University of Nottingham
UK | CHINA | MALAYSIA

COMP3065: Computer Vision Coursework Report

Submitted: **6th May, 2024**

Zepei Luo
20321548
scyzl9@nottingham.edu.cn

School of Computer Science
University of Nottingham Ningbo China

Chapter 1

Introduction

This report will illustrate the workflow and key features used to implement **Project (a): Panorama generation from videos**. The author realized the whole process independently without the help of any libraries that generate results directly, such as *cv2.createStitcher()* or *cv2.findHomography()*. Here are some important features implemented beyond the basic requirements:

Key features & functionalities

1. Overcome the excessive distortion problem of conventional methods
2. Smooth the chromatic aberration due to exposure problems (Feathering)
3. Handle videos with persistent shaking
4. Design UI to visualize panorama generating and above functionalities

Structure of the Report

Chapter 2 will illustrate the description of key features and implementation. **Chapter 3** will present and analyze the performance of the proposed algorithm on four different source videos. **Chapter 4** will explain the strengths and weaknesses of the chosen approach and methods in detail.

Chapter 2

Implementation & Analysis

This chapter will include the description of the key features and implementation. Section 2.1 introduces the limitations of traditional methods. Section 2.2 illustrates the workflow of the whole panorama generation process. Section 2.3 contains the introduction of some basic steps. Section 2.4 demonstrates how the proposed new ideas and new features improve this work and being able to work with hard scenarios.

2.1 Foreword

A video can be seen as a sequential presentation of numerous frames. Firstly, several frames will be extracted from the source video with a fixed interval. The solution of the task is translated into how to stitch these frames together to form a panorama. Typically, a stitching algorithm is devised to enable the seamless attachment of one image to the right side of another. By repeatedly invoking this algorithm, all frames are sequentially joined to the right to create a panoramic image. Figure 2.11 shows the performance of this conventional method when applied to short videos, where it can achieve relatively satisfactory results. However, the stitching outcome becomes markedly inadequate when this method is employed on longer videos.

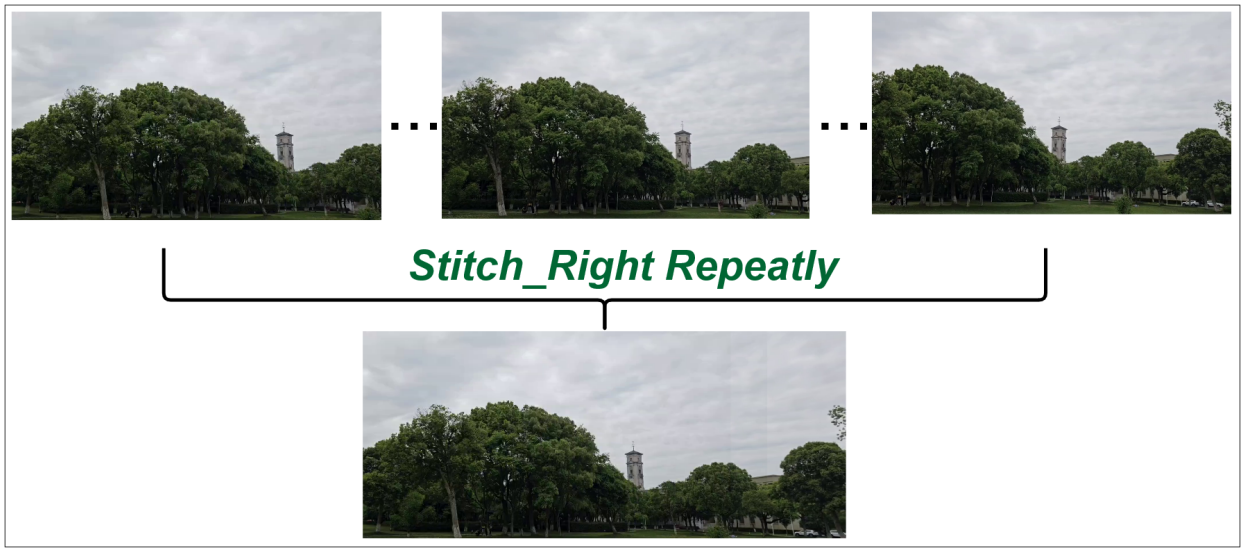

Figure 2.1: Typical Stitching Method Result on Short Video

As shown in Figure 2.2, the area at the far right of the panorama produces severe distortions. This is because the right frame's perspective needs to be warped at each stitching (will be explained in detail in later sections). In the process of continuous rightward stitching, the cumulative effect of this deformation has resulted in distortions on the far right side that have reached an unacceptable level. In this work, the author eliminated this traditional left-to-right order method.

Figure 2.2: Typical Stitching Method Result on Long Video

2.2 Overall Workflow & Feature 1

Figure 2.3 displays the workflow of the panorama generation process improved by the author. The primary distinction from traditional methods lies in selecting a central frame from the video as the base, to which other frames from both the left and right sides are appended. This approach could disperse the distortions to both the left and right sides, resulting in a more aesthetically pleasing panoramic image.

The algorithm employs two stack structures to store frames that need to be stitched from

both the left and right sides, subsequently achieving the final result through iterative invocations of the "*Stitch_Right*" and "*Stitch_Left*" functions.

