Abstract

This project has two main components, the first one is to create a 3D head pose estimation software. This software is based on random regression forest, which was proposed in the research of Gabriele Fanelli [2,5] et al.. The key points of this program are: i) 3D data, the program must be based on range images. ii) real-time, this program will detect the people with a 3D camera, Kinect, and estimate the orientation of people's head in real-time. And iii) good accuracy of estimation, find good method to increase the accuracy as much as possible.

The second component is to expand the algorithm in the research of Gabriele Fanelli et al. to estimate the orientation of moving people's head. In this part, I have made improvements in 3 aspects: i) head tracking, I added Skeleton Tracking technic into this program, which can track at most two people's heads and return the position as a point. ii) determine the head zone, find a method to create a rectangle area to frame the head and it will change its size along with the head. iii) use the positional relationship between head and shoulder centre to increase accuracy.

- I created a piece of software to estimate head pose (see Section 3 and 5) adapted from an algorithm proposed by Gabriele Fanelli et al. in [2,5], see Section 2.1.3.

- I created a method to track people's head based on Skeleton Tracking, which is provided by Kinect for Windows SDK, see Section 5.1.3.

- I created a method to help estimating the head pose by comparing the positional relationship between head and shoulder centre, see page 46-47.

Contents

1. Introduction

1.1. Aims and Objectives

The objective of this project is to use fast and accurate estimators that are able to do a regression on the 3D data, to estimate the orientation in space of people's head. Estimating head pose with Random Regression Forests is a robust, accurate and real-time way. My work is based on the theory of Gabriele Fanelli's (will be described in Section 2), while some changes have been made to fit simple and crude working conditions (hard to capture data, low quality of training images, etc.), which I will describe in detail in the *Implementation* part. There are still some limitations in Gabriele Fanelli's work, with the most obvious one being that the person who is in testing must be sitting in a pre- selected position in front of the camera. Thus, I also tried to improve this method to fit the situation of moving people and more than one person. This will be shown in the following sections.

1.1.1. Implementation Objectives

i) Implementation of Random Regression Forests system to estimate head orientations.

ii) Find fast and accurate methods to improve the estimation system.

iii) Test the correctness of the system.

1.1.2. Research Objectives

i) Expand the algorithm to fit the situation of moving people.

1.2. Structure

This report consists of 5 main sections. Firstly, time is spent explaining the background of human body orientation detection, as the most important part of which, head pose estimation will be introduced. The research of Gabriele et al., Random Regression Forest, will also be explained in this section as a typical example of 3D image-based head pose estimation algorithm. The next section will describe the technic basis of this project, as well as several alternative methods, and how they are chosen or abandoned during implementation. In Section 4, I will introduce the 3rd party hardware and software that are used in this project, including their technical basis and function. How this project is realized, as well as the result of it, will be discussed in Section 5. The testing results of the alternative methods that I mentioned in Section 3, and the comparison between each method will also be listed and discussed in Section 5. Finally, the result of my project and future work will be fully discussed in Section 6.

2. Background

Human behaviour analysis techniques, which is considered to be the most popular issue in the field of pattern recognition and computer vision research, had kept offering new services and improving human life in the last decades. Estimating the head pose has become one of the fastest-growing techniques, which played an important role in human body estimation and Human-Computer interaction. In recent years, accurate face detection algorithm-combined face orientation detection technology with high reliability, high robustness and low complexity has been wildly used in Human-computer interaction, intelligent security, content-based video indexing, traffic monitoring and many other fields.

Human body orientation detection methods mainly function through detecting the angles in three directions in the space where the human head is in, while posture and gaze direction are also important information. Figure 2.1 shows the analysis process.

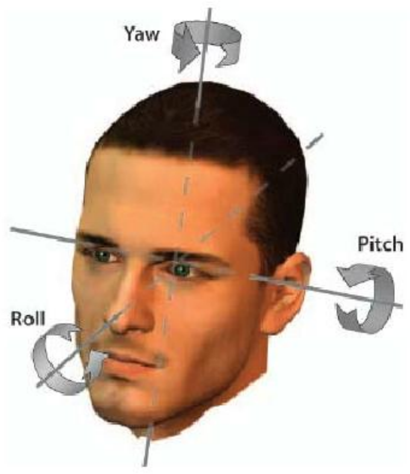


Figure 2.1. Three degrees of head freedom: pitch, roll and yaw. Image from [1].

Human head orientation estimation is an algorithm which uses face detection methods to get face images and then detect the angles in space through template matching, machine learning, morphological characterising and predicted tracking algorithm, etc. Sketchy face orientation estimation algorithms are commonly used to classify head posture, such as looking towards left, right and front. However, some algorithms relying on 3D modelling, target tracking and priori knowledge of angle can detect angles from three directions in the space accurately [1].

A common algorithm uses the Gaussian filter template to process images to get the gray value of every pixel, and then pupils and nose can be localized using Hough Transform [4]. After getting the coordinates of eyes and nose, we can calculate the middle point of two eyes and connect this point to nose tip (Figure 2.2) to build different models for this issue. The simplest one is to use the angle between target vector and eyes and nose. Certainly, some other methods can be used here too. No matter which method would be used, the data that after preprocess can be used as training set.

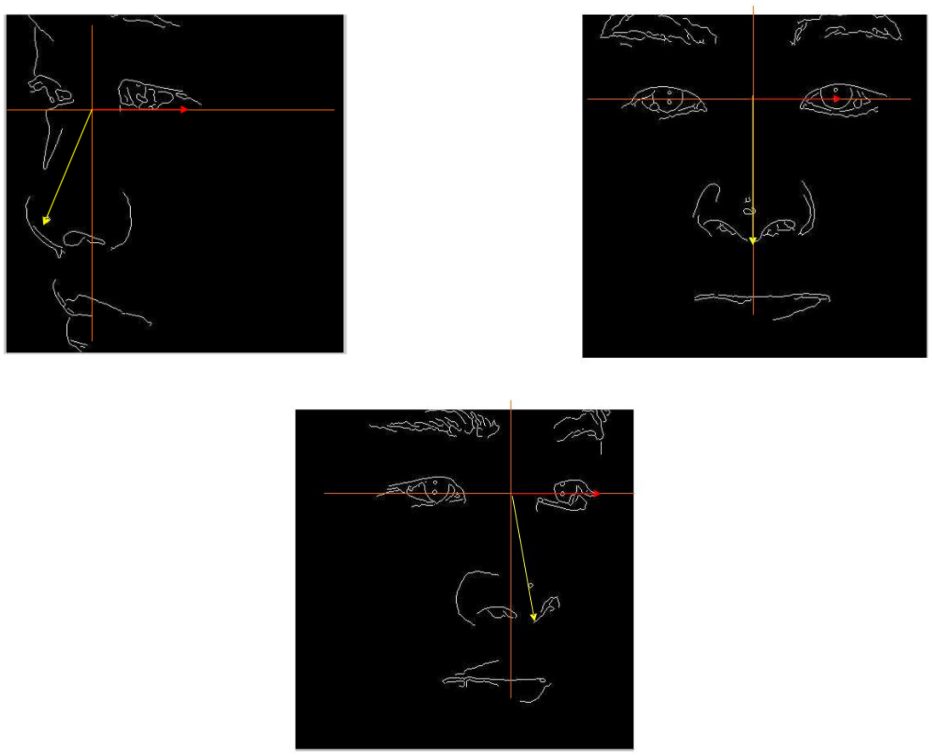


Figure 2.2. Faces with different orientations. Image from [1].
Top left is a leftward face, top right is a front face and bottom is a rightward face.

2.1.1. The problem with 2D images

The method discussed above is widely used in many cases. However, we can find that color and some other factors could be the destabilizing factors of this method.

Methods relying solely on standard 2D images face serious problems, especially when illumination keeps changing and face regions being textureless [2]. A 2D face image is converted from the environmental lights that reflected by face surface, via the photoelectric conversion and quantification. Therefore, such an image contains not only essential factors of the face, but also external factors (i.e. illumination). Since most of the 2D image-processing methods are based upon color, lights from different directions and lights in different intensities might cause the changing of contrast of the image, leading to the difference of color extraction.

In addition, it will be a devastating problem if the head is covered by another object in practical situations. For instance, motion-sensing games often have multiple players simultaneously. Limited space and large movements make it inevitable that some body parts of one player will be covered by another.

2.1.2. Work in 3D images

Due to the relevance of head pose estimation and to the challenges posed by the problems described above, considerable effort has been made in the computer vision community to develop fast and reliable algorithms to solve them. With the availability of 3D cameras and structured light sensors such as the MS Kinect, it is possible to do many interesting things more easily when compared with only looking at 2D images. In recent years, 3D sensing technologies are becoming ever more affordable and reliable. The additional depth information finally allows us to overcome some of the problems inherent of methods based on 2D data. However, existing depth-based methods still need to be improved in handling large pose variation, being real time or being real time but over-reliance on advanced hardware devices, etc. [2, 3].

For instance, Breitenstein, et al. has achieved real-time monitoring using a geometric descriptor that will provide nose location hypotheses, and then compare them to a mass of renderings of a generic face template, both of which are done by a GPU, based on its power of massive processing [2, 3].

Hardware-dependency limits such method's promotion for certain kind of application, since the GPUs will bring both high power consumption and high price. Therefore, using a method to estimate the orientation in space of people's head rapidly, accurately and in real time could be a challenge.

Gabriele Fanelli et al. described their work in their CVPR paper (i.e. [2, 5]), which uses the Random Regression Forests to spilt the big problem into small ones. Their work solves this problem well in real-time and accuracy. In addition, their method does not require advanced hardware devices [5].

2.1.3. Head Pose Estimation with Random Regression Forests

Random forests [6] have become a popular method in computer vision, given their capability to handle large training datasets, high generalization power, fast computation, and ease of implementation. Recent works showed the power of random forests in mapping image features to votes in a generalized Hough space [8] or to real-valued functions [7, 9]. Recently, multiclass random forests have been proposed in [9] for real-time head pose recognition from 2D video data. The approach that uses random regression forests for the task of head pose estimation from range data has also been presented lately.

Gabriele et al. [2] used random regression forest to compute head pose by jointly estimating the 3D coordinates of the nose tip and the angles of head rotation in a range image. This could be achieved due to the capability of the classification and regression trees to map complex input spaces into discrete or continuous output spaces, by splitting the original problems into smaller ones that could be solved with simple predictions [2].

Random Regression Forest

A regression tree is consisted of a series of nodes, which directs the training data into left or right child according to the results generated by the test conferred to the node. Thus, diverse training data can be grouped into clusters along with the selecting procedure, which finally reach and be stored into the leaves that can achieve good prediction.

Breiman [6] claims that Random forests surpass standard decision trees in high generalization power, which enables the introduction of randomness either or both in the set of examples provided to each tree and the tests available for optimization at each node.

In their work, a patch-pool is established by many patches which are selected randomly, and a forest is consist of many binary trees. A patch is sent down to two trees and ending up in a leaf node. Leaf nodes are divided into two different classes, one is head leaf which contains the maximum-depth leaf nodes and such nodes are only available for the patches which are in the head zone (showed in Figure 2.3, blue rectangle), another one is non-head leaf, such leaf nodes are aimed to accept the patches which are out of head zone (red one). Figure 2.3 shows a training image of their work and Figure 2.4 shows an Example of regression of forest.

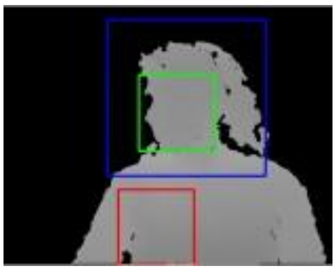


Figure 2.3. A training image [5].
The blue bounding box enclosing the head species
where to sample positive (green - inside) and
negative patches (red - outside) [5].

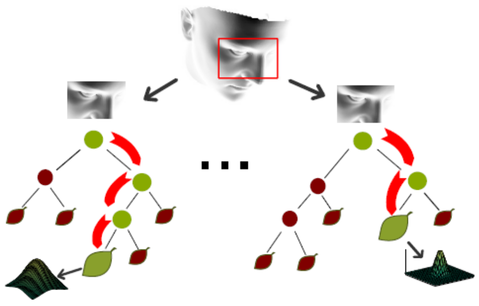


Figure 2.4. Example of regression forest. Image from [2].

For each tree, the tests at the non-leaf nodes direct an input sample towards a leaf, where a real-valued, multivariate distribution of the output parameters is stored. The forest combines the results of all leaves to produce a probabilistic prediction in the real-valued output space [2].

Training

In the research of G. Fanelli et al., they focused on head pose estimation by assuming the head has been detected in the image and setting up the training examples with range images of faces annotated with 3D nose location and head rotation angles. Also, they set up one to four feature channels, i.e., depth values and the X, Y and Z values of the geometric normals computed over neighboring, non-border pixels [2].

Parameter Patches $\{P_i = (I^f_i, \theta_i)\}$ was introduced by random sampling from the training examples that acts as a element to construct trees (Ƭ) in the forest [2]. I^f_i are the extracted visual features for a fix-sized patch, and the real-valued vector θ_i contains the pose parameters associated to each patch, i.e., the θ_x, θ_y, θ_z represent an offset vector from the center point of the training patch in the range image, to the nose position in 3D, while the θ_{yaw}, θ_{patch}, and θ_{roll} represent the head rotation angles denoting the head orientation [2].

