Multi-Cam Video Stitching: Panoramic Video Option During a Skype Call

by

Daniel Appello and Danielle Erickson

Submitted in partial fulfillment of the requirements
For the Computer Science Major
Dickinson College, 2013-2014

Professor John MacCormick, Advisor
Professor Timothy Wahls, Supervisor

May 9, 2014

Abstract

Multi-Cam Video Stitching: Panoramic Video Option During a Skype Call

by
Daniel Appello and Danielle Erickson

The goal of this research is to expand on the work done previously by Professor John MacCormick that incorporated multiple cameras into a Skype call and allowed each participant on the two-way call to choose how they view themselves as well as opposite party. Our research intends to extend this existing system by adding the option of a panoramic video for each user that is comprised of the feeds from each of the cameras of that particular user. The research aspect of this implementation goal concerns the delicate balance of speed and quality of the output video. The challenges to overcome include relying on equipment that the typical user has available and outputting the stitched video in real time. This paper introduces the concepts of image stitching and video stitching, as well as explains the current progress and intended future work.

Chapter 1

Introduction

1.1. Research Question

In the first two months we spent working on our research project, we were able to effectively isolate not only the desired end goal of our work, but also understand and generate the particular question we wish to investigate. As previously stated in the project proposal, our research falls under the scope of computer vision and image processing. The work extends the previous work done by Professor John MacCormick, which allows a user to control multiple cameras on both ends during a Skype conversation. The end goal of our research project is to integrate an option into his existing system that will allow a user to combine the input from multiple attached video cameras into a single, seamless panoramic video. This option could be applied to conference calls, lectures over Skype, etc. The panoramic video option must operate in real time, which led us to our research question: Can a good panoramic video be stitched in real time during a Skype call with the equipment of the average person?

1.1.1. Definitions

Before explaining our progress, it is important to first define a few terms. As used in our research question, we define a "good panoramic video" to be one video that is the compilation of two or more separate video streams. This video should be seamless, meaning there should be no traces of the placement or existence of the input videos. A good panoramic video should be in real-time, meaning that the output video should have the same frame rate as the input videos. Additionally, there should be no noticeable

differences in exposure or lighting in each stitched frame of the output video, or in other words, there should be unified exposure. Lastly, a "good panoramic video" should be stitched such that the videos are in-sync. This means that each frame of the output video should be comprised of frames from the input videos that were taken at the exact same point in time. Another clarification should be made regarding the implications of the terms "equipment of the average person." We intend to make our system marketable to the general public and therefore will be working with components that are commonly used and relatively inexpensive. In particular, we define these components to be computers without dedicated GPUs (graphics processing units) and basic webcams ranging from about $15-$30. To reiterate, the research to be done with this project is to find the most time-efficient way to stitch multiple video streams together seamlessly on minimum hardware machines.

1.2. Literature Review

1.2.1. Image Stitching and Video Stitching Examples

The first action taken regarding the aforementioned research question was to investigate different image stitching techniques as well as existing implementations of image stitching and panoramic video stitching. One such example is the system built by M. Adam, C. Jung, S. Roth and G. Brunnett described in their paper "Real-time Stereo-Image Stitching using GPU-based Belief Propagation." Their system stitched together multiple HD videos at a respectable (between 10 fps and 45 fps) frame rate. To do so, they applied a new algorithm for video stitching that encompassed many different techniques for accuracy, but relied heavily on the capabilities of a distinct GPU (graphics

processing unit) exploiting the nVIDIA CUDA Framework. By using the GPU and its multiple cores, which allow the use of parallel computing, they were able to output a video with a very efficient frame rate. By splitting up the various tasks of the algorithm for analyzing and altering the incoming video stream, they state the GPU approach "is about 40-50 times faster than the not optimized, single core CPU based implementation" (Adam, Jung, Roth, Brunnett, 5). The use of the GPU enabled speed without compromising the quality of the output video. Because their research requires the GPU to function in real-time and because we have chosen to make our system compatible with the equipment of the average user, we cannot adopt their stitching approach. However, if in the future we wish to increase stitching quality further, we may consider adding an option for users with a GPU.

