Detecting a Square in an Image
Find pose of a Square

• First find the square in the image
• Then compute its pose from the corner points

• Squares are frequently used as fiducial targets

“ARToolkit”, www.hitl.washington.edu/artoolkit
Example video http://www.youtube.com/watch?v=5M-oAmBDcZk
A simple method

- Get the boundary of a connected component
- Find the point that is furthest from the centroid – assume that is a corner
- Then find the point furthest from that detected corner – assume that is the opposite corner
- Then find the points that are furthest from the line connecting the two opposite corners
- Assume that the square is not too large and not too small
• Threshold image
• Find connected components

clear all
close all

I = imread('square.jpg');
I = rgb2gray(I);
BW = im2bw(I, graythresh(I));
BW = ~BW;  % invert the image so we look for a white square
imshow(I,[]);

% Find connected components
[L,nBlobs] = bwlabel(BW);
blobs = regionprops(L);
% Look at the blobs and find the first one that looks like a square.
FoudSquare = false;   % This will be set to true if a square is found
for i = 1:nBlobs
    if blobs(i).Area > 40000 || blobs(i).Area < 2000
        continue;
    end

    % Find a point on the boundary of this blob
    [rows, cols] = find(L==i);
    [r0, i0] = min(rows);
    c0 = cols(i0);

    % Get coordinates (row, col) along boundary
    pts = bwtraceboundary(BW, [r0 c0], 'N');

    N = size(pts,1);    % Number of points along the boundary
    c = mean(pts);      % Get centroid

    % Find the point furthest from centroid
    dp = pts - repmat(c,N,1);
    d = dp(:,1).^2 + dp(:,2).^2;
    [~, i1] = max(d);    % Assume that this is a corner
    p1 = pts(i1,:);

    % For each blob, get its boundary
    % Find point furthest from centroid
• Find point furthest from first corner
• Find the points that are furthest from the line connecting the two opposite corners

% Get vectors from the first corner to all other points
r = pts - repmat(p1,N,1);

% Find the point furthest from first corner
d = r(:,1).^2 + r(:,2).^2;
 [~,i3] = max(d); % Assume that this is the opposite corner
p3 = pts(i3,:);

% Find the points that are the furthest from the line from i1 to i3.
v = [p3(2)-p1(2); -(p3(1)-p1(1))]; % A vector perpendicular to that line

% The signed distance from each point p to the line is just dot(v,r)
d = v(1)*r(:,1) + v(2)*r(:,2);
 [~,i2] = max(d); % Point 2 is to the right of 1->3
p2 = pts(i2,:);
~,i4] = min(d); % Point 3 is to the left of 1->3
p4 = pts(i4,:);
% So the order of point is 1,2,3,4 in counterclockwise order
• As partial verification, check side lengths
• Break out of loop if a square is found

% Verify that this is a square - there are many ways to do this.
% We will just check to see if the sides are all about the same length.
s12 = norm(p1-p2);
s23 = norm(p2-p3);
s34 = norm(p3-p4);
s41 = norm(p4-p1);
smax = max([s12,s23,s34,s41]);
sthresh = 0.7;    % minimum ratio of lengths
if (s12/smax > sthresh) && (s23/smax > sthresh) && ...
         (s34/smax > sthresh) && (s41/smax > sthresh)
    fFoundSquare = true;
    break;
end
end
Choose the first point to be the one closest to the top left
Draw a colored marker on each corner

```matlab
if fFoundSquare
    c = [p1; p2; p3; p4];

    % Arbitrarily choose the first point to be the point closest to top left
    d = c(:,1).^2 + c(:,2).^2;
    [~,i0] = min(d);
    for i=1:4
        i1 = i0+(i-1);
        if i1>4  i1=i1-4;  end
        corners(i,:) = c(i1, :);
    end

    mycolors = ['r', 'g', 'b', 'y'];
    hold on
    for i=1:4
        plot(corners(i,2), corners(i,1), 's', 'Color', mycolors(i));
    end
else
    fprintf('Can't find square\n');
end
```
Try this

• See if you can detect the square in each image of the video “square.wmv”

• Recall – to read and process video images

```matlab
movieObj = VideoReader('square.wmv'); % open file
images = read(movieObj); % get all images
nImg = size(images,4); % Number of images read
fprintf('Read in %d images from video files\n', nImg);

for iImg=1:nImg
    I = images(:,:,:,iImg); % Get next image
    :
```
Try this

• Compute the pose of the square and display its coordinate axes
  – Assume the square is 2.4 inches on a side
  – Assume the camera has intrinsic parameters:
    • focal length = 528 pixels
    • center (x,y) = (320,240)
  – You can use the pose that was found in the previous image as the initial guess for the pose in the current image
clear all
close all

% These are the points in the model's coordinate system (inches)
P_M = [
    0     0      2.4     2.4  ;
    0     2.4     2.4     0  ;
    0     0       0       0  ;
    1     1       1       1   ];

% Define camera parameters
f = 528;      % focal length in pixels
cx = 320;
cy = 240;

K = [ f 0 cx; 0 f cy; 0 0 1 ];   % intrinsic parameter matrix

% Make an initial guess of the pose [ax ay az tx ty tz]
x = [0; 0.0; 0.0; 0; 0; 20];

movieObj = VideoReader('square.wmv'); % open file
images = read(movieObj);    % get all images
nImg = size(images,4);      % Number of images read
fprintf('Read in %d images from video files
', nImg);

for iImg=1:nImg
    I = images(:,:,:,iImg);     % Get next image
    I = rgb2gray(I);
    BW = im2bw(I, graythresh(I));
    BW = ~BW;       % invert the image so we look for a white square
    imshow(I,

