BACHELOR PAPER

Term paper submitted in partial fulfillment of the requirements for the degree of Bachelor of Science in Engineering at the University of Applied Sciences Technikum Wien - Degree Program TW

Development of a hand gesture recognition plugin for the AsTeRICS framework

By: Leonhard Hauptfeld

Student Number: 1510768031

Supervisor: Ing. Martin Deinhofer, MSc

Vienna, February 21, 2018

FH University of Applied Sciences
TECHNIKUM WIEN

Declaration

“As author and creator of this work to hand, I confirm with my signature knowledge of the relevant copyright regulations governed by higher education acts (see Urheberrechtsgesetz /Austrian copyright law as amended as well as the Statute on Studies Act Provisions / Examination Regulations of the UAS Technikum Wien as amended).

I hereby declare that I completed the present work independently and that any ideas, whether written by others or by myself, have been fully sourced and referenced. I am aware of any consequences I may face on the part of the degree program director if there should be evidence of missing autonomy and independence or evidence of any intent to fraudulently achieve a pass mark for this work (see Statute on Studies Act Provisions / Examination Regulations of the UAS Technikum Wien as amended).

I further declare that up to this date I have not published the work to hand nor have I presented it to another examination board in the same or similar form. I affirm that the version submitted matches the version in the upload tool.“

Vienna, February 21, 2018 Signature

Kurzfassung

Handgesten sind ein wichtiger Teil der menschlichen Körpersprache. Ihr Nutzen umfasst ein weites Feld, von Grundfunktionen wie der Assistenz von linguistischer Kommunikation, bis zu einer komplett eigenen Sprache für taube Menschen. Demnach ist die Erkennung von Handgesten ein großer Teil in dem Feld der "Computer Vision". Sie bilden oft einen wichtigen Teil von Softwarelösungen im Bereich der Assistiven Technologien (AT). Trotzdem ist mit Gestenerkennung oft eine steile Lernkurve und eine hohe Eintrittsbarriere verbunden. Sie erfordert komplexe Algorithmen, implementiert in schwierigen hardwarenahen Programmiersprachen, und normale Computer-Webcams sind den Anforderungen oft nicht gewachsen. Lösungen für diese Probleme existieren teilweise in Form von AT-Toolkits (Frameworks für die schnelle Implementierung von Lösungen) und 3D-Kameras mit Tiefensensoren. Diese Arbeit beschäftigt sich mit der Implementierung eines Handgestenerkennungs-Plugins für das AT-Toolkit "Assistive Technology Rapid Integration & Construction Set" (AsTeRICS). Die Software verwendet dafür eine 3D-Kamera ausgestattet mit Intel RealSense-Technologie. Die Funktionalität beschränkt sich auf die Erkennung einer Handform und eines Zählens der ausgestreckten Finger allerdings wird das Plugin offen für Erweiterungen implementiert. In der Einführung wird ein kurzer Überblick über State-of-the-art-Technologien passend für die Implementierung gegeben. Anschließend wird eine Methode für die Konzeptionierung und Implementierung der Software entwickelt und angewendet. Das resultierende Plugin wird im Rahmen einer kleinen Test-AT-Lösung auf die Funktionalität hin evaluiert. Die angewendete Implementierungsmethode wird analysiert und Verbesserungsmöglichkeiten sowie aufgetretene Fehler in der Konzeptionierung werden in Ratschläge für weitere Entwicklung der Methode intepretiert. Zuletzt werden Zukunftsmöglichkeiten für die Software wie z.B. mögliche Verbesserungen aufgelistet.

Schlagworte: Assistive, Technologie, Software, Integration, Einbindung, Bewegungserkennung, Computer Vision

Abstract

Hand gestures are an important part of human body language. Their uses range from assisting verbal expression to a complete form of communication for the deaf. As a result, recognition of hand gestures by a machine is a large segment of computer vision research and often an important part of assistive technology (AT) solutions. However, there is a steep learning curve and high barrier of entry associated with gesture recognition. The processing requires complex algorithms programmed in difficult low-level programming languages and normal color cameras such as webcams are often ill suited for the task. Solutions for these problems partly exist in the form of AT-Toolkits that provide frameworks for quick implementation of assistive solutions and advanced computer cameras with the ability to not only capture images but also depth information. This paper pursues the implementation of a hand gesture recognition plugin for the AT-Toolkit known as "Assistive Technology Rapid Integration & Construction Set" (AsTeRICS) utilizing a depth camera equipped with Intel RealSense technology. The functionality of this plugin is limited to recognizing the shape of a hand and detecting the number of extended fingers on it. The software is built with expandability in mind. In the introduction, a short overview of state-of-the-art technologies suited for this task is given. A method for approaching and executing the implementation is created. This method is applied and the resulting plugin is evaluated for its functionality in an example AT solution. The method of implementation is discussed and flaws uncovered in it are interpreted into advice for approaching further plugin development. Finally, the future prospects for the developed software are listed.

Keywords: assistive, technology, software, integration, motion recognition, computer vision

Acknowledgements

I would like to thank my Supervisor Ing. Martin Deinhofer for supporting the project through my continued communications blackouts and odd scheduling. I would also like to thank the developers of elementaryOS, MinGW64 and CMake for making my development environment pleasant to work in. Additionally, I would like to thank the local pizzeria for being open until 1AM.

Finally, I would like to extend my gratitude to FH Technikum Wien for providing me with the means to learn about, study, and experiment with topics that interest me and that I want to make my life about.

