Implementação de um algoritmo de rastreamento de objetos móveis em vídeo

João Felipe Santos
05141273

01 de julho de 2010

Universidade Federal de Santa Catarina
Centro Tecnológico
Departamento de Engenharia Elétrica
EEL7815 - Projeto Nível I em Controle e Processamento de Sinais I
Professor: Joceli Mayer, Ph.D.

---

1 Introdução

O problema de rastreamento de objetos em vídeo é considerado importante atualmente, especialmente ao se considerar suas aplicações em visão computacional e análise automática de vídeos. Este problema pode ser dividido basicamente em três partes [1]:

- Detecção de objetos de interesse em movimento
- Rastreamento destes objetos quadro a quadro
- Análise da trajetória dos objetos para reconhecimento de seu comportamento

Como áreas de aplicação, pode-se citar o monitoramento de segurança automático, reconhecimento de gestos, auxílio a navegação de veículos e detecção automática de objetos baseada em seu padrão de movimento.

Este procedimento pode ser feito em diversos níveis de complexidade e refinamento. Para certas aplicações, é importante que o processamento possa ser realizado em tempo real, isto é, há um tempo relativamente curto para o processamento de um novo quadro a partir do momento em que ele foi adquirido. É importante levar em consideração o compromisso entre as funcionalidades do algoritmo e sua complexidade.

O algoritmo proposto neste trabalho visa o rastreamento de pessoas em vídeos de monitoramento de segurança. Para detecção de objetos de interesse em um quadro, é utilizado um extrator

de características chamado SURF (*Speeded-Up Robust Features*), e para localização destes objetos a cada quadro, é feita uma busca multidimensional. Pelo fato de ambos os algoritmos serem complexos computacionalmente, a ideia básica deste trabalho é setorizar a região de busca para que a quantidade de pixels a serem processados por quadro seja menor do que a imagem inteira.

O restante do relatório está organizado da seguinte forma: as seções 2 e 3 detalham melhor os procedimentos de extração de características e rastreamento utilizados e traçam rapidamente um paralelo entre outras técnicas. A seção 4 mostra detalhes de implementação, incluindo a estrutura do algoritmo e o ambiente computacional utilizado para programação. A seguir são descritos os resultados obtidos, elaboradas algumas considerações finais sobre os resultados e os principais problemas encontrados.

2 Extração de características

A extração de características de uma imagem busca detectar pontos ou formas de interesse que possam ser utilizados para rastreamento ou reconhecimento. Existe um grande número de propostas na literatura para este fim, baseando-se em características da imagem como cor, textura, bordas ou cantos. Um detector tem como principal característica a repetibilidade, isto é, é necessário que detecte pontos semelhantes em imagens distintas. Adicionalmente, um detector pode ser invariante a escalamento e rotação.

Após a detecção destes pontos, é necessário associar a eles descritores, para que se possa identificá-los. O tipo de descritor depende muito do formato de características extraído. Descritores precisam ser robustos o bastante, para que pontos similares apresentem descritores similares. Também é necessário que pontos muito diferentes apresentem descritores também diferentes.

2.1 Alguns exemplos de características

Borda o detector Canny identifica pontos de interesse em bordas da imagem. A imagem original é filtrada com um filtro Gaussiano para redução de ruído. A seguir, são calculadas a magnitude e o ângulo dos gradientes da imagem. Então, são suprimidos os valores não-máximos dos gradientes, de modo que restem somente as bordas afinadas, que são analisadas e conectadas [2]. Este algoritmo é somente um detector, sendo necessário algum método para geração de descritores a partir das bordas detectadas. A figura 1 mostra um exemplo de imagem processada com esse detector.

Cantos/curvaturas alguns detectores/descritores, como Harris, detectam locais com alta curvatura, que podem ou não ser cantos de objetos. A figura 2 mostra marcações de pontos detectados com um detector Harris sobre a imagem original.

