Finger and gesture recognition with Microsoft Kinect

Daniel James Ryan

Department of Electrical and Computer Engineering

University of Stavanger

Abstract

The Kinect is a cheap and wildly spread sensor array with some interesting features, such as a depth sensor and full body skeleton tracking.

The Kinect SDK does not support finger tracking; We have therefore created algorithms to find finger positions from depth sensor data. With the detected finger position we use dynamic time warping to record and recognize finger gestures.

We have created an API that can record and recognize finger gestures. The API was created with focus on ease of use and the possibility to customize and change core algorithms. Even the Kinect can be changed for other similar devices. This opens up a new field of applications utilizing the Kinect and finger gestures.

Acknowledgements

I would like to thank professor Reggie Davidrajuh for providing valuable feedback on writing this thesis.

I would also like to thank Glenn F. Henriksen at EVRY for providing constructive feedback on coding the API and for programming advice in general.

Contents

List of Figures **vii**

Glossary **ix**

1 Introduction **1**

1.1 Inspiration 1

1.2 Existing solutions 2

1.3 Vision 2

2 The Kinect sensor **3**

2.1 Kinect versions 3

2.1.1 Commercial and development usage 3

2.2 Specs 3

2.3 Kinect for Windows SDK 4

2.3.1 Data streams 4

2.4 Possibilities with the Kinect 4

3 Contour tracking **7**

3.1 Existing algorithms 7

3.2 Finding the initial pixel 7

3.3 Tracking the contour 8

3.3.1 Search directions 8

3.3.2 Single pixel lines 10

3.3.3 Backtracking 11

3.3.4 Track two hands 12

3.4 Termination states 12

3.4.1 Termination state 1: A pixel was discovered twice. 13
3.4.2 Termination state 2: A fixed number of pixels were discovered. . 13

4 Finger recognition 15
4.1 Related works 16
4.2 Curve detection 16
4.3 Finger detection 17

5 Gesture Recognition 19
5.1 Related works 19
5.2 Note 19
5.3 Recording a gesture 20
5.4 Recognizing a gesture 20
5.4.1 Step 1: Finding a candidate gesture 20
5.4.2 Step 2: Euclidean distance matrix 21
5.4.3 Step 3: Accumulated distance matrix 21
5.4.4 Step 4: Lowest distance path 22
5.4.5 Optimizations and improvements 22
5.4.5.1 Euclidean distance 23
5.4.5.2 Accumulated cost matrix 23
5.4.5.3 Lowest distance path 24
5.5 Store gestures 24
5.6 Retrieve stored gestures 25

6 Auto depth interval detection 27

7 Enhancements 31
7.1 Smoothing 31
7.2 Prediction 31
7.2.1 Exponential moving average 32

8 Virtual reality 33
8.1 Related works 33
8.2 Introduction 33
8.2.1 On-axis perspective 33

8.2.2 Off-axis perspective 34
8.3 View frustum 34
8.4 Implementation 36
8.5 Limitations 36

9 Results **37**
9.1 Program flow 37
9.2 Events 37
9.2.1 DepthDistanceUpdated 37
9.2.2 ContourDataReady 38
9.2.3 FingertipLocationsReady 39
9.2.4 GestureRecognized 39
9.2.5 GestureRecorded 39
9.2.6 Performance 40
9.3 Modularity 40
9.3.1 Interfaces 41
9.3.2 Modules 41
9.3.2.1 Kinect 42
9.3.2.2 ContourTracking 43
9.3.2.3 CurveDetection 44
9.3.2.4 FingerRecognition 45
9.3.2.5 GestureRecognition 46
9.3.3 Tying modules together 47
9.4 Virtual reality 47
9.5 Code 48

10 Discussion **49**
10.1 Contour tracking 49
10.1.1 Finding the initial pixel 49
10.2 Curve detection 50
10.2.1 The k-curvature algorithm 50
10.3 Detecting Fingertips 50
10.3.1 False fingertip 50
10.4 The Kinect 50

10.5 Commercialization 51

11 Conclusion **53**

References **55**

Appendix A Create line segment **57**

Appendix B **59**

B.1 ContourDataReady event with auto interval scan. 60

B.2 Program flow 61

B.3 Main class 63

List of Figures

3.1 Local search grid. 9
3.2 Single pixel lines. 11
3.3 Minimum hand contour pixels (the red colored pixels). 14

4.1 The raw and processed images. 15
4.2 Finding the fingertips and their pointing direction. 18

5.1 Hand frame matrix. 22
5.2 Warping Constraints. 24

6.1 Example of a scan sequence. 28
6.2 Depth frames from an auto interval scan. 29

8.1 Perspective. 35
8.2 Frustum extents. 36
8.3 VR space. 36

9.1 IKinect and the implementation. 42
9.2 IContourTracking and the implementation. 43
9.3 ICurveDetection and the implementation. 44
9.4 IFingerRecognition and the implementation. 45
9.5 IGestureRecognition and the implementation. 46

Glossary

Kinect	Kinect is a sensor device from Microsoft for Xbox 360 and Windows. It has a microphone array, RGB camera and a depth sensor.
XNA	XNA is a product from Microsoft that facilitates video game development for Windows, Xbox 360 and Windows Phone.
NUI	Natural User Interface.
API	Application Programming Interface.
SDK	Software Development Kit.
RGB	An additive color model; Red, Green, Blue.
YUV	A color model used to encode video images.
DTW	Dynamic time warping. Recognition algorithm between two time series.
View frustum	The region of space a camera sees in a 3D environment.
VR	Virtual Reality.

LIST OF FIGURES

1

Introduction

With the launch of Kinect for Xbox 360 in November 2010 and for Windows in February 2012, ordinary consumers have the ability to cheaply buy a sensor array and, with some programming knowledge, easily play with the received data.

The Kinect for Windows SDK provides, amongst other things, out of the box skeleton tracking. However it does not provide finger tracking or gesture recognition. We have therefore created an API that enable users to find fingertip locations and pointing direction and to record and recognize finger gestures. In addition it also gives the ability to create virtual reality using head tracking and the XNA framework. We will talk more about the Kinect in chapter 2.

1.1 Inspiration

The inspiration to do finger tracking and gesture recognition was found in the science fiction movie "Minority Report" (1) where the main character uses his hands and fingers to navigate and manipulate video and pictures on a large horizontal screen. However his hands and fingers were not touching anything but the air. This gives us some insight on how NUI can be used to interact with a computer. In addition there did not exist any satisfying solution that would enable someone, with relative ease, to create a NUI as seen in the movie or any NUI based on finger tracking with Kinect. This gave us the opportunity to be the first to create a tool to facilitate easy and quick NUI development, specifically with the use of fingers, with Kinect.

Inspiration to create virtual reality came from a video (2) were a Nitendo Wii remote was used to achieve VR on an old television by utilizing the users head movement.

1.2 Existing solutions

There exists some open source libraries that try to do finger tracking and gesture recognition, but most of these were either not working correctly, performed poorly due to poor programming or were not documented and hard integrate into other applications. We were not able to find any libraries that combined finger tracking and gesture recognition that would perform on a reasonably level.

1.3 Vision

Our vision is to create an API that can do finger tracking and gesture recognition with reasonably good results and make it available to a broad spectrum of people. To do this the API should be easy to use and the code should be well documented and easy to read. We will talk more in depth about the API in chapter 9.

2

The Kinect sensor

2.1 Kinect versions

There are two different versions of the Microsoft Kinect sensor; Kinect for Windows and Kinect for Xbox 360. The Kinect for Windows sensor does not work with the Xbox 360 gaming console.

2.1.1 Commercial and development usage

Both versions can be used for development, but only the Kinect for Windows can be used for commercial purposes. The Kinect for Xbox sensor will not work with the Kinect for Windows runtime (v1.0 at the time of writing), however it does work with the Kinect for Windows SDK. The Kinect for Windows runtime needs to be installed by the end user to enable the Kinect for Windows sensor to work with the application.

The commercial license of the v1.0 release allows us to develop, distribute and sell our applications using the Kinect for Windows sensor on Windows platforms (3).

2.2 Specs

The Kinect has a RGB camera, depth sensor (IR) and a microphone array. The RGB camera and depth sensor has a resolution of 640×480 pixels at 30 Hz. The sensor array stand also features a motor so the tilt can be changed without physical interaction.

2.3 Kinect for Windows SDK

The Kinect for Windows beta 2 was released on November 1, 2012. and gave us an official way to create applications using the Kinect sensor. This beta release is licensed only for research and development purposes. On February 1. Microsoft simultaneously released version version 1 of the SDK and the Kinect for Windows sensor. This would allow anyone to create and sell applications using the Kinect sensor. A version 1.5 was released May 22.

The SDK allows a programmer to access the video, depth and sound stream produced from the Kinect sensor.

With the SDK we can get skeleton tracking data from up to two players who are in front of the Kinect sensor. The skeleton tracking data gives us the position of 20 joints and has a depth rage of about $0.7 - 6$ meters. The SDK also provides sound triangulation.

2.3.1 Data streams

The depth data stream is given as an array of size $640 \times 480 = 307200$ with distances in millimeters for each pixel. There is a difference between the Kinect for Xbox 360 and the Kinect for Windows depth stream. With the Kinect for Windows and its Near Mode, we can see objects as close as 40 cm from the sensor. The Kinect for Xbox can only see objects that are further than 80 cm from the sensor.

The video data stream is also given as an array of size $640 \times 480 = 307200$ with the color value for each pixel. We can choose between RGB and YUV for the color value.

We have not used the audio stream in this thesis.

2.4 Possibilities with the Kinect

Two researchers have created a system that uses multiple Kinect depth streams to keep track of objects as they are moved around in a building (4). This may be the end of searching after our misplaced car keys.

A group of independent game makers has created a physical sandbox with an image projected on to the sand from above. The image adds a virtual world to the sandbox by for instance showing snow on high peaks of sand or adding virtual water to lower regions of the sandbox. The water is fully animated and dynamic, meaning the water flows with terrain changes. They have achieved this by using the depth stream from a Kinect mounted above the sandbox (5).

This show what can be done with the Kinect.

3

Contour tracking

Contour tracking will find the contour of objects in range of the depth camera.

This algorithm works by scanning the depth image from the bottom and up for a valid contour pixel. When a valid contour pixel is found it will begin to track the contour of the object that the pixel is a part of. This is done by searching in a local grid around the pixel. After the contour tracking algorithm has terminated the tracking it will return an ordered list with the positions of the contour pixels.

3.1 Existing algorithms

There exists several contour tracing algorithms, such as (6), (7) and (8), but we have chosen to implement our own algorithm.

3.2 Finding the initial pixel

To start the contour tracking we need to find a valid contour pixel. A valid contour pixel is a pixel that is in a specified range from the Kinect sensor and has at least one neighboring pixel that is not in the specified range. The initial pixel is found by scanning the depth image, from bottom and up, after a valid pixel. This means the scan for the initial pixel will have to scan a substantial amount of pixels if the hands are high up on the image. To reduce the time it takes to find the initial pixel we can make an assumption about the hands position. We assume the hand is somewhere in the middle of the picture. With this assumption we can add a height offset to the start height of the scan. In the algorithm we set the height offset to be 20% of the image height. Now we don't have to scan so many pixels and to make it scan even less pixels we only scan every fifth row. The effect can be seen if we have an empty image, i.e no objects are visible to the depth camera. This is our worst case scenario.

The best resolution we can get from the depth camera is 640×480. Worst case number of pixels we have to scan if we don't optimize the scan is 307200 pixels. See equation 3.1.

$$imageWidth \times imageHeight = 640 \times 480 = 307200 \text{ pixels} \tag{3.1}$$

Worst case with the optimized version, as shown in equation 3.2, is 48640 pixels.

$$\left\lfloor \frac{imageHeight \times 0.80}{5} \right\rfloor \times imageWidth = \left\lfloor \frac{480 \times 0.80}{5} \right\rfloor \times 640 = 76 \times 640 = 48640 \text{ pixels} \tag{3.2}$$

This is an 84% reduction of pixels we have to scan.

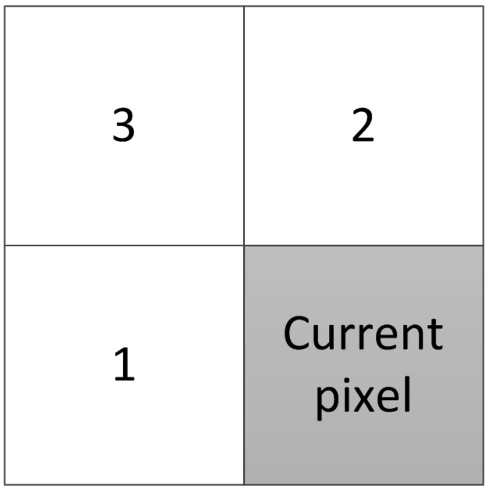
3.3 Tracking the contour

After the initial contour pixel is found we can begin the search for the next contour pixel. The algorithm will search in a local grid pattern with the current found pixel at the origin. The grid extends one pixel in each direction.

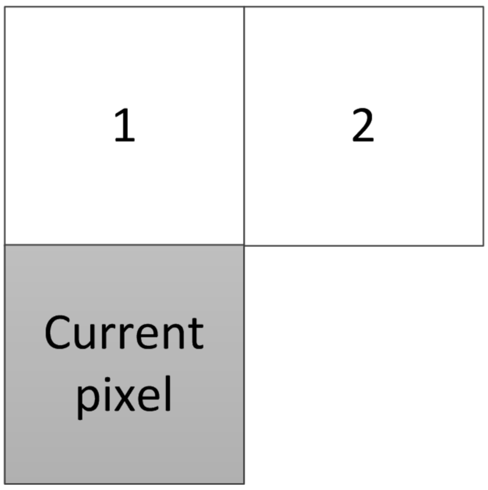
3.3.1 Search directions

We assign a search direction to each found contour pixel to speed up the discovery process. The search directions are up-left, up-right, down-right and down-left; they are relative to the center of the screen. The starting direction is set to up-left due to the V-shape of the hand above the wrist. The search for a new contour pixel will begin in the same direction as the last pixel was found in. If we don't find a new pixel in this direction, we begin searching in the next most probable direction. The most probable direction after up-left is up-right: After we search up along a finger we will hit the beginning of the fingertip; here the contour direction changes to go up-right. At the apex of the fingertip it curves down-right, making this direction the most probable direction after up-right. After we traverse down to a finger valley a new finger begins, making the next most probable direction up-right. If the algorithm doesn't find a new pixel in the next most probable direction, it will search in all the directions, beginning at the last found pixel direction and moving clockwise. The search will stop when a valid pixel is found.