Contents

1 Introduction

1.1 Premise

Body language is a way of communication natural to most beings in nature, including humans. Even though we have mastered the use of verbal language, written language, and even LaTeX, a big part of communication when physically interacting with another person is done using body language. This was in part researched and demonstrated by Paul Watzlawick in the mid-20th century[1], resulting in five axioms describing language, the first of which is "One cannot not communicate"[2], meaning in part that body language is always present.

Hands make up a portion of that language. As such, it is natural to aid verbal communication with hand gestures to express emphasis or emotion. As part of the quest to make interaction with a computer more natural, fields of research and software development have developed to incorporate such technology into applications that need it. One of these uses lies in the field of assistive technologies, where it can be used to help people with certain disablities convey their meaning to a computer by movement of hand or fingers.

Development of such applications and integration into AT-Solutions is no easy task due to the complexity of both hand recognition and integration into AT-solutions. For this reason, libraries like OpenCV are being developed to ease development of computer vision software and AT-Toolkits like the Assistive Technology Rapid Integration & Construction Toolkit created. These advancements allow for rapid prototyping and implementation of assistive solutions.

The AsTeRICS framework is made up of sensor, actor and processor components that perform the duties they were designed to do independent of one another while exchanging data through a certain set of rules set by the AsTeRICS toolkit. The framework lacks a sensor module for Gesture Recognition using an Intel RealSense 3D camera and as such this is the software being researched and developed in this paper. The resulting component should be able to track the number of extended fingers on a hand shown to the sensor to a reasonable degree of accuracy.

It starts with a list of state of the art technologies that can be used to accomplish this task. Then, 3 integration methods are developed that, applied step by step, help to create a concept of the plugin that is later implemented. These methods are for analyzing the target software, choosing the right technology to implement the add-in components in and finally combining the findings to find a suitable way of attachment to the target software. Additionally, methods for choosing ways to implement a User Interface and development environment are developed.

In the following implementation the methods are applied to create the concept for the AsTeRICS Intel RealSense gesture recognition plugin. It consists of a native library written in C++

utilizing OpenCV and librealsense to recognize the gestures, the implementation of which is described first. Then, implementation of the Java plugin utilizing the Java Native Interface to exchange data with the library is described.

Finally, the results of this implementation are evaluated in two use cases, left clicking by making a fist and entering numbers using the number of extended fingers on a hand. Flaws like fluctuations in the resulting number are collected.

The final conclusion lists some improvements for the conceptualization methods found through application to the component and ends with a list of possible improvements to it.

1.2 State of the art

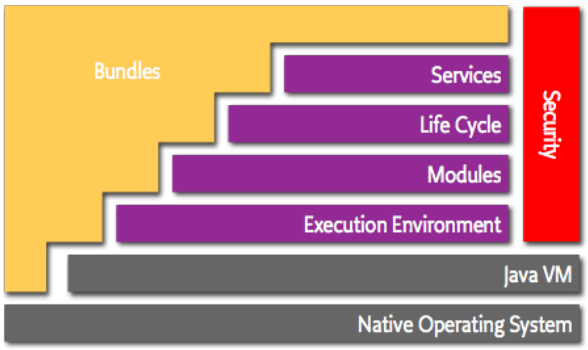
1.2.1 AT-Frameworks

AsTeRICS is a framework for building AT-Solutions that is based entirely on different plug-ins and their interaction. Possible plug-in types are sensors and actors. It is written in Java, utilizing native C++ libraries where necessary or advantageous.

OSGi

The runtime application is implemented as a framework utilizing the OSGi platform concept by the OSGi Alliance. It therefore provides a middleware layer to all the sensor and actor plugins that they can use to connect to the main application. These plugins represent the component model and provide their services to the main application over the middleware library. The assembled software is provided as a bundle, consisting of all the seperate components and the middleware. The OSGi layer architecture is shown in Figure 1.

Figure 1: The OSGi architecture (Source: [3])



1.2.2 Computer Vision

Various computer vision libraries for research exist, but OpenComputerVision (OpenCV) has established itself as the dominant library used for most computer vision applications, becoming almost synonymous for the technology. It contains a vast selection of structures and algorithms for loading, storing and transforming computer vision relevant data.

OpenCVs core is implemented in C, making it a very resource-efficient library. It is available for every major operating system and processor architecture. Official bindings with documentation exist for C++, Python and Java with many more unofficial ones for almost every other important programming language.

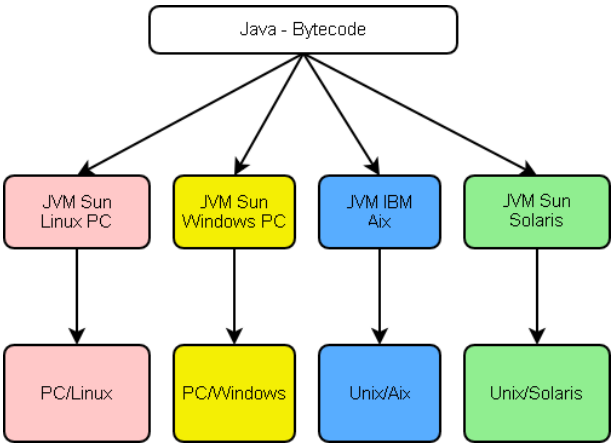
1.2.3 Programming languages

Java

Java[4] is a programming language invented by Sun Microsystems, now owned by the Oracle Corporation, which also supports its most prominent implementation. It compiles into non-native Java Bytecode that runs on a Java Virtual Machine (JVM). As a result, any compiled Java Code can run on any hardware that runs such a JVM, making Java almost entirely platform independent, as illustrated in Figure 2.

