Advanced Vision Assignment 3

Boris Mitrovic, Alexandros Gouvatsos

Introduction

This report presents the system that was developed for the augmented reality task of Advanced Vision. The report is broken down into several sections. In the ‘Methodology’ section, the general workflow of the implemented system is briefly explained followed by an explanation of the individual algorithms used. In the ‘Results’ section, some sample images are presented displaying the individual stages of the final approach, as well as results from different approaches that were tested. Finally, in the ‘Discussion’ section the results are critically evaluated and the different techniques are compared. Assessment of the strengths and weaknesses of the system is also included in that section, followed by some directions for future work. The source code of the system is included as an appendix at the end of the report.

Methodology

General implementation

The system is built in a modular way, differentiating between the different steps. This allows for easy extension of the system and it also simplified the development process as individual components could be updated individually without affecting the performance of the rest of the system [1]. All algorithms and techniques are merely mentioned in this part as they are explicitly explained in the next section.

There are three tasks which are carried out on the given video-data. The first task is to overlay a video over the quadrilateral which appears in the video. The second task is to fix certain gaps which appear in the image data, mostly around the foreground. This is necessary because kinect doesn’t always find the range data, in which case all the information for that pixel is missing. Finally, the third task is to overlay a given image over the actual background wall.

The first step of the workflow is to import the data in order to be able to work with it. The data is broken into three parts. Firstly, there is the actual data, which contains the image data and the 3D range data from the video to be augmented. Secondly, there is the background image which is to be placed instead of the real background of the video. Thirdly, there are the frames of a well-known video [2] which are re-mapped over the quadrilateral that appears in the video. For

the rest of this report the quadrilateral is also referred to as a suitcase. The next step is to transform the imported data in order for them to be displayed correctly. The original data is rotated by 90 degrees to the right.

Having finished with the initialisation step, the program enters the main loop. The way the system works is it selects each individual frame from the data (36 frames in total) and carries out the augmented reality process. During this main loop the first task (as described earlier) is completed.

Using the iterative approach as described in the ‘Algorithms and techniques’ section, the plane of the suitcase is first extracted together with all the points which are contained within. Having extracted these two sets of data, the four corners of the suitcase are found and ordered in a clockwise fashion.

Knowing the four corners of the quadrilateral and the equation of the plane on which it lies, the remapping algorithm is used to overlay a frame of the video over it. The video which is overlayed is also 36 frames long and each frame is directly mapped to a frame of the original video.

Having completed the first task and ended the main loop, all the pixels which have missing values due to unavailable range data are filled in, as described in the next section. Following that, using the same remapping algorithm which was used for the suitcase, a background image is overlayed over the background wall. While the quadrilateral identification is automatic, due to the fact that the camera is still at all times, the corner data of the background are specified manually during the initialisation step. Moreover, even though the same plane growing algorithm as before is used, the initial points needed are also hard-coded.

Finally, with the individual tasks completed and the individual frames of the video being augmented, an .AVI format video is created.

Algorithms and techniques

RGB and HSV Thresholding

Thresholding in terms of the RGB or the HSV colour space is used for certain tasks. It is a very straightforward technique where individual Red, Green and Blue or Hue, Saturation and Value values of given pixels need to be within a given range. All the threshold ranges were adapted through multiple iterations of testing.
While RGB was used originally, due to fluctuations in the illumination between different frames HSV was considered to be better. More information about the comparison of the two is in the ‘Discussion’ section.

It is worth noting that the thresholding in the final version of the system is adaptive HSV. Additionally, thresholding is done only with regards the Value (brightness property). The adaptive element of the technique works by calculating the average Value of the entire frame. Then the actual threshold T is set as a constant k (which models how close to mean the Value of the pixel that is being checked can be) multiplied by the mean. This can be formalised as follows:

$$T = k * m(V)^2$$

where m(V) corresponds to the mean of the HSV colour-space Value plane. It is worth noting that although for the final implementation, raising the mean value to the power of 2 yielded better results, it is not necessary.

Corner detection and ordering

In order to detect the corners, an ad-hoc algorithm has been developed. The way it works is it takes the collection of points of the quadrilateral as detected through plane selection. It finds the centre point of the suitcase as the mean of all the given points. Then the point which has the largest euclidean distance from the centre is found and assumed to be the first corner. Having found the corner, the algorithm removes all the points which are closer to the found corner than the centre. The same process is repeated four times, once for each corner.

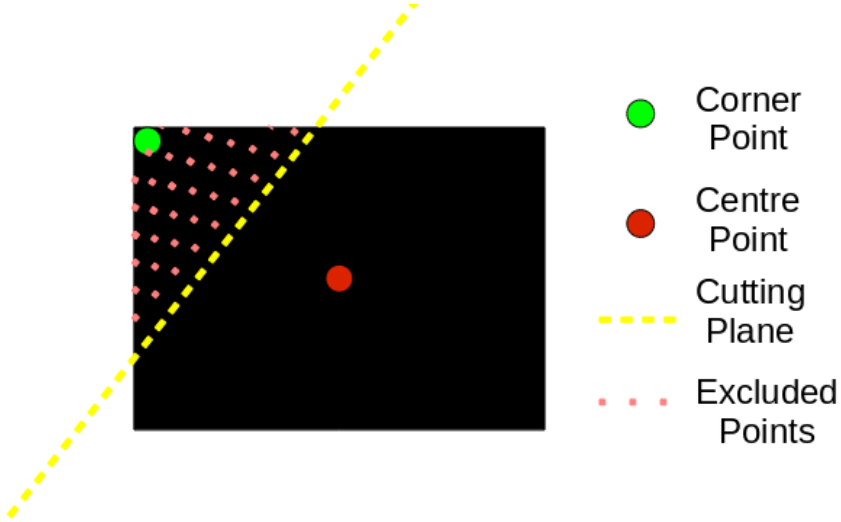

Figure 1: Corner detection

Having detected the four corners, it is important to sort them in a clockwise manner, from the top-left to the bottom-left corner. The reason for doing this is due to the input format required by the remapping function. The main idea behind ordering them is based on geometry. Firstly, the centre point of the four corners is found as the average coordinate value between them.

