StitcHD

Final report

Sam Hatfield
Luke Yeager
Heather Young

Department of Computer Science and Engineering
Texas A&M University

May 8, 2012

Table of Contents

1. Executive summary

This NASA funded project is intended to develop a video system which will ultimately replace windows on their spacecrafts. They have requested that we attack the problem of stitching high-definition video feeds together as part of the goal of the overall system. While keeping in mind the end goal of a comprehensive video system, we will focus on the mechanics of streaming several high-bandwidth, high-definition camera feeds and stitching them together quickly and reliably into a panoramic video stream.

Our goal is to provide NASA with a program that helps a user to quickly learn how the mechanics of stitching videos together work. A user should be able to use our graphical user interface (GUI) to explore the different parameters used in the different modules of our system to determine how each parameter affects the speed and quality of the resulting stitched video stream.

We implemented video stitching by separating our code into several threads. First, there is a separate CameraCapture thread for of the cameras, allowing them all to stream simultaneously. Then, once the images are retrieved from each camera, there are two operations which must be done to create a stitched panorama from the images. The images must be analyzed to find the relationship between them, and the images must be stitched together into one panoramic image.

First, the images have to be analyzed based on the overlap region between the cameras to calculate how much each image needs to be transformed to align the images appropriately. This is done by utilizing some of OpenCV's computer vision algorithms to calculate a perspective transformation between two images. These parameters are called homographies, so we have dubbed these threads, "Homographiers." Several of these homographies are calculated between several pairs of images so that in the end, all of the images can be stitched together. Each of these calculations can be done independently, so they are each done in a separate thread.

Once the perspective transformation parameters have been calculated, the main thread can take images from each of the CameraCapture threads, and parameters from each of the Homographier threads, and use them to stitch the images together. The stitching is done on the GPU, since it is an easily parallelizable problem. And the output images are displayed on a GUI built using the Qt framework. This GUI not only displays the stitched video stream, but it also allows the user to make changes to various configuration options on the fly, and see the resulting changes in the output video immediately. These configuration options let the user specify several options about how the images are stitched together, and a plethora of options regarding how the Homographiers should calculate the homographies.

The results for this project can be seen in the attached demo videos. The GUI allows the user to adjust the parameters until they are content with the results, up to the theoretical limits of the algorithms used and the physical limits of the hardware used. Our project also helps the user to understand how the positioning of the cameras and the relative distance to different objects in the scene affect the quality and stability of the output video. We feel we have met our goal of creating a useful tool for learning the mechanics of video stitching.

The timing results of this project, however, are disappointing on several points, due not to errors in the code, but to limitations introduced by our USB 2.0 cameras and from sharing a single GPU with other processes on the computer. Solutions to these problems are given in the results section below. We had initially aimed to stitch high-definition video at high-definition speeds, but

quickly realized that hardware limitations would not let us reach that objective. So, rather than spending more money on expensive hardware, we have provided mechanisms for measuring these limitations alongside developing a program that is useful for educational purposes regardless of what performance the hardware is able to deliver.

The experience we had working together on a team was a positive one, as was our relationship working closely with NASA. It was rough in the beginning trying to discover the most productive way to communicate and get things done, but these problems were noticed and resolved about a fourth of the way into the semester. The biggest issue was time management, and finding a time that every member of the team was available. The most rewarding part of our project to all of us was being able to work for a real client, especially such a well-renowned company like NASA. The fact that our research will go on to serve important endeavors such as space exploration was excellent motivation this semester to develop a quality product.

2. Project background

This NASA funded project will help to develop a video system which will ultimately replace windows on their spacecrafts. NASA has split the design idea into two projects, with each project assigned to a separate team. The SkyView team, another CSCE 483 Senior Design group, has been assigned the task of controlling the camera with a multi-touch interface. NASA has requested that our team, the StitcHD team, attack the problem of stitching high definition video feeds together. While keeping in mind the end goal of a comprehensive video system, we will focus on the mechanics of streaming several high-bandwidth, high-definition camera feeds and stitching them together quickly and reliably into one, higher-definition stream.

2.1. Needs statement

In NASA's proposed solution to replace spacecraft windows, there is a need to stitch multiple high-definition videos together. Currently, no solution exists which can account for camera movements dynamically.

2.2. Goal and objectives

The goal of this project is to provide NASA with a helpful research tool to explore the effects of different parameters in the stitching pipeline on the quality and stability of the output video. This should be presented with an intuitive user interface and provide detailed timing information in order to make informed decisions about which configuration best suits the user's needs.

Our first objective is to build a user interface that allows a user to stitch videos together with minimal effort or previous knowledge about computer vision. We also aim to provide detailed timing information about each step in the stitching pipeline which can be easily analyzed to calculate the performance of our project according to different hardware and stitching parameter configurations. We aim to support four input camera streams simultaneously, according to our given project assignment. The output must be a single panoramic image that is stitched using real-time video, not recorded video as some other projects have done. Real-time signifies that there should be minimal lag between retrieving a camera frame and displaying it to the user. We also aim to maximize the speed of the stitching, even for high-definition camera resolutions (720p). In addition, the edges between the individual streams should be seamless: the user should not be able to distinguish between the images from the different cameras in the output video, but it should appear as though the video came from a single camera, at least in the center

of the image. And finally, we should be able to allow for camera and scene movement, since the final camera system will allow each of the cameras to pan and tilt independently.

2.3. Design constraints and feasibility

Though our solution to the video stitching problem should apply to an arbitrary number of cameras, we have constrained ourselves to four cameras for this project. This was done partially for budgeting concerns, but also to keep the complexity at a manageable level for a semester's work and to temper the bandwidth requirements. We used USB 2.0 cameras in this project, and the USB 2.0 architecture has known bandwidth limitations. We avoided this issue by buying additional hardware with extra USB controllers and by spreading the cameras out to separate controllers. In addition to the general USB limitations, the Orbit AF cameras we ordered are known not to support high frame-rates when running in high resolution, so we were unable to truly explore the limits of our hardware using these cameras.

Simply stitching video together is extremely feasible, and we had the basics of the problem solved in a few weeks. The difficult part was building the framework to allow arbitrary camera movements and, at the same time, perform the calculations in parallel to maximize the use of advanced hardware.

Another constraint that we encountered this semester was the scenes which we were able to observe. For most of the semester we were restricted to viewing the scene of our lab, which turned out to be one of the most difficult scenes from which to create a panorama, due to the relative variation of the distance of objects from the camera array. Later in the semester we were able to take some demo videos of various scenes around campus, but we were never able to test our project on a star field or a flat horizon, which would be the scenes observed on a spaceship or on a rover.

2.4. Literature and technical survey

Among all the research our team completed, one project stood out as particularly relevant to what we are trying to accomplish. The Imaging System for Immersive Surveillance (ISIS) is a cutting-edge 360-degree camera array developed by MIT and the Pacific Northwest National Lab for the United States Department of Homeland Security. ISIS does essentially all that our project hopes to: stitches multiple HD video feeds into a large panorama in real time. The system is truly impressive, producing a final image of 100 megapixels. Operators can access and manipulate the video feed in real time, zooming into a specific area and tracking individual people as the move across the room. The fact that ISIS was built with off-the-shelf commercial products is encouraging, but the price isn't: the Department of Homeland Security spent about $3 million in funding the research and installation of ISIS.

A previous Computer Engineering senior design group, PanoVision, stitched the output of seven webcams together to create one panoramic image. They were able to achieve this on very minimal hardware and with some very basic webcams. Our own project differs in several ways. NASA has asked us to stitch HD video feeds together, while this project was able to use low-resolution cameras and were further able to sacrifice video quality for the sake of minimizing computational intensity while stitching. They were also able to do calibration during initialization, whereas we want to do calibration in real-time. This is because the cameras we will be using will be, eventually, controlled from within the spacecraft in terms of zoom, tilt, etc. We want to be able to do calibration in real-time, constantly recalculating the stitching parameters.

We also do not want to sacrifice video quality from the cameras. Therefore, we will have to explore using more expensive hardware than did members of this previous project.

Immersive Media is a company that provides 360-degree panoramic video solutions to a variety of clients. One of their most well-known products is Google Street View, a feature of Google Maps that allows people to move from the birds-eye perspective of a map to a ground-level view. Immersive Media achieves this by driving the streets of a city in a car mounted with a special multi-angle camera which is valued at $45,000. These pictures are a static panorama, but Immersive also specializes in 360 degree video. Most of their video is stitched in post-processing, but they say they can stitch live video "with some effort".

Jonathan Clark, an amateur blogger, has taken on the herculean task of creating a room with multiple virtual windows. Each window will display a piece of an extremely high-resolution video, and the windows will adjust their content based on where the viewer's head is located in the room, creating an illusion that the video is really existing right outside these windows. His goal is to display video at 100 megapixels and 60 frames per second, though he admits it may not be possible. The overlap between this project and our own is mostly over the stitching of the HD images together. Unfortunately, Clark has not reported any progress on this front. He has, however, commented on some of his research into feature matching. He is trying to use it to find the location of the viewer's head, while we want to use the algorithm on the HD video feed itself. Also, he plans to capture all the videos separately from each other, and then stitch them together into one enormously high resolution video later, while we aspire to do that stitching in real-time. His major concerns over processing power are over the calculation of which image to display in each "window," while our concerns are over the capturing and processing of simultaneous video. This project was started in 2009, and hasn't seen any development since September of 2010, so Clark may have abandoned the project.

The MindTree foundation has developed a Video Analytics Suite which stitches multiple HD camera feeds into one panoramic video in real-time. They boast a superior algorithm which uses dynamic feature matching to stitch images together preserving geometric integrity throughout the stitched image, at the loss of some clarity. Shown below is an impressive demonstration. Note how the roof is flat across the image, but also note the loss of image quality.

Figure 1: MindTree

The MindTree solution can only stitch 3 camera feeds together, whereas we want to design a system which can stitch arbitrarily many video feeds together. Also, MindTree depends on Texas Instruments' DM6467 platform, which is capable of analyzing and stitching two feeds in real time, but which will not be sufficient for our needs.

Several researchers from the University of Hong Kong published an article in 2005, "A System for Real-Time Panorama Generation and Display in Tele-Immersive Applications," on their proposed system for tele-immersion. Their goal was to combine multiple video feeds into a single panoramic image to be displayed on a screen and simulate reality as if looking through a wall. An example they used was for drivers of excavators whose field of vision is currently very limited by the surrounding framework of the vehicle. Their research will aid the operator by projecting his surroundings onto a panoramic "immersive display," or a concave screen. They use a total of eight cameras, going through two video splitters which combine four cameras into one feed, and then through two computers which feed into another central computer through an Ethernet connection. All of this connectivity is necessary because, at the time of publication, "a PC can handle only one input video stream without relying on some expensive specialized video capture boards." This is no longer the case, as most motherboards have 2 to 4 PCI Express slots, each of which can process an input stream, though the bandwidth is still limited. Our goal is to perform the same sort of research, while feeding all the HD cameras into a single computer. We believe that this is achievable with recent advances in hardware.

