Human-Machine Interaction Through Gesture Recognition

by

Dmitrii Cucleschin

Bachelor Thesis in Computer Science

Prof. Andreas Birk

Name and title of the supervisor

Date of Submission: May 10, 2015

Jacobs University — School of Engineering and Science

With my signature, I certify that this thesis has been written by me using only the indicates resources and materials. Where I have presented data and results, the data and results are complete, genuine, and have been obtained by me unless otherwise acknowledged; where my results derive from computer programs, these computer programs have been written by me unless otherwise acknowledged. I further confirm that this thesis has not been submitted, either in part or as a whole, for any other academic degree at this or another institution.

Signature   Place, Date

Abstract.

Nowadays, it's hard to imagine modern human's life without all the devices we use. Mobile phones, laptops, cameras and many other things have become our daily companions that we use to communicate, work and play. Despite there being so many different kinds of devices, there are only few traditional ways to control them: via physical buttons (e.g. mouse and keyboard), touch screen input, or, since recently, voice recognition. However, what should one do if the device can't support any of the above?

In this research project we will focus on implementing a gesture recognition system for an underwater diving assisting robot. Equipped with such software, robot can be controlled hands-free, while it monitors the diver's safety and performs its assigned tasks. Such way of interaction is so natural for people with diving experience, that it requires almost no special training and can assist in communicating actions quicker. We are going to implement a prototype of such application, using Microsoft Kinect depth sensor. Step-by-step approach to recognizing gestures will be presented and the results will be evaluated according to criteria defined in the thesis proposal.

Contents

1 Introduction.

1.1 Motivation.

Communication underwater is much more complicated than on the surface. Divers often need to learn the entire diver's sign language to understand each other. Often to communicate more complex messages, they need to carry waterproof boards and markers to communicate. There were a lot of developments with regards to making diving easier and safer recently. One of the possibilities is an assistant underwater robot, capable of tracking the diver, performing research, communicating with other divers and more.

Robotics department of Jacobs University Bremen has been working on several projects involving underwater robots. One of such projects, CADDY (Cognitive Autonomous Diving Buddy), is meant as an assistant for the diver, allowing to transport objects, take photographs and scan the surrounding area. However, equipment like this can be quite bulky to carry and can possibly put the diver in danger in critical situations. Therefore, a concise, hands-free communication method has to be implemented between the human and the robot to initiate tasks and report current danger status. Since CADDY is equipped with a depth camera, in this project we will be focusing on implementing one of such methods - communication using hand gestures. Equipped with an ability to process diver's sign language, CADDY can be controlled from a safe distance, while having virtually no learning curve.

1.2 Research Question.

Based on the problem discussed above, the following research question arises, that I will be solving throughout my bachelor's thesis project:

Can a robust gesture-based communication system be implemented for a use in the underwater robot?

Since in a system like this reliability is very critical, it is not enough to just implement the application logic. Thorough testing has to be performed to make sure all of the components of the system are functioning as expected. Therefore, the scope of this project will not be limited to research and implementation, but also creating proper design specifications and quality assurance.

1.3 Expected Research Contribution.

As a result of this research project, standalone gesture recognition application will be implemented. CADDY can be equipped with such software, which will definitely be a very valuable addition to already impressive features of the device. It will be possible to easily alter specific parts of the application and reuse the code in different projects.

I feel that enabling people to communicate better and more naturally with machines may simplify a lot of routine tasks, as well as potentially improve the speed of such communications, which may be crucial in the event of emergency situations.

2 State of the art.

Robot development has been advancing a lot over the course of last 10 years. We transcended all the way from simple programmable robots to truly autonomous machines, that can perceive the environment around them and interact with it. A lot of research has been done to perfect the communication with these "new-generation" robots. In 2006, a report [1] has been published, that shows that natural interaction (speech recognition, face recognition, gesture processing, etc.) is a great way of performing such a task, because of the variety of input data and emotional feedback as a result of such communication. Furthermore, artificial intelligence researchers [2] have discovered that when gesture recognition is present, humans tend to engage more with a robot and have better reviews, regarding their experience. However, such natural interfaces are not only useful for generic communication, but in some cases also for therapy. Robot AURORA [3] aims to aid children with autism by being a salient observer with changeable behavior. Researchers note, that the emotional feedback is one of the most important variables in this scenario and natural interaction through speech and gestures help to achieve that.

Even beyond research projects, the idea of implementing interface for consumer devices, controlled by natural interactions, isn not novel. At the moment, more and more manufacturers are investigating the possibility of adding voice or gesture control in their devices or applications. When the first Microsoft Kinect came out for Xbox 360, people were skeptical about the applications of a depth camera. Skeletal recognition and hand tracking were still quite choppy (partially due to hardware specifications and partially due to beta software) and controls using the camera were not intuitive and fluid. However, with the release of Xbox One and updated Kinect sensor, gesture control became one of the easiest ways to interact with the console. Kinect recognizes one of the three basic gestures: open hand, closed hand (fist) and lasso (2 fingers open), and bases interactions with the console on these. IR sensor with a better resolution allows for recognition in low- to no-light conditions. However, Microsoft is not the only company, that is interested in providing such features. In their newest lineup of Smart TVs, Samsung introduced a feature [4], which allows basic control of your television with the gestures. Using a regular camera, that is built-in in the unit, TV can recognize simple gestures, like flip movement, thumbs up, selection etc.

However, the potential of this technology is not limited to just entertainment. Number of startups are working on the devices, which would allow developers to embed gesture control in any application (most notable examples are Leap Motion sensor and Myo armband). With the powerful SDKs, it is just a matter of time until the technology is perfected enough for gestural interaction to become an everyday part of our lives. There are already some impressive experiments using the power of gestures: Microsoft Research has presented a Kinect sign language translator for American and Chinese languages [5], as well as a tool for the hospitals that allows touchless interaction with the computer during surgeries [6]. Nowadays, Microsoft Research is working on a new tool, called Handpose [7], that uses complex machine learning algorithms to enable developers to track hand gestures in complete 3D with the upcoming iterations of Kinect.

As we can see, even though the technology is still maturing and is not available to a lot of people, number of useful applications based on it is growing every day and the overall potential is very high.

3   Preparation.

3.1   Overview of Kinect 2.0.

For the implementation of this project we will need a depth camera, that is able to produce a clear depth frame for the hand segmentation and processing. While there are number of options to choose from, I chose **Microsoft Kinect 2.0**, because of its widespread availability and impressive specifications.

Second version of Kinect improved a lot over its predecessor, offering the following features:

- Color frames at 15-30 fps, depending on the light conditions (1920x1080)
- Depth frames at 30 fps (512 x 424) with improved fidelity
- Infrared frames at 30 fps (512 x 424)
- Skeletal tracking of up to 6 users (25 joints per person)
- Sound capture with an array of microphones

Kinect uses the following coordinate systems, that will be applied thorough the course of implementation of the project:
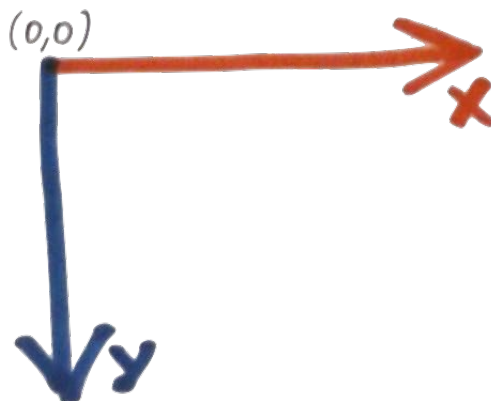

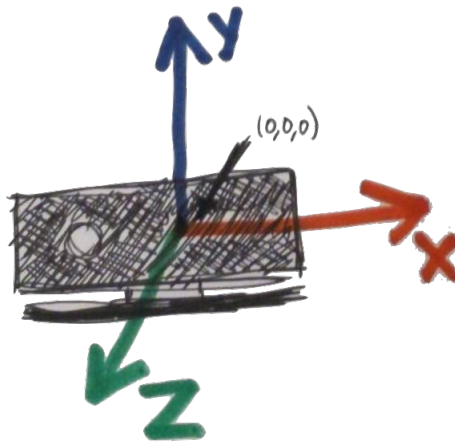Figure 1: Coordinate system for a 2D depth frame

Figure 2: Coordinate system for a 3D skeleton tracking frame

3.2 Choosing SDK.

Microsoft has developed an **in-house SDK**, that is available as free download for every registered Windows/Kinect developer. It is used to take full advantage of all hardware capabilities of the sensor and includes a lot of helper utility functions, that ease the processing of the data. Drawback to using this approach is a strict software limitation, allowing to only use it on Windows 8+ in the native C# environment (wrappers for Java and other languages exist, but are not mature enough in their implementation to suffice for this project).

