Natural Human-Computer Interaction
with Kinect and Leap Motion

Xietong LU
Supervisor: Yuchao DAI

COMP8790: Software Engineering Project
Australian National University Semester 1, 2014

May 29, 2014
Abstract

In the recent years, a growing number of devices have been invented to improve human-machine interaction. Kinect by Microsoft may be most popular one among them. The other device, Leap Motion, is relative new and getting more and more focus. These two devices have different advantages and defects. In the project, a concerted effort is made to combine these two devices for the purpose of enhancing the user experience during the interaction with PC (personal computer). For the constraint on time, the project has not accomplished the combination of Kinect and Leap Motion. However, the project succeeds to extract the hand information from both the Kinect and the Leap Motion. The collaboration of the hand information from different sources is implemented as well. Finally, the project succeeds to visualise the collaboration using the Point Cloud Library (PCL).

**Keywords:** Kinect, Leap Motion, Human-computer Interaction, Computer Vision
Contents

1 Introduction .............................................. 3
  1.1 Background ........................................ 3
  1.2 Kinect .............................................. 3
  1.3 Leap Motion ........................................ 6
  1.4 Motivation ........................................... 7

2 Simple Leap Motion Application ....................... 8
  2.1 Application description ............................. 8
  2.2 Software Construction ............................... 9
  2.3 FSM .................................................. 12
  2.4 Conclusion .......................................... 14

3 Hand Recognition Application Using Kinect .......... 16
  3.1 Application Description ............................. 16
  3.2 Depth Calibration .................................... 22
  3.3 Skin Colour Detection ............................... 24
  3.4 Palm Recognition ..................................... 26
  3.5 Fingertips recognition ............................... 27

4 Kinect and Leap Motion .................................. 30
  4.1 Application Description ............................. 30
  4.2 Transformation from Leap Motion to Kinect ....... 32

5 Hand Gesture Recognition Application ................. 35

6 Future Improvement .................................... 38

7 Conclusion ............................................... 41
Chapter 1

Introduction

1.1 Background

A natural user interface (NUI) is a system for human-machine interaction that enables users to operate machines with everyday behaviours. The principle of NUI is to provide a more intuitive way for users to interact with machines. It is not a new concept. In the 1970s, Steve Mann has started to develop strategies for NUI [1]. Since last decade, this concept becomes very popular. Different attempts have been made to improve the user experience using NUI. On the one hand, some companies focus on interface through speech, such as Google Now by Google and Siri by Apple. On the other hand, some companies pay attention to the interaction through gestures. There are many inventions in this field, such as Kinect, Leap Motion, MYO, etc. This project will mainly focus on the interaction through gestures. Considering the popularity and availability of different inventions, Kinect and Leap Motion are chosen for the project.

1.2 Kinect

Kinect is a line of motion sensing input devices by Microsoft. It is firstly designed for the user-machine interaction in video games (XBOX 360). Since it was released in 2010 [2], it has become more and more popular in the academic areas, especially in Computer Vision, Robotics and Human-Computer interaction.

The popularity of Kinect comes from its unique combination of different sensors. The device consists of an RGB camera, a depth sensor and multi-array microphone. Because multi-array microphone is not involved in the project, more attention will be drawn on the RGB camera and the depth sensor. The RGB camera is similar
CHAPTER 1. INTRODUCTION

a normal webcam, providing real-time colour images. The depth sensor basically comprises an infrared ray (IR) emitter and a receiver (illustrated in figure 1.1). By measuring the receiving time of the reflected ray, the depth, the distance between an object and the Kinect, can be calculated. Moreover, a pixel in the RBG image can find its corresponding depth data. Therefore, the Kinect provides both 3D and colour information of the captured objects.

The other significant and powerful feature of Kinect is its capability to recognise human bodies and track users skeleton. Kinect can recognise up to 6 people and keep tracking two skeletons among them (illustrated in figure 1.2). By calling some built-in functions in SDK, developers can retrieve the real-time position of each joint conveniently. The function provides more possibilities for improving user experience in HCI. In the market, there are already many video games taking advantage of this feature and they have earned praise from family users.

The third feature making Kinect popular could be its software side. Since the first release of non-commercial Kinect software development kit (SDK) by Microsoft in 2011, several updated versions have been released. There are different new added features in each update. For example, in the newest version (1.8), the SDK provides API (application programming interface) to remove background automatically. In Version 1.7, a feature called Kinect Interactions was introduced in the SDK. It provides API to detect some specific gestures, such as pressing the palm forward and gripping the hand. Apart from the powerful features, the basic functions in SDK are also well constructed. All in all, Kinect is a developer-friendly device.

However, Kinect has some defects as well. First, the depth sensor has a highly limited range. In the normal mode, the depth sensor can sense objects between 0.8 meters and 4 meters. In the near mode, its range is from 0.4 meters to 3 meters. Therefore, it cannot detect objects very close to it. Second, the data from the depth sensor is interfered by noise severely making it less reliable. This defect also affects other features relying on the depth data, such as the skeletal tracking. Third, the image quality from the RGB camera is not entirely satisfactory, especially in low light environment. The colour rendering index (CRI) is very low at night, which means the image from Kinects RGB camera is like a black-white image.
CHAPTER 1. INTRODUCTION

Figure 1.1: A Kinect sensor [credit: Microsoft]

Figure 1.2: Kinect can recognise 6 players and track two among them [credit: Microsoft]
1.3 Leap Motion

The Leap Motion controller is another hardware sensor device that detects hands and fingers (illustrated in figure 1.3). It is designed to be connected to a PC and placed on a physical desktop. Then it can detect users hands and fingers, while the user does not touch it.

