August | 12

3D Skeleton extraction using one Kinect camera.

Constantinos Glynos
MSc Computer Animation & Visual Effects

NCCA 2012
Bournemouth University

Abstract

Nowadays, it is well established that Computer Vision is used in many applications and devices. One area of huge interest is the detection and reconstruction of the human skeletal structure. In short words, the camera will identify the human parts and construct the joints of the person standing in front of it. So far, scientists have managed to build various systems that reconstruct the human skeleton representing it as lines and spheres in 2D using one camera and 3D using two or more cameras. Although there are still a few areas requiring improvements for such methods, such as computational cost and stability, the results are generally accurate and efficient.

Currently, the reconstruction of the human skeleton in 3D using one camera is still under research. In this master thesis a relatively novel approach is proposed, using image-processing techniques, mesh extraction, line skeletonization, 2D Skeleton extraction and depth detection methods, resulting into a 3D skeleton reconstruction of the left hand.

Acknowledgments

I would like to specially thank Dr. Hammadi Nait-Charif who supervised this project's progress from the very beginning, for his expertise on computer vision and his advice and guidance to my problem solving stages. Most of all I would like to thank him for his patience and belief.

Table of Contents

Table of Figures

Chapter 1

INTRODUCTION

Computer vision is a field in computer science yet to be fully discovered. It can be defined as the science of processing, analyzing and understanding images. Over decades many researchers have been working on developing new ideas or improving existing applications in order to "take it" to the next level. Initially computer vision would work on a two dimensional environment using one camera and should the programmer wished to add the third dimension he or she would have to use two or more cameras depending on the nature of the application and the amount of detail needed to be accessed. Generally, the use of computer vision has great prospects and could be used in almost any application, from army surveillance to games and films to medical research and training to only educational purposes.

One area of great interest is the detection and reconstruction of the human skeletal structure. In short, the computer will identify the human parts and construct the joints of the person standing in front of it in a three dimensional environment. This method would give so much freedom to the user to perform tasks that require great detail and accuracy. So far scientists have managed to build various systems that reconstruct the human skeleton representing it as lines and spheres in 2D using one camera and 3D using two or more cameras. They have been using simple RGB cameras and advanced depth sensors, which are very expensive and hard to access.

In 2009, Microsoft released the Kinect, a motion sensing camera device which captures the movement of objects in a 3D environment. This three dimensional scanner was cheap enough to allow more researchers to test their theories in order to accomplish their goal.

As far as skeletonization is concerned, there are still a few areas requiring improvements, such as computational cost and stability, but the results are generally accurate and efficient. Some of the techniques will be discussed in the next chapter.

Currently, the reconstruction of the human skeletal structure in 3D using one camera is still under research because the problem of self-occluded objects is yet to be solved. In this research thesis the basic idea is to construct a fully functioning 2D skeleton, which would also solve some of the issues of stability, accuracy and computational cost, and then using the Kinect's depth sensor and detect the self-occluded points. Furthermore, various techniques to combine the two methods will be introduced, discussing each ones advantages and disadvantages.

Chapter 2

RELATED WORK

Skeleton extraction techniques in a 2D or 3D co-ordinate system are researched for various applications either using a set of cameras, or other scans such as CT and MRI, or straight from software using voxels to avoid having self occluded objects.

TEASAR (tree-structure extraction algorithm for accurate and robust skeletons) was introduced by Sato M. et al. in 2000. In their method they used CT and MRI scans as input data to retrieve a full view of the object. They gather data as voxels and define the skeleton as a tree composed of paths. Using the Dijkstra algorithm to determine the global minimal weight path and the Euclidean distance transform they extracted the minimum cost path from voxel to root and finally labeled the voxels near that path. The algorithm seems to be very accurate (Figure 1) and according to their analysis, very efficient.

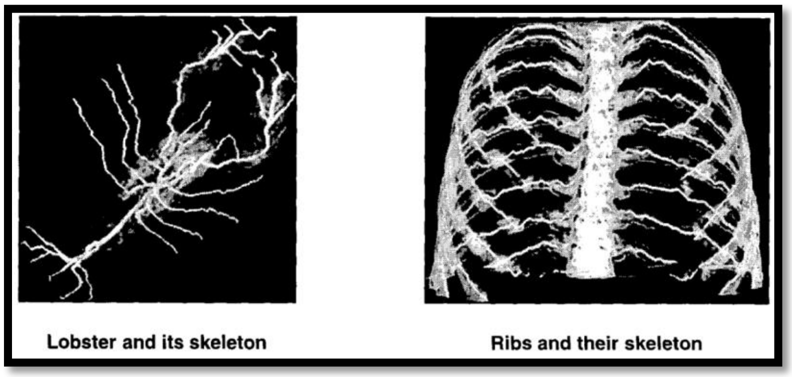


Figure 1: TEASAR algorithm skeleton extraction.

In 2002, a method to retrieve a 2D skeleton out of (Yi Sun et al. 2000) "Freeman Chain Code (FCC)" was introduced, suggesting that the best way to determine a skeleton is to find the boundary contours of the silhouette after having removed the background. They divide the body parts into several sub parts and they label them in order to retrieve the co-ordinates of the joints. They tried to solve the issues caused by self occlusion by estimating the position (Figure 2) of the joints using motion analysis.

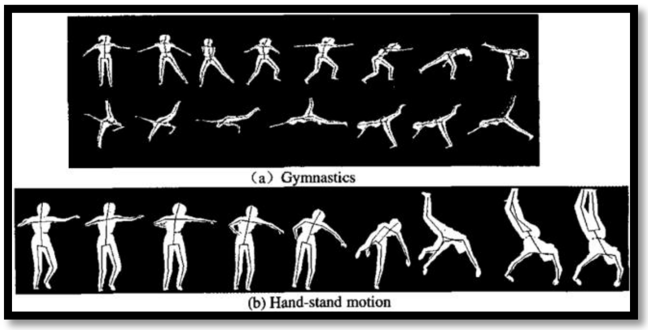


Figure 2: 2D skeleton extraction using FCC.

A technique to retrieve the 1D line skeleton was presented in 2008 by Yu-Shuen and Tong-Yee. Essentially they shrink the mesh by contracting the edges between the adjacent voxels and they add forces to preserve the original positions of the boundary voxels. Although the resulting skeleton is accurate and straight because of the method shrinking the mesh to make the model thinner, it causes (Yu-Shuen et al. 2008) "the skeleton to diverge from the center of the model" (Figure 3).

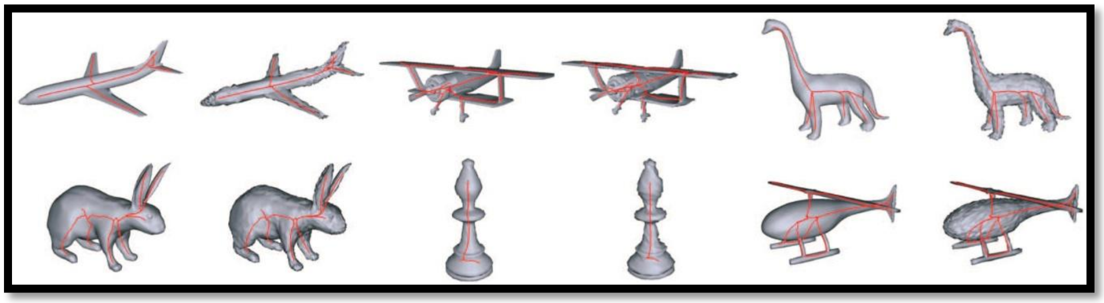


Figure 3: Curve skeleton extraction using iterative least squares optimization.

In 2010, Le Zhang et. al presented a hybrid method of the Euclidean Distance Transform (EDT) and the thinning technique for topology presentation. It is a very basic technique, which was used to obtain the 2D skeleton for any object but the results (Figure 4) are accurate and efficient.

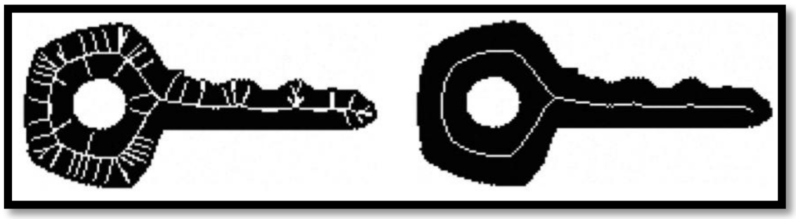


Figure 4: Euclidean distance-ordered thinning for skeleton extraction.

In 2012, three Kinects were used to scan the entire human body outlining the camera's advantages and disadvantages (Figure 5). It explains the disadvantages and the weaknesses of the Kinect device, such as low resolution and some inaccuracy in depth information (Jing T. et al. 2012) "The quality of the reconstructed models in our system is still poor for some specific applications due to low quality for depth data captured by the Kinects."

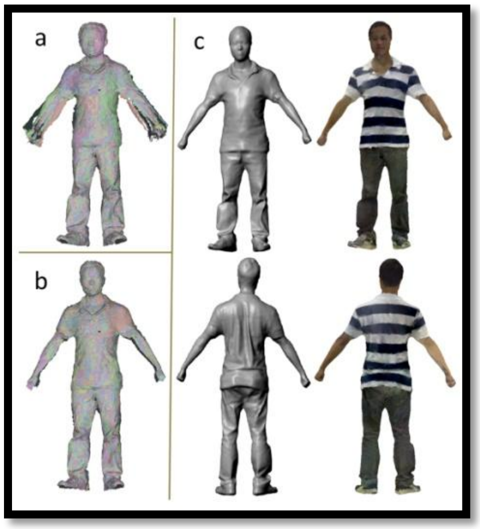


Figure 5: Scanning 3D Full Human Bodies Using Kinects.

According to the historical background, self-occlusion still remains the greatest issue of skeleton extraction, which is the focus of this thesis. Additionally, a 2D skeleton will be constructed and a relatively novel approach to the formation of a 3D skeleton will be presented. Three methods for combining the two skeletons will be presented explaining each ones limitations and facilitations.

Chapter 3

THE KINECT DEVICE

Microsoft currently being the largest software corporation was founded in 1975 providing the well-known "windows" operating system, producing video games, mobile phones, digital services and the Kinect camera (Figure 6).

The Kinect was first announced in 2009 having the ability to detect motion in a three dimensional environment using the IR (Infra Red) projector, RGB camera and depth sensor.

Figure 6: The Microsoft Kinect camera.

3.1 RGB stream

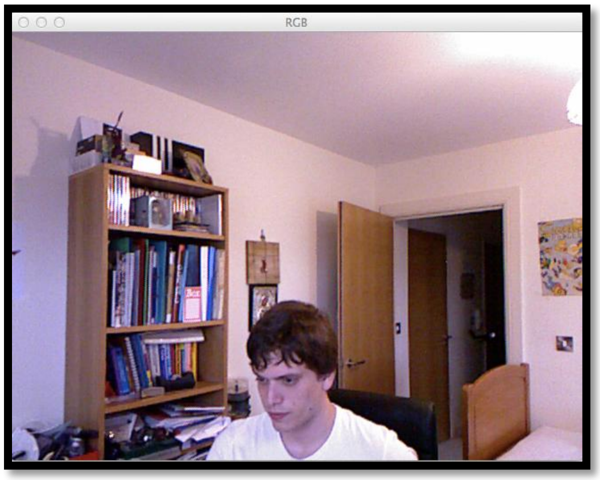


Figure 7: The RGB stream of the Kinect device.

The Kinect's RGB stream (Figure 7) has a frame rate of 30 Hz and a resolution of 640 x 480 pixels. It is used mostly for colour isolation and skin extraction. In object tracking, the RGB is commonly used to track coloured locators positioned on a dark monochromic background. On the other hand the RGB stream is used only under certain lighting conditions, it is very unstable when it comes to template matching and it is very sensitive to noise.

3.2 IR and Depth

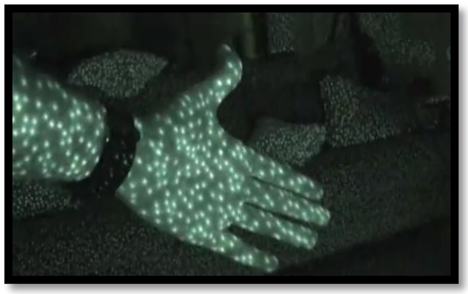


Figure 8: The Infra Red grid projection.

The Kinect is equipped with an Infra Red (IR) (Figure 8) projector working along with a monochrome CMOS sensor that captures all the data in a 3D environment.

(Wikipedia 2012) "The sensing range of the depth sensor is adjustable, and the Kinect software is capable of automatically calibrating the sensor based on game play and the player's physical environment".

The next most important feature of the Kinect is the depth stream (Figure 9). The depth can also be visualized with colour gradients expressing blue as the furthest point and white the nearest to the camera.

Although the resolution of the depth is the same as in the RGB stream, it outputs an 11-bit depth resulting to (Wikipedia 2012) "2,048 levels of sensitivity".

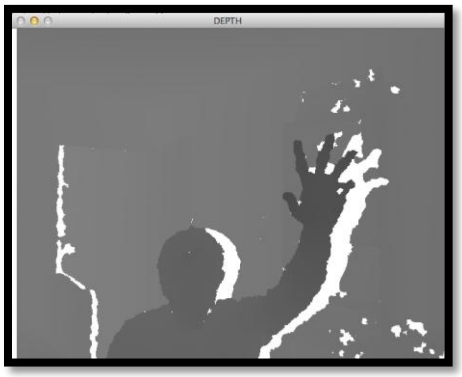


Figure 9: The raw depth stream from the Kinect.

3.3 Hardware limitations

Initially, the working distance begins from 0.2 m and ends at 1.5 m from the camera. Any higher than that, the camera loses track of the object. Lighting is very important too as the camera produces a lot of miss-detections when it is facing directly at a light source. Additionally, the resolution and small disparity of the IR projector and Depth sensor are responsible for the white noise seen in Figure 12, returning nan (not a number) values which can be very threatening.

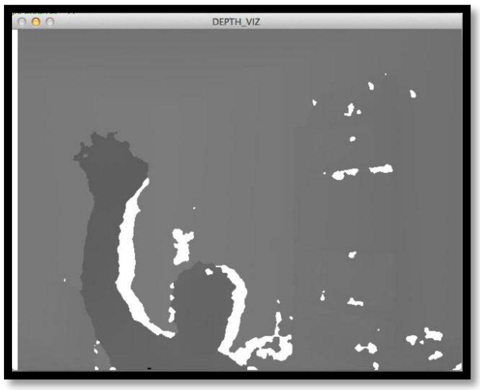


Figure 10: Kinect limitations.

Although the frame rate of the camera is fairly acceptable (30 Hz), it does not work well for fast movements (Figure 10) in the depth stream unless the object is big enough, like an arm. It is very problematic for the fingers as it merges the meshes and the skeletonization algorithm gets confused.

Chapter 4

HAND ANATOMY

Extracting the hands skeletal structure is essentially the detection of certain points that when connected with simple lines, would resemble the shape of the actual hand.

The hand (Figure 11) has a very distinctive shape. It consists of 32 bones including the ones located in the elbow and wrist region. Even if the Kinect was able to detect all 32 joints without any errors it would still be very difficult to keep track of each joint separately. For that reason it was suggested to select the most crucial joints that would determine the hand's position, orientation and movement.

It was decided that only 8 of all the joints were necessary to detect and track the hand (Figure 12). The elbow would be the starting joint followed by the wrist and the palm.

As far as the fingers are concerned, only the fingertips were needed since detecting the fingers starting point would complicate the identification process.

Figure 11: Left Hand.

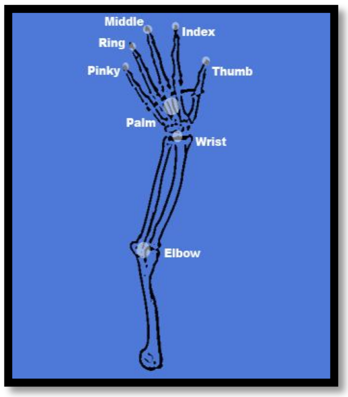


Figure 12: Left Arm Skeleton.

Figure 13 displays the hand with the desired skeleton. It is accurate enough and would allow the main focus to be on solving self-occlusion, which is, detecting the thumb in front of the palm region.

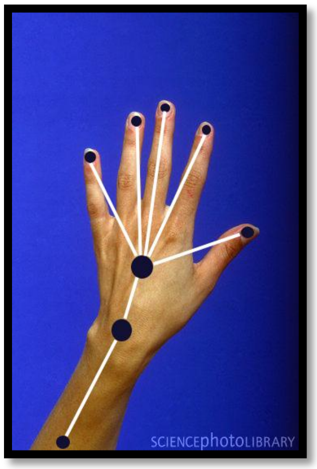


Figure 13: Left Hand showing the points to be detected.

Overall the skeleton will be constructed with 8 points determining the position of each joint of the hand.

Chapter 5

PRE-PROCESSING

At this stage the Kinect device needs to be initialized streaming the data from the depth and RGB cameras. Depending on the source that will be used, some image processing is required in order to set the camera up correctly and to dismiss any small amounts of noise. It is also necessary to store the video stream into a matrix, which then needs to be converted, thresholded and flipped in order to get a visual of the depth with no noise that mirrors the motion detected.

5.1 Image Matrix Conversion

The Kinect outputs an 11-bit depth for each pixel on the 640 x 480 resolution size also being the matrix size, resulting to 2048 sensitivity levels. These attributes are not very efficient to work with, as it is not possible to get a visual output (Figure 14). It is necessary to be converted to an 8-bit depth (Figure 15).

Additionally to converting the original pixels to the ones wanted, (OpenCV API documentation 2010) "a saturate cast is applied at the end to avoid any possible overflows".

$$m(x, y) = saturate\_cast < rType > (\alpha(*this)(x, y) + \beta)$$

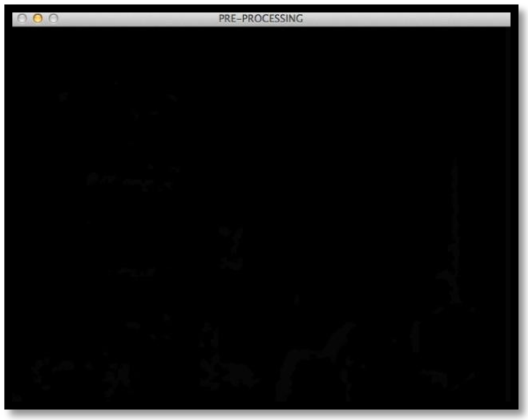


Figure 14: 11-bit depth stream - Original

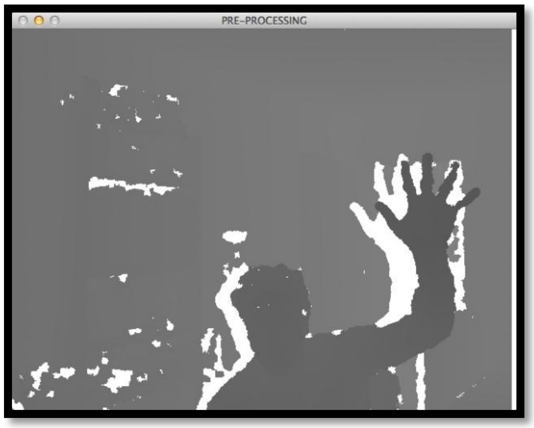


Figure 15: 8-bit depth stream – Reduced

The reduced matrix that stores the depth stream has 640x480 (rows x columns) and now has an 8-bit depth with 255.0/2048 levels of sensitivity.