Alternative option is using **libfreenect2**, a UNIX library based on *libusbx*, that allows basic interfacing with Kinect. The current version allows the export of raw unprocessed color, depth and IR data. It also requires basic reprogramming of Kinect sensor to bypass the software requirements and has no methods for synchronizing frames with each other and converting points from one coordinate space to the other. Typically, as done with the first version of Kinect sensor, this data is later used with a generic image processing library like **OpenNI** to calculate skeleton data and allow the use of other features. However, Kinect v2 is still on very early stages of support in OpenNI, not allowing the full skeletal tracking, needed for the scope of this application. Experiments also proved, that frame rate is far below fluent (average 3-5 fps), which is not far below real-time and thus suitable for this project.

As a result, we will base the implementation on native SDK, to allow us to unlock the full potential of the Kinect hardware, while allowing the fluid UI. We will also include **EmguCV**, a C# wrapper of OpenCV to allow us to easily use complex algorithms, like Contours Detection, Convex Hull and Pairwise Geometric Histogram computation.

3.3 Gesture set

The following basic gesture set has been selected for the purposes of this project. Default semantics of those gestures are described, and some suggested basic actions are

4

assigned to them, based on some of the recommendations presented in CADDIAN language draft [8]. Let's separate all the gestures into two kinds: **static**, corresponding to a single hand pose expression and **dynamic**, building upon the static ones, but analyzing a sequence of frames (allowing us to represent hand movement or a sequence of changing gestures). This list is meant only as the proof-of-concept and should be easily changeable or extensible, once more features are needed.

3.3.1 Static gestures

Figure 3: Static gesture set

- **s1.** Request to initiate communication with the ground crew.
- **s2.** Take a picture of the area.
- **s3.** Start/stop recording sensor data.
- **s4.** Stop current tasks and check whether the area is safe.
- **s5.** Ascend or descend *(depending on direction)*.
- **s6.** Confirm action or abort action *(depending on direction)*.
- **s7.** Set/change the active user.
- **s8.** Synchronize recorded data with the server.
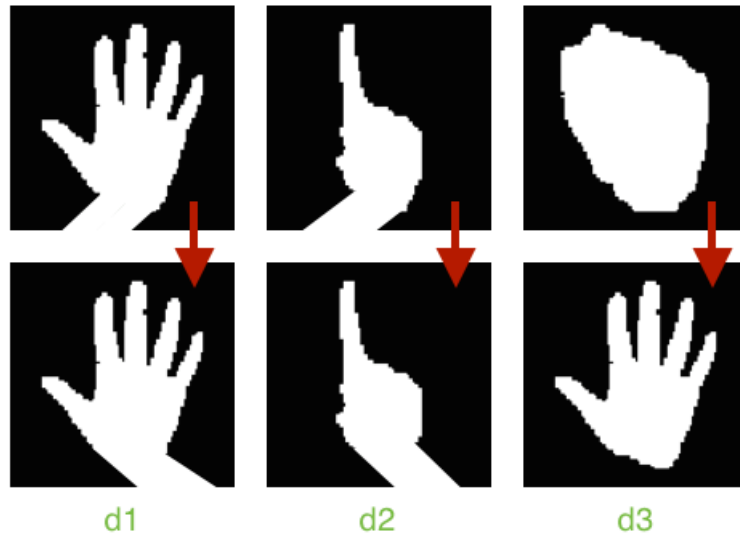
3.3.2 Dynamic gestures

Figure 4: Dynamic gesture set

- **d1.** Waving hand indicates that robot should listen for and perform a new task with a bigger priority over a current one.

- **d2.** Waving pointer finger indicates emergency. Immediate abort of current operations and ascend to the surface are requested.

- **d3.** Closing and opening hand rapidly is an indication that user's O_2 is running out.

3.4 Environment assumptions.

For the purpose of this research project, the following assumptions will be made about the setup and the environment:

- Kinect is positioned to minimize the background noise (uncertain values).

- User is located directly in front of the sensor.

- View is not obstructed by any object in front of the user.

- There is only one user in the frame *(multiple user tracking can be easily added, but adds computational intensity)*.

- When performing a gesture, hand is extended at least 30cm in front of the body.

- Gesture recognition will be less robust in the scenario of a fast moving hand, due to the motion blur produced by the sensor.

3.5 System requirements.

As a result of choosing our depth sensor and the SDK, the following system requirements need to be enforced to ensure the correct functioning of the application.

- 64-bit dual-core processor

- Dedicated USB 3.0 controller (Kinect will use its full capacity)

- > 4 GB of RAM

- Graphics card with support of DirectX 11

- Windows 8 or newer

- Kinect v2 SDK and EmguCV installed

3.6 Code License

All the code, written as the part of this research project will be available under **GPL v2** license. After the final submission, code will be available freely as an open-source repository hosted on Github. Anyone is free to use, modify or distribute this software and its parts, as long as they provide an attribution back and license it under the same terms.

4 Implementation.

4.1 Project structure.

For more advanced overview of the classes, that I have implemented, including definitions of fields and methods, please refer to the class diagram in *Appendix, section 10.3*.

- **App**
  Main class, responsible for launching the application.

- **MainWindow**
  Includes rendering of the main application window, initializing Kinect sensor and displaying the results of the recognition.

- **FrameBuffer**
  Stores last n frames (in our case, 35) and provides easy access to interpolated frame data and gesture recognition results, that can be used to improve fluidity of the system and are needed for the dynamic gesture recognition.

- **FrameProcessor**
  Designed as a black box and is responsible for receiving the frame from Kinect, performing all the needed operations on it and storing it in the FrameBuffer. Event-based system is used, allowing to call MainWindow's delegate functions to update the UI when needed, removing the necessity to wait for whole frame to process to get visual feedback.

Events: *DepthFrameReady, LeftHandReady, RightHandReady, LeftGestureRecognized, RightGestureRecognized, LeftDynamicGestureRecognized, RightDynamicGestureRecognized.*

- **Utility**
  Static class, including most global constants and helper functions, needed in the other classes.

- **HandRecognizer**
  Being initialized with an overall depth frame, this class is responsible for returning the segmented hand's binary mask, resized to a specific size for easier processing and comparison later on.

- **Hand**
  Processes the hand mask to locate and identify points of interest and other features to build a simplified model of a hand.

- **Gesture**
  Represents a configuration of the hand that corresponds to a defined gesture.

- **DynamicGesture**
  Represents prerequisites for a dynamic gesture of one of the pre-defined types.

- **GestureRecognizer**
  Analyzes the model of the hand against stored gestures to return possible matches.

4.2   Capturing the frames.

After the Kinect sensor is initialized, a reader is created to retrieve multi-source frames. Since the robot will be submerged underwater, some frame sources may become less effective for our implementation. For example, color frames will be very dark, where silhouette and background are almost indistinguishable and infrared frames will be distorted due to different propagation speed of the IR rays underwater. Therefore, we will be capturing depth frames (which Kinect implements using measuring response time of laser beams) and skeleton tracking data, based on them. Another advantage of such an approach is an opportunity to swap Kinect for any other compatible device. Given any device to capture depth (could be a sonar or well-calibrated stereo camera) and a framework to track body joints (OpenNI, for example, is adaptable to almost any device), the following implementation can be adapted to work in these conditions with little changes to the main algorithms.