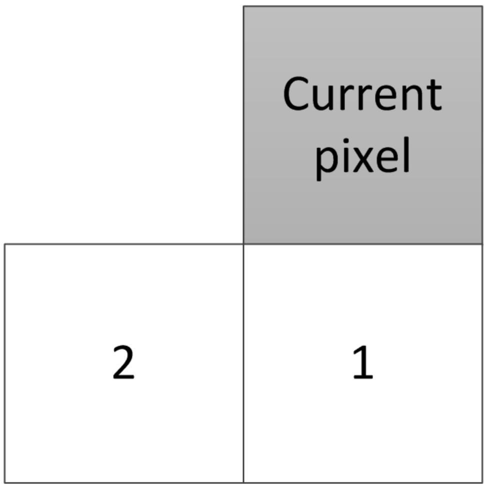
With this method we can run into the problem of discovering a pixel that has already been found. If we found a pixel that has previously been discovered we begin to search from the last found pixel direction, but this time moving counterclockwise. If we still find a previously discovered pixel we check if we are on a single, vertical or horizontal, pixel line. If we are not on a single pixel line, we can assume that there are no more contour pixels to discover and terminate the search.



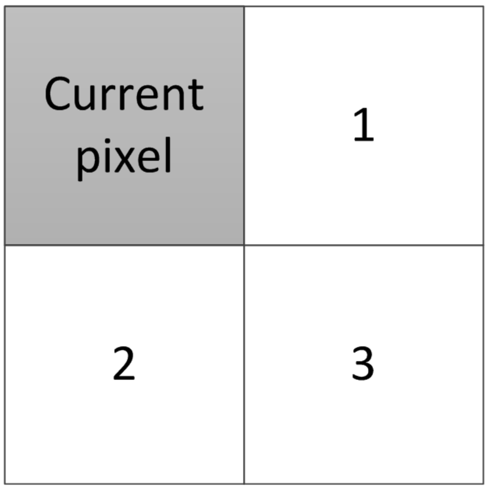
(a) SearchUpLeft algorithm.



(b) SearchUpRight algorithm.



(c) SearchDownLeft algorithm.



(d) SearchDownRight algorithm.

Figure 3.1: Local search grid.

Figure 3.1 shows the local grid pattern used by the contour searching algorithm. Figure 3.1a displays where the algorithm SearchUpLeft will search for new contour pixels. It will first check pixel 1 (in figure 3.1a) if it's a valid contour pixel. If not, the search continues to pixel 2 and then 3. Figure 3.1b, 3.1c and 3.1d also displays pixel search order, where 1 is the first pixel it checks.

In figure 3.1 we can see that the algorithms overlap pixel searching in the vertical direction. With a hand contour we will be mostly searching in the vertical direction. The overlap enables all algorithms to search directly vertically. By overlapping we save time by not changing search directions too often.

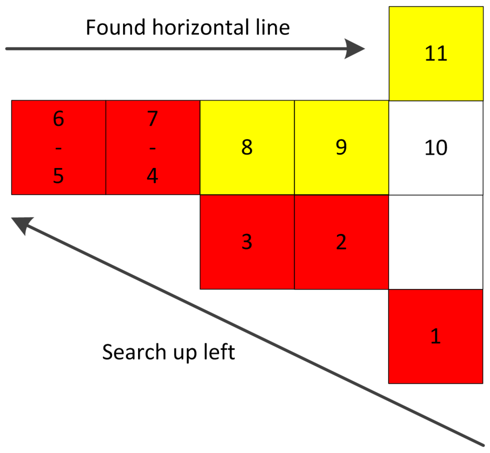
3.3.2 Single pixel lines

One problem with the four search direction algorithms is; If we traverse up a single pixel line we are unable to traverse down again and continue searching. Therefore if we cant find any new pixels using all four search direction algorithms, we must check if we are on a single pixel line. This is done by checking the pixel above and under the current pixel. If both of them are not valid pixels we have a horizontal pixel line. To check if we are on a vertical single pixel line we check the left and right pixels.

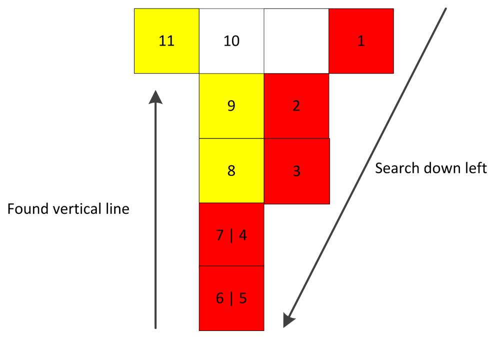
Once we have established that we are on a single pixel line the single pixel line algorithm will traverse down to the base. It will break off the search for the base if we find a valid pixel on the other side of the line from where we came from. It has then successfully found the base. When it is finished it will return a new direction to continue the search.

Including the newly found pixel it will also return a search direction to continue the search in.

Figure 3.2 demonstrates how the single pixel line algorithm works. In 3.2a the SearchUpLeft algorithm finds contour pixels 1 to 5. After it has found pixel 5 it will search in the next most probable search direction, up-right. This leads to finding pixel 4, which has already been discovered. It will then try to find a new pixel by searching counterclockwise. This leads to finding pixel 4, again. It will now find if we are on a horizontal or vertical single pixel line by checking which neighbor pixel is a valid pixel. In this case we find that the right pixel is our valid neighbor, this means we are on a horizontal line. The algorithm also checks which general direction we are searching in;



(a) Horizontal single pixel line.



(b) Vertical single pixel line.

Figure 3.2: Single pixel lines.

These are up, down, left and right, and determines in which direction we must traverse to find the base.

In figure 3.2a the single pixel line algorithm will traverse to the right, discovering contour pixels 7, 8 and 9. Note that the algorithm will not break the search when we find pixel 8 because it knows the general search direction is up and will therefore only break if there is a valid pixel above the pixel line. When it is on pixel 10, it will discover pixel 11 and stop; this is the end of the line. The algorithm will return a new search direction, up-right since the general search direction were up and right, and continue local grid search from this direction.

In figure 3.2b we have a vertical single pixel line. The only differences here are the searching directions. When the algorithm is finished the new searching direction will be down left since the general directions were down and left.

3.3.3 Backtracking

We also have a backtracking algorithm to ensure continuous contour tracking even if we encounter unknown valid pixel configuration that would halt the tracking. The backtracking algorithm will backtrack pixels up to a user specified pixel count. This algorithm utilizes the search clockwise algorithm with some additions; It will continue until it finds a new valid contour pixel or until it has reach its specified pixel count limit. If no new valid contour pixel is found, then we can say the contour tracking is finished. If the backtracking finds a new contour pixel, it will return the newly found pixel with the search direction it was found in. This enables us to continue normal contour tracking from here.

Backtracking is used as a last resort to find new contour pixels. This algorithm could also substitute the find single pixel lines algorithm (chapter 3.3.2), although it would not perform as well.

3.3.4 Track two hands

For the contour tracking to work with two hands in the depth frame simultaneously, we have to do two separate searches for the initial pixel. First we follow the steps described in section 3.2 to find the initial pixel and track the contour of the left hand. If it did not find any pixels we know there aren't any objects in the frame and don't need to continue. If one or more valid pixels were discovered we continue to search for a second hand.

The initial pixel scan for the second hand has some additional properties. Since we started the scan from the left side it is reasonable to assume the discovered hand is on the left side of the depth frame. With this in mind, it will be faster to detect the right hand if we start the second scan from the right side. There is still the possibility to find one of the contour pixels of the left hand. To avoid detecting the contour of the same hand twice we check if the pixel has already been discovered. If the pixel is part of the first hand we know the second hand must be higher up on the depth frame. The algorithm will jump 20 pixel rows up and continue the scan from there.

If the pixel has not previously been discovered it must be a contour pixel on the right hand. This pixel will be our second initial pixel and the contour tracking can begin on the second hand.

3.4 Termination states

The algorithm has two termination states; if it finds a pixel that has already been discovered or a fixed number of pixels have been discovered.

3.4.1 Termination state 1: A pixel was discovered twice.

We can make two assumptions if we find a pixel that has previously been discovered. The first assumption we make is that the algorithm has tracked the whole contour and found the initial pixel that began the tracking. If this is the case we can say the tracking was successful and the whole contour was found. The other assumption is that the algorithm is stuck and can't find any new contour pixels in its local search grid. It can become stuck if the pixel configuration that make up the contour has undiscovered edge cases that were not accounted for. Although this is unlikely, there is a possibility for this to occur.

To provide a fast and easy method to check if a pixel has previously been discovered, we also store all discovered pixels in a hash table. Hash tables has $O(1)$ lookup and insertion time and is an unordered collection; making it good for duplicate checking but unusable to use as the primary collection of discovered pixels since we need to have them sorted in the order they were found.

3.4.2 Termination state 2: A fixed number of pixels were discovered.

Since we only are interested to find the contour around the fingers, we only need to discover a limited number of pixels. At the closest detection range of the Kinect for Xbox sensor all fingers is made up of about 450 pixels total, see figure 3.3. If we were to move further away from the Kinect sensor this value would decrease. With this information we can stop the tracking algorithm after a fixed number of pixels are discovered.

There is also a risk with this method. The tracking could be terminated too early, before finding all the finger contours. Other body parts could also be included in the picture, such as the forearm, using most of the available pixels. The algorithm is set to discover max 700 pixels per hand.

The termination value is also important to the performance of the algorithm. The more pixels it need to discover the more time the algorithm will use. By limiting the number of pixels the algorithm can discover we can calculate the worst case running time. This way we can make adjustments to meet the maximum expected running time.

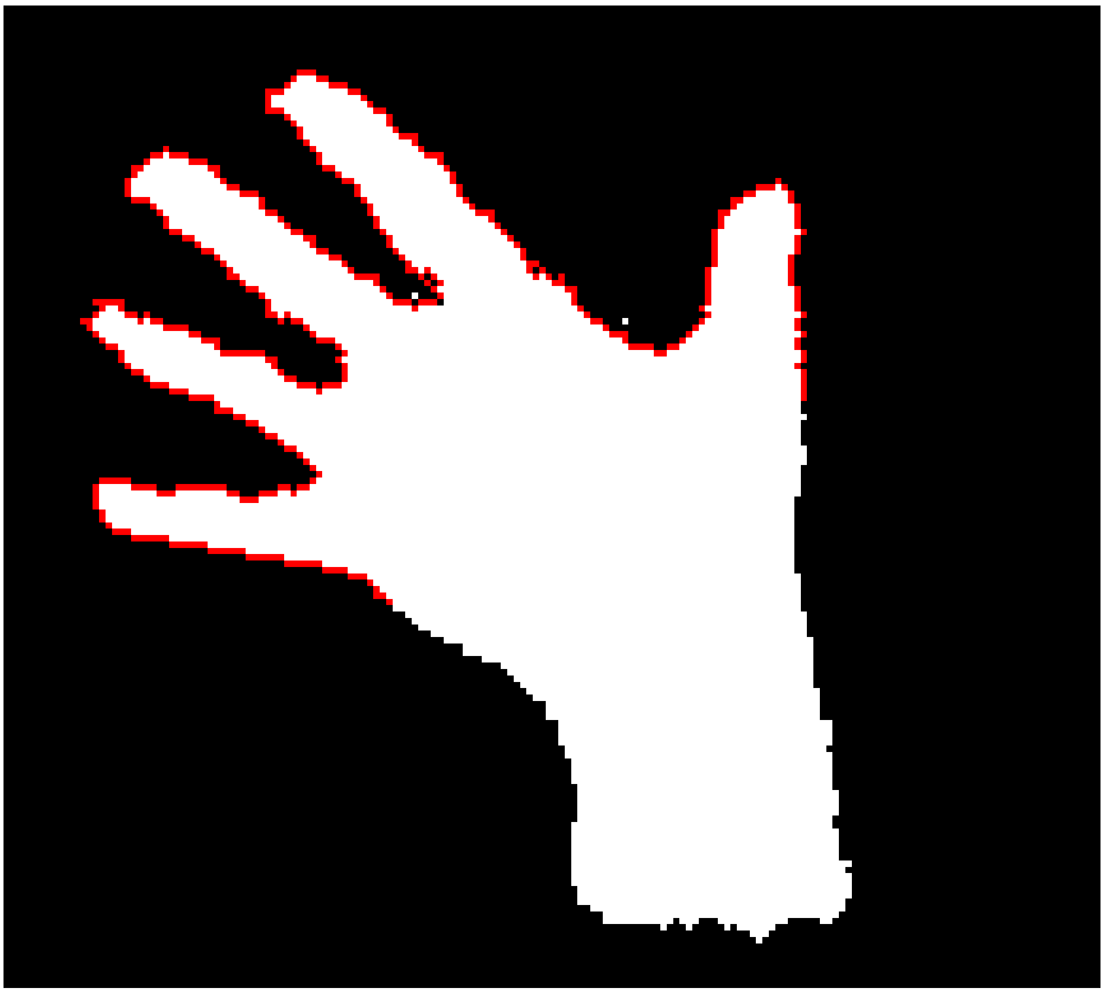
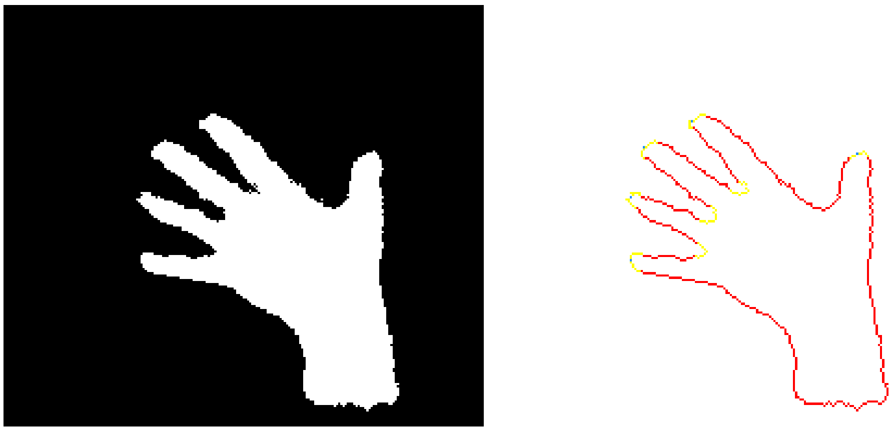


Figure 3.3: Minimum hand contour pixels (the red colored pixels).

