2020 Computer Vision
Assignment 2: 3D Reconstruction

The process of recovering camera geometry and 3D scene structure is a significant, perennial problem in computer vision. This assignment will explore the challenges of recovering 3D geometry when errors are made in estimating correspondences between two images in a stereo camera rig. The main aims of this assignment are:

1. to understand the basics of projective geometry, including homogeneous co-ordinates and transformations;
2. to gain practical experience with triangulation and understand how errors in image space manifest in three-dimensional space;
3. to analyse and report your results in a clear and concise manner.

This assignment relates to the following ACS CBOK areas: abstraction, design, hardware and software, data and information, HCI and programming.

1 Stereo cameras

This assignment will experiment with camera rigs modelled by two identical cameras that are separated by a Euclidean transformation. Individual cameras are modelled by a 3×4 *projection matrix* that is parameterised by internal, or *'intrinsic'* parameters—such as focal length and principal point—and external, or *'extrinsic'* parameters for modelling the camera's position in the world. See Lecture 4 and Section 2.1.5 of the textbook for more information on this.

The intrinsics matrix of the cameras is given by

$$\mathbf{K} = \begin{bmatrix} f & 0 & 0 \\ 0 & f & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{1}$$

where f is the *focal length* of the cameras. The two camera projection matrices are given by $\mathbf{P}_1 = \mathbf{P}$ and $\mathbf{P}_2 = \mathbf{PM}$, where

$$\mathbf{P} = \mathbf{K}[\mathbf{I}_{3\times3}|\mathbf{0}], \tag{2}$$

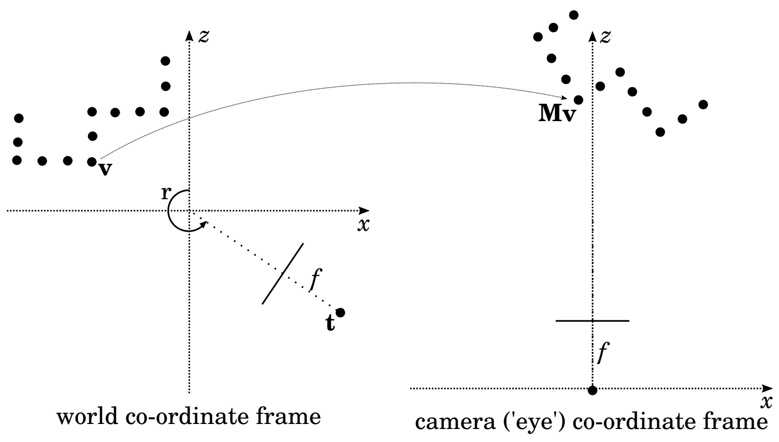

Figure 1: The camera extrinsics matrix transforms the scene so that the camera remains at the origin. Overhead view shown.

and $\mathbf{M}$ is a 4×4 'extrinsics matrix' that re-orientates the world relative to the camera. Therefore, it is the *inverse* of the camera's transformation from its coordinate system into the world. This is illustrated in Figure 1 which depicts the same scene in world and eye co-ordinates, where the transformation from eye co-ordinates into world co-ordinates is given by $\mathbf{M}^{-1}$.

The geometry of the stereo camera rig demonstrating the projection of a 3D point in both cameras is illustrated in Figure 2, where the left camera is located at the origin of the world co-ordinate system and the right camera has been transformed by the relative transformation $\mathbf{M}^{-1}$. Homogeneous 3D points $\mathbf{v}$ are projected into both cameras giving pairs of homogeneous image co-ordinates $\mathbf{v}' = \mathbf{P}\mathbf{v}$ and $\mathbf{v}'' = \mathbf{P}\mathbf{M}\mathbf{v}$. Given the corresponding projections $\mathbf{v}'$ and $\mathbf{v}''$ in the two views, the 3D point $\mathbf{v}$ can be reconstructed with triangulation provided that the camera geometry is known. Point triangulation (and, more broadly, *'bundle adjustment'*) is, itself, a separate process, but for this assignment you are free to use OpenCV's or Matlab's triangulation function.

