Gesture Recognition for Vision Based Interaction

BACHELORARBEIT 2

Student Christian Mayr, 0910601022
Betreuer DI Robert Praxmarer

Salzburg, 7. Mai 2012

Eidesstattliche Erklärung

Ich erkläre hiermit eidesstattlich, dass ich die vorliegende Bachelorarbeit selbständig und ohne fremde Hilfe verfasst, und keine anderen als die angegeben Quellen und Hilfsmittel benutzt habe. Weiters versichere ich hiermit, dass ich die den benutzten Quellen wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungskommission weder im In- noch im Ausland vorgelegt und auch nicht veröffentlicht.

Datum Unterschrift

Kurzfassung

Vor- und Zuname:	Christian MAYR
Institution:	FH Salzburg
Studiengang:	Bachelor MultiMediaTechnology
Titel der Bachelorarbeit:	Gesture Recognition for Vision Based Interaction
Begutachter:	DI Robert Praxmarer

Seit mit Erscheinen des Kinect–Sensors die Idee der gestengesteuerten Interaktion den Massenmarkt durchdrungen hat, besteht hohes Interesse an Erforschung dieser Materie. Den daraus folgenden Fortschritten ist es zu verdanken, dass es heute möglich ist, für die breite Masse Anwendungen zu entwickeln, mit denen ohne zusätzliche Eingabegeräte interagiert werden kann. Diese auf Gesten basierenden Eingabemethoden können in vielen Bereichen zum Einsatz kommen, wie etwa in Computerspielen oder Anwendungen für Virtual Reality.

Wir haben eine Anwendung entwickelt, in der die teilnehmenden Benutzer und Benutzerinnen durch eine Kinect–Kamera mit einem virtuellen Kolibri interagieren können. Streckt der Benutzer oder die Benutzerin seine oder ihre Hand aus, so wächst aus seinem oder ihrem Handballen eine Blume, die die Aufmerksamkeit des Kolibris erweckt.

Nachdem die Erkennung der Geste einer ausgestreckten Hand die Basis für die funktionierende Interaktion unseres Programms bildet, konzentriert sich unsere Arbeit auf diesen speziellen Anwendungsfall. Zusätzlich tut sich in unserem Szenario die Notwendigkeit auf, für eine korrekte Gestenerkennung die Orientierung der Hand mit Hilfe der Fingerspitzenpositionen zu ermitteln.

Im Zuge unserer Forschung vergleichen wir verschiedene Algorithmen zur Erkennung von Gesten und Fingerspitzen und stellen die Vor– und Nachteile ihrer Einsatzfähigkeit unserer speziellen Anforderung gegenüber. Die Erkenntnisse dieses theoretischen Vergleiches bilden die Entscheidungsgrundlage zur finalen Implementierung. Danach bewerten wir unsere Lösung auf ihre Praxistauglichkeit, indem wir unseren Gestenerkennungsalgorithmus in verschiedenen Größen– und Sichtverhältnissen auf ihre Robustheit testen.

Schlagwörter: Gesture Recognition, Kinect, Depth Imaging, Convex Hull, Finger–Earth Mover's Distance, Curvature Morphology, K–Curvature Estimation

Abstract

Since the release of Microsoft's Kinect technology gesture recognition has become widely accessible to the public. As a result of this increased research interest it is possible to develop applications for the masses, which can be controlled without any additional input devices. Those input methods can be used within various setups such as computer games or applications for virtual reality.

We developed a software, where users can interact with a hummingbird through a Kinect camera. When the user stretches out his or her arm, a flower grows out of his or her palm attracting the bird's attention.

As the robust detection of the gesture of an outstretched arm is one of the key features for flawless interaction, our thesis will emphasize on this topic. Additionally, for stable gesture detection we need to extract the fingertip positions to gather information about the hand's orientation.

In our research, we compare various algorithms detecting gestures and fingertip positions and draw conclusions from their advantages and disadvantages to usage in our practical setup. The results of this theoretical comparison lead us to a decision for an algorithm implemented in our final setup. In the end, we evaluate our solution by testing the robustness of our gesture detection algorithm within various scales and point of views.

Keywords: *Gesture Recognition, Kinect, Depth Imaging, Convex Hull, Finger–Earth Mover's Distance, Curvature Morphology, K–Curvature Estimation*

Contents

1 Overview **1**

1.1 Motivation 1

1.2 Practical Setup 2

1.3 Requirements 2

1.4 Research Question and Scientific Methods 3

2 Gesture Recognition **3**

2.1 Hardware Setup 4

2.2 Survey of General Methods and Algorithms 5

2.2.1 Dynamic Time Warping 5

2.2.2 Hidden Markov Models 5

2.2.3 CONDENSATION 6

2.2.4 Body Part Recognition based on Depth Imaging 7

2.3 Recognition of Hand Gestures and Finger Positions 9

2.3.1 Shape Detection using Convex Hull 9

2.3.2 Finger–Earth Mover's Distance 10

2.3.3 Curvature Morphology and Support Vector Machines 12

2.3.4 K–Curvature Estimation 14

3 Implementation **15**

3.1 Overview 15

3.1.1 Scene States 15

3.1.2 Interaction States 16

3.2 Used Methods and Algorithms 17

3.2.1 Pose Detection 17

3.2.2 Fingertip Detection 18

3.3 Final Results 22

4 Conclusion and Answer to the Research Question **24**

List of Abbreviations

CADET Center for Advances in Digital Entertainment Technologies
CONDENSATION Conditional Density Propagation
CPU Central Processing Unit
DTW Dynamic Time Warping
e.g. exempli gratia
FEMD Finger–Earth Mover's Distance
GUI Graphical User Interface
GPU Graphics Processing Unit
HCI Human Computer Interaction
HD High Definition
HMM Hidden Markov Model
i.e. id est
LCD Liquid Crystal Display
NITE Natural Interaction
OpenCV Open Source Computer Vision
RGB Red Green Blue
SDK Software Development Kit
XML Extensible Markup Language

1 Overview

1.1 Motivation

Gesture recognition provides a natural way in communication between humans and machines. Widely available and affordable components like Microsoft's Kinect make this technology widely accessible to the public, which gives real–world computer vision applications a boost (Doliotis et al. 2011). Beside Microsoft also other companies like for instance GestureTek[1] or Reactrix[2] offer hardware solutions for gesture controlled interaction.

Other domains where Human Computer Interaction (HCI) benefits of the usage of gesture recognition are for instance applications in virtual reality, sign language recognition, computer games or human–robot interaction (Ren, Yuan, and Zhang 2011; Kim et al. 2010). Also, gesture recognition can be useful for interaction over long distances where no speech information is available (Kim et al. 2010).

When concentrating on the aspect of advertising it has become more and more popular to display electronic advertisements on large displays like plasma panels. But most of these "electronic posters" provide only one–way information lacking interactivity (Fukasawa, Fukuchi, and Koike 2006). In combination with the latest approaches in computer vision it is possible to let the user interact with those large screens by gestures. This active involvement attracts attention in a way static advertisements can't and is therefore a promising market sector for the future.

We created an implementation of an advertisement program where a hummingbird flies around the shape of interacting users. When one of them stretches his or her arm, a flower grows out of his or her fingertip and attracts the bird's attention (Figure 1).

Figure 1: In our implementation, a hummingbird flies around a flower growing out of the user's hand.

To achieve this, we need to implement a robust algorithm detecting the gesture of an outstretched arm. This thesis will concentrate on this topic and analyze different approaches and ideas, which can be used for an implementation of a proper algorithm. We prove that, with the usage of Microsoft's Kinect sensor and the established and well researched driver setup, the extraction of

1. http://www.gesturetek.com
2. http://www.reactrix.com

the user's skeleton data works reliable using the OpenNI Framework[3] or the Kinect for Windows SDK[4]. Because of that, the recognition of the arm position can be done pretty easily, but to identify an outstretched arm correctly, it is important to take the orientation of the fingertips also in account as you can see in Figure 2.

(a) Fingertips pointing down (b) Fingertips pointing up

Figure 2: The hand's orientation of an outstretched arm can be identified by extracting the fingertip positions. In our use case only the gesture of an outstretched arm with fingertip positions pointing upwards should be detected.

As a result, the recognition of the user's fingertips remains a challenging task because of the low resolution of the Kinect sensor in comparison to the size of a human hand in the scene (Ren, Yuan, and Zhang 2011). So we need to evaluate different approaches for solving this issue too. Based on our theoretical evaluation we will implement an algorithm that detects the gesture of an outstretched arm and integrates it as crucial part for our advertisement program. Finally, we will review the practical results gained by executing our final implementation for justifying the decisions we made in our theoretical evaluation.

1.2 Practical Setup

Our program has been implemented using the Cinder library[5] in combination with CADET's 2RealKinectWrapper[6]. The decision to this wrapper has been made because it natively supports both the OpenNI Framework and the Kinect for Windows SDK. As a result, the user benefits of this high level of abstraction by choosing the drivers of his or her choice.

The software is targeted on being displayed on a plasma or LCD panel in Full HD resolution while a Kinect sensor (or any other depth camera that is supported by the framework) tracks the user's actions. Users, who enter or leave the screen, should be tracked and recognized in real time while a hummingbird flies around their shape and reacts accordingly. More detail on the hummingbird's degree of interaction will be proposed in Chapter 3.1.2.

One of the most important points for seamingless and believable interaction within the context of our setup is the implementation of a stable and reliable gesture detection algorithm as pointed out in Chapter 1.1.

1.3 Requirements

Since this thesis emphasizes on the correct recognition of hand gestures, we'll provide a basic overview over the requirements that the algorithm of our choice needs to fulfill:

3. `http://www.openni.org`
4. `http://www.kinectforwindows.org`
5. `http://libcinder.org`
6. `http://www.cadet.at`

- No usage of markers, gloves or any other supporting devices that need additional attachment to the user. His or her hand should be trackable without the help of any other devices. A fast and stable interaction between user and software should be ensured without any boundaries.
- The gesture detection should be translation and scale invariant, allowing occurrence in any part of the image as long as skeleton data is available.
- It should perform well even in challenging environments with difficult lighting conditions and cluttered backgrounds allowing robust hand tracking.
- Fingertip detection should work well within different point of views to the user's hand, especially to the side view, which is required for the detection of an outstretched arm.
- No usage of an extensible amount of training data, because it just adds an unwanted and unrequired amount of complexity to our solution, which should only be capable of the detection of one simple gesture.

1.4 Research Question and Scientific Methods

Because of the vast amount of research and solutions for gesture detection we mainly concentrate on the field of fingertip and hand gesture recognition using depth cameras. Solutions for this problem include algorithms based on convex hull, the calculation of the Finger–Earth Mover's Distance, curvature morphology and k–curvature estimation. Before introducing these topics we'll give a basic overview over techniques handling gesture recognition in general. Later on we'll evaluate all these efforts in comparison to the requirements outlined in Chapter 1.3. The algorithm which fulfils the requirements best within our theoretical evaluation will be considered for our practical implementation.

Finally, our research leads to following research question, which should both evaluate the feasibility and the algorithm of our choice for stable arm detection in connection to our use case:

How can the detection of fingertips, derived from the Kinect's depth image, be used to improve a stable detection of an outstretched hand in an interactive setup?

2 Gesture Recognition

Gesture recognition has become a widely and overly well researched topic within the last decade. Because of the enormous amount of information we'll provide general definitions and restrains regarding our use case.

