Final report
SIGB Spring 2014

Marcus Gregersen
mabg@itu.dk

Martin Faartoft
mlfa@itu.dk

Mads Westi
mwek@itu.dk

May 18th 2014
IT University of Copenhagen

Chapter 1

Assignment 1 - Eye Tracking

Eye Tracking
SIGB Spring 2014

Marcus Gregersen
mabg@itu.dk

Martin Faartoft
mlfa@itu.dk

Mads Westi
mwek@itu.dk

March 26th 2014
IT University of Copenhagen

Contents

1 Introduction

In the following, we will experiment with, and discuss different approaches to detecting major eye features. Figure 1 gives the names of the eye features that are used throughout this report.

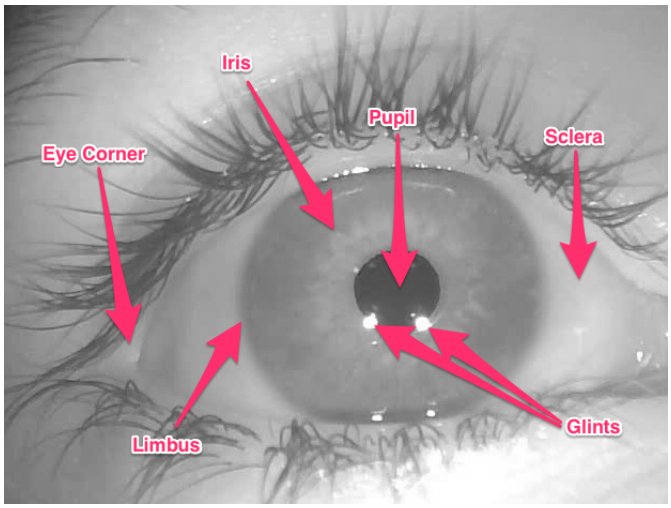


Figure 1: Names and positions of major eye features

2 Pupil Detection

In this section, we will investigate and compare different techniques for pupil detection.

2.1 Thresholding

An obvious first choice of technique, is using a simple threshold to find the pupil, then do connected component (blob) analysis, and finally fit an ellipse on the most promising blobs.

Figure 2 shows an example of an image from the 'eye1.avi' sequence and the binary image produced by, using a threshold that blacks out all pixels with intensities above 93. This manages to separate the pupil nicely from the iris.

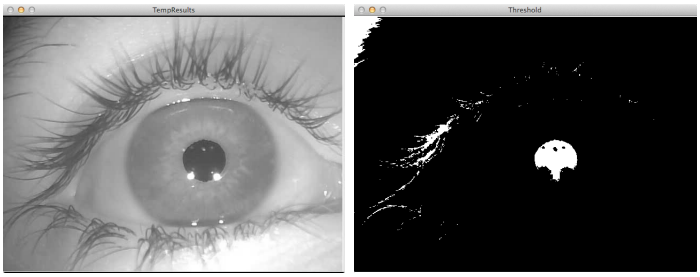


Figure 2: Thresholding eye1.avi

The next step, is to do connected component analysis, and fit an ellipsis through the blobs. As seen in Figure 3, this succesfully detects the pupil, but is extremely prone to false positives.

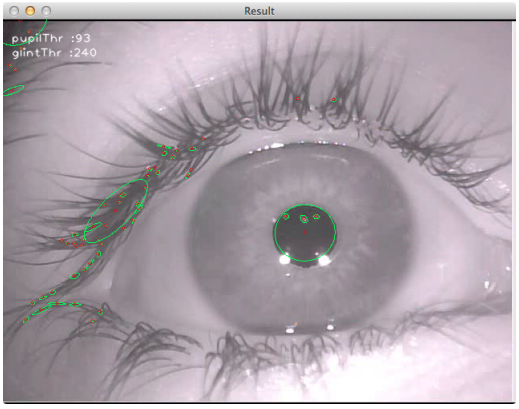


Figure 3: Fitting ellipses on blobs from eye1.avi (green figures are ellipses fitted through blobs, red dots are the centerpoint of each blob)

By experimenting, we find that requiring that the area of the blob lies in the interval $[1000 : 10000]$, and the extent between $[0.4 : 1.0]$, we eliminate most false positives on the entire eye1 sequence, while still keeping the true positive.

This approach has several problems, however. Note how the true positive on Figure 3 fails to follow the bottom of pupil correctly. This is due to the glints obscuring part of the boundary between pupil and iris. It also makes some sweeping assumptions:

The pupil has size at least size 1000 If the person on the sequence leans back slightly, the pupil will shrink and we will fail to detect it.

A threshold of 93 will cleanly separate pupil from iris This is true for eye1.avi, but does not generalize to other sequences. If this approach is

to be used across multiple sequences recorded in different lighting conditions, the threshold will have to be adjusted by hand for each one.

This problem can be mitigated somewhat with Histogram Equalization. A threshold of 25 on Histogram Equalized images, fares considerably better across several sequences.

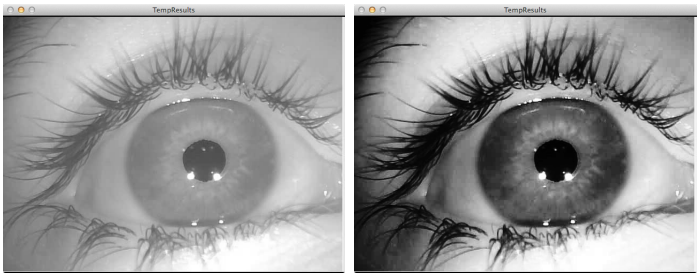


Figure 4: Eye1 before and after Histogram Equalization

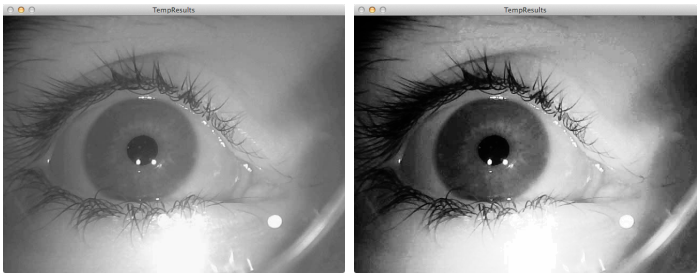


Figure 5: Eye3 before and after Histogram Equalization

Morphology Using Morphology, we can improve the detected pupil. The problem with the glints obscuring part of the boundary can be mitigated with the 'closing' operator - used to fill in holes in binary images. Figure 6 shows binary images before and after applying the closing operator. Notice how the noise inside the pupil is completely removed, and the glints are mostly removed. A downside to using the closing operation, is that adjacent, sparse structures may merge into something resembling a circle, thereby giving a false positive.

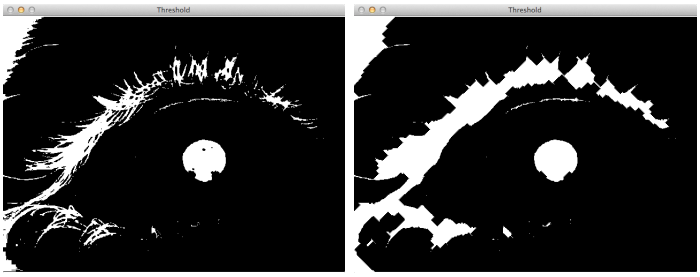


Figure 6: Eye1 before and after Closing (5 iterations, 5x5 CROSS structuring element)

Tracking The pupil tracker can be further improved, by using information about the pupil positions from the previous frame. We do it as follows:

1. Search within some threshold distance from each pupil in previous frame

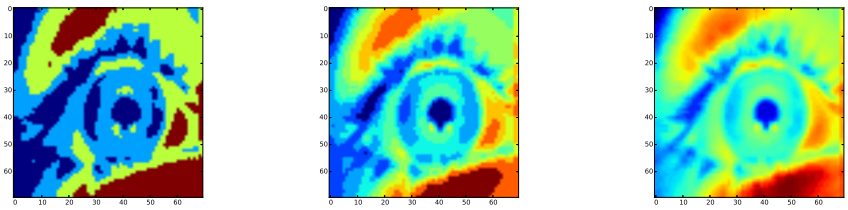
2. One or more pupils were found within the distance, return those

3. No pupils were found within the distance, search the entire image

Because of the fallback clause in '3', it is very unlikely that the true positive is not detected in each frame. The only case where this approach fails, is if the pupil is obscured for a frame (subject blinking for example), while a false positive is still detected. In that case, the pupil will be improperly detected for as long as the false positive continues to be present.

2.2 Pupil Detection using k-means

A method to enhance the BLOB detection of pupil detection is k-means clustering. The method separates the picture in K clusters. Each cluster is a set of pixels, which have values closer to the cluster center, than to other cluster centres - a cluster center corresponds to mean value of the pixels in the cluster. The value of K is arbitrarily chosen, so that for a sufficiently large number, K the pupil is evaluated as a single separate cluster. If the pupil is a single cluster a binary image can easily be created and BLOB detection would only need to look at the one object. The following figure illustrates how different values of K impacts the segmentation.



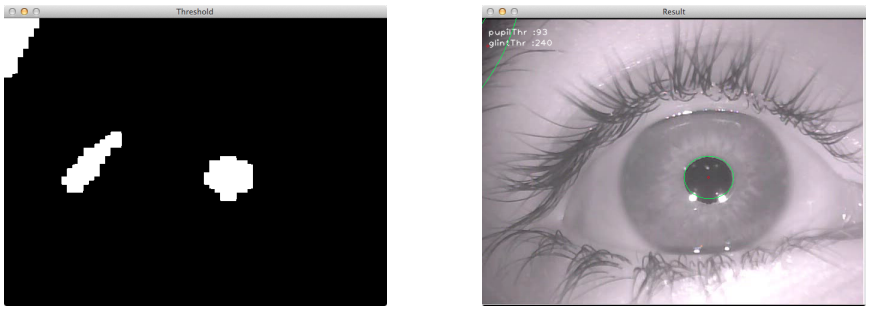
(a) $K = 4$ (b) $K = 10$ (c) $K = 80$

Figure 7: K-means for different values of K

It would be infeasable to calculate the k-means procedure on the original image. The clustering is therefore done on a downsampled 70x70 pixel image. To reduce the impact of noise, the resized image is filtered with a Gaussian filter. It is clear in Figure 7 that even for a very high K, the pupil is not a separate cluster, Experiments has revealed that at a K value in the range of 10 to 20 gives a reasonably clustered image, while not having a massive impact on performance.

Another parameter to tweak would be the size of the downsampling. A smaller downsampling yields less pixels to classify, which in some cases would yield a better performance, and vice versa.

Each pixel in the reduced picture is assigned to a cluster(label), selecting the pixels in the label with the lowest mean value, we can create a binary image, which now can be resized to the original image size. Because the cluster also contains pixels from other regions than the pupil area, the resulting binary image, on which we can do BLOB detection, is not optimal. Figure 8 shows this.



(a) Binary image (b) Resulting image

Figure 8: Pupil detection using k-means

Eyelashes and shadows become a part of the pupil cluster, and much like the issues with morphology, they often become connected components, which makes BLOB detection very difficult.

In conclusion the use of k-means did not yield a better result than ordinary thresholding, in many cases the result was actually worse, This is unfortunate because, if the pupil could be found as a single cluster, the amount of evaluation on the BLOB could be reduced and give better scalability on the position of the eye relative to the camera.

2.3 Pupil Detection using Gradient Magnitude

So far, we have been looking at the intensity values of the image. This has yielded reasonable approximate results, but is not as robust as we would like. In the following, we investigate what happens if we look at the change in intensity (the image gradient / first derivative), instead of the absolute intensity value at a given point. The gradients in the X and Y directions, are easily calculated with a Sobel filter. And from these, we can calculate the Gradient Magnitude as: $\sqrt{x^2 + y^2}$ (the Euclidean length of the vector x + y), and the orientation as: $arctan2(y, x)$. Figure 9 shows a subsampled cutout of the Gradient image of Eye1, featuring the pupil and glints.

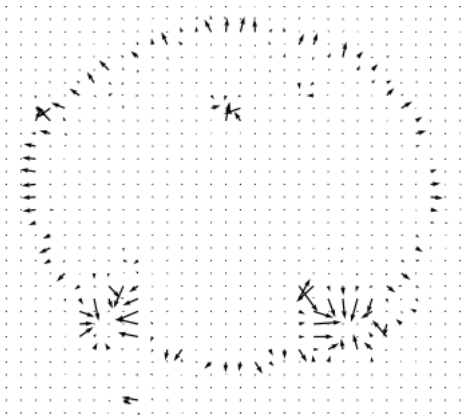


Figure 9: Quiver plot of Eye1 gradients (zoomed on pupil area)

Note that the pupil boundary is clearly visible on Figure 9. We will attempt to use this information as follows: given an approximate centerpoint and radius for the pupil, scan in a number of directions, d from the centerpoint, find the location of the maximum gradient magnitudes along the line-segments that are described by the centerpoint, a direction from d and the radius. Use this set of points to fit an ellipse, and use that as improved pupil detection.

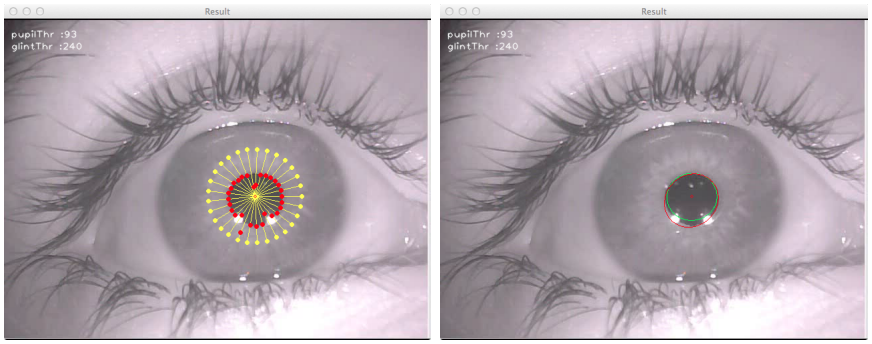


Figure 10: Left: Eye1 showing line-segments for gradient magnitude maximization (yellow) and maximum gradient values along the lines (red). Right: Eye1 showing pupil approximation (green), and new pupil detection (red)

Figure 10(left) shows the lines considered, and the max gradient points found. Figure 10(right) shows the old and new pupil detections. This approach suffers the same problem as earlier. When the glints obscure part of the boundary, the pupil detection fails to follow the lower boundary. On top of that, there are also issues with noise, notice the red dots inside the top part of the pupil on Figure 10, these are caused by non-system light reflecting off the pupil.

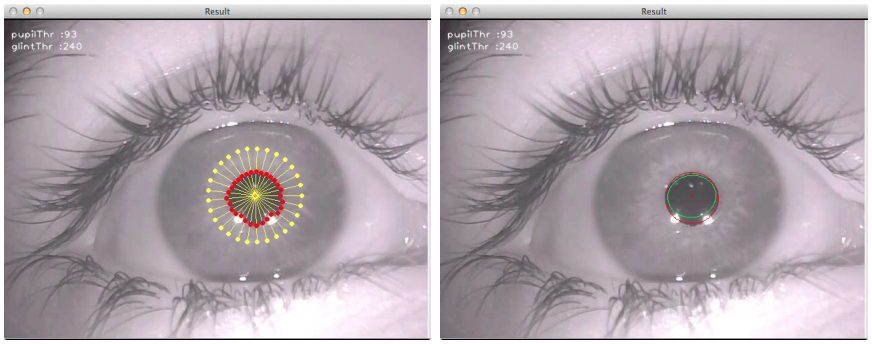


Figure 11: as Figure 10, but pre-processed with a 9x9 Gaussian Blur

The noise issues can be drastically reduced with proper pre-processing. Figure 11 shows much improved results, when blurring the image beforehand.

We experimented with ignoring gradient points where the orientation was too far from the orientation of the circle normal, but did not see any improvements to the pupil detection.

2.4 Pupil Detection by circular Hough transformation

In an attempt to make our pupil detection more robust we now investigate the result of applying a circular Hough transformation on the eye images.

The main challenge is finding the correct parameters for the process. We consider the following parameters:

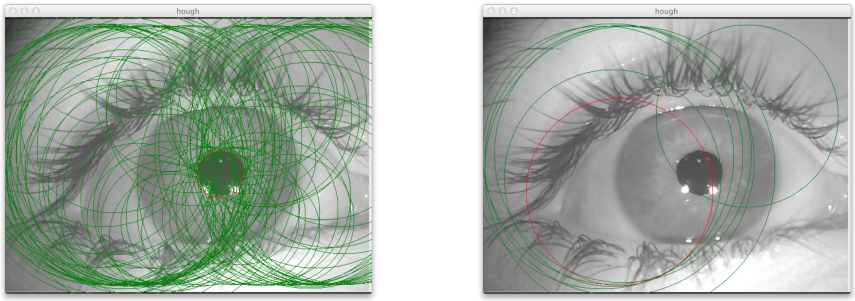
Gauss kernel size The size of the gaussian kernel that is applied to the image before the Hough transformation

σ - value the standard deviation value used to construct the gaussian kernel.