4

Finger recognition

Finger recognition consists of three steps. Step one is to find curves on the hand contour. Step two is to find which curves are fingertips and the last step is to find the middle of the fingertip curves. In addition we will also get the pointing direction of the fingertips. Figure 4.1b shows the results of these algorithms. The red pixels are the extracted hand contour, the yellow pixels are the curve points and the blue pixels indicate where the fingertip are located.



(a) Raw depth image.

(b) Hand contour with curve points and fingertip location.

Figure 4.1: The raw and processed images.

4.1 Related works

A group at Massachusetts Institute of Technology (MIT) has developed an open source graphical interface with the use of finger tracking with the Kinect. They have however used a different approach to the finger tracking problem; They use point clouds to find the hand and fingers (9).

In the paper (10) they use the k-curvature algorithm to find curves. They have used this approach to find the fingertip from binary images. We have used the same approach to curve detection in this thesis.

4.2 Curve detection

The curve detection is implemented using the k-curvature algorithm. The k-curvature algorithm detects the angle between two vectors.

The implemented version of the algorithm takes in three parameters; An ordered list of contour points, a constant k and an angle ω, in radians. The constant k and the angle ω values are application specific, k was found by trail and error. In the application k is set to 20 and ω to 0.95993 radians, or 55 degrees. The angle ω was found by measuring fingertip angles in depth frames. The average fingertip angle was measured to 39 degrees and the lowest fingertip angle was measured to 25 degrees.

The algorithm works by creating two vector at each contour point. One vector, vector $\vec{a}$, points to a contour point k points in front of the current point in the list. The other vector, vector $\vec{b}$, points to a contour point k points behind the current point. If the contour point list is cyclic, i.e. the contour closes on itself, we can create vectors across the start and end boundary of the list. If the list is not cyclic we set $\vec{a}$ to point to the first contour point in the list when the index of the current point is less than k. We must also do the same at the end; We set $\vec{b}$ to point to the last contour point if we are less than k points from the end of the list. See algorithm A.0.1 in the appendix. We also create a third vector, $\vec{c}$, between $\vec{a}$ and $\vec{b}$.

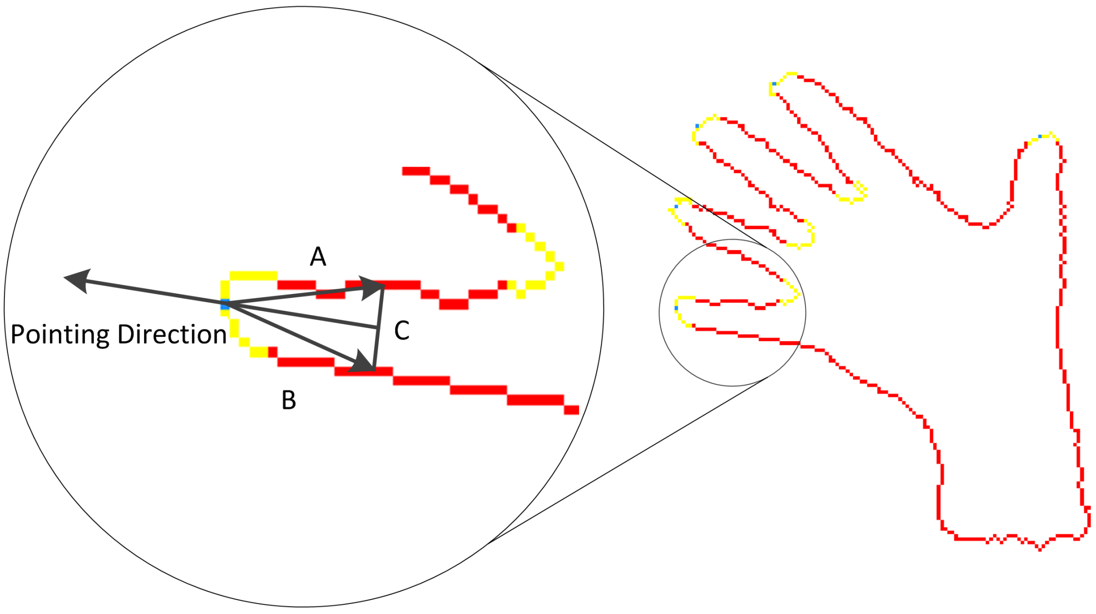
After the vectors are created we need to find the angle between $\vec{a}$ and $\vec{b}$. If this angle is less than ω we have a curve point. The screen coordinates of the current contour point and the three vectors, $\vec{a}$, $\vec{b}$ and $\vec{c}$ are stored in a list.

4.3 Finger detection

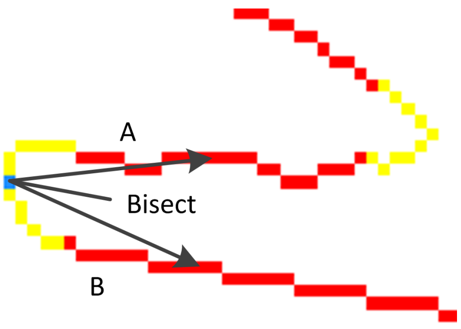
To find the fingertips we iterate through the curve point list and try to find curve point segments. Curve point segments consists of points that are next to each other.

When the start and end point of a curve segment is found we find the middle point of the segment. This will be the fingertip location. However, not all segments are fingertips, they can also be finger valleys. To find if the segment is a fingertip we create a bisect between $\vec{a}$ and $\vec{b}$. If the bisect points to a pixel that is in the specified depth range we know that it must be a fingertip otherwise it is a finger valley, see figure 4.2b and 4.2c.

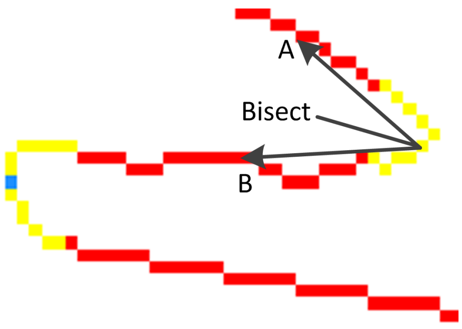
We use $\vec{c}$ to find the pointing direction of the finger. We create a new vector between the curve point and the middle of $\vec{c}$ and reverse the vector. This vector will be the pointing direction of the finger. See figure 4.2a. The curve point and the pointing direction is stored in a list.



(a) Fingertip direction.



(b) Fingertip with bisect.



(c) Finger valley with bisect.

Figure 4.2: Finding the fingertips and their pointing direction.

5 Gesture Recognition

To recognize gestures, we have implemented a variant of the dynamic time warping algorithm - DTW.

The DTW algorithm recognizes similarities between two time series. The two time series do not need to be synchronized in time, enabling a user to do gestures slower or faster than the recorded gesture speed.

5.1 Related works

A group of researchers at Microsoft Research used dynamic time warping to recognize dance gestures using the Kinect. Their gesture classifier has an average accurace of 96.9%. This shows that DTW can be used to achieve high accuracy with data from the Kinect (11).

5.2 Note

From here on a frame implies that we have processed the depth frame and found the finger positions in the frame.

When we say cost, we mean the total euclidean distance in one frame. See equation 5.2 for calculating total euclidean distance between two frames. In equation 5.2 we assume only one hand will be used for gestures, although gestures with multiple hands are possible with our solution. We also assume that the finger count n throughout a gesture is $1 \leq n \leq 5$.

In equation 5.1, p and q denotes a finger position in a reference gesture frame and in a input gesture frame.

5.3 Recording a gesture

To record gestures we store finger positions and direction from all frames within a user specified time frame or until a max frame count has been reached. When a user starts recording a gesture, finger positions and direction from each frame is stored in a queue collection. A queue collection was chosen for the removal speed of the first inserted data, which is $O(1)$. Fast removal speed is needed if a maximum frame count is set together with a time limit, where in the time limit a greater number of frames will be produced than the maximum frame count number. If this should happen, the oldest frame will be removed before inserting the new frame. This can be seen as a safety property to avoid using too much memory if the time limit is too large.

A likely scenario where this could happen is when the recording is set to start and end by a button click. If the button for stopping the recording is never pressed, we have a way to prevent memory leak.

After the recording time has passed, all the data from the queue collection is extracted and stored in an array in its own object.

5.4 Recognizing a gesture

We have divided the DTW recognizing algorithm down to several steps for easy understanding and explanation.

5.4.1 Step 1: Finding a candidate gesture

We find the cost (euclidean distance) of the current frame and compare it to the last frame from all the gestures. The cost indicates how similar the two frames are. If the two frames are identical, the cost will be zero. The more difference between the frames the higher the cost will be.

To find a candidate gesture we calculate the cost of the current frame and the last frame from all the gestures. The lowest of the calculated costs will be our candidate gesture for full DTW processing. However, if the lowest cost is too great, we don't select any candidate. As the gesture might be the closest match but the cost difference is too high.

This is an performance optimization so we don't need to do full DTW calculations that would yield a not recognized gesture.

5.4.2 Step 2: Euclidean distance matrix

We now have our two gestures, a prerecorded gesture (reference gesture) and the newly performed gesture (input gesture), to do DTW calculations. The first thing we have to do is to calculate the cost between each reference and input frames. This can be visualized by using a matrix, see figure 5.1a.

The matrix will show how similar each frame is to another between the two gestures.

$$\text{Euclidean distance} = d(p,q) = \sqrt{(p_x - q_x)^2 + (p_y - q_y)^2 + (p_z - q_z)^2} \tag{5.1}$$

$$\text{Total euclidean distance} = \sum_{i=1}^{5} d(p_i, q_i) \tag{5.2}$$

5.4.3 Step 3: Accumulated distance matrix

After the matrix is filled with the cost, we can begin computing the lowest accumulated cost matrix. In this matrix we compute the lowest cost to reach a cell. There are three different ways we can reach a cell; From the left, bottom or the diagonal down cell, see figure 5.1b. In figure 5.1b we see that c can only be reached by cell 1, 2 and 3. The accumulated cost for these cells must be calculated before we can calculate it for c.

The cost to reach a cell c is the accumulated cost of the one of the three cells that can reach c plus the existing cost in c (calculated in section 5.4.2). The lowest of the three calculated accumulated cost for c gives us the final accumulated cost for c, see equation 5.3. We have to do this for all the cells, except for cell [0,0] which we will explain later.

In figure 5.3, c is the cell we want to calculate the accumulated cost (ac) for and $c^{'}$ is one of the three cells that can reach c. The lowest of the three c_{ac} will be chosen as the final value for c.

The left most column and the bottom row must have their accumulated cost calculated first. These cells can only be reached from the bellow cell and the left cell, respectively, and thus only depend on cell [0,0] to have an initial accumulated cost. The accumulated cost of cell [0,0] is set to zero since it is not reachable by any cell.

$$c_{ac} = c'_{ac} + c_{cost} \tag{5.3}$$

5.4.4 Step 4: Lowest distance path

We now need to find the lowest accumulated cost path from the last cell ($[m, n]$) to the first cell ([0,0]). Beginning at the last cell, we always choose the cheapest cell of the three we can choose from (left, down and the diagonal down cell) as the next cell. The accumulated cost for each cell in the path is added together to be the total path cost.

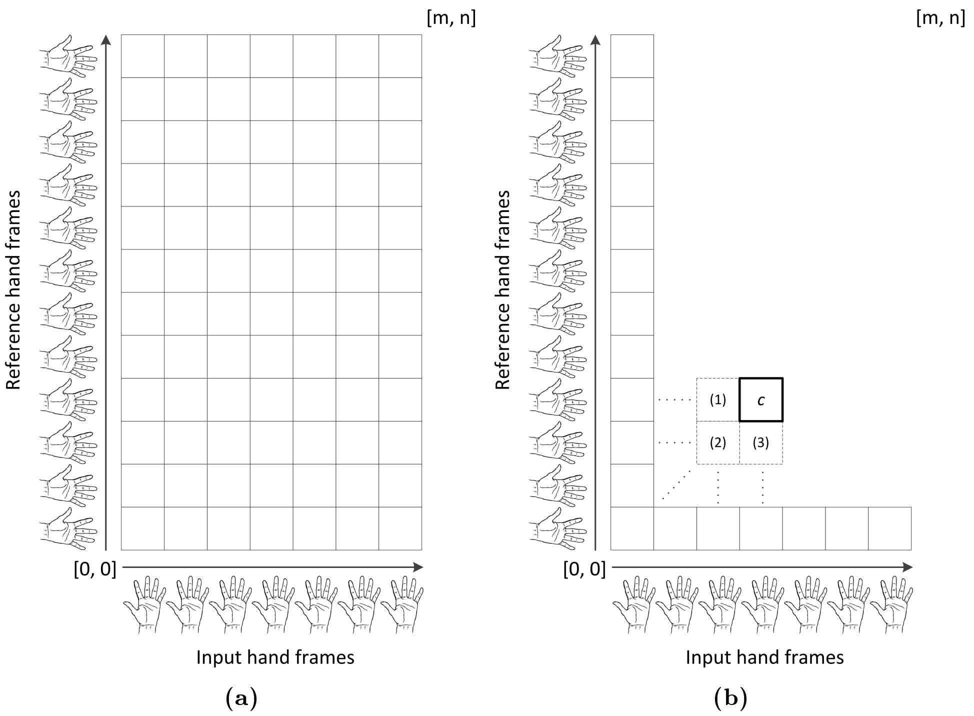


Figure 5.1: Hand frame matrix.