The most obvious feature of Leap Motion is its size. Although there are two monochromatic IR cameras and three IR emitters inside the device, it is actually a 1.230.5 inch box. Its tiny size makes it more convenient to cooperate with a PC.

The other feature of the Leap Motion controller is its range. Different from Kinect, it has a range much closer to itself. The devices observes a hemispherical area to a distance around 3 feet. It can detect fingers even when the fingers are just 5cm away. Because of the close range, Leap Motion has higher accuracy for the position of fingers and data from Leap Motion is highly reliable.

The third advantage of Leap Motion is its API. The SDK of Leap Motion provides some high-level APIs. Position of fingers and centre of palms can be returned directly. The SDK also has some built-in functions for gesture detection. It can detect gesture like swiping, drawing circles etc.

However, the third advantage of the Leap Motion controller can be its disadvantage from the other view of point. Raw data, the images from two IR cameras, is not accessible by the developers. It means that developers can only improve their applications in the high level.
CHAPTER 1. INTRODUCTION

The second drawback of Leap Motion is that it can only detect fingers in certain poses. A hand is in the ideal pose when its palm is facing down above of Leap Motion. The hand can be construed as in parallel with the Leap Motion. In that case, the data returned by Leap Motion is consistently reliable. However, when the angle between the hand and Leap Motion increases, the projection of hand onto Leap Motion becomes smaller, making it more difficult to observe the hand for Leap Motion. For example, when a hand poses like figure 1.4a, the finger positions returned by Leap Motion is illustrated on figure 1.4b.

![Figure 1.4: An example for the post constraint.](image)

1.4 Motivation

As mentioned in the previous sections, both Kinect and Leap Motion have different advantages and drawbacks. In the project, an effort is made to combine these two devices in order to improve the human-computer interaction. Specifically, the project tries to improve the hand gesture recognition, using the advantages of Kinect to counteract Leap Motion’s defects.

**Project Organisation.** The project is separated into four phases. In Phase 1, a simple Leap Motion application is developed for the purpose of getting familiar with the Leap Motion controller. In Phase 2, a hand recognition application using Kinect is developed. In Phase 3, the application in Phase 2 is extended to coordinate hands positions from Kinect and Leap Motion. The result is visualised in 3D space in the application. In Phase 4, the final phase, a hand gesture recognition application is designed using Kinect and Leap Motion. Detail for each phase is described in separated section. The future improvement is discussed in Chapter 6.
Chapter 2

Simple Leap Motion Application

The chapter will give details of the project’s first phase. The objective of this phase is to develop a gallery application, which interacts with users through Leap Motion. The chapter is constructed into 3 parts. Section 2.1 gives a more detailed description of the application including its GUI (graphic user interface) and how users interact with the application through Leap Motion. Section 2.2 is an introduction of the application software structure. Section 2.3 introduces a software mechanism called finite-state machine, which is used in the application. The last section is the conclusion for this phase.

2.1 Application description

figure 2.1 illustrates the GUI of the application. figure 2.1a is the initial view of the application. 16 photos are displayed. One and only one of them will be highlighted, which indicates a cursor is pointing to the photo. In figure 2.1a, the highlighted photo is the one on the top right corner. Users can move the cursor by a swipe gesture. Swiping up, down, left or right will lead the cursor to move in the corresponding direction. For example, swiping from left to right can move the cursor to next photo on the right side. To browse the detail of the selected photo, users can zoom in by drawing a clockwise circle in the air using one finger. Then an enlarged photo will pop up as figure 2.1b. Users can return to the previous view by drawing a counter clockwise circle.
CHAPTER 2. SIMPLE LEAP MOTION APPLICATION

2.2 Software Construction

Figure 2.2 illustrates the construction of the application. The application is separated into two processes.

There are many options for to implement a GUI application, such as using external libraries Qt, FLTK and wxWidgets. These libraries are in programming languages like C++ or Java. These libraries are costly to learn and the program may require a high source lines of codes (SLOC). Therefore, they may not be the ultimate option for the application. In order to spare time for the later phases of the project, the GUI part is designed as a Web application, using HTML5, CSS3 and Javascript. With help of JQuery (an external library in Javascript), the GUI part can be implemented in around 100 lines of codes, with fair appearance and animated transition.

The second part of the application is to control Leap Motion and process data fetched from Leap Motion. Leap Motion controlling and data processing are decoupled into two modules and each module runs on an independent thread. The reason for this design is to enhance the encapsulation for each module so that modules can be reused conveniently in the latter phase.

A Leap Motion listener is running on Thread 1. The listener controls the Leap Motion device. It receives information from the device by a callback function. It means that the updated data is pushed by the Leap Motion device automatically whenever it is available and the listener does not need to check if there is new data continually. Data received from Leap Motion includes positions of fingertips and the detected gestures.

The main logic of the application is running on Thread 2. It first pulls information
from the listener. Because the information is data shared between two threads, the mutex mechanism is introduced to ensure that only one thread will manipulate the data each time. Leap Motion can recognise gestures and the detected gesture is included in the information. However, the detected gesture is not highly reliable and not entirely suitable for the scenario in the application. For example, when a user does a single swiping gesture with a single hand, the detected gestures by Leap Motion is visualized in figure 2.3. Leap Motion interprets the single gesture as three swiping gestures and one circle gesture, which is undesired for the application. Therefore, the detected gestures are further processed using FSM. More details of FSM will be introduced in 2.3. The FSM returns the processed gesture corresponding to the gesture done by the user.