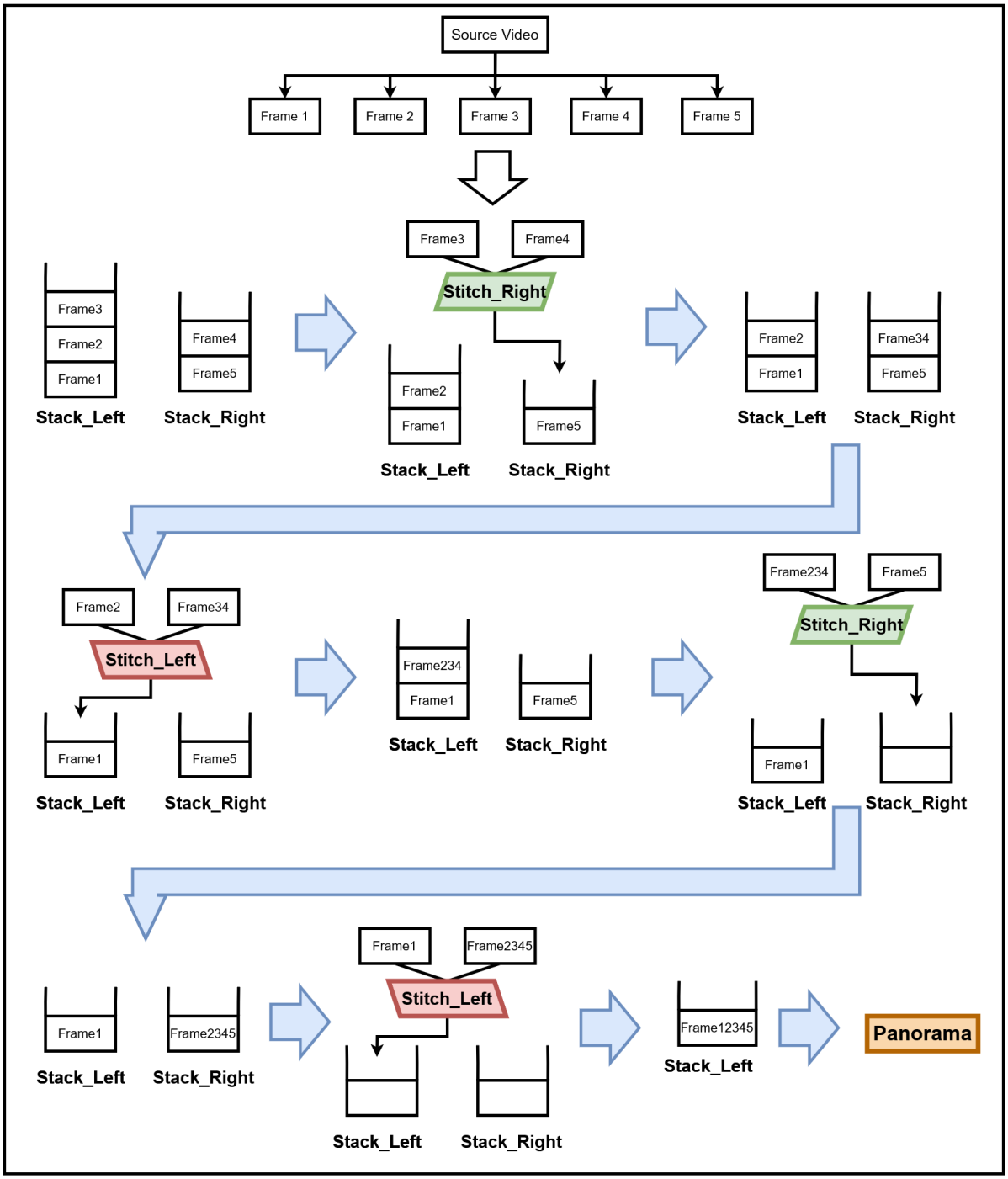

Figure 2.3: Panorama Generation Workflow

As shown in Figure 2.4, it can be observed that the improved algorithm retains more information from the video compared to the traditional algorithm. The resulting panoramic images exhibit a larger effective area and less noticeable distortion. Conversely, the results of the traditional algorithm display significant stretching distortions in the direction indicated by the red arrows in the figure.

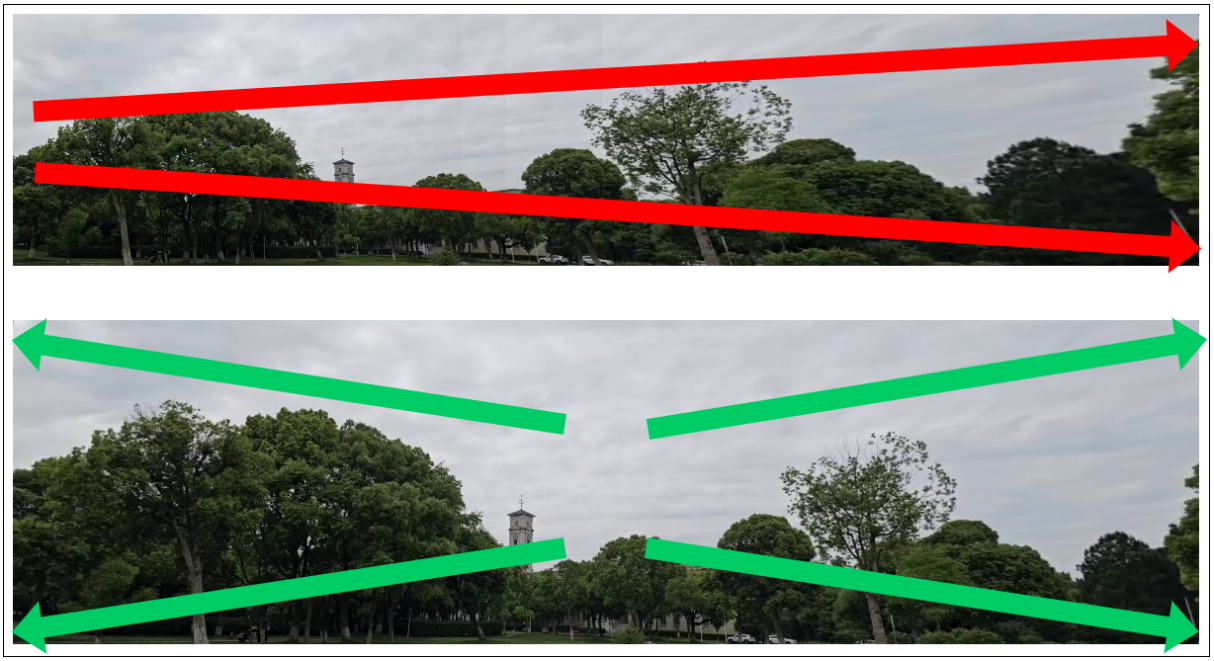

Figure 2.4: Comparison of optimization effect (Above: traditional method's result; Below: improved method's result)

2.3 Basic Stitching Algorithm

The previous section described the workflow for generating panoramas, and this section describes the most basic implementation process of the "*Stitch_Right*" function. It means that the achievement of the much harder "*Stitch_Left*" function and other optimization techniques and features will be explained in later chapters.

