# CS4243 Assignment 3

# ===================

# Name: Tay Yang Shun

# Matric: A0073063M

import numpy as np

import matplotlib.pyplot as plt

from math import \*

# Camera Intrinsic Parameters

u\_0 = 0

v\_0 = 0

B\_u = 1

B\_v = 1

k\_u = 1

k\_v = 1

f = 1

# Figure Plotting Constants

PLOT\_HORIZONTAL\_COUNT = 2

PLOT\_VERTICAL\_COUNT = 2

PLOT\_PADDING = 0.5

PLOT\_MARGIN = 0.2

PLOT\_FONT\_SIZE = 18

NUMBER\_OF\_POINTS\_CUBE = 8

############

# Part 1.1 #

############

def pts\_set\_1():

pts = np.zeros([11, 3])

pts[0,:] = [-1, -1, -1]

pts[1,:] = [1, -1, -1]

pts[2,:] = [1, 1, -1]

pts[3,:] = [-1, 1, -1]

pts[4,:] = [-1, -1, 1]

pts[5,:] = [1, -1, 1]

pts[6,:] = [1, 1, 1]

pts[7,:] = [-1, 1, 1]

pts[8,:] = [-0.5, -0.5, -1]

pts[9,:] = [0.5, -0.5, -1]

pts[10,:] = [0, 0.5, -1]

return pts

def pts\_set\_2():

def create\_intermediate\_points(pt1, pt2, granularity):

new\_pts = []

vector = np.array([x[0] - x[1] for x in zip(pt1, pt2)])

return [np.array(pt2) + (vector \* (float(i)/granularity)) for i in range(1, granularity)]

pts = []

granularity = 20

# Create cube wireframe

pts.extend([[-1, -1, -1], [1, -1, -1], [1, 1, -1], [-1, 1, -1], \

[-1, -1, 1], [1, -1, 1], [1, 1, 1], [-1, 1, 1]])

pts.extend(create\_intermediate\_points([-1, -1, 1], [1, -1, 1], granularity))

pts.extend(create\_intermediate\_points([1, -1, 1], [1, 1, 1], granularity))

pts.extend(create\_intermediate\_points([1, 1, 1], [-1, 1, 1], granularity))

pts.extend(create\_intermediate\_points([-1, 1, 1], [-1, -1, 1], granularity))

pts.extend(create\_intermediate\_points([-1, -1, -1], [1, -1, -1], granularity))

pts.extend(create\_intermediate\_points([1, -1, -1], [1, 1, -1], granularity))

pts.extend(create\_intermediate\_points([1, 1, -1], [-1, 1, -1], granularity))

pts.extend(create\_intermediate\_points([-1, 1, -1], [-1, -1, -1], granularity))

pts.extend(create\_intermediate\_points([1, 1, 1], [1, 1, -1], granularity))

pts.extend(create\_intermediate\_points([1, -1, 1], [1, -1, -1], granularity))

pts.extend(create\_intermediate\_points([-1, -1, 1], [-1, -1, -1], granularity))

pts.extend(create\_intermediate\_points([-1, 1, 1], [-1, 1, -1], granularity))

# Create triangle wireframe

pts.extend([[-0.5, -0.5, -1], [0.5, -0.5, -1], [0, 0.5, -1]])

pts.extend(create\_intermediate\_points([-0.5, -0.5, -1], [0.5, -0.5, -1], granularity))

pts.extend(create\_intermediate\_points([0.5, -0.5, -1], [0, 0.5, -1], granularity))

pts.extend(create\_intermediate\_points([0, 0.5, -1], [-0.5, -0.5, -1], granularity))

return np.array(pts)

pts = pts\_set\_1()

def deg\_to\_rad(deg):

# Converts an angle from degrees to radians

return float(deg)/180 \* pi

def conjugate(quat):

# Calculates the conjugate of a quaternion

return quat[0:1] + np.negative(quat[1:]).tolist()

def approx(value):

# Approximate a floating point value

val = round(value, 6)

return 0.0 if val == 0.0 else val

def approx\_mat(mat):

# Approximate every value in a matrix

for pt in np.nditer(mat, op\_flags=['readwrite']):

pt[...] = approx(pt)

return mat

############

# Part 1.2 #

############

def quatmult(p, q):

# Performs the multiplication of two quaternions

s\_p, v\_p = p[0], np.array(p[1:])

s\_q, v\_q = q[0], np.array(q[1:])

s\_pq = s\_q \* s\_p - np.dot(v\_q, v\_p)

v\_pq = np.cross(v\_p, v\_q) + s\_q \* v\_p + s\_p \* v\_q

out = [s\_pq]

out.extend(v\_pq)

return out

def quatmult\_2(p, q):