It features a very strict object oriented programming model with, in comparison with similar languages like C#, little convenience features, making it one of the more challenging programming languages for development. Additionally, the platform independence of the Java Virtual Machine means that Java programs are quite more resource intensive than those of natively executed languages[5].

Figure 2: The Java Virtual Machine (Source: [6])



Java can be coupled to native Machine Code written in C or C++, giving it additional versatility at the cost of complete platform independence. As such, bindings exist for many popular C/C++ libraries, including OpenCV. The OpenCV binding to Java is officially supported by the core OpenCV team.

C++

C++ is a programming language defined and standardized by the International Organization for Standardization (ISO)[7]. There are multiple implementations of this standard (each differing slightly), the most prominent ones being as part of the free GNU Compiler Collection project and Visual C++ by Microsoft.

Python

Python[8] is an interpreted programming language with a simple syntax, partially derived the language "ABC", a simple language with the original purpose of teaching children programming. Despite its simple appearance it is closely tied in with C and C++, with many libraries being direct ports from these programming languages. Prominent examples of this include OpenCV (supported by the core development team), QT and many more. Furthermore, it also has a number of libraries for scientific computing available exclusively to it, such as the hugely popular numpy.

Being an interpreted language, Python is not as fast as most compiled languages. However, single methods of the C/C++ libraries will perform almost as fast as their native counterparts, since they just refer back to their native assemblies. As such, program logic is mostly implemented in Python, while performance critical algorithms are written in C/C++.

The versatility coupled with the easy syntax makes Python a very attractive technology to use for prototyping and scientific computation.

1.2.4 Data exchange formats

JSON

JavaScript Object Notation is a text-based data exchange format based on the object definition syntax in the programming language JavaScript. It is defined by the Standards RFC8259[9] and ECMA-404[10]. The simple syntax with only 4 object types (string, number, array, object) makes it easily human-readable while the same object syntax is native to a few programming languages (like JavaScript and Python) leading to fast and easy processing.

JSON data is most commonly transmitted via the Hypertext Transfer Protocol (HTTP) from a web service utlizing the Representational State Transfer (REST) model. The exact format of JSON and the means of exchange are not standardized, leading to vastly different types of usage and often completely incompatible web services merely able to parse the data, not

process it. No acknowledged standards exist. An attempt at providing standardized JSON schemas has been made through the JSON-Schema project[11].

XML

The Extensible Markup Language is also a text-based data exchange format specified[12] by the World Wide Web Consortium (W3C). It uses a system of tags and attributes to represent hierarchical data.

Its wide use in commercial applications gives it a large advantage over JSON when it comes to standardization. There is an official XML Schema recommendation and a Document Type Definition (DTD) standard by the W3C. Furthermore, web services can be described in the Web Services Description Language (WSDL), which is a derivative of XML. A standardized way of transforming XML exists in the XML Stylesheet (XSLT) standard. XLST provides the means to transform any XML document from one web service into a completely different XML arrangement for use with another.

2 Integration method

2.1 Analyzing target software

This proposed method starts with an analysis of the technology used to build both the target software and the technology to be integrated at different tiers.

At the core level, this is the programming language used to implement it. Integration is naturally easiest when these are the same or very similar. A method used in one program to encode an output is guaranteed to have an accompanying decoding method in the other program.

A level above this lie any direct extensibility interfaces. This includes all methods of directly attaching your own code to the target software. Examples for this are loading of shared libraries (dll, so, etc.) or Java JAR files.

The last level to analyze is any networking components. The application might expose a REST API via a local HTTP server or other information via a custom protocol and a local socket.

These levels are sorted from lowest to highest level interfacing from a technical standpoint. This is by no means a way to rank their usability, as the other software might have better support for the networking components or another layer might have shortcomings in a certain technology that makes it unable to support usage for assistive technology. For example, a JavaScript extension for an application is much more suited for interfacing via the networking layer than ones on a lower level. However, if the target software does not implement some kind of security protocol on that layer, it might not be suited for exchanging medical data.

Those are some reasons and an example why the integration of software, especially complex technology, might not be extremely straightforward and why it is important to choose the right technology.

2.2 Choosing the right technology

The simplest proposed methodology is to find the lowest level interface that the software to be integrated supports at its own lowest level, and use these end points to integrate the two pieces of software. This crude approach might be well suited if the two pieces of target software are very similar in nature. A directly extensible C++ program would not use a web layer to extend another C++ application when it can be directly connected via shared library plugin extension and the target software supports that method.

Another point to consider is the difference of closeness to hardware between the target software and the software to be integrated. If the target software is of a higher level it us usually easier to extend it with a lower level language because high level programming languages are built on low level technology. A good example of this is the integration of native C++ code into Java applications using the Java Native Interface. With this in mind, keeping the level of the software to be integrated at a higher level is more desirable, due to high level programming languages usually being easier to develop with and the loss of high level functionality when coupled with low level technology, such as the loss of easy platform independence when using the Java Native Interface.

The existing technology available to the programming language to be integrated is also important. While it is not usually possible to directly change things like dependencies about the target software, the software to be integrated can often be quite freely extended.