Weinland, Ronfard and Boyer classify gesture recognition as sub–topic of visual action recognition besides facial expression recognition and movement behavior recognition for video surveillance (Weinland, Ronfard, and Boyer 2011, 4). Furthermore, they define the generic term action recognition as following:

> "Action recognition is the process of naming actions, usually in the simple form of an action verb, using sensory observations. Technically, an action is a sequence of movements generated by a human agent during the performance of a task. As such, it is a four–dimensional object, which may be further decomposed into spatial and temporal parts." (Weinland, Ronfard, and Boyer 2011, 4)

For our implementation in Chapter 3 and the theoretical debate within the current chapter we are only concerned with visual observations by means of a video camera. The decomposition and analysis of spatial and temporal sequence of movements will be handled by the software using one or more suitable algorithms, which will be introduced in the following paragraphs.

Additionally, we need to define the general terminology of movement and activity. Bobick defines those terms within the context of our research as following:

> "Movements are the most atomic primitives, requiring no contextual or sequence knowledge to be recognized; movement is often addressed using either view–invariant or view–specific geometric techniques. Activity refers to sequences of movements or states, where the only real knowledge required is the statistics of the sequence." (Bobick 1997)

These very basic definitions are important for our research since they form the foundation for all of the topics treated within this thesis.

2.1 Hardware Setup

Our implementation is based upon the usage of the Kinect sensor, which has initially been introduced by Microsoft in November 2010 for the Xbox 360 architecture. The device includes a RGB camera, a depth sensor and a multi–array microphone, which for instance allows interaction by capturing gestures, body motion, facial expressions or the usage of spoken commands (Microsoft Corporation 2010). Figure 3 offers a basic overview over the hardware architecture of the Kinect hardware.

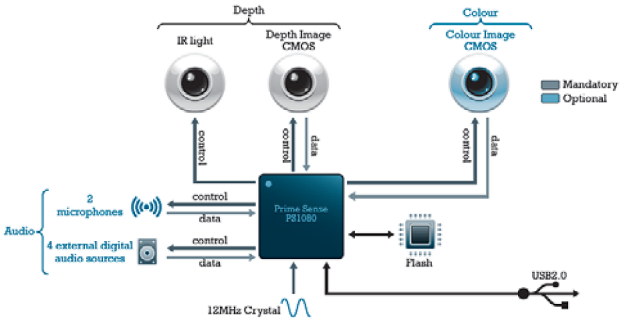


Figure 3: This figure contains an overview over Kinect's hardware components (Raheja, Chaudhary, and Singal 2011).

Our implementation is based upon this technology because of the following reasons:

- There is lot of research in the context of gesture recognition and Kinect in progress, allowing us the usage of various well developed and proven frameworks like NITE or skeletal tracking outlined in Chapter 3.2.1.
- The consumer price is relatively low for a depth camera and so the financial costs even for multiple setups of our program are still affordable.
- Microsoft actively supports community development of various applications for this hardware and developers by providing a well documented and extensive SDK.
- Kinect is accessible by usage of drivers developed by the company PrimeSense, which additionally offers compatibility to other depth cameras which can also be accessed through these drivers.

2.2 Survey of General Methods and Algorithms

In this chapter we will evaluate some of the most popular approaches in gesture recognition algorithms in general and more specifically how they can be used for detecting and interpreting hand gestures. The advantages and limitations of each solution will be evaluated with attention to our implementation.

2.2.1 Dynamic Time Warping

Doliotis et al. deployed a translation– and scale–invariant gesture recognition algorithm based on Dynamic Time Warping (DTW) calculated upon depth and color images gathered from the Kinect sensor (Doliotis et al. 2011).

DTW is a robust distance measure for time series and detection of similar shapes. A big advantage of this technology is its flexibility with usage in several fields such as science, medicine, industry and finance. All these applications have in common that they need to find the best match to a query time series from a pool of candidates (Keogh and Ratanamahatana 2005).

Doliotis et al. describe the similarity measure of Dynamic Time Warping as following:

> "The DTW algorithm temporally aligns two sequences, a query sequence and a model sequence, and computes a matching score, which is used for classifying the query sequence." (Doliotis et al. 2011)

There is a vast amount of further literature available which tackles this algorithm in more detail, like for instance Keogh's and Ratanamahatana's paper handling indexing using DTW (Keogh and Ratanamahatana 2005).

We won't examine DTW any further, because the disadvantages discussed by Doliotis et al. make algorithms based on DTW unsuitable for our implementation. DTW requires a perfect hand detector and assumes that it receives the perfect hand location for every frame. Another restriction is the viewpoint invariance of their implementation. They assume that the user always faces the camera for a frontal view of the gesture (Doliotis et al. 2011).

This is a highly unwanted limitation for our setup, where gestures should be traceable within various point of views.

2.2.2 Hidden Markov Models

Derpanis mentions that "Hidden Markov Models (HMMs) by far have received the most attention in literature for classifying gestures" (Derpanis 2004, 9). Before Starner, Weaver and Pentland firstly introduced HMMs for hand gesture recognition they were prominently and successfully used for speech and handwriting recognition (Starner, Pentland, and Weaver 1998).

The statistical methods of Markov source or hidden Markov modeling have initially been studied and introduced in the late 1960s and early 1970s. They try to characterize real–world signals (like for instance speech samples, temperature measurements or music) in terms of signal models (Rabiner 1989).

Derpanis defines the architecture of HMMs as following:

> "An HMM consists of a number of hidden states, each with a probability of transitioning from itself to another state. The transitioning probabilities are modeled as nth order Markov processes (i.e. the probability to transitioning to a new state only depends on the n previous states visited)." (Derpanis 2004, 10)

Also, the topology allows states to transition to themselves, which give HMMs a high degree of time–scale invariance (Figure 4). For recognizing gestures "an HMM is constructed for each of the gestures under consideration" (Derpanis 2004, 10).

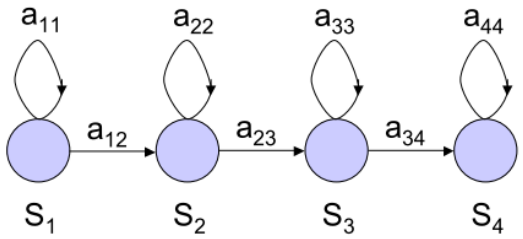


Figure 4: An overview over hidden states of a Markov model: s_i denotes a state and a_{ij} the probability of a state transition from state i to j (Derpanis 2004, 10).

For gathering more information about HMMs and an overview over algorithms we recommend Rabiner's and Juang's introductional article (Rabiner and Juang 1986). We won't go into more detail here because of the disadvantages discussed in the next paragraph.

In practice, Starner's, Weaver's and Pentland's attempt in using HMMs for the detection of American Sign Language features high accuracy in detection of various gestures. But in order to understand different subjects with their own variations of language they require collecting enormous amounts of different data (Starner, Pentland, and Weaver 1998).

In comparison to the requirements for our implementation the collection of training data is a major flaw. The amount of work in collecting this data is disproportionate to the complexity of our implementation where only one gesture needs a stable detection. Additionally, Derpanis mentions two more disadvantages leading to unwanted complexity:

> "A significant problem is that there is no principled way of defining the topology (i.e. number of states, number of transitions). Instead, defining the topology relies on educated guesses or trial and error, which is a non–trivial task when attempting to model a large gesture language. Another disadvantage lies in the Markovian assumption of transitioning from one state to the next. This assumption does not in general map well to real–world processes." (Derpanis 2004, 10–11)

Because of these reasons we won't rely on algorithms based on HMMs for finding the gesture of an outstretched arm.

2.2.3 CONDENSATION

Black and Jepson employed a framework that uses the CONDENSATION algorithm for gesture recognition. Their approach can be interpreted as a generalization of HMMs discussed in Chapter 2.2.2, because they also allow a set of states and transitions between them. As for state recognition, their algorithm "involves the probabilistic matching of an entire temporal trajectory model that represents a portion of the gesture" (Black and Jepson 1998, 911). They mention that their interpretation works "similar to DTW but within a unified probabilistic framework" (Black and Jepson 1998, 911).

The CONDENSATION algorithm itself has initially been proposed by Isard and Blake. They define CONDENSATION as following:

> "It uses 'factored sampling', a method previously applied to interpretation of static images, in which the distribution of possible interpretations is presented by a randomly

> generated set of representatives. The CONDENSATION algorithm combines factored sampling with learned dynamical models to propagate an entire probability distribution for object position and shape, over time." (Isard and Blake 1996, 343)

They note that their implementation of motion tracking within image or movie sequences works effective in clutter (Figure 5) (Isard and Blake 1996, 354). Black and Jepson used CONDENSATION only for recognition, while the trajectories of gestures have been estimated using color markers (Black and Jepson 1998, 923). Doliotis et al. state as disadvantage of the usage of CONDENSATION in comparison to DTW that the algorithm needs additional knowledge about the observation and propagation density (Doliotis et al. 2011).

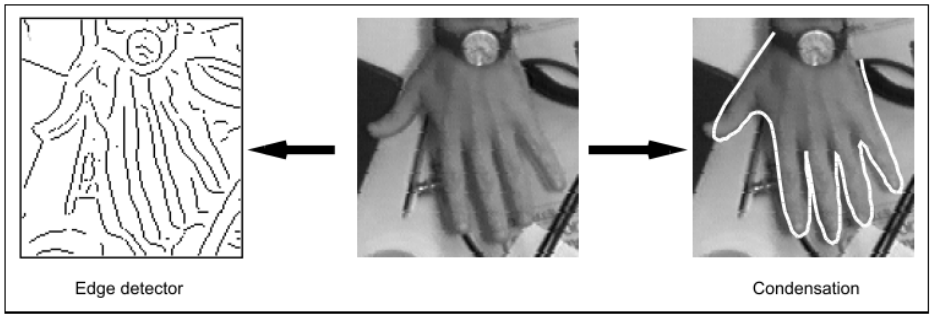


Figure 5: When only tracking the edges it occurs that there are many clutter edges that distract the system, while the CONDENSATION algorithm succeeds in tracking the hand through this distractions (Isard and Blake 1996, 355).

A main limitation of CONDENSATION in reference to our use case is once more the need to record user data before being able to recognize gestures accurately. Additionally, the examples performed within Black's and Jepson's framework for testing and evaluating gesture recognition are rather coarse as they concentrate on tracking the hand position using some sort of physical icon (Black and Jepson 1998, 915). Both the constrains on using supporting devices for the tracking and comparison of gestures and the limitation, that their framework is not able to interpret gestures with a high amount of samples in real time (Black and Jepson 1998, 923), lead us to the decision to refrain from the idea of using CONDENSATION algorithms for gesture recognition.

2.2.4 Body Part Recognition based on Depth Imaging

Since our implementation is based upon Microsoft's Kinect sensor, an analysis of the methods used for skeletal tracking provided by the SDK and a verification if they are reliable enough for stable gesture recognition seems reasonable.

Shotton et al. from Microsoft Research recognize the human shape by depth imaging, which offers several advantages over traditional intensity sensors like the ability to work in low light levels or being color and texture invariant. They captured a large database of motion capture of human actions and defined several localized body parts (Figure 6). Algorithms based on randomized decision forests are responsible for the task in disambiguating these trained body parts (Shotton et al. 2011).

