연 • 재 • 순 • 서

1회 2004.12 | 쉽고 빠른 OpenCV 영상처리의 첫걸음

2회 2004.1 | OpenCV의 고급 영상처리 알고리즘의 활용

연 • 재 • 가 • 이 • 드

운영체제 | 윈도우 98/ME/NT/2000/XP

개발도구 | 다이렉트쇼 프트 비주얼 스튜디오 6

기초지식 | C, 비주얼 스튜디오, 영상처리에 대한 기초 개념

황미란 | mirang@vision.hanyang.ac.kr

이혼진 | hjlee@vision.hanyang.ac.kr

한양대학교 영상공학연구실의 동기로서 MPEG-7, 생체인식, DRM분야에서 임베디드 시스템 및 핸드폰 카메라에서의 영상처리로 연구 분야를 확장하고 있다. 한양대 전자전기컴퓨터 공학과 학술모임 바라미의 정회원이기도하다.

OpenCV 라이브러리를 이용한 영상처리 2

OpenCV 고급 영상처리 알고리즘의 활용

OpenCV 라이브러리의 가장 큰 매력은 지속적인 업데이트 및 온라인 그룹을 통한 정보 공유, 그리고 무엇보다도 미리 구현된 고급 영상처리 알고리즘을 소스 레벨까지 다룰 수 있다는 점이다. 그러나 OpenCV 라이브러리가 영상처리 알고리즘을 구현했기 때문에 100% 활용을 위해서는 기본 이론에 대한 이해가 필수적이다. 이번 호에서는 OpenCV에서 제공하는 고급 알고리즘에 대한 이해를 통해 OpenCV와 좀 더 친숙해지도록 하자.

지난 호에서는 OpenCV 라이브러리와 그 기초적인 활용 방법에 대해 알아보았다. 이번 호에는 한 걸음 더 나아가 템플릿 매칭, 옵티컬 플로우, 얼굴 검출 등 좀 더 난이도 있는 내용을 다루도록 하겠다. 각각의 알고리즘이 깊이가 있는 내용이기 때문에 영상처리를 배우지 않은 사람이 제대로 이해하기 위해서는 상대적으로 어려움이 있을 수 있으므로 영상처리 관련 인터넷 사이트나 관련 서적, 논문 등을 참고하기 바란다.

영상의 반전, 색상계 변환

〈화면 1〉과 같은 영상을 인터넷에서 종종 보아 왔을 것이다. 이것은 반전된 영상으로 영상의 최대값에서 현재 픽셀의 밝기 값을 빼주는 것으로 쉽게 구현할 수 있다. 이번 예제를 통해 OpenCV에서 그레이, 컬러 영상의 픽셀을 어떻게 처리하는지 알아보자.

IplImage의 픽셀 값을 직접 접근하기 위해서 imageData라는 내부 변수를 사용한다. imageData는 1차 배열이지만 영상은 2차원이기 때문에 2차원 배열처럼 다루는 것이 편하므로 〈리스트 1〉에서는 2차원 배열처럼 다루고 있다. 단, IplImage는 생성될 때 비트맵 영상처럼 영상의 너비가 4바이트의 배수가 되도록 너비를 정렬하기 때문에 src->width 대신에 src->widthStep을 사용해야함을 유의하자. 즉, src->width는 단순히 영상의 너비를 가리키며 src->widthStep는 너비의 실제 바이트 수를 가리킨다. 그레이 영상은 픽셀당 1바이트씩 할당되지만, 컬러 영상은 BGR 순서로 픽셀당 3바이트가 할당된다. 즉, 컬러 영상 배열은 (B0,G0,R0), (B1,G1,R1), ..., (Bn,Gn,Rn) 순서로 저장되어 있으며 (B0,G0,R0)는 첫 번째 픽셀의 파란색, 녹색, 빨간색을 나타낸다. 예를 들어 두 번째 픽셀의 녹색에 접근하기 위해서는 src->imageData[4]를 사용하면 된다.

또한, cvCvtColor() 함수를 통해 RGB 색상계를 컬러 영상 분석할 때 주로 사용하는 HSV 혹은 YCbCr 색상계로도 쉽게 변환할 수 있다(참고자료 ❶).

<화면 1> 영상의 반전

〈리스트 1〉 영상의 반전 및 색상계 변환

```
void COpenCVTestDlg::OnBtnInverse()
{
   IplImage* src=cvLoadImage("dog.jpg", -1);
   IplImage* dst=cvCreateImage( cvGetSize(src), IPL_DEPTH_8U, src->nChannels);
    IplImage*  YCbCr=cvCreateImage(  cvGetSize(src),  IPL_DEPTH_8U,  src->nChannels);
   IplImage* Cb =cvCreateImage( cvGetSize(src), IPL_DEPTH_8U, 1);

   // 반전 영상 구하기: 최대 밝기값(255) - 영상의 밝기값
   for (int i=0; i<src->height; i++){
   for (int j=0; j<src->widthStep; j++){
      dst->imageData[(i*dst->widthStep) + j]
         = 255-src->imageData[(i*src->widthStep) + j];
   }}

   // BGR 색상 좌표를 YCrCb 영상으로 변환
   cvCvtColor(src, YCbCr, CV_BGR2YCrCb);

   // Cb만 추출, Y, Cr, Cb를 따로 보면, 그레이 영상과 큰 차이가 없다.
   for (i=0; i<Cb->height; i++){
   for (int j=0; j<Cb->widthStep; j++){
      Cb->imageData[(i*Cb->widthStep) + j]
       = YCbCr->imageData[(i*YCbCr->widthStep) + (j+2)*3]; //+0:Y, +1:Cr, +2:Cb
   }}
// cvNamedWindow(),cvShowImage()를 이용한 결과 화면 출력 및 IplImage의 리소스 해제
   cvNamedWindow("src", CV_WINDOW_AUTOSIZE);   cvShowImage("src", src);
   cvNamedWindow("dest", CV_WINDOW_AUTOSIZE);  cvShowImage("dest", dst);
   cvNamedWindow("Cb", CV_WINDOW_AUTOSIZE);     cvShowImage("Cb", Cb);
   cvReleaseImage(&src);   cvReleaseImage(&dst);   cvReleaseImage(&Cb);
}
```

이산 푸리에 변환, DFT

신호처리에서 절대로 빠질 수 없는 것 중에 하나가 바로 푸리에 변환(Fourier Transform)이다(참고자료 ❷). 푸리에 변환은 신호를 주파수 영역에서 분석하기 위해서 사용된다. 푸리에 변환의 기본 개념은 하나의 신호는 여러 개의 sin 신호와 cos 신호의 합으로 표현할 수 있다는 것이고, 이 sin 신호와 cos 신호의 주파수 성분이 어떻게 구성되어 있는지를 파악하여 신호를 분석한다. 다음은 2차원 푸리에 변환과 그의 역변환을 나타내고 있으며 sin과 cos을 오일러 공식을 통해 exp로 표현하고 있다. 푸리에 변환을 통해 우리는 신호 해석을 쉽게 할 수 있다.