2.3 Choosing the attachment point

Having chosen layer and technology through the previous two methods, choosing an attachment point should now be the simple matter of looking at extendability options on the chosen layer and choosing the option best suited to the selected technology. Usually, this will only result in only one way to reasonably combine all the pieces.

If there are multiple options, the use case for the plugin should be evaluated and the best option chosen. Generally, the more independent from the target software the attachment point is, the more versatile and maintainable the resulting software will be. This is because a mostly self-contained plugin can more easily be integrated into a different system than something deeply intertwined with the target software.

This will also result in more maintainability because more of the source code is independent from the target and as such will not suffer from any failures that the target software might develop. Developers of the extendability interface will always seek to provide maximum compatibility, shifting workload from the plugin developer to the core development team who usually have a better understanding of the target software's internal processes.

3 Plugin development method

3.1 User Interface

Choosing where to have a user configure the application to be integrated is not always straightforward. Sometimes the target software provides a means to integrate a user interface. This approach is almost always the most desirable one, as it provides the most streamlined experience. Sometimes, the target software does not provide the means for this or they are not sufficient for the task. In this case, the software to be integrated has to provide some own means of configuration and / or interface. The easiest solution comes in the form of a configuration file to be edited by the user. More functionality like a graphical user interface can be added as well to the degree that the guidelines of the target software and the possibilities in the source software allow.

3.2 Development Platforms & Tools

The platform and software used to develop the add-in should ideally be the same as the one used to develop the target software. This is a problematic approach when the original development software is outdated or not available on the platform. Familiarity with certain software like a particular Integrated Development Environment plays a role too because it can dramatically affect the speed of development. The choosing of development software should take all these factors into account and result in the best compromise between these factors.

4 Implementation

4.1 Hardware

The hardware used to test this implementation was an Intel RealSense RS300 camera manufacture by Creative Technology Ltd. The Intel RealSense library used requires a DS400 series or RS300 camera[13]. Older versions supporting the RealSense cameras F200, R200, LR200 and ZR300 may work but this was not tested.

A desktop computer running Windows 10 64-Bit and a Linux laptop with elementaryOS 0.4.1 were used to test the component functionality and for development.

4.2 Software

For C++ development, the IDE "CLion" by IntelliJ was chosen, together with the integrated CMake build system. There was no particular requirement for features of the C++ IDE because the native library is mostly independent from AsTeRICS and its plugin system. The libraries used in the native C++ support library were OpenCV, librealsense and the Java Native Interface due to their versatility and ability to integrate into one another[14].

The C++ compiler used was the "GNU Compiler Collection 7" running either natively on Linux or in MinGW64 on Windows. Using the same compiler on both development computers streamlined the development process.

For Java development, the IDE "IDEA" by IntelliJ was selected due to familiarity with it. The build system of the ASTeRICS framework is Ant. IDEA supports this build system and the integration of Ant into it was set up for the build process. The IDE also provides support for the OSGi framework. These factors meant that the development environment could be used for AsTeRICS plugin development even if the AT-Toolkit wasn't originally developed using it.

The only external dependencies of the Java program is the ASTeRICS middleware and the native library developed for the recognition.

4.3 Conceptualization

The methods described earlier are now applied to the AsTeRICS framework to find a suitable layer to integrate the plugin into. At the core, AsTeRICS is written in Java. To provide support for extensibility through plugins on the layer above, it is written as an OSGi middleware. Finally, AsTeRICS can interact with network components through special plugins such as the WebSocket, used to connect to the AsTeRICS Runtime, for example via JavaScript in a browser.

All AsTeRICS components are plugins written in Java using the OSGi middleware layer provided by AsTeRICS. It is possible to implement additional functionality by using the JavaScript WebSocket from a browser window. For gesture recognition, JavaScript requires opening a seperate browser window, which is a nuisance to the end user. Since AsTeRICS is open source, it would also be possible to directly implement the functionality into the software itself. However, the gesture recognition does not need functionality not available to plugins. For these reasons, the implementation of the gesture recognition plugin was to use the OSGi plugin interface.

Initially, the chosen programming language for the plugin was to be purely Java. AsTeRICS provides the JavaCV Java bindings for OpenCV[15]. These are not the official Java bindings from OpenCV, but they include a Java version of the Intel "librealsense" library used for communication with the camera. After some prototyping, it was discovered that this approach had several severe drawbacks:

- The main Java OpenCV binding was not the official OpenCV Java binding and as such may become outdated or suffer other flaws
- The Java "librealsense" binding was outdated and no Windows version was available, making it impossible to run the plugin on Windows

Consequentially the Java prototype was scrapped and the functionality replaced with an implementation using OpenCV in one of its and librealsense's native environments, the programming language C++. The functionality in this library would be coupled to the Java plugin using the Java Native Interface. This solution had the advantage of good library support and superb performance. The only drawback was the additional complexity of the Native Interface and the difficulty of cross-platform programming on a low level language like C++, resulting in more time spent setting up the toolchain.

