a snapshot of a storage depot. As an example, stills from the actual **ISSv1** installation hosting multimedia data (documentary videos) users can call out by voice or gestures to examine the contents as in Figure 9 [67]. We propose to reorganize the latter into more structured spaces so that the investigators can create semantic links to group the relevant evidences together and for subsequent evaluation by the distributed GIPSY backend engine.

Available gesture-based interactions using Kinect and similar depth cameras with **OpenCV** [51] are the enabling HCI aspects for the investigator to link the evidential items in the 3D space. In our general approach, we propose an architecture to enable interactive visual windowing into the digital evidence processing as an investigator aid tool as seen in Figure 10. Thus, the preferred method of interaction during analysis and human insight phases prior to or after distributed processing of the evidence and event reconstruction algorithms [4].

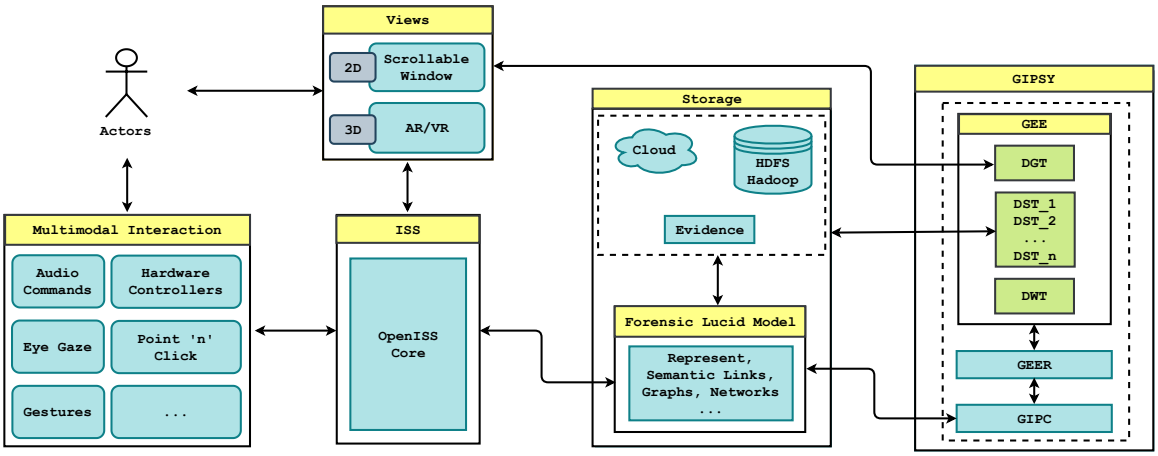


Figure 10: High level architecture of the proposed solution in [4]

However, as already discussed briefly in the Section 1.2 in Chapter 1, we **partially** base our research in this thesis on [4] work to enable a subset of this proposed solution from a gesture interaction perspective and a baseline platform to develop

for a digital investigator to emulate visualization of digital evidence objects in a 3D application for observation sequencing and their semantic evaluation by the GIPSY's GIPC. Figure 10 also exposes the FORENSIC LUCID *Evidential Storage and Compute Infrastructure* in Figure 4, page 23 in Chapter 1. Additionally, out of the multi-modal interactions modes in the figure we enable the gesture interaction as a part of our solution and recommend which currently available middleware and gesture dictionaries are suitable for this type of task.

2.1.5 OpenISS depth camera as a near-realtime broadcast service for performing arts and beyond

Recently, we extended the **OpenISS** by enabling SOAP and REST APIs, which brought us closer to a more flexible and scalable architecture for being available as an interactive broadcast service over the Internet to a wider audience [34, 38]. Due to a few limitations we entitle our work as near-realtime and learning from our experience with this work, in our current solution we deliver the data as a stream that can be broadcast over the ROS network in a client-server fashion.

2.1.6 Gesture Tracking

A gesture is a non-verbal form of communication essentially used to convey a message via movement of a specific part of body, such as hands, face, etc. Sign language is a standard example of how gestures can enable people who have a speaking or hearing disability to be able to communicate through the sign language itself [82]. However, in terms of interaction with machines a gesture is any physical movement that can be detected via a motion sensor and classified. The role of gestures in Human Computer Interaction in general was first seen as early as in the mid 1960's and caught the limelight with the Apple Newton in 1992 [83].

2.1.7 Vision-Based Gesture Recognition

Definition. *Gesture Recognition is a machine's ability to infer gestures via some mechanism and perform one or more tasks.*

However, within the scope of this work we will essentially talk about vision based gesture recognition observed as early as the 1980's [86].

Definition. *Vision-based gesture recognition is the process of recognizing meaningful human movements from image sequences that contain information useful in human-human interaction or human-computer interaction. [87]*

This is distinguished from other forms of gesture recognition based on input from a computer mouse, pen or stylus, sensor-based gloves, touch screens, etc. [88].

2.1.8 Nielsen Usability Heuristics

In 1995, author Jakob Nielsen defined 10 usability heuristics for interface design and in [29] the authors have stated that these heuristics apply equally well to APIs as to regular user interfaces. These are called heuristics because they are broad rules of thumb and not specific usability guidelines. [32].

- **Visibility of system status** – User must be aware of the current status of the system or provided with appropriate feedback.
- **Match between system and the real world** – User language, system should match the mental model and expectations.
- **User control and freedom** – Undo and redo operations to enable user to switch between states.
- **Consistency and standards** – Consistency in naming conventions, actions, menus.
- **Error prevention** – Prevent users from making mistakes, pop-up dialog when deleting something.

- **Recognition rather than recall** – Visibility of actions, operations, options, such as stop action in red color written big and highlighted.
- **Flexibility and efficiency of use** – Enable shortcuts, tutorials or hints/tips for novice users.
- **Aesthetic and minimalist design** – Relevant and non-redundant design via essential information only.
- **Help users recognize, diagnose, and recover from errors** – Error or log messages in simple language describing exact error and suggesting a solution.
- **Help and documentation** – Necessary resources to explain usage and help documentation with appropriate drawings if necessary.

2.2 Libraries, Tools and Middleware

In this section we will talk about various tools, middleware and libraries that we will directly or indirectly use to realize our research work.

2.2.1 OpenKinect

OpenKinect is an open source community of people primarily providing cross-platform device drivers specifically for the Microsoft Kinect Sensor [90–92]. The community has provided open source drivers `libfreenect` [93] and `libfreenect2` [94] and is actively maintaining both for the Kinect v1 and Kinect v2 respectively which work on different technologies namely *structured light* [95] and *time-of-flight* [96]. These drivers were also the first ones to be published for the Kinect line of sensor devices. Additionally, there is an interesting story behind the birth of `libfreenect`. It all started with *Adafruit Open Kinect* bounty for the first open source drivers for the Kinect, which was claimed by Hector Martin and henceforth setting a foundational proof of concept to leverage the cheap yet powerful Microsoft Kinect device for its applications in specific domains.

2.2.1.1 libfreenect and libfreenect2

The `libfreenect` provides device driver for the Microsoft Kinect v1, which had models 1414, 1473, and 1517. It also supports and provides access to the following characteristics for the Kinect devices:

- RGB and Depth Images
- Motors
- Accelerometer
- LED
- Audio

Whereas, `libfreenect2` provides device driver for the Microsoft Kinect v2 and provides support for the following:

- RGB image transfer
- IR and depth image transfer
- Registration of RGB and depth images [94]

OpenNI2-FreenectDriver

OpenNI2-Freenect Driver [97] acts as a bridge to `libfreenect` and `libfreenect2` and is implemented as an OpenNI2.0 driver; thus, allowing OpenNI2.0 to use the Microsoft Kinect hardware on Linux, OSX, and Windows operating systems.

2.2.2 PrimeSense

OpenNI2.0 and NiTE2.0 were released collectively by PrimeSense in response to the efforts of the open source community with `libfreenect`. OpenNI, stands for Open Natural Interaction, which was eventually shutdown after Apple, Inc. [98] acquired PrimeSense [99], the founding member of OpenNI on November 24, 2013. Prior

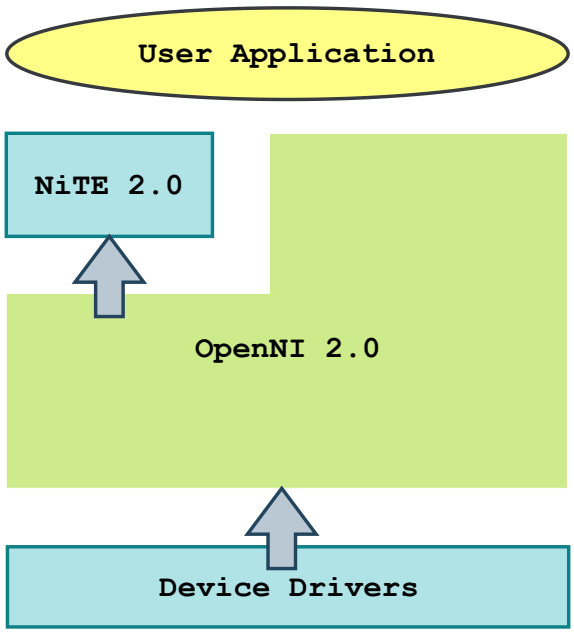


Figure 11: OpenNI2.0 and NiTE2.0 stack

to its purchase the API had versions OpenNI1.0 and OpenNI1.5 and subsequently OpenNI2.0. However, Occipital [100], a former partner of PrimeSense kept the OpenNI2.0 alive [101] as a forked version of the same as an open source software for their own depth sensor product called the Structure Sensor [102]. Figure 11 illustrates a typical application using the OpenNI2.0 and NiTE2.0 stack.

2.2.2.1 OpenNI 2.0

OpenNI2.0 introduced major changes to the existing API by making it more device driven rather than data driven. OpenNI2.0 provides the following functionalities via its interface:

- Enabling connection with a device and provide IR, Color and Depth video streams.
- Encapsulate stream data into a frame of data along with stream info and configurations.
- Capture one or more stream output into a file of type ONI (OpenNI file format).

- Event driven depth access.

2.2.2.2 NiTE 2.0

NiTE2.0 acts as a middle ware integration layer which provides rich processing features over the sensor data and is based on OpenNI2.0 with more focus on natural interaction. Overall, NiTE2.0 provides the following functionalities via its API.

- Scene Segmentation
- Skeleton Tracking
- Pose Detection
- User and Hand Tracking
- Gesture Detection or Recognition

From an implementation point of view NiTE2.0 is a closed source proprietary middleware now owned by Apple. However, via its documentation available over the Internet and texts one can say that NiTE2.0 provides those above mentioned functionalities via its two main classes namely, the `UserTracker` class, which provides information related to users and their bodies and the `HandTracker` class, which provides information regarding hand tracking and hand gestures on the most recent scene.

The important classes that provide the majority of functionality are:

- `HandTracker` – Recognizes hand gestures and initiates hand tracking.
- `HandTrackerFrameRef` – Represents a frame of data with currently recognized hands and gestures.
- `GestureData` – Represents a gesture with associated data such as state, position coordinates, and, gesture type.
- `HandData` – Represents a real world human hand with associated hand tracking data such as id, state, position coordinates, and, tracking status.

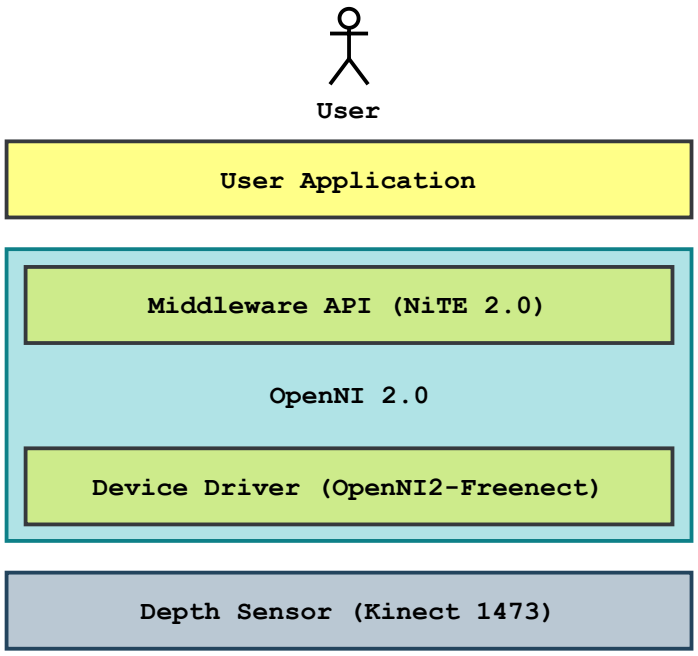


Figure 12: OpenNI and NiTE device and driver specific stack

Typically, as seen in Figure 12, the data are delivered from the depth sensor that is the Microsoft Kinect, which can interface to the Linux and MacOS operating systems via the OpenNI2-Freenect Driver [97] and then the data is passed onto the NiTE2.0 middleware which contains the *Gesture recognition and Hand tracking* algorithms that extract relevant information from the depth stream and color stream from the sensor device [103].

2.2.3 Nuitrack

Nuitrack API is a well-supported native (C++) synchronous interface based on the asynchronous commercial proprietary middleware layer that conceals all the interactions with 3D sensor and other devices, as well as with an operating system (Linux, Android). As a result, the user only applies Nuitrack API for writing cross-platform applications. Its architecture can be seen in Figure 13 and the various modules that facilitate the essential functionalities can be seen in the Figure 14. Nuitrack makes it possible to receive data from a 3D sensor to Android without root privileges required. The interaction with the Android OS occurs through JNI.

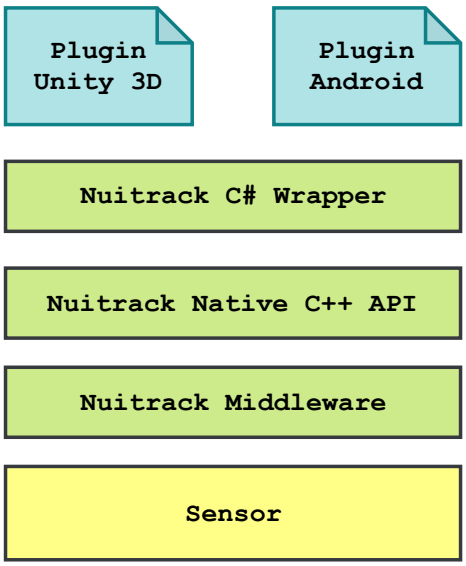


Figure 13: Nuitrack architecture [5]

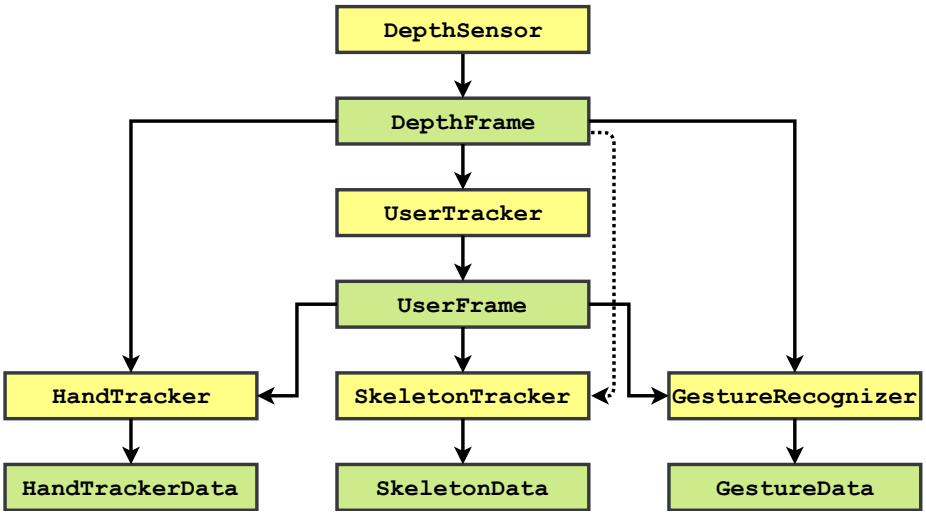


Figure 14: Nuitrack modules [5]

The SDK contains a Nuitrack C# wrapper, a plug-in for Unity3D and a plug-in for Android devices. In its list of dependencies Nuitrack SDK uses both above mentioned OpenNI and OpenNI 2.0 under the hood including `libfreenect`, `libfreenect2`, and `librealsense2`. Similar to NiTE2.0, as seen in Figure 15, the data are delivered from the depth sensor that can interface to the Linux, Windows and Android operating systems via different device sensors depending on the device manufacturer, here PrimeSense Sensor device driver and then the data are passed onto the Nuitrack

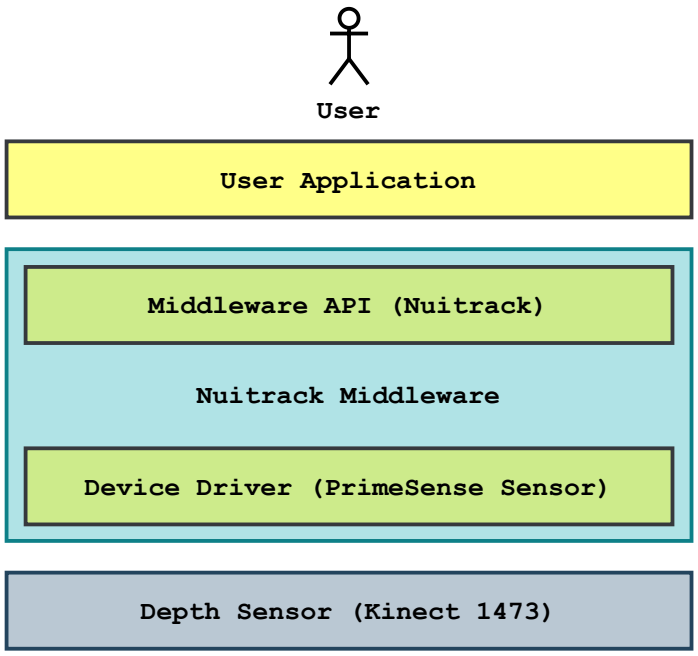


Figure 15: Nuitrack device and driver specific stack

middleware, which contains the *Gesture recognition and Hand tracking* modules that extract relevant information from the depth stream and color stream from the sensor device as seen in Figure 14. It is important to mention that currently Nuitrack supports up to seven different depth devices including the Kinect v1, Kinect v2 and Intel RealSense D415/D435.

2.2.4 Robot Operating System (ROS)

2.2.4.1 Overview

ROS stands for Robot Operating System, which unlike traditional operating systems is an open source meta operating system designed primarily to support code reuse for research related to robotics and computer vision, and provide a baseline to write software for robots by providing services like hardware abstraction, message passing between processes and low-level device control alongside a structured layer of communication over the host operating system typically in a heterogeneous computing cluster [33].

ROS uses a peer-to-peer topology to pass messages (similar to protocol buffers)

between different processes called nodes that are typically distributed across machines and to facilitate a loosely coupled communication infrastructure among off board computers running computation intensive tasks and on board computers on the robot itself carrying out tasks that rely on the output from the off board computers in a typical use case. ROS provides synchronous RPC-style communication over services defined by a name, and request/response message types and provides asynchronous streaming of data over topics.

In ROS code is organized into a granular structure referred to as package, which can be easily shared or distributed and managed via the official build system catkin that extends on top of `cmake` [105] build system to provide ROS specific functionality. ROS is written in C++ and PYTHON and extensively supports C++, PYTHON and LISP languages in terms of main client libraries. However, it does have experimental client libraries in JAVA, Go, HASKELL, Ruby and Node.js as well.

Currently, ROS supports Linux operating systems; however, there are build and install instructions for MacOSX and Windows, which have been titled "experimental" by the ROS community. In a nutshell, ROS is a distributed framework of nodes which can communicate via messages and can publish to or subscribe to a topic of messages of a defined type and are loosely coupled at runtime.

2.2.4.2 ROS Architecture

The ROS architecture has been defined and categorized into three conceptual levels [6]:

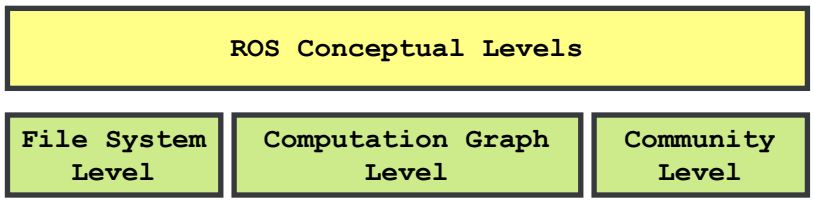


Figure 16: ROS conceptual architecture levels

- **The File System Level** – This level constitutes different kinds of files that are stored on the disk such as packages, messages, services and so forth. Usually, the packages, messages, services, and project manifests reside in the `catkin_`

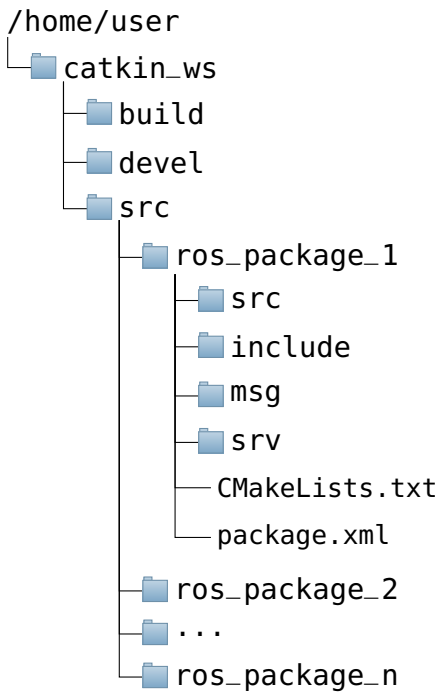


Figure 17: A typical ROS workspace and package structure [6]

`ws` directory with three sub-directories namely, `src`, `devel`, and, `build`. By default, ROS packages reside in `src` directory and after build and compilation the executables are in the `devel` directory and build related files are typically in the `build` directory.

- *Packages*, are the main unit for organizing software in ROS, the most atomic build item, and, the most granular thing that is built and released is a package.
- *Package Manifests*, typically, `package.xml`, provide metadata about a package, including its name, version, description, license information, dependencies, and other meta information like exported packages.
- *Repositories*, are collection of packages which share a common version control system and can only contain one package.
- *Message (msg) types*, are descriptions that define the data structures for messages sent in ROS.
- *Service (srv) types*, are descriptions that define the request and response

data structures for services in ROS.

- **The Computation Graph Level**, ROS creates a network of different connected processes where, any node within the system has access to the network, and can easily interact with other nodes, see the information that they are exhanging, and transmit data to the network itself.

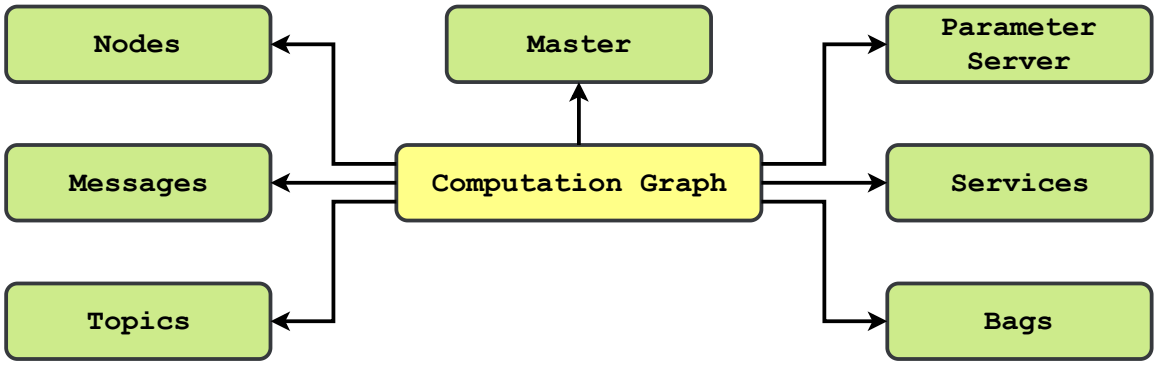


Figure 18: Computation graph level

  - *Nodes*, are processes that perform computation and is written with the use of a ROS client library, such as `roscpp` or `rospy`.
  - *Master*, provides name registration and lookup to the rest of the Computation Graph. Without it, nodes would not be able to find each one another or exchange messages, or invoke services.
  - *Parameter Server*, allows data to be stored by key in a central location.
  - *Messages*, allow nodes to communicate with each other where, a message is simply a data structure, comprised of typed fields.
  - *Topics*, act as channels for messages routed via a transport system with publish / subscribe semantics. A node sends out a message by publishing it to a given topic that is used to identify the content of the message in case a node that is interested in a certain kind of data will subscribe to the appropriate topic. There may be multiple concurrent publishers and subscribers for a single topic, and a single node may publish and/or subscribe to multiple topics.

- *Services*, are defined by a pair of message structures, one for the request and one for the reply and are offered by nodes under a name which is used by clients by sending the request message and awaiting the reply.
- *Bags*, are a format for saving and playing back ROS message data and storing data, such as sensor data [6].

- **The Community level** consists in Distributions, Repositories, ROS Wiki, Bug Ticketing, Mailing Lists, ROS Answers, and, Blog. The community also publishes open-source software on platforms like **GitHub** as well [6].

2.2.5 Processing

Ben Fry and Casey Reas, two graduate students at the MIT Media Lab developed **Processing** IDE [26, 107] in 2001 with an aim to facilitate simpler visual arts production with computer programming [108]. **Processing** is the made development environment for **ISSv2** as it was done in collaboration between software engineering, computer science, and computation arts students for various productions mentioned earlier. It was the best compromise between pure software development IDEs and simplicity of development and rapidly testing out modifications often hours before actual productions [2, 54, 109].

Currently, as an open-source software project, **Processing** has attracted quite a community strength, which contribute to the language actively. Also, **Processing** is based on top of JAVA, which makes it quite powerful yet it is much easier to learn than plain JAVA or C++. It is a software sketchbook and a programming environment for learning how to code within the context of the visual arts and rapid prototyping, which includes a 2D and 3D graphics API based on OpenGL and a JAVA rendering engine [110]. A **Processing** program is usually written as a **Processing** "sketch" as seen in Figure 19. Artists contribute various sketches for real-time graphics on `OpenProcessing.org` using the same sketches as well as `p5.js` to run them in browsers using WebGL. **Processing** IDE was also adapted to C language programming of *Arduino* open source hardware boards. mDreams Stage

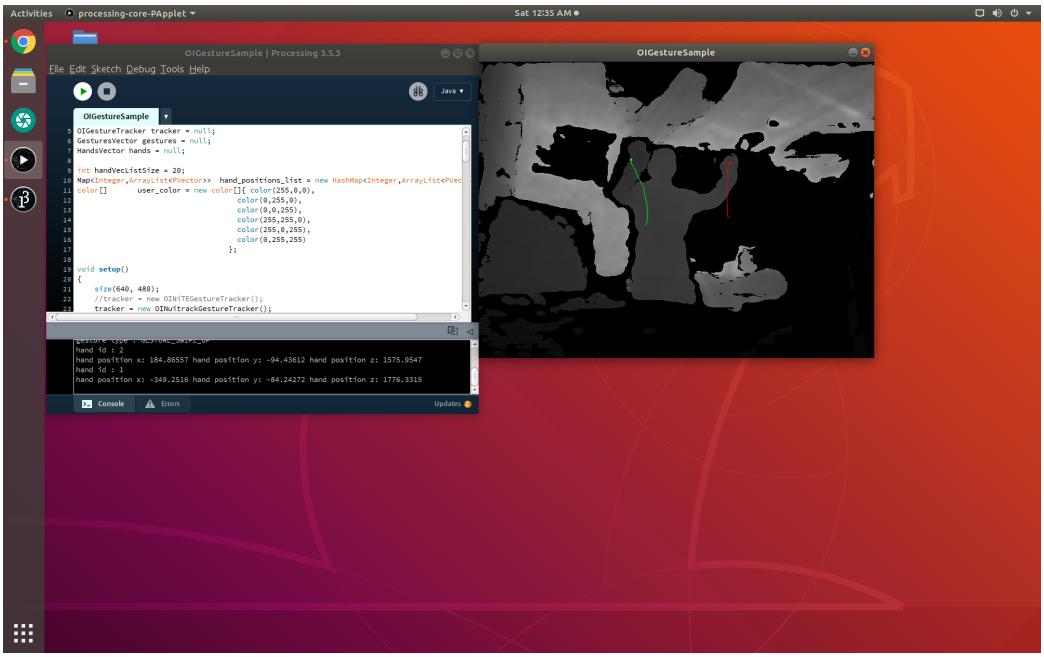


Figure 19: A Processing sketch

Research and Creation Group experimented of using an **ISSv2** extension prototype with a programmable LED light board in *Arduino* [111].

2.2.6 SWIG 4.0

SWIG stands for *Simplified Wrapper and Interface Generator* and is a software development tool that enables programs written in C and C++ to interface with a wide array of high-level programming languages. **SWIG** is a greatly flexible tool, but is also quite complex as well with a steep learning curve. The array of languages it can support can be seen just below:

- *Scripting Languages*

  JavaScript, Perl, PHP, Python, Tcl, Ruby

- *Non-Scripting Languages*

  C#, Go, Java, OCamL, Lua, R

SWIG takes an interface file with an extension *.i* which contains C-Style declarations for which **SWIG** will create wrapper code which acts as a glue between the C/C++

library and the target language.

In a relatively simpler context, **SWIG** is a compiler that can generate wrappers from C or C++ header files, once one lists everything that should be a part of the desired extension module as simple C-Style declarations in the interface file. **SWIG** will parse the interface file including the declarations and generate wrapper code for the target language [116, 117]. Most common use cases of this tool are, building interpreted interfaces to existing C programs and rapid prototyping and application development. However, the latter serves our purpose in this research work.

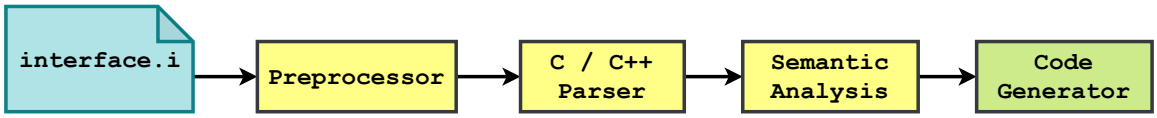


Figure 20: SWIG architecture

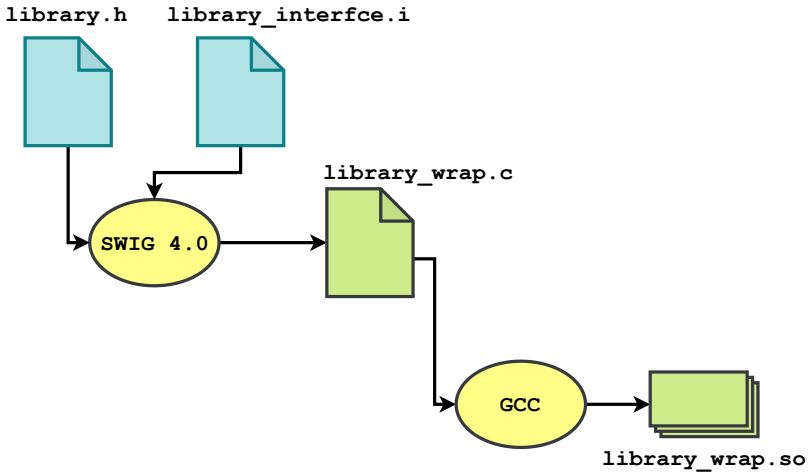


Figure 21: SWIG pipeline

- `interface.i` – This file declares or includes declarations that will be wrapped using **SWIG**. The target Java module name is set using the module directive.
- `interface_wrap.cxx` – Generated by **SWIG** based upon the above interface file and contains the JNI wrapper code that needs to be compiled and linked with the rest of C++ application.
- `library.h` – Header file containing declarations of the desired library that needs to be wrapped via **SWIG**.

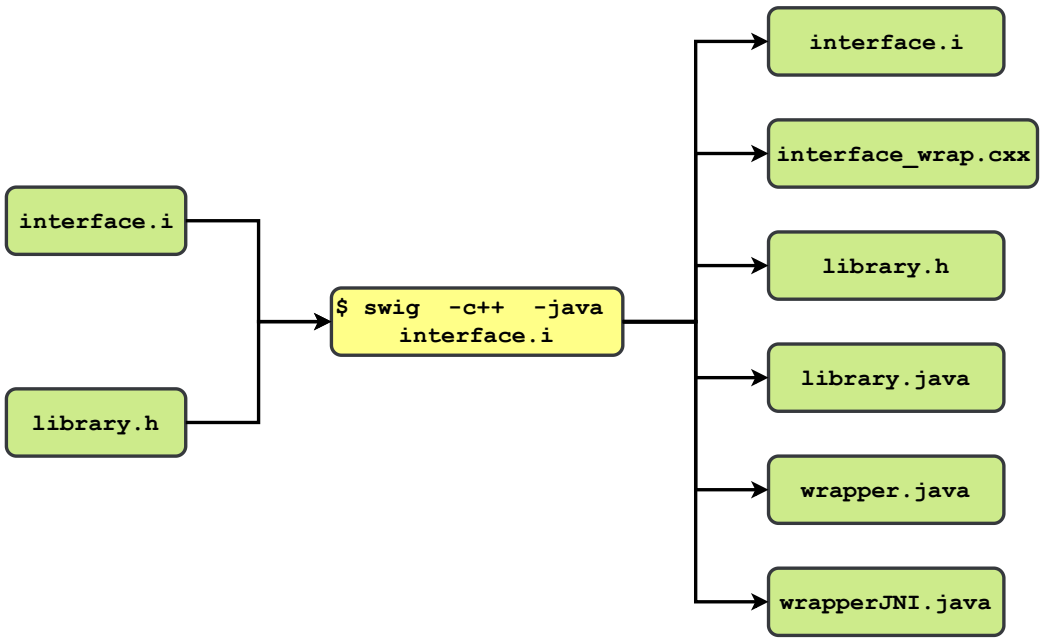


Figure 22: SWIG JAVA demo

- `library.java` – This is a proxy class, generated for each structure, union or C++ class that is wrapped. This class merely holds a pointer (`swigCPtr`) to the underlying C++ object. It contains getters and setters for the public member variables of the class. These functions call the native methods in the intermediary JNI class. Each proxy class has an ownership flag `swigCMemOwn` for the underlying C++ object, and its value determines that is responsible for deleting this object. If the value is false then the object will outlive the destruction of the proxy class. However, when an object is created by a constructor or returned by value, then the ownership of the result goes to JAVA, while pointers or references are handled opaquely.

- `wrapper.java` – This module class contains C global values wrapped as static functions in JAVA and must be accessed as such by using the module name in the static function call.

- `wrapperJNI.java` – For every JNI C function there has to be a static native Java function, which appears in this intermediary JNI class. The intermediary JNI class can be tailored through the use of pragmas, but is not commonly done.

This class contains the complete JAVA–C/C++ interface so all function calls go via this class. As this class acts as a go-between for all JNI calls to C/C++ code from the Java proxy classes, type wrapper classes and module class, it is known as the intermediary JNI class. The functions in the intermediary JNI class cannot be accessed outside of its package. Access to them is gained through the module class for globals otherwise the appropriate proxy class.

Our use of **SWIG** in this work is to provide a maintainable Java wrapper for pure JAVA-based and **Processing**-based OpenISS applications.

2.2.7 Docker

Docker tool consists in a set of services that allows its users to packages up code and all its dependencies so the application runs quickly and reliably from one computing environment to another in a standard unit of software called a container. Whereas, a Docker container image is a lightweight, standalone, executable package of software that includes everything needed to run an application: code, runtime, system tools, system libraries and settings.

Container images become containers at runtime and in the case of Docker containers, whereas images become containers when they run on the Docker Engine [118]. We provide a `Dockerfile` in Appendix A that can be used to assemble an image which contains all the required setup and required files to demonstrate our solution. We are releasing in part various test and deployment versions of OpenISS and its dependencies on DockerHub: `https://hub.docker.com/u/openiss`.

2.3 Summary

In this chapter we introduced the reader to the required background in context to this research work in terms of the related work and software tools and how they pertain to our OpenISS gesture framework design and development.

Chapter 3

Methodology

In this chapter, we introduce the reader with our proposed solution, followed by the reasoning that led us to our proposed solution and the various methods that we used to design, develop and evaluate our solution. This also includes our design decisions that were made based on our previous experience with the **ISSv2** and our review of the related work. Then, we provide an outline of the development and experiments in a procedural way. Eventually, we preview an initial concrete design of our solution.

3.1 Solution Overview

We describe a subset of our core requirements that necessarily drive the solution design decisions.

- **Reusability** – As we already know, this work aims to provide a *common* platform that will not only enable two specific yet diverse applications, but also, has the potential for other applications as well, within the context of natural interaction via vision based gesture recognition. Therefore, we require that our solution must rather be a *reusable* solution that can be applied to enable such diverse applications from a natural interaction perspective through vision based hand-gesture interaction as exemplified in our motivational scenarios Section 1.2.3, page 6 and Section 1.2.4, page 8. This solution trait although

not explicitly mentioned in the form of a non-functional requirement, is equally significant as the others.

- **Extensibility** – One of our key requirements for our proposed solution is *extensibility*, see NFR3, essentially, to accommodate newer gesture recognition algorithms, libraries or middleware that may outperform the present ones. This simply means that our solution must provide a simple mechanism to allow such *extensions* easily. Moreover, once such a mechanism is inbuilt within the solution it must naturally be a uniform, standardized and, easy to follow and implement.