Accumulator threshold The minimum cumulative vote threshold in which some parameters for a circle are considered.

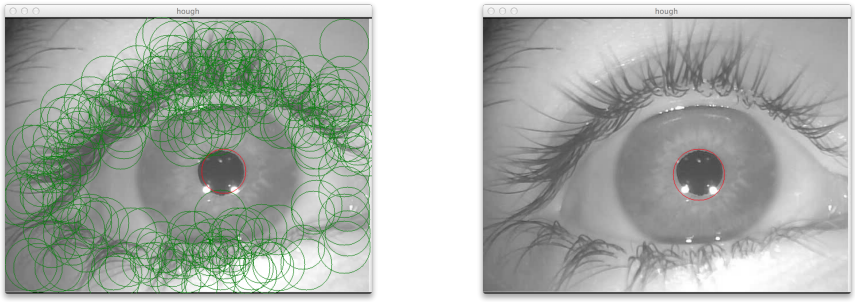
Minimum and Maxiumum Radius The minumum and maximum radius of circles to consider.

The next step is to experimentally find the parameters that yields the best result over all sequences.



(a) Accumulator threshold at 100 (b) Accumulator threshold at 150

Figure 12: Finding the accumulator threshold values



(a) Size=9, $\sigma = 9$ (b) Size=31, $\sigma = 9$

Figure 13: Preprocessing by smoothing with a gaussian kernel

Discussion The circular Hough transformation yields the most robust pupil detection we have been able to produce so far.

The intuition behind this is that in an ideal setting, where the eye for example is not distorted by perspective, the pupil is going to be near-circular, so the process of Hough tranforming and then voting for circles is likely to succeed.

In a non ideal setting, for example in a frame were the eye is seen from the side the pupil is not going to yield the same circular properties, and the process is going to fail.

By preprocessing the image by smooting some of the noise is going to be filtered out. The idea is to choose a kernel that is rougly of the same size as the pupil. In this manner smaller features, such as eye lashes will be smoothed away, but still maintaining the pupil feature.

The drawback of setting a constant size for the gaussian kernel is that is is not scale independent. An ideal kernel in some frame may smoothe away the pupil in some other frame if the had subject moved closer or further away from the camera.

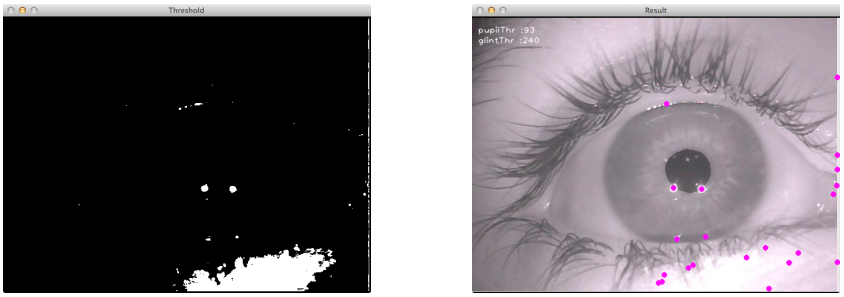
3 Glint Detection

In this section, we will investigate and discuss glint detection.

3.1 Thresholding

Like pupil detection thresholding seems like an obvious place to start. The methods are almost identical. By first creating a binary image from a threshold value, and then do a BLOB analysis, resulting in a set of points where glints are present.

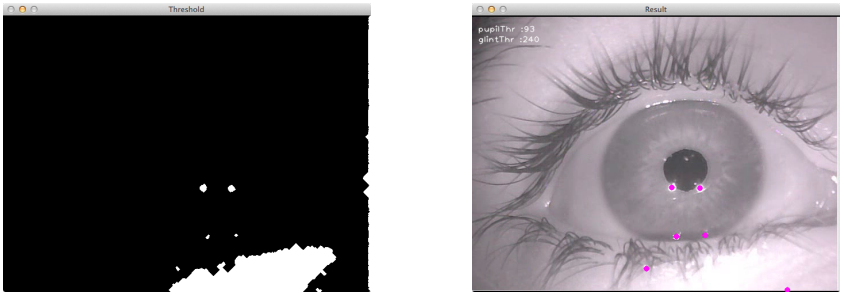
Figure 14 shows the glints in 'eye1.avi'. Because glints are very close to white, a fairly high threshold gives a good result. Furthermore by experimenting we found that, by requiring that the BLOB area lies in the interval $[10 : 150]$, we exclude a great deal of unwanted glint detections, it is although clear that there is still a good amount of false positives.



(a) Binary image (b) Resulting image

Figure 14: Glint detection with threshold of 240

Morphology To mitigate the false positives, we make use of morphology to remove small "spots" of white by first closing and then opening, we get rid of a lot of unwanted glints, as can be seen in Figure 15



(a) Binary image (b) Resulting image

Figure 15: Glint detection with threshold of 240 - using morphology

Filter glints using eye features To further enhance glint detection we have added a function function that excludes glints that have an Euclidian distance from the center of the pupil greater than the largest radius of the elipse representing the pupil. Figure 16

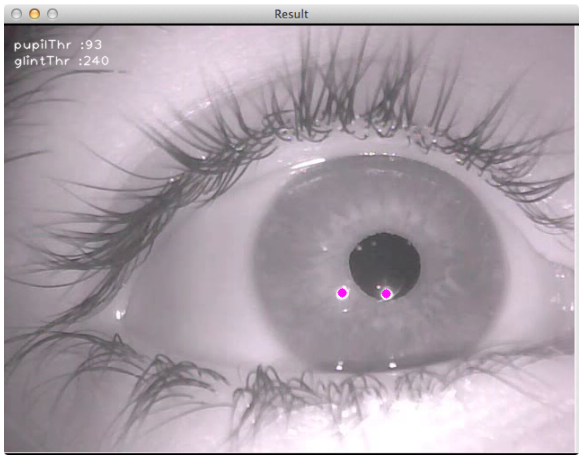
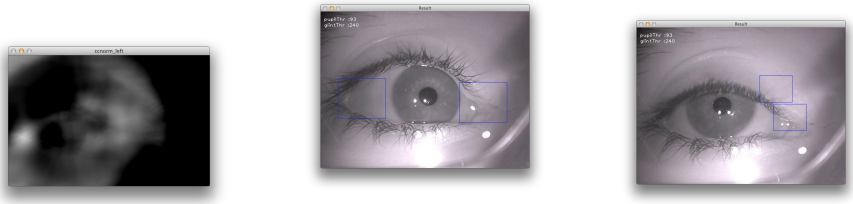


Figure 16: Glint detection with threshold of 240 - using morphology and filter

Discussion The method works very well, but fail when more than one pupil is detected, in other words the stability of glint detection is dependent the quality of the pupil detection.

4 Eye Corner Detection

We detect eye corners simply by template matching. We use template matching by normalized cross-correlation, which is invariant to lighting and exposure conditions, at the expense of performance. Because of the large changes in intensities in the different sequences, we find this to be the preferred method.



(a) image

(b) A good template match of both the left and right patch

(c) A bad match of the left template

Figure 17: Template Matching

Examples The technique works best in a stable environment. Thus if the test subject moves his/her head on any of the 3 axis, or closer/further away

from the camera, the chosen template will of course produce a lower result at the position of the actual eye corner.

The changes in scale can be mitigated by using an image pyramid, i.e. convolving the template with multiple versions of a downsampled or upscaled version of the input frame.

Changes in rotation can be mitigated in a similar manner by rotating the template in a number of steps around its own axis.

Changes in perspective, i.e if the person turns his/her head, could possibly be mitigated by transforming the image by a homography.

The methods can be combined to detect a change in both scale and rotation. A problem with these techniques is that they introduce some computational complexity.

5 Iris / Limbus Detection

5.1 Iris detection using thresholding

We experimented with simple thresholding, to detect the shape and position of the iris. What we found, was that it will work under ideal circumstances, but is extremely brittle with regards to chosen threshold and system lighting. Furthermore, because the iris intensities lies in the middle range (with glints being very bright, and pupil being very dark), many false positives were being picked up in the skin-tone range. This experiment yielded no usable results, and are not discussed further.

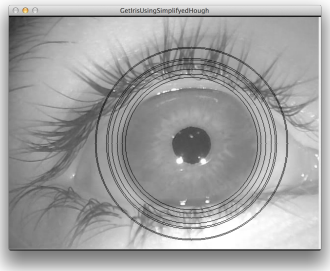
5.2 Hough Transformation with position prior

If we assume that we are somehow able to detect the pupil in a robust way, we can use this position as input to a circular Hough tranformation.

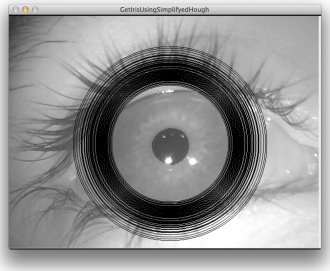
In this way the Hough tranformation will be one dimensional since the only free variable is the radius of the circle.

The procedure is then to iterate over the circles in a range between some minimum and maximum radius with some step size. We then discretize the periphery of the circle and iterate over this discretization. We then increment the value of the current radius, in a parametric accumulator table, if the corresponding point in the Canny edge image is larger than zero.

The radius with the highest value in the accumulator table is thus the circle that yields the best fit.



(a) One correct circle and some false positives



(b) Multiple circular matches

Figure 18: Detecting circles using Hough transformation with position prior

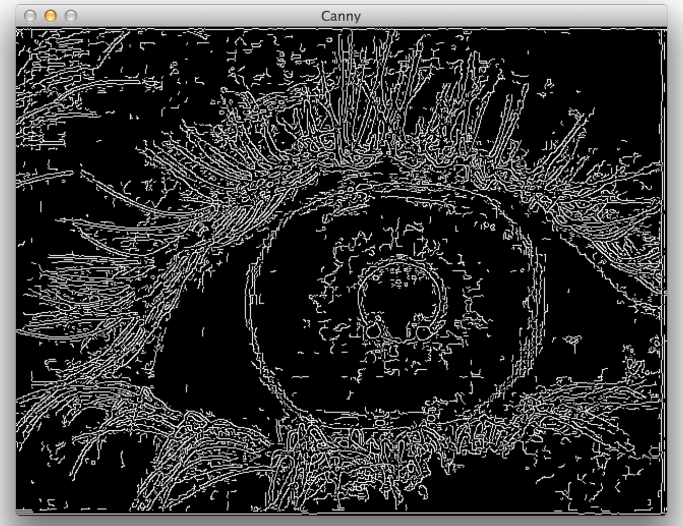


Figure 19: The canny edge image on which the circles in Figure 18a are found

Examples In Figure 18a the method positively identifies the limbus and some false positives.

In Figure 18b the method yields more circles above some given threshold since the perspective distortion exhibits a limbus that appears to be more circular than it actually is.

Discussion The method works but is sensitive on numerous parameters. The threshold should be carefully chosen, and is dependent on the granularity of the Canny edge image it takes as input. A canny edge image with a lot

of edges yields an accumulator table with more votes, and thus more iris candidates.

Looking at Figure 19 we see that one possible improvement could be to smoothe the image before producing the Canny edge image. This would remove the noise produced by eye lashes and other non-circular features.

6 Conclusion

After ample experimentation, we find that the following techniques perform best:

Pupil Hough transform gives a robust pupil detection, that works on image gradient, instead of actual intensity. This makes it much more tolerant of changing lighting conditions. We preprocess the input image with a Gaussian blur, to minimize the effect of noise.

Eye Corners We have only experimented with Template Matching for eye corner detection. We have not attempted to use a generic set of templates across multiple sequences, but have settled with having to define templates for each sequence.

Glints Our thresholding approach is extremely good at finding all the highlights of the image, but results in many false positives. Filtering those with blob detection eliminates most of the false positives. Further, requiring that the glints are close to the pupil center, results in a very robust glint detection (given that the pupil is correctly detected)

Iris / Limbus We use our own, simplified Hough transform, that assumes the pupil center has been found correctly, and that the limbus is approximately circular. This is the least robust of our feature detections, but works well enough in practice to be useful.

Performance We have recorded our own sequence: 'Julie.avi', using an iSlim 321r camera with infrared capabilities. Figure 20 shows an image from 'Julie.avi' where detection is successful, and an image where the detection fails. Enclosed on the CD-ROM is the full sequence 'Julie.avi' (unedited recording), and 'Julie_Detection.avi' that shows the performance of our eye tracker across the entire sequence.

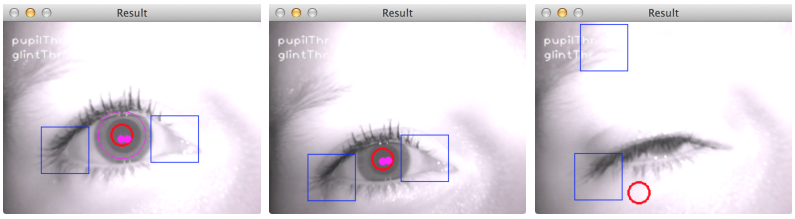


Figure 20: Left: working detection, middle: Iris detection failed, right: Totally failed detection (blink)

There is certainly room for improvement, especially in the areas of eye corner and iris detection. But overall we are happy with the performance of our eye tracker, especially the various Hough transforms we apply, perform very well.