Sharp managed to implement a method that can evaluate and train decision trees entirely on the GPU, which creates results for object recognition — identical to those obtained on a CPU — within one percent of the time compared to the calculation on CPUs (Sharp 2008, 606). He explains the main course of actions as following:

> "In computer vision techniques, the input data typically correspond to feature values at pixel locations. Each parent node in the tree stores a binary function. For each

Figure 6: Shotton et al. captured a large database of motion capture of human actions with attention to variety in pose, shape, clothing and crop (Shotton et al. 2011).

> data point, the binary function at the root node is evaluated on the data. The function value determines which child node is visited next. This continues until reaching a leaf node, which determines the output of the procedure. A forest is a collection of trees that are evaluated independently." (Sharp 2008, 596)

The trees are trained on sets of these synthesized pictures of user's body parts with a random subset of example pixels from each image. They consist of split and leaf nodes as you can see in Figure 7. In more detail, the following steps are performed within the generation of training data:

> "Each split node consists of a feature f_θ and a threshold τ. To classify pixel x in image I, one starts at the root [...], branching left or right according to the comparison to threshold τ. At the leaf node reached in tree t, a learned distribution $P_t(c|I,x)$ over body part labels c is stored. The distributions are averaged together for all trees in the forest to give the final classification $P(c|I,x) = \frac{1}{T}\sum_{t=1}^{T} P_t(c|I,x)$." (Shotton et al. 2011)

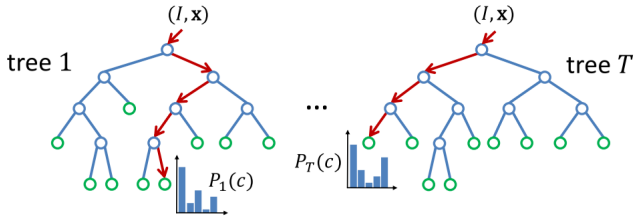


Figure 7: "A forest is an ensemble of trees. Each tree consists of split nodes (blue) and leaf nodes (green). The red arrows indicate the different paths that might be taken by different trees for a particular input" (Shotton et al. 2011).

Shotton et al. come to the conclusion that "proposals for the 3D locations of body joints can be estimated in super real–time from single depth images" (Shotton et al. 2011). Their results "show high correlation between real and synthetic data, and between the intermediate classification and the final joint proposal accuracy" (Shotton et al. 2011). Additionally, they mention the high performance their algorithm can achieve when the computation takes place on the GPU.

Because of these promising results we'll rely on data gathered by their implementation of body part recognition for our pose estimation. As already pointed out, the usage of the Kinect's depth image for skeletal tracking guarantees stable and reliable results even in challenging environments. Another reason crucial for our decision is the fact that we don't need to gather any training data for our implementation, because the work has already been done within the creation of trained decision forests.

Summing up, body part recognition by Shotton et al. provides a setup that fulfills almost all of our requirements defined in Chapter 1.3. Beside the usage of their data for our implementation of pose estimation we need to take care of an implementation of a stable algorithm for detecting fingertips.

2.3 Recognition of Hand Gestures and Finger Positions

As stated in Chapter 2.2.4, the usage of skeleton data allows us an easy detection of arm gestures, but we additionally need to track the user's finger positions for a reliable detection of an outstretched arm. Ren, Yuan and Zhang describe the main problem for the implementation of a stable and reliable algorithm, which is able to track hand gestures and finger positions, as following:

> "As for the gesture recognition, even with the Kinect sensor, it is still a very challenging problem. Because typically, the resolution of a Kinect sensor is only 640 x 480. Although it works well to track a large object, e.g. the human body, it is difficult to detect and segment precisely a small object from an image at this resolution, e.g., a human hand that occupies a very small portion of the image." (Ren et al. 2011)

Good examples for this issue are illustrated in Figure 8.

Figure 8: Here are a few examples of challenging cases listed, where hand gesture recognition using depth cameras may fail. While the first and second hand have the same gestures, the third hand confuses the recognition (Ren, Yuan, and Zhang 2011).

Fortunately, a few techniques that overcome these limitations have been introduced. The results of these approaches will be evaluated within the next chapters. As for the requirements of our use case listed in Chapter 1.3 we'll mainly focus on the detection of fingertips.

2.3.1 Shape Detection using Convex Hull

Multiple solutions in extracting fingertip positions around a hand position by usage of a convex hull have been published (Cristina Manresa and Perales 2005; Frati and Prattichizzo 2011).

By definition, the convex hull of a set of points is the smallest convex set containing them (Buss 2003, 117). Figure 9 illustrates an example.

Frati and Prattichizzo explain an easy and straightforward way in extracting the fingertip positions by using functions provided by the OpenCV framework. Since we already use functionality of this framework for background subtraction within our implementation, their solution could easily be integrated into our work without generating any additional dependencies.

They determine the convex hull by the following steps:

> "The function used to find the contour for objects in OpenCV is cvFindContours(): it takes a binary image and returns the number of retrieved contours. The binary image is computed from the cropped image. Once the contour is obtained, it is possible to compute its convex hull using the OpenCV function cvConvexHull(). The points of the hull represent the external contour of the hand, from the wrist to the fingers if the hand is open." (Frati and Prattichizzo 2011)

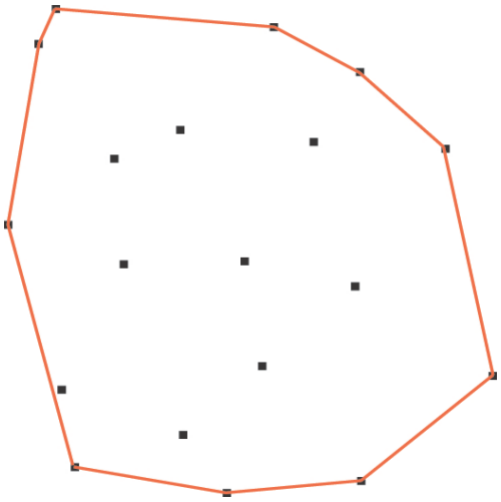


Figure 9: The convex hull of a planar set of points representing the smallest convex set containing them (Fisher 2004).

Next, they need to determine convexity defects, which represent points that are farthest away from the edge of the hull (Bradski and Kaehler 2008, 260) as you can see in Figure 10.

Fratti and Prattichizzo explain the corresponding implementation in OpenCV as following:

> "This set of points is necessary for the function used to identify the fingers: cvConvexityDefects(). The routine takes the contour and the convex hull and it computes the defects of convexity returning for each defect a structure with the start point, corresponding to the tip of a finger, the depth point and the end point, corresponding to the tip of the adjacent finger." (Frati and Prattichizzo 2011)

While their results seem promising, we detected a major disadvantage, which will make solutions based on segmentation via convex hull and convexity defect not suitable for our implementation: They assume that the largest part of the hand needs to be visible for correct results. Also, no occlusions of the hand should occur. And, even more discouraging, they mention that "the worst condition generating occlusions is when the fingers are perpendicular to the Kinect camera plane xy" which may occur approximately in our use case within the pose of an outstretched arm.

As a result, an implementation based on finger detection via convex hull and convexity defect doesn't fulfill our requirements of a stable fingertip detection within different point of views and won't be considered for implementation.

2.3.2 Finger–Earth Mover's Distance

For addressing the main problem of tracking hand and finger positions discussed in Chapter 2.3 Ren, Yuan and Zhang introduced a shape distance metric called Finger–Earth Mover's Distance (FEMD).

In the beginning, they collect hand gesture datasets for each gesture they want to detect. They collect the data in uncontrolled environments, where the subject poses with variations. For correct data retrieval the user's hand must be the frontmost object facing the sensor and he or she needs to wear a black belt around the gesturing hand. Next, they record the relative distance between each contour vertex to the hand's center point and represent it as a time series curve (Figure 11) (Ren, Yuan, and Zhang 2011).

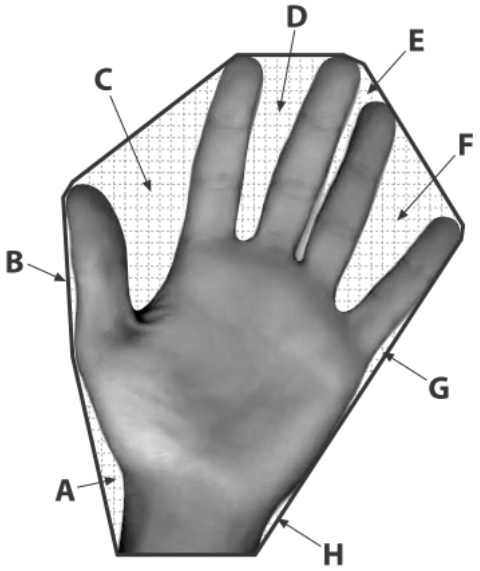


Figure 10: While the dark contour is the convex hull around the hand, the gridded regions (A–H) are convexity defects relative to the convex hull (Bradski and Kaehler 2008, 259).

As for gesture recognition they do the following:

> "With the hand shape and its time–series representation, we apply template matching for robust recognition, i.e., the input hand is recognized as the class with which it has the minimum dissimilarity distance: $c = arg\min_{c} FEMD(H, T_c)$, where H is the input hand; T_c is the template of class c; FEMD(H, T_c) denotes the proposed Finger–Earth Mover's Distance between the input hand and each template." (Ren, Yuan, and Zhang 2011)

The Finger–Earth Mover's Distance is based upon the Earth Mover's Distance, which measures the distance between signatures of histograms (Rubner, Tomasi, and Guibas 2000).

Ren, Yuan and Zhang describe the basics of the Earth Mover's Distance as following:

> "It is named after a physical analogy that is drawn from the process of moving piles of earth spread around one set of locations into another set of holes in the same space. The locations of earth piles and holes denotes the mean of each cluster in the signatures, the size of each earth pile or hole is the weight of cluster, and the ground distance between a pile and a hole is the amount of work needed to move a unit of earth. To use this transportation problem as a distance measure, i.e., a measure of dissimilarity, one seeks the least cost transportation — the movement of earth that requires the least amount of work." (Ren, Yuan, and Zhang 2011)

Based on this idea they altered the features of an Earth Mover's Distance algorithm applied for contour matching to make it work for hand gesture recognition by considering "the input hand as signature with each finger as a cluster". More detail covering the calculation of this procedure can be found in their paper (Ren, Yuan, and Zhang 2011).

Before measuring the FEMD they need to detect the fingers from the hand shape by extracting the peaks of the time series curve. As mentioned before, the gestures will be detected within the minimum dissimilarity distance of the FEMD (Ren, Yuan, and Zhang 2011).

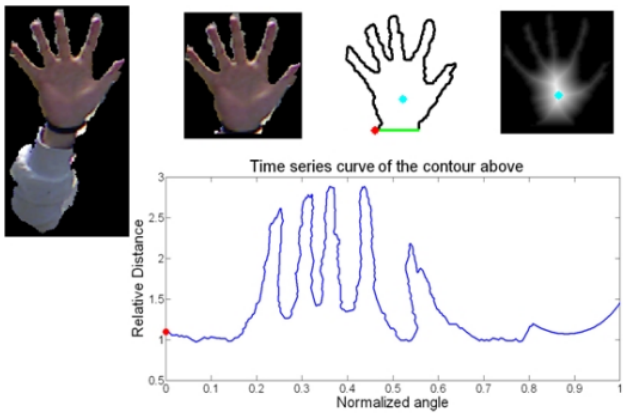


Figure 11: First of all, the rough hand is segmented by depth thresholding. The detection turns out being more accurate by using the black belt. With the help of the initial point (red) and the center point (cyan) a time–series curve represenation is drawn (Ren, Yuan, and Zhang 2011).

