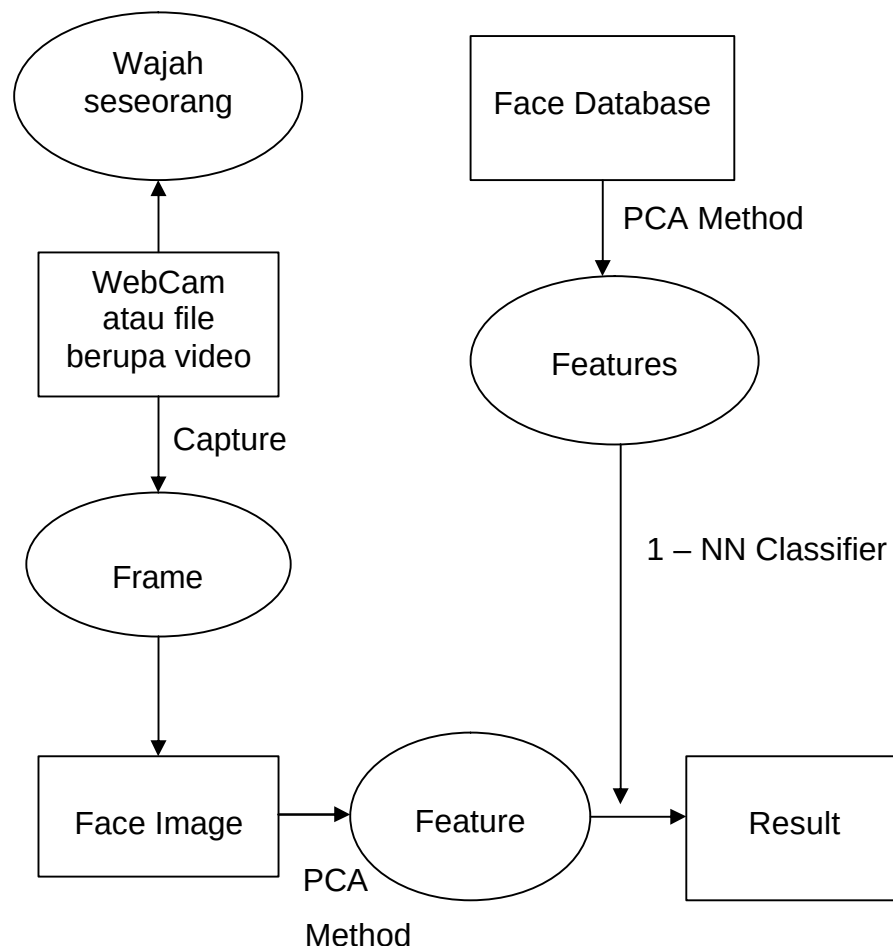


3. DESAIN SYSTEM

Pada bab ini akan dibahas desain dan cara kerja dari system pengenalan wajah berdasarkan metode *Principal Component Analysis* (PCA). Sistem ini dibagi menjadi tiga bagian modul yaitu *Camcap*, *database training*, dan *face detection*. Bagian dari *Camcap* terdiri dari spesifikasi kamera dan mengambil frame dari kamera atau video. Modul *database training* meliputi perhitungan *eigen object* dan *average object* serta perhitungan *feature database*. Sedangkan modul yang terakhir yaitu modul *face detection* meliputi *eigen object* dan *average object* dari frame yang akan diteset lalu kemudian membandingkannya dengan *feature-feature database*. Blok diagram secara sederhana dapat digambarkan sebagai berikut :



Gambar 3.1

Block Diagram Sistem secara sederhana

1. MODUL CAMCAP

Modul ini bertujuan untuk melakukan pengambilan gambar yang berasal dari kamera atau file video. Dalam proses ini terdiri dari dua sub program, yang pertama adalah proses untuk menerima *input* baik dari kamera atau file video dan yang kedua adalah proses untuk melakukan pengambilan frame dari *input* tersebut. Kedua proses tersebut menggunakan fasilitas *library* yang telah disediakan oleh Microsoft® DirectShow® yaitu *directshow*. Penggunaan *library* ini sangat bermanfaat karena proses *input* dapat berlangsung secara *realtime*.