2.5. Evaluation of alternative solutions

Initially, an objective for this project was to find the ideal parameters for the Homographier and Stitcher modules. We eventually settled instead on building a user interface to let the user decide which parameters best suited their application. This was decided because it quickly became apparent that the "best" configuration is different depending on the scene. Also, there is a trade-off between most of the options between speed and stitching quality. The priority may change depending on the application.

A simple solution to the problem of providing a panoramic video would be to use one high-definition fish-eye lens camera. These give a wide viewing angle and, depending on the price, can provide very high resolution images. However, NASA asked us to explore stitching multiple videos together to create a panorama in our research, so we were constrained to using standard, webcam-like cameras.

OpenCV can be built with GPU support, which allows the user to specify in some cases that a function should be executed on the GPU rather than the CPU. This is true of several functions used in the Homographier to create a homography from two images. We decided not to take advantage of these GPU-enhanced functions after encountering issues in the GpuStitch module with competing for processing time on the GPU. If there were a third GPU which could be dedicated to the Homographiers, then the homographies could be calculated in series on the GPU and likely execute more quickly, but since typically a maximum of two graphics cards can be installed on any one computer, this solution is probably infeasible. We opted to leave the Homographier calculations all on the CPU, since the latency in the Homographier module is not as critical as it is in the Stitcher module anyway.

The GUI could be implemented using wxWidets or QT. WxWidgets was the desired choice due it being more malleable as far as making a more original GUI, but there was no stable version available yet for Visual Studio 2010.

3. Final design

3.1. System description

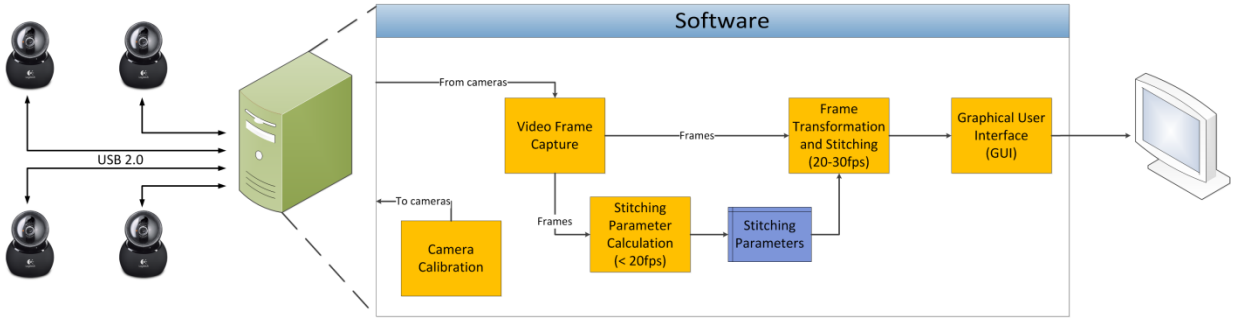

Figure 2: High-level system diagram

The above flow chart illustrates how our system works in general. The cameras are connected to our custom-built computer by USB 2.0. As the program runs, independent threads save frames from the cameras and store them in shared memory. Stitching parameter calculation threads, or "Homographiers" as we call them, calculate the perspective difference between each pair of adjacent frames. These calculated parameters, called homography matrices, are also stored in shared in memory. The stitching thread, running on the GPU in parallel with the other threads, retrieves the frames and parameters and creates a single combined image. The stitched images

are displayed in succession through a graphical user interface as often as possible, creating a video stream.

3.2. Complete module-wise specifications

3.2.1. Hardware

- Cameras:
  - Prototype Cameras: 2 x Playstation Eye
    - These were not used because of shipping delays
  - Final Cameras: 4 x Logitech Orbit AF USB 2.0 Cameras
- Computer Hardware:
  - LG Super-Multi DVD Burner 22X DVD+R CD-ROM Black SATA Model
  - ASUS Sabertooth X58 LGA 1366 SATA Intel Motherboard
  - 12GB (3 x 4GB) DDR3 SDRAM Desktop Memory
  - Crucial M4 2.5" 64GB SATA III MLC Internal Solid State Drive
  - EVGA GeForce GTX 560 2GB Video Card
  - Intel Core i7-950 Bloomfield 3.06GHz LGA 1366 Quad-Core Processor
  - Black Aluminum ATX Full Tower Computer Case
  - OCZ ZS Series 650W 80PLUS Bronze High Performance Power Supply

3.2.1.1. Cameras

The cameras we decided upon for the project are retail, off-the-shelf Logitech Orbit AF webcams. As we researched different camera alternatives, connection bandwidth, resolution, and cost were our major considerations. In the end, however, the SkyView team's requirements drove our final decision. The Orbit AF offers built-in pan/tilt motors and zoom functionality for the SkyView team, along with 720p HD video capture, so it emerged as the front-runner for our final camera.

One particular issue that we discussed at length was the camera connection bandwidth. Immediately after receiving our assignment, we expected that we would need a higher bandwidth connection than USB 2.0 could provide. Upon further research, we found that cameras with alternate data transfer protocols, such as Firewire, Gigabit Ethernet, or USB 3.0, were expensive and limited in selection. Given the tradeoffs involved, we determined that USB 2.0 cameras would suffice provided we could build a custom PC architecture to support them.

Later on, we realized some of the limitations of our cameras. These problems included autofocus issues, differences in light levels between cameras, and bandwidth issues (which we address in later sections). A particular problem with these cameras specifically is error in the pan/tilt functionality of the cameras. Once we started implementing a module to automatically align the cameras for stitching, we realized that the pan/tilt movements generated through the DirectShow API weren't always reliable or accurate. After several runs of the alignment module, some cameras didn't reset to a centered position. At times, we also found that the cameras wouldn't respond to commands to tilt lower or higher, even when the next camera over showed that it was possible to tilt to that level.

3.2.1.2. Custom PC Hardware

As mentioned above, custom computer architecture was needed to support 4 simultaneous HD video streams under USB. Through bandwidth calculation (and testing on the lab machines), we determined that only one of the Orbit AF camera feeds can be supported by each USB controller.

In other words, USB 2.0 can only support 1 camera per USB controller, of which there is typically one or two in a standard PC, regardless of the amount of USB ports.

To solve this bandwidth issue in our custom PC, we chose a motherboard chipset which has two built-in EHCI USB 2.0 controllers, one USB 3.0 controller, and several PCI Express 2.0 expansion slots. We outfitted one of these PCI Express slots with a dedicated USB card containing several of its own USB controllers.

Surprisingly, even with all of these bandwidth solutions in place, we still experienced some problems. Because our USB PCI card appeared to have a dedicated USB controller for each of its four ports, we tried hooking all four cameras into it at first. This somehow caused the entire computer to freeze up and crash! After that, we distributed the cameras over all the sets of USB ports on the computer, which solved the crashing problem. Still, the camera access speeds that we have measured don't seem as fast as we had initially hoped.

Because the processing of four HD video streams requires extensive computing power, our custom PC utilizes an Intel i7 Quad-Core processor and an Nvidia GeForce GTX 560 graphics card. OpenCV, a computer vision library we are using (and will discuss below), provides GPU-enhanced algorithms that allow us to use the parallel processing power of the GPU for much of our heavy processing. For other processes, we have the quad-core Intel i7 CPU, one of the most cutting-edge processors on the market today.

3.2.1.3. Display

Originally, we wanted to have a large, ultra-high definition monitor to display our stitched video. However, we have decided to just use the best monitor available to us since an outsize display would be too high an expense for our proof of concept. The entire HD output will still be accessible through the new GUI design explained in a later section.

3.2.2. Software

3.2.2.1. Camera Calibration

The camera calibration module is an auxiliary mode, accessed by the "View Cameras" button on the main menu, which allows the user to manually adjust the angles of each camera view. Adjustment is done with the built-in pan/tilt motors in each camera, accessed in the back-end code through Logitech-provided sample code and the DirectShow library of the Windows SDK. This allows the user to set their own level of overlap and make sure that the cameras are aligned correctly, which is particularly important. For example, for two cameras mounted to the left and right of each other, our system is designed to accommodate a wide range of overlap over the x axis. However, the way we stitch images is ill-equipped to deal with difference in the y position of two horizontally aligned cameras. Thus, when our program is first opened, the user should enter "View Cameras" mode and align the cameras before running the full stitching program.

When we presented our design at CDR, we had a more ambitious plan for the camera calibration unit than what we finally implemented. We then wanted to develop an automatic subroutine which would initialize the camera at startup of our program. As originally planned, this module would turn off automatic focus and lighting adjustment (aperture, etc.) and use the pan/tilt motors to move the cameras to a given overlap level. If possible, we also hoped to find a way to let the automatic adjustment run on one camera, and sync those camera settings to all the other cameras.

However, time and technical restraints prevented us from realizing this plan for the camera calibration module. Because other features were more pressing, we didn't have time to dig into the DirectShow documentation and find a way to control camera properties besides pan and tilt. Also, the previously mentioned issues with pan/tilt accuracy made an automatic alignment system unreliable. Given these restraints, a user-controlled camera calibration unit seemed like the best way to address the accuracy problems and still use the motors for convenient camera alignment.

3.2.2.2. Camera Capture Threads

Incoming to the software will be four concurrent video feeds. We want to capture an image from each of these feeds at the same instant. To achieve this, we will use separate synchronized threads, one for each of the cameras, which will take one frame from the cameras at the same time as each other, and store them into some location in memory. From here, both the stitching parameter calculation module and the stitching module can access the images.

Each CameraCapture object is a thread which owns a cv::VideoCapture object. The VideoCapture's *grab()* method tells the camera to store a new video frame, then the *retrieve()* method, which runs more slowly, decodes and returns the grabbed video frame. Using separate threads will ensure that the *grab()* calls execute at the same time, so that all the frames represent the scene at the same moment. And by running the retrieve methods in parallel, we ensure that we are utilizing the full bandwidth potential of the USB controllers.