Finally, the processed gesture output by FSM should be passed to the GUI. As mentioned above, the GUI is a Web application, which is running in a browser, while the controlling part is an independent C++ program. Therefore, they are two separated programs and the communication between programs is not trivial. The solution is that when an action is triggered by FSM, the interface component (in figure 2.2) will simulate a global key pressing event. A global event can be captured by all other programs in the operating system (OS), including the browser. For the Web application, each operation is mapped to a unique corresponding key (e.g. moving the cursor to the right is mapped to the right arrow key). An example
CHAPTER 2. SIMPLE LEAP MOTION APPLICATION

Figure 2.3: Leap Motion recognises a single swiping gesture as be multiple gestures.

is illustrated in figure 2.4. It may not be an optimal solution, because a global key event may influence other programs. However, it is still an efficient solution balancing the feasibility and functionality. The lines of source code for this part is less than 30 and it fully performs the required functions.

Figure 2.4: An example of how the interface component works.
CHAPTER 2. SIMPLE LEAP MOTION APPLICATION

2.3 FSM

A FSM is model of computation commonly used in gesture recognition. A particular FSM is defined with a finite set of states and a finite set of events. It can be in only one state at a time. Transition of states is triggered by an event. One of the states will be defined as initial state and a subset of the states is defined as final states. The FSM starts from the initial state and it terminates if it achieves one of the final states. An example of a particular FSM is illustrated in figure 2.5. The FSM has 3 states and 2 events. The initial state is State A pointed by a dot arrow. If Event 1 happens, the machine remains in State A. The state of the machine does not change until Event 2 happens. Then the machine transits to State B. Similar to State A, the machine stays in State B, until another Event 2 happens. Then machine transits to State C. The double-line edge of C indicates that it is a final state. Therefore, the machine terminates.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{fig2_5.png}
\caption{An example of a particular FSM.}
\end{figure}

The advantage of using FSM for gesture recognition is that it is not necessary to restore the all the history data. A gesture normally is a sequence of movement. The straightforward method to determine whether a gesture is performed is to check if the current movement combining with the movements in the past can be a meaningful gesture. The cost in computation time and space can be extremely high. With the help of FSM, the information of the past movement is encoded in the current state. There is no need to store all the history information. FSM is far more efficient than the straightforward method if it is well designed.

In the application, the updated data from Leap Motion is input to the FSM as events, which can trigger the transition between states of the FSM. If a gesture is detected by Leap Motion, detail of the gesture will be included in the updated data. The gesture information includes the gesture type (swiping or drawing circle), gesture ID, gesture state and other details. A unique ID will be assigned to the newly detected gesture. The gesture state may be start, update or stop. If a swiping gesture is detected, an ID is assigned and it has state start. Leap Motion keeps tracking the gesture and its state changes to update. When the hand stops moving, the gesture finishes and its state becomes stop. To avoid confusion between the terminologies of gesture information and terminologies of FSM, the gesture information returned
by Leap Motion is called \textit{raw gesture}. The gesture output by the FSM is called \textit{processed gesture}. The state in raw gesture is in notation \textit{g-state}. The state of FSM is called \textit{f-state}.

A simplified FSM for the swiping gesture is illustrated on figure 2.6. The application will collect raw gestures periodically. In each interval, 0, 1 or more than one raw gestures may be collected (only raw gestures in swiping type will be collected by this FSM). The collected raw gestures are transformed into FSM events using the following rules:

- Rule 1. If no raw gesture is collected during the interval, indicating that no gesture is performed by users, then the generated event is \textit{E0}.
- Rule 2. If more than one raw gestures are collected, an event is generated for each gesture using following rules.
  - Rule 3. If g-state is \textit{start}, then an event \textit{E1} is generated.
  - Rule 4. If g-state is \textit{update}, then an event \textit{E2} is generated.
  - Rule 5. If g-state is \textit{stop}, then an event \textit{E3} is generated.

Figure 2.6: The simplified FSM for swiping gesture.

S0 is the initial f-state. It is the idle state, which means it keeps waiting for the first raw gesture input into the FSM. Once the first raw gesture with g-state \textit{start} is detected, an E1 event is generated and the FSM transits to S1 indicating the machine is running. As mentioned before, Leap Motion interprets a single swiping gesture into multiple gestures and they are returned by Leap Motion at different time points. Therefore, it is possible for the FSM to receive E1 events in S1. Then E1 events would not trigger the state transition. The FSM stays in S1 until a raw gesture with g-state \textit{stop} is collected. In this case, an Event \textit{E3} is generated and the FSM transits to S2, which indicates it is waiting to stop. The FSM does not terminate in S2 is because some of the raw gestures do not finish yet. The FSM should keep collecting the unfinished raw gestures. The machine stays in S2, until an Event \textit{E0} is generated, indicating all gestures are already collected. Then a processed gesture is output by the FSM.
The implementation of FSM takes advantage of OOP (object-oriented programming). To enhance the reusability, the FSM is designed as an abstract class, making it more general and suitable for different usage. In the class, the state and event are two template types. The class only defines one method called updateState. It takes an event as input and updates the current state of the FSM. The class also declares several virtual methods as the interface to initialise the class. To use the FSM, a derived class of the abstract class, the state type and the event type are defined. Taking the FSM for swiping as example, two enumerated types, SwipeState and SwipeEvent, are created. The class, SwipeFSM, is defined which inherits FSM with the newly created state and event type representing the template types. SwipeFSM overrides the virtual methods to specify the initial state and final states. It also overrides initStateTable in order to specify the transitions between states. The class diagram for the FSM in the application is illustrated in figure 2.7.

