Occlusion Filling – Reverse Projection

By: Wen De Zhou
Stu#: 1001713274

Introduction

Occlusion filling, or X-ray vison in fictional term is an interesting project idea to me because it presents us with some very helpful applications that most of us may not even find useful until we have such features. For instance, no one felt the need of a larger smartphone display until we got a smartphone with a larger display. Similarly I think occlusion filling can greatly improve our daily lives by giving us the ability to fill important objects that has been occluded by other unimportant objects when we look around every day.

A great example of this is to look into the near distant future where everyone is wearing some type of vision assistant/optical display (i.e. Google glass or even contacts), and since we are in the information age everything will be connected to the cloud. This allows user to stream what they see to the server, and query their search problems to the server and get back a result in real time. Assume a student named Bob is studying in University of Toronto, and he was looking at the slides in class and someone just blocked his view. The occlusion filling algorithm could simply send a query to the server and retrieve an image moment before his vision was blocked and use that image to fill the blackboard that has being occluded now. (Of course he can just download the slides, but assuming he did not manually do so before class.) Another application is that it can also be used as an approximation algorithm for computer vision to approximate the class of the object that has been occluded.

I chose to pick this project idea also because it uses many of the areas we have learned in class about computer vision. This involves key feature detection and matching, object class recognition, and homography. In particular, the basic idea I had in mind before starting was to use Deformable Part Model to detect key objects in the database, and use SIFT to find the transformation to project that object onto the object that has been occluded in the current image.

Methods

Myself (Wen De Zhou) and my partner (Hugh Matsubara) has discussed ways to solve the problem of occlusion filling, we have choose different approach in solving this problem. In the end I used the vision approach of object detection and key points matching (which only requires visual information), where Hugh used the approach to project 3D pixel cloud from one image to another (also requires depth information, described in his report).

There are pros and cons for each of our approaches, in particular for my approach:

Pros:

1. Only requires visual information.

2. Can apply occlusion filling for particular class of key objects. (i.e. If you are looking at an important object in in the foreground, and another unimportant object in the background passes by and got occluded by the important object, the unimportant object will not be filled on top of the important object.)

3. Able to detect visual patterns/shadings of a flat object. (i.e. 3D pixel cloud matching cannot.)

Cons:

1. Only works well on object occlusions of flat (or partial flat) surfaces. (I.e. Object with complex Y-axis (depth) information might result in a filling that is tilted or distorted.)

Below will contain MATLAB code along with comment inside the code to explain each individual parts, as well as a paragraph on top of each segment of the code to describe the general procedure of each individual step.

Every lines of MATLAB code below is programmed by me, as me and my partner has completely different approach on this problem. Libraries I used for this problem include DPM, and SIFT they will be cited at the end of this report.

Occlusion Filling Algorithm

1. Get an image with occlusion of object(s) that we wish to fill using this algorithm and call this image the source image, then WOLG (∗ 1) get an image from the database which contains the same object(s) but unconcluded and call this image the data image. (∗ 1) As we can simply compute more image from the database and select the image with the most key point matches.

```
source = imread('../data/project/source3.jpg');
data = imread('../data/project/data3.jpg');
```

2. Use the Deformable Part Model (DPM) to find object(s) of interest. Where DPM is a convolutional neuro network that can detect class of objects using masks of different properties and parameters adjusted using training data. This allows us to find the object(s) of interest given the corresponding object class detector (WOLG we set class car as the object(s) of interest, given that we have the car detector).

```
% Parameters and thresholds.
recompute = 1; % 1 implies to recompute, 0 implies not to recompute.
dpm_resize_factor = 1; % Resize the image bigger to detect smaller objects.
% This is used to modify the threshold given by the dpm algorithm, since
% the original threshold will detect false positive objects.
dpm_model_threshold = 0.4;
dpm_nms_threshold = 0.3;


% Read the test images, where source is the image with occlusion, data is
% the image without occlusion.
source = imread('../data/project/source3.jpg');
data = imread('../data/project/data3.jpg');

% Use DPM to detect the objects (Cars). If it has not yet being detected
% before. Results are stored in the folder /results.
% If recompute == 1 then this algorithm will recompute the DS of DMP.
if (recompute == 1 || exist('results/ds_source.mat', 'file') == 0 || ...
    exist('results/ds_data.mat', 'file') == 0)
    dataCar = getData([], [], 'detector-car');
    model = dataCar.model;
```

```
    % Detect object (Cars) on source image.

    % Resize the data image, it works better for detecting small objects in
    % DPM.
    f = dpm_resize_factor;
    sourcer = imresize(source,f);

    % Detect car objects.
    % You may need to reduce the threshold if you want more detections.
    model.thresh = model.thresh*dpm_model_threshold;
    [ds, bs] = imgdetect(sourcer, model, model.thresh);
    % Non-maximum suppression to eliminate overlapping object detection.
    nms_thresh = dpm_nms_threshold;
    top = nms(ds, nms_thresh);
    if model.type == model_types.Grammar
        bs = [ds(:,1:4) bs];
    end
    if ~isempty(ds)
        % resize back
        ds(:, 1:end-2) = ds(:, 1:end-2)/f;
        bs(:, 1:end-2) = bs(:, 1:end-2)/f;
    end
    ds_source = ds(top, :);

    % Save ds in results folder.
    filename = sprintf('ds_source.mat');
    save(['results/' filename],'ds_source');

    % Detect object (Cars) on data image.

    % Resize the data image, it works better for detecting small objects in
    % DPM.
    f = dpm_resize_factor;
    datar = imresize(data,f);

    % Detect car objects.
    % You may need to reduce the threshold if you want more detections.
    model.thresh = model.thresh*dpm_model_threshold;
    [ds, bs] = imgdetect(datar, model, model.thresh);
    % Non-maximum suppression to eliminate overlapping object detection.
    nms_thresh = dpm_nms_threshold;
    top = nms(ds, nms_thresh);
    if model.type == model_types.Grammar
        bs = [ds(:,1:4) bs];
    end
    if ~isempty(ds)
        % Resize back.
        ds(:, 1:end-2) = ds(:, 1:end-2)/f;
        bs(:, 1:end-2) = bs(:, 1:end-2)/f;
    end
    ds_data = ds(top, :);

    % Save ds in results folder.
    filename = sprintf('ds_data.mat');
    save(['results/' filename],'ds_data');

% The DPM has already been computed, and we can simply load them.
else
    ds_source = importdata('results/ds_source.mat');
    ds_data = importdata('results/ds_data.mat');
end

% Visualize the segmented objects on source and data images.
% Note: the objects detected might be different, due to the Non-maximum
% suppression.
figure;
image(source);
axis image;
axis off;
showboxes(source, ds_source, 'r', 'Car');
figure;
image(data);
axis image;
axis off;
showboxes(data, ds_data, 'r', 'Car');
```