Appendix

6.1 Assignment1.py

The code is available online at:
https://github.com/MartinFaartoft/sigb/blob/master/ass1/Assignment1.py

```
import cv2
import cv
import pylab
import math
from SIGBTools import RegionProps
from SIGBTools import getLineCoordinates
from SIGBTools import ROISelector
from SIGBTools import getImageSequence
from SIGBTools import getCircleSamples
import SIGBTools
import numpy as np
import sys
from scipy.cluster.vq import *
from scipy.misc import *
from matplotlib.pyplot import *


inputFile = "own_Sequences/julie.avi"
outputFile = "eyeTrackerResult.mp4"

#seems to work okay for eye1.avi
default_pupil_threshold = 93

#-------------------------
#          Global variable
#-------------------------
global imgOrig,leftTemplate,rightTemplate,frameNr
imgOrig = [];
#These are used for template matching
leftTemplate = []
rightTemplate = []
frameNr =0;


def GetPupil(gray,thr, min_val, max_val):
  '''Given a gray level image, gray and threshold value return a
     list of pupil locations'''
  #tempResultImg = cv2.cvtColor(gray,cv2.COLOR_GRAY2BGR) #used
    to draw temporary results

  #Threshold image to get a binary image

  cv2.imshow("TempResults", gray)
  val,binI =cv2.threshold(gray, thr, 255, cv2.THRESH_BINARY_INV)
  #print val
```

```
45    #Morphology (close image to remove small 'holes' inside the
        pupil area)
      st = cv2.getStructuringElement(cv2.MORPH_CROSS,(5,5))
47    #binI = cv2.morphologyEx(binI, cv2.MORPH_OPEN, st, iterations
        =10)
      #binI = cv2.morphologyEx(binI, cv2.MORPH_CLOSE, st, iterations
        =5)
49    cv2.imshow("Threshold",binI)
      #Calculate blobs, and do edge detection on entire image (
        modifies binI)
51    contours, hierarchy = cv2.findContours(binI, cv2.RETR_LIST,
        cv2.CHAIN_APPROX_SIMPLE)

53    pupils = [];
      prop_calc = RegionProps()
55    centroids = []
      for contour in contours:
57      #calculate centroid, area and 'extend' (compactness of
        contour)
        props = prop_calc.CalcContourProperties(contour, ["centroid"
        , "area", "extend"])
59      x, y = props["Centroid"]
        area = props["Area"]
61      extend = props["Extend"]
        #filter contours, so that their area lies between min_val
        and max_val, and then extend lies between 0.4 and 1.0
63      if (area > min_val and area < max_val and extend > 0.5 and
        extend < 1.0):
          pupilEllipse = cv2.fitEllipse(contour)
65        # center, radii, angle = pupilEllipse
          # max_radius = max(radii)
67        # c_x = int(center[0])
          # c_y = int(center[1])
69        # cv2.circle(tempResultImg,(c_x,c_y), int(max_radius),
        (0,0,255),4) #draw a circle
          #cv2.ellipse(tempResultImg, pupilEllipse,(0,255,0),1)
71        pupils.append(pupilEllipse)

73    #cv2.imshow("TempResults",tempResultImg)

75    return pupils

77  def GetGlints(gray,thr):
      min_area = 2
79    max_area = 150
      ''' Given a gray level image, gray and threshold
81    value return a list of glint locations'''
      #print thr
83    val, binary_image = cv2.threshold(gray, thr, 255, cv2.
        THRESH_BINARY)


85
      st = cv2.getStructuringElement(cv2.MORPH_CROSS,(3,3))
```

```
87    #binary_image = cv2.morphologyEx(binary_image, cv2.MORPH_CLOSE
        , st, iterations=8)
      #binary_image = cv2.morphologyEx(binary_image, cv2.MORPH_OPEN,
         st, iterations=2)
89
91    #cv2.imshow("Threshold", binary_image)
      #Calculate blobs, and do edge detection on entire image (
        modifies binI)
93    contours, hierarchy = cv2.findContours(binary_image, cv2.
        RETR_LIST, cv2.CHAIN_APPROX_SIMPLE)

95    glints = [];
      prop_calc = RegionProps()
97    centroids = []
      for contour in contours:
99
        #calculate centroid, area and 'extend' (compactness of
        contour)
101     props = prop_calc.CalcContourProperties(contour, ["centroid"
        , "area", "extend"])
        x, y = props["Centroid"]
103     area = props["Area"]
        extend = props["Extend"]
105     print x, y, area, extend

107     #filter contours, so that their area lies between min_val
        and max_val, and then extend lies between 0.4 and 1.0
        if area > min_area and area < max_area: #and extend > 0.4
        and extend < 1.0):
109        glints.append((x,y))

111       #cv2.circle(tempResultImg,(int(x),int(y)), 2, (0,0,255),4)
        #draw a circle
      #cv2.imshow("TempResults",tempResultImg)
113
      return glints
115
  def GetIrisUsingThreshold(gray, thr, min_val, max_val):
117   ''' Given a gray level image, gray and threshold
      value return a list of iris locations'''
119   val,binary_image = cv2.threshold(gray, thr, 255, cv2.
        THRESH_BINARY_INV)
      cv2.imshow("Threshold", binary_image)
121
      contours, hierarchy = cv2.findContours(binary_image, cv2.
        RETR_LIST, cv2.CHAIN_APPROX_SIMPLE)
123
      irises = [];
125   prop_calc = RegionProps()
      centroids = []
127   for contour in contours:
        #calculate centroid, area and 'extend' (compactness of
        contour)
```

```
129     props = prop_calc.CalcContourProperties(contour, ["centroid"
        , "area", "extend"])
        x, y = props["Centroid"]
131     area = props["Area"]
        extend = props["Extend"]
133     #filter contours, so that their area lies between min_val
        and max_val, and then extend lies between 0.4 and 1.0
        if area > min_val and area < max_val and extend > 0.5 and
        extend < 1.0:
135       irisEllipse = cv2.fitEllipse(contour)
          # center, radii, angle = pupilEllipse
137       # max_radius = max(radii)
          # c_x = int(center[0])
139       # c_y = int(center[1])
          # cv2.circle(tempResultImg,(c_x,c_y), int(max_radius),
        (0,0,255),4) #draw a circle
141       #cv2.ellipse(tempResultImg, pupilEllipse,(0,255,0),1)
          irises.append(irisEllipse)
143   return irises

145 def circularHough(gray):
    ''' Performs a circular hough transform of the image, gray and
       shows the  detected circles
147   The circe with most votes is shown in red and the rest in
       green colors '''
    #See help for http://opencv.itseez.com/modules/imgproc/doc/
       feature_detection.html?highlight=houghcircle#cv2.HoughCircles
149   blur = cv2.GaussianBlur(gray, (13,13), 9)

151   dp = 6; minDist = 30
      highThr = 10 #High threshold for canny
153   accThr = 20; #accumulator threshold for the circle centers at
       the detection stage. The smaller it is, the more false
       circles may be detected
      maxRadius = 10;
155   minRadius = 20;
      circles = cv2.HoughCircles(blur,cv2.cv.CV_HOUGH_GRADIENT, dp,
       minDist, None, highThr,accThr,maxRadius, minRadius)
157   #Make a color image from gray for display purposes
      gColor = cv2.cvtColor(gray, cv2.COLOR_GRAY2BGR)
159   if (circles !=None):
        #print circles
161     all_circles = circles[0]
        M,N = all_circles.shape
163     k=1
        #for c in all_circles:
165     # cv2.circle(gColor, (int(c[0]),int(c[1])),c[2], (int(k*255/
        M),k*128,0))
        # K=k+1
167     c=all_circles[0,:]
        cv2.circle(gColor, (int(c[0]),int(c[1])),c[2], (0,0,255),5)
169     #cv2.imshow("hough",gColor)
      return [c]
171
```

```
173 def simplifiedHough(edgeImage,circleCenter,minR,maxR,N,thr):
    samplePoints  = 200
175   accumulator_line = {}

177   for radius in range(minR, maxR, N):
        points = getCircleSamples(center=circleCenter, radius=radius
      , nPoints=samplePoints)
179     accumulator_line[radius] = 0
        for point in points:
181       try:
            if edgeImage[point[0], point[1]] > 0:
183           accumulator_line[radius] += 1
          except IndexError:
185         continue
    radii = []
187   for radius in accumulator_line:
        value = accumulator_line[radius]
189     if value > thr:
          radii.append(radius)
191       print value
    return radii
193


195
  def GetIrisUsingNormals(gray,pupil,normalLength):
197   ''' Given a gray level image, gray and the length of the
       normals, normalLength
      return a list of iris locations'''
199   # YOUR IMPLEMENTATION HERE !!!!
    pass
201
  def GetIrisUsingSimplifyedHough(gray,pupil):
203   ''' Given a gray level image, gray
    return a list of iris locations using a simplified Hough
       transformation'''
205   if pupil != None:
        edges = cv2.Canny(gray, 30, 20)
207     pupil = pupil
        pupil_x = int(pupil[0])
209     pupil_y = int(pupil[1])
        cv2.imshow("edges", edges)
211     radii = simplifiedHough(edges, pupil, 10, 100, 1, 50)
        irises = []
213     for radius in radii:
          #cv2.circle(gray, (pupil_x, pupil_y), radius, (0, 0, 0),
       1)
215       #cv2.circle(gray, pupil, radius, (127,127,127), 2)
          #cv2.imshow("GetIrisUsingSimplifyedHough", gray)
217       irises.append((pupil_x,pupil_y,radius))
    return irises
219


221
```

```
def plotVectorField(I):
  g_x = cv2.Sobel(I, cv.CV_64F, 1,0, ksize=3)
  g_y = cv2.Sobel(I, cv.CV_64F, 0,1, ksize=3) # ksize=3 som **
    kwargs

  x_orig_dim, y_orig_dim = I.shape
  x_mesh_dim, y_mesh_dim = (3, 3)

  sample_g_x = g_x[0:x_orig_dim:x_mesh_dim,0:y_orig_dim:
    x_mesh_dim]
  sample_g_y = g_y[0:x_orig_dim:x_mesh_dim,0:y_orig_dim:
    x_mesh_dim]

  quiver(sample_g_x, sample_g_y)
  show()


def getGradientImageInfo(I):
  g_x = cv2.Sobel(I, cv.CV_64F, 1,0)
  g_y = cv2.Sobel(I, cv.CV_64F, 0,1) # ksize=3 som **kwargs

  X,Y = I.shape
  orientation = np.zeros(I.shape)
  magnitude = np.zeros(I.shape)
  sq_g_x = cv2.pow(g_x, 2)
  sq_g_y = cv2.pow(g_y, 2)
  fast_magnitude = cv2.pow(sq_g_x + sq_g_y, .5)

  # for x in range(X):
  #   for y in range(Y):
  #     orientation[x][y] = np.arctan2(g_y[x][y], g_x[x][y]) *
    (180 / math.pi)
  #     magnitude[x][y] = math.sqrt(g_y[x][y] ** 2 + g_x[x][y]
    ** 2)


  #print fast_magnitude[0]
  #print magnitude[0]

  return fast_magnitude,orientation

def GetEyeCorners(orig_img, leftTemplate, rightTemplate,
    pupilPosition=None):
  if leftTemplate != [] and rightTemplate != []:
    ccnorm_left = cv2.matchTemplate(orig_img, leftTemplate, cv2.
    TM_CCOEFF_NORMED)
    ccnorm_right = cv2.matchTemplate(orig_img, rightTemplate,
    cv2.TM_CCOEFF_NORMED)

    minVal, maxVal, minLoc, maxloc_left_from = cv2.minMaxLoc(
    ccnorm_left)
    minVal, maxVal, minLoc, maxloc_right_from, = cv2.minMaxLoc(
    ccnorm_right)