2 Task 1: Creating a camera

Your first task will investigate how projection matrices are constructed with extrinsics matrices to position the camera in the world, and how they are used to project 3D points onto images. Plotting points with scatter plots will help visualise the effects of changing the projection matrices, but remember that you will first need to convert the points from homogeneous co-ordinates into Cartesian co-ordinates. These different co-ordinate systems are described in Lecture 4 and Section 2.1 of the textbook.

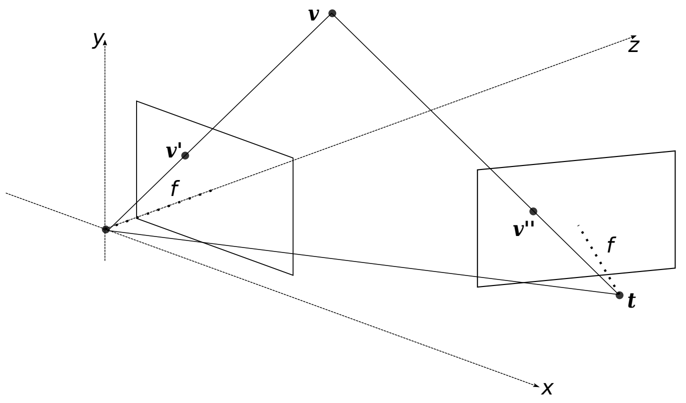

Figure 2: Stereo camera configuration with the left camera at the world origin and the right camera separated by an Euclidean transformation. Here, point *vecp* is projected to $\mathbf{p}'$ and $\mathbf{p}''$ in the left and right views respectively.

The source code included in Section 7 is a simple example of this. In this code, a projection matrix $\mathbf{P} = [\mathbf{I}_{3\times3}|\mathbf{0}]$ corresponding to a focal length of 1, is used to project four homogeneous 3D points with $x = \pm1, y = \pm1, z = 1, w = 1$. The resulting image is displayed as a scatter plot. In this scenario you should expect see the four corners of a square because the image co-ordinates will be $[\frac{\pm1}{1}, \frac{\pm1}{1}]$.

Study and run this code to make sure you understand what it is doing. Next, modify the code to project the four homogeneous points defined by $x = \pm1, y = -1, z = 2 \pm 1, w = 1$ and check the result.

1.1 In your report, show and comment on the effect of modifying the focal length (refer to equation (1)) with both sets of points. How would you obtain the same effect when operating a real camera?

Next, experiment with camera placement by viewing the projection under a variety of extrinsic transformations: in particular translation and rotation. Note that although mathematically you will always get a projection of the scene onto the imaging plane, the image will be upside down if the points are behind the camera (and any points with $z = 0$ will be at infinity). You can check for such cases by plotting each point with a different colour.

To get you started with transforming the camera, begin with the translation matrix

$$\mathbf{T} = \begin{bmatrix} \mathbf{I}_{3\times3} & \mathbf{t} \\ \mathbf{0}^\top & 1 \end{bmatrix} \tag{3}$$

where $\mathbf{t} = [-1, 0, 0]^\top$ translates the *scene* 1 unit to the left—and hence represents a camera one unit to the *right* of the camera defined only by $\mathbf{P}$. Plotting the set of vertices $\mathbf{V} = [\pm1, -1, 2 \pm 1, 1]^\top$ should give you the

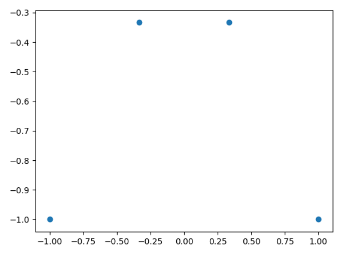
Figure 3: Left view.