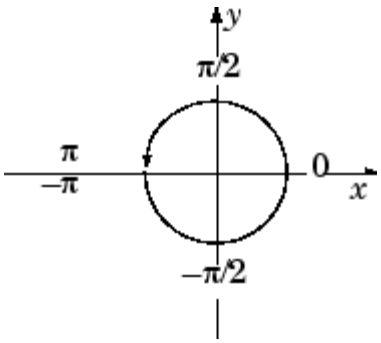

Figure 2: Corner Points Ordering

Then the vectors between each corner and the centre point are calculated. In order to sort the corners, the angles given by the four-quadrant inverse tangent based on the vectors are sorted with the sort() Matlab function in descending order. The angles are calculated using the atan2() Matlab function.

Remapping

Remapping maps an image inside a quadrilateral in the output image. It estimates the homography mapping, and uses the nearest neighbour to map the points. It also performs a plane check, which checks for every pixel to be overlayed, if it lies close enough to the target plane. The code for this function was adapted from the AV website.

Plane selection

The RANSAC method is used to find a dark plane with the least square error. Adaptive HSV Thresholding on the Value (brightness) is used to select two sets of points: a smaller and a bigger set, where a smaller set has a stricter threshold. The code was adapted from the code provided on the AV website.

Initial patch selection

Finds a patch which lies on a plane. It selects a random point from a smaller set of points (which have a stricter relative brightness threshold), and creates a patch (subset of points) which are within a certain distance of the random point selected. If the number of points in the initial patch is sufficient and if the plane's residual error is small enough the patch is selected as the initial patch. Otherwise try another random point or fail if the enough random points were tried (no plane detected).

The residual error tolerance is increased in each iteration, therefore finding a best fitting patch. This avoids the problems associated with setting the threshold manually - the threshold might differ between the frames.

Growing patch

Growing patch method finds all the points which are connected to the patch and lie in the same plane as the patch. In each iteration it adds to the patch all the points which are very close to any point which was added to the patch in the previous iteration and which are close enough to the plane of the patch. It stops growing once there are no more points to add.

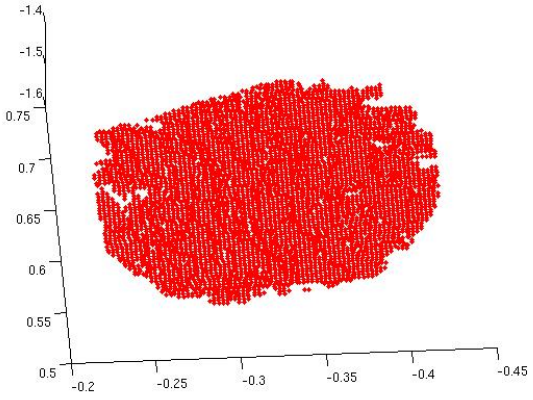

Figure 3: Growing patch middle step

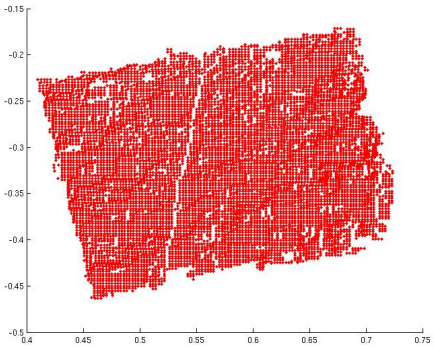

Figure 4: Fully grown patch

Quadrilateral detection

Quadrilateral is detected using the plane selection method. If that method returns a set of points which is insufficiently large, or whose plane residual error is too high, it will try to find the quadrilateral again, using the same method. This should help in a highly unlikely case of the wrong initial patch being selected. In the most recent implementation the quad seems to be found in the first iteration every time, if it is present in the image, however in the previous implementations which didn't have very good thresholds, it was helpful, as sometimes a part of the background would be dark enough, and a plane would be found there, or a part of the trousers would happen to be linear enough to pass the initial patch selection.

No-range data

The data for which there is no range information is replaced by the data from an early frame, with a brightness normalisation. If there is no range information, there isn't any colour information either, so such pixels in the original image are black. The brightness is normalised by simply multiplying by the ratio between the current frame's and early frame's average brightness. This doesn't always have the desired effect, as in a few frames parts of the person's hair have no range data.

Results

The program correctly finds the suitcase in all the frames, where the whole suitcase is present in the image. The image is correctly overlayed on the background wall. No-range data isn't always estimated correctly as it is always assumed to be background.

The runtime of the program (processing all 36 frames) is 8 minutes. It was significantly reduced from 2.5 hours by optimising the code for speed. This made debugging and parameter adjustment much faster.