```

```
    l_x,l_y = leftTemplate.shape
    max_loc_left_from_x = maxloc_left_from[0]
    max_loc_left_from_y = maxloc_left_from[1]

    max_loc_left_to_x = max_loc_left_from_x + l_x
    max_loc_left_to_y = max_loc_left_from_y + l_y

    maxloc_left_to = (max_loc_left_to_x, max_loc_left_to_y)

    r_x,r_y = leftTemplate.shape
    max_loc_right_from_x = maxloc_right_from[0]
    max_loc_right_from_y = maxloc_right_from[1]

    max_loc_right_to_x = max_loc_right_from_x + r_x
    max_loc_right_to_y = max_loc_right_from_y + r_y
    maxloc_right_to = (max_loc_right_to_x, max_loc_right_to_y)

    return (maxloc_left_from, maxloc_left_to, maxloc_right_from,
     maxloc_right_to)

bgr_yellow = 0,255,255
bgr_blue = 255, 0, 0
bgr_red = 0, 0, 255
def circleTest(img, center_point):
  nPts = 20
  circleRadius = 100
  P = getCircleSamples(center=center_point, radius=circleRadius,
     nPoints=nPts)
  for (x,y,dx,dy) in P:
    point_coords = (int(x),int(y))
    cv2.circle(img, point_coords, 2, bgr_yellow, 2)
    cv2.line(img, point_coords, center_point, bgr_yellow)

def findEllipseContour(img, gradient_magnitude,
    gradient_orientation, estimatedCenter, estimatedRadius, nPts
    =30):
  center_point_coords = (int(estimatedCenter[0]), int(
    estimatedCenter[1]))
  P = getCircleSamples(center = estimatedCenter, radius =
    estimatedRadius, nPoints=nPts)
  for (x,y,dx,dy) in P:
    point_coords = (int(x),int(y))
    cv2.circle(img, point_coords, 2, bgr_yellow, 2)
    cv2.line(img, point_coords, center_point_coords, bgr_yellow)

  newPupil = np.zeros((nPts,1,2)).astype(np.float32)
  t = 0
  for (x,y,dx,dy) in P:
    #< define normalLength as some maximum distance away from
    initial circle >
    #< get the endpoints of the normal -> p1,p2>
    point_coords = (int(x),int(y))
    normal_gradient = dx, dy
    #cv2.circle(img, point_coords, 2, bgr_blue, 2)
```

```
313         max_point = findMaxGradientValueOnNormal(gradient_magnitude,
             gradient_orientation, point_coords, center_point_coords,
            normal_gradient)
            cv2.circle(img, tuple(max_point), 2, bgr_red, 2) #locate the
             max points
315         #< store maxPoint in newPupil>
            newPupil[t] = max_point
317         t += 1
        #<fitPoints to model using least squares- cv2.fitellipse(
          newPupil)>
319     return cv2.fitEllipse(newPupil)

321 def findMaxGradientValueOnNormal(gradient_magnitude,
        gradient_orientation, p1, p2, normal_orientation):
        #Get integer coordinates on the straight line between p1 and
         p2
323     pts = SIGBTools.getLineCoordinates(p1, p2)
        values = gradient_magnitude[pts[:,1],pts[:,0]]
325     #orientations = gradient_orientation[pts[:,1],pts[:,0]]
        #normal_angle = np.arctan2(normal_orientation[1],
          normal_orientation[0]) * (180 / math.pi)
327
        # orientation_difference = abs(orientations - normal_angle)
329     # print orientation_difference[0:10]
        # max_index = 0 #np.argmax(values)
331     # max_value = 0
        # for index in range(len(values)):
333     #   if orientation_difference[index] < 20:
        #     if values[index] > max_value:
335     #       max_index = index
        #       max_value = values[index]
337     #print orientations[max_index], normal_angle
        max_index = np.argmax(values)
339     return pts[max_index]
        #return coordinate of max value in image coordinates
341
    def FilterPupilGlint(pupils,glints):
343     ''' Given a list of pupil candidates and glint candidates
          returns a list of pupil and glints'''
        filtered_glints = []
345     filtered_pupils = pupils
        for glint in glints:
347       for pupil in pupils:
            if (is_glint_close_to_pupil(glint, pupil)):
349           filtered_glints.append(glint)

351     return filtered_pupils, filtered_glints

353 def is_glint_close_to_pupil(glint, pupil):
        x, y, radius = pupil
355     center = (x,y)
        distance = euclidianDistance(center, glint)
357     return (distance < radius* 1.5)
```

```
359 def filterGlintsIris(glints, irises):
    new_glints = []
361   if glints and irises:
        for glint in glints:
363       for iris in irises:
            iris_x, irix_y, iris_radius = iris
365         print glint
            iris_vector = np.array([iris_x, irix_y])
367         distance = np.linalg.norm(glint - iris_vector)
            if distance < iris_radius:
369           new_glints.append(glint)
          #print iris
371   return new_glints


373


375 def update(I):
    '''Calculate the image features and display the result based
      on the slider values'''
377   #global drawImg
    global frameNr,drawImg, gray
379   img = I.copy()
    sliderVals = getSliderVals()
381   gray = cv2.cvtColor(img,cv2.COLOR_RGB2GRAY)
    gray = cv2.equalizeHist(gray)
383
    # Do the magic
385   #pupils = GetPupil(gray,sliderVals['pupilThr'], sliderVals['
      minSize'], sliderVals['maxSize'])
    pupils = circularHough(gray)
387   glints = GetGlints(gray,sliderVals['glintThr'])
    pupils, glints = FilterPupilGlint(pupils,glints)
389   #irises = GetIrisUsingThreshold(gray, sliderVals['pupilThr'],
      sliderVals['minSize'], sliderVals['maxSize'])

391   K=10
    d=40
393   #labelIm,centroids = detectPupilKMeans(gray,K=K,distanceWeight
      =d,reSize=(70,70))
    #pupils = get_pupils_from_kmean(labelIm,centroids,gray,
      sliderVals['minSize'],sliderVals['maxSize'])
395
    #magnitude, orientation = getGradientImageInfo(gray)
397   if pupils:
        irises = GetIrisUsingSimplifyedHough(gray, pupils[0])
399
    #plotVectorField(gray)
401   #Do template matching
    global leftTemplate
403   global rightTemplate

405   corners = GetEyeCorners(gray, leftTemplate, rightTemplate)

407   #detectPupilHough(gray, 100)
```

```
    #irises = detectIrisHough(gray, 400)
409
    #glints = filterGlintsIris(glints,irises)
411
    #Display results
413 global frameNr,drawImg
    x,y = 10,10
415 #setText(img,(x,y),"Frame:%d" %frameNr)
    sliderVals = getSliderVals()
417
    # for non-windows machines we print the values of the
      threshold in the original image
419 if sys.platform != 'win32':
      step=18
421   cv2.putText(img, "pupilThr :"+str(sliderVals['pupilThr']), (
      x, y+step), cv2.FONT_HERSHEY_PLAIN, 1.0, (255, 255, 255),
      lineType=cv2.CV_AA)
      cv2.putText(img, "glintThr :"+str(sliderVals['glintThr']), (
      x, y+2*step), cv2.FONT_HERSHEY_PLAIN, 1.0, (255, 255, 255),
      lineType=cv2.CV_AA)
423 cv2.imshow('Result',img)

425 #Uncomment these lines as your methods start to work to
      display the result in the
    #original image
427
    #Elipse
429 # for pupil in pupils:
    #   #cv2.ellipse(img,pupil,(0,255,0),1)
431 #   C = int(pupil[0][0]),int(pupil[0][1])
    #Circle
433 for pupil in pupils:
      cv2.circle(img, (int(pupil[0]),int(pupil[1])),pupil[2],
      (0,0,255),2)
435
    #   contour = findEllipseContour(img, magnitude, orientation,
      C, 70)
437 #   cv2.ellipse(img, contour, bgr_red, 1)
    #   cv2.circle(img,C, 2, (0,0,255),1)
439   #circleTest(img, C)
    for glint in glints:
441     C = int(glint[0]),int(glint[1])
        cv2.circle(img,C, 2,(255,0,255),5)
443


445
    if corners:
447   left_from, left_to, right_from, right_to = corners
      cv2.rectangle(img, left_from , left_to, 255)
449   cv2.rectangle(img, right_from , right_to, 255)

451 for iris in irises:
      #cv2.ellipse(img,iris,(0,255,0),1)
453   C = int(iris[0]),int(iris[1])
```

```
      radius = int(iris[2])
455     cv2.circle(img, C, radius, (255,0,255), 1)

457   cv2.imshow("Result", img)

459     #For Iris detection - Week 2
    #
461
    #copy the image so that the result image (img) can be saved in
       the movie
463   drawImg = img.copy()


465
  def printUsage():
467   print "Q or ESC: Stop"
    print "SPACE: Pause"
469   print "r: reload video"
    print 'm: Mark region when the video has paused'
471   print 's: toggle video  writing'
    print 'c: close video sequence'
473
  def run(fileName,resultFile='eyeTrackingResults.avi'):
475
    ''' MAIN Method to load the image sequence and handle user
      inputs'''
477   global imgOrig, frameNr,drawImg, leftTemplate, rightTemplate,
      gray
    setupWindowSliders()
479   props = RegionProps();
    cap,imgOrig,sequenceOK = getImageSequence(fileName)
481   videoWriter = 0

483   frameNr =0
    if(sequenceOK):
485     update(imgOrig)
    printUsage()
487   frameNr=0;
    saveFrames = False
489
    while(sequenceOK):
491     sliderVals = getSliderVals();
      frameNr=frameNr+1
493     ch = cv2.waitKey(1)
      #Select regions
495     if(ch==ord('m')):
        if(not sliderVals['Running']):
497         roiSelect=ROISelector(imgOrig)
          pts,regionSelected= roiSelect.SelectArea('Select eye
      corner',(400,200))
499         if(regionSelected):
            if leftTemplate == []:
501           leftTemplate = gray[pts[0][1]:pts[1][1],pts[0][0]:
      pts[1][0]]
          else:
```

```
503             rightTemplate = gray[pts[0][1]:pts[1][1],pts[0][0]:
    pts[1][0]]

505     if ch == 27:
          break
507     if (ch==ord('s')):
          if((saveFrames)):
509         videoWriter.release()
            saveFrames=False
511         print "End recording"
          else:
513         imSize = np.shape(imgOrig)
            videoWriter = cv2.VideoWriter(resultFile, cv.CV_FOURCC('
    D','I','V','3'), 30.0,(imSize[1],imSize[0]),True) #Make a
    video writer
515         saveFrames = True
            print "Recording..."
517


519
      if(ch==ord('q')):
521     break
      if(ch==32): #Spacebar
523     sliderVals = getSliderVals()
        cv2.setTrackbarPos('Stop/Start','Controls',not sliderVals[
    'Running'])
525   if(ch==ord('r')):
        frameNr =0
527     sequenceOK=False
        cap,imgOrig,sequenceOK = getImageSequence(fileName)
529     update(imgOrig)
        sequenceOK=True
531
      sliderVals=getSliderVals()
533   if(sliderVals['Running']):
        sequenceOK, imgOrig = cap.read()
535     if(sequenceOK): #if there is an image
          update(imgOrig)
537     if(saveFrames):
          videoWriter.write(drawImg)
539 if(videoWriter!=0):
    videoWriter.release()
541     print "Closing videofile..."
#------------------------

543
def detectPupilKMeans(gray,K=2,distanceWeight=2,reSize=(40,40)):
545 ''' Detects the pupil in the image, gray, using k-means
    gray            : grays scale image
547 K               : Number of clusters
    distanceWeight  : Defines the weight of the position
    parameters
549 reSize          : the size of the image to do k-means on
  '''
551 #Resize for faster performance
```

```
    smallI = cv2.resize(gray, reSize)
553 smallI = cv2.GaussianBlur(smallI,(3,3),20)
    M,N = smallI.shape
555 #Generate coordinates in a matrix
    X,Y = np.meshgrid(range(M),range(N))
557 #Make coordinates and intensity into one vectors
    z = smallI.flatten()
559 x = X.flatten()
    y = Y.flatten()
561 O = len(x)
    #make a feature vectors containing (x,y,intensity)
563 features = np.zeros((O,3))
    features[:,0] = z;
565 features[:,1] = y/distanceWeight; #Divide so that the distance
      of position weighs less than intensity
    features[:,2] = x/distanceWeight;
567 features = np.array(features,'f')
    # cluster data
569 centroids,variance = kmeans(features,K)
    #use the found clusters to map
571 label,distance = vq(features,centroids)
    # re-create image from
573 labelIm = np.array(np.reshape(label,(M,N)))
    return labelIm,centroids
575

def get_pupils_from_kmean(labelIm, centroids, gray, min_val,
    max_val):
577 result = np.zeros((labelIm.shape))
    label = np.argmin(centroids[:,0])
579 result[labelIm == label] = [255]
    y,x=gray.shape
581 result = cv2.resize(result,(x,y))
    semi_binI = np.array(result, dtype='uint8')
583 #remove gray elements created from the linear interpolation
    val,binI =cv2.threshold(semi_binI, 0, 255, cv2.THRESH_BINARY)
585 cv2.imshow("Threshold",binI)
    #Calculate blobs, and do edge detection on entire image (
      modifies binI)
587 contours, hierarchy = cv2.findContours(binI, cv2.RETR_LIST,
      cv2.CHAIN_APPROX_SIMPLE)

589 pupils = [];
    prop_calc = RegionProps()
591 for contour in contours:
      #calculate centroid, area and 'extend' (compactness of
      contour)
593   props = prop_calc.CalcContourProperties(contour, ["centroid"
      , "area", "extend"])
      x, y = props["Centroid"]
595   area = props["Area"]
      extend = props["Extend"]
597   #filter contours, so that their area lies between min_val
      and max_val, and then extend lies between 0.4 and 1.0
```

```
        if (area > min_val and area < max_val and extend > 0.4 and
        extend < 1.0):
599         pupilEllipse = cv2.fitEllipse(contour)
            pupils.append(pupilEllipse)
601     return pupils

603 def detectPupilHough(gray, accThr=600):
      #Using the Hough transform to detect ellipses
605     blur = cv2.GaussianBlur(gray, (9,9),9)
      ##Pupil parameters
607     dp = 6; minDist = 10
      highThr = 30 #High threshold for canny
609     #accThr = 600; #accumulator threshold for the circle centers
        at the detection stage. The smaller it is, the more false
        circles may be detected
      maxRadius = 50;
611     minRadius = 30;
      #See help for http://opencv.itseez.com/modules/imgproc/doc/
        feature_detection.html?highlight=houghcircle#cv2.
        HoughCirclesIn thus
613     circles = cv2.HoughCircles(blur,cv2.cv.CV_HOUGH_GRADIENT, dp,
        minDist, None, highThr,accThr,minRadius, maxRadius)
      #Print the circles
615     gColor = cv2.cvtColor(gray, cv2.COLOR_GRAY2BGR)
      pupils = list(circles)
617     if (circles !=None):
        #print circles
619       all_circles = circles[0]
        M,N = all_circles.shape
621       k=1
        for c in all_circles:
623         cv2.circle(gColor, (int(c[0]),int(c[1])),c[2], (int(k*255/
        M),k*128,0))
            K=k+1
625         #Circle with max votes
        c=all_circles[0,:]
627       cv2.circle(gColor, (int(c[0]),int(c[1])),c[2], (0,0,255))
      cv2.imshow("hough",gColor)
629     return pupils

631 def detectIrisHough(gray, accThr=600):
      #Using the Hough transform to detect ellipses
633     blur = cv2.GaussianBlur(gray, (11,11),9)
      ##Pupil parameters
635     dp = 6; minDist = 10
      highThr = 30 #High threshold for canny
637     #accThr = 600; #accumulator threshold for the circle centers
        at the detection stage. The smaller it is, the more false
        circles may be detected
      maxRadius = 150;
639     minRadius = 100;
      #See help for http://opencv.itseez.com/modules/imgproc/doc/
        feature_detection.html?highlight=houghcircle#cv2.
        HoughCirclesIn thus
```

```
641     circles = cv2.HoughCircles(blur,cv2.cv.CV_HOUGH_GRADIENT, dp,
          minDist, None, highThr,accThr,minRadius, maxRadius)
        #Print the circles
643     gColor = cv2.cvtColor(gray, cv2.COLOR_GRAY2BGR)
        irises = []
645     if (circles !=None):
          #print circles
647       all_circles = circles[0]
          M,N = all_circles.shape
649       k=1
          for c in all_circles:
651         irises.append(c)
            cv2.circle(gColor, (int(c[0]),int(c[1])),c[2], (int(k*255/
          M),k*128,0))
653         K=k+1
            #Circle with max votes
655       c=all_circles[0,:]
          cv2.circle(gColor, (int(c[0]),int(c[1])),c[2], (0,0,255))
657     cv2.imshow("hough",gColor)
        return irises
659 #----------------------------
    #          UI related
661 #----------------------------

663 def setText(dst, (x, y), s):
      cv2.putText(dst, s, (x+1, y+1), cv2.FONT_HERSHEY_PLAIN, 1.0,
        (0, 0, 0), thickness = 2, lineType=cv2.CV_AA)
665   cv2.putText(dst, s, (x, y), cv2.FONT_HERSHEY_PLAIN, 1.0, (255,
         255, 255), lineType=cv2.CV_AA)

667 vertical_window_size = 523
    horizontal_window_size = 640
669
    def setupWindowSliders():
671   ''' Define windows for displaying the results and create
        trackbars '''
      cv2.namedWindow("Result")
673   cv2.moveWindow("Result", 0, 0)
      cv2.namedWindow('Threshold')
675   cv2.moveWindow("Threshold", 0, vertical_window_size)
      cv2.namedWindow('Controls')
677   cv2.moveWindow("Controls", horizontal_window_size, 0)
      cv2.resizeWindow('Controls', horizontal_window_size, 0)
679   cv2.namedWindow("TempResults")
      cv2.moveWindow("TempResults", horizontal_window_size,
        vertical_window_size)
681   #Threshold value for the pupil intensity
      cv2.createTrackbar('pupilThr','Controls',
        default_pupil_threshold, 255, onSlidersChange)
683   #Threshold value for the glint intensities
      cv2.createTrackbar('glintThr','Controls', 240, 255,
        onSlidersChange)
685   #define the minimum and maximum areas of the pupil
```

```
    cv2.createTrackbar('minSize','Controls', 20, 2000,
      onSlidersChange)
687 cv2.createTrackbar('maxSize','Controls', 2000,2000,
      onSlidersChange)
    #Value to indicate whether to run or pause the video
689 cv2.createTrackbar('Stop/Start','Controls', 0,1,
      onSlidersChange)

691 def getSliderVals():
    '''Extract the values of the sliders and return these in a
      dictionary'''
693 sliderVals={}
    sliderVals['pupilThr'] = cv2.getTrackbarPos('pupilThr', '
      Controls')
695 sliderVals['glintThr'] = cv2.getTrackbarPos('glintThr', '
      Controls')
    sliderVals['minSize'] = 50*cv2.getTrackbarPos('minSize', '
      Controls')
697 sliderVals['maxSize'] = 50*cv2.getTrackbarPos('maxSize', '
      Controls')
    sliderVals['Running'] = 1==cv2.getTrackbarPos('Stop/Start', '
      Controls')
699 return sliderVals

701 def onSlidersChange(dummy=None):
    ''' Handle updates when slides have changed.
703  This function only updates the display when the video is put
      on pause'''
    global imgOrig;
705 sv=getSliderVals()
    if(not sv['Running']): # if pause
707   update(imgOrig)

709 def euclidianDistance(a,b):
    a_x, a_y = a
711 b_x, b_y = b
    return math.sqrt((a_x - b_x) ** 2 + (a_y - b_y) **2)
713
    #--------------------------
715 #          main
    #--------------------------
717 run(inputFile)
```

Chapter 2

Assignment 2 - Homographies, Textures, Projection and Augmentation

Homographies, Textures, Projection and Augmentation
SIGB Spring 2014

Marcus Gregersen
mabg@itu.dk

Martin Faartoft
mlfa@itu.dk

Mads Westi
mwek@itu.dk

April 23rd 2014
IT University of Copenhagen

Contents

Contents

1 Introduction

In this report we will experiment with homographies, texture mapping, projections and image augmentation. The report is divided into two parts. In the first part we cover homographies (2) and texture mapping (3). In the second part we focus on projections and augmentation (4).

2 Person Location on Map

In this section, we will take a video of a person walking around the ITU atrium, and map his movement onto an overview map of the ITU building. This effectively eliminates the perspective, and allows for easy measurement of speed, acceleration and heading.

2.1 Theory

Both the ground floor of the ITU, and the overview map, can be described as a plane. To transfer a point from one plane to the other, the homography that describes the transformation is needed. A homography is a 3×3 matrix that is capable of describing the transformation between any two planes.

$$H = \begin{bmatrix} h_{1,1} & h_{1,2} & h_{1,3} \\ h_{2,1} & h_{2,2} & h_{2,3} \\ h_{3,1} & h_{3,2} & h_{3,3} \end{bmatrix}$$

When the correct homography is found, points (x, y) in the original plane, can be mapped to points (x', y') in the new plane, by multiplication as follows:

$$\begin{bmatrix} w \cdot x' \\ w \cdot y' \\ w \end{bmatrix} = H \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

The homography has 8 degrees of freedom ($h_{3,3}$ is always 1), meaning that we need to solve 8 equations with 8 unknows, to find H. This is done by choosing at least 4 points in one of the planes, and the corresponding 4 points in the other plane. This is enough to calculate the homography. We used opencv to solve the linear set of equations.[1]

2.2 Experiment

Using the calculated homography, we can map coordinates from the video sequence `sunclipds.avi` to the corresponding coordinates on the overview

[1] We experimented with using just 4 points, but got very inconsistent results, so we ended up using 6 pairs of reference points for the homography calculation

map `ITUMap.bmp`. Figure 1a shows a frame from the video sequence, and the corresponding position of the tracked person on the overview map.

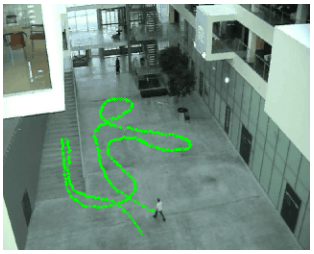
(a)

(b)

Figure 1: The green dot on the map, corresponds to the position of the green box in the video frame.

It is important to note, that this approach only holds for movement within a plane, so when the subject in the video starts moving up the stairs, he leaves the plane that is the floor of the atrium, and the overview map coordinates stops being accurate. In the actual sequence, this is hard to detect, because the stairs have a very small slope.

Figure 2 shows the path taken by the person walking, both in the video view and in the overhead map view.



(a)

(b)

Figure 2: The path described by the tracked person walking.

A video of the views in Figure 1 (a) and (b) called `displaytrace.mp4` can be found in the attached .zip file.

3 Linear texture mapping

Texture mapping on static object

The function `textmapGroundFloor` takes a texture an maps it to each frame of the sequence `sunclipds.avi`. A homography is created using the four corners of the the texture, and four points in the sequence selected by clicking on the first frame. The result is a texture placed 'statically' in the sequence. Figure 3 show two different frames from the sequence.

Figure 3: Linear texture mapping to sequence

The function uses SIGBTools `getHomographyFromMousepoints` to create the homography, it should be noted that the order of the points are essential for the homography to work - the order of the four mouse clicks should be clockwise, the first point corresponds to the top left corner of the texture. Figure 4 illustrates the distortion as a result of click ordering.

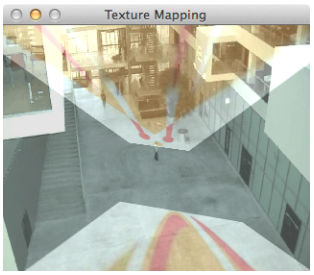


Figure 4: Distorted texture

This approach does not try to make it seem like the person walking, is walking on top of the texture. This effect could be achieved by calculating an average intensity of the texture mapped area over the entire sequence (effectively removing all moving objects), and then only texture map a pixel, if its intensity is close to the average.

4 Augmentation

In the following, we will experiment with, and discuss, different techniques that can be used for augmenting images. (adding objects to an image that seems to behave as if real, with regards to perspective and position).

4.1 Cameras and Calibration

The camera matrix consists of two main parts, the intrinsic and the extrinsic parameters.

The extrinsic parameters describe the rotation and translation needed to transform objects from world coordinates to camera coordinates.

The intrinsic parameters describe the focal length, principal point, and format of the projection plane.

To find the calibration matrix, we capture a frame with a camera pointing at a chessboard figure, with a fixed square size of 2 centimeters.

The OpenCV function `calibrateCamera` can infer the focal length, the distortion coefficients, the translation and rotation matrix from the position of the chessboard squares.

4.2 Camera Matrix via Homography

After calibrating the camera, the following data is available. The camera matrix K, rotation and translation vectors, one for each sample done in the calibration, the first frame of the samples and a set of points corresponding to the corners of the chess board in world coordinates.

The world coordinate system is defined according to the chessboard, meaning that the origin of the coordinate system is defined as the inner corner of one of the corner tiles of the chess board and that the chess board spans the xy-plane of the coordinate system. In other words all points that are on the same plane as the chess board in world coordinates are located in the xy-plane of the world coordinate system, meaning that for all of these points $z = 0$.

Adding to this, that a frame in a video sequence is a 2D representation, meaning a plane, of the real world. This means that if we can find set corresponding points on the chess board and on the current frame, we can create a homography, which can be used to map any 2D structure on the the chess board to the current frame.

Because the transformation is from one plane to another the projection matrix P_1, from the first sample in the calibration, which is known due to the calibration data, the two projection matrices can be related $P_2 = HP_1$ where $H = K[r_1, r_2, t]$, from this it is possible to calculate the third rotation vector $r_3 = r_1 \times r_2$ in the the rotation matrix R, meaning we can recalculate P_2, so that z values in world coordinates can be mapped as well. By using P_2, which is calculated for each frame in a sequence, we can now map any point in world coordinates to that given frame. Figure 5 shows an image augmented with a cube in 3 dimensions, using this technique.