2.2 SURF - Speeded Up Robust Features

O algoritmo utilizado neste trabalho para extração de características foi o SURF [3]. Este algoritmo é invariante a escalamento e rotação, e utiliza-se de alguns recursos para redução de complexidade computacional. Esta seção dá uma ideia geral de seu funcionamento, para maior detalhamento recomenda-se a leitura do artigo original.

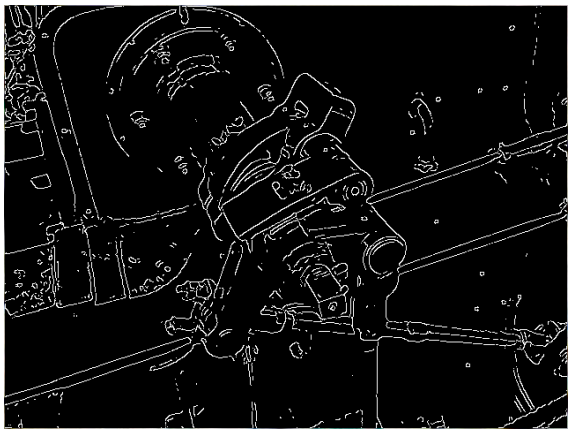


Figura 1: Imagem processada com o detector de bordas Canny

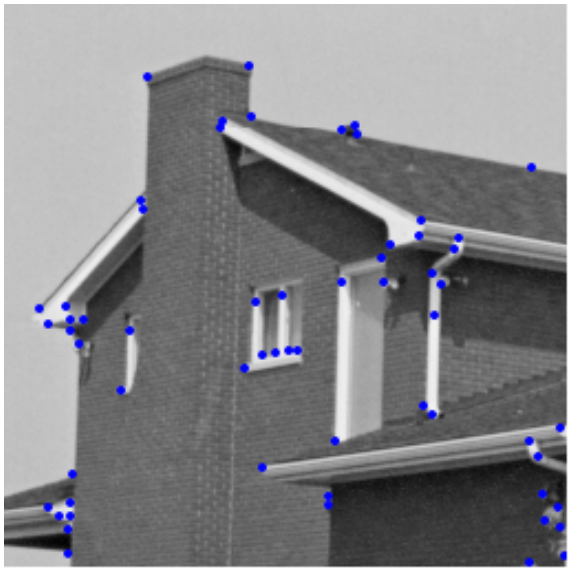
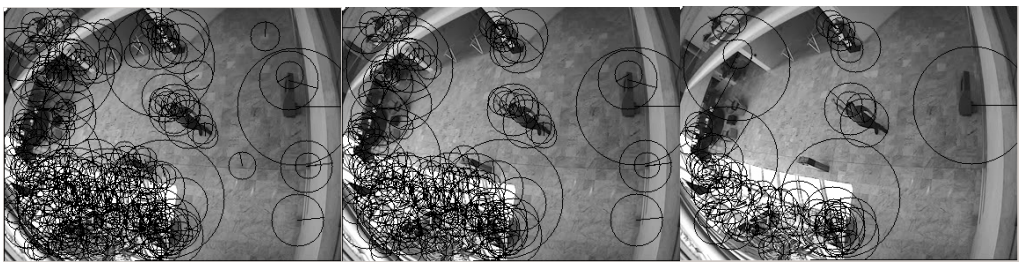


Figura 2: Pontos de interesse encontrados pelo detector Harris



(a) (b) (c)

Figura 3: Pontos de interesse com determinante maior que 500, 1000 e 3000 (da esquerda para a direita)

O detector de pontos de interesse do SURF é baseado em uma aproximação da matriz Hessiana, calculada para cada ponto da imagem. Os pontos de interesse encontrados são estruturas do tipo *blob*, que ocorrem nas regiões onde o determinante da matriz é máximo. O cálculo desta matriz é baseado na convolução da imagem em escala de cinza com a derivada parcial de segunda ordem de uma Gaussiana truncada. É possível definir um nível de detecção para o algoritmo baseado em um valor mínimo do determinante, o que resulta na diminuição da quantidade de pontos encontrados. No entanto, percebe-se que pontos com o determinante mais elevado são mais robustos. A figura 3 mostra, respectivamente, a detecção de pontos com determinante maior que 500, 1000 e 3000.