4.4 Implementation of Handtracking

4.4.1 RealSense and OpenCV

The open source library "librealsense"[13] by Intel used provides an easy means of accessing the RealSense camera directly from C++ code. A pipeline object is created and used in a loop to get the newest image data. The initial code for the project was taken from the "Getting Started with OpenCV" guide[16] from the librealsense Github documentation. For this project, only data from the camera's depth sensor is needed. The newest frame of depth data is received every loop iteration and converted to a standard OpenCV Matrix ("Mat") using a built-in librealsense method.

```
1
2 [...]
3
4 // Declare RealSense pipeline, encapsulating the actual device and sensors
5 rs2::pipeline pipe;
6 // Start streaming with default recommended configuration
7 pipe.start();
8
9 isRecognizing = true;
10 int previousFingers = 0;
11
12 using namespace cv;
13 while (gesture_visualizer.is_open() && isRecognizing)
14 {
15    rs2::frameset data = pipe.wait_for_frames(); // Wait for next set of frames
          from the camera
16    rs2::frame depth = color_map(data.get_depth_frame());
17    //rs2::frame depth = data.get_depth_frame();
18
19    // Query frame size (width and height)
20    const int w = depth.as<rs2::video_frame>().get_width();
21    const int h = depth.as<rs2::video_frame>().get_height();
22
23    // Create OpenCV matrix of size (w,h) from the colorized depth data
24    Mat image(Size(w, h), CV_8UC3, (void*)depth.get_data(), Mat::AUTO_STEP);
25    [...]
26 }
```

Code 1: RealSense image capture and conversion

The default capture mode of the depth sensor outputs a BGR (Blue, Green and Red color components) image in 8 bit depth, as indicated by the 8-bit depth 3 channel ("CV_8UC3") format of the matrix. For easier processing, a grayscale image is more suited. Therefore, a color scheme is set using a "colorizer" class built into the librealsense library.

```
1 // Declare depth colorizer for pretty visualization of depth data
2 rs2::colorizer color_map;
3 // Use color scheme option 2 (grayscale, distant black, close white)
4 color_map.set_option(RS2_OPTION_COLOR_SCHEME, 2);
```

Code 2: RealSense image colorizer

The resulting captured image in OpenCV Matrix format is then passed to the library's own "recognizer" class for further processing.

4.4.2 Gesture recognition

The "recognizer" class is where all of the image segmentation and hand detection occurs. Its method "get_hand_model" takes the Matrix depth frame and outputs a "hand_model" structure, containing all the necessary info about how the hand was detected and, as a final result, how many fingers were detected as extended. The structure is as follows:

```
1 struct hand_model{
2    int num_fingers;
3    std::vector<cv::Point> hand_contour;
4    std::vector<int> hand_hull_indexes;
5    std::vector<cv::Vec4i> hand_defects;
6    std::vector<int> finger_defects_indexes;
7    cv::Mat display_frame;
8 };
```

Code 3: Hand Model Structure

The following steps, taken from the article "Hand Gesture Recognition Using a Kinect Depth Sensor" by Michael Beyeler[17], are then applied to get the hand model from the passed Matrix:

Conversion The grayscale BGR matrix is converted to a single grayscale channel

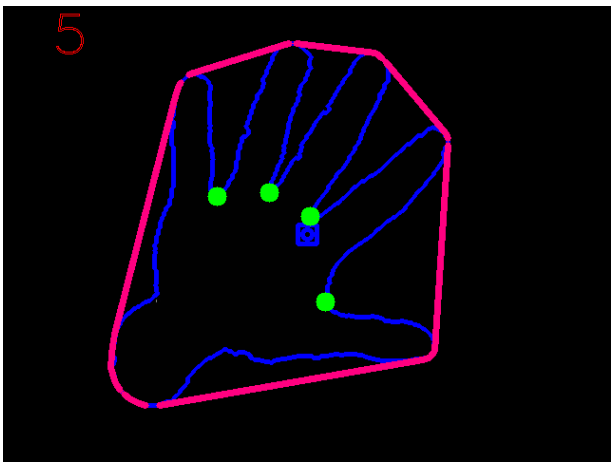
Segmentation The arm is segmented from background noise. This is achieved by getting the median value of a small center region, and then removing all depth data within a certain threshold of that median. Gaps caused by an insufficient threshold are filled in and everything is colored gray. The segment of the image touching the middle (the hand) is then filled white and all remaining gray background noise is removed. The result is an image with a white hand contour in the middle.

Defect Recognition Contours are drawn around the white segments in the frame. The contour with the biggest area is selected as the probable shape of the hand. It is smoothed to remove noise and a hull is drawn around it. The hull is a contour drawn around the outside of the hand, not accounting for "valleys" in the actual contour. This hull is then analyzed for convexity defects, which represent those valleys.

Hand Processing The convexity defects are analyzed for the angle of the "valley". If it is smaller than a threshold, it is counted as an extended finger. Otherwise, it is discarded.

Figure 3 below visualizes some of these steps. The contour of the hand from the defect recognition process is drawn as a blue line. The hull from the same step is represented by the pink lines. Green dots illustrate the convexity defects that are detected and found to have an angle smaller than the threshold.

Figure 3: Visualization of hand recognition



All the above mentioned processes can be found in the "recognizer.cpp" class file, split into appropriately named methods. They utilize OpenCV functionality. Comments in those methods describe the workings in greater detail.

4.4.3 Tracking and Information Window

Sensible additional functionality for hand tracking software is some kind of user interface. This is because the person in front of the computer sometimes requires feedback on what the camera is seeing and how it is interpreting the data, so they can correct any malfunctions in the detection by re-aligning their hand. The simplest form of interface for this application would be a small window showing the video feed from the camera and a representation of the tracking model, and this was implemented using OpenCV's own HighGUI library. This library is only meant for debugging purposes and caused problems when called from C++ code due to threading issues.