3.2.2.3. Stitching Pipeline

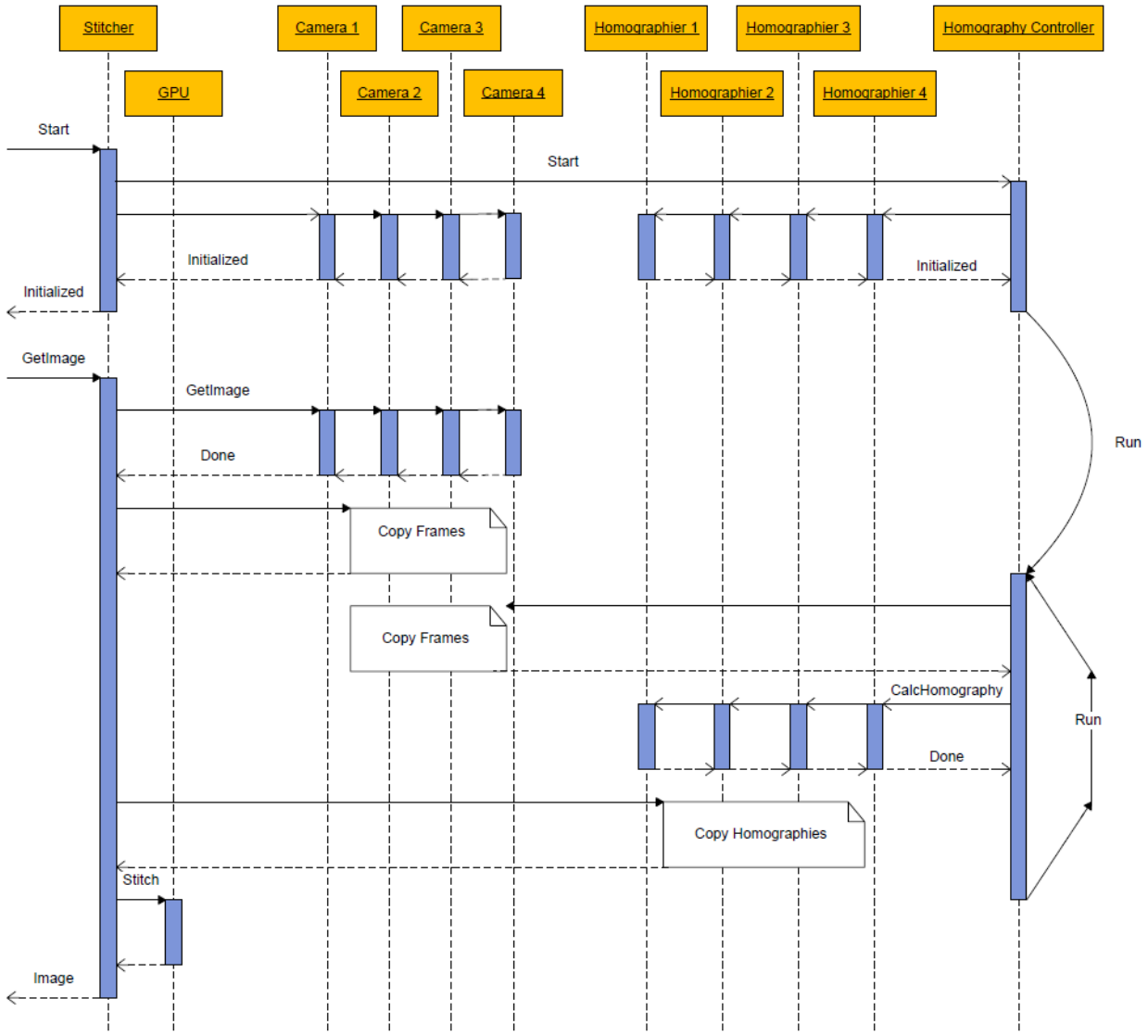

Figure 3: Stitching pipeline details

The bulk of our software design resides within the stitching pipeline. This consists of multiple modules which work together to analyze and stitch the videos together. These modules were designed to run as parallel threads, since many of the necessary calculations are easily parallelized. While running, our project takes full advantage of all six cores in our i7 processor. If this project were extended to more than four cameras, the six cores might actually be a significant limitation.

A more serious limitation of our implementation would be exposed if more cameras were added, however. Our design is built around the idea of perspective transformations between the images which can be represented with a homography matrix. This type of transformation represents a planar warp of the images. Choosing one of the images as a root image, all other images are projected onto the flat plane of the root image. This results in trapezoidally warped images stemming outwards from the root image. Examples of this can be seen in Appendix A. This type of transform works well with a few centrally focused images, as with the 2x2 camera array used

for this project. But, as more cameras are added, the overall viewing angle increases and the images far from the root image become more and more dramatically warped.

The reason for this is that the Stitcher is trying to depict a panorama described solely by camera rotations about a central point. If the camera were allowed to rotate fully, the only way this could be represented would be to stitch the images onto a spherical surface. If the cameras are constrained to a low viewing angle in both the vertical and horizontal direction about a central point, then the planar projection works well. If the cameras are constrained only in the vertical direction, as in a 4x1 array or an 8x2 array, for example, then a cylindrical projection may be sufficient. The cylindrical projection results in an image that is warped like the standard depiction of the world map, where countries at the north and south extremes of the map appear larger than those at the vertical center.

The only way to ensure that the projection is accurate for an arbitrary number of cameras and configurations is to use a spherical projection. This lets you display stitched video from as many rotated cameras as you could want. It also ensures that straight lines shared amongst the frames are preserved as continuous, curved lines. By comparison, in a planar warping straight lines are preserved in each image, and continuous between images, but they are bent at the intersection between the images. And cylindrical warping preserves straight lines in the vertical direction, but displays them as curved lines in the horizontal direction. Szeliski explains all of this in "Image Alignment and Stitching: A Tutorial" in Section 2: Motion Models, and explains how to choose between the different types of projections in Section 6.1: Choosing a Compositing Surface. In our design, we assume that a homography matrix is sufficient to specify the transformation of an image.

3.2.2.4. Data Flow between Modules

At all times, when our project is running with four cameras, there are ten concurrently executing threads. So, significant effort was spent ensuring that communication between the threads is well-defined and thread-safe. To do this, we took advantage of many core .NET structures: threads, sockets, events, and mutexes. The specific implementation of these structures is explained below, in the implementation notes. Here, only the flow of data between the modules will be explained.

The VideoStitcher class is the parent class. It owns each of the 9 background threads: the four CameraCapture threads, the four Homographier threads, and the HomographierController thread. When the VideoStitcher calls its *start()* method, it kicks off all nine of those threads, which continue executing until the VideoStitcher tells them to terminate in its *stop()* method.

As soon as the VideoStitcher starts, the HomographierController starts to request parameters from each of the Homographier objects. The HomographierController serves to maintain synchronization between the independent Homographier threads. Each time the four Homographiers finish calculating new parameters, the controller tells them each to start again on a new set of images. This controller operates entirely separately from the main thread, calculating new parameters as often as it can.

Meanwhile, whenever a request from the main GUI thread comes in for a new stitched image, the VideoStitcher first tells each of the CameraCapture objects to retrieve a new image, in parallel. Then, the homographies are copied from the Homographier objects. Then, the four images and the three homographies are passed on to the GPU for stitching.

Here, I must explain how the homographies for each frame are actually calculated. Each Homographier takes a pair of images, frame A and frame B, and calculates the transformation of frame B onto frame A. So, in a (2,2) camera array, the following four projections are calculated:

- Frame(1,2) onto frame(1,1)
- Frame(2,1) onto frame(1,1)
- Frame(2,2) onto frame(1,2)
- Frame(2,2) onto frame(2,1)

Then, from these four homographies, parameters for each of the four cameras can be calculated. The root image, frame(1,1) simply uses the 3x3 identity matrix as its transformation, since it does not need to be transformed at all. Frames (1,2) and (2,1) use the homographies from the first and second Homographiers. The last frame, (2,2) is a little more complicated. There is no directly calculated transformation onto the plane of the root image, so what we do is take the average of the projections onto frame (1,1) through frames (1,2) and (2,1).

Since these homographies are matrices, you can use matrix multiplication to chain projections. The resulting homography for frame (2,2) is:

$$\frac{Homography1 * Homography3 + Homography2 * Homography4}{2}$$

So, these are the four homographies that are used in the GPU to stitch the four frames together. When extending to more cameras, this will need to become more sophisticated. You could either choose to analyze the transformation between all pairs of images and determine programmatically which cameras have an overlap region (requires n! Homographiers), or you could continue to specify manually which frames are adjacent by pointing each Homographier to two specific frames. We suggest implementing an SSSP algorithm to find the chain of homographies for each frame, or some other mechanism which allows the user to change the Homographier targets at runtime without having to recompile the VideoStitcher module to specify the chains manually.

Currently, these targets are specified in a configuration file, the same file that saves all the configuration changes made in the GUI in between subsequent program executions. This is misleading, however, because if you change the targets in the configuration file, the VideoStitcher will still try to use the results of each Homographier in the fashion specified above. This has been labeled as future work: to extend the VideoStitcher to work with arbitrarily specified Homographier targets.

Finally, when the GPU has taken these frames and homographies and stitched them according to the given GUI parameters, this stitched frame is sent back to the GUI for display.

3.2.2.5. Homographier

To make all four frames share the same perspective, our software must calculate the perspective difference between frames. This is done in separate parameter calculation threads, one for each neighboring pair of cameras. In general, each thread starts by finding mathematically interesting features, or keypoints, in both frames. Next, the thread compares the two sets of keypoints and tries to find keypoints that have a match in the other image. After finding the set of common features between frames, the thread can use that data to calculate a homography matrix, a 3x3 set

of values that specifies the perspective difference between frames. This homography matrix is then saved to shared memory, where it is used by the Stitcher module.

3.2.2.6. Feature Detection

For feature detection, our software uses the OpenCV version of the SURF (Speeded-Up Robust Features) algorithm. Given an input frame, SURF finds interesting features in a frame using a specific math-based metric. These features are output as two sets of data: a vector of keypoints and a vector of descriptors.

There are several parameters for the SURF algorithm, and our software allows the user to change them dynamically, as it does elsewhere in the code. Two particularly important parameters are the hessian threshold and the frame overlap.

In general, the hessian threshold determines how many keypoints are found by SURF. Raising the hessian threshold reduces the number of keypoints, while lowering the threshold will increase the number of keypoints. The hessian threshold also affects system performance: lowering the threshold tends to slow down the system, and vice versa.

The frame overlap percentage specifies how much of each frame overlaps with the neighboring frame. This percentage is used to create a mask which tells SURF to only calculate keypoints within the overlapping region and ignore the parts of the frame which aren't shared. This is a useful function because it saves calculation time from the non-overlapping regions and reduces the amount of false positive matches in later stages.