A detecção de pontos se dá em diferentes escalas. Para evitar os problemas associados ao *downsampling*, ao invés de reduzir a imagem iterativamente, a máscara do filtro é aumentada. O espaço de escalamentos é dividido em oitavas, cada uma representando a convolução da saída do filtro inicial com filtros aumentados em escalas intermediárias. Para o algoritmo implementado, foram usadas 3 oitavas. Na figura 3, o tamanho dos círculos representa a escala onde a característica foi detectada, e o eixo representa a direção.

Os descritores são baseados na soma das respostas wavelet de Haar. Fundamentalmente, os descritores representam a distribuição espacial dos gradientes na região de interesse. Cada região é dividida em subregiões 4×4, e para cada divisão da subregião são calculadas respostas wavelet Haar na horizontal e na vertical. Os descritores resultantes são a soma das respostas e de seus módulos na horizontal e na vertical $(\sum dx, \sum dy, \sum |dx|, \sum |dy|)$. Desta forma, cada descritor é um vetor com 64 coeficientes. Uma variação do algoritmo possibilita obter descritores com 128 coeficientes, que é mais distintivo mas por ter dimensões maiores tornaria o processo de busca mais lento.

3 Rastreamento de objetos

O rastreamento de objetos visa determinar a posição e o deslocamento de objetos em uma sequência de quadros. Para rastreamento de movimento, existem métodos baseados em busca de descritores e métodos baseados em fluxo óptico.

Métodos baseados em fluxo óptico são capazes de identificar regiões em movimento, sem detectar objetos propriamente ditos. Este fato evita a necessidade das operações de busca multidimensional em um espaço de descritores. O algoritmo de Lucas-Kanade [4], um método deste tipo, considera as propriedades de brilho constante, persistência temporal e coerência espacial e, com isso, faz uma estimativa de vetores de velocidade a partir do movimento de pixels em regiões

selecionadas.

Os algoritmos baseados em busca de descritores determinam um modelo para o objeto a ser rastreado baseado nos descritores que podem ser extraídos de uma imagem inicial. A partir deste modelo, é possível fazer buscas por descritores próximos aos do modelo nos descritores encontrados em quadros subsequentes. Caso o descritor seja suficientemente robusto e a imagem não tenha sofrido alterações bruscas, há grande probabilidade de o conjunto de descritores mais próximo, encontrado em um outro quadro, ainda represente o mesmo objeto.

O rastreamento baseado em descritores pode apresentar grande complexidade computacional caso seja necessário extrair descritores e fazer uma busca em um espaço multidimensional com muitos pontos para cada quadro. Por conta disso, os métodos em geral apresentam alguma maneira de modelar a trajetória do objeto rastreado e reduzir a área da imagem na qual a busca deve ser realizada [5]. Esta foi a estratégia adotada para o algoritmo implementado.

4 Implementação

Esta seção discute algumas características da aplicação implementada e escolhas de projeto.

4.1 Ambiente de programação

Como ambiente de programação para desenvolvimento rápido, foi utilizada a linguagem de programação Python [6]. Esta é uma linguagem de alto nível multiparadigma, permitindo o uso de conceitos como programação orientada a objetos e programação funcional.

Por ser uma linguagem interpretada, seu desempenho não é o ideal para processamento de sinais, especialmente para vídeo. Para estes casos, a linguagem conta com uma interface para bibliotecas escritas em código nativo de uma arquitetura, como C, C++ e FORTRAN, possibilitando unir as facilidades de uma linguagem de alto nível interpretada (não ser necessário compilar o programa a cada alteração, por exemplo) ao alto desempenho de código nativo.

Para operações numéricas matriciais foi utilizada uma biblioteca para computação científica desenvolvida para a linguagem Python, chamada NumPy [7].