# Alternative version of quaternion multiplication just for kicks

out = [0] \* 4

out[0] = p[0]\*q[0] - p[1]\*q[1] - p[2]\*q[2] - p[3]\*q[3]

out[1] = p[0]\*q[1] + p[1]\*q[0] + p[2]\*q[3] - p[3]\*q[2]

out[2] = p[0]\*q[2] - p[1]\*q[3] + p[2]\*q[0] + p[3]\*q[1]

out[3] = p[0]\*q[3] + p[1]\*q[2] - p[2]\*q[1] + p[3]\*q[0]

return out

def quatrot(p, q):

# Performs rotation of two quaternions

# p is the point to be rotated and q is the rotation quaternion

return [approx(x) for x in quatmult(quatmult(q, p), conjugate(q))]

############

# Part 1.3 #

############

def quat2rot(q):

# Returns a 3x3 rotation matrix parameterized with

# the elements of an input quaternion

q\_0, q\_1, q\_2, q\_3 = q

return np.matrix([[q\_0\*\*2 + q\_1\*\*2 - q\_2\*\*2 - q\_3\*\*2, 2\*(q\_1\*q\_2 - q\_0\*q\_3), 2\*(q\_1\*q\_3 + q\_0\*q\_2)],

[2\*(q\_1\*q\_2 + q\_0\*q\_3), q\_0\*\*2 + q\_2\*\*2 - q\_1\*\*2 - q\_3\*\*2, 2\*(q\_2\*q\_3 - q\_0\*q\_1)],

[2\*(q\_1\*q\_3 - q\_0\*q\_2), 2\*(q\_2\*q\_3 + q\_0\*q\_1), q\_0\*\*2 + q\_3\*\*2 - q\_1\*\*2 - q\_2\*\*2]])

# Calculating camera positions for each frame

initial\_pos = [0, 0, 0, -5]

camera\_pos = [initial\_pos]

camera\_rot\_quat = [cos(deg\_to\_rad(-15)), 0, sin(deg\_to\_rad(-15)), 0]

pos\_new = initial\_pos

for i in range(3):

pos\_new = quatrot(pos\_new, camera\_rot\_quat)

camera\_pos.append(pos\_new)

# Camera positions for each frame

pos\_1, pos\_2, pos\_3, pos\_4 = camera\_pos

# Calculating camera orientation for each frame

initial\_orntn = np.identity(3)

camera\_orntns = [initial\_orntn]

camera\_rot\_mat = quat2rot([cos(deg\_to\_rad(15)), 0, sin(deg\_to\_rad(15)), 0])

orntn\_new = initial\_orntn

for i in range(3):

orntn\_new = camera\_rot\_mat \* orntn\_new

camera\_orntns.append(orntn\_new)

camera\_orntns = [np.array(approx\_mat(m)) for m in camera\_orntns]

# Camera orientations for each frame

quatmat\_1, quatmat\_2, quatmat\_3, quatmat\_4 = camera\_orntns

##########

# Part 2 #

##########

def perspective\_proj(s\_p, t\_f, i\_f, j\_f, k\_f):

# Calculate point after perspective projection

sptf = s\_p - t\_f

u\_fp = f \* float(np.dot(sptf, i\_f)) / np.dot(sptf, k\_f) \* B\_u + u\_0

v\_fp = f \* float(np.dot(sptf, j\_f)) / np.dot(sptf, k\_f) \* B\_v + v\_0

return [approx(p) for p in (u\_fp, v\_fp)]

def orthographic\_proj(s\_p, t\_f, i\_f, j\_f, k\_f):

# Calculate point after orthographic projection

sptf = s\_p - t\_f

u\_fp = float(np.dot(sptf, i\_f)) \* B\_u + u\_0

v\_fp = float(np.dot(sptf, j\_f)) \* B\_v + v\_0

return [approx(p) for p in (u\_fp, v\_fp)]

def generate\_projection\_plots(pts, proj\_fn, proj\_name):

fig = plt.figure()

plt.subplots\_adjust(hspace=PLOT\_PADDING, wspace=PLOT\_PADDING)

for i in range(4):

projected\_pts = [proj\_fn(pt, np.array(camera\_pos[i][1:]), camera\_orntns[i][0], \

camera\_orntns[i][1], camera\_orntns[i][2]) for pt in pts]

plt.subplot(PLOT\_HORIZONTAL\_COUNT, PLOT\_VERTICAL\_COUNT, i+1)

plt.margins(PLOT\_MARGIN)

plt.title('Frame ' + str(i+1))

plt.xlabel('x')

plt.ylabel('y')

plt.axis('equal')

for index, pt in list(enumerate(projected\_pts)):

# Use red colour for triangle points for easy identification

plt.plot(pt[0], pt[1], 'bo' if index < 8+(20 - 1)\*12 else 'ro')

plt.suptitle(proj\_name, fontsize=PLOT\_FONT\_SIZE)

fig.savefig(proj\_name + '.png')

generate\_projection\_plots(pts, perspective\_proj, 'Perspective Projection')

generate\_projection\_plots(pts, orthographic\_proj, 'Orthographic Projection')