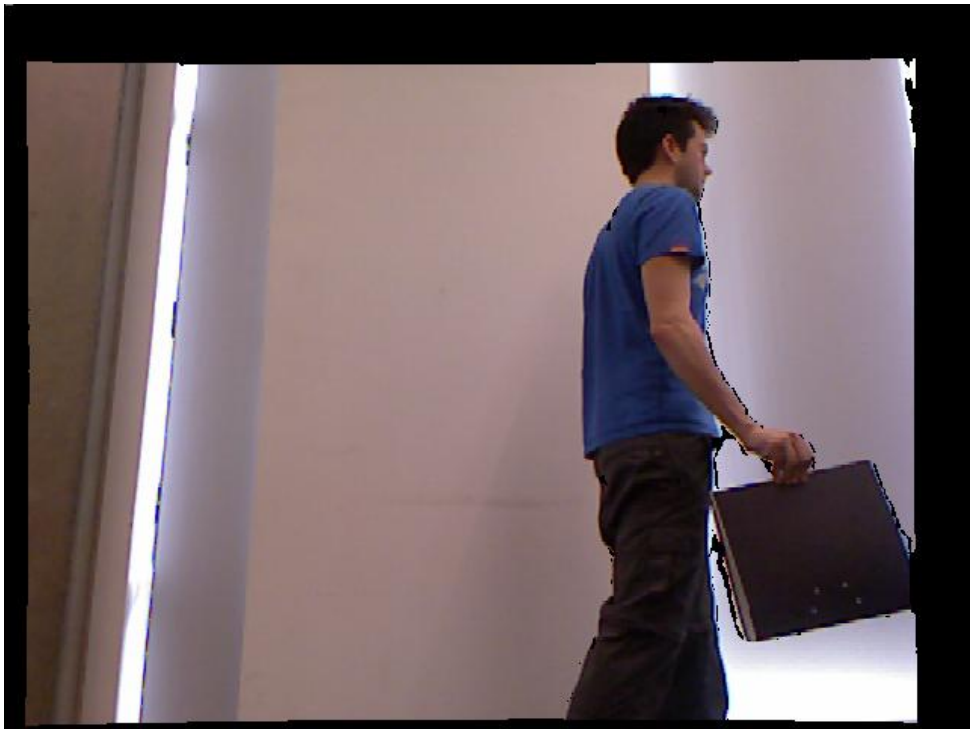

Figure 5: Original Image

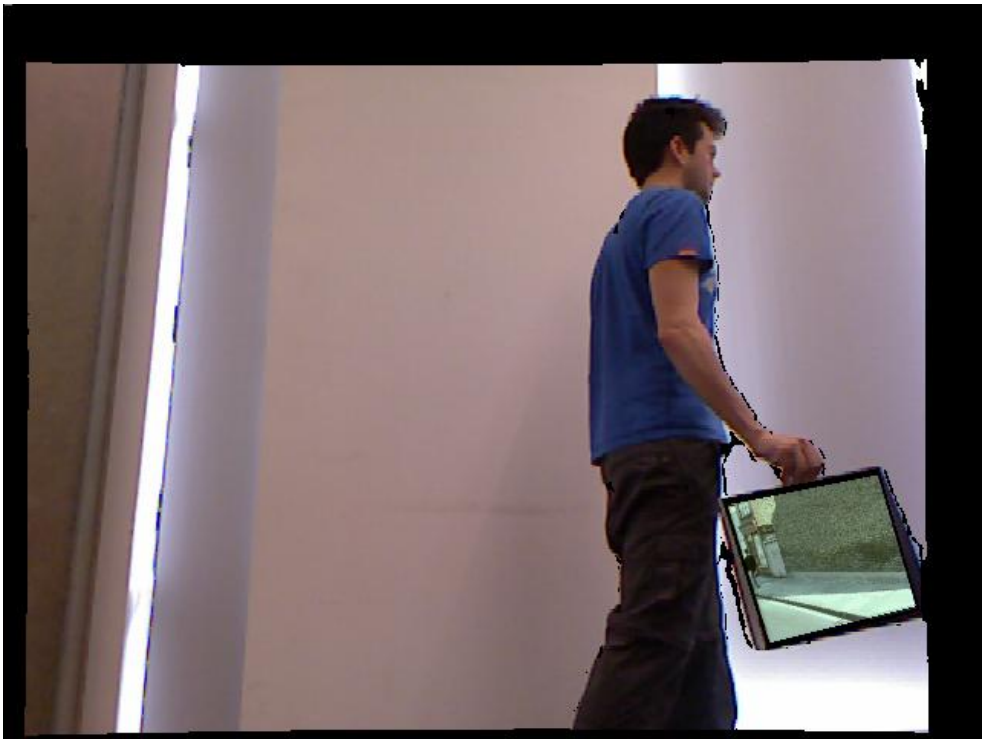

Figure 6: Video Overlayed on the Quad

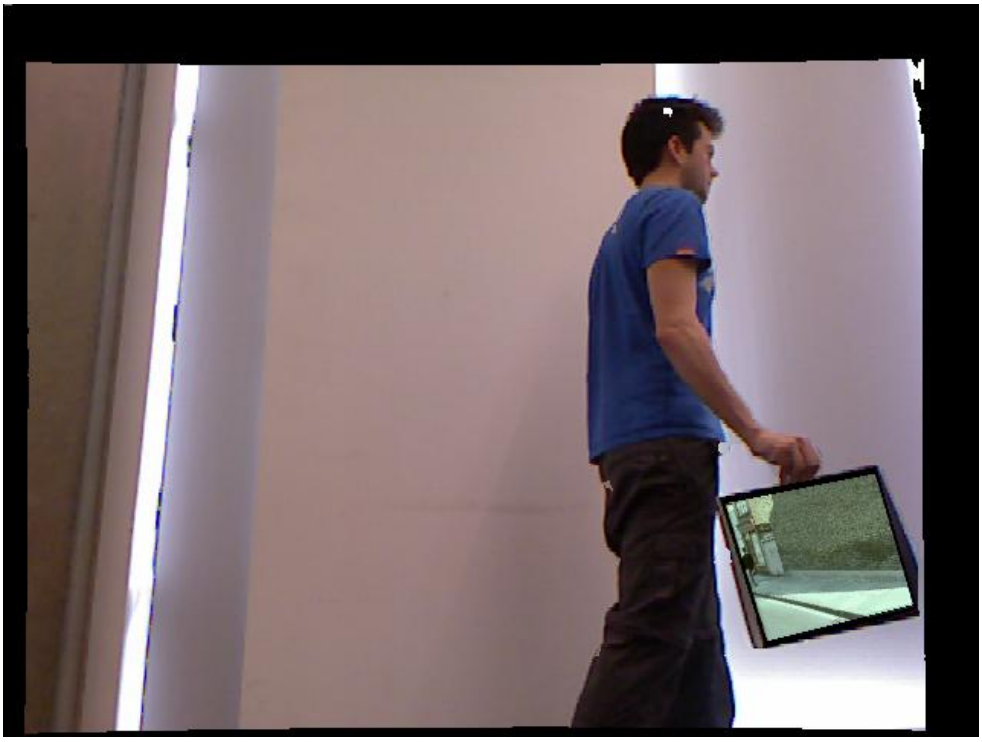

Figure 7: No-range points replaced with the computed background

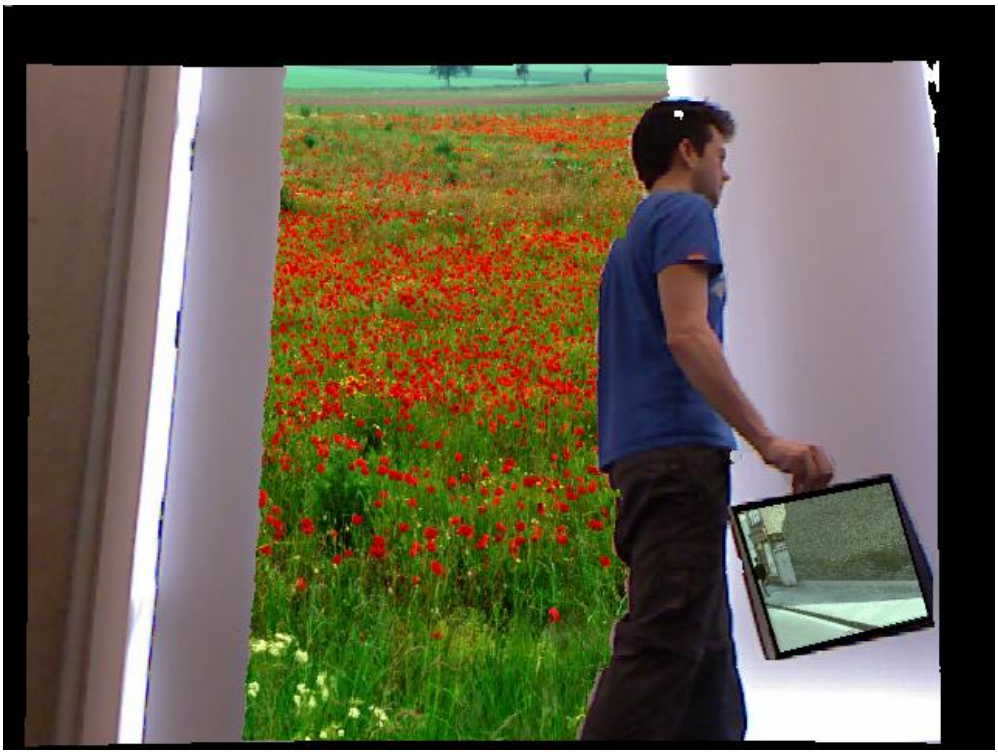

Figure 8: Final frame - background overlayed

Discussion

Augmented reality is gaining research interest quickly over the past few years[3] as cheaper sensor devices hit the market (e.g. Kinect). With the increase in the number of smartphones[4] there is an additional interest, as these devices combine powerful cameras with processing power.

The system presented in this report has achieved all the specified tasks and generates a video showing the results. It is important to note that the techniques used are relatively simple and transferable to a wide range of vision problems but are nonetheless in many cases ad-hoc and heavily relying on threshold values.

Future work can look into generalising the system. This means further research into better adaptation of the HSV thresholds, using background removal, automatic extraction of the background wall plane (as it is done for the quadrilateral), more advanced corner detection and restoration of the video by fixing the missing values. Suggestions for how these steps can be tackled are given later throughout the discussion.

The solution to the corner detection problem for example, while simple and effective for the specific data, has several shortcomings that might affect it when it comes to using it in a more general setting. For example, if the quadrilateral shape would be different there is danger that during the removal of points after finding a corner, one of the four corners could also be removed. Significant research has been carried out for corner detection [5] and as such there are several alternative solutions that can be considered for future work [6][7]. It is important to note however, that the performance of such algorithms could also run into shortcomings. For example the Level Curve Curvature or the Harris & Stephens algorithms would require accuracy during the plane selection of the suitcase. In the case where there are gaps or artifacts in the selection there is the possibility of such an algorithm to find one of them as a corner.

An additional observation when looking at the results, is that there are certain pixels where there is missing data due to the fact that the range data is unavailable. By using these pixels as the selection for remapping the background image, the final system deals with the artifacts which appear around the walking person.

