IPCV Coursework – Part 2 – Report

Introduction

The assignment involved reconstructing 3D spheres from two images of the scene taken by two virtual cameras with known relative pose (i.e. the setup is calibrated). The solution to the assignment obtains from just the two images, and the known position of the cameras, the 3D centre positions and radius lengths of the spheres, which can then be used to draw spheres to a 3D scene in order to visualise the scene.

Completing the assignment required the use of the epipolar line constraint equation, which was used to compute epipolar lines and match corresponding points. It also involved using the formula for 3D reconstruction to obtain a 3D point from two corresponding 2D image points.

Method

Task 3:

The OpenCV Hough circle detector is used in order to automatically identify the centres and radii of the circles seen in the images, which is important since knowing the 2D positions of the circles is key to computing the 3D centres of the spheres in the later tasks. The function provided by OpenCV is dependent on the minimum sphere separation of the spheres (the actual spheres that the virtual cameras took an image of), since the function is given a minimum circle distance to detect in the images, and the minimum separation of the spheres in the 3D geometry will have an impact on how close the circles are in the images. The function is also dependent on the minimum and maximum sphere radius, since the function is given a minimum and maximum circle radius, and the 3D sphere radius range will impact the range of circle radii observed in the images.

Task 4:

In order to draw epipolar lines in the viewing VC for each identified circle centre point in the reference VC, I made use of the epipolar line constraint equation:

$$\hat{\mathbf{p}}_R^T F \hat{\mathbf{p}}_L = 0$$

Where F is the fundamental matrix:

$$F = M_R^T E M_L$$

M_R and M_L are the intrinsic matrices for the right and left cameras respectively. I have denoted in the code that the reference VC is the left camera and the viewing VC is the right camera. In our case, both cameras have identical intrinsic matrices, so both of them consisted of the following:

$$\begin{bmatrix} s_x & 0 & -s_x\hat{o}_x/f \\ 0 & s_y & -s_y\hat{o}_y/f \\ 0 & 0 & 1 \end{bmatrix}$$

Where s_x and s_y are the width and height of a pixel respectively, o_x and o_y defines the principal point (the centre of the image plane) and f is the focal length of the camera.

These intrinsic matrices can be obtained from the cameras directly in the code, so the only thing left to construct in order to get the fundamental matrix F is the essential matrix E.

E is the rotation matrix R (the rotation to get from the reference VC to the viewing VC) multiplied with S, where S is composed of elements of the translation vector T (the translation to get from the reference VC to the viewing VC).

We know the matrix H0_wc for transforming points from the world coordinate system to the reference camera's coordinate system, and we know the matrix H1_wc for transforming points from the world coordinate system to the viewing camera's coordinate system, so to obtain the matrix for transforming points from the reference camera's coordinate system to the viewing camera's coordinate system, we multiply H1_wc with the inverse of H0_wc (the inverse gives a transformation from reference camera to world, which is then converted by H1_wc from world to viewing camera).

R can be extracted from the resulting matrix by taking the first 3 rows and first 3 columns. The first 3 elements of the last column of the matrix represent RT, so we multiply this by the inverse of R to obtain T.

We can now plug everything in to obtain the fundamental matrix F.

For each circle centre in the reference VC image, we can set p_L in the epipolar line constraint equation to this centre point (note that we convert it from a 2D point to a 3D point by setting the z component to the focal length f).

We know the z component of p_R is also f, and so we can expand and rearrange to obtain an equation for the y component of p_R in terms of the x component of p_R.

We can get the start point of the line in the image by plugging in 0 for the x component (this is for the far left side of the image where x=0) to get the y component, and we can get the end point of the line by plugging in the image width (this is for the far right side of the image) to get the y component.

We can join the start and end points fo the line using the OpenCV line function.

Task 5:

To find corresponding circles across the two images, we can use the epipolar line of a circle in the reference VC image and see which circle centre in the viewing VC image is closest to sitting on this line. The one that sits closest is considered a corresponding point.

Rather than plugging in the x of each circle centre in the viewing VC image into the epipolar line equation in the form y=… to obtain the y, and see which circle's y matches the closest to the ones obtained from the equation, I decided to use the epipolar line constraint directly. I did this by plugging into the constraint, for each circle centre p_L in the reference VC image, all circle centre's p_R one by one, and seeing which circle gives a value closest to 0 (since all points that lie on an epipolar line should give a value of 0 when put into the line constraint equation). The circle in the viewing VC that gives a value closest to 0 is chosen as the corresponding circle for the circle in the reference VC.