According to their test results the gesture detection of the FEMD algorithm of Ren, Yuan and Zhang works very well even within different orientation and scale changes of the hand. As final result their detection works pretty reliable with detection rates over 90 percent (Ren, Yuan, and Zhang 2011).

In relation to our implementation an algorithm based on FEMD will most certainly work very well, but since the recorded training data is not available to the public we would need to record and interpret our own set of gestures for an outstretched arm. Since this task increases the effort for a successful implementation to an unwanted level of complexity we'll continue evaluating other approaches.

2.3.3 Curvature Morphology and Support Vector Machines

Ambrus and Mohamed use the concept of curvature morphology as silhouette filter to detect the positions of fingertips of a hand image (Ambrus and Mohamed 2011, 20).

Their research is based on the conclusions of Leymarie's and Levine's research on this topic. They extract curvatures of planar curves for representation and interpretation of objects in an image (Figure 12) (Ambrus and Mohamed 2011, 20). Curvatures are "a measure of the rate of change in orientation at each point along a curve" (Leymarie and Levine 1988, 1). The main flow of actions is stated as following:

> "In a typical computer vision system, discrete contours of objects are first extracted from an image. Curvature of these discretized contours is then approximated and used to detect important features of the boundary of an object." (Leymarie and Levine 1988, 1)

Ambrus and Mohamed analyze the curvature along the boundary of a hand shape for gaining the finger data. Leymarie and Levine use functions of mathematical morphology to extract dominant shapes within a curve (Leymarie and Levine 1988, 9). With attention to our implementation the extraction of fingertips might work out, but for detecting our gesture it is enough when we concentrate on finding the local maxima and minima of this curve (Munshi 2011) as proven in Chapter 3.2.2.

For guaranteeing a precise fingertip detection the problem of classifying which peaks are fingertips and which are not is still remaining. Ambrus and Mohamed use support vector machines for solving this issue (Ambrus and Mohamed 2011, 21).

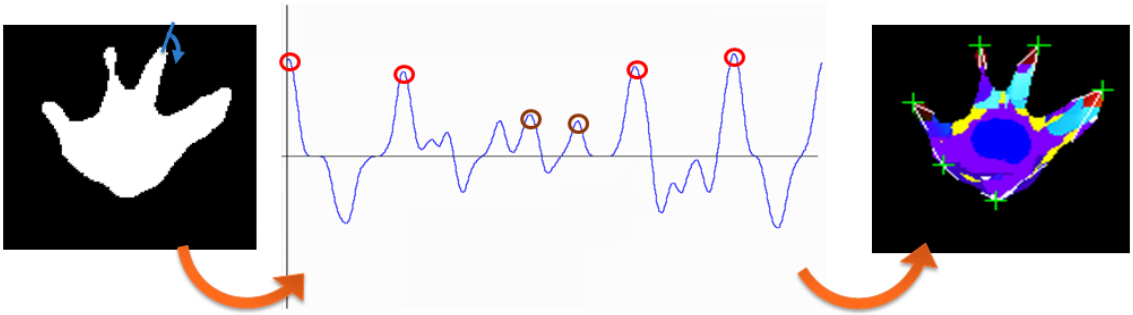


Figure 12: With the help of curvature morphology the peaks and valleys of the shape are analyzed to differentiate fingers (Ambrus and Mohamed 2011, 20).

Cortes and Vapnik define such a support–vector network as "a learning machine for two–group classification problems" (Cortes and Vapnik 1995, 273). In other words, support vector machines are used to classify data that belong to one of two classes, which would be in our use case the classification of the peak height of the curve and whether or not it belongs to the "class" fingertips (Ambrus and Mohamed 2011, 21). The following idea is implemented:

> "It [the support–vector network] maps the input vectors into some high dimensional feature space Z through some non–linear mapping chosen a priori. In this space a linear decision surface is constructed with special properties that ensure high generalization ability of the network." (Cortes and Vapnik 1995, 274)

In order to find such a separating hyperplane, a small amount of training data (which are the support vectors) has to be taken in account for defining this margin (Figure 13) (Cortes and Vapnik 1995, 275). For our implementation the finding of such a separating hyperplane remains a challenge as we need to conduct training data for gathering finger data.

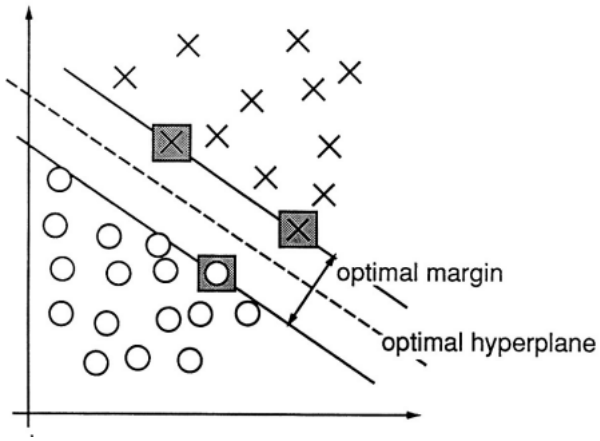


Figure 13: In this example of a seperate problem the support vectors (marked with grey squares) "define the margin of largest seperation between the two classes" (Cortes and Vapnik 1995, 275).

Additionally, we won't necessarily need to implement a support vector machine for correct identification of fingertips of an outstretched arm , since we are only interested in the orientation of the fingers. Also the collection and elaboration of training vectors might also be a too extensive task for the detection of a single gesture.

2.3.4 K–Curvature Estimation

As stated in Chapter 2.3.3 the stable detection of a single fingertip might be enough for a robust implementation of the recognition of an outstretched arm, since we are only interested in the orientation of the fingers.

For their implementation of a virtual 3D blackboard Wu, Shah and Vitoria Lobo were also only interested in an implementation of a stable detection of the user's fingertip. While traversing along the pixel of an arm contour they use an approximation to k–curvature, "whose measure is defined by the angle between two vectors $[P(i;k);P(i)]$ and $[P(i);P(i+k)]$, where k is a constant and $P(i) = (x(i), y(i))$, the list of contour points" (Wu, Shah, and Vitoria Lobo 2000). See Figure 14 for details on finding features of a human hand using k–curvatures.

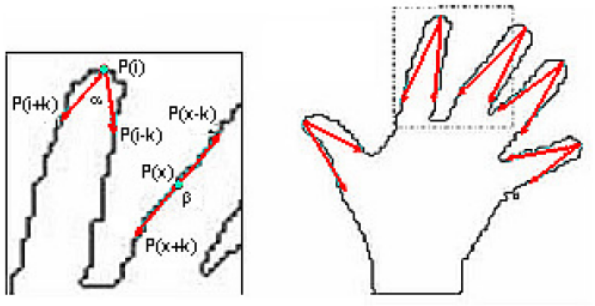


Figure 14: In k–curvature estimation fingertips are detected by measuring the angle between two vectors $[P(i;k);P(i)]$ and $[P(i);P(i+k)]$ (Trigo and Pellegrino 2010).

While they concentrate only on finding one single finger point, they calculate the dot product for each point on the outline:

> "For appropriate values of k, the value of the dot product can be used to find the fingertip. If we take a pixel near the finger and extend two vectors k pixels away along the outline, the vectors will point in similar directions and thus will have a high value of a dot product. The vectors formed at a pixel on the arm will have a large but negative dot product. Thus, we compute the dot product for all outline pixels and find the highest values. Typically, there is a unique highest value, which is the fingertip. When multiple pixels have equally high dot products, we choose the pixel that is closest to the previous finger position." (Wu, Shah, and Vitoria Lobo 2000)

The basic idea behind this approach sounds promising and easy to implement but indicates also two troubles. For gesture recognition the number of detected fingers is detected properly but can produce false positive answers for gestures with similar curves. Since we won't distinguish between a multiple set of constants within our implementation this trouble is irrelevant. Another problem is that the constant k must be defined properly in order to make the recognition work appropriate (Trigo and Pellegrino 2010). Since the scaling of the hand analyzed by our implementation tends to be highly invariant from user to user the constant needs to be defined in comparison to the size of the user's hand to make it work properly.

Summing up, fingertip detection based on the k–curvature algorithm offers us a straightforward to implement solution for our problem without requiring any training data by simply detecting the local maxima of the curve around the hand's shape. The only remaining problem, which needs evaluation throughout our implementation, is the calculation of the value for the constant k. The value should be based and altered upon the size of the user's hand in order to gather reliable data for guaranteeing scale invariance.

3 Implementation

We implemented a program that uses the Kinect sensor for interaction with users. In this setup the scene is recorded and displayed on a plasma or LCD panel in Full HD resolution. The program has been implemented in C++ using Visual Studio 2010.

The display and interaction logic is based upon the Cinder library, because it is a free and open source library that is widely in use for the implementation of interactive setups. Also it natively delivers a wide variety of features, which have been used during the implementation, like for instance a XML parser or the possibility to add GUI elements for altering and storing configuration parameters during execution.

As for image processing we use the OpenCV library, because it "was designed for computational efficiency and with a strong focus on real–time applications" (Bradski and Kaehler 2008, 1). Also there are appropriate functions available allowing native usage of OpenCV functions within the Cinder framework.

In our test environment the data of the Kinect sensor will be accessed through PrimeSense's OpenNI drivers. We do this by usage of the 2RealKinectWrapper developed by the CADET research team. The wrapper allows easy access on depth and RGB images and also user's skeleton data provided by the Kinect hardware. Additionally, it also supports the Kinect for Windows SDK[7] deployed by Microsoft.

Because of these features, possible future versions of our implementation, which has currently only been built and tested on Windows 7 Professional using OpenNI drivers, can be ported for other platforms like Linux or Mac or used for one of the two major Kinect frameworks without deploying major changes in the code base.

3.1 Overview

On startup, only the RGB image of the Kinect sensor will be displayed on a screen. When the user enters the scene and the Kinect driver detects a new user shape a hummingbird appears. The bird itself consists of flying and turning animations stored in a 2D spritesheet. As next step the hummingbird will be scaled according to the current user size by detecting skeleton points of the user's neck and the user's head and calculating the length of a vector between these two positions as seen in Listing 1.

As long as no gesture is detected, the bird flies around the user's shape randomly and switches between the states discussed in Chapter 3.1.2. When we detect the gesture of an outstretched arm, the bird knocks against the user's fingertips and activates a flower, which will grow out of the user's palm (Figure 1). As long as the user maintains his or her gesture the bird will be attracted to the flower. Otherwise it will continue flying around the user's shape. The whole flow of actions is demonstrated in Figure 15.

3.1.1 Scene States

In the beginning, our setup will be executed in an init state, where all the logic for initialization and asset loading is performed. In more detail, the most important tasks operated are the loading of the sprite images for the bird and flower animations and the initialization of the 2RealKinectWrapper for accessing camera data. When any step during initialization fails, our program will switch into an error state, where the cause of failure will be prompted before it stops execution.