4.4.4 Independent testing of the native library

For faster testing and therefore faster development of the native library, a small executable program was developed that called the loop function of the library to run it for testing purposes. The loop function has to take into account if it has been given the Java environment reference variable. If it hasn't, the library was not called from a Java Native Interface context and therefore operates in testing modes, executing no native interface code.

4.5 Integration into framework

4.5.1 Anatomy of an AsTeRICS plugin

AsTeRICS plugins are located in folders named after their last two package names. This is usually results in a "<componentType>.<name>" format. In this folder, the required files are:

- "build.xml" file for the ant build system, specifying the build options for the plugin
- "bundle_descriptor.xml" file located in the "src/main/resources folder", describing the AsTeRICS plugin characteristics such as input ports, output ports, etc.
- Standard Java Manifest file "MANIFEST.MF" located in the "src/main/resources/META-INF" folder
- Main Java source file in the "src/main/java" folder, located within the appropriate package directory

The main plugin class, in this case called "RealSenseGesturesInstance", should have a package name that starts with "eu.asterics.component". The last two parts of the package name are the component type, e.g. "sensor", and a descriptive component package name, e.g. "realsensegestures". These last two parts of the package name should also be the same as the main directory name for the plugin.

4.5.2 Plugin Creation

To ease plugin creation, AsTeRICS comes with a Plugin Creation Wizard. This is a tool that provides a GUI for setting all plugin properties and then generates an empty "stub" plugin with the given configuration in the correct directory.

This tool is a Windows executable called "AsTeRICS_PluginCreationWizard.exe" located in the "bin/ACS/tools" directory of the main AsTeRICS installation.

It is possible to use it within Linux using the "wine" emulator application. This was tested and working using "wine" version 2.4 under elementaryOS 0.4.1, an operating system closely related to Ubuntu 16.04 LTS "Xenial".

4.5.3 Plugin Configuration and Options

AsTeRICS plugins can have input ports where they can receive data and output ports where they can return data. Additionally, AsTeRICS supports an event system within which plugins can fire events to other plugins using "Event Trigger Ports" and receive said events from other plugins through "Event Listener Ports". Furthermore, plugins can be configured via a set of defined properties. All of the ports and properties have fixed primitive data types.

What follows is a short overview of the most important configuration options of an AsTeRICS plugin:

Table 1: AsTeRICS Plugin Configuration

Option Name	**Description**	**Configured Value**
Plugin Name	Name of the plugin as it appears in the AsTeRICS configuration suite "ACS".	123
Type	Describes what type the component is. Possible values are "actuator", "processor" and "sensor".	345
Subcategory	Denotes the category within the Type in which the component subsides. Used for selection the ACS.	456
Input Ports	List of component data input endpoints. Each of them has to have a name and a primitive data type assigned.	567
Output Ports	List of component data output endpoints. Each of them has to have a name and a primitive data type assigned.	678
Event Listener Ports	List of component endpoints used by the plugin to receive events sent by other plugins if connected to a "Trigger" port.	789
Event Trigger Ports	List of component endpoints used by the plugin to send events to other plugins that will receive the events on their "Listener" ports.	8910
Properties	List of component configuration properties that can be set by the configuration in the ACS. Each of them has to have a name and a data type assigned.	8910

4.5.4 Main Plugin Class

The main class of the AsTeRICS RealSense gesture recognition plugin is called "RealSenseGesturesInstance" and extends the abstract class "AbstractRuntimeComponentInstance" provided by the AsTeRICS middleware layer. This forces it to implement functionality used by AsTeRICS to retrieve metadata and output by the plugin as well as methods to receive data sent to it by the Toolkit.

```
1 public class RealSenseGesturesInstance extends AbstractRuntimeComponentInstance
2 {
3    final IRuntimeOutputPort opDetected = new DefaultRuntimeOutputPort();
4    final IRuntimeOutputPort opExtended = new DefaultRuntimeOutputPort();
5    [...]
6    private RealSenseNativeConnector native_conn;
7    [...]
```

Code 4: Main plugin class and member variables

A "stub" class with basic empty implementations was generated by the Plugin Creation Wizard. The "Runtime Output Port" classes provide functionality for outputting data from the plugin. Two standard output ports were used here because that is the plugin configuration and no advanced functionality for the ports was needed. The default ports provide simple methods for outputting data in bit form.

The rest of the main plugin class functionality is left to default. Most of the workload is delegated to the class managing the connection to the native support library, "RealSenseNativeConnector", described in the chapter below. This class was created to keep the functionality in the component more compartmentalized. The main class methods are restricted to interaction with the AsTeRICS middleware and the connector class.

4.5.5 Native Connector Class

Interaction with native code

The class "RealSenseNativeConnector" implements the method called by the native library through the Java Native Interface. It is used for relaying the number of detected extended fingers to the running Java application.

```
1 // Callback called from C++ code when extended finger numbers change
2 public void fingerNumberChanged(int fingersExtended){
3    if(transmitting) {
4       AstericsModelExecutionThreadPool.instance.execute(new
            PassFingersRunnable(fingersExtended));
5    }
6 }
```

Code 5: Java Native Interface callback method

This method, also known as a "Callback", is called on every iteration of the native recognition loop, running inside a separate thread within Java. As a parameter it is given the number of currently detected extended fingers. It confirms that the AsTeRICS model is not currently paused through the "transmitting" member variable of the connector class and, on fulfillment of that condition, passes the given number to the main AsTeRICS model execution thread via a class that extends the Java "Runnable" class. This has to be done because AsTeRICS does not accept output of values from a separate thread. The "Runnable" forwards these values to the output port in the main plugin class.

