COMPGV15 Coursework 2: Controllable Heliotrope Video

Name: Jiacheng.Liu ID: 17025135

Due date: Mar. 6th

Basic section

In this section, I implemented algorithm based on given procedure. The specific working steps of the algorithm could be seen as below.

- Load image sequence and down scale to 30%.
- Load flows and compute distance matrix.
- Ask user to select path on shown image.
- Find shortest path and interpolate frames.

Two interesting points in basic section are distance matrix calculation and shortest path comparison.

1. Distance matrix

In the basic section, only image differences are considered, which means only pixel value is taken account into the calculation. The traditional method is making use of Euclidean distance,

$$dist_{RGB} = \sqrt{dist_R^2 + dist_G^2 + dist_B^2}$$

In Figure 1, each pixel represents the similarity between two images and it could be found that the darker point represents higher similarity.

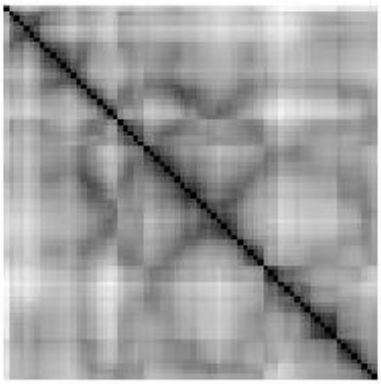

Figure 1: Distance matrix

2. Shortest path comparison

This algorithm uses selected points to evaluate shortest path. For the selected paths which have less direction changes, such as a vertical line, the obtained shortest path is smooth. Figure 2 and Figure 3 show such an example whose selected path is a vertical line from chin to mouth. It could be found that the frame series relized quite smooth required movements.

Figure 2: Selected path (left) and estimated path (right)

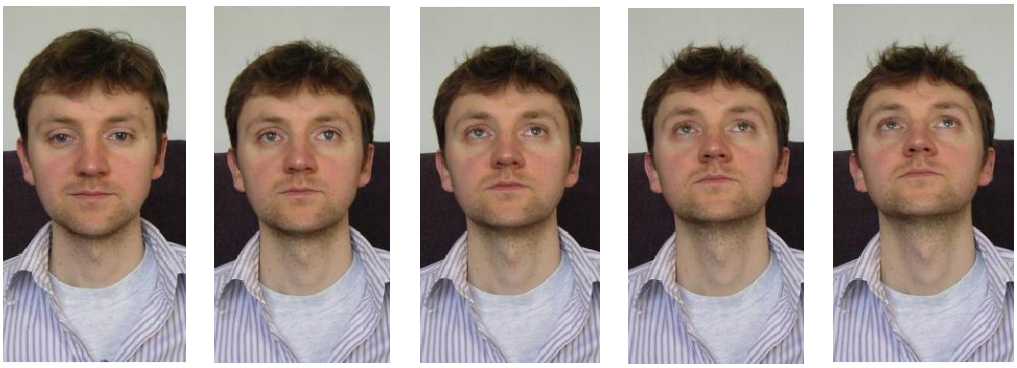

Figure 3: Output image sequence

However, for some selected paths which have obvious direction changes, some additional frames may be included into the obtained series though the output image sequence completes the overall required movements. Figure 4 and Figure 5 illustrate an example whose selected paths are complex. We could learn from Figure 4 that the latter half of estimated path is quite different from selected ones. Additionally, the fifth interpolated frame in Figure 5 is unnecessary and made the output sequence less smooth. The reason may be the distance calculation. Furthermore, another possible solution for this problem is that using selected path direction instead of points to evaluate shortest path.

Figure 4: Selected path (left) and estimated path (right)