When the initialization succeeds, the program switches to an idle state, where the RGB image of the Kinect sensor will be displayed without any further interaction. After Kinect's driver setup

7. http://www.kinectforwindows.org

Listing 1: KinectManager.cpp: calculating the birdheight by detecting the skeleton points of the user's neck and head

```
267 for(unsigned int i = 0; i < kinect->getNumberOfSkeletons(0); ++i)
268 {
269     _2RealKinectWrapper::_2RealPositionsVector3f skeletonPositions;
270     skeletonPositions = kinect->getSkeletonScreenPositions(0, i);
271
272     if( kinect->isJointAvailable
273             (_2RealKinectWrapper::_2RealJointType::JOINT_HEAD) &&
274         kinect->getSkeletonJointConfidence
275             (0, i, _2RealKinectWrapper::_2RealJointType::JOINT_HEAD)
276                 .positionConfidence > 0.0f &&
277         kinect->isJointAvailable
278             (_2RealKinectWrapper::_2RealJointType::JOINT_NECK) &&
279         kinect->getSkeletonJointConfidence
280             (0, i, _2RealKinectWrapper::_2RealJointType::JOINT_NECK)
281                 .positionConfidence > 0.0f)
282     {
283         ci::Vec2f headPosition = ci::Vec2f(
284           skeletonPositions[_2RealKinectWrapper::_2RealJointType::JOINT_HEAD].x,
285           skeletonPositions[_2RealKinectWrapper::_2RealJointType::JOINT_HEAD].y);
286
287         ci::Vec2f neckPosition = ci::Vec2f(
288           skeletonPositions[_2RealKinectWrapper::_2RealJointType::JOINT_NECK].x,
289           skeletonPositions[_2RealKinectWrapper::_2RealJointType::JOINT_NECK].y);
290
291         birdHeight += ((headPosition - neckPosition) * scaleVector).length();
292     }
293 }
294
295 // when there are more users use the average size
296 if(hasUsers())
297     birdHeight /= kinect->getNumberOfUsers(0);
```

was able to detect a new user in the scene, our setup will switch into a playing state where the hummingbird with all the interaction states discussed in chapter 3.1.2 will emerge. Figure 16 contains a class diagram containing all those states.

3.1.2 Interaction States

By default, the hummingbird is flying around the user's shape. This behavior is achieved by flying to a random end point on the screen. When the end position has been arrived, the bird waits for a random amount of time and continues flying towards the next random position. When the user's shape overlaps with the bird's current position (for instance if the user stretches his arm towards the hummingbird) its state will change into a leaving shape state where it seeks the shortest path out of the user's shape. This feature allows an additional form of interaction where the bird can be driven through the scene by the user's movements. The state changes between these important states is shown in Figure 17.

When a new user is detected by the Kinect framework, the hummingbird will interrupt the flying behavior and fly to a position near the newly entered user's head. As for gesture recognition, after detection of an outstretched hand the hummingbird flies towards the user's hand and "activates" the flower. Then it keeps on circling the flower till the user stops stretching his arm. All the possible bird states are displayed in Figure 18.

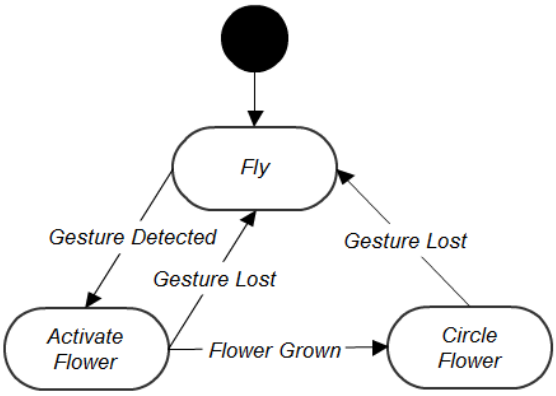


Figure 15: This diagram demostrates the bird's state changes when a gesture of an outstretched arm is detected.

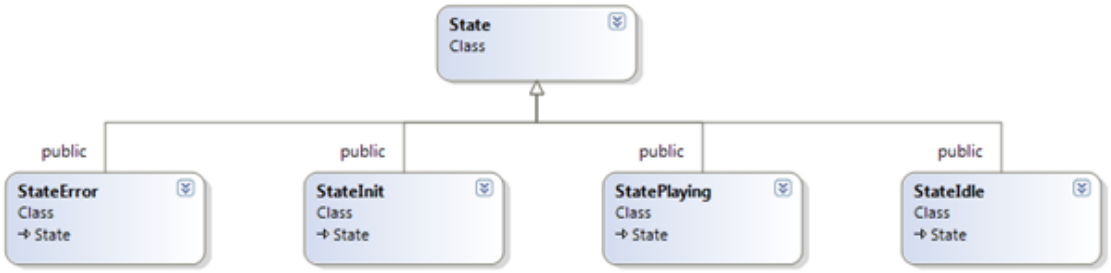


Figure 16: This figure features all classes that have been implemented as scene states.

3.2 Used Methods and Algorithms

After delivering a general overview over the features implemented in our software (as outlined in Chapter 3.1) we'll go into more detail here. Within the context of our research, we'll specifically deal with the implementation of the gesture recognition of an outstretched arm in more detail.

To achieve this, we split the problem into two major tasks. First of all, we roughly estimate the user's pose for detecting whether the user's arm is in a valid position for an outstretched arm or not. When this first check is positive, we extract the position of the user's fingertip to derivate the hand's orientation. When both criteria are met, we interpret the user's gesture as the gesture of an outstretched arm.

3.2.1 Pose Detection

In chapter 2.2 we estimated algorithms capable of general estimation and detection of human shapes and gestures. As we came to the conclusion that skeleton data provided by the Kinect driver software offers us a well researched and reliable fundament for our implementation, we'll extract and interpret skeleton points provided by the 2RealKinectWrapper.

Several joint positions of the user are accessible through the 2RealKinectWrapper. As you can see in Listing 2, a few of them are not supported by one of the two target frameworks, while the detection of fingertip positions is currently unsupported by both frameworks.

Since the hummingbird should not intersect with the shape of the user, we are only interested in finding an outstretched arm where the arm is not aiming towards the camera (Figure 1).

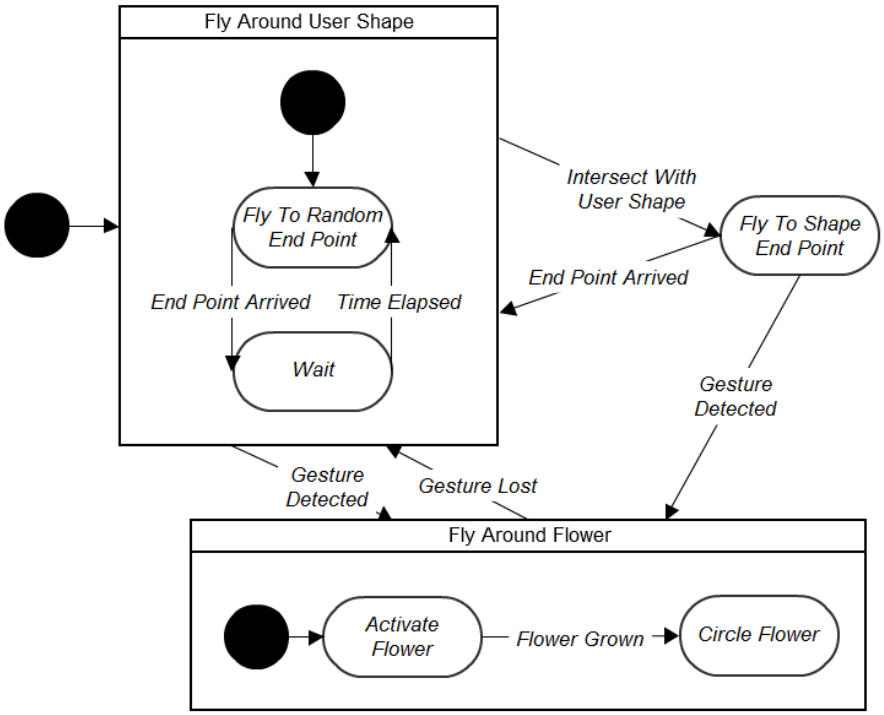


Figure 17: Once the user's shape overlaps with the bird's current position or a gesture is detected it interrupts it's default behavior.

Additionally, the framework of the 2RealKinectWrapper offers us easy access to the joint positions outlined in Listing 2 once the user's skeleton data has been calibrated.

We came up with good results by calculating the angle between arm and elbow position and checking whether the value lies between a threshold defining how "far" the arm can be stretched. To achieve this, we calculate the arc tangent of the arm vector by using the atan2-function provided by C++[8], which gives us the orientation of the vector considering the current quadrant. This works for both hands, but we need to consider that we need to subtract 180 degrees from the left hand vector since it faces in the opposite direction (Listing 3).

3.2.2 Fingertip Detection

In Chapter 2.3.4 we came to the conclusion, that the usage of an algorithm based on k–curvatures for fingertip detection might be a reliable solution for our implementation. As for curvature calculation, our implementation is based on Wu's, Shah's and Vitoria Lobo's research on finger tracking using an approximation of k–curvature by estimating the angle between two vectors using the dot product (Wu, Shah, and Vitoria Lobo 2000).

The main idea behind our algorithm is to find the fingertip positions on a binary image displaying the user's arm. Additionally, we need to measure the relative position of the hand and elbow position within this image to outline the hand image. In practical usage, the binary image is equivalent to the user image, which can easily be accessed through the 2RealKinectWrapper and is computed on interpretation of the camera's depth image. The hand and elbow positions are skeletal positions we already discussed in Chapter 3.2.1.

8. `http://www.cplusplus.com/reference/clibrary/cmath/atan2/`

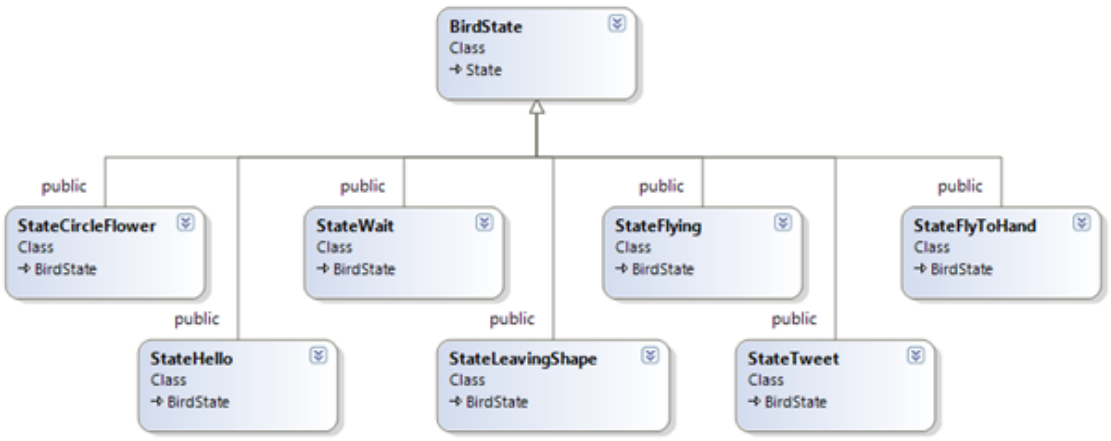


Figure 18: This figure features all classes that have been implemented as bird states.