A Binary test will be randomly generated from a set of possible tests at each non-leaf node, which is defined as $t_{f,\,F1,\,F2,\,\tau}(I)$:

$$|F_1|^{-1}\sum_{q\in F_1} I^f(q) - |F_2|^{-1}\sum_{q\in F_2} I^f(q) > \tau,$$

where I^f indicates the feature channel, F_1 and F_2 are two rectangles within the patch boundaries, and τ is a threshold [2].

Thus, when a patch satisfies the test, it will be passed to the right child while it can be directed to the left child if not. The difference between the two average values of rectangular spaces is chosen to estimate the head pose in order to reduce the sensitivity to the nose tip [2]. This is shown in Figure 2.5.

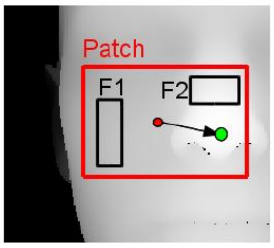


Figure 2.5. Example of a training patch. Image from [2].

A Training patch (larger, red rectangle) with its associated offset vector (arrow) between the 3D point falling at the patch's centre (red dot) and the ground truth location of the nose (marked in green). The rectangles F1 and F2 represent a possible choice for the regions over which to compute a binary test [2].

The values of f, F_1, F_2 and τ are randomly generated from a pool of binary tests $\{t^k\}$ at each non-leaf node, thus when the patches arrive at the node, they can be evaluated by all binary tests in the pool and the test maximizing a predefined measure will be assigned to the node. Furthermore, they optimize the trees by maximizing the information gain, which is defined as the difference between the differential entropy of the set of patches at the parent node P and the weighted sum of differential entropies at the left and right node P_L and P_R [2].

$$IG = H(p) - (w_L H(p_L) + w_R H(p_R)),$$

Where $P_{i\in\{L,R\}}$ is the set of patches reaching node I and WI is the ratio between the number of patches in i and in its parent node, i.e., $w_i = |P_i| / |P|$.

The vectors θ at each node are modelled as realizations of a random variable with a multivariate Gaussian distribution, which lead to the equation [2]:

$$IG = \log\left|\sum(p)\right| - \sum_{i\in\{L,R\}} w_i \log\left|\sum{}_i(p_i)\right|,$$

This decreases the uncertainty of the output parameters cast by each set of patches due to the minimizing the determinant of the covariance matrix Σ. Then, the authors assume the matrix is block-diagonal $\sum = \begin{pmatrix} \sum^{v} & 0 \\ 0 & \sum^{a} \end{pmatrix}$, which allows covariance among offset vectors (Σ^{v}) and among head rotation angles (Σ^{a}), rather than between them, and the equation above can be written as:

$$IG = \log(\left|\sum^{v}\right| + \left|\sum^{a}\right|) - \sum_{i \in \{L,R\}} w_i \log(\left|\sum_i^{v}\right| + \left|\sum_i^{a}\right|),$$

The leaf node is introduced to store the mean value of all angles and offset vectors reached it, together with their covariance to create a multivariate Gaussian distribution [2].

Testing

As described above, the test could be carried out given a new, unseen range image of a head. Patches with the same size as the training are sampled densely and passed through all the trees in the forest from the root to each node, where they can be evaluated by the stored binary test and sent either to the right or left child, until they reach the leaf *l* while giving estimates for the pose parameters [2].

In order to reduce the noise of the estimation generated by less informative leaves with high variance, the researchers discard all Gaussians whose total variance are greater than an empiric threshold max_v. Also, they locate the nose position before computing other parameters to remove outliers. A 10 mean-shift [10] iterations is carried out using a spherical kernel and only random variables with means falling within the mean shift kernel are kept. The radius of the kernel is defined as a fraction of the smallest sphere size capability to enclose the average human face, which is regarded as the mean of the PCA model constructed from range scans of many different people [11]. The initialization for the mean-shift is defined as the mean of all votes returned by the forest, where the votes cluster is considered to be close to the true location of nose. It is shown in their experiments that removing outliers is essential in the test images when head undergoes large rotations, and it might be partially occluded by glasses or hair [2].

Finally, the researchers sum all the remaining random variables θ, producing a Gaussian whose mean represents the output parameters and whose covariance reflects the measure of uncertainty extent for the estimate. This can be simply explained in Figure 2.6, where the blue spheres are all votes cast by the forest, while the green ones are all votes selected after mean-shift. The green cylinder represents the final estimate of the nose position and head orientation.

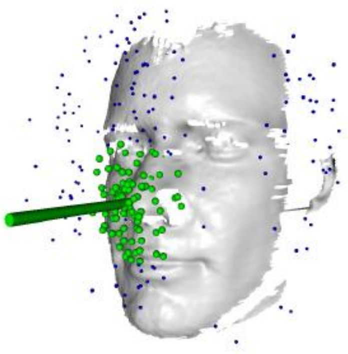


Figure 2.6. One example of testing image. Image from [2]. The green spheres are the ones selected after the filtering of the outliers (blue spheres) by mean shift. The large green cylinder stretches from the final estimate of the nose center in the estimated face direction.

3. Technical Basis

The primary focus of this project is the head pose estimation algorithm, while additional vital components of the proposed program will be considered too. The program process can be divided into five clearly defined stages，which are executed in the following order:

1. Data capture:
Collect images of people with their heads in different orientations and preprocess the data.

2. Calculate the test thresholds:
Calculate the value of τ for every non-leaf node.

3. Training:
Build up distribution for every leaf node.

4. Testing:
Estimate the orientation for new, unseen images.

5. Expanding:
Find out methods to expand the program.

Below I will describe the technical basis of each listed stage, detailing how the stage works and how it is implemented in the final prototype, as well as alternative methods considered where appropriate.

3.1. Data Capture

The key point of building up an accurate and complete model is the training data, which must be ensured that the training images are accurate in angles and sufficient in quantity. The implementation of capturing head images with the same orientation will be explained in Section 5.1.

3.1.1. Range Images from Kinect

The project was implemented with the 3D camera Kinect, which is a programmable 3D camera supplied by Microsoft. The Kinect sensor features an "RGB camera, depth sensor and multi-array microphone running proprietary software", which renders Kinect with the capabilities of full-body 3D motion capture, facial recognition and voice recognition [10]. The protagonist of this project is the depth sensor. It utilizes the Light Coding technology to encode the testing space with continuous light that could be read by the sensor and calculated, decoded by a chip sequentially, and finally produces a depth image. Figure 3.1 (a) shows the Kinect sensor, 3.1 (b) shows Light Coding technology.

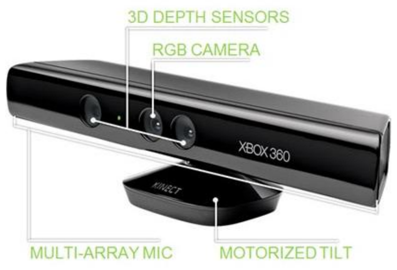


Figure 3.1 (a). The structure of the Kinect.

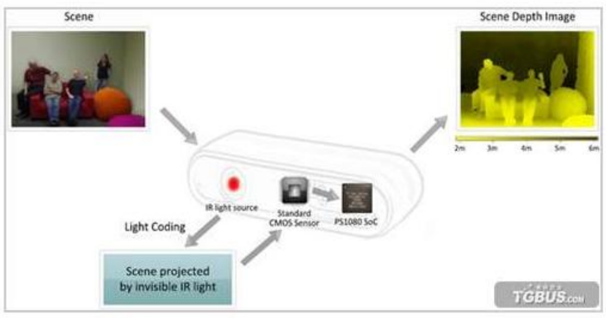


Figure 3.1 (b). How does the depth sensor work.
Figure 3.1 shows the hardware of a Kinect device
and illustrates the Light Coding technology.

Kinect is a programmable device. Microsoft has launched Kinect for Windows SDK so that the Kinect software can be developed by C# and .NET Framework 4.0, by which we can get the depth value of every pixel in the program. A pixel of the Kinect depth data frame is 16 bits in size (i.e. each pixel occupies 2 bytes). However, not all of the 16 bits records depth information, only the first 13 bits are Depth Bits and the last 3 are Player Index (shown in Figure 3.2). We can calculate the depth value, which is the distance between the depth sensor and the testing object, by using bit computing (shift the data to the right by three).

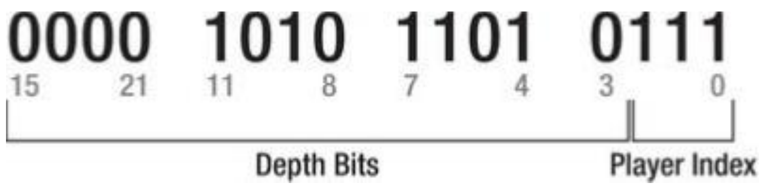


Figure 3.2. The structure of depth data.
It shows the structure of depth data, only the first 13 bits are Depth Bits and the last 3 are Player Index.

3.1.2. Data preprocessing

Collecting the depth information of a specified pixel is one of the prerequisites of data preprocessing, and the other one is creating the depth image frames. RGB images were used to create the depth image frames instead of gray ones (see Section 5.1), since they will guarantee the good visual effects on the depth frame, as well as simplify the process of extracting color information from pixels. However, the drawback of this design arose at the same time. When the depth frame stream began, the frame would be updated every 1/30 second, making that all the options I wanted to do with the depth frame must be done in this short period of time. Also, conversing the 16-bit gray level images into 32-bit ones occupied time, which obviously decreased time for computing the depth value of each frame and deciding which tree to go through. These problems were solved by adjusting the number and the depth of the trees.

Data storage comes after the image creation. How to and in what form to save the training data are two aspects that worth for consideration because the construction of the forest is based on them.

Gabriele Fanelli [2,5] proposed a method in their research [5], they set a database containing 20 person, 14 of which are males, 6 are females and 4 are wearing glasses, sitting in front of the camera and were asked to rotate their heads to all possible ranges of angles. They first built a template of human head using the technology provided by faceshift [12], and then used a 3D morphable model [13] with a subsequent graph-based non-rigid ICP [14], in order to adapt a generic face template to the point cloud [15]. This process resulted in a template representing the shape of the head, which is called the personalized templates, and thanks to such ICP-based head tracking technology, which made it possible to estimate head pose for each frame. Figure 3.3 shows the face tracking processes.

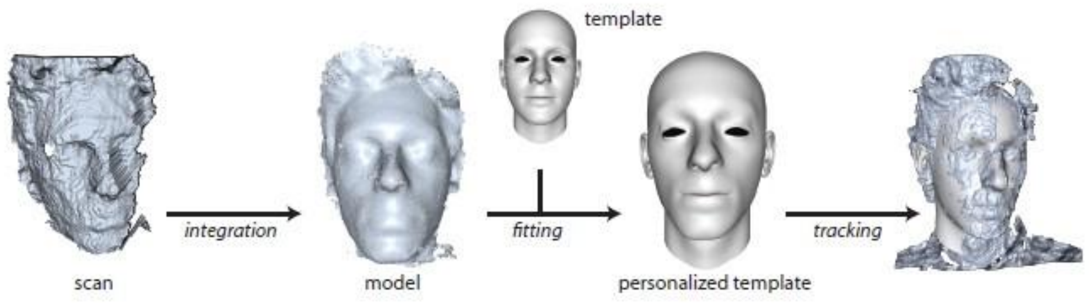


Figure 3.3. Training data based on template. Image from [5].

Automatic pose labelling: A user turns the head in front of the depth sensor, the scans are integrated into a point cloud model and a generic template is fit to it. The personalized template is used for accurate rigid tracking [5].

Technical support and teamwork facilitates their research by offering a huge amount of data with small deviation (the mean translation and rotation errors were around 1 mm and 1 degree respectively). In this project, the lack of human resource and technical support limits the work in several aspects. For instance, implementing the head pose estimation needs to be improved continually, so that it is possibly that I need to collect the training data and change the format of the training data for many times, this makes it hard to find many people to set up a huge and sundry training set. In addition, human face templates are not available so that it is difficult to detect the rotation angle for each frame. However, data that in some special orientations and containing few people can test the accuracy and correctness of the algorithm, despite its limited capability. Considering the factors above, there are two feasible ways for data storage: i) save the training data as depth images, and ii) save the training data as depth values in a file. The method that I chose in this project is the latter one, which is obviously advantageous when compared to the other one (corresponding distance value to color, it will be described in detail below), since it saves the depth value directly, ensures data accuracy, and is easy to handle. However, this method also contains some drawbacks manifesting in two aspects. First, once the program runs to the stages of calculating threshold and training, reading the distance of every pixel to an array and searching the array is a time-consuming task. Second, the data collected in this way will take more storage space. As for method i), it associates distance value with color, and then save the images (an image saved in this way is shown in Figure 3.4) as the training data. When the depth values are needed to calculate the thresholds, the getPixel() method can be used to get the color of the specified pixels and this color value represents the depth attribute. Although this method saves time in the follow-up work and reduces the size of the database, I found it could not be adopted in this project after several tests. The reason and the experiment detail will be described in Section 5.1.