Additionally, M. Brown and D. G. Lowe wrote a paper entitled “Automatic panoramic image stitching using invariant features” which described a system they built for stitching still images that used the image stitching technique called feature matching. Brown and Lowe primarily strove to create an effective solution to the problem of automatic image stitching. Automatic image stitching requires that unlike many current solutions, there is no initialization or user input required before stitching. In an attempt to produce outputs that are more accurate as well as aesthetically pleasing, they viewed their problem as a multi-image matching problem. In doing so, their solution is insensitive to ordering, orientation, scale, and illumination of the input images. In addition, they achieved outputs that eliminated noise through a technique called multi-band blending, which produced panoramas of good quality. However, it took 15 minutes for the output panorama to be generated. Because we wish to stitch together the frames of our videos in

real time, our stitching algorithm must produce an output image in milliseconds rather than in minutes. Consequently, this article implies that we may have to sacrifice some aspects of quality in the image stitching in order to meet our time constraints and to have our stitching happen automatically.

Another inspirational example of video stitching was the existing software VideoStitch. VideoStitch takes input videos from multiple cameras and stitches them into one panoramic video, but does not use its own image-stitching algorithm. Instead, it requires the user to specify or create a calibration (using either PTGui or Hugin image stitching software). This allows the user to have a lot of control over the stitching and also allows for customizability in terms of lens used, crop parameters, orientation and position, curve, etc. Although we aim to make our system automatic (no user input is required), we may wish to add similar features of customizability in the future. VideoStitch is user friendly and uncomplicated but processes images too slowly to work with a live feed.

1.2.2. Image Stitching Techniques

In addition to giving us insight on features to add or techniques to deal with timing, through our literature review we learned the basics about image. Chapter 9 in the book *Computer Vision* by R. Szeliski contains an abundance of information on how images are stitched, the different types of algorithms in existence that accomplish the task, as well as information on adjusting or fixing the image and the possible applications of image stitching. One of the most important things we obtained from the chapter by Szeliski was the explanation of a homography. As described by Szeliski, a homography is

the mathematical relationship that maps the pixel coordinates of one image to those of another image. These relationships can be computed using different motion models, "from simple 2D transforms, to planar perspective models, 3D camera rotations, lens distortions, and mapping to non-planar (e.g., cylindrical) surfaces," (Szeliski, 2011, p 378). Fortunately, we have decided to use an existing algorithm in OpenCV to compute the homography, eliminating the necessity to understand the mathematical complexities that exist in these image-stitching algorithms. However, we may find after computing and applying the homographies generated by the OpenCV algorithm to the frames of our videos that adjustments are desired to improve the image. If this is the case, we may choose to try new algorithms that have features described by Szeliski such as gap closing, global alignment, parallax removal, and bundle adjustment.

1.2.3. OpenCV

OpenCV, which stands for Open Source Computer Vision, is a framework written in C++, Java, Python and Matlab that allows high-level analysis and modification of images and videos. There are many built-in classes that facilitate image processing in areas such as facial detection, facial recognition, motion tracking, object identification and image stitching. We will be using these built-in classes, mainly the image stitching class, as a black box since our research focuses more on the application of image stitching rather than the actual algorithms involved.

Chapter 2

Fall Semester Progress

2.1. Preliminary Work with OpenCV

As mentioned previously, we are using the OpenCV Framework to compute the homographies for the frames of the videos to be stitched. Having never used OpenCV before, we decided we needed to familiarize ourselves with the framework before doing anything more complicated. The first step in this process was to install OpenCV, which turned out to be a quite complicated process on a Mac. Although Multi-Cam operates only on Windows, at this stage in our project we decided to work with Mac out of convenience and lack of access to Windows PCs. The installation process involved using MacPorts to install OpenCV via command line. After installation, there was a lot of configuration in Xcode, the IDE we chose to use, that had to be done in order to have access to the correct frameworks, libraries, and header files.

After successfully installing OpenCV, we began to familiarize ourselves with its capabilities. Fortunately, OpenCV provides a sizable amount of tutorials, which were helpful to run while testing our setup. Additionally, there was a tutorial that contained an example of a simple image stitch and led us to investigate the Stitcher class. Using the provided example as an instruction on how to import images, what methods to use to compute and apply a homography, and how to display an output, we stitched together our own images as seen in Figure 2.1 and produced the output shown in Figure 2.2. We were very satisfied with the output and felt that at this time no improvements to the quality of

the image were needed. However, with more tests, we may find that we wish to add some of the features and image adjustment techniques as described in Chapter 1.

Figure 2.1: These images were taken by a camera that rotated from a single point.