The implementation of stitching one frame f_A to the right side of another f_B requires the following five steps:

1. Identify the key points and derive local invariant descriptors of the two frames

2. Match the two frames' descriptors and get matched feature vectors

3. Utilize feature vectors to obtain the homography matrix

4. Execute warping transformation to the frame f_A and combine it to the right of f_B

5. Remove invalid black parts

Step 1: Identify key points and descriptors

The first step is to detect key points and extract local invariant descriptors of the two frames. The first line converts the input image from BGR color space to greyscale color space as the luminance information is sufficient for feature extraction. After that, the author used $SIFT\_create()$ and $detectAndCompute()$ functions from the OpenCV library. They would implement the Difference of Gaussian (DoG) [3] keypoint detector and the Scale Invariant Feature Transform (SIFT) [2] feature extractor algorithm directly and return. It is worth noticing that KeyPoint objects need to be converted to a NumPy array.

```
def detect_describe(self, frame):
    # Convert one color space to another color space
    gray = cv.cvtColor(frame, cv.COLOR_BGR2GRAY)

    sift = cv.SIFT_create()
    (key_points, descriptors) = sift.detectAndCompute(gray, None)

    key_points = np.float32([kp.pt for kp in key_points])
    return key_points, descriptors
```

Figure 2.5: Code implementation of Step 1

Step 2: Match the descriptors

Firstly, the $DescriptorMatcher\_create(BruteForce)$ method in OpenCV was applied. It can exhaustively iterate through the descriptors of both frames, calculate the Euclidean distances between them, and identify the minimum distance for each descriptor pair. Then the top 2 matches for each feature vector are obtained by calling KNN_matching. To avoid the case that some of the matches of these pairs are false positives, David Lowe's ratio test [1] is adopted to prune them. The ratio is set at 0.75 in this work.

```
def match_key_points(self, key_points_1, key_points_2, descriptors_1, descriptors_2):
    # Initialize BruteForce, using the default parameters
    # raw knn matches calculation
    descriptor_matcher = cv.DescriptorMatcher_create('BruteForce')
    knn_matches = descriptor_matcher.knnMatch(descriptors_1, descriptors_2, 2)

    # Get ideal match
    ideal_match = []
    for m in knn_matches:
        if len(m) == 2 and m[0].distance < self.__ratio * m[1].distance:
            ideal_match.append((m[0].trainIdx, m[0].queryIdx))
```

David Lowe' s ratio test

Figure 2.6: Code implementation of Step 2

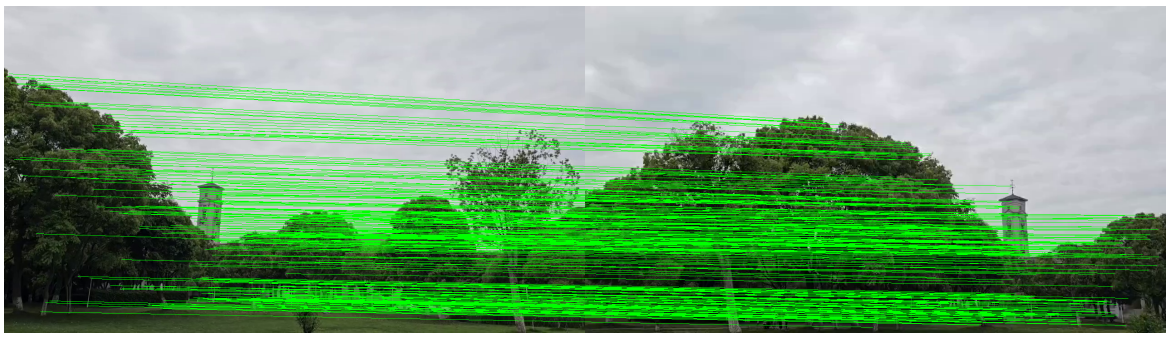

Figure 2.7: Matching Result Visualization

Step 3: Obtain homography matrix

This step utilizes the OpenCV's $findHomography()$ function to calculate the homography matrix. To accurately compute a homography between two point sets, it is essential to start with at least four matching pairs. In this work, the author uses more than just four matched points to enhance the reliability of the homography estimation. $reprojThresh$ is set with 4, which defines the maximum pixel "wiggle room" allowed by the RANSAC algorithm.

```
# A minimum of four matches are needed to compute a homography.
if len(ideal_match) > 4:
    points1 = np.float32([key_points_1[i] for (_, i) in ideal_match])
    points2 = np.float32([key_points_2[i] for (i, _) in ideal_match])

    # find the homography between the point1 and point2, S means the status of each matched point
    (homography_matrix, S) = cv.findHomography(points1, points2, cv.RANSAC, self.__reprojThresh)
    return (ideal_match, homography_matrix, S)
# Return None if at least four are not met
return None
```

Figure 2.8: Code implementation of Step 3

Step 4: Warping transformation

This step uses the function $warpPerspective()$ from OpenCV directly. Figure 2.9 displays the warped frame f_A, then it will be added to the right side of f_B. This process can be optimized by designing masks to eliminate chromatic aberration, it will be explained in the next section.

Figure 2.9: Code implementation & visualization of Step 4

Step 5: Remove black parts

This step utilizes the $findContours()$ function in OpenCV to obtain the coordinates of the main body's contour. Then the algorithm designed by the author would traverse the first row of pixels from right to left until the color is not pure black, deleting the right side. The obtained picture is the outcome of stitching f_A to the right side of f_B and it will be pushed into stack for further stitches. The next chapter will explain how to optimize this method so that videos with persistent shaking can be handled.

2.4 Optimization Method

This section will introduce four additional features of this work, which can get better panorama results, handle more complex input videos, and optimize the user experience.

2.4.1 Stitch_Left Algorithm

Contrary to what people might intuitively assume, code of *Stitch_Left* cannot be directly reused or rewritten for code of *Stitch_Right*. The key difference occurs when executing step four: warping transformation. The left frame will be warped and added to the left side of another frame. However, figure 2.10 displays that the warped frame in *Stitch_Left* would exceed specified boundaries if rewritten *Stitch_Right*'s codes directly.