Para processamento de imagens, foi utilizada a biblioteca OpenCV [8], em sua versão 2.1. Esta biblioteca conta com grande variedade de funções para implementação de aplicações para visão computacional em tempo real. Neste projeto, foram utilizadas basicamente funções para leitura de imagens e vídeos, operações matriciais e extração de características SURF.

Além destas, foi utilizada a biblioteca FLANN (*Fast Library for Approximate Nearest Neighbors*) para busca por pontos vizinhos em espaços multidimensionais [9].

Tanto o interpretador para a linguagem Python quanto as bibliotecas e ferramentas utilizadas são publicados sob licenças livres e estão disponíveis na página dos respectivos projetos.

4.2 Premissas utilizadas na implementação

Para redução da complexidade computacional do algoritmo, visando sua execução em tempo real, algumas premissas foram adotadas. Além disso, algumas escolhas foram feitas para que o algoritmo possa ser utilizado em situações gerais.

- O objeto a ser rastreado encontra-se em uma região geométrica regular pré-determinada (na implementação, um quadrado). Somente descritores encontrados nesta região serão observados.
- Objeto move-se com velocidade relativamente baixa, de modo que seu deslocamento não faz com que ele saia completamente da região onde estava no quadro anterior.
- Não há um modelo do objeto definido previamente. Este modelo é extraído de um quadro da própria sequência.

4.3 Estrutura geral do algoritmo

O programa opera de acordo com a seguinte sequência de passos:

1. Recebe do usuário o limiar mínimo do determinante da matriz Hessiana e a sequência a ser processada.
2. Extrai as características do quadro inicial e mostra o quadro com marcação de características para o usuário.
3. Recebe do usuário a região inicial a ser rastreada (coordenadas da origem do quadrado e tamanho em pixels) e gera uma máscara binária correspondente (figura 4).
4. Para cada quadro subsequente:
   (a) Extrai as características da seção demarcada pela máscara ampliada de 10% do quadro.
   (b) Caso seja o primeiro quadro, gera o índice para busca. Caso contrário, calcula as distâncias entre os pontos encontrados e os pontos indexados.
   (c) Caso não seja o primeiro quadro, remove os vizinhos mais distantes da lista de vizinhos, isto é, a lista de vizinhos fica com um único vizinho para cada ponto do quadro original.
   (d) Calcula o centróide dos pontos na lista de vizinhos e coloca este ponto como centróide da máscara da região rastreada.
   (e) Marca a região rastreada no quadro e salva o quadro na sequência processada.
5. Exibe a sequência processada (figura 5).

5 Resultados

O algoritmo foi testado através de sequências de imagens de câmeras de segurança, adquiridas no site do projeto CAVIAR [10]. As sequências utilizadas mostram pessoas circulando em um