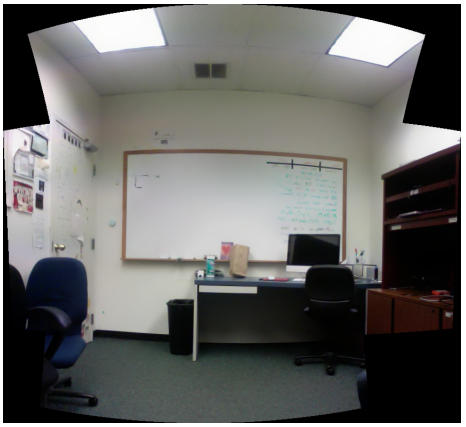

Figure 2.2: This is the output image obtained by our program that took as input the images in Figure 2.1 and combined them using the Stitcher class.

2.2. Video Stitching

2.2.1. Simple Videos

The next step after stitching together two images was to attempt to stitch together two pre-recorded videos. To keep things simple and to test the efficiency of the OpenCV algorithm we recorded two separate videos with still cameras and limited motion. We chose to work with stationary video cameras initially because this ensures that the consecutive frames have almost the exact same stitching parameters. This allowed us to experiment with how often the homographies are recomputed (or simply applied) in an attempt to balance time and accuracy of the output.

For the first experiment with videos recorded with stationary video cameras, the homography was computed for each frame. The output (with 640 x 480 resolution) was

displayed frame by frame as soon as the computation finished and the stitching parameters were applied (this will be referred to as Approach #1). We found that this technique was too slow because the output video displayed a new stitched frame about every second. Consequently, due to the fact that with still images and still cameras the homography is essential the same from frame to frame, we tried an approach (Approach #2) in which the homography was only computed once at the beginning of the video. After this computation, these stitching parameters were used to stitch together the remaining frames. As expected, this approach resulted in a faster output. This result can be seen in Figure 2.3 indicated by the fact that first data point for the blue line (Approach #1) is lower than the first data point for the red line (Approach #2).

2.2.2. Output Speed

In order to compare the results of the two aforementioned approaches we calculated the Frames Per Second. The FPS for each approach was calculated using OpenCV's built-in timing methods, getTickCount() and getTickFrequency(). The FPS was calculated by dividing the number of frames in the video (which happened to be 133) by the total time it took to produce the output. To get the total time, we subtracted the tick count at the start from the tick count at the end (both obtained by getTickCount()) and then divided by the tick frequency (obtained by getTickFrequency()).

For comparison, film is typically shot at 24 FPS while our video was outputting at about 2 FPS using Approach #2. The output of Approach #1 had an even slower frame rate of .89 FPS. Even though there was improvement in speed with Approach #2, we were still dissatisfied with the speed of the output.

2.2.3. Experimentation with Resolution

In attempts to improve the FPS of the output, we experimented with different resolutions for both Approach #1 and Approach #2. The results shown in Figure 2.3 indicate that as the resolution decreased, the frame rate increased. Additionally, there was a very significant increase in FPS for the resolution of 240 x 180 for both approaches when compared to the other resolution. Because there was only a difference of 1.93 in the FPS between the two approaches for the smallest resolution, in the future, if we choose to compute the homography on frames of this resolution, we may reconsider Approach #1, especially when accounting for the possibility of moving cameras.

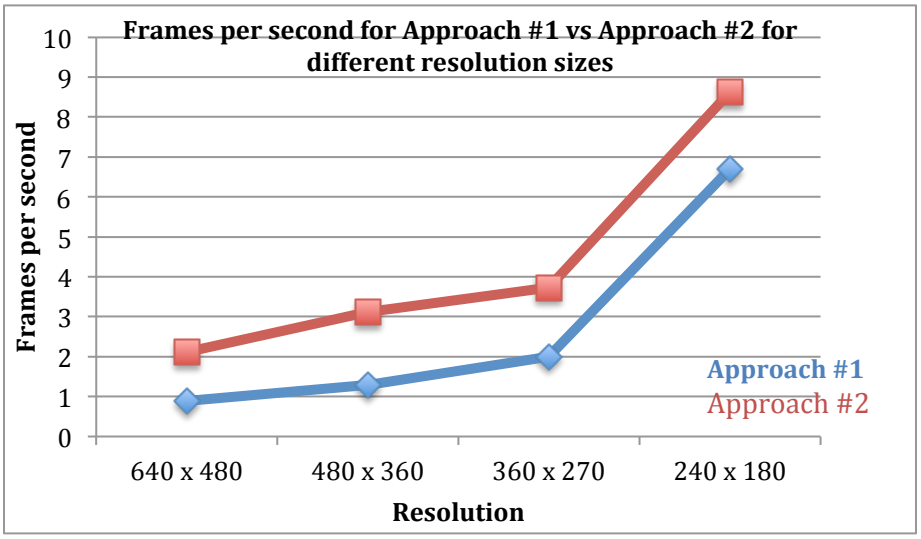

Figure 2.3: This graph shows the difference in FPS of our two approaches as well as the effects of resolution size on FPS.