An alternative for passing data from the native library to the Java code was considered. The information can be passed between the environments by using a local networking socket on the computer. However, this would result in additional implementation overhead, as the format for the data exchange needs to be defined and implemented on both sides. It is also not as fast as the callback method, although due to the small amount of information being passed this was not really a considered factor. The approach was discarded because of the additional time it would have taken to implement and the little benefits it would have brought.

```
1 public class PassFingersRunnable implements Runnable {
2    int extended;
3    public PassFingersRunnable(int fingers_extended) {
4       this.extended = fingers_extended;
5    }
6
7    @Override
8    public void run() {
9       parent.opExtended.sendData(ConversionUtils.intToBytes(extended));
10   }
11 }
```

Code 6: Forwarding runnable

Furthermore, the "RealSenseConnector" class also defines all the methods that have their implementation in the support library. All these methods have equivalents in the native library that interact with the main loop function of the C++ code.

```
1 // Native methods for recognition control
2 public native void start_recognition();  // Blocking call that starts
     recognition loop
3 public native void pause_recognition();  // Does the same as stop
4 public native void stop_recognition();  // Stops the recgonition loop
5 // Visualization - not implemented
6 public native void start_visualization();
7 public native void stop_visualization();
```

Code 7: Java Native Interface methods

The visualization part is not implemented in C++ and not used by the Java plugin for that reason. The cause for this decision is explained later in the results chapter.

Threading native code in Java

Components of AsTeRICS need to be able to interact with the graphical user interface (GUI). For this reason, the main plugin class runs on the same thread as the GUI. This means that the hand recognition had to be done on a seperate thread. There are two sensible ways of approaching this problem.

The first way is to perform all threading inside the native C++ support library. This has been made significantly easier to do in C++ since the introduction of the "C++11" standard that is now widely supported by compilers. Nevertheless it is still harder to interact and share data between threads than in higher level languages such as Java. Additionally, this approach requires more interaction between Java and C++ in this case because the Java code needs more precise control of the threading.

The second way involves performing the threading within Java. The native functionality in the library is called from within a Java thread and executed there. This results in less direct control of the C++ code but is faster and less complex to implement.

Considering both ways, the second approach has considerably less overhead because it features fewer JNI functions. It is also faster and easier to implement due to the more advanced threading functionality in Java. For these reasons, it was the approach chosen for the implementation of the gesture recognition plugin.

Start of the native recognition loop blocks execution of its thread while running. It is executed in a Java Thread implemented as the "RealSenseNativeConnectorThread" class. This class is initialized with the instance of the native connector thread class as "parent" and a start delay. This delay is sometimes used by the native connector when restarting the native loop after an error. In various instances during testing a random failure of the USB connection occurred and thus the delay is used to give the user some time to rectify the problem.

```
1 // Do the blocking recognition loop in thread
2 try{
3    if(startDelay > 0) { Thread.sleep(startDelay); }
4    this.parent.start_recognition();
5    System.out.println("[RealSenseGestures] Recognition loop stopped - Thread
         exiting");
6 }catch (Exception e){
7    System.out.println("[RealSenseGestures] Error in recognition loop - stack
         trace below");
8    e.printStackTrace();
9    parent.recognitionThreadExited();
10 }
```

Code 8: Functionality inside the native connector threading class

When the thread is started it starts the recognition loop inside the native library. This blocks thread execution until it is either stopped by the native connector class or runs into an error, in which case the catch block is triggered, the error reported and the parent notified. The parent will attempt to start a new thread after a delay.

5 Result

5.1 Achieved functionality

What follows is a list of all the components implemented as well as their respective functionality and purpose.

- A native library for Windows and Linux in 64-bit architecture that can connect to an IntelRealSense camera and use its depth sensor to recognize a hand and the number of extended fingers using OpenCV. It also contains functionality for being controlled through a Java class and reporting information back to it through use of the Java Native Interface.
- A small test application used to test that library with a visualization of the recognition process.
- A Java AsTeRICS component plugin that runs the library in a seperate thread to recognize the number of fingers extended and relay that data to an AsTeRICS model.

5.2 Evaluation

5.2.1 Fist mouse click

The first evaluation was done using an AsTeRICS model created to determine the basic accuracy of the extended finger detection. Whenever the extended finger count drops below 3 for 2 seconds, a left mouse click is executed.

In testing, this was found to be reliable. While the gesture recognition component does drop below 3 fingers when more are extended quite often, these errors never persist for a long duration of time. The plugin is able to accurately detect a fist and allowed the AsTeRICS environment to act upon that information in a predictable manner.

5.2.2 Number entry

To test functionality that requires more precise and predictable measuring, a second model was created that recognizes the number of fingers extended and adds that detection count to a text field if a value persists through a threshold of one second.

The plugin failed at providing accurate enough data for this use case. The returned numbers fluctuate too much to trigger through the time threshold and are often too inaccurate to be of use.

Smoothing functionality in the model could provide a better result. This was not tested because if a such a feature is implemented it should be part of the plugin, since it is the responsibility of the component to provide accurate and predictable measurements.

