Computer Vision (COMP3065 UNNC) (SPC1 23-24): Coursework Report

Submitted April 30, 2024.

Nuo Xu
20320188
Git Link: https://github.com/Lincoooln/Computer-Vision-Coursework.git

School of Computer Science
University of Nottingham

Contents

List of Figures

Chapter 1

Description of Key Features

1.1 Key Features

1.1.1 Objectives

I selected the project 'a) Panorama generation from videos' to work on. The main objectives of the project are:

1. **Develop a basic program to generate panoramas from videos:** It involves extracting key frames from the video, image stitching and post processing stitched image.

2. **Processing of vertically shot videos:** Ability to rotate and process frames.

3. **Blurred video for panorama generation:** Blurring smoothes edges and protects edge information.

4. **Design a User Interface:** The interface allows the user to select the video, the type of video, the setting of the number of keyframes and the intensity of the blur.

1.1.2 Extract Key Frames

The *'extract_key_frames'* function is designed to extract key frames from a video at certain intervals. At first, *'cv2.VideoCapture()'* captures the video from the given video path. The extraction interval is then calculated using *'cv2.CAP_PROP_FRAME_COUNT'* to calculate the total number of frames in the video divided by the number of key frames

set by the user. An iteration is then set up to capture the frame positions in each iteration using *'cv2.CAP_PROP_POS_FRAMES, i'*. The frames were then captured using *'cap.read()'* and if the frame was successfully captured, it was added to the list of key frames. Finally, the function returns the list of key frames for further processing. After debugging, my personal recommended setting for the number of key frames is 30.

1.1.3 Image Stitching

The *'image_stitching'* function is designed to stitch the extracted key frames into a panorama image using OpenCV's built-in stitching functionality. At first, a stitcher object is created using *'cv2.Stitcher_create()'*. Then use *'stitch()'* to stitch together a series of key frames, eventually returning a stitched together picture.

1.1.4 Post Processing Stitched Image

Obviously, this simple generation of panoramas is not enough. Taking the test video 'Horizontal3.mp4' as an example, as shown in Figure 1.1, there are many black parts around the panorama, which are due to the blank areas left in the stitched image in order to maintain the coherence of the stitched image. At the same time, the image is binarized and it is found that there is actually a lot of noise in the image, as shown in Figure 1.2. So I designed the *'post_process'* function to deal with these problems.

Figure 1.1: The panorama with black parts

The *'post_process'* function is design to remove the black part around the panorama as