3. We use scale-invariant feature transform (SIFT) to find matching key points between the data image and the source. The general idea of SIFT is to compute the difference of Gaussian at different scales, and pick local extrema as the key points. Then for each of the key points computer a feature vector and normalize it, so that if two feature vectors in different images match, it means we have found one key point pair. Details on what key points prioritize over the other is shown on the comments below in the MATLAB code.

```
% Convert images to greyscale due to the definition of SIFT feature vector.
source_grey = single(rgb2gray(source));
data_grey = single(rgb2gray(data));

% Extract key points and features
[F_source_grey, D_source_grey] = vl_sift(source_grey);
[F_data_grey, D_data_grey] = vl_sift(data_grey);

% Find keypoint matching pairs.
% distances is the the two smallest distance between a point in D_source_grey
% with all other points in D_data_grey. Indices of indices tells which vertex
% in Df is being matched.
[distances, indices] = pdist2(transpose(D_data_grey), transpose(D_source_grey), ...
    'euclidean', 'Smallest',2);
% Creat matches to stores the matching pair, for simplicity.
matches = [];
% threshold is usually 0.8, indicate how similar the most similar match
% compared to the second most similar pair.
thrshold = 0.3;
for i = 1:size(indices, 2)
    % smallest is the most similar match of keypoint i, and seSmallest is
    % the second most similar match.
    smallest = distances(1, i);
    secSmallest = distances(2, i);
    matchVal = smallest/secSmallest;
    if thrshold > matchVal
        matches = [matches; i indices(1, i) matchVal];
    end
end

% matchedPoints1, and matchedPoints1 are pair of matched points pointed
% from matches.
matchedPoints1 = zeros(size(matches, 1), 2);
matchedPoints2 = zeros(size(matches, 1), 2);
for i = 1:size(matches, 1)
    currMatch = matches(i, :, :);
    matchedPoints1(i, :) = [F_source_grey(1, currMatch(1)), F_source_grey(2, currMatch(1))];
    matchedPoints2(i, :) = [F_data_grey(1, currMatch(2)), F_data_grey(2, currMatch(2))];
end

% Show the matched pairs.
figure; ax = axes;
showMatchedFeatures(source,data,matchedPoints1,matchedPoints2, ...
    'montage','Parent', ax)
```