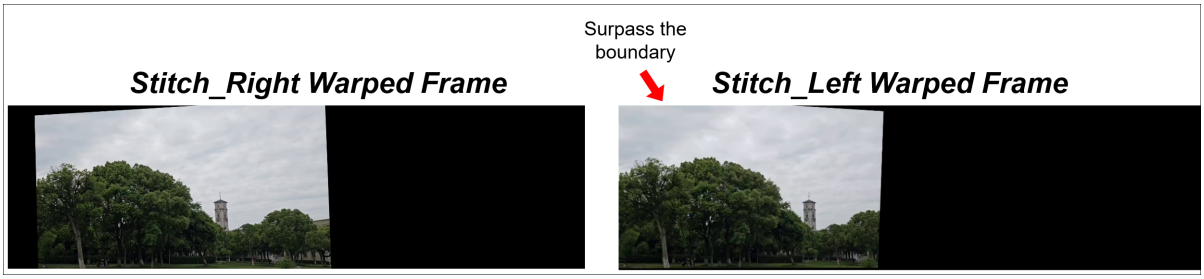

Figure 2.10: Difference of obtaining warped frame

The solution is to give the frame to be stitched a sufficient length of black area to the left in advance. The resulting graph and the stitching frame execute "match the descriptors" and "obtain homography matrix". This way the warped frame will fall within the valid boundaries. It is worth stating that the features of this section have been explained clearly earlier in Section 2.2.

```
def expand_right_frame(self, frame1, frame2):
    # expand frame2 in this function
    right_frame_height, _ , num_channels = frame2.shape
    # Determine the left fill width to add
    expanding_width = frame1.shape[1]
    expanding_area = np.zeros((right_frame_height, expanding_width, num_channels), dtype=np.uint8)
    expanded_right_frame = np.concatenate((expanding_area, frame2), axis=1)
    return expanded_right_frame
```

Figure 2.11: Key Step of Stitching Left

2.4.2 Eliminate chromatic aberration: Feathering

Figure 2.12 demonstrates the stitching result of the "basic stitching algorithm" introduced in Section 2.3. In the position indicated by the red arrow, there is a visible chromatic aberration discernible to the naked eye. This is due to the fact that the camera automatically adjusts the exposure when recording video, so there is chromatic aberration from frame to frame.

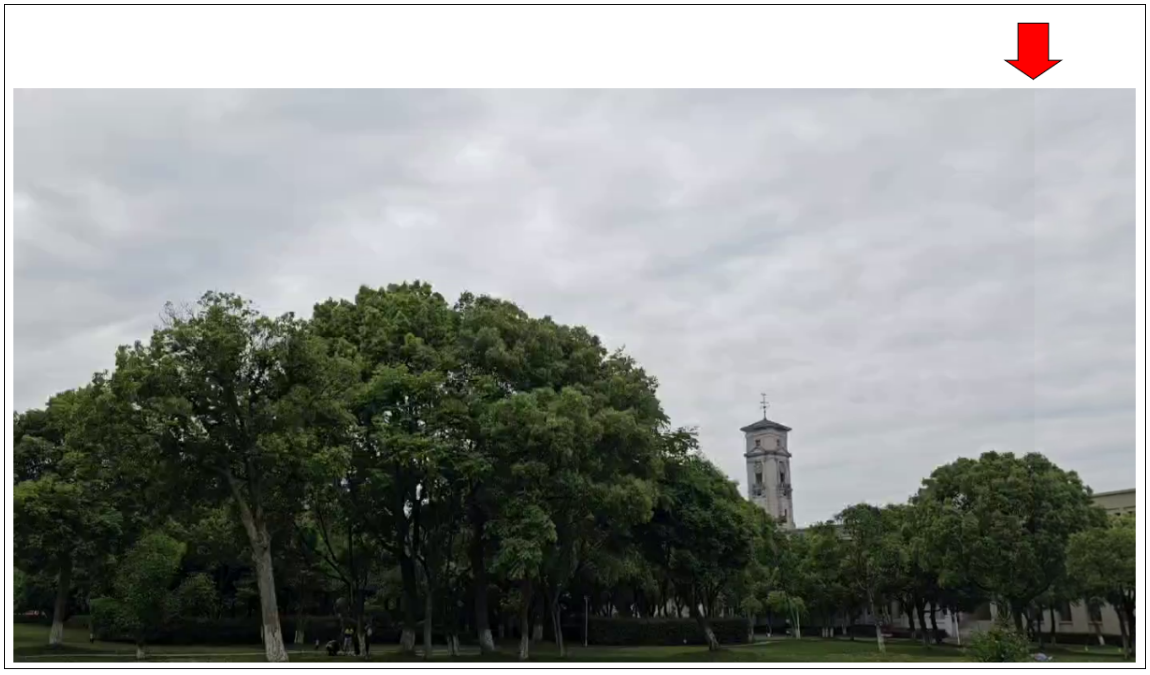

Figure 2.12: Chromatic Aberration Problem of Unoptimized Algorithm

In this work, the author designed a feathering algorithm to generate masks that could smooth the stitching performance. Smooth linear gradient masks are obtained respectively for both f_B and warped f_A. Figure 2.13 displays the *mask_generation* function for both "*Stitch_Right*" and "*Stitch_Left*", which has significant difference.

