MachineVision HW2 - Image Stitching

Author

- 00657113 Chao, Jun-Kai (Compeador)

- Done at 2020/05/20

External Link

- Video on Youtube (https://youtu.be/msms463CDmM)
- Source on Github (https://github.com/ken1882/NTOU2020_MachineVision/tree/master/hw1_filters)

Abstract

This report has differnt approaches to do the job, the first image is easy however the second one is relatively very hard.
First approach is implement the stiching with moving and stacking the images by the coordinate offset of matched features. Second one is by the homography.

Due to the time restriction, codes are unrefined.

Primary Code Explaination

Stitching by offset

```
1 def merge_matrix(big, small, st_r, st_c, op='=')
```

Operation to merge a small matrix to a big one.

- op == '=': replace
- op == '+': adds

- op == 'x': replace if zero

```
class ImageFragment:
  def __init__(self, file, dx=None, dy=None):
    self.file = file
    self.dx   = dx
    self.dy   = dy
    self.next = None
    self.next_score = 0
```

This class assumes images are sequential and much identical that can be stitiched by moving image fragments.

- file: image filename
- dx: delta x compared to last linked image
- dy: delta y compared to last linked image
- next: next linked image
- next_score: Possibility score of next linked image is closer to current one.

```
def stitich_ordered(files, method, offset):
  pass
  # ...
```

Stitiching images by moving fragments.
The feature detection and matching methods are the follows, and here enabled cross check for (possible) better result when matching.

```
FeatureDetector = cv2.xfeatures2d.SIFT_create()
FeatureMatcher  = cv2.BFMatcher(cv2.NORM_L2, crossCheck=True)
```

In first image, fragments are obviously ordered and has no big difference, so went gray might help SIFT/BFMatcher goes faster.

```
gray2 = cv2.cvtColor(img2, cv2.COLOR_BGR2GRAY)
kp2, dt2 = FeatureDetector.detectAndCompute(gray2, None)
```

And next sort the matching features by its distance in ascending order, since the shorter one means two descriptors has higher similarity.

```
matches = sorted(FeatureMatcher.match(dt2, dt1), key=lambda x:x.distance)
```

Then since we know the images are horizontal, so only take x offsets into account.
Then pick first n descriptors calculating their average Δx, get the offset of two images.
Here also used a concept from Bresenham's algorithm to drawing matching preview

```
dx = 0
for i, m in enumerate(matches):
  if i > CANDIDATE_NUM:
    break
  delta = np.array(kp2[m.queryIdx].pt) - np.array(kp1[m.trainIdx].pt)
  dx += delta[0]
dx /= CANDIDATE_NUM
errorx += dx - int(dx)
if errorx > 1.0 or errorx < -1.0:
  dx += int(errorx)
  errorx -= int(errorx)
curx = curx - int(dx)

frag1 = ImageFragment(file, dx, 0)
frag0.next = frag1
fragments.append(frag0)
frag0 = frag1
```

Finally calculating the bounding box of fragments to generate final image, also has 3 different options

```
def generate_final_image(fragments, images, method):
  if method == METHOD_STACKING:
    "Stacking-up images"
  elif method == METHOD_ADD:
    "Calculate average pixel value of images coverage"
  elif method == METHOD_FILLING:
    "Reversed stacking, filling blanks of previous images"
```

This method has good result in first image, but poor in second one.

Stitching by homography

Due to the deviation of homography will be incrased over time, so starting from the middle and extend to both side will get a better result.
So the initial homogeaphy matrix is

$$\begin{bmatrix} 1 & 0 & m \\ 0 & 1 & h \\ 0 & 0 & 1 \end{bmatrix}$$

Where $m =$ `(<canvas width> - <image width>) // 2` (makes the first image in center of the canvas)
And $h = 0$ since images are changing horizontally.