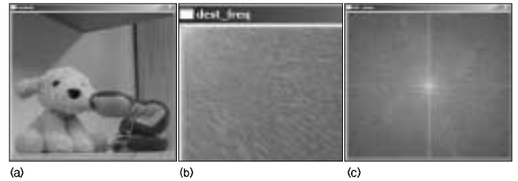
$$F(u,v)=\int_{-\infty}^{\infty}\int_{-\infty}^{\infty} f(x,y)\exp[-j2\pi(ux+vy)]dxdy$$

$$f(x,y)=\int_{-\infty}^{\infty}\int_{-\infty}^{\infty} F(u,v)\exp[j2\pi(ux+vy)]dudv$$

OpenCV에서는 푸리에 변환을 지원하기 위해 cvDFT()라는 함수를 제공하고 있다. DFT는 이산 푸리에 변환(Discrete Fourier Transform)으로서 이론적인 푸리에 변환을 실제 컴퓨터에서 실행 가능하게 만든 알고리즘이다. 이 DFT를 훨씬 빠르게 처리하기 위한 것이 바로 고속 푸리에 변환(FFT, Fast Fourier Transform)이며 길버트 스트랭(Gilbert Strang)은 FFT를 가리켜, "우리 세대의 가장 중요한 알고리즘"이라고 말하기도 했다(참고자료 ❸).

cvDFT() 함수는 1차원, 2차원 신호를 모두 변환, 역변환할 수 있다. 〈화면 2〉에는 임의의 영상에 대해 cvDFT()한 결과를 보여주고 있다. 보통 DFT 변환된 결과를 화면에 보여줄 때에는 보통 주파수 신호의 크기(magnitude)를 출력한다. 또한, 원래 DFT의 결과인 〈화면 2〉의 (b)보다는 (c)와 같은 영상을 많이 보게 된다. DFT의 결과는 주파수 영역에서 주기적이라는 성질이 있기 때문에 (b)를 반 주기만큼을

<화면 2> 이산 푸리에 변환(cvDFT) 결과



(a) (b) (c)

(a) 원본 영상 (b) 주파수 신호의 크기를 출력한 결과 (c) (b)를 반주기 이동한 영상

이동시킨 (c)는 서로 동일하다. 주파수 영역에서의 밝은 부분과 어두운 부분의 의미는 밝은 부분일수록 그 주파수에 해당하는 sin과 cos 성분이 영상에 많이 포함되어 있다는 것을 의미한다. 자세한 것은 푸리에 변환에 관한 서적을 참조하자.

```
// 이산 푸리에 변환(DFT)을 실행한다.
void cvDFT(const CvArr* src,      // 원본 영상
           CvArr* dst,            // 푸리에 변환 결과 주파수 값
           int flags );           // DFT 또는 Inverse DFT 결정
```

cvDFT() 함수에서 src와 dst는 동일한 자료형이어야 하고 복소수 연산을 하기 때문에 자료형을 실수 형태로 생성하는 것이 좋다. flag는 DFT를 할 것인지(CV_DXT_FORWARD), Inverse DFT를 할지(CV_DXT_INV_SCALE)를 결정한다. 여기서 Inverse DFT는 주파수 변환된 결과를 원래 영상으로 역변환한다. 〈리스트 2〉에서는 실수 연산을 위해 CvMat 자료형으로 채널이 2인 64비트 실수형 데이터를 생성하여 DFT한다. 채널이 2인 변수를 만드는 이유는 푸리에 변환의 결과인 주파수 영역의 값이 실수, 허수 형태로 저장되기 때문이다. 실수를 R, 허수를 I라고 할 때, 채널이 2개인 변수는 앞의 컬러 영상과 유사한 형태로 (R0, I0), (R1, I1), (R2, I2), …, (Rn, In) 형태로 배열이 생성된다. 그러므로 freq->data.db[0], freq->data.db[1]은 src->imageData[0]를 주파수 영역으로 변환했을 때 나오는 실수, 허수 값이며, freq->data.db[2], freq->data.db[3]은 src->imageData[1]에 해당한다.

〈리스트 2〉 이산 푸리에 변환(DFT) ↔ Inverse DFT

```
void COpenCVTestDlg::OnBtnDft()
{
   IplImage *src=cvLoadImage("dog.jpg", 0);
    IplImage *dst_inverse=cvCreateImage(cvGetSize(src), IPL_DEPTH_8U, src->nChannels);
    IplImage *dst_freq =cvCreateImage(cvGetSize(src), IPL_DEPTH_8U, src->nChannels);
    IplImage *dst_swap =cvCreateImage(cvGetSize(src), IPL_DEPTH_8U, src->nChannels);

   // spatial:입력 영상, freq: 입력 영상을 주파수 영역으로 변환한 결과
   CvMat * spatial = cvCreateMat(src->height, src->widthStep, CV_64FC2);
   CvMat * freq         = cvCreateMat(src->height, src->widthStep, CV_64FC2);

   // DFT  ------------------------------------------------------------
   for (i=0; i<src->imageSize; i++) {
  spatial->data.db[i*2] = (double)(BYTE)src->imageData[i];
  spatial->data.db[i*2+1] = 0; // 복소수 연산을 하기 때문에 허수 영역도 함께 초기화함
   }

   cvDFT(spatial, freq, CV_DXT_FORWARD); // DFT 수행

   // 주파수 크기의 변화가 크기 때문에 log를 통해 화면의 그레이 레벨 범위(0~255)로 변환하여 출력
   double tmp, max = -10000000, min = 1000000000;
   for (i=0; i<src->imageSize; i++) {
         tmp  =log10(1+sqrt(SQUARE(freq->data.db[i*2]) + SQUARE(freq->data.db[i*2+1])));
      if(tmp <min) min = tmp;
      if(tmp >max) max = tmp;
   }
   for (i=0; i<src->imageSize; i++) {
      dst_freq->imageData[i] = (BYTE)(256/(max-min)*
          log10(1+sqrt(SQUARE(freq->data.db[i*2]) + SQUARE(freq->data.db[i*2+1])));
   }

   // 화면 출력을 위해서 주파수 영역의 결과를 반 주기 만큼 이동하여 출력
   FreqShift(dst_freq, dst_swap);

  // inverse-DFT ------------------------------------------------------------
   cvDFT(freq, spatial, CV_DXT_INVERSE_SCALE);

   for (i=0; i<src->imageSize; i++) { // 역변환 결과를 IplImage에 저장
      dst_inverse->imageData[i] = (char)spatial->data.db[i*2];
   }

 // cvNamedWindow(), cvShowImage()를 이용한 결과 화면 출력 및 IplImage의 리소스 해제

}

// 주파수 영역에서 반 주기만큼 이동한 효과를 줌:
// 좌측상단 ↔ 우측 하단, 우측상단 ↔ 좌측상단 교환
void FreqShift(IplImage* src, IplImage* dst)
{
   int halfHeight=src->height/2, halfWidth=src->width/2;
   BYTE* imagedata = (BYTE*)src->imageData;
   BYTE* dstImage  = (BYTE*)dst->imageData;

   for(int v=0; v<halfHeight; v++) { // 행 연산
   for(int u=0; u<=halfWidth; u++) { // 열 연산
          dstImage[v*src->widthStep+u]=  imagedata[(v+halfHeight)*src->widthStep+u+halfWidth];
       dstImage[(v+halfHeight)*src->widthStep+u+halfWidth]=imagedata[v*src->widthStep+u];
        dstImage[(v+halfHeight)*src->widthStep+u]  =  imagedata[v*src->widthStep+u+halfWidth];
       dstImage[v*src->widthStep+u+halfWidth] = imagedata[(v+halfHeight)*src->widthStep+u];
   }}
}
```