- **API Usability** – Yet another detail, mentioned before, is that one of the design goals of the **ISSv2** was to enable digital media artist to create applications via the popular platform **Processing**. This essentially requires an API that is highly abstract yet simple enough to understand by not only artists, easy to use and requires minimal effort to access hand gesture recognition functionality, enabling good API usability (NFR2). This includes abstracting complex tasks away to also reduce errors by users, **for example**, in the **Template Method** design pattern, a method exists solely to provide a *recipe* or skeleton of the process defined in it, and defines the order of calls to different functions. Thus, reducing the chance of messing up the order of function calls by the user.

  At the simplest level, an API that can provide the required gesture recognition functionality in a minimal number of steps would be a good start towards its usability. However, to provide such abstraction that facilitates the user to create applications via an API that is easy to use, our solution must govern the program control flow. This is relevant since the solution must take care of initializing different devices and middleware, which bring gesture recognition functionality to the user application by making corresponding calls to specific methods. The API user should only care about what action can be assigned to specific gestures depending upon the use case. Therefore, we will simply plug application specific code to our solution where it takes care of initializing,

resource allocation and populating the data structures that provide essential data. This is also commonly known as the *Hollywood Principle* or *Inversion of Control* [119].

Now, these specific characteristics discussed above that must be inherent to our solution, certainly lead us to a well known *Software Engineering* concept called **Frameworks**, and, especially, in case of our scenarios, **Application Frameworks**. Now, attributes such as code reuse, extensibility, flexibility, modularity and inversion of control are naturally inherent to frameworks. At this point, we, therefore put forward our decision to provide our solution as a **Gesture Framework** [120]. The overview of this solution described visually can be seen in Figure 23. Now, we will

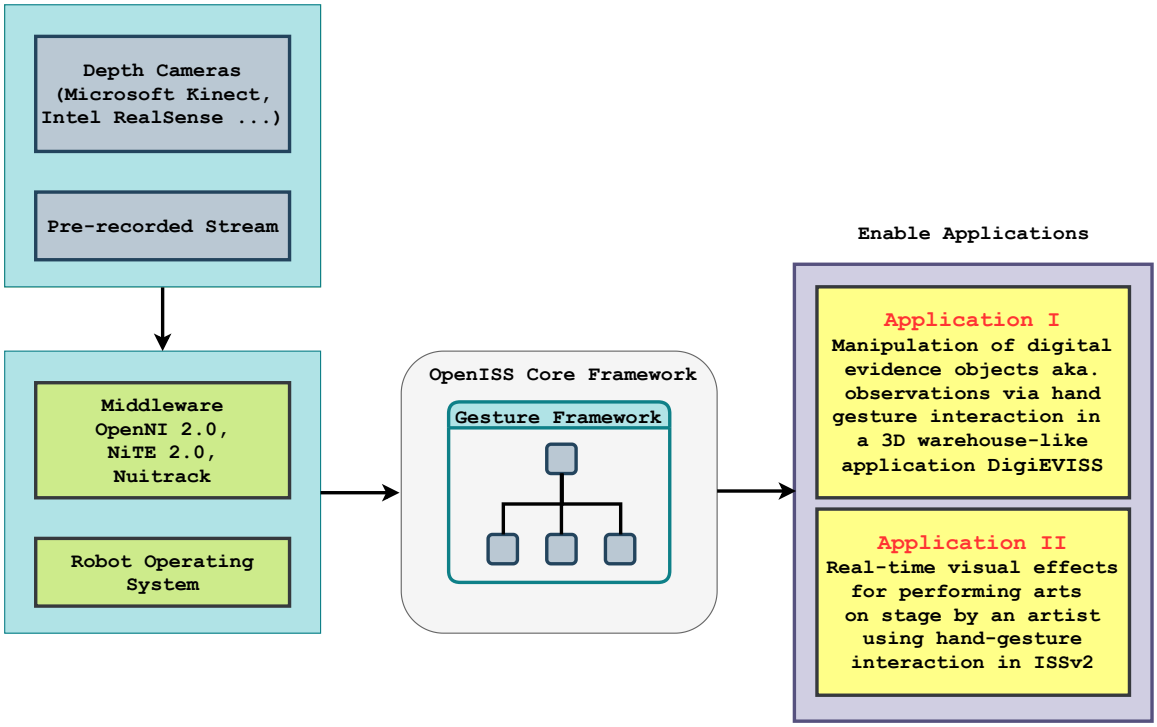


Figure 23: Solution overview

discuss about **Object Oriented Application Frameworks** and how they enable the above mentioned attributes that are desired in our research work listed in detail in Section 1.2.6, page 12.

3.2 Framework Design Approach

Frameworks are *semi-finished* architectures that are reusable and applicable to various domains. They are application generators that enable an array of applications for a specific domain. Generally, the main idea is to describe the framework design and its constituent classes while obscuring the implementation details. This is primarily done to reuse common code across applications to reduce some level of complexity and promote faster prototyping. The run-time architecture of a framework is characterized by an inversion of control that allows the framework, rather than the application, to determine which set of application-specific methods to invoke in response to external events. A framework consists of **Frozen Spots** and **Hot Spots**, coined by Wolfgang Pree in 1994 [121].

- **Frozen Spots** – the fixed components of the framework that define its architecture and the object protocols that govern the control flow are referred to as the *frozen spots*. These spots in the framework are immutable components that define its kernel and are already implemented within the framework that calls and invokes one or more hot spots provided by the implementer. This kernel remains constant and is a fundamental part of each instance of the framework [7].

- **Hot Spots** – a framework demonstrates extensibility and flexibility via extension points, where the adaptation takes place, and are referred to as the *hot spots* of the framework. They are the flexible and generic aspects of a framework that enable specialization through inheritance or composition. A well-designed framework must offer domain-specific hot spots and adequate flexibility for adaptation. They take the form of hook classes and hook methods, which are abstract methods or classes that provide a default empty implementation. Hook methods give a subclass the ability to *hook* into the algorithm as desired. The subclass may or may not use these hooks to adapt to implement a desired behavior. They allow decoupling the stable interfaces and behaviors

of an application domain from the variations required by instantiations of an application in a specific context [7].

Broadly, framework development consists of three stages given below [7], as seen in the Figure 24:

1. Domain Analysis
2. Framework Design
3. Framework Instantiation

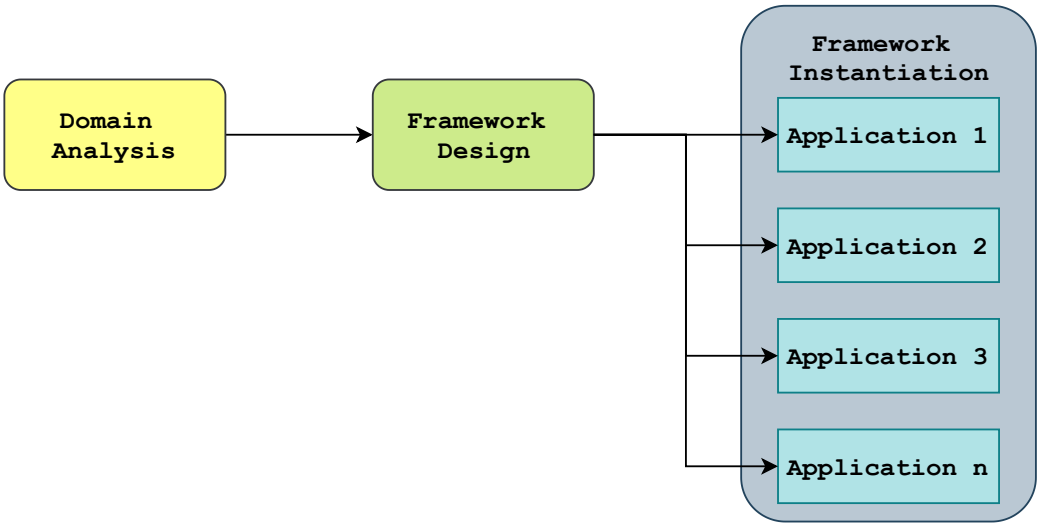


Figure 24: Framework development process [7]

Frameworks themselves are not executable. Rather, they generate executables when instantiated via the implementation of application-specific code for each hot spot by extending them. Also, these hot spots typically are abstract classes or methods that are required to be inherited and overridden to define application-specific behavior. This is normally referred to as the *Dependency Injection Principle* [122]. However, one must know that during domain analysis, the hot spots and frozen spots are partially uncovered and the framework's abstractions are defined. Authors in [7, 123] concluded that hot-spot approach helped in enhancing flexibility and extensibility of the framework. An overview of the gesture framework can be seen in Figure 25.

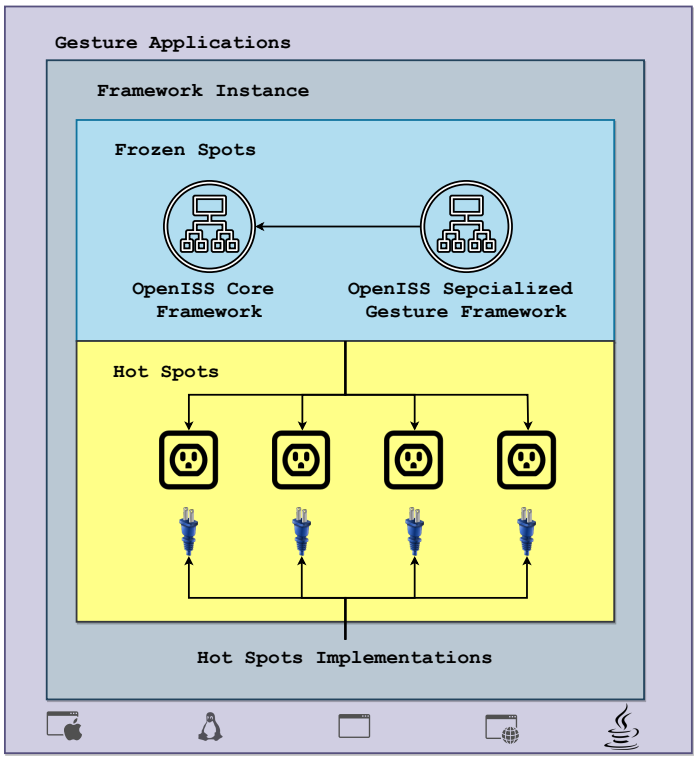


Figure 25: Gesture framework overview [7]

3.2.1 OpenISS Core Framework

The **OpenISS** core framework, a part of which our Gesture Framework will be a specialization of, consists of five modules which represent the primary frozen spots of the framework itself. Most of these modules are a direct abstract mapping to our solution overview shown in Figure 23. Thus, in this work, we provide the **Gesture Framework** that will sit alongside the **person re-identification** [47] by Haotao Lai and the **facial recognition framework** [48] by Yiran Shen, pushing **OpenISS** towards its long-term design goals and as a significant addition to its set of specialized frameworks that enable diverse applications in their respective domains. Next, we describe our core **OpenISS** Framework modules that define the overall kernel of the framework itself.

- **Device Module** – In this module, abstractions are present for depth and color information sources, whether it be a physical device, a pre-recorded stream, or a node publishing similar data, specifically a ROS node. In addition to

the existing abstractions, we provide abstractions for ROS. These abstractions play a vital role in taking care of device specific drivers, device initialization and setting up the device specifics if any, and at the same time, encapsulate away these complexities away from the user and give the end user more of a plug-and-play feel, see NFR6.

- **Common Data Structures** – It is quite obvious that different gesture recognition provider middleware will provide data somewhat differently, essentially in the form of different data types and user-defined types with non-compatible interfaces. This module represents the core data structures of the **OpenISS** core framework as well as the constituent specialized frameworks such as the **Gesture Framework** to decouple the source from the application and at the same time provide interoperability by adapting different data into a uniform data that the framework defines. After this work, the set of data representations will include our representations of a real world `Hand` and `Gesture` alongside `Frame` (depth or color), `Face`, `Skeleton` and `User`, see NFR4.

- **Cross-Language Module** – In this module, we will provide the necessary wrapper code and **SWIG**-related interface files that help generate wrapper classes in JAVA and enable a bridge between C++ and JAVA in order to send essential data from our gesture framework defined types to their corresponding types in JAVA. This will enable interoperability between our gesture framework in C++ and **ISSv2** applications in **Processing** and JAVA, specifically for our scenario mentioned in Section 1.2.4, page 8, also to fulfill our requirements NFR4 and NFR8.

- **Tracker Module** – In this module, abstractions are present primarily for skeleton trackers and face trackers. We will thus provide the gesture tracker abstractions under this module. A tracker factory object can instantiate the desired tracker specific to the application, in our case NiTE2.0, Nuitrack or ROS, and directly fulfill our requirements FR1 and FR2.

- **Viewer Module** – In this module, we provide abstractions for graphics and windowing APIs such as OpenGL, WebGL, Unity, and OpenCV built-in modules that simplify the viewing of frames coming from a source, whether in real-time, over the network or a pre-recorded stream in the end-user applications primarily to visualize the color and depth frames.

3.2.2 OpenISS Gesture Framework

The **OpenISS** specialized Gesture Framework uses the frozen spots of the core by extension. However, the variant components of our solution based on our requirements as seen in the first chapter are namely:

- *Depth data providers*, such as the Microsoft Kinect v1 and Kinect v2, and Intel Real Sense devices, ROS Sever Node, or a pre-recorded dataset.
- *Gesture providers*, such as the NiTE2.0 and Nuitrack middleware.
- *Middleware adapters* for each middleware, essentially for interface and data adaptation.

Now, these variations constitute the various hot spots of our **Gesture Framework** itself. In other words, the **OpenISS** core framework becomes the calling framework, and, our **Gesture Framework** is the one that gets called, and the specialization interface is enabled via the different adapter classes as seen in Figure 26. Also, the relationship between the core framework and the specialized framework is that the latter will take the core as its dependency to define its desired functionality. Now, based on our knowledge so far, we illustrate our domain model in Figure 27. In the next section, we will describe our design and evaluation methodologies.

3.3 Framework Design and Evaluation Methodology

In this section, the reader will first learn about the different methods that we follow in this research work to design, develop and evaluate our gesture framework. Then we

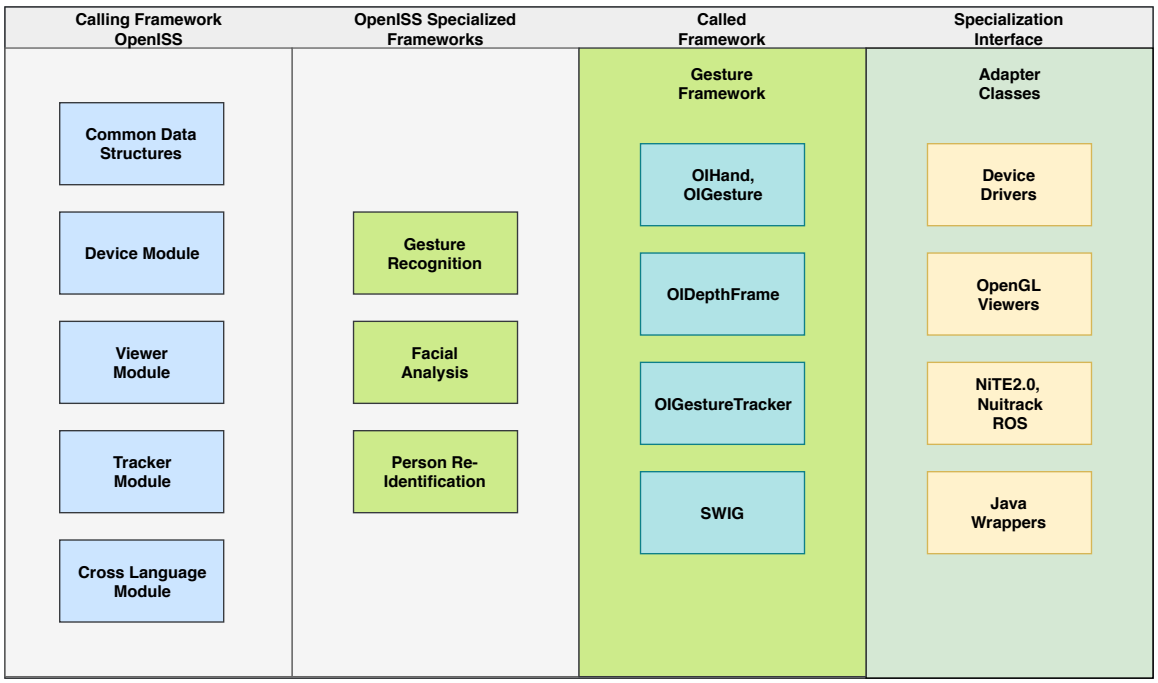


Figure 26: OpenISS core and specialized gesture framework

will enumerate the entire procedure in consecutive steps, for the design and evaluation methodologies respectively. The next two chapters will essentially expand on the technical aspects of the design and implementation, and their evaluation with regards to the requirements.

3.3.1 Top-Down and Bottom-Up Approaches

During domain analysis, one can only partially uncover the frozen and hot spots of the framework itself. In [124] authors have pointed out that although framework development can be slow and unpredictable, the framework design and patterns must be discovered using a *bottom-up* approach. Still, one can leverage existing frameworks to conceive, design and implement high-quality frameworks rapidly. This is among the reasons we choose to build our solution on top of the existing middleware, specifically, NiTE2.0 and Nuitrack. Moreover, in this research work we will employ both top-down and bottom-up design approaches in parallel for specific purposes.

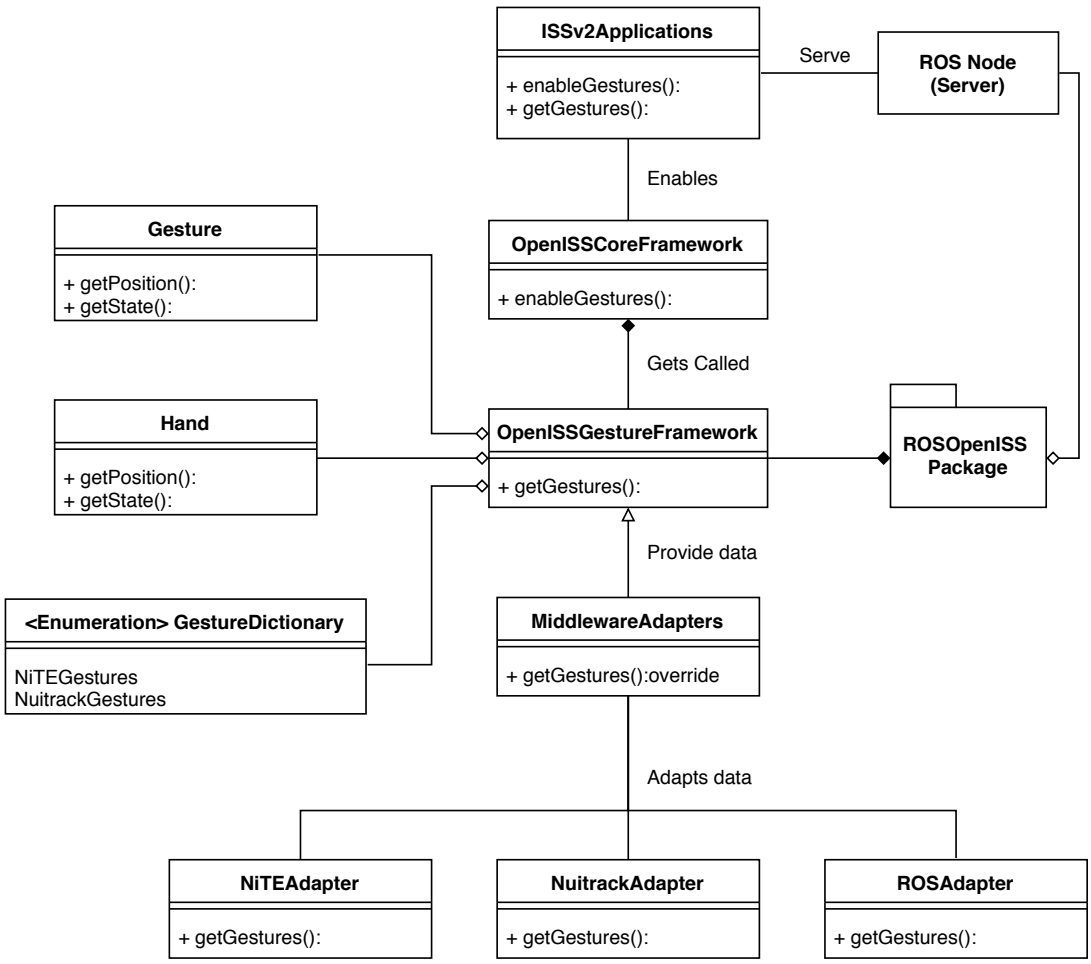


Figure 27: Domain model

Top-Down Framework Analysis and Design

Strictly speaking, the top-down approach to framework design requires complete understanding of the problem at hand, and our problem is to provide real-time hand and gesture data in a flexible and extensible manner to enable the development of our specific applications in Section 1.2.3, page 6 and Section 1.2.4, page 8. Now, beginning with the domain model described in Figure 27, and by factoring in our experience with the previous Illimitable Space System versions alongside the existing middleware solutions, namely NiTE2.0 and Nuitrack with their respective resources such as online documentation and sample applications, we identify the necessary information to infer and define an abstract design of our **Gesture Framework**.

Moreover, one of the previously mentioned **ISSv2** *backends* is the `SimpleOpenNI`

backend, which is essentially a JAVA wrapper implementation of OpenNI2.0 and NiTE2.0 middleware, which solely provides gesture recognition in the **ISSv2** presently, thus, bolstering our decision to choose NiTE2.0 first, which also facilitates the comparison with existing applications. On the other hand, Nuitrack provides more gestures and supports multiple devices, which easily becomes a suitable choice in contrast to NiTE2.0 as well as our requirements, especially for multiple devices (cf. NFR6). However, this essentially means that we will require a intermediate interface to bridge the gap between the interfaces of our framework's API with the API of these specific middleware.

The framework development and design patterns often go hand in hand and specific problems of the frameworks are usually solved using one or more of the design patterns that best suit the use case under consideration [125]. Since we are dealing with incompatible interfaces, undoubtedly, the **Adapter Pattern** exists to solve the very exact problem of adaptation between different interfaces. The top-down approach also supports in designing of our API. For instance, we certainly need API calls for our end user applications such as, `getGestures()`, `getHands()` and `getDepth()` which, by the way, broadly resolve to our first three functional requirements (FR1, FR2, and FR3). Now, these function must exactly do what their names describe, that is get depth and/or detected and recognized gestures and/or tracked hands to the end user simply by making calls to these functions. Similarly, we also proceed by abstraction to fulfill the need to contain this information in corresponding data structures. Likewise, the top-down approach will keep us on track when it comes to fulfilling the rest of the remaining requirements.

Bottom-Up Framework Analysis and Design

A bottom-up approach promotes early implementation with smaller, individual components that are gradually composed together into one complex system using abstraction. It is only when a framework is used to build applications that its lack of generality might show up. To find its weaknesses in applicability to different situations, it must be designed and reused for different applications. Moreover,

an integral part of any library, middleware or framework is its documentation, which enables other developers to use it by referring to the details provided in the documentation about the architecture, classes, types, etc., and do something with the functionality they provide. Both NiTE2.0 and Nuitrack provide documentation and sample code to demonstrate how these middleware can be used. Therefore, we must do the same for our gesture framework as well. We will reuse these samples from NiTE2.0 and Nuitrack into our own single universal sample application that must work with every instance of our **Gesture Framework** and demonstrate similar behavior across these framework instances.

Thus, our sample application will serve as the qualitative analysis testbed that will enable us to track our progress towards our goals and visualize and validate our results. This includes the early discovery of the weaknesses of the framework itself and removing them successively [126]. Broadly, the idea is to first encapsulate NiTE2.0, followed by Nuitrack and eventually ROS as gesture providers via our solution framework and compare the behavior of the sample application using the different middleware. Both these above-mentioned approaches enable us to define an entire procedure that we will follow throughout this research work to develop our gesture framework and the functionality and the applications it enables for the **ISSv2**. In the coming section, we will discuss the different evaluation methods that we will use to evaluate our requirements and demonstrate the same capability through experiments when possible.

3.3.2 Design and Implementation

In this section, we put forward the methodology steps that we follow to develop the detailed design and implementation of our gesture framework:

1. We start with defining our abstract classes that will define the overall kernel of the gesture framework based upon our domain model in Figure 27. These classes essentially are the identified frozen spots for the gesture framework with the required *hook* methods that constitute the identified hot spots for

our gesture framework. These *hook* methods can be overridden to provide our extensibility requirement (see NFR3) for the hot spot that is susceptible to change, in particular the gesture provider hot spot, which is an essential requirement for the gesture framework itself to be flexible and extensible.

2. Further, we define the required data structures to enable our interoperability requirement (see NFR4) and uniformity for our gesture framework. This is essentially to provide uniform data to our framework applications since the underlying middleware NiTE2.0 and Nuitrack have some differences in the way they provide hand, gesture and depth information. Similarly, prospective gesture providers may provide data differently, which must comply with our gesture framework so as to not break existing code by complying with a newly integrated middleware.

3. Next, we create our integrated sample application for our gesture framework based on the insights from the sample applications of NiTE2.0 and Nuitrack. The sample application will necessarily serve two purposes: first, following the bottom-up design approach we will demonstrate and test individual functionalities iteratively with our sample application. Secondly, this sample application will eventually serve as a resource alongside the framework documentation for the end users of **OpenISS** as an SDK, to use it effectively.

4. At this point, we must encapsulate NiTE2.0 and Nuitrack middleware to provide the required real-time gesture recognition and hand-tracking functionality for our **Gesture Framework**. Essentially, we require adapter classes for each of these middleware to perform the intermediate adaptation of depth, gesture and hand data to populate our gesture framework's data structures and further provide the data to the applications. Thus, the integrated sample application that uses an instantiation of our gesture framework gesture provider hot spot extensions, either NiTE2.0 or Nuitrack must exhibit similar or identical behavior when either of these *extensions* are used as gesture providers for the application.

5. Next, after we have a baseline working **Gesture Framework**, we start building our FORENSIC LUCID visual interaction application **DigiEVISS** against this baseline framework to create the application as per the usage scenario mentioned in Section 1.2.3, page 6. This process further supports our solution framework from a developer's perspective. This essentially means that not only this process will bring certain current shortcomings of the API to the surface, but also provide an insight in to streamlining the process of application development using our framework.

6. After our review on ROS in Chapter 2, we found out that at the time of writing this thesis no such ROS package was available that provides real-time hand-gesture detection, recognition and hand tracking functionality. However, ROS packages have existed for long enough that do provide access to raw video data (depth and color frames) for devices such as the Microsoft Kinect. Moreover, various ROS packages exist for OpenNI2.0 and NiTE2.0 and even one or two for Nuitrack but, they all focus on providing *skeleton tracking* only instead.

   Therefore, we rather create a ROS package version of our **Gesture Framework** and *publish* real-time gesture detection and recognition data via the package and create a corresponding ROS adapter client class to *subscribe* to the publisher and obtain necessary data for adaptation between the ROS package and the **OpenISS** gesture framework itself and eventually made available for the sample application. In this scenario, our framework as a ROS package is the server that provide gesture, depth and hand information over the ROS network stack and the application is the one that instantiates the gesture framework ROS adapter client that can subscribe to this data.

7. Next, we create an interface file for the **SWIG** tool to generate wrapper classes for our gesture framework in JAVA so as to enable the performing arts application in **ISSv2** itself. We then bundle these classes as a JAR file to simply place as is in the **Processing** [26] environment. We then create a backend class specifically for our gesture framework within the **ISSv2** and move on to refactor

existing application only to instantiate our backend and replace pre-existing ones, such as `SimpleOpenNI` in the **ISSv2** and analyze the behavior.

3.3.3 Evaluation

In this section, we define the methods we incorporate to evaluate our work and compliance with the requirements. This includes quantifiable and non-quantifiable properties of our gesture framework strictly corresponding to our functional and non-functional requirements. Within the context of this research we employ both *quantitative and qualitative evaluation* followed by the experimentation we conduct.

3.3.3.1 Qualitative Evaluation

Broadly speaking, in qualitative analysis we test the non-quantifiable properties of our **Gesture Framework** *mainly* through observations and visual feedback via the previously mentioned integrated sample application as a testbed for our requirements evaluation. Essentially, the sample application will not only enable us to visually track our progress, but also to identify problems, improve on design limitations and essential refactoring that can be addressed early and successively across iterations. Moreover, to answer our research questions in Section 1.5, page 24, it is mandatory to demonstrate and evaluate our gesture framework by providing working application prototypes conforming to their respective scenarios mentioned in Section 1.2.2, page 6. Thus, creating two separate and specific application prototypes strictly in the context to our scenarios using our proposed gesture framework that will be a proof-of-concept that we can indeed design a common solution for both these applications and enable comparisons among hot spot extensions. However, we will focus on the feasibility on **enabling** such applications via a common platform by overcoming the challenges in this research work.

Then we compare how our hot spot extensions namely NiTE2.0, Nuitrack and ROS affect the behavior of our applications. We essentially want to compare these **adapter backends** against our applications and make some useful inferences in terms

of desired functionalities of a solution for these applications. We compare these backends on their gesture dictionaries, visual responsiveness, and whether or not the gesture count matters? If yes, then what is the minimum number of gestures that are required to enable both these applications in their complete form?

Note that usability evaluation with an actual investigator or even formal API usability evaluation involving artists is beyond the scope of this thesis.

3.3.3.2 Quantitative Evaluation

Common run-time performance for interactive graphical applications is measured in frames-per-second (FPS) i.e., how many frames the graphical application is able to render to enable smooth appearance of rendered animated graphical imagery. Modern day games typically aim for **60FPS** or higher, where the highest FPS possible is desirable. However, the render frame rate is affected by various factors such as the type of visual effects that are used, transparency, realistic lighting, shadows etc., applied on the objects in the scene as well as the geometry of the said objects or in graphics jargon "polygon count" and the final resolution of the rasterized image. If the objects in the environment are dynamic and their topology changes due to deformation and interactions, it may affect the frame rate. This affect is directly proportional to the amount of objects, interaction and special effects there are and if frames-per-second fall below 24, the end-users may experience what is known as *lag or jitter*. This rate in the reality is the only one metric we consider in our evaluation as it is the most user-perceptable. However, there are other things that may affect our specific application types:

- **Sensor rate:** The rate at which the sensor devices give out sensor data such as the Kinect provides **30FPS** or **15FPS** whereas Intel RealSense D435 provides **60FPS** with better resolution than the former. However, this rate is naturally independent of the render FPS as mentioned above, but may affect the interactivity in the end-user application, which may still render at high FPS, but the interaction may appear to be lagging if the sensor data delivery

rate is significantly slower. We can call this effect of sensor FPS on render FPS as **perceived FPS** by the end user in reference to our real-time requirements. Nonetheless, the two rates, render FPS and sensor FPS are otherwise fairly decoupled.

- **Sensor data processing:** Now, processing of sensor data is another aspect that may adversely affect the **perceived FPS**. A part of the processing takes place in the middleware such as segmentation of hands or skeleton joints and related transformations, e.g., using real-time physics engines for dynamic scene or VFX generation. However, this relies on sensor data delivery rate, and any additional overhead, may slow down the overall appearance of the generated perceived *and* rendered FPS by the application.

To begin with quantifiable evaluation, we consider the render frame rate as well as the perceived frame rate where necessary and their ratio for each and every instantiation of the gesture framework, which entirely constitutes the four stages mentioned in NFR1 namely:

- Motion or Gesture Capture
- Motion or Gesture Interpretation
- Visual Effects Generation
- Visual Effects Rendering

It is crucial to evaluate whether or not the framework delivers desired functionality in real-time or not. A frame rate of **15FPS** or more would be a preferred rate for our applications to be considered real-time while **10FPS** as acceptable for motion perception. Moreover, as we will use ROS to provide the service aspect of our solution simply by publishing data over the ROS network stack and the allowing the application to subscribe to it. However, a bearable latency [127] is still expected when it comes to adding multiple abstraction layers on top of one another. Thus, when it comes to evaluating whether our solution is interoperable with middleware

like ROS we want to calculate this overhead introduced to the process by ROS. This includes recording the frame rate yet again, but over the ROS network and finding out the overhead in terms of FPS as compared to the framework abstractions alone. To be acceptable, FPS needs to be minimally **10FPS** so that it can be perceived as motion and we are expecting an overhead. However, as long as the frame rate doesn't drop below the acceptable threshold, the overhead is considered bearable. Moreover, the peer-to-peer nature of ROS's architecture forces us to decouple the data publisher and subsriber client application, thus, it is quite natural that even if the publisher is not publishing any data the subscriber application can still render at **60FPS** or more. This might affect the evaluation of the overhead introduced by ROS.

To address this issue, we further instrument capturing callbacks-per-second on both publisher and subscriber ends. To further explain this reader must know that, middleware such as NiTE2.0 and Nuitrack have callback mechanisms that exist to enable asynchronus processing where specific callback methods respond to certain events such as availability of a new frame from the sensor device. Moreover, Nuitrack provides even more similar callbacks for other events such as introduction of a new user in the frame and so forth. Thus, we will count the number of times these callbacks methods are called, which corresponds to the sensor device frame rate and if a new frame is not available usually the previous one is reused. Thus, it will be quite helpful in case of ROS where we essentially want to count the overhead introduced to the sensor FPS as delivered to the client over the ROS network stack. So, we define our metric **effective FPS** as the average of the sum of render FPS and sensor FPS as delivered to the client application divided by 2 to measure the overhead of different middleware on the **perceived FPS** by the user and the reaction time. However, this assumes we include the sensor data related processing in this average.

$$effectiveFPS = (sFPS + rFPS)/2 \tag{1}$$

We can summarize our different requirements under qualitative and quantitative analyses as seen in Table 3.

Table 3: Requirements evaluation classification

Requirements	**Qualitative**	**Quantitative**
Real-Time		X
Device Adaptability	X	
API Usability	X	
Cross-Platform	X	
Cross-Language	X	
Extensibility	X	
Interoperability		X
Structural Scalability	X	

3.3.3.3 Experimental Evaluation

To evaluate our framework in practice, we put forward the following experiments that enable a qualitative or quantitative analysis of the gesture framework itself:

1. Earlier, we mentioned about our integrated sample application. Now, we use it to first obtain the depth frames, and then gestures and eventually hands via our gesture framework. Then, we test the application by switching between NiTE2.0 and Nuitrack gesture provider instances of our gesture framework and note down the analysis and key points and make corrections if necessary.

2. Then, once we have built the digital evidence interaction application **DigiEVISS** for FORENSIC LUCID investigations, we compare both NiTE2.0 and Nuitrack gesture provider instances to make inferences and note down the key aspects of their comparisons over the same application in terms of gesture dictionary sizes and whether or not we meet our functional requirements for this scenario. For minimal API usability evaluation, we release the initial basic application with only two evidential items, to be extended to four and two observation sequences and observe them completing the task and integrate their modifications back into the main application.

3. Subsequently, we produce the essential JAVA wrappers for our framework instances, which we initially test using our integrated sample application custom built for **Processing** for a qualitative evaluation. Then only we test them with our existing visual effects pipeline in the **ISSv2** and ensure

the proper functionality and data transfer between the two language layers. This is where we also test the real-time aspect of the **Gesture Framework**, firstly, qualitatively via the visual feedback compared between the existing `SimpleOpenNI` backend compared to our **OpenISS** backend and in terms of render FPS, and, secondly, comparing the frame rate with our minimum threshold for real-time live performances on stage recorded on average through the same sequence of visual effects present in **ISSv2**.

4. Having successfully built our applications conforming to our scenarios, and done evaluating them on various aspects, we can move on to evaluate our integrated sample application as a client to our framework delivered as a ROS package over its network stack. The idea is to calculate the overhead introduced by ROS and its effect on the real-time aspect of the application. Specifically here, we compare the callbacks-per-second on the server side that is the ROS package providing gesture data vs the integrated sample application on the client side consuming that data. This means counting the total number of frames produced and published by the publisher/server vs the total number of frames received by the subscriber/client.

5. Last but not least, we choose a different unrelated test application to demonstrate how easily we are able to plug our solution in as a library to this test application and describe our experience in replacing existing interaction functionality with hand gestures. For instance, this can easily be a 2D game that requires keyboard direction keys to control and play the game, for which we can replace the key strokes with hand gestures for direction control, made available via our gesture framework. At this point we perform a qualitative API usability evaluation against applicable Nielsen's heuristics for usability evaluation.

6. After having done all these evaluations we should be able to make concrete comparisons and suggest our experiences as a set of guidelines and recommendations for these specific application types to application developers along with the future work that can be done to extend and enhance the

capabilities of these applications and our own framework.

3.4 Summary

In this chapter, we gave the reader an overview of the various approaches we follow to achieve our end goals and fulfill our various functional and non-functional requirements in a procedural and systematic fashion. In the next chapter we will further expand on these steps to design and develop our framework and how we can instantiate our framework.

Chapter 4

Framework Design and Instantiation

In this chapter we provide the reader with the detailed design and implementation-specific details of our gesture framework, its constituent classes, and its Application Programming Interface (API). Now, we mentioned in the previous chapter that there are three main stages of framework development, **domain analysis, framework design, and framework instantiation**. In the previous chapter, we primarily focused on the domain analysis for the framework solution and define a domain model. Then, we put forward an action plan that described our steps to build our **Gesture Framework** and perform the qualitative and quantitative analysis, the findings of which we will discuss about in the next chapter. However, in this chapter we describe our concrete framework design along with UML class diagrams and its different instantiations namely, NiTE2.0, Nuitrack and ROS. In the last section and its subsections we describe the look and feel of the integrated sample application and realizations of our two different application scenarios as described in the first chapter.