A few additional notes on frame overlap: Since the real amount of overlap is set by the user through the pan/tilt motors, this value can be changed to any value the user prefers. Also, it should be noted that this value sets the frame overlap on both the horizontal axis and the vertical axis. Since these values will most likely differ in reality, future work might include separating this parameter into vertical frame overlap and horizontal frame overlap.

3.2.2.7. Feature Matching

Next, the Homographier module tries to match features that can be found in both frames using a FLANN-based (Fast Library for Approximate Nearest Neighbors) matcher. This OpenCV algorithm takes as input the descriptor set from each frame and searches for mathematically similar features with one of several search methods. OpenCV supplies several ways to use FLANN, and we selected a few of the most interesting to expose to the user through our UI.

The first of these search methods is the brute-force, or linear, search. This search method doesn't build a special data structure to aid with faster search; it just uses a straightforward brute-force approach.

The next method is a KD-Tree search. This is the default OpenCV option when creating an OpenCV FlannBasedMatcher and it involves building randomized kd-trees to be searched in parallel.

Last is the Autotuned index build option. When Autotuned is selected, the FLANN algorithm automatically creates a linear index, kd-trees, or a k-means tree based on desired user parameters like accuracy, build time vs. search time, and speed vs. memory. While such a level of customization seems like a great option, we found that this option ran the slowest of all three FLANN index types.

3.2.2.8. Homography Calculation

Finally, after receiving a set of matched keypoints, our code uses the OpenCV *findHomography()* method to find the perspective difference between frames. To increase the accuracy of this calculation, we attempt to filter out false positive matches with two parameters: the match distance tolerance and the RANSAC threshold. We also attempt to smooth out differences in sequential sets of homographies by using a time average.

During the feature matching operation, one of the parameters saved by the matching algorithm is a distance value for each set of matched features. This value specifies the distance between the (x, y) position of the feature in one frame and the (x, y) position of the corresponding feature in the other frame. In a completely accurate system, all matches between a pair of frames should share roughly the same distance (unless perhaps there is a perspective difference between foreground objects and the background). In reality, however, the matcher returns several false positive matches. Changing the match distance tolerance to less than 100% screens out some of these false positives on the basis of variance from the mean distance. For example, when the match distance tolerance is set to 60%, our code calculates the average distance and removes outliers outside the middle 60%, centered at the average.

The RANSAC threshold is another way to reduce error in the homography calculation. This parameter is used in *findHomography()* to filter outliers based the re-projection error from an initial calculation of the homography matrix. Lowering the threshold lowers the maximum allowed error for an inlier, so a lower threshold theoretically leads to a more accurate calculation. A higher threshold allows more points to be considered as inliers.

Finally, before saving the newly calculated homography matrix to shared memory, we combine it with the previous homography using a weighted average. Despite efforts to reduce error, our homography calculations typically vary significantly, which caused the output image to shift erratically before we implemented a time average. The time average weight refers to the percentage of the overall average which is determined by the new homography. We have found that setting the time average weight to 1-10% produces the smoothest results. Setting the weight this low means that the output takes longer to adjust to scene changes, but we find slower updating preferable to a jumpy output.

3.2.2.9. Stitching Module

The stitching module resides in a separate .dll project, called GpuStitch. While a CPU implementation exists, it lacks many of the useful features in the GPU stitching module and runs very slowly. This was only developed as an initial proof of concept and was not intended for production use.

The GPU module takes as input a vector of images (cv::Mat's) and a vector of homographies (also cv::Mat's), one for each image. The output image is twice the size of the root frame, which the stitching module expects to be the first in the vector of input images. This canvas size was chosen because it typically gives a large enough output image to display the full content of each of the four images. We could have chosen to output the entire content of the stitched image, but since the homographies change every second or so, the output frame size would also change about once a second. Since we want to be able to scroll through a portion of the large image in the GUI, we chose to fix a constant output image size, at the cost of occasionally losing some pixel information at the edges of the stitched image.

We desired to center the stitched image in the output frame, and this proved to be a fairly interesting technical challenge. We needed to shift each of the frames to the center of the output image beforehand by adjusting the homographies. What we ended up doing was calculating the maximum and minimum values in both the x and y directions by transforming the four corner points of each of the four input frames through their inverse homographies. The reason we used the inverse of these homographies is that normally the matrices are used to transform points in the plane of the root image onto the plane of the input image. We needed to do the opposite: transform points on the input image onto points in the projection plane to find out how much space each image would map out on the output image. Once we had all of these maximum and minimum values, a translational homography matrix was created, explained in Szeliski's article in section 2.1. Then, each of the homographies was multiplied by this translation matrix, shifting each of the four images by the same amount.

The GPU was utilized by breaking up the output image into blocks of 32x16 pixels. Without going into the details of how CUDA splits up GPU SIMD executions into grids and blocks, it will suffice to say that each pixel in the output image gets its own thread for calculating its pixel information. Each thread transforms its point using each of the homographies into each of the other frames, then the resulting pixel is calculated according to the blending and interpolation details explained below.

3.2.2.10. Bilinear Interpolation

When using a homography to warp an image, what actually happens mathematically is that you use the 3x3 matrix to transform a coordinate in one plane into a coordinate in a different plane. The GPU assumes that each of the homographies that it has received is already constructed to project a frame onto the plane of the root image. In our design, this is the plane of the first image, or the "root" image. All other frames are transformed to be drawn onto the plane of the first image.

When the points are transformed, the result is a point represented with two floating point numbers. Here the user has a choice about which pixel to select as the value for this frame at that particular point in the projection plane. The resulting point can either be rounded to the nearest pixel, in which case the BGR value at that pixel is returned, or bilinear interpolation can be used to calculate a more accurate color value for that point by taking the weighted average of the four surrounding pixels.

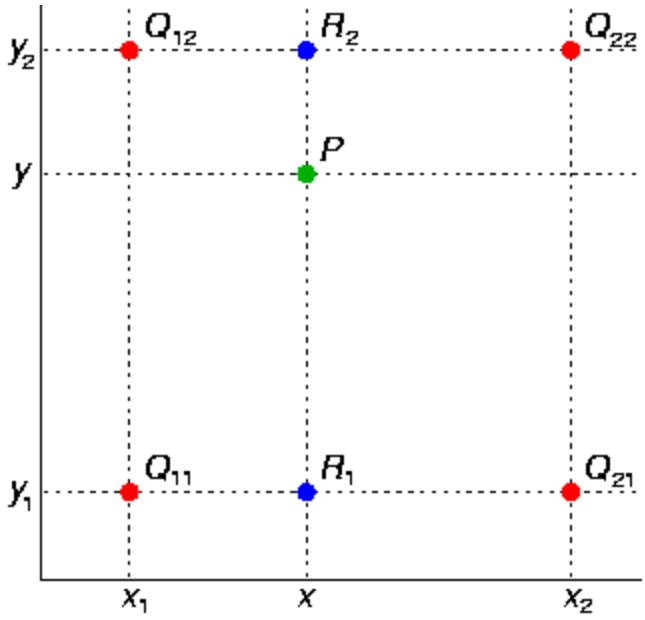

Figure 4: Bilinear interpolation

Since each floating point x and y pair lands in some region in between four points, the resulting pixel value should be weighted more heavily towards whichever neighboring point is closer to the desired point. In particular,

$$P = Q_{11}(1-x)(1-y) + Q_{21}x(1-y) + Q_{12}(1-x)y + Q_{22}xy$$

In theory, this should result in a more accurate transformation of the image, while it may result in a more blurry image, due to each pixel being calculated as an average. In practice, however, this setting doesn't make any visible difference. It turns out that for reasonably high camera resolutions the difference between using rounding and using interpolation is nearly indistinguishable to the human eye. But, this is a very inexpensive operation, and it has been left in the stitch kernel so that the GUI could support digital zoom in the future. In that case, the difference would likely be noticeable.

3.2.2.11. Image Blending

When two or more images are stitched together, there is always an area in each image that overlaps with another. Deciding the best mechanism for blending the images is not simple. The goal is a seamless stitch, where someone cannot identify the delineation region between the two images, and the entire image looks like a seamless panoramic taken with one wide-angle camera.

Since our Stitcher runs in a CUDA kernel, each thread needs to be able to calculate the BGR (blue, green, red) value at a specific pixel independently from other threads, while the blending needs to be smooth between neighboring pixels. There are many different techniques for blending images in a panorama, several of which are enumerated in Szeliski's paper. Some of these techniques are helpful in our case, and some are not. We need blending techniques which are simple enough to be calculated in real-time, and which can be calculated independently yet uniformly for each pixel.

The options available through the GPU Stitcher module are: overlapped, average, linear and exponential blending. The details for each are outlined in the appendix below. We suggest using half-power exponential blending for most scenes.

3.2.2.12. Image Tinting

One feature in the GPU stitching module which is very useful is the ability to tint the images. What started as a simple debugging tool turned into a viable feature for explaining how the stitching parameters function. We support two types of tinting. First, the user can adjust a tinting slider which tints each of the three non-root images in either the red, green or blue directions. This can help to make the stitching regions more apparent between the images by delineating the images from one another. This tinting is done in the Stitcher module after all of the image processing has already been done by the Homographiers, so the stitching does not affect the homographies.

The other option is to do a "Maximum Tint." What happens here is that all of the actual pixel information in each image is thrown away and each image is simply assigned a color. This is most helpful for analyzing the strengths of the different blending techniques, as explained in the appendix below.

3.2.2.13. Graphical User Interface

Our final stitched video stream will be viewed through a custom user interface. In our original proposal, we hoped to purchase an oversized monitor capable of displaying an image at roughly two times the resolution of 720p. However, we later decided to cut the oversized monitor from our budget and use a standard monitor. Because most monitors can't display an image of the size we are creating, we considered reducing the resolution of the output stream. Ultimately though, we felt this defeated the purpose of using HD input and decided against it. Instead, we plan to develop an interface for users to see part of our stitched feed in full resolution, while still proving we can process all four cameras at once. In general, we plan for the user interface to display one section of the feed at full resolution, while allowing the user to scroll to other sections of the feed.

For most of the project, we were trying to perfect setting of each parameter to produce the most desired output. However, after messing with the parameters for a while, we discovered that we basically had to choose between fast output speeds (higher frames/sec) or better stitching quality. So instead of making the decision on which to sacrifice for the other, we decided to leave the parameter controls open to the user. As a result, in the GUI the parameter controls are all listed for the user to change as needed. In addition to this, the user stop and start the stitching at will, hit a record button to record sections of the outputted video, and decide whether to see the entire output or a small section with the capability to scroll through the image.