Figure 5: Depth frame returned by Kinect

After the frames are captured, the raw depth data in retrieved as *ushort[]* with distances in millimeters to every pixels of a frame, ranging from 0 to 8000. Those values will be filtered to only make use of reliable values returned by sensor, which usually are within the range of 500 - 4500. After that, the body joints returned by the body frame will be mapped to depth frame's 2D coordinate space, using the provided *Kinect.CoordinateMapper* class.

4.3   Hand segmentation

As a next step, we will analyze the pixel neighborhood, corresponding to the hand joints, returned by the sensor. If those pixels are not a part of the body (in the other words, are in the background), we can judge that Kinect did not return the right values for this frame and, therefore, can skip all the further processing. In the other case, a modified Scanline FloodFill algorithm (see *Appendix, section 10.1*) will be used to segment and draw the binary mask of the hand. This particular version of algorithm was chosen for its superior efficiency, analyzing the lines of an image as opposed to individual pixels, resulting in much smaller queue size and thus, less iterations.

Instead of using a generic threshold in both directions, two separate thresholds will be used: one large one to apply to values closer than the hand's center (to ensure capturing of all the possible hand positions) and a smaller one to analyze values further away. That can be used only based on our previous assumption, that the hand is the closest object in sensor, allowing us to assume that no objects will interfere with forward thresholding. Additional check is implemented to check the distance from the hand's Z coordinate to the corresponding shoulder's Z coordinate (or head, if not available), that is aimed to minimize the number of cases, where the backwards threshold is large enough to segment bigger regions of the body, returning the data not suitable for proper gesture recognition.

In the algorithm, there is an additional check (denoted with *BeyondWrist*) is made to crop values behind the wrist line. That is performed by disregarding all the pixels, projections

9

of which on the plane, described by $v = (handX - wristX, handY - wristY)$, as returned by the Kinect sensor, are below the wrist point. However, due to the noise in processing of such a small region, often mistakes occur where the wrist point jumps and some fingers are being mistakenly cropped. That is the reason why the current implementation does not rely on this and instead attempts to achieve different means of classifying the wrist region later on. Should a special identifying marker be allowed or additional calibration of Kinect to return more precise coordinates, this method would return a very accurate representation of the wrist line.

Figure 6: Hand mask, its contour and minimum area rectangle

Resulting region then will be resized to a constant size (defined as 120x120 in my implementation) to remove variance based on hand's size and orientation of the sides of the bounding box (as suggested by [9]).

4.4 Palm segmentation

Before identifying points of interest on our segmented binary image, let's consider which degrees of freedom does the hand allow. On the image below, we can see the general model of hand, as well as all its 26 degrees of freedom. While this model is mostly accurate, there are some people who might anatomically have an extra degree of freedom in their thumb. We will not consider such cases and instead focus on simplified, 2D version of the general model with bending angles achievable for an average person.
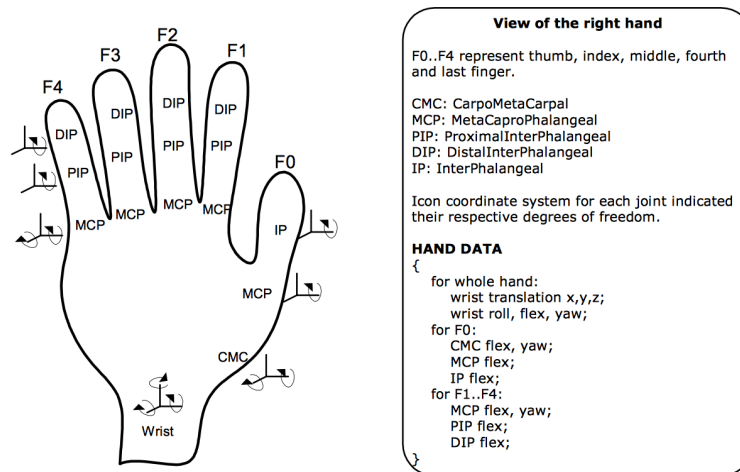
Figure 7: General model of the hand (taken from [10])

Next step is to locate so-called "palm point", the center point of the palm, which will help us to identify the palm, segment it and determine approximate orientation of the hand. To do that, we will use an optimized 2D distance transform algorithm (as derived by [11], implementation details in *Appendix, section 10.2*), where the matrix of the same size as the mask is filled with distances to the closest zero pixel boundary. The point with $d[i] = max(d)$ (or closest one to Kinect's hand joint if multiple points of this kind exist) is defined to be a palm point.

Now, let's draw a circle around this point, increasing radius every time until 5% or more of the sampling points correspond to non-skin values . Such a circle with a maximal radius will encompass the entire palm and is called inner circle. Once we have determined that, we can draw another circle, with the radius larger than inner circle's. Experimental factor of 1.60 was selected to separate the fingers, but retain the thumb even at its larger angle. This larger circle will function as a great mask, allowing us to analyze pixels outside of it for potential finger candidates.

The result of palm segmentation is illustrated at the image below:

Figure 8: Palm center, inner and outer circles on the hand contour

11

4.5 Finger recognition

To find the regions, corresponding to fingers, we will sample the values, located on the outer circle. If the pixel belongs to the mask of the hand, a connected region will be analyzed using a simplified version of Flood Fill to yield a position, occupied by the finger. At this point, regions smaller than the specified area cutoff (defined experimentally to be 15 pixels), will be discarded as noise to avoid detecting extra fingers. It is possible then to traverse presumed finger regions to find a fingertip (which is anatomically defined to be the point, furthest away for the center of the palm), as well as directional vector pointing from the base of the finger to that point (representing a simplified model of a finger).

Figure 9: Model after finger recognition

However, in some of the cases this will not be enough to make a correct judgement of a finger. If fingers are grouped together (located very close to each other), one bigger region will be returned, encompassing multiple fingers. In that case, rotated minimum area rectangle will be analyzed to make a prediction about a number of fingers inside the region. Let's define base width of that rectangle to be a length of a side, completely bordering with the outer circle of the hand. Now we can compare that value with a radius of the inner hand circle (representing the width of a segmented palm), decreased by 10%, making it roughly the same as the width of 4 connected fingers. This computation will yield the following ratio (denoted λ, suggested by Vitali Henne [12]):

$$\lambda = \frac{BaseWidth * 4}{InnerCircleRadius * 2}$$

The values of this ratio can be analyzed to make the final approximation regarding the number of fingers in the region.

- $\lambda < 1.320$: 1 finger in the region

- $1.320 <= \lambda < 2.315$: 2 fingers in the region

- $2.315 <= \lambda < 2.815$: 3 fingers in the region

- $2.815 <= \lambda < 4.000$: 4 fingers in the region

- $\lambda >= 4.000$: Discard the case as anatomically impossible.

Then, we can separate the base, according to number of fingers in the regions, connect those points with discovered fingertip and process each of the inferred fingers separately. An example of a case, where such processing is done is displayed on an image below.