Figure 5: Augmented cube using homography to find the current P

As Figure 5 shows, the technique works, but at extreme angles the map-

ping of the z-coordinates fails, Figure 5 shows this.

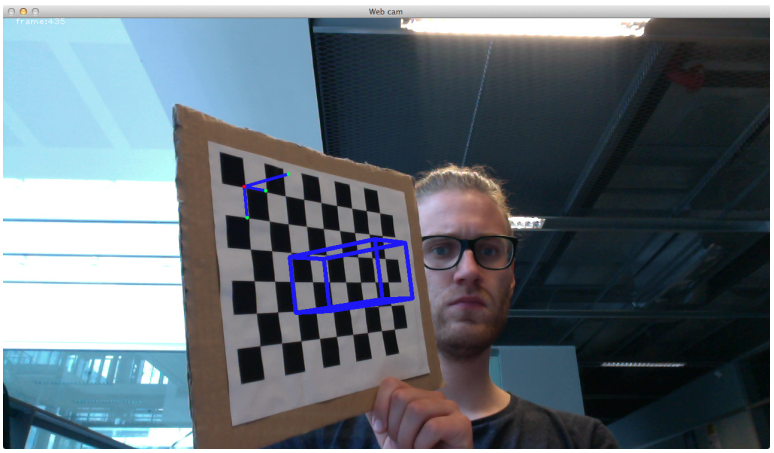


Figure 6: Homography technique failing

4.3 P from Object Pose

The OpenCV function `solvePnP` can estimate the object "pose" (position and orientation in 3D space). To do this, it needs a set of points in the ChessBoard space (x, y, z), the set of corresponding points in the image where we want to determine the pose (x, y), and finally the camera matrix and distortion coefficients found via calibration. The result from `solvePnP`, is a rotation vector - r_{vec}, and a translation vector t_{vec}. The rotation vector can be converted to a rotation matrix, using the `Rodrigues` function.

Given the rotation matrix, R the translation vector, t and the camera matrix, K we can determine P as: $P = K[R|t]$. This gives us a camera that allows us to project directly from the 3-dimensional ChessBoard space (not just plane), to any image where a chessboard pattern can be detected. Figure 7 shows an image augmented with a cube in 3 dimensions, using this technique.

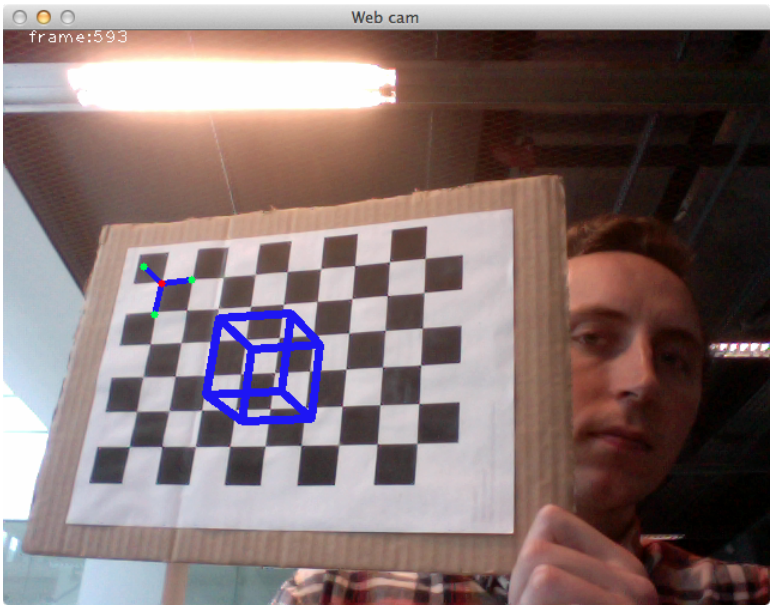


Figure 7: Cube From Pose

The `solvePnP` technique appears to be extremely robust, and produces accurate results in all of our experiments.

4.4 Comparison

As mentioned in the previous sections, using homographies to calculate P, appear to be working in most cases, but at extreme angles the z-dimension in the object is distorted, whereas the object pose, appears robust in all our tests, the conclusion is therefore that the object pose technique is best.

4.5 Alternative Objects

The main challenge is calculating P for the current frame, when this is achieved, it is possible to map any point in world coordinates to the current frame. We have implemented a function called `getPyramidPoints()`, which defines a pyramid, that can be drawn similarly to the cube. More generally, we have implemented `drawObjectScatter`, which takes a list of points in world coordinates.

Using this function we are able to draw a point cloud of the Utah teapot as shown in Figure 8

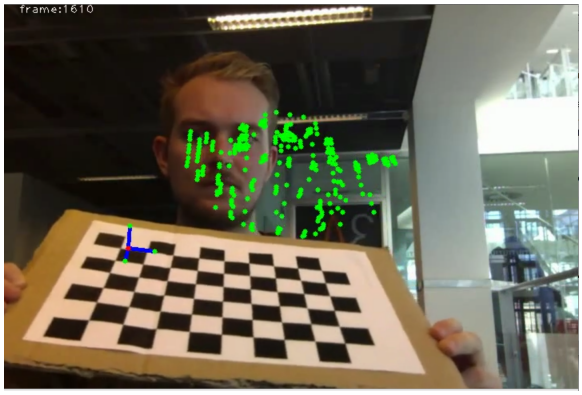


Figure 8: Augmenting the chessboard with the Utah teapot point cloud

A point for further improvement, would be to add the possibility of drawing lines between corresponding points, thus being able to create a wireframe for any object. Pressing the "L"-button toggles the teapot.

4.6 Rotation, Scaling and Translation

Now that we can accurately project arbitrary points from the ChessBoard space onto any image where a chessboard can be found, we can easily transform the objects in ChessBoard coordinates, before projecting.

Our method `transformFigure` accepts as input: a set of points to be transformed, the amount to rotate around the 3 axes, the amount to scale each of the 3 axes by, and a translation vector, and gives as output the transformed points. (Rotation is achieved by using the 3-dimensional basic rotations, scaling by scalar multiplying and translation by addition).

To achieve animation, we can use the current framenumber as input to `transformFigure`, as follows.

```
angle = frameNumber * (math.pi / 50.0)
scale = 2 + math.sin(angle)
```

The result of this, is that each frame will rotate the object by $\theta = \pi/50$ radians, describing a full revolution every 100 frames. The scale of the object will vary smoothly between a factor 1 and a factor 3.

A recorded sequence, showcasing animations, transformations and alternative objects, can be found in the attached .zip file in a file called `projected_objects.mp4`

5 Conclusion

In this report we successfully found a homography from one plane to another, that allowed us to display tracking data from one plane, in the other.

- We used the found homography to map textures onto planes in a realistic fashion, using transparency to heighten the illusion.
- We successfully calibrated a camera, and used this calibration (along with homographies or object poses) to perform real-time augmentation of a video-stream.
- We expanded the augmentation from a static image of a box, to include animations (rotation, scaling and translation) and multiple simultaneous, independent objects.
- We added a new object shape (a pyramid), and implemented point-cloud rendering - exemplified by a teapot.
- We compared two different methods for determining the projection matrix P, and found the one based on object pose to be superior in both precision and robustness.

We are especially proud of our work on animations, but would have liked to extend our point-cloud rendering to draw polygons, instead of just points.

Appendix

https://github.com/MartinFaartoft/sigb

Assignment2.py

```
#from scipy import ndimage
import cv2
import cv
import numpy as np
from pylab import *
from matplotlib import *
from matplotlib.pyplot import *
from scipy import *
import math
import SIGBTools

#found by createHomography with n=6
H = np.array([[  1.47453831e-01,   9.78826731e-01,   2.04270412e
    +02],
       [ -5.10193725e-01,   4.10523270e-01,   1.95623806e+02],
       [  1.70178472e-04,   1.12945293e-03,   1.00000000e+00]])
calibration = [[(34.983870967741922, 208.40322580645164),
    (84.661290322580641, 198.4677419354839), (249.30645161290323,
     162.98387096774195), (296.14516129032256,
    229.69354838709677), (202.4677419354839, 253.82258064516128),
     (172.66129032258067, 180.01612903225805)],
    [(331.11290322580658, 212.53225806451621),
    (331.11290322580658, 187.69354838709694),
    (324.01612903225828, 109.62903225806463),
    (359.50000000000023, 106.08064516129048),
    (366.59677419354853, 148.66129032258073),
    (334.66129032258073, 148.66129032258073)]]

def frameTrackingData2BoxData(data):
    #Convert a row of points into tuple of points for each
    rectangle
    pts= [ (int(data[i]),int(data[i+1])) for i in range(0,11,2)
    ]
    boxes = [];
    for i in range(0,7,2):
        box = tuple(pts[i:i+2])
        boxes.append(box)
    return boxes


def simpleTextureMap():

    I1 = cv2.imread('Images/ITULogo.jpg')
    I2 = cv2.imread('Images/ITUMap.bmp')

    #Print Help
    H, Points  = SIGBTools.getHomographyFromMouse(I1,I2,4)
```

```
    h, w,d = I2.shape
    overlay = cv2.warpPerspective(I1, H,(w, h))
    M = cv2.addWeighted(I2, 0.5, overlay, 0.5,0)

    cv2.imshow("Overlayed Image",M)
    cv2.waitKey(0)

def textureMapGroundFloor():
    #create H_T_G from first frame of sequence
    texture = cv2.imread('Images/ITULogo.jpg')

    fn = "GroundFloorData/sunclipds.avi"
    sequence = cv2.VideoCapture(fn)
    running, frame = sequence.read()

    h_t_g, calibration_points = SIGBTools.getHomographyFromMouse
    (texture, frame, -4)
    print h_t_g
    #fig = figure()
    while running:
        running, frame = sequence.read()

        if not running:
            return

        #texture map
        h,w,d = frame.shape
        warped_texture = cv2.warpPerspective(texture, h_t_g,(w,
    h))
        result = cv2.addWeighted(frame, .7, warped_texture, .3,
    50)

        #display
        cv2.imshow("Texture Mapping", result)
        cv2.waitKey(1)

    #run sequence and map texture onto it according to H_T_G

def showImageandPlot(N):
    #A simple attenmpt to get mouse inputs and display images
    using matplotlib
    I = cv2.imread('groundfloor.bmp')
    drawI = I.copy()
    #make figure and two subplots
    fig = figure(1)
    ax1  = subplot(1,2,1)
    ax2  = subplot(1,2,2)
    ax1.imshow(I)
    ax2.imshow(drawI)
    ax1.axis('image')
    ax1.axis('off')
    points = fig.ginput(5)
    fig.hold('on')

```

```
        for p in points:
            #Draw on figure
            subplot(1,2,1)
            plot(p[0],p[1],'rx')
            #Draw in image
            cv2.circle(drawI,(int(p[0]),int(p[1])),2,(0,255,0),10)
        ax2.cla
        ax2.imshow(drawI)
        draw() #update display: updates are usually defered
        show()
        savefig('somefig.jpg')
        cv2.imwrite("drawImage.jpg", drawI)

blue = 255, 0, 0
green = 0, 255, 0

def displayTrace(squareInVideo, writer=None):
    ituMap = cv2.imread('Images/ITUMap.bmp')
    #print ituMap.shape
    p1, p2 = squareInVideo
    p1_map = multiplyPointByHomography(p1, H)
    p2_map = multiplyPointByHomography(p2, H)
    cv2.rectangle(ituMap, p1_map, p2_map, green)
    cv2.imshow("Tracking", ituMap)
    writer.write(ituMap)
    return ituMap


def multiplyPointByHomography(point, homography):
    #homography = np.linalg.inv(homography)
    p = np.ones(3)
    p[0] = point[0]
    p[1] = point[1]

    p_prime = np.dot(homography, p)
    #print p_prime
    p_prime = p_prime * 1 / p_prime[2]
    #print p,p_prime
    return (int(p_prime[0]), int(p_prime[1]))

def texturemapGridSequence():
    """ Skeleton for texturemapping on a video sequence"""
    fn = 'GridVideos/grid1.mp4'
    cap = cv2.VideoCapture(fn)
    drawContours = True;

    texture = cv2.imread('Images/ITULogo.jpg')
    texture = cv2.pyrDown(texture)


    mTex,nTex,t = texture.shape

    #load Tracking data
    running, imgOrig = cap.read()
```

```
    mI,nI,t = imgOrig.shape

    cv2.imshow("win2",imgOrig)

    pattern_size = (9, 6)

    idx = [0,8,45,53]
    while(running):
    #load Tracking data
        running, imgOrig = cap.read()
        if(running):
            imgOrig = cv2.pyrDown(imgOrig)
            gray = cv2.cvtColor(imgOrig,cv2.COLOR_BGR2GRAY)
            found, corners = cv2.findChessboardCorners(gray,
    pattern_size)
            if found:
                term = ( cv2.TERM_CRITERIA_EPS + cv2.
    TERM_CRITERIA_COUNT, 30, 0.1 )
                cv2.cornerSubPix(gray, corners, (5, 5), (-1, -1)
    , term)
                cv2.drawChessboardCorners(imgOrig, pattern_size,
     corners, found)

                for t in idx:
                    cv2.circle(imgOrig,(int(corners[t,0,0]),int(
    corners[t,0,1])),10,(255,t,t))
            cv2.imshow("win2",imgOrig)
            cv2.waitKey(1)

def calibrateSharpening():
    frame = cv2.imread("failed_frame_224.png")
    new_frame = sharpen(frame)
    found, _ = cv2.findChessboardCorners(new_frame, (9,6))
    print found
    cv2.imshow("sharpened", new_frame)
    cv2.waitKey(0)

def sharpen(gray):
    #sharpening idea 1: use the laplacian to sharpen up the
    image
    #sharpen_mask = copy(gray)
    #cv2.cv.Laplace(cv2.cv.fromarray(gray), cv2.cv.fromarray(
    sharpen_mask), 3)
    #return sharpen_mask + gray

    #sharpening idea 2: subtract a blurred version from the
    original
    blur = cv2.GaussianBlur(gray, (0,0), 10)
    return cv2.addWeighted(gray, 1.5, blur, -.5, 0)

def textureOnGrid():
    texture = cv2.imread('Images/ITULogo.jpg')
    texture = cv2.pyrDown(texture)
    m,n,d = texture.shape
```

```
186     fn = "GridVideos/grid1.mp4"
        sequence = cv2.VideoCapture(fn)
188     running, frame = sequence.read()
        pattern_size = (9, 6)
190     idx = [0,8,53,45]
        frame_no = 1
192     failed = 0
        while running:
194         running, frame = sequence.read()
            frame_no += 1
196         if not running:
                print "FAILED: ", failed
198
            frame = cv2.pyrDown(frame)
200         gray = cv2.cvtColor(frame,cv2.COLOR_BGR2GRAY)
            gray = sharpen(gray)
202         found, corners = cv2.findChessboardCorners(gray,
    pattern_size)

204         #Define corner points of texture
            ip1 = np.array([[0.,0.],[float(n),0.],[float(n),float(m)
    ],[0.,float(m)]])
206
            if not found:
208             failed += 1
                print "GEMMER LIGE"
210             cv2.imwrite("failed_frame_{}.png".format(frame_no),
    frame)
                continue
212
            r = []
214
            c = [(0,0,0),(75,75,75),(125,125,125),(255,255,255)]
216         for i,t in enumerate(idx):
                r.append([float(corners[t,0,0]),float(corners[t
    ,0,1])])
218             cv2.circle(frame,(int(corners[t,0,0]),int(corners[t
    ,0,1])),10,c[i])

220         ip2 = np.array(r)

222         h_t_s,mask = cv2.findHomography(ip1, ip2)

224         #texture map
            h,w,d = frame.shape
226         warped_texture = cv2.warpPerspective(texture, h_t_s,(w,
    h))
            result = cv2.addWeighted(frame, .6, warped_texture, .4,
    50)
228
            #display
230         cv2.imshow("Texture Mapping", result)
            cv2.waitKey(1)
```

```
232
    #run sequence and map texture onto it according to H_T_G
234


236
def realisticTexturemap(H_G_M, scale):
238     map_img = cv2.imread('Images/ITUMap.bmp')
        point = getMousePointsForImageWithParameter(map_img, 1)[0]
240
        texture = cv2.imread('Images/ITULogo.jpg')
242     #texture = cv2.cvtColor(texture,cv2.COLOR_BGR2GRAY)
        H_T_M = np.zeros(9).reshape(3,3)
244     H_T_M[0][0] = scale
        H_T_M[1][1] = scale
246
        H_T_M[0][2] = point[0]
248     H_T_M[1][2] = point[1]

250     H_T_M[2][2] = 1

252     H_M_G = np.linalg.inv(H_G_M)

254     H_T_G = np.dot(H_M_G, H_T_M)

256     fn = "GroundFloorData/sunclipds.avi"
        cap = cv2.VideoCapture(fn)
258     #load Tracking data
        running, frame = cap.read()
260     while running:
            running, frame = cap.read()
262         h,w,d = frame.shape

264         warped_texture = cv2.warpPerspective(texture, H_T_G,(w,
    h))

266         result = cv2.addWeighted(frame, .6, warped_texture, .4,
    50)

268         cv2.imshow("Result", result)
            cv2.waitKey(0)
270

272 def createHomography():
        img1 = cv2.imread('Images/ITUMap.bmp')
274
        fn = "GroundFloorData/sunclipds.avi"
276     cap = cv2.VideoCapture(fn)

278     #load Tracking data
        _, img2 = cap.read()
280
        print SIGBTools.getHomographyFromMouse(img2, img1, 6)
282
```

```
284
def showFloorTrackingData():
286     #Load videodata
        fn = "GroundFloorData/sunclipds.avi"
288     cap = cv2.VideoCapture(fn)
        running, imgOrig = cap.read()
290     first = copy(imgOrig)
        imSize = np.shape(imgOrig)
292     print imSize
        map_img = cv2.imread('Images/ITUMap.bmp')
294
        resultFile = "a.avi"
296     resultFile_2 = "b.avi"
        videoWriter = cv2.VideoWriter(resultFile, cv.CV_FOURCC('D','
        I','V','3'), 30.0,(imSize[1],imSize[0]),True) #Make a video
        writer
298     videoWriter_2 = cv2.VideoWriter(resultFile_2, cv.CV_FOURCC('
        D','I','V','3'), 30.0,(800,348),True) #Make a video writer
        #load Tracking data
300
        dataFile = np.loadtxt('GroundFloorData/trackingdata.dat')
302     m,n = dataFile.shape

304     fig = figure()
        for k in range(m):
306         running, imgOrig = cap.read()
            if(running):
308             boxes= frameTrackingData2BoxData(dataFile[k,:])
                boxColors = [(255,0,0),(0,255,0),(0,0,255)]
310             for kx in range(0,3):
                    aBox = boxes[kx]
312                 cv2.rectangle(imgOrig, aBox[0], aBox[1],
        boxColors[kx])
                cv2.imshow("boxes",imgOrig);
314             videoWriter.write(imgOrig)
                disp_img = displayTrace(boxes[1], videoWriter_2)
316             print boxes[1][0]
                p_vid = boxes[1][0]
318             cv2.circle(first, p_vid, 1, boxColors[1])
                p_map = multiplyPointByHomography(p_vid, H)
320             cv2.circle(map_img, p_map, 1, boxColors[1])

322             #print k
                if k == 1:
324                 cv2.imwrite("displayTrace.png",disp_img)
                cv2.waitKey(1)
326     cv2.imwrite('le_tracking.png', first)
        cv2.imwrite('le_map.png', map_img)
328
def angle_cos(p0, p1, p2):
330     d1, d2 = p0-p1, p2-p1
        return abs( np.dot(d1, d2) / np.sqrt( np.dot(d1, d1)*np.dot(
        d2, d2) ) )
332
```

```
def findSquares(img,minSize = 2000,maxAngle = 1):
    """ findSquares intend to locate rectangle in the image of
    minimum area, minSize, and maximum angle, maxAngle, between
    sides"""
    squares = []
    contours, hierarchy = cv2.findContours(img, cv2.RETR_LIST,
    cv2.CHAIN_APPROX_SIMPLE)
    for cnt in contours:
        cnt_len = cv2.arcLength(cnt, True)
        cnt = cv2.approxPolyDP(cnt, 0.08*cnt_len, True)
        if len(cnt) == 4 and cv2.contourArea(cnt) > minSize and
    cv2.isContourConvex(cnt):
            cnt = cnt.reshape(-1, 2)
            max_cos = np.max([angle_cos( cnt[i], cnt[(i+1) %
    4], cnt[(i+2) % 4] ) for i in xrange(4)])
            if max_cos < maxAngle:
                squares.append(cnt)
    return squares

def DetectPlaneObject(I,minSize=1000):
    """ A simple attempt to detect rectangular
    color regions in the image"""
    HSV = cv2.cvtColor(I, cv2.COLOR_BGR2HSV)
    h = HSV[:,:,0].astype('uint8')
    s = HSV[:,:,1].astype('uint8')
    v = HSV[:,:,2].astype('uint8')

    b = I[:,:,0].astype('uint8')
    g = I[:,:,1].astype('uint8')
    r = I[:,:,2].astype('uint8')

    # use red channel for detection.
    s = (255*(r>230)).astype('uint8')
    iShow = cv2.cvtColor(s, cv2.COLOR_GRAY2BGR)
    cv2.imshow('ColorDetection',iShow)
    squares = findSquares(s,minSize)
    return squares

def texturemapObjectSequence():
    """ Poor implementation of simple texturemap """
    fn = 'BookVideos/Seq3_scene.mp4'
    cap = cv2.VideoCapture(fn)
    drawContours = True;

    texture = cv2.imread('images/ITULogo.jpg')
    #texture = cv2.transpose(texture)
    mTex,nTex,t = texture.shape

    #load Tracking data
    running, imgOrig = cap.read()
    mI,nI,t = imgOrig.shape

    print running
    while(running):
```

```
        for t in range(20):
384         running, imgOrig = cap.read()

386     if(running):
            squares = DetectPlaneObject(imgOrig)
388
            for sqr in squares:
390             #Do texturemap here!!!!
                #TODO
392
                if(drawContours):
394                 for p in sqr:
                        cv2.circle(imgOrig,(int(p[0]),int(p[1]))
    ),3,(255,0,0))
396

398         if(drawContours and len(squares)>0):
                cv2.drawContours( imgOrig, squares, -1, (0, 255,
     0), 3 )
400
            cv2.circle(imgOrig,(100,100),10,(255,0,0))
402         cv2.imshow("Detection",imgOrig)
            cv2.waitKey(1)
404
def getMousePointsForImageWithParameter(image, points=1):
406     '''GUI for inputting 4 points within an images width and
    height

408     image is the image to input points within
    '''
410 #Copy image
    drawImage1 = copy(image)
412
    #Make figure
414 fig = figure("Point selection")
    title("Click 4 places on a plane in the image")
416
    #Show image and request input
418 imshow(drawImage1)
    clickPoints = fig.ginput(points, -1)
420
    #Return points
422 return clickPoints

424 #createHomography()
showFloorTrackingData()
426 #simpleTextureMap()
#textureMapGroundFloor()
428 #realisticTexturemap(H, 0.2)
#texturemapGridSequence()
430 #textureOnGrid()
```