Stitch_Right Code Snippet

```python
def mask_generation(self, frame1, frame2, type_mask):
    stitch_outcome_height = frame1.shape[0]
    # Main difference 1!!!
    stitch_outcome_width = frame1.shape[1] + frame2.shape[1]
    # Main difference 2!!!
    mask_barrier = frame1.shape[1] - self.MASK_OFFSET

    blend_mask = np.zeros((stitch_outcome_height, stitch_outcome_width))

    if type_mask == "Left":
        linear_gradient_right = np.tile(np.linspace(1, 0, 2 * self.MASK_OFFSET).T, (stitch_outcome_height, 1))
        blend_mask[:, mask_barrier - self.MASK_OFFSET:mask_barrier + self.MASK_OFFSET] = linear_gradient_right
        blend_mask[:, :mask_barrier - self.MASK_OFFSET] = 1
    elif type_mask == "Right":
        linear_gradient_left = np.tile(np.linspace(0, 1, 2 * self.MASK_OFFSET).T, (stitch_outcome_height, 1))
        blend_mask[:, mask_barrier - self.MASK_OFFSET: mask_barrier + self.MASK_OFFSET] = linear_gradient_left
        blend_mask[:, mask_barrier + self.MASK_OFFSET:] = 1

    return cv.merge([blend_mask, blend_mask, blend_mask])
```

```python
stitch_outcome_height = frame1.shape[0]
stitch_outcome_width = frame1.shape[1] + frame2.shape[1]

# Generate the left part of the stitched picture
mask_for_left_part = self.mask_generation(frame2, frame1, type_mask = "Left")
stitch_outcome_left_part = np.zeros((stitch_outcome_height, stitch_outcome_width, 3))
stitch_outcome_left_part[0:frame2.shape[0], 0:frame2.shape[1], :] = frame2
stitch_outcome_left_part_masked = stitch_outcome_left_part * mask_for_left_part

# Generate the right part of the stitched picture
stitch_outcome_right_part = cv.warpPerspective(frame1, homography_matrix, (stitch_outcome_width, stitch_outcome_height))
mask_for_right_part = self.mask_generation(frame2, frame1, type_mask = "Right")
stitch_outcome_right_part_masked = stitch_outcome_right_part * mask_for_right_part
```

Stitch_Left Code Snippet

```python
def mask_generation(self, frame1, frame2, type_mask):
    stitch_outcome_height = frame1.shape[0]
    # Main difference 1!!! Since frame1 has been expanded
    stitch_outcome_width = frame1.shape[1]
    # Main difference 2!!!
    mask_barrier = self.MASK_OFFSET + frame2.shape[1]

    blend_mask = np.zeros((stitch_outcome_height, stitch_outcome_width))

    if type_mask == "Left":
        linear_gradient_right = np.tile(np.linspace(1, 0, 2 * self.MASK_OFFSET).T, (stitch_outcome_height, 1))
        blend_mask[:, mask_barrier - self.MASK_OFFSET: mask_barrier + self.MASK_OFFSET] = linear_gradient_right
        blend_mask[:, : mask_barrier - self.MASK_OFFSET] = 1
    elif type_mask == "Right":
        linear_gradient_left = np.tile(np.linspace(0, 1, 2 * self.MASK_OFFSET).T, (stitch_outcome_height, 1))
        blend_mask[:, mask_barrier - self.MASK_OFFSET: mask_barrier + self.MASK_OFFSET] = linear_gradient_left
        blend_mask[:, mask_barrier + self.MASK_OFFSET:] = 1

    return cv.merge([blend_mask, blend_mask, blend_mask])
```

```python
stitch_outcome_height = frame2.shape[0]
stitch_outcome_width = expanded_frame1.shape[1]  # different with "right" since frame1 has been expanded

stitch_outcome_left_part = cv.warpPerspective(frame2, homography_matrix, (stitch_outcome_width, stitch_outcome_height))
mask_for_left_part = self.mask_generation(expanded_frame1, frame2, type_mask = "Left")
stitch_outcome_left_part_masked = stitch_outcome_left_part * mask_for_left_part

mask_for_right_part = self.mask_generation(expanded_frame1, frame2, type_mask = "Right")
stitch_outcome_right_part = np.zeros((stitch_outcome_height, stitch_outcome_width, 3))
stitch_outcome_right_part[0:expanded_frame1.shape[0], 0:expanded_frame1.shape[1], :] = expanded_frame1
stitch_outcome_right_part_masked = stitch_outcome_right_part * mask_for_right_part
```

Figure 2.13: Code Snippet for Feathering

Figure 2.14 shows the processed f_A and f'_B, and the optimized stitching outcome. It can be observed that the chromatic aberration of the original position is eliminated and the whole picture looks smoother and more beautiful.