3.3. Approach for design validation

Our main goal for this project was to provide the user with enough tools to understand the limitations and available options for stitching videos together. We have put great effort into building a user interface which allows the user to explore as many aspects as possible of the available options, while providing meaningful graphical and timing results for instant feedback. However, this has not been truly validated since only the developers have been operating the program: we didn't spend any time conducting usability tests. Hopefully the technically competent people at NASA will be able to interpret our GUI well enough for it to be useful.

Initially, our approach to validation for this project was to make sure that we could display full HD stitched video (four 720p streams) at 25-30 frames per second. This was infeasible for several reasons. We were never able to even display one unprocessed 720p stream at 25 fps, much less stitch four streams at that speed. This is mostly due to USB issues. Streaming uncompressed HD video is extremely taxing on a USB connection. We could use cameras which compress the video before transmitting it, but they we would have to decompress it before processing it. The best solution would be to find cameras which use a different protocol, like USB 3.0, FireWire, or Thunderbolt.

Physical metrics are still helpful for our project, however. Our socketed timing module provides detailed, accurate timing information about each module, and lets us make some exact claims about the speed of each section of our program. However, the live-updated latencies for both the VideoStitcher module and the HomographierController module displayed on the main output window are of more practical use. These two metrics tell the user almost everything they will be interested to know about timing. When they make a change in the Homographier options, or start up the program with a different frame resolution, the change in the latency is immediately apparent

The other tool for design validation is also immediately available, but less objective. We also want to give the user the ability to evaluate the stitching quality of the output video stream. We tried to maximize the usefulness of the output video by centering the image and by providing mechanisms for tinting the different images. Again, this requirement is more difficult to validate than the timing requirements, but we believe we have given the user all the tools necessary to determine whether the stitching quality they are getting is satisfactory.

4. Implementation notes

4.1. Software Architecture

The StitcHD solution is composed of three projects: GUI, StitcHD and GpuStitch. The GUI project is a .exe project, it is the program entry point containing the *main()* function. The GUI project is dependent on Qt and OpenCV; it is primarily responsible for providing a graphical front for the StitcHD project.

The StitcHD project is nearly a stand-alone project. We had compiled it as a .exe project for most of the development cycle with a command-line interface for changing the configuration. Since adding the GUI portion of our project, several key features have been added that would need to be added to the StitcHD project to get the command-line version fully functional. Now it exists as a .dll project which exports several crucial classes needed by the GUI project, most notably the VideoStitcher class, which controls the stitching pipeline.

And finally, the GpuStitch project is another .dll project which exports a single function to the StitcHD project. The function is *stitch_gpu()*, and it is called in the ImageStitcher class within the StitcHD project.

The code was broken up in this way for several reasons. For one, the project takes a very long time to compile from scratch. By breaking up the code into various projects, you can minimize the compile time when making a small change to one part of the code. Also, we wanted to separate the code by their dependencies. Only the GpuStitch project depends on the CUDA SDK, and only the GUI project depends on the Qt SDK. The idea was that the project could be compiled without GPU support if so desired, and also without Qt support if so desired.

4.2. Software Implementation

4.2.1. MainWindow (GUI)

The program starts in the GUI project's main.cpp. It simply starts up a MainWindow, which inherits from a QMainWindow. This window lets you make changes to some of the most basic parameters, like the screen resolution and the number of cameras. These configuration settings matter for both of the available options from this window: "Show Cameras" and "Run".

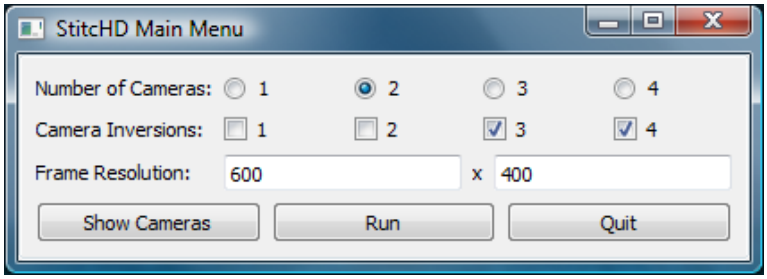

Figure 5: GUI MainWindow

This MainWindow owns a Config object as a member of its class. Config is a class declared in the StitcHD project which stores all of the configuration options for the stitching. This Config object is read from a file (unless the file does not exist, in which case default values are assigned) when the MainWindow is created, and written back to file when the window is destroyed. A reference to this object is passed to each of its child windows, so that they can make changes to the configuration and have them written to the file when the program exits.

4.2.2. DisplayCameras (GUI)

Choosing "Show Cameras" starts up a DisplayCameraController, which is a QWidget defined in DisplayCameras.h. This widget starts up several DisplayCameraFrames, also defined in DisplayCameras.h, one for each camera specified in the MainWindow. Each of these DisplayCameraFrames is a QMainWindow which owns a CameraCapture object from the StitcHD module for retrieving frames from the camera and a DirectShow IAMCameraControl object for sending pan and tilt commands to the camera. These windows have several buttons for adjusting the camera's pan and tilt, and also a button for resetting a camera: making it point straight ahead. In each of these buttons' callbacks, the IAMCameraControl object is needed to send the necessary commands.

Figure 6: GUI DisplayCameraFrame

These windows are linked together such that if you close one, then all of them will close and take you back to the MainWindow. This is done by connecting each window's *destroyed* signal to the *closed* slot of the DisplayCameraController. Signals and slots are Qt's mechanism for sending synchronous signals between different modules. In order for the *destroyed* signal to be emitted when the window closes, a specific attribute must be specified for that object, Qt::QA_DeleteOnClose. This forces the window to be destroyed when you hit the "X," rather than simply hiding the widget and waiting for the program to exit before destroying it. This is a crucial addition in several places, since many of the objects in the StitcHD module depend on the destructor to end threads and clean up memory.

The updating of the frames in the DisplayCameraFrame windows happens by looping through the windows in the DisplayCameraController. Each time a window finishes updating, the controller posts a new event for the next window to update whenever possible. The reason for using events here is so that other GUI changes like dragging or closing windows can occur. If the event loop is not allowed to update other GUI widgets, the entire application appears to freeze.

Since the updating of these windows happens in series, this functionality may not give an accurate idea of the frame-rate when viewing multiple cameras. Doing this in parallel would be ideal, but Qt complains when you try to display an image using a background thread. This is because Qt uses the GPU to display images, and it doesn't like for background threads to compete for the GPU.

4.2.3. SettingsWindow (GUI)

The settings window represents a large amount of GUI code. The purpose of this screen is to adjust all the parameters in the Config object that are specific to the actual stitching of images which aren't changed on the MainWindow. The file SettingsWindow.cpp alone is nearly 1000 lines of code, just to deal with creating the layouts and widgets for the GUI, and implementing all of their callbacks. Each checkbox, radio button and slider on the SettingsWindow corresponds to some value in the Config object that is passed to the SettingsWindow by the MainWindow.

While a large amount of effort was put into designing this window to make it clear visually how the settings are separated, the window is still overwhelming at first glance. So, for each widget on this window, we have also added a tool-tip which appears when the user hovers over it. One way to improve this window visually would be to increase the visual separation between the different settings groups. The Stitcher settings are separated from the Homographier ("Parameter Calculation") settings, but only subtly.

At the bottom are three buttons. The "Set Defaults" button resets all of the Config values to their defaults, which are specified in Config.cpp, not in the SettingsWindow module. This function was frustratingly difficult to implement, because when you change the value of a slider, you still have to invoke the callback for moving the slider manually. This is not true for the other widgets on the page, like the radio buttons or spin-boxes.

The "Save Frame" button saves the current image being displayed to the hard-drive, in the Pictures/ folder, sorted by date and time. Similarly, the "Record" button records the video being displayed. When you click "Record," the text on this button changes to "Stop Recording" to let you stop recording the video whenever you like. Behind the scenes, the video recording actually happens in the VideoStitcher module. When you toggle this button, it actually sends an event to the DisplayStitcHD window telling it to toggle recording mode.

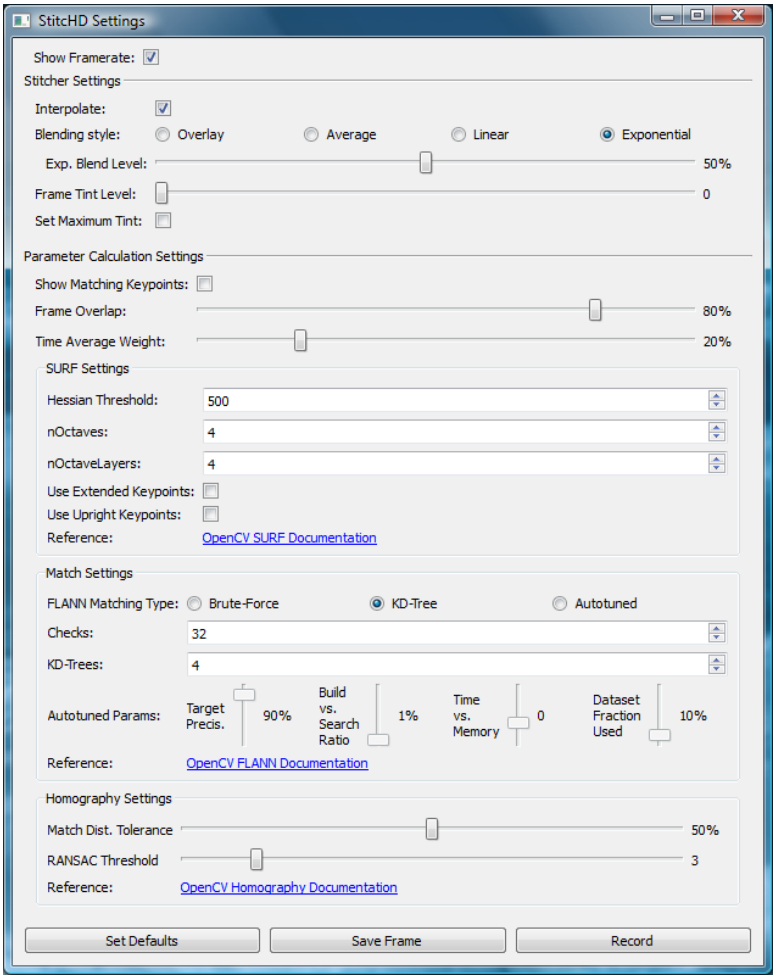

Figure 7: GUI SettingsWindow

4.2.4. DisplayStitcHD (GUI)

This final GUI window is the one which actually displays the stitched video. The widget layout is fairly simple, just a QLabel to assign the image to, wrapped within a QScrollArea, with an additional two QLabels at the top which are used to display the latency of the VideoStitcher and HomographierController modules.