Chapter 3

Spring Semester Progress

3.1. Pre-Recorded Videos

3.1.1. Speed

Our main concern as we entered the spring semester was the speed of our outputted videos. At the end of the Fall semester our fastest videos were roughly 9 fps with the homography computed only once initially (Approach #2) and a resolution of 240 x 180. Because the frame rates for the cameras used (Apple MacBook Pro iSight built-in camera, HP Webcam 2100, Logitech Webcam C500) fall between 15 fps and 30 fps, it was essential to stitch together pre-recorded videos with a frame rate of over 15 fps.

Our approach to further increase the frame rate was to build upon the previous work done with resolution and the frequency of the homography computation (computing the homography once with a resolution of 240 x 180). This solution was to modify the parameters within the Stitcher class, specifically the three main parameters: compositionResol, registrationResol and seamEstimationResol. These parameters determine the resolution, or size, that inputted images are resized to during the associated stitching process. RegistrationResol is used in the registration of the images to be stitched while compositionResol and seamEstimationResol are used to place the images together. We decided to try modifying all three of these parameters, but hypothesized that it was the composition parameters that would have the greatest effect on performance. To perform this test, each parameter was modified while the other two remained at their default values.

RegistrationResol : Default .6		SeamEstimationResol : Default .1		CompositionResol : Default -1	
Value	Frame Rate	Value	Frame Rate	Value	Frame Rate
1	5.75 fps	.01	16.06 fps	-1	5.45 fps
.6	6.66 fps	.025	9.78 fps	-.5	6.97 fps
.1	7.14 fps	.05	7.23 fps	-.25	6.09 fps
		.1	6.26 fps	-.1	0 fps
		1	5.60 fps		

Figure 3.1: This table displays each Stitcher parameter, its default value, its modified values and the frame rates that were experienced at each value.

As evidenced in Figure 3.1, the greatest improvement was found when changing the seam estimation to .01 instead of .1. With this change, we successful stitched together pre-recorded videos with a frame rate of 16.06. Since this method of modifying the seam estimation parameter seemed to be the most successful in increasing performance, we decided to carry out all further experiments with seamEstimationResol set to .01. This approach will be referred to as Approach #3. Figure 3.2 shows further tests of Approach #3 in comparison to Approaches 1, 2 and 3. These videos had moving subjects but still cameras.