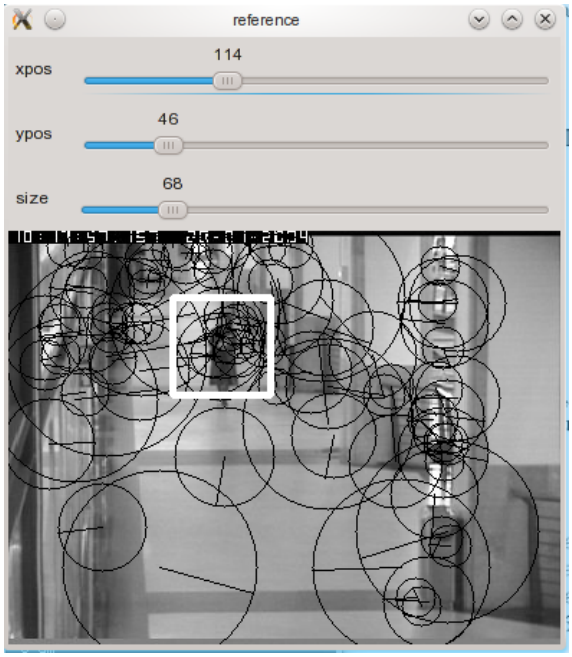


Figura 4: Janela do programa para seleção da região de interesse

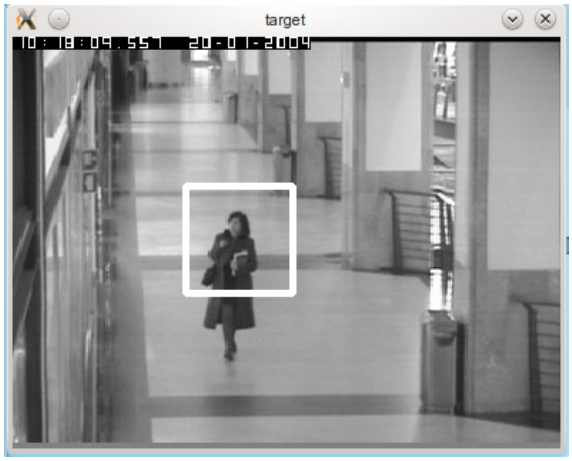


Figura 5: Janela do programa mostrando a sequência processada

corredor e um hall. O objetivo foi verificar se o algoritmo era capaz de rastrear o movimento de uma ou mais pessoas em um quadro até que as pessoas saíssem da cena.

As sequências utilizadas foram "Fight_RunAway", "Walk2", "OneStopEnter1cor" e "TwoEnterShop1cor", sendo os dois primeiros gravados com uma mesma câmera na mesma posição e os dois últimos com outra câmera em outra posição. A resolução dos vídeos é 384×288 pixels, 25 quadros por segundo (padrão PAL). Foram utilizadas as sequências de imagens em JPEG, extraídas dos arquivos MPG previamente para facilitar a visualização e indexação dos quadros fora da execução do programa.

Os resultados em geral foram satisfatórios. São necessários ajustes do valor de limiar do determinante da Hessiana para obter melhores resultados em cada sequência, pois para rastrear pessoas que aparecem em menor escala ou menor contraste com o cenário, é necessário obter pontos com valor de determinante menor. A região rastreada não segue uma trajetória suave, pois não foi utilizado nenhum método de modelagem de trajetória do objeto.

Nas sequências "Fight_RunAway" e "Walk2", ocorreram alguns problemas ao rastrear as pessoas que se aproximam da região iluminada diretamente pela janela. Pelo fato desta região possuir uma distribuição de intensidades dos pixels diferente do modelo original, os descritores extraídos nessa região devem ser bastante diferentes dos originais, o que explicaria o problema obtido.

6 Considerações finais

O rastreamento de objetos em sequências de vídeo utilizando características extraídas com o algoritmo SURF mostrou-se satisfatório para as sequências testadas. Com as premissas assumidas para o algoritmo, foi possível rastrear pessoas em vídeos com resolução baixa. Existe um ruído alto na trajetória encontrada por não ter sido realizado nenhum tipo de estimativa "inteligente" de trajetória.

6.1 Problemas encontrados e possíveis melhorias

Alguns problemas encontrados e possíveis melhorias são os seguintes:

- Ruído na trajetória estimada: como já foi citado, algum tipo de modelagem para estimação de trajetória poderia suavizar o rastreamento obtido.
- Adaptações no modelo: para evitar problemas como o ocorrido nas sequências onde havia uma variação brusca na iluminação, poderia-se tentar adaptar o modelo inicial baseando-se em informações do quadro atual, descartando pontos que estejam com a distância muito elevada e adicionando novos pontos.
- Oclusão: o algoritmo atual não é capaz de tratar oclusão. Uma proposta simples seria executar a extração de características para quadros inteiros, saltando quadros para reduzir a complexidade computacional, até que seja localizado uma nova área onde existam distâncias curtas entre os descritores encontrados e o modelo inicial.

Referências

[1] Alper Yilmaz, Omar Javed, and Mubarak Shah. Object tracking. *ACM Computing Surveys*, 38(4):13–es, December 2006.