5.2 Gaussian and Median Blur

Blurring is used mainly to smooth the image and eliminate some detail and most of the noise detected in edges. OpenCV facilitates two of the most important types of blurs, the Gaussian using the Gaussian filter and the Median, which uses the median filter.

The Gaussian filter (Figure 16) can be described as a bell shaped function that uses the Gaussian function to compute its impulse response.

The main Gaussian function is:

$$f(x) = ae^{-\frac{(x-b)^2}{2c^2}}$$

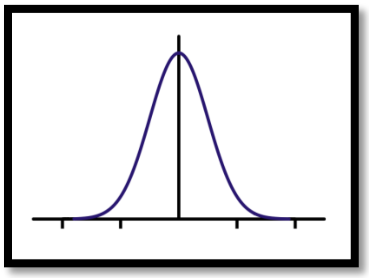


Figure 16: A simple Gaussian filter.

Where a, b and c are real numbers greater than 0 and “e” is the Euler’s number.

Since the image matrix from the depth is a 2D array, the function for computing the Gaussian filter is:

$$g(x,y) = \frac{1}{2\pi\sigma^2} \cdot e^{-\frac{x^2+y^2}{2\sigma^2}}$$

Where x and y are the co-ordinates (width and height), σ is the standard deviation parameter and e is Euler's number.

One the other hand the Median filter is mostly used to remove noise from the image instead of smoothing it. Essentially it calculates the median value of each pixel by taking the two neighboring pixels.

$$median = \frac{a+b}{2}$$

Where a and b are the neighboring pixels inside the matrix.

Applying these two blurs in the original depth matrix (Figure 18) the result is much smoother with less noise and seems more accurate (Figure 17). Yet the white noise still remains as the Kinect still cannot see all the areas.

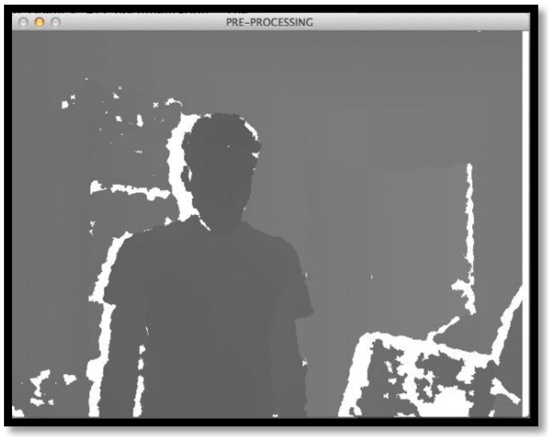


Figure 18: Original depth stream.

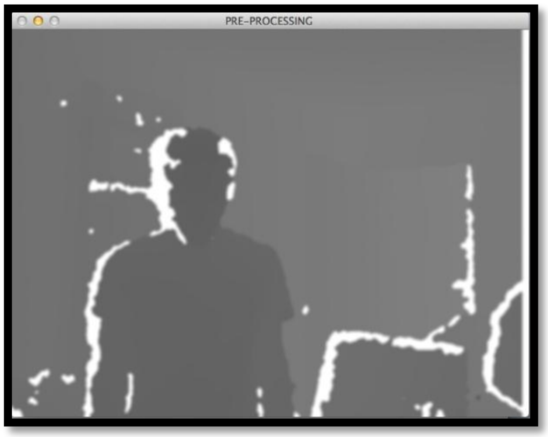


Figure 17: Blur filtered depth.

5.3 Threshold

Threshold converts the original image into a binary one. It is the best way to segment an image leaving only the major shape in the screen. Each pixel in the depth stream has a colour value that varies from 0 to 255. The threshold iterates through each pixel and converts it into binary (0, 1) according to whether that pixels value is smaller or greater than the threshold value given by the user (Figure 19).

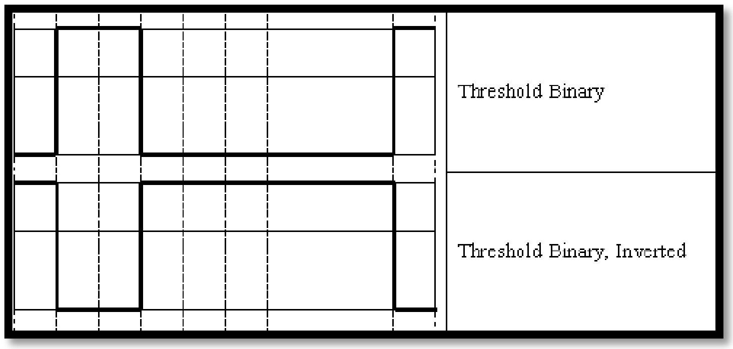


Figure 19: Threshold graph of Binary and its Inverse.

The threshold method is:

$$\mathtt{dst}(x,y) = \begin{cases} \mathtt{maxval} & \text{if } \mathtt{src}(x,y) > \mathtt{thresh} \\ 0 & \text{otherwise} \end{cases}$$

The inverse threshold method is:

$$\mathtt{dst}(x,y) = \begin{cases} 0 & \text{if } \mathtt{src}(x,y) > \mathtt{thresh} \\ \mathtt{maxval} & \text{otherwise} \end{cases}$$

In this case the threshold would serve well as it would automatically ignore the background and only focus on what is inside the depth range (Figure 20).

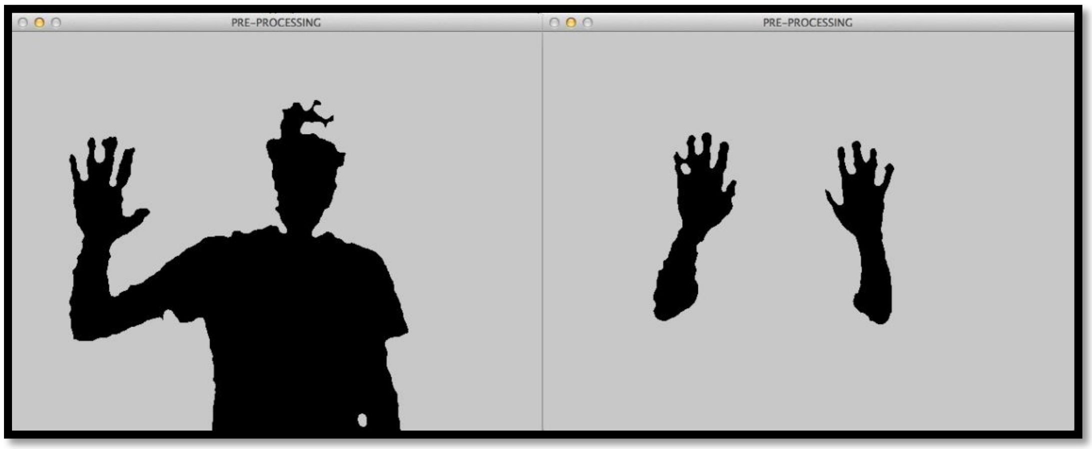


Figure 20: Binary threshold on the image matrix.

5.4 Flip Matrix

It is observed that the image mirrors the environment opposite to the actual one. This is a matter of flipping the image matrix so that when the object moves on the left side, it would appear on the left side of the image too.

In mathematics, in order to accomplish the reflection of a matrix about the x-axis, the following should be calculated:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

In the case of the Kinect's depth stream it would flip the image matrix (Figure 21) so that when the person lifts his/her left hand it would appear the same in the image, just like a normal mirror.

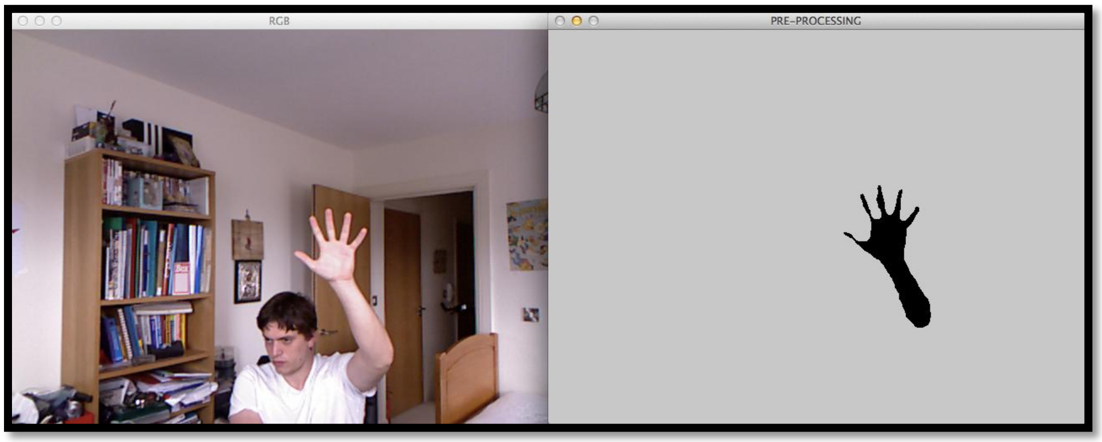


Figure 21: The mirrored matrix with the thresholded values.

Chapter 6

MESH EXTRACTION

Computer vision deals with pixel values translated in points, vectors, matrices but all have to do with the geometry of the object. The word “mesh” is more suitable to define a polygonal model but it was used here because what the Kinect detects is just a line shape that resembles a hand.

At this point the mesh of the hand will be retrieved, as it would help with the formation of the Region of Interest (ROI) and with the elimination of noise as it will be able to remove the shapes with really small area when it comes to skeletonization.

6.1 Canny Edge Detection

In 1986, John F. Canny created an algorithm to detect all the edges inside an image. His algorithm is meant to recognize all the real edges, to have good localization and should only detect the same edge once as well as not confusing noise for an edge.

The algorithm follows certain steps to acquire the accurate edges:

- Noise reduction
- Intensity gradient
- Non-maximum suppression
- Edge trace through image and hysteresis threshold

The canny edge detection considers edges to be vectors in any directions and it uses four filters to recognize any edges diagonally, vertically and horizontally. (Wikipedia 2012) "The edge detection operator (Roberts, Prewitt, Sobel for example) returns a value for the first derivative in the horizontal direction (Gx) and the vertical direction (Gy)". The intensity gradient therefore can be computed by the following:

$$\mathbf{G} = \sqrt{{\mathbf{G}_x}^2 + {\mathbf{G}_y}^2}$$

$$\mathbf{\Theta} = \operatorname{arctan}\left(\frac{\mathbf{G}_y}{\mathbf{G}_x}\right).$$

Using the Canny edge detection function provided by OpenCV, the edges of the thresholded image can be detected (Figure 22).

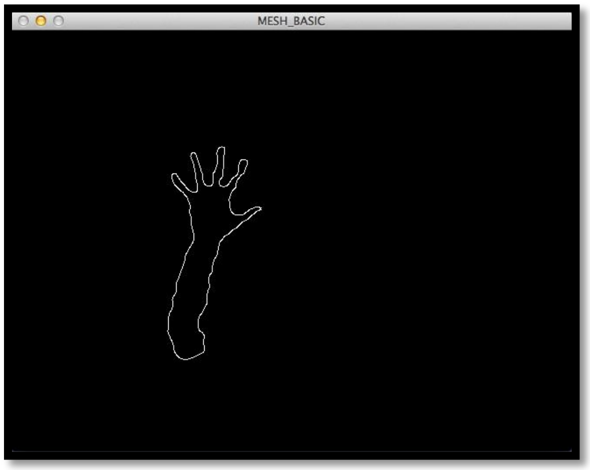


Figure 22: Canny Edge Detection.

6.2 Contours

The edge detection function does not allow for any control over that shape nor does it provide any particular information. All it returns are the pixels that define the edges in a binary manner. Contours are very important in computer vision since they allow for the manipulation of a shape. A contour is a set of points in a sequence, thus being declared as a vector list of points. According to Bradskli and Kaehler (2008) "Contours are represented in OpenCV by sequences in which every entry in the sequence encodes information about the location of the next point on the curve". Since the canny function returned pixel points for the edges of the hand as a binary image, the contours would be efficiently retrieved (Figure 23).

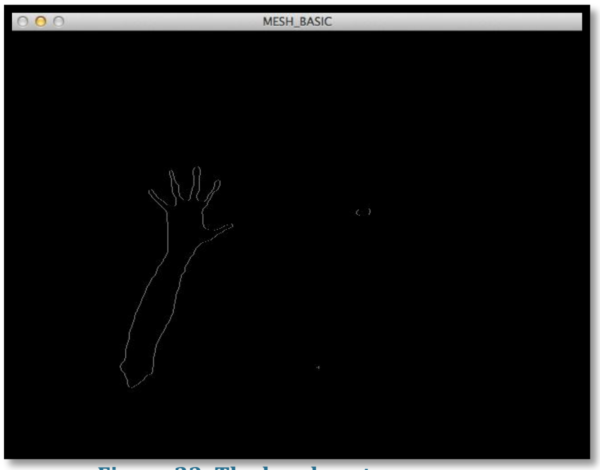


Figure 23: The hand contours.

It was noticed that there was a lot of noise in the image and the edge detection algorithm was picking up small areas that were not there. In order to remove it from the image, the area of the hand contour was found and a filter was added that would only draw the contours with big enough area (Figure 24).

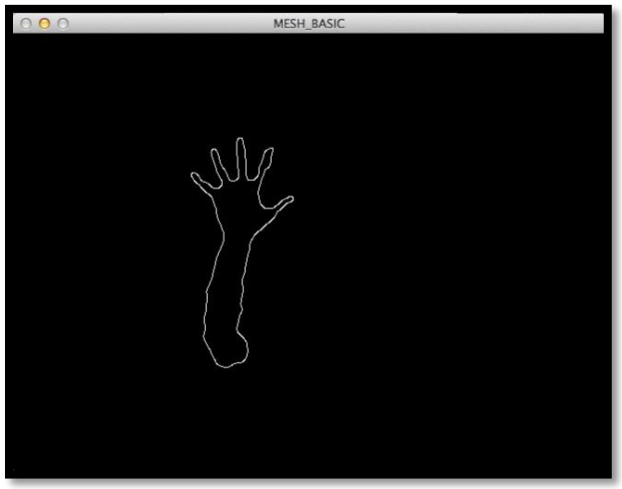


Figure 24: The contours drawn without the noise.

6.3 Circular ROI (Region of Interest)

Some noise that flickers every other n frames can cause big problems in the skeletonization process. Thus a region of interest was formed to capture most of these scattered noises and remove them.

An ROI is essentially a rectangle that surrounds the object's mesh. OpenCV facilitates two types of rectangles that can be used as ROIs. The first is a simple rectangle (Figure 26), which has its origin on the top left corner and can only be drawn with straight lines. The second ROI is a rotated rectangle (Figure 26), much more functional than the simple one but it takes more computational time.

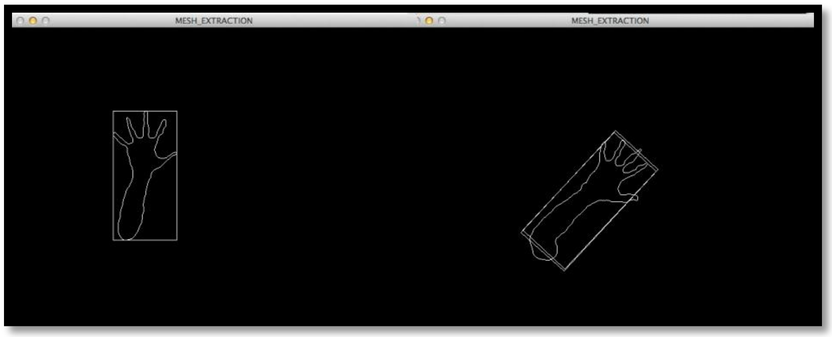


Figure 26: Simple ROI. Rotated ROI.

A rotated ROI will be used in order to construct a circle (Figure 25) around the hand. This method will be useful to eliminate any noise that got through the previous filter. The circular ROI will only surround the contours with six or more points while only the ones with big enough area are drawn.

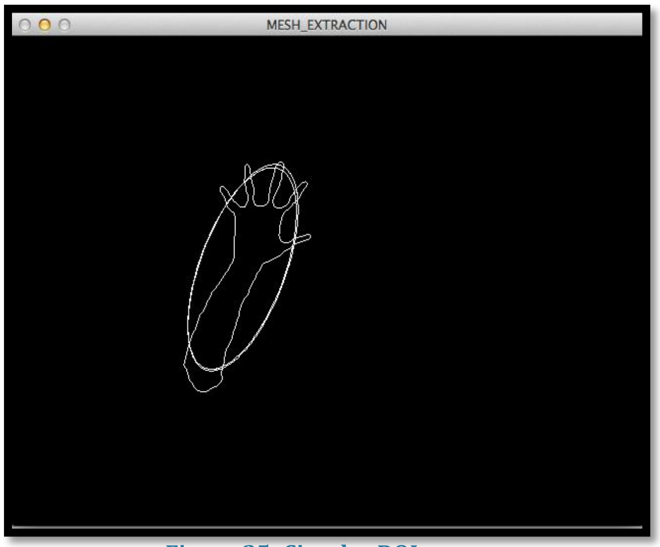


Figure 25: Circular ROI.

Currently the circle surrounds the outer points of the hand mesh and when the skeleton is extracted the ROI will cover the index finger causing the skeleton to lose track of the fingers.

For this purpose the ROI was scaled down (Figure 27) to cover only the inside of the mesh. Also in order to remove the unwanted detections, only the ROIs that passed the filtering would be drawn.

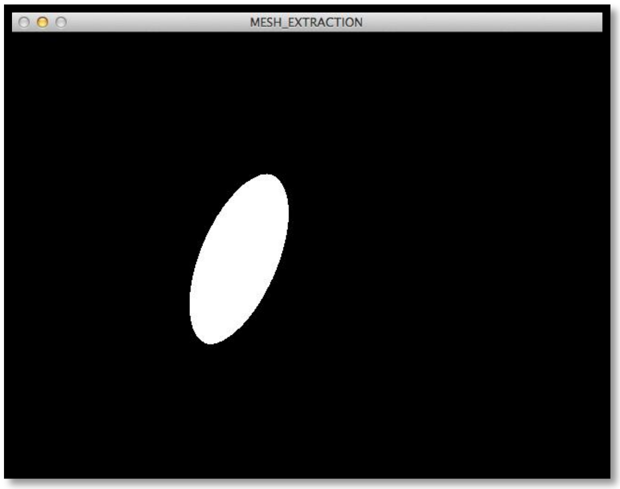
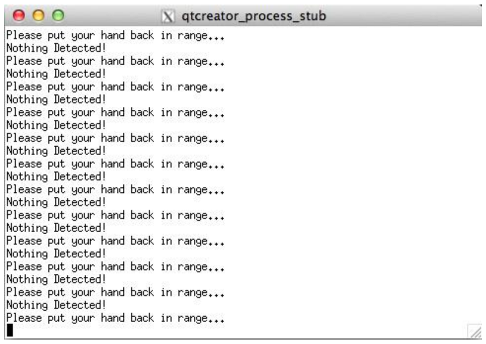


Figure 27: Circular ROI on the new matrix.