Figure 9: Difficult frame - background overlayed on the missing values for the person's head; Part of the suitcase is just ignored (rather than attempting to estimate where the other two corners should be)

However, there are certain frames where there are pixels on the walking person. One ad-hoc solution would be to look at the neighbouring frames (e.g. the previous or the next) and to fill-in the missing values with the values from the corresponding past or future pixels. Nonetheless, not only does such a solution create the problem of having an automated way to select the previous or the next frame but it also cannot deal with all the cases. There are certain cases where the missing values are from pixels of the arm of the person, however the same pixels in the previous and the next frames belong to the shirt of the person. An alternative and more complex look at the problem is to examine inpainting algorithms and techniques [8][9]. Inpainting or image interpolation can look at the neighbouring pixels of the missing data and interpolate the most probable value. One relatively simple approach for implementing this approach is to use a form differential equations with Dirichlet boundaries to establish a continuous and 'seamless' result [10].

Another part worth discussing is the remapping of the video over the suitcase in terms of depth. The walking person is holding the quadrilateral and inevitably some fingers appear over it. This can potentially be a problem when selecting the plane as the fingers might be included if the restrictions on growing of the plane are too relaxed. In the system reported in this paper this is avoided by having optimized the tolerance level for growing. In the case where the quadrilateral is perfectly identified but the video is overlayed over the fingers, the solution is to make the restrictions in terms of the depth location of the plane stricter. Since the fingers are slightly more forward than the rest of the quadrilateral then this can provide a solution. It is important to note that the final system generates results that are not affected by this.

The benefits of using two separate sets of points for patch selection and patch growth can be seen on the following example. If figure 11 was used for initial patch selection, there would be a possibility of finding a plane, which is in fact not a quadrilateral which the person is holding. By restricting the choice of the initial points to a figure 10 (using stricter threshold), all the other planes are successfully removed, however the suitcase doesn't include all the points. This would be a problem for patch growth, but isn't a problem for patch selection.

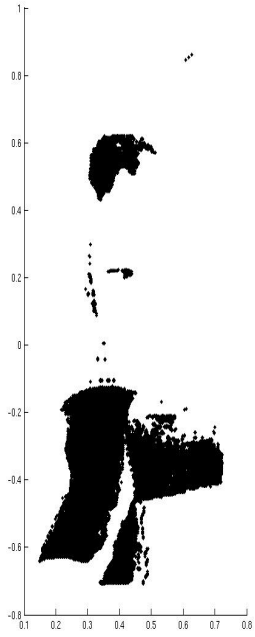

Figure 10: Pointset for patch selection

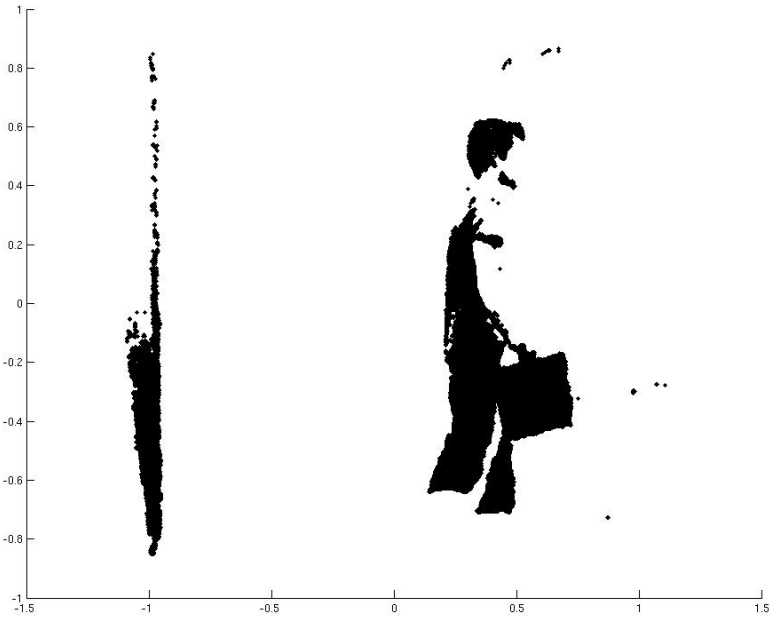

Figure 11: Points used for growth of the plane

In our initial implementation we used RGB thresholds for detecting the suitcase. But these threshold didn't scale very well, as they differed significantly between different frames. Therefore we started experimenting with the HSV thresholds, and we soon discovered that a value threshold was enough to get a satisfactory result. Further improvement here is possible as these thresholds won't generalise well.

Bibliography

[1] D.L. Parnas, On the Criteria To Be Used in Decomposing Systems into Modules, Carnegie-Mellon University, Reprinted from Communications of the ACM, Vol. 15, No. 12, December 1972 pp. 1053 - 1058, Association for Computing Machinery Inc.

[2] Monty Python's Flying Circus, "The Ministry of Silly Walks", episode 14: "Face the Press".

[3] Augmenting Indoor Spaces Using Interactive Environment-aware Handheld Projectors.

[4] IDC Press Release 06 Feb 2012 http://www.idc.com/getdoc.jsp?containerId=prUS23299912.

[5] T. Lindeberg (1994). *Scale-Space Theory in Computer Vision*. Springer. ISBN 0-7923-9418-6.

[6] C. Harris and M. Stephens (1988). "A combined corner and edge detector". *Proceedings of the 4th Alvey Vision Conference*. pp. 147–151.

[7] L. Kitchen and A. Rosenfeld (1982). "Gray-level corner detection"". *Pattern Recognition Letters* **1** (2).

[8] M. Bertalmío, G. Sapiro, V. Caselles and C. Ballester., "Image Inpainting", Proceedings of SIGGRAPH 2000, New Orleans, USA, July 2000.

[9] Homan Igehy and Lucas Pereira, Image Replacement through Texture Synthesis, Stanford University, Appears in the Proceedings of the 1997 IEEE International Conference on Image Processing.

[10] Peterson, Ivars (May 11 2002). "Filling in Blanks". *Science News* (Society for Science &#38) **161** (19): 299–300. doi:10.2307/4013521. JSTOR 4013521. Retrieved 2008-05-11.