〈화면 3〉 DCT 결과(a) 원본 영상 (b) DCT의 주파수 신호의 크기를 출력 결과

〈리스트 3〉 DCT ↔ Inverse DCT 결과

```
void COpenCVTestDlg::OnBtnDct()
{
   IplImage *src=cvLoadImage("tulip.jpg", 0);
    IplImage *dst_inverse =cvCreateImage(cvGetSize(src), IPL_DEPTH_8U, src->nChannels);
    IplImage *dst_freq =cvCreateImage(cvGetSize(src), IPL_DEPTH_8U, src->nChannels);
   CvMat * spatial= cvCreateMat(src->width, src->height, CV_64FC1);
   CvMat * freq   = cvCreateMat(src->width, src->height, CV_64FC1);

   // DCT  ------------------------------------------------------
   for (int i=0; i<src->imageSize; i++) {
      spatial->data.db[i] = (double)(BYTE)src->imageData[i];
   }

   cvDCT(spatial, freq, CV_DXT_FORWARD); // DCT 수행

   // inverse-DCT ------------------------------------------------------
   cvDCT(freq, spatial, CV_DXT_INVERSE_SCALE);

   for (i=0; i<src->imageSize; i++) {
      dst_inverse->imageData[i] = spatial->data.db[i];
   }

// cvNamedWindow(),cvShowImage()를 이용한 결과 화면 출력 및 IplImage의 리소스 해제
}
```

CvMat형 변수를 생성할 때는 cvCreateMat() 함수를 사용하며, 생성될 자료형의 종류는 CV_64FC2의 값으로 표시한다. 64는 64비트, F는 실수형, C2는 채널이 2임을 나타낸다. CvMat은 행렬을 다루는 자료형으로 영상과 같이 2차원 배열을 사용하면 편리하며 행렬 연산은 물론 역행렬, SVD, 고유 값을 계산하는 함수도 지원한다. CvMat형 데이터의 접근은 spatial->data를 이용하는데, 64비트 자료형이기 때문에 spatial->data.db[i] 형태로 행렬의 값에 접근한다. 만약에 32비트 실수형이라면 spatial->data.fl[i], 정수형이라면 spatial->data.i[i]로 사용하면 된다. 이것은 CvMat 구조체에서 행렬 값을 저장하는 data 변수가 union으로 정의되어 있기 때문이다.

DFT의 결과는 freq 변수에 저장되며 변환 결과에 이상이 없는지는 보통 Inverse DFT나 크기 값을 화면에 출력하여 확인한다. 단, 주파수 영역의 값은 수천에서 수만까지로 모니터에 표시하기엔 그 값의 변화가 너무 크기 때문에 로그(log)를 이용하여 화면에 출력한다. 〈화면 2〉의 (c)와 같이 보여주기 위해서 반 주기 이동을 Shift() 함수에서 수행한다. 이 함수는 푸리에 변환의 결과가 원점 대칭이라는 점을 이용하여 좌상, 좌하, 우상, 우하, 네 영역에 대해서 좌상 ↔ 우하, 우상 ↔ 좌하 영역으로 상호 교환하여 반 주기만큼 이동한 효과를 낸다.

마지막으로 cvDFT(freq, spatial, CV_DXT_INVERSE_SCALE)를 통해 역변환을 한다. CV_DXT_INVERSE 옵션에는 전체 픽셀 수로 나눠주는 과정이 생략되어 있으므로 꼭 CV_DXT_INVERSE_SCALE을 사용한다.

DFT와 FFT는 신호처리에서 유명한 알고리즘 중에 하나이기 때문에 인터넷에서 쉽게 소스코드를 얻을 수 있으며 특히, 속도와 정확성에서 우수한 성능을 보여 주는 FFT 중에서 MIT에서 만들어져 공개되고 있는 FFW도 있으므로 한번 찾아 실행해보길 바란다(참고자료 ❹).

참고로 〈리스트 2〉와 같이 IplImage와 CvMat를 함께 사용할 때는 영상의 너비와 행렬의 컬럼 수에 대해서 주의해야 한다. 앞에서 설명한 것처럼 IplImage는 생성될 때, 너비가 4바이트의 배수가 되도록 width 대신에 widthStep만큼 자동 생성되지만, CvMat는 설정한 크기만큼만 생성된다. 그러므로 〈리스트 2〉에서는 IplImage와 CvMat 간의 연산에서 생길 수 있는 실수를 줄여주기 위해서 CvMat의 컬럼 수를 widthStep만큼을 생성하였다.

이산 코사인 변환, DCT

DCT(Discrete Cosine Trasnform)는 cos을 이용한 변환이며 DFT 및 다른 변환과 비교했을 때, 〈화면 3〉의 (b)와 같이 변환 후에 좌측 상단 저주파 영역(밝은 부분)에 에너지가 집중되기 때문에 데이터 압축 효과가 우수하다. 이것은 (b)의 우측 하단의 어두운 영역의 주파수 성분들은 영상에서 사람의 눈에 잘 띄지 않는 부분이나 노이즈에 해당하므로 이 영역을 버림으로써 압축률을 높일 수 있기 때문이다. 예를 들어 256×256 크기의 영상을 이루는 주요 cos 신호가 좌측상단 30×30 영역에 집중되어 있다고 가정하면, 256×256개의 원본 데이터를 30×30개의 데이터만으로 거의 동일하게 표현 가능하므로 약 73배로 압축하는 효과를 얻을 수 있다. 이러한 DCT의 특성을 JPEG에서 이용하고 있으며, 포토샵 등에서 JPEG으로 저장할 때 압축률을 정하는 것은 바로 이 저주파 영역의 개수를 얼마만큼 사용할 것인가와 연관이 있다. 참고로 JPEG2000에서는 DCT보다 압축률이 우수한 웨이블릿(Wavelet)을 사용한다. 좀 더 자세한 내용은 영상 처리 서적의 압축 관련 부분이나 JPEG 관련 서적을 살펴하자(참고자료 ❺).

```
// 이산 코사인 변환(DCT)을 실행한다.
void cvDCT(const CvArr* src,        // 원본 영상의 데이터
           CvArr* dst,              // 주파수 영역
           int flags )              // DCT 또는 Inverse DCT 결정
```

cvDCT는 DFT와 달리 sin 성분이 없기 때문에 주파수 영역의 결과값은 실수 값이고 CvMat의 채널은 1로 설정하면 된다.