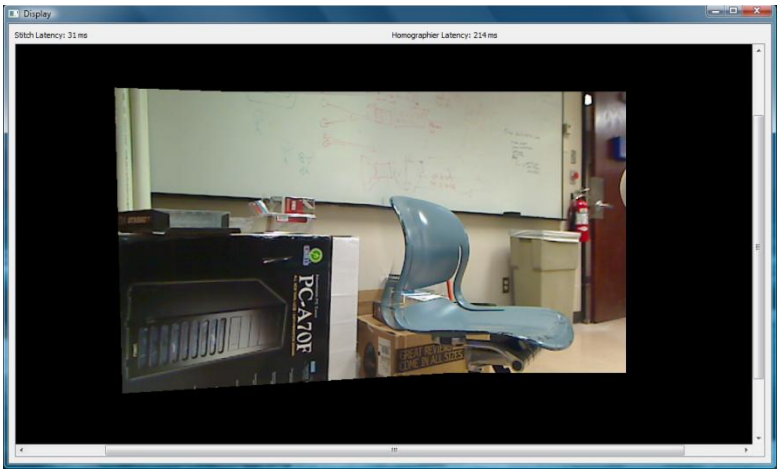

Figure 8: GUI DisplayStitcHD

Behind the scenes, this window owns a VideoStitcher object, which is initialized when the window is created and deleted when the window is destroyed. This window also gets a reference to the same Config object being modified by the SettingsWindow so that the VideoStitcher uses an updated copy of the Config object each time it stitches a new image or calculates some new homographies.

Like the DisplayCameraFrame windows, this window is linked with the SettingsWindow so that when either of the two windows is deleted, the other one gets destroyed as well. And again, the updating of these images is done by chaining events to Qt's event loop. Once a frame is calculated and displayed, a new event for getting a new frame is pushed to the event loop to be processed once the other GUI events have been processed.

4.2.5. VideoStitcher (StitcHD)

Now we will walk through the modules in the StitcHD project, starting with the VideoStitcher class. This class is the parent class for all of the threads running in the StitcHD project. When you call the *start()* method, each of these threads gets kicked off. While the VideoStitcher is running, you can call *getImage()* and the VideoStitcher will put together the next stitched video frame for you. First, it will tell each of the CameraCapture modules to retrieve a new frame from its camera, then it will copy the homographies from each of the Homographiers, and finally, it will calculate a stitched image using a function in the ImageStitcher class and store the frame in its *displayFrame* member object.

4.2.6. Timer (StitcHD)

The first thread which is started by the VideoStitcher is the Timer thread. This thread exists to record timing information from all of the other modules in the StitcHD project. We wanted to design a timing mechanism that required very little overhead cost, which was simple to add in the code in the other modules, and which was thread-safe. We could have chosen to use a

multithreaded pipe, but we chose instead to use a socket. In each of the timed modules, all that is necessary to send a new piece of timing data is a simple call to a static method in the Timer class, *send()*.

```
324
325     Timer::send(Timer::Stitch, 0, Timer::StitchTimeval::Start);
326
327     // Do stitching...
328
329     Timer::send(Timer::Stitch, 0, Timer::StitchTimeval::End);
330
```

This method sends a UDP packet to a local socket containing four numbers. The first three are those seen above, an enumeration for specifying the module, an identifier (since there are multiple CameraCapture and Homographier modules), and another enumeration specifying which step in the process has been reached. The fourth number is the time value when the packet was sent. This is calculated before the packet is sent rather than after it is received to ensure that the time value is accurate even if many packets have stacked up on the socket, though in practice this shouldn't ever make a difference.

The Timer thread then is in charge of listening for incoming UDP packets, receiving them, turning the strings into integers, and updating the vectors of timing data as necessary. A vector is stored for each of the modules being timed, and each element in the vector contains a vector of time values, one for each of the steps in the process. For example, for each Homographier, a vector is stored, and each element in that vector contains a vector of timing values for *Start*, *Detect*, *Match*, and *End*.

When the Timer is done, you can print the results out to a file. This file contains some information at the top about what the Config looks like for this execution of the program, though if the Config changed in the middle of the execution, those changes aren't reflected in the output file. Then, a long table of numbers follows, in several sections, one for the CameraCapture values, one for the Homographiers, and one for the Stitcher. These numbers can be easily copied into Excel or some other graphing program for analysis.

4.2.7. CameraCapture (StitcHD)

The next set of threads which the VideoStitcher is responsible for are the CameraCapture threads. Each of these serves as a wrapper for a cv::VideoCapture object that represents a camera. When you *start()* a CameraCapture object, it opens and initializes a camera, which can take some time. Alternately, if no cameras exist, they can also open some stock video that has been pre-recorded. This comes in handy sometimes when testing the project on alternate computers.

The mechanism for keeping these threads synchronized is using three manual-reset events. Two of these events are created and owned by the VideoCapture object, the *startEvent* and *stopEvent*. The *startEvent* signals all of the CameraCapture threads to get a new frame. Then, each CameraCapture object has its own *doneEvent* which is set when the camera has finished grabbing a new frame. Once all of these *doneEvents* have been set, the VideoStitcher sets the *stopEvent* which simply tells each of the CameraCapture events to start waiting for the *startEvent* again. Both the *startEvent* and the *stopEvent* are necessary to avoid a race condition where a

CameraCapture finishes grabbing a frame before the *startEvent* is reset. This event cannot be an auto-reset event because multiple threads need to query it.

4.2.8. HomographierController (StitcHD)

This module title is a little misleading because it does not actually represent a C++ class but rather a set of methods in the VideoStitcher. The *runHmgController()* function happens in a separate thread that also gets kicked off by the main VideoStitcher thread in *start()*. This thread is only responsible for facilitating synchronization between the Homographiers. The VideoStitcher first starts all of the Homographier objects, then starts the HomographierController thread to monitor them.

Similar to the CameraCapture objects, the Homographiers are controlled with several events, a *startHmgEvent*, a *stopHmgEvent*, and several *doneEvents* in the Homographiers. The *startHmgEvent* is set by the HomographierController signifying the Homographiers should all begin calculating new parameters for the most recent frames. Before this event is set, the frames are copied to the Homographiers so that they don't try to read the frames while they are being overwritten in another thread by a CameraCapture object. After setting the *startHmgEvent*, the controller waits for all of the *doneEvents* to be set, at which point the *stopHmgEvent* is set and the Homographiers start to wait for the *startHmgEvent* once again, ad infinitum until the program terminates.

4.2.9. Homographier (StitcHD)

This module is where most of the interesting computer vision algorithms come into play. There are three basic components of the Homographiers: detecting keypoints, matching keypoints, and calculating a homography from the keypoints.

First we use OpenCV's implementation of the SURF algorithm to calculate keypoints. We transform the two images into grayscale cv::Mat's so that SURF can analyze them, and we create masks for each of the images which, according to the frame overlap configuration parameter, marks out a region of the image where SURF will search for keypoints. Once these preliminary steps are complete, keypoints and descriptors are calculated for each of the two images.

Next, the descriptors are matched together using a FlannBasedMatcher. Based on options specified in the SettingsWindow, the matcher is built as a brute-force matcher, a kd-tree matcher, or an Autotuned matcher. The matcher processes the descriptors, and outputs a vector of matches.

These matches still have to be pre-processed a bit before calculating a homography. Based on the match tolerance value specified through the GUI, points are filtered out according to their distance from one another, with those furthest from the average distance being discarded. From the remaining matches, a vector of good matches is created, which the *cv::findHomography()* method uses to calculate the homography.

Once the homography is calculated, it gets averaged with the previous homography value back in the *run()* method which invoked the *findHomography()* method initially. This is done to create a smoother blending between the images. Without this averaging factor, the homographies vary enough between subsequent iterations to make the output video look jumpy.

4.2.10. ImageStitcher (StitcHD)

The ImageStitcher is responsible for actually stitching images together. This class consists of a bunch of public static methods that can be called from anywhere, though they are only called from within the VideoStitcher's *getImage()* method. When the GPU is disabled, this module can stitch two or four images together using the CPU to calculate each pixel sequentially. This is slow, but fast enough to prove the concept with a decent CPU. The code to stitch using the CPU was mostly copied from source code found online.

When the GPU is enabled, the method used is *stitchImages_GPU()*. This method calculates the homography to be used for each of the input frames, and then passes the images and their homographies on to the GpuStitcher module for stitching. The way that these homographies are calculated is explained above in the design section. Before invoking the GpuStitch method, a StitchParams object must also be created, copying several stitcher-related values from the Config object into the StitchParams object. This object is used in the GPU kernel to specify the stitching parameters.

4.2.11. stitch_gpu (GpuStitch)

The GpuStitch .dll project only exports one method, *stitch_gpu()*, but has a handful of internal methods used to perform the stitch. For this explanation, you will need to have a basic understanding of Nvidia's device architecture and how CUDA code is structured. If you already understand this, skip over the next paragraph.

Nvidia GPUs have several multiprocessors (8 in the GTX 560), each of which have hundreds of threads (up to 1024 for a compute architecture of 2.1 chip like the GTX 560). Each these multiprocessors runs 32 threads at a time in parallel (called a *warp*), using the single-instruction, multiple-data (SIMD) paradigm. So, for 512 threads, there are 16 *warps* in each *block*. Each multiprocessor executes one *block* at a time, where a block is a part of a *grid*. A *grid* is an array of *blocks* which all run the same *kernel* of GPU code. With 8 multiprocessors, 8 *blocks* in the *grid* are computed at a time, until the entire *grid* has been computed. So, if we had 300 *blocks* in a *grid*, then 38 waves of 8 *blocks* would run sequentially. GPU code consists of *global* functions and *device* functions. A *global* function is an entry point to the GPU, it represents a kernel of code executed by all *blocks* in a *grid*. The *global* function may call several *device* functions, each of which is compiled into the invoking *global* function as inline functions to create one kernel program. The number of registers required by this kernel restricts the number of threads which can run in each block, since each multiprocessor only has 32K registers, which run out pretty quickly when you have 1K threads.

Our kernel is restricted to using 512 threads, so our *block* size is 32x16 threads. Each thread represents a different pixel, so a *block* represents a rectangular portion of the output image, and the *grid* represents the entire image, an array of *blocks* covering all of the pixels. The GPU entry point is the *host* function, *stitch_kernel()*, which calls several *device* functions such as *addFrameToPixel()* and *applyHomographyToPoint()*.

The choice for the output frame size and the corresponding translation of the images to center them in the output frame is explained above in the design section.

