I named the project 3T to stand for Two to Three, which alludes to two-dimensional images being converted into a three-dimensional model. I tested the algorithms on a provided data set and used my own photos after I calibrated my iPhone XS, running on iOS 13.4.1.

**Project task**

Given a pair of 2D images, convert the 2D common points into a 3D plot of points. The wider aim of this project is to help architectural students/professors/professionals to map buildings using 2D images.

**Desired outcome**

The desired outcome is a good 3D model. This takes the form of a **point cloud**. Two examples are provided below.
Datasets

Any data set with two images that “surrounds” an object to be analysed is helpful. I have, however, depended on this specific set of images found here. I use them because they are high-resolution and specifically taken for SfM projects.

Literature review

Structure from motion (SfM) is a technique used to derive three-dimensional structures of a scene from a set of two-dimensional images. In this project, structure from motion from multiple views is used, because it yields a more accurate depiction of the 3D object. The following resources have been referenced as part of this project.


Additionally, the following documentation was referenced from OpenCV.

- FLANN matching
- Lucas-Kanade algorithm
- Feature matching
- Epipolar geometry
Algorithm

Developing a usable algorithm

The strategy used was to identify features, then track the features with optical flow across the sequence of images. I assumed that the movement of the camera is predictably stable, i.e., from left to right, top to bottom, or vice versa. This meant that there was less "jerky" motion, which would interfere with the identification of features from image to image.

Changes from P2 update: In P2, I decided to resize images to a width of 700 or below due to processing time, however, I discovered an OpenCV function of down sampling, which is cv2.pyrdown(img). This proved to be effective to down sample the image, while maintaining the accuracy of the algorithm.

1. Load two images, $I_1$ and $I_2$, that are used to identify common features.
2. Select features to match. For this part, I have been experimenting with several different feature detection algorithms, including but not limited to: (a) canny edge detection (see previous lecture), (b) Harris corner/edge detection (see this), (c) Shi-Tomasi detector (see cv2.goodFeaturesToTrack()), and (d) FAST corner detection (see this).
3. Use optical flow algorithm to estimate the motion between $I_1$ and $I_2$. Currently, I am relying on the Lukas-Kanade algorithm, which is provided by the cv2.calcOpticalFlowPyrLK() function.
4. Compute the translation and rotation of the camera from the motion of the images and translate the features accordingly.
5. Derive the common points from (4) into a 3D matrix and plot them onto a graph using a graph library.

Feature detection

What was used

I extracted features using four different algorithms:

- The Harris corner detector;
- The Shi-Tomasi corner detector
- The Canny edge detector + Shi-Tomasi detector
- The fastest accelerated segment test (FAST) detector.

Some examples are provided below. The main testing was done in P2. It is reproduced in the Appendix. For the FAST detector:
For the Canny edge + Shi-Tomasi detector:

The rest of the examples can be found in the appendix.

**Code**

For the basic code, we tracked the strong corners of the image:

```python
def extract_features_lk_img(img):
    feature_params = dict(maxCorners = 300, qualityLevel = 0.25,
                           minDistance = 7, blockSize = 10)

    color = np.random.randint(0, 255, (1000, 3))
    img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    kp_1 = np.int0(cv2.goodFeaturesToTrack(img, mask = None, **feature_params))

    return kp_1
```

There were variations on this, for instance, applying the Canny edge detector beforehand.

```python
def extract_features_harris_lk(img):
    feature_params = dict(maxCorners = 300, qualityLevel = 0.25,
                           minDistance = 7, blockSize = 10, useHarrisDetector=True)

    gray_img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    kp_1 = np.int0(cv2.goodFeaturesToTrack(gray_img, mask = None, **feature_params))

    return kp_1
```

**Tests run for feature detection**

Experimentation informed by research was used to determine what specific algorithm to use. The following algorithms had to be tested (see above also) for feature detection:

- Tracking using Canny combined with Shi-Tomasi edge detection
• Tracking using only Harris corner/edge detection
• Tracking using only Shi-Tomasi detection
• Tracking using FAST detection

As for the optical flow algorithm:
• Lukas-Kanade algorithm was tested.

I measured the average number of seconds to process each algorithm. I had to balance between speed and accuracy for each case.