Right hand   5   Right hand   4

No gestures detected!   Gesture detected:
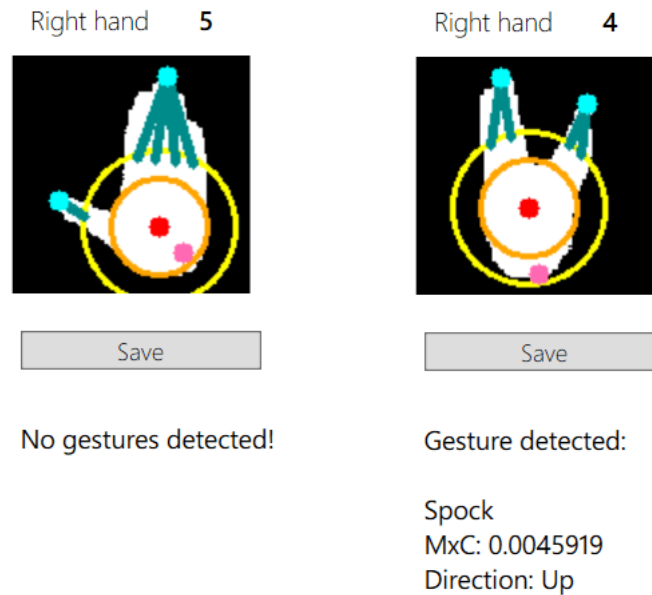
Spock
MxC: 0.0045919
Direction: Up

Figure 10: Inferred fingers detected and processed

In some cases it is possible that overall variation to the width of extracted regions can lead to errors, where one extra finger is detected or some fingers are missing. In that case, no gestures will be recognized for that particular frame and algorithm will retry with the next frame.

4.6 Recognizing static gestures

Now that we can build a simplified model of the hand, we can try to compare the hand to the set of predefined gestures to return the closest match. All the gestures are stored as JPEG files in *Gestures* directory, representing the mask of the hand. When initialized, additional arguments are passed to the constructor, such as gesture name and number of fingers displayed (to avoid further computations in case of mismatch).

Then, for every recognized model of the hand, the following methods are used to determine whether it represents a gesture:

- **Number of fingers comparison.**

  Obviously, if the number of fingers on the hand does not match the desired one for the particular gesture, we should halt the computation immediately and jump to the next gesture. This check ensures that fewer contours are being analyzed and compared and allows us to increase the threshold constants for the next two methods

13

to allow more variation in the overall hand shape.

- **Contour match.**

  As a first step, we extract the contours from both the gesture image and original hand's mask image. Then, we can use the following formula to return the similarity between two contours:

  $$I(A, B) = max_{i=1..7} \frac{|m_i^A - m_i^B|}{|m_i^A|},$$

  $$m_i^A = sign_i^A * log(h_i^A)$$
  $$m_i^B = sign_i^B * log(h_i^B)$$

  Here, h_i^A and h_i^B represent seven Hu moments [13] of an image. Since the specific properties of Hu moments are rotation and scale invariance (as well as skew invariance for seventh moment), this will allow us to detect gestures even when the hand is rotated.

  It is important to note, however, that since we are detecting gestures in 2D space of a depth frame, rather than 3D space, some gestures may stop being recognized if the overall contour changes too much due to different perspective of a hand for the camera. If the environment assumptions are fulfilled and hand is fully extended in front of the body, these problems should not arise.

  After analyzing the output of the algorithm (smaller values are better), a cutoff threshold for similarity was chosen to be 0.80.

- **Histogram match.**

  Since contour of the hand can be quite similar for different gestures, it is not enough to reliably determine whether a particular gesture has been shown. To solve this problem we will need to find another comparison metric, that would add confidence to our recognized gesture.

  One of such metrics can be derived by calculating and comparing the 2D pair-wise geometrical histograms (PGH) [14] of the images. The algorithm will calculate the angle and minimum/maximum distances for every pair of edges in the contour. This is done by selecting each of the edges as a base, while the function is looping through all other edges, recording distances from the points on the non-base edge and line of the base edge. The angle will define a row of the histogram in which all the bins that represent the distance between retrieved minimum and maximum distances are incremented.

  The results are then normalized, to prevent the errors in the situation when two images are not of the same size. Finally two resulting PGHs are compared using Bhattacharyya distance, that is commonly used to measure similarity between two probability distributions:

  $$d(H_1, H_2) = \sqrt{1 - \frac{1}{\sqrt{H_1^- H_2^- N^2}} \sum_I \sqrt{H_1(I) H_2(I)}}$$

14

After analyzing the output of the algorithm (smaller values are better), a cutoff threshold for similarity was chosen to be 0.20.

In addition, another small threshold value is used for the product of the results of contour match and histogram match (cutoff threshold = 0.0125). That ensures, that even if one of the values fluctuates, the overall similarity factor is very high.

Using this comparison method, the hand will be evaluated against the entire gesture set and the closest match satisfying the cutoff constraints will be returned.

Additionally, in the case where at least one finger is displayed, direction of the hand will be determined by averaging all the fingertips and observing the distance between the resulting point and extrema (up-, down-, left- and right-most points). In some cases (like with gestures **s5** and **s6**), a completely different action can be performed on a recognized gesture, depending on its direction.

Since the algorithm relies heavily on shape and contour of the hand, it is possible that recognition results and precision may vary from a person to person. However, because of this nature of storing gestures as pictures, it is trivial to adapt the system to a particular user's hand to improve the reliability of the recognition algorithm. To do that, user has to show a desired gesture, press the "Save" button, and replace the existing gesture image (or add the new one) with the one outputted to the directory, from which the program is launched.
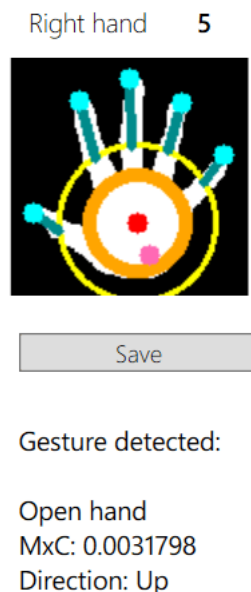

Figure 11: Results of gesture recognition (with option to save the mask).

Some other methods have been used for this purpose, such as Dynamic Time Warping [15], Earth Mover's Distance [16] or SURF feature detection [17]. However, despite robustness and excellent results, that these algorithms produce, their computational complexity often does not allow us to use reliably them in real-time and multi-hand recognition scenario.

4.7 Recognizing dynamic gestures

Since dynamic gestures are happening across multiple frames, simple frame-by-frame analysis (as used for static gesture recognition) is not enough to reliably judge if a gesture has been recognized. Instead, we will retrieve the latest n frames, stored in the frame buffer and analyze whether their features correspond to the requirements set by the gesture. Here are some of the features, that we are interested in:

- Depth data of the frame.

- Position and relative offset of the joints.

- Position of hand's bounding box.

- Number of fingers shown.

- Static gesture(s) recognized.

Due to the nature of dynamic gestures, evaluation function may differ a lot between defined dynamic gestures. For that reason we can introduce a notion, called dynamic gesture types, encompassing and implement a certain recognition behaviors. Two of such types are needed to implement our defined dynamic gesture set: **Wave** (as seen in **d1** and d2) and **Alternation** of static gesture states (as seen in **d2**).

When evaluating **Wave** gestures, we will analyze the last 35 stored frames (corresponding to roughly 1-2 seconds of motion). Since the static gesture can not be successfully recognized in some cases due to motion blur, we will not impose strict requirements for all frames to match the required gesture. Instead, let's say that if gesture has been recognized in more than 70% of frames, it will count as a successful recognition. Additionally, we track relative positions of elbow and hand joints, allowing us to analyze at which points has the hand been to the left and to the right of the elbow. If the pattern is noticed where there is an alternation between these two states, gesture will be successfully recognized.

In a case of **Alternation** gestures, we can only consider the static gestures, that have been recognized. If both gestures prevail in the analyzed frame window, and there is a trend of switching between them, we can conclude that Alternation gesture has been shown.

5 Final deliverable.

5.1 Installation.

Either of the following two methods can be used to try the final application for yourself. Please ensure that your hardware exceeds the minimum requirements and Kinect 2.0 sensor is connected.

5.1.1 Download built executable (32-bit).

- Install drivers from Kinect SDK 2.0 (if needed).

- Download and extract *the latest build*.

- Run KinectGR.exe. That is it!

5.1.2 Build the solution.

- Install Kinect SDK 2.0, EmguCV and Visual Studio 2013+ (if needed).
- Download the solution from *the Github repositiory*.
- Launch the solution and ensure that EmguCV and Kinect SDK references are set correctly, and OpenCV dlls are being copied to the build path.
- Copy **Gestures** folder to *C:/Gestures* (or change the paths in FrameProcessor.cs).
- Build (in x86 mode) and run!
  Do not forget to connect your Kinect to a compatible USB 3.0 port.

