

# 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.

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 really performance intensive, 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 illumination 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.

### 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

eksempel  
uden  
culling

mention  
strength  
falloff  
some-  
where, if  
we use  
it

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  $x$ .

### 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  $x$ .

### 3.3 Specular Reflection

Specular reflection models the light reflected by glossy surfaces. It falls off 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  $\alpha$  is the reflectance constant of the material (higher values means more reflectance).

## 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. Figure TODO shows an example of a scene with flat shading, and *flat\_shading.avi* on the DVD shows a sequence with flat shading.

## 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 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 length of the face being shaded, since it will simply be downsampled if too big.
- 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.
- 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.
- 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.

show example of shading matrix

show whitemask image

As shown in Figure, Phong shading achieves a much more realistic shading. `phong_shading.avi` on the DVD shows a sequence shaded by our implementation of Phong shading.

figure here

## 5 Conclusion

In this report, we have documented our implementation of illumination and shading, and given example frames and sequences.

An orange rounded rectangular callout box with a black border, containing the text "har vi gjort mere?". A thin orange line connects the end of the sentence in the paragraph above to the top-left corner of the box.

har vi  
gjort  
mere?

## Appendix

<https://github.com/MartinFaartoft/sigb>

### Assignment\_Cube.py

```
1  '''
   Created on March 20, 2014
3
   @author: Diako Mardanbegi (dima@itu.dk)
5  '''
   from numpy import *
7  import numpy as np
   from pylab import *
9  from scipy import linalg
   import cv2
11 import cv2.cv as cv
   from SIGBTools import *
13 import math

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

23
def findChessBoardCorners(image):
25     pattern_size = (9, 6)
        flag = cv2.CALIB_CB_FAST_CHECK + cv2.cv.
CV_CALIB_CB_NORMALIZE_IMAGE
27     gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
        return cv2.findChessboardCorners(gray, pattern_size, flags=
flag)

29
def update(img):
31     image=copy(img)

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

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

41         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
        cam2 = Camera(P)
49
        if ShowText:
51            ''' <011> Here show the distance between the
camera origin and the world origin in the image'''

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

55            center = cam2.center()

57            distance = np.linalg.norm(center)
            cv2.putText(image, str("distance: %02d" %
distance), (20,30), cv2.FONT_HERSHEY_PLAIN, 1, (255, 255, 255))
#Draw the text

59
61            ''' <008> Here Draw the world coordinate system in
the image'''
            cam2 = Camera(P)
63            coordinate_system = getCoordinateSystemChessPlane()
            transformed_coordinate_system =
projectChessBoardPoints(cam2, coordinate_system)
65            drawCoordinateSystem(image,
transformed_coordinate_system)

67            if TextureMap:
                ''' <012> calculate the normal vectors of the
cube faces and draw these normal vectors on the center of
each face'''
69                face_normals = calculate_face_normals()
                draw_face_normals(image, cam2, face_normals)
71
                ''' <013> Here Remove the hidden faces'''
73                idx = back_face_culling(face_normals, cam2)
                faces_to_be_drawn = np.array(Faces)[idx]
75                textures_to_be_drawn = np.array(FaceTextures)[
idx]
                face_corner_normals = np.array(CornerNormals)[
idx]
77                ''' <010> Here Do the texture mapping and draw
the texture on the faces of the cube'''
                for i in range(len(faces_to_be_drawn)):
79                    face = faces_to_be_drawn[i]
                    translate_to = [8, 6, -1]
81                    f = copy(face)