Assignment_Cube.py

```
'''
Created on March 20, 2014

@author: Diako Mardanbegi (dima@itu.dk)
'''
from numpy import *
import numpy as np
from pylab import *
from scipy import linalg
import cv2
import cv2.cv as cv
from SIGBTools import *
import math

def DrawLines(img, points):
    for i in range(1, len(points[0])):
        x1 = points[0, i - 1]
        y1 = points[1, i - 1]
        x2 = points[0, i]
        y2 = points[1, i]
        cv2.line(img, (int(x1), int(y1)), (int(x2), int(y2)),
    (255, 0, 0),5)
    return img

def findChessBoardCorners(image):
    pattern_size = (9, 6)
    flag = cv2.CALIB_CB_FAST_CHECK + cv2.cv.
    CV_CALIB_CB_NORMALIZE_IMAGE
    gray = cv2.cvtColor(image,cv2.COLOR_BGR2GRAY)
    return cv2.findChessboardCorners(gray, pattern_size, flags=
    flag)

def update(img):
    image=copy(img)

    if Undistorting:  #Use previous stored camera matrix and
    distortion coefficient to undistort the image
        ''' <004> Here Undistort the image'''
        image = cv2.undistort(image, cameraMatrix,
    distortionCoefficient)

    if (ProcessFrame):
        ''' <005> Here Find the Chess pattern in the current
    frame'''
        patternFound, corners = findChessBoardCorners(image)

        if patternFound == True:
            ''' <006> Here Define the cameraMatrix P=K[R|t] of
    the current frame'''
            if debug:
                P = findPFromHomography(corners)
            else:
                P = createPCurrentFromObjectPose(corners)
```

```
48          if ShowText:
                ''' <011> Here show the distance between the
    camera origin and the world origin in the image'''
50
                cv2.putText(image,str("frame:" + str(frameNumber
    )), (20,10),cv2.FONT_HERSHEY_PLAIN,1, (255, 255, 255))#Draw
    the text
52
            ''' <008> Here Draw the world coordinate system in
    the image'''
54          cam2 = Camera(P)
            coordinate_system = getCoordinateSystemChessPlane()
56          transformed_coordinate_system =
    projectChessBoardPoints(cam2,coordinate_system)
            drawCoordinateSystem(image,
    transformed_coordinate_system)
58
            if TextureMap:
60
                ''' <010> Here Do he texture mapping and draw
    the texture on the faces of the cube'''
62
                ''' <012>  calculate the normal vectors of the
    cube faces and draw these normal vectors on the center of
    each face'''
64
                ''' <013> Here Remove the hidden faces'''
66
68          if ProjectPattern:
                ''' <007> Here Test the camera matrix of the
    current view by projecting the pattern points'''
70              cam2 = Camera(P)
                X = projectChessBoardPoints(cam2,
    points_from_chess_board_plane)
72
                for p in X:
74                  C = int(p[0]),int(p[1])
                    cv2.circle(image,C, 2,(255,0,255),4)
76
78          if WireFrame:
                ''' <009> Here Project the box into the current
    camera image and draw the box edges'''
80              cam2 = Camera(P)
                if (Teapot):
82                  teapot = parse_teapot()

84                  #rotated_box = rotateFigure(box, 0, 0, angle
    , scale, scale, scale)
                    drawObjectScatter(cam2, image, teapot)
86              ## stop
                else:
```

```
88                      #figure = box if frameNumber % 100 < 50 else
       pyramid

90                      angle = frameNumber * (math.pi / 50.0)
                        scale = 2 + math.sin(angle)
92                      box1 = transformFigure(box, 0, 0, -angle, 1,
       1, 1)
                        box2 = getPyramidPoints([0, 0, -1], 1,
      chessSquare_size)
94                      box2 = transformFigure(box2, 0, 0, angle, 1,
       1, 1)
                       #rotated_box = rotateFigure(figure, 0, 0,
      angle, scale, scale, scale)
96                      drawFigure(image, cam2, box1)
                        drawFigure(image, cam2, box2)
98
       cv2.namedWindow('Web cam')
100    cv2.imshow('Web cam', image)
       videoWriter.write(image)
102    global result
       result=copy(image)
104

106 def drawFigure(image, camera, figure):
       X = figure.T
108    ones = np.ones((X.shape[0],1))
       X = np.column_stack((X,ones)).T
110
       projected_figure = camera.project(X)
112    DrawLines(image,projected_figure)

114 def getImageSequence(capture, fastForward):
       '''Load the video sequence (fileName) and proceeds,
      fastForward number of frames.'''
116    global frameNumber

118    for t in range(fastForward):
           isSequenceOK, originalImage = capture.read()  # Get the
      first frames
120        frameNumber = frameNumber+1
       return originalImage, isSequenceOK
122

124 def printUsage():
       print "Q or ESC: Stop"
126    print "SPACE: Pause"
       print "p: turning the processing on/off "
128    print 'u: undistorting the image'
       print 'g: project the pattern using the camera matrix (test)
       '
130    print 'x: your key!'

132    print 'the following keys will be used in the next
      assignment'
```

```
    print 'i: show info'
    print 't: texture map'
    print 's: save frame'


def run(speed,video):
    '''MAIN Method to load the image sequence and handle user inputs'''
    #------------------------------video
    capture = cv2.VideoCapture(video)
    resultFile = "recording.avi"


    image, isSequenceOK = getImageSequence(capture,speed)
    imSize = np.shape(image)
    global videoWriter
    videoWriter = cv2.VideoWriter(resultFile, cv.CV_FOURCC('D','I','V','3'), 30.0,(imSize[1],imSize[0]),True) #Make a video writer
    if(isSequenceOK):
        update(image)
        printUsage()
    while(isSequenceOK):
        OriginalImage=copy(image)

        inputKey = cv2.waitKey(1)
        if inputKey == 32:#  stop by SPACE key
            update(OriginalImage)
            if speed==0:
                speed = tempSpeed;
            else:
                tempSpeed=speed
                speed = 0;
        if (inputKey == 27) or (inputKey == ord('q')):#  break by ECS key
            break
        if inputKey == ord('p') or inputKey == ord('P'):
            global ProcessFrame
            if ProcessFrame:
                ProcessFrame = False;

            else:
```

```
                ProcessFrame = True;
184             update(OriginalImage)

186         if inputKey == ord('u') or inputKey == ord('U'):
                global Undistorting
188             if Undistorting:
                    Undistorting = False;
190             else:
                    Undistorting = True;
192             update(OriginalImage)
            if inputKey == ord('w') or inputKey == ord('W'):
194             global WireFrame
                if WireFrame:
196                 WireFrame = False;

198             else:
                    WireFrame = True;
200             update(OriginalImage)

202         if inputKey == ord('i') or inputKey == ord('I'):
                global ShowText
204             if ShowText:
                    ShowText = False;
206
                else:
208                 ShowText = True;
                update(OriginalImage)
210
            if inputKey == ord('t') or inputKey == ord('T'):
212             global TextureMap
                if TextureMap:
214                 TextureMap = False;

216             else:
                    TextureMap = True;
218             update(OriginalImage)

220         if inputKey == ord('g') or inputKey == ord('G'):
                global ProjectPattern
222             if ProjectPattern:
                    ProjectPattern = False;
224
                else:
226                 ProjectPattern = True;
                update(OriginalImage)
228
            if inputKey == ord('x') or inputKey == ord('X'):
230             global debug
                if debug:
232                 debug = False;
                else:
234                 debug = True;
                update(OriginalImage)
236
```

```
        if inputKey == ord('l') or inputKey == ord('L'):
238         global Teapot
            Teapot = not Teapot
240         update(OriginalImage)


242
        if inputKey == ord('s') or inputKey == ord('S'):
244         name='Saved Images/Frame_' + str(frameNumber)+'.png'
            cv2.imwrite(name, result)
246
        if (speed>0):
248         update(image)
            image, isSequenceOK = getImageSequence(capture,speed
    )
250
def loadCalibrationData():
252     global translationVectors
    translationVectors = np.load('numpyData/translationVectors.
    npy')
254     global cameraMatrix
    cameraMatrix = np.load('numpyData/camera_matrix.npy')
256     global rotatioVectors
    rotatioVectors = np.load('numpyData/rotatioVectors.npy')
258     global distortionCoefficient
    distortionCoefficient = np.load('numpyData/
    distortionCoefficient.npy')
260     global points_from_chess_board_plane
    points_from_chess_board_plane = np.load('numpyData/
    obj_points.npy')[0]
262     return cameraMatrix,rotatioVectors[0],translationVectors[0]

264 def calculateP(K,r,t):
    R,_ = cv2.Rodrigues(r)
266     Rt = np.hstack((R,t))
    P = np.dot(K,Rt)
268     return P

270 def displayNumpyPoints(C):
    points = np.load('numpyData/obj_points.npy')
272
    img = cv2.imread('01.png')
274
    x = projectChessBoardPoints(C,points[0])
276
    for p in x:
278         C = int(p[0]),int(p[1])
        cv2.circle(img,C, 2,(255,0,255),4)
280
    cv2.imshow('result',img)
282     cv2.waitKey(0)

284 def projectChessBoardPoints(C, X):
    ones = np.ones((X.shape[0],1))
286     X = np.column_stack((X,ones)).T
```

```
    x = C.project(X)
    x = x.T
    return x

def getCoordinateSystemChessPlane(axis_length = 2.0):
    o = [0., 0., 0.]
    x = [axis_length, 0., 0.]
    y = [0., axis_length, 0.]
    z = [0., 0., -axis_length] #positive z is away from camera,
    by default
    return np.array([o,x,y,z])

def drawObjectScatter(C,img, points):
    points = points.T
    ones = np.ones((points.shape[0],1))
    points = np.column_stack((points,ones)).T
    points = C.project(points)
    points = points.T

    for point in points:
        cv2.circle(img, (int(point[0]),int(point[1])), 3, (0,
    255, 0), -1)


def drawCoordinateSystem(img, coordinate_system):
    o = coordinate_system[0]
    x = coordinate_system[1]
    y = coordinate_system[2]
    z = coordinate_system[3]

    cv2.line(img, (int(o[0]),int(o[1])), (int(x[0]),int(x[1]))
    ,(255, 0, 0),3)
    cv2.line(img, (int(o[0]),int(o[1])), (int(y[0]),int(y[1])),
    (255, 0, 0),3)
    cv2.line(img, (int(o[0]),int(o[1])), (int(z[0]),int(z[1])),
    (255, 0, 0),3)

    cv2.circle(img, (int(x[0]),int(x[1])), 3, (0, 255, 0), -1)
    cv2.circle(img, (int(y[0]),int(y[1])), 3, (0, 255, 0), -1)
    cv2.circle(img, (int(z[0]),int(z[1])), 3, (0, 255, 0), -1)
    cv2.circle(img, (int(o[0]),int(o[1])), 3, (0, 0, 255), -1)

def createPCurrentFromObjectPose(corners):
    found, r_vec, t_vec = cv2.solvePnP(
    points_from_chess_board_plane, corners, cameraMatrix,
    distortionCoefficient)
    return calculateP(cameraMatrix, r_vec, t_vec)

def findPFromHomography(corners_current):
    cam1 = C

    img = cv2.imread("01.png")
    _, corners_1 = findChessBoardCorners(img)
    H,_ = cv2.findHomography(corners_1, corners_current)
```

```
334
        cam2 = Camera(np.dot(H,cam1.P))
336     A = np.dot(linalg.inv(K),cam2.P[:,:3])

338     r1 = A[:,0]
        r2 = A[:,1]
340     r3 = np.cross(r1,r2)
        r3 = r3/np.linalg.norm(r3)
342
        A = np.array([r1,r2,r3]).T
344     cam2.P[:,:3] = np.dot(K,A)
        return cam2.P
346
    def parse_teapot():
348     points = []
        with open("teapot.data", "r") as infile:
350         lines = infile.read().splitlines()
            for line in lines:
352             line = line.split(",")

354             x = float(line[0]) + 5
                y = float(line[1]) + 5
356             z = (float(line[2]) * -1) - 5
                points.append([x, y, z])
358
        result =  np.array(points).T
360     return result * 2

362 def transformFigure(figure, theta_x, theta_y, theta_z, scale_x,
        scale_y, scale_z):
        translate_to = [8, 6, -1]
364

366     rotation_matrix_x = np.array([ [1, 0, 0], [0, cos(theta_x),
        -sin(theta_x)], [0, sin(theta_x), cos(theta_x)] ])
        rotation_matrix_y = np.array([ [cos(theta_y), 0, sin(theta_y
        )], [0, 1, 0], [-sin(theta_y), 0, cos(theta_y)] ])
368     rotation_matrix_z = np.array([ [cos(theta_z), -sin(theta_z),
         0], [sin(theta_z), cos(theta_z), 0], [0, 0, 1] ])

370     rotated_x = []
        rotated_y = []
372     rotated_z = []
        rotation = np.dot(rotation_matrix_x, np.dot(
        rotation_matrix_y, rotation_matrix_z))
374     for i in range(len(figure[0])):
            p = np.array([figure[0][i], figure[1][i], figure[2][i]])
376         p_rot = np.dot(rotation, p)
            rotated_x.append(scale_x * p_rot[0] + translate_to[0])
378         rotated_y.append(scale_y * p_rot[1] + translate_to[1])
            rotated_z.append(scale_z * p_rot[2] + translate_to[2])
380
        result = np.array([rotated_x, rotated_y, rotated_z])
382     return result
```

```
384
def getPyramidPoints(center, size,chessSquare_size):
386     points = []

388     tl = [center[0]-size, center[1]-size, center[2]]
        bl = [center[0]-size, center[1]+size, center[2]]
390     br = [center[0]+size, center[1]+size, center[2]]
        tr = [center[0]+size, center[1]-size, center[2]]
392     top = [center[0], center[1], center[2] - size * 2]

394     #bottom
        points.append(tl)
396     points.append(bl)
        points.append(br)
398     points.append(tr)
        points.append(tl)
400
        #top
402     points.append(top)

404     #diagonals
        points.append(bl)
406     points.append(br)
        points.append(top)
408     points.append(tr)
        points=dot(points,chessSquare_size)
410     return array(points).T


412


414 '''--------------------MAIN BODY
        ----------------------------------------------------------------------------
        '''
    '''
        ----------------------------------------------------------------------------
        '''
416


418
    '''-------variables-------'''
420 global cameraMatrix
    global distortionCoefficient
422 global homographyPoints
    global calibrationPoints
424 global calibrationCamera
    global chessSquare_size
426
    ProcessFrame=False
428 Undistorting=False
    WireFrame=False
430 ShowText=True
    TextureMap=True
432 ProjectPattern=False
```

```
debug=True
Teapot = True

tempSpeed=1
frameNumber=0
chessSquare_size=2


'''-------defining the figures-------'''

box = getCubePoints([0, 0, 1], 1,chessSquare_size)
pyramid = getPyramidPoints([0, 0, 1], 1,chessSquare_size)


i = array([ [0,0,0,0],[1,1,1,1] ,[2,2,2,2] ]) # indices for
    the first dim
j = array([ [0,3,2,1],[0,3,2,1] ,[0,3,2,1] ]) # indices for
    the second dim
TopFace = box[i,j]


i = array([ [0,0,0,0],[1,1,1,1] ,[2,2,2,2] ]) # indices for
    the first dim
j = array([ [3,8,7,2],[3,8,7,2] ,[3,8,7,2] ]) # indices for
    the second dim
RightFace = box[i,j]


i = array([ [0,0,0,0],[1,1,1,1] ,[2,2,2,2] ]) # indices for
    the first dim
j = array([ [5,0,1,6],[5,0,1,6] ,[5,0,1,6] ]) # indices for
    the second dim
LeftFace = box[i,j]

i = array([ [0,0,0,0],[1,1,1,1] ,[2,2,2,2] ]) # indices for
    the first dim
j = array([ [5,8,3,0], [5,8,3,0] , [5,8,3,0] ]) # indices for
    the second dim
UpFace = box[i,j]


i = array([ [0,0,0,0],[1,1,1,1] ,[2,2,2,2] ]) # indices for
    the first dim
j = array([ [1,2,7,6], [1,2,7,6], [1,2,7,6] ]) # indices for
    the second dim
DownFace = box[i,j]


'''-----------------------------------------'''
'''-----------------------------------------'''