4. Find 4 key points to each object found in data image, such that they are the best to represent the project of the object in the data image to the source image (if any). I made the choice of finding 4 key points due to the shape of the object detected by DPM is rectangle, and can be constructed with 4 points, thus a 4 point detection best represent this model. The 4 key points can be chosen differently, but the best way (among the ones I've thought) is shown on the MATLAB code below, other less efficient ways are also shown, but they are commented out below (with explanations in the comments).

```
% For each object detected in the data image, find 4 key points corners
% such that they are the most precise bounds of the location of the object.
for i = 1:size(ds_data, 1)
    top_left = [0 size(data, 1)];
    top_right = [size(data, 2) size(data, 1)];
    bottom_left = [0 0];
    bottom_right = [size(data, 2) 0];
```

```
top_left_source = top_left;
top_right_source = top_right;
bottom_left_source = bottom_left;
bottom_right_source = bottom_left;
obj_max_y = ds_data(i,4);
obj_min_y = ds_data(i,2);
obj_max_x = ds_data(i,3);
obj_min_x = ds_data(i,1);
top_left_dist = abs(pdist([top_left; obj_min_x obj_max_y]));
top_right_dist = abs(pdist([top_right; obj_max_x obj_max_y]));
bottom_left_dist = abs(pdist([bottom_left; obj_min_x obj_min_y]));
bottom_right_dist = abs(pdist([bottom_right; obj_max_x obj_min_y]));
flag = [0 0 0 0];
%{
% Gen 1.
% This is scrapped for clean reason, in the worst case curr_y-obj_max_y
% can be minimal, but curr_x-obj_max_x could be very big, causing this
% algorithm to select this key point, where as there might be better
% ones out there.
for j = 1:size(matchedPoints2, 1)
    curr_y = matchedPoints2(j , 2);
    curr_x = matchedPoints2(j , 1);
    if ((top_left(2) > curr_y) &&  ...
        (curr_y > obj_max_y) && ...
        (top_left(1) < curr_x) &&  ...
        (curr_x < obj_min_x))
        top_left = matchedPoints2(j, :);
        top_left_source = matchedPoints1(j, :);
    end
    if ((top_right(2) > curr_y) &&  ...
        (curr_y > obj_max_y) && ...
        (top_right(1) > curr_x) &&  ...
        (curr_x > obj_max_x))
        top_right = matchedPoints2(j, :);
        top_right_source = matchedPoints1(j, :);
    end
    if ((bottom_left(2) < curr_y) &&  ...
        (curr_y < obj_min_y) && ...
        (bottom_left(1) < curr_x) &&  ...
        (curr_x < obj_min_x))
        bottom_left = matchedPoints2(j, :);
        bottom_left_source = matchedPoints1(j, :);
    end
    if ((bottom_right(2) < curr_y) &&  ...
        (curr_y < obj_min_y) && ...
        (bottom_right(1) > curr_x) &&  ...
        (curr_x > obj_max_x))
        bottom_right = matchedPoints2(j, :);
        bottom_right_source = matchedPoints1(j,:);
    end
end
%}

%{
% Gen 2.
% This algorithm has the flaw of not being able to find keypoints
% inside the object's bound if that part is not occluded. Thus the
% result is not optimal.
for j = 1:size(matchedPoints2, 1)
    curr_y = matchedPoints2(j , 2);
    curr_x = matchedPoints2(j , 1);
    if ((top_left_dist > abs(pdist([obj_min_x obj_max_y; curr_x curr_y]))) &&  ...
        (curr_y > obj_max_y) && ...
        (curr_x < obj_min_x))
        top_left = matchedPoints2(j, :);
        top_left_source = matchedPoints1(j, :);
        top_left_dist = abs(pdist([obj_min_x obj_max_y; curr_x curr_y]));
        flag(1) = 1;
    end
    if ((top_right_dist > abs(pdist([obj_max_x obj_max_y; curr_x curr_y]))) &&  ...
        (curr_y > obj_max_y) && ...
        (curr_x > obj_max_x))
        top_right = matchedPoints2(j, :);
        top_right_source = matchedPoints1(j, :);
        top_right_dist = abs(pdist([obj_max_x obj_max_y; curr_x curr_y]));
        flag(2) = 1;
    end
    if ((bottom_left_dist > abs(pdist([obj_min_x obj_min_y; curr_x curr_y]))) &&  ...
```

```
        (curr_y < obj_min_y) && ...
        (curr_x < obj_min_x))
        bottom_left = matchedPoints2(j, :);
        bottom_left_source = matchedPoints1(j, :);
        bottom_left_dist = abs(pdist([obj_min_x obj_min_y; curr_x curr_y]));
        flag(3) = 1;
    end
    if ((bottom_right_dist > abs(pdist([obj_max_x obj_min_y; curr_x curr_y]))) &&  ...
        (curr_y < obj_min_y) && ...
        (curr_x > obj_max_x))
        bottom_right = matchedPoints2(j, :);
        bottom_right_source = matchedPoints1(j,:);
        bottom_right_dist = abs(pdist([obj_max_x obj_min_y; curr_x curr_y]));
        flag(4) = 1;
    end
end
%}

% Gen 3.
% Find the keypoints with the closest manhattan distance between the
% corners of the detected object bound.
for j = 1:size(matchedPoints2, 1)
    curr_y = matchedPoints2(j , 2);
    curr_x = matchedPoints2(j , 1);
    if ((top_left_dist > abs(pdist([obj_min_x obj_max_y; curr_x curr_y]))) )
        top_left = matchedPoints2(j, :);
        top_left_source = matchedPoints1(j, :);
        top_left_dist = abs(pdist([obj_min_x obj_max_y; curr_x curr_y]));
        flag(1) = 1;
    end
    if ((top_right_dist > abs(pdist([obj_max_x obj_max_y; curr_x curr_y]))) )
        top_right = matchedPoints2(j, :);
        top_right_source = matchedPoints1(j, :);
        top_right_dist = abs(pdist([obj_max_x obj_max_y; curr_x curr_y]));
        flag(2) = 1;
    end
    if ((bottom_left_dist > abs(pdist([obj_min_x obj_min_y; curr_x curr_y])))  )
        bottom_left = matchedPoints2(j, :);
        bottom_left_source = matchedPoints1(j, :);
        bottom_left_dist = abs(pdist([obj_min_x obj_min_y; curr_x curr_y]));
        flag(3) = 1;
    end
    if ((bottom_right_dist > abs(pdist([obj_max_x obj_min_y; curr_x curr_y])))  )
        bottom_right = matchedPoints2(j, :);
        bottom_right_source = matchedPoints1(j,:);
        bottom_right_dist = abs(pdist([obj_max_x obj_min_y; curr_x curr_y]));
        flag(4) = 1;
    end
end

% Visualize the bound made by the 4 keypoints choosen.
figure; imshow(data)
hold on
line([top_left(1), top_right(1), bottom_right(1), bottom_left(1), top_left(1)], ...
    [top_left(2), top_right(2), bottom_right(2), bottom_left(2), top_left(2)], ...
    'color', 'green', 'LineWidth',3)
hold off
```

5. For each object detected in the data image, use the 4 key point matches found in previous step to compute the projective transformation from the data image to the source image (for each objects). Then project the center of that object to the source image to check if any object already exist in that location. If not, simply apply the projective transformation to each pixel within the object bound in data image to the source image.

```
% Use those 4 key points and find the transformation matrix, from
    % data image to source image.
    % Construct P and Pp.
    if (~(flag(1) == 1 && flag(2) == 1 && flag(3) ==1 && flag(4) ==1))
        continue
    end
    k = 4;
    P = zeros(2*k, 6);
    Pp = zeros(2*k, 1);
```

