Project Report

1. Introduction

This project is designed to construct a “spatio-temporal” image (STI) through several ways in order to find and characterize video transitions. To carry out the construction of STI, we have tried six different methods. Firstly, we copied the pixels from the middle column or row to the STI. Then we established the chromaticity histogram to compute histogram intersections of each column or row in each frame with the previous time instant. In addition, we also implemented the threshold versions of histogram intersections. The language used in this program is **C++** and the main library used is opencv2.

2. STI construction

2.1 Copying pixels

The first approach we adopted was copying the pixels directly from each frame’s middle column or row to the target STI.

1. Pseudocode

```
1. For each frame
2.     Copying the middle column of frame to the STI_C
3.     Copying the middle row of frame to the STI_R and side away
4.  Display the STI_C, STI_R
```

The STI_C has the same number of rows as does the original video frame and the column number equal to the number of frames in the video. The STI_R has the same number of rows as the number of columns in the original video frame and the number of columns equal to the number of frames in the video.

2.2 Histogram intersection

The second approach we adopted was constructing a chromaticity histogram for each column or row of each frame and comparing it to the previous frame’s column at the same location and then adding up the smaller one to the STI.

1. Pseudocode

```
1.  For each frame
2.    For each row or column or row
3.      Generate previous chromaticity histogram
4.      Generate current chromaticity histogram
5.      Compare previous and current histogram and add the smaller
          one to the intersection sum
6.    Divide the intersection sum by the number of columns or rows
7.    Copy the value of intersection sum of each frame to STI_H
8.  Display the STI_H
```

The STI_H has the same number of rows as the number of columns in the original video frame and the number of columns equal to the number of frames in the video.

2.3 Histogram intersection with threshold

The third approach we adopted was based on the second method but added a boundary condition on the intersection value. We set the boundary to be 0.3.

1. Pseudocode

```
1.  For each frame
2.    /the same as histogram intersection/
3.    If the intersection sum is smaller than the threshold then set
    it to 0
4.    Else set it to 1
5.    Copy the value of intersection sum of each frame to STI_HI
6.  Display the STI_HI
```

The STI_HI has the same number of rows as the number of columns in the original video frame and the number of columns equal to the number of frames in the video.

3. Test result

The STI_HI has the same number of rows as the number of columns in the original video frame and the number of columns equal to the number of frames in the video. To test the actual performance of each method, we selected a test MPEG video, which has 640*360 pixels per frame. To test the correctness of each test, we added a wipe in the video manually.

3.1 Copying pixels

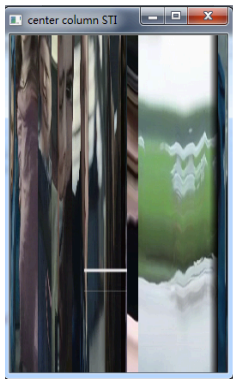


Figure (a)

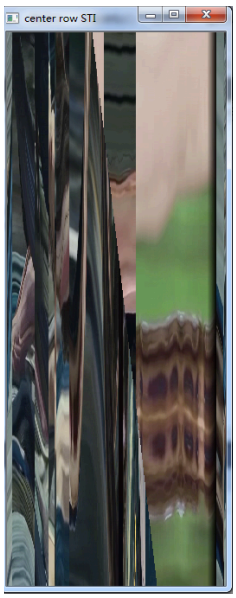
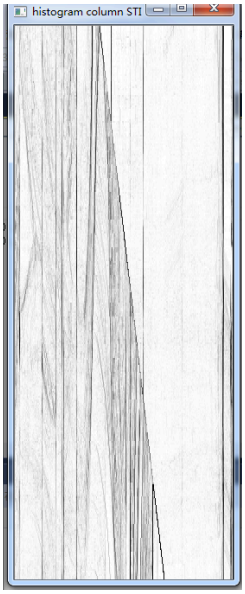


Figure (b)

Figure (a) is the result of copying pixels from the middle column and Figure (b) is the result of copying pixels from the middle row and side away. From two figures, we find that the width of two are identical corresponding to the same number of frames. Due to nice quality of video, both column and row copying clearly show the abrupt transition of the video.