6.4 Conclusion of Mesh Detection

Both hand mesh and circular ROI can be accessed any time during the program. They will be useful when constructing the mask area for the 3D skeleton and when eliminating most of the palm noise in the 2D skeleton. Additionally, the detection of the ROI is used to determine whether the user has gone out of range. Instead of crashing when no objects are detected, the program waits for the user to place his/her hand back in range for 1 minute before closing.

Overall, the hand mesh and its circular ROI are both efficient, accurate and all the noise from that image has been removed securing the accuracy of the joints.



```
Please put your hand back in range...
Nothing Detected!
Please put your hand back in range...
Nothing Detected!
Please put your hand back in range...
Nothing Detected!
Please put your hand back in range...
Nothing Detected!
Please put your hand back in range...
Nothing Detected!
Please put your hand back in range...
Nothing Detected!
Please put your hand back in range...
Nothing Detected!
Please put your hand back in range...
Nothing Detected!
Please put your hand back in range...
Nothing Detected!
Please put your hand back in range...
Nothing Detected!
Please put your hand back in range...
Nothing Detected!
Please put your hand back in range...
```

Figure 28: Waiting 1 min for the user to place his/her hand back in range.

Chapter 7

SKELETONIZATION

Skeletonization is the process of representing the actual mesh using a set of lines. In many cases it can be retrieved using topological thinning which essentially uniform scales the mesh until it becomes thin as a line.

The most important attributes of skeletonization are its connectivity, accuracy, stability, topology, length and direction.

In this thesis, topological thinning will be used to bring the hands mesh at a point where the contours will not merge, the Euclidean distance transform will be used to calculate the distance of the points in the contour, the Laplace operator will construct the line and finally the probabilistic Hough line transform will be used to improve the skeleton's connectivity.

7.1 Finding Contours

At this point it is essential to begin with a new set of contours in order to avoid any conflictions with the previous detections. For the detection of the contours (Figure 29), only the thresholded matrix is needed and a few median and Gaussian blurs to smooth the mesh.

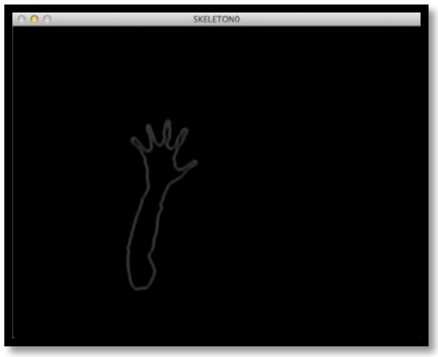


Figure 29: New set of contours based on the threshold values of the matrix.

7.2 Mesh Erosion / Thinning

Thinning is defined as the transformation of the original mesh to one more simplified yet retaining the original topology. The goal is to reduce the hand a really slim mesh while trying to keep the original proportions and shape.

Like any other morphological operation the most important factor besides the binary image that determines its accuracy and detail is the structuring element. According to Fisher R et al. (2003), "The structuring element consists of a pattern specified as the coordinates of a number of discrete points relative to some origin. Normally Cartesian coordinates are used and so a convenient way of representing the element is as a small image on a rectangular grid" (Figure 30).

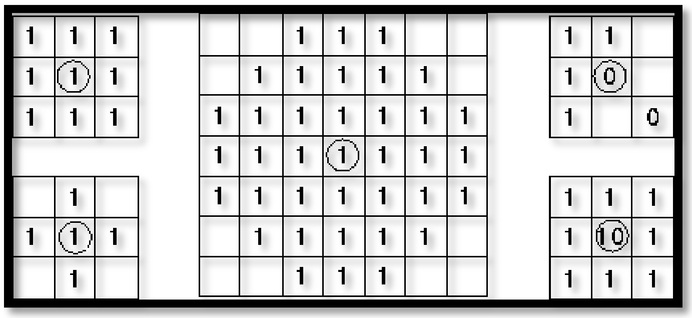


Figure 30: Different structuring elements of various sizes.

The thinning method works by taking the original binary image and uses the hit-and-miss transform to determine its structuring element. In contrast to a simple structuring element, which only uses the foreground pixels (1), the hit-and-miss transform contains both 0 and 1 values (Figure 31) indicating the background and foreground pixels respectively.

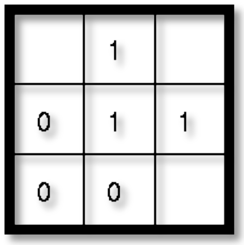


Figure 31: hit-and-miss transform grid.

Following it subtracts the structuring element from the image:

$$thin(I,J) = I - hit\_and\_miss(I,J)$$

Where **I** is the binary image, **J** is the structuring element and the subtraction is logical.

Once more OpenCV facilitates both functions to get the structuring element from the image matrix and to erode the mesh according to the structuring element resulting to a thinner mesh of the hand (Figure 32).

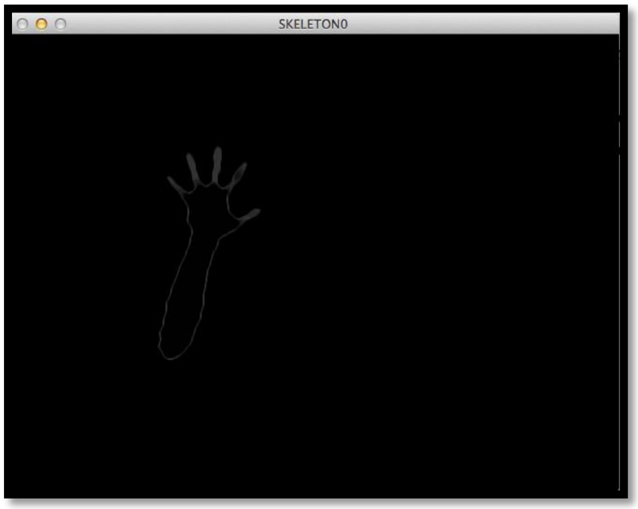


Figure 32: The hand mesh after thinning.

It can be noticed that the fingers have become a lot thinner closing to a line, yet the palm and elbow has little difference than before.

7.3 Euclidean Distance Transform

In short, a distance transform calculates the distance between two points as if they were connected with a straight line.

$$d = \sqrt{dx^2 - dy^2} = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

Where x and y are co-ordinates of the two points in a 2D space.

Using the distance transform will allow for the conversion of the Euclidean space to a metric space. For any two points A(x,y,z) and B(x,y,z), the Euclidean distance between them can be described as the length of the imaginary line that connects them.

Similarly:

$$d(A,B) = \sqrt{(Bx - Ax)^2 + (By - Ay)^2 + (Bz - Az)^2}$$

$$\sqrt{\sum_{i=1}^{n} (B_i - A_i)^2}$$

In computer vision, points are defined as vectors with origin the top left corner of the screen being (0, 0) thus allowing for the calculation of the magnitude between the two points ||AB||.

The distance transform is generally applied on binary images returning a gradient gray-scale image where the white regions are the points closest to the boundary. There are a couple more distance transforms available to be used in order to extract the skeleton of an object such the chessboard and the city-block metrics.

$$D_{Euclid} = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

$$D_{Chess} = \max(|x_2 - x_1|, |y_2 - y_1|)$$

$$D_{City} = |x_2 - x_1| + |y_2 - y_1|$$

OpenCV facilitates a function to compute the distance transform from every pixel to the closest 0 pixel in the binary image matrix, including the Euclidean Distance Transform (CV_DIST_L2). Figure 33 displays the human body using the EDT on the previous binary image.

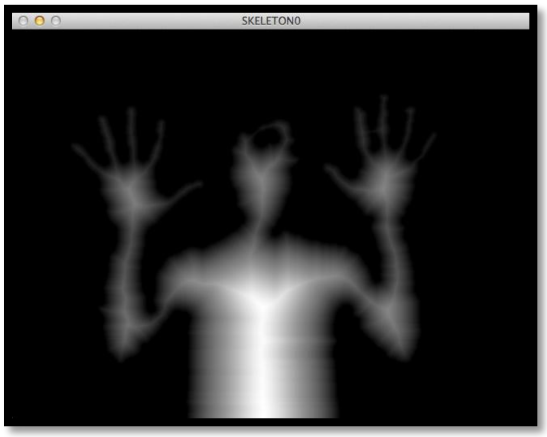


Figure 33: The Euclidean distance transform.

The thinning process decreased the amount of misdetections and false calculations of the distances.

7.4 Laplace Operator / Laplacian

The Laplace operator was named after Pierre-Simon de Laplace, a French mathematician who focused on celestial mechanics.

The original Laplacian formula is:

$$\Delta f = \sum_{i=1}^{n} \frac{\partial^2 f}{\partial x_i^2}$$

And for a 2D environment in Cartesian co-ordinates is:

$$\Delta f = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2}$$

In the previous part, the Euclidean distance transform of the image returned a gray-scale gradient image indicating the skeleton of the mesh. In order to access that skeleton and extract it from the rest of the image, the Laplace operator was used (Figure 34).

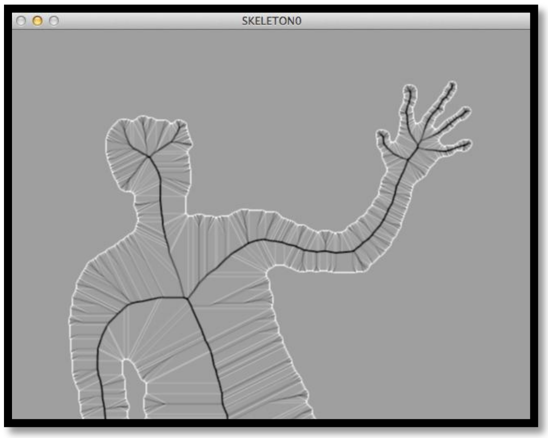


Figure 34: The Laplace Operator.

7.5 Normalize Min/Max

Since the operations are performed on a binary matrix, the EDT and Laplacian cannot be visualized in the image because both convert the original image to a gradient going from white to black while the image matrix can only accept binary pixels (0, 1). In order to be able to see what is happening to the mesh of the hand and keep track of the skeleton, the image matrix had to be normalized.

Although the normalization of the image matrix takes place after the EDT and Laplacian where performed, Figures 33 and 34, have been normalized before hand for demonstration purposes.

Following the matrix was thresholded leaving only the line skeleton (Figure 35).

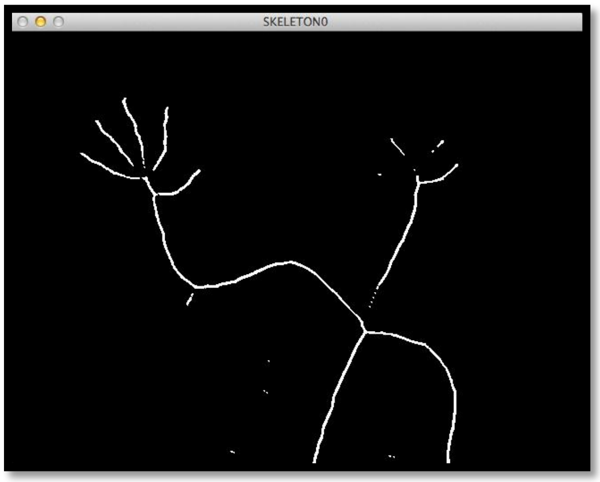


Figure 35: Normalized and Thresholded matrix.

It can be observed that the current line skeleton lacks in connectivity and that small noises are detected on the side of the body.

7.6 Mesh Dilation

The dilation works at the exact opposite of erosion, which is used for the thinning. In order to increase the line skeleton's connectivity it was suggested that the line defined in the matrix needed to become stronger and bigger. With the use of a structuring element, similar to erosion, mesh dilation enlarges the size of the skeleton (Figure 36).

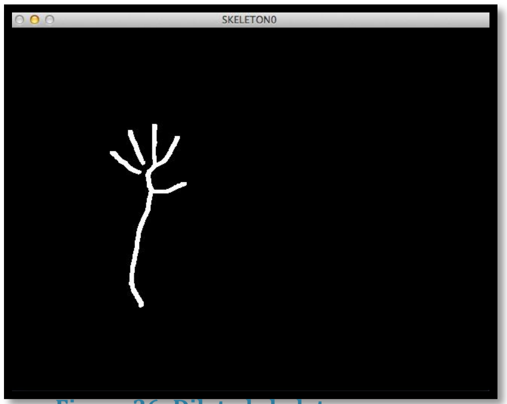


Figure 36: Dilated skeleton.

Although the dilation manages to fill in most of the spaces while preserving the shape of the original skeleton, it is still noticeable that there are a few gaps around the finger area.

7.7 The Probabilistic Hough Line Transform

The Probabilistic Hough line transform is derived from the simple Hough Transform, which detects all possible lines that can pass through a point and if a set of points in the binary image are positioned collinearly, then the line is created.

According to Stephens (1991) "The Probabilistic Hough Transform H(y) is defined as the log of the probability density function of the output parameters, given all available input features".

The Hough line transform formula is the following:

$$H(y) = \sum_{i=1}^{n} \ln[p(x_i \mid y)] + \ln[p(y)] + C$$

Where p(y) is a uniform probability distribution and C is a constant.

The PHLT will work better with a binary image, as there is limited noise and confusion. It will detect the points of the skeleton that form a line, for instance the fingers, including the first point of the other line, that being the palm joint, filling the gap between them (Figure 37).

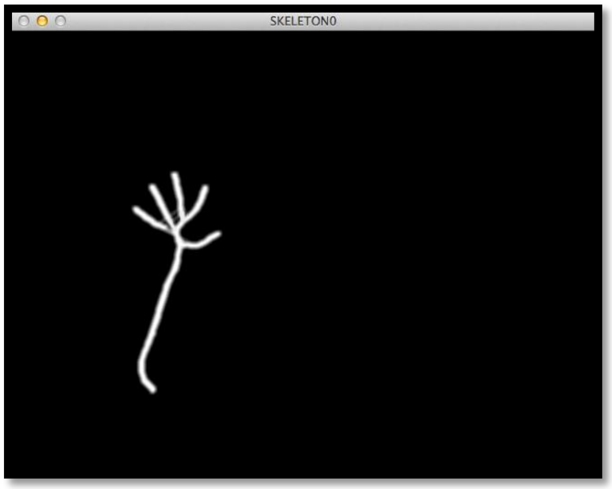


Figure 37: The connected skeleton with the PHLT.

7.8 Conclusion of Line Skeletonization

Overall the line skeleton was extracted using thinning methods, the Euclidean distance transform, the Laplace operator and the probabilistic Hough lines transform. It is accurate, efficient, noise free, stable and the computational cost is low (Figure 38, 39).

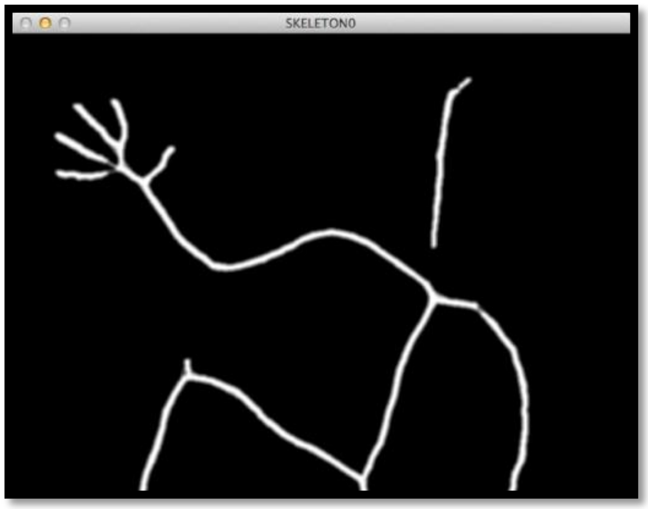


Figure 38: Full body line skeletonization.

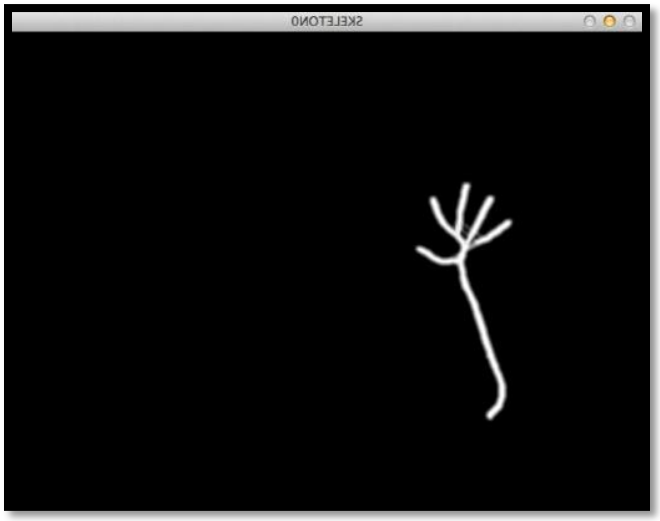


Figure 39: Hand skeletonization.

Chapter 8

2D Skeleton tracking

The line skeleton obtained is a representation of the hand mesh, but its current state does not support accessibility or control. Skeleton tracking is the method of detecting certain points on the line skeleton and use them as separate controllers.

In this chapter, the points will be detected using a corner detection algorithm, the convex hull of the points will be constructed along with a filtering method to remove the noise and then using geometrical equations the elbow, wrist, palm and finger tips will be tracked while a simple representation of the joints using lines will be demonstrated. Finally a short conclusion identifying the limitations and advantages of this technique will be discussed.

8.1 Harris Corner Detection

Generally a corner is defined as a point in the image which variation gradient is very high. The corner is detected by projecting on a u, v area a patch with x, y co-ordinates which will identify the variation using the formula bellow:

$$E(u,v) = \sum_{x,y} w(x,y)[I(x+u,y+v) - I(x,y)]^2$$

Where w(x, y) is the patch and I is the intensity.

Using the binary image of the line skeleton and OpenCV's function cornerHarris, the corners of the hand can be detected accurately (Figure 40).

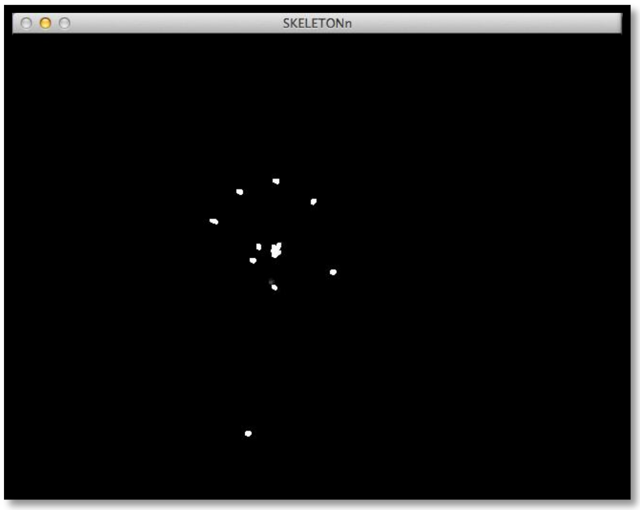


Figure 40: Corner Harris detection.