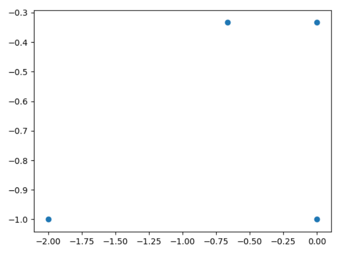
Figure 4: Right view.

graphs in Figure 3 and 4 for the left and right cameras respectively for a camera with unit focal length.

You should also experiment with rotating the scene. The rotation matrix about the y-axis is defined by

$$\mathbf{R}_y = \begin{bmatrix} \cos(\theta) & 0 & \sin(\theta) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(\theta) & 0 & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \tag{4}$$

and there are similar matrices for rotation about the x and z axes (as well as rotating about arbitrary vectors). You can also combine chains of transformations, but note that *the order matters.* For example, the transformation $\mathbf{M} = \mathbf{T}\mathbf{R}_y$ first rotates the scene about the y-axes before it is translated, whereas the transformation $\mathbf{M} = \mathbf{R}_y\mathbf{T}$ will orbit the camera around the origin.

When completing this task, remember that the transformation $\mathbf{M}^{-1}$ maps the eye co-ordinate frame, where the optical centre is at the origin, into world co-ordinates. It may help your understanding to define scenes with a very familiar structure, such as cubes, spheres and sets of parallel lines. This will help you understand how the projection matrices are projecting 3D points onto image planes.

1.2 In your report, include multiple transformations, describing their composition in terms of the type and order of operations, and showing their effect on projection.

The main purpose of this exercise is to understand how $\mathbf{M}$ is used to position the camera. You should experiment with these transformations until you are comfortable with controlling the camera placement, because understanding this concept will be invaluable for the next task.

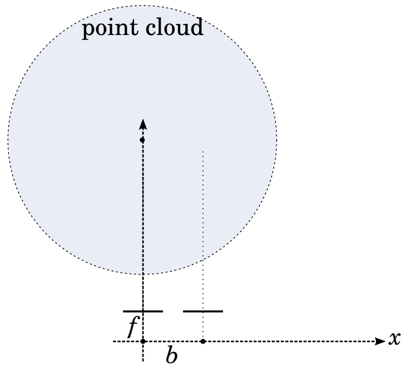

Figure 5: Configuration for task 2.

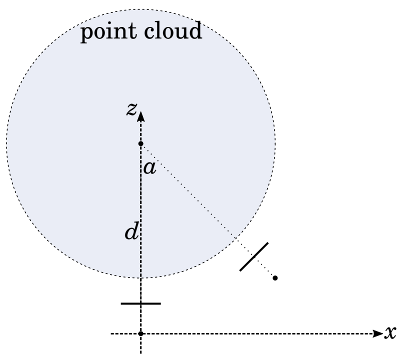

Figure 6: Cnfiguration for task 3.

3 Task 2: reconstruction from parallel cameras

This task will experiment with triangulating image points with known camera geometry but imperfect point correspondences. Implement a pair of stereo cameras sharing the same focal length and a fixed distance b (the 'baseline') between them. Generate a cloud of points within a sphere in front of both cameras and project these points into both views. An illustration of this configuration is shown in Figure 5.

2.1 In your report, describe how you generated the point-cloud and what considerations you gave to its construction.

Use OpenCV/Matlab's triangulation function to reconstruct the 3D points from each pair of corresponding image points and measure the residual (i.e. 3D distance) between the reconstructed 3D points and the ground truth. This should be 0, since you are simply reversing the process of projection! Next, add random Gaussian noise to the projected image point locations, re-triangulate the image points and measure the residual again.

Repeat the experiment by systematically varying the baseline (ie. the distance between the two cameras), focal length and amount and/or type of noise. Although calculating the average error over all points may give *some* insight into the errors introduced by noise, you are encouraged to find a more nuanced analysis by visualising the original and reconstructed point-clouds using a 3D scatter plot. Pay careful attention to where the reconstruction diverges, and comment in your report on any pattern that emerges from these experiments, and suggest a reason for these errors.