```

```

83         f[0,:] = f[0,:] + translate_to[0]
84         f[1,:] = f[1,:] + translate_to[1]
85         f[2,:] = f[2,:] + translate_to[2]
86
87         texture = textures_to_be_drawn[i]
88         corner_normals = face_corner_normals[i]
89         image = textureFace(image, f, cam2, texture)
90         image = ShadeFace(image, f, corner_normals,
cam2)
91
92         if ProjectPattern:
93             ''' <007> Here Test the camera matrix of the
current view by projecting the pattern points '''
94             cam2 = Camera(P)
95             X = projectChessBoardPoints(cam2,
points_from_chess_board_plane)
96
97             for p in X:
98                 C = int(p[0]),int(p[1])
99                 cv2.circle(image,C, 2,(255,0,255),4)
100
101         if WireFrame:
102             ''' <009> Here Project the box into the current
camera image and draw the box edges '''
103             cam2 = Camera(P)
104             if (Teapot):
105                 teapot = parse_teapot()
106
107                 #rotated_box = rotateFigure(box, 0, 0, angle
, scale, scale, scale)
108                 drawObjectScatter(cam2, image, teapot)
109             ## stop
110             else:
111                 #figure = box if frameNumber % 100 < 50 else
pyramid
112
113                 angle = frameNumber * (math.pi / 50.0)
114                 scale = 2 + math.sin(angle)
115                 box1 = transformFigure(box, 0, 0, -angle, 1,
1, 1)
116                 box2 = getPyramidPoints([0, 0, -1], 1,
chessSquare_size)
117                 box2 = transformFigure(box2, 0, 0, angle, 1,
1, 1)
118                 #rotated_box = rotateFigure(figure, 0, 0,
angle, scale, scale, scale)
119                 drawFigure(image, cam2, box1)
120                 drawFigure(image, cam2, box2)
121
122         cv2.namedWindow('Web cam')
123         cv2.imshow('Web cam', image)
124         videoWriter.write(image)
125         global result

```



```

127         result=copy(image)

129 def drawFigure(image, camera, figure):
    X = figure.T
131     ones = np.ones((X.shape[0],1))
    X = np.column_stack((X,ones)).T
133
    projected_figure = camera.project(X)
135     DrawLines(image,projected_figure)

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

    for t in range(fastForward):
        isSequenceOK, originalImage = capture.read() # Get the
141         first frames
        frameNumber = frameNumber+1
143         return originalImage, isSequenceOK
145

147 def printUsage():
    print "Q or ESC: Stop"
149     print "SPACE: Pause"
    print "p: turning the processing on/off "
151     print 'u: undistorting the image'
    print 'g: project the pattern using the camera matrix (test)
    ,
153     print 'x: your key!'

    print 'the following keys will be used in the next
    assignment'
    print 'i: show info'
157     print 't: texture map'
    print 's: save frame'
159

161 def run(speed, video):
163     '''MAIN Method to load the image sequence and handle user
    inputs'''

165     #-----video
167     capture = cv2.VideoCapture(video)

169     resultFile = "recording.avi"

171

173     image, isSequenceOK = getImageSequence(capture, speed)

```

```

175     imSize = np.shape(image)
176     global videoWriter
177     videoWriter = cv2.VideoWriter(resultFile, cv.CV_FOURCC('D', 'I', 'V', '3'), 30.0, (imSize[1], imSize[0]), True) #Make a video writer
178
179     if (isSequenceOK):
180         update(image)
181         printUsage()
182
183     while (isSequenceOK):
184         OriginalImage=copy(image)
185
186
187         inputKey = cv2.waitKey(1)
188
189         if inputKey == 32:# stop by SPACE key
190             update(OriginalImage)
191             if speed==0:
192                 speed = tempSpeed;
193             else:
194                 tempSpeed=speed
195                 speed = 0;
196
197         if (inputKey == 27) or (inputKey == ord('q')):# break
198             by ECS key
199             break
200
201         if inputKey == ord('p') or inputKey == ord('P'):
202             global ProcessFrame
203             if ProcessFrame:
204                 ProcessFrame = False;
205
206             else:
207                 ProcessFrame = True;
208                 update(OriginalImage)
209
210         if inputKey == ord('u') or inputKey == ord('U'):
211             global Undistorting
212             if Undistorting:
213                 Undistorting = False;
214             else:
215                 Undistorting = True;
216                 update(OriginalImage)
217         if inputKey == ord('w') or inputKey == ord('W'):
218             global WireFrame
219             if WireFrame:
220                 WireFrame = False;
221
222             else:
223                 WireFrame = True;
224                 update(OriginalImage)
225
226         if inputKey == ord('i') or inputKey == ord('I'):

```

```

227         global ShowText
228         if ShowText:
229             ShowText = False;
230
231         else:
232             ShowText = True;
233             update(OriginalImage)
234
235     if inputKey == ord('t') or inputKey == ord('T'):
236         global TextureMap
237         if TextureMap:
238             TextureMap = False;
239
240         else:
241             TextureMap = True;
242             update(OriginalImage)
243
244     if inputKey == ord('g') or inputKey == ord('G'):
245         global ProjectPattern
246         if ProjectPattern:
247             ProjectPattern = False;
248
249         else:
250             ProjectPattern = True;
251             update(OriginalImage)
252
253     if inputKey == ord('x') or inputKey == ord('X'):
254         global debug
255         if debug:
256             debug = False;
257
258         else:
259             debug = True;
260             update(OriginalImage)
261
262     if inputKey == ord('l') or inputKey == ord('L'):
263         global Teapot
264         Teapot = not Teapot
265         update(OriginalImage)
266
267     if inputKey == ord('s') or inputKey == ord('S'):
268         name='Saved Images/Frame_' + str(frameNumber)+' .png'
269         cv2.imwrite(name, result)
270
271     if (speed>0):
272         update(image)
273         image, isSequenceOK = getImageSequence(capture, speed
274 )
275
276 def loadCalibrationData():
277     global translationVectors
278     translationVectors = np.load('numpyData/translationVectors.
279     npy')
280     global cameraMatrix