1 Appendix: Code Listings

1.1 File: main.m

```
% Main

% Initialize data locations
imagesDir = 'data/bindermat';
backgroundDir = 'data/field.jpg';
videoDir = 'data/video';

% manually selected corners of the overlaying background image
backgroundCorners=[[40,182]',[39,429]',[474,453]',[474,155]']';

% Loading data
images = readData(imagesDir);
backgroundImage = double(imread(backgroundDir));

[width,height,nChannels,nImages] = size(images);

% Transforming images so that they are displayed correctly
images = transformData(images);
earlyFrame = images(:,:,:,2);

writeImgs(images,1:nImages, 'original');

% Find a big patch for estimating the plane based on the four corners
sizePatch= 100;
middlePoint = round(mean(backgroundCorners));
xRange = [middlePoint(1)-sizePatch, middlePoint(1)+sizePatch];
yRange = [middlePoint(2)-sizePatch, middlePoint(2)+sizePatch];
xSize = xRange(2) - xRange(1) + 1;
ySize = yRange(2) - yRange(1) + 1;

% Compute Background overlaying plane
initialPoints = earlyFrame(xRange(1):xRange(2),yRange(1):yRange(2),1:3);
initialPoints = reshape(initialPoints(:,:,1:3), xSize*ySize, 3);

[plane, fit] = fitplane(initialPoints);


% Iterate over the images
for i=1: nImages

    frame = i

    % In case of a very improbable event of not finding the best fitting
    % plane, while there is one, run the algorithm again
    for j=1:1
        attempt = j
        [ quadPoints, suitcasePlane ] = planeExtraction(images(:,:,:,i));
        if isSuitcase(quadPoints)
            break;
        end
    end

    if isSuitcase(quadPoints)
        % find ordered corners of the suitcase
        suitcaseCorners = getCorners(quadPoints);
        pixelVals = getPixelVals(suitcaseCorners, images(:,:,:,i));
        orderedCorners = orderCorners(pixelVals);

        % Load corresponding video frame
        listing = dir( videoDir );
        imagePath = strcat(videoDir, '/', listing( i+3 ).name);
        videoFrame = double(imread(imagePath));

        % Remap video on the suitcase
        images(:,:,:,i) = remap(videoFrame, images(:,:,:,i), suitcasePlane, ↩
            orderedCorners, 0.02);
    end

```

```
68      writeImgs(images,i, 'quad');
69
70      % Make pixels without any value into background pixels
71      images(:,:,:,i) = fillMissingVals(images(:,:,:,i), earlyFrame(:,:,4:6));
72      writeImgs(images,i, 'fill');
73
74      % Remap field image as background image
75      UV=backgroundCorners;    % target points
76      images(:,:,:,i) = remap(backgroundImage, images(:,:,:,i), plane, UV, 0.05);
77
78      imshow(images(:,:,4:6,i));
79  end
80
81  % create the AV.avi video of all the frames
82  generateVideo(images);
83
84  % write all the final images to the output folder
85  writeImgs(images,1:nImages, 'final');
```

1.2 File: planeExtraction.m

```
1   function [ oldlist, plane ] = planeExtraction( image )
2   % Finds a plane with the lowest residual errror using a RANSAC method.
3   % First finds a candidate patch, which it grows until it satisfies the
4   % planar property, or there are no more points to add.
5
6
7   im3d = image(:,:,1:3);
8   imRGB= image(:,:,4:6);
9
10  % Two thresholds, strict for initial points and relaxed for all the points
11  initialPtsBinary = hsvThresh(imRGB, 0.8) & hasRangeData(im3d);
12  inclusiveBagPtsBinary = hsvThresh(imRGB, 1.6) & hasRangeData(im3d);
13
14  initialPts = im3d(repmat(initialPtsBinary,[1 1 3]));
15  inclusiveBagPts = im3d(repmat(inclusiveBagPtsBinary,[1 1 3]));
16
17  R = reshape(im3d, 640*480,3); % 3d points
18  initialPts = reshape(initialPts, length(initialPts)/3,3);
19  inclusiveBagPts = reshape(inclusiveBagPts, length(inclusiveBagPts)/3,3);
20
21  [NPts, ~] = size(R);
22
23  remaining = inclusiveBagPts;
24
25  % select a random small surface patch
26  [oldlist,plane] = select_patch(initialPts);
27
28  % grow patch
29  stillgrowing = 1;
30
31  DISTTOL  = 0.01;            % grow slowly to avoid jumping to another plane
32  PLANETOL = 0.01;
33
34  pointsAdded = size( oldlist, 1 );
35
36  while stillgrowing
37
38  % find neighbouring points that lie in plane
39    stillgrowing = 0;
40    [newlist,remaining] = getallpoints(plane,oldlist((end-pointsAdded + 1):end,:), ↩
          remaining,NPts,DISTTOL,PLANETOL);
41    newlist = [ oldlist ; newlist ];
42
43    [NewL,W] = size(newlist);
44    [OldL,W] = size(oldlist);
45
46    pointsAdded = NewL - OldL;
47
48
49    if (pointsAdded > 10) % TODO: look at this param
```

```
    % refit plane
    [newplane,fit] = fitplane(newlist);
    if fit > 0.04*NewL       % bad fit - stop growing
      break
    end
    stillgrowing = 1;
    oldlist = newlist;
    plane = newplane;
  end
end

end
```