5.2 GUI.

Figure 12: Application's interface

1. Depth frame from Kinect data. If Kinect is not connected an error is displayed here.
2. Box representing position of the detected hand.
3. Mask image of a hand with important points marked.
4. Counter indicates number of detected fingers.
5. Static gestures that were recognized. Similarity product and direction are displayed for recognized gestures.
6. Dynamic gestures that were recognized *(none in the displayed frame)*.
7. Save the current hand mask to an image file (used for changing gesture sets).

17

5.3 Features.

When user launches the application, an immediate request for Kinect's status is being issued. If sensor is not connected, an error message will be shown, otherwise depth data will be displayed on the screen and frames will be analyzed for the potential gestures.

When the hand is being extended in front of the body, hand segmentation algorithm will run, returning display image for the hand and its position, that will be highlighted on the depth image. Then, algorithm will try to match hand's contour with gesture set, that is defined within the gesture recognizer. If a match is found, user will be notified real-time about the name, direction and similarity value of detected gesture. Finally, stored frames will be analyzed with respect to requirements set by the dynamic gestures to return any possible matches.

Since application is tailored for use with gestures, there are almost no interactable elements in application's interface. Only notable exception in the "Save" button, that, as mentioned before, can be used to save hand masks for later use in the gesture sets.

For more information regarding the flow of the application, please, refer to the sequence diagram in *Appendix, section 10.4*.

6 Evaluation.

Evaluation criteria for this research project are strongly coupled with the project requirements and implementation quality. The project is assessed against the following criteria, that can be used to evaluate the results of this thesis:

6.1 Performance.

Visual Studio's built-in profiler was used to analyze the performance of the implementation, focusing on the following:

- **CPU usage** to test whether algorithms have been properly optimized.

- **Memory consumption** to check for possible memory leaks and analyze performance of the garbage collector.

All the measurements have been done on a personal laptop with the following hardware specifications:

- **Model:** MacBook Pro (Retina, 13-inch, Late 2013)

- **CPU:** 2.6 GHz Intel Core i5

- **RAM:** 16 GB 1600 MHz DDR3

- **GPU:** Intel Iris 1536 MB

Following sections present the results, as well as short explanations and findings.

6.1.1 CPU usage.

As we can see, despite Kinect's high system requirements, overall CPU usage stays consistent and quite low. It was achieved as a result of heavy optimization of all recurring algorithms (e.g. Flood Fill, Distance Transform, Contour Matching etc.). Highest possible values were detected in the range of 40-60%, for cases when the depth data was very noisy and both hands have been tracked for complex set of gestures.

Predictably, gesture recognition is the most computationally intensive part of the implementation, corresponding to 31.20% of overall CPU usage, followed by hand recognition and rendering depth frames into bitmaps.
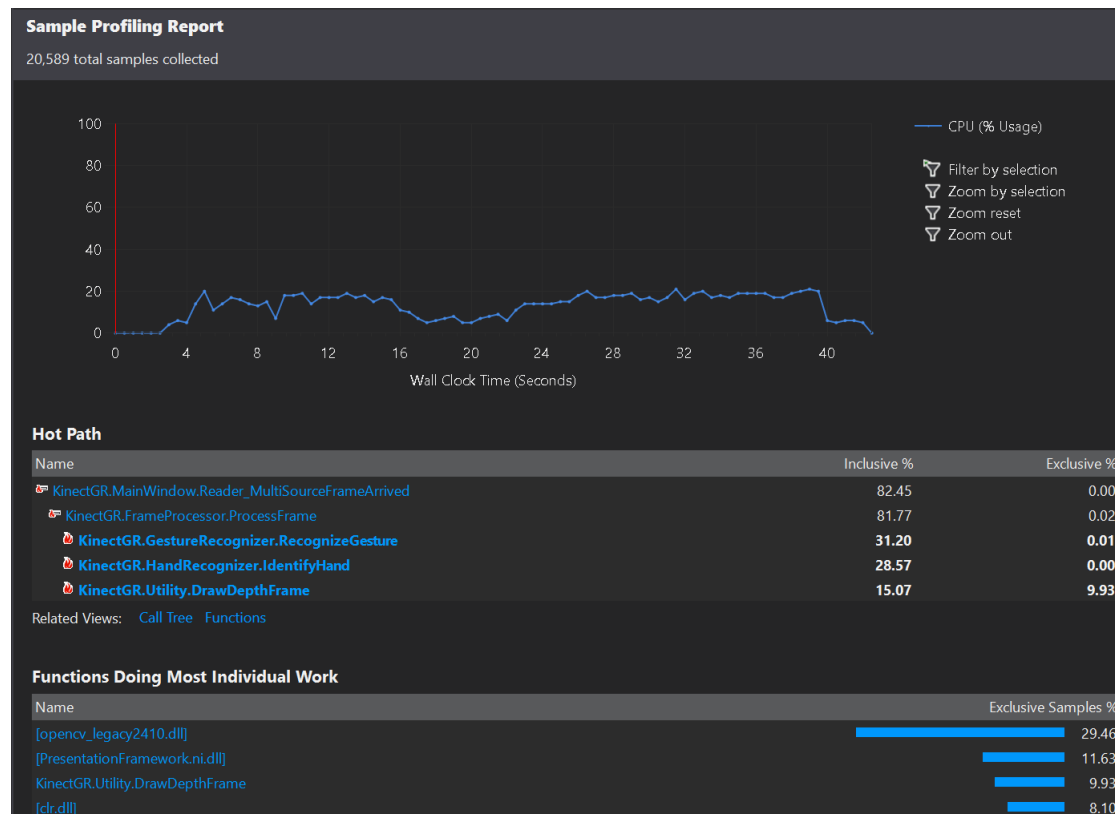
Figure 13: Results of CPU profiling.

6.1.2 Memory consumption.

On a graph below, we can see that memory usage is consistent and conclude the absence of memory leaks. Usage slowly increases, as the frame buffer is populated with processed frames and then converges, in this case, to about 170 Mb (size of the full buffer and all the temporary storage for the algorithms).
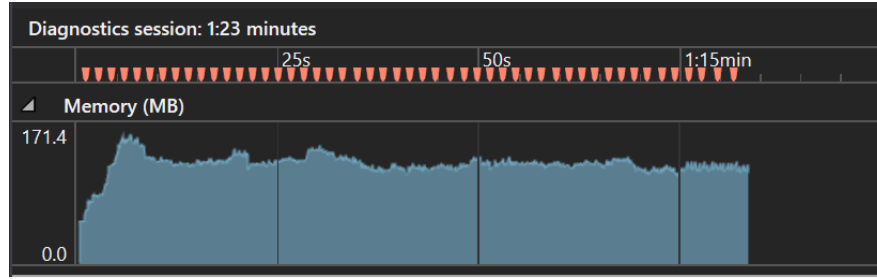
Figure 14: Results of memory profiling.