What happens in the kernel itself is that each thread calculates its own pixel index based on its *block* and *grid* indices. Then, for each frame, that point is transformed by the frame's homography into a point on the plane of the frame. If the point lies within the boundaries of the image, that pixel value is returned, along with a multiplier for the frame at that point. This

multiplier is calculated according to the StitchParams, as explained in the appendix, and the resulting pixel value may be adjusted according to the specified tinting options.

4.3. Hardware Implementation

4.3.1. Camera Connections

Our original fear with the USB cameras was that there would not be enough bandwidth to stream all four simultaneously. We realized there has to be a separate USB controller for each camera. We attempted to fix this using a computer expansion bus called PCI Express, but the server would not boot with the PCIe card installed. Therefore, the hardware being used is required to already have four separate USB controllers, and it must be ensured they are all plugged in accordingly.

4.3.2. Mounting

A fixed position for our cameras was required to keep our stitching as consistent as possible. In order to keep the eyes of the cameras as close as possible, we mounted two upside down directly above the other two as seen below. The best stitching output will occur if the eyes are as close together and equidistant as possible.

Figure 9: Camera Mounting

5. Experimental results

They say a picture is worth a thousand words, so a two minute video at 20 fps is worth 2.4 million words. Especially for our video stitching application, the results are most appropriately presented as a video of the application running. So, the example videos attached with this report are the most helpful for understanding what we were able to accomplish. Together, they clearly display all of the available stitching options, and most of the tests we performed to analyze the stitching quality of our project.

The two videos were chosen to display different strengths and weaknesses of our project. The video overlooking the intersection displays a scene for which our project performs well. Since all of the visible objects are far from the cameras, there is virtually no parallax error between the cameras, and the perspective transformation is well modeled by the rotational camera paradigm that is assumed by our project. Once the Homographiers reach a steady state, their homographies are very accurate transformations between the four images. Since the majority of the scene is motionless, the homographies are also very similar between subsequent frames, which adds to the stability of the stitching. This video also does a good job of displaying the different blending techniques and makes a powerful case for exponential blending over the other available options.

The second video does a better job of displaying the weaknesses of our program. This video shows the lobby of the Texas A&M library, with students walking through in the background. Students far from the cameras can walk through the stitching seams with virtually no distortion, but those close to the cameras have portions of their bodies cropped out where they pass through the stitching seams. This is due to parallax errors between the cameras that could only be corrected by computationally expensive 3D analysis or by mounting the cameras even more closely together.

5.1. Timing Data

Our timing module let us record timing information for every time the program gets executed, so we have much more data than we can process. Some of the data has been represented here in graphical form so that you can gain some insight into the kind of numbers we are dealing with for runtimes in the various modules.

5.1.1. Frame Retrieval

This first graph represents the time needed to retrieve a frame from the cameras dependent on the resolution of the frame requested. We would expect the time required to retrieve an image to increase with the size of the frame, since more data needs to be transferred.

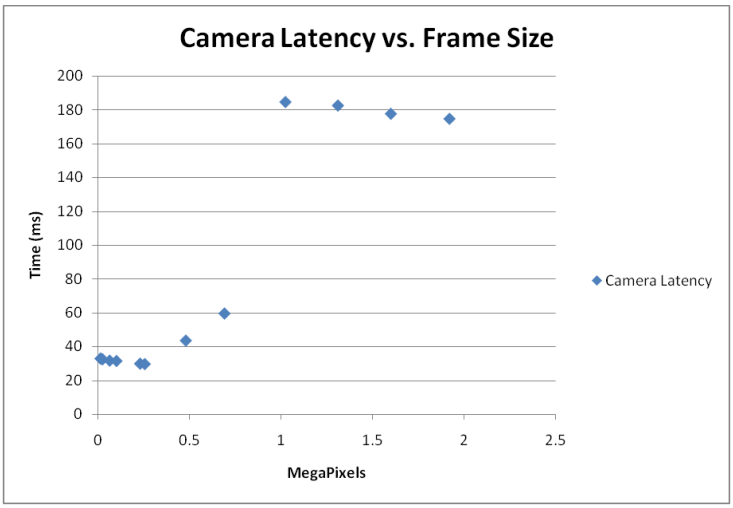

Figure 10: Camera Latency vs. Frame Size

This is one of our most puzzling results. For low camera resolutions (< 1 Mpixel), the results match our hypothesis, but then for greater resolutions, the latency increases dramatically and stays somewhat constant at about 180 ms after that. We observed strange behavior in the USB

cameras all semester long. Our explanation for this behavior is that the USB bandwidth is maxed out at this point, which slows the execution dramatically, forcing the request to take longer than is really needed for the data transfer. If we had a camera which supported higher resolutions, we could continue to take more data points and see if the latency continues to increase for higher resolutions. For now, this result leads us to encourage exploration of other data transfer protocols like USB 3.0 or FireWire which are better suited for large data transfers.

The next graph shows the latency when four cameras are running in parallel. Each of the cameras ought to take roughly the same amount of time to retrieve each frame, since each camera is set to the same resolution.

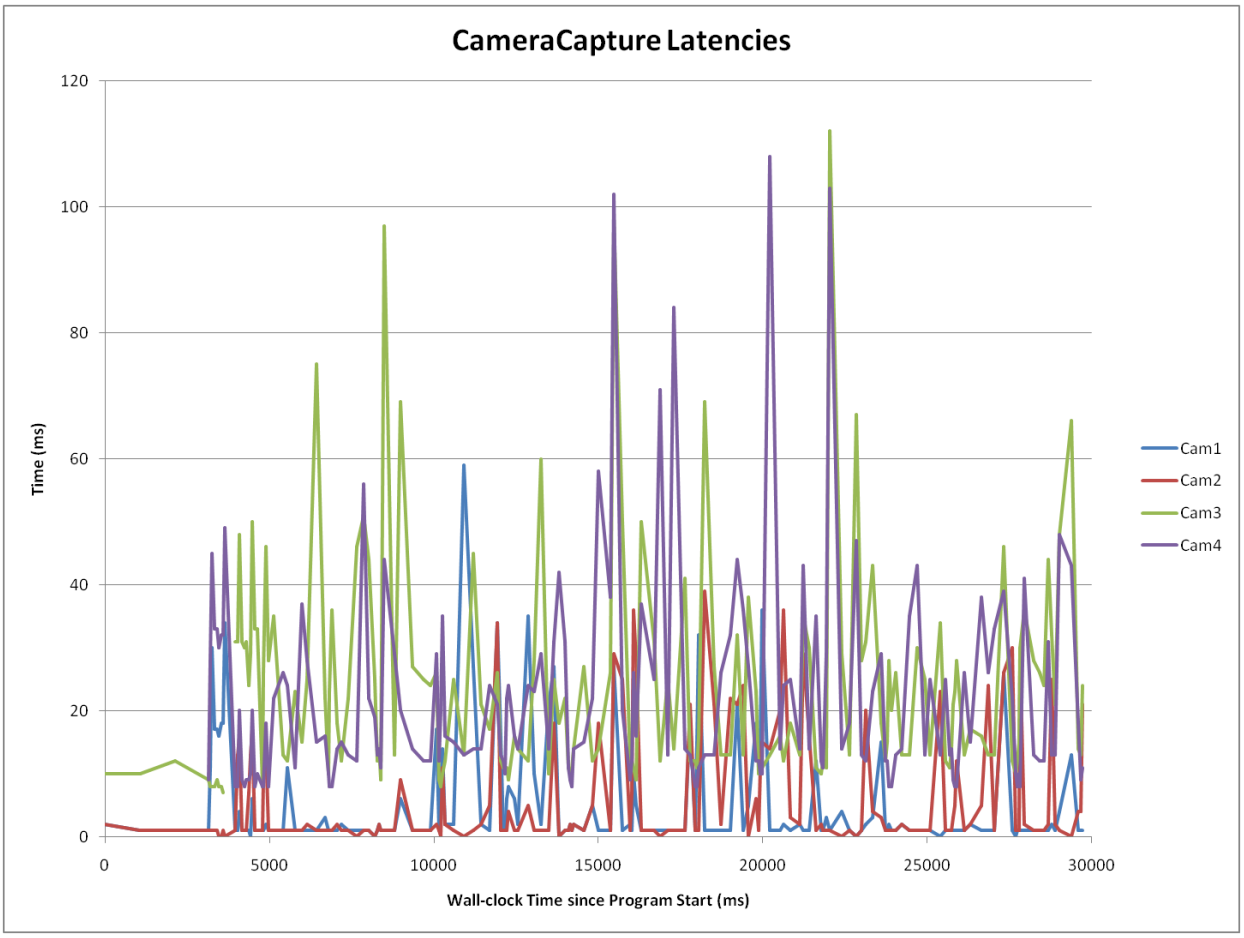

Figure 11: CameraCapture Latencies

Again, the results are disappointing, and likely due to USB 2.0 limitations. The time needed to retrieve an image varies widely between successive requests, and some of the cameras take more time on average than others, rather than being uniformly distributed. Worse than that, in some tests we ran, we saw that the cameras which were going slowly switched: for a while camera 1 would be slow, and then at an arbitrarily juncture camera 2 would be slow instead. Again, we chalk these issues up to the USB cameras.

5.1.2. Homography Calculation

Timing information was taken at more points in the homography calculation than was done in other modules, since the steps in this module are easily separable. This graph is one of our

earliest results for the project, and still does a good job of displaying the latency in each of the parts of the Homographier process.