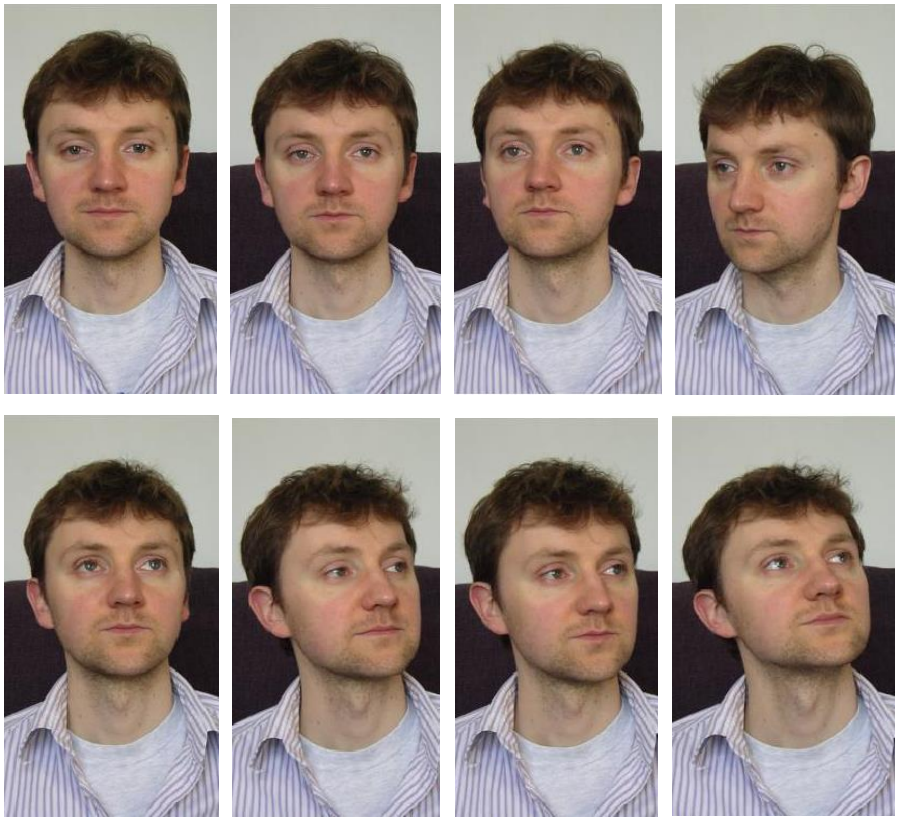

Figure 5: Output image sequence

Advance section

1. Updated distance matrix

In this section, it is required to compute better distance matrix based both image differences and trajectory similarity. To update the distance matrix, I added flow differences into the distance. Each element in optical flow matrix is a vector (vx, vy), which represents the pixel position changes between two certain frames. The shorter vector length represents the higher similarity between frames. The updated distance matrix is calculated by using below equation,

$$dist_{final} = \sqrt{dist_R^2 + dist_G^2 + dist_B^2 + vx^2 + vy^2}$$

Each element in final distance matrix is in the range [0,1]. The specific algorithm could be seen as *'advanced_mats.m'*. The obtained updated distance for given image sequence could be seen as Figure 6.

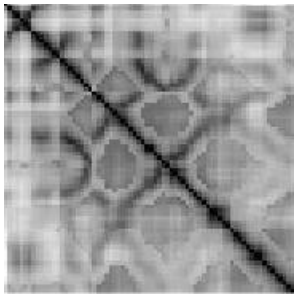

Figure 6: Updated distance matrix

2. Generate own data

This part requires to collect own data and apply them into the algorithm. To complete the task, I collected data with Xinyi YU. We put an orange on the table which is covered by non-reflective dark cloth and collected series of data by changing position of the orange. Three samples of collected data could be seen as below.

Figure 7: Own data examples

By applying '*get_flow()*' on collected data, the flows of own data have been generated and saved as *'own_flows.mat'*. Through multiple tests, I found an interesting point. For some selected paths, the algorithm works well. Both estimated paths and output image sequence are very close to selected paths. However, for some selected paths, the estimated paths show the similar trace as selected ones, but the actual output image sequence illustrates different movement from required paths. It means that the algorithm could realize correct estimated path based on flows, but the computed flows do not match the object movement in image sequence. The reason for this problem might is we did not use tripod in the photo shooting process so slight background moving caused by camera shake leads to computational errors in flows. In addition, the lighting situation for data collection is a possible reason as well. In the process of data organization, it could be found that lighting and shadow situation for photos are not the same. In the future tasks which are similar to this one, above mentioned situations should be considered and try to avoid these possible problems.

Because of space limitation for uploading, resolution of photos were down scaled from 5500×3500 to 1200×763 and some photos were removed final from the test set. However, the flows matrix for own data still too large for uploading. Therefore, the obtained it was removed from uploaded file as well. The algorithm will compute and save flows matrix for own data automatically when it is ran for the first time.

[Data collected with Xinyi YU].