4.1 Framework Design

The key aspect of any framework is that it is essentially a reusable design for a specific class of problems that can be customized via creating application specific subclasses of a parent abstract class of the framework. Not only the framework describes the architecture of the application, it calls application code, instead of

having the application call specific APIs to implement a specific behavior [7]. Moving on, we will start with providing an overview on how the data will flow from its source and propagate through our data structures to the end user application as seen in the Figure 28.

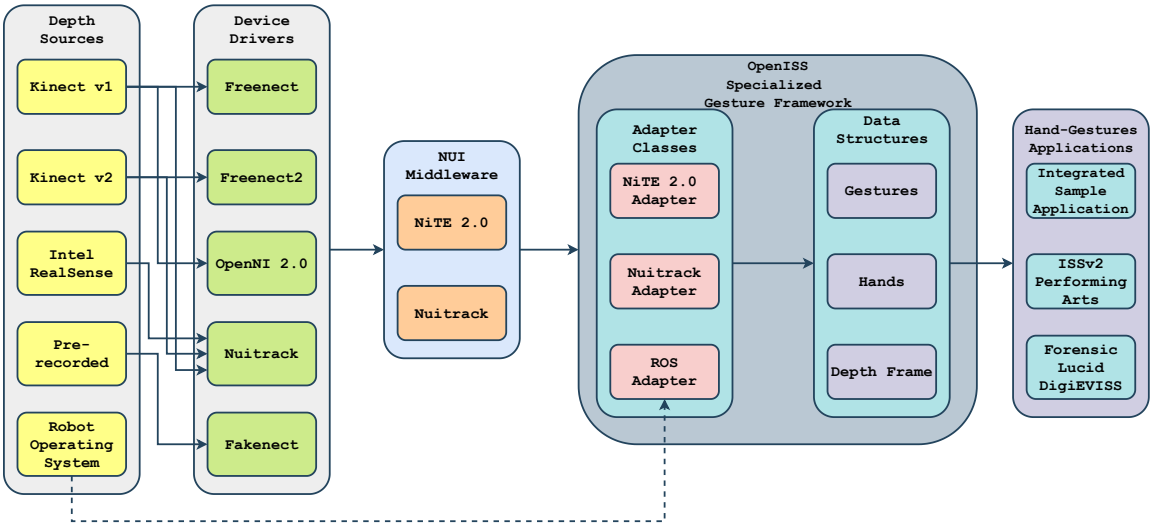


Figure 28: Data flow from device to application

In our solution, the **OpenISS** Specialized Gesture Framework, we can broadly classify the flow of control into three different stages: **data acquisition**, **data adaptation**, and **data delivery**, roughly following the general high-level design principles in Figure 6, page 32.

4.1.1 Data Acquisition

In this phase, we acquire the data, essentially, the depth frames that contain crucial information that can be extracted via algorithms provided by NiTE2.0 and Nuitrack middleware and related solutions to classify gestures, track hands, track users and so forth. Now the *source*, of these gesture data providing depth frames can be:

- A depth sensing device like the Microsoft Kinect or the Intel RealSense that can provide depth and color streams along with related meta information.
- A pre-recorded video stream from a depth sensing device such as the Microsoft Kinect using the `libfreenect`'s library **Fakenect** or `libfreenect2`'s

`streamer_recorder`, which emulate a real Kinect device and plays a prerecorded stream on a continuous loop. A similar tool exists for RealSense.

- A ROS node that can publish depth data in the form of ROS messages over ROS topics that can be subscribed and get data over the ROS network stack.

The data acquisition phase requires different device drivers for different devices usually provided either by the device manufacturers or community driven open source drivers. These drivers bridge the gap between the device and the operating system and then this depth data can be further passed on to the NiTE2.0 and Nuitrack middleware. Such middleware contain hand-gesture recognition algorithms which process the aforementioned depth data and extract relevant information such as detected and recognized gestures, hand tracking information, and associated meta information such as a hand ID, gesture type, and their respective states.

However, before we go further, we would like to briefly talk about the role of ROS in the implementation context. It is quite obvious to expect a delay related to marshalling and de-marshalling of data back and forth between different systems that are ROS and **OpenISS** over the ROS network stack. As mentioned before at the time of writing this thesis there were no such ROS packages available that can provide real-time gesture recognition and hand tracking functionalities. Now, to address this, ideally we would create ROS packages for NiTE2.0 and Nuitrack only to publish their respective data and subscribe to this data via **OpenISS** adapters for each of these and instantiate within the client application that is the integrated sample application. However, we instead decided to write a ROS package instead for our **OpenISS** gesture framework which so far can still provide NiTE2.0 and Nuitrack data but via our own API.

Not only can this be passed over to the ROS community as a extensible and flexible gesture provider (first of its own kind) within the ROS package ecosystem but also enables us to calculate the accumulative overhead of or framework abstractions and ROS. One can speculate that incase our framework does not pose a significant overhead it won't affect the overall overhead introduced by ROS alone. Lastly, in

simple words, whatever functionality and data the **OpenISS** gesture framework can provide the **OpenISS** ROS package must provide as well.

4.1.2 Data Adaptation

In the data adaptation phase the adapter classes enable adaptation of data between the different interfaces of the underlying middleware with the **OpenISS** specialized gesture framework API via intermediate or common data representations. This is accomplished by the aforementioned **Adapter Design Pattern** [125], where the framework instance adapts an existing object to a new use-context by means of an intermediate **Adapter object**. A Client uses the Adapter operations only, and the Adapter implements them in terms of the domain functionality of the **Adaptee** [128] as seen in Figure 29. Thus, the idea is to create an object of the gesture provider (here NiTE2.0 or Nuitrack) and simply copy the data directly from its data members to our data structures. Once our data structures are populated, they can then uniformly provide the essential data to a set of related hand-gesture applications including our own realizations of application scenarios mentioned before in the Chapter 1, Section 1.2.3, page 6 and Section 1.2.4, page 8.

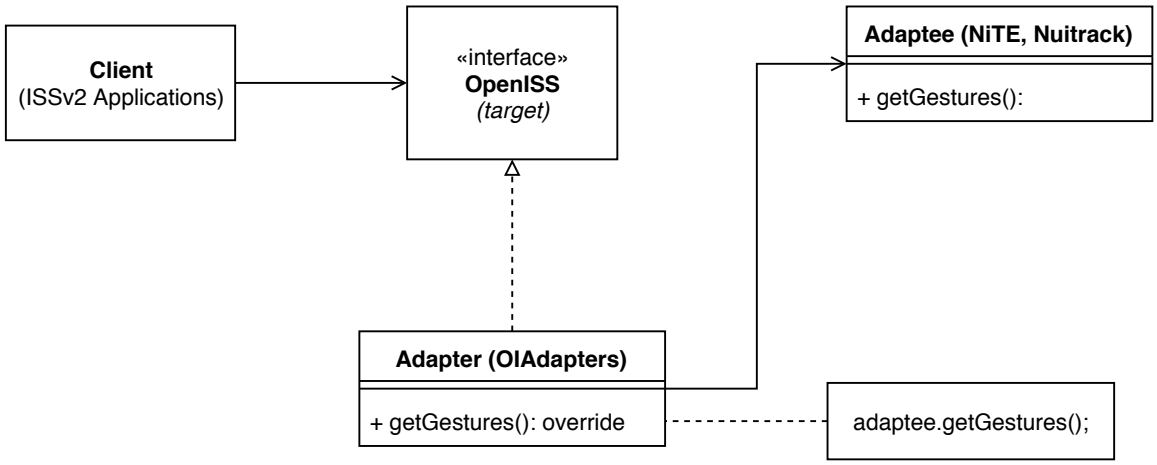


Figure 29: Adapter object adaptation via Composition

4.1.3 Data Delivery

After the data adaptation stage, comes the last stage, that is the data delivery stage where an application can choose one of the framework instantiations and access relevant adapted data directly from the **OpenISS** data structures coming from the underlying gesture provider middleware such as NiTE2.0 and Nuitrack but via the **OpenISS** API. The data are available directly via our simpler API that abstracts away mundane tasks such as device initialization and getting the relevant gesture and hand data associated with some form of action in the end user application. The data are only accessible through the framework's API and is governed by the gesture framework itself. An overview of the **OpenISS** Gesture Framework API can be seen in Table 4. In simple terms, the collection of gestures and hands in every depth frame are available as a `std::vector` (C++ Standard Template Library) of their respective data representations in **OpenISS**.

Table 4: OpenISS gesture framework API overview

API Call	**Functionality**
`init()`	Initialize devices, middleware and objects construction.
`startGestureDetection()`	Start the gesture detection functionality.
`stopGestureDetection()`	Stop the gesture detection functionality.
`update()`	Call all the callbacks and update the data structures with most recent data.
`startHandTracking()`	Start the hand tracking functionality.
`stopHandTracking()`	Stop the hand tracking functionality.
`getGestures()`	Get the collection of gestures and related data in the recent past frame.
`getHands()`	Get the collection of hands and related data in the recent past frame.
`stop()`	Release allocations or resources including the depth devices, destroy objects.

4.1.4 Design Structure and Layers

The entirety of the solution provided in this work is not merely centric to the sole gesture framework itself. In Figure 30 we describe the different layers that are an integral part of our solution stack. These different layers pass data from bottom to top along with providing concrete details of the components within each layer. Starting from the bottom-most layer we will next describe the role that different layers play to realize the solution.

- **Hardware Layer** – This layer constitutes the depth sensing devices such as

the Kinect v1, Kinect v2 and Intel RealSense devices including the **Fakenect** library from OpenKinect that comes with the **Freenect** library and emulates a real device running a pre-recorded stream on a continuous loop.

- **Device Driver Layer** – This layer contains different device drivers that enable access to device data captured by the devices in the hardware layer. The drivers facilitate the transfer of data from the device to the operating system which can further be processed by algorithms available via middleware like NiTE2.0 and Nuitrack.

- **Middleware Layer** – In this layer, essentially, gesture provider middleware such as NiTE2.0 and Nuitrack reside. Although, Intel RealSense's middleware `librealsense2` itself does not provide any gestures yet, it provides wrappers for OpenNI2.0; thus, eventually being compatible with NiTE2.0 also by extension. This layer is responsible for extraction of useful information from the depth frames received via the device driver layer and perform functionality like gesture classification.

- **Framework SDK Layer** – This layer constitutes the entirety of the gesture framework along with the core framework including essential data representations of crucial data such as depth frame, hands, gestures and so forth. Data processed by the middleware is adapted into our gesture framework data representations and available via our API to the end user application.

- **Application Layer** – The top most layer contains the different instantiations of the framework itself including the sample application and the ones mentioned in Section 1.2.2, page 6.

In Table 5 we describe the structure of the work itself where, often, code is grouped into directories or packages to manage and group different classes that work with one another. In this work, directories like `src/` and `include/` contain all the class definitions and declarations of the Gesture Framework including data structures,

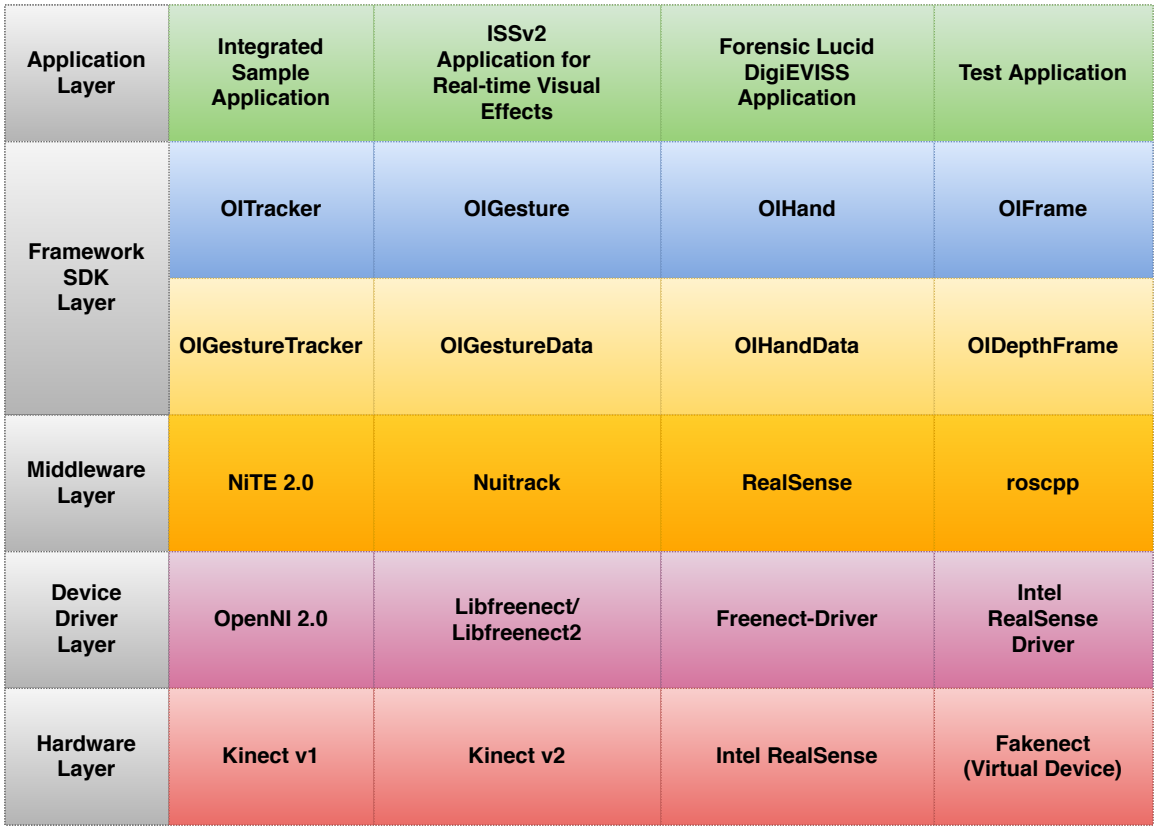


Figure 30: Layered architecture

gesture enumerations, API definition, class definitions, adapter classes and so forth. Under the `samples/` directory we have grouped all the applications that were enabled using the **OpenISS** gesture framework, including the integrated sample application, and the applications realizing the scenarios mentioned in Section 1.2.2 and a test application with the sole purpose to demonstrate the ease to plug our solution as a library to an existing application and replace its interaction mode with hand gestures. When providing a generic solution, especially, in this work there are lot of complex configurations that can be easily managed by build systems like `cmake`. Thus, the `cmake-modules/` directory contains essential configuration files, that help lookup and linking to of different dependencies that are required by a generic solution. Further, the `swig/` directory contains interface files and related `cmake` configurations to produce JAVA wrappers for our gesture framework and produce a JAR file and related compiled code packaged as libraries, thus, essentially, cross-language modules.

Table 5: OpenISS gesture framework concrete implementation structure

Directory	Constitutes
src/	All the C++ source files, class definitions etc.
include/	All the C++ header files, declarations etc.
samples/	End user applications built using the framework itself.
cmake-modules/	Custom cmake scripts for looking up and linking with dependencies.
swig/	Interface files for the framework classes for Java JNI wrapper code.
CMakeLists.txt	Top level cmake-compliant setup file.

Further, in Table 6, we describe the **OpenISS** gesture framework solution but packaged as a ROS-package-compliant structure built on top of the roscpp, ROS's C++ API. Just like the project structure mentioned above for gesture framework, the directories src/ and include/ contain essential class definitions for the ROS *Node* (here the **OpenISS** gesture framework) and the msg/ directory contains the data representations of gesture and hand collections that are published on the ROS **OpenISS** package custom define topics for **OpenISS** itself. In the cmake-modules/ directory exist various scripts that enable looking up and linkage to with various dependencies of the node itself. Whereas, package.xml defines properties about the package such as its name, version number, author, dependencies on other catkin packages and so forth. The reader may recall from Chapter 2, Section 2.2.4, page 50, that this package will reside in the catkin_ws, that is, the *catkin workspace*.

Table 6: ROS OpenISS package

Directory	Constitutes
src/	All the C++ source files, class definitions, using roscpp, C++ API.
include/	All the C++ header files, declarations etc.
msg/	Custom ROS publisher messages, essentially, **OpenISS** framework data.
cmake-modules/	Custom cmake modules for building the package with external dependencies.
CMakeLists.txt	Top level cmake-compliant setup file.
package.xml	Package manifest file.

As partially described before in Table 4, now, the complete concrete design of the interface and its API, where, OIGestureTracker is the abstract class, that defines the base kernel of the gesture framework, can be seen as an UML class diagram in Figure 31.

In C++, one can define an abstract class by making one or more member functions, *pure virtual* methods using the virtual keyword right in front and

assigning it to zero, as seen in Figure 31. Then these *pure virtual* functions can be overridden in different subclasses or concrete classes in other words. This is simply polymorphic behavior, also known as dynamic dispatch mechanism, where, we access the subclass member methods via a pointer or reference of its superclass as seen in Figure 29, page 83.

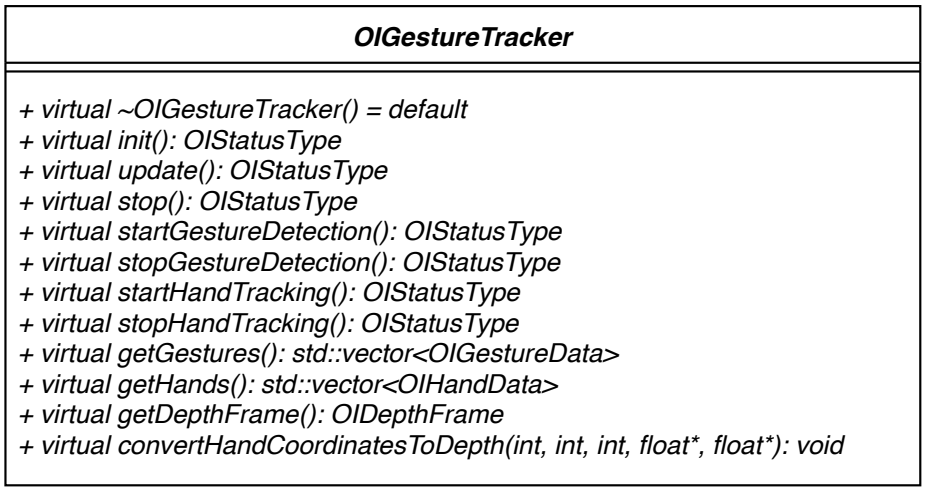


Figure 31: `OIGestureTracker` UML class diagram

Thus, in framework development terms, these concrete classes are the points of variation or so-called hot spot implementations, that can be polymorphically switched. The bare-bones framework kernel can be seen in the UML class diagram presented in Figure 32 and constitutes the frozen spots of our gesture framework. Classes `OIGestureData`, `OIHandData`, and, `OIDepthFrame` are representations of a real world gesture, hand and depth frame as seen by the depth sensor, respectively.

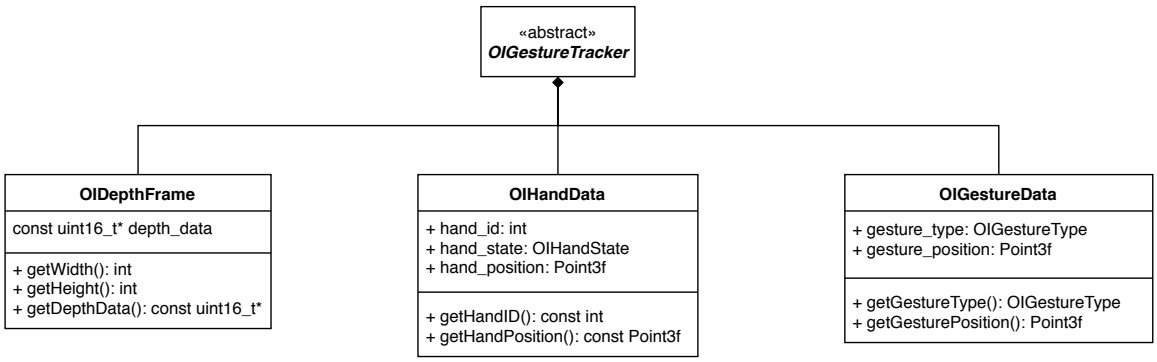


Figure 32: OpenISS gesture framework kernel in UML

Although, Table 4 provides an overview of the different API calls in a few words, we will further expand on their functionality from the perspective of defining desired behaviors via these methods in the various framework instantiations.

- `init()` – In this method, all the initializations such as device drivers, gesture recognition, and hand tracker modules, especially those from NiTE2.0 and Nuitrack, are done. The same goes for the **roscpp**, ROS's C++ API, which is initialized here as well.

- `stop()` – As the name suggests, in this method, we stop the functionality such as gesture recognition or hand tracking, release held resources, and destroy objects of gesture recognition and hand tracker modules created in the constructor and initialized in the `init()` function.

- `update()` – This method is responsible for providing the most recent data as seen by the sensor device as well as the information extracted by the middleware by managing calls to callback methods that respond to events generated by gesture recognition or hand tracking modules. Moreover, both NiTE2.0 and Nuitrack provide callback mechanisms, but differently, so as to create a non-blocking application and even delegate control to windowing libraries such as based on OpenGL.

- `startGestureDetection()` – As the name suggests, we simply summon the underlying middleware objects to commence the process of gesture recognition or detection in real time with appropriate feedback to the user. It internally initializes the algorithms that perform gesture classification and extract useful information from the incoming depth frames. Both NiTE2.0 and Nuitrack are closed-source, therefore, we cannot really comment on the kind of algorithms they are using to do so.

- `stopGestureDetection()` – As opposed to the above, this method exists to stop the process of gesture recognition or hand tracking on the incoming depth

frames. This is highly desirable to manage application resources since gesture classification algorithms can be resource-extensive.

- `startHandTracking()` – In this method, we call the underlying middleware objects to start the process of hand tracking, if any. This essentially means that algorithms within the middleware will normally first recognize a hand and then assign it an identity value to differentiate between multiple hands or users and then track it until such information is no more available due to some reason such as the hand is out of the *field of view* of the sensor itself. Now, in case of NiTE2.0, one must first perform a gesture and once it is detected the coordinates are used as a starting point to start tracking the hands. On the other hand, in Nuitrack once the user itself is tracked successfully, hands are recognized and tracked by default.

- `stopHandTracking()` – In this method, we stop tracking hands that are currently being tracked in real-time. It is desirable to do so in case a hand is no longer required to be tracked.

- `getGestures()` – This method adapts the gesture data from the middleware's data structures and populates our own, and then return the collection of recognized gestures as a `std::vector` of type `OIGestureData`. It is crucial to make a call to `update()` before in order to get the recognized gestures extracted from the most recent depth frame.

- `getHands()` – Similarly here, we obtain the hand data from the middleware's data structures and populate our own and return the collection of hands recognized or tracked as a `std::vector` of type `OIHandData`. Just like with `getGestures()`, it is crucial to make a call to `update()` before in order to get the currently tracked hands from the most recent depth frame.

- `getDepthFrame()` – In the first chapter, we established that for gesture recognition and hand tracking depth data are crucial and adequate as well.

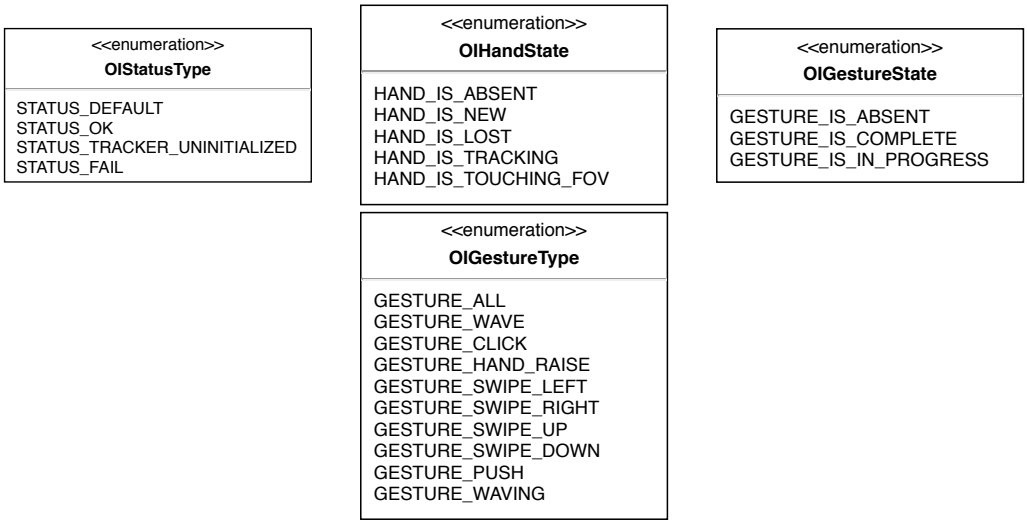


Figure 33: OpenISS gesture framework enumerations

By providing the depth frame exclusively we explore the possibility of further extending the gesture dictionaries, preferably, by object composition.

- `convertHandCoordinatesToDepth()` – The coordinates that we obtain from a depth sensors are usually real world coordinates, which need to be converted into projective coordinates in order to display those 3D points on a 2D space especially for our applications. However, with some precise measurements one can compute the conversion factor for various devices. A concrete application would be projection mapping by calculating camera intrinsic and extrinsic parameters.

The different kinds of enumerations defined for our gesture framework can be seen in Figure 33. Further, we will elaborate on these enumerations and data that they represent.

- `OIStatusType` – It is extremely desirable to get hold of the feedback that whether or not a method executed successfully, and if not, what is the error or warning that can point the user in a direction towards solving the problem and making things work as they are expected to do so. This directly supports two of Nielsen's usability heuristics [32] **visibility of the system status** and **help**

users recognize, diagnose and recover from errors. In this particular enumeration, we have four values, namely:

1. `STATUS_DEFAULT` – It represents a default status value, or a value for initialization itself.
2. `STATUS_OK` – It represents a successful execution of the function.
3. `STATUS_TRACKER_UNINITIALZED` – A special value that reports that the underlying NiTE2.0 or Nuitrack tracker object was not initialized.
4. `STATUS_FAIL` – It represents that a method or an operation has failed to execute.

- `OIGestureType` – This class constitutes to what we refer to as our **gesture dictionary**, which currently is an aggregation of gestures from NiTE2.0 and Nuitrack middleware. However, it is crucial to keep prospective additions to this dictionary simple and efficient. For now we have the following enumerations of different gesture types, namely:

  1. `GESTURE_DEFAULT` – Represents the default value for a gesture object's type which essentially means an empty gesture object. It is generally a good programming practice to provide such initial values for user defined types.
  2. `GESTURE_WAVE` – The simple action of waving a hand is represented by this value.
  3. `GESTURE_CLICK` – This value represents the click gesture with hand, which can be thought of as pressing a large invisible button with your palm.
  4. `GESTURE_HAND_RAISE` – This value represents the hand raise gesture.
  5. `GESTURE_SWIPE_LEFT` – This value represents the hand swipe gesture to the left.
  6. `GESTURE_SWIPE_RIGHT` – This value represents the hand swipe gesture to the right.

7. `GESTURE_SWIPE_UP` – This value represents the hand swipe gesture in upwards direction.
8. `GESTURE_SWIPE_DOWN` – This value represents the hand swipe gesture in downwards direction.
9. `GESTURE_PUSH` – This value is similar to the one above that is, `GESTURE_CLICK`, where the only difference is that this belongs to Nuitrack middleware.
10. `GESTURE_WAVING` – This value is similar to `GESTURE_WAVE`, but, belongs to Nuitrack whereas the other comes from the NiTE2.0 middleware.

- `OIHandState` – This enumeration represents different states a `OIHandData` object, which is our representation of a real world hand, can have at a given instant in time. These states are crucial to create sophisticated applications based on our experience with this work. One can leverage these states to create effective interaction between the user and the application.

  1. `HAND_IS_ABSENT` – This value indicates that there is no hand present in the most recent depth frame received.
  2. `HAND_IS_NEW` – This value indicates the presence of a newly tracked hand in the most recent depth frame received.
  3. `HAND_IS_LOST` – This value indicates the a previously tracked hand is lost due to some reason.
  4. `HAND_IS_TRACKING` – This value indicates that the hand is currently being tracked.
  5. `HAND_IS_TOUCHING_FOV` – This value indicates that the hand is currently on the boundary of the sensor's *field of view*.

- `OIGestureState` – This enumeration represents different states of a `OIGestureData` object, which is our representation of a real world hand gesture.

1. `GESTURE_IS_ABSENT` – This value indicates that there is no gesture detected in the most recent depth frame received.
2. `GESTURE_IS_COMPLETE` – This value indicates that the detected gesture is complete.
3. `GESTURE_IS_IN_PROGRESS` – This value indicates that the gesture is currently in progress.

Now that we have laid out the definitions of the baseline framework structure and important classes, we can see the different instantiations in the UML representation shown in Figure 34. This essentially means that these three instantiations are the subclasses that provide specific behavior to the methods of the superclass as per the underlying middleware. However, the kernel will remain the same, and still the control flow will be governed by the framework itself in all of its instantiations. In the next section, we will further elaborate on these different instantiations and eventually our different applications built using our gesture framework API.

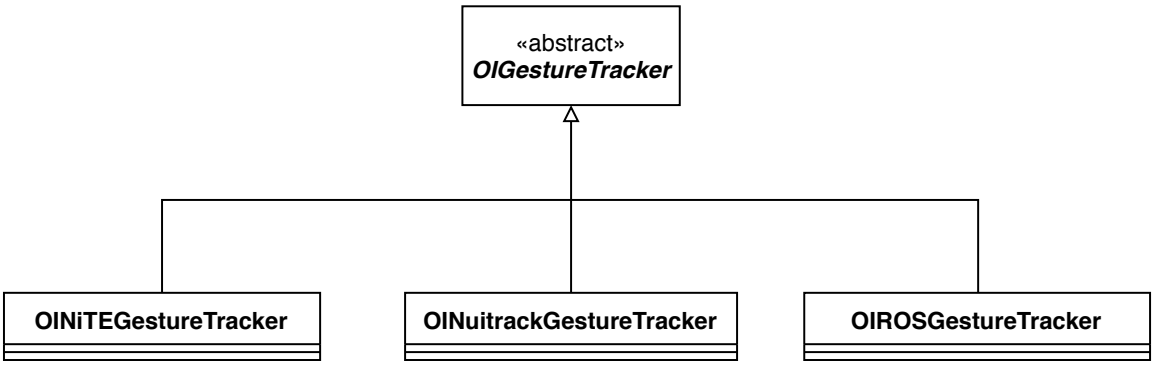


Figure 34: OpenISS gesture framework class Hierarchy

4.2 Framework Instantiations

In this section, we will elaborate on the hot spot implementations provided on top of the NiTE2.0 and Nuitrack middleware and then corresponding adapters for ROS to publish the same data over its network stack. Now, as seen previously in Figure 29, in C++, the instantiations can take the following forms as seen in Listing 4.1.

```
openiss::OIGestureTracker* gesture_tracker = new openiss::OINiTEGestureTracker;
openiss::OIGestureTracker* gesture_tracker = new openiss::OINuitrackGestureTracker;
openiss::OIGestureTracker* gesture_tracker = new openiss::OIROSGestureTracker;
```

Listing 4.1: OpenISS Gesture Framework Polymorphism

This is a classic polymorphism example, where the class `OIGestureTracker` defines the API, but is unaware of which particular instantiation will provide the functionality.

4.2.1 NiTE2.0

In this section, we shed light on how we leverage the NiTE2.0 middleware to be one of the many (including prospective middleware) gesture provider to our framework. We then uniformly populate the data from its API into our much simpler API provided by our gesture framework. The `OINiTEGestureTracker` class can be referred to as a derived class or a subclass of the abstract class `OIGetsureTracker`. The sole purpose of this class is to adapt between the two different interfaces, that is, NiTE2.0 middleware's gesture recognition and hand tracking API and our own framework's API. At the same time, this class also takes care of populating the essential data structures required by the end user applications. As NiTE2.0 has a rather small gesture dictionary of only three gestures, namely:

1. `GESTURE_WAVE`
2. `GESTURE_CLICK`
3. `GESTURE_HAND_RAISE`

Now, limited gestures as low as three in case of NiTE2.0 can be restricting for applications requiring complex interactions. However, it is still adequate for testing our gesture framework against existing visual effects applications in **ISSv2** because a few of our existing **ISSv2** applications previously used `SimpleOpenNI` [37], which is essentially a JAVA wrapper of OpenNI2.0 and NiTE2.0, for gesture interaction. Thus, this enables us to compare our gesture framework as a backend with the existing

ISSv2 `SimpleOpenNI` backend. By default, NiTE2.0 provides gesture detection and hand tracking through the `HandTracker` and `HandTrackerFrameRef` classes, where the class `NewFrameListener` reacts to events generated by the `HandTracker` class. To use this class, one must derive a class from it that implements the `onNewFrame()` method as seen in Figure 35. This is the method that will be called when an event is generated.

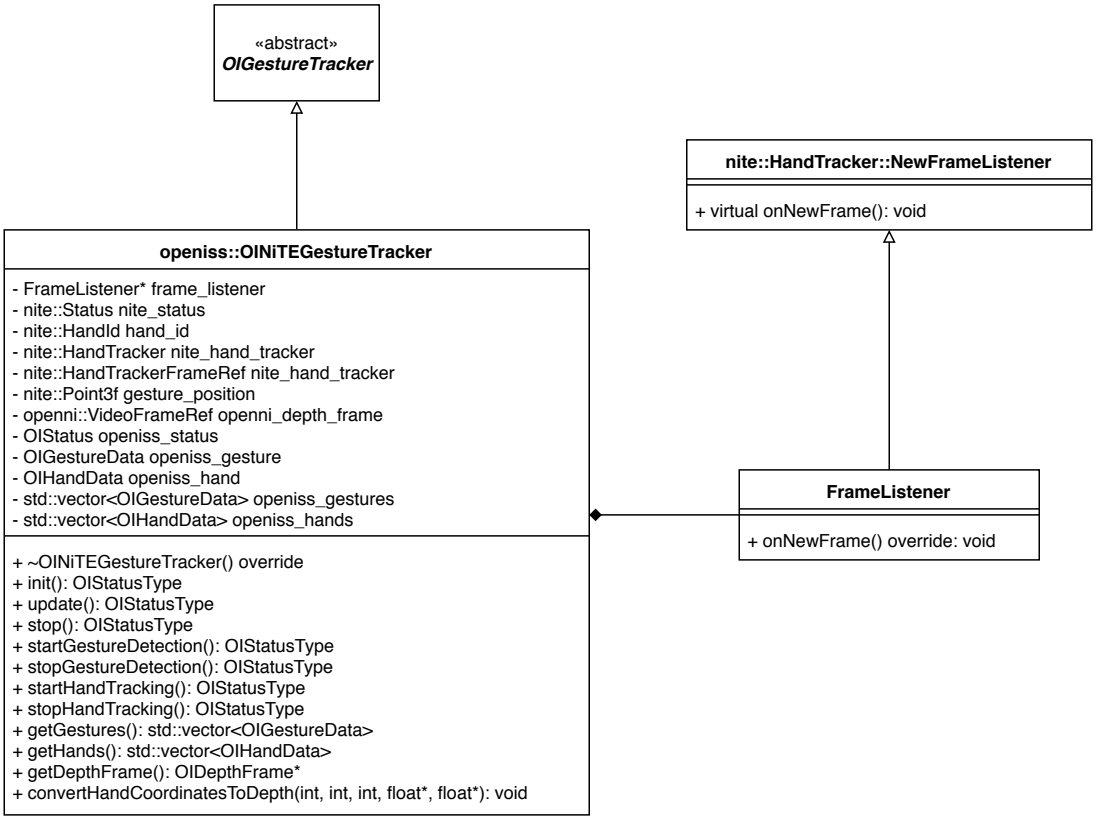


Figure 35: NiTE2.0 instantiation in UML for OpenISS gesture provider hot spot adapter

We will now further expand on the procedure followed to build this hot spot implementation adapter.

1. We started by inheriting the class `OIGestureTracker` publicly in our class `OINiTEGestureTracker`. Next, we state all the methods as `override`, including constructor and destructor. This promotes **RAII** (*Resource Acquisition is Initialization*) to manage object lifetimes through constructors and destructors. Therefore, resources like the depth device can be released

```
nite::HandTracker hand_tracker;
nite::hand_tracker.startGestureDetection(nite::GESTURE_CLICK);
nite::hand_tracker.startGestureDetection(nite::GESTURE_WAVE);
nite::hand_tracker.startGestureDetection(nite::GESTURE_HAND_RAISE);
```

Listing 4.2: Start gesture detection NiTE2.0

effectively once it is no longer required or the object holding the device has completed its lifetime. However, this is more of a general programming practice strictly in case of C++, so all our class definitions leverage this concept for efficient object lifetime management. This may seem like a redundant detail, but it is of utmost importance. Now, the next step would be to define these overridden methods to attain desired functionality.

2. We start with the method `init()` and initialize NiTE2.0 in here including the initialization of the device either using `OIDevice` or by default taken care of by the middleware itself. To initialize NiTE2.0 we call `nite::NiTE::initialize()` within this method. Subsequently, `stop()` to destruct or release the device and other related resources by making a call to `nite::NiTE::shutdown()`.