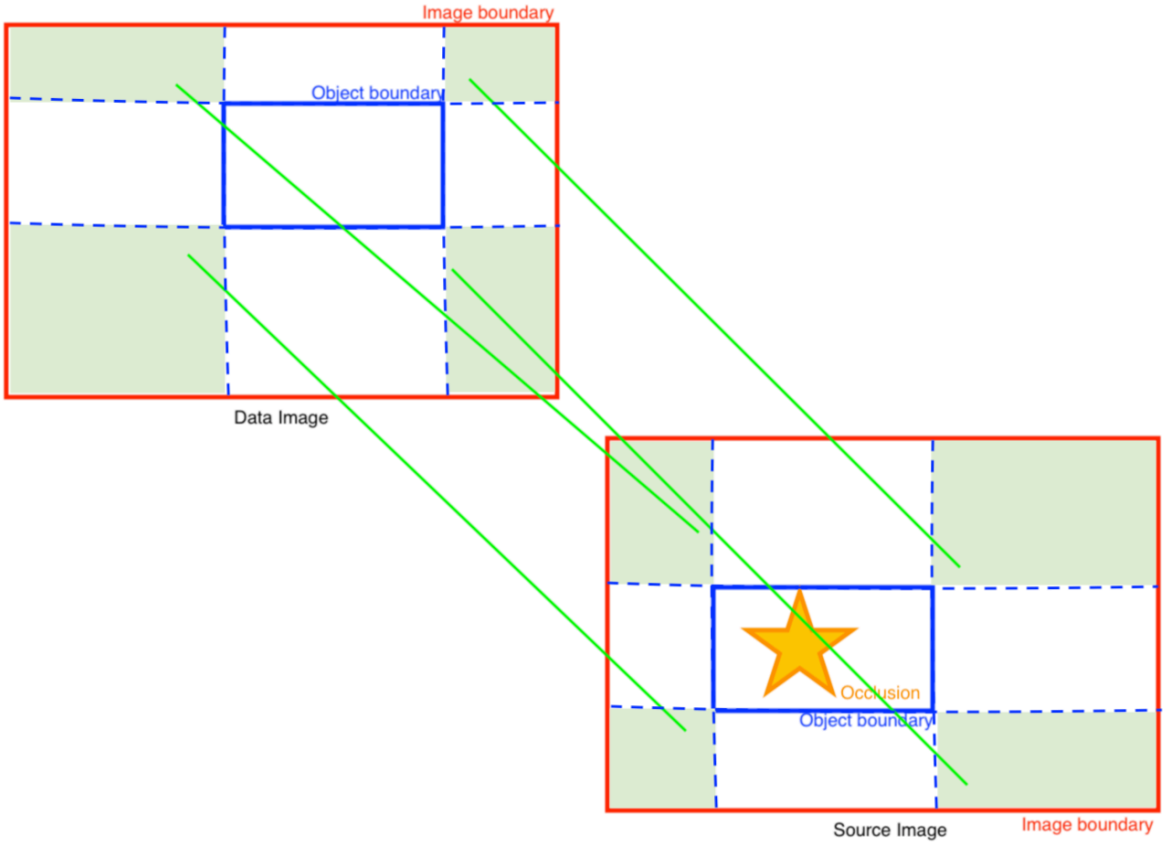
```
P(1, :) = [top_left(1), top_left(2), 1, 0, 0, 0];
P(2, :) = [0, 0, 0,top_left(1), top_left(2), 1];
P(3, :) = [top_right(1), top_right(2), 1, 0, 0, 0];
P(4, :) = [0, 0, 0,top_right(1), top_right(2), 1];
P(5, :) = [bottom_left(1), bottom_left(2), 1, 0, 0, 0];
P(6, :) = [0, 0, 0,bottom_left(1), bottom_left(2), 1];
P(7, :) = [bottom_right(1), bottom_right(2), 1, 0, 0, 0];
P(8, :) = [0, 0, 0,bottom_right(1), bottom_right(2), 1];
Pp(1) = top_left_source(1);
Pp(2) = top_left_source(2);
Pp(3) = top_right_source(1);
Pp(4) = top_right_source(2);
Pp(5) = bottom_left_source(1);
Pp(6) = bottom_left_source(2);
Pp(7) = bottom_right_source(1);
Pp(8) = bottom_right_source(2);
a = (inv(transpose(P)*P))*transpose(P)*Pp;
a = reshape(a, 3,2).';

% Find the centre point of the object, and project it with the
% transformation matrix 'a' to see if the object exist in the source
% image.
obj_centre = [obj_min_x + (obj_max_x + obj_min_x)/2; ...
    obj_min_y + (obj_max_y + obj_min_y)/2; ...
    1];
obj_centre_proj = a*obj_centre;
% Check if obj_centre_proj is within any object boundaries in source
% image.
has_obj_projection = 0;
for ii = 1:size(ds_source, 1)
    obj_max_y = ds_source(ii,4);
    obj_min_y = ds_source(ii,2);
    obj_max_x = ds_source(ii,3);
    obj_min_x = ds_source(ii,1);
    if ((obj_max_x > obj_centre_proj(1)) && ...
        (obj_centre_proj(1) > obj_min_x) && ...
        (obj_max_y > obj_centre_proj(2)) && ...
        (obj_centre_proj(2) > obj_min_y))
        has_obj_projection = 1;
    end
end
% If not it means that there does not exist an object in
% computer vision (meaning the object is occluded), thus we can start
% filling the occlusion using the object in the data image.
if has_obj_projection == 0
    obj_max_y = round(ds_data(i,4));
    obj_min_y = round(ds_data(i,2));
    obj_max_x = round(ds_data(i,3));
    obj_min_x = round(ds_data(i,1));
    for n = obj_min_x:obj_max_x
        for m = obj_min_y:obj_max_y
            pixel = [n; m; 1];
            pixel = round(a*pixel);
            if pixel(2) <= size(source, 1) && pixel(1) <= size(source, 2)
                source(pixel(2), pixel(1), :) = data(m, n, :);
            end
        end
    end
end
```

Main Challenges

The main challenge of this project is to figure out a way to detect occluded object in the source image. As by the definition of occlusion, there are a very high chance of the computer not being able to detect the object that has being occluded, thus to computer vision that object is nonexistent. Thus I had to figure out a way to use the information that can be detected using computer vision to figure out which object that has been occluded.