5.4.5 Optimizations and improvements

To get a more desirable result we can apply some optimizations to the steps above.

5.4.5.1 Euclidean distance

To make movement along a certain axis more important than other, we can apply weighing to the euclidean distance calculation. We modify equation 5.1 by multiplying each axis with a weight $w_{\{x,y,z\}}$, where $0 \leq w_{\{x,y,z\}} \leq 1$. Equation 5.4 shows the modified euclidean distance with weighting.

$$\text{Weighted euclidean distance} = d_w(p,q) = \sqrt{w_x(p_x - q_x)^2 + w_y(p_y - q_y)^2 + w_z(p_z - q_z)^2} \tag{5.4}$$

With this improvement we can better recognize gestures that do not rely on all three axis. Such a gesture could be a swiping gesture from left to right as this gesture would mainly use the x-axis. It is therefore desirable to let the other axis have a lower impact on the cost.

A weighting value of zero on a axis would be to not take the axis into consideration when calculating the cost.

5.4.5.2 Accumulated cost matrix

Since our goal is to find the lowest accumulated cost path and optimally that would be a straight diagonal line from $[m, n]$ to [0,0], we can do a small optimization to the diagonal cost in section 5.4.3. Instead of adding the cost of c to the diagonal cells accumulated cost, we add $\frac{c}{2}$, thus making going diagonal more desirable. See equation 5.5.

$$c_{ac} = c^d_{ac} + \frac{c_{cost}}{2} \tag{5.5}$$

To make things clear; we use equation 5.3 to calculate the accumulated cost to c from the left and bellow cell and equation 5.5 for the diagonal down cell.

To generalize the equations we can use a weighting factor w for each direction e.g w_h for horizontal movement (left cell), w_v for vertical movement (below cell) and w_d for diagonal movement (diagonal down cell). This gives us equation 5.6.

$$c_{ac} = c^{'}_{ac} + w^{'} c_{cost} \tag{5.6}$$

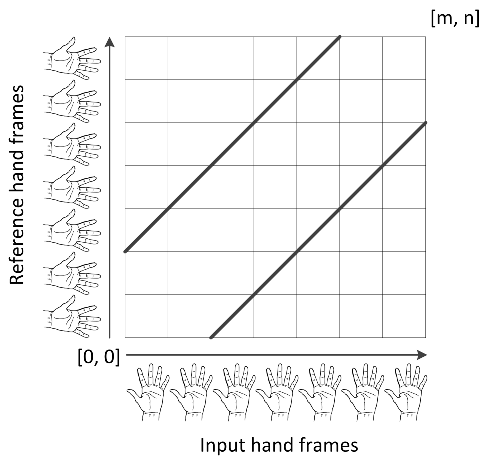
To use equation 5.6 to get a bonus to the diagonal cell, as shown in equation 5.5, and no bonus to the left or bottom cell, we set $w_{\{h,v,d\}}$ to $w = \{1, 1, 0.5\}$.

5.4.5.3 Lowest distance path

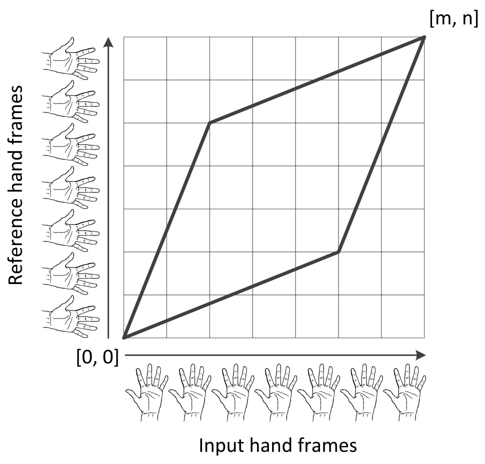
We can optimize the path finding by constructing some constraints, such as the Sakoe-Chiba band (12) (figure 5.2a) or the Itakura parallelogram (13) (figure 5.2b).

Constraints as those will force the warping path to not deviate too much from the straight diagonal path. The path finding can also be aborted should it go beyond the constraints. A problem can arise if the optimal path has some cells outside of the constraints. This would lead to a higher total path cost that could impact our decision if the gesture should be seen as recognized.

Generally constraints will speed up the DTW algorithm since we do not need to calculate the cells outside of the constraint. This could lead to a significant performance gain.



(a) Sakoe-Chiba band



(b) Itakura parallelogram

Figure 5.2: Warping Constraints.

5.5 Store gestures

The recorded gestures can also be stored to disk. It is saved as a XML-file to make it available to be used in other applications or to be read by humans. The XML-file will contain finger positions and direction for each frame in a gesture. One file can contain multiple gestures.

5.6 Retrieve stored gestures

Since we can store gestures to disk, we can also retrieve them. It will read a XML-file with finger positions and direction and add each gesture from the file to the list of active gestures to be recognized.

6

Auto depth interval detection

The auto depth range calibration will create a depth interval that is limited by two hands. When the depth interval is found it will discard all data outside this range, only allowing objects to be detected in the newly found depth interval. The algorithm works by detecting the fingers of the hand closest to the sensor and then detecting the fingers of the second hand; which is further away from the sensor. The depth interval is set from the position of the first hand to the second hand. After the scan is complete we can freely move our hands in the depth interval.

The algorithm works by detecting how many fingers were found at a determined depth interval. It will start the scan from the minimum possible distance from the sensor. In our case with the Kinect for Xbox sensor it is at 800 millimeters. The first pass of the auto scan will start at 800 millimeters with an interval, b, of 150 millimeters. The interval value is based on how wide a hand is from the tip of the thumb to the tip of the little finger when all the fingers are spread out. We need the interval to be at least this wide because the hand could be rotated around the y-axis and we need to be able to detect all the fingers in at least one pass. At the subsequent passes the interval will begin at $prevStart + \frac{b}{2}$ millimeters and end at $prevStart + \frac{b}{2} \times 3$, where $prevStart$ is the previous start depth. This will overlap the previous interval, allowing us to correctly detect the number of fingers if the hand should be split in half at the end depth.

When we correctly find the first hand we set the final start depth p to be the start depth of the current pass. Then in the next pass the start depth will be $p + b$, this is to avoid detecting the found fingers twice. When we then correctly find the second hand we set the final end depth q to be the end depth of the current pass. The algorithm will return p and q, which in turn will be used as the depth interval for the range finder.

If only one hand was found, or none at all, it will return default values for p and q. The default start depth is the minimum detection distance of the sensor. The default end depth is the default start depth $+\ b$.

We encounter some problems if we go beyond 1500 millimeters from the Kinect for Xbox sensor. At this range there will be a significant amount of artifacts around objects. This leads to poor finger detection quality and that will impact the ability to correctly find the depth interval. To make the algorithm more flexible we only require four fingers to be detected for a hand to be correctly detected. This will give a higher rate of correct detections.

There is also support to use only one hand to create a depth interval. After it correctly detects the closest hand it will set the final start depth to the start value of the current pass. The final end depth is then the start depth $+\ u$, where u is a user specified value in millimeters.

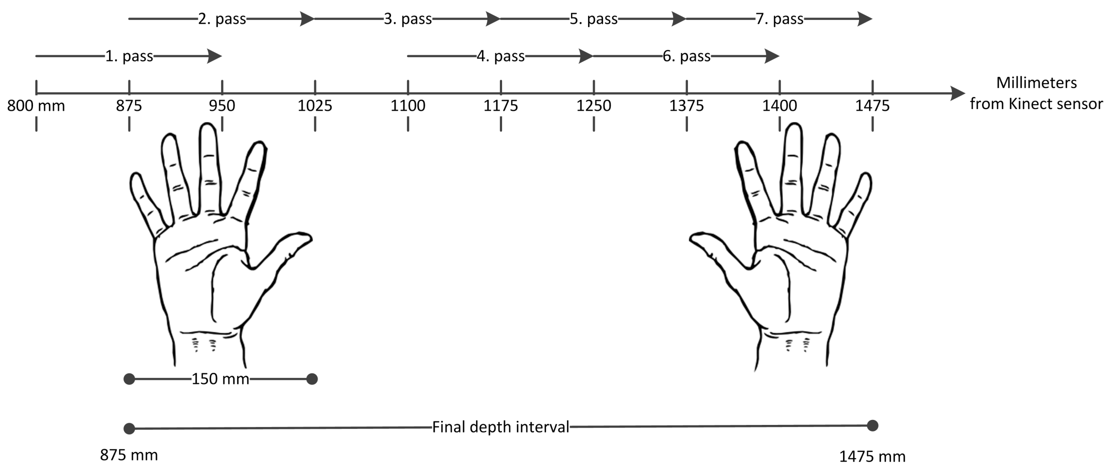


Figure 6.1: Example of a scan sequence.

In figure 6.1 we see an example of how the auto scanning would find the depth interval between two hands. The first hand is found in the second pass and we set the final start depth to the current pass start depth, $p = 875mm$. At the next pass we skip the overlap to avoid the possibility of detecting the first hand again. In the seventh pass we find the second hand and set the final end depth to the current end depth, $q = 1475mm$. The algorithm is now finished and we will only be able to detect objects that are in the interval p to q. Figure 6.2 shows the actual depth data from the scan. The first hand is found in 6.2b and the second hand in 6.2g.

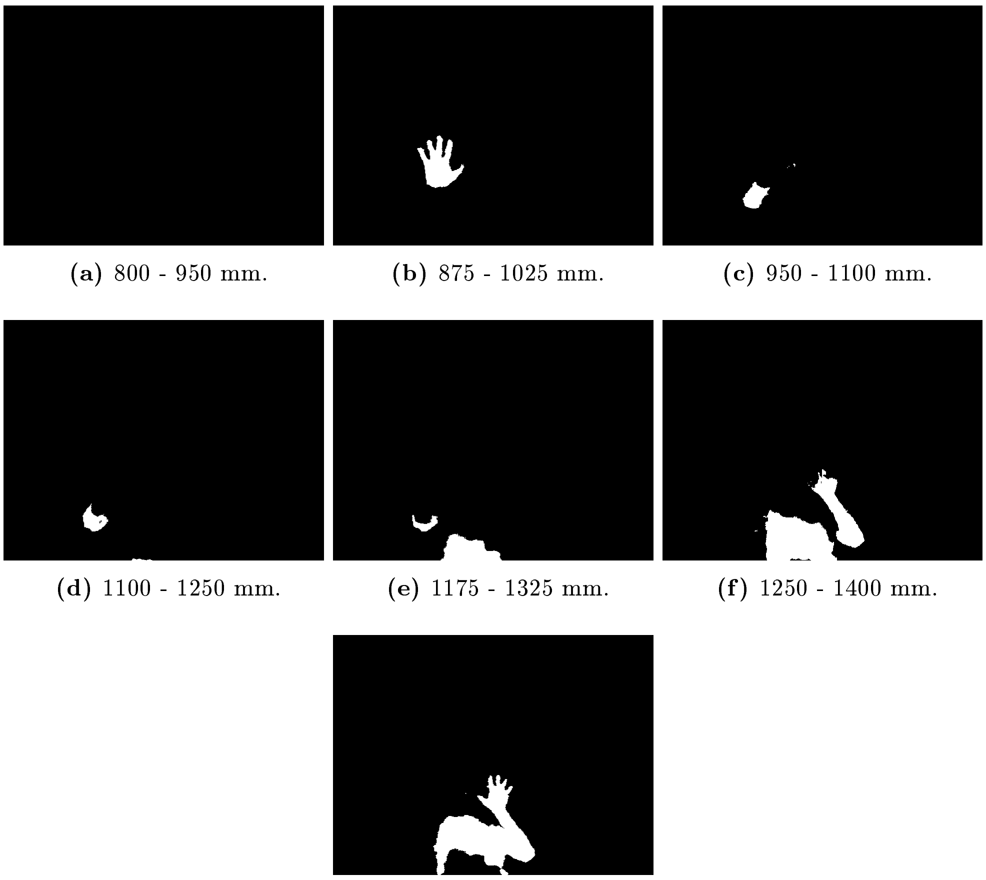

(a) 800 - 950 mm.
(b) 875 - 1025 mm.
(c) 950 - 1100 mm.
(d) 1100 - 1250 mm.
(e) 1175 - 1325 mm.
(f) 1250 - 1400 mm.
(g) 1325 - 1475 mm.

Figure 6.2: Depth frames from an auto interval scan.

7

Enhancements

The depth images from the Kinect produces jitter and artifacts. This makes the result from contour tracking unpredictable. This effect is cascading, meaning other algorithms depending on contour data will also be negatively affected.

To counter these effects we have created some algorithms that will help give more consistent results.

7.1 Smoothing

To make the fingers position more consistent when jitter occurs, we can apply a smoothing algorithm to the fingers positions. A smoothing algorithm reduces the change between the new and old point.

$$p^{'} = q + s(p - q) \tag{7.1}$$

We have implemented exponential smoothing, as shown in equation 7.1. In equation 7.1 we have the current point p, the previous point q, the new point with smoothing applied $p^{'}$ and the smoothing factor s, where $0 \leq s \leq 1$.

7.2 Prediction

We have also implemented an algorithm that enables us to predict where a fingertip is going to be in the next frame. The algorithm detects if one or more finger positions has been missing in up to n frames, where n is a small number, and replaces them with a predicted finger position.

7.2.1 Exponential moving average

We use exponential moving average (EMA) to predict where a finger is going to be in the next frame. The equation is shown in equation 7.2, where t is the frame number (time period), y is the observation, s is the prediction and w is the weight factor $(0 < w < 1)$.

$$s_t = wy_t + (1 - w)s_{t-1} \tag{7.2}$$

The EMA algorithm needs a set of values to produce a good prediction. Every correct frame is stored in a queue collection that holds 30 frames. When the queue is saturated and an inconsistent finger count has been detected, all the frames in the queue is passed to the EMA algorithm. It will then produce the next frame and use those values for the missing fingers.