After the detection of the corners, the points were dilated using a structuring element in order to become more noticeable. It can be observed that there is a lot of noise in the palm region due to the many corners of the hand. This effect could jeopardize the stability of the skeleton and since only one point is needed to manipulate the palm, this noise needed to be cleared out.

8.2 Convex Hull

An object's convex hull is defined as the smallest possible area that contains that object. Convex hulls are constructed by straight lines that go through the maximum co-ordinates of the object edges. A similar approach occurs for any set of points in an image. According to Wikipedia (2012) "For a point set S, its convex hull is the convex combination of its points:

$$\left\{ \sum_{i=1}^{k} \alpha_i x_i \,\middle|\, x_i \in S,\ \alpha_i \in \mathbb{R}^+ \cup \{0\},\ \sum_{i=1}^{k} \alpha_i = 1 \right\}.$$

A convex combination (in contrast to an affine combination) requires all coefficients, α_i, to be non-negative".

In this case the convex hull will be used to surround each point corner separately allowing for better area check. It will not make much difference to the fingers or the elbow but as far as the palm area is concerned, the convex hull will encompass all the points that collide and the ones with really small distance (Figure 41) in order to be removed accurately.

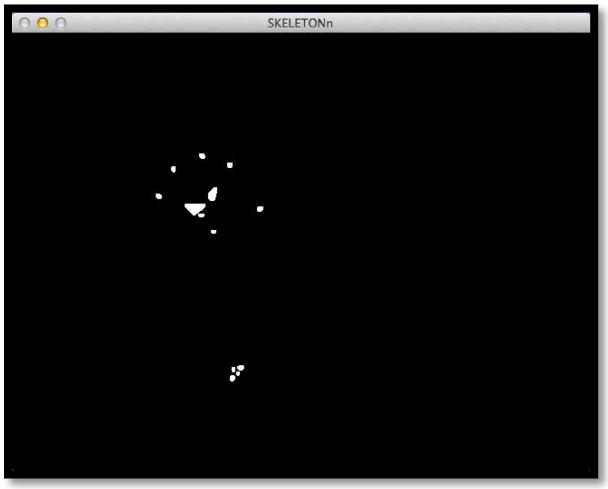


Figure 41: Convex hull on points.

8.3 Filtering the Hull Areas

In order to dismiss any big chunks of hulls detected in the palm region, the area must be computed for each hull and if it is bigger than a certain size that defines a finger, then that chunk must be removed. For that to succeed, a double layered filter method took place right after the detection of the hulls.

The first layer was given the size of the fingers hulls and anything that was bigger than that got erased. The remaining hulls where detected as contours and were assigned to a temporary rectangle ROI array so that each point could be accessed separately.

The second layer compared the size of each ROI to the maximum ROI of the fingers so that if any points that would pass through the first layer, would get stopped in the second (Figure 42).

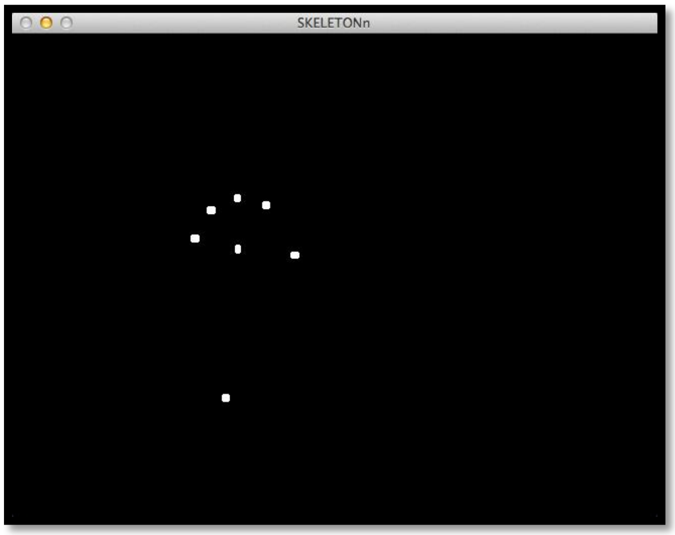


Figure 42: Filtered hulls.

It is observed that most of the noise in the palm area has been erased apart from a couple of small bits that will be used as references to the palm joint later.

8.4 First Object Detection

At this point the contours remaining in the image matrix are used as reference to the skeleton controller positions.

In computer vision, the points tend to be detected from bottom up, meaning in an array

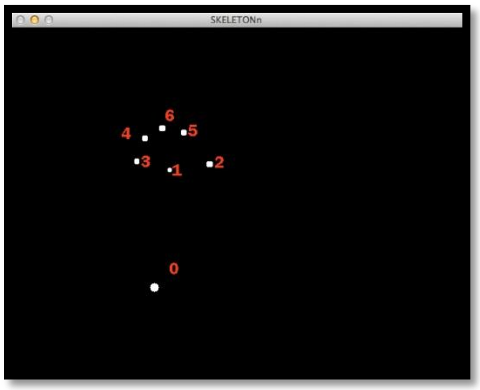


Figure 43: Detection in computer vision.

of points, 0 will be the most bottom point aka the elbow (Figure 43).

In order to isolate the first object from the rest of the points, initially the first contour of the point array was marked with a circle as shown in Figure 44. Then the minimum and

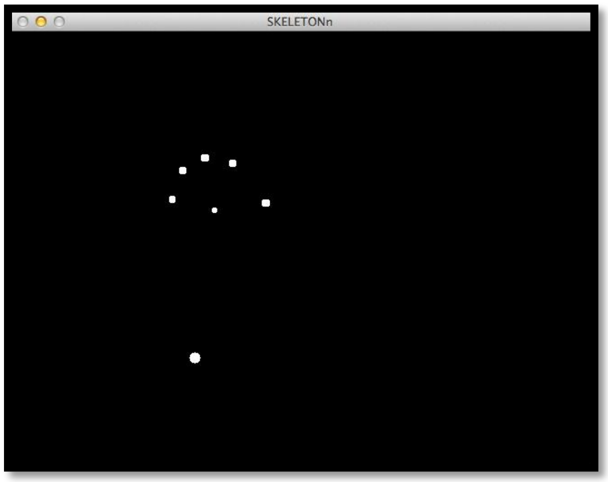


Figure 44: The circle indicates the first contour in the point array.

maximum points on the x-axis of the entire array were determined using the first contour as the initialization value of the min/max.

$$min_x < p_{(i)x} < max_x$$

It was also noticed that some noise existed near the elbow region which affected its stability. This was handled by a virtual locator positioned in that area while the rest of the points were removed.

8.5 The Elbow

The elbow according to the discussion in Chapter 4 will be the starting point of the virtual skeleton, thus it should be isolated from all other points. So far, the elbow is represented with a locator assigned to the first contour in the point array but as soon as the hand rotates more than 90 degrees the locator jumps to the "thumb" or the new first contour of the array. For that the min/max values were found to distinguish the elbow point according to both position and angle.

8.5.1 Joint Orientation

Since in this application there is no shoulder detection, the elbow's rotation had to be estimated according to simple movements. The minimum and maximum points were used to assign the elbow controller to the correct point according to the hand's angle (Figure 45).

There were three conditions indicating the position of the elbow. Should these condition

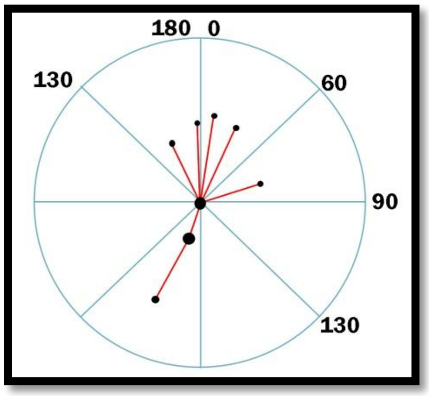


Figure 45: angle of hand to determine the elbow.

were exceeded, the hand would appear as broken. Let R be the hand's angle of rotation:

1) If $R < 60^0$, then the elbow is the first point.
2) If $60^0 < R < 130^0$, then the elbow is the MIN point on the x-axis.
3) If $R > 130^0$, then the elbow is the MAX or first point on the x-axis.

Finally the elbow joint was indicated in the image as a new circle ROI (Figure 46) and its position will be used to detect the palm joint.

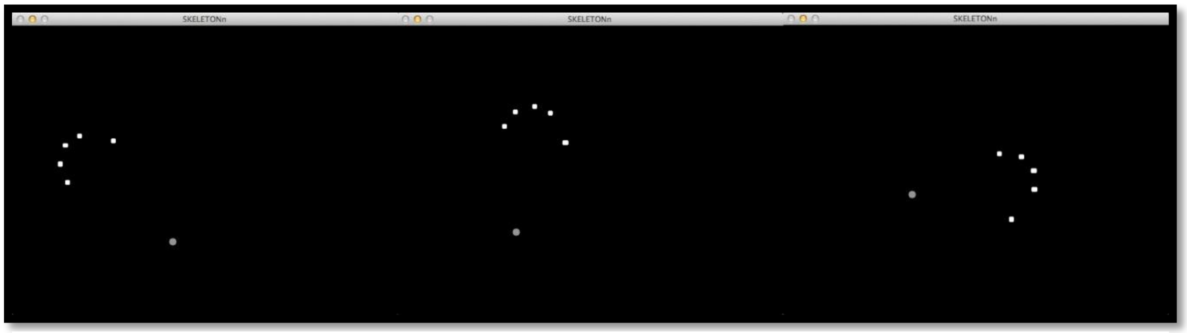


Figure 46: Gray elbow locator at different hand rotations.

8.6 The Palm

The circular ROI from the mesh extraction method helps to determine the position of the palm. Initially the ROI was used to cover any noise left in the palm region clearing out the area for misdetections and confusion. Next the center of the hand's ROI along with the elbow controller where used to position the controller of the palm (Figure 47).

Let P be the center of the palm:

$$P = (2 \times ROI_c) - \frac{E_c + ROI_c}{2}$$

Where ROI_c is the center of the mesh and E_c is the center of the elbow.

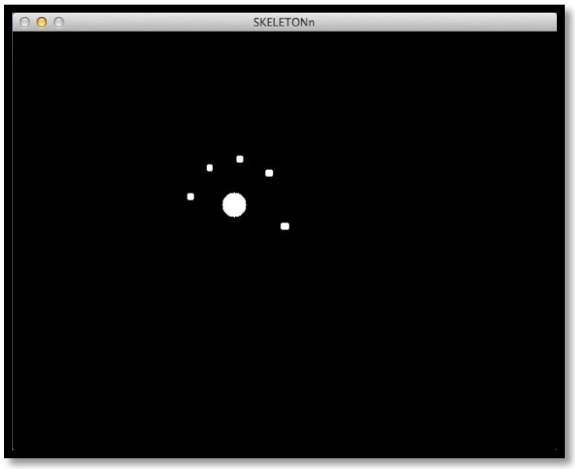


Figure 47: The palm controller.

8.7 The Wrist

In this thesis the wrist is used later in the 3D skeleton extraction to calculate each finger's angle when they come in front of the palm. It has no other use since the corners to detect the wrist were not accurate enough and were very sensitive to noise. Thus an estimated position for the wrist (Figure 48) was added.

Let W be the center of the wrist:

$$W = \frac{\frac{E_c + P_c}{2} + P_c}{2}$$

Where P_c is the center of the palm and E_c is the center of the elbow.

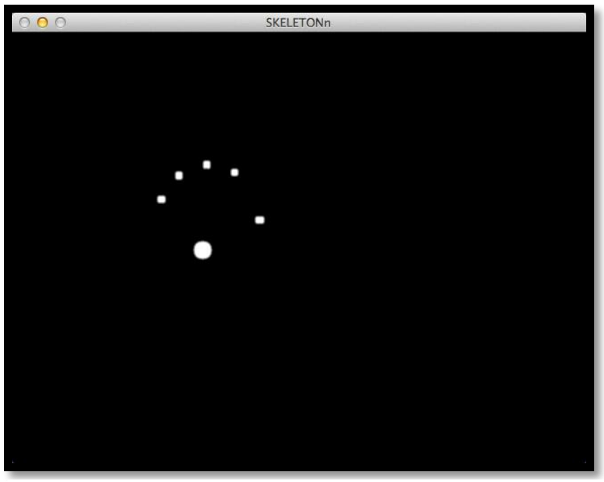


Figure 48: The wrist controller. (Estimated position)

For demonstration purposes the controllers of the palm and elbow have been hidden to focus on the wrist.

8.8 The Finger Tips

At this point the finger tips have not been identified and are only used as a five particle system stored in a vector container whose size is determined by the number of objects in the image matrix. The elbow, wrist and palm points have been hidden and only their locators are shown.

8.9 Visualization of the 2D skeleton

The gathering of the elbow, wrist, palm and fingers was added in a new image matrix with colours and lines indicating the connected joints. It is a simple visualization of the skeleton formed in order to be later on used by the 3D detection.

Figure 49 demonstrates the joints of the 2D skeleton, with the Elbow being the green locator, the wrist is the purple one, the palm is the red locator and the fingers are distinguished with all given a yellow colour.

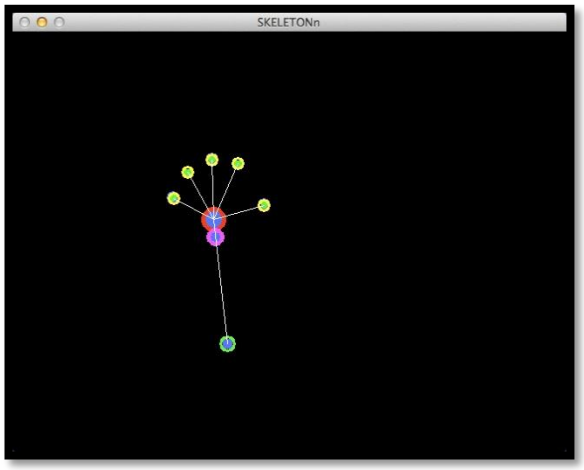


Figure 49: Final 2D virtual skeleton.

8.10 Conclusion and Limitations

In previous cases the 2D skeleton was formed using connectivity defects and the convex hull of the hand (see OpenCV 2.4 documentation). This was cheap towards the computational cost and efficient for some applications. Yet it is unstable when motion is applied and very inaccurate as the fingers are detected only when they collide with the convex hull.

The proposed method for developing a 2D skeleton solves a lot of the issues as it is based on the detection points of the line skeletonization. At this point the skeleton is accurate, efficient, and cheap and does not lose track of the fingers unless they enter the palm region, due to self-occlusion.

The limitations with this technique are:

- Some misdetections occur when the hand is in motion due to Kinect's small frame rate as explained in chapter 3.3, resulting to a sensitive skeleton.
- The Euclidean distance transform will create a finger (branch on the line skeleton) as soon as the mesh of the palm forms a big blob (Figure 49.1).
- Self-occlusion remains (Figures 49.5 – 49.8).

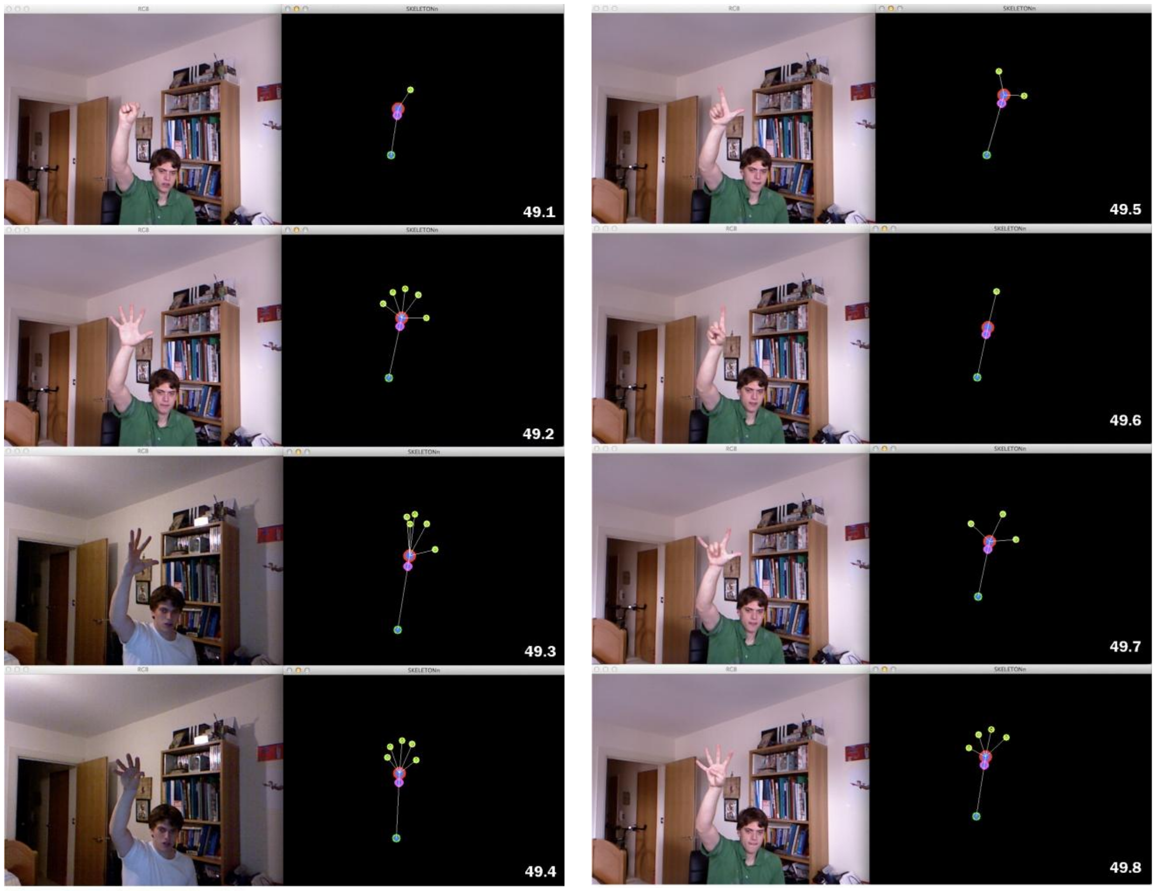


Figure 50: Final Results of the 2D skeleton extraction method.

The 3D skeleton extraction will use this method as a reference to locate the joints while additionally it will detect the self-occluded fingers.

Chapter 9

3D SKELETON TRACKING

The idea behind 3D skeleton extraction is its ability to detect the fingers that have been self-occluded by the palm region. Most applications have tried to overcome this problem using motion analysis estimation. The Kinect has the ability to detect the hand's depth with some limitations such as white noise, frame rate, low resolution and limited range of detection.

In this chapter, a new approach to solve self-occlusion will be presented along with several techniques to improve the Kinect's detection limitations. Once the fingers are detected the two skeletons will be merged to one using only each best attributes to construct the final 3D skeleton of the hand.