K-Means 알고리즘을 통한 클러스터링

클러스터링이란 물리적 혹은 추상적 객체를 비슷한 객체군으로 묶는 과정이다. 두 개 이상의 컴퓨터를 여러 대 묶어서 한 대의 컴퓨터로 사용할 때도 클러스터링이란 용어를 사용하는데, 영상처리에서는 섞여 있는 데이터에 대해서 그룹을 정해주는 것을 클러스터링이라고 한다. 예를 들면 3종류의 강아지 500마리가 무작위로 섞여 있을 때, 3개의 그룹으로 분류하는 것이 바로 클러스터링이다. 또한, 사람이 강아지를 분류할 때엔 그 기준이 털색이나 다리의 길이, 얼굴 모양 등 여러 가지가 될 수 있다. K-Means는 영상 처리에서 클러스터링 방법 중에서 그룹의 중심들과의 유클리디언 거리를 비교하여 그룹을 분류하는 방법이다. 〈화면 4〉와 같이 임의의 500개의 데이터에 대해서 3개의 그룹으로 분리하려고 한다면 K-Means 방법은 다음과 같이 이루어진다.

❶ 3개의 그룹의 중심점 A, B, C를 임의로 잡는다.

❷ 우선 첫 번째 데이터에 대해 중심점 A, B, C와의 유클리디언 거리를 각각 측정한다.

❸ 만약 A와의 거리가 10, B와의 거리가 3, C와의 거리가 170이 나왔다면, 첫 번째 데이터는 거리가 가장 가까운 B 그룹으로 분류한다.

❹ 500개의 데이터에 대해서 2번과 3번의 과정을 반복하여 그룹을 결정한다.

❺ 모든 데이터에 대해서 그룹이 정해지면, 각 그룹의 평균좌표를 계산한다.

❻ 각 그룹에서 새롭게 계산된 평균 좌표를 새로운 중심점 A, B, C로 삼는다.

〈리스트 4〉 K-Means 알고리즘의 간단한 활용

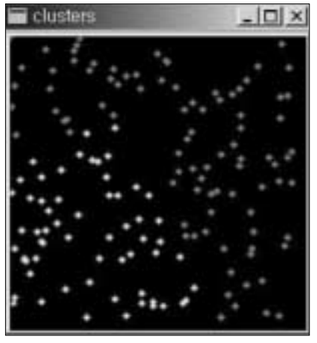
```
void COpenCVTestDlg::OnBtnKmeans()
{
   #define MAX_CLUSTERS 5
   CvScalar color_tab[MAX_CLUSTERS];
   IplImage* img = cvCreateImage( cvSize( 210, 210 ), 8, 3 );
   CvRNG rng = cvRNG(-1);
   CvPoint ipt;

   color_tab[0] = CV_RGB(255,0,0);
   color_tab[1] = CV_RGB(0,255,0);
   color_tab[2] = CV_RGB(100,100,255);
   color_tab[3] = CV_RGB(255,0,255);
   color_tab[4] = CV_RGB(255,255,0);
   cvNamedWindow( "clusters", 1 );

   int key, k, cluster_count= 3;
   int i, sample_count = 500;
   // 2-Channel이므로, x, y좌표를 구현: (x0, y0)-> (points[0], points[1])
   CvMat* points  = cvCreateMat( sample_count, 1, CV_32FC2 );
   CvMat* clusters= cvCreateMat( sample_count, 1, CV_32SC1 );

   for( k = 0; k < sample_count/2; k++ ) { // 입력 데이터 500개를 임의로 생성한다
      points->data.fl[2*k]   = rand()%200+1; // 임의의 x좌표
      points->data.fl[2*k+1] = rand()%200+1;// 임의의 y좌표
   }
   // K-Means 실행
   cvKMeans2( points, cluster_count, clusters,
               cvTermCriteria( CV_TERMCRIT_EPS+CV_TERMCRIT_ITER, 10, 1.0 ));

   cvZero( img );
   for( i = 0; i < sample_count; i++ ) { // 분류된 그룹에 맞는 색상으로 화면 출력
      int cluster_idx = clusters->data.i[i];
      ipt.x = (int)points->data.fl[i*2];
      ipt.y = (int)points->data.fl[i*2+1];
      cvCircle( img, ipt, 2, color_tab[cluster_idx], CV_FILLED, CV_AA, 0 );
   }

   cvReleaseMat( &points );
   cvReleaseMat( &clusters );
   cvShowImage( "clusters", img );
}
```


❼ 종료 조건을 만족할 때까지 2~6번의 과정을 반복한다.

참고로 K-Means라고 불리는 것은 이와 같이 초기에 주어진 K개의 그룹의 중심(mean)을 통해 클러스터링 하기 때문이다. K-Means 대한 설명 및 영상에 적용한 인터넷 사이트도 있으니 찾아가 보도록 하자(관련자료 ❻).

```
// K-Means 알고리즘 수행. 임의의 데이터를 cluster_count 개의 그룹으로 분리한다.
void cvKMeans2( const CvArr* samples,          // 임의의 데이터
                int cluster_count,             // 분리될 그룹 수
                CvArr* labels,                 // 데이터에 매겨질 그룹 번호
                CvTermCriteria termcrit ); // 종료 조건
```

〈리스트 4〉에서 Samples 변수는 입력 데이터로서 임의의 차원을 가질 수 있다. 〈화면 4〉와 같이 2차원 데이터에 대해 클러스터링한다고 가정했기 때문에 cvCreateMat()의 채널 값을 2로 주었다. 3개의 그룹으로 분리시키기 위해 그룹을 나타내는 cluster_count 변수를 3으로 정하고, K-Means의 결과로 매겨질 그룹 번호가 저장되는 label 변수는 입력 데이터의 개수만큼 생성한다. 마지막 CvTermCriteria는 K-Means 알고리즘의 종료 조건을 정한다. K-Means와 같이 반복 계산법(Iterative Method)을 사용하는 알고리즘은 종료 조건을 알고리즘 수행 최대 반복 회수(CV_TERMCRIT_ITER, max_iter 변수)나 알고리즘을 얼마나 정확한 정도까지 수행할 것인지를 나타내는 정확도(CV_TERMCRIT_EPS, epsilon 변수)로 정한다. 〈리스트 4〉에서는 실행 회수가 10회 이상이거나 정확도가 1.0이하이면 K-Means가 종료되도록 하였다.



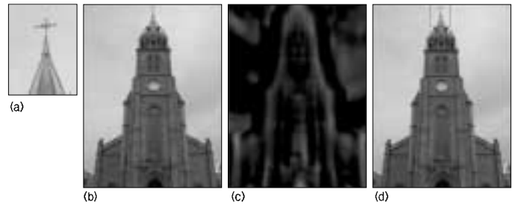
〈화면 4〉
K-Means 알고리즘을 이용하여 3개의 그룹으로 분리한 결과

```
typedef struct CvTermCriteria
{
    int    type;         // CV_TERMCRIT_ITER 혹은 CV_TERMCRIT_EPS의 OR연산
    int    max_iter;     // 최대 반복 회수, 여기선 10회
    double epsilon;      // 정확도, 여기선 0.1
} CvTermCriteria;
```

템플릿 매칭을 통한 객체 검출

템플릿 매칭(Template matching)은 우리가 찾고자 하는 객체를 템플릿으로 정의하고 목표 영상에서 템플릿의 위치를 찾는다(참고자료 ❼). 즉, 〈화면 5〉의 (b)와 같은 영상에서 (a)라는 객체를 찾으려고 할 때 사용하는 방법 중에 한 가지로서, 템플릿을 목표 영상의 전 영역을 훑어가면서(scan) 템플릿과 비교를 한다. 비교 방법은 주로 두 영상의 유사도를 계산하는 상관계수(Correlation Coefficient)를 주로 사용한다. 즉, 유사도가 높을수록 객체일 확률이 높다는 것을 이용하여 위치를 찾는다. (c)는 템플릿 매칭의 결과로서 밝을수록 유사도가 높다는 것을 의미한다. (d)는 cvThreshold()를 적용했으며 상단 중앙의 밝은 점은 최고의 유사도를 보이는 위치로서 객체의 위치를 정확히 찾은 것을 확인할 수 있다.