```

```

cameraMatrix = np.load('numpyData/camera_matrix.npy')
279 global rotatioVectors
rotatioVectors = np.load('numpyData/rotatioVectors.npy')
281 global distortionCoefficient
distortionCoefficient = np.load('numpyData/
distortionCoefficient.npy')
283 global points_from_chess_board_plane
points_from_chess_board_plane = np.load('numpyData/
obj_points.npy')[0]
285 return cameraMatrix, rotatioVectors[0], translationVectors[0]

287 def calculateP(K,r,t):
R,_ = cv2.Rodrigues(r)
289 Rt = np.hstack((R,t))
P = np.dot(K,Rt)
291 return P

293 def displayNumpyPoints(C):
points = np.load('numpyData/obj_points.npy')
295
img = cv2.imread('01.png')
297
x = projectChessBoardPoints(C, points[0])
299
for p in x:
301     C = int(p[0]),int(p[1])
cv2.circle(img,C, 2,(255,0,255),4)
303
cv2.imshow('result',img)
305 cv2.waitKey(0)

307 def projectChessBoardPoints(C, X):
ones = np.ones((X.shape[0],1))
309 X = np.column_stack((X,ones)).T
x = C.project(X)
311 x = x.T
return x
313

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

321 def drawObjectScatter(C,img, points):
points = points.T
323 ones = np.ones((points.shape[0],1))
points = np.column_stack((points,ones)).T
325 points = C.project(points)
points = points.T
327
for point in points:

```

```

329         cv2.circle(img, (int(point[0]),int(point[1])), 3, (0,
255, 0), -1)

331
332 def drawCoordinateSystem(img, coordinate_system):
333     o = coordinate_system[0]
334     x = coordinate_system[1]
335     y = coordinate_system[2]
336     z = coordinate_system[3]
337
338     cv2.line(img, (int(o[0]),int(o[1])), (int(x[0]),int(x[1])),
,(255, 0, 0),3)
339     cv2.line(img, (int(o[0]),int(o[1])), (int(y[0]),int(y[1])),
,(255, 0, 0),3)
340     cv2.line(img, (int(o[0]),int(o[1])), (int(z[0]),int(z[1])),
,(255, 0, 0),3)
341
342     cv2.circle(img, (int(x[0]),int(x[1])), 3, (0, 255, 0), -1)
343     cv2.circle(img, (int(y[0]),int(y[1])), 3, (0, 255, 0), -1)
344     cv2.circle(img, (int(z[0]),int(z[1])), 3, (0, 255, 0), -1)
345     cv2.circle(img, (int(o[0]),int(o[1])), 3, (0, 0, 255), -1)

347 def createPCurrentFromObjectPose(corners):
348     found, r_vec, t_vec = cv2.solvePnP(
349         points_from_chess_board_plane, corners, cameraMatrix,
350         distortionCoefficient)
351     return calculateP(cameraMatrix, r_vec, t_vec)

352 def findPFromHomography(corners_current):
353     cam1 = C
354
355     img = cv2.imread("01.png")
356     _, corners_1 = findChessBoardCorners(img)
357     H,_ = cv2.findHomography(corners_1, corners_current)
358
359     cam2 = Camera(np.dot(H,cam1.P))
360     A = np.dot(linalg.inv(K),cam2.P[:, :3])
361
362     r1 = A[:,0]
363     r2 = A[:,1]
364     r3 = np.cross(r1,r2)
365     r3 = r3/np.linalg.norm(r3)
366
367     A = np.array([r1,r2,r3]).T
368     cam2.P[:, :3] = np.dot(K,A)
369     return cam2.P

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