9.1 Pre-Processing the depth

In order to process the Kinect's depth stream and retrieve the distance of the hand from the camera, an entirely new set of matrices is needed. The pre-processing stage is very similar to the previous technique with the only difference being that in this method, the raw data from the depth stream is accessed. To accomplish that, the original image matrix needs to be converted to a 16-bit depth. Additionally, the image had to go though median blurs smoothing the output and finally the matrix was flipped similar to chapter 5.4 (Figure 51).

It was mentioned previously that it was not possible to get a visual of the depth stream unless it was an 8-bit depth and in order to access it's the raw data, the Kinect needs to be set at a 16-bit depth. To solve this conflict, two different matrices were constructed, one that holds the depth values and one used for visualization purposes.

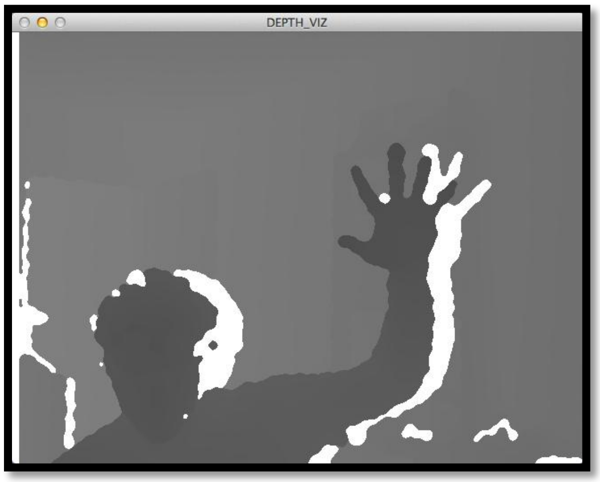


Figure 51: The pre-processing of the new depth.

9.2 Matrix Separation

The two matrices used in this method are the "DepthZval" which stores the raw data from the Kinect's depth sensor and the "DepthZvis" used for drawing, which is essentially a clone of the DepthZval converted to an 8-bit depth.

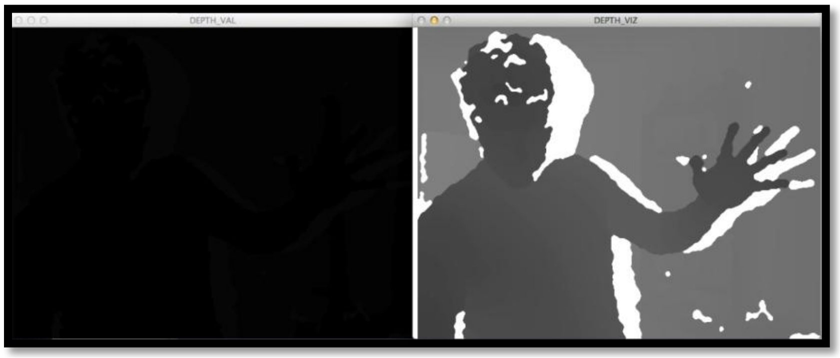


Figure 52: The 16-bit depth Values is on the left and the 8-bit depth Visuals are on the right.

The division of the image to values and visuals (Figure 52) would benefit by preventing any conflicts that could have been presented when manipulating the controllers.

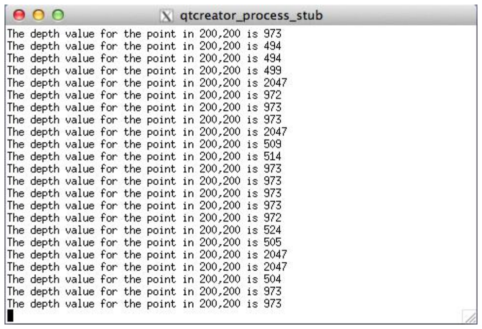
9.3 Depth Values Conversion

Using the 16-bit depth, a 2D array was declared which took the column and row number as its parameters and returned the depth value of the pixel. To return all the values of the image, a nested for-loop which went through each pixel was performed and the depth value of each pixel was returned by:

$$depthPixel[i][j] = DepthZval.at < unsigned\ short > (j, i)$$

Figure 53 prints the depth value for the point (200,200). These values are called raw depth data and represent the Kinect's sensitivity levels.

To convert them into something meaningful such as meters or centimeters, Burrus (2012) presented a conversion formula where if **RD≤2047**:



```
qtcreator_process_stub
The depth value for the point in 200,200 is 973
The depth value for the point in 200,200 is 494
The depth value for the point in 200,200 is 494
The depth value for the point in 200,200 is 499
The depth value for the point in 200,200 is 2047
The depth value for the point in 200,200 is 972
The depth value for the point in 200,200 is 973
The depth value for the point in 200,200 is 973
The depth value for the point in 200,200 is 2047
The depth value for the point in 200,200 is 509
The depth value for the point in 200,200 is 514
The depth value for the point in 200,200 is 973
The depth value for the point in 200,200 is 973
The depth value for the point in 200,200 is 973
The depth value for the point in 200,200 is 973
The depth value for the point in 200,200 is 972
The depth value for the point in 200,200 is 524
The depth value for the point in 200,200 is 505
The depth value for the point in 200,200 is 2047
The depth value for the point in 200,200 is 2047
The depth value for the point in 200,200 is 504
The depth value for the point in 200,200 is 973
The depth value for the point in 200,200 is 973
```

Figure 53: Raw depth data for point (200,200).

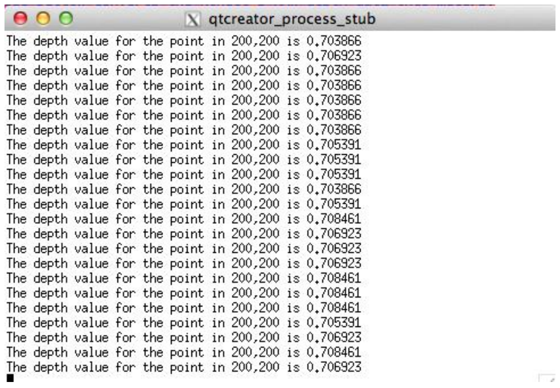
$$depth = \frac{1}{[RD \times (-0.0030711016)] + 3.3309495161}$$

Where ***RD*** is the raw depth value.

Figure 54 displays the same depth value of the point (200,200) in meters.

This use of the depth will allow for the accurate detection of any point in the matrix such as a finger point or even the point right next to it.

Additionally, with the declaration of the 2D array there was no need to convert the values again, saving a lot of time and computations.

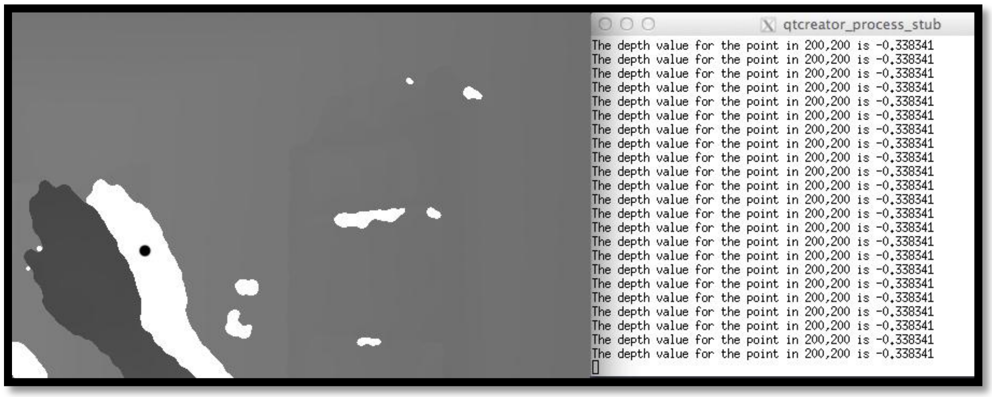


```
qtcreator_process_stub
The depth value for the point in 200,200 is 0.703866
The depth value for the point in 200,200 is 0.706923
The depth value for the point in 200,200 is 0.703866
The depth value for the point in 200,200 is 0.703866
The depth value for the point in 200,200 is 0.703866
The depth value for the point in 200,200 is 0.703866
The depth value for the point in 200,200 is 0.703866
The depth value for the point in 200,200 is 0.705391
The depth value for the point in 200,200 is 0.705391
The depth value for the point in 200,200 is 0.705391
The depth value for the point in 200,200 is 0.703866
The depth value for the point in 200,200 is 0.705391
The depth value for the point in 200,200 is 0.708461
The depth value for the point in 200,200 is 0.706923
The depth value for the point in 200,200 is 0.706923
The depth value for the point in 200,200 is 0.706923
The depth value for the point in 200,200 is 0.708461
The depth value for the point in 200,200 is 0.708461
The depth value for the point in 200,200 is 0.708461
The depth value for the point in 200,200 is 0.705391
The depth value for the point in 200,200 is 0.706923
The depth value for the point in 200,200 is 0.708461
The depth value for the point in 200,200 is 0.706923
```

Figure 54: Depth value in meters.

9.4 White Noise Elimination

White noise represents the areas the Kinect cannot see returning negative or nan values (Figure 55). This was very threatening to the project as the skeleton could not be stabilized.

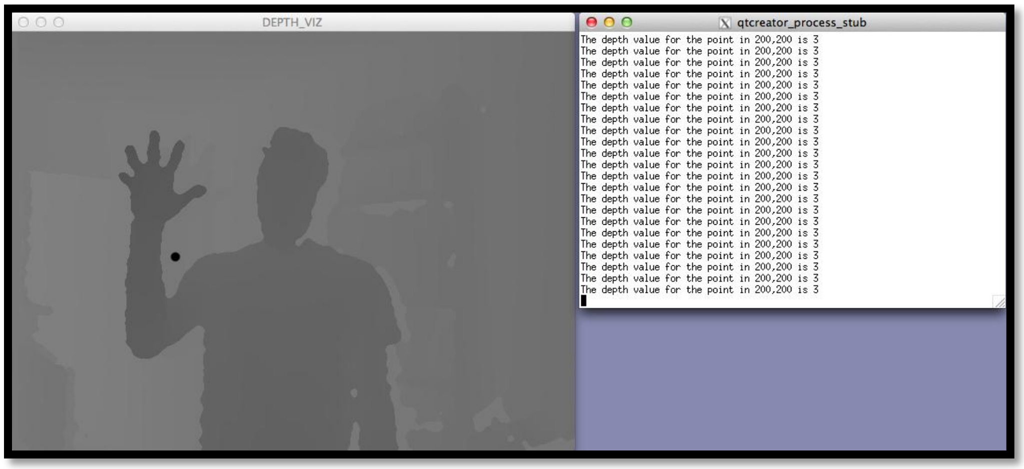


```
The depth value for the point in 200,200 is -0.338341
The depth value for the point in 200,200 is -0.338341
The depth value for the point in 200,200 is -0.338341
The depth value for the point in 200,200 is -0.338341
The depth value for the point in 200,200 is -0.338341
The depth value for the point in 200,200 is -0.338341
The depth value for the point in 200,200 is -0.338341
The depth value for the point in 200,200 is -0.338341
The depth value for the point in 200,200 is -0.338341
The depth value for the point in 200,200 is -0.338341
The depth value for the point in 200,200 is -0.338341
The depth value for the point in 200,200 is -0.338341
The depth value for the point in 200,200 is -0.338341
The depth value for the point in 200,200 is -0.338341
The depth value for the point in 200,200 is -0.338341
The depth value for the point in 200,200 is -0.338341
The depth value for the point in 200,200 is -0.338341
The depth value for the point in 200,200 is -0.338341
The depth value for the point in 200,200 is -0.338341
The depth value for the point in 200,200 is -0.338341
The depth value for the point in 200,200 is -0.338341
The depth value for the point in 200,200 is -0.338341
The depth value for the point in 200,200 is -0.338341
```

Figure 55: Depth returned from white noise. The dot is the point (200,200).

The proper way of solving this problem is to identify the pixel with white noise and fill it with the median value of the eight surrounding pixels but it is very time consuming.

In this case all it was needed was to tell the Kinect that these values have a very high depth. Therefore iterating through each pixel, the white noise was isolated and given a value of 3 meters. Also these pixels were thresholded to roughly fit the assigned distance (Figure 56).



```
The depth value for the point in 200,200 is 3
The depth value for the point in 200,200 is 3
The depth value for the point in 200,200 is 3
The depth value for the point in 200,200 is 3
The depth value for the point in 200,200 is 3
The depth value for the point in 200,200 is 3
The depth value for the point in 200,200 is 3
The depth value for the point in 200,200 is 3
The depth value for the point in 200,200 is 3
The depth value for the point in 200,200 is 3
The depth value for the point in 200,200 is 3
The depth value for the point in 200,200 is 3
The depth value for the point in 200,200 is 3
The depth value for the point in 200,200 is 3
The depth value for the point in 200,200 is 3
The depth value for the point in 200,200 is 3
The depth value for the point in 200,200 is 3
The depth value for the point in 200,200 is 3
The depth value for the point in 200,200 is 3
The depth value for the point in 200,200 is 3
The depth value for the point in 200,200 is 3
The depth value for the point in 200,200 is 3
The depth value for the point in 200,200 is 3
```

Figure 56: The new depth stream.

Overall there was no noise in the depth stream and the skeleton was now working without any false detections. This was a great improvement for the project that prevented numerous errors.

9.5 ROI Formation and Masking

The ROI's formation is very crucial to this project. Instead of searching for points in the entire matrix, it limits the code to the ones located inside that region. It optimizes the algorithm and lowers the computational cost, increasing accuracy and preventing false detections.

Masking the ROI was done by creating another matrix used as the original's inverse and subtracting the ROI from the inverse. It sets everything to 0 values (black colour) and only the objects inside the ROI are viewed.

Figures 57 and 58 demonstrate the ROI and its mask.

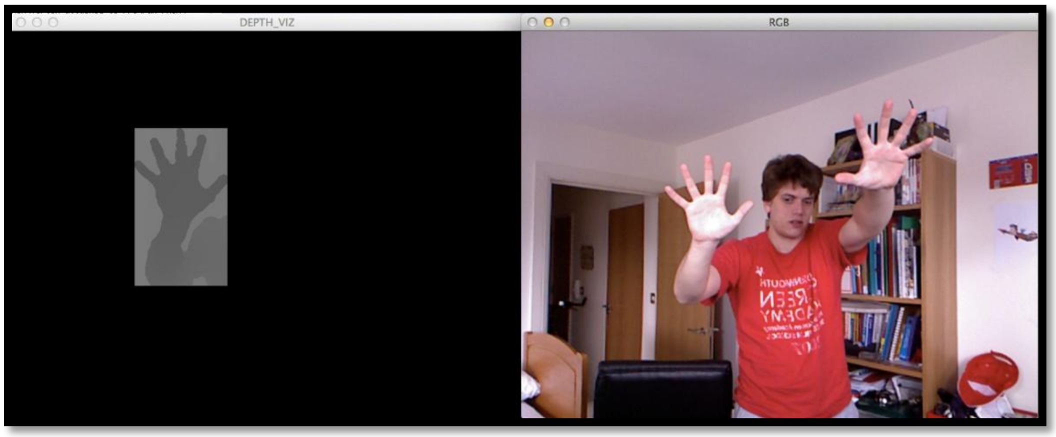


Figure 58: The right hand is NOT affecting the detections. Only left hand is seen.

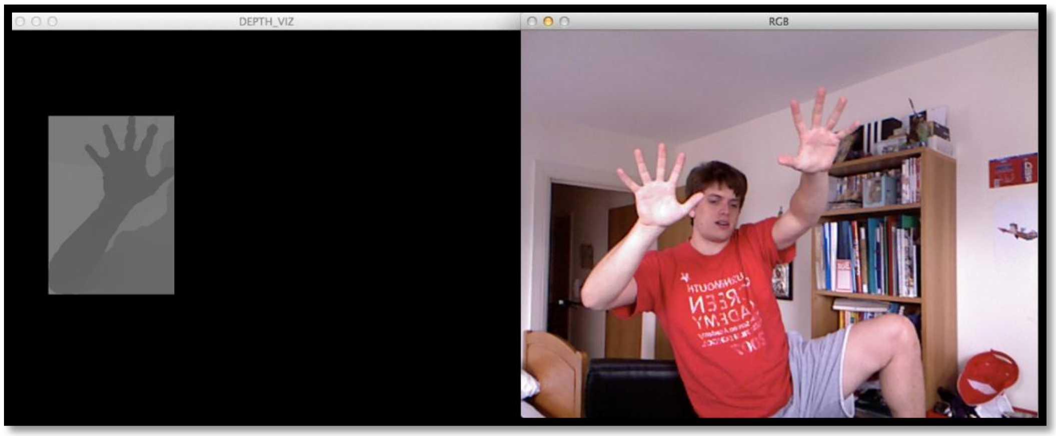


Figure 57: Right hand and Leg are NOT detected.

With the above optimizations such as the white noise removal and ROI formation, the depth stream is clean from most threats to false detections and is ready to build the skeleton.

9.6 Joints Depth Retrieval

At this point the 2D skeleton extraction has x and y co-ordinates but using the pixel array declared previously, their depth position (z axis) was computed.

9.6.1 Elbow Joint

The elbow's z position will be computed by its x and y co-ordinates:

$$E_z = pixelValue[E_x][E_y]$$

Where E is the elbow controller and *pixelValue* is a float. Figure 59 demonstrates the elbow joint with X, Y and Z co-ordinates.

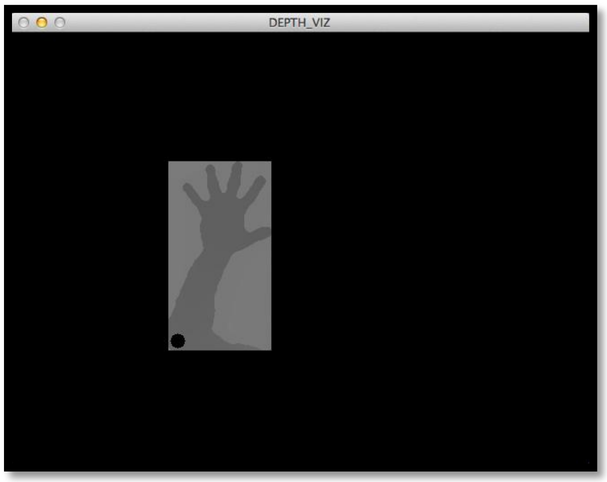


Figure 59: The elbow with x, y, z co-ordinates.

9.6.2 Palm Joint

The palm joint was retrieved in a similar way to the elbow with the only difference that its depth was slightly changed. Anatomically, the palm has a lot of muscles forming small bumps on the hand, and because the depth values are very accurate, these small bumps were detected as objects in front of the palm.

As soon as the z-axis of the palm was formed, it was subtracted by a small value of 0.02 which represents the height of those muscles (Figure 60).

$$P_z = pixelValue[P_x][P_y] - 0.02$$

The goal was to use the palm as a flat surface so that only the fingers would be detected as self-occluded objects.