```

```
''' <000> Here Call the calibrateCamera from the SIGBTools to
    calibrate the camera and saving the data'''
#calibrateCamera(5, (9,6), 2.0, 0)
''' <001> Here Load the numpy data files saved by the
    cameraCalibrate2'''
K,r,t = loadCalibrationData()
''' <002> Here Define the camera matrix of the first view image
    (01.png) recorded by the cameraCalibrate2'''

P = calculateP(K,r,t)
C = Camera(P)

''' <003> Here Load the first view image (01.png) and find the
    chess pattern and store the 4 corners of the pattern needed
    for homography estimation'''
#displayNumpyPoints(C)


''' <003a> Find homography H_cs^1 '''

run(1, 0)
#run(1,"sequence.mov")
```

Chapter 3

Assignment 3 - Shading

Assignment 3 - Shading
SIGB Spring 2014

Marcus Gregersen
mabg@itu.dk

Martin Faartoft
mlfa@itu.dk

Mads Westi
mwek@itu.dk

May 18th 2014
IT University of Copenhagen

1 Introduction

In this report, we will implement and document a simple shader, able to render a textured cube, and shade it realistically.

2 Back-Face Culling

When we render the cube, at most 3 of the faces will be visible to camera, regardless of the positions of cube and camera. Attempting to render all 6 will waste a lot of performance, and furthermore require us to render the faces in the correct order, to get the correct perspective. Figure 1 shows a rendered frame where back-face culling has been disabled, and a sequence with disabled back-face culling can be found in the attached .zip file with the name *all_and_no_backface.mp4*.

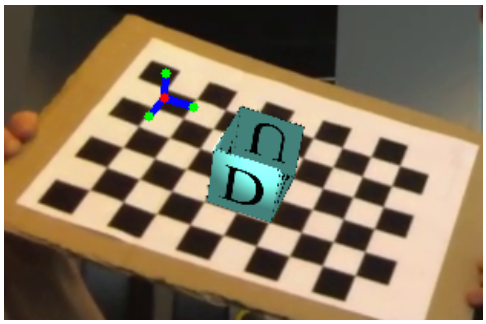


Figure 1: Back-face culling disabled

The idea behind back-face culling, is to calculate the dot-product between the camera view vector, and each of the face normals. If the result is negative, the face should be drawn - otherwise it faces away from the camera, and should be discarded.

$$cos(\theta) = V_{view} \cdot \hat{F}$$

Because the cube is convex, back-face culling is sufficient to ensure the scene will be rendered correctly, assuming that the cube is the only object in the scene.

3 Illumination

The first step in realistic shading, is to calculate the intensity of the light going towards the camera, from each point on each object in the scene.

The most realistic illumination is achieved by using a global illumination model, such as *ray tracing*, that calculates the path of a number of light rays

as they bounce around in the scene. Unfortunately, ray tracing is is really performance intensitive, so most 3D shaders apply a local illumination model instead. Local illumination cuts corners by approximating the illumination, disregarding reflection between objects in the scene (hence *local*).

We use the Phong illumination model in the following, it is a local illumination model, that calculates the illumnation of each point as:

$$I_{Phong} = I_{ambient} + I_{diffuse} + I_{specular}$$

In the following we will describe how to calculate each component of the Phong illumination model. Note that we do not model the falloff in intensity, as the distance to the light source increases.

3.1 Ambient Reflection

Ambient reflection is used when we want all parts of the scene to be illuminated. Without ambient reflection, any point not hit by light from a light source will simply be black.

$$I_{ambient}(x) = I_a \cdot k_a(x)$$

Where x is the point calculating the intensity for I_a is the intensity of the ambient light in the scene, and $k_a(x)$ is the material properties in x. Figure 2 shows a frame rendered with only ambient reflection.

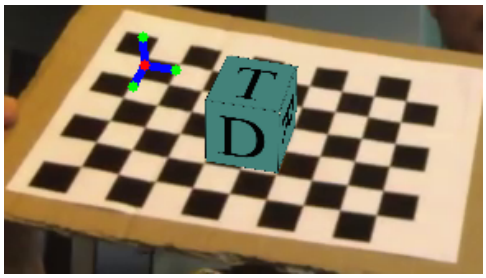


Figure 2: Cube with only ambient reflection

ambient.mp4 on the attached .zip file, shows a rendering using only ambient lighting.

3.2 Diffuse Shading / Lambertian Reflectance

Lambertian reflectance models matte surfaces, that scatter incoming light in all directions. The Lambertian is calculated as

$$I_{diffuse}(x) = I_l(x) \cdot k_d(x) \cdot max(n(x) \cdot l(x), 0)$$

Where $I_l(x)$ is the intensity of incoming diffused light in the point x, $k_d(x)$ is the material properties of the surface in x, $n(x)$ is the normal in x, and $l(x)$ is the direction of the incoming light in x. Figure 3 shows a frame rendered with only diffuse shading.

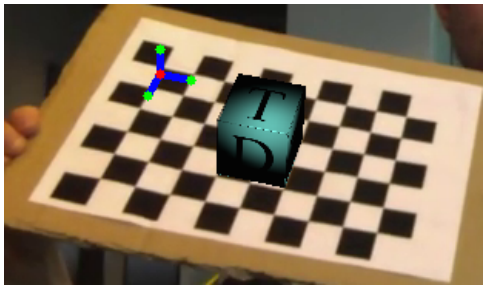


Figure 3: Cube with only diffuse shading

diffuse.mp4 on the attached .zip file, shows a rendering using only diffuse lighting.

3.3 Specular Reflection

Specular reflection models the light reflected by glossy surfaces. It falls of when the angle between the reflection r and the view vector v increases.

The reflection vector can be found as:

$$r = 2 \cdot (n \cdot i \cdot n - i$$

Where n is the normal of the surface the light is being reflected in, and i is the direction of the incoming light.

The reflection vector can then be used to calculate the specular reflection.

$$I_{glossy}(x) = I_s(x) \cdot k_s(x) \cdot (r \cdot v)^{\alpha}$$

Where $I_s(x)$ is the intensity of the incoming specular light in the point x, $k_s(x)$ is the material properties in x, v is the view vector, and α is the reflectance constant of the material (higher values means more reflectance). Figure 4 shows a frame rendered with only specular reflection.

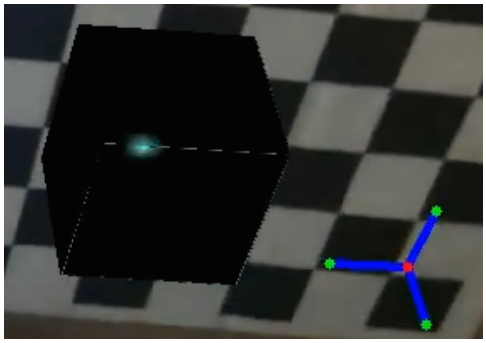


Figure 4: Cube with only specular reflection

specular.mp4 on the attached .zip file, shows a rendering using only specular lighting.

4 Shading

After calculating the illumination, what remains is to calculate the shading in each point of the scene.

4.1 Flat Shading

Flat shading is simply calculating a single shading value for each polygon. Flat shading gives unrealistic results, but is extremely quick to calculate.

4.2 Phong Shading

To improve on flat shading, Phong shading calculates multiple shading values for each polygon, resulting in a more realistic shading of the scene.

Phong shading works as follows: the normals for each corner point in the polygon is calculated, and then for each point in the polygon, the normal for that point is interpolated from the corner points. The shading is then calculated for each point, using the interpolated normal.

4.3 Explanation of the ShadeFace Function

As part of the assignment, we were given a function that applies a shadematrix to an image. The function is called `shadeFace`, and in the following we will outline how it works.

- Create a square of size $shadeRes \cdot shadeRes$, where $shadeRes$ is the resolution of the shading to be applied to the face that is currently being

shaded. A higher value of *shadeRes* results in a smoother shading, at the cost of increased computation time. *shadeRes* should be chosen to be smaller than or equal to the size of the face being shaded, since it will simply be downsampled otherwise.

- Project the corner points in the face, using the camera in the scene.
- Estimate a homography from the shade square, to the projected face.
- Calculate the shading matrix for each of the three channels R, G and B. Figure 5 shows an example of a shade matrix.

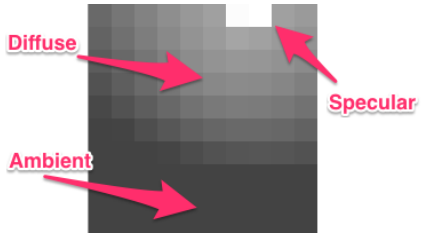


Figure 5: 10x10 Shade Matrix taken from our implementation

- Warp and interpolate the shading for each channel, using the estimated homography, so the shading fits the face being shaded.
- Create a whitemask image, that is white in the pixels corresponding to the face being shaded, and black everywhere else. Figure 6 shows a cropped whitemask image for one of the 3 faces being rendered.

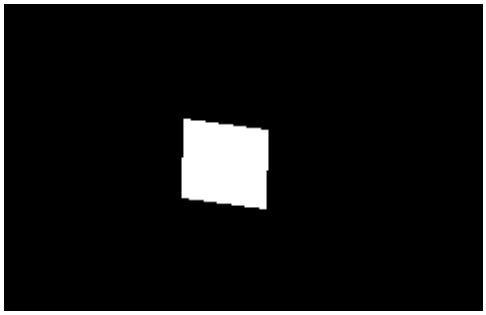


Figure 6: Whitemask (cropped)

- Apply the shading, by multiplying each color channel with the corresponding shading channel, and ensure each pixel lies in the interval $[0; 255]$.
- Finally merge the three channels back into a single image, and return it.

The `ShadeFace` algorithm only works for shading rectangular faces, it could be improved by working on triangles instead. This improvement would allow `ShadeFace` to handle any polygon, since a polygon can be split into triangles, using 'polygon triangulation'. The calculations of the shade matrix would not need to be changed dramatically because of this. It could work on triangles with no changes to the basic idea (bilinear interpolation inside a convex face).

As shown in Figure 7, Phong shading achieves a quite realistic shading. *all.mp4* in the attached .zip file shows a sequence shaded by our implementation of Phong shading.

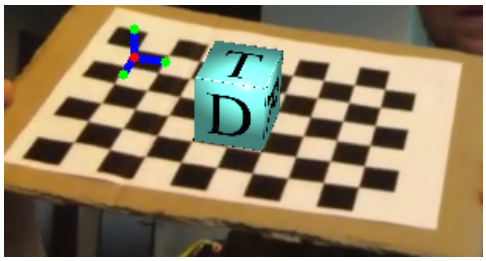


Figure 7: Phong Shading

4.4 Moving the Light Source

We have implemented a way to move the light source around in the scene, the *moving_light.mp4* shows a sequence where the light describes a circle above and around the center of the cube. The camera can also be moved by clicking the mouse, but this is not shown in the sequence.

5 Conclusion

In this report, we have documented our implementation of illumination and shading, and given example frames and sequences. The solution works as expected, but only in a very specific setting. A point of further improvement would be to offer a more generic solutions. Firstly the order of which the faces are drawn is essential for the cube to be drawn correctly, the top face must be drawn last, at the moment the ordering of the faces are hard coded, which is fine for the cube, but any other figure could possibly fail. A solution would be to implement the painters algorithm, ensuring that the back most polygon is drawn first and the outmost polygon drawn last. Secondly it is only possible to calculate the shading of a square polygon in the function `CalculateShadeMatrix`, which means that if we wanted to shade another type of geometrical figure it must be constructed of squares, or the code should be refactored to handle all kinds of polygons, the last is of course preferable.