3. A peculiar detail of the NiTE2.0 API is that to start tracking a hand or hands one must perform a NiTE2.0 recognized gesture with the very hand that must be tracked. Additionally, the gesture type must be mentioned that the tracker object should expect the user to perform. Henceforth, in the method `startGestureDetection()` we initiate gesture detection for all three gesture types provided by NiTE2.0, see Listing 4.2.

4. The method `onNewFrame()`, as mentioned previously, also see Figure 35, is responsible for providing the most recent data, as part of the design of the NiTE2.0 middleware; therefore, the adaptation of data must take place in this method. Now, in NiTE2.0, the two main functionalities of class `nite::HandTracker` are recognizing hand gestures and tracking hands. Whereas, `nite::HandTrackerFrameRef` represents a frame of data from

`nite::HandTracker`, which contains the currently recognized gestures and hands in the scene. Additionally, `nite::HandTracker` is able to track hands independent of the user's body.

This means by creating instances of these classes, one can obtain the gestures and hands information via the class methods and data members through these instances and simply copy the information over into our data structures and making it available by methods `getGestures()` and `getHands()`, which are essentially our two functional requirements FR1 and FR2. For our another functional requirement FR3, we access it via `openni::VideoFrameRef::getData()`, which returns a pointer with the undefined data type `(void*)` to the first pixel of the frame. It is important to note that the reader might wonder about the `update()` method, which is expected to update the data every time a frame is received essentially, an array of pixels which can be of type depth, color, but in this case we simply return the control back to the program. This means that although the methods are overridden, based on specific use cases they may or may not be defined in the subclasses. However, for depth pixels received we convert it into an array of depth pixels of type `uint16_t` and provide it via our API. This also enables serializing some desired frames if required. Thus, essentially we are wrapping underlying middleware NiTE2.0 functionality within our API calls.

4.2.2 Nuitrack

The `OINuitrackGestureTracker` class is a derived class or a subclass of the abstract class `OIGetsureTracker`, just like the `OINiTEGestureTracker` earlier. In this class, we bridge between the Nuitrack middleware's gesture recognition and hand tracking API with our own framework's API. At the same time, this class also takes care of populating the essential data structures required by the end user applications. Moreover, Nuitrack has a gesture dictionary of six gestures, namely:

1. `GESTURE_WAVING`

```
// Create required Nuitrack modules instances for data access
depth_sensor = tdv::nuitrack::DepthSensor::create();
gesture_recognizer = tdv::nuitrack::GestureRecognizer::create();
hand_tracker = tdv::nuitrack::HandTracker::create();
user_tracker = tdv::nuitrack::UserTracker::create();
```

Listing 4.3: Essential Nuitrack modules instances

2. `GESTURE_SWIPE_LEFT`
3. `GESTURE_SWIPE_RIGHT`
4. `GESTURE_SWIPE_UP`
5. `GESTURE_SWIPE_DOWN`
6. `GESTURE_PUSH`

This provides the end user a broader scope of actions that can be associated with gestures. However, there is a tradeoff: this isn't directly proportional, that is, more gestures does not necessarily mean more usability via mapping to diverse actions, rather it can have an effect on the learning curve and recalling all of them might be difficult; thus, negatively impacting usability. Unlike NiTE2.0, we don't need to explicitly start hand tracking in Nuitrack rather once a user is tracked successfully, hands are tracked automatically.

1. Just like `OINiTEGestureTracker` we started by inheriting class `OIGestureTracker` publicly in class `OINuitrackGestureTracker` marking all the methods with the keyword `override`.

2. Next, we start with the method `init()` and wrap Nuitrack's own initialization method `tdv::nuitrack::Nuitrack::init()`. After, we create instances of four essential modules using their respective `create()` methods, namely:

   which provides access to the various callback methods (see Listing 4.4) defined by the Nuitrack API, that respond to corresponding events and provide the most recent data including depth frames as seen in Figure 36.

```
// Various Nuitrack callback functions
void onNewDepthFrame(tdv::nuitrack::DepthFrame::Ptr depth_frame);
void onNewGestures(tdv::nuitrack::GestureData::Ptr gesture_data);
void onHandUpdate(tdv::nuitrack::HandTrackerData::Ptr hand_data);
void onUserUpdate(tdv::nuitrack::UserFrame::Ptr user_frame);
void onUserStateChange(tdv::nuitrack::UserStateData::Ptr user_state);
void onNewUser(int user_id);
void onLostUser(int user_id);
```

Listing 4.4: Nuitrack callbacks providing essential data

Then the callback methods provide access to the most recent information extracted from the depth frame including the frame itself captured by the depth sensing device. Since these methods are providing the essential data, the adaptation itself takes place within these individual methods that populate our data structures. Additionally, it is important to make a call to the `tdv::nuitrack::Nuitrack::run()` to start processing the data provided by the sensor asynchronously. Subsequently, in our method `update()`, we must call `tdv::nuitrack::Nuitrack::update()`, to request recent data from all created Nuitrack modules mentioned above, and all the callback functions are called to fetch the most recent data.

3. Finally, in the method `stop()` we release allocations and destroy module instances created prior and initialized in `init()` to free resources such as the depth sensing device.

4.2.3 ROS

As we mentioned prior, we provide the **OpenISS** gesture framework as a ROS package that is the first of its kind to provide a flexible solution for gesture recognition applications where essential data can be published and subscribed to over ROS's network stack. Yet, another reason was to be able to uniformly evaluate ROS vs non-ROS applications for both NiTE2.0 and Nuitrack middleware. However, most importantly, this is to fulfill our requirements, NFR4, NFR5, and NFR6. The first step was to adapt between the **OpenISS** Gesture framework API and `roscpp` API. ROS provides `Message`s, which essentially are data structures that are `roscpp`-compliant.

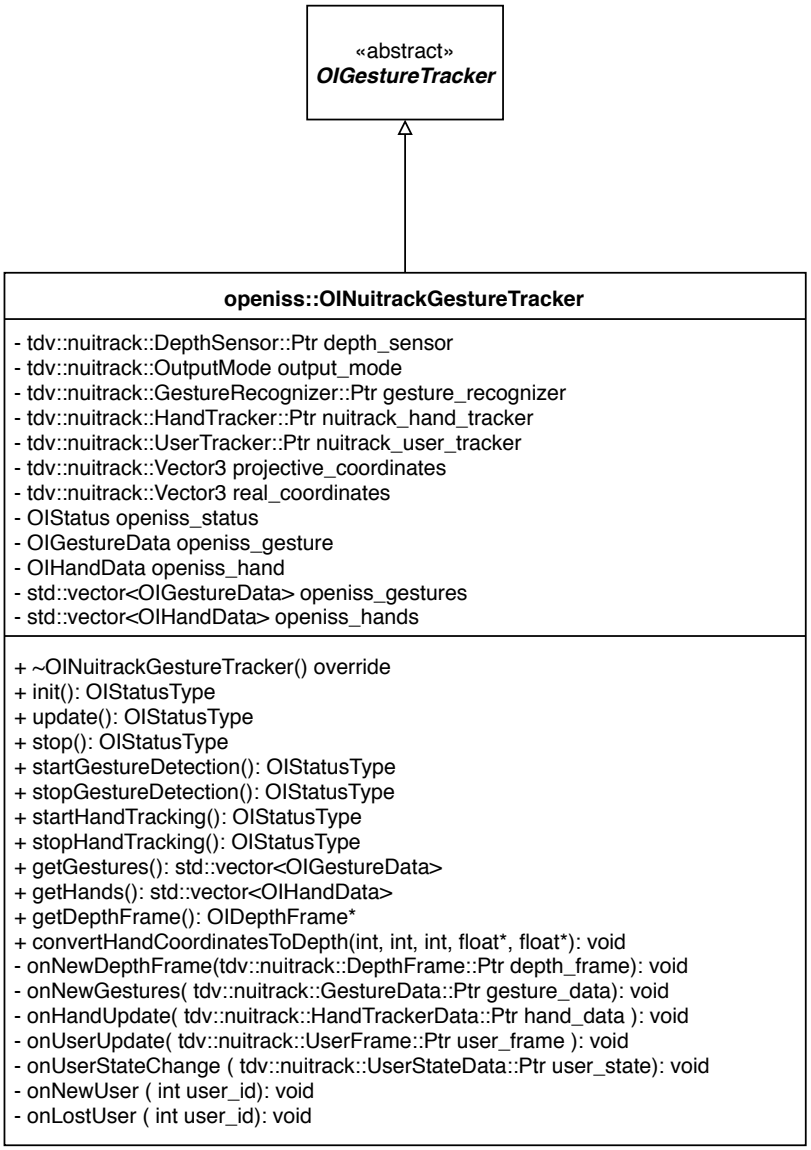


Figure 36: Nuitrack instantiation in UML for OpenISS gesture provider hot spot adapter

```
// Gesture.msg
uint16 gesture_type
geometry_msgs/Point gesture_position
```

Listing 4.5: OpenISS ROS message (gesture)

For instance, for our data types `OIGetsureData`, `OIHandData` and `OIDepthFrame` as seen in Figure 32 an equivalent of all in `roscpp` messages can be seen in Listing 4.5, Listing 4.6 and Listing 4.7 respectively.

```
// Hand.msg
uint16 hand_id
bool hand_state
geometry_msgs/Point hand_position
```

Listing 4.6: OpenISS ROS message (hand)

```
// DepthFrame.msg
uint16[] depth_data
uint16 frame_width
uint16 frame_height
uint16 frame_id
time frame_timestamp
```

Listing 4.7: OpenISS ROS message (depth frame)

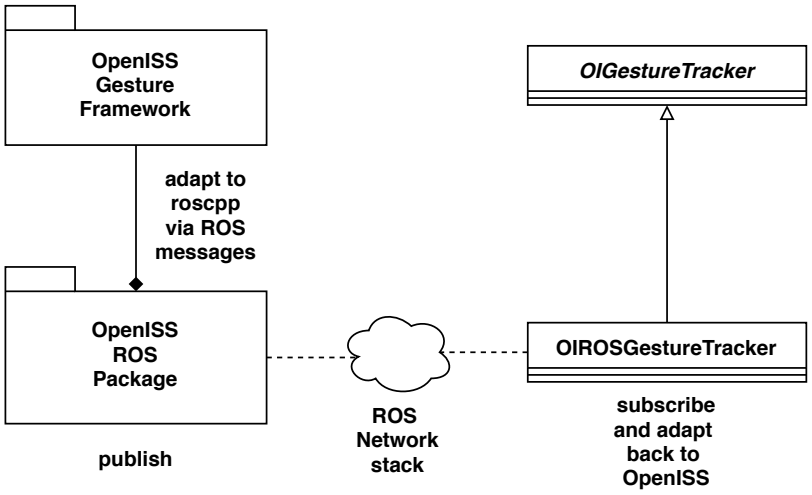


Figure 37: ROS adaptation on publisher/subscriber ends

However, this adaptation can be referred to as the server-side adaptation, that is adaptation from **OpenISS** data to ROS data but, on the other side, which is the **OpenISS** framework itself, requires yet another adapter to adapt data between `roscpp` back to the **OpenISS** API as seen in Figure 37. Moving on, these messages now can be published over custom defined topics for our **OpenISS** ROS package as seen in Figure 38 where `ROSAdapter` is the adapter class that subscribes to the published data and adapts it back to the **OpenISS** gesture framework API. On a side note, ROS's `catkin` relies heavily on `cmake` and contains numerous `cmake` macros that enable package management, dependencies management, linking to essential libraries and so forth.

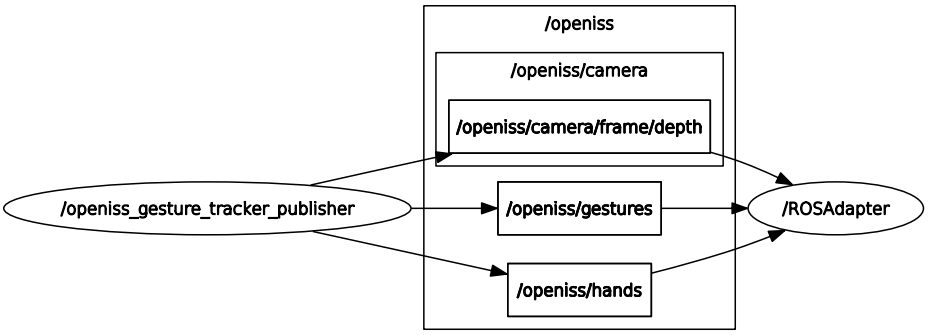


Figure 38: ROS OpenISS node and topics graph

After defining the corresponding data structures, they are populated from the **OpenISS** API, and published using the `ros::Publisher` class from the *roscpp* API, and should always be created via a call to `NodeHandle::advertise()`. Essentially, we publish collection of gestures, tracked hands that can have different status, depth frame including individual gesture and hand data as seen in Figure 39.

On the other end, we have various `ros::Subscriber` class instances, that subscribe to the aforementioned publishers on their individual topics as seen in Figure 40. Essential data, that is depth, gestures and hands are adapted to the **OpenISS** API and populate its data structures and made available to the applications via methods namely, `adaptDepthData()`, `adaptGestureData()` and `adaptHandData()` respectively also can be seen in Figure 40.

To summarize, one can easily notice a trend that can be generalized for prospective solution say X that can be leveraged via our framework by creating suitable adapters enabling extensibility, see NFR3, and intermediate data representations for interoperability, see NFR4:

1. Publicly inherit the abstract class `OIGestureTracker` and then override and implement its various methods depending on the use case.

2. Adapt the interface between **OpenISS** API and X's API via a suitable design pattern such as the Adapter Object Pattern as seen previously in Figure 29.

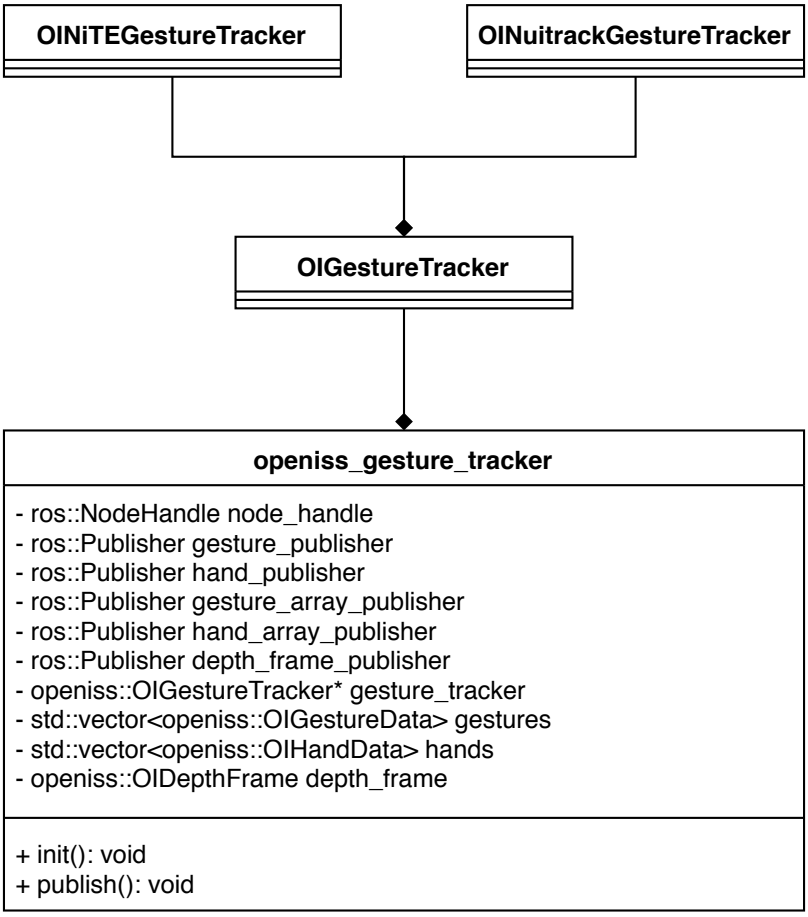


Figure 39: Publisher: **OpenISS** ROS NiTE2.0 and Nuitrack instantiations

3. Create applications that can use one or the other form of gesture recognition and hand tracking for natural interaction using hands.

4.2.4 Framework Cross Language Module

One of our non-functional requirements from Chapter 1 is **Cross-Language**, see NFR8, and, in Chapter 2 we talked about a mature tool named **SWIG**, which stands for Simplified Wrapper and Interface Generator. Now, **SWIG** can generate JNI (JAVA Native Interface) code from C/C++ code by creating interface files for corresponding C++ classes that must be wrapped into JNI code. This essentially means a JAVA program can call into C/C++ code from JAVA. For real-time requirements, see NFR1, we chose C++ as the primary implementation language; however, as mentioned prior

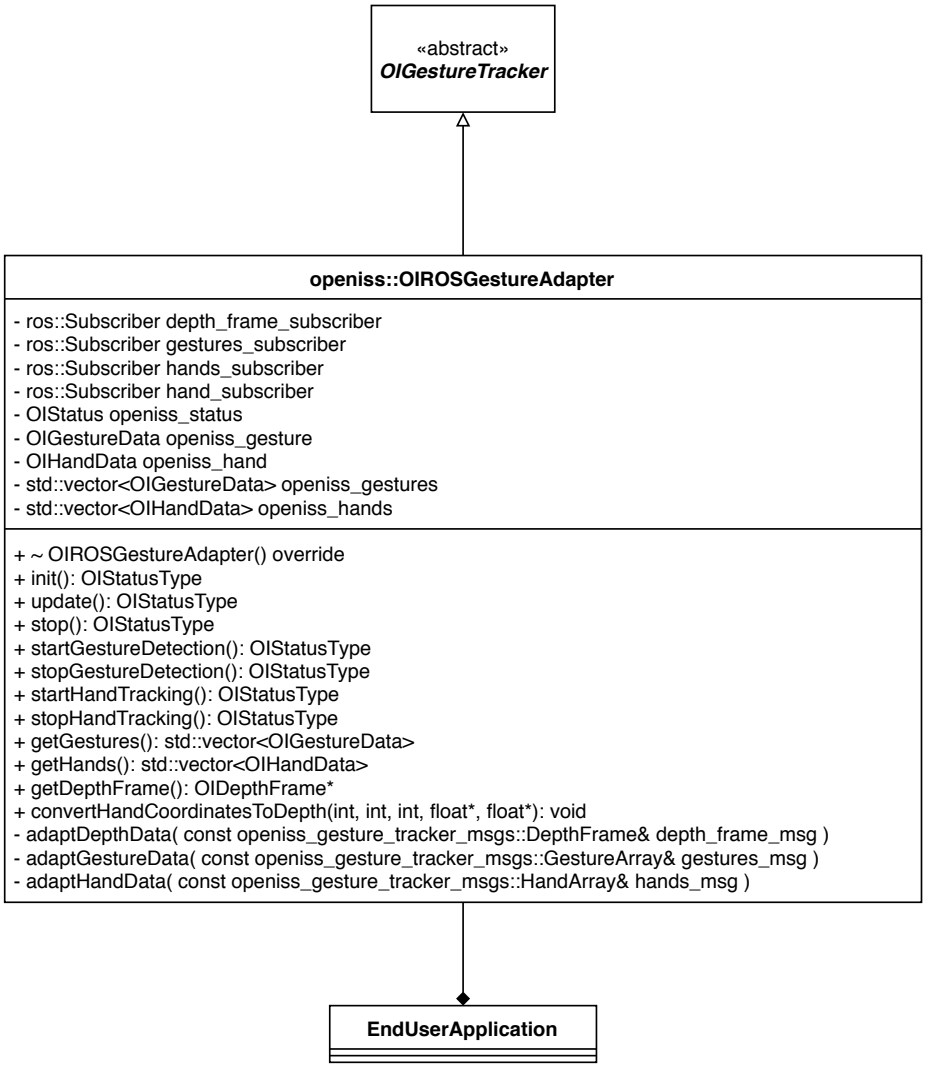


Figure 40: Subscriber: applications instantiating NiTE2.0 or Nuitrack

ISSv2 is JAVA-based. Thus, we leverage **SWIG** to generate such JNI code which we bundled as a JAR file that can be placed into **Processing**'s libraries directory. The main challenge is to define suitable interface files and **SWIG** `typemaps`, which enable a mapping between native and user defined types in C++ code and corresponding types in JAVA which can be created to exchange data back and forth.

The most essential data that is required for **ISSv2** application for our scenario (see Section 1.2.4, page 8), is the depth frames, and all the recognized gestures and hands within those frames see Figure 32. Since we are using the `std::vector` type from C++ STL, it was relatively easy to wrap using **SWIG** library itself. It provides an

```
// SWIG library for std::vector
%include "std_vector.i"

// Typemaps OpenISS types to corresponding Java types
%template(GesturesVector) std::vector<openiss::OIGestureData>;
%template(HandsVector) std::vector<openiss::OIHandData>;

...

%include "OIGestureTracker.h"
%include cpointer.i
```

Listing 4.8: **SWIG** interface file for `OIGestureTracker`

interface for the `std::vector` type named `std_vector.i` that can be included in the interface file for the desired C++ class that uses `std::vector` type. Therefore, we can define an interface file, see Listing 4.8 for the abstract class `OIGestureTracker`.

The next challenge is the conversion of our class `OIDepthFrame`, for which we leverage the `typemaps.i` library from **SWIG** itself. Now, the reader can see in Listing A.1 various kinds of typemaps such as `jni`, `jtype`, `jstype` and `javaout`. Curious readers can read about these in detail in the official **SWIG** documentation.

- `jni` – JNI C types that provide default mapping of types from C/C++ to JNI.
- `jtype` – JAVA *intermediary* types that provide default mapping of types from C/C++ to JAVA.
- `jstype` – JAVA types that provide default mapping of types from C/C++ to JAVA.
- `javaout` – converts from `jstype` to `jtype`, but, from native method call return type.
- `out` – converts method return values from C/C++ to JAVA. This is where we create a JAVA type array from C++ pointer to array of pixels as seen in Listing A.1.

`cmake`'s built-in macros for **SWIG** are used to help generate and manage the various JAVA related files created in the process. Since we are only interested in

```
//
%typemap(jni) unsigned short* "jshortArray"
%typemap(jtype) unsigned short* "short[]"
%typemap(jstype) unsigned short* "short[]"
%typemap(javaout) unsigned short* {
   return $jnicall;
}
%typemap(out) unsigned short*
{
   long lSize = (arg1)->getWidth() * (arg1)->getHeight();
   if ((arg1)->getDepthData() != nullptr)
   {
      // Create a new short[] object in Java
      jshortArray data = JCALL1(NewShortArray, jenv, lSize);
      if (data == nullptr)
      {
         jclass excep = jenv->FindClass("java/lang/NullPointerException");
         if (excep)
            jenv->ThrowNew(excep, "SIGSEV null pointer exception!");
         $result = 0;
         return $result;
      }
      // Copy pixels from image buffer
      JCALL4(SetShortArrayRegion, jenv, data, 0, lSize, (jshort*)result);
      $result = data;
   }
   else { $result = 0; }
}

...

%include "OIDepthFrame.h"
```

Listing 4.9: **SWIG** interface file for `OIDepthFrame`

the essential data required to run **ISSv2** applications, we have only described those in the above seen interface files. The generated JAVA code once bundled into a dynamic library can be used alongside a JAR, which is essentially a JAVA archive with the generated JAVA class like `wrapperJNI.java` seen in Chapter 2, Section 2.2.6, page 55 using `cmake`'s command `add_jar`. This JAR can now be directly used in the **Processing** environment including **ISSv2**, but provided the **OpenISS** Gesture Framework is bundled along with it in the form of a library just like the one mentioned above including middleware dependencies. It is important to mention the role of `rpath`, that is run-time search path that gets encoded with such bundled generated libraries. If this path is not handled and set properly it affects the portability of the create libraries. However, just like the **OpenISS** ROS package this JAVA bundle must provide similar functionality to the **OpenISS** Gesture Framework itself.

4.3 Framework Applications

In this section, we talk in detail about the various applications built using our **OpenISS** Gesture Framework to test all the functional and non-functional requirements mentioned in Chapter 1 (Section 1.2.6, page 12) and realize our motivational scenarios.

4.3.1 Integrated Sample Application

The integrated sample application is crucial to this research work in different ways. As mentioned before in Chapter 3, Section 3.3.1, page 69, it enables iterative testing of features as we progress implementing our work and serve as a visual qualitative testbed for a subset of non-functional requirements. The main components of the application can be seen in the UML diagram shown in Figure 41.

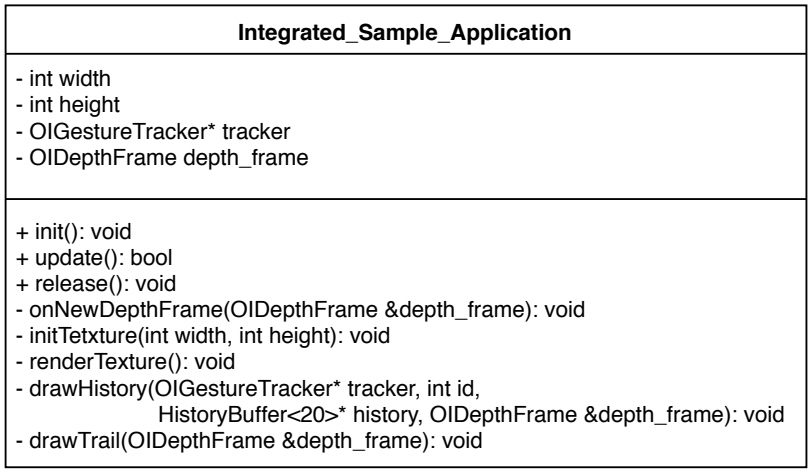


Figure 41: Integrated sample application UML diagram

Now, we will first elaborate on the application from the development perspective and then describe what exactly the application does and what it represents in context to the functionality provided by the Gesture Framework:

1. We start by creating one of the three instances as seen in Listing 4.1 before and calling the methods `init()` and `startGestureDetection()` in the `init()` of the application itself.

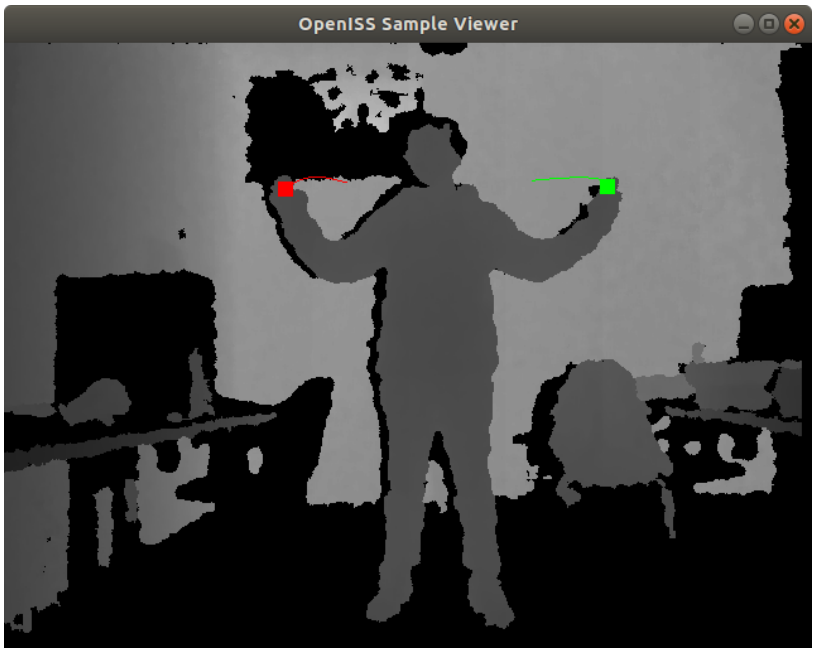


Figure 42: Integrated sample application viewer

2. Next, in the `update()` method of the application we initialize the texture once and then call the private methods `onNewDepthFrame()`, `renderTexture`, and `drawTrail()` where in `onNewDepthFrame()` we call the `update()` method of the Gesture Framework API and get the depth frame data and copy it to the texture to visualize by calling the `renderTexture` method see Figure 42. Finally, in the `drawTrail()` method we obtain the `std::vector` of gestures and hands. Now, every time a gesture is performed or a user is tracked successfully, it will trigger hand tracking, which in context to the application means a colored `GL_POINT`, just like the green and red on the left and right hand as in Figure 42. Moreover, a tracked hand will be followed by a trail of points which denote the most recent past positions the hand went through.

3. Then, we simply need to switch between the three different instances of the Gesture Framework, see Listing 4.1, and the ideally the behavior of the application should remain the same. The findings of this are discussed in the next chapter. However, in case of the **OpenISS** ROS package, one must first start the master ROS node called `roscore` and then run the package itself using

the `roscpp` command `rosrun` followed by the name of the package and then the name of the ROS node. Once all of this is done, we can run this application and see the results.

4.3.2 DigiEVISS

In Chapter 1 under Section 1.2.3, page 6, we presented the reader with a specific scenario that involves a forensic investigator wishing to manipulate preloaded visualized digital evidence objects in a gamified warehouse-like application, **DigiEVISS**. These manipulations help the investigator to create a 3D visual representation of a partial dataflow graph that further is a visual reminiscent of FORENSIC LUCID encoded program's evidential context that contains one or more observation sequences in an evidential statement. However, to perform these manipulations and create such a representation the investigator prefers natural interaction via hand gestures. Henceforth, our gesture framework that essentially provides a solution to this interactivity problem by providing hand gesture interaction abilities that are used in the application to perform such manipulations. The application is designed primarily keeping in mind the **MVC pattern** which stands for model, view and controller components of an application [129]. Currently, the model component of our **DigiEVISS** application is assumed to be preloaded digital evidence whereas the controller component is hand gesture interaction and the viewer is the 3D application window in which we render our objects and manipulations. The best illustration of the desired interaction for this application is shown in a related a **YouTube** video [8] showing the creator placing 3D models of various planets within a 3D application via hand gestures (see Figure 43) and we strongly encourage the reader to watch it. The newest illustration of the desired level of interaction and data processing is presented in the aforementioned *A Clever Label* [81] work.

Now, based on the nature of the application required, that is a 3D warehouse-world-like application (the "warehouse" is a potentially large navigable game space), we choose the well known cross-platform development library **SDL** [45] (*Simple*

Figure 43: 3D application [8] (arranging planets with hands and gestures)

DirectMedia Layer) that provides low level access to audio, input devices, and graphics hardware via OpenGL. At the same time, plenty of boilerplate code such as setting up the camera view point, near and far planes, field of view, shaders and basic geometry such as the *spheres* (or originally referred to as *bubbles* in [130–132]) for the 3D warehouse-like application has been reused from Open Source code available on platforms like **GitHub**. We assume our readers have a fair understanding of the OpenGL specification and libraries such as **SDL** in general. Thus, we will simply skip related details since the goal is to provide a solution that can enable the aforementioned scenario. This essentially means that irrespective of how sophisticated or elementary the application itself can be, our solution must be capable of enabling natural interaction via real-time hand gestures and we will only focus on the parts, which enable such natural interaction in this application. However, the structure of the application itself and related classes can be seen in Figure 44.

Broadly, `Camera` class takes care of the viewpoint of the application whereas `ApplicationRenderer` class is responsible for objects that will be rendered within the application and within this class we instantiate our framework instance. The `Scene` class essentially represent the viewer window which simply leverages both classes to visualize rendered objects. Since, NiTE2.0 and Nuitrack have quite different gesture dictionaries, it is imperative, for comparisons, to design the application

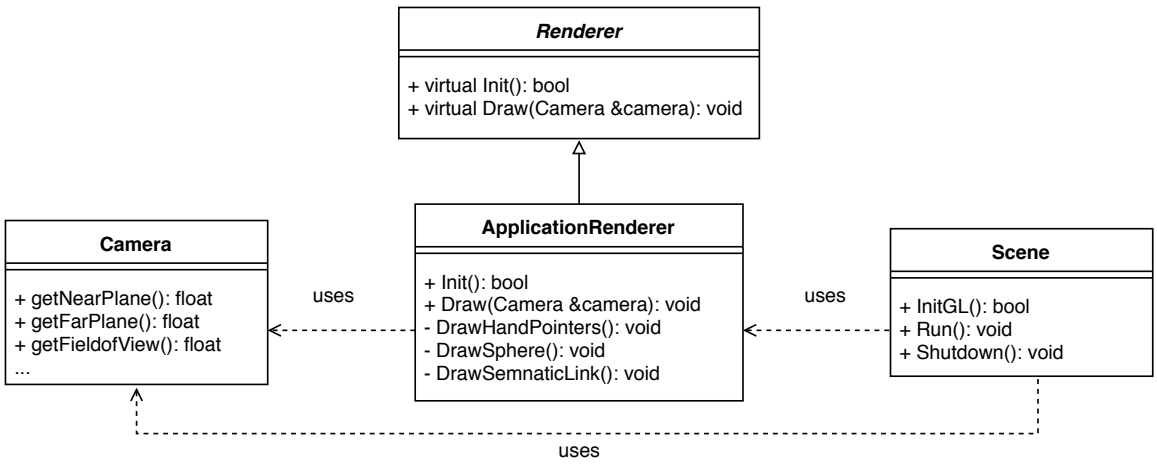


Figure 44: **DigiEVISS** structure in UML

somehow accommodating to both these gesture dictionaries. Later, based on our experiments, we will recommend, which middleware is better for this type of application.

The first preview sketch of our application can be seen in Figure 45 where the reader can spot multiple spheres in blue that are dispersed in a virtual 3D environment. These spheres represent the preloaded digital evidence objects representing observations that the forensic investigator will arrange using hands and gestures into one or more observation sequences. On the left bottom corner, there are two buttons that will **generate** the FORENSIC LUCID program similar to one seen in Listing 2.1 based upon the order of observations defined in the observation sequence created by placing the spheres in a desired order and then **compile** it with the GIPC, the details of which are provided further below. (The "preloading" aspect refers to either parsing an index of evidence identifiers from a source like a database or using GIPC to parse FORENSIC LUCID representations of the same, and representing these identifiers as the spheres with a possibility to render the content of some of the media types there – similarly to **ISSv1** the *Tangible Memories* [130] interactive documentary film idea. However, the preloading is outside the scope of this thesis.)

Initially, following the bottom-up approach we first created a rather primitive version of our application consisting of only two spheres, which at most could create one observation sequence of two observations as represented by the two

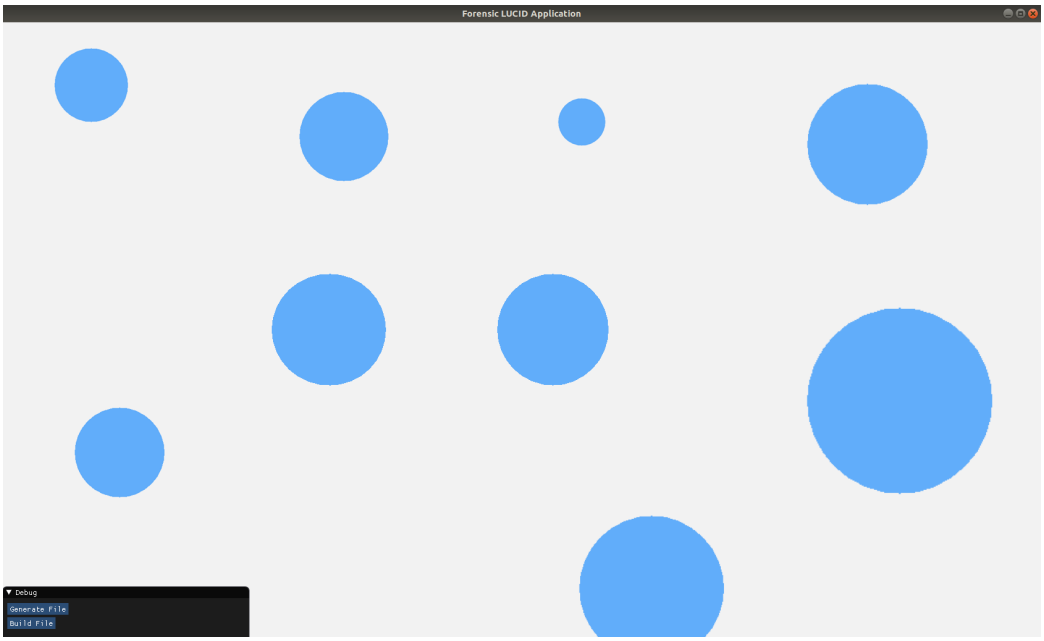


Figure 45: Scenario I: **DigiEVISS** application

aforementioned spheres. Recently, an opportunity to evaluate our gesture framework's gesture dictionary and the application surfaced in the second half of the Summer 2019 term the **INSE6610** (*Cybercrime Investigations*) course that was delivered by Dr. Serguei Mokhov here at *Concordia University*. Normally, students are given a set of various project work choices that they can choose from which is relevant to the course work. As our application is relevant to the course itself, the **OpenISS** gesture framework and the FORENSIC LUCID application were presented as one of the project work choices to employ its API usability aspects to extend the application to include more than two spheres as to create observation sequences with two or more spheres using the gesture set made available by the **OpenISS** gesture framework.