Test Application For testing and estimating the detection within our requirements, we set up a small test application, which reads a binary image of an arbitrary size and calculates the fingertip positions. One of the main advances of an independent test application is, that we can test and adjust our algorithm and its values fast and easily within an environment only focusing on fingertip detection. Additionally, our implementation can easily be integrated into our interactive setup using the Kinect. All we need to do is to replace the binary image with the user image provided by the Kinect software.

First of all, we calculate a parameter called hand size by determining the length of the vector pointing from arm to elbow. We use this size for calculating a rough estimation of the region around the user's hand (Figure 19). Next, we use functions of the OpenCV library to find all the contour points of the user's hand. To improve the accuracy of this step, we ignore points that are most likely not part of the hand's contour, such as points around the border or when the amount of certain contour points is beyond a certain threshold. This eliminates erroneous behavior which may for instance occur, when parts of other user shapes are part of the region of the target hand. The whole process of contour retrieval and filtering is outlined as commented code in Listing 4.

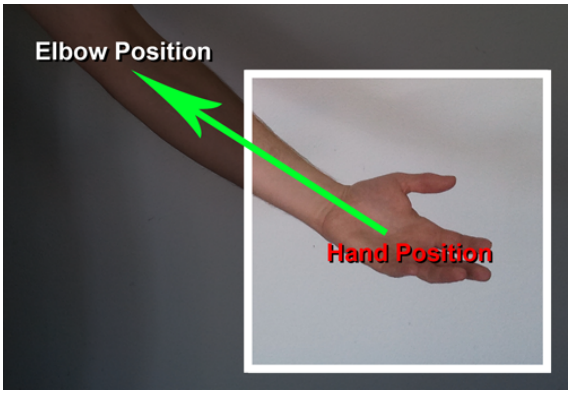


Figure 19: We use the length of the vector pointing from hand to elbowposition (green arrow) for estimating the hand's boundary. Every side of the white square surrounding the user's hand has the length of this vector.

As discussed in Chapter 2.3.4, we need to define the constant k in comparison to the size of the user's hand. The calculation of a value for this constant turned out to be a challenging task, as our solution needs to be scale invariant and also deliver robust results within different point of

Listing 2: _2RealTypes.h: an overview over all joint types supported by the 2RealKinectWrapper

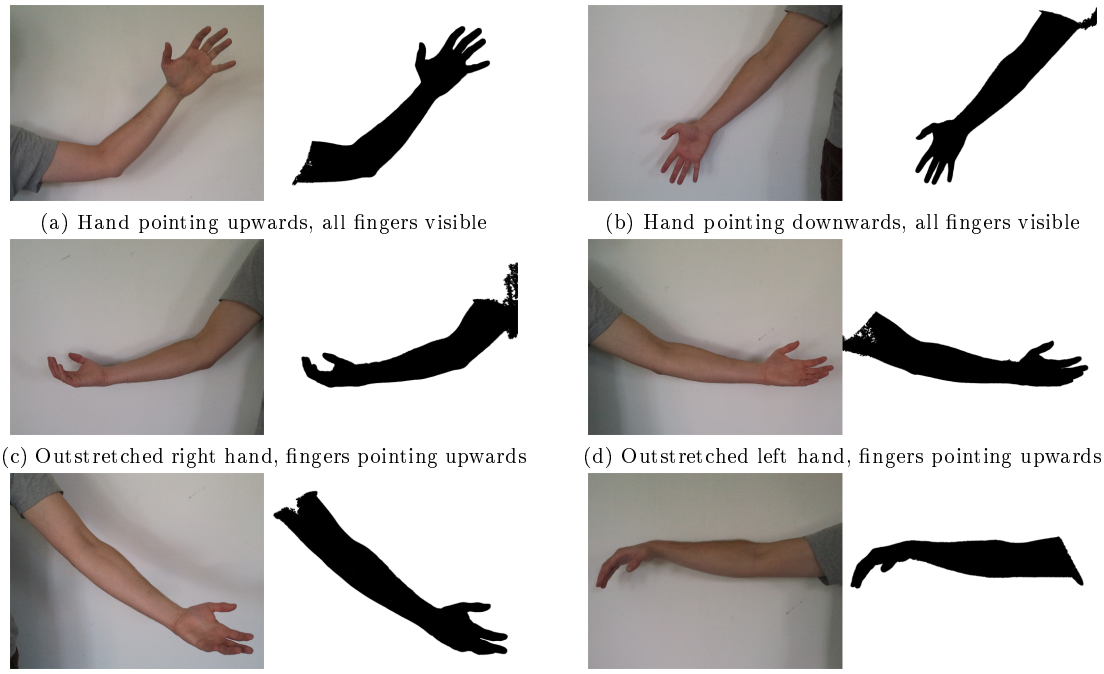
```
56 enum _2RealJointType
57 {
58     JOINT_HEAD              = 0,
59     JOINT_NECK              = 1,
60     JOINT_TORSO             = 2,
61     JOINT_WAIST             = 3, // currently not available for OPENNI
62
63     JOINT_LEFT_COLLAR       = 4, // currently not available for OPENNI; WSDK
64     JOINT_LEFT_SHOULDER     = 5,
65     JOINT_LEFT_ELBOW        = 6,
66     JOINT_LEFT_WRIST        = 7, // currently not available OPENNI
67     JOINT_LEFT_HAND         = 8,
68     JOINT_LEFT_FINGERTIP    = 9, // currently not available for OPENNI; WSDK
69
70     JOINT_RIGHT_COLLAR      = 10, // currently not available for OPENNI; WSDK
71     JOINT_RIGHT_SHOULDER    = 11,
72     JOINT_RIGHT_ELBOW       = 12,
73     JOINT_RIGHT_WRIST       = 13, // currently not available for OPENNI
74     JOINT_RIGHT_HAND        = 14,
75     JOINT_RIGHT_FINGERTIP   = 15, // currently not available for OPENNI; WSDK
76
77     JOINT_LEFT_HIP          = 16,
78     JOINT_LEFT_KNEE         = 17,
79     JOINT_LEFT_ANKLE        = 18, // currently not available for OPENNI
80     JOINT_LEFT_FOOT         = 19,
81
82     JOINT_RIGHT_HIP         = 20,
83     JOINT_RIGHT_KNEE        = 21,
84     JOINT_RIGHT_ANKLE       = 22, // currently not available for OPENNI
85     JOINT_RIGHT_FOOT        = 23
86 };
```

views. In general, the value for k needs to be high for a big hand size and low for a small hand size. After evaluating several test images with different scale and orientation we came to the conclusion, that the k-value must have a limit value it should not go below (too small k-values won't produce reliable results, as they are too susceptible to jittering) and also be in direct connection to the amount of filtered contour points. The final solution we came up with is described in Listing 5.

When we've calculated the k–value, we can finally start iterating through the contour points and draw a vector between $[P(i-k;k);P(i)]$ and $[P(i);P(i+k)]$ as described in chapter 2.3.4. Like Wu, Shah and Vitoria Lobo we use the dot product to draw conclusions to the angle between the two vectors (Wu, Shah, and Vitoria Lobo 2000). The points with the highest negative values are those with the highest peaks or highest valleys and therefore fingertips or positions between them.

Also we experimented around with a value defining a minimum dot value which should indicate the value from which fingertips should be detected. After testing around with various images we left it to zero, but it can be altered for possible future releases or adjustments. Whenever a negative dot product switches back to a positive result, we know that we detected a finger position within the last dot products. The position where the calculation leads to the biggest value will be detected as a position of interest. Listing 6 illustrates the whole course of actions.

Generating and Interpreting Test Data For testing and configuring parameters of our algorithm we took pictures of certain hand gestures behind a white wall and transformed them into black and white displaying only the hand contour (Figure 20).



(a) Hand pointing upwards, all fingers visible

(b) Hand pointing downwards, all fingers visible

(c) Outstretched right hand, fingers pointing upwards

(d) Outstretched left hand, fingers pointing upwards

(e) Outstretched hand pointing downwards

(f) Outstretched arm, fingers pointing downwards

Figure 20: We took pictures of six different hand gestures, which we transformed into black and white images for contour evaluation and fingertip detection.

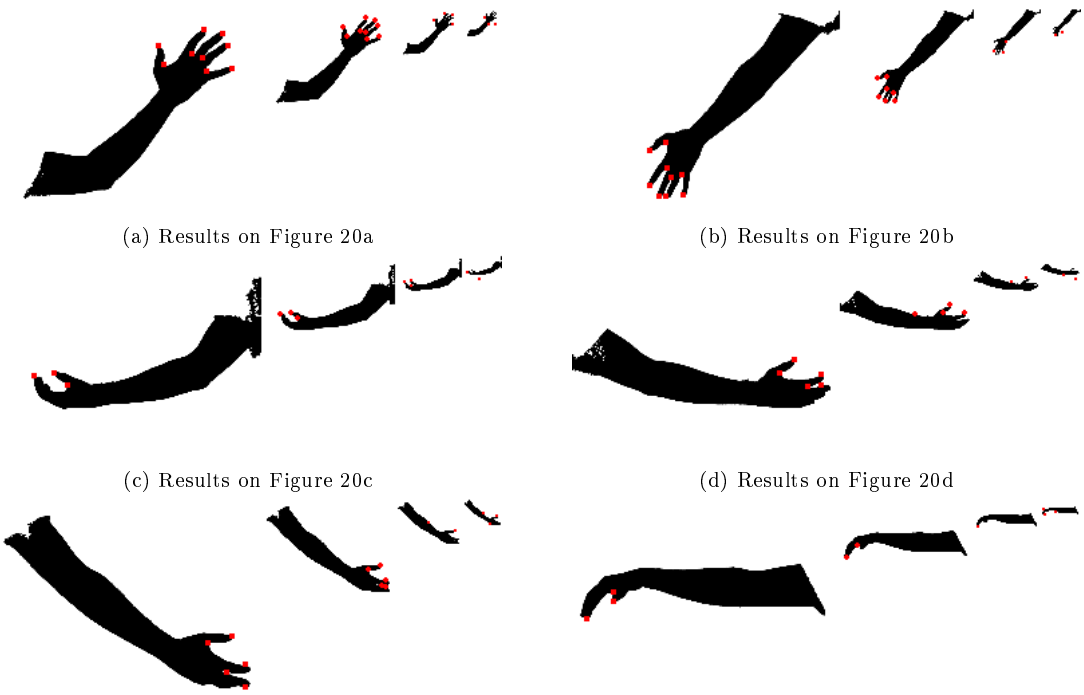
For testing the accuracy within different hand sizes we scaled these images to different sizes, always with attention to the resolution in our final setup. Since the depth image of the Kinect has a minimum size of 640 x 480 pixels, we decided to scale the hand images to the sizes 200 x 154, 100 x 77, 50 x 38 and 30 x 23 pixels to simulate realistic proportions.

Within the first two sizes we expect accurate results, but we also expect our algorithm to fail on too small images since the hand details can't be extracted precisely on too small image sizes. For every picture we measured a rough pixel position for the palm and the elbow position. With this information, our algorithm should be able to detect fingertip positions within these different contour images.

We captured six different images handling typical situations that may occur frequently within our final implementation. Figures 20a and 20b are not typical gestures of an outstretched arm and will never be evaluated in our final implementation as the angle between palm and elbow exceeds the threshold as discussed in chapter 3.2.1. Nevertheless, they are valuable for benchmarking the robustness of our algorithm since they are the only ones where all fingers are clearly visible in the contour image.