3. Slow motion interpolation

This task required to render slow motion interpolation based initially on the flow between real images in the collection. To realize slow motion interpolation, I used 'griddata()' and 'interp2()' to achieve interpolation in both forward and back directions. In this case, there are 5 additional frames interpolated into interval between 2 successive images. After get interpolation results from both two directions initially, to show smoother switch

between successive frames, the final interpolated frame is constructed by blending results based on certain times step within the interval. In the main program, the algorithm will get each pair of two successive images from computed shortest path and interpolate pre-defined number of slow motion frames between them.

An example of slow motion interpolation between two given images can be seen as Figure 8.

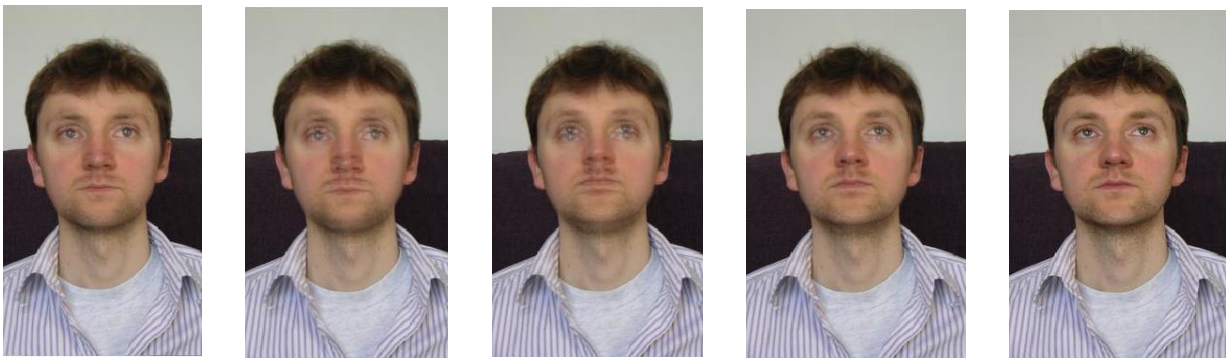

Figure 8: Slow motion interpolation

From above series images, it could be learned that there are some ghosting occurs during the interpolation due to blending. If two given images have quite large differences, the motion will become more obvious. The specific code could be seen in function *'slow_motion.m'* and *'get_interpolated_frame.m'*.

4. Multi-node interpolation

This task was done based on 'slow motion interpolation'. It was required to interpolate additional frames after last output selected image to move points to end pixel. In order to obtain the correct path from end point to expected end point, the stop images of second and third shortest paths to the last point are used to compute additional flows to final expected point.

$$(\mathrm{vx}, \mathrm{vy})_{\mathrm{to\_expect}} = (vx, vy)_{to\_2nd} + (vx, vy)_{to\_3rd}$$

Next, I used similar approach as applied in slow motion interpolation to interpolate additional frames from last image. The interpolated frames are added at the end of frames set with slow motion obtained previously. The specific code could be seen in function *'multi_node_interpolation()'*. However, if the expected point does not at the middle point of end points of second and third closest paths, the final stopped point may be not reach expected point correctly.

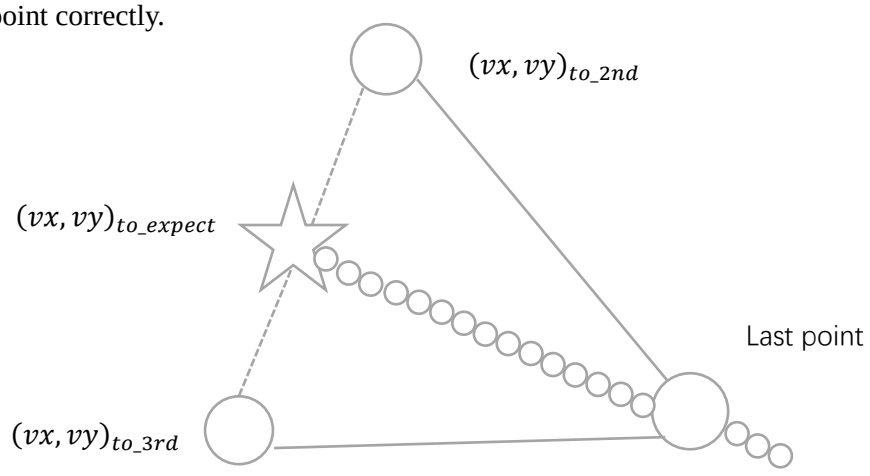

Figure 9: Illustration for multi-node interpolation