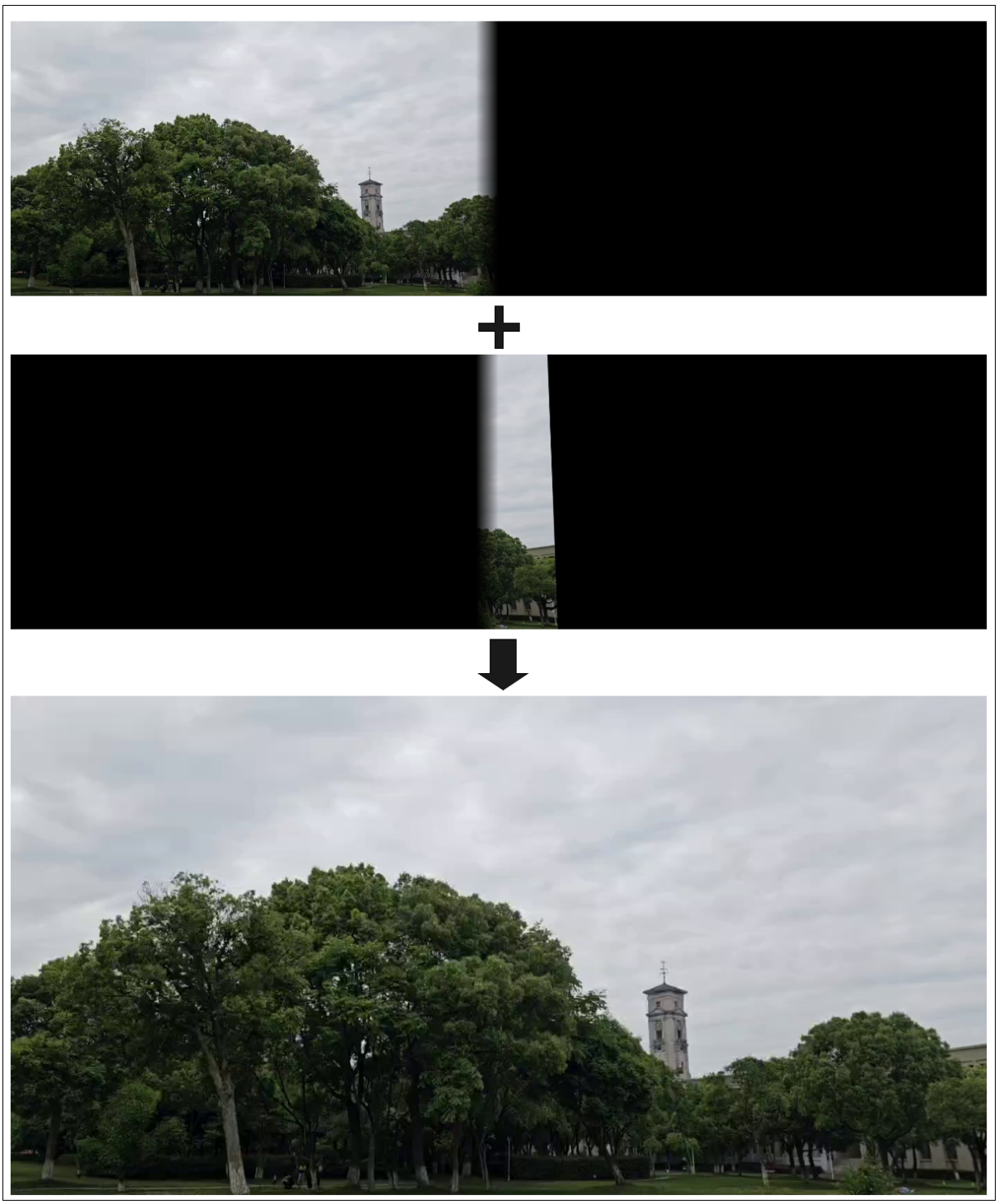

Figure 2.14: Feathering Outcome

2.4.3 Handle videos with persistent shaking

When the input video has continuous shaking, the obtained frames will have large differences in the vertical direction. As shown in Figure 2.15, there is a significant difference in the distance from the tiles at the top of the wall to the bottom of the image. This causes the stitched panorama to produce one or more black invalid areas that detract from the overall aesthetics of the image (as shown in Figure 2.16).

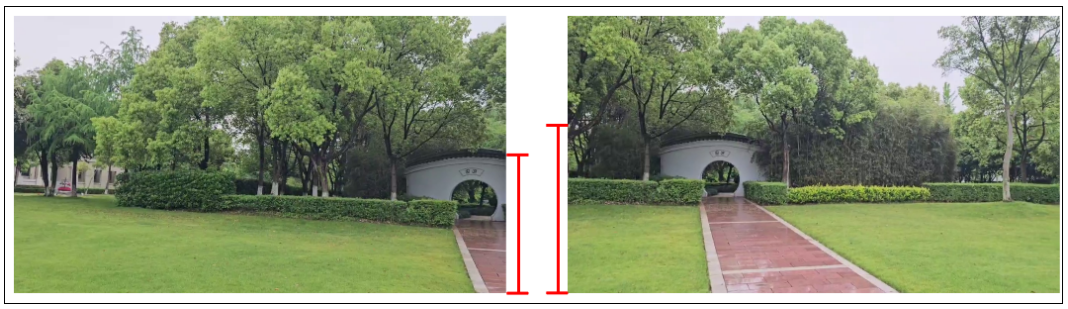

Figure 2.15: Frames in Shaking Video

Figure 2.16: Result of Shaking Video before Optimization

The first step to optimize this problem is to perfect step five, which is to remove black parts. The designed algorithm will traverse the entire stitched image and perform minimum cropping where there is black noise. For example, the noise can be wrapped in a $W * H$ rectangle, if $W > H$, then crop the entire H row, and vice versa if $H > W$, then crop the entire W column.

Once the width of the image has been cropped to a smaller size, all subsequent stitched images need to be cropped to the same width. In order to maximize information retention, and assuming a width difference of X, the image will be cut $X/2$ from the top and $X/2$ from the bottom. The final panorama is rightfully narrower than the width of the video.

```python
    def remove_black_borders(self, frame):
        gray = cv.cvtColor(frame, cv.COLOR_BGR2GRAY)
        # the grayscale image is converted to a binary image using threshold operations
        # Distinguish between non-black areas of the image and black backgrounds
        _, thresh = cv.threshold(gray, 1, 255, cv.THRESH_BINARY)

        # Step1
        contours, _ = cv.findContours(thresh, cv.RETR_EXTERNAL, cv.CHAIN_APPROX_SIMPLE)
        if contours:
            largest_contour = max(contours, key=cv.contourArea)
            # x, y are the coordinates of the top left point of the matrix
            # w, h are the weight and height of the matrix
            x, y, w, h = cv.boundingRect(largest_contour)

            # Step2
            for n in range(w):
                if frame[0, n, 0] == 0 and frame[0, n, 1] == 0 and frame[0, n, 2] == 0:
                    continue
                else:
                    begin_num = n
                    break

            #print(begin_num)
            cropped_frame_1 = frame[y:y+h, begin_num: x+w]

            # Step3
            for m in reversed(range(h)):
                if cropped_frame_1[m, 0, 0] == 0 and cropped_frame_1[m, 0, 1] == 0 and cropped_frame_1[m, 0, 2] == 0:
                    continue
                else:
                    num = m
                    break

            cropped_frame_2 = cropped_frame_1[:num, :]
            return cropped_frame_2
        # Return the original if no contours found
        return frame


def alignment_height(fig1, fig2):
    fig1_height = fig1.shape[0]
    fig2_height = fig2.shape[0]

    if fig1_height == fig2_height:
        return fig1, fig2
    elif fig1_height > fig2_height:
        diff = fig1_height - fig2_height
        top = math.ceil(diff/2)
        sub = diff - top
        aligned_fig1 = fig1[top: fig1_height-sub, :]
        return aligned_fig1, fig2
    elif fig1_height < fig2_height:
        diff = fig2_height - fig1_height
        top = math.ceil(diff/2)
        sub = diff - top
        aligned_fig2 = fig2[top: fig2_height-sub, :]
        return fig1, aligned_fig2
```

Figure 2.17: Code Snippet of Handling Shaking Video

2.4.4 UI Design

In order to facilitate better video selection, this work uses **tkinter** and **os** libraries to design a simple user-interactive interface. Figure 2.18 shows the main window of the UI, the user can choose any video on his personal computer (the path name cannot contain Chinese) by clicking the button “Select video file”.