All the correct frames is also passed to the gesture algorithm. This ensures the consistent finger count needed for gesture recording and recognition. The gesture recognition will however lag up to n frames.

8

Virtual reality

The virtual reality we have created attempts to use the screen as a physical window to a simulated environment. This chapter gives an basic overview on how we have achieved this. More in depth explanation can be found in (14) and (15).

8.1 Related works

Johnny C. Lee brought the idea of virtual reality with head tracking and cheap consumer hardware to the masses with a video he posted on YouTube in 2007. He used the Nitendo Wii remote and a head mounted sensor to track the users head position. The perspective on the screen changes with head movement to create a virtual environment inside the screen (16).

8.2 Introduction

Our virtual reality works by creating an off-axis perspective that follows the head movement of a user.

8.2.1 On-axis perspective

On-axis perspective is the perspective most of us is know with. It is the perspective used in most 3D computer and console games. This perspective is shown in figure 8.1a. There the users eyes, or the camera origin, p are always centered to the screen. The view frustums size does not change if we move the virtual camera around.

To visualize on-axis perspective we can think of us looking through a window. We have our head centered and perpendicular to the center of the window. Now if we want to view something else, that is outside of the view space we can see through the window, we must also move the window to keep our head centered and perpendicular to it.

8.2.2 Off-axis perspective

Off-axis perspective is when the eye position does not need to be at the center of the screen. This perspective is shown in figure 8.1b. There we can see that the users eye position, or the camera origin, p does not need to be centered to the screen. In this perspective the size of the view frustum changes depending on the users head position.

To visualize off-axis perspective we can think of us looking through a window. Now the window is in a fixed position and does not move. If we want to see something outside of the initial view space through the windows, we need only to move our head.

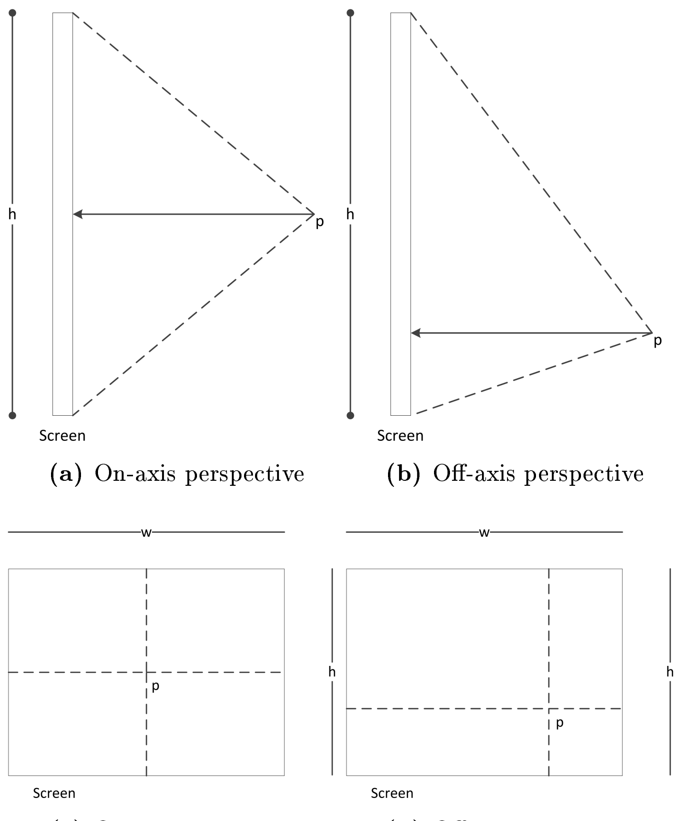
8.3 View frustum

In off-axis perspective the view frustum size changes depending where we have our head relative to the screen.

The major graphics library DirectX has built in support to create off-axis perspective. This means that we don't have to manually create the projection matrix needed. As XNA uses DirectX we have an easy way to create the matrix (17).

To create the matrix we need to determine the view frustum extents. The extents that needs to be calculated is seen in figure 8.2. There t, b, r and l are the top, bottom, right and left extents, respectively. In addition we also need to choose a near and far view plane.

The extents are calculated in equation 8.1. Where r is the aspect ratio of the screen, v is the near view plane and p is the position of the users head. This equation is the



(a) On-axis perspective **(b)** Off-axis perspective

(c) On-axis perspective **(d)** Off-axis perspective

Figure 8.1: Perspective.

same used by (2).

$$\begin{aligned}
\text{top} &= \frac{v(^1/_2 - p_y)}{p_z} \\
\text{bottom} &= \frac{v(-^1/_2 - p_y)}{p_z} \\
\text{right} &= \frac{v(^1/_2 r - p_x)}{p_z} \\
\text{left} &= \frac{v(-^1/_2 r - p_x)}{p_z}
\end{aligned} \tag{8.1}$$

Figure 8.3 displays an top down view of a user and a screen. We can see that the screen works like a physical window into the virtual environment.

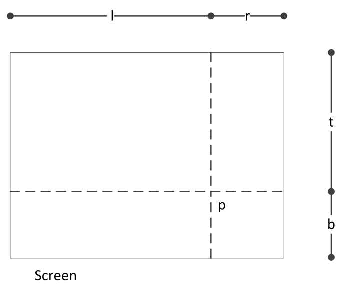


Figure 8.2: Frustum extents.

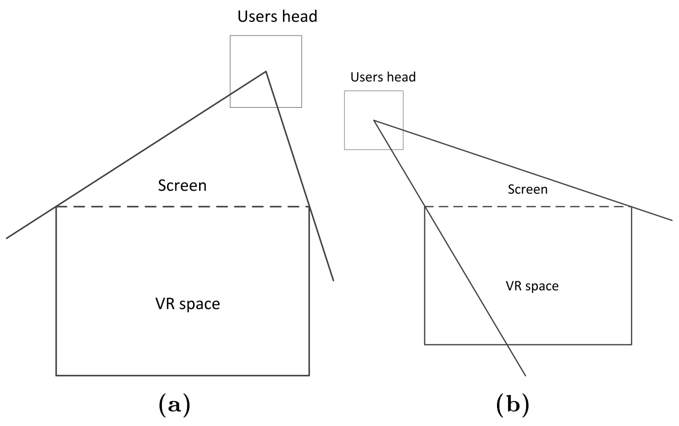


Figure 8.3: VR space.

8.4 Implementation

The head tracking is performed by the Kinect and the perspective camera is implemented as a XNA game component. XNA was used since it simplifies the implementation and allows the code to be used on different .NET platforms, such as the Xbox 360 and Windows Phone.

VR can be used in combination with gesture recognition.

8.5 Limitations

A drawback with creating virtual reality this way is that it can only be used by one user at a time. If someone else watches the screen, it will look weird as the objects appear skewed.

9

Results

The result is an easy to use and easy to understand API. A modular programming technique with the use of interfaces was used to make the API to be flexible and easy to use (18). Code examples in this chapter uses C# syntax.

9.1 Program flow

An overview of the implemented application flow can be seen in appendix B.2.

9.2 Events

The API is event-driven. This gives us complete control over when we want the algorithms to run and what we want to do after it has finished its task. This also matches how we are notified of new data from the Kinect with the Kinect for Windows SDK.

However, the API does not force a user to use events. All methods that starts processing algorithms will also return the result. I.e instead of using the event triggered to get contour data when contour tracking is finished, a user can also get it directly from the method call: $contourData = IContourTracking.StartTracking(...)$.

9.2.1 DepthDistanceUpdated

The first event we use is from the Kinect SDK, *DepthFrameReady*. This fires every time a new depth frame is produced from the Kinect; A new frame is produced every 33 milliseconds. From there we get the width and height of the depth image and filter the received depth data to only contain the depth for each pixel. After its done we fire our own event, *DepthDistanceUpdated*, to indicate new depth data is available. *DepthDistanceUpdated* takes an array of depth distances and the width and height of the depth image as parameters. See code block 9.1.

When *DepthDistanceUpdated* is triggered, we filter the depth data to only contain data in a specified depth interval and begin contour tracking. The depth filter (*RangeFinder.PixelsInRange*(...)) returns an array with a status for each pixel, telling if a pixel is in the range interval or not. When the application is running we should position our hands in the depth interval.

```
void DepthDistanceUpdated(depthDistanceData, width, height)
{
  // Filter depth data to only contain data in a specified depth interval.
  var pixelsInRange = RangeFinder.PixelsInRange(depthDistanceData,
      minDistance, maxDistance);

  IContourTracking.StartTracking(pixelsInRange, width, height);
}
```

Code block 9.1: DepthDistanceUpdated event.

9.2.2 ContourDataReady

The event *ContourDataReady* is fired when contour tracking is completed; It has the contour points and the filtered depth image as parameters. The depth image is not altered during contour tracking.

We perform curve and finger detection when *ContourDataReady* is triggered. See code block 9.2. *ICurveDetection.FindCurves*(*contourPoints*) has an event that fires when curve detection has finished, but we chose not to use it here. The code looks cleaner when we use the direct reference. *ICurveDetection.FindCurves*(...) returns a list with curve points and *IFingerRecognition.FindFingertipLocations*(...) returns a list of *Fingertip* objects. The *Fingertip* object contains the fingertip position and pointing direction of the finger.

If we want to use the auto interval scan, the code for it should run when this event is triggered as it uses the fingertip count to determine when the interval starts and ends. See code block B.1 in the appendix for how this can be done.

```
void ContourDataReady(contourPoints, pixles)
{
  var curves = ICurveDetection.FindCurves(contourPoints);

  IFingerRecognition.FindFingertipLocations(curves, pixels, width, height);
}
```

Code block 9.2: ContourDataReady event.

9.2.3 FingertipLocationsReady

The event FingertipLocationsReady is fired when the fingertip detection algorithm has finished. It has a list of fingertips as parameter. See code block 9.3. When the event is triggered the fingertips are converted to a $Hand$ object. By converting it to a $Hand$ object, we do additional processing within the object. This could be to determine exact which fingers are shown from the fingertip points.

Here we can also try to enhance the result by enabling smoothing or the prediction algorithm. If $preventHandInconsitencies$ is set to $false$ and the gesture recognition or recording is enabled, we feed the $Hand$ object to the $IGestureRecognition.AnalyzeFrame$(...) method. The method will analyze the frame and if it finds a gesture match it will fire the event *GestureRecognized*. If we have successfully recorded a gesture the *GestureRecorded* event will fire.

If $preventHandInconsitencies$ is set to $true$, the prediction module will handle the passing of frames to the gesture recognizer. The prediction module has a reference to the gesture recognizer.

9.2.4 GestureRecognized

The event GestureRecognized is triggered when a gesture has been recognized. It has the recognized gesture as parameter. See code block 9.4.

9.2.5 GestureRecorded

When a gesture has been successfully recorded, the *GestureRecorded* event is triggered. It has the recorded gesture as parameter. Here we can, if we wish, manipulate the *recordedGesture* object before storing it, such as giving it a name. See code block 9.5.

```
void FingertipLocationsReady(fingertips)
{
  // Convert fingertips to a Hand object.
  var currentHand = new Hand(fingertips);

  if (EnableSmoothing)
    currentHand = Smoothing.ExponentialSmoothing(currentHand, prevHand,
        smoothingFactor);

  // This is the prediction algorithm.
  if(preventHandInconsitencies)
    HandEnhancements.PreventHandIncosistency(currentHand);
  else if (IGestureRecognition.Recognizing || IGestureRecognition.Recording)
    IGestureRecognition.AnalyzeFrame(currentHand);

  prevHand = currentHand;
}
```

Code block 9.3: FingertipLocationsReady event.

```
void GestureRecognized(recognizedGesture)
{
  // Here goes code that act upon the recognized gesture.
}
```

Code block 9.4: GestureRecognized event.

9.2.6 Performance

There is no noticeable performance gain or loss by using events over direct reference calls (19).

9.3 Modularity

One of the key elements to making the API easy to use is to divide it into modules. The functionality of the module should, preferably, be accessed with only one line of code. I.e in code block 9.2 to find the fingertips, we only need to call one method, *FindFingertipLocations*, in the *IFingerRecognition* module to get the fingertip locations from a list of contour points. Thus all the complexity are inside the module, hidden from the user. This makes the code look clean and also makes it easy to work

```
void GestureRecorded(recordedGesture)
{
  IGestureRecognition.StoreGesture(recordedGesture);
}
```

Code block 9.5: GestureRecorded event.

with.

9.3.1 Interfaces

The modules implements interfaces specific to its own module. This is also a object-oriented design principle; Interface segregation principle (ISP) (20). This allows users to create their own modules that can easily be interchanged with the existing one. The user only needs to implement the interface. All interfaces use standard C# naming convention; Interfaces are prefixed with the letter "I" (21).

9.3.2 Modules

The main modules are: *Kinect*, *ContourTracking*, *CurveDetection*, *FingerRecognition* and *GestureRecognition*. The main modules, interfaces and implementation, are shown in the sub chapters.