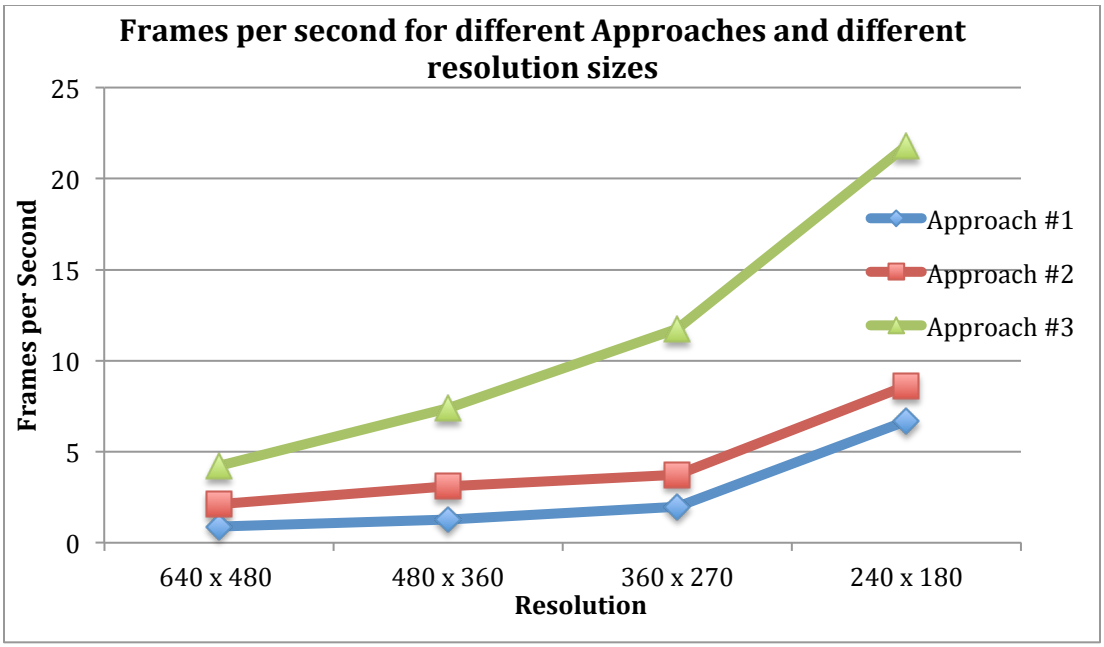

Figure 3.2: This graph shows the typical frame rates of each approach.

As evidenced by Figure 3.2, Approach #3 is much faster than our previous two approaches, especially for the smallest resolution (240 x 180). The speed reached was 21.8 fps, which can be considered real-time.

3.1.2. Image Quality

Throughout our work with pre-recorded videos, there were noticeable issues concerning quality. For example, there were fluctuations between frames. In other words, from frame to frame a certain area of the output image would have slight changes in color and/or location. Because the homography was computed only once and applied for each subsequent frame it was not surprising that there were some quality issues. We attempted to solve this problem by adjusting the exposure of the frames, but did not have the required information on the input frames to use the tools available in OpenCV.

We predicted that the cause of these fluctuations was a combination of exposure and lack of synchronization of the frames. Because the frames being stitched together were not recorded at the exact same time (by nature of being pre-recorded), variables in the image such as light and shadow were not the same. That said, we predicted that this issue would be solved when working with live feeds because the images to be stitched together would be recorded at the same time. This work with live feeds is described in Section 3.2.

3.1.3. Multiple Cameras

Because this program will likely be used for remote conference calling, it is necessary to make the program scalable. In other words, it is desirable to make the panorama as large as possible, thus incorporating multiple cameras. We chose to stitch together three pre-recorded videos. The code needed to update the program from supporting two cameras to three cameras is relatively simple and concise and therefore easily scalable. Additionally, there was only a slight effect on speed: the average frame rate for 3 pre-recorded videos was 7.2 fps and the frame rate of two pre-recorded videos was 8.62 (this data was taken with Approach #2 as described in Section 2.2.1 and a resolution of 240 x 180).

3.2. Live Feeds

3.2.1. Speed

Because our goal was to stitch in real-time, we aimed for an output video frame rate above 15 frames per second. We chose this number because for all of our tests we

used the built in iSight of the MacBook Pro. As mentioned in Section 3.1.1, this built in camera has a maximum frame rate of 30 fps but adjusts the frame rate to 15 frames per second in low-light conditions. For this reason, we deemed an output real time if it surpassed 15 fps. With the alterations applied as discussed in Section 2.2.3 (Approach #3 and resolution 240 x 180), live feeds were able to obtain up to 22 fps. However, stitching the homography once for live feeds isn't practical because there is no limit on how long the program will run and what a user will do while it runs. To address this issue, we decided to compute the homography every *x* frames. When tried after every 20 frames, there was a negligible change in fps, but there was a clear pause in the output video each time the homography was computed. To eliminate this pause, we created a thread that would compute the homography in the background.

3.2.2. Multi-threaded approach

After researching a few different ways of implementing threads in C++, we decided to use POSIX threads, or Pthreads, which are actually designed for C. A Pthread was created and designed to run in the background and recalculate the homography every five seconds. Because the goal of the Pthread is to eliminate any waiting on the homography computation, the background thread was implemented without any locks. Instead of using locks to modify shared variables, the thread creates temporary copies of the captured frames to make sure the input data isn't modified in the middle of the homography calculation. After the homography is computed, the thread is designed to sleep for a given time before repeating the process. This allows us to control how often a new homography is computed.

Because the calculation of the homography is performed by the background Pthread, the performance of the main thread performing the composition of the panorama is not affected. As can be seen in Figure 3.3, the amount of time the thread sleeps before recomputing the homography in the background does not have any noticeable effect on the main thread's frame rate. Each trial was run for 3 minutes with Approach #3 and a resolution of 240 x 180. The differences in frame rates between trials can possibly be attributed to the previously discussed fact that the iSight automatically changes its FPS depending on available light (Section 3.2.1).