_Canny edge detector_

Following this research paper, I believed that Canny edge detection matched with optical flow would provide a good approximation of the correct points to identify. Using a Canny edge detector, I converted the input image into a series of edges. A series of features were then extracted using the Shi-Tomasi algorithm. An example is provided of the detection is provided below.

A sample of six images of the tested set is reproduced below.
The processing times were recorded as follows: 0.0053, 0.0064, 0.0071, 0.0057, 0.0058, 0.0072, 0.0059, 0.0043, 0.0044. The average time was: 0.0058 seconds.

Notice that in the image, the lines record the lines of motion, which indicate the correct vector that the camera should be moving.

**Harris corner/edge detection**

The Harris corner detection did not fare well. After a couple of tests, using different block sizes and $k$ values, I decided to abandon it. This is evident from the images below. For example, it did not recognise the main features and got quite messy towards the end.
The processing times were: 0.0037, 0.0059, 0.0042, 0.0039, 0.0033, 0.0033, 0.0037, 0.0023, 0.0038. The average time was: 0.0038 seconds.

*Shi-Tomasi*

The following results were obtained using the Shi-Tomasi feature-matching algorithm, using the `goodFeaturesToTrack()` function provided by OpenCV. On a large image, it is relatively fast.
The processing times were recorded as follows: 0.0035, 0.0028, 0.0028, 0.0041, 0.0032, 0.0028, 0.0033, 0.0027, 0.0023. The average time was: 0.0031 seconds.

*FAST detection*

The features from accelerated segment test (FAST) corner detection method was also used to extract feature points and then subsequently, track and map these points. A FAST feature detection is seen below.
The points which have been detected in this image are correct. The FAST detector detects more points than the Harris, Shi-Tomasi or Canny edge with Shi-Tomasi. The more features detected, the easier the tracking.
The processing times were: 0.1341, 0.1407, 0.1562, 0.1620, 0.1697, 0.1668, 0.1634, 0.1453, 0.2011. The average time was: 0.1599 seconds.

**Camera calibration**

The data set that I used came with the camera matrix and therefore there was no need for calibration. This camera matrix was provided as:

\[
\begin{bmatrix}
689.87 & 0 & 380.1725 \\
0 & 691.04 & 251.7025 \\
0 & 0 & 1
\end{bmatrix}
\]

However, I decided to go one step further and calculate the camera matrix for my iPhone to test it. Several checkerboard tests were carried out, resulting in a camera matrix as follows. The result was as follows.
This created a camera matrix of:

\[
\begin{bmatrix}
1.43049115e+03 & 0.00000000e+00 & 7.85999676e+02 \\
0.00000000e+00 & 1.48194785e+03 & 8.42038371e+02 \\
0.00000000e+00 & 0.00000000e+00 & 1.00000000e+00 \\
\end{bmatrix}
\]

This matrix was used for my own testing.

**Finding the matching points**

Once the optical flow was calculated, I was able to find matching points from one set of the first image to the another set in the second image.

```python
first_key_list = [i.pt for i in kp_1]
kp_arr_1 = np.array(first_key_list).astype(np.float32)
kp_arr_2, status, err = cv2.calcOpticalFlowPyrLK(img1, img2, kp_arr_1, None)
```
What this code does is to extract points that had a small error (< 5.0) with a status of 1. A status of 1 indicates that the flow for the corresponding features has been found; correspondingly a flow status of 0 means that it has not been matched.

Visualising the optical flow is as such (this is for the FAST detector):

---

**Determining the relevant matrices for translation and rotation**

The observation that the camera remains the same (i.e., same camera matrix) allows us to plot the common points. We do so by first determining the *fundamental* and *essential matrices*. These images help to relate corresponding sets of points in the two images, which then allows us to calculate the rotation matrix and translation vector for the camera. This is completed by the following code:

```python
F, f_mask = cv2.findFundamentalMat(match_pts1, match_pts2, cv2.FM_RANSAC, 0.1, 0.99)
E, e_mask = cv2.findEssentialMat(match_pts1, match_pts2, K, cv2.RANSAC, 0.999, 1.0)
```

To extract the matrices, I used the random sample consensus (RANSAC) algorithm. The point correspondences were converted to homogeneous coordinates and solved by putting
the system in form of \( Ax = b \). Once the vector is solved, we need to interpret the pose (which refers to the Rotation and Translation) from the essential matrix. There are a few important properties of the fundamental matrix with regards to epipolar geometry.