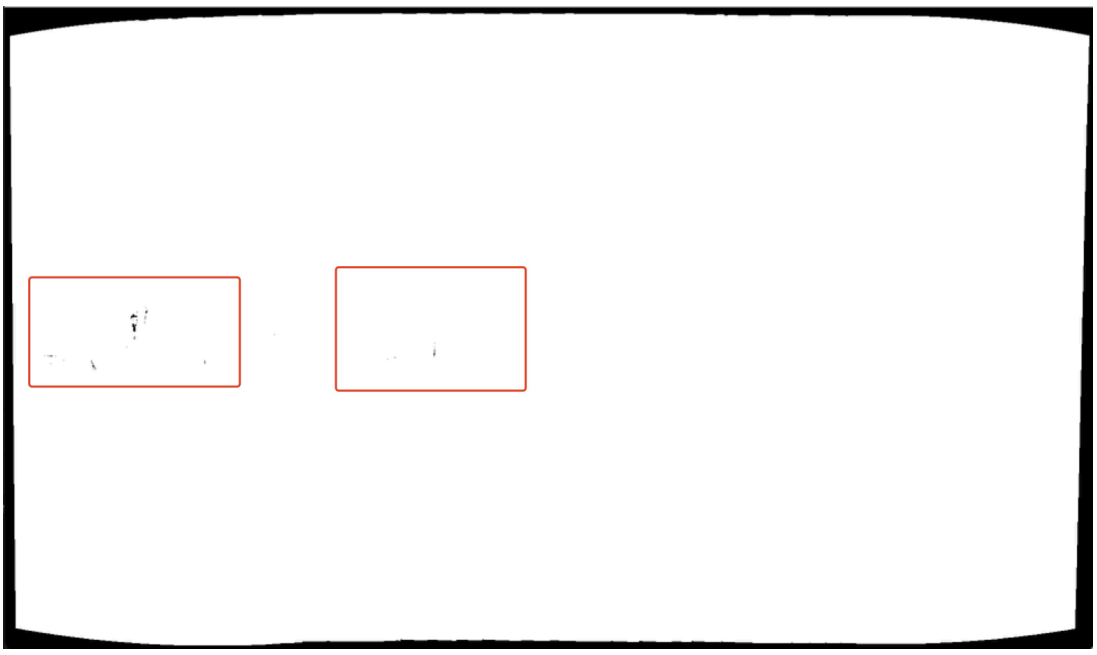

Figure 1.2: The binarized panorama with noise

well as the noise in the image. At first, use *'cv2.copyMakeBorder'* to add a black border to the image to ensure that subsequent processing doesn't affect the edges. Then use *'cv2.cvtColor'* to convert the image to grayscale, and then use *'cv2.threshold'* to distinguish the image we want to get from the black background; if the pixel value is greater than the threshold, it is set to white, and vice versa for black. As shown in Figure 1.2, we are now able to clearly observe the black part around the image and the noise in the image.

Next, using the elliptical kernel from *'cv2.getStructuringElement'*, I went through the debugging kernel size and found that 15x15 was the best fit. Then perform a Morphological closing operation on the image using *'cv2.morphologyEx'*. Morphological closing is achieved by first dilating the image and then eroding the image using the kernel. As shown in Figure 1.3, the noise is removed from the image.

As a follow, we are going to proceed with the processing of the black parts around the image. At first, use *'cv2.findContours'* to find the contours and then *'cv2.contourArea'* to find the main image area. Then utilise *'np.zeros'* to create a blank mask of the same size as the closing image. Then utilise *'cv2.boundingRect'* to calculate the smallest rectangle that can contain the largest contour detected in the image. Finally utilise *'cv2.rectangle'*