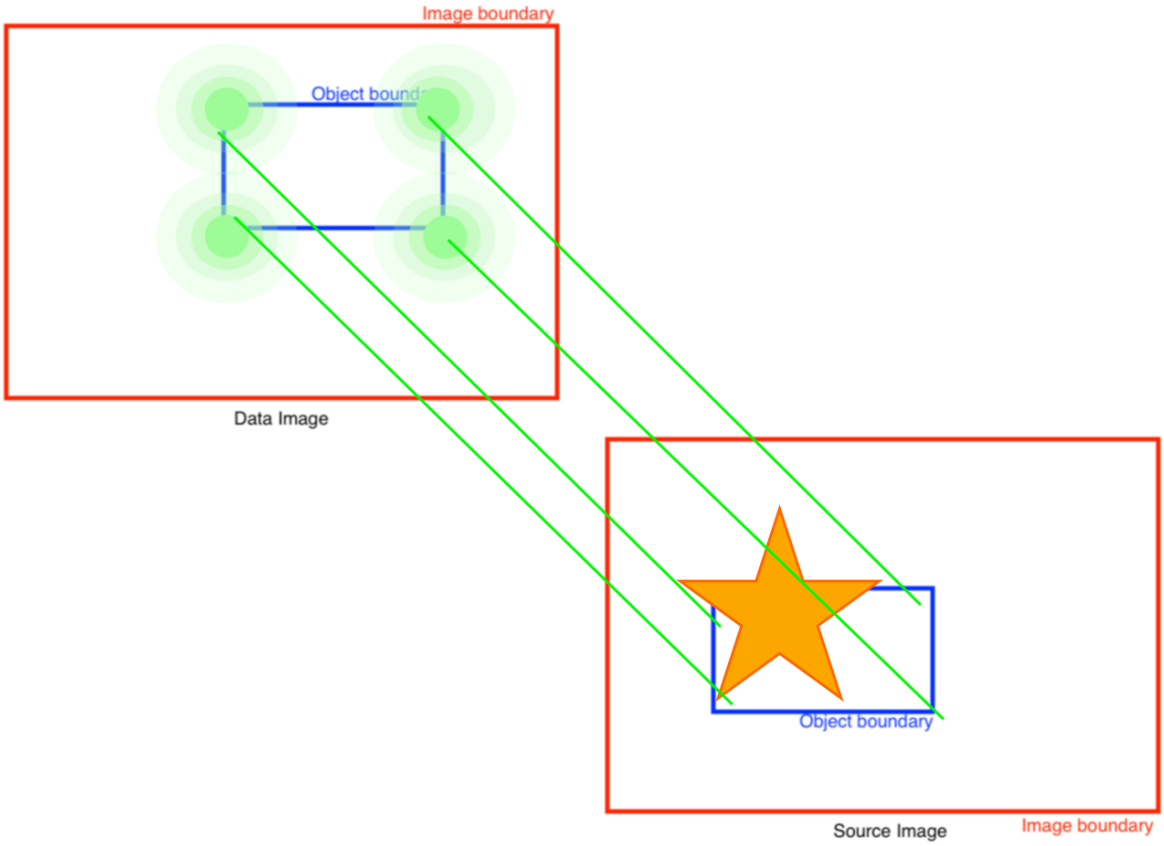
The first thing I had in mind is to use SIFT to find key points to compute the projective transformation that can map one object in data image to the source image. The difficult part is to pick the minimal amount of key point matches to compute the optimal transformation for the detected object, as things besides the object may move in position (between the data and source image), and picking more key point matches has a higher chance of picking one of those key point matches. Picking the top most similar key point match also doesn't work due to this reason. The best solution I have come up with is to compromise picking the most similar key point matches in favor of key point matches that are closer to the corners of the object detected in data image.

However there are still multiple ways to pick key point matches that are close to the corners of the object boundary. The first way I had in mind is to only consider matches that are within the light green area indicated in the figure below. (Shown as labeled Gen 1. and Gen 2. in step 4 of the algorithm in the Method section.)
(Green lines indicate the key point matching pair.)



Since this way we won't get key pointed matches such that they are all saturated in one corner of the object boundary. Also we does not allow any match selection that are within the object boundary, since it might contain occlusion.

After many different modifications and testing I found that the best algorithm is to simply find the key point matches with the lowest manhattan distance to each of the four corners, shown by the figure below. (Shown as labeled Gen 3. in step 4 of the algorithm in the Method section.)



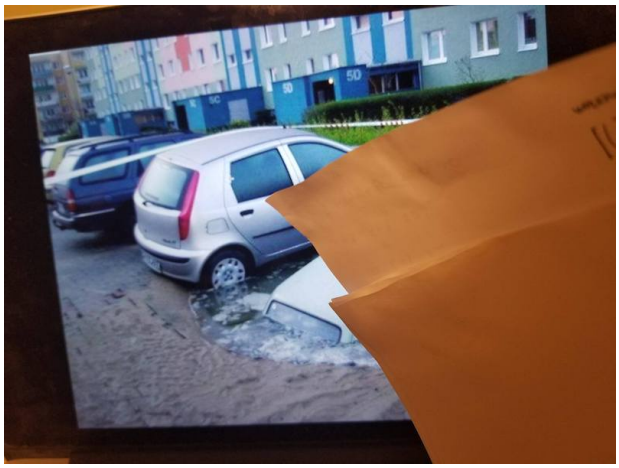
We do not even have to consider the key point matching pair's position, since features of occlusion will not be matched to any points in data image. Additionally allowing the selection of matchings that are within the object boundary also gives the chance of selecting a key point matching of the object, which will give a more accurate projective transformation for the particular object.