This project work was undertaken by a group of students namely, *Chen Ling, Yuaho Mao and Chao Wang* [133] that not only extended the application to include more than two spheres and create observation sequences with two or more spheres, but also evaluated the **OpenISS** gesture framework's gesture dictionary in context to the application's HCI requirements (essentially usability of certain gesture types for this application). However, the findings of this work have been deferred to the next chapter for better context in Section 5.3, page 130. For now, the difference

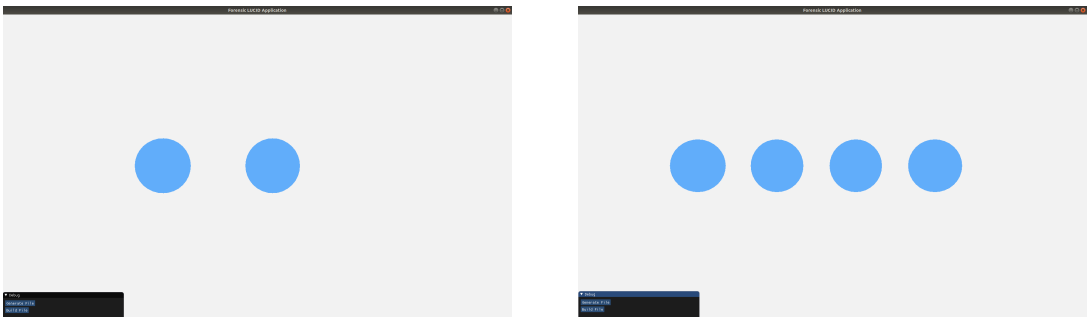


Figure 46: **DigiEVISS** application first and second iterations

between the first iteration and second iteration of the FORENSIC LUCID application can be seen in Figure 46. However, although the most recent version can manipulate multiple (but limited number of) spheres, we will illustrate essential information using only four spheres for the purpose of better clarity and visibility in the related figures. The same goes for not yet displaying preloaded data contained within each of these spheres until later. It is well understandable that currently the reader won't be able to differentiate between the spheres but the intent is to emphasize on other functional aspects such as gestures and enabling the requirements of the application.

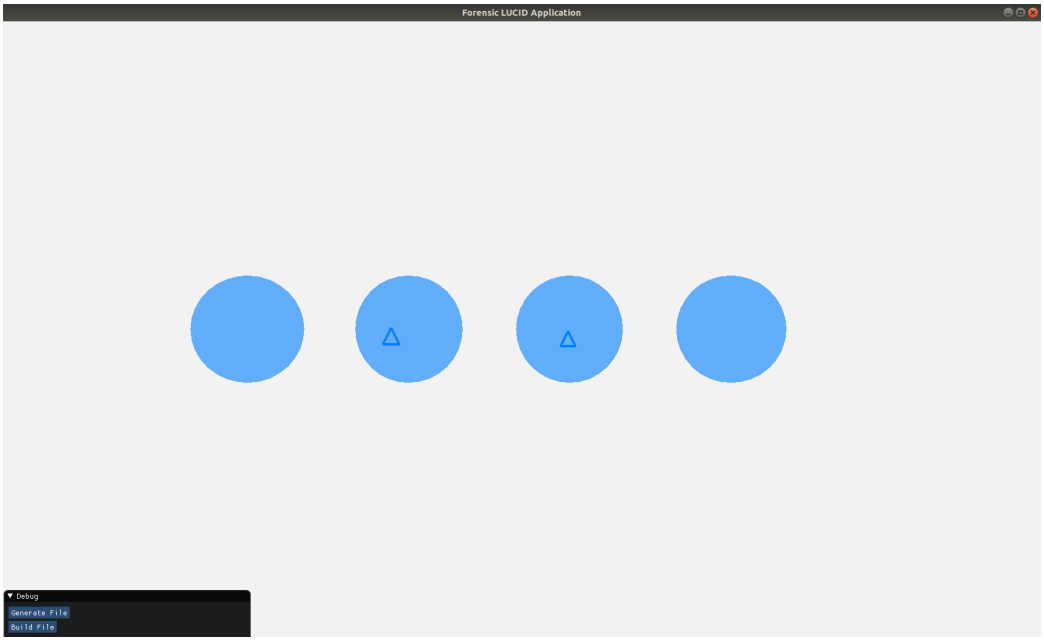


Figure 47: **DigiEVISS** application (tracked hand pointers)

Now, we will reiterate through the manipulations required that are described in detail in Section 1.2.3, page 6:

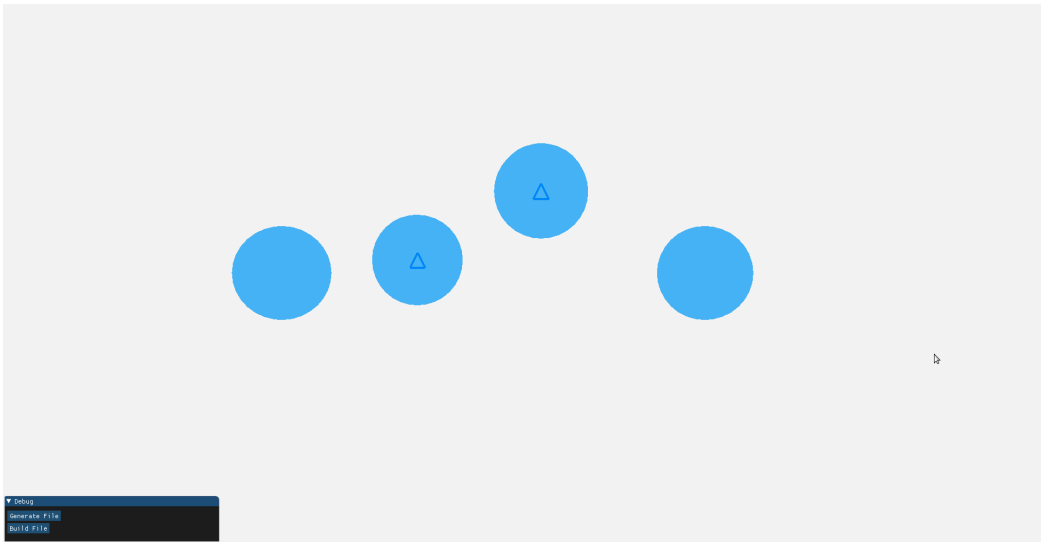


Figure 48: **DigiEVISS** application (grabbing a sphere)

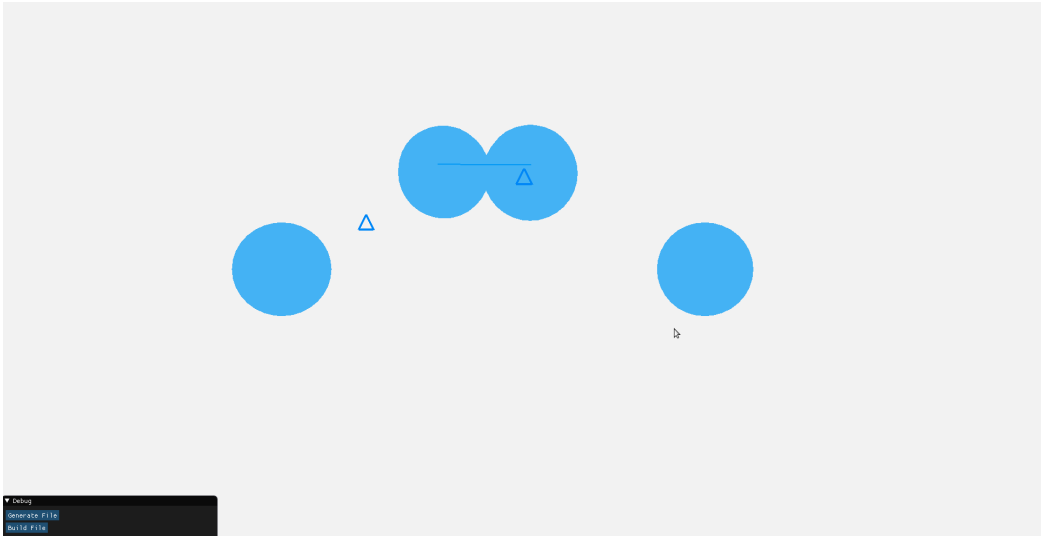


Figure 49: **DigiEVISS** application (semantic link)

1. **Select up to two digital evidence representation objects (known as "observations" in FORENSIC LUCID parlance), one at a time, via a gesture, and hold each one with each hand so that they can be naturally moved around with the hands that are holding these objects to be arranged in a desired sequence within the virtual 3D space.**

   Firstly, to select any digital evidence object we require a recognized hands that are currently being tracked. The tracking hands in this application are

represented using a small blue triangle representing a hand cursor or a hand pointer in other words, as can be seen in Figure 47. Now, once a tracking hand is above and close to the sphere it can be picked or grabbed by the hand by performing a gesture on it as seen in Figure 48. This is essentially when the distance between the sphere origin and the tracked hand position is less than or equal to the radius of the sphere.

2. **Then create a *semantic link* (a time successor relationship) between these selected objects by bringing them closer and placing them in a desired order within the 3D application. This is required because FORENSIC LUCID observation sequence evidential context is a chronological ordered collection of observations.**

   Every sphere in the application is an instance of the `Sphere` class, which has attributes such as `sphere_id` and `sphere_data`, using these we can maintain and store the order of the observations in the collection, which is essential as the end result of the observation sequence is used in the generation of the corresponding FORENSIC LUCID program. Specifically, the `sphere_id` consists of the character **o** (as in observation) followed by an integer specifying the identity number starting from 0. Therefore, within these figures the four spheres will have their `sphere_id`'s as **o0**, **o1**, **o2**, **o3** and so forth starting from left to right. Now, to semantically link two evidential objects, the user must grab the spheres with hands that are currently being tracked and are visible as triangle cursors in the 3D application and bring them close enough. In this case, when the distance between the origins of two spheres is less than or equal to the sum of their radii. In Figure 49 the reader can see the link between two spheres represented as a fine blue line stretched between their origins.

3. **Once the investigator is done creating an observation sequence with those semantically linked digital evidence objects, generate a FORENSIC LUCID encoded program including the evidential statement and compile it with the corresponding compiler within the General**

```
os
  where
```

Listing 4.10: FORENSIC LUCID program generation (header)

```
  evidential statement es1 = os2;
  os = os2;
end
```

Listing 4.11: FORENSIC LUCID program generation (footer)

Intensional Program Compiler (GIPC) framework.

Referring to the simple FORENSIC LUCID program in Listing 2.1, page 38 we divide the program generation logic into three parts. For now, the entire program is generated by appending different strings together where some of them are being dynamically generated by the manipulations performed within the application into one final `std::string` that is output as the **`simple-flucid-program.ipl`**. However, to create this final output `std::string` that is the FORENSIC LUCID program itself we use a **header** string, a **footer** string and an **observation sequence** string that is partially built dynamically based on how the spheres representing the observations (**o0**, **o1**, **o2**, **o3**) were arranged within the 3D application using hands and gestures. To further elaborate, the **header** and **footer** will consist of Listing 4.10 and Listing 4.11 from Listing 2.1 respectively. Now the **observation sequence** string itself is further built using two strings that are built dynamically based on the order, in which the observations are semantically linked. A pseudo-code representation of the same can be seen in Listing 4.12.

It should be noted the original simple program from Listing 2.1 [80], is just an illustration of syntactical constructs for observations and their sequences and an equivalence of those sequences. In itself, it simply constructs two observation sequences using two notations – a list and the `fby` operator, returns the second observation sequence, and its evidential statement as only a single observation sequence (because both sequences are actually same in that program). For

```
...
for(int i = 0; i < spheres.size(); i++)
{
  "observation "
  + std::to_string(spheres[i].getSphereID())
  + " = "
  + "(" + spheres[i].getSphereData()
  + ");\n"
}
...
"observation sequence os1 = {" + ob_seq_comma + "};\n"
"observation sequence os2 = {" + ob_seq_fby + "};\n"
...
```

Listing 4.12: FORENSIC LUCID program generation (observations and observation sequence)

its bare simplicity we chose it to use for dynamic generation of its evidential context instead. We plan to extend the application to parse and work with more complex and numerous evidential objects like those in some other samples in [80].

4.3.3 Scenario II Application

Similarly, in Chapter 1 under Section 1.2.4, page 8, we present the reader with our second specific scenario, in which an artist requires real-time hand gesture recognition and detection to create visual effects within the **ISSv2** (see Section 2.1.1.2, page 32) that is built on top of **Processing** and JAVA. We will yet again reiterate through the scenario and elaborate on how our gesture framework will enable this application scenario and fulfill the requirements.

1. **First, select OpenISS as a backend instance "driver" to instantiate a desired middleware/library that provides real-time hand-gesture recognition, hand-tracking and necessary depth information functionalities via our solution, and, this is a configurable selection.**

   So far we introduced the reader to the notion of backends in **ISSv2**. Surprisingly, there are seven experimental backends in total and **OpenISS** gesture framework is the eighth one; however, the reader does not need to

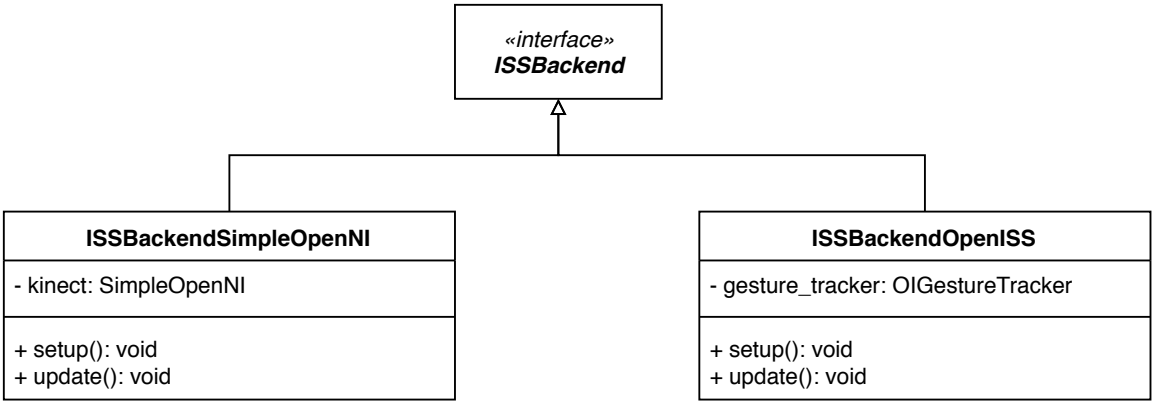


Figure 50: ISSv2 backends `SimpleOpenNI` and **OpenISS**

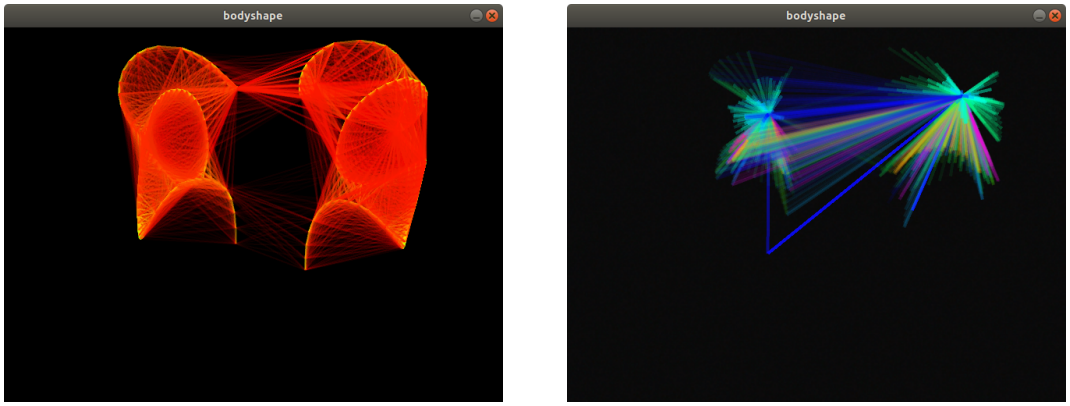


Figure 51: ISSv2 application scenario II visual effect 1 and 2

be aware of the others except the `SimpleOpenNI` backend mentioned a few times before as it provide limited gesture functionality that we alleviate with our solution and make comparisons. To select **OpenISS** as a backend instance "driver" we created one in the **ISSv2** itself, see Figure 50, and, then eventually replacing the existing `SimpleOpenNI` backend by gradually testing one feature at a time with existing **ISSv2** applications.

2. **Then, execute and interact with the existing ISSv2 performing arts applications for performance visualization using hand-gesture interaction provided via our solution within the OpenISS backend as per usual with other backends such as the `SimpleOpenNI` backend mentioned before with limited gesture interaction.**

In **Processing**, there are two ways to import a library into its environment. Now, when **Processing** is run for the first time after installation, it creates a directory named `sketchbook/` under the `home/` directory of the user that installed it. However, the another way to do so is by creating a directory named `code/` that can be placed alongside the *sketch* with extension `.pde` and place all the required shared order dynamic libraries and JAR files under that directory. This essentially means that the libraries and **OpenISS** gesture framework JAVA classes generated by **SWIG** bundled into a JAR in Section 4.2.4 must be placed in similar fashion to the `SimpleOpenNI` libraries and JARs and then **OpenISS** gesture framework functionality can be called into JAVA code from C++. Then we simply replaced the `SimpleOpenNI` instances with our gesture framework instances and the results can be seen in Figure 51. We also created an equivalent **Processing** *sketch* of integrated sample application in Section 4.3.1.

3. **Or, create new ISSv2 performing arts application profiles for performance visualizations using completely new hand-gestures previously not possible via our solution that must provide an easy to use Application Programming Interface (API) for the computation artists to do so.**

   We also created a new **ISSv2** performing arts application profile for performance visualizations that uses the **OpenISS** gesture framework provided hand gestures to create visual effects with not only OpenNI2.0 and NiTE2.0 just like `SimpleOpenNI`, but also Nuitrack, which contains newer gestures that were not present in the **ISSv2** prior to our gesture framework's cross language module as detailed in Section 4.2.4. Moreover, we are the first ones to provide Nuitrack functionality in **Processing**. However, as mentioned prior in Section 1.6 we will not evaluate the usability of the API formally with a computation artist.

4.4 Summary

In this chapter, we only focused on the design and implementation details, concrete classes, their definitions and corresponding UML diagrams related to the overall solution of this work that is the gesture framework. Then we describe the various applications powered by our gesture framework. In the next chapter, we discuss the analysis and evaluation of all the functional and non-functional requirements, whether qualitative or quantitative.

Chapter 5

Evaluation and Results

In this chapter, we present the various findings of our two different evaluation approaches namely, qualitative and quantitative evaluation as presented in Chapter 3, under Section 3.3.3, page 73. Moreover, we will essentially present the results in the same order as described in the evaluation steps in Section 3.3.3. Following the bottom-up approach, we will start by using our integrated sample application described earlier that not only enables us to visually track our progress, but also serves as a testbed to evaluate various functional and non-functional requirements in terms of rendering frame rate and sensor data frame rate. Moreover, we have an equivalent sample application in JAVA for **Processing** to qualitatively evaluate a subset of our specific requirements, **for example**, see NFR7 and NFR8.

5.1 Evaluation Testbed Specifications

Before we commence our discussions regarding our evaluation, we describe its environment including the operating system used, processor power, memory, hardware used, devices and so forth. Except for the ROS overhead evaluation experimentation where we have two machines/nodes (one serving as a master node and publisher on the same where as the other as a client with a data subscription to the aforementioned publisher node), the rest of our experiments and evaluations we employ the client machine only. Henceforth, the hardware specifications of both client and server can be

seen in Table 7; whereas, the various middleware/libraries/tools used in this research work are described in Table 8 and the different depth sensor devices used and tested are listed in Table 9 along with their hardware specifications. It is vital to mention that various steps were taken to minimize affect on our evaluation in terms of running as less processes as possible.

Setting	**Name**	**Device**
Client	Memory	8 GB
	Processor	Intel Core i5-2400 CPU @3.10GHz × 4
	Graphics	Nvidia Quadro 600 (1 GB)
	OS	Ubuntu 18.04.3 LTS 64-bit
Server	Memory	12 GB
	Processor	Intel Core i7-920 CPU @2.67GHz × 8
	Graphics	GeForce GTX1080 Ti (12 GB)
	OS	Ubuntu 18.04.3 LTS 64-bit

Table 7: Environment hardware specifications

Type	**Name**	**Version**
Middleware	OpenNI2.0	2.2.0.33 Beta x64
	NiTE2.0	2.2.0.5 x64
	Nuitrack	0.2.4
	ROS	Melodic Morenia
Libraries	Libfreenect	0.6.0
	Libfreenect2	0.2.0
	Processing	3.5.3
	OpenGL	4.6.0
	SDL2	2.0.8
Tools	CMake	3.12.2
	SWIG	4.0.1

Table 8: Middleware, libraries, and tools used

Depth Sensor Device	Model Version	Depth Resolution (pixels)	Frame Rate (Hz)	FOV (h*v)
Kinect v1	1414, 1473	640 * 480	30	57° * 43°
Kinect v2	1656	512 * 424	30	70° * 60°
Intel RealSense	D435	1280 * 720	60	87° * 58°

Table 9: Depth sensors specifications

5.2 Integrated Sample Application as Evaluation Testbed

As described in Section 3.3.3.3, page 77 we performed specific experiments to qualitatively and quantitatively evaluate our various functional and non-functional requirements successively using our integrated sample application as described in Section 4.3.1, page 108, which also serves as an important resource for the end users to illustrate usage of our gesture framework to obtain real-time gesture interaction via different NiTE2.0 or Nuitrack middleware. As seen prior in Figure 42 the reader can spot the depth information (rendered as background texture within the application) of the scene with the user waiving his hands. There, the red and the green points on the user's hands depict that these hands are currently being tracked and the trail following these points that appears like a fine line of the same color as the points represents recent past positions of the hand as it moved to its current position. Further, we compare the behavior of the application between NiTE2.0 and Nuitrack (and their ROS equivalent) backends of our gesture framework. As the reader can recall from Listing 4.1, page 95 to instantiate either backend one will only require to instantiate the corresponding adapter class object.

NiTE2.0

In Figure 52, the reader can see the NiTE2.0 gestures detected being notified at the console, but provided via our gesture framework. In NiTE2.0 a successfully detected gesture is required to start real-time hand tracking as seen in Figure 53. Essentially,

the real-world position coordinates of successfully detected gestures are passed on to the hand tracking algorithm that takes on from that point and tracks the hands as long as it is not occluded or somehow lost or went out the sensor's field of view. As mentioned in Section 4.2.1, page 95 it has three gestures namely:

1. `GESTURE_WAVE`
2. `GESTURE_CLICK`
3. `GESTURE_HAND_RAISE`

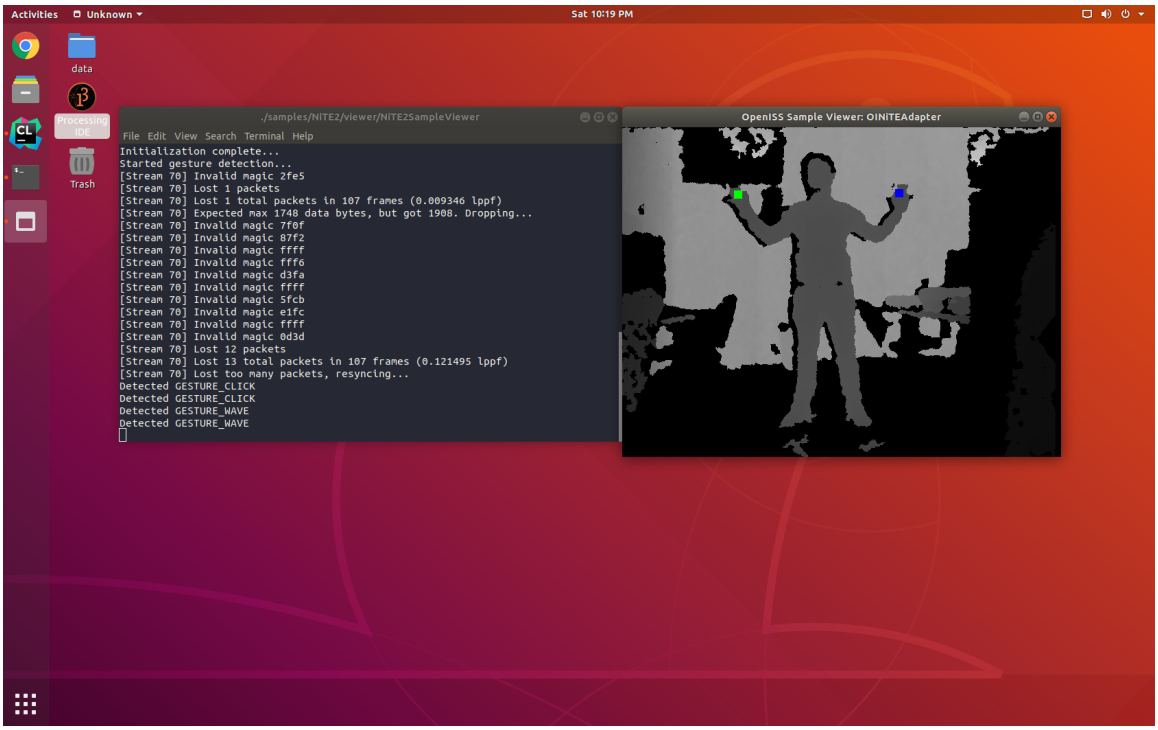


Figure 52: Integrated sample application NiTE2.0 backend gesture recognition

Nuitrack

Within the same sample application we comment out the previously instantiated NiTE2.0 adapter class object and instantiate Nuitrack adapter class object instead. This essentially demonstrates that minimally affecting the application one can switch from one gesture provider backend to another given their corresponding adapter class which exists within the framework to adapt required functionality. It must

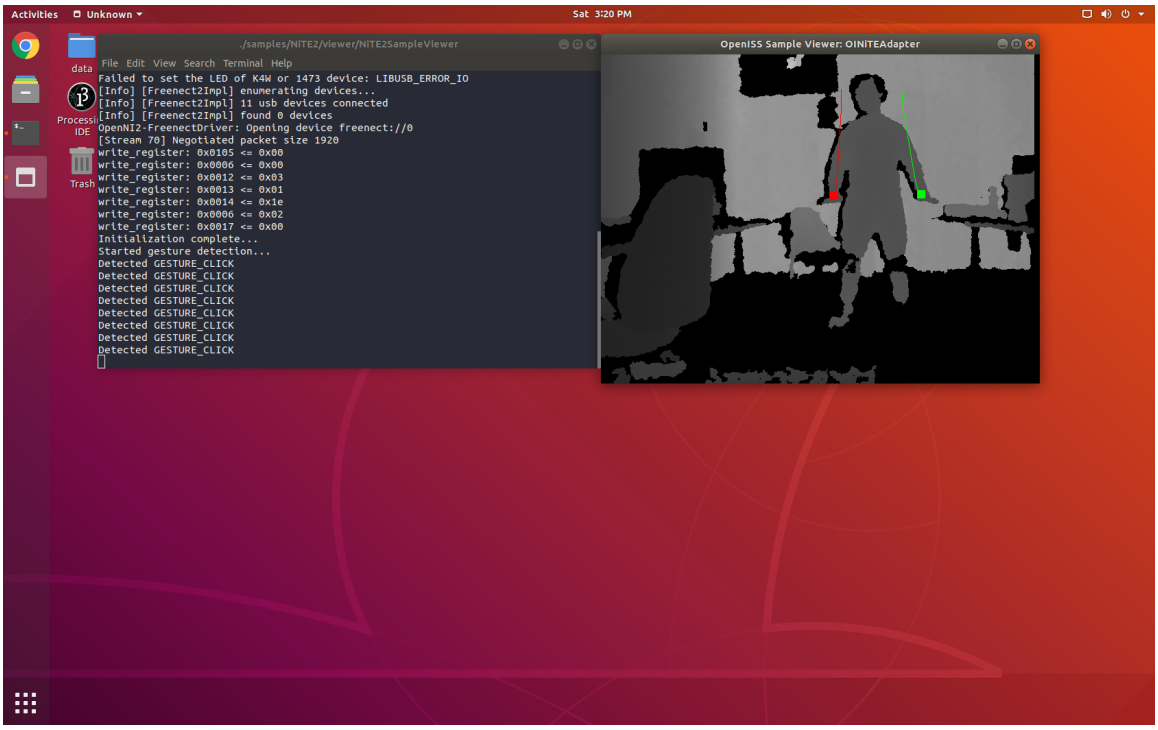


Figure 53: Integrated sample application NiTE2.0 backend hand tracking

be noted that in case of the sample integrated application the functionality is trivial for demonstration purposes only; however, it is still true that if two gesture providers with a common subset of gestures in their gesture dictionaries can be easily switched between with minimal modifications to the end user application.

In Figure 54, the reader can see the Nuitrack gestures detected are being notified at the console. In Nuitrack, there is no such requirement to first perform a gesture for hand tracking as seen in Figure 55. Once a user is detected within the frame, the hands are tracked by default and gestures can be recognized and detected at any instance as long as they are performed within the sensor's field of view. Also, as described previously in Section 4.2.2, page 98 Nuitrack has six gestures in its gesture dictionary namely:

1. `GESTURE_WAVING`
2. `GESTURE_SWIPE_LEFT`
3. `GESTURE_SWIPE_RIGHT`

4. `GESTURE_SWIPE_UP`
5. `GESTURE_SWIPE_DOWN`
6. `GESTURE_PUSH`

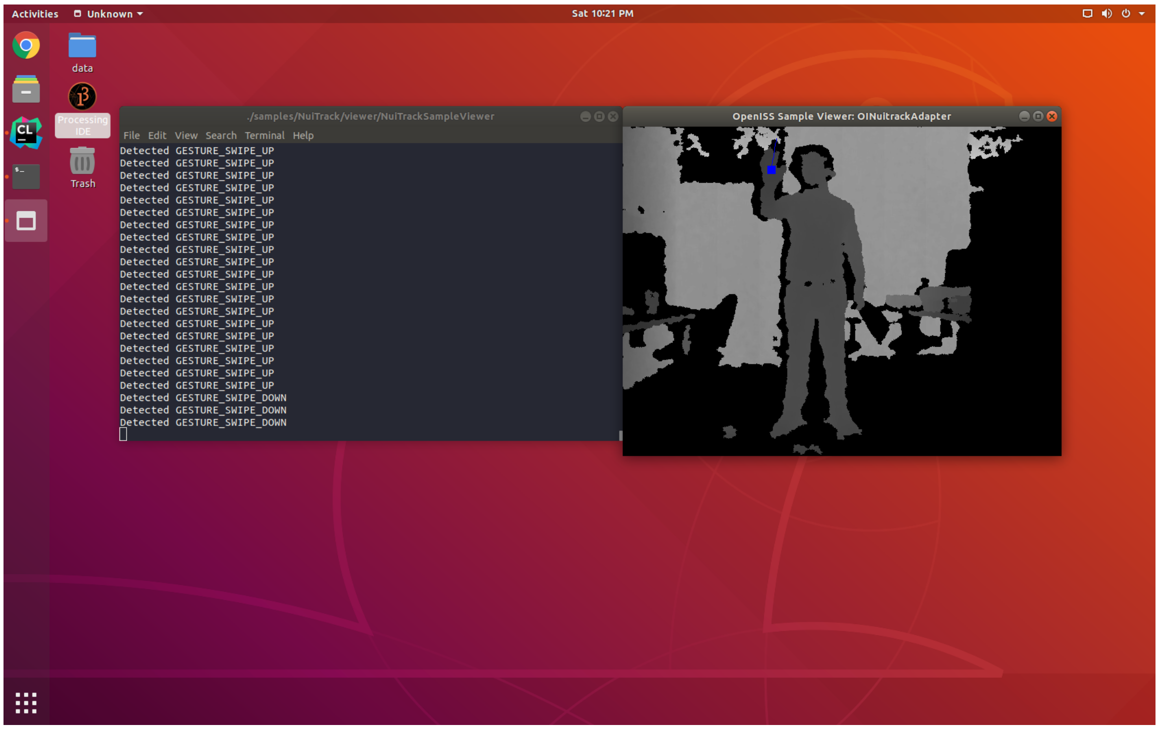


Figure 54: Integrated sample application Nuitrack gesture adapter (gestures)

NiTE2.0 and Nuitrack Key Comparisons

- **Current status**: Although NiTE2.0 has not been maintained or suported for a few years and research involving use of OpenNI2.0, NiTE2.0 and the Microsoft Kinect stack for motion tracking and gesture interaction is on the decline[1] it is still used by the academia. Whereas, Nuitrack is relatively a newer gesture provider middleware that is currently being developed, maintained and supported by the author company as well. Even though Nuitrack requires a commercial license to work, it still can be used for a total of *three* minutes in an unlicensed version, which we found sufficient for our research work.

[1] The use of the OpenNI2/NiTE2/Kinect stack is on the decline, Kinect alone is still used adequately.

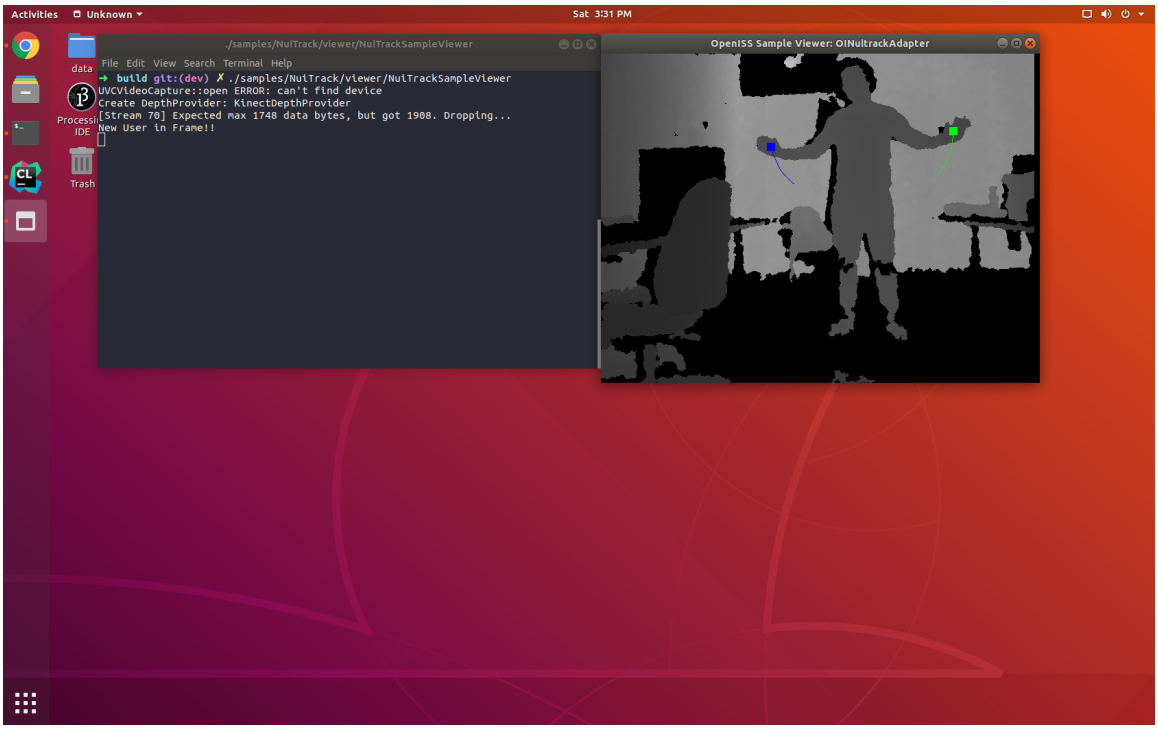


Figure 55: Integrated sample application Nuitrack gesture adapter (hands)

- **Platforms supported**: NiTE2.0 binaries are available for MacOSX, Linux and Windows. In this work we personally tested it on Linux and MacOSX platforms so we will not comment on its working on the Windows platform. Nuitrack is available for Windows and Linux platforms only and does not supports MacOSX operating system which however is abundantly used for most of our **ISSv2** applications.

- **Gesture dictionaries**: So far the reader is aware that NiTE2.0 provides three gestures in total, however, specifically the hand raise gesture is not useful since it gets detected always even if the user raise their hand to perform the other two. Whereas, Nuitrack provide a total of six gestures out of which two waving and click/push are similar to the remaining NiTE2.0 gestures.

- **Supported depth devices**: In terms of the amount of depth devices that either of these middleware can support, Nuitrack is clearly a winner. However, in context to this research work, three of our devices namely, Kinect v1, Kinect v2

and Intel RealSense D435 can work with both these middleware. As previously mentioned, Intel RealSense SDK2.0 now officially supports OpenNI2.0 via its wrappers as thus, by extension NiTE2.0 as well [134]. However, in the end Nuitrack provides for more devices than NiTE2.0 such as Asus Xtion, Orbbec Astra, and Orbbec Persee as well.

- **Functionality**: By default, both NiTE2.0 and Nuitrack fail to tell which hand performed the gesture between the two of the same user. We strongly felt the need of such information whilst developing **DigiEVISS** specifically. Moreover, in NiTE2.0 by default there is no such notion of a *left* or a *right* hand rather left/right hand joints available via its `Skeleton` class, whereas Nuitrack explicitly provides both hands. However, it is not very precise as we observed that if the user turns around the left becomes right and the right becomes left.

Irrespective of these, the reader can see in Figure 52, Figure 53, Figure 54, and, Figure 55 the behavior of the application doesn't really change except the color of the points which gets initialized at random from a set of few colors.

Further, in terms of real-time response as observed qualitatively and visually the results are very satisfying and seem adequately real-time. Now, to quantify the rendering frame rate per second we employed a frame rate counter within our application and we run the application for an average of *three* minutes. There are various ways to calculate the frame rate but all are based on the same equation to simply count the number of frames rendered in a minute and divide it by the number of seconds that is 60 and in our case 180. The reason to run it for exactly three minutes is that since we didn't purchase a commercial license for Nuitrack, and without one Nuitrack only runs for three minutes and then terminates with a `LicenseNotAcquiredException` signal as mentioned previously; thus, for consistent results across NiTE2.0 and Nuitrack providers we settled to run our application for three minutes for measurements. Now, the average **measured** frame rates of the same sample application, but with NiTE2.0 and Nuitrack adapters via our gesture

framework can be seen in Table 10.