1.1 Proses Input Video dan Kamera

DirectShow dibuat berdasarkan *Component Object Model* (COM). Oleh sebab itu pemograman Aplikasi DirectShow hampir sama dengan pemograman *Component Object Model* (COM). Pembuatan aplikasi DirectShow diawali dengan `CoInitialize(NULL)` pada awal pembuatan dan diakhiri `CoUninitialize()` pada penghancuran program.

Untuk membuat sebuah filter graph, selalu dimulai dengan membuat sebuah *instance* dari *filter graph manager* dan mendapatkan *pointer* ke *IGraphBulder interface* dari *Filter Graph Manager* tersebut.

IGraphBulder interface, seperti namanya, mempunyai *method-method* untuk membangun seluruh *filter graphs*. *Method-method* ini menyediakan tiga pendekatan dasar untuk membangun graph :

Pertama, aplikasi menginstruksikan *Filter Graph Manager* untuk membangun seluruh graph secara otomatis. Pendekatan pertama adalah untuk scenario ketika hanya ingin memainkan sebuah file dari format yang dikenali seperti AVI, MPEG, WAV, MP3 dan seterusnya. Teknik ini digunakan untuk memainkan sebuah file. Dalam DirectShow, istilah *render* lebih sering digunakan untuk memainkan (*play*) untuk menjelaskan proses menampilkan video data pada monitor atau audio

data melalui *speaker system*. Untuk memerintahkan *filter Graph Manager* secara otomatis membuat filter graph yang dapat me-render sebuah file tertentu, sebuah aplikasi menggunakan `IGraphBuilder::RenderFile` method.

Kedua, Aplikasi dari *Filter Graph Maneger* membagi pekerjaan untuk membangun *Filter Graph Maneger*. Ketika filter graph yang diperlukan untuk melakukan sesuatu yang berbeda daripada sekedar me-render file, maka aplikasi harus melakukan paling tidak beberapa tugas pembangunan filter. Sebagai contoh, jika hendak membangun sebuah filter graph yang mengkonversi AVI ke MPEG file, *Filter Graph Manager* masih dapat membuat sebuah AVI *playback graph* dengan memanggil `IGraphBuilder::RenderFile`, tetapi harus memodifikasi graph ke *output* sebuah MPEG file daripada monitor dan speaker. Ini melibatkan men-*disconnect* dan menghapus video dan audio *renderer filters* (dengan menggunakan `IGraphBuilder::Disconnect` dan `IGraphBuilder::RemoveFilter`) dan menggantikan itu dengan MPEG video dan *audio compressor filters*, MPEG *Multiplexer filter*, dan sebuah *file writer filter*.

Terkadang juga mungkin diperlukan untuk menambah sebuah *filter intermediate* untuk mengubah tipe media kesalah satu bentuk tipe media dimana MPEG *compressor* dapat menerima. Aplikasi dapat baik secara langsung menambah *filter intermediate* ini atau menggunakan `IGraphBuilder::Connect` method. Ketika method ini dipanggil, *Filter Graph Manager* akan mengkoneksi filter secara langsung, dan jika mereka tidak dapat setuju dengan sebuah tipe media, maka *Filter Graph Manager* akan mencari *intermediate filter*. Ketika *Filter Graph Manager* mencari sebuah filter yang dapat menangani sebuah tipe media, *Filter Graph manager* akan pertama kali mencoba filter yang telah ditambahkan ke graph. Untuk mencegah *Filter Graph Manager*

dari mencoba menambahkan *intermediate filter*, gunakan `IGraphBuilder::ConnectDirect` method.

Ketiga, Aplikasi membangun seluruh graph dengan sendirinya yaitu dengan cara menambah dan menkoneksi masing masing filter secara individual. Dalam beberapa scenario, aplikasi mungkin membutuhkan untuk membangun seluruh graph dengan menambahkan dan menghubungkan setiap filter secara individual. Dalam kasus ini, filter yang akan ditambahkan telah diketahui secara khusus, jadi tidak diperlukan untuk menggunakan *Filter Mapper Object* untuk menemukan filter yang bersangkutan. Hanya dengan menambahkan setiap filter ke graph dengan menggunakan `IGraphBuilder::AddFilter` method dan menkoneksi mereka dengan menggunakan baik `Connect` maupun `ConnectDirect`.