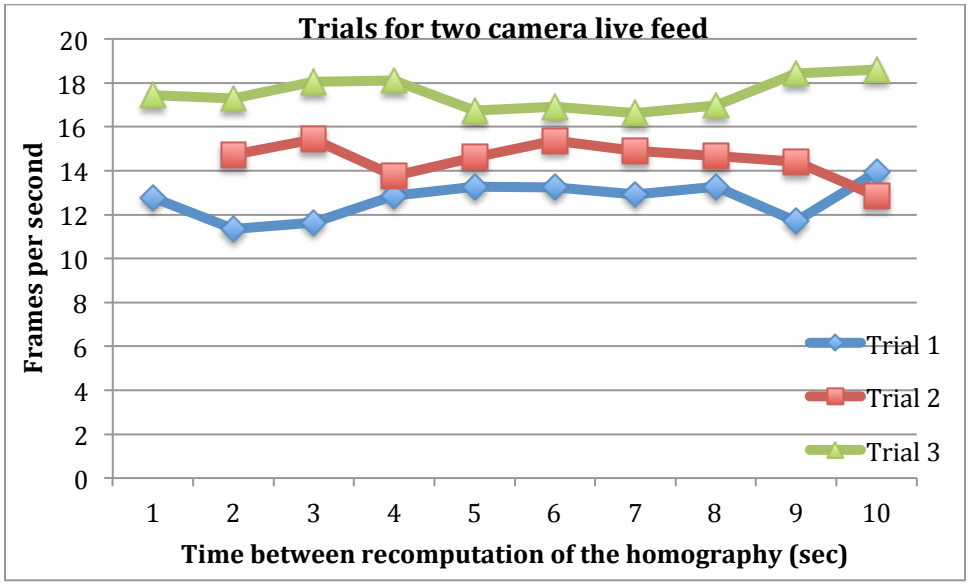

Figure 3.3: This graph shows the frame rates of three trials with differing amounts of time between recomputation of the homography by the background thread.

Implementing a thread into our project resolved the issue of pauses in the video, but resulted in an occasional error caused by the composePanorama() method of the Stitcher class that never appeared when our program was single-threaded. The error appears to be caused by an empty frame being inputted to the method, but our tests have

not been able to detect any errors with the inputted frames. This must be investigated further in the future as mentioned in Chapter 4.

3.2.3. Multiple Cameras

After securing the frames per second with live feeds, we decided to add a third camera. Similar to our results with pre-recorded videos, the live feed with three cameras was slower than the live feed with two cameras. With Approach #3 and the thread sleeping for 4 seconds before recomputing the homography, stitching live feeds from three cameras had an average frame rate of 9.18 fps. In contrast, stitching live feeds from only two cameras had a frame rate of 18.11 fps. Because three pre-recorded videos had a slower frame rate than two pre-recorded videos (see Section 3.1.3), it was not surprising that three live feeds stitched together with a slower frame rate than two live feeds, however with the live feeds the difference in frame rates was much greater: a difference of 9.07 fps versus the difference of 1.42 fps with pre-recorded videos. This could be attributed to the overhead of having to fetch the frame from the camera.

3.2.4. Quality

As mentioned before, the issue of synchronization was solved by moving to live feeds and subsequently resulted in fewer fluctuations for stitching still images. When moving subjects were introduced, fluctuations sometimes appear where the cameras overlap. They can be described as a portion of the image switching between the feed from one camera to the feed from the other camera for two consecutive frames. We predicted that one reason this was happening was because one of our cameras had a yellowish tint

whereas the other appeared much clearer. In an attempt to remedy this, we experimented with different ways to normalize the frames before they are stitched. However, when trying to run this code, we received an error related to memory access. This error is similar to the error mentioned earlier, in Section 3.2.2, which we believe to be caused by the thread.

One quality issue that occurred only with live feeds was the appearance of occasional black screens or flashes of just one camera's image. This issue appeared to be hardware related: the cameras would drop frames and have errors decompressing the videos. In an attempt to understand these issues and possibly learn how to remedy them, tests were run to determine if application use affected these errors.

The results of these tests indicated that heavy application usage did indeed have a strong impact on the number of dropped frames and decompression errors. With heavy application usage (opening and closing Microsoft Excel, iPhoto, PhotoBooth, Google Chrome, and Mac Mail) there were on average 24.25 decompression errors and 290 dropped frames each minute. In contrast, when there was minimal app usage (Microsoft Excel, Microsoft Word, Google Chrome, and Mac Mail open but not being actively used) there were on average 2.5 decompression errors and 3.2 dropped frames each minute.