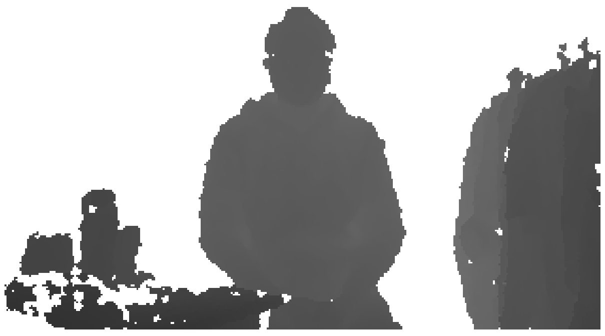


Figure 3.4. A training image.
One training data uses method i). The too far away distance and the too close distance are filtered out.

3.2. Calculate the Test Thresholds

The main purpose of this part is to assign one threshold to each non-leaf node. This part will use all images in the training data and the result will provide support for the following parts. Also, this part is the most time-consuming in the project. The final method was decided after considering various factors, but it still took about 22 hours when the training data contained 15 orientations and 300 images for each orientation, or about 15 hours when the training data contained 6 orientations and 500 images for each. The technical basis of this part will be described in detail below.

In Gabriele Fanelli's research, the first step in their program is to build up the forest. In their work, the authors gave a patch to each tree and randomly selected two rectangular areas (F_1 and F_2, named feature zones below), and as they described, "at each non-leaf node, a pool of binary tests $\{t^k\}$ is generated with random values for *f*, F1, F2, and τ"[2]. Thus, each patch would be evaluated by all binary tests in this pool when it arrived at the non-leaf node, and then, information attached to the node would be selected by the authors, according to how they wanted to build the non-leaf nodes. The huge amounts of training data and patches and tests pools made their method accurate and complete. However, building the forest in this way takes huge mass of time, which I could not afford in my project. The alternative method implemented in this project let all of the training data go into a node and subsequently calculate a value by the left side of the inequality listed below, which is mentioned in Section 1.1.3, and will also be used in the following steps.

$$|F_1|^{-1} \sum_{q \in F_1} I^f(q) - |F_2|^{-1} \sum_{q \in F_2} I^f(q) > \tau,$$

The threshold of this node would be set as the mean value of all τs. Same process would be applied to all non-leaf nodes. There are two reasons why I renounced Gabriele Fanelli's method. Firstly, the method I used in this project allowed each non-leaf node to be calculated by the equation listed above for only n+1 times (n=number of training data, i.e. once for each training data and once for average), while Gabriele Fanelli's method requires to calculate for n*m times (n=number of training data, m=size of tests pool), which consumes much more times than the former one. Secondly, Gabriele Fanelli's method made it difficult to determine the range of τ, and in what range to randomly select the τ value directly affects computing time. We can make the following reasonable assumptions: i) If I set a not too big range for the randomly selection, then the good values will fall outside of this range, ii) If I set a very big range to ensure that all good values are included, then it takes a long time to determine the thresholds for each non-leaf node, which I possibly could not afford. Thus, I did not use Gabriele Fanelli's method and it will be introduced in Section 5.2.

There are different factors affecting the calculation of τ, including which feature channels to use, how many feature channels to use at the same time and how to set the relationship between each feature channel. In this project, I have tried several ways to calculate τ and some of them work with different accuracy, while some are broken. I will briefly introduce them in this section and describe the test results of them in the section of Implementation.

i) The first formula I used is based on images, rather than depth files described above. Because there is a mapping between depth and color, I just summed up the blue value (because when the color of a RGB pixel is grey, the red, green and blue value will be equivalent) of each pixel in feature zones F_1 and F_2. This formula is the simplest one and takes the shortest calculation time. However, this formula only contains one feature, i.e. the depth value, and does not use a reference point.

ii) The second formula I used is also based on images but differs with the first one in that I added the center point of the head zone as reference point. This formula sums up the difference between each pixel in the two feature zones and the reference point, and still, the blue values represent the depth values. Introducing the reference point makes each pixel taken into the calculation more specific, since the reference point associate the calculation results with the head position.

iii) Both of the formulas described above are based on images, but file data was preferred due to several defects of the image data. Also, this formula was improved by introducing a new technology that supports the tracking of many human body parts, including head, and will be described in detail in Section 3.5. Since the head zone moves with the head position rather than staying within a fixed zone, I changed the reference point from the center of head zone to the head point (i.e. the head position will be recorded as a point when the head is tracked). The advantage to do so is that, if the reference point is a special point of the head in stead of a special point of the head zone, the calculation result will be more closely linked to the head itself, as well as reducing the impact of changing head position.

iv) As described above, the head zone moves with the head, meaning that it would become bigger when the head moves closer to the camera, and would in contrast become smaller when the head moves away from the camera, which causes the change of feature zones' sizes, too. Then we can make an assumption: if there are two frames of head with the same pose but in different depth, then the feature zones of the two frames would contain different number of pixels, and the sum of each pixel therefore can not be compared. Thus, I changed this formula from summing up each pixel into calculating the average of all the pixels. So the τ value would be the sum of the difference between each pixel in the two feature zones and the reference point, divided by the number of the pixels.

v) The last formula is also based on file data, but an extra feature channel, the position of F_1 and $F_{2,}$ is added, in order to confer higher specificity to the eigenvalue of each pixel in the feature zones. Then the τ value can be calculated as: the difference between each pixel in the two feature zones and the reference point multiplied by the position of feature zones, and then divide the sum of which by the umber of pixels within two feature zones.

3.3. Training

This part is the last step of training works, which mainly aims at using the result of all the work done above to build leaf nodes. Figure 3.5 shows the training process. After one training data enters the root node, it will be directed into left or right child node by the calculated τ value, and finally arrives at a leaf node. Then the orientation of this training data will be added to the corresponding array in this leaf. The information attached to these leaf nodes is statistical result of different orientations from the training data, and I have tried two alternative methods to save such information, which are listed below.

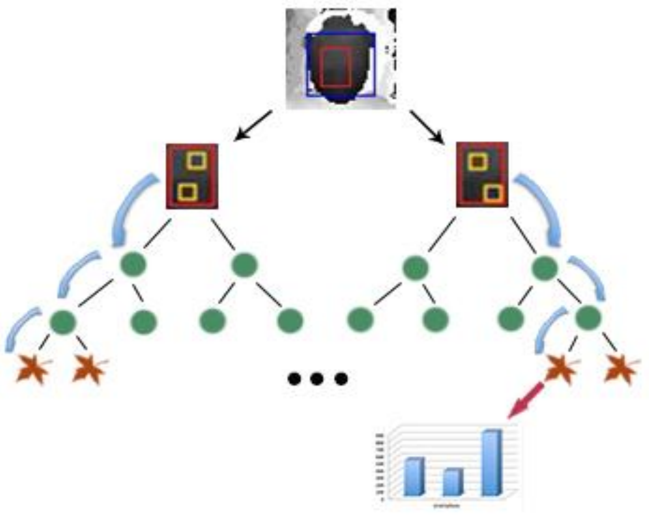


Figure 3.5. Training process.
It shows the training process of one tree. One of the training data goes into the tree from the root node and and arrives at a leaf node via some non-leaf nodes, then the orientation of this training data will be added to the corresponding array in this leaf. After all training data finished going through the tree, there will be a distribution in each leaf node.

i) The first method only saves the orientation that has the highest number of votes in the statistical distribution. We can make two reasonable assumptions here: 1) the distribution in a leaf node is biased, meaning that there is one orientation with the highest value, and this value significantly exceeds the others. 2) the distribution is even, i.e. there are several orientations with similar statistical values. In the first situation, this leaf node should be obviously sensitive to the orientation with the highest votes, and in turn, this orientation could represent the orientation of this leaf node. However, the tree in the second situation can not distinguish some orientations clearly, so the orientation with the highest votes may mislead the final result.

ii) The second method saves the statistical distribution of every orientation in one leaf node, so that we can calculate the proportion of each orientation in the leaf node at which the testing data arrives. Although this method solves the problem brought about by method i), the final results may still be confused if there are some trees not sensitive to some orientations and some trees are not sensitive to some other orientations.

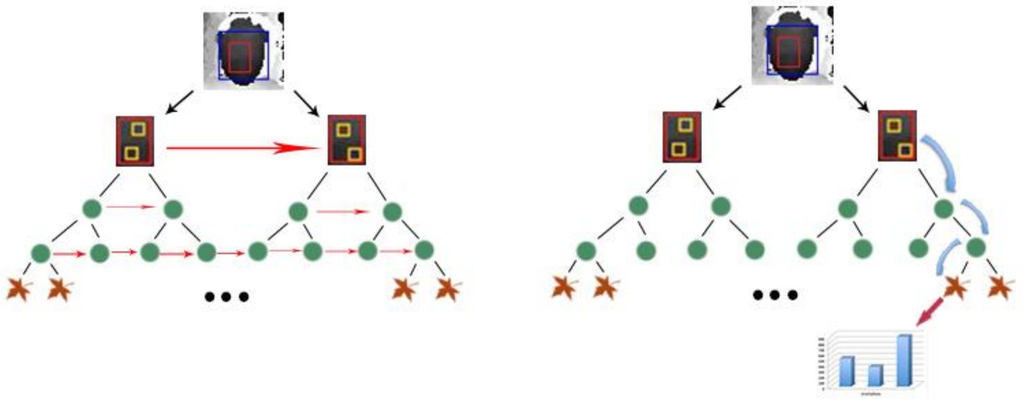
Both of these two methods contain pros and cons, and we can only determine which one is suitable for this project through experiment, the experiment result and analysis will be described in Section 5.

3.4. Testing

Testing is at the application layer. The main purpose of this step is to give the trained forest new, unseen images of head, then the program will return the estimation of the head orientation, as well as to measure the accuracy of the program.

One thing needs to be introduced before describing how the estimation runs is the creation of testing data. There is no doubt that the testing data must be in the same format as the training data, only that the training data is saved into files, while we just use the testing data without saving. This means that after calculating one testing frame, the next frame can be processed without saving the previous ones. According to the Kinect for Windows SDK, there are three events can be registered for different kind of images, they are ColorFrameReady(), DepthFrameReady() and SkeletonFrameReady(). In addition, AllFramesReady() can monitor multiple image events at the same time. Once the camera generates a frame, the event will be triggered， and the trigger frequency of these events is 30 frames per second, since the Kinect camera can create 30 frames per second.

Estimating the orientation of an image requires this image to go through all the trained trees. Due to the trigger frequency of the event, the whole-forest calculation of this image must be finished within 1/30 second, which includes classifying the image in each tree and counting the leaf nodes at which the image arrives. In theory, making full use of the 1/30 second leads to the highest accuracy and the largest range of rotation detection, because only in such situation can the deepest trees and the largest number of trees be achieved. Another thing needs to be taken into consideration is the training time. As mentioned above, the process of calculating the τ value is different from testing, which requires training data to traverse all the trees. This means that the training data enter one node to calculate the τ value one by one, and each training data needs to go through all of the nodes in the whole forest. Figure 3.6 illustrates the difference between training and testing.



(a). Training. **(b). Testing.**

Figure 3.6 Process of training and testing.
It shows the difference between training and testing. The data will arrives at all nodes in training step, while there is only one node will be visited in each layer of the tree in testing step.

All trees in the forest are full binary trees. Thus, if the depth of the trees increases by one, the times of calculation would increase by one for each tree during testing, and if there are 50 trees in the forest (50 trees is the final size of the forest in this project), the times of calculation would only increase to 50 in total during testing. However, when it comes to training steps, if the depth of the forest increase by one, the number of nodes in each tree would double, which in turn duplicates the time cost in this step. Therefore, how to balance the accuracy and time-consumption is the issue brought about by the size of the trees. How to determine the size and what the size is will be described in Section 5.