<화면 5> 템플릿 매칭을 통한 물체의 검출 결과



(a) 찾으려는 물체 (b) 물체가 포함된 영상, (c) cvMatchTemplate() 함수의 결과 영상 (d) cvThreshold() 함수를 통해 상관계수가 가장 큰 위치 표시

```
void cvMatchTemplate(const CvArr* image,          // 원본 영상
                     const CvArr* templ,          // 찾으려는 객체 영상 - 템플릿
                     CvArr* result,               // 결과 영상
                     int method );                // 유사도 측정 방법
```

cvMatchTemplate()을 사용할 때 주의할 부분이 바로 유사도 측정 방법이다. OpenCV의 도움말을 살펴보면 유사도 측정 방법으로 유클리디언 거리를 이용하는 방법(CV_TM_SQDIFF, CV_TM_SQDIFF_ NORMED), 상관도(Correlation)를 이용하는 방법(CV_TM_ CCORR, CV_TM_CCORR_NORMED), 상관계수를 이용하는 방법(CV_TM_CCOEFF, CV_TM_CCOEFF_ NORMED) 등을 제공한다. 여기서 NORMED의 의미는 결과 값의 크기를 0~1 사이로 정규화했다는 뜻이다.

〈리스트 5〉에서는 상관계수를 사용했으며 상관계수의 범위는 -1에

서 1까지다. 상관계수 값이 -1이나 1에 가까워질수록 상관성이 커지며, 0에 가까워질수록 상관성이 없어진다. 특히, 템플릿 매칭에서는 1에 가까울수록 물체와 유사하다. cvMinMaxLoc() 함수를 이용하면 상관계수의 최소, 최대 값이 나온 위치를 찾을 수 있다.

템플릿 매칭은 영상에서 물체의 위치를 찾을 때 사용하는 기본적인 방법이다. 단, 이와 같이 찾고자 하는 영상이 템플릿과 유사할수록 정확한 위치를 찾을 수 있지만, 영상에 변형이 생기거나 배경의 노이즈가 많은 경우에는 원하는 결과를 얻을 수 없는 경우가 발생한다. 따라서 템플릿 매칭의 성능을 향상시키기 위해서 템플릿의 모양을 지능적으로 변화시킨다거나, 유사도 비교 방법을 개선시킨 다양한 알고리즘이 연구되고 있다.

옵티컬 플로우를 이용한 영상의 매칭

옵티컬 플로우(Optical Flow)는 〈그림 1〉과 같이 영상의 한 점이 다른 영상에서는 어디에 위치하는지를 찾는 알고리즘으로 〈그림 1〉 (c)와 같이 영상 간의 움직임을 분석할 수 있다 (참고자료 ❽). 옵티컬 플로우는 이동된 점의 위치를 찾기 위해서 이동된 점은 원래 위치에서 멀리 움직이지 않았다는 것과 점이 이동된 후에도 색상은 변하지 않는다는 것을 가정한다. 또한, 점이 이동할 때 주변 K×K개의 점들도 함께 이동하는 것을 가정하며 최소자승법(least Square)으로 이동된 점의 위치를 찾는다.

OpenCV에서는 옵티컬 플로우를 위해 CalcOpticalFlowHS(), CalcOpticalFlowLK(), CalcOpticalFlowBM(), CalcOpticalFlowPyrLK() 등의 함수를 지원하며 여기선 피라미드를 이용한 Lukas-Kanade의 방법에 대해 설명하겠다.

이러한 옵티컬 플로우는 〈그림 2〉와 같이 연속된 장면을 담은 사진들을 파노라마 영상으로 만드는 프로그램은 물론 2차원 영상과 3차원 영상의 매칭, 세그맨테이션(segmentation) 등의 분야에서 사용되고 있다.

```
void cvCalcOpticalFlowPyrLK(
      const CvArr* prev,                    // 첫 번째 영상
      const CvArr* curr,                    // 두 번째 영상
      CvArr* prev_pyr,                      // 첫 번째 영상의 피라미드
      CvArr* curr_pyr,                      // 두 번째 영상의 피라미드
      const CvPoint2D32f* prev_features,    // 첫 번째 영상에서 원래 점의 위치
      CvPoint2D32f* curr_features,          // 두 번째 영상에서 찾은 점의 위치
      int count,                            // 찾으려는 점의 개수
      CvSize win_size,                     // search window의 크기
      int level,                            // 피라미드 레벨 지정
     char* status               // status=1: 이동된 위치 찾은 경우,
                                // status=0: 이동된 위치 찾지 못한 경우
      float* track_error,                  // NULL
      CvTermCriteria criteria,             // 종료 조건
      int flags );                          // CV_LKFLOW_INITIAL_GUESSES등
```

옵티컬 플로우 함수에서 prev와 curr는 첫 번째, 두 번째 영상의 포인터로 포맷은 그레이 영상이다. 옵티컬 플로우의 가정에 따라 영상은 〈그림 1〉과 같이 움직임이 있는 영상이며, 움직임의 차이가 적은 것이 좋다. 다르게 말하면 움직임이 클수록 오류가 생길 확률이 높아진다.

prev_pyr와 curr_pyr은 첫 번째, 두 번째 영상의 피라미드에 대한 포인터이다. 피라미드란 영상을 여러 단계로 축소시켜서 쌓은 것을 말하는데 영상 전체를 비교하여 점을 찾기에는 연산량이 많으므로 축소시킨 영상에서 후보 영역을 찾고 이 후보 영역을 검증하여 이동된 점의 위치를 찾아 연산량을 줄인다. 피라미드에 관한 자세한 내용은 영상처리 서적 및 관련 사이트를 참고하자(참고자료 ❽).

〈리스트 6〉에서는 NULL 값을 사용하며 이것은 내부적으로 피라미드를 생성해서 사용한다는 뜻이다. NULL 값이 아닌 경우 함수는 피라미드 생성하는 작업을 prev_pyr 혹은 curr_pyr 변수에서 한다. 이 경우 CV_LKFLOW_PYRA_READY, CV_LKFLOW_PYR_B_READY 옵션을 사용하며 피라미드를 생성하기에 충분한 메모리 공간을 잡아줘야 한다. count는 찾고자 하는 점의 개수로 1개를 지정하며, 점의 위치를 피라미드 레벨은 5로 주었다. status는 점을 찾은 경우 1, 찾지 못한 경우는 0을 되돌려주며, track_error는 에러 값을 저장하는데 필요가 없으므로 NULL 값을 지정했다.