1.3 File: select_patch.m

```
function [fitlist,plane] = select_patch(points)
% Finds a candidate planar patch.
% Iteratively increases residual error tolerance until a good fit is found.

[L,D] = size(points);
tmpnew = zeros(L,3);

initialErrTol = 0.01;
maximumErrTol = 0.04;
nSteps        = 500;

residErrTol   = initialErrTol;
stepSize      = (maximumErrTol - initialErrTol) / nSteps;

% pick a random point until a successful plane is found or no plane can
% be found (allow for 500 random point selections)
for nStep = 1:nSteps
    idx = floor((L-1)*rand)+1;   %BUGFIX: avoiding access of unexisting elements
    pnt = points(idx,:);


    % find points in the neighborhood of the given point
    DISTTOL = 0.08;
    fitcount = 0;
    for i = 1 : L
      dist = norm(points(i,:) - pnt);
      if dist < DISTTOL
        fitcount = fitcount + 1;
        tmpnew(fitcount,:) = points(i,:);
      end
    end

    if fitcount > 1500
      % fit a plane
      [plane,resid] = fitplane(tmpnew(1:fitcount,:));

      if resid < residErrTol
        fitlist = tmpnew(1:fitcount,:);
        return
      end
    end

    %increase allowed error tolerance
    residErrTol = residErrTol + stepSize;
end

% no plane found - return points which will fail.
fitlist=points(1:5,:);
plane=[1; 1; 1; 1];
```

1.4 File: getallpoints.m

```
function [newlist,remaining] = getallpoints(plane,oldlist,P,NP, DISTTOL,PLANETOL)
% selects all points in pointlist P that fit the plane and are within
% TOL of points added newly to the list of points in the plane.

  pnt = ones(4,1);
  [N,W] = size(P);
  [Nold,W] = size(oldlist);

  tmpnewlist = zeros(NP,3);
  tmpremaining = zeros(NP,3);           % initialize unfit list
  countnew = 0; %Nold;
  countrem = 0;

  for i = 1 : N
    pnt(1:3) = P(i,:);
    notused = 1;


    if abs(pnt'*plane) < PLANETOL
        % speed optimisation of vectorising the for loop was suggested
        % by Cristian Cobzarenco
        if any( sum((oldlist - repmat( P(i,:), Nold,1)) .^ 2, 2 ) < DISTTOL^2 )
            countnew = countnew + 1;
            tmpnewlist(countnew,:) = P(i,:);
            notused = 0;
        end
    end

    if notused
      countrem = countrem + 1;
      tmpremaining(countrem,:) = P(i,:);
    end
  end

  newlist = tmpnewlist(1:countnew,:);
  remaining = tmpremaining(1:countrem,:);
```

1.5 File: getCorners.m

```
function [ corners3d ] = getCorners( quadPts )
% This function gets the corners of a quad given a list of points. It finds
% the point which is further away from the centroid, and then removes all
% the points which are closer to that corner than to the centroid. Then it
% repeats this three more times.

corners3d = zeros(4,3);
nPts = length(quadPts);
centroid = mean(quadPts, 1);

dists3 = abs(quadPts - repmat(centroid, [nPts 1]));
dists = get3dLen( dists3 );


for i= 1: 4
    [~, idx] = max(dists);
    corners3d(i,:) = quadPts(idx, :);

    distsCor3 = abs(quadPts - repmat(corners3d(i,:), [length(quadPts) 1]));
    distsCor = get3dLen( distsCor3 );

    L = repmat(dists < distsCor, [1 3]);

    quadPts = quadPts( L );
    quadPts = reshape(quadPts,length(quadPts)/3, 3);

    dists3 = abs(quadPts - repmat(centroid, [length(quadPts) 1]));
    dists = get3dLen( dists3 );
end

end
```

1.6 File: orderCorners.m

```
function [ outputCorners ] = orderCorners( inputCorners )
% This function gets a list 4 of corners and using their angle from the
% centre of the quad, sorts them from top left to bottom right (clockwise).

corners(:,1) = inputCorners(:,2);
corners(:,2) = inputCorners(:,1);

Cx = sum(corners(:,1)) / 4;
Cy = sum(corners(:,2)) / 4;

centre = [Cx, Cy];
centre = repmat(centre, 4, 1);

tmp = corners - centre;

angles = atan2(tmp(:,1),tmp(:,2));
[sortAngles, idx] = sort(angles,'descend');

almostOrderedCorners = corners(idx,:);
orderedCorners(1,:) = almostOrderedCorners(4,:);
orderedCorners(2:4,:) = almostOrderedCorners(1:3,:);

outputCorners(:,1) = orderedCorners(:,2);
outputCorners(:,2) = orderedCorners(:,1);

end
```

1.7 File: hsvThresh.m

```
function vThresh = hsvThresh(inImage, val)

% Performs Value Tresholding on a given image relative to the average
% brightness.

% Convert the image to the HSV space
hsvImage = rgb2hsv(inImage);

% Get the value plane and threshold
vPlane = hsvImage(:,:,3);
valThresh = val * mean(mean(vPlane))^2;

% Find indeces that will be removed based on thresholding
vThresh = (vPlane < valThresh ) & (vPlane>0);

end
```

1.8 File: remap.m

```
function [ finalImage ] = remap( inimage,image, plane, UV, DISTTOL )
% Remaps inimage onto the quad in image. UV specifies the four corners  of
% the quad in the clockwise order. DISTTOL specifies the distance tolerance
% from the plane of the quadrilateral.

outimage = image(:,:,4:6);


% get sizes of the input and output images
[IR,IC,D]=size(inimage);
[OR,OC,D]=size(outimage);

% corners of the input image
XY=[[1,1]',[1,IC]',[IR,IC]',[IR,1]']';    % source points