Task 6:

To compute the 3D sphere centre point for each pair of corresponding circles, we use the following equation:

$$a\underset{3x1}{\begin{bmatrix}\bullet\\ \mathbf{p}_L\\ \bullet\end{bmatrix}} - b\underset{3x1}{\begin{bmatrix}\\ R^T\mathbf{p}_R\\ \end{bmatrix}} - c\underset{3x1}{\begin{bmatrix}\\ \mathbf{p}_L \otimes R^T\mathbf{p}_R\\ \end{bmatrix}} = \underset{3x1}{\begin{bmatrix}\\ \mathbf{T}\\ \end{bmatrix}}$$

Where the three 3x1 vector on the left hand side can be arranged into a single 3x3 matrix H, where each column of H is one of the 3x1 vectors, in the order they appear in the above equation. This gives us a way to compute a, b and c by multiplying both sides of the equation by the inverse of H:

$$\begin{bmatrix}a\\ b\\ c\end{bmatrix} = H^{-1}\,\mathbf{T}$$

T is the translation vector that we have already obtained in task 4. R is the rotation vector that we have also already obtained in task 4.

p_L and p_R are the centres of the circles in the reference VC and viewing VC respectively (each converted into a 3D vector by setting the z component to the focal length f). However, these p's don't have hats, meaning that they are coordinates on the image plane and are not pixel coordinates. To convert them to image plane coordinates, we simply multiply them by the intrinsic matrices M_L and M_R respectively.

So we can plug everything into the equation to obtain the values a, b and c.

'a' is a scalar to multiply p_L by to reach the near the 3D point, and 'b' is a scalar to multiply p_R in the reference camera's coordinate system by to reach near the 3D point. 'c' is the distance between these two vectors (i.e. the length of the vector that is perpendicular to both of these vectors). Note: Due to slight inaccuracies (since we are only estimating the 3D point), the two vectors don't exactly reach the 3D point, hence why I am using the term "near the 3D point".

To get the vector p_R into the left camera's coordinate space, we multiply it by the transpose of the rotation matrix R and add the translation vector T. Let p_R_L denote this vector.

To obtain an estimate of the 3D point, we take the average of the vector (a * p_L) and (b * p_R_L) to get the midpoint.

We are not quite done yet, since this 3D point is with respect to the left camera's coordinate system, but we want a point in the world coordinate system. To obtain this, we simply multiply the 3D point by the inverse of the transformation matrix H0_wc (the inverse gives us a matrix that takes us from the reference camera to the world).

Task 7:

I chose to compute the sphere centre estimate errors as the distance between the estimated and ground truth points. This is because we are dealing with 3D positions, so distance seemed to be the most intuitive metric to assess how far off the points are.

Task 8:
In order to estimate the radius of each sphere, I decided to use the perspective projection equations. My idea was that while we don't have corresponding points across both images for points along the edges of the circles, we now know the 3D sphere centres, the circle centres in the images used to obtain the 3D centres, and we know the radius of the circles in the images (from the Hough circle detection in task 3).

Starting with the reference VC, for each circle we can subtract the y component of the centre coordinates to get the pixel coordinates for the point on the edge of the circle vertically above the centre. We can convert this into image plane coordinates by multiplying by the intrinsic matrix M_L, and denote this vector as p_L. We can multiply the 3D point by the transformation matrix that brings it into the reference camera's coordinate system, and we then know that the point some unknown radius R above the sphere centre projects onto the image plane at p_L.

Using the perspective projection equation, we can calculate this R, which is the sphere radius that we want to find.

If we denote the y and z coordinates of the sphere centre as Y and Z respectively, the focal length of the camera as f, and the y-coordinate of the point on the image plane p_L as y, then the perspective projection equation tells us that:

$$\frac{y}{f} = \frac{Y + R}{Z}$$

Which if we rearrange for sphere radius R we get:

$$R = \frac{Zy}{f} - Y$$

We can plug into this equation for every circle in the reference VC image to get an estimate for every sphere.

We can repeat this for all circles in the viewing VC image, and then since we now have two estimates for each sphere radius, we can take the average to get the final estimate for each sphere radius.