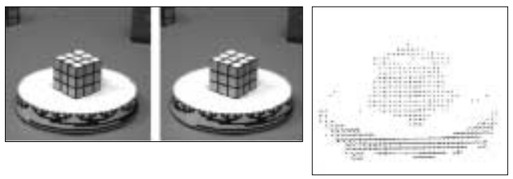
〈리스트 5〉 템플릿 매칭

```
void COpenCVTestDlg::OnBtnMatchTemplate()
{
   IplImage* src=cvLoadImage("c:/myung.jpg", 0);
   IplImage* src2=cvLoadImage("c:/myung2.jpg", 0);
   IplImage* dst_image=cvCreateImage( cvSize(src->width-src2->width+1,
                                    src->height-src2->height+1), IPL_DEPTH_32F,
1 );

   //상관계수를 이용하여 템플릿 매칭
   cvMatchTemplate(src, src2, dst_image, CV_TM_CCOEFF_NORMED);

   double min_val, max_val;
   CvPoint max_val_pos;
   cvMinMaxLoc(dst, &min_val, &max_val, NULL, &max_val_pos);//유사도가 최대인 위
치 얻음

      cvThreshold(dst_image,  dst_image,  max_val-max_val*0.3,    100,
CV_THRESH_TOZERO);

   // cvNamedWindow(), cvShowImage()를 이용한 결과 화면 출력 및 IplImage의 리소스
해제
}
```

〈리스트 6〉 Optical Flow를 이용한 점의 이동된 위치 찾기

```
/* 옵티컬 플로우를 위한 전역 변수 */
IplImage* g_pIpl1, *g_pIpl2;
IplImage* g_pIplColor1, *g_pIplColor2;
POINT g_pt;                          // 현재 선택된 점의 좌표
char* g_szWnd1= "FirstWindow";       // 윈도우 이름 1
char* g_szWnd2= "SecondWindow";      // 윈도우 이름 2
void  OnFirstMousePoint(int event, int x, int y, int flags);
void  OpticalFlowMatching(IplImage* ipl1, IplImage* ipl2, POINT pt);

// 옵티컬 플로우의 초기화:영상을 출력하고 마우스 이벤트를 등록한다.
void COpenCVTestDlg::OnBtnOpticalFlow()
{
   //cvCalcOpticalFlowPyrLK() 알고리즘은 그레이 영상에서 수행되기 때문에
   //화면에 보여줄 컬러 영상을 따로 생성한다.
   g_pIplColor1 = cvLoadImage("scene1.jpg", 1);  // 화면에 보여줄 영상
   g_pIpl1  = cvLoadImage("scene1.jpg", 0);      // 옵티컬 플로우를 처리할 영상
   cvNamedWindow(g_szWnd1,  CV_WINDOW_AUTOSIZE  );
   cvShowImage(g_szWnd1, g_pIplColor1);

   g_pIplColor2 = cvLoadImage("scene2.jpg", 1);    // 화면에 보여줄 영상
   g_pIpl2  = cvLoadImage("scene2.jpg", 0);        // 옵티컬 플로우를 처리할 영상
   cvNamedWindow(g_szWnd2, CV_WINDOW_AUTOSIZE );   //0 );
    cvShowImage(g_szWnd2, g_pIplColor2);

   cvSetMouseCallback(g_szWnd1, OnFirstMousePoint);   /* 마우스 클릭에 반응 */
}

/* 마우스 오른쪽 버튼에 반응하는 함수 */
void OnFirstMousePoint(int event, int x, int y, int flags)
{
   if (event==CV_EVENT_LBUTTONDOWN) {
      g_pt.x = x; // 선택된 점의 좌표를 옵티컬 플로우 함수에 넘겨줌
      g_pt.y = y;
      OpticalFlowMatching(g_pIpl1, g_pIpl2, g_pt);
   }
}

/* 옵티컬 플로우를 수행하는 함수 */
void OpticalFlowMatching(IplImage* ipl1, IplImage* ipl2, POINT pt)
{
   POINT xy; xy.x=-1; xy.y=-1;
   if (!ipl1 || !ipl2) return ;
   if (pt.x==-1 || pt.y==-1) return ;

   CvPoint2D32f featureIpl1, featureIpl2;
   featureIpl1.x = (float)pt.x;
   featureIpl1.y = (float)pt.y;
   featureIpl2 = featureIpl1;

   CvSize winSize;
   winSize.height = 5;
   winSize.width  = 5;

   char status;
   CvTermCriteria criteria = { CV_TERMCRIT_ITER|CV_TERMCRIT_EPS, 10, 0.1 };

   /* 피라미드를 이용한 Lukas-Kanade 옵티컬 플로우를 이용한 이동된 점의 위치 찾기 */
   cvCalcOpticalFlowPyrLK(ipl1, ipl2, NULL, NULL,&featureIpl1, &featureIpl2, 1,
                  winSize, 5, &status, NULL, criteria, CV_LKFLOW_INITIAL_GUESSES
);

   if (status==1) { /* 매칭되는 점을 찾은 경우 화면에 표시해줌 */
      CvPoint ptSelected;
      ptSelected.x = pt.x;
      ptSelected.y = pt.y;

      int r= rand()%256, g= rand()%256, b= rand()%256; // 점의 색상 선택
      // cvCircle()은 화면에 원을 그려주는 함수
      cvCircle(g_pIplColor1, ptSelected, 3, CV_RGB(r,g,b), -1);
      ptSelected.x = (int)featureIpl2.x;
      ptSelected.y = (int)featureIpl2.y;
      cvCircle(g_pIplColor2, ptSelected, 3, CV_RGB(r,g,b), -1);
      cvShowImage(g_szWnd1,  g_pIplColor1);
      cvShowImage(g_szWnd2, g_pIplColor2);
   }
}
```

<그림 1> 옵티컬 플로우를 이용한 움직임 추정



<그림 2> 피노라마 영상의 예

CvTermCriteria는 알고리즘의 종료 조건으로 최소자승법과 같은 반복 계산법으로 문제를 푸는 경우 사용한다. 앞의 K-Means의 경우처럼 해의 정확도 혹은 해를 찾는 최대 반복 횟수를 지정한다. flag는 CV_LKFLOW_INITIAL_GUESSES를 사용하였다. 이것은 curr_features의 초기 위치를 첫 번째 영상의 점의 위치인 prev_features 동일하게 잡겠다는 의미이다.

CV_LKFLOW_PYR_A_READY, CV_LKFLOW_PYR_B_READY 옵션은 첫 번째, 두 번째 영상에서 피라미드를 미리 생성할 것인지 말것인지를 정한다.

cvCalcOpticalFlowPyrLK() 함수의 인자가 다소 복잡한 감은 있지만, 리스트에서 사용된 방법처럼 인자를 설정하여 사용하면 된다. 〈화면 6〉은 옵티컬 플로우를 이용하여 이동된 점을 찾은 결과이다. 옵티컬 플로우의 가정에서처럼 영상의 움직임의 차가 크지 않고 변화가 적을수록 이동된 점의 위치를 더 정확하게 찾는다.