One hypothesis for the increase in these issues when there is more application usage has to do with the increased competition for CPU time. Because the camera is not allotted as much CPU time, less frames can be processed, and therefore they appear to have been dropped. However, regardless of the increased number of dropped frames and errors, there were no perceivable differences in quality between the two tests. To clarify, each dropped frame or decompression error did not result in a black screen or flash of one

camera's input, and there was no difference in the number of these occurrences between the two tests. This fact implies that perhaps it was not the dropped frames or decompression errors that caused the aforementioned quality issues.

However, we still believed that the reason behind the occasional black screens or flashes of one camera's input was the dropping of frames or decompression errors. Consequently we investigated a solution to the dropped frames. We researched the possibility of repeating the most recently stitched image whenever a frame is dropped. We found the location in the code where the camera input was being processed and where it was determined that a frame was dropped. This code was an .mm file written in Objective C. Unfortunately we were unable to trace this code to our own and therefore could not tell when a frame was being dropped.

3.2.5. Shape

An additional quality issue that became apparent with live feeds was the stitching shape. Occasionally the images would stitch together in untraditional or unhelpful orientations. Some of these shapes can be seen in Figure 3.4.

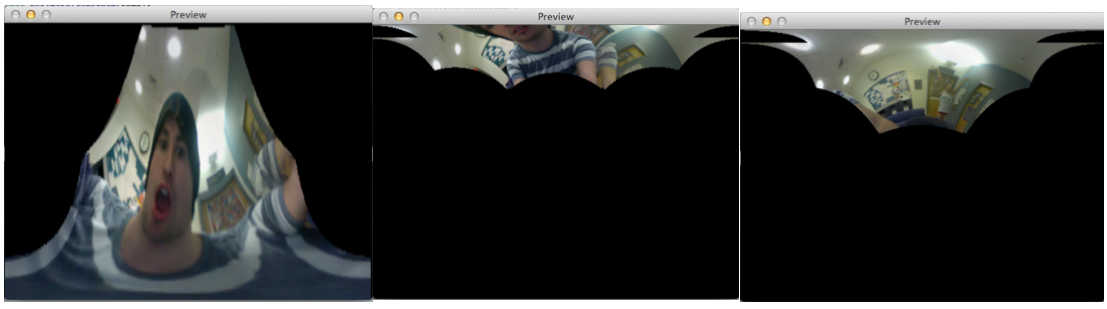

Figure 3.4: These images show various shapes our videos were rendered in.

We believe that these stitching issues have to do with the camera placement. However, the poor stitching is not consistent with certain camera positions.

3.2.6. Performance

Tests were run to determine the overall effect on the frame rate if other applications are running. Twenty trials were run with the application stitching for 3 minutes while Mac Mail, Microsoft Word, Google Chrome, and Microsoft Excel were being used. Additionally, twenty trials were run with no additional applications running. These trials were run with a resolution of 240 x 180. As expected, the frame rate was slower when other apps were being used because of the competition for the CPU. The average frame rate with the aforementioned apps running was 11.73, whereas the frame rate with no other apps running was 16.39. Additionally, when other apps were running, the program quit 2 out of 10 times because of an unexpected exception. With no apps running, the program quit 0 out of 10 times. This data suggests that there is a significant effect on both speed and reliability when other apps are running as well as a tendency to error out.

3.2.7. Adding a Mutex lock

To eliminate the possibility of the program erroring out, we decided to add a Mutex lock. We implemented the Mutex lock that exists with Pthreads. The lock ensures that the homography is not calculated by the background thread at the same time the main program is stitching together the images, or in other words it ensures mutual exclusion.

After running the program containing the lock for 3-minute trials over a period of 7 hours, there were no instances of erroring out.

Additional tests were done to test the efficiency of the program with a lock versus without a lock. Without the lock, 3-minute trials were run intermittently for 4 hours without the program crashing. Further tests were done comparing the program with the lock and without the lock. 10 1-minute trials were run for each program. Out of the ten trials, the program without the lock errored out 2 times and the program with the lock errored out 0 times. On average, there was no significant difference in the frame rate (13.86 with the lock, 14.85 without the lock), however there was a notable difference in the number of times the homography was recomputed. With the lock, there were 20 recalculations, in other words the homography was re-calculated roughly every 3 seconds. In contrast, in the program without the lock, the thread recomputed the homography 10.63 times on average, or every 5.64 seconds.

