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

2 Back Face Culling

3 Lighting

3.1 Ambient Lighting

3.2 Diffuse Lighting

3.3 Specular Lighting

4 Shading

4.1 Flat Shading

4.2 Phong Shading

5 Conclusion

Appendix

https://github.com/MartinFaartoft/sigb

Assignment_Cube.py

```python
'''
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)

        cam2 = Camera(P)

        if ShowText:
            ''' <011> Here show the distance between the camera origin and the world origin in the image'''

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

            center = cam2.center()

            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


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

        if TextureMap:
            ''' <012>  calculate the normal vectors of the cube faces and draw these normal vectors on the center of each face'''
            face_normals = calculate_face_normals()
            draw_face_normals(image, cam2, face_normals)

            ''' <013> Here Remove the hidden faces'''
            idx = back_face_culling(face_normals, cam2)
            faces_to_be_drawn = np.array(Faces)[idx]
            textures_to_be_drawn = np.array(FaceTextures)[idx]
            face_corner_normals = np.array(CornerNormals)[idx]
            ''' <010> Here Do he texture mapping and draw the texture on the faces of the cube'''
            for i in range(len(faces_to_be_drawn)):
                face = faces_to_be_drawn[i]
                translate_to = [8, 6, -1]
                f = copy(face)
```

```
                f[0,:] = f[0,:] + translate_to[0]
                f[1,:] = f[1,:] + translate_to[1]
                f[2,:] = f[2,:] + translate_to[2]

                texture = textures_to_be_drawn[i]
                corner_normals = face_corner_normals[i]
                image = textureFace(image, f, cam2, texture)
                image = ShadeFace(image, f, corner_normals, cam2)

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

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


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

                #rotated_box = rotateFigure(box, 0, 0, angle, scale, scale, scale)
                drawObjectScatter(cam2, image, teapot)
            ## stop
            else:
                #figure = box if frameNumber % 100 < 50 else pyramid

                angle = frameNumber * (math.pi / 50.0)
                scale = 2 + math.sin(angle)
                box1 = transformFigure(box, 0, 0, -angle, 1, 1, 1)
                box2 = getPyramidPoints([0, 0, -1], 1, chessSquare_size)
                box2 = transformFigure(box2, 0, 0, angle, 1, 1, 1)
                #rotated_box = rotateFigure(figure, 0, 0, angle, scale, scale, scale)
                drawFigure(image, cam2, box1)
                drawFigure(image, cam2, box2)

cv2.namedWindow('Web cam')
cv2.imshow('Web cam', image)
videoWriter.write(image)
global result
```

```
    result=copy(image)


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

    projected_figure = camera.project(X)
    DrawLines(image,projected_figure)

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

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


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

    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;
            update(OriginalImage)

        if inputKey == ord('u') or inputKey == ord('U'):
            global Undistorting
            if Undistorting:
                Undistorting = False;
            else:
                Undistorting = True;
            update(OriginalImage)
        if inputKey == ord('w') or inputKey == ord('W'):
            global WireFrame
            if WireFrame:
                WireFrame = False;

            else:
                WireFrame = True;
            update(OriginalImage)

        if inputKey == ord('i') or inputKey == ord('I'):
```

```
            global ShowText
            if ShowText:
                ShowText = False;

            else:
                ShowText = True;
            update(OriginalImage)

        if inputKey == ord('t') or inputKey == ord('T'):
            global TextureMap
            if TextureMap:
                TextureMap = False;

            else:
                TextureMap = True;
            update(OriginalImage)

        if inputKey == ord('g') or inputKey == ord('G'):
            global ProjectPattern
            if ProjectPattern:
                ProjectPattern = False;

            else:
                ProjectPattern = True;
            update(OriginalImage)

        if inputKey == ord('x') or inputKey == ord('X'):
            global debug
            if debug:
                debug = False;
            else:
                debug = True;
            update(OriginalImage)

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


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

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

def loadCalibrationData():
    global translationVectors
    translationVectors = np.load('numpyData/translationVectors.npy')
    global cameraMatrix
```

```python
    cameraMatrix = np.load('numpyData/camera_matrix.npy')
    global rotatioVectors
    rotatioVectors = np.load('numpyData/rotatioVectors.npy')
    global distortionCoefficient
    distortionCoefficient = np.load('numpyData/distortionCoefficient.npy')
    global points_from_chess_board_plane
    points_from_chess_board_plane = np.load('numpyData/obj_points.npy')[0]
    return cameraMatrix,rotatioVectors[0],translationVectors[0]

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

def displayNumpyPoints(C):
    points = np.load('numpyData/obj_points.npy')

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

    x = projectChessBoardPoints(C,points[0])

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

    cv2.imshow('result',img)
    cv2.waitKey(0)

def projectChessBoardPoints(C, X):
    ones = np.ones((X.shape[0],1))
    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)

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

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

def back_face_culling(face_normals, camera):
    #center = camera.c
    box_center  = [8, 6, -1]
    camera_center = camera.center()
    camera_x = camera_center[0,0]
    camera_y = camera_center[1,0]
    camera_z = camera_center[2,0]

    camera_center = np.array([camera_x, camera_y, camera_z])

    view_vector = box_center - camera_center
    view_vector = view_vector / np.linalg.norm(view_vector)

    angles = [np.dot(view_vector, face) for face in face_normals]
    angles = np.array(angles)

    idx = angles <= 0
    #print angles
```

```
    return idx

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

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

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

    I = copy(image)

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

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

    return image

def ShadeFace(image, points, faceCorner_Normals, camera):
    global shadeRes
    shadeRes=10
    videoHeight, videoWidth, vd = array(image).shape
    #................................
    points_Proj=camera.project(toHomogenious(points))
    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])]])
    square = np.array([[0, 0], [shadeRes-1, 0], [shadeRes-1, shadeRes-1], [0, shadeRes-1]])
    #................................
    H = estimateHomography(square, points_Proj1)
    #................................
    Mr0,Mg0,Mb0=CalculateShadeMatrix(image, shadeRes, points, faceCorner_Normals, camera)
    # HINT
    # type(Mr0): <type 'numpy.ndarray'>
    # Mr0.shape: (shadeRes, shadeRes)
    #................................
```

```
    Mr = cv2.warpPerspective(Mr0, H, (videoWidth, videoHeight), flags=cv2.INTER_LINEAR)
    Mg = cv2.warpPerspective(Mg0, H, (videoWidth, videoHeight), flags=cv2.INTER_LINEAR)
    Mb = cv2.warpPerspective(Mb0, H, (videoWidth, videoHeight), flags=cv2.INTER_LINEAR)
    #................................
    image=cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
    [r,g,b]=cv2.split(image)
    #................................
    whiteMask = np.copy(r)
    whiteMask[:,:]=[0]
    points_Proj2=[]
    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])])
    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])])
    cv2.fillConvexPoly(whiteMask,array(points_Proj2),(255,255,255))
    #................................
    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)])
    b[nonzero(whiteMask>0)]=map(lambda x: max(min(x,255),0),b[nonzero(whiteMask>0)]*Mb[nonzero(whiteMask>0)])
    #................................
    image=cv2.merge((r,g,b))
    image=cv2.cvtColor(image, cv2.COLOR_RGB2BGR)
    return image

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

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



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