Results and Discussion

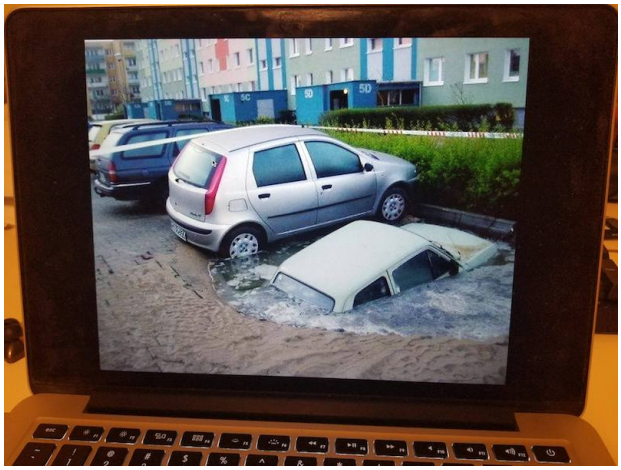
This occlusion filling algorithm works best under some conditions of the environment, while performs slightly worse under some other environments. Below I will first demonstrate and discuss in what conditions does this algorithm performs the best, and what parameter of the algorithms I can tweak to make it perform better. I will start by visualizing the results for all steps listed in the Method section for the optimal case example below, then discuss all the thresholds used in this algorithm, and finally showing an example for the least optimal case.

The Optimal Case

In this case the surface of occlusion for the key object(s) must be flat, and the data image must show a complete unconcluded key object(s). The occluded key object(s) in the source image must show some edges or vertices (as in their object boundary by eyeballing) so that the algorithm can obtain an optimal data to source object projection.

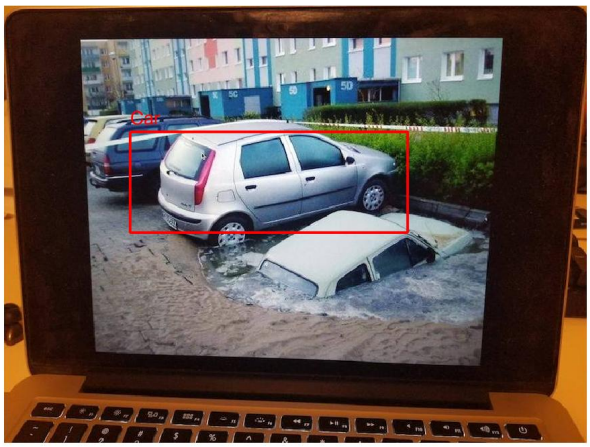


Source Image, before any manipulation in Step 1.



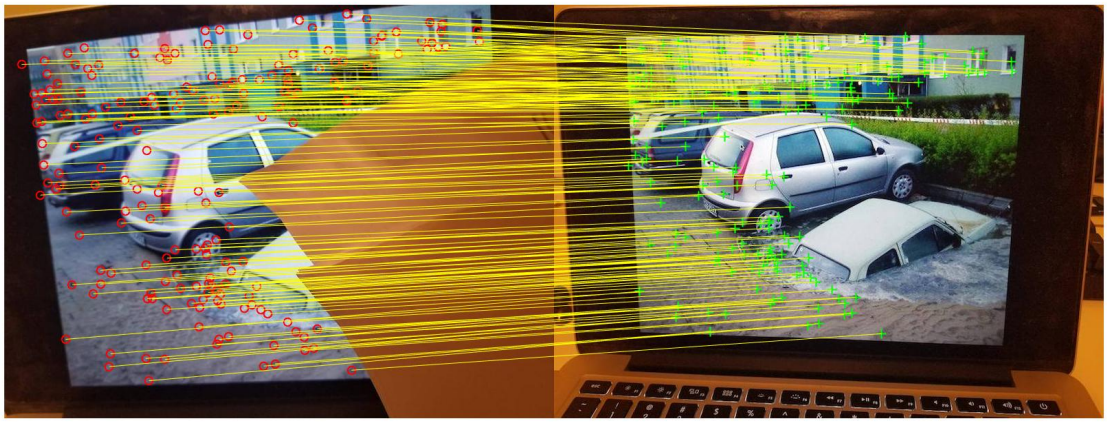
Data Image, before any manipulation in Step 1.

To start we use DPM to segment key objects of class car in both images. The DPM algorithm uses three thresholds, the image resize factor – for detecting smaller objects, the DPM model threshold – a parameter for object recognition, and non-maximum suppression threshold – to eliminate multiple detections on the same object. However since this is the best case, we do not need to tweak any of the parameters (default values shown in the code, and in General Thresholds discussion below).



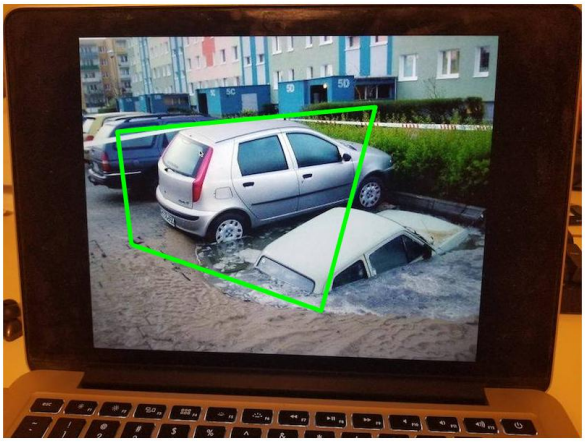
Car object detected with DPM in data image in step 2.