![Class Diagram](image)

Figure 2.7: The class diagram for the FSM in the application.

### 2.4 Conclusion

The final version of the application fully satisfies the requirement in the application description. For the design of the interaction, the selection operations are mapped to the swiping gestures, which should be intuitive for users. The expansion operation is mapped to circle gesture because it is the second reliable gesture supported by Leap Motion API. However, this operation may not be intuitive for all users. In terms of the experience of using the application, first, there is obvious latency between the gesture and the animation. It is caused by the design of the FSM. As mention in Section 2.3, the FSM does not terminate until the gesture is entirely finished. When a user think the gesture already finishes, Leap Motion may still detect the
slightly movement of the hand and does not return raw gesture with g-state *stop* immediately. This defect can be overcome with a more complicated FSM. Second, the accuracy of gesture recognition is acceptable. The swiping left and right gestures have accuracy rate above 95%. The accuracy rate of the other two swiping gestures is above 85%. The recognition of drawing circle gesture is not as reliable as the swiping gesture. Therefore, the accuracy rate is around 70%. This problem may be alleviated using the version Leap Motion API. The new version is a beta version, so it is not used in the project.
Chapter 3

Hand Recognition Application Using Kinect

The purpose for this phase is to get familiar with Kinect’s API. The objective of the phase is to develop an application that can recognise hands in front of Kinect and return the position of palms and fingertips. This chapter is organized into 5 parts. Section 3.1 gives a brief description of the application and the technical details involved in the project are introduced in the following sections including depth calibration (Section 3.2), skin colour detection (Section 3.3), palm recognition (Section 3.4) and fingertips recognition (Section 3.5).

3.1 Application Description

The application takes advantage of Kinect’s features to fulfil the objective. The application is designed to recognise hands and return information similar to Leap Motion, which means it should returns a hand data structure for each detected hand, consisting of the central position of the palm, the number of fingertips and their positions.

An external library, OpenCV, is used in the application. OpenCV is an open source cross-platform library, commonly applied on real-time image processing. It implements most of the popular algorithms in image processing and computer vision. Most of the implementation is optimized and efficient. The other benefit from OpenCV is its API for matrix calculation. Unlike Matlab, a language designed for matrix manipulation, the standard C++ library does not include tools for this purpose, while image processing and computer vision are highly related to matrix. A heavy effort is required to avoid problems like memory leak and access violation. OpenCV has a series of data structures and functions related to matrix calculation,
which does compensate for the drawback of the standard C++. Last but not least, OpenCV provides some basic but sufficient GUI API. Therefore, it is not necessary to use any other complicated library for the GUI of the application.

The sequence diagram of the application is illustrated in figure 3.1. The application mainly consists of 4 components. The component with name Application is the main handler, cooperating with other components. HandDetector has the main logic for hand recognition. KinectSensor is responsible for communication with the Kinect device. GUI is the component interacting with users.

As illustrated in the figure, the major part of the application is inside a loop. For each iteration, the application first updates Kinect’s data, which will be accessed later. A flag called calibrated is initialized to be false. After updating the Kinect data, the application checks if the flag is set or not. If not, Application informs HandDetector to calibrate the depth threshold, which is used for filtered pixels behind the hands. Then in each iteration, HandDetector accesses the updated skeleton data from Kinect and stores it in a vector. If the number of data in storage is not enough for calibration, HandDetector continues to sample the skeleton data in the next iteration. When the samples are sufficient, HandDetector will calibrate the depth and gets the depth threshold, which is the depth value indicating the distance between hands and Kinect. The technical detail of calibration is in Section 3.2. Then the flag calibrated is set, indicating the calibration is complete. Once the depth threshold is found, the calibration function will not be run in the later iterations.

In iterations after calibration, the program follows a sequence to recognise the hands captured in the RGB image. As mentioned in the introduction of Kinect, the SDK can map a pixel in the RGB image into the depth data, getting corresponding depth value. Therefore, HandDetector first filters pixels with the depth value farther than the threshold from the RGB image and draws the filtered image on the canvas. The filtered image is similar to the original RGB image, expect that points behind the detected hands are drawn in black. Therefore, the body or the face of the user will be filtered if they are captured in the RGB image.

The depth-filtered RGB image is actually prepared for the skin colour filtering. Objects besides the hands may be included in the depth-filtered image as well. It will be the case in two situations. First, objects locates even closer to Kinect than the hands. Second, the depth threshold is not accurate enough so the sleeve of the user is kept in the image. The solution is using skin colour detection to filter pixels expect the hands. Skin colour has been proven to be a robust cue for different applications, including face detection and hand recognition [3]. There are different approaches [4] invented to find the skin colour in an image and the method mentioned by [5] is used in the application. The detail for this part is introduced in Section 3.3. For convenience purpose, the parameters involved in skin colour filtering are adjustable.
through the interface (illustrated in figure 3.2). Therefore, HandDetector will fetch the parameters from GUI before the processing.

With the image only containing hands, HandDetector can start to find the position of the palms and fingertips. It creates a Hand data structure to store the output information for each detected hand. Then several algorithms are involved in the process, referring to the approach used in [6]. The detail is introduced in Section 3.4 and Section 3.5.
chapter 3. Hand Recognition Application Using Kinect

(a) continue on next page.
CHAPTER 3. HAND RECOGNITION APPLICATION USING KINECT

(b) continue on next page.
CHAPTER 3. HAND RECOGNITION APPLICATION USING KINECT

(c) The final part.