```

```

377         x = float(line[0]) + 5
378         y = float(line[1]) + 5
379         z = (float(line[2]) * -1) - 5
380         points.append([x, y, z])
381
382     result = np.array(points).T
383     return result * 2
384
385 def transformFigure(figure, theta_x, theta_y, theta_z, scale_x,
386                    scale_y, scale_z):
387     translate_to = [8, 6, -1]
388
389     rotation_matrix_x = np.array([ [1, 0, 0], [0, cos(theta_x),
390 -sin(theta_x)], [0, sin(theta_x), cos(theta_x)] ])
391     rotation_matrix_y = np.array([ [cos(theta_y), 0, sin(theta_y)
392 ), [0, 1, 0], [-sin(theta_y), 0, cos(theta_y)] ])
393     rotation_matrix_z = np.array([ [cos(theta_z), -sin(theta_z),
394 0], [sin(theta_z), cos(theta_z), 0], [0, 0, 1] ])
395
396     rotated_x = []
397     rotated_y = []
398     rotated_z = []
399     rotation = np.dot(rotation_matrix_x, np.dot(
400 rotation_matrix_y, rotation_matrix_z))
401     for i in range(len(figure[0])):
402         p = np.array([figure[0][i], figure[1][i], figure[2][i]])
403         p_rot = np.dot(rotation, p)
404         rotated_x.append(scale_x * p_rot[0] + translate_to[0])
405         rotated_y.append(scale_y * p_rot[1] + translate_to[1])
406         rotated_z.append(scale_z * p_rot[2] + translate_to[2])
407
408     result = np.array([rotated_x, rotated_y, rotated_z])
409     return result
410
411 def back_face_culling(face_normals, camera):
412     #center = camera.c
413     box_center = [8, 6, -1]
414     camera_center = camera.center()
415     camera_x = camera_center[0,0]
416     camera_y = camera_center[1,0]
417     camera_z = camera_center[2,0]
418
419     camera_center = np.array([camera_x, camera_y, camera_z])
420
421     view_vector = box_center - camera_center
422     view_vector = view_vector / np.linalg.norm(view_vector)
423
424     angles = [np.dot(view_vector, face) for face in face_normals]
425     angles = np.array(angles)
426
427     idx = angles <= 0
428     #print angles

```

```

425     return idx

427 def textureFace(image, face, currentCam, texturePath):
428     #translate_to = [8, 6, -1]
429     #f = copy(face)
430     texture = cv2.imread(texturePath)
431     m,n,d = texture.shape
432     mask = zeros((m,n)) + 255
433     face_corners = np.array([[0.,0.],[float(n),0.],[float(n),
float(m)],[0.,float(m)]])

435     #f[0,:] = f[0,:] + translate_to[0]
436     #f[1,:] = f[1,:] + translate_to[1]
437     #f[2,:] = f[2,:] + translate_to[2]

439     X = face.T
440     ones = np.ones((X.shape[0],1))
441     X = np.column_stack((X,ones)).T
442     projected_face = currentCam.project(X).T
443     projected_face = projected_face[:, :-1]

445     I = copy(image)

447     H,_ = cv2.findHomography(face_corners, projected_face)

449     h,w,d = image.shape
450     warped_texture = cv2.warpPerspective(texture, H,(w, h))
451     warped_mask = cv2.warpPerspective(mask, H,(w, h))
452     idx = warped_mask != 0
453     image[idx] = warped_texture[idx]

455     return image

457 def ShadeFace(image, points, faceCorner_Normals, camera):
458     global shadeRes
459     shadeRes=10
460     videoHeight, videoWidth, vd = array(image).shape
461     #.....
462     points_Proj=camera.project(toHomogenous(points))
463     points_Proj1 = np.array([[int(points_Proj[0,0]),int(
points_Proj[1,0])],[int(points_Proj[0,1]),int(points_Proj
[1,1])],[int(points_Proj[0,2]),int(points_Proj[1,2])],[int(
points_Proj[0,3]),int(points_Proj[1,3])]])
464     square = np.array([[0, 0], [shadeRes-1, 0], [shadeRes-1,
shadeRes-1], [0, shadeRes-1]])
465     #.....
466     H = estimateHomography(square, points_Proj1)
467     #.....
468     Mr0,Mg0,Mb0=CalculateShadeMatrix(image, shadeRes, points,
faceCorner_Normals, camera)
469     # HINT
470     # type(Mr0): <type 'numpy.ndarray'>
471     # Mr0.shape: (shadeRes, shadeRes)
472     #.....