Table 10: Integrated sample application (average rendering FPS vs. sensor FPS)

OpenISS gesture provider	rFPS	sFPS	effective FPS
NiTE2.0	58FPS	29FPS	43FPS
Nuitrack	50FPS	28FPS	39FPS

We observe, as NiTE2.0 is relatively smaller and less complex than Nuitrack; hence, greater FPS than Nuitrack, which is more resource-intensive than NiTE2.0 simply because it recognizes more gestures than NiTE2.0 and provides more functionality as well. Moreover, this essentially constitutes the **qualitative and quantitative evaluation** of the following requirements, but from the perspective of our integrated sample application following the bottom-up approach as described in Section 3.3.1:

- FR1: **The system shall provide real-time hand gesture recognition.**
- FR2: **The system shall provide real-time hand tracking.**
- FR3: **The system shall provide real-time depth information of user's hands.**
- NFR3: **The ability of a system to accommodate prospective solutions easily.**

Next, as described in the step 2 in Section 3.3.3, page 73, we will discuss our observations and evaluation results for our **DigiEVISS** application on both NiTE2.0 and Nuitrack middleware.

5.3 DigiEVISS

We mentioned previously that the application must be accommodating for both middleware so as to perform a fair comparison that is uniform among both middleware. Therefore, the primary actions required to be performed in the **DigiEVISS** application can be performed using either of the middleware. As we observed and evaluated our **DigiEVISS** application against the two middleware and

depth devices, such as the Microsoft Kinect v1, Kinect v2, and Intel RealSense D435 we discovered the need to make slight changes in the application from the perspective of usability. As we know that the user will essentially perform manipulations to digital evidence objects represented as spheres in the application. Now, an attached sphere that has been grabbed by a currently tracking hand and is attached to it will appear *red* instead of the usual blue. Moreover, once the semantic link is created between two spheres they will appear as *green*. At the same time, the `sphere_id` is visible on the sphere itself to enable the reader to differentiate between different *observations*.

We also mentioned before that we essentially design the application in such a way that the size of gesture dictionaries of NiTE2.0 and Nuitrack will not affect at least picking up two spheres and creating a semantic link between them. However, Nuitrack is certainly far better than NiTE2.0 when it comes to this application. Not only the direction gestures of Nuitrack can be prospectively used to navigate around in the 3D warehouse-like application, but it also directly fulfills our another non-functional requirement NFR6, since Nuitrack itself supports an array of different devices whereas NiTE2.0 is restricted only to the Microsoft Kinect devices by default. We also observed that using the Intel RealSense D435 performs better than the Kinect devices in terms of responsiveness and ease of use due to better hardware (especially large field of view for interaction) and middleware. In terms of gesture survey mentioned in the previous chapter the authors surveyed the **DigiEVISS** application as to applicable gestures and found that **OpenISS** yet is not adequate enough in terms of the type of gestures it provides via NiTE2.0 or Nuitrack to handle complex actions. The survey suggests integrating **OpenPose** library that has a gesture dictionary of 28 gestures. However, we defer the integration of the same for future work.

Moreover, in this application we found the frame rate results pretty consistent with as seen in Table 10 of the integrated sample application. This totally makes sense because both the sample integrated application and the **DigiEVISS** in their present states are relatively basic and the scenes or objects being rendered are simple geometric types and thus their render rates are quite similar to as seen in Table 10.

The sample FORENSIC LUCID program generated by the manipulations of the

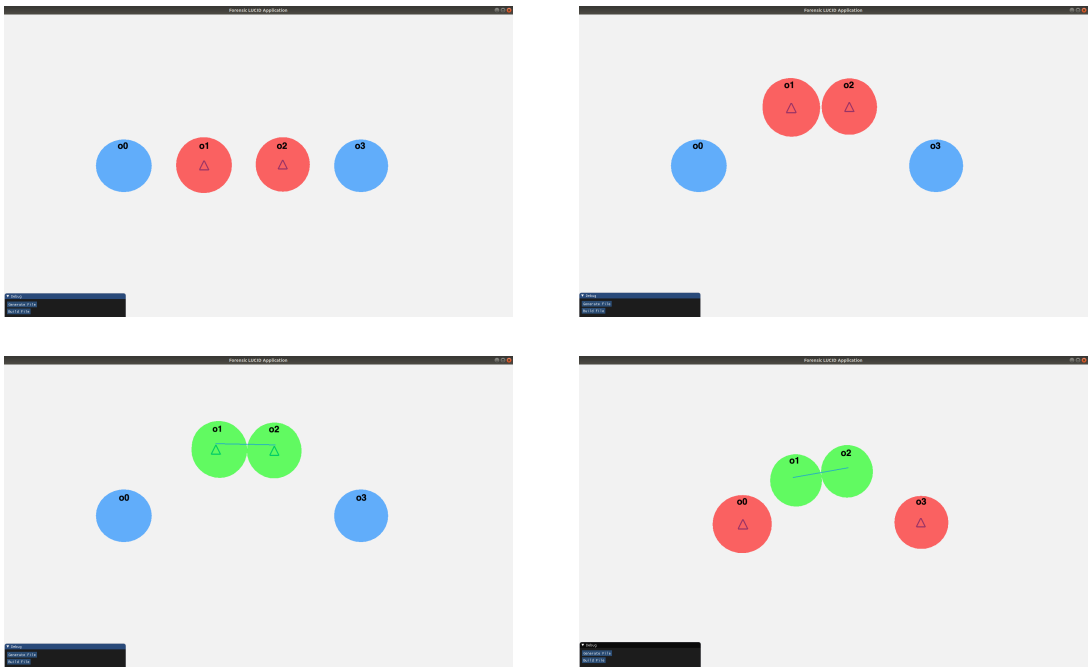


Figure 56: **DigiEVISS**

observations as depicted in Figure 56 can be seen in Listing 5.1. It is important to mention that this generated program can be successfully compiled with GIPC for semantic correctness as long as the required binaries are placed at the same location where the generated program is saved.

```
es
where
  observation o0 = ("A printed" => "very well", 1, 0, 0.85, 1234);
  observation o1 = ("B printed" => "well", 1, 0, 0.65, 2341);
  observation o2 = ("C printed" => "not very well", 1, 0, 0.35, 3412);
  observation o3 = ("D printed" => "not well", 1, 0, 0.15, 4123);
  observation sequence os1 = {o0, o3};
  observation sequence os2 = o1 fby o2;
  evidential statement es1 = {os1,os2};
  es = es1;
end
```

Listing 5.1: Generated FORENSIC LUCID observation sequence context

5.4 ISSv2

Based on our work described in Section 4.2.4, page 104, we will now test our gesture framework capabilities in our equivalent sample application, but in **Processing**. As we observe similar results as expected in contrast to our sample application in C++

we will further discuss the results of our gesture framework in the existing **ISSv2** visual effects pipeline.

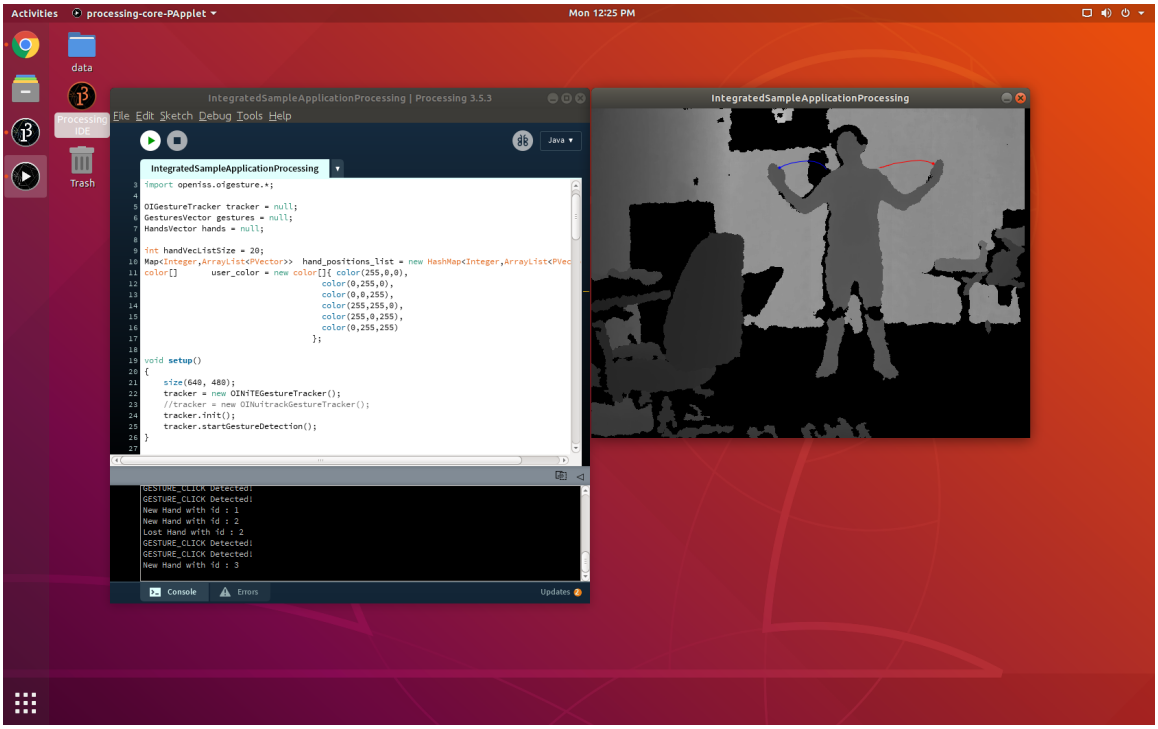


Figure 57: Integrated sample application Processing (NiTE2.0)

Despite the language bridge, the results in terms of render frame rate were yet again consistent with our previous results with integrated sample application. As seen in Figure 57 and Figure 58 the reader can notice that the only difference between them is where we instantiate NiTE2.0 vs. Nuitrack gesture provider via our gesture framework. The **Processing** application behaves exactly as expected to its C++ counterpart. This bolsters our claim that high rendering rate is observed due to minimal rendering computations.

However, specifically in **ISSv2**, there is an abundance of such computations in the form of data transformation, computer vision algorithms from **OpenCV** such as edge detection or contouring, VFX generation and so forth. Thus, **ISSv2** is much more intensive processing application that generates VFX using dynamic shapes, physics, based on hand positions or skeleton positions as delivered to it from the sensor. Therefore, we will now evaluate our gesture framework as a backend in the existing **ISSv2** pipeline to observe:

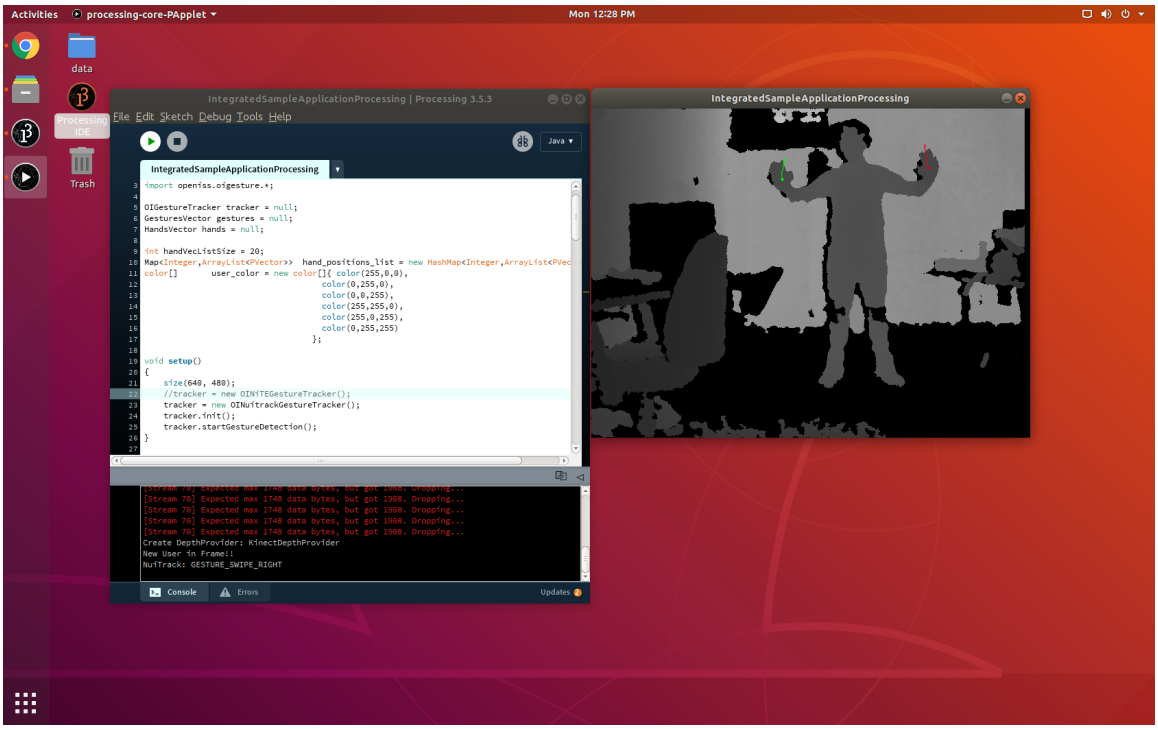


Figure 58: Integrated sample application Processing (Nuitrack)

- Whether the existing visual effects in **ISSv2** work with the newer **OpenISS** gesture backend?
- Whether the visual rendering response is real-time and render frame rate is equal to or greater than 15FPS?

As we can see in Figure 59 various existing visual effects in the **ISSv2** pipeline work well with our backend. Moreover, we noticed that the rendering frame rate of the application averages at **39FPS** using **OpenISS**, which is still better than the average frame rate of **29FPS** as seen with the existing `SimpleOpenNI` backend. One can only speculate that if `SimpleOpenNI` is merely trying to synchronize with device rate. Moreover, `SimpleOpenNI` relies on a relatively older version of `libfreenect` which is a plausible explanation as well for a lower reder rate. However, **39FPS** is less than compared to the trend seen so far with our integrated sample application and **DigiEVISS**. This is essentially due to the heavy computations that take place within the VFX pipeline of **ISSv2**. As we repeated the same experiment with Intel RealSense, which provides **30FPS** as the *default* sensor data rate the rendering frame

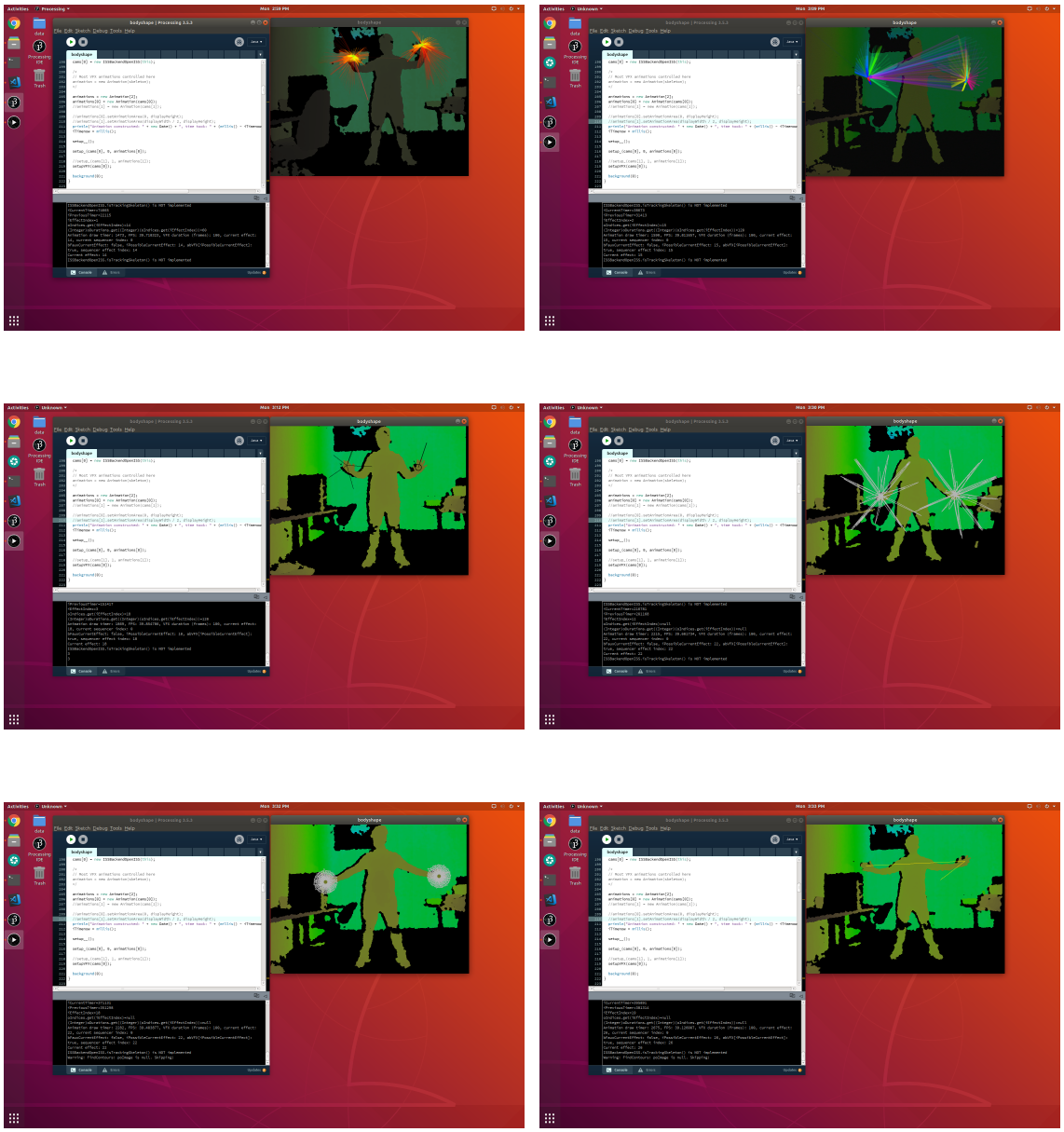


Figure 59: ISSv2 visual effects via gestures framework

rate still remained the same. Moreover, as mentioned briefly prior, via this research work we provide access to the Nuitrack API data via our API, which is first of its kind to do that. This already broadens the scope of **ISSv2** devices that it can work with apart from only restricted to the Microsoft Kinect devices.

Not only the answer to the above posed questions is *yes*, but at the same time this also constitutes an evaluation for the following attained requirements qualitatively

Table 11: ISSv2 (average rendering FPS vs. sensor FPS)

OpenISS gesture provider	rFPS	sFPS	effective FPS
NiTE2.0	39FPS	29FPS	34FPS
Nuitrack	39FPS	28FPS	33FPS

and quantitatively wherever applicable:

- FR4: **The system shall serve as an easy-to-switch-to backend with the existing ISSv2 pipeline.**
- NFR1: **The system should be able to process the frames at a speed that is at least 10 frames-per-second (FPS)**
- NFR4: **The ability of a system to inter-operate with other software systems and share meaningful data.**
- NFR6: **The ability of a system to switch between devices only minimally affecting the end user application.**
- NFR7: **The ability of a system to function on more than one platform.**
- NFR8: **The ability of a system to share data from its implementation language to a different programming language application.**

5.5 Integrated Sample Application as ROS Client

In case of the ROS adapter there are a few extra steps that we need to perform to evaluate gestures being published over ROS. We will demonstrate this with a single machine setup first where the same machine acts as a server and a client or in other words the **OpenISS** ROS package that essentially publishes depth, gesture and hand data and the client application reside on the same computer. Later we will test this setup on separate machines.

Now, first and foremost it is crucial for the `master` node to be up and running by using the command `roscore`. Next, in an another terminal window we run the **OpenISS** ROS package node that will essentially publish gesture data over the ROS

network by using the `rosrun` command. Now, ROS provides another command `rostopic`, which essentially prints information regarding the current ROS topics on which `messages` are being published, as visually illustrated in Figure 60.

Finally, we run our sample application that instantiates a ROS client adapter as seen in Figure 61 and Figure 62. The reader can spot the gestures being detected over the ROS network and available to the client application that is the integrated sample application.

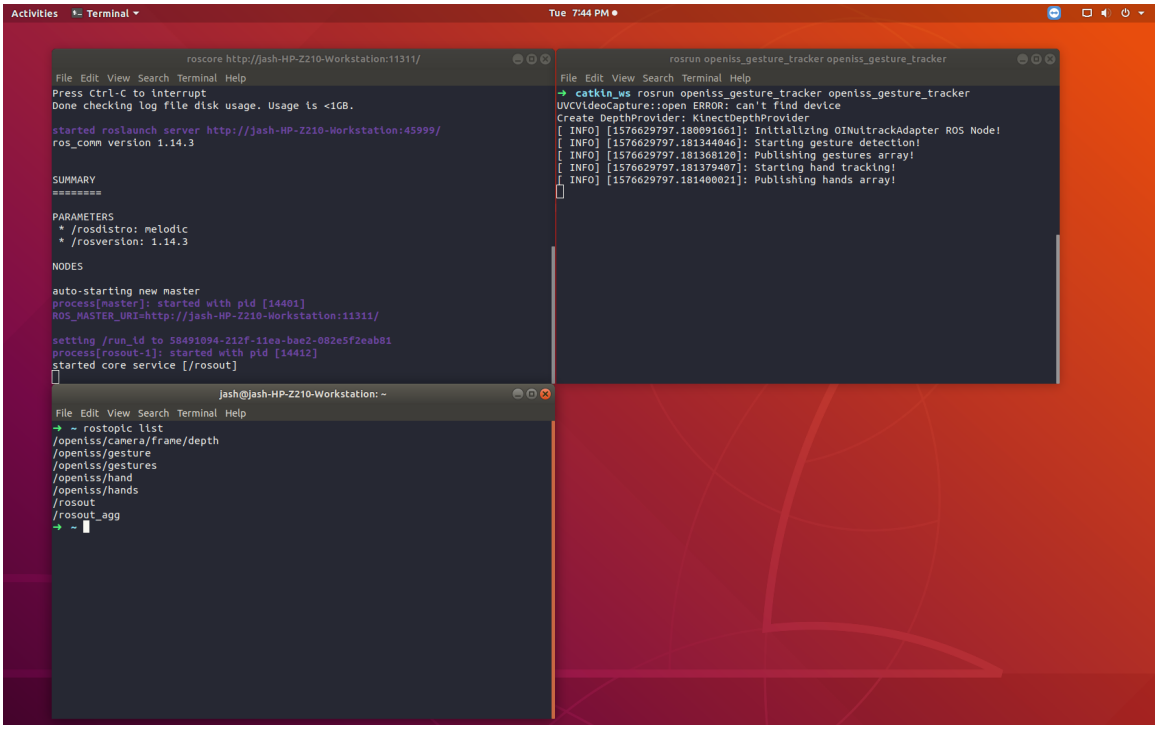


Figure 60: Master node and **OpenISS** ROS node publishing data on various topics

Now, as mentioned before, an interesting aspect of working with ROS is that it decouples the publisher from the subscriber where the publisher is the **OpenISS** gesture framework disguised as a ROS package and the publisher is the end user application which in our case is the integrated sample application. This essentially means that the client can still render at **60FPS**, while receiving zero frames from the sensor itself. Thus, measuring pure render frame rate (rFPS) was not sufficient in our scenarios. This, especially exhibited by ROS, where the data acquisition server application and the render application are physically or logically separated as different processes completely decoupling the rendering and sensor processes. To measure the

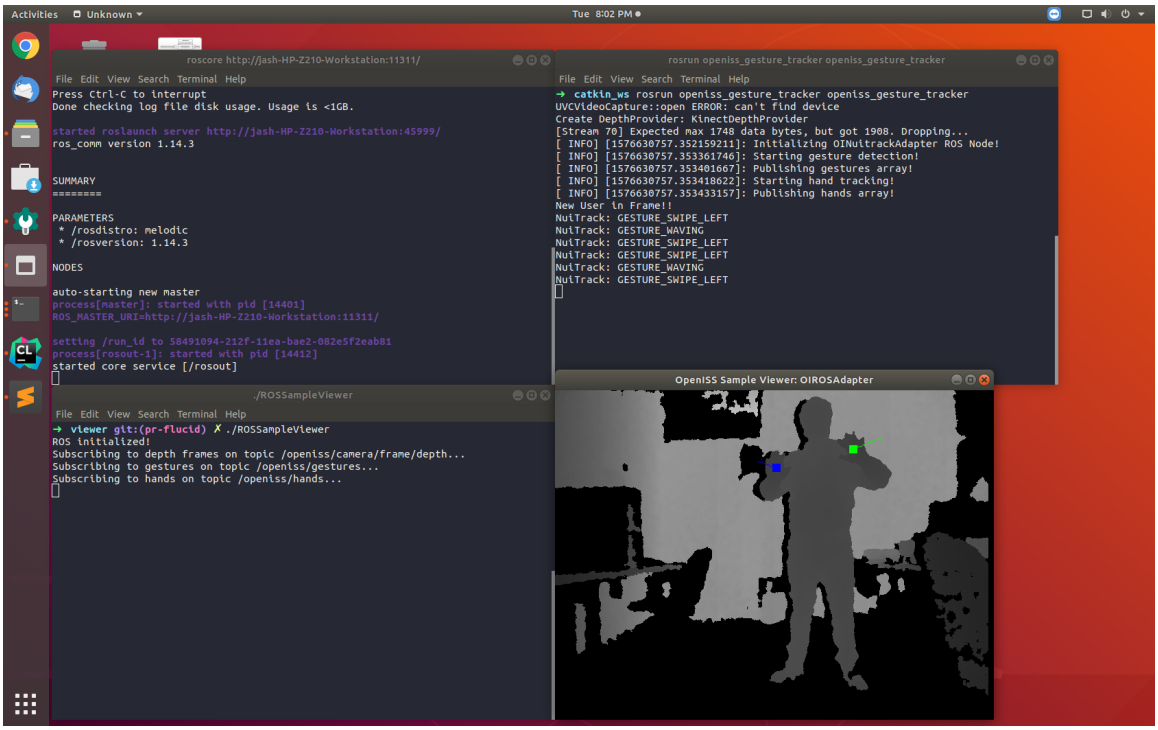


Figure 61: Integrated sample application ROS (Nuitrack) gesture adapter

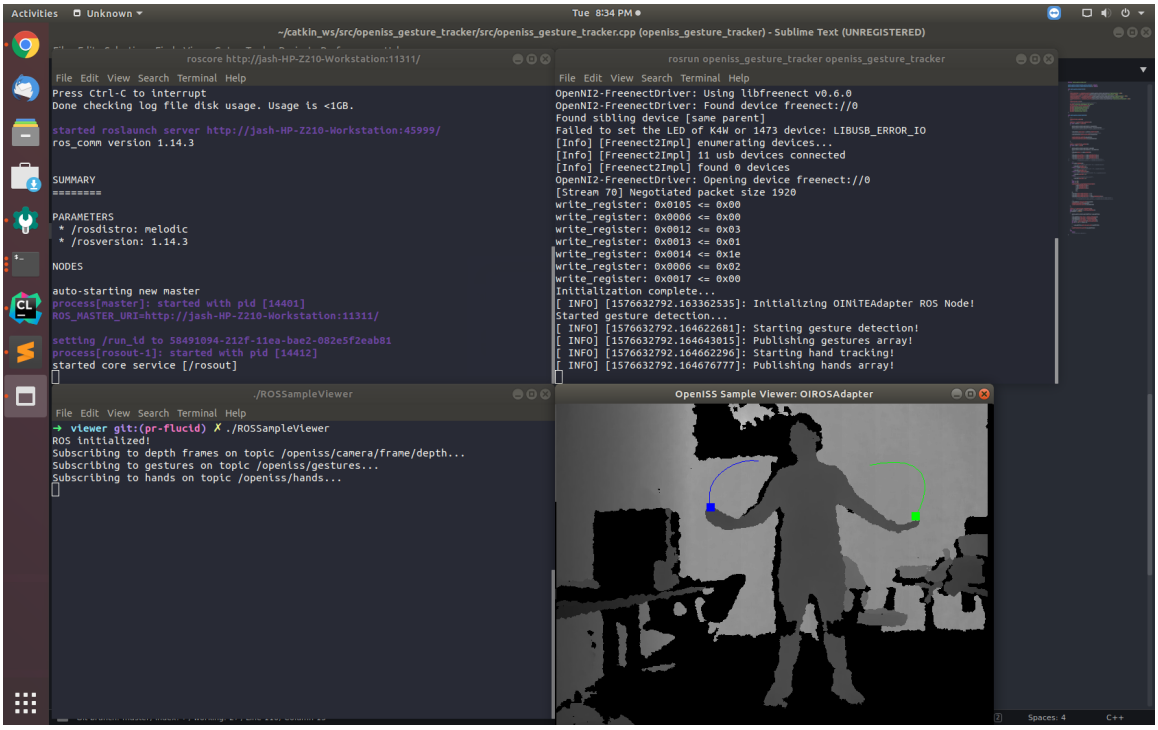


Figure 62: Integrated sample application ROS (NiTE2.0) gesture adapter

overhead introduced by ROS we leverage additional instrumentation to measure the data delivery rate to the application from the middleware, which essentially boils down to measuring the amount of data frames received vs. the number of frames produced on the server side. Thus, this resulted in counting callbacks-per-second via the various callbacks provided by NiTE2.0, Nuitrack and ROS.

Next, we setup our experiment on separate machines where one acts as the server and the other as the client. Once we had fully bidirectional communication between the two, so we can start the `master` node on the server. Next, on the client we setup two essential ROS environment variables, namely, `ROS_MASTER_URI` and `ROS_IP`. Since all nodes must talk to the same `master` node we simply set both these values corresponding to the server on the client. The URI is usually `http://localhost:11311` and the IP address can be easily found via a network utility. However, instead of localhost we provide the actual IP address of the server running the `master` node. This enables the master node to easily let other nodes register to it and transfer data among themselves.

After completing the setup once we started recording results, they quickly became really interesting. Initially we noticed a huge drop of data in terms of dropped depth frames. However, gesture and hand data being much smaller in size as compared to the depth frames, did not show a significant loss. We speculated bandwidth restriction and numerous firewalls were the factors responsible since both the machines were initially not on the same subnet. However, upon further analysis we found out that our server was publishing data over a **100Mbps** network bandwidth despite availability of a **1Gbps** wired network bandwidth within the *Gina Cody School of Concordia University*. As soon as we switched to the latter we started getting favourable results as seen in Table 12, Table 13, Table 14 and Table 15. However, we did notice a **35% loss** of depth frame data only over the **100Mbps** network. However, in case of a **1Gbps** network we did not record any such non-negligible overhead neither did we notice significant differences between the values recorded with NiTE2.0 vs. Nuitrack. However, since we timestamped the published and received frames, we observed a minute difference between the timestamps which turned out to be an average **16ms**

delay and definitely it is network delay which may decrease on even higher bandwidth networks. Last but not the least, we collected results over message queue sizes of 0, 1, 10 and 1000. Particularly in context to ROS message queue size of 0 means almost infinite number of data buffering (to the available memory).

At the same time it is also possible to control the publishing and subscribing rate of data in ROS. This is essential to take note of since it affects the values that one will encounter in terms of number of published frames vs. actual frames captured by the sensor. If we do not control the rate of the ROS publisher node, we observed that between two frames captured by the sensor we published the same frame over and over for an approximate of 5 or 6 times at an average. Whereas, if we limit the publisher rate to **30Hz**, which is almost similar and slightly more than the sensor frame rate by a factor of 1 or 2 on an average, we do not publish duplicate frames; however, the loss of data is still negligible. Publishing duplicate data enables the client to catch the same data more than once in case previously the client was not able to catch it somehow making it more resilient. However, more bandwidth was used and also, the queue size of the client has been kept zero in all cases to buffer as much data as possible for the client as not to loose anything that arrives. As mentioned prior we have run these experiments for three minutes each due to Nuitrack free version restriction. More importantly, in NiTE2.0 capture of depth, hand, and gesture date are synchronized to the depth frame capture.

Table 12: Callbacks-per-second when message queue size is 0

Data Type	**Server**	**Client**
Depth Frame	27132	27132
Gesture	24011	24011
Hand	26676	26675

Table 13: Callbacks-per-second when message queue size is 1

Data Type	**Server**	**Client**
Depth Frame	27365	27365
Gesture	26361	26361
Hand	26632	26632

Table 14: Callbacks-per-second when message queue size is 100

Data Type	**Server**	**Client**
Depth Frame	27236	27236
Gesture	26736	26736
Hand	26003	26003

Table 15: Callbacks-per-second when message queue size is 1000

Data Type	**Server**	**Client**
Depth Frame	27314	27313
Gesture	26907	26907
Hand	26969	26969

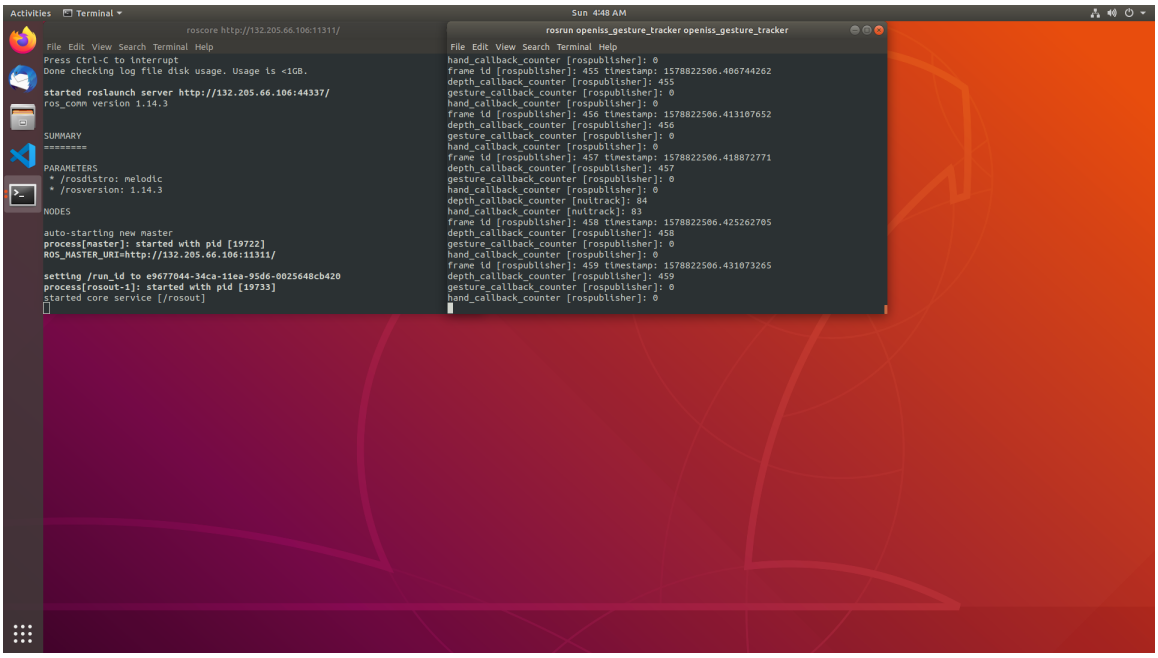


Figure 63: ROS server/publisher

Table 16: ROS overhead

Network Bandwidth	**Depth Frame Loss%**	**Gesture Frame Loss%**	**Hands Frame Loss%**
100 Mbps	35	0	0
1 Gbps	0	0	0

Both the server and the client in between an experiment can be seen in Figure 63 and Figure 64. As per this evaluation we can say that the following requirements have been evaluated qualitatively and quantitatively as applicable:

- NFR4: **The ability of a system to inter-operate with other software systems and share meaningful data.**
- NFR5: **The ability of a system to expand in a chosen dimension without major modifications to its architecture.**
- NFR6: **The ability of a system to switch between devices only minimally affecting the end user application.**

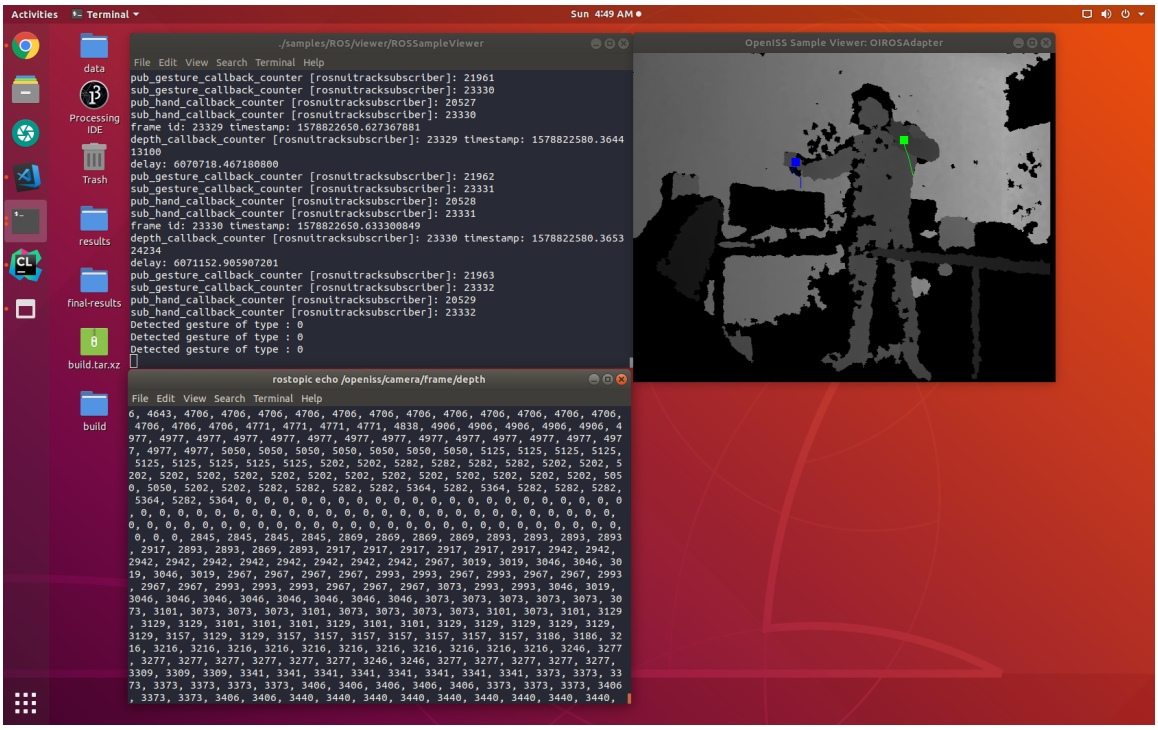


Figure 64: ROS client/subscriber

5.6 Test Game Application

In Section 3.3.3.3, page 77 we mentioned about an unrelated application that we will leverage to demonstrate the ease of use of our gesture framework API. Now, we found one such application on the **GitHub** platform that is available with a public and free to use license [135]. The application is essentially a classic 2D snake game, in which the snake grows every time it encounters a food object while moving in the four directions that is, **up, down, left and right** using a hardware controller such as the keyboard. Not only we easily replaced the keyboard controller with our Nuitrack backend, but also achieved the real-time gesture interaction in a completely unrelated application. The various direction controls were replaced using Nuitrack direction gestures. The game is built using the SDL library and uses `cmake` to build and link to dependent libraries. The game consists of the following classes:

- `Controller`
- `Snake`

- `Renderer`
- `Game`

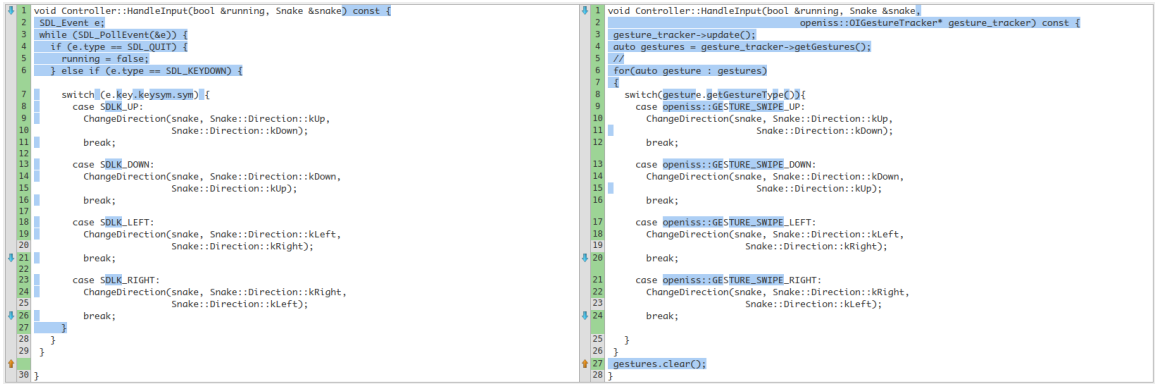
To integrate our gesture framework with this application we first started by extending the `cmake` file `CMakeLists.txt` to look for and link to Nuitrack and our gesture framework as a library. Next, we explored the code base of the game itself and find a suitable place to plug in gesture interaction via our solution. Upon initial understanding, we found that the `Snake` class itself represents the "snake" object and the `Controller` class is responsible for handling the input and hence the direction of the snake object. Whereas, the `Renderer` class takes care of the visualization and windowing via the SDL library and the `Game` class represents the game itself and manages all other aforementioned classes.