5.2.3 Evident Limitations

Both use case tests provide unpredictable results when the hand is completely removed from the center of the tracking area. Furthermore, the tracking only recognises a hand held perpendicular to the camera with the front or back of the hand fully exposed to the sensor and the fingers pointing up. Furthermore, a distance to the camera of 15 centimeters provides the best results in both tests and deviation from that distance can also lead to more unpredictable behaviour. Lastly, when the hand is held at an angle with the face or the back facing towards the sensor, the hull defect angle measuring algorithm fails and the defects are now longer recognized, falsifying the extended finger count.

6 Conclusion

6.1 Effectiveness of method

Analyzation of the AsTeRICS framework proved the first method to work very well. The three proposed tiers are easily distinguishable while being seperated enough to be of use for decision-making. Components of the framework being exclusively written as plugins following the OSGi architecture made this process more straightforward than could be the case in other applications, in which cases the layer definitions might need to be expanded.

Choosing technology for the gesture recognition component proved more complicated than predicted by the second method. An add-in might not want to settle on one technology - as an example, one programming language - for implementation. This is due to factors like library availability, which weren't considered in the method for complexity reasons. The failed Java implementation of the developed plugin is proof of this. An improvement for this method would be an easy way to define a collection of technologies that combine well (as was done with Java and C++ through the JNI), rather than a single one.

The attachment point is highly dependent on the result of the first two methods. Attaching to the OSGi layer was a prudent decision and further proved that the layer method works well. This would not have been so easily achieved if Java as the base technology of the plugin had to have been completely removed. As a result, a good improvement for the 3rd method would be a way to make the step of finding an attachment point more independent of the other two steps.

A change of layer or technology should not affect the chosen attachment point as drastically as it does with the currently proposed method.

6.2 Possibilities for improvement

To complete the paper, below is a list of possible improvements for further development of the AsTeRICS RealSense Hand Gesture Recognition component:

- Provide a better algorithm for measuring the angle of defects in the hand hull that works at any angle
- Add smoothing functionality that removes some of the finger count fluctuation, toggleable through a component property
- Implement tracking of the hand as it moves around the sensor's field of view and use this information for more gestures that trigger AsTeRICS events
- Provide additional compiled native libraries for more processor architectures and operating systems

Bibliography

[1] MultiArt PR-Agentur GmbH, “Paul Watzlawick - Ein Baumeister der eigenen Wirklichkeit.” [Online]. Available: http://www.watzlawickehrenring.at/paul-watzlawick.html

[2] Dr Gordon Coates, “Notes on Communication, Appendix 3: Watzlawick’s Five Axioms.” [Online]. Available: http://www.wanterfall.com/Communication-Watzlawick%27s-Axioms.htm

[3] [Online]. Available: https://www.osgi.org/developer/architecture/layering-osgi/

[4] Oracle Corporation, “Java, Official Website.” [Online]. Available: https://java.com/en/

[5] M. Fourment and M. R. Gillings, “A comparison of common programming languages used in bioinformatics,” *PubMed Central®, US National Library of Medicine*, 2008. [Online]. Available: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC2267699/

[6] [Online]. Available: https://commons.wikimedia.org/wiki/File:Java-jvm.png

[7] International Organization for Standardization, “C++ language specification.” [Online]. Available: https://www.iso.org/standard/68564.html

[8] Python Software Foundation, “Python Programming Language, Official Website.” [Online]. Available: https://www.python.org

[9] Internet Engineering Task Force, “Rfc 8259 – the javascript object notation (json) data interchange format,” 2017.

[10] Ecma International, “Ecma-404 – the json data interchange syntax,” Genf: Ecma International, 2017.

[11] JSON Schema Project, “JSON Schema Website.” [Online]. Available: http://json-schema.org

[12] World Wide Web Consortium, “Extensible markup language (xml) 1.0 (fifth edition),” World Wide Web Consortium, 2008. [Online]. Available: https://www.w3.org/TR/REC-xml/

[13] Intel Corporation, “librealsense, Github repository.” [Online]. Available: https://github.com/IntelRealSense/librealsense

[14] IBM, “IBM Knowledge Center: Overview of JNI.” [Online]. Available: https://www.ibm.com/support/knowledgecenter/en/SSYKE2_8.0.0/com.ibm.java.lnx.80.doc/diag/understanding/jni_overview.html

[15] Samuel Audet and contributors, “JavaCV, Github repository.” [Online]. Available: https://github.com/bytedeco/javacv

[16] Intel Corporation, “librealsense "Getting Started with OpenCV" guide.” [Online]. Available: https://github.com/IntelRealSense/librealsense/blob/master/doc/stepbystep/getting_started_with_openCV.md

[17] Michael Beyeler, “Hand Gesture Recognition Using a Kinect Depth Sensor.” [Online]. Available: https://www.packtpub.com/books/content/hand-gesture-recognition-using-kinect-depth-sensor

List of Figures

List of Tables

List of Code

List of Abbreviations

AT Assistive Technology

W3C World Wide Web Consortium

JSON JavaScript Object Notation

XML Extensible Markup Language

AsTeRICS Assistive Technology Rapid Integration & Construction Set

OpenCV Open Computer Vision

IDE Integrated Development Environment

REST Representational State Transfer

API Application Programming Interface

HTTP Hypertext Transfer Protocol

OSGi Open Service Gateway Initiative

JNI Java Native Interface

GUI Graphical User Interface

A Anhang A

B Anhang B