Untuk proses ini, pertama kali adalah membuat beberapa *variable* dengan tipe data object yang disediakan oleh *library* directshow antara lain :

- `IGraphBuilder*` `m_CamBuilder1`

Berfungsi sebagai *interface* dari *Filter Graph manager*

- `IMediaControl*` `m_CamControl1`

Berfungsi untuk menangani control terhadap *stream* yang sedang berlangsung.

- `IVideoWindow*` `m_CamView1`

Berfungsi untuk membangun *interface output video window*

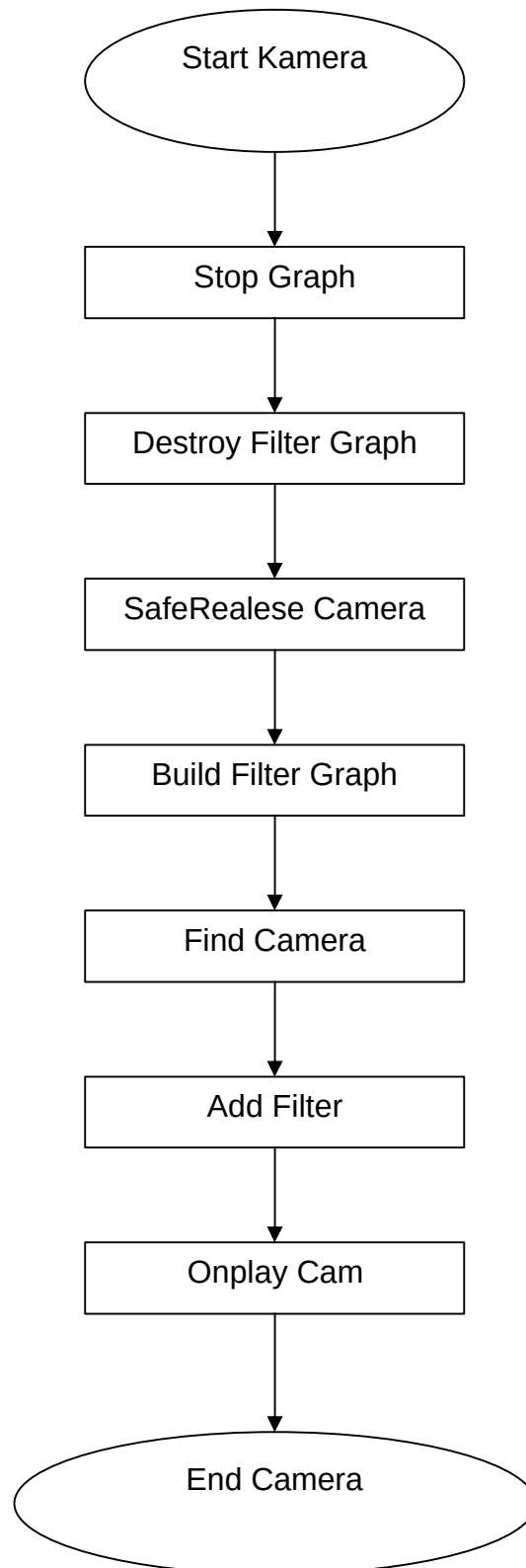
- `IBaseFilter*` `m_Cam1, m_CamTrans1`

Berfungsi untuk pengontrol filter dan pada aplikasi berfungsi untuk mengspesifikasikan *pin* dan *query* informasi filter.

- **IFilterGraph*** m_CamGraph1

Berfungsi untuk membangun filter. Pada aplikasi digunakan untuk menambahkan filter pada graph, menghubungkan dan memutuskan hubungan dengan filter, menghapus filter dan melakukan beberapa operasi dasar lainnya.