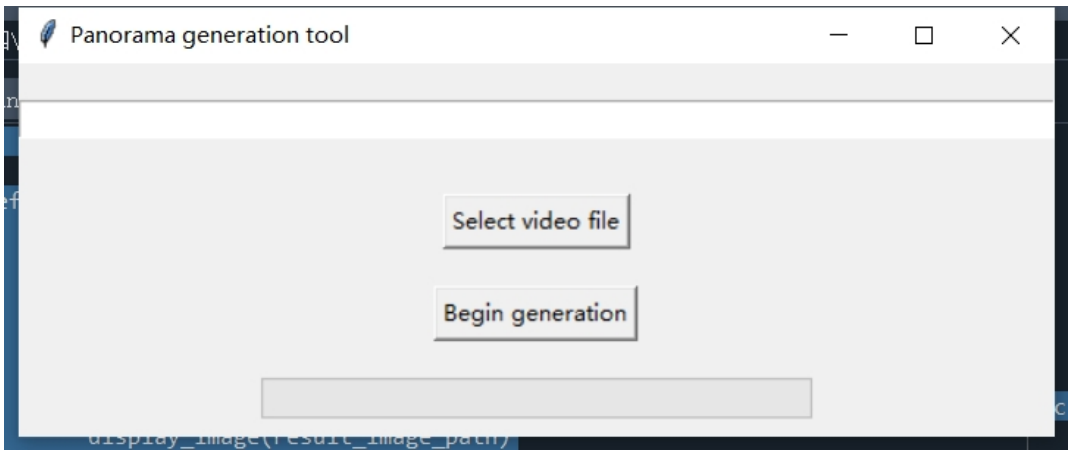

Figure 2.18: Main Window

After clicking the button "Begin generation", the *panorama_gen* will be called, and also pass in a function that reflects the progress of the run. The UI has a progress bar display section shown in figure 2.19.

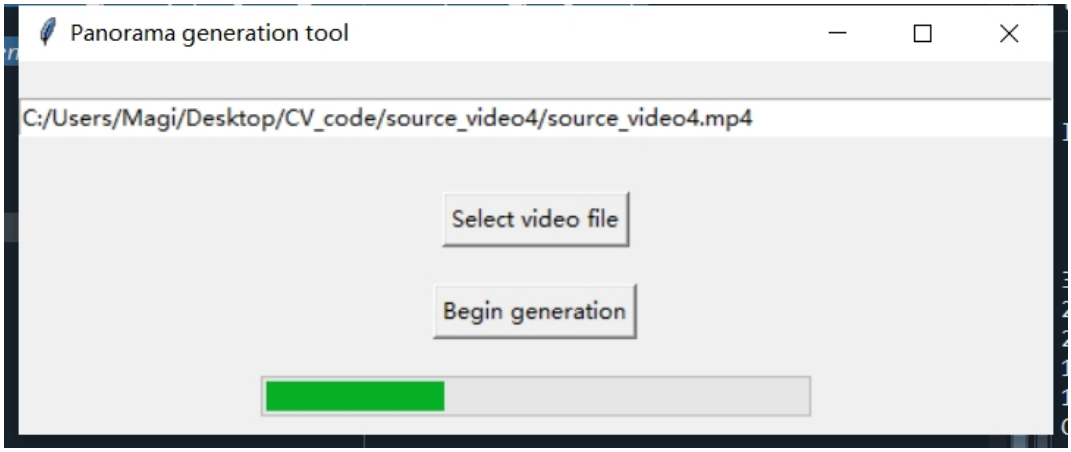

Figure 2.19: Progress Bar

If the generation is successful, the UI prints the success message and pops up the resulting panorama. If it fails an error message is returned.

Figure 2.20: UI when success

Chapter 3

Results Presentation and Analysis

This chapter will present the algorithm performance on different source videos. Here are 4 videos containing various scenes. Source video 1 and video 2 hope to demonstrate that the algorithm works for both natural and man-made landscapes. Source video 3 has persistent shaking in the vertical direction. Source video 4 has the same scene as video 1, but was shot at night.