Figure 3.1: The sequence diagram of the hand detection application.

Figure 3.2: The GUI for adjusting skin colour filtering parameters.
3.2 Depth Calibration

The applications for hand detection normally just use skin colour detection to filter pixels except the hands. However skin colour detection may not always provide robust performance. One scenario could be that user’s hand is just in front of it face. In this case, the face interferes the extraction of the hand because they are both in skin colour. In addition to skin colour detection, this application takes advantage of depth data from Kinect. The image can be filtered with a depth threshold, eliminating pixels behind the hands. Therefore, the threshold value should be a value close to the depth of hands.

In order to determine value for the depth threshold, the skeletal tracking feature of Kinect is used. As mentioned in the introduction, Kinect can track the skeletal data of the user, including the rough positions of hands. In the ideal case, the application is supposed to filter pixels behind the hands with real-time skeletal data. However, the skeletal data is not reliable. There are two kinds of errors. One is very minor, within 1cm. It may be caused by the noise in the depth data, because the skeletal data is analysed using the depth data. The other is much more significant. The position of the hands returns from Kinect could be around 20cm away from the realistic position. It happens when Kinect tracks the skeleton wrongly. Normally, this kind of error lasts for a while. Therefore, rather than using the real-time skeletal data, the application samples enough data and calculate a fixed depth threshold.

In the calibration phase, the user is required to face their hands towards Kinect and move their hands slowly around the preferred position. After the calibration, the user is supposed to interact with the application in the same area. The skeletal data has the three-dimension position of hands. Using Kinect’s API, the position can be translated into the depth value. During the calibration, the translated depth value is stored. Once sufficient samples are collected, HandDetector calculates the depth threshold. An assumption is made that the majority of the samples reflect the real depth value. Because the depth value is actually floating-point number, the desired depth value cannot be calculated by find the mode of the samples. DBSCAN, an algorithm in data clustering, is used to find the desired depth value.

DBSCAN is short for density-based spatial clustering of applications with noise. It is an algorithm proposed in 1996 [7] and, according to the Microsoft Academic Search database, it is the one of the most cited algorithm in scientific literature among all the clustering algorithm. Before explaining the detail of the algorithm, some terminologies are introduced as follows:

- The $\epsilon$-neighbour of a point $p$ is the area within a radius centred at $p$. $\epsilon$ is a user-specified parameter.
- The density of a neighbour is measured by the number of points inside the
neighbour.

- The $\epsilon$-neighbour of a point $p$ is dense if the density is at least $MinPts$, another user-specified parameter.
- If the $\epsilon$-neighbour of a point $p$ is dense, then point $p$ is a core object.
- A point $q$ is directly density-reachable from core object $p$ if $q$ is within the $\epsilon$-neighbour of $p$.
- A point $p$ is density-reachable from $q$ if there is a list of core objects $p_1, \ldots, p_n$, where $p_1 = p$, $p_n = q$ and $p_{i+1}$ is direct density-reachable from $p_i$ with respect to $\epsilon$ and $MinPts$.
- Two points $p$ and $q$ are density-connected if there is a point $o$ that $p$ and $q$ are density-reachable from $o$.

Then a cluster in DBSCAN is a subset of the data that satisfies two properties: (a) all points in the cluster are density-connected to each other; and (b) if a point is density-reachable from arbitrary point of the cluster, then it is included in the cluster as well.

To implement DBSCAN, all points in the data set are marked as ‘unvisited’. Then it selects an unvisited point $p$, marks it as ‘visited’ and checks the $\epsilon$-neighbour of $p$ is dense or not. If not, then $p$ is in a sparse area and is a noise point. If yes, then $p$ is a core object and DBSCAN creates a new cluster $C$ with $p$. To expand the cluster $C$, all points in $\epsilon$-neighbour of $p$ are added to a candidate set. Then DBSCAN searches other ‘unvisited’ core objects in the candidate set. If a core object, $p$, is found, DBSCAN marks it as ‘visited’ and points in $\epsilon$-neighbour of $p$ are added into the candidate set. The expansion of cluster $C$ when the candidate set is empty. Then DBSCAN finds another unvisited point from the remaining data and repeats the steps above until no unvisited points are left.

There are different alternative algorithms in data mining but DBSCAN is the most suitable one in the application. The distribution and histogram for a set of samples is illustrated in figure 3.3. The data contains two dense regions. The region around 0.67 represents the correct depth of hands, while the other dense region is collected when Kinect tracks the hand skeleton wrongly. DBSCAN can find these two regions easily with suitable parameters and some noise points (e.g. the point around 0.7 in figure 3.3a) can be removed as well. Then the cluster with the largest number of points is selected as the correct region and its mean is used to be the depth threshold. K-means can be a suitable solution in this case. However, Kinect does not always track the hand position wrongly. Then there will be only one region in the sample. At the same time, Kinect may track more than one wrong positions. Therefore, the number of clusters are not determined and K-means is not generally suitable for the calibration.
CHAPTER 3. HAND RECOGNITION APPLICATION USING KINECT

3.3 Skin Colour Detection

After filtering pixels behind the hands, the image is filtered again using the skin colour detection technique. There are two methods for skin detection, either pixel-based methods or region based methods. Pixel-based methods normally classify each pixel independently, while region based methods take the neighbouring pixels into account. Because the region based methods require more computation and may not suitable for this real-time application, the pixel-based method is chose.