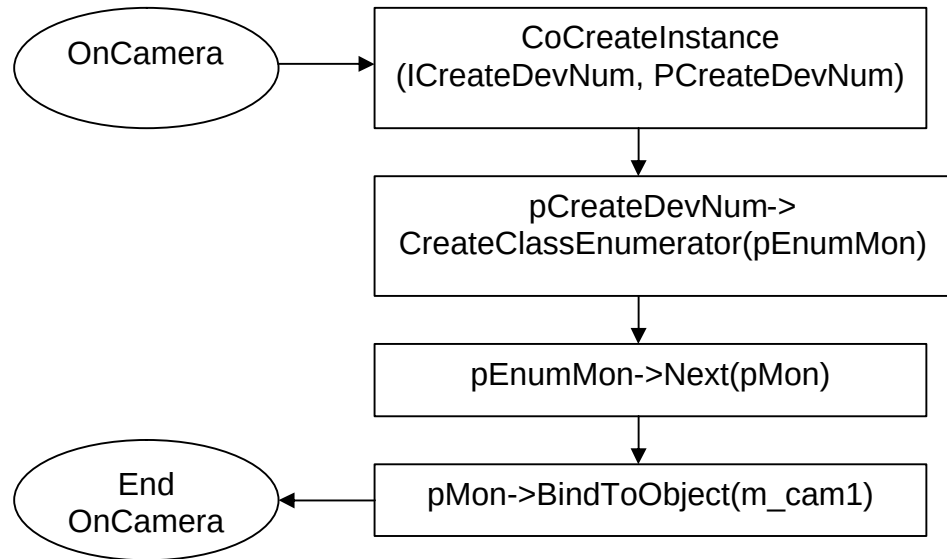
Program aplikasi DirectShow yang dibuat dalam tugas akhir ini, berdasarkan sumber *inputnya* (*Source filter*) secara garis besar dapat dibagi menjadi dua bagian, yaitu *input* dari file berupa AVI File , dan *input* dari kamera. Bagian *input* AVI File dibangun dengan pendekatan pertama, sedangkan bagian *input* kamera dibangun dengan menggunakan pendekatan ketiga. Kedua bagian ini secara garis besar memiliki prosedur utama yang hamper sama. Berikut diagram alir dari bagian *input* :



Gambar 3.2
Diagram Alir bagian Input

Secara garis besar, baik bagian *input* kamera maupun *input* AVI File menjalani prosedur-prosedur sebagai berikut : StopGraph, DestroyFilterGraph, CreateFilterGraph, dan StartGraph. StopGraph berfungsi untuk menstop aliran dengan menggunakan IMediaControl, kemudian mematikan video *output* melalui IVideoWindow *interface*. DestroyFilterGraph melakukan *SafeRelease* terhadap variable yang dipakai. SafeRelease berarti jika *pointer* tidak sama dengan NULL maka dilakukan *release* *pointer* yang bersangkutan dan *pointer* tersebut di-set menjadi NULL. CreateFilterGraph berfungsi membangun *graph* yang digunakan pada bagian input AVI File. StartGraph berfungsi untuk memulai *graph*. Perbedaan antara bagian *input* kamera dan input AVI File adalah pada void CCamCapDlg::FindCamera. Proseduer ini berfungsi untuk mendapatkan *source filter* apakah dari kamera atau dari file video. Diagram alir dari bagain kamera dapat dilihat pada gambar 3.3.