Increasing the accuracy of head rotation estimation is as important as achieving estimation itself. Some methods that may help increase estimation accuracy and their assessment are listed below:

i) Increase the depth of the trees. One benefit brought by this method is that it dramatically increases one tree's capacity in recognizing different orientations. Basically, increasing the depth of the trees increases the number of leaf nodes, which means that the capacity of information storage will be doubled. For instance, if a tree has 4 layers, the number of the leaf nodes would be 8, so that this tree can only classify 8 orientations. However, if the number of layer increases by 1, the number of leaf nodes would become 16 and this tree therefore can classify 16 orientations. Nevertheless, time limitation will be another factor that must be taken into consideration.

ii) Increase training data. There are two ways to increase training data. The first one is to increase the number of training data. The low quality of Kinect device manifests as that even for static objects, the depth value keeps fluctuating. Thus, increasing the size of training set can decrease the impact caused by this fluctuation. However, when the size of training set arrives at a certain level, the increasing of accuracy will slow down or even stop. The second one is to increase the range of training set, i.e. the person is not only in one pre-selected position. This method fits for moving people, which is an expanding work and will be described below. Since the τ value I used is an average value, it cannot reflect this kind of training data, which requires the equation in non-leaf nodes to be changed. Thus, this method was abandoned in this project, but how to change the equation will be described in the next paragraph.

iii) Improve the equation. There are no standard criteria for how to improve the equation, but several factors may be important. Adding new feature channels and changing the relationship between feature channels would affect the result. Using two τ values or more instead of one τ could be an alternative way for the method described in the previous paragraph. If multiple τ values are used in the equation, the breadth of the trees will increase. Figure 3.7 shows the change

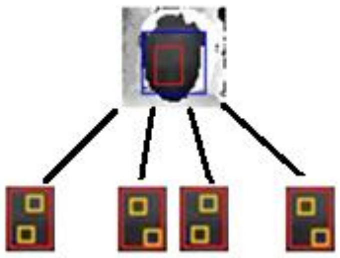


Figure 3.7. Multiple thresholds.
There are two τ values, τ_1 and τ_2, the four children are greater than τ_1 and greater than τ_2, greater than τ_1 and smaller than τ_2, smaller than τ_1 and greater than τ_2 and smaller than τ_1 and smaller than τ_2 respectively.

iv) Use skeleton. This method uses the SkeletonFrame in

Kinect for Windows SDK to monitor the positional relationship between head and shoulders during head pose estimation. The SDK provides API to track 20 joints, including head, shoulders and many other parts of the body. This technic will be fully described in below. Using this method helps determine some specific orientations. Figure 3.8 shows one image, within which the head, shoulders and the shoulder centre joints are displayed.

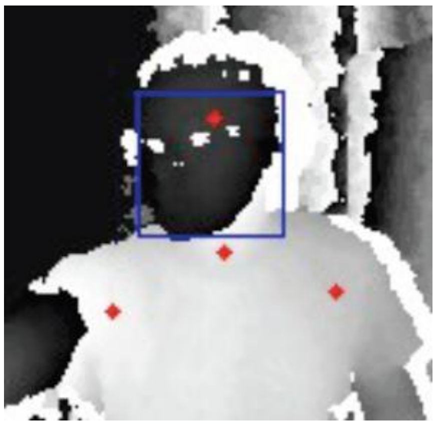


Figure 3.8. Data with skeleton.
It shows one image with skeleton tracked. 4 red points are head, left-shoulder, right-shoulder, shoulder-centre respectively.

The last task in this step is to work out the correctness. This program runs in real-time so that it is difficult to record or calculate the result. The best way to test it is through cross-validation, the details and result of which will be described in Section 5.

3.5. Expanding

This step is an open-ended issue, and the main purpose of this step is to expand the work done above. Everything can be added to the program as long as it is useful and valuable. In this project, I planed to expand the algorithm to estimate moving people, and in additionally, I will briefly discuss about my idea on estimating the orientations of two or more heads.

I would like to introduce the Skeleton Tracking provided by Kinect for Windows SDK before introducing the two ideas above, since this technic will be used in both of them.

Since there are functional deficiencies in the depth data returned by Kinect, and much needs to be done other than depth data in order to achieve human-computer interaction and produce attractive applications, the Skeleton Tracking has been introduced as the core technology of Kinect. Thanks to this technology which leads to the born of many interesting applications, such as Kinect Adventures [10, 18, 19].

By now, Kinect can track 20 joints of a human body. They are (from top to bottom respectively): head, shoulder-centre, left and right shoulder, left and right elbow, left and right wrist, left and right hand, spine, hip_centre, left and right hip, left and right knee, left and right ankle and left and right foot. Figure 3.9 (a) [19] shows the 20 joints of a human body that Kinect can detect. Figure 3.9 (b) shows a human skeleton that is tracked by a sample program provided by Kinect Developer Toolkit Browser v1.5.1.

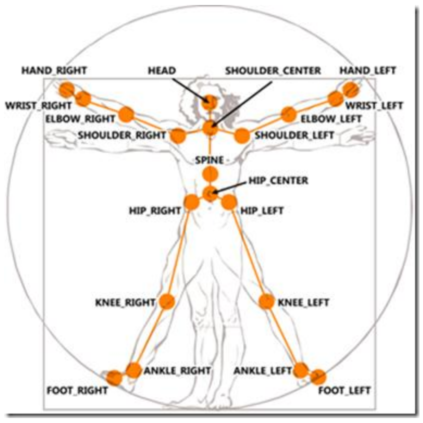


Figure 3.9 (a). Kinect skeleton. Image from [19]
It shows the 20 joints of a human body that the Kinect can detect.

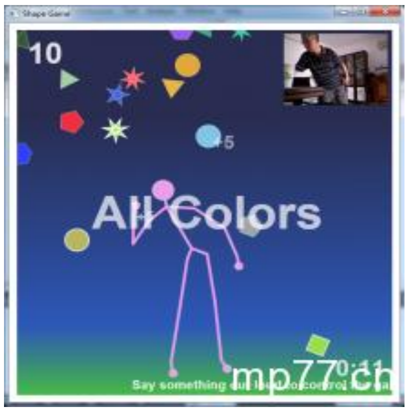


Figure 3.9 (b). Skeleton sample.
It shows a human skeleton that is tracked by a sample program provided by Kinect Developer Toolkit Browser v1.5.1.

Skeleton Tracking can localize the position of people's head and shoulders, which can be used to track head orientation and to increase the accuracy (use the positional relationship between head and shoulders that I described above). In addition, Skeleton Tracking is able to track two people at the same time, and this can be used for estimating more people.

3.5.1. Moving People

The algorithm I described above has achieved the estimation of the orientations of people's head. But several limitations do exist, with the most prominent one being that the person who is in testing must sit in a pre-selected position. Thus, I intended to expand this algorithm to estimate moving people. The changes that I have made for this algorithm are mainly in three aspects: i) track the head, since we must know where the head is before we can process the head image. ii) Set the feature value to be proportional to the distance, due to that if the head is close to the camera, the head would be a big one in the image, while if the head moves away from the camera, it would gradually become smaller. Thus it must be ensured that the size of head zone changes with the head and the F_1 and F_2 change with the head zone. iii) Use the Skeleton Tracking to increase the accuracy, which will be described in Section 5.

Head tracking is the foundation of tracking moving people, it is because that data calculations and processing is largely depend on knowing where the head is. By now, although there are many head tracking algorithms, mainly of them are based on color or 2D images, e.g. CamShift [17], which is an algorithm based on color. In this project, Due to the special characteristic of the training and testing images (they are 3D images), there are two alternative ways to achieve head tracking: i) use algorithm which is based on 2D images and merge the color frames to the depth frames. ii) use the Skeleton Tracking method provided by Kinect for Windows SDK.

i) Using the algorithm based on 2D images has several advantages. For example, in the 2D images based method, the color histogram are established by the training set of people's face color, which is used to distinguish the face from background. This enables the detection of people outside the capable range of Kinect's depth data, since Kinect can only get the depth value when a person's distance from the camera is within 1-4 metres. However, this method is severely affected by illumination. Although merging the depth frame to the color frame is an advisable way to avoid the drawbacks of both methods, it will be a complex work that needs huge time to process.

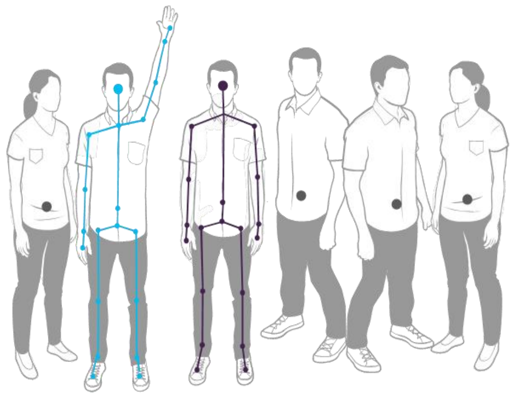
ii) The most obvious disadvantage of Skeleton tracking method is that, the scope of detection is rather small, especially for objects that are close to the Kinect camera, the depth value that Kinect returns would be -1, which means that the objects can not be detected. On the other hand, using skeleton tracking has its own advantages. Firstly, it is easy to use. Kinect for Windows SDK provides API for Skeleton Tracking. Secondly, the running time it requires is less. This can be attributed to its fastness when compared the other algorithm, and also to that since Skeleton Tracking will also be used to increase accuracy, using the same method would reduce calculations. Thirdly, Skeleton Tracking tracks people's head as a point, which is the head joint of the 20 joints of the human body that I mentioned above. This point can be used as the reference point during calculating the feature value (which is described in Section 3.2).

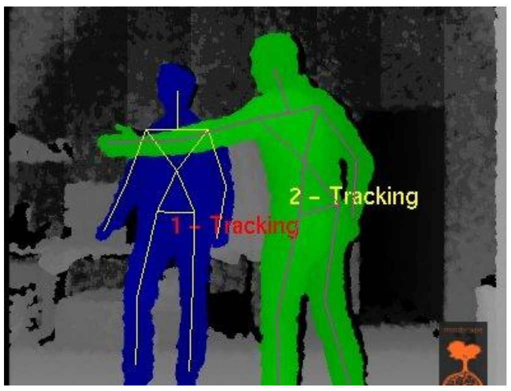
After weighing the advantages and the disadvantages of these two methods, I think the latter one is advisable for this project.

3.5.2. More People

Estimating the orientation of two or more people's head simultaneously requires tracking more than one heads at the same time, as well as a bigger training set and a faster way for image processing.

There are several multiple-face tracking methods, but in this project Skeleton Tracking will be preferred. Skeleton Tracking technology provides the ability of recognizing 6 people at most and tracking 2 people at the same time [20]. Figure 3.10(a) shows how Skeleton Tracking works with multiple people, and Figure 3.10(b) shows skeleton of two people are tracked by Kinect. The two head points can be used to track the corresponding two heads simultaneously, and then we can calculate the feature value and do other processes for the two head zone respectively.





(a). recognizing 6 people and tracking 2. **(b). tracking two people in an application.**

Figure 3.10 (a) shows how Skeleton Tracking works with multiple people, 3.10 (b) shows two people's skeletons are tracked in an application.

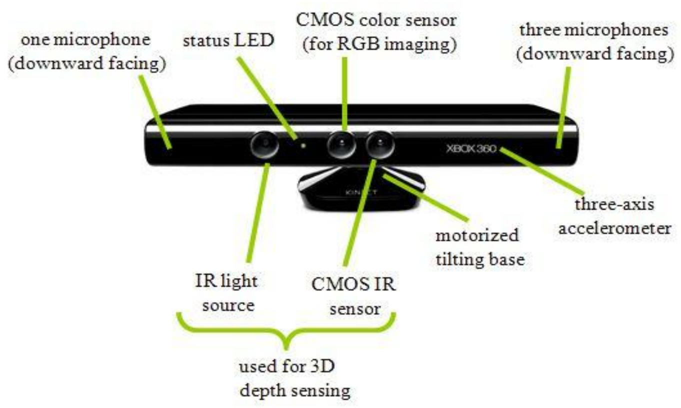
The running time problem can be improved by using two threads for the two head zones respectively. Thus, one head zone does not need to wait until the processing for the other one ends.

All above in this section are the technic basis of this project and some alternative methods can be used in this project. The main purpose of this section is to describe how the algorithm I used works, the implementation of each part will be discussed in detail in Section 5.

4. Summary of 3rd Party Hardware/Software

1. XBOX 360 Kinect sensor.

    * RGB VGA Camera with resolution of 640*480.
    * IR depth sensor.
    * Multi-array microphone.



one microphone (downward facing)
status LED
CMOS color sensor (for RGB imaging)
three microphones (downward facing)
three-axis accelerometer
motorized tilting base
IR light source
CMOS IR sensor
used for 3D depth sensing

Figure 4.1. The Kinect hardware. Image from [21].
This project mainly uses the IR light source and CMOS IR sensor to get depth data.

2. Kinect for Windows SDK v1.5.

    * http://www.microsoft.com/en-us/kinectforwindows/
    * Include divers for developing a Kinect sensor on Windows 7, Windows Embedded Standard 7, and Windows 8 Developer Preview.
    * It provides APIs that necessary to develop applications with Kinect using C#, C++ or VB programming language.
    * It provides documents.

3. Developer Toolkit Browser.

    * It contains many source code samples that are related to all functions of the Kinect device.
    * It provides some SDKs to simplify the developing on Kinect for Windows SDK, e.g. Face Tracking SDK.

4. Coding4Fun Kinect Toolkit.

    * http://c4fkinect.codeplex.com/.
    * It provides several methods for programs on WPF and WinForm to

simplify the developing.

5. Emgu CV

* http://file.emgu.com/wiki/index.php/Main_Page.
* Emgu CV is a cross platform .Net wrapper to the OpenCV image processing library [22].
* Emgu CV provides many image processing methods, image format conversion methods are frequently used in this project.