More in-detail analysis allows us to check which functions allocated bigger amounts of memory and try to optimize their space complexity. Looking at the results, we can note that rendering depth frame to the screen and Flood Fill consume the most memory, due to their need to allocate large matrixes for their computations. C#'s built-in Garbage Collector often executes only after a certain time has passed, thus keeping these objects in memory for a bit longer.
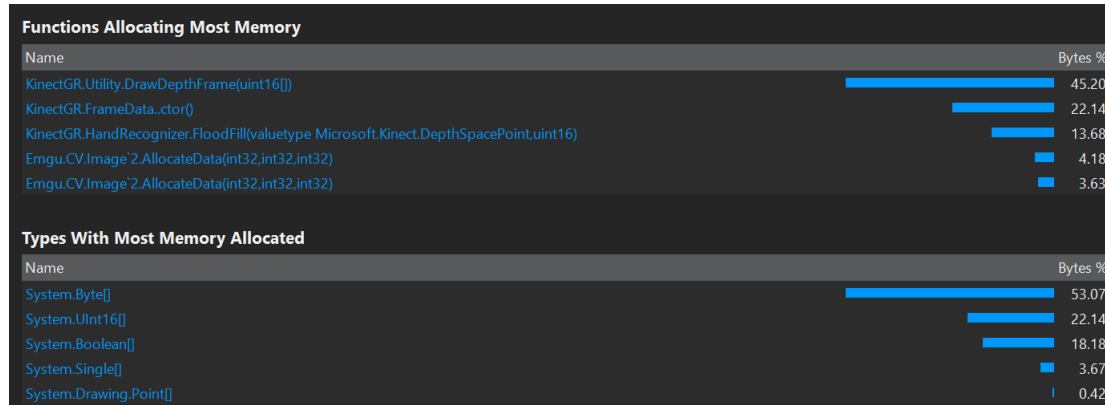

Figure 15: Details about memory consumption.

6.2 Functionality.

Application successfully implements all the features, described in this thesis paper. One user is being tracked and hand segmentation starts when all the environment assumptions are fulfilled (e.g. hand extended at least 30 cm in front of the body). Then, it attempts to recognize a static gesture and displays the results. Finally, sequence of frames is analyzed to notify the user about dynamic gestures recognized (if any).

6.3 Correctness.

5 test participants of different gender and varying hand features were asked to use the application to analyze it for correctness. Following observation has been made as the result:

- Hand was correctly recognized and segmented in all of the cases.

20

- Hand was correctly ignored, based on the environment conditions (frontmost object in the frame, far enough from the body). The only case, that failed for some of the participants, was the one, where user would be standing sideways (which is not a defined use case for our application). Then, shoulder Z coordinate would not be a reliable reference anymore, causing a bigger region (the entire arm) to be segmented. This can be improved by analyzing more joint values to detect the incorrect position of user with respect to the sensor.

- Fingers were successfully recognized in all cases, where each finger has its individual region. In a case with inferred fingers present, some participants reported that slight adjustments to λ ratio constants were needed for optimal performance.

- Static gestures were recognized most of the time, using the normal position of the hand. When trying to rotate the hand, gesture would disappear sometimes but promptly reappear. When swapping gesture set for the one optimized for a particular user's hand, 90%+ recognition rate has been achieved.

- Since dynamic gestures rely heavily on static gesture recognition, the results of this section highly depend on the results of the previous one. Gestures have been recognized in vast majority of the cases, where static gesture recognition succeeded and hand displacement was within the defined threshold.
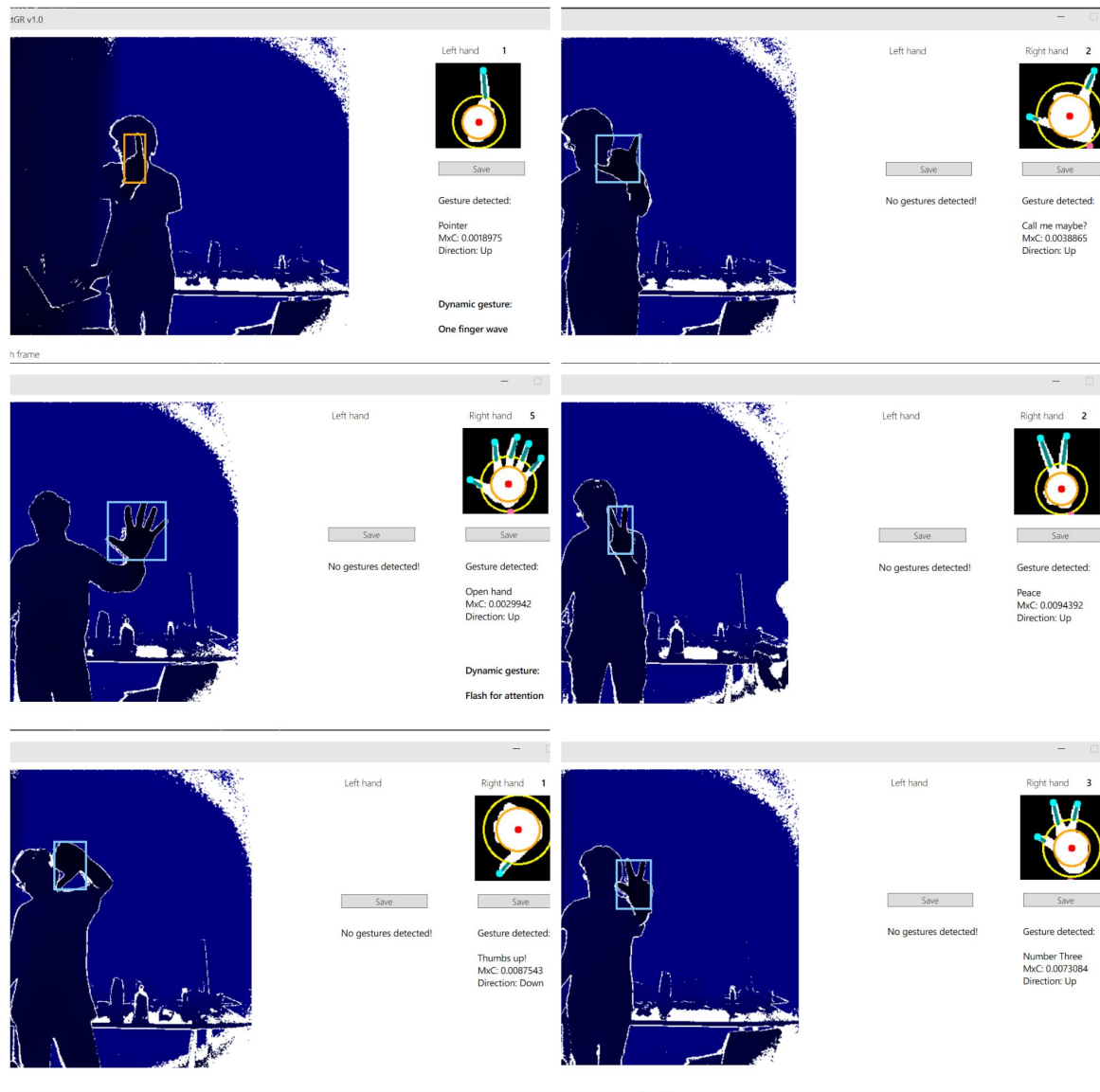
Figure 16: Some screenshots from testing and evaluation phase.

6.4 Modularity.

Because of the way the application is structured, it is very easy to alter specific parts of the system, while leaving the others intact. As described in the previous section, changing the gesture set of the application and adapting the program for a particular need is intuitive and simple. This successfully fulfills the goals set in the proposal part of the project and allows for easier adaptation later on.

6.5 Code Quality and Documentation.

Strict C# code design guidelines were enforced over the course of development, using JetBrains' Resharper plugin. Each class has its own purpose and scope, variables and

methods have descriptive names, and comments are supplied in places where code is non-trivial. Finally, XML-style comments for classes and methods allow to easily generate HTML documentation pages (using Sandcastle).

6.6   Testing.

Over the course of development I was adhering to **test-driven development** to minimize the time spent on fixing bugs and narrowing the source of potential problems. Separate test suite was written with unit tests for every component of the application. That helped me over the course of development to assess the result of the operations algorithms were relying on (e.g. reading data from hardware or performing a Flood Fill algorithm). That way I could be more confident about the algorithms' correctness and instead focus on improving the overall recognition success rate. Most of the testing has been performed with a live Kinect stream to account for potential noise in the data stream.

7   Discussion.

7.1   Improving the results.

Kinect SDK, that is used for my implementation, is not yet optimized for closed-range tracking of the specific body joints and instead, it focuses on tracking skeleton as a whole. That results in the number of situations, when inferred wrist joint location is completely out of sync with the actual position of the wrist. That, in turn, affects the precision of the gesture recognition by mistakenly classifying some fingers as wrist in the worst cases.