Then we use the SIFT algorithm to detect key point matchings between the two images. SIFT uses one threshold, the matching threshold – used to distinguish how good a matching is. We can again tweak this threshold to produce more matching when the image quality is bad, but we don't have to since in this example our image quality is decent enough.



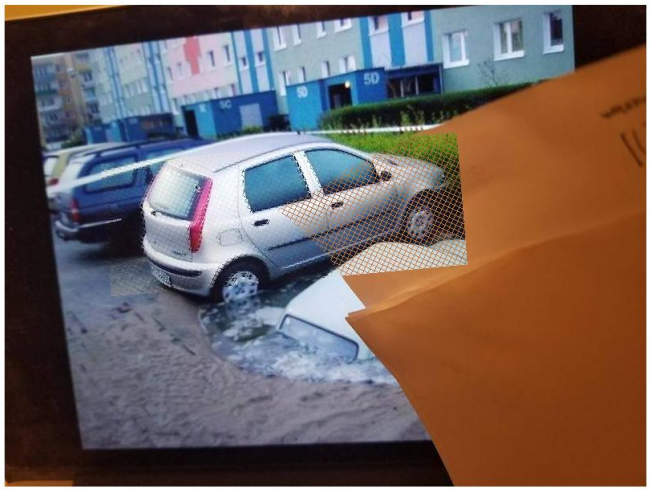
SIFT key point matchings detected in step 3.

After this we use the key point matching pair selection algorithm (details described in the Methods and Main challenges section) to find 4 key point matchings for each object detected in the data image that best describes the projective transformation of that object. In this example there is only one car object detected, so we are going to visualize the 4 key points chosen for that object below.



Visualization of the 4 key point matches selected in step 4. The vertex of the green trapezoid indicate the 4 key points of the matches on the data image respectively.

Finally we use these key point matches to compute the projective transformation matrix, and check if there exist an object on the source image by applying this matrix to the center of the detected object in the data image. In this example we found that there is no object in the place of the projection, implies that the object has been occluded (i.e. object is undetectable by computer vision). Thus we simply apply the projective transformation to each pixel of the object in the data image, to project them onto the source image.



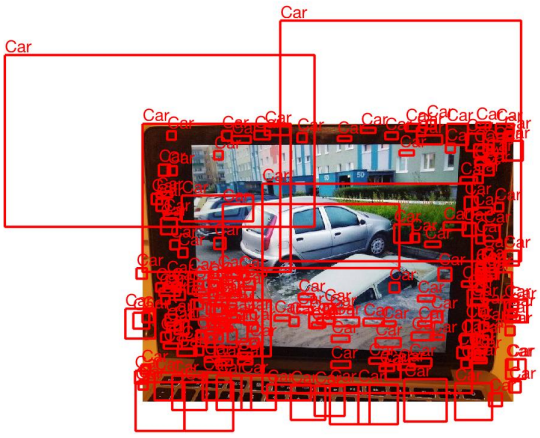
Final result of occlusion filling at the end of step 5.

Note: pixels projected is not smoothened, or have their opacity reduced in order to preserve more information.

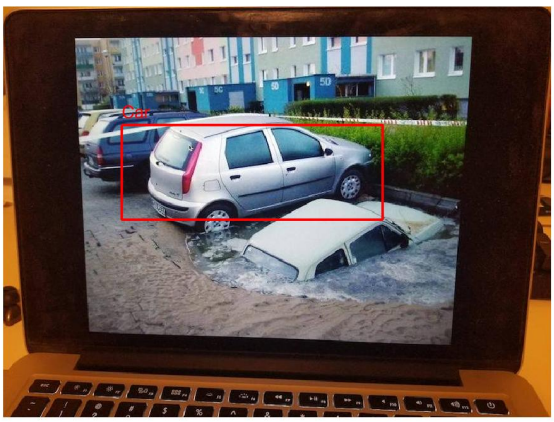
General Thresholds

There are four thresholds in this occlusion filling algorithm. The optimal case above uses the 'general' thresholds, such that they works for work for most test case scenarios. However we can tweak those thresholds for special case scenarios, such as image with small objects or crowded objects.

DPM model threshold is used to distinguish whether or not to classify an detection as an object of that class. The given DMP model threshold for the car detector is way too high, this will result many false positive detections (as shown by figures below), thus we have to scale to 40% of its original value to get a more sensible result.

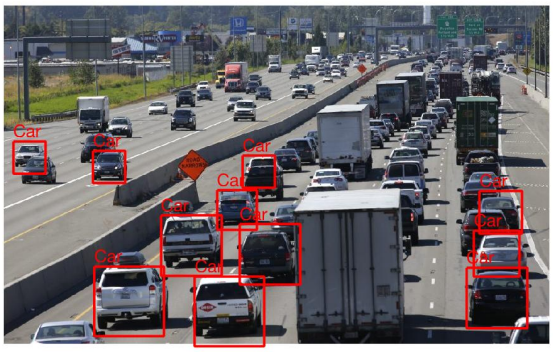


DPM detection with original model threshold.

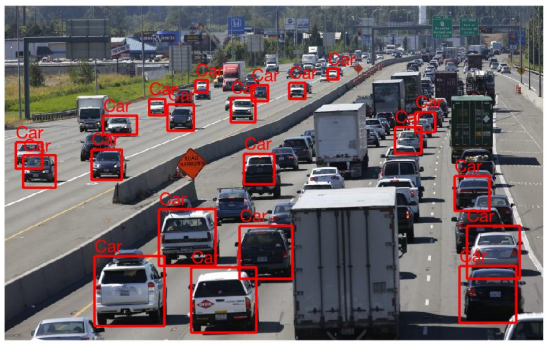


DPM detection with 40% scaled original model threshold.