Appendix

https://github.com/MartinFaartoft/sigb

Assignment_Cube.py

```
'''
Created on March 20, 2014

@author: Diako Mardanbegi (dima@itu.dk)
'''
from numpy import *
import numpy as np
from pylab import *
from scipy import linalg
import cv2
import cv2.cv as cv
from SIGBTools import *
import math

def DrawLines(img, points):
    for i in range(1, len(points[0])):
        x1 = points[0, i - 1]
        y1 = points[1, i - 1]
        x2 = points[0, i]
        y2 = points[1, i]
        cv2.line(img, (int(x1), int(y1)), (int(x2), int(y2)),
    (255, 0, 0),5)
    return img

def findChessBoardCorners(image):
    pattern_size = (9, 6)
    flag = cv2.CALIB_CB_FAST_CHECK + cv2.cv.
    CV_CALIB_CB_NORMALIZE_IMAGE
    gray = cv2.cvtColor(image,cv2.COLOR_BGR2GRAY)
    return cv2.findChessboardCorners(gray, pattern_size, flags=
    flag)

def update(img):
    image=copy(img)

    if Undistorting: #Use previous stored camera matrix and
    distortion coefficient to undistort the image
        ''' <004> Here Undistort the image'''
        image = cv2.undistort(image, cameraMatrix,
    distortionCoefficient)

    if (ProcessFrame):
        ''' <005> Here Find the Chess pattern in the current
    frame'''
        patternFound, corners = findChessBoardCorners(image)

        if patternFound == True:
```

```
            ''' <006> Here Define the cameraMatrix P=K[R|t] of the current frame'''
43          if debug:
                P = findPFromHomography(corners)
45          else:
                P = createPCurrentFromObjectPose(corners)
47
            if ShowText:
49              ''' <011> Here show the distance between the camera origin and the world origin in the image'''

51              cv2.putText(image,str("frame:" + str(frameNumber)), (20,10),cv2.FONT_HERSHEY_PLAIN,1, (255, 255, 255))#Draw the text

53          ''' <008> Here Draw the world coordinate system in the image'''
            cam2 = Camera(P)
55          coordinate_system = getCoordinateSystemChessPlane()
            transformed_coordinate_system = projectChessBoardPoints(cam2,coordinate_system)
57          drawCoordinateSystem(image, transformed_coordinate_system)

59          if TextureMap:

61              ''' <010> Here Do he texture mapping and draw the texture on the faces of the cube'''

63              ''' <012>  calculate the normal vectors of the cube faces and draw these normal vectors on the center of each face'''

65              ''' <013> Here Remove the hidden faces'''


67
            if ProjectPattern:
69              ''' <007> Here Test the camera matrix of the current view by projecting the pattern points'''
                cam2 = Camera(P)
71              X = projectChessBoardPoints(cam2, points_from_chess_board_plane)

73              for p in X:
                    C = int(p[0]),int(p[1])
75                  cv2.circle(image,C, 2,(255,0,255),4)


77
            if WireFrame:
79              ''' <009> Here Project the box into the current camera image and draw the box edges'''
                cam2 = Camera(P)
81              if (Teapot):
                    teapot = parse_teapot()
```

```
83
                    #rotated_box = rotateFigure(box, 0, 0, angle
    , scale, scale, scale)
85                  drawObjectScatter(cam2, image, teapot)
                ## stop
87              else:
                    #figure = box if frameNumber % 100 < 50 else
     pyramid
89
                    angle = frameNumber * (math.pi / 50.0)
91                  scale = 2 + math.sin(angle)
                    box1 = transformFigure(box, 0, 0, -angle, 1,
     1, 1)
93                  box2 = getPyramidPoints([0, 0, -1], 1,
    chessSquare_size)
                    box2 = transformFigure(box2, 0, 0, angle, 1,
     1, 1)
95                  #rotated_box = rotateFigure(figure, 0, 0,
    angle, scale, scale, scale)
                    drawFigure(image, cam2, box1)
97                  drawFigure(image, cam2, box2)

99      cv2.namedWindow('Web cam')
        cv2.imshow('Web cam', image)
101     videoWriter.write(image)
        global result
103     result=copy(image)


105
    def drawFigure(image, camera, figure):
107     X = figure.T
        ones = np.ones((X.shape[0],1))
109     X = np.column_stack((X,ones)).T

111     projected_figure = camera.project(X)
        DrawLines(image,projected_figure)
113
    def getImageSequence(capture, fastForward):
115     '''Load the video sequence (fileName) and proceeds,
    fastForward number of frames.'''
        global frameNumber
117
        for t in range(fastForward):
119         isSequenceOK, originalImage = capture.read()  # Get the
    first frames
            frameNumber = frameNumber+1
121     return originalImage, isSequenceOK


123
    def printUsage():
125     print "Q or ESC: Stop"
        print "SPACE: Pause"
127     print "p: turning the processing on/off "
        print 'u: undistorting the image'
```

```
129     print 'g: project the pattern using the camera matrix (test)
        '
        print 'x: your key!'
131
        print 'the following keys will be used in the next
        assignment'
133     print 'i: show info'
        print 't: texture map'
135     print 's: save frame'

137

139 def run(speed,video):

141     '''MAIN Method to load the image sequence and handle user
        inputs'''

143     #------------------------------video
        capture = cv2.VideoCapture(video)
145
        resultFile = "recording.avi"
147


149
        image, isSequenceOK = getImageSequence(capture,speed)
151
        imSize = np.shape(image)
153     global videoWriter
        videoWriter = cv2.VideoWriter(resultFile, cv.CV_FOURCC('D','
        I','V','3'), 30.0,(imSize[1],imSize[0]),True) #Make a video
        writer
155
        if(isSequenceOK):
157         update(image)
            printUsage()
159
        while(isSequenceOK):
161         OriginalImage=copy(image)


163
            inputKey = cv2.waitKey(1)
165
            if inputKey == 32:#  stop by SPACE key
167             update(OriginalImage)
                if speed==0:
169                 speed = tempSpeed;
                else:
171                 tempSpeed=speed
                    speed = 0;
173
            if (inputKey == 27) or (inputKey == ord('q')):#  break
        by ECS key
175             break
```

```
177         if inputKey == ord('p') or inputKey == ord('P'):
                global ProcessFrame
179             if ProcessFrame:
                    ProcessFrame = False;
181
                else:
183                 ProcessFrame = True;
                update(OriginalImage)
185
            if inputKey == ord('u') or inputKey == ord('U'):
187             global Undistorting
                if Undistorting:
189                 Undistorting = False;
                else:
191                 Undistorting = True;
                update(OriginalImage)
193         if inputKey == ord('w') or inputKey == ord('W'):
                global WireFrame
195             if WireFrame:
                    WireFrame = False;
197
                else:
199                 WireFrame = True;
                update(OriginalImage)
201
            if inputKey == ord('i') or inputKey == ord('I'):
203             global ShowText
                if ShowText:
205                 ShowText = False;

207             else:
                    ShowText = True;
209             update(OriginalImage)

211         if inputKey == ord('t') or inputKey == ord('T'):
                global TextureMap
213             if TextureMap:
                    TextureMap = False;
215
                else:
217                 TextureMap = True;
                update(OriginalImage)
219
            if inputKey == ord('g') or inputKey == ord('G'):
221             global ProjectPattern
                if ProjectPattern:
223                 ProjectPattern = False;

225             else:
                    ProjectPattern = True;
227             update(OriginalImage)

229         if inputKey == ord('x') or inputKey == ord('X'):
                global debug
```

```
231             if debug:
                    debug = False;
233             else:
                    debug = True;
235             update(OriginalImage)

237         if inputKey == ord('l') or inputKey == ord('L'):
                global Teapot
239             Teapot = not Teapot
                update(OriginalImage)
241

243         if inputKey == ord('s') or inputKey == ord('S'):
                name='Saved Images/Frame_' + str(frameNumber)+'.png'
245             cv2.imwrite(name, result)

247         if (speed>0):
                update(image)
249             image, isSequenceOK = getImageSequence(capture,speed
    )

251 def loadCalibrationData():
    global translationVectors
253     translationVectors = np.load('numpyData/translationVectors.
    npy')
    global cameraMatrix
255     cameraMatrix = np.load('numpyData/camera_matrix.npy')
    global rotatioVectors
257     rotatioVectors = np.load('numpyData/rotatioVectors.npy')
    global distortionCoefficient
259     distortionCoefficient = np.load('numpyData/
    distortionCoefficient.npy')
    global points_from_chess_board_plane
261     points_from_chess_board_plane = np.load('numpyData/
    obj_points.npy')[0]
    return cameraMatrix,rotatioVectors[0],translationVectors[0]
263
def calculateP(K,r,t):
265     R,_ = cv2.Rodrigues(r)
    Rt = np.hstack((R,t))
267     P = np.dot(K,Rt)
    return P
269
def displayNumpyPoints(C):
271     points = np.load('numpyData/obj_points.npy')

273     img = cv2.imread('01.png')

275     x = projectChessBoardPoints(C, points[0])

277     for p in x:
            C = int(p[0]),int(p[1])
279         cv2.circle(img,C, 2,(255,0,255),4)
```

```
281     cv2.imshow('result',img)
        cv2.waitKey(0)
283
    def projectChessBoardPoints(C, X):
285     ones = np.ones((X.shape[0],1))
        X = np.column_stack((X,ones)).T
287     x = C.project(X)
        x = x.T
289     return x

291 def getCoordinateSystemChessPlane(axis_length = 2.0):
        o = [0., 0., 0.]
293     x = [axis_length, 0., 0.]
        y = [0., axis_length, 0.]
295     z = [0., 0., -axis_length] #positive z is away from camera,
        by default
        return np.array([o,x,y,z])
297
    def drawObjectScatter(C,img, points):
299     points = points.T
        ones = np.ones((points.shape[0],1))
301     points = np.column_stack((points,ones)).T
        points = C.project(points)
303     points = points.T

305     for point in points:
            cv2.circle(img, (int(point[0]),int(point[1])), 3, (0,
        255, 0), -1)
307

309 def drawCoordinateSystem(img, coordinate_system):
        o = coordinate_system[0]
311     x = coordinate_system[1]
        y = coordinate_system[2]
313     z = coordinate_system[3]

315     cv2.line(img, (int(o[0]),int(o[1])), (int(x[0]),int(x[1]))
        ,(255, 0, 0),3)
        cv2.line(img, (int(o[0]),int(o[1])), (int(y[0]),int(y[1])),
        (255, 0, 0),3)
317     cv2.line(img, (int(o[0]),int(o[1])), (int(z[0]),int(z[1])),
        (255, 0, 0),3)

319     cv2.circle(img, (int(x[0]),int(x[1])), 3, (0, 255, 0), -1)
        cv2.circle(img, (int(y[0]),int(y[1])), 3, (0, 255, 0), -1)
321     cv2.circle(img, (int(z[0]),int(z[1])), 3, (0, 255, 0), -1)
        cv2.circle(img, (int(o[0]),int(o[1])), 3, (0, 0, 255), -1)
323
    def createPCurrentFromObjectPose(corners):
325     found, r_vec, t_vec = cv2.solvePnP(
        points_from_chess_board_plane, corners, cameraMatrix,
        distortionCoefficient)
        return calculateP(cameraMatrix, r_vec, t_vec)
327
```

```
def findPFromHomography(corners_current):
    cam1 = C

    img = cv2.imread("01.png")
    _, corners_1 = findChessBoardCorners(img)
    H,_ = cv2.findHomography(corners_1, corners_current)

    cam2 = Camera(np.dot(H,cam1.P))
    A = np.dot(linalg.inv(K),cam2.P[:,:3])

    r1 = A[:,0]
    r2 = A[:,1]
    r3 = np.cross(r1,r2)
    r3 = r3/np.linalg.norm(r3)

    A = np.array([r1,r2,r3]).T
    cam2.P[:,:3] = np.dot(K,A)
    return cam2.P

def parse_teapot():
    points = []
    with open("teapot.data", "r") as infile:
        lines = infile.read().splitlines()
        for line in lines:
            line = line.split(",")

            x = float(line[0]) + 5
            y = float(line[1]) + 5
            z = (float(line[2]) * -1) - 5
            points.append([x, y, z])

    result =  np.array(points).T
    return result * 2

def transformFigure(figure, theta_x, theta_y, theta_z, scale_x,
    scale_y, scale_z):
    translate_to = [8, 6, -1]


    rotation_matrix_x = np.array([ [1, 0, 0], [0, cos(theta_x),
    -sin(theta_x)], [0, sin(theta_x), cos(theta_x)] ])
    rotation_matrix_y = np.array([ [cos(theta_y), 0, sin(theta_y
    )], [0, 1, 0], [-sin(theta_y), 0, cos(theta_y)] ])
    rotation_matrix_z = np.array([ [cos(theta_z), -sin(theta_z),
     0], [sin(theta_z), cos(theta_z), 0], [0, 0, 1] ])

    rotated_x = []
    rotated_y = []
    rotated_z = []
    rotation = np.dot(rotation_matrix_x, np.dot(
    rotation_matrix_y, rotation_matrix_z))
    for i in range(len(figure[0])):
        p = np.array([figure[0][i], figure[1][i], figure[2][i]])
        p_rot = np.dot(rotation, p)
```

```
377             rotated_x.append(scale_x * p_rot[0] + translate_to[0])
                rotated_y.append(scale_y * p_rot[1] + translate_to[1])
379             rotated_z.append(scale_z * p_rot[2] + translate_to[2])

381         result = np.array([rotated_x, rotated_y, rotated_z])
            return result
383

385 def getPyramidPoints(center, size,chessSquare_size):
            points = []
387
            tl = [center[0]-size, center[1]-size, center[2]]
389         bl = [center[0]-size, center[1]+size, center[2]]
            br = [center[0]+size, center[1]+size, center[2]]
391         tr = [center[0]+size, center[1]-size, center[2]]
            top = [center[0], center[1], center[2] - size * 2]
393
            #bottom
395         points.append(tl)
            points.append(bl)
397         points.append(br)
            points.append(tr)
399         points.append(tl)

401         #top
            points.append(top)
403
            #diagonals
405         points.append(bl)
            points.append(br)
407         points.append(top)
            points.append(tr)
409         points=dot(points,chessSquare_size)
            return array(points).T
411


413 '''--------------------MAIN BODY
        ----------------------------------------------------------------------------------
        '''
415 '''
        ----------------------------------------------------------------------------------
        '''


417


419 '''-------variables-------'''
    global cameraMatrix
421 global distortionCoefficient
    global homographyPoints
423 global calibrationPoints
    global calibrationCamera
425 global chessSquare_size
```

```
427 ProcessFrame=False
    Undistorting=False
429 WireFrame=False
    ShowText=True
431 TextureMap=True
    ProjectPattern=False
433 debug=True
    Teapot = True
435
    tempSpeed=1
437 frameNumber=0
    chessSquare_size=2
439


441
    '''-------defining the figures-------'''
443
    box = getCubePoints([0, 0, 1], 1,chessSquare_size)
445 pyramid = getPyramidPoints([0, 0, 1], 1,chessSquare_size)


447
    i = array([ [0,0,0,0],[1,1,1,1] ,[2,2,2,2] ]) # indices for
        the first dim
449 j = array([ [0,3,2,1],[0,3,2,1] ,[0,3,2,1] ]) # indices for
        the second dim
    TopFace = box[i,j]
451


453 i = array([ [0,0,0,0],[1,1,1,1] ,[2,2,2,2] ]) # indices for
        the first dim
    j = array([ [3,8,7,2],[3,8,7,2] ,[3,8,7,2] ]) # indices for
        the second dim
455 RightFace = box[i,j]


457
    i = array([ [0,0,0,0],[1,1,1,1] ,[2,2,2,2] ]) # indices for
        the first dim
459 j = array([ [5,0,1,6],[5,0,1,6] ,[5,0,1,6] ]) # indices for
        the second dim
    LeftFace = box[i,j]
461
    i = array([ [0,0,0,0],[1,1,1,1] ,[2,2,2,2] ]) # indices for
        the first dim
463 j = array([ [5,8,3,0], [5,8,3,0] , [5,8,3,0] ]) # indices for
        the second dim
    UpFace = box[i,j]
465


467 i = array([ [0,0,0,0],[1,1,1,1] ,[2,2,2,2] ]) # indices for
        the first dim
    j = array([ [1,2,7,6], [1,2,7,6], [1,2,7,6] ]) # indices for
        the second dim
469 DownFace = box[i,j]
```

```
471
473 '''_____'''
    '''_____'''
475
477
    ''' <000> Here Call the calibrateCamera from the SIGBTools to
        calibrate the camera and saving the data'''
479 #calibrateCamera(5, (9,6), 2.0, 0)
    ''' <001> Here Load the numpy data files saved by the
        cameraCalibrate2'''
481 K,r,t = loadCalibrationData()
    ''' <002> Here Define the camera matrix of the first view image
        (01.png) recorded by the cameraCalibrate2'''
483
    P = calculateP(K,r,t)
485 C = Camera(P)

487 ''' <003> Here Load the first view image (01.png) and find the
        chess pattern and store the 4 corners of the pattern needed
        for homography estimation'''
    #displayNumpyPoints(C)
489

491 ''' <003a> Find homography H_cs^1 '''

493 run(1, 0)
    #run(1,"sequence.mov")
```