Overall robustness of the system could be further improved by implementing a calibration phase, that would calculate specific constants and save hand masks specifically for user's hand, tailoring the experience for him. Finally, using more methods for gesture recognition could bring the overall accuracy up, reducing the amount of false classifications, requiring better hardware specification in process.

7.2   Extending the application for more complex cases.

Depth mask of the hand can be reused to calculate the differences in depth at midpoints and towards the fingertip to yield a better, 3D-augmented approximation of the general model of the hand. That will allow to define more features to describe the gesture and thus, expand the recognition far beyond the simple gesture set. Multiple sources familiar with gesture recognition recommend using Hidden Markov Models and Bayesian Networks [18] for such cases to implement a machine learning algorithm that automatically gathers a set of relevant features from all of the available data. Application could then be extended to include a "learning mode", where user can train the algorithm on the number of applicable cases under different environment conditions to include new static and dynamic gestures to the gesture set.

8   Conclusion.

Summarizing all the information presented above, we can see that gesture recognition, despite being a pretty novel technology, has a lot of useful applications. In our case, communicating with robot underwater can be complicated due to a number of natural events. Gesture recognition system allows to control the robot from the safe distance and keep track of safety of the diver. Step-by-step implementation of gesture recognition was presented, starting from hand segmentation and ending with identifying different kinds of gestures. It proved quite reliable in specific cases and further work will be performed to improve the application to be as robust as possible.

9   Acknowledgements.

10 Appendix

10.1 FloodFill implementation

```
FloodFillCheck (x, y, startDepth, mask) : Bool
    if (!IsValidPoint(x, y)) return False
    if (depth[x, y] is not reliable) return False
    if (mask[x, y] is already marked) return False
    if (depth[x, y] > startDepth + bwdThreshold) return False
    if (depth[x, y] < startDepth - fwdThreshold) return False
    // if (BeyondWrist(x, y) return False (!)
    return True

FloodFill (startPoint, startDepth) : Hand
    Queue q, Bool mask[w*h]
    minX = minY = +Infinity
    maxX =  maxY = -Infinity

    mask[startPoint.X, startPoint.Y] = true
    q.Add(startPoint)

    while (q is not empty)
        p = q.Pop()

        w = p.X, e = p.X
        while (FloodFillCheck(w-1, y, startDepth, mask)
            w = w-1
            mask[w, p.Y] = true
            if (w < minX) minX = w
        while (FloodFillCheck(e+1, y, startDepth, mask)
            e = e+1
            mask[e, p.Y] = true
            if (e > maxX) maxX = e

        for (i in range (w, e))
            if (FloodFillCheck(i, y-1, startDepth, mask)
                if (y-1 < minY) minY = y-1
                q.Push(i, y-1)
            if (FloodFillCheck(i, y+1, startDepth, mask)
                if (y+1 > maxY) maxY = y+1
                q.Push(i, y+1)

    position = Rectangle(minX, minY, maxX - minX, maxY - minY)
    return Hand(position, mask)
```

10.2 2D Distance Transform implementation

```
DistanceTransform1D (arr, n) : Double[]
  Int v[n], Double z[n+1], Int k = 0, Double s = 0
  v[0] = 0, z[0] = -Infinity, z[1] = +Infinity

  for (int i =1; i < n; ++i)
      s = ((arr[i] + i * i) - (arr[v[k]] + v[k] * v[k])) /
          (2.0 * i - 2.0 * v[k]);

      while (s <= z[k])
          --k, s = ((arr[i] + i * i) - (arr[v[k]] + v[k] * v[k])) /
                   (2.0 * i - 2.0 * v[k]);

  ++k, v[k] = i, z[k] = s, z[k + 1] = +Infinity

  k = 0, Double result[n]

  for (int i = 0; i < n; i++)
      while (z[k + 1] < i) ++k
      result[i] = ((i - v[k]) * (i - v[k]) + arr[v[k]])

  v = null, z = null
  return result

DistanceTransform2D (mask) : Double[]
  Double result[w*h], Double tmp[w*h]

  ### For columns ###
  for (int x = 0; x < w; ++x)
      for (int y = 0; y < h; ++y)
          tmp[y] = mask[y*w+x] ? MAX_DOUBLE - 1 : 0;

      Double d[] = DistanceTransform1D(tmp, h);

      for (int y = 0; y < h; y++)
          result[y*w+x] = d[y];

  ### For rows ###
  for (int y = 0; y < h; y++)
      for (int x = 0; x < w; x++)
          tmp[x] = result[y*w+x];

      Double d[] = DistanceTransform1D(tmp, w);

      for (int x = 0; x < w; x++)
          result[y*w+ x] = d[x];

  return result
```
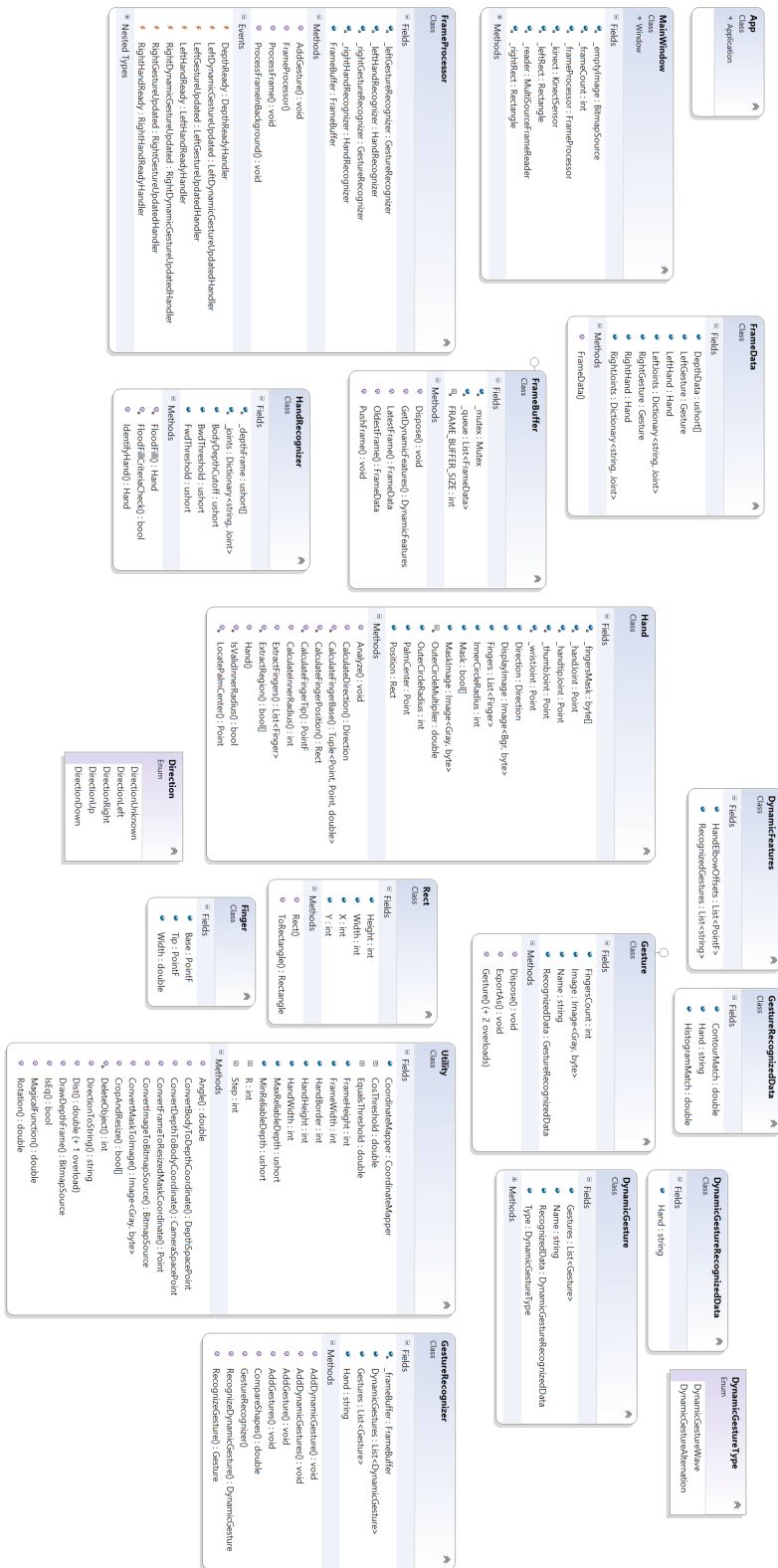
10.3 Class diagram.
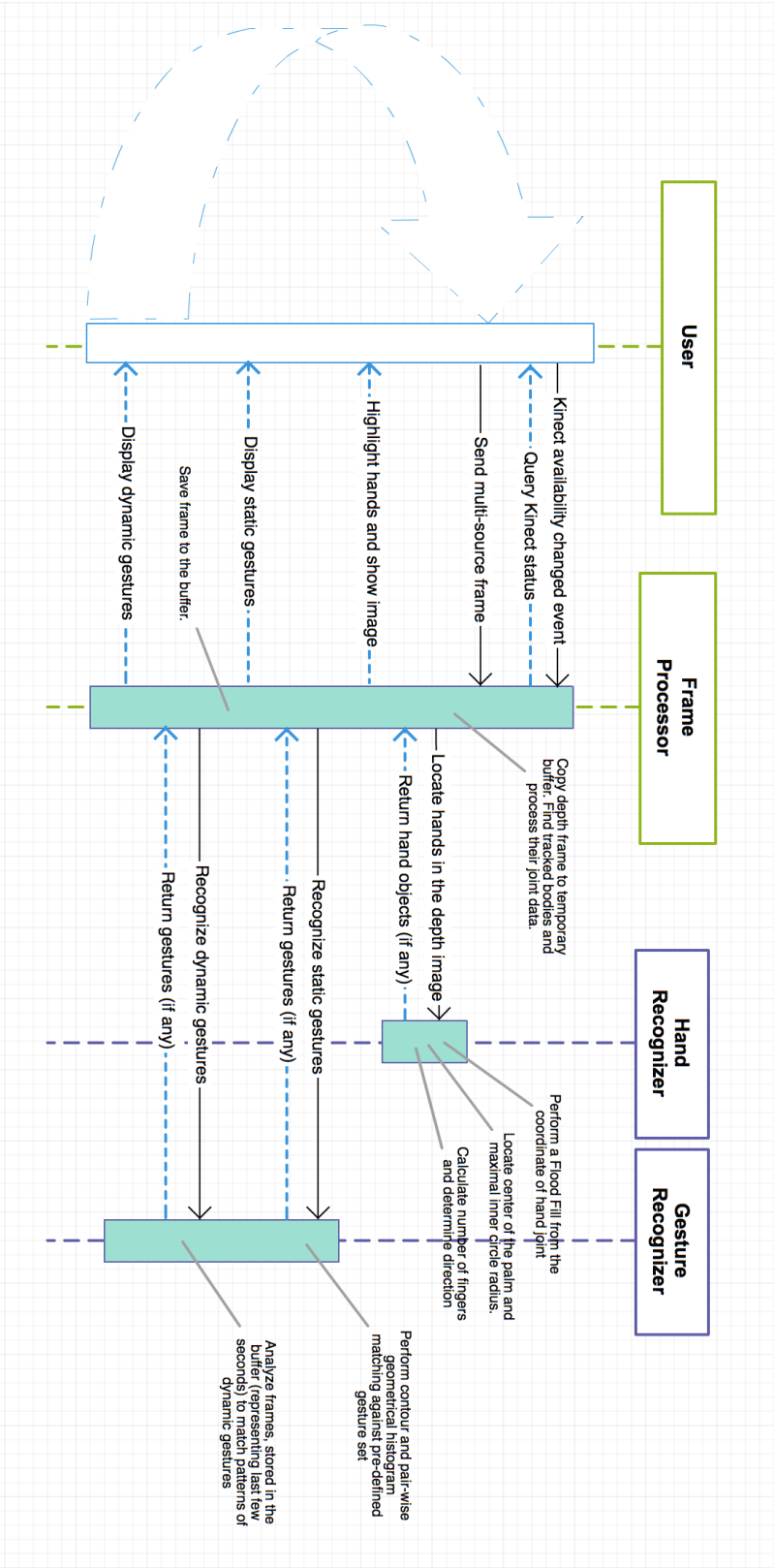