3.2 Histogram intersection



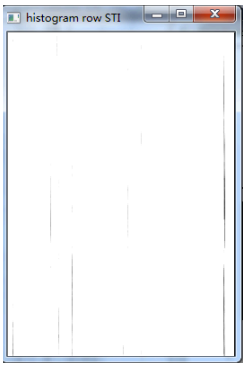
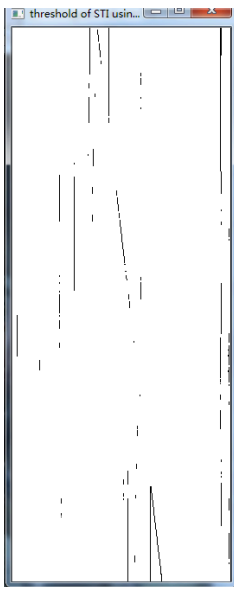
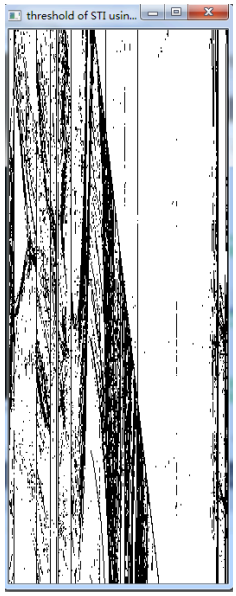


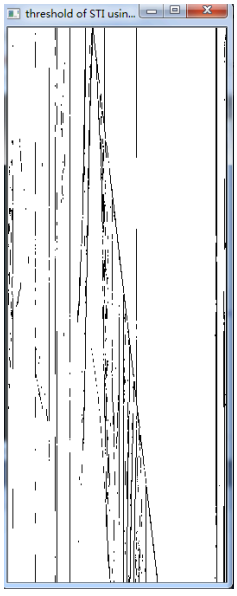
Figure (c) Figure(d)

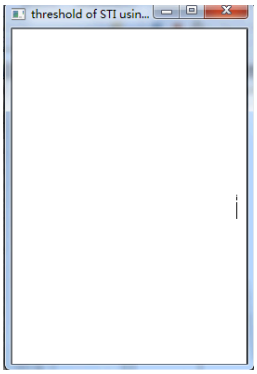
Figure (c) is the result of doing histogram intersection for each column of each frame and Figure (b) is the result of doing histogram intersection for each row of each frame. Similarly the width of two figures are the same as the number of frames. Figure (c) shows a cleaner output compared to the copying ones and provides a clean sets of zero values down a straight diagonal. However, the result of histogram intersection on rows is quite bad and useless for us to find the wipe. We think the uselessness of row intersection depends on the actual direction of wipe in the video.

3.3 Histogram intersection with threshold









Figure(e): 0.3-col Figure(f): 0.9-col Figure(g):0.7-col Figure(h):0.3-row

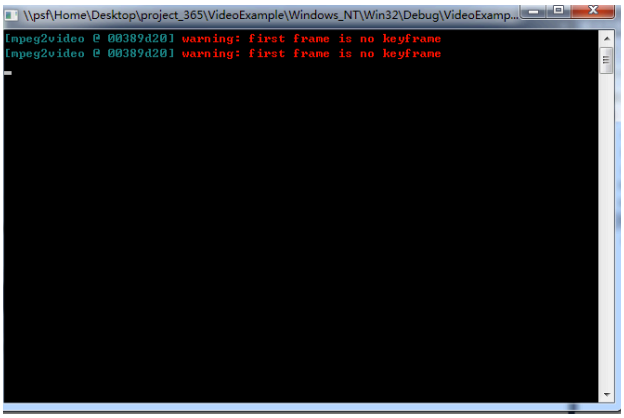
Figure (e), (f) and (g) is the result of doing histogram intersection for each column of each frame with threshold holds on 0.3, 0.9 and 0.7 and Figure (h) is the result of doing histogram intersection for each row of each frame with threshold on 0.3. Similarly the width of these figures are the same as the number of frames. To investigate the relationship between the result and the threshold, we tried different thresholds in the range of [0.01,1]. Comparing the first three figures, we find that threshold around 0.7 has the relative best performance.

3.4 Discussion

Firstly, the effect of using column or row to construct each STI through different ways depends on the actual direction of transition in video. Secondly, the performance of copying pixels depends on the quality of the video or noisy level. So it is not good enough to find the transition compared to other methods. Thirdly, the threshold version has the best performance but it is crucial to select an appropriate threshold to implementation.

4. Solved problems

When we ran the program, we found that the control desk always showed the following sentence. “first frame is no keyframe”



To find the reason for this problem, we did research on this strange result. In order to calculate the intersection between two frames, the program should have both the data of current frame and previous one. At the beginning, we try to reduce the amount of memory the whole program needed to store frames of the video by using the function: VideoCapture::set(CV_CAP_PROP_POS_FRAMES). This function can set the index of frame which is to be captured next. However, when we apply this function to the MPEG video, it will lost several (approximately 10) frames and we find the reason. The reason is that OpenCV uses a “key frame” technology to decode video when reading some high compression video format like MPEG. Therefore, it will skip some frames at the beginning to find the first key frame. According to this reason, we choose to store the previous frame in a matrix variable and read frames form video in sequence.