Source Video1: trees and the bell tower

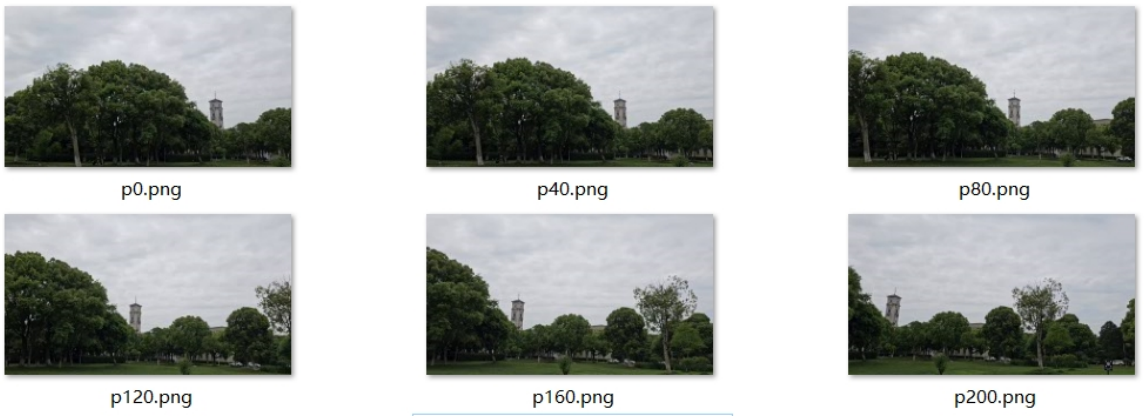

Figure 3.1: Frame Set of Source Video 1

Figure 3.2: Generated Panorama from Source Video 1

Source Video2: Starbucks Building

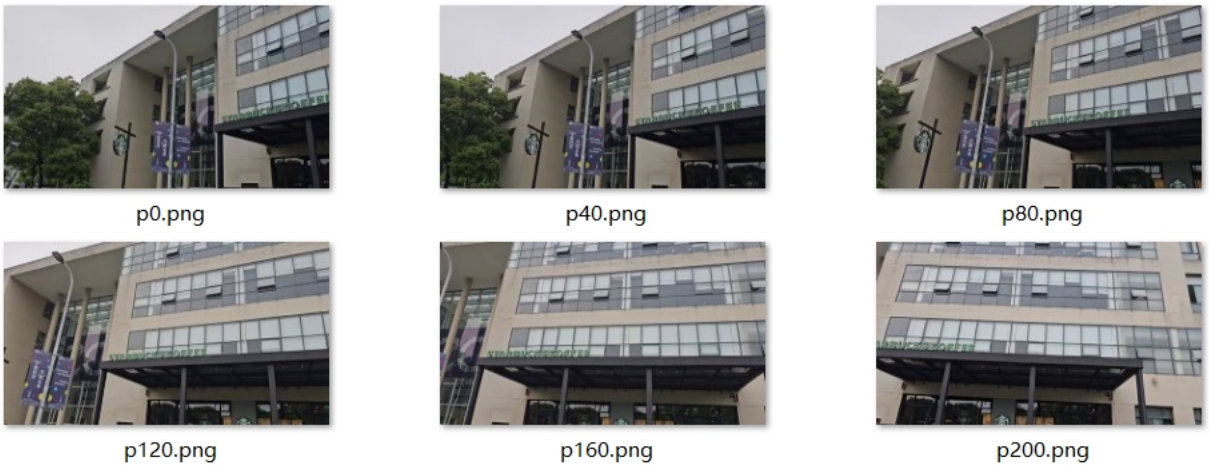

Figure 3.3: Frame Set of Source Video 2

Figure 3.4: Generated Panorama from Source Video 2

Source Video3: Walls and lawns (with persistent shaking)

Figure 3.5: Frame Set of Source Video 3

Figure 3.6: Generated Panorama from Source Video 3

Source Video4: Night mode of Source Video 1

Figure 3.7: Frame Set of Source Video 4

Figure 3.8: Generated Panorama from Source Video 4

It can be seen that the algorithm completed by the authors can be applied to a wide range of complex scenarios, and the resulting panoramas are free of distortions visible to the naked eye and in line with the aesthetics of the general public. The final images are free of chromatic aberrations. A detailed analysis will be developed in the next chapter.

Chapter 4

Strengths and Weakness

4.1 Strengths

The strengths of the author's algorithm are based on the proposed features or improvement methods introduced in Chapter 2.

Firstly, the author has overcome the limitations of the traditional left-to-right stitching algorithm by implementing a novel algorithm based on two stack structures that perform stitching from the middle frame outwards. This approach effectively disperses distortions to both sides, resulting in panoramic images that align more closely with human aesthetic preferences and appear as though they were captured directly, rather than being generated through computer vision techniques. In scenarios where the traditional method can handle video lengths of t without significant distortion, the author's improved algorithm can process videos of up to $2t$, thus doubling its performance capability.

Secondly, the author has implemented a feathering process during the image stitching, resulting in seamless panoramic images with no chromatic aberration. The designed feathering algorithm exhibits robust versatility across various environments, whether they be natural landscapes or high-rise buildings, regardless of the color tones, be they green or blue, or the time of day, whether daylight or nighttime. The generated panoramic images are indistinguishable to the naked eye from those taken directly through photography.

Thirdly, the author's algorithm is capable of processing videos with persistent shaking and generates panoramas that are completely unaffected by the quality of the video source.

Even in the presence of significant shaking, the algorithm trims away the mismatched segments, ensuring that the resultant image is devoid of any black noise.

Finally, the author has utilized expertise in human-computer interaction to design a clear and concise user interface for the panoramic image generation algorithm. Users can effortlessly select the original video, and the integration of a progress bar has significantly enhanced satisfaction.

4.2 Weakness

Firstly, the author conceived and implemented the generation of panoramic images solely in the horizontal direction. However, the creation of vertical panoramas is equally critical. This significant aspect was overlooked in the initial phase of the project. In future work, the author intends to address this oversight and enhance the comprehensiveness of the problem analysis, ensuring that solutions are not restricted to the most immediately apparent aspects.

Secondly, despite the algorithm proposed by the author significantly mitigating the issue of distortion at the far end, it still fails to fundamentally resolve the issue when dealing with particularly long videos. Visible distortions are still evident at the extreme right of Figure 3.6. In future work, the author intends to preprocess each frame by applying a reverse distortion to neutralize the cumulative warping effects during the stitching process.

Finally, as shown in figure 4.1, the author initially considered only vertical shaking during shooting, neglecting the potential variations in camera angles that occur in practical scenarios. Future work will aim to address the effects of such rotational movements.

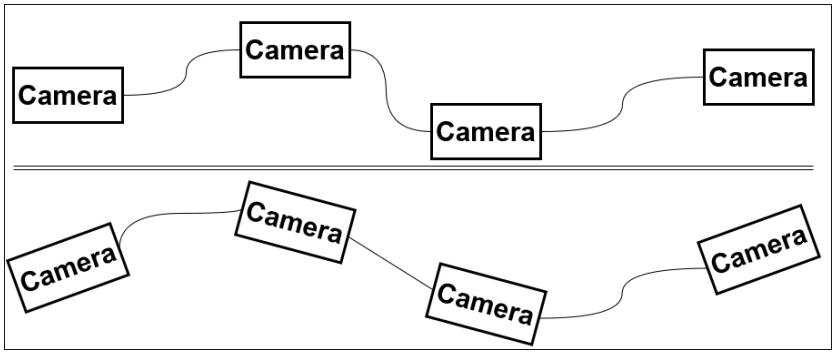

Figure 4.1: Two different Shooting methods

Bibliography

[1] Lowe, D. G. Distinctive image features from scale-invariant keypoints. *International journal of computer vision 60* (2004), 91–110.

[2] Lowe, G. Sift-the scale invariant feature transform. *Int. J 2*, 91-110 (2004), 2.

[3] Marr, D., and Hildreth, E. Theory of edge detection. *Proceedings of the Royal Society of London. Series B. Biological Sciences 207*, 1167 (1980), 187–217.