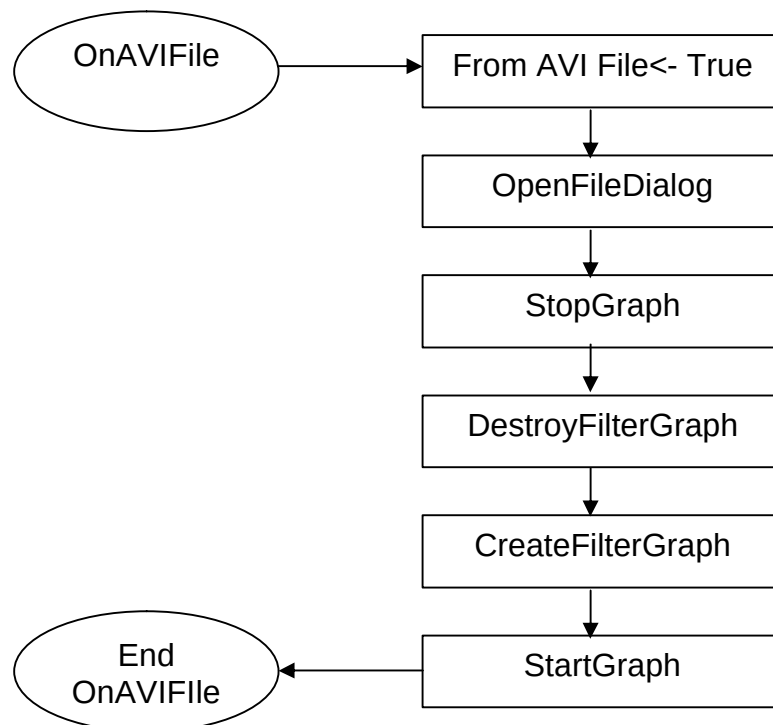
Dimulai dengan membuat *instance* dengan tipe ICreateDevNum, yang berfungsi untuk men-*enumerate device* yang terdapat di computer saat ini. Setelah itu ICreateDevNum, membuat ClassEnumerator dengan nama pCreateDevNum. pCreateDevNum akan mengambil *device* pertama dari daftar *device* dan kemudian *membind object* dengan variable m_cam1 yang bertipe IBaseFilter.



Gambar 3.3

Diagram Alir bagian Input dari Kamera

Diagram Alir dari bagian AVI File secara garis besar mirip dengan diagram alir bagian kamera. Berikut Diagram Alir dari bagian AVI File :



Gambar 3.4

Diagram Alir bagian input dari AVI File

Seperti yang telah dijelaskan diatas, secara garis besar prosedur yang dijalankan oleh bagian *input* AVI File sama dengan bagian *input* kamera. Perbedaan yang mencolok adalah procedure `OpenFileDialog` yang tidak dimiliki oleh bagian kamera. Bagian ini berfungsi untuk membuka `OpenFileDialogBox` dan jika ada file yang dipilih maka `OpenFileDialog` akan menghasilkan nama file yang telah dipilih. Setelah itu, procedure yang dilakukan sama dengan prosedur yang ada pada bagian input kamera.

1.2 Proses pengambilan frame

Aliran *frame* dari aplikasi `DirectShow` ini nantinya yang akan digunakan dalam modul *face detection* untuk mengenail wajah seseorang. Adapun parameternya `void CCamCapDlg::CallBackCam1 (IplImage* frame)`.

Frame yang dihasilkan merupakan gambar gambar yang dihasilkan dari video sehingga kita akan mudah untuk mengambil informasi melalui gambar daripada video yang nantinya akan kita proses lebih lanjut.

1.3 Proses Pemberian Noise

Inputan yang dihasilkan berupa frame dapat kita tambahkan noise untuk menguji ketepatan pencarian wajah yang cocok. Noise yang dipakai merupakan noise titik-titik dengan jumlah yang dirandom. Untuk menambahkan noise ini digunakan perintah :

```
IplNoiseParam p;
iplNoiseUniformInit(&p, rand(), 0, 100);
iplNoiseImage(frame, &p);
```

Gambar 3.5

Perintah Pemberian Noise kepada Frame

2. MODUL DATABASE TRAINING

Bagian ini adalah proses untuk melakukan proses *training* pada *database*. Dimana *database* ini berupa sekumpulan gambar wajah wajah *alphanumeric* yang terdiri dari beberapa macam variasi untuk masing masing wajah. Disini saya memiliki 20 orang dan tiap orangnya memiliki kurang lebih 4-9 gambar *database*. Database yang digunakan total ± 109 gambar. Dalam modul *training* ini dihasilkan *output* berupa *Eigen Object* dan *Average Object* serta bobot masing masing wajah yang terdapat dalam *database*. Hasil bobot ini disimpan kedalam suatu *list array* dan kedalam text file. Variasi pembagian masing masing wajah adalah sebagai berikut :