〈리스트 6〉은 첫 번째 영상의 임의의 점에서 클릭하면 동일한 점의 위치를 두 번째 영상에서 찾아 화면에 표시해준다. 영상에서 클릭된 점의 위치는 OpenCV의 Highgui.h의 cvSetMouseCallback() 함수를 이용하여 얻을 수 있다. 즉, cvSetMouseCallback(m_szWnd1, OnFirstMousePoint())를 통해 m_szWnd1 윈도우에서 마우스 클릭 이벤트가 일어났을 때마다 사용자 정의 함수인 OnFirstMousePoint 함수를 호출한다. 따라서 마우스 클릭시마다 일어나는 이벤트는 OnFirstMousePoint() 함수에서 처리하면 되고, 〈리스트 6〉에서는 cvCircle()를 통해 화면에 점을 찍어준다. 단, OnFirstMousePoint() 함수의 이름은 사용자가 마음대로 지정할 수 있지만 3개의 인자는 동일하게 설정해주어야 한다. 마우스 이벤트의 경우 왼쪽 버튼 클릭(CV_EVENT_LBUTTONDOWN), 오른쪽 버튼 클릭(CV_EVENT_RBUTTONDOWN) 등으로 정의되어 있다. 이벤트의 종류에 대한 설명은 도움말에서 cvSetMouseCallback() 함수에 나와 있다.

CBCH를 이용한 얼굴 검출

마지막으로 〈화면 7〉과 같이 영상에서 얼굴을 찾는 방법에 대해 알아보자. 얼굴 검출은 영상 처리 중 패턴 인식 분야에 해당하며 영상의 모든 영역을 비교하여 얼굴인지 아닌지를 판단하는 방법이다. 물체를 찾는 간단한 방법은 앞의 템플릿 매칭에서도 배웠다. 템플릿 매칭에서는 유사도를 계산하는 수식을 통해 영상을 구분했지만, 패턴 인식에서는 영상의 종류를 판별하는 분류기(classifier)를 학습시켜서 원하는 패턴인지 아닌지 구별한다. 예를 들면, 얼굴 검출에서는 분류기에 여러 얼굴 영상을 보여줘서 얼굴의 특징을 학습시킨 뒤, 임의의 영상이 얼굴인지 아닌지를 분류기가 판단하게 한다. 이러한 방법에는 신경망 회로(Neural Network), SVM(Support Vector Machine), PCA(Principle Component Analysis), LDA(Linear Discriminant Analysis) 등 많은 알고리즘이 있으며 이번에 소개하는 Cascade of boosted classifiers working with haar-like features를 이용한 객체 검출 알고리즘은 빠른 속도와 높은 정확도가 특징이다(참고자료 ❾). 알고리즘의 이름이 길기 때문에 여기서부터는 간단히 CBCH라고 부르도록 하겠다.

CBCH를 이용한 검출 방법의 특징은 분류기의 연산이 단순하다는 것이다. 〈그림 3〉과 같이 검정색 영역의 합과 흰색 영역의 합의 차이가 Haar 분류기의 특징 값에 해당하며 이 값을 비교하여 얼굴 영상인지 아닌지를 판별한다. 이 Haar 분류기는 연산이 단순한 만큼 얼굴 검출률은 높지 않다. 그러나 이러한 Haar 분류기를 100개 혹은 1000개 이상을 적절히 조합함으로써 분류기의 성능을 높일 수 있다는 것이 CBCH의 핵심이다. 또한, 이 Haar 분류기의 특징값은 누적된 밝기 값을 가지는 영상(Integral Image)을 이용하기 때문에 검정색 영역의 합과 흰색 영역의 합의 차는 몇 번의 덧셈 뺄셈으로 이뤄질 수 있어서 다른 패턴 인식 방법에 비해 계산 속도가 굉장히 빠르다.

얼굴 검출에 적절한 Haar 분류기의 조합을 찾기 위해서 아다부트스(Adaboost)라는 알고리즘을 사용한다. CBCH의 boosted는 바로 아다부스트 알고리즘과 같이 여러 개의 분류기를 조합해서 사용하는 부스트(Boost) 알고리즘을 사용하기 때문이다. 아다부스트는 얼굴 영상에서 가능한 모든 형태의 Haar 분류기에 대해서 얼굴 판별 능력이 뛰어난 순서대로 Haar 분류기를 추출해준다. 물론, 각각의 Haar 분류기의 성능이 다르기 때문에 가중치도 함께 계산되어 나온다. 그리고 추출된 분류기들은 임의의 영상에 대해서 각각 얼굴 영역인지 아닌지를 판별하게 되고 다수결에 따라서 얼굴 영상인지 아닌지 판별되게 된다. 예를 들면, 100개의 분류기가 있다면, 임의의 영상에 대해서 60개의 분류기가 얼굴이라고 판단하고 40개가 얼굴이 아니라고 판단하면 다수결에 따라 얼굴이 아니라고 판단하게 된다. 여기서 적절한 분류기의 조합을 찾기 위해서 적게는 몇 백장에서 수 천장의 테스트 얼굴 영상이 필요하며, 이런 과정을 바로 분류기를 학습(Learning 혹은 Training)시킨다고 한다.

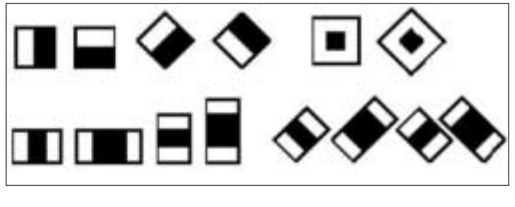
1000개의 Haar 분류기를 사용한다고 했을 때도 한 번에 1000개 모두를 비교하는 것이 아니라 처음엔 1개, 그 다음 10개, 25개, 25개, 50개 등 그 개수를 증가시키면서 차례로 비교하는 방법을 사용한다. 이렇게 단계를 나누어서 비교하는 것을 cascade라고 하며 얼굴 속도를 가속화시킬 수 있다. 예를 들어 얼굴 영상이 아닌 것을 판단하기 위해 1000개를 한꺼번에 비교한다면 연산량이 1000번이 될 것이다. 그러나 이처럼 단계를 나눠 비교하면, 처음 1개에서 얼굴이 아니라고 판단되면 999번의 연산을 하지 않아도 되므로 검출 속도를 높일 수 있게 된다.

OpenCV에서는 이 CBCH를 구현하기 위해서 도움말에서 설명된 함수 외에도 많은 함수를 사용하지만 〈리스트 7〉과 같이 주요 함수 2개로 얼굴 검출을 구현할 수 있다. 우선 cvLoad() 함수를 이용하여 정면 얼굴 검출에 대한 CBCH파일 (haarcascade_frontalface_default.xml)을 읽어온다. 이 파일은 CBCH를 학습시킨 파일로써 OpenCV 4.1로 버전업되면서 다양한 *.xml 을 OpenCV\data\haarcascades 폴더에서 제공한다. 다음으로 CvHaarClassifierCascade * cascade와 cvHaarDetectObjects() 함수를 이용하여 실제 얼굴 검출을 수행한다.