```
    ks=np.matrix([0.7, 0.7, 0.7]).T
    alpha = 100

    #ambient: I_ambient(x) = I_a * k_a(x)
    r = np.zeros((shadeRes, shadeRes))
    g = np.zeros((shadeRes, shadeRes))
    b = np.zeros((shadeRes, shadeRes))

    #Ambient
    r_ambient = r + IA[0] * ka[0]
    g_ambient = g + IA[1] * ka[1]
    b_ambient = b + IA[2] * ka[2]

    #Diffuse

    point = points.T[0]

    #point = points[0,0]
    point_normal = np.mean(faceCorner_Normals, axis=1)

    point_normal = point_normal / np.linalg.norm(point_normal)

    i_diffuse = diffuse(point, point_normal, light_source) * kd[0]

    i_spectral = speculate(point, point_normal, light_source, camera_position, alpha)

    r_final = r_ambient + i_diffuse + i_spectral #+ r_specular + r_diffused
    g_final = g_ambient + i_diffuse + i_spectral
    b_final = b_ambient + i_diffuse + i_spectral

    return (r_final, g_final, b_final)


def diffuse(point, point_normal, light_source):
    # Regn vector ud fra Light source til point

    light_vector = light_source - point

    # Calculate distance from point to light
    r = np.linalg.norm(light_vector)

    # Normaliser vector
    light_direction = light_vector / r

    #a,b,c = (0.1,0.1,0.1)

    #i_l = 1 / float(a * r ** 2 + b * r + c)
    i_l = 1


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

```
    , 0)

    return i_diffuse

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

    view_vector = camera_position - point
    view_vector = view_vector/np.linalg.norm(view_vector)

    i_s = 1

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

    return i_spectral

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

    #print "top", top_normal
    #return np.array([top_normal])

def draw_face_normals(image, camera, normals):
    cube_center = [8, 6, -1]
    size = 2
    #find pairs of points (cube_center -> cube_center + normal)
    #project and draw
    for normal in normals:
        p1 = cube_center + normal * size
        p2 = p1 + normal * 4
        #print p1, p2
        fig = np.array([p1, p2])
        drawFigure(image, camera, fig.T)

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

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

```
    #bottom
    points.append(tl)
    points.append(bl)
    points.append(br)
    points.append(tr)
    points.append(tl)

    #top
    points.append(top)

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



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



'''----------variables---------'''
global cameraMatrix
global distortionCoefficient
global homographyPoints
global calibrationPoints
global calibrationCamera
global chessSquare_size

ProcessFrame=True
Undistorting=False
WireFrame=False
ShowText=True
TextureMap=True
ProjectPattern=False
debug=False
Teapot = True

tempSpeed=1
frameNumber=0
chessSquare_size=2



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

```python
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'''
Faces = [RightFace, LeftFace, UpFace, DownFace, TopFace]
FaceTextures = ['Images/Right.jpg', 'Images/Left.jpg', 'Images/
    Up.jpg', 'Images/Down.jpg', 'Images/Top.jpg']

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

```

#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")
```