9.3.2.1 Kinect

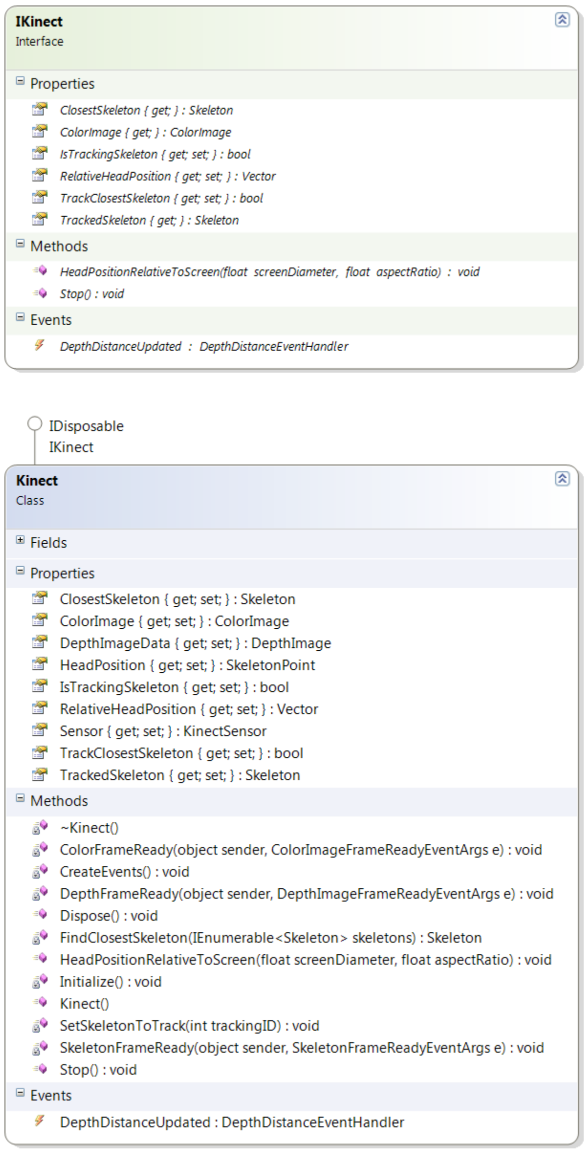
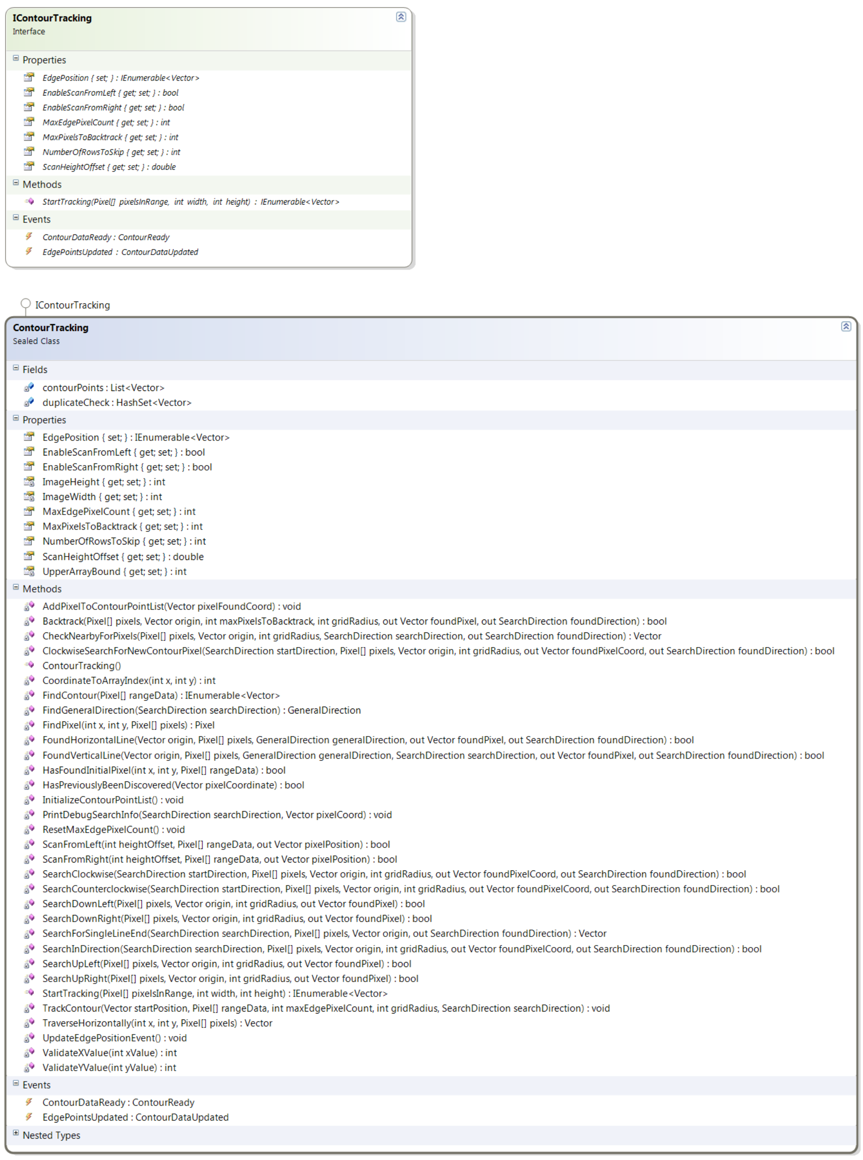

IKinect
Interface
Properties
ClosestSkeleton { get; } : Skeleton
ColorImage { get; } : ColorImage
IsTrackingSkeleton { get; set; } : bool
RelativeHeadPosition { get; set; } : Vector
TrackClosestSkeleton { get; set; } : bool
TrackedSkeleton { get; } : Skeleton
Methods
HeadPositionRelativeToScreen(float screenDiameter, float aspectRatio) : void
Stop() : void
Events
DepthDistanceUpdated : DepthDistanceEventHandler
IDisposable
IKinect
Kinect
Class
Fields
Properties
ClosestSkeleton { get; set; } : Skeleton
ColorImage { get; set; } : ColorImage
DepthImageData { get; set; } : DepthImage
HeadPosition { get; set; } : SkeletonPoint
IsTrackingSkeleton { get; set; } : bool
RelativeHeadPosition { get; set; } : Vector
Sensor { get; set; } : KinectSensor
TrackClosestSkeleton { get; set; } : bool
TrackedSkeleton { get; set; } : Skeleton
Methods
~Kinect()
ColorFrameReady(object sender, ColorImageFrameReadyEventArgs e) : void
CreateEvents() : void
DepthFrameReady(object sender, DepthImageFrameReadyEventArgs e) : void
Dispose() : void
FindClosestSkeleton(IEnumerable<Skeleton> skeletons) : Skeleton
HeadPositionRelativeToScreen(float screenDiameter, float aspectRatio) : void
Initialize() : void
Kinect()
SetSkeletonToTrack(int trackingID) : void
SkeletonFrameReady(object sender, SkeletonFrameReadyEventArgs e) : void
Stop() : void
Events
DepthDistanceUpdated : DepthDistanceEventHandler

Figure 9.1: IKinect and the implementation.

9.3.2.2 ContourTracking



IContourTracking
Interface

Properties

EdgePosition { set; } : IEnumerable<Vector>
EnableScanFromLeft { get; set; } : bool
EnableScanFromRight { get; set; } : bool
MaxEdgePixelCount { get; set; } : int
MaxPixelsToBacktrack { get; set; } : int
NumberOfRowsToSkip { get; set; } : int
ScanHeightOffset { get; set; } : double

Methods

StartTracking(Pixel[] pixelsInRange, int width, int height) : IEnumerable<Vector>

Events

ContourDataReady : ContourReady
EdgePointsUpdated : ContourDataUpdated

IContourTracking

ContourTracking
Sealed Class

Fields

contourPoints : List<Vector>
duplicateCheck : HashSet<Vector>

Properties

EdgePosition { set; } : IEnumerable<Vector>
EnableScanFromLeft { get; set; } : bool
EnableScanFromRight { get; set; } : bool
ImageHeight { get; set; } : int
ImageWidth { get; set; } : int
MaxEdgePixelCount { get; set; } : int
MaxPixelsToBacktrack { get; set; } : int
NumberOfRowsToSkip { get; set; } : int
ScanHeightOffset { get; set; } : double
UpperArrayBound { get; set; } : int

Methods

AddPixelToContourPointList(Vector pixelFoundCoord) : void
Backtrack(Pixel[] pixels, Vector origin, int maxPixelsToBacktrack, int gridRadius, out Vector foundPixel, out SearchDirection foundDirection) : bool
CheckNearbyForPixels(Pixel[] pixels, Vector origin, int gridRadius, SearchDirection searchDirection, out SearchDirection foundDirection) : Vector
ClockwiseSearchForNewContourPixel(SearchDirection startDirection, Pixel[] pixels, Vector origin, int gridRadius, out Vector foundPixelCoord, out SearchDirection foundDirection) : bool
ContourTracking()
CoordinateToArrayIndex(int x, int y) : int
FindContour(Pixel[] rangeData) : IEnumerable<Vector>
FindGeneralDirection(SearchDirection searchDirection) : GeneralDirection
FindPixel(int x, int y, Pixel[] pixels) : Pixel
FoundHorizontalLine(Vector origin, Pixel[] pixels, GeneralDirection generalDirection, out Vector foundPixel, out SearchDirection foundDirection) : bool
FoundVerticalLine(Vector origin, Pixel[] pixels, GeneralDirection generalDirection, SearchDirection searchDirection, out Vector foundPixel, out SearchDirection foundDirection) : bool
HasFoundInitialPixel(int x, int y, Pixel[] rangeData) : bool
HasPreviouslyBeenDiscovered(Vector pixelCoordinate) : bool
InitializeContourPointList() : void
PrintDebugSearchInfo(SearchDirection searchDirection, Vector pixelCoord) : void
ResetMaxEdgePixelCount() : void
ScanFromLeft(int heightOffset, Pixel[] rangeData, out Vector pixelPosition) : bool
ScanFromRight(int heightOffset, Pixel[] rangeData, out Vector pixelPosition) : bool
SearchClockwise(SearchDirection startDirection, Pixel[] pixels, Vector origin, int gridRadius, out Vector foundPixelCoord, out SearchDirection foundDirection) : bool
SearchCounterclockwise(SearchDirection startDirection, Pixel[] pixels, Vector origin, int gridRadius, out Vector foundPixelCoord, out SearchDirection foundDirection) : bool
SearchDownLeft(Pixel[] pixels, Vector origin, int gridRadius, out Vector foundPixel) : bool
SearchDownRight(Pixel[] pixels, Vector origin, int gridRadius, out Vector foundPixel) : bool
SearchForSingleLineEnd(SearchDirection searchDirection, Pixel[] pixels, Vector origin, out SearchDirection foundDirection) : Vector
SearchInDirection(SearchDirection searchDirection, Pixel[] pixels, Vector origin, int gridRadius, out Vector foundPixelCoord, out SearchDirection foundDirection) : bool
SearchUpLeft(Pixel[] pixels, Vector origin, int gridRadius, out Vector foundPixel) : bool
SearchUpRight(Pixel[] pixels, Vector origin, int gridRadius, out Vector foundPixel) : bool
StartTracking(Pixel[] pixelsInRange, int width, int height) : IEnumerable<Vector>
TrackContour(Vector startPosition, Pixel[] rangeData, int maxEdgePixelCount, int gridRadius, SearchDirection searchDirection) : void
TraverseHorizontally(int x, int y, Pixel[] pixels) : Vector
UpdateEdgePositionEvent() : void
ValidateXValue(int xValue) : int
ValidateYValue(int yValue) : int

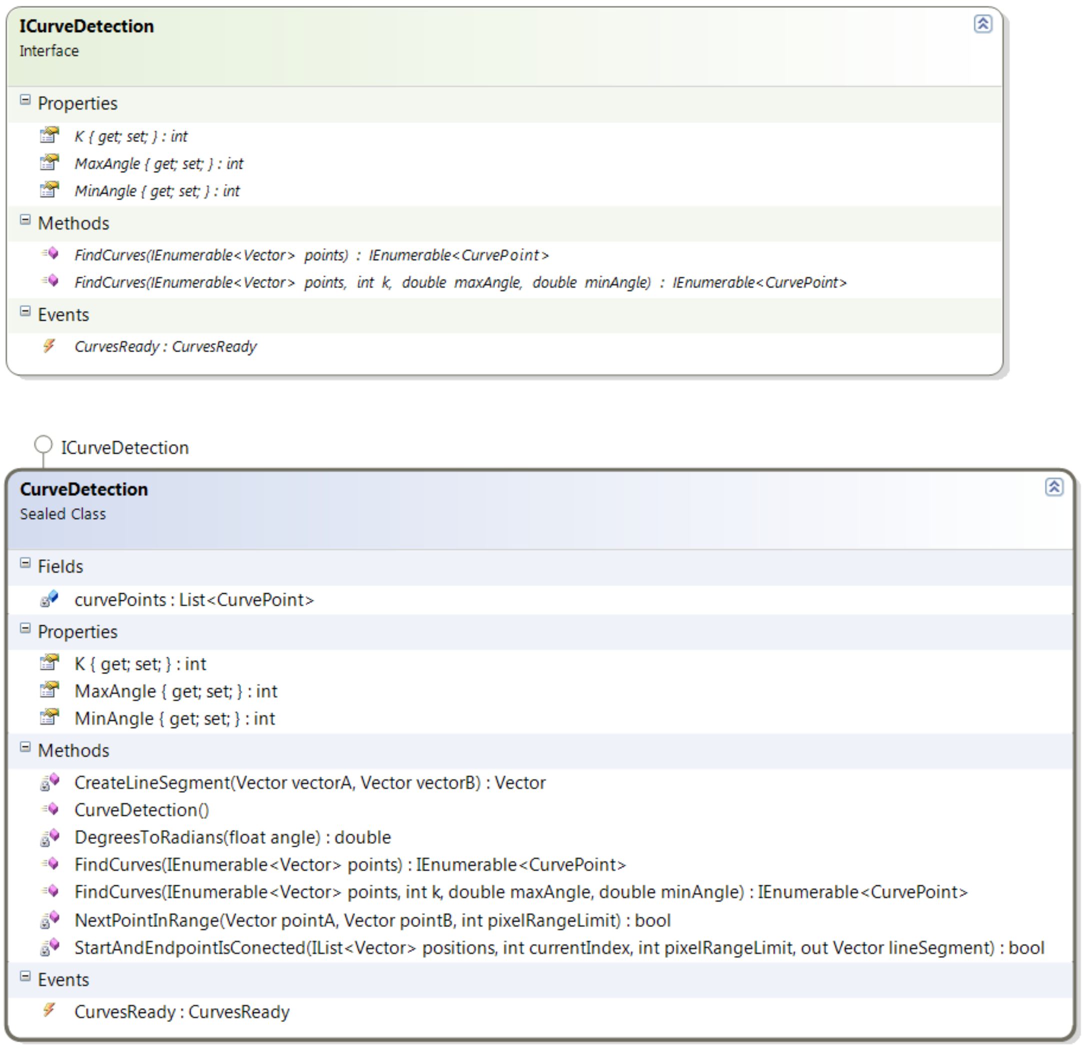
Events

ContourDataReady : ContourReady
EdgePointsUpdated : ContourDataUpdated

Nested Types

Figure 9.2: IContourTracking and the implementation.