Task 9:

I decided to show the spheres as wireframes (line sets in Open3D), so when you zoom in closer you can see through the spheres to see how much the estimated and ground truth spheres deviate.

I decided to compute the radius estimate errors as simply the absolute difference between the estimated and ground truth radii, since this seemed like an intuitive metric for error in radius lengths.

Task 10:

In terms of sphere radius, a min sphere radius below 4 makes circles that are too small for the Hough circle detector to detect (at least with the parameters I used when detecting the circles), so many spheres are not even considered. With a max sphere radius greater than 30, spheres generated towards the back of the plane (furthest part away from camera 1 (the viewing VC in my case)) can potentially go slightly out of view and have a risk of not being detected by the Hough circle detector. Also at this size, spheres obscuring other spheres is common, leading to spheres that are missed, or spheres that are detected in the reference VC (and hence have epipolar lines generated) but are missed in the viewing VC, so other sphere centres are matched against these epipolar lines instead, causing odd and very inaccurate results.

Min sphere separations that are too small cause a lot of spheres to obscure others quite frequently, causing the same issue as when sphere radii are too big.

Task 11:

Adding noisy relative pose causes more inaccurate sphere estimates (as you would expect). Adding small noise with standard deviation 0.1 to the translation T between the two cameras causes quite a bit of inaccuracy in the estimates, so the code is quite sensitive to noise in the relative camera pose.

Experiments and Results

Task 4:

Implementing this task was a case of implementing the epipolar line equation, being careful to prepare all the correct parts of the equation in the code, such as the fundamental matrix.

My code for the epipolar lines works quite well, producing lines for each reference VC circle on the viewing VC image, cutting close to if not exactly through the circle centres in the viewing VC image.

Task 5:

Implementing this task was a case of using the epipolar line constraint, being careful to prepare all the correct parts of the equation in the code.

My code for task 5 can sometimes fail. Firstly, it can fail if a sphere is obscuring another sphere in the viewing VC image, meaning that no circle is detected for this obscured sphere, which means that a different circle in the viewing VC is matched with the circle detected in the reference VC, which is incorrect behaviour an leads to incorrect matchings.

Secondly, it can fail if two circles in the viewing VC happen to lie very close to or completely on each other's epipolar line (i.e. their epipolar lines line up). This means that there is no way to know which circle should match with which line (at least with the implementation I have made that only uses epipolar line information), so there is a possibility that the circles will be incorrectly matched.

Task 6:

Implementing this task was a case of implementing the formula for 3D reconstruction, being careful to prepare all the correct parts of the equation in the code.

My code for this task works well, provided the circle matchings are correct from task 5.

Task 7:

When the matchings are correct from task 5, I get a low error on average, such as 0.094, meaning that my code is quite good at estimating 3D sphere locations from circle matchings, provided that the matchings from task 5 are correct. If incorrect matchings are given, then it can throw the estimated very off, leading to averages such as 2.86.

Task 8:

Implementing this task involved coming up with my own solution, rather than use any obvious formulae such as the epipolar line constraint. It took me some trial and error to come up with a solution, but in the end it turned out to be a relatively simple application of the perspective projection equation.

Task 9:

The results are not quite as accurate as it is for finding corresponding points (when the matchings from task 5 are correct), since typically the average radius estimate error is larger than the average centre estimate error, but the error is still relatively low, typically around 0.10 to 0.30.

Conclusion

Overall, I have created a relatively successful program for doing 3D reconstruction of spheres given only two camera viewpoint images, and the relative pose of the cameras.

The performance is good when the circle matchings from task 5 are accurate, but as discussed, there are issues that can lead to these matchings being incorrect, which has a chain reaction with the rest of the program, causing the end result to be quite inaccurate. My code does not handle well the case where the matching are wrong, and can assign the same circle in the viewing VC image to multiple epipolar lines in the case of these issues.

To improve the implementation, I would ensure that each circle from each image is assigned only once, and in terms of dealing with the issue of spheres obscuring others in one of the VCs, there isn't anything that I can think of with the current setup, except for limiting the original sphere geometry that is generated in the first place, such as ensuring the minimum sphere separation is sufficiently high and the maximum sphere radius is sufficiently low.

The pose between the two cameras could also be altered, so that their view is more similar, so it is less likely that a sphere that is visible in one VC would be obscured in the other.