```

```

473 Mr = cv2.warpPerspective(Mr0, H, (videoWidth, videoHeight),
    flags=cv2.INTER_LINEAR)
Mg = cv2.warpPerspective(Mg0, H, (videoWidth, videoHeight),
    flags=cv2.INTER_LINEAR)
475 Mb = cv2.warpPerspective(Mb0, H, (videoWidth, videoHeight),
    flags=cv2.INTER_LINEAR)
# .....
477 image=cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
[r, g, b]=cv2.split(image)
479 # .....
whiteMask = np.copy(r)
481 whiteMask[:, :]= [0]
points_Proj2=[]
483 points_Proj2.append([int(points_Proj[0,0]),int(points_Proj
[1,0])])
points_Proj2.append([int(points_Proj[0,1]),int(points_Proj
[1,1])])
485 points_Proj2.append([int(points_Proj[0,2]),int(points_Proj
[1,2])])
points_Proj2.append([int(points_Proj[0,3]),int(points_Proj
[1,3])])
487 cv2.fillConvexPoly(whiteMask, array(points_Proj2)
,(255,255,255))
# .....
489 r[nonzero(whiteMask>0)]=map(lambda x: max(min(x,255),0),r[
nonzero(whiteMask>0)]*Mr[nonzero(whiteMask>0)])
g[nonzero(whiteMask>0)]=map(lambda x: max(min(x,255),0),g[
nonzero(whiteMask>0)]*Mg[nonzero(whiteMask>0)])
491 b[nonzero(whiteMask>0)]=map(lambda x: max(min(x,255),0),b[
nonzero(whiteMask>0)]*Mb[nonzero(whiteMask>0)])
# .....
493 image=cv2.merge((r, g, b))
image=cv2.cvtColor(image, cv2.COLOR_RGB2BGR)
495 return image

497 def CalculateShadeMatrix(image, shadeRes, points,
    faceCorner_Normals, camera):
    #given:
499 #Ambient light IA=[IaR,IaG,IaB]

501 cc = camera.center()
camera_position = np.array([cc[0,0], cc[1,0], cc[2,0]])
503 light_source = np.array([cc[0,0], cc[1,0], cc[2,0]])

505

507 IA = np.matrix([5.0, 5.0, 5.0]).T
#Point light IA=[IpR,IpG,IpB]
509 IP = np.matrix([5.0, 5.0, 5.0]).T
#Light Source Attenuation
511 fatt = 1
#Material properties: e.g., Ka=[kaR; kaG; kaB]
513 ka=np.matrix([0.2, 0.2, 0.2]).T
kd= np.matrix([0.3, 0.3, 0.3]).T

```



```

515     ks=np.matrix([0.7, 0.7, 0.7]).T
      alpha = 100
517
      #ambient: I_ambient(x) = I_a * k_a(x)
519     r = np.zeros((shadeRes, shadeRes))
      g = np.zeros((shadeRes, shadeRes))
521     b = np.zeros((shadeRes, shadeRes))
523
      #Ambient
      r_ambient = r + IA[0] * ka[0]
525     g_ambient = g + IA[1] * ka[1]
      b_ambient = b + IA[2] * ka[2]
527
      #Diffuse
529
      point = points.T[0]
531
      #point = points[0,0]
533     point_normal = np.mean(faceCorner_Normals, axis=1)
535
      point_normal = point_normal / np.linalg.norm(point_normal)
537
      i_diffuse = diffuse(point, point_normal, light_source) * kd
      [0]
539
      i_spectral = speculate(point, point_normal, light_source,
      camera_position, alpha) * ks[0]
      #i_spectral = 0
541
      r_final = r_ambient + i_diffuse + i_spectral #+ r_specular +
      r_diffused
543     g_final = g_ambient + i_diffuse + i_spectral
      b_final = b_ambient + i_diffuse + i_spectral
545
      return (r_final, g_final, b_final)
547
549 def diffuse(point, point_normal, light_source):
      # Regn vector ud fra Light source til point
551
      light_vector = light_source - point
553
      # Calculate distance from point to light
555     r = np.linalg.norm(light_vector)
557
      # Normaliser vector
      light_direction = light_vector / r
559
      #a,b,c = (0.1,0.1,0.1)
561
      #i_l = 1 / float(a * r ** 2 + b * r + c)
563     i_l = 1
565