In computer, there are different ways to represent colour. Different colour spaces can be used for skin colour detection. RGB is one of the most popular colour spaces and the original image in the application is stored in RGB space. However, RGB is not the most convenient option in the application. An example is illustrated in figure 3.4. The skin of a face is illustrated in figure 3.4a. There is shadow on the face. When representing the image in RGB colour space, the distribution is shown in figure 3.4b. It is obvious that the values in three dimensions have large ranges. It is difficult to extract the skin colour from others by defining upper bounds and lower bounds for the RGB values. In conclusion, the RGB value for the skin colour varies a lot with different brightness. Another colour space, HSV (hue saturation value), is chosen in the application. The colour distribution in HSV colour space is illustrated in figure 3.4c. Because the V value reflects the brightness of the pixels, values in V dimension has a large range. However, the projection on H dimension is very dense, from range 0 to 25. In HSV space, H value is relative stable in different lighting environment. Therefore, when representing images in HSV colour space, the skin colour can be extracted efficiently by defining an upper bound and lower bound for the H value.
Figure 3.4: The colour distribution for a face in RGB and HSV spaces.
3.4 Palm Recognition

The method to location the palm centre is by searching the largest possible circle inscribed in the hand. The centre of the circle is perceived as the centre of the palm (figure 3.5). To find the possible circle, the `pointPolygonTest()` function in OpenCV is used.

The `pointPolygonTest()` function requires the contour of the hand as one of the function input. In this application, a contour can be regarded as the edge of a shape. Then the application finds the contours in the image using `findContours()` function in OpenCV. Although the image has been filtered twice, there would be still some noise left in the image. Therefore, more contours are expected to be returned by the function. Fortunately, the region size of the noise tend to be relative small. A solution is to calculate the inner size of the contours and remove the ones with small size.

Then each contour is passed to the `pointPolygonTest()` function. The `pointPolygonTest()` function takes another parameter, a point in the image. The function returns the distance between the point and the nearest point in the contour. It returns a negative number if the point is outside of the contour. To find the maximum inscribed circle, each point in the image is tested and the point with the largest positive distance is the centre of the palm and the distance is the radius.

However, the computation is so costly that it is infeasible in the real-time application. The performance is improved with two strategies. First, the application iterates points on the contours and find the maximum and minimum values for the x and y dimensions. Then a rectangle can be determined for each contour that enclose the contour. The search for the centre point is restricted in the rectangle, which is much smaller than the entire image. Although the searching region is shrunk to a much smaller size, the number of point are still massive. Therefore, the second strategy is to sample points uniformly in the area and only the sampling points are tested by `pointPolygonTest()`. An example for these two strategies are illustrated in figure 3.6. The rectangle in blue is the searching region and the black dots inside are the sampling points. Then the computation for searching the centre of palm is largely reduced.
3.5 Fingertips recognition

The contours of hands are useful for fingertip recognition as well. To find the fingertips, the application first finds the convex hull for the contours as the polygon with cyan colour in figure 3.7. Then the application finds the convexity defects using the contour and the convex hull with function \texttt{convexityDefects()}. A convexity defect is a point on the contour. A convex hull has several intersections with the contour. For example, in figure 3.7, point \( A \) is one of the intersections and point \( B \) is the next intersection in the counter clockwise direction. Then the convexity defect, point \( C \), is the point on the contour between \( A \) and \( B \) with the largest distance to line section \( AB \). All the convexity defects in figure 3.7 are drawn in red. Some defects are in the gap between two fingers so they can be clues about the location of fingertips.

However, there are many defects in addition to the ones located in the gaps because \texttt{convexityDefects()} finds a defect for each pair of adjacent of intersections. To extract the defects between fingers, the application takes advantage of the other information returned by \texttt{convexityDefects()}. The function returns the distance between the defect and the line section along with the location of the defect point. The distance of defects in gaps tends to have a larger distance than other defects. For examples, the distance of \( C \) in figure 3.8 is 93, while the distance of \( F \) to \( DE \) is just 15. By filtering the defects with small distance from the set of defects, the result is illustrated on figure 3.8.

As mentioned above, a defect point has two corresponding intersections of the contour and the convex hull. In figure 3.8, the corresponding intersections are drawn in blue and green. These points are candidates for fingertips. In the figure, a green
Figure 3.7: The convex hull and convexity defects for a hand.

Figure 3.8: Candidate points for fingertips (in green and blue).

point may be so close to a blue point. For this case, these two points are merged into one point and the merged point is a point on the contour in the middle of the two points.

In some cases, the candidate points for fingertips may include undesired points. One possible case is illustrated in figure 3.9. Only one suitable defect point is found and there are two candidate points. Obviously, the blue one is not a fingertip. In order to filter undesired points in the fingertips, the k-curvature algorithm is used in the application. The algorithm starts from a candidate point $A$ and searches two points, $B$ and $C$, that $k$-points away from $A$ on the contour ($k$ is a user-specified parameter). One is on the left side of the candidate and the other is on the right. Then it calculates the angle formed by vectors $AB$ and $AC$. In the application, $k$ is normally set to 5. If the candidate is a fingertips, the angle tends to be less than 60 degree (the green point in figure 3.9). In this way, the blue candidate point in figure 3.9 is filtered because its angle is larger than 120 degree.

In conclusion, the position of the fingertips are found with the combination of these methods.
Figure 3.9: An example for the k-curvatures algorithm.
Chapter 4

Kinect and Leap Motion

The purpose for this phase is to find a method to implement the coordination between Kinect and Leap Motion. Then objective is to extend the Kinect application in the previous phase to an application combining Kinect and Leap Motion. This chapter includes 2 parts. Section 4.1 describes the details of the application, including the structure of the application. Then Section 4.2 discusses the scheme used for transformation between two systems.