Moving on, within the `Controller` class, a method named `HandleInput` exists that uses SDL's API to control the snake via the keyboard keys. We overloaded this method and introduced our gesture framework instance as a parameter to this method and a "diff" of both the methods can be seen in Figure 65. In less than 20 lines of code we were able to replace existing keyboard controller functionality with gesture interaction. The updated or modified application is available on **GitHub** as well [136] `https://github.com/OpenISS/CppND-Capstone-Snake-Game`. This essentially wraps up our evaluations qualitatively evaluating the left out requirement that is:

- NFR2: **The ability of an Application Programming Interface provided by a system to be easy to use and learn by developers that wish to implement it.** [29, 30]

The results can be seen in Figure 66, Figure 67, Figure 68, Figure 69 where the reader can spot the snake being guided to different directions via gestures provided by Nuitrack through the **OpenISS** gesture framework. In terms of rendering frame rate the average FPS recorded is **59FPS**.

Now, qualitatively we will describe the effects of a subset of applicable Nielsen's heuristics that affected our API design [29]:



```
void Controller::HandleInput(bool &running, Snake &snake) const {
  SDL_Event e;
  while (SDL_PollEvent(&e)) {
    if (e.type == SDL_QUIT) {
      running = false;
    } else if (e.type == SDL_KEYDOWN) {

      switch (e.key.keysym.sym) {
        case SDLK_UP:
          ChangeDirection(snake, Snake::Direction::kUp,
                          Snake::Direction::kDown);
          break;

        case SDLK_DOWN:
          ChangeDirection(snake, Snake::Direction::kDown,
                          Snake::Direction::kUp);
          break;

        case SDLK_LEFT:
          ChangeDirection(snake, Snake::Direction::kLeft,
                          Snake::Direction::kRight);
          break;

        case SDLK_RIGHT:
          ChangeDirection(snake, Snake::Direction::kRight,
                          Snake::Direction::kLeft);
          break;
      }
    }
  }
}
```

```
void Controller::HandleInput(bool &running, Snake &snake,
                             openiss::OIGestureTracker* gesture_tracker) const {
gesture_tracker->update();
auto gestures = gesture_tracker->getGestures();
//
for(auto gesture : gestures)
{
  switch(gesture.getGestureType()){
    case openiss::GESTURE_SWIPE_UP:
      ChangeDirection(snake, Snake::Direction::kUp,
                      Snake::Direction::kDown);
      break;

    case openiss::GESTURE_SWIPE_DOWN:
      ChangeDirection(snake, Snake::Direction::kDown,
                      Snake::Direction::kUp);
      break;

    case openiss::GESTURE_SWIPE_LEFT:
      ChangeDirection(snake, Snake::Direction::kLeft,
                      Snake::Direction::kRight);
      break;

    case openiss::GESTURE_SWIPE_RIGHT:
      ChangeDirection(snake, Snake::Direction::kRight,
                      Snake::Direction::kLeft);
      break;

  }
}
gestures.clear();
}
```

Figure 65: Diff between existing vs updated controls

- **Visibility of system status**: In our API the various calls return a `OIStatusType` object, which helps the user to know the return status of the call.

- **Match between system and the real world**: Our API defines methods like `getGestures()` and `getHands()`, that essentially match the real world action of getting something.

- **User control and freedom**: We have API calls `init()` and `stop()`, `startGestureDetection()` and `stopGestureDetection()` to initialize, abort or reset operations.

- **Consistency and standards**: Irrespective of the adapter or middleware behind the scenes our API is consistent across applications. We also rely on the OpenNI2 standard.

- **Error prevention**: As we know our solution is a framework and it governs the control flow; hence, preventing the users from making errors by providing them a simple recipe of API calls required to be called in a specific order to enable interaction applications.

- **Recognition rather than recall**: Clear and understandable names for methods describing exactly what a method is capable of doing.

- **Flexibility and efficiency of use**: Our API is relatively compact and provides high abstraction to complex tasks making it efficient to use and flexible polymorphically.

- **Aesthetic and minimalist design**: Although an API cannot be necessarily aesthetic however consistent naming with polymorphic method names that can take different set of parameters can improve usability. However, our API is adequately abstract and minimal already.

- **Help users recognize, diagnose, and recover from errors**: Similar to our `OIStatusType` we have various log and error statements that enable user to rectify their errors by clearly stating them the issue.

- **Help and documentation**: Our various applications including the integrated sample application provide the essential knowledge on how to properly use our API.

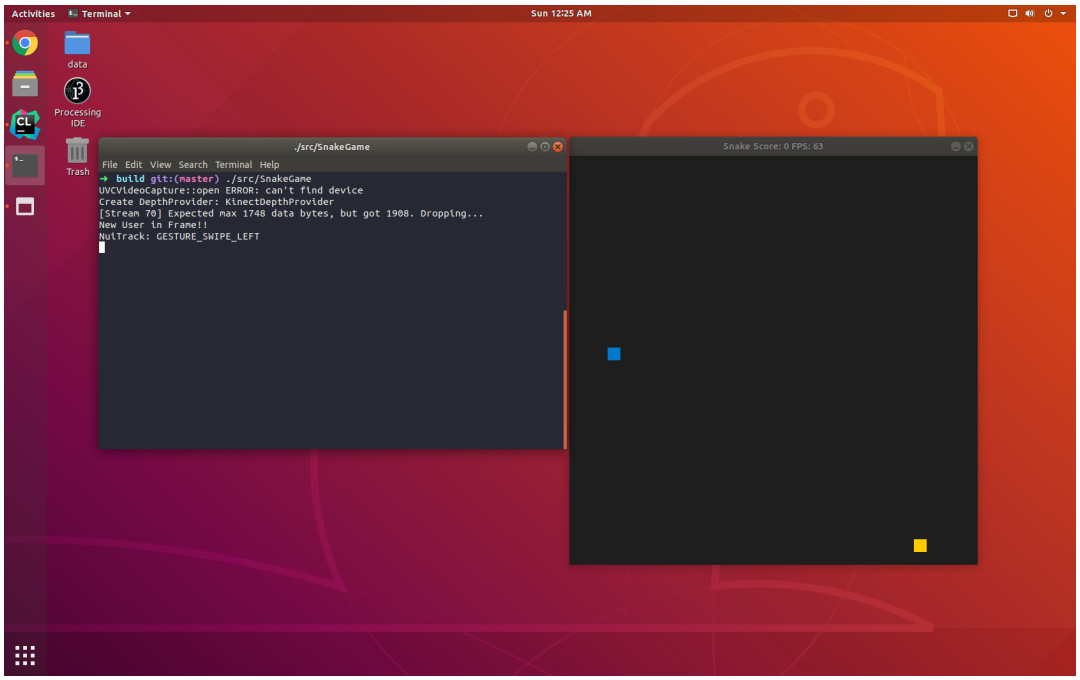


Figure 66: Gesture swipe left

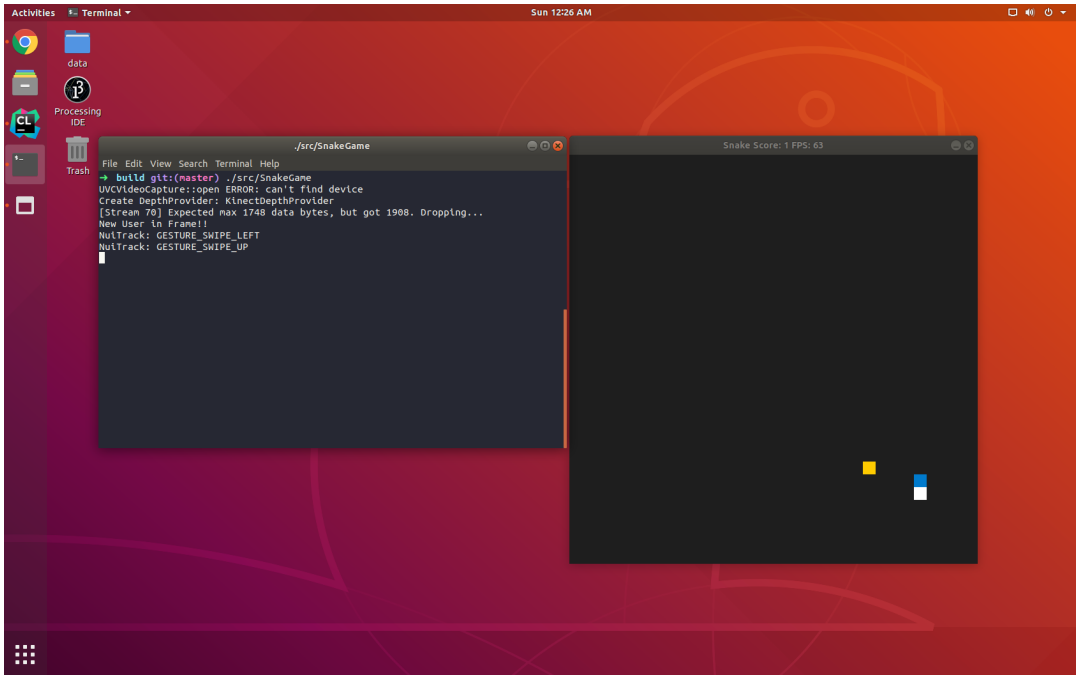


Figure 67: Gesture swipe up

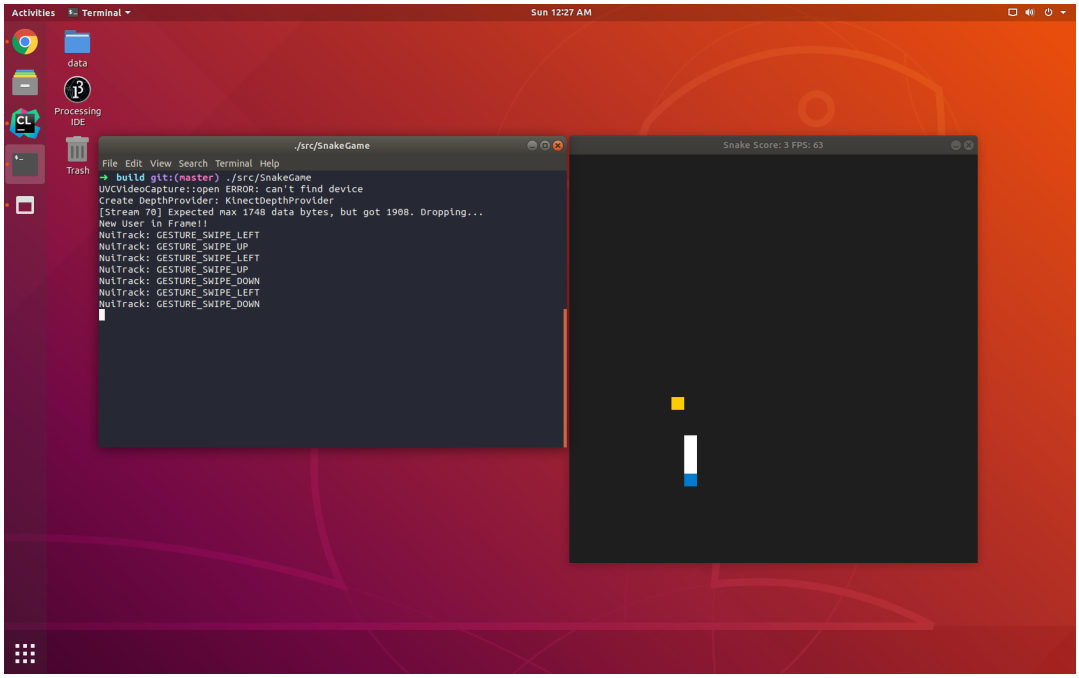


Figure 68: Gesture swipe down

5.7 Summary

In this chapter, we discussed our various evaluation results, whether qualitative or quantitative and provided their interpretations. Moreover, the evaluations done

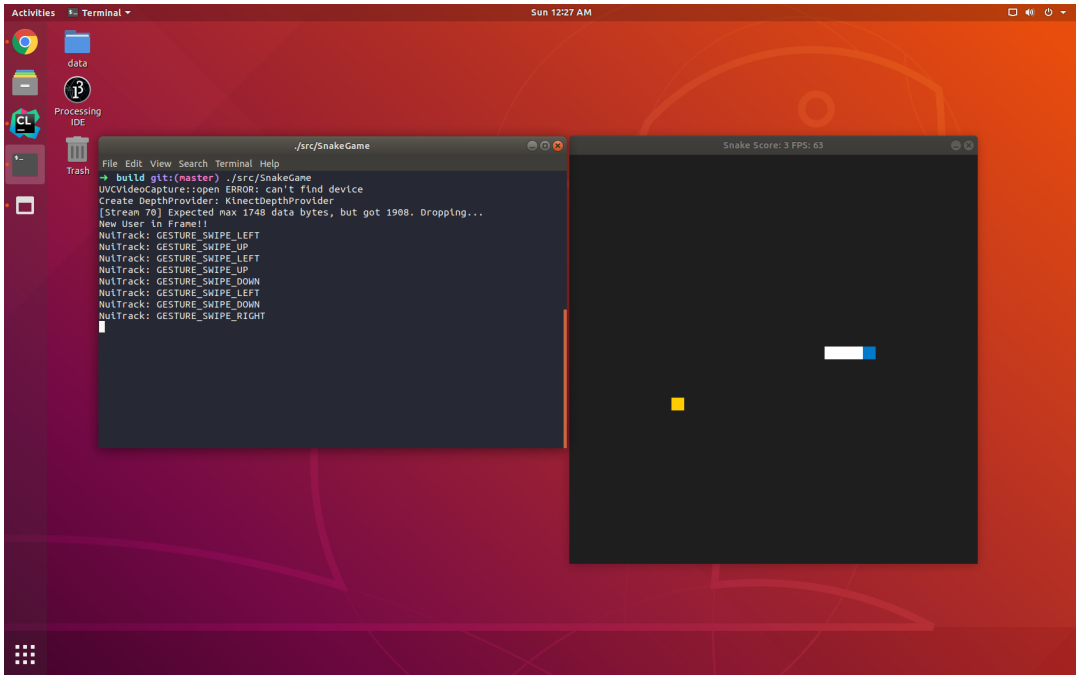


Figure 69: Gesture swipe right

enables us to answer the research questions we posed in the first chapter in Section 1.5, page 24 that we have answered in the next and the last chapter. In the next chapter we will conclude our work and describe current limitations that must be alleviated in the future work. Lastly, we can summarize that all our functional and non-functional requirements have been successfully fulfilled and evaluated qualitatively or quantitatively as applicable and illustrated in Table 17.

Table 17: Requirements evaluation summary

Requirements	Qualitative	Quantitative	Addressed
FR1	X		Section 5.2
FR2	X		Section 5.2
FR3	X		Section 5.2
FR4	X		Section 5.4
NFR1		X	Section 5.4
NFR2	X		Section 5.6
NFR3	X		Section 5.2
NFR4		X	Section 5.4, Section 5.5
NFR5	X		Section 5.5
NFR6	X		Section 5.4
NFR7	X		Section 5.4
NFR8	X		Section 5.4

Chapter 6

Conclusion and Future Work

In Chapter 1, we presented the reader with our problem statement based on our motivational scenarios in Section 1.2.2, page 6 and the various requirements elicited from these scenarios in Section 1.2.6, page 12. Then, in the next chapter Chapter 2 we provided the related work and various tools and software libraries used in this work. Next, in Chapter 3, we introduced the reader with our solution, that is an application framework, or, in other words, a gesture application framework along with the different approaches followed in the development and evaluation of this solution in the context of **OpenISS**. Then, we evaluated this framework qualitatively and quantitatively in Chapter 5 ensuring the fulfillment of all our functional and non-functional requirements with the integrated sample application built specifically for such evaluations and our applications derived from our motivational scenarios mentioned earlier. Finally, in this chapter, we conclude on our work done based upon our qualitative and quantitative evaluation results in the previous chapter in Section 6.1 and answer the various research questions that were posed in the first chapter under Section 1.5, page 24 and make recommendations. Then, we describe the various limitations of our research work that will be prospectively addressed in the near future in Section 6.2. Finally, we elaborate on the few aspects of the work that we deferred for the future in Section 6.3 respecting the scope of this research work.

6.1 Concluding Remarks

Based on our evaluation results in the previous chapter we will conclude our research work in this very section. First, we begin our concluding remarks by answering all the different research question posed in Chapter 1 under Section 1.5, page 24 one by one.

Question. *Can we design a general extensible solution that meets all our stated requirements and enable us to realize both of our motivation scenarios described in Section 1.2.2, page 6?*

Yes, we did indeed design a general and extensible solution, that is the **gesture application framework** that **enabled** us to realize both of our motivational scenarios exemplified in Section 1.2.3, page 6 and Section 1.2.4, page 8 in the form of a standalone application **DigiEVISS** (see Figure 56) and a gesture provider backend for the **ISSv2** (see Figure 59). In our solution that is the **Gesture Framework** we leverage object polymorphism, inheritance and design patterns such as the **Adapter pattern** to make it extensible among others and via requirements NFR7 and NFR8 to support two of our very different aforementioned applications.

Question. *Can we enable comparison among different gesture libraries or middleware via our solution to recommend which one is better for which application type in a uniform manner?*

Yes, we did enable comparison among NiTE2.0 and Nuitrack middleware to make recommendations and built our framework in a way that it is fairly simple to extend it further to another gesture providing middleware/libraries for making such comparisons and recommendations as well. These comparisons support decision making for choosing gesture dictionaries, device compatibility, platform compatibility, interaction effectiveness and so forth.

Question. *Specifically, which gesture library or middleware is better suited for Motivation Scenario 1, in Section 1.2.3, page 6?*

Nuitrack, although we built our application in a way that it can still work with NiTE2.0 as well, but, surely Nuitrack is better in this case. Nuitrack provides relatively larger gesture dictionary that is *six* gestures as compared to *three* in NiTE2.0, out of which only *two* work well. The application described in Section 1.2.3, page 6 is meant to be a complex application, for which Nuitrack's *six* gestures are much better suited than *three* NiTE2.0 gestures. The four direction related Nuitrack gestures (up, down, left, right) alone can be easily leveraged for enabling *navigation* within the warehouse-like 3D application via hand gestures. This is certainly not feasible when using NiTE2.0's relatively much smaller gesture dictionary without modifications. Moreover, it is quite obvious that since Nuitrack is currently being maintained and provided support for by its developers, NiTE2.0 is rather obsolete, but, both of these are closed source middleware.

Question. *Specifically, which gesture library or middleware is better suited for Motivation Scenario 2, in Section 1.2.4, page 8?*

Although, ***Nuitrack***, provides more gestures than the NiTE2.0 middleware as stated above, and at the same time it can enable creation of newer visual effects that can be used by computation artists for various stage performances by using the larger gesture dictionary it provides, we cannot currently use it in production in **ISSv2**. Nevertheless, this is the first ever solution that the Nuitrack middleware can be used in the **Processing** environment. This is quite an achievement that broadens the scope of **ISSv2** over an array of devices that Nuitrack supports. However, the core production platform of **ISSv2** resides on MacOS X and computational artists primarily use the same platform and Nuitrack does not currently run on it. Moreover, as we know Nuitrack requires a commercial license, whereas NiTE2.0 does not require one. Also, **ISSv2** does not currently require a rich gesture dictionary due to its application domain. Therefore, in this very scenario NiTE2.0 is suitable unless the **ISSv2** production is moved completely to the Linux or Windows platforms.

Question. *How much overhead does our solution and ROS introduce to the system?*

Overall, we did not observe any noticeable overhead when communicating over

a network bandwidth of **1Gbps**. However, a network bandwidth of **100Mbps** introduced a 35% loss of depth frames as seen in Table 16. Moreover, we also noticed an average network delay of **16ms** between consecutive depth frames at 1Gbps. However, gesture and hand data remain unaffected and delivered without any such measurable loss over both networks. We mentioned in our evaluation that we are publishing depth frames at a rate approximately 5 times higher than the actual device rate; however, we noticed that the trail in our integrated application stays longer when the publish rate is set to 30. The trail logically within the code is collection of 20 hand coordinate values. When we publish at much higher rate the values of the collection are updated faster and the trail gets lesser chance to fill up. Although, we did not anticipate this observation it does raise the need of testing it with more use cases which we defer to the future work.

6.2 Limitations

Currently, in our research work although we fulfilled our requirements and goals there are some present limitations that must be resolved in the near future:

- Even though our framework has no such a dependency, but the extensions points may have dependencies such as NiTE2.0 and Nuitrack middleware in our adapter classes and it is non-trivial and painful to manage these dependencies and link to them in an effective manner. Thus far, our `cmake` files can take care of these dependencies and manage them; however, it still requires installing or placing a bunch of libraries and header files or define environment variables to specific locations so that the `cmake` can help compiler to look for their declarations and the linker can see corresponding definitions. However, to overcome this issue we can provide a complete **Docker** image, but it has its own limitations of connecting with USB devices such as the depth sensors used in this research work. However, one can still easily use the **Docker** image and have pre-recorded stream played with the `fakenect` library. But, currently `fakenect` cannot work with Nuitrack. Nevertheless, we still provide our **Docker** image

that facilitates majority of the setup and is described in Appendix A. Moreover, once we created our corresponding JAVA wrappers for our framework we quickly came across the issue of making JARs and dynamic libraries portable. Normally, a runtime path guides the linker, but it must be carefully set. In short, supporting scripts or tools are required to automate this process as much as possible.

- The core of a framework is its API and we have not done any usability other than number lines of code required to get gesture recognition and hand tracking functionality in terms of ease of use. However, when we compare the effort required (in terms of methods employed) to enable gesture interaction in **Processing** our API is definitely smaller than using the NiTE2.0 and Nuitrack middleware API directly vs the **OpenISS** API.

- Our **DigiEVISS** application requires a lot of work visually and in flexibility, as it currently does not load and pre-parse FORENSIC LUCID programs or evidential objects from other sources like file storage or a database prior visualization. Once such a functionality is enabled we can work with complete data-flow graphs instead of partial ones. To compile the generated FORENSIC LUCID program we need to enable natural user interaction as well. Moreover, it is still gesture-deficient to be able to release a grabbed sphere (in case the investigator grabs a wrong one or changes his mind) as for now we are dependent on hiding/occluding the hand that is holding one.

- To provide an ideal testing environment it would have been easier to and effective to pre-record a stream with some performed gestures and tracked hands as a dataset that uniformly works with both NiTE2.0 and Nuitrack middleware. However, currently it is not possible as not only the `fakenect` footage was not available in Nuitrack, but similar tool `streamer_recorder` for `libfreenect2` or `librealsense2`'s recorder, or a corresponding one from Nuitrack. Else, we could have recorded the same footage over different devices simultaneously and then feed them to both middleware for a identical captured dataset.

- The ROS equivalent adapter for either NiTE2.0 or Nuitrack is not yet ported using **SWIG**; thus, **ISSv2** cannot interoperate with ROS presently.

- This work does not enable custom gesture tracking additions similar to Microsoft SDK for Kinect or the **OpenPose** project with a gesture dictionary of 28 gestures. Also, currently adding gestures requires adding them to our enumerations, instead of dynamically learning and naming them.

- So far we did not evaluate a server pure device ROS like original packages, with "heavy" client adapters. Essentially, this assumes the publisher is a low power device merely publishing sensor data and the processing and extraction of gesture and hand information takes place on the client side.

- Similarly, we did not test multiple devices with the ROS environment instead a simple client-server setup with one device.

- This work only provides user's hands and gestures information; however, there is yet another way to get the hand coordinates – from the skeleton joints via a skeleton tracking middleware. Both, NiTE2.0 and Nuitrack provide user and skeleton tracking, but they are more resource intensive than their hands only counterparts.

- We did not update our work in [34] to expose gesture and hand information as REST services.

- We do not provide an API for Probabilistic values for ambigious gestures, essentially, how accurately the gesture has been performed.

- Our **OpenISS** backend for **ISSv2** currently cannot play all the effects it could with `SimpleOpenNI` that require user image nor did we test it in a live performance yet. Also, the projection mapping is not working yet either as it requires scaling that we do not support currently in our backend solution.

6.3 Future Work

Our present work, that is described in this dissertation is dedicated towards *enabling* a common and extensible solution that can provide vision based gesture recognition data to create applications that require natural interaction via hand gestures such as our applications created in this research work. Even though we achieved all our goals and objectives, we did defer a few things along the way for future work that were beyond the scope of this research work at the time. Aside from addressing the limitations stated in the previous section, we list several future work items in more detail:

- Now, the first and foremost is certainly the usability evaluation of the **DigiEVISS** application with an actual forensic investigator and of the API itself by an artist for insights on their mental models using Nielsen's 10 Heuristics for Usability [32].
- Moreover, the immediate future work would be to finish the release of the framework to the open source community along with the **OpenISS** ROS package and the JAVA wrapper classes for our framework all bundled as one.
- It would be also be interesting and extremely effective to enable the user to record and save their own custom gestures.
- The **DigiEVISS** has potential to scale and load real digital evidence from a distributed storage. Currently, FORENSIC LUCID program generation is in primitive stages and does not cover a significant portion of the language itself, which possibly can be done by interacting with GIPC to get a parse tree into C++.
- Certain wrappers or plugins can be written to convert our framework's data into **Unity**-like data formats for gesture data delivery into **Unity** applications for real-time VFX and character animation rigging, like Nuitrack and `librealsense` do. It is also desirable to deliver real-time data for the same in **Blender** and **Unreal Engine** for animation specific applications.

- Integrate **OpenPose**'s gesture tracker.
- Similar to ROS, enable **OpenISS** data and features in medical applications that use OpenIGTLink protocol [137].
- Enable GIPSY's DST-based storage and server for replay and contextual image analysis and playback in time (depth frames are preserved in the DST) as a service equivalent testing of demand-driven OpenISS GIPSY backend.
- Lastly, it would be quite interesting to test the service aspect of the framework with heavy data load such as depth, color and IR (infrared) frames together in higher resolution on a larger network and multiple devices and corresponding applications.

6.4 Summary

To summarize, in this work we provide our common, flexible and extensible solution that necessarily enables gesture interaction for different yet specific application that require such functionality in **OpenISS**. To conclude we make the following recommendations based on our results and evaluations in this research work:

- For applications like **ISSv2** we recommend **OpenISS** with NiTE2.0 backend.
- For applications like **DigiEVISS** we recommend **OpenISS** with Nuitrack backend.
- For the applications involving requirements such as scalability and decoupling of sensor and application processes we recommend the **OpenISS** with ROS backend.

In the immediate future, we will release our **OpenISS** gesture framework as open source and we look forward to integrate with other systems and frameworks and enable a wider array of relevant applications in entertainment, serious games, research, computer vision, VFX, and artistic performances. Follow the project on GitHub at this URL: `https://github.com/OpenISS/OpenISS`

Bibliography

[1] M. J. Cheok, Z. Omar, and M. H. Jaward, "A review of hand gesture and sign language recognition techniques," *International Journal of Machine Learning and Cybernetics*, vol. 10, no. 1, pp. 131–153, 2019.

[2] S. A. Mokhov, M. Song, S. Chilkaka, Z. Das, J. Zhang, J. Llewellyn, and S. P. Mudur, "Agile forward-reverse requirements elicitation as a creative design process: A case study of Illimitable Space System v2," *Journal of Integrated Design and Process Science*, vol. 20, no. 3, pp. 3–37, Sep. 2016. doi: 10.3233/jid-2016-0026

[3] S. A. Mokhov, "Intensional cyberforensics," Ph.D. dissertation, Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada, Sep. 2013, online at http://arxiv.org/abs/1312.0466.

[4] S. A. Mokhov, M. Song, J. Singh, J. Paquet, M. Debbabi, and S. Mudur, "Toward multimodal interaction in scalable visual digital evidence visualization using computer vision techniques and ISS," in *Proceedings of the International Conference on Pattern Recognition and Artificial Intelligence (ICPRAI)*. CENPARMI, Concordia University, Montreal, May 2018. ISBN 978-1-895193-04-6 pp. 151–157, https://users.encs.concordia.ca/~icprai18/, arXiv:1808.00118.

[5] Nuitrack Team, "Nuitrack SDK Architecture," [online; accessed 27-May-2019], ...–2019, http://download.3divi.com/Nuitrack/doc/Architecture_page.html.

[6] ROS Wiki, "ROS Concepts," [online, accessed 27-May-2019], 2019, http://wiki.ros.org/ROS/Concepts.

[7] M. E. Markiewicz and C. J. d. Lucena, "Object oriented framework development," *Crossroads*, vol. 7, no. 4, pp. 3–9, 2001.

[8] A. Tkachuk, "3D OpenGL project for Kinect," [online], YouTube, 2016, http://youtube.com/watch?v=9PcDp5HctnQ.

[9] Interaction Design Org., "Human Computer Interaction," [online], 2019, https://www.interaction-design.org/literature/topics/human-computer-interaction. [accessed 27-May-2019].

[10] J. Coutaz, "Evaluation techniques: Exploring the intersection of HCI and software engineering," in *Software Engineering and Human-Computer Interaction*, R. N. Taylor and J. Coutaz, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1995. ISBN 978-3-540-49173-6 pp. 35–48.

[11] E. A. Buie and A. Vallone, "Integrating HCI engineering and software engineering: A call to a larger vision," in *HCI (2)*, 1997, pp. 525–530.

[12] P. Garg, N. Aggarwal, and S. Sofat, "Vision based hand gesture recognition," *World Academy of Science, Engineering and Technology*, vol. 49, no. 1, pp. 972–977, 2009.

[13] M. S. Dias, S. Gibet, M. M. Wanderley, and R. Bastos, Eds., *Gesture-Based Human-Computer Interaction and Simulation, 7th International Gesture Workshop, GW 2007, Lisbon, Portugal, May 23-25, 2007, Revised Selected Papers*, ser. LNCS, vol. 5085. Springer, 2009. doi: 10.1007/978-3-540-92865-2

[14] S. S. Rautaray and A. Agrawal, "Vision based hand gesture recognition for human computer interaction: a survey," *Artificial intelligence review*, vol. 43, no. 1, pp. 1–54, 2015.

[15] O. Wasenmüller and D. Stricker, "Comparison of Kinect v1 and v2 depth images in terms of accuracy and precision," in *Asian Conference on Computer Vision*. Springer, 2016, pp. 34–45.

[16] K. K. Biswas and S. K. Basu, "Gesture recognition using Microsoft Kinect®," in *The 5th International Conference on Automation, Robotics and Applications*. IEEE, 2011, pp. 100–103.

[17] J. P. Wachs, M. Kölsch, H. Stern, and Y. Edan, "Vision-based hand-gesture applications," *Communications of the ACM*, vol. 54, no. 2, pp. 60–71, 2011.

[18] J. Wachs, H. Stern, Y. Edan, M. Gillam, C. Feied, M. Smith, and J. Handler, "Gestix: a doctor-computer sterile gesture interface for dynamic environments," in *Soft Computing in Industrial Applications*. Springer, 2007, pp. 30–39.

[19] G. C. S. Ruppert, L. O. Reis, P. H. J. Amorim, T. F. de Moraes, and J. V. L. da Silva, "Touchless gesture user interface for interactive image visualization in urological surgery," *World Journal of Urology*, vol. 30, no. 5, pp. 687–691, 2012.

[20] Z. Zhang, "Microsoft Kinect sensor and its effect," *IEEE Multimedia*, vol. 19, no. 2, pp. 4–10, 2012.

[21] S. Fuhrmann, A. MacEachren, J. Dou, K. Wang, and A. Cox, "Gesture and speech-based maps to support use of GIS for crisis management: A user study," *AutoCarto 2005*, 2005.

[22] X. Yin and M. Xie, "Finger identification and hand posture recognition for human–robot interaction," *Image and Vision Computing*, vol. 25, no. 8, pp. 1291–1300, 2007.

[23] M. Song, S. A. Mokhov, P. Grogono, and S. P. Mudur, "Illimitable Space System as a multimodal interactive artists' toolbox for real-time performance," in *Proceedings of the SIGGRAPH ASIA 2014 Workshop on Designing Tools for Crafting Interactive Artifacts*, ser. SIGGRAPH ASIA'14. New York, NY, USA:

ACM, Dec. 2014. doi: 10.1145/2668947.2668953. ISBN 978-1-4503-3215-6 pp. 2:1–2:4.

[24] M. Song, "Computer-assisted interactive documentary and performance arts in illimitable space," Ph.D. dissertation, Special Individualized Program/-Computer Science and Software Engineering, Concordia University, Montreal, Canada, Dec. 2012, online at http://spectrum.library.concordia.ca/975072 and http://arxiv.org/abs/1212.6250.

[25] S. A. Mokhov *et al.*, "OpenISS – Open Illimitable Space System," [online], [accessed 24-Oct-2019], 2016–2020, https://github.com/OpenISS/OpenISS.

[26] B. Fry and C. Reas, "Processing – a programming language, development environment, and online community," [online], 2001–2020, http://www.processing.org/.

[27] J. Paquet, "Scientific intensional programming," Ph.D. dissertation, Department of Computer Science, Quebec City, Canada, 1999.

[28] Except on Tuesdays, "Gestures with Microsoft Kinect for Windows SDK v1.5," [online], Jul. 2012, http://blog.exceptontuesdays.com/post/27989563563/gestures-with-microsoft-kinect-for-windows-sdk-v1-5.

[29] B. A. Myers and J. Stylos, "Improving API usability," *Communications of the ACM*, vol. 59, no. 6, pp. 62–69, 2016.

[30] T. Grill, O. Polacek, and M. Tscheligi, "Methods towards api usability: a structural analysis of usability problem categories," in *International conference on human-centred software engineering*. Springer, 2012, pp. 164–180.