Table 3.1

Jumlah Variasi Wajah pada Database

Wajah	Jumlah Variasi	Contoh Variasi
Adi	5	
Adrian	5	
Badam	5	
Charles	7	
Danny	5	

Hadi	6	
Immanuel	5	
Jenny	5	
Kosdi	6	
Lewi	6	
Liki	5	
Nora	5	
Raymond	10	
Rono	5	

Stefanus	6	
Teguh	5	
Tera	5	
Tjandra	5	
Vincent	4	
Yuniawan	4	

Proses *training* ini dibagi menjadi dua tahapan proses yaitu :

2.1 Perhitungan Eigen Object dan Average Object.

Pada tahap ini program akan melakukan suatu perhitungan untuk mendapatkan beberapa fitur dan gambar rata-rata dari *database*. Fitur yang dimaksud adalah *Eigen Object* itu sendiri dan gambar rata-rata (*mean*) adalah *Average Object* yang digunakan untuk proses pengenalan (*recognition*). Perhitungan yang digunakan adalah menggunakan fungsi

`cvCalcEigenObjects(int nObjects, void* input, void* output, int ioFloags, int ioBufSize, void* UserData, cvTermCriteria* calcLimit, IplImage* avg, float* eigVals)`. Dalam penggunaan fungsi ini (gambar 3.7) terdapat syarat yang harus dipenuhi, yaitu jumlah feature yang akan dihasilkan harus kurang dari jumlah *database* yang akan *ditrainingkan*. Sebagai contoh apabila kita memiliki 20 gambar *database*, maka fitur yang dapat dihasilkan antara 1 hingga 19 buah.

```
//=====
//INITIALIZE EIGEN OBJECT & FEATURE
//=====
m_nEigs=40;
m_nFeature=40;
m_eig = new IplImage* [m_nEigs]; //depth = IPL_DEPTH_32F
IplImage **img_face = new IplImage* [m_nItem]; // 1 channel
images
IplImage *temp1, *temp2;
CvSize size = cvSize(320, 240); //size of image training
//=====
// prepare eigen image
for (i=0; i<m_nEigs; i++)
    m_eig[i] = cvCreateImage(size, IPL_DEPTH_32F, 1);

// prepare average image
m_avg = cvCreateImage(size, IPL_DEPTH_32F, 1);

// prepare PCA criteria -> the number of principal
components
CvTermCriteria criteria;
criteria.type = CV_TERMCRIT_ITER;
criteria.maxIter = m_nEigs; // set jumlah eigen image
criteria.epsilon = 0.1f;
```

Gambar 3.6

Inisialisasi Awal Penggunaan Fungsi `cvCalcEigenObjects()`

```

//=====
// CALCULATE EIGEN IMAGE AND AVERAGE IMAGE FROM DATABASE
//=====
cvCalcEigenObjects( m_nItem, //number of input
                    img_face, // input image
                    m_eig, // output eign[] image
                    = principal components
                    CV_EIGOBJ_NO_CALLBACK,
                    0,
                    NULL,
                    &criteria,
                    m_avg, //output average image
                    NULL); //output eigen value
//=====

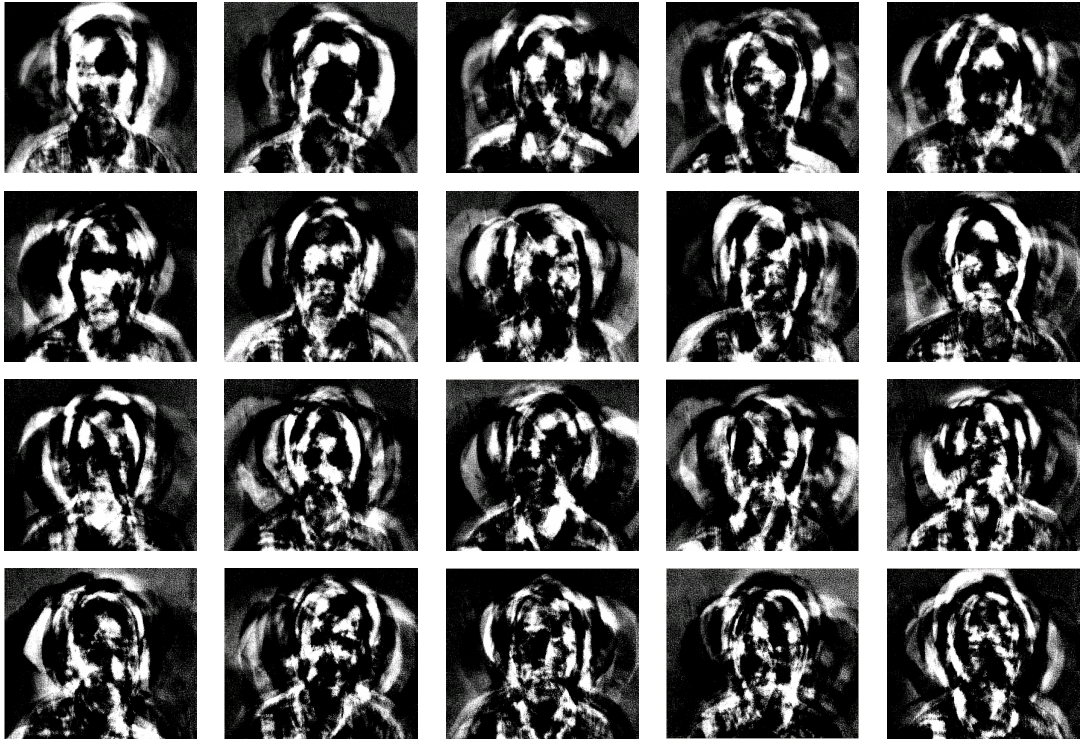
```