4.1 Application Description

The functionality of the application includes transforming positions in Leap Motion system into Kinect coordinate system and visualising data from both systems in a single visualiser. From the technical view, positions in both systems can be transformed into positions in the other system with the same method. For the visualisation part, however, 480,000 points from Kinect are rendered in each frame, while no more than 12 points from Leap Motion are rendered. Therefore, it is much more efficient to transform the data from Leap Motion than the ones from Kinect.

The application consists of 6 components, Application, KSensor, KHandDetector, LMLListener, LM2K and PCV. As mentioned before, the application is an extension of the application in the last phase. Source code in the previous phase is reused. The details for each component is followed:

- **Application.** The component is the owner of the other components. It controls other components and helps them communicate with each other.

- **KSensor.** It is short for Kinect sensor. It is the same as the one in the previous application.
• **KHandDetector.** It is short for Kinect hand detector. It is the *HandDetector* in the previous application with some extra interfaces to output the detected hand information. It does not output the hand data until it finishes the calibration.

• **LMLListener.** It is short for Leap Motion listener. It is the same as the one in Phase 1, which has its own thread to update the Leap Motion device frequently.

• **LM2K.** The component transforms the Leap Motion data to Kinect coordinate system. Because both data from Leap Motion and *KHandDetector* are required for finding the relationship, it will not start to process until *KHandDetector* finishes the calibration. The process has two phases. In Phase 1, it collects data from both systems for five seconds. Then it calculates the relationship between two coordinate systems and goes to Phase 2. In Phase 2, it takes only the data from Leap Motion as input and transforms the data into positions in Kinect system using the relationship calculated in Phase 1. More details for this component is in Section 4.2.

• **PCV.** It is the visualiser in the project and it is short for point cloud visualiser. Because it requires the Kinect data and the transformed Leap Motion data, it does not start to process until LM2K goes to phase 2. It uses the external library, the Point Cloud Library. Each element in Kinect’s depth data can be transformed into a point in 3D space, indicating the relative position to the device. Moreover, each element in depth data can map to a pixel in RGB image. Then data from Kinect can be represented with many colourful points in a 3D space. *PCV* takes the points as input and uses the Point Cloud Library to visualise them. Then it draws the transformed Leap Motion data in the same space so that users can observes how well the Leap Motion data matches the Kinect data.

The final view of the application is illustrated in figure 4.1.
4.2 Transformation from Leap Motion to Kinect

As the discussion in the previous phases, both the Kinect application and the Leap motion application can return position of fingertips. However, the relative position between hands and Kinect is different from the one between hands and Leap Motion, which determines that the fingertip positions from two systems cannot be mapped directly. The application needs to find the relationship between two systems and transforms one to the other.

In addition to transformation, the error in both systems should be taken into account. Ideally, a fingertip position in one system should fully match the position in the other system after transformation. However, there exists errors in both systems. It is infeasible to find a transformation that maps points in one system exactly into points in the other.

The application uses the method proposed in [8] to solve the problem. The method names the two systems the left system and the right system. There are n points in both systems and they are denoted by

\[ \{r_{l,i}\} \text{ and } \{r_{r,i}\}, i = 1 \ldots n \]

Points in the left system are transformed to the right system. For \( r_{l,i} \) in the left system, \( r_{r,i} \) is the corresponding point in the right system. Therefore,

\[ r_{r,i} = sR(r_{l,i}) + r_0 \]  

(4.1)
s is the scalar factor, R is the rotation matrix and $r_0$ is the translation vector. As mentioned before, a point in the left system cannot map to the point in the right system exactly. The error is expressed as,

$$e_i = r_{r,i} - sR(r_{l,i}) - r_0.$$ 

Then the problem is to find a set of $s$, $R$ and $r_0$ that minimises the sum of the squares of these errors

$$\sum_{i=1}^{n} \| e_i \|^2.$$ 

The method first finds the mean coordinate for each system:

$$\bar{r}_l = \frac{1}{n} \sum_{i=1}^{n} r_{l,i} \text{ and } \bar{r}_r = \frac{1}{n} \sum_{i=1}^{n} r_{r,i}.$$ 

Then $\sum_{i=1}^{n} r'_{l,i} = \sum_{i=1}^{n} r'_{r,i} = 0$. Using this property in addition to the properties of matrices, the method gets a closed-form solution as follows:

$$s = \sqrt{\frac{\sum_{i=1}^{n} \| r'_{l,i} \|^2}{\sum_{i=1}^{n} \| r'_{r,i} \|^2}}$$

$$r_0 = \bar{r}_r - sR(\bar{r}_l)$$

$$R = M(M^T M)^{-1/2},$$

where

$$M = \sum_{i=1}^{n} r'_{r,i}(r'_{l,i})^T.$$ 

Once the $s$, $R$ and $r_0$ are known, points in the left system can be transformed to the right system using equation (4.1).

To implement the method, the application first collects data from both systems for 5 seconds. Because of the interference of the environment, both Kinect and Leap Motion may return unreliable data. Therefore, users are required to hold their single hand or both hands above Leap Motion during the phase 1 of LM2K. The palms should be open, be facing Kinect and tilt forwards slightly. The application collects data only when Leap Motion and Kinect detect the same number of hands and both of them detect 5 fingertips for each hand.