[2] Rafael C. Gonzalez and Richard E. Woods. *Digital Image Processing (3rd Edition)*. Prentice Hall, 2007.

[3] Herbert Bay, Andreas Ess, Tinne Tuytelaars, and Luc Van Gool. Speeded-Up Robust Features (SURF). *Computer Vision and Image Understanding*, 110(3):346–359, June 2008.

[4] G.R. Bradski and A. Kaehler. *Learning OpenCV*. O'Reilly, 2008.

[5] Huiyu Zhou, Yuan Yuan, and Chunmei Shi. Object tracking using SIFT features and mean shift. *Computer Vision and Image Understanding*, 113(3):345–352, March 2009.

[6] Python Software Foundation. Python Programming Language. Disponível em http://www.python.org, visitado em julho de 2010.

[7] NumPy. Disponível em http://numpy.scipy.org, visitado em julho de 2010.

[8] OpenCV. Disponível em http://opencv.willowgarage.com/, visitado em julho de 2010.

[9] Marius Muja. FLANN - Fast Library for Approximate Nearest Neighbors. Disponível em http://www.cs.ubc.ca/~mariusm/index.php/FLANN/FLANN, visitado em julho de 2010.

[10] Robert Fisher. CAVIAR Test Case Scenarios. Disponível em http://homepages.inf.ed.ac.uk/rbf/CAVIARDATA1/, visitado em julho de 2010.