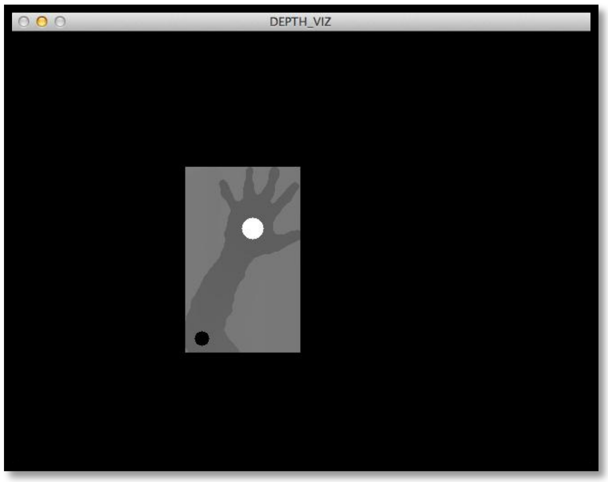


Figure 60: The palm with x, y, z co-ordinates.

The palm and elbow are very accurate and very stable due to the size of the arm.

9.6.3 Wrist Joint

The wrist was retrieves similarly to the elbow joint using the array to return its depth value (Figure 61).

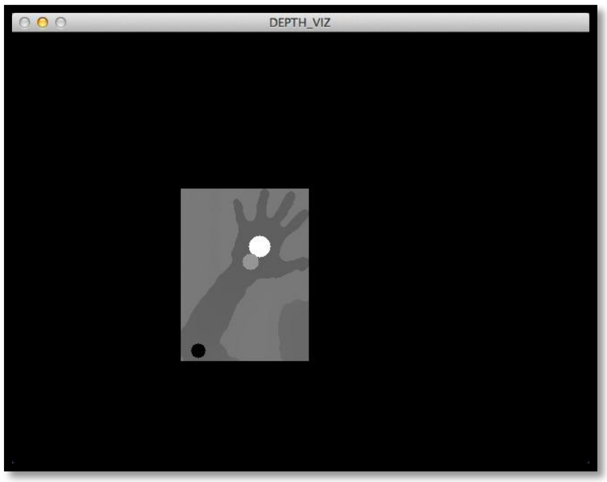


Figure 61: The wrist in the 3D environment.

The locators were created for demonstration purposes. They are not the final controllers of the 3D skeleton.

9.6.4 Temporary finger storage

The fingers have not been assigned to any locators yet, as the next step involves sorting them out according to a counter-clockwise identification. Five variables were created inside the function's scope to process the computations of the fingers. These variables are deleted when the final positions of the fingers are assigned to their according locators.

9.7 The Point4f

OpenCV facilitates several data types that use points as vectors. The most common are cv::Point for 2D points and cv::Point3f for 3D points including all functions that can be performed with those vectors.

A finger in this application is defined by its length, angle and position on the image. Unfortunately the data types in the library do not support these extra attributes, thus a new structure called "Point4f" was created that holds these elements:

```
Struct Point4f
{
        cv::Point3f pos;
        float angle;
        float length;
        bool operator<(const Point4f& _p) const { return angle<_p.angle; }
};
```

This new data structure allows for an easier management of the points and will be used to store the final controllers of the skeleton.

9.8 Constructing the 2D skeleton in a 3D environment

At this point the elbow, wrist and palm have been positioned in the 3D environment but the fingers have not yet been located or identified. In order to compute the angle of the fingers according to the palm, the dot product and magnitude need to be calculated along with the vector of the elbow to the palm.

9.8.1 Elbow to Palm vector and magnitude

To find the vector from the elbow to the palm ($\overrightarrow{EP}$) the elbow vector should be subtracted from the palm by:

$$\overrightarrow{EP} = \vec{P} - \vec{E} = \begin{pmatrix} Px \\ Py \\ Pz \end{pmatrix} - \begin{pmatrix} Ex \\ Ey \\ Ez \end{pmatrix}$$

The magnitude of this vector can be calculated by:

$$\parallel EP \parallel = \sqrt{EP_x^2 + EP_y^2 + EP_z^2}$$

Where EP is the vector from Elbow to Palm, P is the palm and E is the elbow.

This vector and its magnitude will be used as reference to determine the finger angles.

9.8.2 The Points from 2D skeletonization

A Point4f dynamic array was declared to hold the values of the fingers. With the use of a simple ROI as explained in chapter 6.3, the fingers were located in the three dimensional environment and assigned to the advanced finger controller.

```
Point4f *fingerController = new Point4f[contoursSkeleton.size()];
```

The fingers now have x, y and z co-ordinates using the pixelValue depth.

9.8.3 Fingers to Palm vector

The vector from every finger to the palm is calculated by:

$$\overrightarrow{FP_i} = \vec{P} - \overrightarrow{F_i} = \begin{pmatrix} Px \\ Py \\ Pz \end{pmatrix} - \begin{pmatrix} F_i x \\ F_i y \\ F_i z \end{pmatrix}$$

Where $\overrightarrow{FP}$ the vector from fingers to palm, P is the palm and F is the Point4f finger controller. Also each length was computed by:

$$\| FP_i \| = \sqrt{FP_{ix}^2 + FP_{iy}^2 + FP_{iz}^2}$$

The variable “i” is the number of fingers in the container.

9.8.4 Finger Angles

In order to compute the angles, the dot product of the $\overrightarrow{EP}$ and each finger must be calculated:

$$\overrightarrow{EP} \odot \overrightarrow{FP_i} = \begin{pmatrix} EPx \\ EPy \\ EPz \end{pmatrix} \odot \begin{pmatrix} F_i x \\ F_i y \\ F_i z \end{pmatrix} = (EPx \times F_i x) + (EPy \times F_i y) + (EPz \times F_i z)$$

The angle of each finger is computed by the cosine function:

$$\cos \theta_i = \frac{A \odot B}{|A| \times |B|}$$

Where θ is the angle, A is the $\overrightarrow{EP}$ vector, B is the $\overrightarrow{FP_i}$ vector.

Finding the angle and length of each finger to the palm is essential for the counter-clockwise identification.

9.8.5 Identification and Sorting

At this point all the information about the fingers is stored in the Point4f finger controller.

The data structure contains a Boolean operation which is used to sort the fingers from the smallest angle being the thumb, to the biggest being the pinky.

```
std::sort(&fingerController[0].&fingerController[contoursSkeleton.size()]);
```

Each finger was assigned then to the temporary variable.

9.8.6 Final fingers in 3D environment

The 2D skeleton is now fully brought to the 3D environment with each joint having three co-ordinates and the fingers having an angle and a length from the palm (Figure 62).

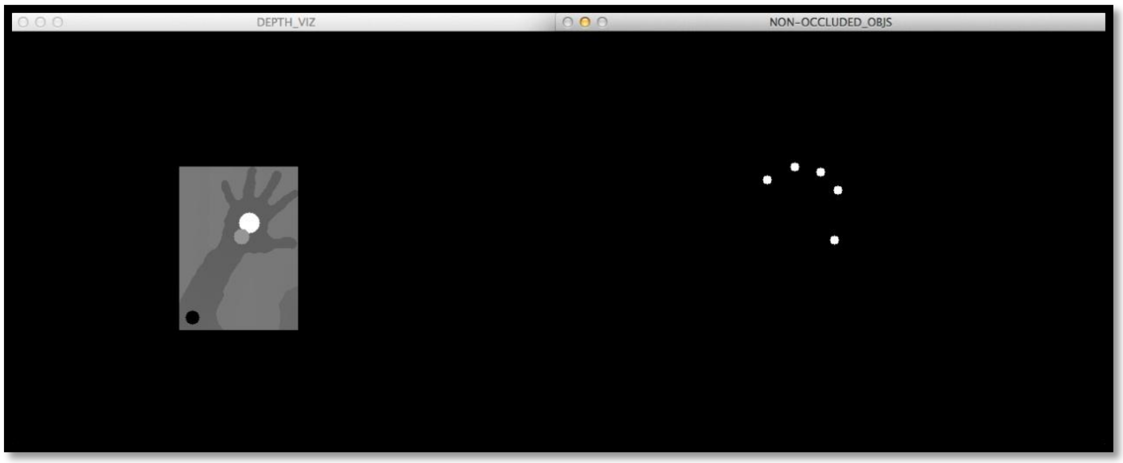


Figure 62: The 2D fingers in the 3D environment with x, y, z co-ordinates.

The fingers were drawn in a new image matrix called “nonOccludedObj” so that they could later on be distinguished from the self-occluded fingers.

9.9 Self-Occlusion

Even though the current skeleton is operating in the 3D environment, it still depends on the 2D skeletonization method to detect the fingers, which means that as soon as the fingers are not detected due to self-occlusion by the palm, they will not be detected in the 3D either.

The proposed method involves an optimized search inside the ROI detecting all the points that have a greater depth value than the palm since that region is the only place that self-occlusion will occur with the fingers.

9.10 Optimization of Search

Since the Kinect's resolution is 640x480 pixels it means that in each frame the program will have to go through 307,200 pixels to detect any objects. This is very expensive and inefficient. Although the ROI constructed earlier lowers the amount of search into only inside its boundaries, the calculations that have to be computed are still enough to slow down the program's efficiency.

9.10.1 Grid Projection

The target was to lower the search even more without dismissing any detail. To accomplish that a set of grids were projected each of a different size to test which one would best fit the criteria.

The table below lists different grids projected along with the calculations they performed in each frame.

Grid Size	Pixels on matrix	Calculations/f	Results
1	640 x 480	307,200	SC / HA
2	320 x 240	76,800	SC / HA
4	160 x 120	19,200	FC / HA
5	128 x 96	12,288	FC / HA
8	80 x 60	4,800	FC / LA
10	64 x 48	3,072	FC / LA

FC=fast computations, SC=slow computations, HA=high accuracy, LA=low accuracy.

According to the table above, the best solution would be to use a grid of a size 4 or 5, but since there was not much difference in the computational cost, it was decided to go with 4 as it gave more detail (Figure 63).

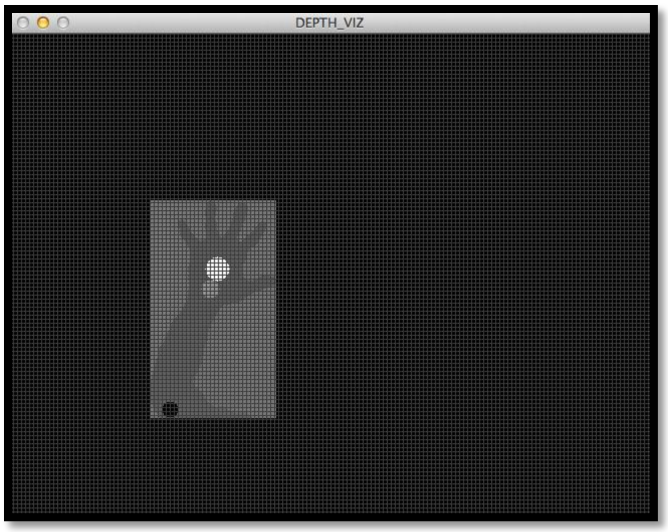


Figure 63: The size 4 grid projected on the image.

9.10.2 Cube ROI

Using the grid and the hand ROI, it was now possible to construct a cubic ROI which would also consider the depth values. That way the search will focus even more than before. In this stage the cube ROI takes the hand ROI x and y co-ordinates and is set to detect all the points only from 0.2 m to 1.4 m depth (Figure 64).

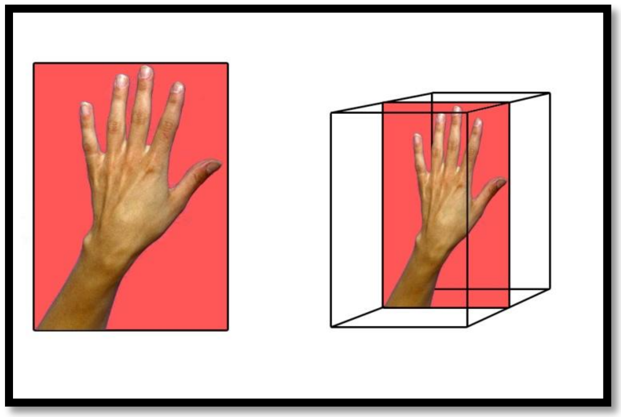


Figure 64: Left is a simple ROI checking all depth. On the Right is the cube ROI checking from 0.2 – 1.4 meters.

This method will also discard the white noise since it was given a value of 3 meters earlier.

9.10.3 Points inside ROI

These optimization methods have really improved the computational time and the algorithm only has to check for a small portion of the image. At this point, all the grid points on the hand are drawn (Figure 65) indicating that the hand is in the correct position.

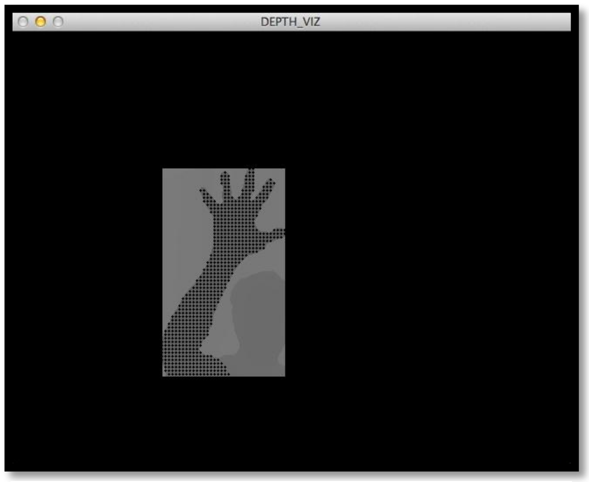


Figure 65: Points on hand.

It can also be observed that although the head is in the frame the points are not drawn on it as it is outside the cube, suggesting less miss-detection.

9.11 Self-Occluded Fingers

Initially all the points on the hand that have a smaller (closer to the camera) depth value than the palm, are indicated with a white circle whereas the rest are black (Figure 66).

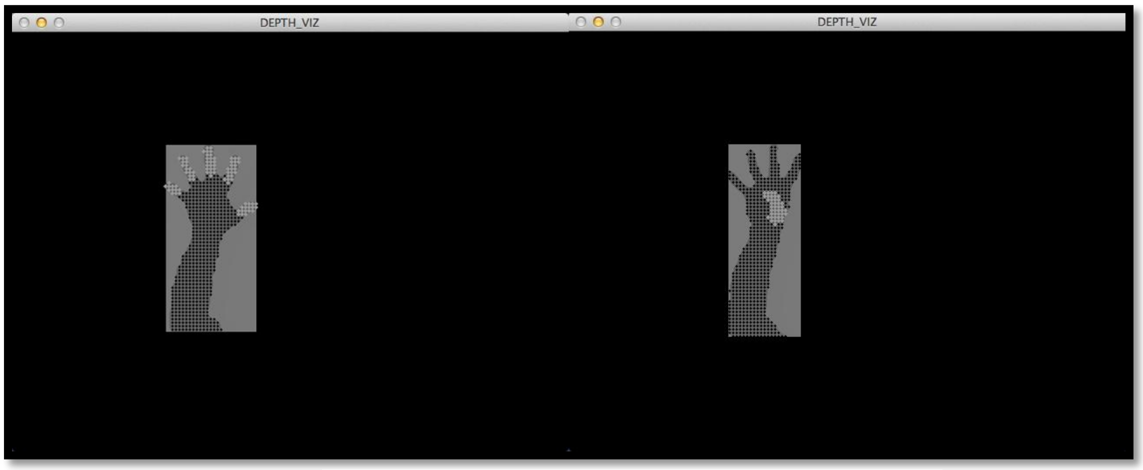


Figure 66: Self-Occluded grid points.

This proves that the points can be detected accurately in the image, but the real task lies with tracking. So far, the algorithm just points out which points on the grid have a smaller depth value from the palm, them being the fingers. Tracking those points with the skeleton is something very difficult to achieve and it is in this part of the project that three attempts were made.

9.11.1 Points on new matrix

The working method involves creating a new matrix called “selfOccludedObj” which will be used to perform all the calculations that would track these points. The first step was to draw the detected points in the new matrix thus isolating them from the rest of the points (Figure 67).

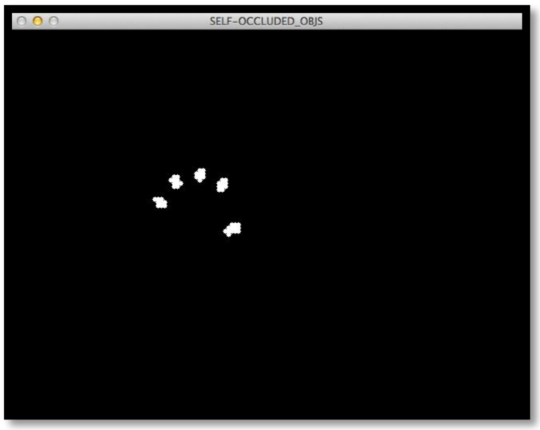


Figure 67: Grid points on new matrix.

9.11.2 Blob Formation

This technique involves using the points as blobs to track their position. It is not the best way to track those points and it has a lot of limitations. Yet it was the only method of the three that gave positive results and due to time constraints it was not possible to develop other methods. Figure 68 demonstrates the points as contours.

Figure 68: Points as 5 contours.

Following, the blobs were assigned to a Point4f array similarly to the previous methods in order for them to be identified.

9.11.3 Elbow to Wrist vector

At this point, when the fingers enter the palm region, the locator of the palm would not be detected, thus the wrist will be used to identify the fingers. To find the vector from the elbow to the wrist ($\overrightarrow{EW}$) the elbow vector should be subtracted from the wrist by:

$$\overrightarrow{EW} = \vec{W} - \vec{E} = \begin{pmatrix} Wx \\ Wy \\ Wz \end{pmatrix} - \begin{pmatrix} Ex \\ Ey \\ Ez \end{pmatrix}$$

The magnitude of this vector can be calculated by:

$$\parallel EW \parallel = \sqrt{EW_x^2 + EW_y^2 + EW_z^2}$$

Where EW is the vector from elbow to wrist, W is the wrist and E is the elbow.

This vector and its magnitude will be used as reference to determine the self-occluded finger angles.

9.11.4 Points Angles and Lengths

Similarly to the previous method, each finger's to wrist vector was determined along with their magnitude. The information will be used to compute the angles using the dot product of the $\overrightarrow{EW}$ by:

$$\overrightarrow{EW} \odot \overrightarrow{FW_i} = \begin{pmatrix} EWx \\ EWy \\ EWz \end{pmatrix} \odot \begin{pmatrix} F_i x \\ F_i y \\ F_i z \end{pmatrix} = (EWx \times F_i x) + (EWy \times F_i y) + (EWz \times F_i z)$$

The angle of each finger is computed by the cosine function:

$$\cos \theta_i = \frac{A \odot B}{|A| \times |B|}$$

Where θ is the angle, A is the $\overrightarrow{EW}$ vector and B is the $\overrightarrow{FP_i}$ vector.

Now that each of the blobs holds similar information to the skeleton, the self-occluded fingers can be sorted and identified.

9.11.5 Identification and Sorting

Since the fingers are stored in a Point4f data type variable, the use of the Boolean operator will sort them from smallest angle to biggest similarly to the other method explained earlier.

```
std::sort(&newFingerController[0].&newFingerController[contoursDepth.size()]);
```

9.11.6 Final self-occluded fingers

The sorted fingers are assigned to the temporary finger storage similarly to the skeleton so that if one method loses track of a finger, the other will detect it. At the moment the two methods work individually and accurately (Figure 69, 70).