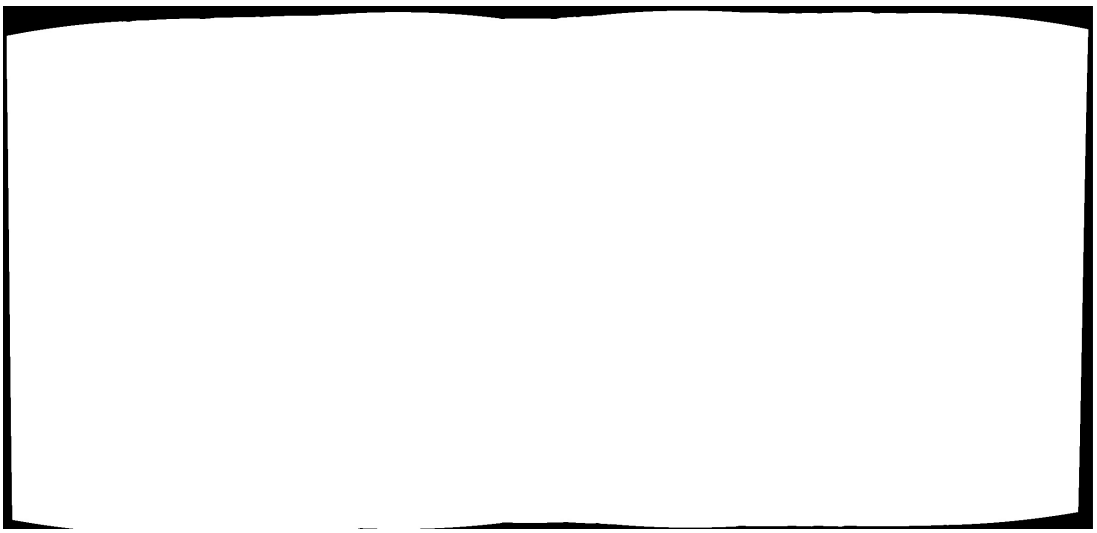

Figure 1.3: The binarized panorama after morphological closing operation

to create a white rectangle. Then create two copies of the mask, one called 'minRectangle' and the other called 'sub'. Then use a while loop to perform the erode operation, if the eroded copy of the mask covers the area outside the main content of the image, the erosion continues, where the rectangle is eroded using *'cv2.erode'*. The final rectangle is shown in Figure 1.4. Finally, use *'cv2.boundingRect'* to calculate the smallest rectangle that can fully contain the specified contour, and then crop the image according to its returned coordinates, width, and height. It can be simply understood as using this rectangle to crop out the black edges around the image. This way a basic panorama is done.

Figure 1.4: The rectangle after erosion

1.2 Additional Features

1.2.1 Processing Vertical Video

In order to be able to process video shot vertically and generate vertical panoramas, the *'rotate_image'* function is designed to rotate the image at a specified angle and automatically resize the canvas to ensure that no part of the image is lost due to rotation. At first, use *'image.shape[:2]'* to return the height and width of the frame, then divide them by 2 to get the coordinates of the centre point of the image, around which the image will rotate. Then *'cv2.getRotationMatrix2D'* generates an affine transformation matrix which is constructed to rotate the image around the centre point by a certain angle. The cosine and sine values are then extracted from the rotation matrix, and the width is used as the horizontal component and the height as the vertical component to calculate the size of the new canvas. Since the rotated image may move out of the boundaries of the initial frame, the image is re-centred into the new canvas, and finally the affine transformation defined by the matrix M is applied to the original image using *'cv2.warpAffine'* to return the rotated frame. When working with vertically shot video, I was first rotating each frame 90 degrees clockwise around the centre point, and then 90 degrees counterclockwise after stitching and post-processing to produce a panorama.

1.2.2 Blurred Video

The *'bilateral_filter'* function is designed to apply bilateral filters to a list of key frames. The bilateral filter reduces noise in the picture while keeping the edges sharp. For each frame, it is filtered using *'cv2.bilateralFilter'*. The parameter *'d'* represents the diameter of the pixel neighbourhood used in the filtering process, the parameter *'sigmaColor'* represents the filter sigma in colour space, and *'sigmaSpace'* represents the filter sigma in coordinate space. In applying this function, I tried three different sets of parameters, (5, 50, 50), (9, 75, 75), and (15, 150, 150). I found that the (9, 75, 75) setting was the most appropriate, the (5, 50, 50) blurring effect was not obvious, and the (15, 150, 150) blurring effect was too blurry. Therefore, I set (9, 75, 75) as 'Moderate Blur'.

1.2.3 User Interface

For the user interface, as shown in Figure 1.5, I used pyqt5 tools to manually design the interface and components, generate the .ui file, and then convert it to a .py file. The user can arbitrarily select a video from the computer, set the number of key frames, the type of video, and the intensity of the blur to generate the panorama. There are two blur intensity choices 'No Blur' and 'Moderate Blur' and two video type choices 'horizontal' and 'vertical'. It is worth noting that I have designed the *'process_horizontal_video'* function in processor.py to generate horizontal panoramas and the *'process_vertical_video'* function to generate vertical panoramas, which serve as a link between *'processor.py'* and *'mian.py'*, the user interface. These two functions are designed to process the frames and generate the panorama using the above mentioned functions in a sequential manner.