    % Find connected components
    [L,nBlobs] = bwlabel(BW);
    blobs = regionprops(L);
% Look at the blobs and find the first one that looks like a square.
fFoundSquare = false;   % This will be set to true if a square is found
for i=1:nBlobs
    if blobs(i).Area > 40000 || blobs(i).Area < 2000
        continue;
    end

    % Find a point on the boundary of this blob
    [rows,cols] = find(L==i);
    [r0,i0] = min(rows);
    c0 = cols(i0);

    % Get coordinates (row,col) along boundary
    pts = bwtraceboundary(BW, [r0 c0], 'N');

    N = size(pts,1);    % Number of points along the boundary
    c = mean(pts);      % Get centroid

    % Find the point furthest from centroid
    dp = pts - repmat(c,N,1);
    d = dp(:,1).^2 + dp(:,2).^2;
    [~,i1] = max(d);    % Assume that this is a corner
    p1 = pts(i1,:);

    % Get vectors from the first corner to all other points
    r = pts - repmat(p1,N,1);

    % Find the point furthest from first corner
    d = r(:,1).^2 + r(:,2).^2;
    [~,i3] = max(d);    % Assume that this is the opposite corner
    p3 = pts(i3,:);

    % Find the points that are the furthest from the line from i1 to i3.
    v = [p3(2)-p1(2); -(p3(1)-p1(1))];  % A vector perpendicular to that line

    % The signed distance from each point p to the line is just dot(v,r)
    d = v(1)*r(:,1) + v(2)*r(:,2);
    [~,i2] = max(d);    % Point 2 is to the right of 1->3
    p2 = pts(i2,:);
    [~,i4] = min(d);    % Point 3 is to the left of 1->3
    p4 = pts(i4,:);
% So the order of point is 1,2,3,4 in counterclockwise order

Complete code (2 of 5)
% Verify that this is a square - there are many ways to do this.
% We will just check to see if the sides are all about the same length.
s12 = norm(p1-p2);
s23 = norm(p2-p3);
s34 = norm(p3-p4);
s41 = norm(p4-p1);
smax = max([s12,s23,s34,s41]);
sthresh = 0.7;      % minimum ratio of lengths
if (s12/smax > sthresh) && (s23/smax > sthresh) && ...
   (s34/smax > sthresh) && (s41/smax > sthresh)
   fFoundSquare = true;
   break;
end
if fFoundSquare
    c = [p1; p2; p3; p4];
    % Arbitrarily choose the first point to be the point closest to top left
    d = c(:,1).^2 + c(:,2).^2;
    [~,i0] = min(d);
    for i=1:4
        il = i0+(i-1);
        if il>4  il=il-4;  end
        corners(i,:) = c(il, :);
    end
    mycolors = ['r', 'g', 'b', 'y'];
    hold on
    for i=1:4
        plot(corners(i,2), corners(i,1), 's', 'Color', mycolors(i));
    end
else
    fprintf('Can''t find square
');
    continue;
end
\[ y_0 = [ \]
\[
corners(1,2); \quad \% \text{point 1, col (x)}
corners(1,1); \quad \% \text{point 1, row (y)}
corners(2,2); \quad \% \text{point 2, col (x)}
corners(2,1); \quad \% \text{point 2, row (y)}
corners(3,2); \quad \% \text{point 3, col (x)}
corners(3,1); \quad \% \text{point 3, row (y)}
corners(4,2); \quad \% \text{point 4, col (x)}
corners(4,1); \quad \% \text{point 4, row (y)}
\]

%%%%%%%%%%%%%%%%%%%%%%%
% Find pose

for i=1:10
    % Get predicted image points
    y = fProject(x, P_M, K);
    
    % Estimate Jacobian
    e = 0.00001; \quad \% \text{a tiny number}
    J(:,1) = ( fProject(x+[e;0;0;0;0],P_M,K) - y )/e;
    J(:,2) = ( fProject(x+[0;e;0;0;0],P_M,K) - y )/e;
    J(:,3) = ( fProject(x+[0;0;e;0;0],P_M,K) - y )/e;
    J(:,4) = ( fProject(x+[0;0;0;e;0],P_M,K) - y )/e;
    J(:,5) = ( fProject(x+[0;0;0;0;e],P_M,K) - y )/e;
    J(:,6) = ( fProject(x+[0;0;0;0;0],P_M,K) - y )/e;

    % Error is observed image points - predicted image points
    dy = y0 - y;
    
    % Ok, now we have a system of linear equations \quad dy = J \ dx
    % Solve for dx using the pseudo inverse
    dx = pinv(J) * dy;
    
    % Stop if parameters are no longer changing
    if abs( norm(dx)/norm(x) ) < 1e-6
        break;
    end
    
    x = x + dx; \quad \% Update pose estimate
end
u0 = fProject(x, [0;0;0;1], K); % origin
uX = fProject(x, [1;0;0;1], K); % unit X vector
uY = fProject(x, [0;1;0;1], K); % unit Y vector
uZ = fProject(x, [0;0;1;1], K); % unit Z vector

line([u0(1) uX(1)], [u0(2) uX(2)], 'Color', 'r', 'LineWidth', 3);
line([u0(1) uY(1)], [u0(2) uY(2)], 'Color', 'g', 'LineWidth', 3);
line([u0(1) uZ(1)], [u0(2) uZ(2)], 'Color', 'b', 'LineWidth', 3);

% Also print the pose onto the image.
text(30,450,sprintf('ax=%.2f ay=%.2f az=%.2f tx=%.1f ty=%.1f tz=%.1f', ...
    x(1), x(2), x(3), x(4), x(5), x(6)), ...
    'BackgroundColor', 'w', 'FontSize', 15);

pause(0.1);
end