<화면 6> Optical Flow를 이용한 연속된 두 장의 사진에서 이동된 점의 위치 찾기

<화면 7>
CBCH를 이용한 얼굴 검출 결과

<그림 3> Haar 분류기의 종류



〈리스트 7〉 Boosted Haar classifier를이용한 얼굴 검출 방법

```
void COpenCVTestDlg::OnBtnHaar()
{
   IplImage* image=cvLoadImage("lena.jpg", -1);

   // Boosted Haar classifier의 정보 읽기
   CvHaarClassifierCascade* cascade
      =(CvHaarClassifierCascade*)cvLoad("haarcascade_frontalface_default.xml");

   // 얼굴 검출
   detectObjects( image, cascade );

   cvNamedWindow( "test", 0 );
   cvShowImage( "test", image );

   cvReleaseHaarClassifierCascade( &cascade );
   cvReleaseImage( &image );
}

// cvHaarDetectObjects()를 이용해 얼굴 검출하고 화면에 표시
void detectObjects( IplImage* image, CvHaarClassifierCascade* cascade )
{
   CvMemStorage* storage = cvCreateMemStorage(0);
   CvSeq* faces;
   int i, scale = 1;

   /* use the fastest variant */
   faces = cvHaarDetectObjects( image, cascade, storage,
                                        1.2, 2, CV_HAAR_DO_CANNY_PRUNING );

   /* draw all the rectangles */
   for( i = 0; i < faces->total; i++ )
   {
      /* extract the rectanlges only */
      CvRect face_rect = *(CvRect*)cvGetSeqElem( faces, i);//, 0 );
      cvRectangle( image, cvPoint(face_rect.x*scale,face_rect.y*scale),
         cvPoint((face_rect.x+face_rect.width)*scale,
            (face_rect.y+face_rect.height)*scale), CV_RGB(255,0,0), 3 );
   }
   cvReleaseMemStorage( &storage );
}
```

```
CvSeq* cvHaarDetectObjects(
      const CvArr* image,                    // 원본 영상
      CvHaarClassifierCascade* cascade,      // CvHaarClassifierCascade 객체
      CvMemStorage* storage,                 // 후보 얼굴 영역을 위한 저장 공간
      double scale_factor=1.1,               // search window 확대 배율
      int min_neighbors=3,                   // 얼굴 영역 판단 기준
      int flags=0,                           // CV_HAAR_DO_CANNY_PRUNING
      CvSize min_size=cvSize(0,0) );         // search window 사이즈
```

cvHaarDetectObjects() 함수를 이해하기 위해서는 CBCH의 얼굴 검출 방법에 대한 이해가 필요하다. CBCH와 같은 얼굴 검출 방법은 〈화면 7〉에서 보이는 사각형 영역의 크기를 변화시키면서 영상 전 영

<화면 8> Cascade of boosted classifiers working with haar-like features를 이용한 얼굴검출결과

(a) 정지영상에서의 얼굴 검출

(b) 동영상에서의 얼굴 검출

역에 대해서 얼굴인지 아닌지 검출한다. 이 사각형 영역을 search window라고 한다. search window의 크기가 변화하기 때문에 작은 얼굴부터 큰 얼굴까지 검출할 수 있으며, 영상의 전 영역을 검사하기(scan) 때문에 얼굴이 어느 위치에 있어도 검출이 가능하다. scale_factor는 이 search window의 확대 비율이며, 1.1이라면 search window를 1.1배씩 확대시키면서 얼굴을 검출한다. 물론 함수 내에서는 search window 대신 영상의 크기를 변화시키는 기교를 부려서 연산량을 줄인다. 얼굴을 검출하다 보면, 얼굴 주변 영역일수록 얼굴이라고 판단하는 확률이 높아진다. 여기서 min_neighbors=3의 의미는 임의의 영역을 얼굴 영역이라고 정의하기 위해서는 최소한 3번 이상 얼굴 영역이라고 판단되어야 한다는 뜻이다. flags는 현재 CV_HAAR_DO_CANNY_PRUNING 옵션만이 가능하며, Prune이 군더기를 제거한다는 뜻인 만큼 얼굴 후보 영역에 대해서 케니 에지(canny edge)를 구해보고 에지가 너무 많으면 얼굴 후보 영역에서 제거한다. 마지막으로 min_size는 search_window의 초기 크기로서, OpenCV에서 제공하는 xml 파일의 CBCH는 20×20으로 학습되었기 때문에 20×20 이상으로 설정하는 것이 좋다.

이제 CBCH를 통해 정지 영상과 동영상에서 각각 얼굴 검출을 해보자. 우선 〈리스트 7〉은 〈화면 8〉 (a)와 같이 CBCH를 이용하여 얼굴을 성공적으로 검출한다. OpenCV에서 기본적으로 제공하는 CBCH의 경우에도 얼굴 검출 성능이 높은 만큼 필요시 적극 활용하면 좋을 것이다. 〈화면 8〉의 (b)는 동영상에서 얼굴 검출하는 예로서 지난 호의 〈리스트 2〉에서 단지 detectObjects(frame_copy , cascade, 1)만 추가해준 결과이다.

이번 연재에서 다루지는 않았지만 OpenCV\apps\HaarTraining 폴더에는 CBCH의 학습된 결과가 저장된 *.xml 파일을 생성하는 툴이 있다. 자신만의 Boosted Haar classifier를 만들고 싶다면 HaarTraining를 분석해 보자.

시행착오를 두려워하지 말자

지금까지 OpenCV에서 제공하는 고급 영상처리 함수들에 대해 알아보았다. OpenCV 라이브러리가 방대한 만큼 다루지 못한 함수가 더 많은 것이 사실이다. 그러나 지금까지 설명한 것만이라도 잘 사용할 수 있다면 다른 함수들 또한 몇 번의 시행착오만 겪으면 쉽게 사용할 수 있을 것이다. 물론, 그 함수를 자유자재로 사용하기 위해서는 그 함수의 바탕이 되는 영상처리 이론 또한 뒷받침이 되어야 할 것이다. 참고자료에 영상처리 알고리즘에 대해 유용한 사이트들을 언급했으니 참고하길 바란다.

정리 | 강경수 | elegy@korea.com

이 + 달 + 의 + 디 + 스 + 켓
openCVTest02.zip http://www.imaso.co.kr

참 + 고 + 자 + 료

❶ 색상계에 관한 설명, http://developer.apple.com/documentation/mac/ACI/ACI-48.html
❷ 푸리에 변환에 관한 설명, http://mathworld.wolfram.com/FourierTransform.html
❸ DFT, FFT에 관한 설명, http://astronomy.swin.edu.au/~pbourke/analysis/dft/)
❹ FFTW에 관한 설명, http://www.fftw.org
❺ JPEG에 관한 설명,
http://www.cs.sfu.ca/CourseCentral/365/li/material/notes/Chap4/Chap4.2/Chap4.2.html
❻ K-Means에 관한 설명, http://www.ece.neu.edu/groups/rpl/projects/kmeans/
❼ 템플릿 매칭에 관한 설명, http://www.cis.temple.edu/~latecki/CIS601-03/Lectures/TemplateMatching03.ppt
❽ 옵티컬 플로우에 관한 설명,
http://www.cs.ucf.edu/courses/cap6411/cap5415/spring03/lecture18.pdf
http://cs223b.stanford.edu/CS%20223-B%20L9%20Optical%20Flow.ppt
❾ Cascade of boosted classifiers working with haar-like features에 관한 논문,
http://www.ai.mit.edu/~viola/research/publications/CVPR-2001.pdf