9.3.2.3 CurveDetection



ICurveDetection
Interface

Properties
K { get; set; } : int
MaxAngle { get; set; } : int
MinAngle { get; set; } : int

Methods
FindCurves(IEnumerable<Vector> points) : IEnumerable<CurvePoint>
FindCurves(IEnumerable<Vector> points, int k, double maxAngle, double minAngle) : IEnumerable<CurvePoint>

Events
CurvesReady : CurvesReady

ICurveDetection

CurveDetection
Sealed Class

Fields
curvePoints : List<CurvePoint>

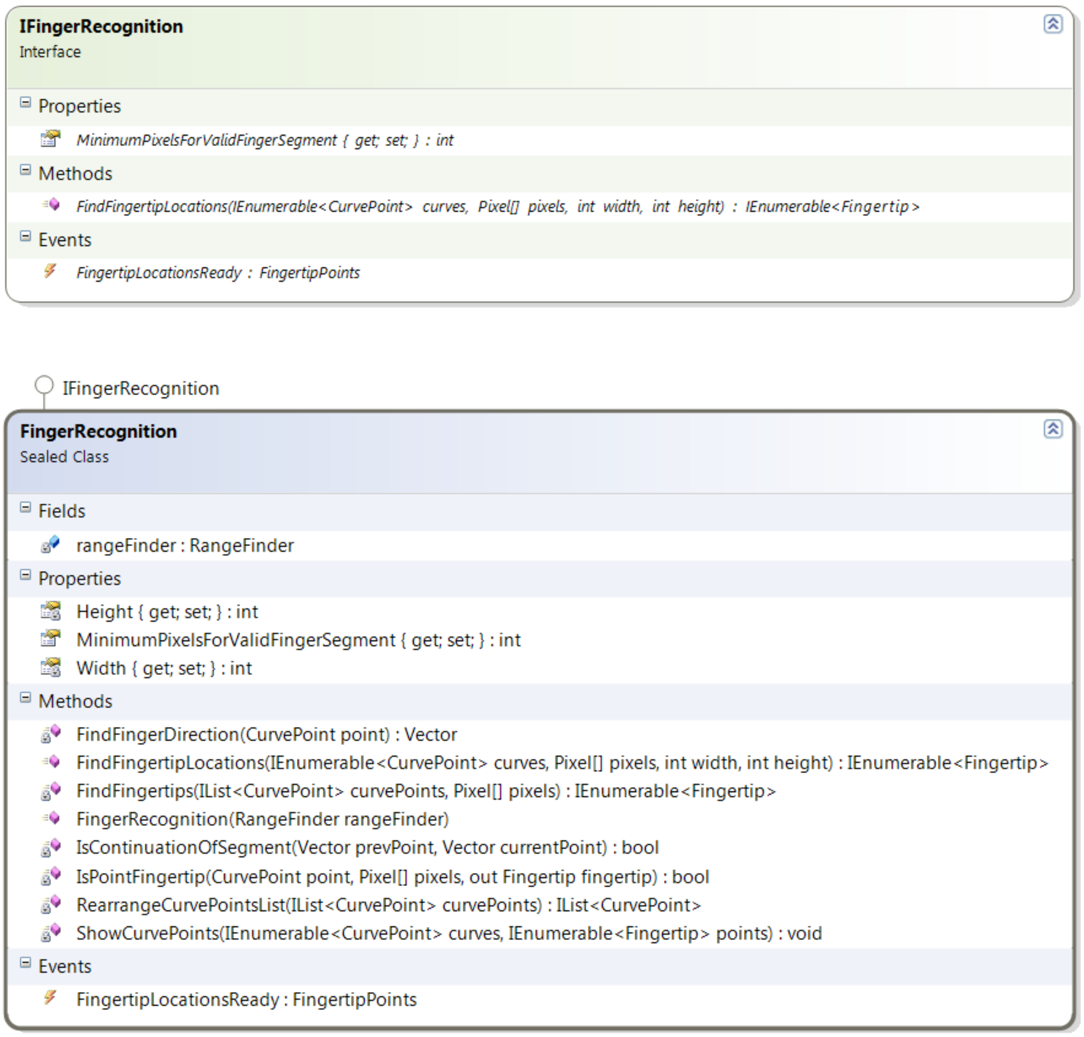
Properties
K { get; set; } : int
MaxAngle { get; set; } : int
MinAngle { get; set; } : int

Methods
CreateLineSegment(Vector vectorA, Vector vectorB) : Vector
CurveDetection()
DegreesToRadians(float angle) : double
FindCurves(IEnumerable<Vector> points) : IEnumerable<CurvePoint>
FindCurves(IEnumerable<Vector> points, int k, double maxAngle, double minAngle) : IEnumerable<CurvePoint>
NextPointInRange(Vector pointA, Vector pointB, int pixelRangeLimit) : bool
StartAndEndpointIsConected(IList<Vector> positions, int currentIndex, int pixelRangeLimit, out Vector lineSegment) : bool

Events
CurvesReady : CurvesReady

Figure 9.3: ICurveDetection and the implementation.

9.3.2.4 FingerRecognition


IFingerRecognition
Interface

Properties
- *MinimumPixelsForValidFingerSegment { get; set; } : int*

Methods
- *FindFingertipLocations(IEnumerable<CurvePoint> curves, Pixel[] pixels, int width, int height) : IEnumerable<Fingertip>*

Events
- *FingertipLocationsReady : FingertipPoints*

IFingerRecognition

FingerRecognition
Sealed Class

Fields
- rangeFinder : RangeFinder

Properties
- Height { get; set; } : int
- MinimumPixelsForValidFingerSegment { get; set; } : int
- Width { get; set; } : int

Methods
- FindFingerDirection(CurvePoint point) : Vector
- FindFingertipLocations(IEnumerable<CurvePoint> curves, Pixel[] pixels, int width, int height) : IEnumerable<Fingertip>
- FindFingertips(IList<CurvePoint> curvePoints, Pixel[] pixels) : IEnumerable<Fingertip>
- FingerRecognition(RangeFinder rangeFinder)
- IsContinuationOfSegment(Vector prevPoint, Vector currentPoint) : bool
- IsPointFingertip(CurvePoint point, Pixel[] pixels, out Fingertip fingertip) : bool
- RearrangeCurvePointsList(IList<CurvePoint> curvePoints) : IList<CurvePoint>
- ShowCurvePoints(IEnumerable<CurvePoint> curves, IEnumerable<Fingertip> points) : void

Events
- FingertipLocationsReady : FingertipPoints

Figure 9.4: IFingerRecognition and the implementation.

9.3.2.5 GestureRecognition

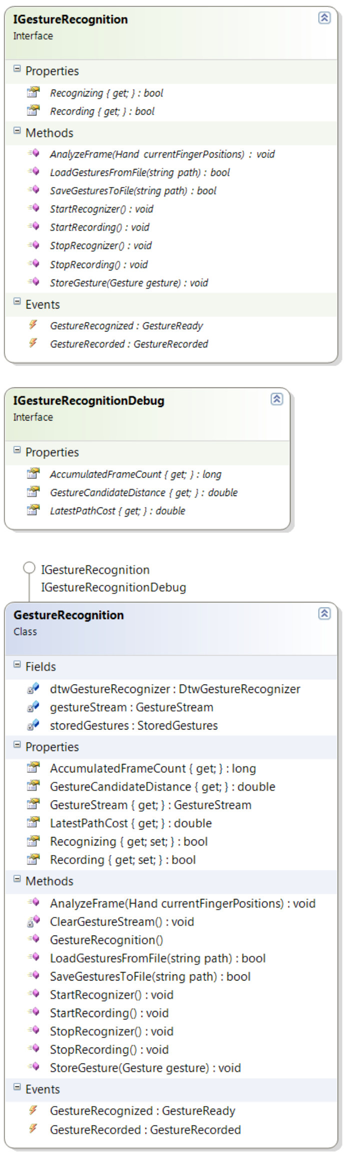

IGestureRecognition
Interface
Properties
Recognizing { get; } : bool
Recording { get; } : bool
Methods
AnalyzeFrame(Hand currentFingerPositions) : void
LoadGesturesFromFile(string path) : bool
SaveGesturesToFile(string path) : bool
StartRecognizer() : void
StartRecording() : void
StopRecognizer() : void
StopRecording() : void
StoreGesture(Gesture gesture) : void
Events
GestureRecognized : GestureReady
GestureRecorded : GestureRecorded
IGestureRecognitionDebug
Interface
Properties
AccumulatedFrameCount { get; } : long
GestureCandidateDistance { get; } : double
LatestPathCost { get; } : double
IGestureRecognition
IGestureRecognitionDebug
GestureRecognition
Class
Fields
dtwGestureRecognizer : DtwGestureRecognizer
gestureStream : GestureStream
storedGestures : StoredGestures
Properties
AccumulatedFrameCount { get; } : long
GestureCandidateDistance { get; } : double
GestureStream { get; } : GestureStream
LatestPathCost { get; } : double
Recognizing { get; set; } : bool
Recording { get; set; } : bool
Methods
AnalyzeFrame(Hand currentFingerPositions) : void
ClearGestureStream() : void
GestureRecognition()
LoadGesturesFromFile(string path) : bool
SaveGesturesToFile(string path) : bool
StartRecognizer() : void
StartRecording() : void
StopRecognizer() : void
StopRecording() : void
StoreGesture(Gesture gesture) : void
Events
GestureRecognized : GestureReady
GestureRecorded : GestureRecorded

Figure 9.5: IGestureRecognition and the implementation.

9.3.3 Tying modules together

An example on how the modules can come together in one class, is given in the appendix B.3.

9.4 Virtual reality

The off-center perspective camera is implemented as a XNA game component. To use the camera one does only need to initialize the game component and add it to the XNA game components list. To update the camera position, one can call the *IKinect.RelativeHeadPosition*(...) method to get the head position relative to the screen and the Kinect. See code block 9.6. If we lose the skeleton tracking the camera will return to origin, which is at the middle of the screen. We use a smooth step function from the XNA framework to achieve a more elegant return to the origin.

Code block 9.6 shows the update method in the main XNA class.

```
void Update(GameTime gameTime)
{
  var origin = new Vector3(0, 0, 1);
  var smoothWeighting = 0.1f;

  if (IKinect.IsTrackingSkeleton)
    PerspectiveCamera.Position =
        IKinect.RelativeHeadPosition(horizontalOffset, floorToKinectOffset,
        screenToKinectOffset, screenWidth);
  else
    SmoothStepCameraPosition(origin, smoothWeighting); // Return to origin.

  base.Update(gameTime);
}
```

Code block 9.6: Off-center perspective camera.

The virtual reality gives the user a feeling of real 3D inside the monitor. A demo application was created for IT-Galla 2012, a IT conference in Stavanger Norway, to showcase what can be done with the Kinect.

9.5 Code

While writing the code, clean code principles were followed (22). This resulted in a code base that is clean, easy to read and easy to follow. It should be easy for a programmer to understand what the code does regardless of the programmers familiarity with the algorithms. Method and variable names are descriptive, ensuring fast and easy reading. Additional comments were added when necessary to further explain code intension and flow.

The main modules are divided into folders to keep the Visual Studio solution clean and organized.

A class for holding debugging information was also created to make it easier and cleaner to access relevant debugging information.

10

Discussion

Currently the API does what we wanted it to do when we started the project. There are room for improvements and new features that can enhance it capabilities. Due to the time constraint of this project not all enhancements that is nice to have were implemented. In this chapter we describe some of the enhancements that can be implemented to improve to results from the API.

10.1 Contour tracking

This is were most of the work went into; To create a reliable contour tracking. It works well, is reliable and efficient in finding the contour.

10.1.1 Finding the initial pixel

To find the initial pixel we only scan every fifth line beginning from the bottom with a height offset of 20% of the image height (chapter 3.2). There are multiple ways the initial pixel can be found. One alternative can be to start at the middle of the image and probe up and down to find the edge of the object. With this alternative method we eliminate the possibility that the object could be under the 20% height offset and rendering it invisible to the current method of finding the initial pixel. However, with the current Kinect hardware we are not able to produce a good enough interpretation of the hand if it were to fit in 20% of the image height pixels.

10.2 Curve detection

10.2.1 The k-curvature algorithm

The curve detection is implemented with the k-curvature algorithm (chapter 4.2). The constant k in the algorithm determines the distance, in pixels, between the start and end point of the line segments. If we were to hold our hand further from the Kinect, the resolution will be lower; Reducing the number of pixels that a fingertip is made up of. If k is static we risk not correctly detecting the fingertip because the line segments will be too long. A solution to this problem is to let k dynamically change depending on how far the hand is from the Kinect. Currently the user has the ability to change k at runtime, it should not take too much effort to implement it to be dynamic. The default k value works well with different depth distances, but a dynamic value would produce more reliable results.

10.3 Detecting Fingertips

The fingertip detection works well when there are not too many artifacts in the image. If there are too much artifacts around a finger, it will have trouble detecting it correctly due to deformation of the finger.

10.3.1 False fingertip

There is also a possibility that other curves might be mistaken for fingertips if it has the correct curvature. One solution to prevent false curves from being detected as correct is to do some more calculation to determine if a curve is part of a finger. This could be to have a proximity constraint between the fingertips; If a detected fingertip is too far from the others it will be discarded.

10.4 The Kinect

Everything is dependent on the depth image feed from the Kinect. With the Kinect for Xbox 360 we only have a small interval where we will get good results from the depth sensor. With the Kinect for Windows and its near mode we can have our hands closer to the Kinect.