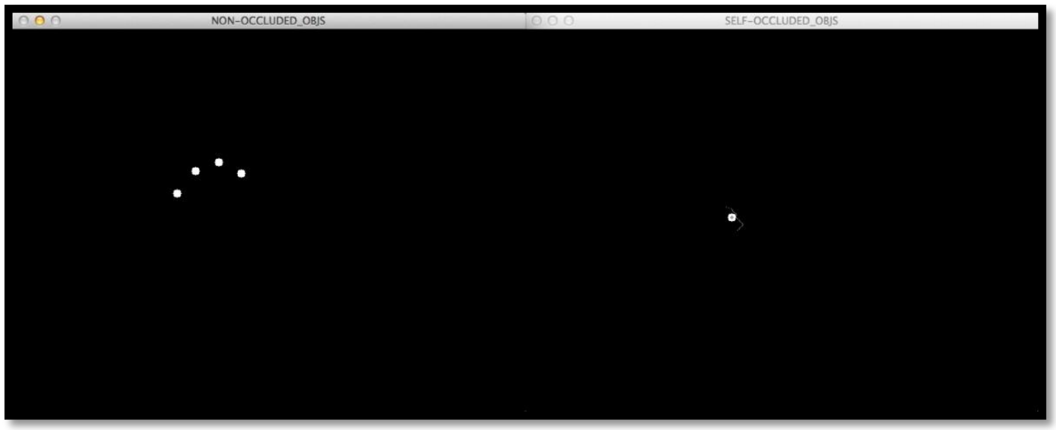


Figure 70: The thumb is lost in the original skeleton but detected in the depth.

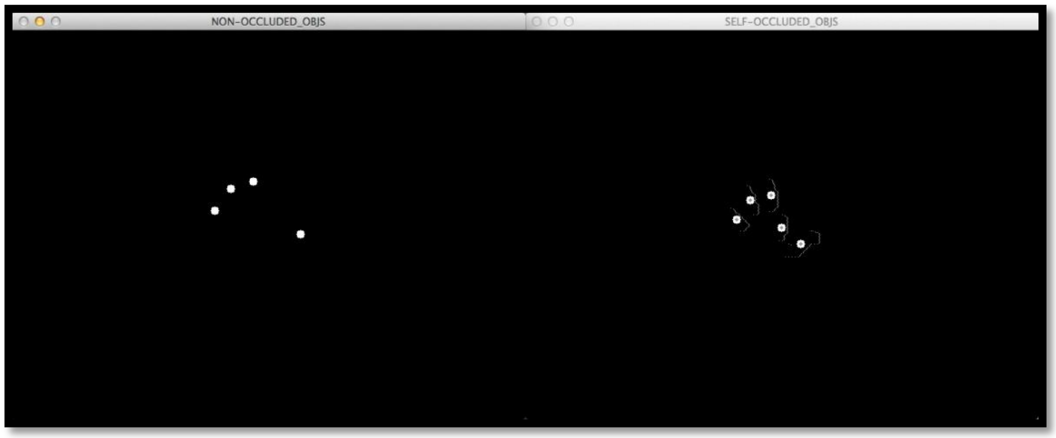


Figure 69: The Index is lost on the original skeleton but detected by the depth.

The next step is to merge the two methods forming one new skeleton that would detect all the fingers without them being self-occluded.

9.12 Merging the techniques

In mathematics to merge two matrices, a simple addition is performed:

$$C = A + B = \begin{bmatrix} A_{(0,0)} & \cdots & A_{(640,0)} \\ \vdots & \ddots & \vdots \\ A_{(0,480)} & \cdots & A_{(640,480)} \end{bmatrix} + \begin{bmatrix} B_{(0,0)} & \cdots & B_{(640,0)} \\ \vdots & \ddots & \vdots \\ B_{(0,480)} & \cdots & B_{(640,480)} \end{bmatrix}$$

Where C is the final gathering of methods, A is the nonOccludedObj and B is the selfOccludedObj. In this case C includes the detected points from both techniques, which suggests no loss of finger detection.

9.12.1 Filter Objects

This technique requires filtering since it was observed that when the palm was closed, one point was detected at its centre, which confused the rest of the skeleton. This detection is a result from the Euclidean distance transform used in the line skeletonization method. In order to prevent the program from recognizing two fingers whereas only one is used, the size of the hand's blob was taken into consideration, resulting into points being drawn only if the area of the object fits the size of a finger tip.

9.12.2 Identification and Sorting

In a similar procedure as was demonstrated previously, the identification and sorting of the 3D skeleton was done using the Point4f structure to store the points and the cosine formula to detect their angle.

$$\cos\theta_i = \frac{A \odot B}{|A| \times |B|}$$

Where θ is the angle, A is the $\overrightarrow{EP}$ vector and B is the new $\overrightarrow{FP_i}$ vector.

Additionally, each finger's length from the palm was found and assigned to the temporary finger variables.

9.12.3 Final finger assignment

At this point the 3D skeleton has been formed combining the two methods preventing from any data loss or miss-detections (Figure 71).

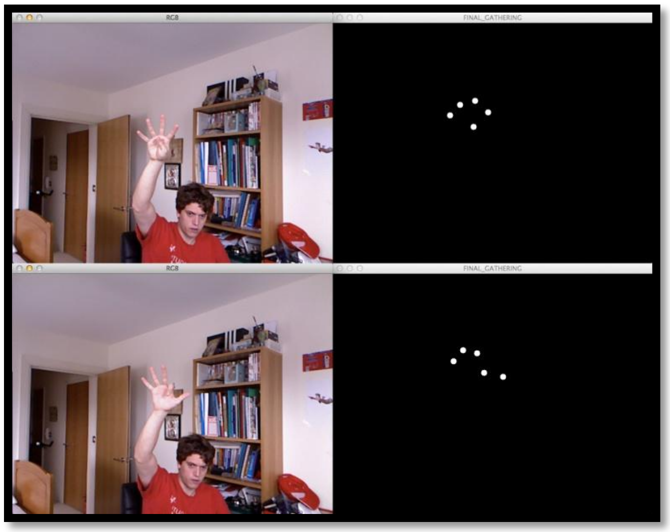


Figure 71: 3D skeleton merging the two methods.

The points detected are assigned to the temporary finger variables so that they can be used by the virtual locators of the skeleton.

9.14 Position Rules

Due to the z position of the palm, which was set to be 0.02 more than what it is in reality to dismiss the small muscles of the hand, as soon as the palm was closed, the fingers would be positioned at (0, 0).

The rules applied to the fingers only consider their length so that they would never exceed 1.5 meters or that when they move to (0, 0) they should be re-positioned at the centre of the palm. These constraints are very simple and only prevent the skeleton's finger from being miss-positioned when the palm is closed.

9.15 Visualization and Final Gathering of Joints

The 3D skeleton is complete, noise free, there are no miss-detections and the fingers are detected even though they are in front of the palm, suggesting that self-occlusion is solved. Yet there are a few limitations with this technique that will be discussed later in this thesis, such as sensitivity, blob accuracy, finger identification and line skeletonization constraints.

A simple visualization of lines and circles has been created to demonstrate the usage of the skeleton.

Figure 72 demonstrates the result of the currently working 3D skeleton extraction.

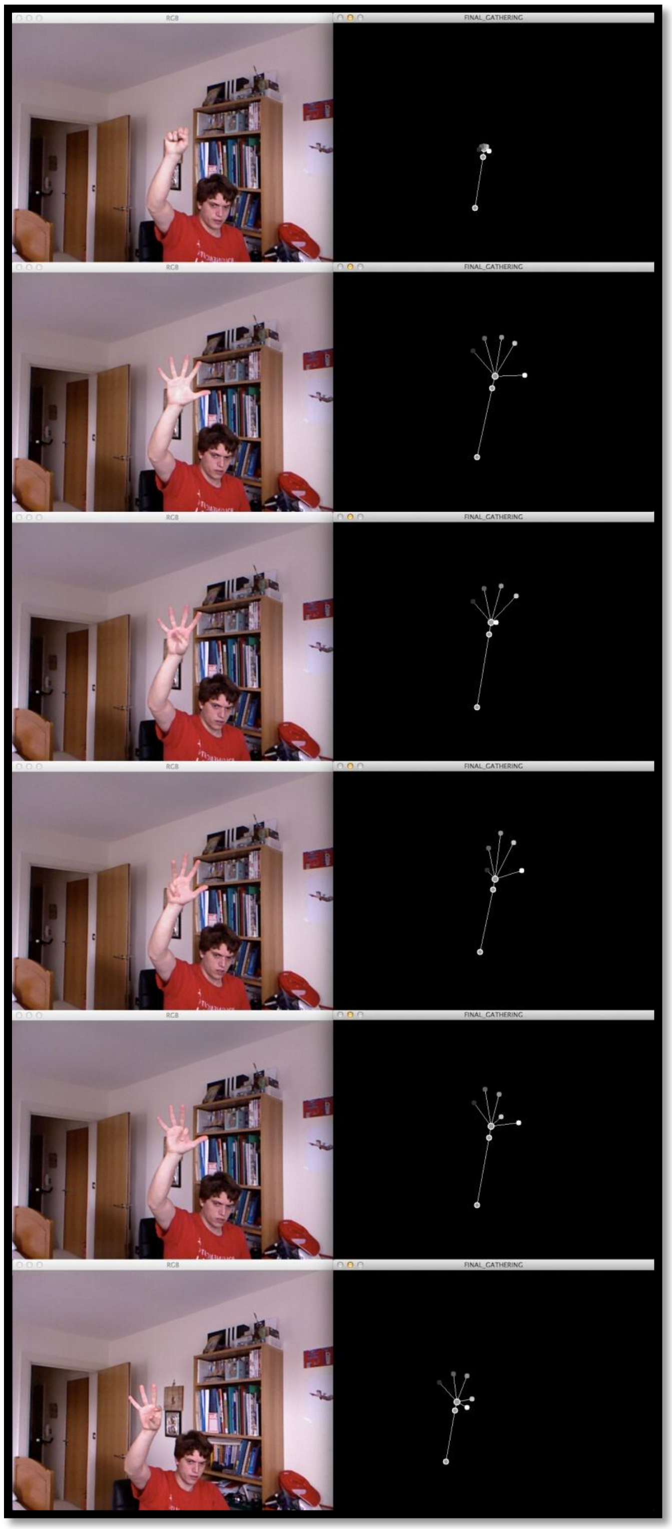


Figure 72: 3D skeleton extraction.

Chapter 10

FAILED METHODS

Although the method used for constructing the 3D skeleton works relatively well, it is worth mentioning the previous methods that did not succeed and the major techniques that caused issues.

10.1 Skeletonization Constraints

As soon as the palm closes, the Euclidean distance transform splits the skeleton in two branches, due to its area becoming big enough to consider each corner as a separate joint. This resulted to an extra finger every time one was raised or a finger being detected and formed when the palm was closed. This problem was sorted by the thinning technique performed right before the EDT and adding rules to prevent such operations, but is still present in some cases.

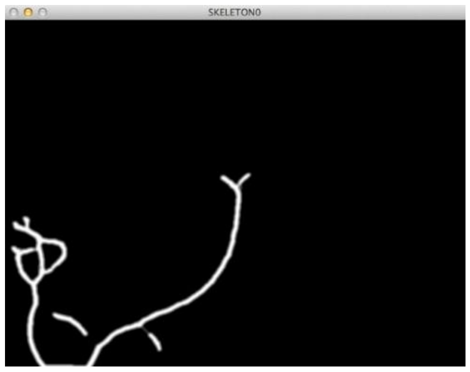


Figure 73: EDT on palm closed.

Additionally, even though the line skeletonization deals with the mesh as a single line, it is created in a 2D environment, thus exposed to self-occlusion threats. This was the case when trying to connect the self-occluded points to the fingers of the skeleton.

10.2 Drawing Limitations of 3D Points

Using the Kinect to detect the points in depth was the key element to the construction of the skeleton. One difficulty that was faced during its development was that even though the points were located in x, y, z co-ordinates, they had to be drawn using only x and y as computer vision deals only with 2D images. This became a problem when using blobs to track the fingers. When the contours of the blobs were formed, the program could not determine if one of the points is closer to the camera than the other, thus merging the point as it would do normally. This is why when the fingers get really close to each other they merge recognizing only one shape, also being one of the main reasons that the method used is very sensitive. This problem was partially solved because the number of grid points tracked, were accurately positioned in the centre of the blob. This issue is visible only when the fingers touch each other.

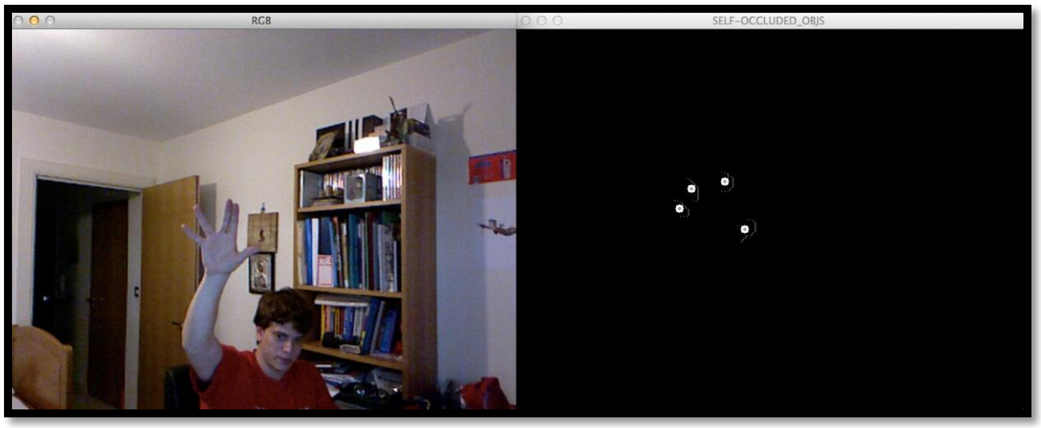


Figure 74: Contour merging.

10.3 Method 1: Tracking the grid points

The original method proposed to track the self-occluded fingers, used the 2D skeleton and converted its locators in 3D using the pixelValue. Furthermore it would track the points on the grid according to their angle, meaning that the hand would be divided into segments and the points would be assigned to the according segment. This was very unstable, inaccurate and the fingers were not being identified correctly.

Additionally the skeletonization method caused issues when the line got self-occluded and the locators where no longer accessible. The problem with this attempt was that there was no way to track the grid points using the skeleton extracted from the other methods. The best result coming from this method was one finger being tracked at a time. If many fingers were bending, the locators would get confused and lose track.

10.4 Method 2: 3D Skeletonization

As soon as the above problems were identified, the next attempt involved forming the line skeletonization in the 3D environment from scratch. It was considered that if the lines had x, y, z co-ordinates from initialization, only the lines in front of the palm would be drawn thus solving self-occlusion and also to discriminate whether a line was covering the other. This method was much more efficient than the previous one and the fingers were tracked with more accuracy.

Unfortunately, due to drawing constraints the lines had to be drawn in a 2D matrix and the Harris corner-detection assigned x and y co-ordinates to the points resulting to the locators being self-occluded. Similar dead-end as before but with more accuracy. Should it been possible to draw 3D points on a three dimensional image matrix, this method would have been very successful.

10.5 Method 3: 3D Blobs

The final attempt to track those grid points involved drawing them in a new image and converting them to blobs as has been examined previously. This is the current working technique used for this thesis. Although this works, it consists of a few limitations such as:

- Contours merging when colliding
- EDT splitting the joints
- Sensitive due to Kinect's frame rate on the depth
- Counter-clockwise finger identification

Most of the issues listed above were solved with thinning methods, and by adding rules/constraints to the skeleton so that it would not break. Yet once the fingers get really close to one another the mesh gets merged producing only one locator. Overall the grid points were accurately detected but the problems occurred when trying to track them with the other skeleton. The issue lies with the positioning of the locators. At this point, the blob locators have x, y, z co-ordinates but they are positioned on a 2D matrix cancelling out the depth which is given by the Kinect. This is why they merge when their distance becomes really small.

A possible solution to this problem would be to analyze the depth values and create a database which would allocate each depth point to the according position on the hand. That way, when the finger collide the depth values that would determine the collision can be cancelled out thus discriminating one finger from the other. This requires a lot of time and testing and it is going to be one of the future developments.

Chapter 11

THE MAIN APPLICATION

To demonstrate the 3D skeleton, eight locators were created in the main window linked to each of the joints co-ordinates. The virtual skeleton is composed of different coloured spheres and simple lines. Three cameras were set up, showing the hand from front, side and perspective view. There is no particular GUI involved as the application is only demonstrating the hand skeleton and nothing else. The shader used is a simple colour-phong shader just to represent the different locators.