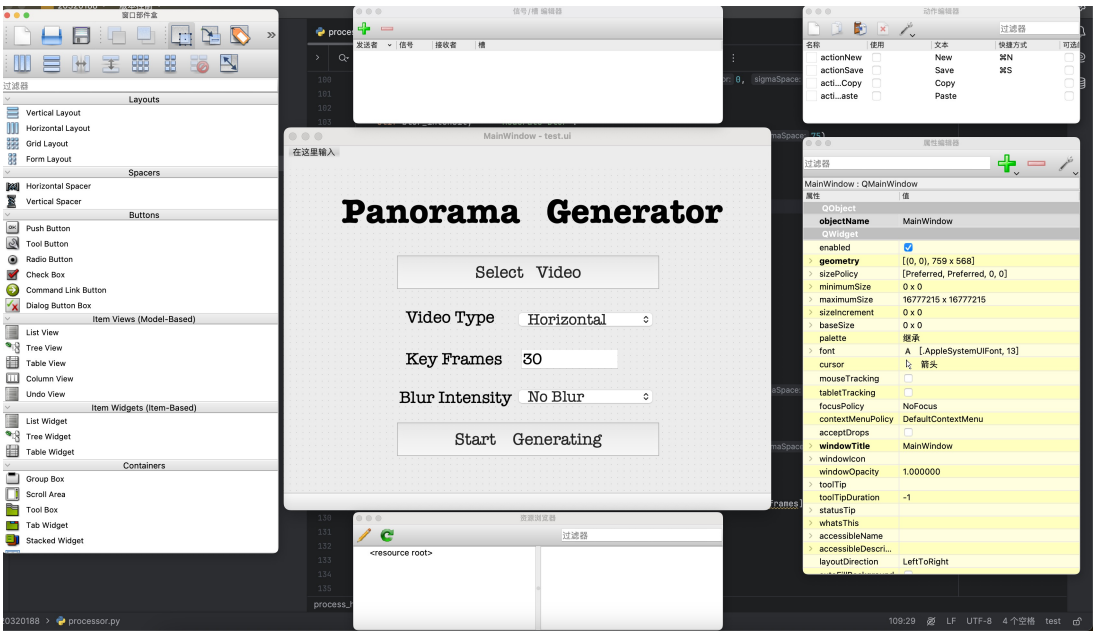

Figure 1.5: pyqt5 tools

Chapter 2

Explanation of the Results

2.1 Processing of horizontally captured video

From Figure 2.1 to Figure 2.6, the horizontally captured video generates panoramic images with no blur and moderate blur. It can be seen that the panoramic images are generated well in the horizontal state from short videos. For the comparison of blur intensity, Figure 2.1 and Figure 2.2 are the most different and I will use them as examples. The smoothing of the street I have circled in red in Figure 2.2 is better than the same portion in Figure 2.1, which suggests that the bilateral filtering is acting as a smoothing agent.

Figure 2.1: Horizontal Panorama with No Blur from Horizontal1.mp4

Figure 2.2: Horizontal Panorama with Moderate Blur from Horizontal1.mp4

2.2 Processing 360-degree horizontally captured video

The above dealt with short videos of 5 to 6 seconds rotated at a smaller angle, followed by a test of 29 seconds of longer video rotated 360 degrees. As shown in Figures 2.7 and 2.8, the generated results are good.

2.3 Processing of vertically captured video

The above is a panorama generated from a horizontally shot video, and the next is a panorama generated from a vertically shot video. As shown in Figure 2.9 and Figure 2.10, the vertical panoramas are also generated quite well.

Figure 2.3: Horizontal Panorama with No Blur from Horizontal2.mp4

Figure 2.4: Horizontal Panorama with Moderate Blur from Horizontal2.mp4

Figure 2.5: Horizontal Panorama with No Blur from Horizontal3.mp4

Figure 2.6: Horizontal Panorama with Moderate Blur from Horizontal3.mp4

Figure 2.7: 360-degree Horizontal Panorama with No Blur from Horizontal4 (360).mp4

Figure 2.8: 360-degree Horizontal Panorama with Moderate Blur from Horizontal4 (360).mp4