A better sensor with higher resolution will greatly improve the range where one can hold its hands and be detected by the API. The Kinect module in the API can be changed to be used with other sensor hardware as long as it fulfills the *IKinect* interface properties. This makes the API more future proof as new hardware enters the marked.

10.5 Commercialization

GesturePak is a gesture recording and recognition toolkit that cost $99 for a single developer license (23). It does gesture recording and recognition, and provides an API. However, it utilizes the skeleton points from the Kinect, thus finger recognition is not supported.

This shows that there is a market for gesture recording and recognition API that is based on the Kinect. Since we have created an API with finger tracking capabilities, we have the market to our selves. There is nothing that prevents us from selling our API. However, if we open source the API we can give something back to the community for free. Others may be inspired to create new and innovating applications and contribute to the Kinect community with their findings.

11

Conclusion

We have created an API that extends the usage of the Kinect, making it possible to create applications for a new field where finger gestures play a significant role. The API is made to suit anyone, from amateur programmers with its ease of use, to the professional programmer with the use of modularity and its flexibility. With better sensor hardware, we will be able to increase the physical working limit of the API.

This shows what can be done with consumer hardware and a SDK in a relative short time span. It really is only our imagination that limits what we can do with the Kinect.

References

[1] S. Spielberg, P. Dick, and J. Kaminski, "Minority report," 2002. 1

[2] J. C. Lee, "Head tracking for desktop vr disply using the wii remote." Website. 2, 35

[3] Microsoft, "Microsoft kinect for windows sdk eula." Website. 3

[4] S. Nirjon and J. Stankovic, "Kinsight: Localizing and tracking household objects using depth-camera sensors," 4

[5] S. van der Linden, "Project mimicry, the ultimate sandbox game by monobanda." Press release, 2011. 5

[6] M. Ren, J. Yang, and H. Sun, "Tracing boundary contours in a binary image," *Image and vision computing*, vol. 20, no. 2, pp. 125–131, 2002. 7

[7] F. Chang, C. Chen, and C. Lu, "A linear-time component-labeling algorithm using contour tracing technique," *Computer Vision and Image Understanding*, vol. 93, no. 2, pp. 206–220, 2004. 7

[8] L. Yan and Z. Min, "A new contour tracing automaton in binary image," in *Computer Science and Automation Engineering (CSAE), 2011 IEEE International Conference on*, vol. 2, pp. 577 –581, june 2011. 7

[9] MIT, "Kinect finger detection." Website. 16

[10] T. Trigo and S. Pellegrino, "An analysis of features for hand-gesture classification," 2010. 16

[11] M. Raptis, D. Kirovski, and H. Hoppe, "Real-time classification of dance gestures from skeleton animation," in *Proceedings of the 10th Annual ACM SIGGRAPH/Eurographics Symposium on Computer Animation, SCA 2011*, pp. 147–156, 2011. 19

[12] H. Sakoe and S. Chiba, "Dynamic programming algorithm optimization for spoken word recognition," *Acoustics, Speech and Signal Processing, IEEE Transactions on*, vol. 26, no. 1, pp. 43–49, 1978. 24

[13] F. Itakura, "Minimum prediction residual principle applied to speech recognition," *Acoustics, Speech and Signal Processing, IEEE Transactions on*, vol. 23, no. 1, pp. 67–72, 1975. 24

[14] R. Kooima, "Generalized perspective projection," August 2008. 33

[15] H. Tiedemann, "Head tracking based 3d illusion and possibilities of interaction on tabletop systems." Master thesis. 33

[16] J. Lee, "Hacking the nintendo wii remote," *Pervasive Computing, IEEE*, vol. 7, pp. 39 –45, july-sept. 2008. 33

[17] MSDN, "Create perspective off center matrix." Website. 34

[18] F. Miller, A. Vandome, and M. John, *Modular Programming*. VDM Verlag Dr. Mueller e.K., 2010. 37

[19] P. Rieck, "Performance implication of .net events." Website. 40

[20] R. Martin, *Agile software development: principles, patterns, and practices*. Prentice Hall PTR, 2003. 41

[21] P.-E. Dautreppe, "C# coding conventions." PDF, 2005. 41

[22] R. C. Martin, *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall, 2008. 48

[23] C. Franklin, "Gesturepak: Gesture recording and recognition toolkit." website, March 2012. 51

Appendix A

Create line segment

Algorithm A.0.1: CreateLineSegments(*contourPoints*, *k*, *omega*)

```
$curvePoints = \emptyset$
for $i_{current} \leftarrow 0$ to $contourPoints.Size$
do
    if $contourPoints$ is cyclic
    then
        if $i_{current} < k$
          then $i_b = contourPoints.Size - k + i_{current}$
          else
          then $i_b = i_{current} - k$

        $\vec{b} = CreateVector(contourPoints[i_{current}], contourPoints[i_b])$

        if $i_{current} > contourPoints.Size - k$
          then $i_a = k - contourPoints.Size + i_{current}$
          else $i_a = i_{current} + k$

        $\vec{a} = CreateVector(contourPoints[i_{current}], contourPoints[i_a])$
    else
    then
        if $i_{current} < k$
          then $i_b = 0$
          else
          then $i_b = i_{current} - k$

        $\vec{b} = CreateVector(contourPoints[i_{current}], contourPoints[i_b])$

        if $i_{current} > contourPoints.Size - k$
          then $i_a = k - contourPoints.Size$
          else $i_a = i_{current} + k$

        $\vec{a} = CreateVector(contourPoints[i_{current}], contourPoints[i_a])$

    $\vec{c} = CreateVector(\vec{a}, \vec{b})$

    $curvePoints = curvePoints \cap \{contourPoints[i_{current}], \vec{a}, \vec{b}, \vec{c}\}$

return $(curvePoints)$
```

Appendix B

B.1 ContourDataReady event with auto interval scan.

```
void ContourDataReady(contourPoints, pixles)
{
  var curves = ICurveDetection.FindCurves(contourPoints);

  var fingertips = IFingerRecognition.FindFingertipLocations(curves,
      pixels, width, height);

  var fingerCount = fingertips.Count;

  if (updateDepthDistaceThreshold)
  {
    AutoscanForDistanceThreshold(fingerCount);
    return; // Don't do anything else while scan is in progress.
  }
}

void AutoscanForDistanceThreshold(fingerCount)
{
  // The DistanceThreshold object holds the min and max distance.
  DistanceThreshold newDistanceThreshold;

  bool scanFinished = DistanceScanner.TwoHandScan(fingerCount,
      distanceThreshold, out newDistanceThreshold);

  // distanceThreshold is a global variable used by the range finder.
  distanceThreshold = newDistanceThreshold;

  updateDepthDistaceThreshold = !scanFinished;
}
```

Code block B.1: ContourDataReady event with auto interval scan.

B.2 Program flow

1. User is in front of Kinect
2. Depth stream
   - Get depth data.
   - Filter data to only contain objects that are in a certain interval.
3. Contour tracking
   - Track contour of objects in the depth interval.
   - Track up to two objects.
   - Returns contour point.
4. Curve detection
   - Find curves from the contour point.
   - Return a set of curves.
5. Finger recognition
   - Filter curves based on curve direction.
   - Find center of curve; this is a fingertip.
   - Find pointing direction of fingertip.
   - Returns fingertip position and direction.
6. Enhance processed data
   - Smoothing.
     - Apply smoothing to reduce jitter.
     - Useful for better recognition of gestures.
   - Prediction; Predict finger positions and direction in the next frame.
     - Used if one or more fingers was not detected due to too many artifacts in the depth stream.
     - The missing fingers are replaced with the predicted fingers.

7. Gesture recording and recognition
    - Recording.
        - Store fingertip position and direction for each frame during recording.
    - Recognition, using dynamic time warping (DTW).
        - Store a set of processed frames from the Kinect in a queue.
        - Check if the latest processed frame from the Kinect matches the last frame in a gesture.
        - If it does, we have a candidate gesture.
        - Convert the frame queue to a gesture (input gesture).
        - Do full DTW calculation on the candidate gesture and the input gesture.
        - If they are reasonably similar, we have recognized a gesture.

B.3 Main class

```
public sealed class Main
{
        private readonly IKinect kinectDevice;
        private readonly RangeFinder rangeFinder;
        private readonly IContourTracking contourTracking;
        private readonly ICurveDetection curveDetection;
        private readonly IFingerRecognition fingerRecognition;
        private readonly DistanceScanner distanceScanner;
        private readonly IGestureRecognition gestureRecognition;
        private readonly IPrediction prediction;
        private readonly HandEnhancements handEnhancements;
        private readonly DebugInfo debugInfo;
        private Hand prevHand;

        public Main(IKinect kinectDevice)
        {
            // Distances in millimeter.
            const int minDepthDistance = 800; // The minimum distance where
                the Kinect for Xbox 360 can detect objects.
            const int maxDepthDistance = 4000;

            this.kinectDevice     = kinectDevice;
            var sensorDepthRange = new DistanceThreshold { MinDistance =
                minDepthDistance, MaxDistance = maxDepthDistance };
            rangeFinder           = new RangeFinder();
            contourTracking       = new ContourTracking();
            curveDetection        = new CurveDetection();
            fingerRecognition     = new FingerRecognition(rangeFinder);
            distanceScanner       = new DistanceScanner(sensorDepthRange);
            gestureRecognition    = new GestureRecognition();
            prediction            = new Prediction();
            handEnhancements      = new HandEnhancements(prediction,
                gestureRecognition);
            debugInfo             = new DebugInfo();


            InitializeDistanceThreshold(minDepthDistance);
            InitializeDebugInfo();
            CreateEvents();
        }
```

```
public IContourTracking ContourTracking { get { return
    contourTracking; } }
public ICurveDetection CurveDetection { get { return
    curveDetection; } }
public IFingerRecognition FingerRecognition { get { return
    fingerRecognition; } }
public IGestureRecognition GestureRecognition { get { return
    gestureRecognition; } }
public DebugInfo DebugInfo { get { return debugInfo; } }

private void InitializeDistanceThreshold(int minDepthDistance)
{
    const int distanceInterval = 150;
    DistanceThreshold = new DistanceThreshold
                            {
                                MinDistance = minDepthDistance,
                                MaxDistance = minDepthDistance +
                                    distanceInterval
                            };

    debugInfo.DistanceThreshold = DistanceThreshold;
}

private void InitializeDebugInfo()
{
    debugInfo.GestureDebugInfo = gestureRecognition as
        IGestureRecognitionDebug;
}

private void CreateEvents()
{
    kinectDevice.DepthDistanceUpdated         += new
        DepthDistanceEventHandler(kinectDevice_DepthDistanceUpdated);
    contourTracking.ContourDataReady          += new
        ContourReady(contourTracking_ContourDataReady);
    fingerRecognition.FingertipLocationsReady += new
        FingertipPoints(fingerRecognition_FingertipLocationsReady);
    gestureRecognition.GestureRecognized      += new
        GestureReady(gestureRecognition_GestureRecognized);
    gestureRecognition.GestureRecorded        += new
        GestureRecorded(gestureRecognition_GestureRecorded);
}

private void kinectDevice_DepthDistanceUpdated(short[]
    depthDistanceData, int width, int height)
```

```
{
    Width = width;
    Height = height;

    Pixel[] pixelsInRange =
        rangeFinder.PixelsInRange(depthDistanceData,
        DistanceThreshold.MinDistance,
        DistanceThreshold.MaxDistance);

    debugInfo.RangeData = pixelsInRange;

    contourTracking.StartTracking(pixelsInRange, width, height);
}


private void contourTracking_ContourDataReady(IEnumerable<Vector>
    contourPoints, Pixel[] pixels)
{
    IEnumerable<CurvePoint> curves =
        curveDetection.FindCurves(contourPoints);

    IEnumerable<Fingertip> points =
        fingerRecognition.FindFingertipLocations(curves, pixels,
        Width, Height);
    int fingerCount = ((IList<Fingertip>)points).Count;

    if (UpdateDepthDistaceThreshold)
    {
        AutoscanForDistanceThreshold(fingerCount);
        return; // Dont do anything else while scan is in progress.
    }
}

private void AutoscanForDistanceThreshold(int fingerCount)
{
    DistanceThreshold newDistanceThreshold;

    bool scanFinished = distanceScanner.TwoHandScan(fingerCount,
        DistanceThreshold, out newDistanceThreshold);
    DistanceThreshold = newDistanceThreshold;

    UpdateDepthDistaceThreshold = !scanFinished;

    debugInfo.AutoscanInProgress = UpdateDepthDistaceThreshold;
    debugInfo.DistanceThreshold = DistanceThreshold;
```

```
        }

        private void
            fingerRecognition_FingertipLocationsReady(IEnumerable<Fingertip>
            points)
        {
            Hand currentHand = new Hand(points);

            if (EnableSmoothing)
                currentHand = Smoothing.ExponentialSmoothing(currentHand,
                    prevHand, SmoothingFactor);

            if(PreventHandInconsitencies)
                handEnhancements.PreventHandIncosistency(currentHand);
            else if (gestureRecognition.Recognizing ||
                gestureRecognition.Recording)
                gestureRecognition.AnalyzeFrame(currentHand);

            debugInfo.FoundHandInconsistencies =
                handEnhancements.FixedInconsistencies;

            debugInfo.FingertipLocationsReadyCounter++;

            prevHand = currentHand;
        }

        private void gestureRecognition_GestureRecognized(Gesture
            recognizedGesture)
        {
            debugInfo.GestureRecognized = true;
        }

        private void gestureRecognition_GestureRecorded(Gesture
            recordedGesture)
        {
            gestureRecognition.StoreGesture(recordedGesture);
        }

        public double SmoothingFactor { get; set; }

        public bool EnableSmoothing { get; set; }

        public bool PreventHandInconsitencies { get; set; }

        private DistanceThreshold DistanceThreshold { get; set; }
        public bool UpdateDepthDistaceThreshold { get; set; }

        private int Height { get; set; }
        private int Width { get; set; }
}
```

Code block B.2: Main class.