2.2 In your report, include at least one graph illustrating the relationship between noise, focal length, and reconstruction error. Include an explanation that describes how image noise affects triangulation and how it is related to your results.

4 Task 3: reconstruction from converging cameras

Whereas the second task was concerned with two cameras with *parallel* optical axes, this task will examine how converging axes affects the 3D reconstruction. For a given focal length and convergence angle, position the second camera so the two cameras' optical axes converge on a common point. This configuration is illustrated in Figure 6, where the position of the second camera is parameterised by the angle a and convergence distance d. Note that you will need to rotate and translate the second camera. If you have difficulty defining a suitable transformation, begin by positioning both cameras at the origin and incrementally shift the second camera and observe how the projection of a known[1] 3D scene changes.

3.1 Repeat the experiment from Task 2 by measuring the effect of varying convergence angle on the reconstructed noisy point-cloud projections. Include a graph of your experiments and compare them to the results from Task 2. With reference to your discussion on error in the previous task, suggest an explanation as to why these results are or are not different from the results you saw in Task 2.

5 Task 4: Estimating pose (post-graduate students only)

Tasks 2 and 3 assume that the relationship between the cameras are known. In some applications we need to estimate this relationship from the point correspondences which can be done by decomposing the Essential Matrix. Both OpenCV and Matlab have methods for estimating and decomposing the Essential Matrix, and we recommend you use the methods that check the orientation of the camera with respect to the triangulated point clouds: the relevant functions are '`recoverPose`' in OpenCV and '`relativeCameraPose`' in Matlab. The essential matrix is explained in Lecture 5 and Section 7.2 of the textbook.

4.1 Repeat Task 3, but use the point correspondences to recover the pose using the Essential Matrix and known intrinsics, and compare the results against those collected in Task 3.

[1] ie. one that you are familiar with, such as a cube

6 Report

Hand in your report and supporting code/images via the MyUni page. Upload two files:

1. report.pdf, a PDF version of your report; and
2. code.zip a zip file containing your code.

This assignment is worth 40% of your overall mark for the course. It is due on ***Monday May 11 at 11.59pm.***

7 Task 1.1 Source code

To help you get started with the assignment, we have included the source code that implements the first part of Task 1.

7.1 OpenCV/Python

```
import numpy as np

v=np.array([  [ 1,   1,  -1,  -1 ],    # x (3D scene points)
              [ 1,  -1,   1,  -1 ],    # y
              [ 1,   1,   1,   1 ],    # z
              [ 1,   1,   1,   1 ]])   # w
P=np.array([  [ 1, 0, 0, 0 ],
              [ 0, 1, 0, 0 ],          # } projection matrix
              [ 0, 0, 1, 0 ] ])
i=np.matmul(P, v);                     # project into homogeneous co-ord
i=i[0:2,:]/i[2]                        # convert to Cartesian co-ord

fig=plt.figure()
plt.scatter(i[0,:], i[1,:], c='r', marker='s')
plt.show()                             # behold: a square!
```

7.2 Matlab

```
v=[ 1,  1, -1, -1;         % x (3D scene points)
    1, -1,  1, -1 ;        % y
    1,  1,  1,  1 ;        % z
    1,  1,  1,  1 ];       % w
P=[ 1, 0, 0, 0 ;
    0, 1, 0, 0 ;           % } projection matrix
    0, 0, 1, 0 ];
i=P*v                      % project into homogeneous co-ord
i=i(1:2,:)./i(3,:)         % convert to Cartesian co-ord
scatter(i(1,:), i(2,:))    % behold: a square!
hold on;                   % but wait, there's more:
i2=v'*P'                   % this is equivalent...
i2=i2(:,1:2)./i2(:,3)
scatter(i2(:,1), i2(:,2))
```

John Bastian
7th April 2020