One of the most difficult tasks within testing and evaluating was the calculation of a k–value, which is able to deliver good results within all these different poses. While experimenting, we came up with estimations that worked excellent and were able to fetch out all important finger details even on very low scale images. But they also tend to produce wrong results in other poses. Our final estimation of the k–value is fully explained in Listing 5. While it is not able to maintain all detail at low scale it is still able to detect the most important fingertip orientation, which is crucial for our implementation. The results produced by our implementation are summarized in Table 1, Table 2 and Figure 21.

When comparing the results with the corresponding images in Figure 21, you may recognize that our final results also include parameters for the valleys between the fingers. Those parameters will be ignored in our evaluation because of two reasons: Firstly, these coordinates can easily be



(a) Results on Figure 20a

(b) Results on Figure 20b

(c) Results on Figure 20c

(d) Results on Figure 20d

(e) Results on Figure 20e

(f) Results on Figure 20f

Figure 21: Our algorithm works accurate on the first two images within the higher scale, but slightly loses detail on the last two images within the lower scale. The positions that have been detected as fingertips or valleys between fingers are marked as red dots.

filtered because of their nature (valleys are always between fingertip coordinates) and secondly, we don't need to filter these entries since we are only interested in the fingertip direction as stated in Chapter 2.3.3.

As expected, the images with the smallest size tend to produce wrong results. Even the images with a pixel size of 50 x 38 pixels sometimes include erroneous values. But when comparing the image size of the hand to the pixel size of the Kinect's depth image, we come to the assumption that in most use cases the hand size will fit to the arms displayed in our images with the size 100 x 77 pixels.

In comparison to our analysis in Table 1 and Table 2 we come to the theoretical conclusion, that our implementation of fingertip detection will work accurate in our practical setup. For justifying our assumptions we adapted the algorithm explained in this chapter for our final software setup. The corresponding results will be explained in the next chapter.

3.3 Final Results

The gesture of an outstretched arm should only be recognized when the user's hand is outstretched and the fingers are pointing upwards.

In our final setup, pose detection, as described in Chapter 3.2.1, works pretty well as long as skeleton data is available. When the angle between the user's elbow and hand is between a predefined threshold, we further examine the fingertip positions to draw conclusions from the user's arm orientation.

To accomplish this step, we were able to adopt our algorithm quite easily into our final setup by simply reading the hand images not from files but from the Kinect's user image. Instead of the

	Figure 21a		Figure 21b		Figure 21c	
Pixel Size	Correct	Wrong	Correct	Wrong	Correct	Wrong
200 x 154	5	0	5	0	2	0
100 x 77	5	0	5	0	2	0
50 x 38	4	0	2	1	2	0
30 x 23	2	1	1	1	1	1

Table 1: Results of the first three test images, where the column "Correct" indicates the amount of detected fingertip positions and the column "Wrong" indicates the amount of detected fingertip positions which are invalid.

	Figure 21d		Figure 21e		Figure 21f	
Pixel Size	Correct	Wrong	Correct	Wrong	Correct	Wrong
200 x 154	3	0	3	0	2	0
100 x 77	2	1	3	0	1	0
50 x 38	1	1	1	1	1	0
30 x 23	0	2	1	2	1	1

Table 2: Results of the last three test images, where the column "Correct" indicates the amount of detected fingertip positions and the column "Wrong" indicates the amount of detected fingertip positions which are invalid.

given positions for the user's hand and elbow we rely on the coordinates of the body joint positions of the Kinect's skeletal data.

Unfortunately, within this context the first results turned out to be discouraging, as the coordinate of the hand position tends to jump around the hand position as you can see in Figure 22. Our initial idea was to detect whether the fingertip positions are above or beneath the hand position to check if the fingers of the outstretched arm are pointing up– or downwards. But since the Kinect sensor tends to produce imprecise results once the hand is not fully visible (as it is in the side view of an outstretched arm), we need to implement a workaround to overcome this jitter.

Figure 22: In reality, the hand position delivered by the Kinect's skeleton data tends to jump around the expected hand position. Here, the calculated hand position (marked as red dot) lies several pixels beyond the real palm position.

As in most cases these inaccuracies affect only one of several fingertip positions of a hand, we decided to calculate the average finger position of all theses coordinates and compare its coordinate

with the absolute hand position as seen in Listing 7. This procedure turned out to be quite effective with attention to filtering single fingertip positions that are lying marginal beneath the hand's position assumed by the Kinect software.

The second problem we came up with was, that our approximation to the k–value, evaluated in Chapter 3.2.2, tends to produce erroneous results, because the distortions within the Kinect's depth image turned out being higher than expected. So in most use cases our algorithm evaluated fingertip position's around the user's sleeve, which of course is a highly unwanted behavior we need to avoid.

We were able to solve this issue by increasing the minimum value for estimating the k–value to make it more robust against jitter. This turned out being an acceptable solution for our use case as you can see in Figure 23.

(a) When increasing the limit of the k–value the fingertip can be precisely determined.

(b) Disadvantage of a high k–value: In some use cases not all fingertip positions will be identified.

Figure 23: When examining the final results we see that in some use cases not all fingertip positions (marked with green dots) will be identified, but we are always able to detect the hand's orientation.

The only drawback that remains, is the fact that because of the rather high k–value, we're often not capable to extract all fingertip positions which are visually visible (Figure 23b). This limitation is acceptable, as we sacrifice this lack of accuracy to a high robustness. Also, we only need to evaluate the orientation of the user's hand to detect whether the pose is correct or not.

Summing up, in practice k–curvature estimation comes up with some troubles adjusting the right values for robust detection within different scaling, but we were able to overcome these issues by carefully adjusting the k–value and testing it against various poses.

4 Conclusion and Answer to the Research Question

We measured a variety of well researched algorithms based upon gesture recognition in general and hand and fingertip recognition more specifically. We relied on body part recognition based on depth imaging, because it doesn't require additional training data like HMMs or CONDENSATION. Also — unlike DTW — it is a viewpoint invariant solution.

For hand gesture and fingertip recognition we reviewed and evaluated various algorithms. Those based on convex hull and convexity defect aren't considered for our final implementation because they can't guarantee stable detection within different point of views. The Finger–Earth Mover's Distance algorithm will most likely deliver the most accurate results, but within the restrictions of our use case the complexity of gathering own training data is too high for detecting only a single gesture. Curvature morphology with support vector machines for filtering correct fingertip positions roughly includes the main idea behind our implementation based on k–curvature esti-

mation. Nevertheless, we discovered that we don't need to filter between the peaks and valleys of the detected curve points as we only need information about the orientation of the user's hand.

Our final implementation is based upon detecting the angle between the user's hand and elbow by extracting the skeletal joint positions. For gathering the correct hand orientation we implemented an algorithm based on k–curvatures, which worked well for our use case. One of the most challenging tasks consisted in finding right values for the constant k to support view point invariance.

We set up a test implementation analyzing different gestures, which are stored in black and white images to test the robustness of our algorithm. Nevertheless, we had to implement several adjustments for our final implementation to overcome the jitter and inaccurate hand positions delivered by the framework.

Summing up, after some tweaking k–curvatures turned out to be our solution of choice for a stable detection of an outstretched arm. It doesn't add unwanted complexity by being independent of acquiring training data and it works within different scales and point of views. This mixture of robustness and straightforwardness in implementation overcomes the lack of accuracy as it isn't always able to detect all visible fingertip positions. Within our requirements, k–curvatures are able to give us information about the hand's orientation, we can derive from the position of certain fingertips, crucial for our hand gesture detection algorithm.

Listing 3: KinectManager.cpp: measuring the angle between the user's left hand and elbow

```
209 jointConfidenceHand = kinect->getSkeletonJointConfidence(0, i,
210     _2RealKinectWrapper::_2RealJointType::JOINT_LEFT_HAND);
211 jointConfidenceElbow = kinect->getSkeletonJointConfidence(0, i,
212     _2RealKinectWrapper::_2RealJointType::JOINT_LEFT_ELBOW);
213 if( kinect->isJointAvailable(
214     _2RealKinectWrapper::_2RealJointType::JOINT_LEFT_HAND) &&
215         jointConfidenceHand.positionConfidence > 0.0f &&
216     kinect->isJointAvailable(
217         _2RealKinectWrapper::_2RealJointType::JOINT_LEFT_ELBOW) &&
218         jointConfidenceElbow.positionConfidence > 0.0f)
219 {
220     armVector = ci::Vec2f(
221       skeletonPositions[_2RealKinectWrapper::_2RealJointType::JOINT_LEFT_HAND].x -
222       skeletonPositions[_2RealKinectWrapper::_2RealJointType::JOINT_LEFT_ELBOW].x,
223       skeletonPositions[_2RealKinectWrapper::_2RealJointType::JOINT_LEFT_HAND].y -
224       skeletonPositions[_2RealKinectWrapper::_2RealJointType::JOINT_LEFT_ELBOW].y);
225
226     float angleBetweenHandAndElbow = atan2(armVector.y, armVector.x)
227         * 180 / 3.14159265f - 180;
228
229     if(angleBetweenHandAndElbow < 0)
230         angleBetweenHandAndElbow += 360;
231     else if(angleBetweenHandAndElbow > 360)
232         angleBetweenHandAndElbow -= 360;
233
234     if( angleBetweenHandAndElbow < allowedAngle ||
235         angleBetweenHandAndElbow > 360 - allowedAngle)
236     {
237         temporaryFlowerPosition = ci::Vec2f(
238           skeletonPositions[_2RealKinectWrapper::_2RealJointType::JOINT_LEFT_HAND].x,
239           skeletonPositions[_2RealKinectWrapper::_2RealJointType::JOINT_LEFT_HAND].y)
240           * scaleVector;
241         temporaryFlowerID = i * 2 + 1;
242
243         if(temporaryFlowerID == currentFlowerID)
244         {
245             flowerPosition = temporaryFlowerPosition;
246             flowerID = temporaryFlowerID;
247             return true;
248         }
249     }
250 }
```

Listing 4: KinectManager.cpp: retrieving and filtering the user's hand contour

```
361 // calculating the estimated size of the hand
362 // retrieving the length of predefined arm and elbow positions
363 float handSize = (handPosition - elbowPosition).length();
364 float handROIPosX = ci::math<float>::max(0, handPosition.x - handSize / 2);
365 float handROIPosY = ci::math<float>::max(0, handPosition.y - handSize / 2);
366
367 cv::Rect handROI(handROIPosX, handROIPosY,
368     ci::math<float>::min(userSurface.getWidth() - handROIPosX, handSize),
369     ci::math<float>::min(userSurface.getHeight() - handROIPosY, handSize));
370
371 cv::Mat userMat = ci::toOcvRef(userSurface);
372 cv::Mat handMat = cv::Mat(userMat, handROI);
373 std::vector< std::vector<cv::Point> > contours;
374
375 // "security" functions to ensure that the image is binary...
376 cv::cvtColor(handMat, handMat, CV_RGB2GRAY);
377 cv::threshold(handMat, handMat, 1, 255, CV_THRESH_BINARY);
378
379 cv::findContours(handMat, contours, CV_RETR_LIST, CV_CHAIN_APPROX_NONE);
380
381 // define a set of minimum contour points that should be taken in account
382 // this resolves possible issues with contours that are in the hand's region
383 // of interest but not part of the hand itself
384 int minContourPoints = handSize / 2;
385 ci::Vec2f fingerTip;
386 std::vector<ci::Vec2f> filteredContourPoints;
387
388 for(int i = 0; i < contours.size(); ++i)
389 {
390     // ignore "small" contours beyond our threshold
391     if(contours[i].size() > minContourPoints)
392         for(int j = 0; j < contours[i].size(); ++j)
393         {
394             ci::Vec2f currentContourPoint =
395                 ci::Vec2f(contours[i][j].x, contours[i][j].y);
396
397             // ignore bounding contour points
398             if( currentContourPoint.x > handSize / 10 &&
399                 currentContourPoint.y > handSize / 10 &&
400                 currentContourPoint.x < handROI.width - handSize / 10 &&
401                 currentContourPoint.y < handROI.height - handSize / 10 )
402                 filteredContourPoints.push_back(currentContourPoint);
403         }
404 }
```