Gambar 3.7

Perhitungan Eigen Object dan Averange Object

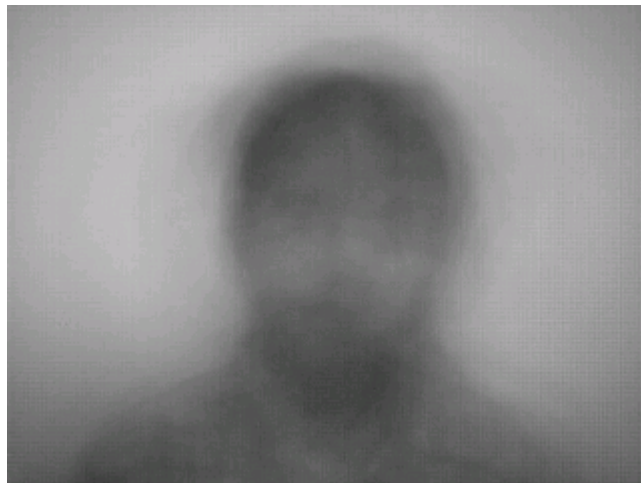
Pada proses pengenalan ini fitur yang akan diambil sebanyak 40 buah atau sekitar 36,66% dari jumlah *database*. Pengambilan fitur 36,66% dari jumlah *database* ini telah memberikan pengenalan yang cukup baik.





Gambar 3.8

40 *Eigen Object* Pertama Dari hasil `cvCalcEigenObjects()`



Gambar 3.9

Average Object dari hasil `cvCalcEigenObjects()`

2.2 Perhitungan Bobot Fiture

Setelah perhitungan *Eigen Object* dan *Average Object* didapatkan, selanjutnya melakukan perhitungan bobot fitur (gambar 3.10) untuk masing masing yang ada pada *database*. Bobot inilah yang akan digunakan sebagai fitur untuk proses pengenalan (*recognition*). Untuk perhitungan bobot ini digunakan :

`cvCalcDecompCoeff(IplImage* obj, IplImage* eigObj, IplImage* avg).`

```
//=====
// CALCULATE FEATURE PER IMAGE DATABASE & SAVE TO FILE
//=====
FILE* file;
file = fopen("../Result\\database.txt", "w");
for (i=0; i<m_nItem; i++)
{
    double weight[100]={0};
    m_ListFiles.GetText(i, filename);
    iplColorToGray(cvvLoadImage(filename), temp1);

    fprintf( file, "%s ",&filename);
    for (j=0; j<m_nFeature; j++)
    {
        weight[j]= cvCalcDecompCoeff(temp1, m_eig[j],
        m_avg);
        m_weight_database[i][j]=weight[j];
        fprintf( file, "%d ",weight[j]);
    }
    fprintf( file, "\n\n");
}
fclose(file);
//=====
```

Gambar 3.10

Perhitungan bobot fitur dengan fungsi `cvCalcDecompCoeff`

Program ini akan melakukan iterasi sebanyak jumlah gambar pada *database* dalam melakukan perhitungan koefisien bobot ini. Dimana setiap gambar akan dihitung koefisien bobotnya sebanyak 40 buah.