As expected, the brief pauses in the videos (Section 3.2.1) that happened each time the homography was computed reappeared when we introduced the lock. This happened because with the lock the main thread could not apply the homography at the same time the background thread recalculated the homography (this was possible without the lock). These pauses are less than a second long and we believe that this is an appropriate sacrifice to guarantee that the program will not crash.

3.2.8. Impact of the thread

As stated in Section 3.2.2, we originally added the background thread as a way to eliminate the pauses in the video that were seen each time the homography was

recomputed. When the lock was introduced, these pauses reappeared. Consequently, it seemed necessary to determine if it was worthwhile to have the thread or if the thread with the lock was making no difference at all. Tests were run comparing the program with the thread and the lock (Approach #3, a resolution of 240 x 180, with the thread sleeping for 3 seconds) and the program without the background thread. Since the program without the thread recalculates the homography every *x* frames, we first ran 1-minute trials to determine the average number of frames between computations for the program with the thread and the lock. We found that in these trials the homography was recomputed on average every 42 frames. Additionally, the average frame rate for these trials was 18.8 fps and there were 19.8 homography calculations on average. Next we ran trials with the program without the thread and set it to recompute the homography every 42 frames. However, this program had an average frame rate of 9.54 fps and recalculated the homography on average 12.4 times in each minute test.

A separate set of one-minute trials was done when the threaded program with the lock recomputed the homography roughly every 60 seconds. The program without the background thread was set to recalculate the homography every 60 seconds as well. Overall it had an average frame rate of 6.17 and 6 recalculations, whereas the threaded program with the lock had an average frame rate of 15.21 fps and 8.18 recalculations.

To reiterate, although each program was recomputing the homography every *x* frames (42 for the first set of trials, 60 for the second set), the threaded program with the lock did more re-calculations of the homography and had a faster frame rate. At this time, this result does not make much sense due to the fact that the lock ensures that the recomputation of the homography does not happen concurrently in the threaded program.

Additionally, the fact that implementation of the non-threaded program is simpler implies that it should run faster than the threaded program.

Chapter 4

Future Work

At this point we have successfully stitched together up to 3 videos with live feeds in real-time. We have no concerns about the frame rate with two cameras, but future work could include explorations of more than three cameras and their respective frame rates. There is also work to be done to ensure that when other applications are open the frame rate does not decrease so drastically.

Additionally, there is potential work to be done on the quality of the output image. So far we have not successfully implemented any image correction techniques such as exposure compensation (mentioned in 3.1.2.). In the future we will continue to investigate this issue in an attempt to remedy quality issues such as fluctuations and changes in color of the output image. We plan to work more on the normalization of the frames (Section 3.2.4) and resolving this error. We can also expand on the work done to reshow a stitched output whenever a frame is dropped by the camera (Section 3.2.4).

We hope to correct the shapes discussed in Section 3.2.5. We found an algorithm that could possibly solve this problem, but we are unsure if it is possible within OpenCV. However, due to time constraints and the complexity of the algorithm, it seems unlikely that we will be able to achieve flattening of the image.

Lastly, although there is remaining work to be done on the quality of the output, the program is potentially ready to be integrated into Prof. MacCormick's Multi-Cam or a video chat software such as Skype.

References

Adam, M., Jung, C., Roth, S., and Brunnett, G. "Real-time Stereo-Image Stitching using GPU-based Belief Propagation." *hArtes,* Seventh Framework Programme of the European Commission.

Brown, M., & Lowe, D. G. (2007). "Automatic panoramic image stitching using invariant features." *International Journal of Computer Vision*, 74(1), 59-73.

Guerrero, M. (2011). "A Comparative Study of Three Image Matching Algorithms: Sift, Surf, and Fast." *DigitalCommons@USU*. Utah State University.

MacCormick, J. (2012). "Video Chat with Multiple Cameras." Dickinson College Technical Report.

"OpenCV, MATLAB, or AForge??" *OpenCV Tutorial C++*. http://opencv-srf.blogspot.com/

Szeliski, R. (2011). "Image stitching (Chapter 9)". *Computer Vision*, pages 375-408. Springer.

Tang, W., Wong, T., and Heng, P. (2005) "A System for Real-time Panorama Generation and Display in Tele-immersive Applications." The Chinese University of Hong Kong.

VideoStitch. "User Guide."(2013) Retrieved from: https://www.videostitch.me/videostitchhow.html