Listing 5: KinectManager.cpp: our final solution for estimating the k–value

```
404 int k = ci::math<int>::max(5, filteredContourPoints.size() / 30);
```

Listing 6: KinectManager.cpp: interpreting contour data by calculating the dot product

```
406 // at current development stage the minimum dot value is zero
407 // when it should have other values, the values should always
408 // be in connection with k as k indicates the lenght of the vectors
409 // from which the dot product is calculated
410 float minDotValue = 0 * k;
411 float minDotProduct = minDotValue;
412 ci::Vec2f currentFingerpoint;
413
414 for(int i = 0; i < filteredContourPoints.size(); ++i)
415 {
416     if(i > k && i + k < filteredContourPoints.size())
417     {
418         ci::Vec2f leftVector =
419             filteredContourPoints[i] - filteredContourPoints[i - k];
420         ci::Vec2f rightVector =
421             filteredContourPoints[i + k] - filteredContourPoints[i];
422         float currentDot = rightVector.dot(leftVector);
423
424         if(currentDot < minDotProduct)
425         {
426             minDotProduct = currentDot;
427             // add the roi offset to the relative coordinate
428             // to get absolute values
429             currentFingerpoint =
430                 ci::Vec2f(handROI.x, handROI.y) + filteredContourPoints[i];
431         }
432         // when "leaving" current fingertip reset dot product
433         else if(currentDot > 0)
434         {
435             // when there has a mindotproduct been set put it into the list
436             if(minDotProduct < minDotValue)
437                 fingerPositions.push_back(currentFingerpoint);
438
439             minDotProduct = minDotValue;
440         }
441     }
442 }
```

Listing 7: KinectManager.cpp: calculating the average finger position by multiplying through all retrieved finger positions

```
531 ci::Vec2f averageFingerPosition = ci::Vec2f::zero();
532 for(int i = 0; i < fingerPositions.size(); ++i)
533     averageFingerPosition += fingerPositions[i] / fingerPositions.size();
534
535 return averageFingerPosition.y <= handPosition.y;
```

List of Figures

1 A hummingbird flies around a flower growing out of the user's hand 1
2 Example images displaying differences between arm orientations 2
3 The architecture of the Kinect 4
4 Hidden states of a Markov model 6
5 The CONDENSATION algorithm used for hand tracking 7
6 Motion capture of human actions and definition of body parts 8
7 Randomized decision forests 8
8 Challenging cases for hand gesture recognition 9
9 Example for the convex hull of a set of points 10
10 Convex hull and convexity defect along a user's hand 11
11 Time series curve represenation within the FEMD 12
12 Curvature morphology on an user's hand 13
13 Example for support vectors 13
14 The idea behind k–curvatures 14
15 Course of actions once a gesture is detected 17
16 Class diagram of all possible scene states 17
17 State changes within the most important interaction states 18
18 Class diagram of all possible bird states 19
19 Estimation of the hand region 19
20 Hand gestures as test data 21
21 Results of our test data 22
22 Inaccuracte hand position 23
23 Final results of our algorithm detecting fingertip positions 24

Listings

1 KinectManager.cpp: calculating the birdheight by detecting the skeleton points of the user's neck and head 16
2 _2RealTypes.h: an overview over all joint types supported by the 2RealKinectWrapper 20
3 KinectManager.cpp: measuring the angle between the user's left hand and elbow . 26
4 KinectManager.cpp: retrieving and filtering the user's hand contour 27
5 KinectManager.cpp: our final solution for estimating the k–value 27
6 KinectManager.cpp: interpreting contour data by calculating the dot product . . . 28
7 KinectManager.cpp: calculating the average finger position by multiplying through all retrieved finger positions 28

List of Tables

1 Results of the first three test images 23
2 Results of the last three test images 23

References

Ambrus, Tony, and Shammi Mohamed. 2011. Kinect Hands: Finger Tracking and Voxel UI. In. Gamefest 2011.

Black, Michael, and Allan Jepson. 1998. A Probabilistic Framework for Matching Temporal Trajectories: CONDENSATION–Based Recognition of Gestures and Expressions. Ed. Hans Burkhardt and Bernd Neumann. *Computer Vision — ECCV'98,* Lecture Notes in Computer Science, 1406:909–924.

Bobick, Aaron F. 1997. Movement, Activity, and Action: The Role of Knowledge in the Perception of Motion. *Royal Society Workshop on Knowledge-based Vision in Man and Machine* 352:1257–1265.

Bradski, Gary, and Adrian Kaehler. 2008. *Learning OpenCV: Computer Vision with the OpenCV Library.* Cambridge, MA: O'Reilly. ISBN: 0596516134.

Buss, Samuel R. 2003. *3D Computer Graphics: A Mathematical Introduction with OpenGL.* New York, NY, USA: Cambridge University Press. ISBN: 0521821037.

Cortes, Corinna, and Vladimir Vapnik. 1995. Support–Vector Networks. *Machine Learning* 20 (3): 273–297. ISSN: 0885-6125.

Cristina Manresa, Ramon Mas, Javier Varona, and Francisco J. Perales. 2005. Hand Tracking and Gesture Recognition for Human-Computer Interaction. *Electronic Letters on Computer Vision and Image Analysis:*96–104.

Derpanis, Konstantinos. 2004. A Review of Vision–Based Hand Gestures.

Doliotis, Paul, Alexandra Stefan, Christopher Mcmurrough, David Eckhard, and Vassilis Athitsos. 2011. Comparing Gesture Recognition Accuracy Using Color and Depth Information. *The Hand.*

Fisher, John. 2004. Visualizing the Connection Among Convex Hull, Voronoi Diagram and Delaunay Triangulation.

Frati, V., and D. Prattichizzo. 2011. Using Kinect for Hand Tracking and Rendering in Wearable Haptics. *IEEE World Haptics Conference 2011* (Istanbul, Turkey) (June): 317–321.

Fukasawa, Tetsuo, Kentaro Fukuchi, and Hideki Koike. 2006. A Vision–Based Non–contact Interactive Advertisement with a Display Wall. *ICEC'06:*394–397.

Isard, Michael, and Andrew Blake. 1996. Contour Tracking By Stochastic Propagation of Conditional Density. Ed. Bernard Buxton and Roberto Cipolla. *Computer Vision - ECCV '96* 1064:343–356.

Keogh, Eamonn, and Chotirat Ann Ratanamahatana. 2005. Exact Indexing of Dynamic Time Warping. *Knowledge and Information Systems* 7 (3): 358–386. ISSN: 0219-1377.

Kim, DoHyung, Jaeyeon Lee, Ho-Sub Yoon, Jaehong Kim, and Joochan Sohn. 2010. Vision–based arm gesture recognition for a long–range human–robot interaction. *The Journal of Supercomputing:*1–17. ISSN: 0920-8542.

Leymarie, F., and M. D. Levine. 1988. Curvature Morphology.

Microsoft Corporation. 2010. Kinect Fact Sheet. June. `http://www.microsoft.com/en-us/news/presskits/xbox/docs/KinectFS.docx`.

Munshi, Juned. 2011. Finger tips kinect – make it easy. [Online; accessed on 03.05.2012]. May. `http://junedmunshi.com/blog/finger-tips-kinect-make-it-easyyy-part1/`.

Rabiner, L., and B. Juang. 1986. An Introduction to Hidden Markov Models. *ASSP Magazine, IEEE* 3, no. 1 (Jan.): 4–16.

Rabiner, Lawrence R. 1989. A tutorial on hidden Markov models and selected applications in speech recognition. *Proceedings of the IEEE:*257–286.

Raheja, Jagdish L., Ankit Chaudhary, and Kunal Singal. 2011. Tracking of Fingertips and Centers of Palm Using KINECT. *Computational Intelligence, Modelling and Simulation, International Conference on* (Los Alamitos, CA, USA) 0:248–252.

Ren, Zhou, Junsong Yuan, and Zhengyou Zhang. 2011. Robust Hand Gesture Recognition Based on Finger–Earth Mover's Distance with a Commodity Depth Camera. *Proceedings of the 19th ACM international conference on Multimedia,* MM '11:1093–1096.

Ren, Zhou, Jingjing Meng, Junsong Yuan, and Zhengyou Zhang. 2011. Robust Hand Gesture Recognition with Kinect Sensor. *Proceedings of the 19th ACM international conference on Multimedia,* MM '11:759–760.

Rubner, Yossi, Carlo Tomasi, and Leonidas J. Guibas. 2000. The Earth Mover's Distance as a Metric for Image Retrieval. *International Journal of Computer Vision* 40:2000.

Sharp, Toby. 2008. Implementing Decision Trees and Forests on a GPU. Ed. David Forsyth, Philip Torr, and Andrew Zisserman. *Computer Vision – ECCV 2008* (Berlin, Heidelberg), Lecture Notes in Computer Science, 5305:595–608. ISSN: 0302-9743.

Shotton, Jamie, Andrew Fitzgibbon, Mat Cook, Toby Sharp, Mark Finocchio, Richard Moore, Alex Kipman, and Andrew Blake. 2011. Real–Time Human Pose Recognition in Parts from Single Depth Images. *Computer Vision and Pattern Recognition* (June). doi:10.1109/CVPR.2011.5995316, http://research.microsoft.com/apps/pubs/default.aspx?id=145347.

Starner, Thad, Alex Pentland, and Joshua Weaver. 1998. Real–Time American Sign Language Recognition Using Desk and Wearable Computer Based Video. *IEEE Trans. Pattern Anal. Mach. Intell.* (Washington, DC, USA) 20, no. 12 (Dec.): 1371–1375. ISSN: 0162-8828.

Trigo, Thiago R., and Sergio Roberto M. Pellegrino. 2010. An Analysis of Features for Hand–Gesture Classification. *17th International Conference on Systems, Signals and Image Processing, IWSSIP 2010:*412–415.

Weinland, Daniel, Remi Ronfard, and Edmond Boyer. 2011. A Survey of Vision–Based Methods for Action Representation, Segmentation and Recognition. *Computer Vision and Image Understanding* (New York, NY, USA) 115 (2): 224–241. ISSN: 1077-3142.

Wu, Andrew, Mubarak Shah, and N. da Vitoria Lobo. 2000. A Virtual 3D Blackboard: 3D Finger Tracking Using a Single Camera. *Proceedings of the Fourth IEEE International Conference on Automatic Face and Gesture Recognition 2000* (Washington, DC, USA), FG '00:536–544.