With the help of OpenCV, most of the matrix calculation can be implemented in C++ easily except the calculation of $R$. The calculation of $R$ requires to solve a matrix to a negative half power. OpenCV does not provide API for this operation.
Therefore, more effort is made on this operation. First, a matrix to the minus one is equal to the inverse of the matrix and OpenCV provides API to solve the inverse of matrices. Then the problem is to solve the square root of the matrix. There are several computation methods and the application solves it by diagonalisation. The method finds the eigenvalues and eigenvectors of the matrix using OpenCV. Then the matrix is expressed as

\[ M^T M = V D V^{-1}. \]

When each column of \( V \) is one of the eigenvectors and \( D \) is a diagonal matrix with the eigenvalues as the diagonal. Then the square root of the matrix is

\[ (M^T M)^{\frac{1}{2}} = V D^{\frac{1}{2}} V^{-1}. \]

After calculating \( s, R \) and \( r_0 \), \( LM2K \) component starts Phase 2. In Phase 2, it transforms data from Leap Motion using equation (4.1). The result is passed to the visualiser.
Chapter 5

Hand Gesture Recognition Application

Because the constraint on time, the hand gesture recognition application is not implemented on time. A brief design of the application is discussed in this chapter.

The structure of the application is illustrated in figure 5.1. The application is separated into two threads. The gesture recognition part is in thread 1, while thread 2 is mainly responsible for other functionalities, such as GUI. Each component is introduced as follows:

- **KLM Hand Detector.** It is a component like the application in the previous phase. It outputs the hand information detected by Kinect and the transformed hand information from Leap Motion. Besides the palm and fingertips positions, Leap Motion provides other information of hands and the information is passed to **Hand Combiner** as well.

- **Hand Combiner.** It merges the information from both systems. It is not a trivial task, because it is necessary to determine which data is more reliable before merging. Data from reliable system should have larger weight during the merging process. The output is a data structure containing the merged information. It should follow the Hand structure of Leap Motion API, providing extra information in addition to positions, such as hand’s velocity and palm’s angle.

- **Gesture Recogniser.** It is an abstract class. It declares the general component for a gesture recogniser. For different gesture recognisers, the same hand information may have different meaning for them. Therefore, each gesture recogniser should define its **Hand Event Convertor.** It takes the merged hand information as input and converts it to an event as input for FSM. Other In-
formation contains specific information useful for different gesture recognisers. For example, the recogniser for drawing circle will store information like the drawing speed, circle radius and so on.

- **Circle Gesture Recogniser and other recognisers.** They are child classes of Gesture Recogniser. In each iteration, the hand information will broadcast to all recognisers and they process independently.

- **Event Handler and Callback function.** *Event Handler* is the interface between two threads. Different callback functions are registered in *Event Handler*. Each callback function is mapped to the corresponding gesture. Once a gesture is recognised, a signal is sent the event handler and the corresponding callback function is run automatically.
Figure 5.1: The software structure of the hand gesture recognition application.
Chapter 6

Future Improvement

The task in the future with the highest priority is to implement the design in Chapter 5. Besides, there are several attempts that can be made in the future in order to enhance the application performance.

Although HSV skin colour detection is applied in the project, the performance of the application is still sensitive to the environment lighting. It is not caused by the algorithms but the defect of Kinect. As mentioned in the beginning of the report, Kinects RGB camera has poor performance in low light environment, especially in term of CRI. An image captured by Kinect in the low light environment is illustrated in figure 6.1. The hand’s colour is close to purple, which is not inside the skin colour threshold. The histogram of hue (H) value of the hand area is illustrated in figure 6.2. The majority has value from 150 to 250, while the value should be around 20 in the normal light environment. There are two attempts to overcome the problems. On the one hand, the problem can be solved from the hardware aspect. One simple way is to only use the depth sensor of Kinect and use a good quality web cam to replace Kinect’s RGB camera. However, attention should be paid for the synchronisation between the web cam and the Kinect depth sensor. On the other hand, the problem can be alleviated using software methods. The skeletal data from Kinect gives a rough position of hands. Then the application can determine the luminance by pixels around the hand areas. Different HSV thresholds are pre-set for different luminance environments. Another attempt is to improve the fingertip detection. The current application in Chapter 4 returns only the number of the detected fingertips and their position. It cannot recognise which finger is the thumb and which is the index finger. Some simple methods have been proposed in [6] and [9]. Some more complicated algorithms can also be used for this purpose, such as pattern recognition.

Last but not least, the performance of the application may be improved using machine learning. It is a popular trend to apply machine learning to computer vision.
CHAPTER 6.  FUTURE IMPROVEMENT

Figure 6.1: An image captured by Kinect at night.

Figure 6.2: Histogram for H value of the hand in figure 6.1
For example, [6] uses machine learning to train a gesture recogniser. Therefore, an effort should be made on attempt at combining technologies in machine learning with the project.
Chapter 7

Conclusion

In this project, three applications are developed. In Phase 1, the gallery application is developed, which is controlled by Leap Motion. It is a simple but convenient application. In the second phase, the hand detection application using Kinect is developed. Different algorithms are implemented to fulfil the requirement and make the application more robust. In Phase 3, the hand detection application is extended and combined with Leap Motion. Data from both systems are visualised in a 3D way.

The project fulfils the requirements in the initial plan except the last phase. The gesture recognition application is not implemented during the project. It is due to the unsatisfied time management and over-estimation of the software development skill.

However, the outcome of the project is meaningful for any follow-up work. Taking advantage of OOP, the structure of the source code is well organised. Detail comment is appended to the source code. Therefore, the code is readable and usable in the relevant projects and other future work.
Bibliography