1. For any point \( x \) in the first image, the corresponding epipolar line is \( l' = Fx \). Then, \( l' = F^T x' \) is the epipolar line that corresponds to \( x' \) in the second image.

2. The epipolar line contains the epipole \( e' \) which satisfies the equation
   \[
   e'^T (Fx) = (e'^T)x = 0, \forall x
   \]

For any two cameras that are not at infinity, \( P = K[I|0], P' = K'[R|t] \). \( F \) has seven degrees of freedom, since a 3 by 3 homogeneous matrix has 8 independent ratios.

**Triangulation**

After that, we want to find the points that lie inside the both images, i.e., the points which overlap between the images. This is described by the f_mask. Then, we use the `cv2.recoverPose` function to find the rotation and translation matrix from the essential matrix as such:

```python
points, R, t, mask = cv2.recoverPose(E, match_pts1, match_pts2, cameraMatrix=K)
```

Finally, we combine this information with the matching points in each image set (inliers_1 and inliers_2) and triangulate them. Before this, we create the two cameras that are necessary for the triangulation to occur.

```python
Rt1 = np.hstack((np.eye(3), np.zeros((3, 1))))      # Canonical camera
Rt2 = np.hstack((R, t.reshape(3, 1)))               # Our camera
```

Then triangulate:

```python
inliers_1 = np.array(inliers_1)[[:, :2]].transpose()
inliers_2 = np.array(inliers_2)[[:, :2]].transpose()
pts = cv2.triangulatePoints(Rt1, Rt2, inliers_1, inliers_2).T
```

The result `pts4D` is in four dimensions because it contains homogeneous coordinates; in order to convert it back to a Euclidean/Cartesian plane, we divide the X-, Y-, Z- coordinates by the value of W and plot it on the graph.

This results in the following image for the above image set, using a FAST detector.

As for the Canny + Shi-Tomasi detector:
Clearly, from this brief example, the FAST detector was more accurate, although slower. Additionally, I tried out the image on the fountain dataset, which is different because it is protruding from a wall. The image is upside-down due to issues with matplotlib.

Carrying out the experiment using my phone

After I calibrated my phone, I also carried out some testing with it. Due to the ongoing restrictions of movement, I was limited to testing images from the confines of my house; thus, I stacked books to recreate a scene. Admittedly, the results obtained were mixed. I felt that the camera matrix that I calculated from calibration was not accurate, even though I went through an arduous process of calibration:

Some of the results from the FAST detection were taken below:
This created a 3D model:

The flat edge of the books can clearly be made out; however, it remains quite ambiguous. More refinement is therefore necessary to make this an effective algorithm for my camera. However, for a well-calibrated camera it seems to work perfectly well.

**Review**

Of the four feature detection algorithms, FAST detection was highly accurate, but contrary to its name, it was very slow. It was nearly 50 times as slow as the fastest algorithm, which is the Shi-Tomasi algorithm. Thus I have chosen to use either the FAST or Canny + Shi-Tomasi algorithms for feature detection.
<table>
<thead>
<tr>
<th>Feature Detection</th>
<th>Average Time</th>
<th>Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Canny + Shi-Tomasi</td>
<td>0.0058 seconds</td>
<td>High</td>
</tr>
<tr>
<td>Harris</td>
<td>0.0038 seconds</td>
<td>Medium</td>
</tr>
<tr>
<td>Shi-Tomasi</td>
<td>0.0031 seconds</td>
<td>Low</td>
</tr>
<tr>
<td>FAST</td>
<td>0.1599 seconds</td>
<td>High</td>
</tr>
</tbody>
</table>

**Improvements**

This project can be further improved upon.

- It is sensitive to light and jerky motion, due to the assumptions of constant brightness and small motion by the Lucas-Kanade algorithm for optical flow;
- Requires a calibrated camera;
- Requires more points for a well-plotted model.

**Conclusion**

In conclusion, this project allowed me to apply many of the concepts that I have learned in CS585 to a practical challenge. It was a very interesting project and I learnt a lot about epipolar geometry, optical flow, and numerous other techniques. It also illustrated the difficulties that computer vision has in inferring structure (3D) from two-dimensional images. Furthermore, it showed certain limitation of techniques such as optical flow for feature tracking as well.