Figure 2.9: Vertical Panorama with No Blur from Vertical1.mp4

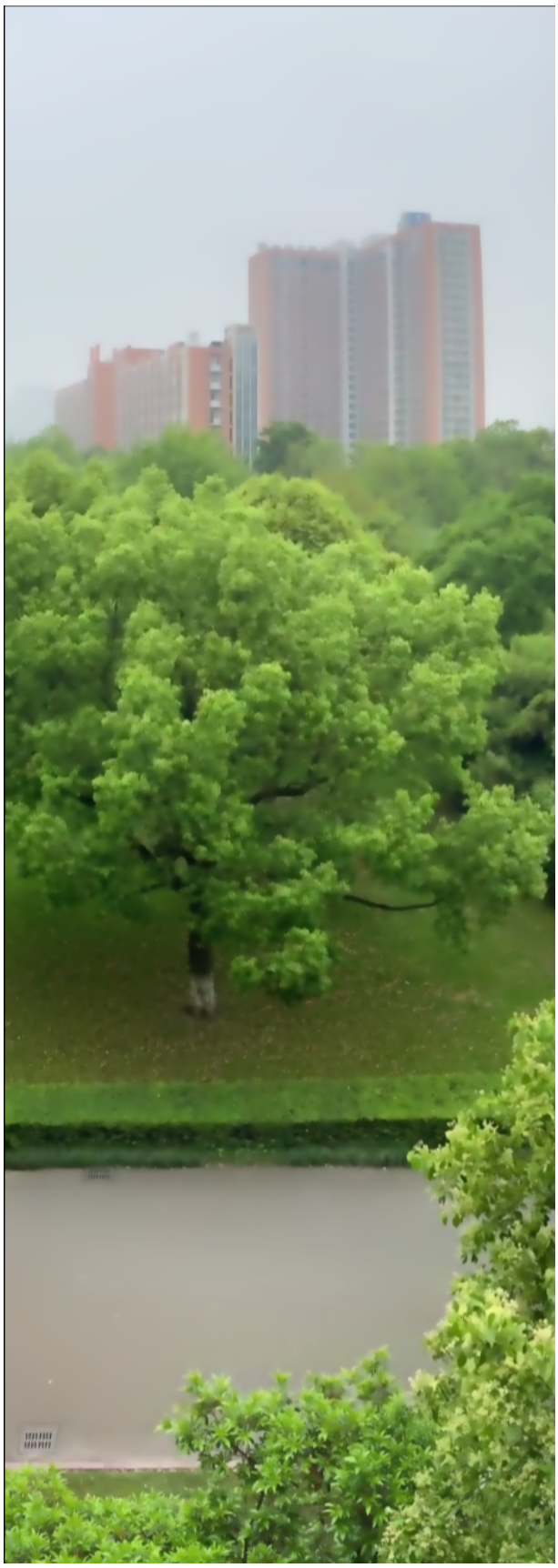

Figure 2.10: Vertical Panorama with Moderate Blur from Vertical1.mp4

Chapter 3

Evaluation

3.1 Strength

Here are what I think are the strengths of my program. First, as shown in Figure 3.1, the program has a very refreshing and user-friendly interface with a high degree of freedom, which allows the users to select the video by themselves, set the number of key frames, the type of video, the intensity of blurring, and get the result they want. Secondly, the program is flexible enough to generate not only horizontal panoramas but also vertical panoramas. Third, from the results, the generation speed is fast and the generated results are relatively good and can be blurred to make the edges smooth. Fourth, the program can handle relatively dark videos due to the handling of noise and black borders.