A Código-fonte do programa implementado

```
import cv, sys
from pyflann import *
from numpy import *

class TrackedRegion():
    xpos = 0
    ypos = 0
    size = 1
    reference = None

    # Setters added so they can be used as callbacks by HighGUI
    def set_x(self, x):
        self.xpos = x

    def set_y(self, y):
        self.ypos = y

    def set_size(self, s):
        self.size = s

    def get_rectangle(self):
        return (self.xpos, self.ypos, self.size, self.size)

    def get_mask(self):
```

```
        width = self.reference.width
        height = self.reference.height
        mask = zeros([height, width], dtype=int32)
        cvmask = cv.CreateMat(height, width, cv.CV_8UC1)
        mask[self.ypos:self.ypos+self.size,
             self.xpos:self.xpos+self.size] = 255
        cv.Convert(mask, cvmask)
        return cvmask

    def __repr__(self):
        return str(self.get_rectangle())

    def __init__(self, reference, xpos=0, ypos=0, size=1):
        self.reference = reference
        self.xpos = xpos
        self.ypos = ypos
        self.size = size

def draw_features(im, k):
    for x in k:
        pos = x[0]
        fsize = x[2]
        theta = radians(x[3])
        cv.Circle(im, pos, fsize, 0)
        cv.Line(im, pos, (pos[0]+round(fsize*cos(theta)),
                          pos[1]+fsize*sin(theta)),0)

def draw_tracked_region(im, tr):
     cv.Rectangle(im, (tr.xpos, tr.ypos),
                  (tr.xpos+tr.size, tr.ypos+tr.size),
                  255, thickness=3)

def get_tracked_region(im):
    tr = TrackedRegion(im)

    cv.NamedWindow("reference")
    cv.CreateTrackbar('xpos', 'reference',
                      0, im.width-1, tr.set_x)
    cv.CreateTrackbar('ypos', 'reference'
                      , 0, im.height-1, tr.set_y)
    cv.CreateTrackbar('size', 'reference',
                      0, im.width-1, tr.set_size)

    # Show position of descriptors on reference image
    cv.ShowImage("reference", im)

    # Selecting tracked region
    while True:
        key_pressed = cv.WaitKey(100)
        if key_pressed == 32:
            cv.Rectangle(im, (tr.xpos, tr.ypos),
                         (tr.xpos+tr.size, tr.ypos+tr.size),
                         255, thickness=3)
            cv.DestroyWindow("reference")
```

```
            break
        elif key_pressed == 27:
            cv.DestroyAllWindows()
            cv.WaitKey(100)
            return
        else:
            im_copy = cv.CreateMat(im.height, im.width, cv.CV_8UC1)
            cv.Copy(im, im_copy)
            cv.Rectangle(im_copy, (tr.xpos, tr.ypos),
                         (tr.xpos+tr.size, tr.ypos+tr.size),
                         255, thickness=3)
            cv.ShowImage("reference", im_copy)

    return tr

def calc_centroid(keypoints):
    x = [k[0][0] for k in keypoints]
    y = [k[0][1] for k in keypoints]
    n = len(keypoints)
    return (sum(x)/n, sum(y)/n)

def main(argv=None):
    if argv is None:
        argv = sys.argv
    flann = FLANN()

    try:
        sample_dir = argv[1]
        sample_name = argv[2]
        start_frame = int(argv[3])
        end_frame = int(argv[4])
        hessian_threshold = int(argv[5])
    except(IndexError):
        print "Argument error\n Usage: python tracker.py sample_dir\
sample_filename start_frame end_frame hessian_threshold"
        return

    # Loading first frame
    im = cv.LoadImageM(sample_dir + sample_name +
                       str(start_frame) + '.jpg',
                       cv.CV_LOAD_IMAGE_GRAYSCALE)

    # Extracting SURF descriptors from reference image
    # to make selecting the tracked region easier
    (k, d) = cv.ExtractSURF(im, None, cv.CreateMemStorage(),
                            (0, hessian_threshold, 3, 1))

    draw_features(im, k)

    frames = [im]

    # Create window and controls
    tr = get_tracked_region(im)
```

```
# Extracting descriptors from each target image and
# calculating the distances to the nearest neighbors
# Tracked region must be updated in each step
for x in range(start_frame,end_frame):
    im2 = cv.LoadImageM(sample_dir + sample_name +str(x)+'.jpg',
                        cv.CV_LOAD_IMAGE_GRAYSCALE)
    # Creating mask for extracting features from new image
    mask = tr.get_mask()
    (k2, d2) = cv.ExtractSURF(im2, mask, cv.CreateMemStorage(),
                              (0, hessian_threshold, 3, 1))
    d2 = array(d2, dtype=float32)
    result = None
    dists = None
    if x == start_frame:
        params = flann.build_index(d2, target_precision=0.9)
        tr.size = 1.1*tr.size
    else:
        if len(d2) > 0:
            result, dists = flann.nn_index(
                d2, 1,checks=params['checks'])
            # Creating full neighbor table
            neighbors = []
            nearest_neighbors = []
            for n in range(len(d2)):
                neighbors.append((d2[n], k2[n],
                                  result[n], dists[n]))
            # Removing not-nearest neighbors
            for k in range(len(d2)):
                k_neighbors = filter(
                    lambda x: x[2] == k, neighbors)
                num_neighbors = len(k_neighbors)
                if num_neighbors >= 1:
                    if num_neighbors > 1:
                        nearest = reduce(
                            lambda x,y: x if x[3] < y[3] else y,
                            k_neighbors)
                    else:
                        nearest = k_neighbors[0]
                    nearest_neighbors.append(nearest)
            nearest_keypoints =  [kp[1] for kp in nearest_neighbors]
            if len(nearest_keypoints) > 0:
                centroid = calc_centroid(nearest_keypoints)
                tr.xpos = centroid[0] - tr.size/2
                tr.ypos = centroid[1] - tr.size/2
            draw_tracked_region(im2, tr)
            #print "Frame %d had %d feature(s)!" % (x, len(d2))
        else:
            #print "Frame %d had no features!" % x
            pass

    frames.append((im2, k2, d2, result, dists))

cv.NamedWindow("target")
```

```
    esc_pressed = False

    while True:
        for frame in frames:
            cv.ShowImage("target", frame[0])
            if cv.WaitKey(50) == 27:
                esc_pressed = True
                break
        if esc_pressed:
            cv.DestroyAllWindows()
            break
    return frames, params

if __name__ == "__main__":
    main()
```