```

```
P=esthomog(UV,XY,4);    % estimate homography mapping UV to XY

pnt = zeros(4,1);
pnt(4) = 1;

% loop over all pixels in the destination image, finding
% corresponding pixel in source image
for r = 1 : OR
    for c = 1 : OC
      v=P*[r,c,1]';         % project destination pixel into source
      y=round(v(1)/v(3));  % undo projective scaling and round to nearest integer
      x=round(v(2)/v(3));

      pnt(1:3) = image(r,c,1:3);

      if (x >= 1) && (x <= IC) && (y >= 1) && (y <= IR) && ...
              (abs(pnt'*plane) < DISTTOL || pnt(3)==0 )
        outimage(r,c,:)=inimage(y,x,:)/255;   % transfer colour
      end
    end
end

finalImage(:,:,1:3) = image(:,:,1:3);
finalImage(:,:,4:6) = outimage(:,:,1:3);

end
```

1.9 File: fillMissingVals.m

```
function [ image6d ] = fillMissingVals( image6d, bcg )
% Replace the missing colour values (range == 0) with the background values.
% Background pixel values are estimated using an early frame normalised for the
% mean brightness (HSV Value).

img = image6d(:,:,4:6);
    I =repmat(image6d(:,:,3)==0, [1 1 3]);    % distance (z-value) == 0

    img(I) = bcg(I) * mean(mean(mean(img))) / mean(mean(mean(bcg)));
    image6d(:,:,4:6) = img;

end
```

1.10 File: getRange.m

```
function [ points3d ] = getRange( img, points )
% Given a list of pixel indeces returns a corresponding list of 3d points.

[nPoints, nDims] = size(points);
points3d = zeros(nPoints,nDims+1);

for i=1:nPoints
    points3d(i,:) = img(points(i,1), points(i,2), 1:3);
end

end
```

1.11 File: hasRangeData.m

```
function [ binary ] = hasRangeData( pts3d )
% Returns a logical of all the points which have some range data.

ranges = pts3d(:,:,3);
binary = ranges ~= 0;
```

```
6
7 end
```

1.12 File: normRGB.m

```
1 function normImage = normRGB(image)
2 % gets the RGB values in the [0,1] range.
3
4 normImage(:,:,1:3) = image(:,:,1:3);
5 normImage(:,:,4:6) = image(:,:,4:6) / 255;
6 end
```

1.13 File: get3dLen.m

```
1 function [ dists ] = get3dLen( dists3 )
2 % Finds the 3d length for all the points
3
4 dists = dists3(:,1).^2 + dists3(:,2).^2 + dists3(:,3).^2;
5
6 end
```

1.14 File: getPixelVals.m

```
1 function [ corners ] = getPixelVals( corners3d, image )
2 % Given the corners of the quad in 3D space it finds corresponding 2d pixel
3 % values of each corner.
4
5 [r1,c1] = find((image(:,:,1) == corners3d(1,1)) & (image(:,:,2) == corners3d(1,2)) & (↩
      image(:,:,3) == corners3d(1,3)));
6 [r2,c2] = find((image(:,:,1) == corners3d(2,1)) & (image(:,:,2) == corners3d(2,2)) & (↩
      image(:,:,3) == corners3d(2,3)));
7 [r3,c3] = find((image(:,:,1) == corners3d(3,1)) & (image(:,:,2) == corners3d(3,2)) & (↩
      image(:,:,3) == corners3d(3,3)));
8 [r4,c4] = find((image(:,:,1) == corners3d(4,1)) & (image(:,:,2) == corners3d(4,2)) & (↩
      image(:,:,3) == corners3d(4,3)));
9
10 corners(1,1:2) = [r1,c1];
11 corners(2,1:2) = [r2,c2];
12 corners(3,1:2) = [r3,c3];
13 corners(4,1:2) = [r4,c4];
14
15 end
```

1.15 File: isSuitcase.m

```
1 function [ isSuitcase ] = isSuitcase( pts3d )
2     % This function simply checks if a set of points in 3D space is large
3     % enough to be assumed to be the correct quad (suitcase of the man).
4
5     nPoints = length(pts3d);
6     nPointsTest = nPoints > 10000;
7     % residual error of the plane is already checked
8
9     isSuitcase = nPointsTest;
10
11 end
```

1.16 File: readData.m

```
function [ images ] = readData( path )
% Loads a sequence of images from a given 'path' (a string specifying a
% path to a folder).

listing = dir( path );

nImages    = size( listing, 1 ) - 3;
images     = zeros( 640, 480, 6, nImages);


for i = 1 : nImages
    var=importdata( [ path '/' listing( i+3 ).name ] );
    images( :, :, :, i ) = reshape(var, 640,480,6);
end

end
```

1.17 File: transformData.m

```
function [ images ] = transformData( imgs )
% Normalises the images (so that RGB is in double as well) and transposes
% the data.

[width, height, channels, nImages] = size(imgs);
images = zeros(height, width, channels, nImages);

for i=1: nImages

    im = normRGB(imgs(:,:,:,i));
    for j=1: channels
        images(:,:,j,i) = im(:,:,j)';
    end

end

end
```

1.18 File: generateVideo.m

```
function generateVideo(images)

    % Generates a video from the individual frames

    vw = VideoWriter('AV_movie.avi');
    vw.FrameRate = 6;
    vw.open();

    for i = 1:36
        image = images(:,:,4:6,i);
        imshow(image);
        writeVideo(vw,getframe(gcf));
    end

    close(vw);
end
```

1.19 File: writeImgs.m

```
function [ ] = writeImgs( images, range, fname )
%Writes the images in the output folder
```

```
3
4 for i=range
5     imwrite(images(:,:,4:6,i), ['output/',int2str(i),fname,'.jpg'] );
6 end
7
8
9 end
```