[31] M. Reddy, *API Design for C++*. Elsevier, 2011.

[32] J. Nielsen, "Ten usability heuristics," *useit.com*, 2005, online at http://www.useit.com/papers/heuristic/heuristic_list.html.

[33] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "Ros: An open-source robot operating system," in *ICRA workshop on Open Source software*, vol. 3, Kobe, Japan, 2009, p. 5.

[34] J. Singh, H. Lai, K. Psimoulis, P. Palmieri, I. Atanasova, Y. Chiter, A. Shirkhodaiekashani, and S. A. Mokhov, "OpenISS depth camera as a near-realtime broadcast service for performing arts and beyond," in *SIGGRAPH Asia 2018 Posters*, ser. SA '18. New York, NY, USA: ACM, 2018. doi: 10.1145/3283289.3283293. ISBN 978-1-4503-6063-0 pp. 25:1–25:2.

[35] Vicon, "Vicon Motion Capture System," [Online], 2019, https://www.vicon.com/. [accessed 27-May-2019].

[36] C. W. Irving and D. Eichmann, "Patterns and design adaptability," in *Pattern Languages of Programs*, vol. 2, 1996, pp. 1–10.

[37] Simple OpenNI Project, "Simple OpenNI – open source Processing library," [online], 2011–2013, https://code.google.com/p/simple-openni/ and https://github.com/totovr/SimpleOpenNI.

[38] S. A. Mokhov, D. Li, H. Lai, J. Singh, Y. Shen, J. Llewellyn, M. Song, and S. P. Mudur, "ISSv2 and OpenISS distributed system for real-time interaction for performing arts," in *ACM SIGGRAPH 2019 Posters*, ser. SIGGRAPH '19. New York, NY, USA: ACM, 2019. doi: 10.1145/3306214.3338539. ISBN 978-1-4503-6314-3 pp. 16:1–16:2.

[39] T. Laleh, E. Garro, J. Singh, G. Raju, M. Usman, S. A. Mokhov, and J. Paquet, "Demand-driven SOA simulation platform based on GIPSY for context-based brokerage," in *Proceedings of Service-Oriented Computing – ICSOC 2016 Workshops and Satellite Events*, ser. Lecture Notes in Computer Science, K. Drira, H. Wang, Q. Yu, Y. Wang, Y. Yan, F. Charoy, J. Mendling, M. Mohamed, Z. Wang, and S. Bhiri, Eds. Springer International Publishing, 2016. ISBN 978-3-319-68136-8 pp. 169–173, demo paper.

[40] M. Tsikkos and J. Glading, "Writing a gesture service with the Kinect for Windows SDK," [online], Aug. 2011, http://blogs.msdn.com/b/mcsuksoldev/archive/2011/08/08/writing-a-gesture-service-with-the-kinect-for-windows-sdk.aspx.

[41] M. Ladly, G. Penn, C. P. C. Chen, P. Chintraruck, M. Ghaderi, B. A. Ludlow, J. Peter, R. Tanyag, P. Zhou, and S. Kazemian, "The CBC Newsworld holodeck," in *CHI'14 Extended Abstracts on Human Factors in Computing Systems*, ser. CHI EA'14. New York, NY, USA: ACM, 2014. doi: 10.1145/2559206.2574795. ISBN 978-1-4503-2474-8 pp. 363–366.

[42] J. J. LaViola, Jr., "Context aware 3D gesture recognition for games and virtual reality," in *ACM SIGGRAPH 2015 Courses*, ser. SIGGRAPH'15. New York, NY, USA: ACM, 2015. doi: 10.1145/2776880.2792711. ISBN 978-1-4503-3634-5 pp. 10:1–10:61.

[43] N. Villaroman, D. Rowe, and B. Swan, "Teaching natural user interaction using OpenNI and the Microsoft Kinect sensor," in *Proceedings of the 2011 Conference on Information Technology Education*. ACM, 2011, pp. 227–232.

[44] M. L. Silveira, T. L. Carvalho, A. F. Neto, and T. Bastos Filho, "A multi-Kinect system for serious game development using ROS and Unity," in *XXVI Brazilian Congress on Biomedical Engineering*. Springer, 2019, pp. 585–591.

[45] S. Lantinga and the SDL Contributors, "SDL – Simple Directmedia Layer," [online], 2008–2014, http://www.libsdl.org/.

[46] S. A. Mokhov, J. Paquet, and M. Debbabi, "On the need for data flow graph visualization of Forensic Lucid programs and forensic evidence, and their evaluation by GIPSY," in *Proceedings of the Ninth Annual International Conference on Privacy, Security and Trust (PST), 2011*. IEEE Computer Society, Jul. 2011. doi: 10.1109/PST.2011.5971973. ISBN 978-1-4577-0582-3 pp. 120–123, short paper; full version online at http://arxiv.org/abs/1009.5423.

[47] H. Lai, "An OpenISS framework specialization for deep learning-based person re-identification," Master's thesis, Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada, 2019, https://spectrum.library.concordia.ca/. [accessed 9-December-2019].

[48] Y. Shen, "Toward a flexible facial analysis framework in OpenISS for visual effects," Master's thesis, Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada, 2019, https://spectrum.library.concordia.ca/.

[49] Y. Ji, "Scalability evaluation of the GIPSY runtime system," Master's thesis, Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada, Mar. 2011, http://spectrum.library.concordia.ca/7152/.

[50] A. Simard, "A framework for interoperability across heterogeneous service description models," Master's thesis, Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada, 2019, https://spectrum.library.concordia.ca/.

[51] Intel Corporation, W. Garage, and Itseez, "Itseez: Image processing algorithms," [online], 2000–2019, https://opencv.org/.

[52] Z. Song, S. A. Mokhov, M. Song, and S. P. Mudur, "Creative use of signal processing and MARF in ISSv2 and beyond," in *ACM SIGGRAPH 2018 Posters*, ser. SIGGRAPH '18. New York, NY, USA: ACM, 2018. doi: 10.1145/3230744.3230752. ISBN 978-1-4503-5817-0 pp. 4:1–4:2.

[53] S. A. Mokhov, M. Song, S. P. Mudur, and P. Grogono, "Hands-on: Rapid interactive application prototyping for media arts and performing arts in illimitable space," in *ACM SIGGRAPH 2019 Studio*, ser. SIGGRAPH '19. New York, NY, USA: ACM, 2019. doi: 10.1145/3306306.3328008. ISBN 978-1-4503-6316-7 pp. 8:1–8:33.

[54] ——, "Dataflow programming and processing for artists and beyond," in *SIGGRAPH Asia 2019 Courses*, ser. SA '19. New York, NY, USA: ACM, 2019. doi: 10.1145/3355047.3359423. ISBN 978-1-4503-6941-1 pp. 134:1–134:33.

[55] M. Song and S. A. Mokhov, "Dynamic motion-based background visualization for the *Ascension* dance with the ISS," [dance show, video], Jan. 2014, http://vimeo.com/85049604.

[56] M. Song *et al.*, "Real-time motion-based shadow and green screen visualization, and video feedback for the *Like Shadows* theatre performance with the ISS," [theatre production, video, news], Apr. 2014, http://www.concordia.ca/encs/cunews/main/stories/2014/06/04/digital-art-thatillustratesthelandofthelivingandthedead.html and http://www.concordia.ca/content/dam/encs/csse/news/docs/like-shadows-cse-academy.pdf.

[57] M. Song, S. A. Mokhov *et al.*, "Illimitable Space System at the Concordia University Virtual Touch exhibition," Eureka! Festival, Old Port, Montreal, Canada, Jun. 2014.

[58] ——, "Illimitable Space System at the Virtual Touch exhibition," Westmount Science Camp, Loyola, Concordia University, Montreal, Canada, Jul. 2014.

[59] M. Song, S. A. Mokhov, and P. Grogono, "Illimitable Space System demo," in *Poster Session Proceedings of Graphics Interface 2014*, C. Batty, Ed., May 2014, pp. 3–4, poster at GI'14, online at http://www.cs.mcgill.ca/~kry/gi2014/GI2014PosterProceedings.pdf.

[60] Cycling '74, "Max/MSP/Jitter," [online], 2005–2015, http://cycling74.com/products/max/.

[61] N. Böttcher, "An introduction to Max/MSP," [online], Medialogy, Aalborg University Copenhagen, 2007–2013, http://imi.aau.dk/~nib/maxmsp/introduction_to_MaxMsp.ppt.

[62] M. Puckette and PD Community, "Pure Data," [online], 2007–2014, http://puredata.org.

[63] Microsoft, "The Kinect for Windows SDK v. 1.5," [online], May 2012, online at http://www.microsoft.com/en-us/kinectforwindows/develop/developer-downloads.aspx and http://msdn.microsoft.com/en-us/library/hh855347.

[64] ——, "The Kinect Studio v. 1.5.0.1," [online], 2012, http://msdn.microsoft.com/en-us/library/hh855389.

[65] ——, "The Kinect for Windows Developer Toolkit v. 1.5.2," [online], Aug. 2012, http://go.microsoft.com/fwlink/?LinkId=259543.

[66] ——, "Microsoft.Kinect Namespace," [online], MSDN Library, 2012, http://msdn.microsoft.com/en-us/library/hh855419.

[67] M. Song, S. A. Mokhov, P. Grogono, and S. P. Mudur, "On a non-web-based multimodal interactive documentary production," in *Proceedings of the 2014 International Conference on Virtual Systems Multimedia (VSMM'2014)*, H. Thwaites, S. Kenderdine, and J. Shaw, Eds. IEEE, Dec. 2014. doi: 10.1109/VSMM.2014.7136675. ISBN 978-1-4799-7227-2 pp. 329–336.

[68] C. Charland, M. Dörfelt, J. Echelman, A. Koblin, M. Song, S. A. Mokhov, and P. Grogono, "Demo hour," *Interactions*, vol. 21, no. 4, pp. 8–11, Jul. 2014. doi: 10.1145/2621929

[69] M. Song, S. A. Mokhov, S. P. Mudur, and P. Grogono, "Rapid interactive real-time application prototyping for media arts and stage performance," in *ACM SIGGRAPH Asia 2015 Courses*, ser. SIGGRAPH Asia'15. New York, NY, USA: ACM, 2015. doi: 10.1145/2818143.2818148. ISBN 978-1-4503-3924-7 pp. 14:1–14:11.

[70] M. Song, S. A. Mokhov, J. Thomas *et al.*, "Dynamic motion-based background visualization for the *Gray Zone* dance with the ISSv2," [dance show, video], Feb. 2015, https://vimeo.com/121177927.

[71] M. Song, S. A. Mokhov, J. Chaffarod *et al.*, "Dynamic motion-based visualization for the *District 3 Demo Day* with the ISSv2 and Processing," [demo, video], Jun. 2015, https://vimeo.com/130122925 and https://vimeo.com/129692753.

[72] S. A. Mokhov, K.-F. Yiu, B. Ye, J. Zhang, H. Lai, and M. Song, "Real-time motion capture for performing arts and stage," [online], TEDxConcordia, Sep. 2017, https://www.youtube.com/watch?v=YgwnEmHFwI8.

[73] S.-S. Bardakjian, M. Song, S. A. Mokhov, and S. P. Mudur, "ISSv3: From human motion in the real to the interactive documentary film in AR/VR," in *Proceedings of the SIGGRAPH ASIA 2016 Workshop on Virtual Reality Meets Physical Reality*, ser. VR Meets PR 2016. New York, NY, USA: ACM, Dec. 2016. doi: 10.1145/2992138.2992139. ISBN 978-1-4503-4548-4/16/12

[74] M. Song, S. A. Mokhov, S. P. Mudur, and J.-C. Bustros, "Demo: Towards historical sightseeing with an augmented reality interactive documentary app," in *Proceedings of the 2015 IEEE Games Entertainment Media Conference (GEM 2015)*, E. G. Bertozzi, B. Kapralos, N. D. Gershon, and J. R. Parker, Eds. IEEE, Oct. 2015. doi: 10.1109/GEM.2015.7377249. ISBN 978-1-4673-7452-1 pp. 16–17.

[75] Motion Arcade, Inc., "The Zigfu Development Kit: Apps with Kinect in HTML5/JavaScript, Unity3D and Flash," [online], 2012–2013, http://zigfu.com/en/zdk/overview/.

[76] B. Baron, C. Brady, R. Gentile, G. Pereyra, J. Mulkin, D. Carrol, L. Spiker, B. Hedayati, C. Phillips, R. Martinez, M. Roy, A. Rader, C. Smith, and N. Robbins, "Initial OpenISS C API and libraries integration," [online],

CSI230, Serguei Mokhov, 2016–2018, https://github.com/OpenISS/OpenISS/tree/master/src/api/c.

[77] K. Psimoulis, P. Palmieri, I. Taushanova-Atanasova, Y. Chiter, A. Shirkhodaei, N. Golabian, M.-A. Eghtesadi, B. Hedayati, P. Annamalai, and A. Laramee, "OpenISS web services API implementation for OpenISS-as-a-service," [online], SOEN487 Team 10 and Team 11, Serguei Mokhov, Apr. 2018, https://github.com/OpenISS/OpenISS/tree/master/src/api/java.

[78] C. Wang, "OpenISS extension with new gesture and AI integration," Tech. Rep. COMP6971-S19, 2019, project report.

[79] J. Paquet and P. G. Kropf, "The GIPSY architecture," in *Proceedings of Distributed Computing on the Web*, ser. Lecture Notes in Computer Science, P. G. Kropf, G. Babin, J. Plaice, and H. Unger, Eds., vol. 1830. Springer Berlin Heidelberg, 2000. doi: 10.1007/3-540-45111-0_17 pp. 144–153.

[80] S. A. Mokhov, "Forensic Lucid encoded examples," [online], 2018–2020, https://github.com/smokhov/atsm/tree/master/examples/flucid.

[81] M. Ledwidge, "A Clever Label," SIGGRAPH Asia 2019: Real-Time Live! Program, Brisbane, Australia, Nov. 2019, http://sa2019.conference-program.com/presentation/?id=real_110&sess=sess230.

[82] Wikipedia, "Sign Language," [onlne, accessed 27-May-2019], 2019, https://en.wikipedia.org/wiki/Sign_language.

[83] B. A. Myers, "A brief history of human computer interaction technology," *interactions*, vol. 5, no. 2, pp. 44–54, 1998.

[84] X. W. Sha, "Resistance is fertile: Gesture and agency in the field of responsive media," *Configurations*, vol. 10, no. 3, pp. 439–472, 2002.

[85] B. Polson, "Pipeline design patterns," in *ACM SIGGRAPH 2015 Courses*, ser. SIGGRAPH'15. New York, NY, USA: ACM, 2015. doi: 10.1145/2776880.2792724. ISBN 978-1-4503-3634-5 pp. 21:1–21:59.

[86] P. Premaratne, "Historical development of hand gesture recognition," in *Human Vomputer Interaction Using Hand Gestures*. Springer, 2014, pp. 5–29.

[87] M. Turk, "Gesture recognition," [online; accessed 12-Jan-2020], 2016, https://link.springer.com/referenceworkentry/10.1007/978-0-387-31439-6_376.

[88] ——, *Gesture Recognition*. Boston, MA: Springer. US, 2014, pp. 346–349. ISBN 978-0-387-31439-6

[89] A. Davison, *Kinect open source programming secrets: Hacking the Kinect with OpenNI, NITE, and Java*. McGraw-Hill New York, 2012.

[90] OpenKinect Developers, "The OpenKinect project," [online, accessed 27-May-2019], 2012–2019, http://openkinect.org.

[91] Microsoft, "Microsoft Kinect," [online; accessed 27-May-2019], 2012–2019, https://support.xbox.com/en-US/browse/xbox-360/getting-started/Kinect.

[92] Wikipedia, "Kinect — Wikipedia, The Free Encyclopedia," [online; accessed 4-February-2012], 2012, http://en.wikipedia.org/w/index.php?title=Kinect&oldid=474626703.

[93] OpenKinect Contributors, "OpenKinect: Open source drivers for kinect v1," [online, accessed 27-May-2019], 2011–2019, http://openkinect.org.

[94] L. Xiang, F. Echtler, C. Kerl, T. Wiedemeyer, Lars, hanyazou, R. Gordon, F. Facioni, laborer2008, R. Wareham, M. Goldhoorn, alberth, gaborpapp, S. Fuchs, jmtatsch, J. Blake, Federico, H. Jungkurth, Y. Mingze, vinouz, D. Coleman, B. Burns, R. Rawat, S. Mokhov, P. Reynolds, P. Viau, M. Fraissinet-Tachet, Ludique, J. Billingham, and Alistair, "libfreenect2: Release 0.2," Apr. 2016.

[95] Wikipedia, "Structured Light," [online; accessed 27-May-2019], 2019, https://en.wikipedia.org/wiki/Structured_light.

[96] ——, "Time of flight," [online; accessed 27-May-2019], 2019, https://en.wikipedia.org/wiki/Time_of_flight.

[97] OpenKinect, "OpenNI2-Freenect Driver," [online; accessed 27-May-2019], 2019, https://github.com/OpenKinect/libfreenect/tree/master/OpenNI2-FreenectDriver.

[98] Apple, Inc., "Apple," [online; accessed 27-May-2019], ...–2019, https://www.apple.com/.

[99] Wikipedia, "PrimeSense," [online; accessed 27-May-2019], 2005, https://en.wikipedia.org/wiki/PrimeSense.

[100] Occipital, "Occipital," [online; accessed 27-May-2019], ...–2019, https://occipital.com/.

[101] ——, "OpenNI 2 SDK binaries and docs," [online; accessed 27-May-2019], 2019, https://structure.io/openni.

[102] ——, "Structure sensor," [online; accessed 27-May-2019], ...–2019, https://structure.io/structure-sensor.

[103] S. Falahati, *OpenNI Cookbook*. Packt Publishing Ltd, 2013.

[104] S. Ravi, "Hand gesture recognition using OpenNI+ROS," [online; accessed 27-May-2019], 2014, https://www.youtube.com/watch?v=mc7qEY60nXY.

[105] B. Hoffman and K. Martin, "CMake," in *AOSA, Volume I*, A. Brown and G. Wilson, Eds. aosabook.org, Mar. 2012, vol. I, http://aosabook.org/en/cmake.html.

[106] A. Brown and G. Wilson, Eds., *The Architecture of Open Source Applications: Elegance, Evolution, and a Few Fearless Hacks*. aosabook.org, Mar. 2012, vol. I, online at http://aosabook.org.

[107] C. Reas and B. Fry, "Processing: programming for the media arts," *AI & SOCIETY*, vol. 20, no. 4, pp. 526–538, Sep. 2006. doi: 10.1007/s00146-006-0050-9

[108] M. Lefebvre, G. Lévesque, R. Pettigrew, J. Segal, and B. Z. Leroux, "Dreaming now," [Théâre Youtheatre show; video], Feb. 2016, http://accesculture.com/activite/Dreaming_now and https://vimeo.com/81609646.

[109] S. A. Mokhov, M. Song, A. Kaur, M. Talwar, K. Gudavalli, and S. P. Mudur, "Managing data and artifacts between software engineers and artists: an issv2 case study," in *Proceedings of the 21st International Database Engineering & Applications Symposium, IDEAS 2017, Bristol, United Kingdom, July 12-14, 2017*, B. C. Desai, J. Hong, and R. McClatchey, Eds. ACM, 2017. doi: 10.1145/3105831.3105862. ISBN 978-1-4503-5220-8 pp. 6–13.

[110] J. Parker, *Introduction to Game Development Using Processing*, Jun. 2015. ISBN 978-1937585402

[111] M. Song *et al.*, "LED matrix demonstration (enhanced)," [online], 2016, https://vimeo.com/157863706.

[112] SWIG Community, "SWIG wiki," [online; accessed 27-May-2019], 2019, https://github.com/swig/swig/wiki.

[113] Klaus Kaempf, "Generating language bindings for C/C++ libraries," [online; accessed 27-May-2019], 2008, https://en.opensuse.org/images/e/eb/Kkaempf_KnowledgeSharing_Swig.pdf.

[114] D. M. Beazley *et al.*, "SWIG: An easy to use tool for integrating scripting languages with C and C++," in *Tcl/Tk Workshop*, 1996, p. 43.

[115] D. M. Beazley, "Automated scientific software scripting with SWIG," *Future Generation Computer Systems*, vol. 19, no. 5, pp. 599–609, 2003.

[116] SWIG Community, "SWIG 4.0 documentation," [online; accessed 27-May-2019], 2019, http://www.swig.org/Doc4.0/index.html.

[117] D. M. S. Beazley, "SWIG master class," [online; accessed 27-May-2019], 2008, http://www.dabeaz.com/SwigMaster/SWIGMaster.pdf.

[118] Docker, Inc., "Docker," [online; accessed 27-May-2019], 2019, https://www.docker.com/resources/what-container.

[119] Wikipedia, "Inversion of control," [online], 2020, https://en.wikipedia.org/wiki/Inversion_of_control.

[120] R. E. Johnson and B. Foote, "Designing reusable classes," *Journal of Object-Oriented Programming*, vol. 1, no. 2, pp. 22–35, 1988.

[121] W. Pree, "Meta patterns – a means for capturing the essentials of reusable object-oriented design," in *European Conference on Object-Oriented Programming*. Springer, 1994, pp. 150–162.

[122] Wikipedia, "Dependency injection," [online], 2019, https://en.wikipedia.org/wiki/Dependency_injection.

[123] K. S. Al-Tahat, S. B. Idris, T. M. T. Sembok, and M. Yousof, "Using hot-spot-driven approach in the development of a framework for multimedia presentation on the web," *Framework*, vol. 2, no. B1, p. A1, 2002.

[124] M. Fayad and D. C. Schmidt, "Object-oriented application frameworks," *Communications of the ACM*, vol. 40, no. 10, pp. 32–38, 1997.

[125] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995, ISBN: 0201633612.

[126] N. Wirth, "Program development by stepwise refinement," in *Pioneers and Their Contributions to Software Engineering*. Springer, 2001, pp. 545–569.

[127] Vicon, "Vicon FAQ: How can i measure real-time latency," [online, accessed 27-May-2019], 2019, https://www.vicon.com/faqs/hardware/how-can-i-measure-real-time-latency.

[128] D. Riehle, "Framework design: A role modeling approach," Ph.D. dissertation, ETH Zurich, 2000.

[129] Wikipedia, "Model–view–controller," [online], 2020, https://en.wikipedia.org/wiki/Model\T1\textendashview\T1\textendashcontroller.

[130] M. Song, "Tangible Memories: Multi-dimensional interactive documentary presentation," Apr. 2011, [Concordia University Humanities Doctoral Student Annual Conference]; http://dislocationsconference.wordpress.com/schedule/.

[131] Miao Song (Director), "Tangible memory bubbles," [online], Oct. 2012, https://vimeo.com/51329588.

[132] M. Song, P. Grogono, J. Lewis, and M. J. Simmonds, "A poor woman's interactive remake of the "I Still Remember" documentary with OpenGL," in *Proceedings of ICEC 2011*, ser. LNCS, J. Anacleto, S. Fels, N. Graham, B. Kapralos, M. S. El-Nasr, and K. Stanley, Eds., no. 6972. Springer, Oct. 2011. doi: 10.1007/978-3-642-24500-8_42. ISBN 978-3-642-24499-5 pp. 362–366.

[133] Y. Mao, C. Ling, and C. Wang, "OpenISS: HCI for forensic investigators," Tech. Rep. INSE6610-S19, 2019, project report.

[134] Intel, "Intel RealSense SDK2.0," [online; accessed 12-Jan-2020], 2020, https://github.com/IntelRealSense/librealsense.

[135] Udacity, "Snake 2D game," [online; accessed 27-May-2019], 2019, https://github.com/udacity/CppND-Capstone-Snake-Game.

[136] Jashanjot Singh, "Snake 2D game," [online; accessed 27-May-2019], 2019, https://github.com/jashanj0tsingh/CppND-Capstone-Snake-Game.

[137] J. Tokuda, G. S. Fischer, X. Papademetris, Z. Yaniv, L. Ibanez, P. Cheng, H. Liu, J. Blevins, J. Arata, A. J. Golby, T. Kapur, S. Pieper, E. C. Burdette, G. Fichtinger, C. M. Tempany, and N. Hata, "OpenIGTLink: an open network protocol for image-guided therapy environment," *The International Journal of Medical Robotics and Computer Assisted Surgery*, vol. 5, no. 4, pp. 423–434, 2009. doi: 10.1002/rcs.274

Index

API
(void*), 98
adaptDepthData(), 103
adaptGestureData(), 103
adaptHandData(), 103
ApplicationRenderer, 111
Camera, 111
Controller, 142, 143
convertHandCoordinatesToDepth(), 91
create(), 99
drawTrail(), 109
Face, 65
fby, 117
Frame, 65
Game, 143
Gesture, 65
GESTURE_CLICK, 92, 93, 95, 125
GESTURE_DEFAULT, 92
GESTURE_HAND_RAISE, 92, 95, 125
GESTURE_IS_ABSENT, 94
GESTURE_IS_COMPLETE, 94
GESTURE_IS_IN_PROGRESS, 94
GESTURE_PUSH, 93, 99, 127
GESTURE_SWIPE_DOWN, 93, 99, 127
GESTURE_SWIPE_LEFT, 92, 99, 126
GESTURE_SWIPE_RIGHT, 92, 99, 126
GESTURE_SWIPE_UP, 93, 99, 127
GESTURE_WAVE, 92, 93, 95, 125
GESTURE_WAVING, 93, 98, 126
GestureData, 47
getDepth(), 69
getDepthFrame(), 90
getGestures(), 69, 84, 90, 98, 144
getHands(), 69, 84, 90, 98, 144
GL_POINT, 109
Hand, 65
HAND_IS_ABSENT, 93
HAND_IS_LOST, 93
HAND_IS_NEW, 93
HAND_IS_TOUCHING_FOV, 93
HAND_IS_TRACKING, 93
HandData, 47
HandleInput, 143

HandTracker, 47, 96
HandTrackerFrameRef, 47, 96
init(), 84, 89, 97, 99, 100, 108, 144
javaout, 106
jni, 106
jstype, 106
jtype, 106
Leap Motion, 25
libfreenect, 134
LicenseNotAcquiredException, 129
master, 136, 139
Message, 100
messages, 137
NewFrameListener, 96
nite::HandTracker, 97, 98
nite::HandTrackerFrameRef, 97
nite::NiTE::initialize(), 97
nite::NiTE::shutdown(), 97
NodeHandle::advertise(), 103
OIDepthFrame, 88, 101, 106, 107
OIDevice, 97
OIGestureData, 88, 90, 93
OIGestureState, 93
OIGestureTracker, 87, 88, 95, 96, 99, 103, 106
OIGestureType, 92
OIGetsureData, 101
OIGetsureTracker, 95, 98
OIHandData, 88, 90, 93, 101
OIHandState, 93
OINiTEGestureTracker, 95, 96, 98, 99
OINuitrackGestureTracker, 98, 99
OIStatusType, 91, 144, 145
onNewDepthFrame(), 109
onNewFrame(), 96, 97
OpenGL, 66, 89, 111, 123
openni::VideoFrameRef::getData(), 98
OpenPose, 25
out, 106
override, 96, 99
Renderer, 143
renderTexture, 109
ros::Publisher, 103
ros::Subscriber, 103
ROS_IP, 139
ROS_MASTER_URI, 139
ROSAdapter, 102
roscore, 109, 136
roscpp, 53, 87, 100–102, 110
rospy, 53
rosrun, 110, 137
rostopic, 137
rpath, 107
Scene, 111
SimpleOpenNI, 8, 9, 22, 32, 68, 73, 78, 95, 96, 119, 120, 134, 153
Skeleton, 65, 129
Snake, 142, 143

Sphere, 116
sphere_data, 116
sphere_id, 116, 131
startGestureDetection(), 84, 89, 97, 108, 144
startHandTracking(), 84, 90
STATUS_DEFAULT, 92
STATUS_FAIL, 92
STATUS_OK, 92
STATUS_TRACKER_UNINITIALZED, 92
std::string, 117
std::vector, 84, 90, 105, 106, 109
std_vector.i, 106
stop(), 84, 89, 97, 100, 144
stopGestureDetection(), 84, 89, 144
stopHandTracking(), 84, 90
swigCMemOwn, 57
swigCPtr, 57
tdv::nuitrack::Nuitrack::init(), 99
tdv::nuitrack::Nuitrack::run(), 100
tdv::nuitrack::Nuitrack::update(), 100
typemaps, 105
typemaps.i, 106
uint16_t, 98
update(), 84, 89, 90, 98, 100, 109
User, 65
UserTracker, 47
virtual, 87

Background, 28

C, 34, 35, 54–58, 104, 106
C++, 20, 34, 35, 48, 51, 54–58, 65, 84, 87, 89, 94, 97, 104–106, 120, 132, 133, 154
C#, 5, 31, 49, 55
Conclusion and Future Work, 148

DigiEVISS, iii, 6, 7, 23, 31, 38, 72, 77, 110, 112–115, 129–132, 134, 149, 152, 154, 155
DST, 155

Evaluation, 122

Files
.pde, 120
build, 52
catkin_ws, 52, 87
cmake-modules/, 86, 87
CMakeLists.txt, 87, 143
code/, 120
devel, 52
Dockerfile, 58
home/, 120
include/, 85, 87
interface.i, 56
interface_wrap.cxx, 56
library.h, 56
library.java, 57

msg/, 87
package.xml, 52, 87
samples/, 86, 87
simple-flucid-program.ipl, 117
sketchbook/, 120
src, 52
src/, 85, 87
swig/, 86, 87
wrapper.java, 57
wrapperJNI.java, 57, 107
Forensic Lucid, iii, 5–7, 18, 22, 28, 31, 35–40, 42, 72, 77, 110, 112–118, 131, 132, 152, 154
Framework Design and Instantiation, 80
Frameworks
GEE, 35, 36
GIPC, 7, 35, 36, 38, 42, 112, 117, 132, 154
RIPE, 35

GEE, 35, 36
GIPC, 7, 35, 36, 38, 42, 112, 117, 132, 154
GIPSY, 27, 35, 36, 39, 41, 42, 155

Haskell, 51

Illimitable Space System, 4, 5, 8, 14, 21, 22, 28–30, 32–34, 68
DigiEVISS, iii, 6, 7, 23, 31, 38, 72, 77, 110, 112–115, 129–132, 134, 149, 152, 154, 155
Forensic Lucid, iii, 6, 7, 23, 31, 38, 72, 77, 110, 112–115, 129–132, 134, 149, 152, 154, 155
OpenISS, iii, 4, 9, 21–23, 28, 30, 33–35, 42, 64–66, 71, 72, 78, 81–84, 86, 87, 100, 102–104, 107–109, 113, 118–120, 131, 134, 136, 137, 143, 148, 152–155
Introduction, 1

Java, 20, 34, 36, 51, 54, 55, 57, 58, 65, 69, 72, 77, 86, 95, 104–107, 118, 120, 122, 152, 154
JavaScript, 34, 55

Kinect v1, 8, 21, 25, 44, 45, 50, 66, 128, 131
Kinect v2, 8, 21, 25, 32, 44, 45, 50, 66, 128, 131

Libraries
OpenGL, 66, 89, 111, 123
LISP, 51
Lucid, 5, 28, 35–38

Methodology, 59

NiTE2.0, 13, 15, 16, 22, 25, 32, 45–49, 65–73, 76, 77, 80–85, 89, 90, 92–100, 104, 105, 111, 120, 124–131, 133, 136, 138–140, 149–153, 155
Nuitrack, 13, 15, 22, 25, 26, 33, 48–50, 65–73, 76, 77, 80–85, 89, 90, 92–94,

98–100, 104, 105, 111, 120, 123–131, 133–136, 138–140, 142, 143, 149–155

OpenCV, 66
OpenGL, 66, 89, 111, 123
OpenISS, iii, 4, 9, 21–23, 28, 30, 33–35, 42, 64–66, 71, 72, 78, 81–84, 86, 87, 100, 102–104, 107–109, 113, 118–120, 131, 134, 136, 137, 143, 148, 152–155
OpenNI, 45, 46
OpenNI1.0, 46
OpenNI1.5, 46
OpenNI2.0, 32, 45–47, 69, 72, 85, 95, 120, 127, 129

Perl, 55
PHP, 55
Python, 51, 55

RIPE, 35
ROS, 17–19, 22, 25, 26, 28, 42, 50–54, 64–66, 70, 72, 73, 75, 76, 78, 80, 82, 83, 87, 89, 94, 100, 102, 122, 123, 136–140, 153–155

Tools
  add_jar, 107
  catkin, 87, 102
  cmake, 51, 86, 87, 102, 106, 107, 142, 143, 151
  fakenect, 151, 152
  libfreenect, 44, 45, 49, 81
  libfreenect2, 44, 45, 49, 81, 152
  librealsense, 154
  librealsense2, 49, 85, 152
  p5.js, 54
  streamer_recorder, 82, 152

Appendix A

Docker File

```
FROM ubuntu:bionic
#
LABEL maintainer="Jashanjot Singh" release="1.0"
#
RUN apt-get update && apt-get install -y \
    build-essential \
    cmake \
    wget \
    git \
    libusb-1.0-0-dev \
    libudev-dev \
    freeglut3-dev \
    ffmpeg \
    unzip \
    ruby-full \
    vim
# Non-ROS
RUN /bin/bash -c " cd ~ \
    && wget http://se.archive.ubuntu.com/ubuntu/pool/main/libp/libpng/libpng12-0_1.2.54-1
     ubuntu1_amd64.deb \
    && dpkg -i libpng12-0_1.2.54-1ubuntu1_amd64.deb \
    && wget http://download.3divi.com/Nuitrack/platforms/nuitrack-ubuntu-amd64.deb \
    && dpkg -i nuitrack-ubuntu-amd64.deb \
    && rm libpng12-0_1.2.54-1ubuntu1_amd64.deb nuitrack-ubuntu-amd64.deb \
    && cd ~ \
    && mkdir libs \
    && cd libs \
    && wget https://s3.amazonaws.com/com.occipital.openni/OpenNI-Linux-x64-2.2.0.33.tar.bz2 \
    && tar -xjvf OpenNI-Linux-x64-2.2.0.33.tar.bz2 \
```

```
    && wget https://sourceforge.net/projects/roboticslab/files/External/nite/NiTE-Linux-x64-2.2.tar.
     bz2 \
    && tar -xjvf NiTE-Linux-x64-2.2.tar.bz2 \
    && wget http://download.3divi.com/Nuitrack/NuitrackSDK.zip \
    && unzip NuitrackSDK.zip \
    && cp -r NuitrackSDK/Nuitrack . \
    && rm NiTE-Linux-x64-2.2.tar.bz2 OpenNI-Linux-x64-2.2.0.33.tar.bz2 NuitrackSDK.zip \
    && git clone https://github.com/OpenKinect/libfreenect.git \
    && cd ~/libs/libfreenect && mkdir build && cd build \
    && cmake .. -DBUILD_OPENNI2_DRIVER=ON \
    && make \
    && cd ~/libs/libfreenect/build/lib/OpenNI2-FreenectDriver \
    && cp * ../ \
    && cd ~/libs/libfreenect/build/lib \
    && cp -rf * ~/libs/OpenNI-Linux-x64-2.2/Redist/OpenNI2/Drivers \
    && cd ~ \
    && wget https://gist.githubusercontent.com/jashanj0tsingh/bfcc092133a1cec3f2043b6ed3b9cb30/raw
     /3415d3928ee65249d4dc79ec9e2c32d21b3d5232/env_setup_openiss.sh \
    && chmod a+x ~/env_setup_openiss.sh \
    && ./env_setup_openiss.sh \
    && cat .bashrc \
    && source .bashrc \
    # uncomment below to install opencv
    # && cd ~ \
    # && git clone https://github.com/opencv/opencv.git \
    # && cd opencv \
    # && git checkout 3.4 \
    # && mkdir build && cd build \
    # && cmake .. -L \
    # && make -j4 \
    # && make install \
    # && rm -rf ~/opencv \
    "
#
RUN bin/bash -c "cd ~ "

# ROS
# setup timezone
RUN echo 'Etc/UTC' > /etc/timezone && \
    ln -s /usr/share/zoneinfo/Etc/UTC /etc/localtime && \
    apt-get update && apt-get install -q -y tzdata && rm -rf /var/lib/apt/lists/*

# install packages
RUN apt-get update && apt-get install -q -y \
    dirmngr \
```

```
    gnupg2 \
    lsb-release \
    python3-pip \
    && rm -rf /var/lib/apt/lists/*

# setup keys
RUN apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv-keys
     C1CF6E31E6BADE8868B172B4F42ED6FBAB17C654
#
# setup sources.list
RUN echo "deb http://packages.ros.org/ros/ubuntu ‘lsb_release -sc‘ main" > /etc/apt/sources.list.d/
     ros-latest.list
# install bootstrap tools
RUN apt-get update && apt-get install --no-install-recommends -y \
    python3-rosdep \
    python3-rosinstall \
    python3-vcstools \
    && rm -rf /var/lib/apt/lists/*
#
# setup environment
ENV LANG C.UTF-8
ENV LC_ALL C.UTF-8
#
# bootstrap rosdep
RUN rosdep init \
    && rosdep update
#
# install ros packages
ENV ROS_DISTRO melodic
RUN apt-get update && apt-get install -y \
    ros-melodic-desktop \
    && rm -rf /var/lib/apt/lists/*
#
CMD ["bash"]
```

Listing A.1: Dockerfile