Figure 17: Class diagram

10.4 Sequence diagram.

Figure 18: Sequence diagram

11 References

[1] Seong-Whan Lee. Automatic gesture recognition for intelligent human-robot interaction. Automatic Face and Gesture Recognition, 2006. FGR 2006. 7th International Conference, April 2006.

[2] Paul; Stribling Penny; Dautenhahn Kerstin Robins, Ben; Dickerson. Robot-mediated joint attention in children with autism: A case study in robot-human interaction. Interaction Studies, Volume 5, Number 2, 2004.

[3] Cory D. Kiddb Neal Lesha Charles Richa Candace L. Sidnera, Christopher Leea. Explorations in engagement for humans and robots. Artificial Intelligence, Volume 166, Issues 12, August 2005.

[4] Samsung. Smart TV Gesture Book. `http://www.samsung.com/global/microsite/tv/common/guide_book_5p_vi/thumb_up.html`.

[5] Microsoft Research. Kinect Sign Language Translator. `http://research.microsoft.com/en-us/collaboration/stories/kinect-sign-language-translator.aspx`.

[6] Microsoft Research. Kinect Launches a Surgical Revolution. `http://research.microsoft.com/en-us/news/features/touchlesssurgery-060712.aspx`.

[7] Microsoft Research. Handpose: Fully Articulated Hand Tracking. `http://research.microsoft.com/en-us/projects/handpose/`.

[8] Marconi L. Chiarella D., Cutugno P. CADDIAN: A Human-Robot Interaction Language Based On Gestures. Draft. CNR-ILC, September 2014.

[9] Jianning Liang Jing Zhang Zhi-hua Chen, Jung-Tae Kim and Yu-Bo Yuan. Real-Time Hand Gesture Recognition Using Finger Segmentation. East China University of Technology, April 2014.

[10] Rob Audenaerde. Towards Ubiquitous Document Scanning. Master's thesis, University of Twente, December 2004.

[11] Daniel P. Huttenlocher Pedro F. Felzenszwalb. Distance Transforms of Sampled Functions. Cornell Computing and Information Science TR2004-1963, 2004.

[12] Vitali Henne. Hand Gesture Recognition with a Depth-Sensing Camera. Bachelor's Thesis, Karlsruhe Institute Of Technology, June 2011.

[13] Jinsong Leng Zhihu Huang. Analysis of Hu's Moment Invariants on Image Scaling and Rotation. Edith Cowan University Research, 2010.

[14] S. Coupe. The current status of pairwise geometric histograms. University of Manchester, Imaging Science and Biomedical Engineering Division, June 2009.

[15] Daniel James Ryan. Finger and gesture recognition with Microsoft Kinect. University of Stavanger, EECS Department.

[16] Yuan Nanyang Zhou Ren Junsong and Zhengyou Zhang. Robust Hand Gesture Recognition Based on Finger- Earth Movers Distance with a Commodity Depth Camera. Conference: Proceedings of the 19th International Conference on Multimedia 2011, Scottsdale, AZ, USA.

[17] Tinne Tuytelaars Herbert Bay and Luc Van Gool. SURF: Speeded Up Robust Features. ETH Zurich, Katholieke Universiteit Leuven.

[18] Zoubin Ghahramani. An Introduction to Hidden Markov Models and Bayesian Networks. University College London, Gatsby Computation Neuroscience Unit.