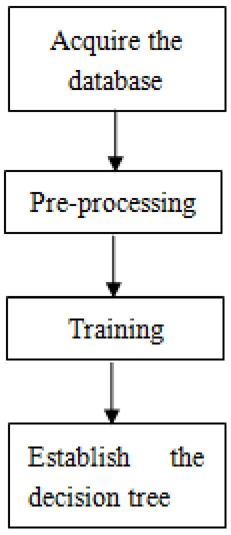
5. Implementation and Results

5.1. Implementation

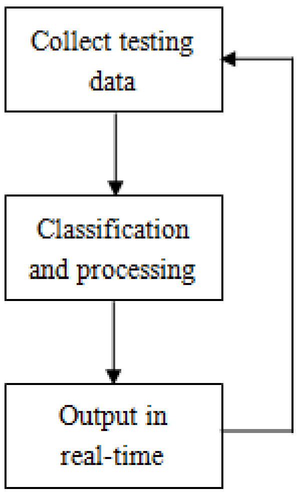
The technic basis, main method and the alternative methods in each part of this project has been described in Section 3. The sequence of these methods is the program framework. In this section, I will introduce how to implement all these methods.

Firstly, I will give the program flowchart. The whole running process can be divided into 3 parts, since there are two places that the program can not proceed without manual intervention, the two places are: i) after building the whole forest, before testing. ii) before the cross validation. Figure 5.1 shows the flow charts of the 3 parts.

In this section I will introduce all parts of them by execution order, i.e. training, testing, and finally cross validation.



Training



Testing

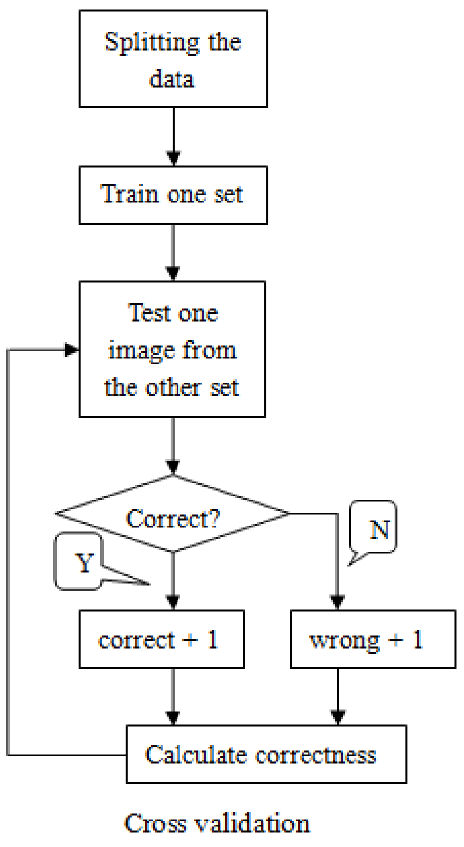


Figure 5.1. Program flow chart.
The program is divided into 3 parts by time. First step is to train the forest by training set. Second one is to estimate human head in real-time. Thirdly, use cross validation to test the correctness.

5.1.1. Training

The first step in training is data capture. I have introduced in Section 3.1.2 that Gabriele Fanelli et al. uses templates and ICP [5, 16] to calculate the orientation of training data. However, there are no templates available in my project. Thus, I must find out a way to guarantee the training data are in the correct orientation. Figure 5.2 shows the room that I worked in. We can see that the Kinect camera was placed on the windowsill. The chair was placed two metres in front of the camera, as the pre-selected position for testing people. I made some tags in this room and each tag represented a specific orientation. Thus, a correct head orientation was guaranteed when a person was sitting on the chair and facing to a tag. Figure 5.2 shows the position of the camera, the chair and 3 tags.

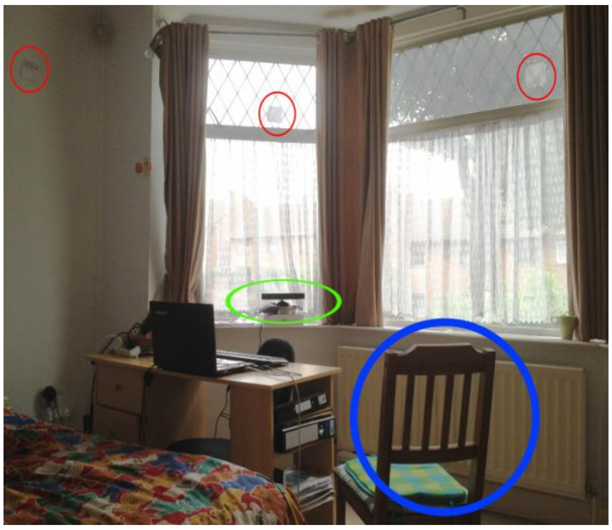


Figure 5.2. Work environment.
This figure shows the room where the project was worked. The three red circles are the position of three tags, as they represent “ul45”, “u0” and “ur45” (they are names of orientations which will be introduced below) respectively. The green circle points out the position of the Kinect. Testing people will sit on the chair where the blue circle indicates.

I have set 6 orientations and 500 images as a training set for each orientation. The 6 orientations are: “ul45”, “u0”, “ur45”, “bl45”, “b0” and “bl45”, “u” means the head is upward, “b” means the head is downward, “l45” and “r45” means left or right 45 degrees, “0” means frontal face. Figure 5.3 shows the 6 orientations in color images.

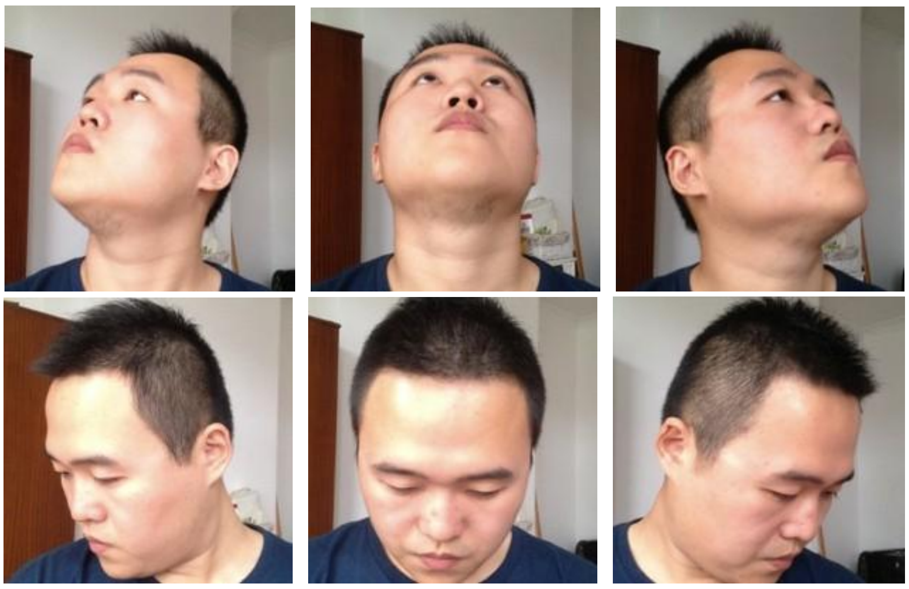
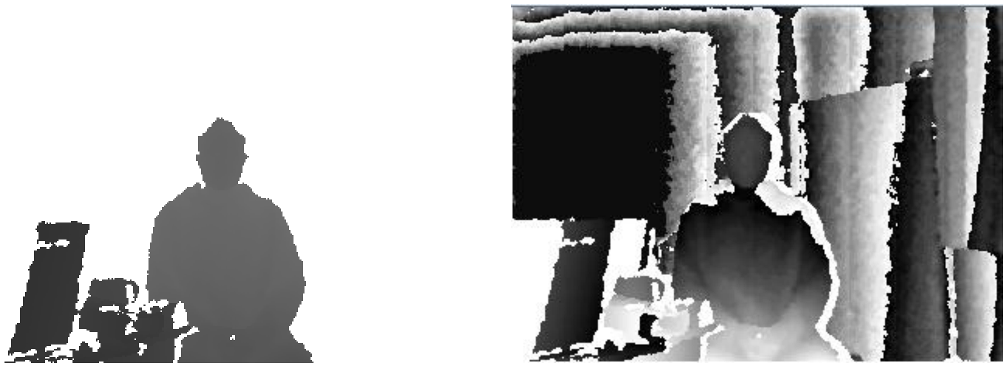


Figure 5.3. 6 Training orientations.
The images are: “ur45” (upper left), “br45” (lower left), “u0”

(middle and upper), “b0” (middle and lower), “ul45” (upper right), “bl45” (lower right).

The size of the training set is determined by the balance between the training time and accuracy, as I described in Section 3.2.

As I mentioned in Section 3, there are two different image formats that can be used, RGB images and Grey ones, and I decided to use RGB images due to its good visual effect. According to Kinect for Windows SDK, the frame monitored by DepthFrameReady() must be stored in a short array, then can a 16-gray-level image be created by calling toBitmap() method. In this project, I used a Byte array to convert the image to Bgr32 format, each of blue, green and red occupies 8 bits and the left 8 bits are empty. Since I set the same value for blue, green and red in order to display only gray on the screen, the gray level increased from 16 to 256. There are two images below, one is RGB image, and the other one is Gray image.



(a) RGB image (b) Gray image

Figure 5.4. One frame in different format.
Figure 5.4 (a) uses Byte array to store images, there are 255 gray levels. (b) uses short array to store images, there are 16 gray levels.

I have evaluated two methods for data storage in Section 3, i.e. saving RGB images and saving depth value in files. Table 5.1 shows the comparison between these two methods.

Table 5.1 Comparison between saving images and files.

	Saving RGB images.	Saving depth value in files.
Size of training data.	6 orientations and 500 images for each orientation.	6 orientations and 500 depth value files for each orientation.
Storage space.	About 7.5 Mb	About 170 Mb
Time for calculating τs.	2 ~ 3 hours	17 ~ 20 hours
Time for build leaf nodes.	About 10 minutes	About 1 hour
Testing result.	Obviously inaccurate in real-time testing, almost can not estimate.	Estimation is not stable in real-time but almost accurate, high correctness in cross validation.

According to this table, the two methods used training sets with the same size, used the same equation to calculate τs and build lead nodes, while the time and storage consumptions and result significantly differed from each other.

Image training set is about 25 times smaller than file training set, while processing images only needs about 1/10 of time needed for processing files. Thus, saving the RGB images as the training set supposed to be better than the other method for the purpose of time and space saving, however, the result of this method is unsatisfactory. As I am concerned, there are several possible reasons:

i) When drawing the head zone in the image, surrounding pixels will be affected. I drew a blue rectangle to indicate the head zone, and the line width was set as two pixels. However, when I printed the green value of each pixel within the image, I find about 1 to 3 adjacent pixels of the line are not gray any more (the blue, green and red value are not equivalent). This problem can be solved by saving a file that records the position of head zone but no longer drawing rectangles. The disadvantage of this solution is that testing people will not know where the head zone is.

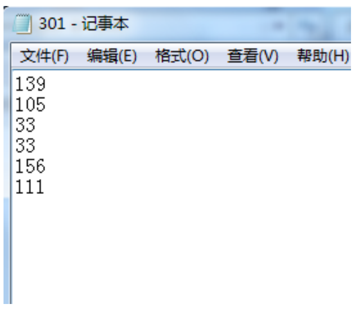
ii) The image changes slightly when being saved or being read. To exam this assumption, I saved the images and then read it immediately, thus there will be no operation between saving and reading. I used Equals() method provided by EmguCV to check whether the current frame changed after saving and reading to the program again. The result showed that they were always not the same. As I am concerned, the reason should be that the image was compressed when being saved. For instance, 1000 continuous pixels (name the first one as *p*) fluctuate around a particular value ε, but they may be saved as "1000 εs starting from pixel *p*". Thus, the change of color led to undesirable results.

iii) Color-level. Bgr32 images were used as training data. To make the images only display gray color, I set the value of blue, green and red equivalent. Since one color occupies 8 bits, there would be 2^8 levels of gray in total in one image. As I mentioned before, the detection range of Kinect's depth camera is between 1000mm and 4000mm, while the color-level is 256, so that every 11.7mm will be represented by one color when mapping the depth value to color. However, 11.7mm is not meticulous enough to reflect a head because one head may only be represented by 1 to 3 colors in this case. An alternative way could be using more colors. For instance, using 3000 colors then one depth value would only correspond to a specific color. However, it makes the program slow and lagging.

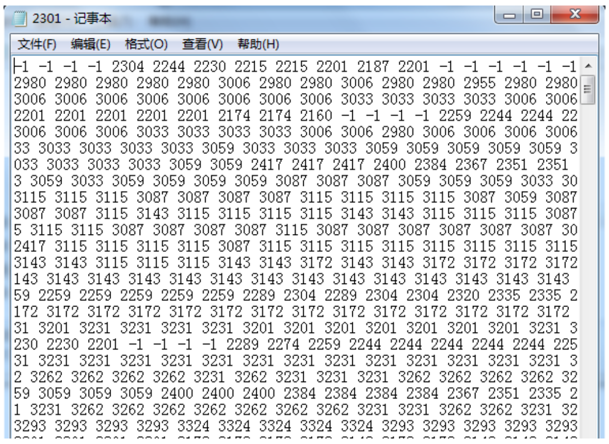
Based on the above reasons, I finally made the decision to give up image data and concentrate on file data.