```

```

        i_diffuse = i_l * max(np.dot(light_direction, point_normal)
        , 0)
567

569     return i_diffuse

571 def speculate(point, point_normal, light_source, camera_position
, alpha):
    #find l
573     incident_vector = point - light_source
    incident_vector = incident_vector/np.linalg.norm(
    incident_vector)
575     #find r
    reflection_vector = incident_vector - 2*np.dot(point_normal,
    incident_vector)*point_normal
577
    view_vector = camera_position - point
579     view_vector = view_vector/np.linalg.norm(view_vector)

581     i_s = 1

583     i_spectral = i_s*np.dot(view_vector, reflection_vector)**
    alpha

585     return i_spectral

587 def calculate_face_normals():
    return np.array([GetFaceNormal(face) for face in Faces])
589     #top_normal = GetFaceNormal(TopFace)

591     #print "top", top_normal
    #return np.array([top_normal])
593
    def draw_face_normals(image, camera, normals):
595         cube_center = [8, 6, -1]
        size = 2
597         #find pairs of points (cube_center -> cube_center + normal)
        #project and draw
599         for normal in normals:
            p1 = cube_center + normal * size
601             p2 = p1 + normal * 4
            #print p1, p2
603             fig = np.array([p1, p2])
            drawFigure(image, camera, fig.T)
605
        def getPyramidPoints(center, size, chessSquare_size):
607             points = []

609             tl = [center[0]-size, center[1]-size, center[2]]
            bl = [center[0]-size, center[1]+size, center[2]]
611             br = [center[0]+size, center[1]+size, center[2]]
            tr = [center[0]+size, center[1]-size, center[2]]
613             top = [center[0], center[1], center[2] - size * 2]

```

```

615     #bottom
        points.append(tl)
617     points.append(bl)
        points.append(br)
619     points.append(tr)
        points.append(tl)
621
        #top
623     points.append(top)

625     #diagonals
        points.append(bl)
627     points.append(br)
        points.append(top)
629     points.append(tr)
        points=dot(points, chessSquare_size)
631     return array(points).T

633

635     '''-----MAIN BODY-----'''
        '''
        '''
        '''
        '''

637

639     '''-----variables-----'''
641     global cameraMatrix
        global distortionCoefficient
643     global homographyPoints
        global calibrationPoints
645     global calibrationCamera
        global chessSquare_size
647
        ProcessFrame=True
649     Undistorting=False
        WireFrame=False
651     ShowText=True
        TextureMap=True
653     ProjectPattern=False
        debug=False
655     Teapot = True

657     tempSpeed=1
        frameNumber=0
659     chessSquare_size=2

661

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

```

```

665 box = getCubePoints([0, 0, 1], 1, chessSquare_size)
    pyramid = getPyramidPoints([0, 0, 1], 1, chessSquare_size)
667

669 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
671 TopFace = box[i,j]

673
    i = array([ [0,0,0,0],[1,1,1,1] , [2,2,2,2] ]) # indices for
    the first dim
675 j = array([ [3,8,7,2],[3,8,7,2] , [3,8,7,2] ]) # indices for
    the second dim
    RightFace = box[i,j]
677

679 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
681 LeftFace = box[i,j]

683 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
685 UpFace = box[i,j]

687
    i = array([ [0,0,0,0],[1,1,1,1] , [2,2,2,2] ]) # indices for
    the first dim
689 j = array([ [1,2,7,6] , [1,2,7,6] , [1,2,7,6] ]) # indices for
    the second dim
    DownFace = box[i,j]
691

693
    '''_____'''
695    '''_____'''

697

699 ''' <000> Here Call the calibrateCamera from the SIGBTools to
    calibrate the camera and saving the data'''
    Faces = [RightFace, LeftFace, UpFace, DownFace, TopFace]
701 FaceTextures = ['Images/Right.jpg', 'Images/Left.jpg', 'Images/
    Up.jpg', 'Images/Down.jpg', 'Images/Top.jpg']

703 t, r, l, u, d = CalculateFaceCornerNormals(TopFace, RightFace,
    LeftFace, UpFace, DownFace)
    CornerNormals = [r, l, u, d, t]
705

```

```

#calibrateCamera(5, (9,6), 2.0, 0)
707 ''' <001> Here Load the numpy data files saved by the
      cameraCalibrate2'''
      K,r,t = loadCalibrationData()
709 ''' <002> Here Define the camera matrix of the first view image
      (01.png) recorded by the cameraCalibrate2'''

711 P = calculateP(K, r, t)
      C = Camera(P)
713
      ''' <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'''
715 #displayNumpyPoints(C)

717
      ''' <003a> Find homography  $H_{cs}^{-1}$ '''
719
      #run(1, 0)
721 run(1, "sequence.mov")

```