DPM resize factor is used to resize the image larger so that the DPM algorithm can detect smaller objects. The general resize factor is simply 1, however as for images with smaller objects, it is recommended resize the image by up to 3 times (as shown by figures below).

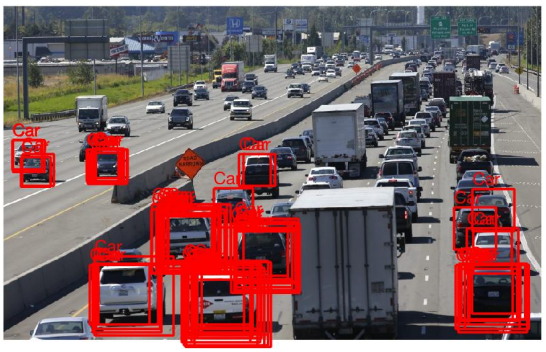


DPM detection with resize factor of 1.



DPM detection with resize factor of 3.

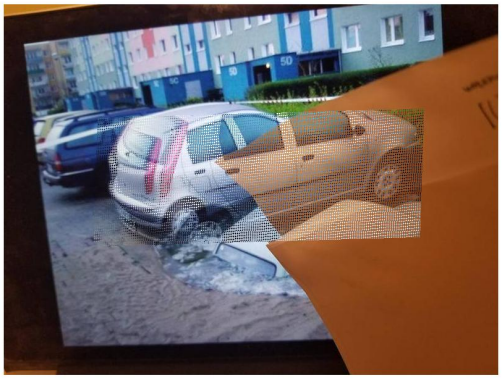
DPM also needs a threshold for Non-maximum suppression, the general NMS threshold is 0.3. In the extreme case if we do not bother with NMS (set threshold to 1), we will get very bad results (as shown by figure below).



DPM detection with no NMS.

The final threshold is the SIFT matching threshold, used as a threshold for the ratio of best matching (lowest distance) and the second best matching (second lowest distance, details described in code). The general SIFT matching threshold is 0.3. Increasing it allows bad matchings to be considered when finding the 4 key matchings used to compute the projective transformation, which will result in bad occlusion filling of the object(s) (as shown by figures below).

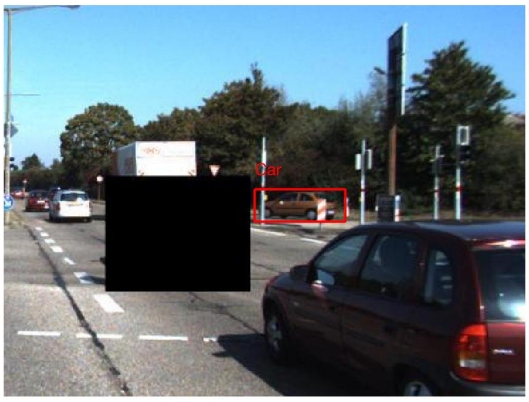
SIFT matchings with matching threshold set to 0.8 .



Occlusion Filling result with SIFT matching threshold of 0.8 .

The Least Optimal Case

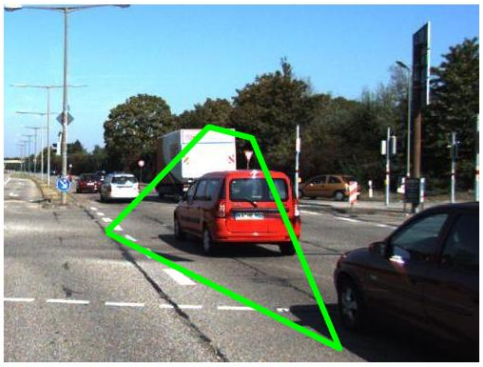
In this case the surface of the object that has been occluded in the source image is not flat, and the whole object has been occluded. This not only makes this algorithm do more guessing based on the data image of where should the occluded object(s) be, it might also cause the result filling to be tilted or distorted due to the complex Y-axis (depth) information of the areas that has been occluded. The example below will demonstrate this case (DPM model threshold is set to 0.5 instead of the general 0.4 as the angle of cars in this image makes it look weird).



Source Image, before any manipulation in Step 1.

Data Image, before any manipulation in Step 1.

SIFT key point matchings detected in step 3.



Visualization of the 4 key point matches selected in step 4. The vertex of the green trapezoid indicate the 4 key points of the matches on the data image respectively.

Final result of occlusion filling at the end of step 5.

Conclusion and Future Work

Occlusion filling is a very important filed in computer vision. It has many applications that are not only useful for filling an occlusion in an image to make it more pleasant to look at, but it can also be used to estimate the class of the occluded object for computer vision. This particular occlusion algorithm requires only visual information, i.e. pictures or videos to fill occluded key objects. This algorithm basically uses DPM to detect all objects of selected key object class, also with the combined key point matches detected by SIFT to compute the estimate projective transformation of objects in data image to the source image. Which allows the detection and projection of occluded objects.

One future improvement for this algorithm is to make it produce better result for filling 3D occluded areas. But this will require stereo images or depth information obtained with non-computer vision methods. A general concept for this extension is that to use DPM use find objects on both source and data images, then use SIFT to computer key point matchings. However instead of using 4 key point matchings to determine the projective transformation, we can use the depth information to select 8 key point matchings that best bound each object in 3D space. Finally we can compute a 3D projection matrix that can map each pixels of the detected object to the source image in 3D space, and transform it back to 2D so that we can fill it on the source image.

Citations

Vedaldi, A., and B. Fulkerson. “VLFeat: An Open and Portable Library of Computer Vision Algorithms.” Http://Www.vlfeat.org/, 2008.
Girshick, et al. “Discriminatively Trained Deformable Part Models, Release 5.” Http://People.cs.uchicago.edu/~Rbg/Latent-release5/, Sept. 2010.