I have introduced the size, classifications and format of training set. After making these decisions, I needed to build the training set and then preprocess the data. There are three steps in the data capture and preprocessing.

i) Saving Training data. There are several ways to save the training data according to different ways to calculate τs (introduced in Section 3.1). I will only introduce the data-saving way corresponding to the last method of calculating τs listed in Section 3.1.2. Firstly, I created 6 folders named by the six orientations introduced above, then the pose of testing person's head was determined by the tags that I introduced before, and frames were saved in the corresponding folder. One frame will create two files, one records the depth value of every pixel in the current frame, the other one records the position of the reference point, the size of head zone and the position of head zone. There will be 800 frames saved for each orientation. Figure 5.5 shows a file-data example of one frame.



```
139
105
33
33
156
111
```

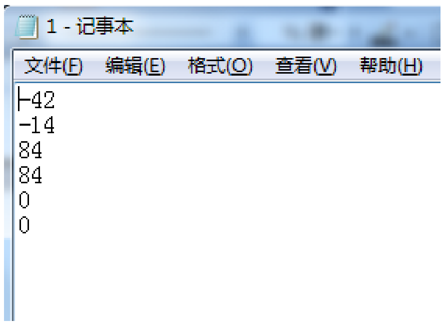


```
-1 -1 -1 -1 2304 2244 2230 2215 2215 2201 2187 2201 -1 -1 -1 -1 -1 -1
2980 2980 2980 2980 2980 3006 2980 2980 3006 2980 2980 2955 2980 2980
3006 3006 3006 3006 3006 3006 3006 3006 3033 3033 3033 3033 3006 3006
2201 2201 2201 2201 2201 2174 2174 2160 -1 -1 -1 -1 2259 2244 2244 22
3006 3006 3006 3033 3033 3033 3033 3006 3006 2980 3006 3006 3006 3006
33 3033 3033 3033 3033 3059 3033 3033 3033 3059 3059 3059 3059 3059 3
033 3033 3033 3033 3059 3059 2417 2417 2417 2400 2384 2367 2351 2351
3 3059 3033 3059 3059 3059 3059 3087 3087 3087 3059 3059 3059 3033 30
3115 3115 3115 3087 3087 3087 3087 3115 3115 3115 3115 3087 3059 3087
3087 3087 3115 3143 3115 3115 3115 3115 3143 3143 3115 3115 3115 3087
5 3115 3115 3087 3087 3087 3087 3115 3087 3087 3087 3087 3087 3087 30
2417 3115 3115 3115 3115 3087 3115 3115 3115 3115 3115 3115 3115 3115
3143 3143 3115 3115 3115 3143 3143 3172 3143 3143 3172 3172 3172 3172
143 3143 3143 3143 3143 3143 3143 3143 3143 3143 3143 3143 3143 3143
59 2259 2259 2259 2259 2259 2289 2304 2289 2304 2304 2320 2335 2335 2
172 3172 3172 3172 3172 3172 3172 3172 3172 3172 3172 3172 3172 3172
31 3201 3231 3231 3231 3231 3201 3201 3201 3201 3201 3201 3201 3231 3
230 2230 2201 -1 -1 -1 -1 2289 2274 2259 2244 2244 2244 2244 2244 225
31 3231 3231 3231 3231 3231 3231 3231 3231 3231 3231 3231 3231 3231 3
2 3262 3262 3262 3262 3231 3262 3231 3231 3231 3262 3262 3262 3262 32
59 3059 3059 3059 2400 2400 2400 2384 2384 2384 2384 2367 2351 2335 2
1 3231 3262 3262 3262 3262 3262 3262 3262 3231 3231 3262 3262 3231 32
3293 3293 3293 3293 3324 3324 3324 3324 3324 3293 3293 3293 3293 3293
```

Figure 5.5 Example of file-data.

These two files correspond to one frame. The 6 numbers in the top one are the abscissa and the ordinate of reference point, height and width of head zone, the abscissa and the ordinate of the upper left point of the head zone. The bottom file contains the depth value of each pixel in this frame, -1 means the object is out of the detection range.

ii) Preprocessing the training set. There were 800 frames saved as training data for each orientation. The first thing was to delete the first 300 frames, this is because that the program may start saving before the head is tracked, the position of the reference point and head zone would not be correct in this situation. Figure 5.6 shows an example of such data. After the first 300 frames were deleted, the 301st frame also needed to be checked to ensure that the rest 500 frames were correct. If incorrectness occurred to the 301st frame, it was necessary to re-collect data for this orientation.



```
-42
-14
84
84
0
0
```

Figure 5.6 A negative data.
This data was saved before the head was tracked. The position of the reference point and head zone are incorrect, the size of the head zone is the maximum value.

iii) Divide the training set into two parts. I used cross validation to test the correctness. I picked out one frame form every five frames as the testing data. The other four frames were saved as they came.

Until now, data capture and data preprocessing were finished. The next step was to establish the forest. There are two structures can be used to establish the forest, one is linked list, the other one is array. Using linked list to build tress has its own advantages since the linked list does not have empty node to consume memory, and is easy to add or remove nodes, while the disadvantage is that searching is slow. In contrast, the advantage of using array is fast searching speed, but the disadvantages exist as it occupies more memory and adding and removing nodes are complex. In this project, all trees in the forest are full binary trees. Thus, there will be no empty nodes in the array, meaning that the array does not occupy more memory than linked list, and is faster than linked list. Therefore, I stored the forest in a three-dimensional array, the first dimension is trees, the second dimension is nodes of the trees, and the third dimension is the information in nodes. The left child of node “n” is “2n+1”, the right child of node “n” is “2n+2”. Figure 5.7 shows the forest.

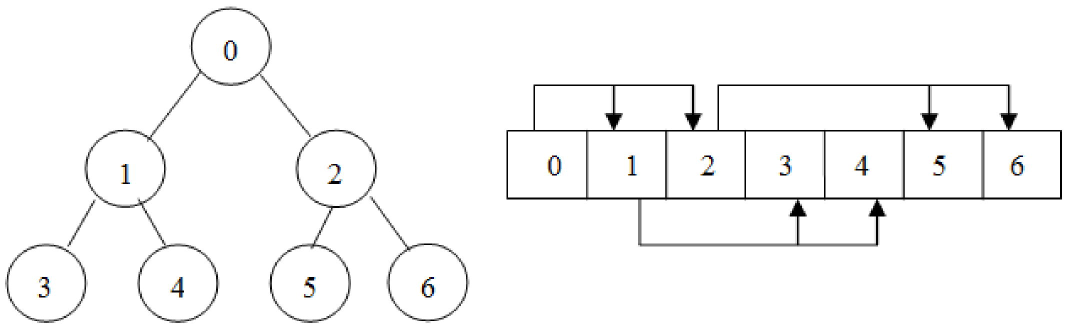


Figure 5.7 Tree stored in array.
There is no empty node in the array when the trees are full binary trees. Left and right child of node "n" are "2n+1" and "2n+2" respectively.

As mentioned in Section 3.4, balancing the training time and accuracy is a time-consuming work. Firstly, I set 5 as the depth of each tree so that every tree will have 15 leaf nodes. Then I tried to run the program with different sizes of forests. Table 5.2 shows the result of different sizes of forests.

Table 5.2 Results of different sizes of forest.

	30 trees	40 trees	50 trees	60 trees
Training time	Less than 15 hours	17 ~ 19 hours	20 ~ 23 hours	More than 25 hours
Testing situation	Fluent	Fluent	Lags only when the head is very close to the sensor	Fluent only when the head is far away from the sensor
Accuracy	Can estimate the orientation but not accurate	Accuracy increases obviously	Accuracy increases but not too much	Not fluent so that can not display results in real-time

As we can see from Table 5.2, setting the forest size as 40 trees and 50 trees are both feasible. If I choose 40 trees, I still can't afford the training time for 6-layer-trees, which is because that one more layer of the trees doubles training time. Thus, I finally made the decision to use 50 trees in total.

By now, there were 50 trees and 31 nodes in each tree, but each of the node-dimension of the three-dimensional array needed one more place to store the position of the patch in this tree. Thus, the sizes of the first two dimensions of the array were determined, i.e. 50 and 32 respectively. The third dimension stores the information within nodes. There are two different kinds of nodes, non-leaf nodes and leaf nodes, while the information they stored are totally different. For this reason, I divided node-building into two parts in Section 3, which are named Calculate the Test Thresholds and Training. Still, I will describe them separately in this section.

Building non-leaf nodes was the most complex work in this project. The time needed for building non-leaf nodes was more than 5 times than that for building leaf nodes. A non-leaf node contains three things, they are: i) a randomly selected patch within the head zone, ii) two randomly selected feature zones within the patch, iii) the threshold of the node.

i) One tree, one patch. A patch is randomly selected for one tree, which means that all the non-leaf nodes in this tree will share this patch. Assume *forest* is the array, *forest*[x,0,0] and *forest*[x,0,1] store the position of the upper left point of the patch. The size of a patch is defined as quarter of the head zone. I randomly created 50 numbers between the smallest and the biggest height of the head zone, and then modulo these numbers by half of the height of head zone, the results are the ordinate values of the upper left point of the patches. The abscissa values of the upper left point are calculated in the same way. Figure 5.8 shows the method of randomly selecting a patch.

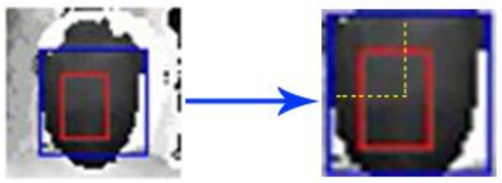


Figure 5.8 Randomly selected patch.
This picture illustrates how to randomly select a patch. Picture on the right is the amplified head zone. The red rectangle is a patch. The upper left point of this patch is randomly selected within the yellow dotted line rectangle, and the side length of patch is half of that of the head zone.

ii) One node, two feature zones. Feature zones are similar to patch, only that they are two areas randomly selected within the patch, and each node has its own feature zones. The way to select these zones is similar to selecting patch, however, if the upper left points of the two feature zones are the same, one of these will be selected again until they are not the same.

iii) The threshold. The threshold is a value used to determine whether a frame should go to the left or right child. As mentioned in Section 3.2, I have tried 5 different equations to calculate the threshold. Table 5.3 shows the result of them respectively. Because it takes too many words to describe which method it is, I just name them 1st to 5th here; same as the order I introduced them in Section 3.2.

Table 5.3 Ways to calculate threshold.

	1st	2nd	3rd	4th	5th
Training time	Less than 2 hours	Less than 2 hours	About 15 hours	More than 15 hours	More than 15 hours
Testing situation	Fluent	Fluent	Fluent	Fluent	Fluent
Accuracy	Could not work	Could not work	Worked but had strict requirements on head position	Higher accuracy than previous methods	The most accurate

As we can see from the table, all of the five methods can be done in 1/30 second in testing. The first two methods were broken in result because they are based on image training data that I have described and analyzed in Section 3.2 and above in this section. The third one is inaccurate because it did not take the average of pixels in feature zones. The last two needed more training time than the third one because there were double variable arithmetic and division operations that occupied more time. The 5th one is more accurate than the 4th one because it used multiple feature channels (depth and relative position instead of depth only).

Building leaf nodes was based on all completed non-leaf nodes. Training data were continuously tested by non-leaf nodes until they finally arrived at a leaf node. As I mentioned in Section 3.3, there are two ways to build leaf nodes, the orientation with the highest proportion represents the node and record the number of occurrences of each orientation. Table 5.4 shows the comparison of the two methods.

Table 5.4 Comparison between two leaf nodes building methods

	Highest one	Record all number of occurrences
Time of building leaf nodes	Less than 2 hours	Less than 2 hours
Testing situation	Fluent	Fluent
Accuracy	Could only estimate 2 or 3 orientations in real-time	Much more accurate than the other one

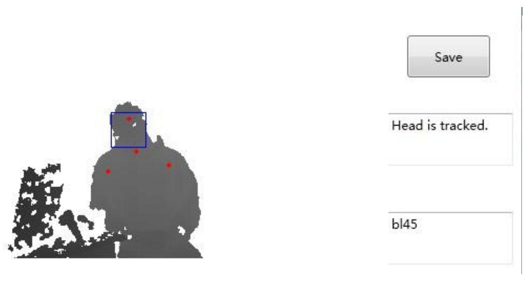
The latter method is obviously better than the other one since it avoided the interference of orientation-insensitive nodes.

5.1.2. Testing and Cross Validation

Then it came to testing after finish building the forest. As I mentioned in Section 3.4, testing is to estimate new, unseen frames with the completed forest. A testing frame will arrive at one leaf node in each tree, so that there will be 50 leaf nodes supporting the estimation of this frame. Sum up the proportion in each leaf node for each orientation, and the orientation with the highest summation will be defined as the estimation result of this frame. The key point of this part was the time interval between two frames, i.e. 1/30 second. Making full use of this time can increase accuracy, however, once the calculations cannot be completed within this time, the fluency of the testing would be broken. In Section 3.4, I have fully described how to make full use of this time.

Cross validation was used to test the correctness of the estimation. This was a time-consuming work, so I decreased the size of data in each orientation from 500 frames to 300 frames. I made the data size used to train 4 times as the data size used to test. In addition, to quantify the result of real-time testing, I sampled new testing frames and tested them with the trees established by training data. All of the testing results will be described in Section 5.2.