Next we initialize the necesary variables.
The images will overlapping in the process, so we have a simple counter of how many times a pixel been covered here, so when in display, divide the number will be the average pixel value.

```
ones = np.ones(tuple(DISPLAY_SIZE[::-1]))
current_canvas = cv2.warpPerspective(img0, T1, (cur_canvas_width, cur_canvas_hei
t_count  = cv2.warpPerspective(ones,T1,(cur_canvas_width, cur_canvas_height), fl
cur_counter += t_count.astype(np.float)
```

Next using Random Sample Consensus to find the homography map to two images, then we can apply perspective transform later.

```
kp2,dt2 = FeatureDetector.detectAndCompute(cv2.cvtColor(img2, cv2.COLOR_BGR2GRAY
matches = sorted(FeatureMatcher.match(dt2, dt1), key=lambda x:x.distance)
src, dst = [], []
for m in matches:
  src.append(kp2[m.queryIdx].pt + (1,))
  dst.append(kp1[m.trainIdx].pt + (1,))
src = np.array(src,dtype=np.float)
dst = np.array(dst,dtype=np.float)
A, mask = cv2.findHomography(src, dst, cv2.RANSAC)
T1 = T1.dot(A)
```

This method is not bad when applied to first image, but in second image it won't be this simple.

The second one due to the different shot angle, distance and lightning of each images, the offset-stitching has poor result, this is where the homography comes in.

And the parameter is specially tuned as the follow:

```
def stitich_pizza(files):
  FeatureDetector = cv2.xfeatures2d.SIFT_create(nOctaveLayers=8,contrastThreshol
  FeatureMatcher  = cv2.BFMatcher(cv2.NORM_L1, crossCheck=True)
```

However thanks to my ignorance I have to do a little hack to makes the final image properly fit in.
The 5th image has most descriptors to the final so:

```
if idx == 5:
  sav_T = T
  sav_dt = dt2
  sav_kp = kp2
  sav_img = image
# ...
if idx == size - 1:
  T = sav_T
  img0 = sav_img
  kp1, dt1 = sav_kp, sav_dt
```

The result is pretty lackluster, there should be better way to do this but that'll be too exhausting to do imo.

References

- https://docs.opencv.org/2.4/modules/imgproc/doc/geometric_transformations.html#void warpPerspective(InputArray src, OutputArray dst, InputArray M, Size dsize, int flags, int borderMode, const Scalar& borderValue) (https://docs.opencv.org/2.4/modules/imgproc/doc/geometric_transformations.html#void%20warpPerspective(InputArray%20src,%20OutputArray%20dst,%20InputArray%20M,%20Size%20dsize,%20int%20flags,%20int%20borderMode,%20const%20Scalar&%20borderValue))
- https://answers.opencv.org/question/178127/matching-colors-between-two-pictures-in-opencv/ (https://answers.opencv.org/question/178127/matching-colors-between-two-pictures-in-opencv/)
- https://answers.opencv.org/question/178127/matching-colors-between-two-pictures-in-opencv/ (https://answers.opencv.org/question/178127/matching-colors-between-two-pictures-in-opencv/)
- https://docs.opencv.org/2.4/modules/features2d/doc/common_interfaces_of_descriptor_matchers.html (https://docs.opencv.org/2.4/modules/features2d/doc/common_interfaces_of_descriptor_matchers.html)
- https://answers.opencv.org/question/172500/descriptormatcher-bruteforce-hamming-with-opencl/ (https://answers.opencv.org/question/172500/descriptormatcher-bruteforce-hamming-with-opencl/)
- https://docs.opencv.org/3.4.9/d5/d3c/classcv_1_1xfeatures2d_1_1SIFT.html (https://docs.opencv.org/3.4.9/d5/d3c/classcv_1_1xfeatures2d_1_1SIFT.html)
- https://docs.opencv.org/master/dc/dc3/tutorial_py_matcher.html (https://docs.opencv.org/master/dc/dc3/tutorial_py_matcher.html)
- https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_feature2d/py_table_of_contents_feature2d/py_table_of_contents_feature2d.html (https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_feature2d/py_table_of_contents_feature2d/py_table_of_contents_feature2d.html)