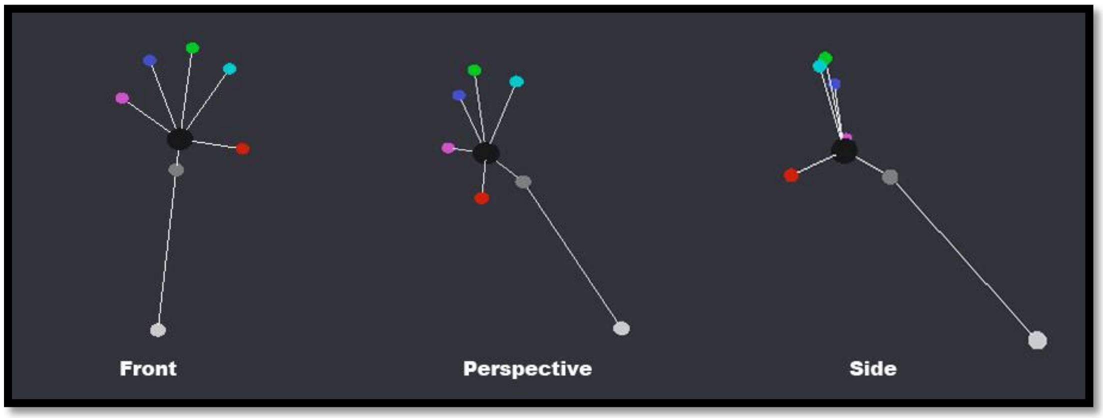
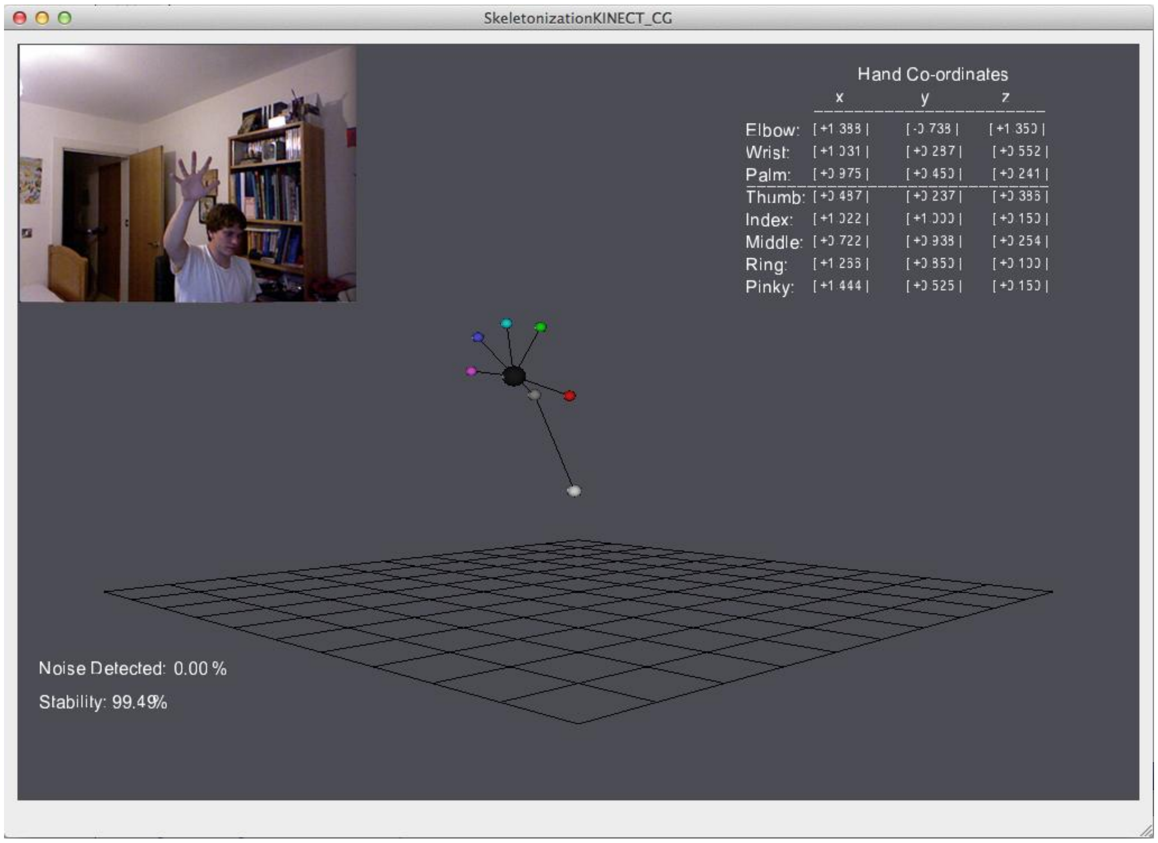


Figure 75: Main Window virtual skeleton.

It was observed that because the depth of the fingers was in centimeters and the difference in the values was very small, it was not possible to see much difference on the main window. The elbow, wrist and palm are very noticeable as the movement of the entire hand is very big compared to the fingers.

Additionally, to demonstrate the use of the technique a simple particle based cloth simulation was used from previous personal projects that allow the user to control and manipulate the objects. A grab gesture was included to “grab” the particles. Find more in the demonstration videos.



Hand Co-ordinates

	x	y	z
Elbow:	[+1.388|	[-0.738|	[+1.350|
Wrist:	[+1.031|	[+0.287|	[+0.552|
Palm:	[+0.975|	[+0.450|	[+0.241|
Thumb:	[+0.487|	[+0.237|	[+0.386|
Index:	[+1.022|	[+1.000|	[+0.150|
Middle:	[+0.722|	[+0.938|	[+0.254|
Ring:	[+1.266|	[+0.850|	[+0.100|
Pinky:	[+1.444|	[+0.525|	[+0.150|

Noise Detected: 0.00 %

Stability: 99.49%

3D SKELETON EXTRACTION APPLICATION

Furthermore, a few useful bits were added that measure the noise and misdetections of the skeleton.

Chapter 12

MORE APPLICATIONS

In general, a 3D skeleton with one camera has great prospects. It can be used in games to increase the playing experience and make the gamer feel more part of the game world, it can be used in training/educational simulations such as in medicine were surgeons can practice their techniques or pilots to learn the basics of how to fly a plane.

Most importantly it can be used in motion capture for movies to decrease the cost of equipment in use and the general limitations currently faced. Other applications of the skeleton would be for smart homes were it would check for the health of the person or for any intruders in the house.

This method could also be supplied with many additional systems such as facial recognition or speech recognition that could improve the final use of the skeleton.

Chapter 13

CONCLUSION

The method used to extract the 3D skeleton with one Kinect camera works and gives positive results, but it is not the most efficient way to tackle this research question.

The skeleton is sensitive due to Kinect's frame rate of the depth sensor as explained in chapter 3.3, thus losing track of the fingers when the hand is in motion.

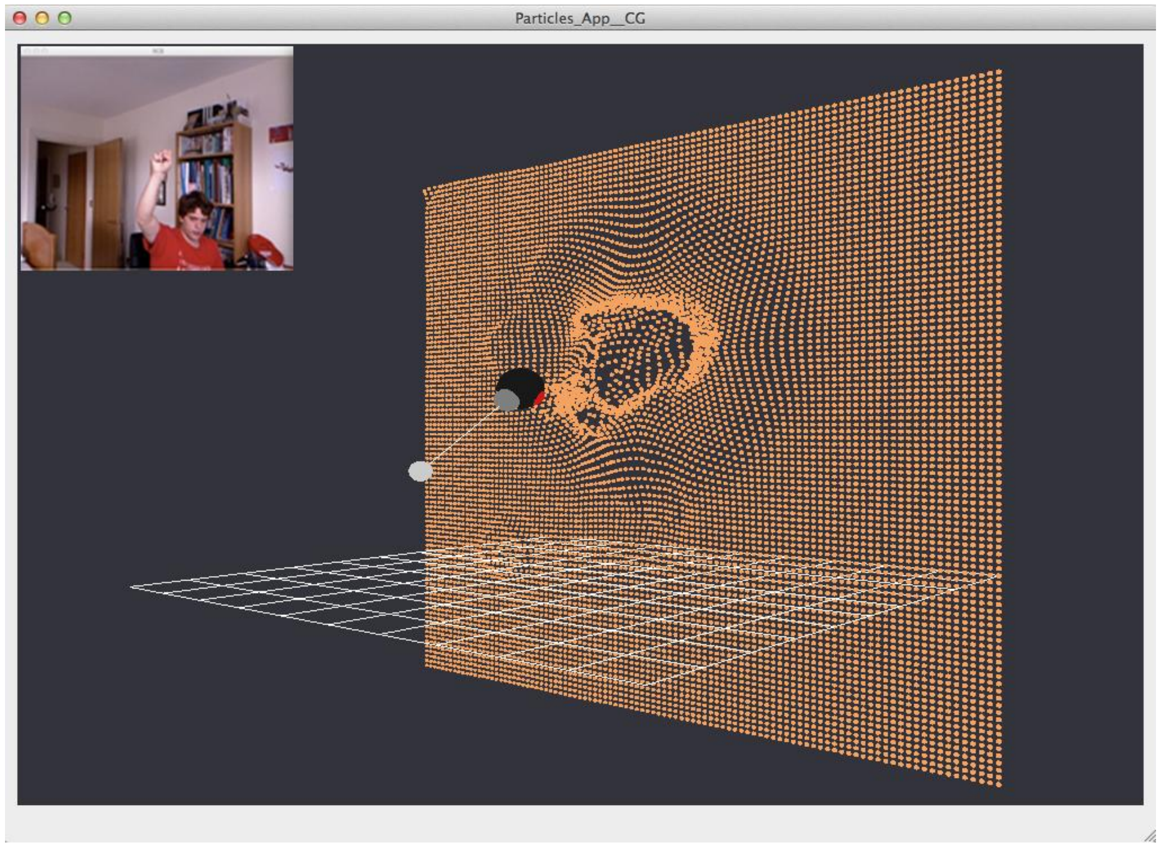
Also the use of blobs to track the fingers from 2D in 3D has many disadvantages such as merging meshes when getting really close, but it was the only method out of the three that worked.

The skeletonization technique resulted into many miss-detections that confused the final skeleton and although it works very well for the 2D, it caused issues to the 3D method such as self-occluded lines. The major problem with the skeletonization method is that it is formed in a two dimensional image whereas the Kinect can develop a three dimensional environment. So when a point was detected in 3D, it had to be drawn in a 2D matrix even though it contained the 3D information, therefore being exposed to merging with other points. It is suggested that should it was possible to draw points on a 3D environment the second method attempted would have worked very efficiently because only the points with the wanted depth value would have been identified.

Overall the 3D skeleton constructed in this application works as a proof of concept, suggesting that in the end it can be done, but it should be approached differently.

It suggests that with the current state of computer vision, line skeletonization should not be used to develop the three dimensional skeleton. Blob extraction method to track the fingers is also not a recommended solution neither is the use of angles as they are both very limited.

As a further development, this research question should be approached with a thorough study in computer vision analyzing all the possible routes and techniques before implementing in code. Detecting the joints and tracking the joints are two totally different areas in skeleton extraction and really hard to combine, but with the proper amount of tests and improvements it can be done.



PARTICLE APPLICATION (grab function)

BIBLIOGRAPHY

Bradski, G. and Kaehler A., 2008. Learning OpenCV: Computer Vision with the OpenCV Library, First edition, US: O'REILLY.

Safaee-Rad R. and Benhabib B. and Smith K.C. and Zhou Z., 1989. *Pre-Marking Methods For 3D Object Recognition*. In: IEEE International Conference on 14-17 Nov. 1989, 10.1109/ICSMC.1989.71366, Page(s): 592 - 595 vol.2, Cambridge, MA.

Giannitrapani R. and Vittorio M., 1989. *Three-dimentional skeleton extraction by point set contraction.* In: Image Processing, 1999. ICIP 99. Proceedings. 1999 International Conference on 24-28 Oct. 1999, Digital Object Identifier: 10.1109/ICSMC.1989.71366, Page(s): 565 - 569 vol.1, Kobe, JA.

Sato, M. and Bitter, I. and Bender, M.A. and Kaufman, A.E. and Nakajima, M., 2000. *TEASAR: tree-structure extraction algorithm for accurate and robust skeletons*. In: Computer Graphics and Applications, 2000. Proceedings. The Eighth Pacific Conference on 3-5 Oct. 2000, 10.1109/PCCGA.2000.883951, Page(s): 281 - 449, Hong Kong, HK.

Weik, S., 2000. *A passive full body scanner using shape from silhouettes*. In: Pattern Recognition, 2000. Proceedings. 15th International Conference on 3-7 Sept. 2000, 10.1109/ICPR.2000.905495 , Page(s): 750 - 753 vol.1, Barcelona, ES.

Jeong-Sun Park and Il-Seok Oh, 2002. *Shape decomposition and skeleton extraction of character patterns*. In: Pattern Recognition, 2002. Proceedings. 16th International Conference on 11-15 Aug. 2002, 10.1109/ICPR.2002.1047934 , Page(s): 411 - 414 vol.3.

Yi Sun and Mei-Hua Li and Jia-Sheng Hu and En-Liang Wang, 2002. *2D recovery of human posture*. In: Machine Learning and Cybernetics, 2002. Proceedings. 2002 International Conference on 4-5 Nov. 2002, 10.1109/ICMLC.2002.1167490 , Page(s): 1638 - 1640 vol.3, Beijing, CN.

Chaichana, T. and Sangworasil, M. and Pintavirooj, C. and Aootaphao, S., 2006. *Accelerate a Dlt Motion Capture System With Quad-Tree Searching Scheme*. In: Communications and Information Technologies, 2006. ISCIT '06. International Symposium on Oct. 18 2006-Sept. 20 2006, 10.1109/ISCIT.2006.339935 , Page(s): 1035 – 1038, Bangkok, TH.

Chih-Chang Yu and Jenq-Neng Hwang and Gang-Feng Ho and Chaur-Heh Hsieh, 2007. *Automatic Human Body Tracking and Modeling from Monocular Video Sequences*. In: Acoustics, Speech and Signal Processing, 2007. ICASSP 2007. IEEE International Conference on 15-20 April 2007, 10.1109/ICASSP.2007.366058, Page(s): I-917 - I-920, Honolulu, HI.

Yu-Shuen Wang and Tong-Yee Lee, 2008. *Curve-Skeleton Extraction Using Iterative Least Squares Optimization*. In: Visualization and Computer Graphics, IEEE Transactions on July-Aug. 2008, 10.1109/TVCG.2008.38, Page(s): 926 – 936 vol.14.

Takahashi, K. and Nagasawa, Y. and Hashimoto, M., 2007. *Remarks on 3D human body's feature extraction from voxel reconstruction of human body posture*. In: Robotics and Biomimetics, 2007. ROBIO 2007. IEEE International Conference on 15-18 Dec. 2007, 10.1109/ROBIO.2007.4522146 , Page(s): 121 - 126, Sanya, CN.

Faming Gong and Cui Kang, 2009. *3D Mesh Skeleton Extraction Based on Feature Points*. In: Computer Engineering and Technology, 2009. ICCET '09. International Conference on 22-24 Jan. 2009, 10.1109/ICCET.2009.71, Page(s): 326 - 329, Singapore, SG.

En Peng and Ling Li, 2008. *Acquiring human skeleton proportions from monocular images without posture estimation*. In: Control, Automation, Robotics and Vision, 2008. ICARCV 2008. 10th International Conference on 17-20 Dec. 2008, 10.1109/ICARCV.2008.4795882, Page(s): 2250 - 2255, Hanoi, VN.

Xin Yuan and Xubo Yang, 2009. *A Robust Human Action Recognition System Using Single Camera*. In: Computational Intelligence and Software Engineering, 2009. CiSE 2009. International Conference on 11-13 Dec. 2009, 10.1109/CISE.2009.5366107, Page(s): 1 - 4, Wuhan, CN.

Sang Min Yoon and Graf, H., 2009. *Automatic skeleton extraction and splitting of target objects*. In: Image Processing (ICIP), 2009 16th IEEE International Conference on 7-10 Nov. 2009, 10.1109/ICIP.2009.5414139, Page(s): 2421 - 2424, Cairo, EG.

Jianhao Ding and Yigang Wang and Lingyun Yu, 2010. *Extraction of Human Body Skeleton Based on Silhouette Images*. In: Education Technology and Computer Science (ETCS), 2010 Second International Workshop on 6-7 March 2010, 10.1109/ETCS.2010.241, Page(s): 71 - 74, Wuhan, CN.

Le Zhang and Qing He and Ito, S.-I. and Kita, K., 2010. *Euclidean distance-ordered thinning for skeleton extraction*. In: Education Technology and Computer (ICETC), 2010 2nd International Conference on 22-24 June 2010, 10.1109/ICETC.2010.5529241, Page(s): V1-311 - V1-315, Shanghai, CN.

Xujia Qin and Xiansheng Sang and Sida Zhu and Shiwei Cheng, 2010. *Line-Skeleton Extraction of 3D Meshes Based on Geometry Segmentation*. In: Cryptography and Network Security, Data Mining and Knowledge Discovery, E-Commerce & Its Applications and Embedded Systems (CDEE), 2010 First ACIS International Symposium on 23-24 Oct. 2010, 10.1109/CDEE.2010.73, Page(s): 354 - 357, Qinhuangdao, CN.

Jing Tong and Jin Zhou and Ligang Liu and Zhigeng Pan and Hao Yan, 2012. *Scanning 3D Full Human Bodies Using Kinects*. In: Visualization and Computer Graphics, IEEE Transactions on April 2012, 10.1109/TVCG.2012.56, Page(s): 643 – 650, vol.18.

Wright M. and Cipolla R. and Giblin P., 1994. *Skeletonisation using an extended Euclidean distance transform.* In: Proceeding BMVC 94 Proceedings of the conference on British machine vision (vol. 2), ISBN:952-1898-1-X, Page(s) 559 - 568, Surrey, UK.

Cao J. and Tagliasacchi A. and Olson M. and Hao Zhang Zhinxun Su, 2010. *Point Cloud Skeletons via Laplacian Based Contraction.* In: Proceeding SMI '10 Proceedings of the 2010 Shape Modeling International Conference IEEE Computer Society Washington, DC, USA ©2010, 10.1109/SMI.2010.25, Page(s) 187-197, Washington, US.

Oscar Kin-Chung Au and Chiew-Lan Tai and Hung-Kuo Chu and Daniel Cohen-Or and Tong-Yee Lee, 2008. *Skeleton extraction by mesh contraction*. In: ACM Transactions on Graphics (TOG), Volume 27 Issue 3, August 2008, 10.1145/1360612.1360643, Page(s): 1 – 10, New York, US.

Panagiotakis C., and Tziritas G., 2004. *Recognition and Tracking of the Members of a Moving Human Body* ⋆. In: F.J. Perales and B.A. Draper (Eds.): AMDO 2004, LNCS 3179, Page(s): 86–98, Berlin, DE.

Aitpayev K., and Gaber J., 2010. *Creation of 3D Human Avatar using Kinect*. In: Asian Transactions on Fundamentals of Electronics, Communication & Multimedia (ATFECM) (ATFECM ISSN: 2221-4305) Volume 01 Issue 05, Jan 2012, Page(s): 1 - 2, Beijing, CN.

Li Ming and Wang Jun and Zhu Meiqiang, 2010. *On skeleton extraction algorithm for path planning of mobile robots in complex planar maps*. In: Control Conference (CCC), 2010 29th Chinese 29-31 July 2010, 11611752, Page(s): 3704 - 3708, Beijing, CN.

Aipeng Qi and Jing Xu, 2010. *Skeleton extraction of cerebral vascular image based on topology node*. In: Biomedical Engineering and Informatics (BMEI), 2010 3rd International Conference on 16-18 Oct. 2010, 10.1109/BMEI.2010.5640001, Page(s): 569 - 573, Yantai, CN.

Chayanurak, R. and Cooharojananone, N. and Satoh, S. and Lipikorn, R., 2010. *Carried object detection using star skeleton with adaptive centroid and time series graph*. In: Signal Processing (ICSP), 2010 IEEE 10th International Conference on 24-28 Oct. 2010, 10.1109/ICOSP.2010.5655765, Page(s): 736 - 739, Beijing, CN.

She, F.H. and Chen, R.H. and Gao, W.M. and Hodgson, P.H. and Kong, L.X. and Hong, H.Y., 2009. *Improved 3D Thinning Algorithms for Skeleton Extraction*. In: Digital Image Computing: Techniques and Applications, 2009. DICTA '09. 1-3 Dec. 2009, 10.1109/DICTA.2009.13, Page(s): 14 - 18, Melbourne, VIC.

Hongbo Jiang and Wenping Liu and Dan Wang and Chen Tian and Xiang Bai and Xue Liu and Ying Wu and Wenyu Liu, 2010. *Connectivity-Based Skeleton Extraction in Wireless Sensor Networks*. In: Parallel and Distributed Systems, IEEE Transactions on May 2010, 10.1109/TPDS.2009.109, Page(s): 710 - 721.

Tierny, J. and Vandeborre, J.-P. and Daoudi, M., 2008. *Fast and precise kinematic skeleton extraction of 3D dynamic meshes*. In: Pattern Recognition, 2008. ICPR 2008. 19th International Conference on 8-11 Dec. 2008, 10.1109/ICPR.2008.4761011, Page(s): 1 - 4, Tampa, FL.