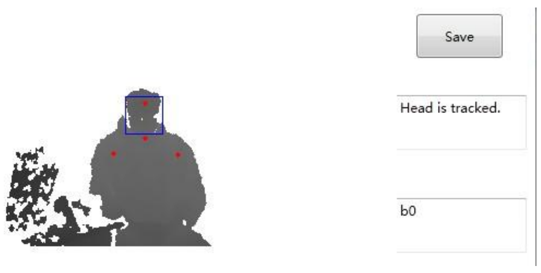
Skeleton Tracking has already been introduced several times in this thesis. One of its functions in this project was to increase the accuracy. My program only displayed 4 of the 20 joints, i.e. head, left shoulder, right shoulder and shoulder centre. I added the positional relationship into the orientation weighting process. As I observed, when the difference between the abscissa value of head point and the abscissa value of shoulder centre point was greater than 2, the head was possibly not frontal. Thus, if the head was close to left shoulder or right shoulder, then I would increase the weight of “l45” or “r45” (they are head orientations described above). The advantage of this method was that it significantly increased the accuracy in real-time testing and was easy to achieve. Figure 5.9 shows some frames with skeleton tracking in different orientations. The limitation of this method was that the rotation in Roll angles would confuse the estimation. Figure 5.10 shows an example in this situation.



Head is tracked.

bl45

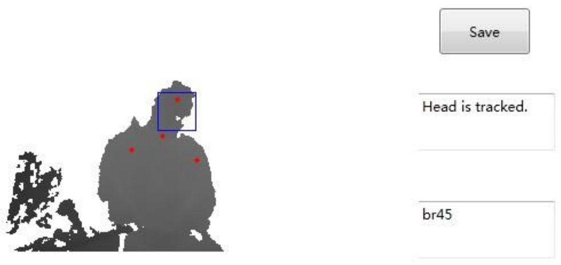
(a)



Head is tracked.

b0

(b)



(c)

Figure 5.9 Real-time estimation with skeleton tracking. Figure 5.9 (a) and Figure 5.9 (c) shows the person rotates head in Yaw and Pitch angles instead of facing frontal to the camera, the head point will not stay in the same abscissa position with shoulder centre. Figure 5.9 (b) shows if the head only rotate in pitch angle from the frontal face, the head point and shoulder centre will stay in same abscissa position.

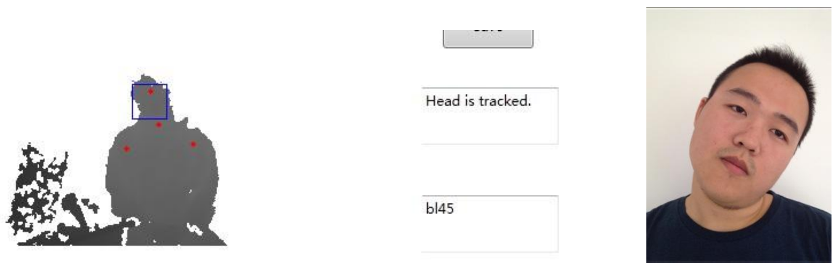


Figure 5.10 Estimation confused by Roll angle rotation. The left picture shows the image displayed by program, while the head in reality is shown in the right picture. The estimation was confused.

5.1.3. Expanding

By now, estimating the orientation of people's head has finished. The next work was to expand the algorithm to fit moving people. As I described before, there were mainly three changes I added into the program for the expanding, head tracking, setting feature value to be proportional to the distance and increasing the accuracy by using the position of shoulders. Actually, if the training data are collected in a pre-selected position and the testing is executed in this position, estimating the people sitting in the pre-selected position can be considered as a special situation of moving people. All of the work I mentioned before is running in the expanded version. Head tracking and increasing accuracy are fully described in the previous section and above in this section. I will describe why and how to set feature value proportional to the distance below.

The detection range was determined as 800mm to 1800mm. When the head is at the maximum distance, it can be framed by a 60*80 rectangle, while the head can be framed by a 30*40 rectangle at the minimum distance. Mapping the detection range, 1000mm, to the incrementals of the width and height of head zone, which comes (1800 – 800) / (80 or 60 – 40 or 30), and then let variable α equals to it. Thus, the height or width of head zone in current distance is 80 or 60 – (depth – 800)* α. This program was mainly based on this formula, but the parameter α has been adjusted and modified. Figure 5.11 shows the head zone in different distances.

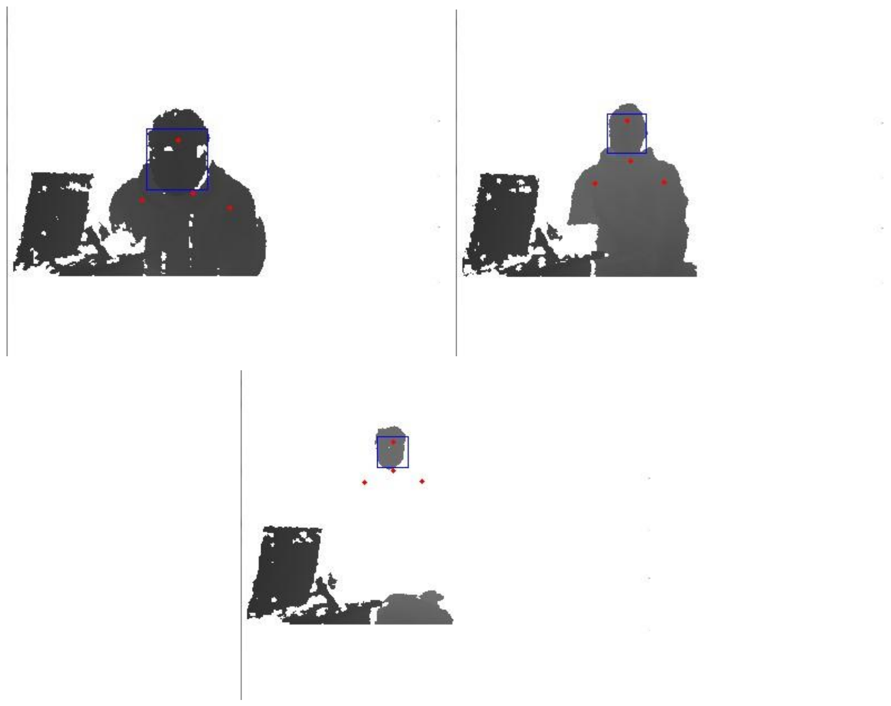


Figure 5.11 Head in different distances.
Objects closer than 800mm and farther than 1800mm are not displayed. The three pictures above shows the head zone proportionally becomes larger or smaller with the head.

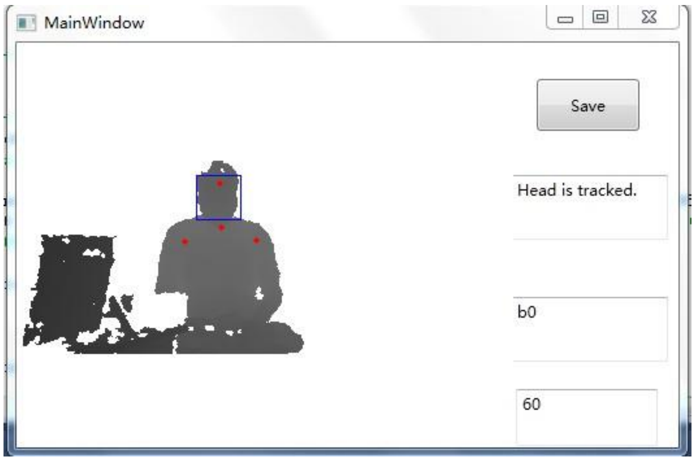
So far, I have introduced what I have done in my project, and the result of my project is in below.

5.2. Testing Results

This section details the testing results in some different situations. These results will be used to analyze the performance of the system of my program.

5.2.1. User Interface

The development of the user interface in this program was based on WPF (Windows Presentation Foundation) [23], which is developed by Microsoft, and provides developers with a unified programming model for building rich Windows smart client user experiences that incorporate UI, media, and documents [23]. The main interface contains one image box, one button and three text boxes. The image box displays image stream, the button is used to save the current frame (it differs from data capture in that once the user click this button, only the current frame will be saved), the top text box shows whether the head is tracked, the middle one shows the estimation result, the bottom box shows the weight of the orientation shown in the middle one. Figure 5.12 shows the UI of this program.



Head is tracked.

b0

60

Figure 5.12 User interface.
The main interface consists of one image box, one button and three text boxes.

5.2.2. Static Object

As training is a time-consuming work, I must test my program with small size training set first, in addition, the testing object was not person but a static object. If the program could not work well on this kind of training set and testing object, the program would not work well on people’s head as well. The object was set in three different poses. Figure 5.13 shows the testing object in three poses.

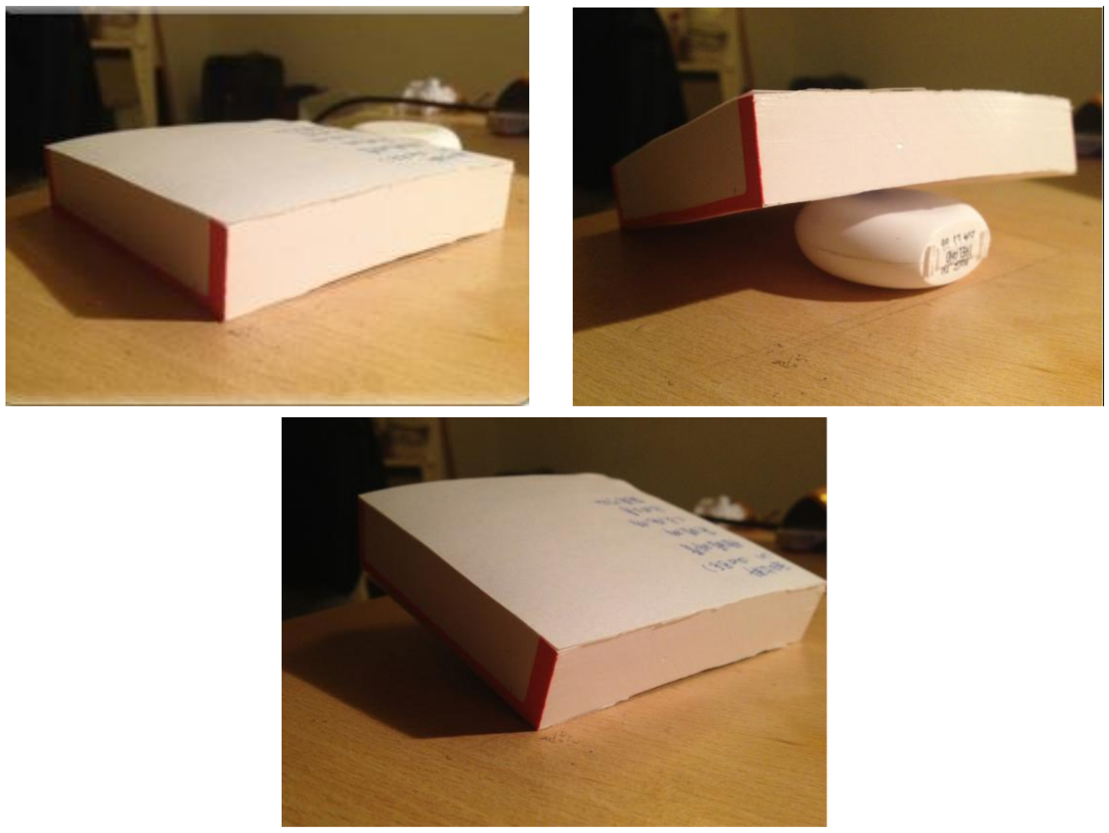


Figure 5.13 Static testing object.
The three figures illustrate the three poses of static testing object. They were named “m0”, “u0” and “b0” respectively.

I named the three poses “m0”, “u0” and “b0” respectively, and collected 100 frames for each pose as the training set. All of the testing results of these three poses were always correct. It means both of the random regression forest I built and the program worked well, the following work is to adjust the program and test it with people.

5.2.3. Head in 3 Orientations

Firstly, I tested whether the program works with people’s head in only 3 orientations, “m0”, “u0” and “b0”, as I have introduced “u0” and “b0”, similarly, “m0” means the head is facing frontal to the camera. The cross validation result in this situation is shown in Figure 5.14. The result of testing in real-time is shown in Table 5.5.

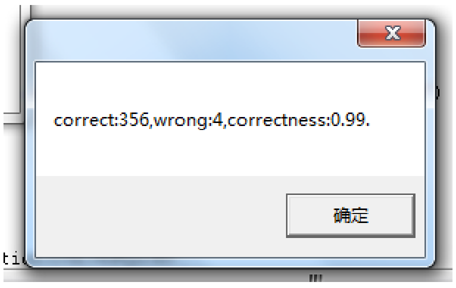

correct:356,wrong:4,correctness:0.99.

Figure 5.14 Cross validation result of 3 orientations.