3. MODUL FACE DETECTION

Setelah kita mendapatkan fitur-fitur dari *database* dan juga kita bisa mendapatkan frame dari *input* yang akan kita deteksi atau test maka tahap selanjutnya adalah tahap pengenalan (*recognition*). Pada bagian ini ada dua bagian yang penting yaitu :

3.1 Image Pre-Processing

Pada tahap ini gambar wajah yang kita dapatkan dari frame tersebut kita rubah menjadi grayscale:

```
m_face = cvCreateImage(cvSize(320,240), IPL_DEPTH_8U, 1);
cvvConvertImage(temp1, m_face, 0);
cvvSaveImage(filename1, temp1);
```

Gambar 3.11

Mengubah dari 3 channel Image menjadi 1 channel

Karena *m_face* merupakan sebuah *IplImage* yang terdiri dari 1 channel maka frame yang kita dapatkan tadi akan berubah dari 3 channel menjadi 1 channel, lalu kemudian kita simpan kedalam file.

3.2 Classifier

Pengenalan wajah ini menggunakan metode *k-Nearest Neighbor Classifier*. Ide dasar dari metode ini adalah melakukan perhitungan bobot pada setiap wajah dan kemudian dicari pada *database*, mana yang memiliki jarak yang paling mendekati dari bobot yang baru dihitung tersebut.

3.2.1 Perhitungan bobot dan jarak masing-masing wajah

Wajah yang kita dapatkan dari frame, dihitung nilai bobotnya (Gambar 3.12) sebanyak 40 fitur dengan menggunakan `cvCalcDecomCoeff(IplImage* obj, IplImage* eigObj, IplImage* avg)`. Fungsi ini

mengembalikan suatu nilai dekomposisi dari hasil perhitungan *eigen object* dan *average object* yang didapat dari modul *database training*. Sehingga dari hasil perhitungan ini maka setiap frame akan memiliki bobot sebanyak 60 nilai. Dan nilai ini nantinya yang akan dipakai untuk mencari jarak terpendek dari *database* yang ada.

```
//=====
//CALCULATE NEW WEIGHT
//=====
for (i=0; i<m_nFeature; i++)
    new_weight[i]= cvCalcDecompCoeff(m_face,
    m_eig[i], m_avg);
//=====
```

Gambar 3.12

Perhitungan bobot wajah dari frame

3.2.2 Mencari jarak terpendek

Setelah 40 fitur didapat maka untuk menghitung jarak terpendek adalah dengan menggunakan persamaan *Euclidean Distance* (Gambar 3.13). Masing masing jarak ini dilakukan pencarian pada *database* mana yang memiliki *minimum distance* (Gambar 3.14) dari hasil perhitungan tersebut. Nilai yang paling minimum adalah merupakan wajah yang memiliki kemiripan dengan data gambar yang bersangkutan.

```
//=====
//CALCULATE DISTANCE
//=====
for (int row=0; row<m_nItem; row++)
{
    total=0;
    for (int col=0; col<m_nFeature; col++)
    {
        temp=(new_weight[col]-
            m_weight_database[row][col]) *
            (new_weight[col]-
            m_weight_database[row][col]);
        total += temp;
    }
    distance[row]=sqrt(total);
}
//=====
```

Gambar 3.13

Menghitung jarak setiap fitur dengan persamaan *Euclidean Distance*

```
//=====
//COMPARE AND FIND MINIMUM DISTANCE
//=====
double min=distance[0];
pos=0;
for (i=1; i<m_nItem; i++)
{
    if (distance[i] < min)
    {
        min=distance[i];
        pos=i;
    }
}
//=====
```

Gambar 3.14

Proses mencari jarak terdekat