3.2 Weakness

Here are what I consider to be the weaknesses of the program. First, the user interface presented by two different operating systems, mac os and windows os, running the same code is not the same. For example, an interface designed in windows os will change in font and text box size if it is run in mac os. Secondly, I can't visualise the points matched by the *'stitch'* method in the program to know if the match is accurate. Also, I've tried stitching using *'cv2.warpPerspective'*, but for some reason the result picture is all black. Thirdly, it is taking too long to process the 29 seconds long video. Fourth, when the parameter of bilateral filtering is set to (5, 50, 50) and (15, 150, 150), processing some videos will report an error, as shown in Figure 3.2, may be there is a nan or inf in the

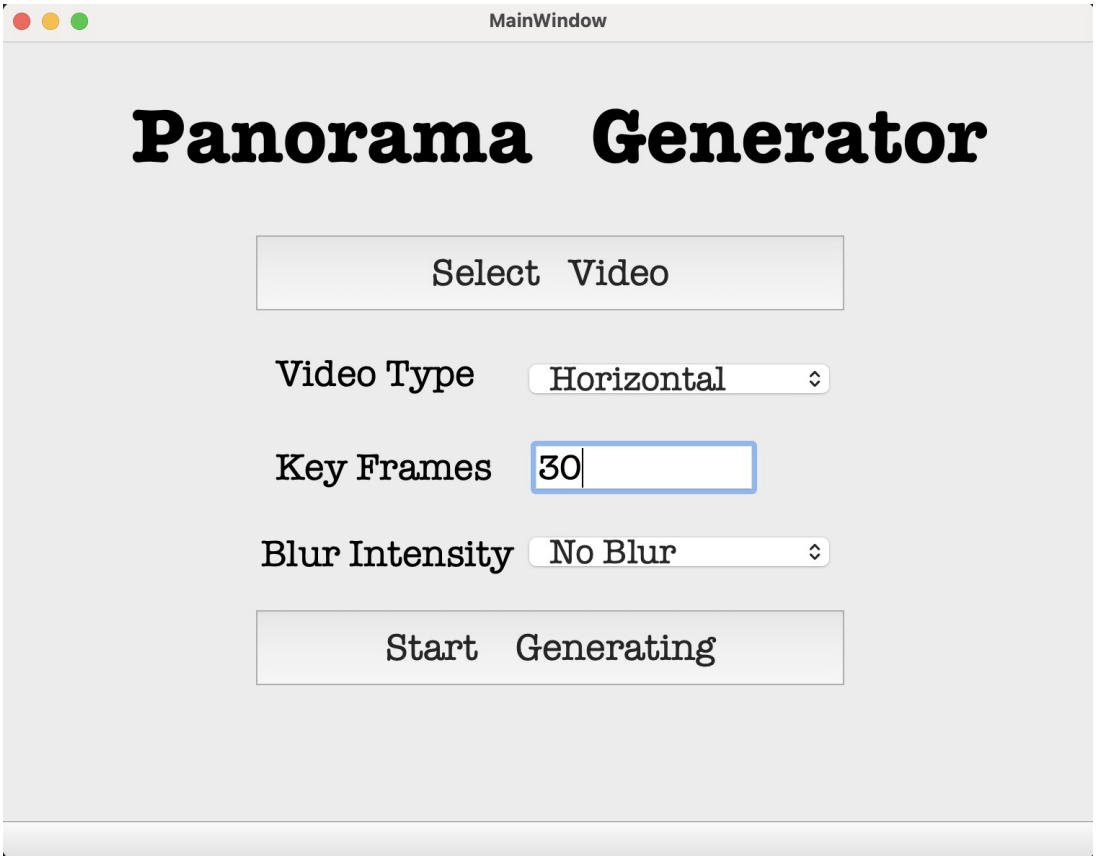

Figure 3.1: User Interface

matrix being inverted, spend a lot of time and still not solved. Note that the panoramas that were successfully generated when set to (5, 50, 50) and (15, 150, 150), I have also saved to the results folder and will not describe them more here. Fifth, the method of selecting key frames is proportional, the generated panorama is subject to the moving speed of the shooting, if the speed of rotation is not uniform when shooting, the panorama will produce a fault, as shown in Figure 2.1.

```
** On entry to DLASCL, parameter number  4 had an illegal value
** On entry to DLASCL, parameter number  4 had an illegal value
** On entry to DLASCL, parameter number  5 had an illegal value
** On entry to DLASCL, parameter number  4 had an illegal value
```

Figure 3.2: Error