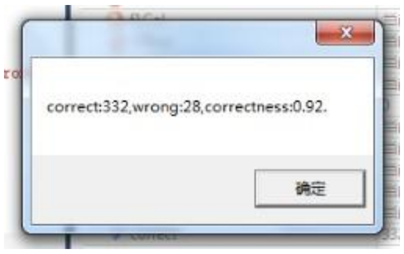
Table 5.5 Three Orientations real-time result.

	u0	m0	b0
Correct	46	43	50
Incorrect	4	7	0
Accuracy	0.92	0.86	1.00

As we can see in Figure 5.14 (c). There were 360 frames used to test the program, and 356 of which were estimated correctly, 4 of which were misestimated, 0.99 is a very high correctness. Table 5.5 shows the real-time testing of 3 orientations estimation. Testing people was asked to turn his head to each of the three orientations for 50 times, then the result was recorded in Table 5.5.

5.2.4. Head in 6 Orientations

I have tested my program with head in 3 orientations and it worked well. The next work is to increase orientations. In this step I used 6 orientations, “bl45”, “b0”, “br45”, “ul45”, “u0” and “ur45”. The cross validation result is shown in Figure 5.15. The result of testing in real-time is shown in Table 5.6.



```
correct:332,wrong:28,correctness:0.92.
```

Figure 5.15 Training result of 6 orientations.

Table 5.6 Six orientations real-time result.

	ul45	ur45	u0	bl45	br45	b0
Correct	45	40	36	43	39	46
Incorrect	5	10	14	7	11	4
Accuracy	0.90	0.80	0.72	0.86	0.78	0.92

As we can see, Figure 5.15 (c) shows the cross validation result, the number of incorrect estimation increased, but the correctness is still satisfactory as being 92%. Table 5.6 shows the real-time testing of 6 orientations estimation. Testing people was asked to turn his head to each of the six orientations for 50 times, then the result was recorded in Table 5.6.

5.2.5. Moving People

I used the 4 testing sets to test the program in this part, three of them were consisted of data captured from people sitting in a place that was not the pre-selected position that used to train before, these three positions are: i) closer to the camera which means the depth value of the head changes and the head zone became bigger, ii) head moves upward, which means the head position only changes in the ordinate, iii) head moves to right, which means head only moves in abscissa. The fourth one is people sitting in the pre-selected position at the beginning, then slightly moves forward, upward, etc. This part used the same forest (the same thresholds of all non-leaf nodes and the same information in leaf nodes) as the testing of 6 orientations. Table 5.7, 5.8 and 5.9 shows the real-time testing results of testing people sat closer, higher and rightward than the pre-selected position. The cross validation results are shown in Figure 5.16.

Table 5.7 Real-time testing result of moving closer to the camera.

	ul45	ur45	u0	bl45	br45	b0
Correct	47	32	9	35	29	12
Incorrect	3	18	41	15	21	38
Accuracy	0.94	0.64	0.18	0.70	0.58	0.24

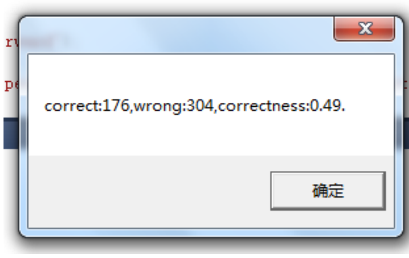
Table 5.8 Real-time testing result of moving upward from the pre-selected position.

	ul45	ur45	U0	bl45	br45	b0
Correct	31	36	30	33	28	34
Incorrect	19	14	20	27	22	16
Accuracy	0.62	0.52	0.60	0.66	0.56	0.68

Table 5.9 Real-time testing result of moving rightward from the pre-selected position.

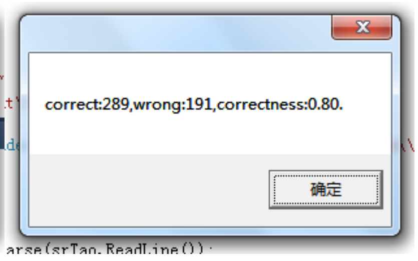
	ul45	ur45	u0	bl45	br45	b0
Correct	44	39	33	39	39	40
Incorrect	6	11	17	11	11	10

Accuracy	0.88	0.78	0.66	0.78	0.78	0.80



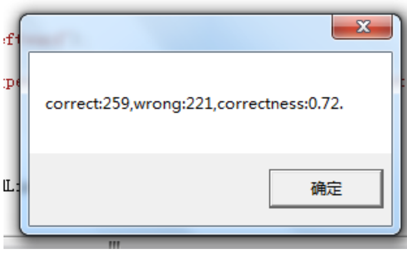
correct:176,wrong:304,correctness:0.49.

(a)



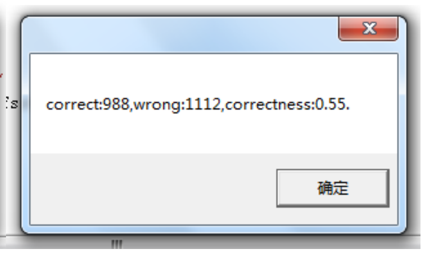
correct:289,wrong:191,correctness:0.80.

(b)



correct:259,wrong:221,correctness:0.72.

(c)



correct:988,wrong:1112,correctness:0.55.

(d)

Figure 5.16 Result of moving people.
Figure 5.16 (a) shows the result of people moves closer to the camera than pre-selected position, 5.16 (b) shows the result of people moves upward, 5.16 (c) shows the result of people moves right, 5.16 (d) shows the result of people keeps moving.

As we can see in, when people sat closer to the camera, higher and rightward, the correctness of cross validation are 49%, 80% and 72% and the correctness of each orientation in real-time testing is shown in Table 5.7 – 5.9. When people sat in the pre-selected position then kept moving the head, the cross validation correctness was 55%.

6. Discussion and Future work

This section will discuss the testing results. In addition, it will review the aims and objectives of the project to judge whether they have been achieved and if so how successfully. Finally, I will list some future work based on the work I did and the discussion of the results.

6.1. Discussion

6.1.1. Static Object Analysis

I used the same method to deal static object as training and testing with people's head. Although the results given by the program would flip occasionally when it was tested in real time, which could be attributed to the low quality of the hardware causing the depth value of the edge of an object keeps changing, the testing results were satisfactory since all of the testing data were estimated correctly. The results proved that using the average of the whole training set in a non-leaf node as the threshold of this node is feasible, also, it showed the feasibility of saving the probability of every orientation in one leaf node. All in all, it proved that the algorithm that this program achieved is sensitive to the changing of range images, so that this program can be used to estimate the orientation of people's head.

6.1.2. People Sitting in Pre-selected Position Analysis

In the last section, the results of head in three orientations and in six orientations came from the same condition where the testing people was sitting in the pre-selected position. When tested in real-time, for example, the head was in "bl45", the result displayed "bl45" in most cases, but some other orientations also occurred occasionally. When tested by cross validation, the results of both of them were satisfactory, meaning that my program is capable to estimate the orientation of people's head in a pre-selected position. However, we can find that the correctness decreased when the number of orientations increased, indicating that limitations exist in the forest I used, and if many orientations were taken into consideration, the correctness of estimation result would be very low. However, this limitation can be solved by following methods: i) improve the way to select threshold or increase the number of threshold in one node. ii) increase the size of training set. iii) increase the depth of each tree in the forest. The last two methods may require more training time, so that how to reduce the training time is another issue that can be listed in the future work.

6.1.3. Moving People

The testing result of this part was shown in Section 5.2.5. The testing of this part was divided into two steps. The first step was to test the head in several new, not pre-selected positions. The main purpose of this step is to test whether the program can estimate the orientation of the head not staying in pre-selected position, the cross validation results of which is shown in Figure 5.16 (a) – (c) and real-time ones in Table 5.7 – 5.9. As we can see, the program estimated well when the head moved upwards and rightwards, but when the head moved towards the camera, the correctness of estimation decreased. Thus, probably the problem is due to change of depth value. When the head moves upward or rightward the depth value changes a little, while it changes dramatically when the head moves towards the camera. As I am concerned, this problem could be solved by adjusting the change of the size of feature zones when the head zone changes its size. Figure 5.16 (d) shows the result of another step, testing the moving head, with the correctness being 55%. When running the program and observing the result in real-time, we can find the program estimated were acceptable when the head moves gently in a certain range (about half metre in the flat parallel to the camera and 20cm perpendicular to the camera). If the head point and the shoulder centre point are added to increase the accuracy, this range would become bigger and the accuracy would increase significantly, but the restriction of the testing people’s pose came along with it.

6.1.4. Evaluation

The first objective of this project is to create a program, which can estimate the orientation of people’s head. This was completed by using the algorithm named random regression forest and therefore this objective can be reckoned as achieved. This program can estimate the orientation of the testing person’s head, the position of which is about 1.5 metres away from the camera and is at the same height with the camera. In addition, the estimation is in real-time and beard, glasses and some other coverings would not affect the testing result too much, and if the number of trees in the forest increases, this affect would even become smaller. In this project, I used 6 orientations to test the program because the training set containing all orientations was not available and the time was limited, and the result was satisfactory.

Another objective of this project is to expand the algorithm. I aimed to expand the algorithm to fit moving people. As the work and the testing result I introduced above, my program can estimate moving head in a certain range, and using the skeleton tracking technology would widen this range and increase the accuracy of estimation, though some limitations. Therefore, this objective can be considered achieved the function with limitations. The problem and method may be helpful for improving this part was introduced in the last section.

6.2. Future Work

6.2.1. Speed and Depth

Although the program achieved estimation of head pose, the algorithm can be improved in several aspects, most effective of which can be decreasing training time. The decreasing of training time means that deeper trees can be afforded, and more thresholds can be used to separate data, so that the performance and efficiency of the program will be improved. There are two ways may be helpful, i) use equipment with high quality, ii) improves the storage method of training set.

6.2.2. Moving People

The performance of estimating moving people still has limitations, and this issue will be taken into further consideration. The program can be adopted in two ways: i) improve the algorithm to find a more effective method to change the size of feature zones when the depth value changes dramatically, ii) take the offset angle caused by head’s movement into consideration. If these two methods work well and the performance of moving head estimation improves significantly, then multiple head estimation can be added by using Skeleton Tracking.

7. Reference

[1] X. Meng, “Robust Head Pose Estimation Algorithm for Embedding Video System”, School of Electronics and Electric Engineering, Shanghai Jiao Tong University, December 2010.

[2] G. Fanelli, J. Gall and L. Van Gool, “Real Time Head Pose Estimation with Random Regression Forests”, In CVPR, August 2011.

[3] M. D. Breitenstein, D. Kuettel, T. Weise, L. Van Gool, and H. Pfister. “Real-time face pose estimation from single range images”. In CVPR, 2008.

[4] J. Illingworth and J. Kittler, A survey of the houghtransform, “Department of Electronics and Electrical Engineering”, University of Surrey, Received 10 August 1987. Revised 19 April 1988.

[5] G. Fanelli, T Weise, J. Gall and L. Van Gool, “Real Time Head Pose Estimation from Consumer Depth Cameras”, In CVPR, August 2011.

[6] L. Breiman. Random forests. “Machine Learning”, 45 (1):5 – 32, 2001.

[7] A. Criminisi, J. Shotton, D. Robertson, and E. Konukoglu. “Regression forests for efficient anatomy detection and localization in ct studies”. In Medical Computer Vision Workshop, 2010.

[8] J. Gall, A. Yao, N. Razavi, L. Van Gool, and V. Lempitsky. “Hough forests for object detection, tracking, and action recognition”. TPAMI, 2011.

[9] C. Huang, X. Ding, and C. Fang. “Head pose estimation based on random forests for multiclass classification”. In ICPR, 2010.

[10] wikipedia. http://en.wikipedia.org/wiki/Kinect

[11] Random Forests for Real Time Head Pose Estimation. [Online; accessed 05 May]. http://www.vision.ee.ethz.ch/~gfanelli/head_pose/head_forest.html#db

[12] http://www.faceshift.com/

[13] Blanz, V., Vetter, T. “A morphable model for the synthesis of 3d faces”. In: SIG-GRAPH '99. pp. 187-194 (1999).

[14] Li, H., Adams, B., Guibas, L.J., Pauly, M. “Robust single-view geometry and mo-tion reconstruction”. ACM Trans. Graph. 28 (5) (2009).

[15] Weise, T., Wismer, T., Leibe, B., Van Gool, L. “In-hand scanning with online loop closure”. In: 3DIM. pp. 1630-1637 (2009).

[16] Besl, P., McKay, N. “A method for registration of 3-d shapes”. IEEE TPAMI 14 (2), 239-256 (1992).

[17] John G. Allen, Richard Y. D. Xu, Jesse S. Jin. “Object Tracking Using CamShift Algorithm and Multiple Quantized Feature Spaces”. University of Sydney, NSW 2006.

[18]http://www.cnblogs.com/yangecnu/archive/2012/04/06/KinectSDK_Skeleton_Tracking_Part1.html.

[19] http://www.360doc.com/content/11/0901/14/4512349_144986127.shtml.

[20] http://msdn.microsoft.com/en-us/library/hh973074.aspx.

[21]http://buildsmartrobots.ning.com/profiles/blogs/building-a-kinect-based-robot-for-under-500-00.

[22] http://file.emgu.com/wiki/index.php/Main_Page.

[23] http://msdn.microsoft.com/en-us/library/ms754130.aspx.