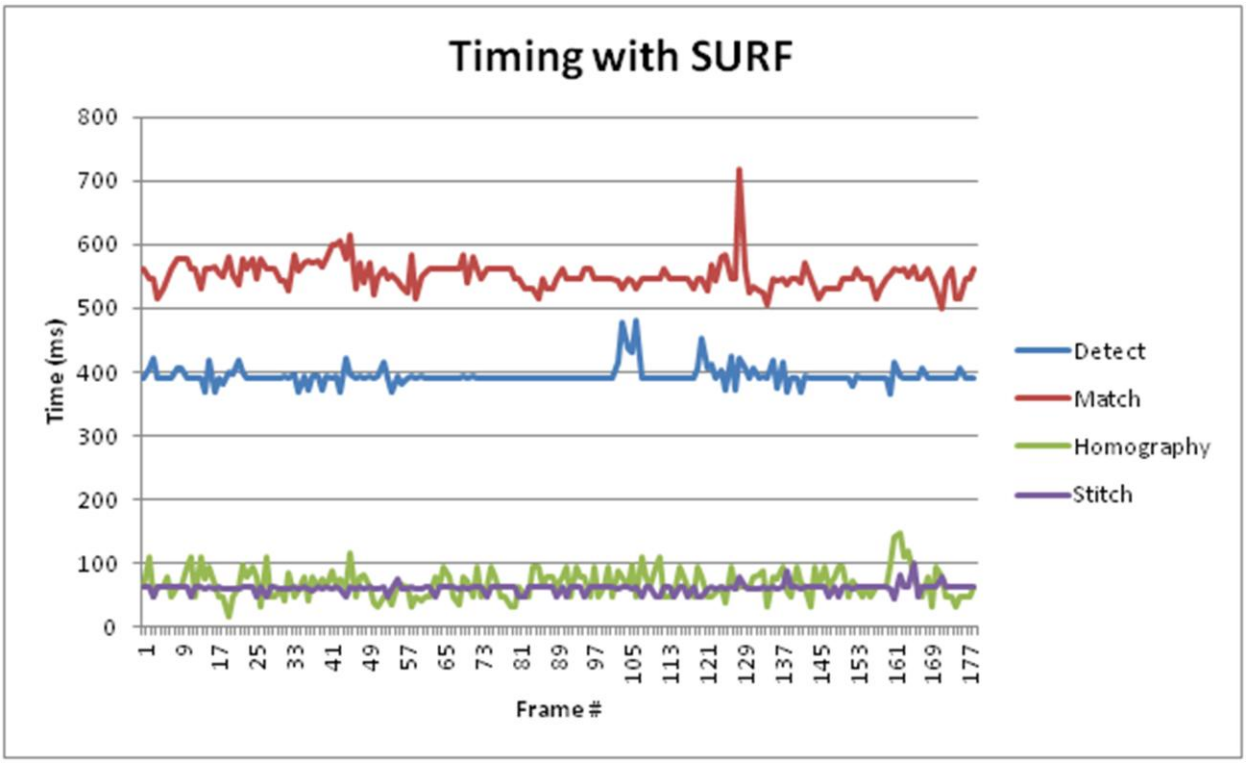

Figure 12: Timing with SURF

Here is a more recent version of the timing data for the Homography module. This shows timing information for the different steps over a longer time interval, during which several of the parameters were changed through the GUI which affected the Homographier.

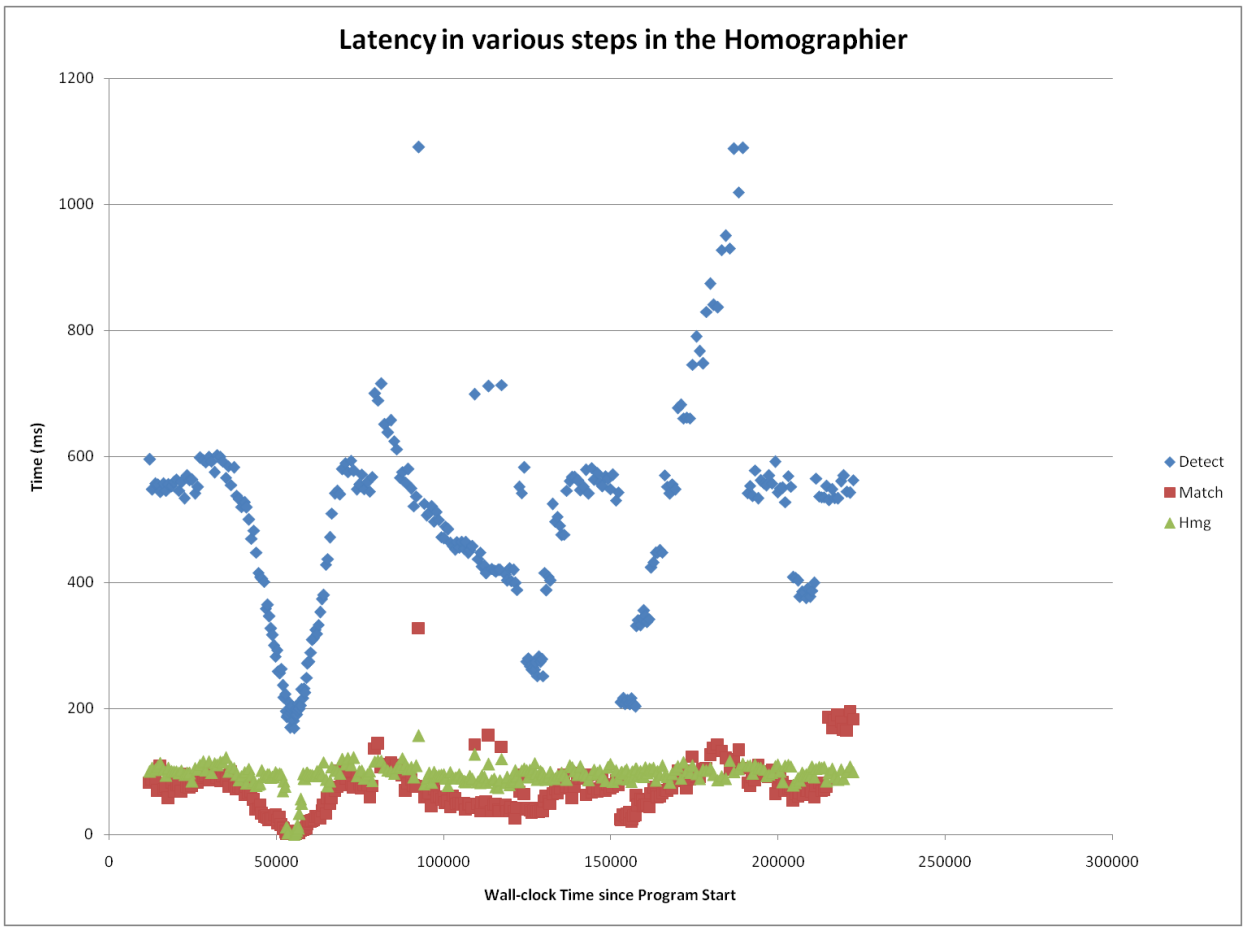

Figure 13: Latency in the Homographier

In general, when the detect phase goes quickly, some unwise shortcut has been taken and the output video is extremely inaccurate. For example, the first dip in the *Detect* phase corresponds to the "frame overlap" value being decreased, thus decreasing the size of the image being analyzed for keypoints. When this value goes to 0, then no part of the image is being analyzed, and no keypoints are detected, so the homography cannot be calculated. The second visible feature in the graph, the section in which the *Detect* time is smoothly decreasing at around 100000 ms, corresponds to the hessian threshold being increased from 100 to 1000, whereas the default value is 500. When the hessian threshold increases to 1000, a large amount of keypoints are discarded due to having a low hessian value. This makes the *Detect* and *Match* phases execute more quickly, but also makes the resulting homography less accurate, in general. And, finally, the last clearly visible feature is the dramatic increase in latency from 200 to 1100 ms in the latter half of the graph. Each of these steps corresponds to an increase in the number of octave layers used to calculate the SURF keypoints. Using more octave layers returns a much higher number of keypoints. Again, when more keypoints are detected, the calculated homography is more accurate, though the computation is much more expensive.

This graph shows how changing the Homographier settings can affect the latency of the Homographier module, but it does not show how the output video varies according to these settings. Neither do the demo videos do a good job of showing the results of changing these parameters. This is because the results are very hard to determine. The user must decide for

himself what the priority is for his application: whether he wants to sacrifice latency in the Homographier module for the sake of accuracy, or vice versa. The default values provided give a good starting point for tinkering with the values, so that the user can validate the software for himself. Another helpful tool for evaluating the Homographier parameters is to view the keypoints that have been detected and matched between two images.

Figure 14: Showing Homographier matches

Viewing these matches can help the user evaluate which changes result in more accurate matching keypoints and which simply increase the latency unnecessarily.

5.1.3. Stitcher

The final module which was timed was the GPU stitching module. We expect this module to run at a constant speed independent of changes in the Homographier settings, very minimally dependent on the Stitcher settings, and linearly dependent with the size of the images being stitched.

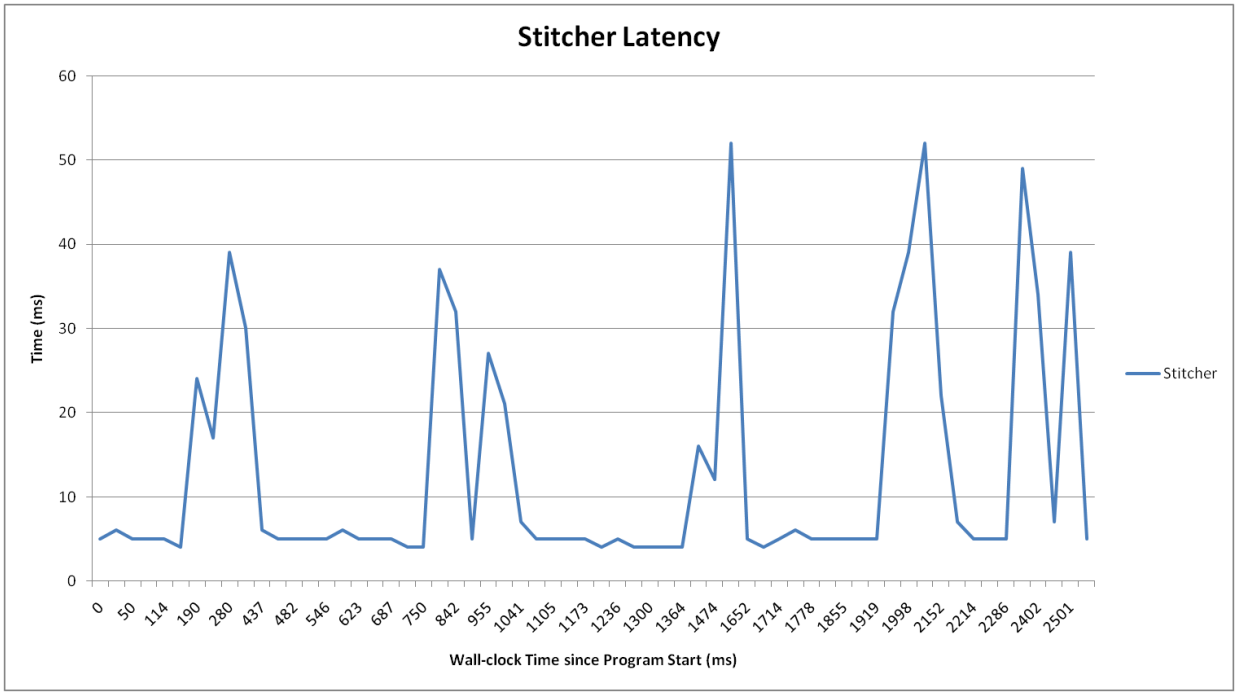

Figure 15: Stitcher Latency

What we see instead is another troubling result. We see that sometimes the stitching is calculated very quickly, while at irregular intervals the time required to stitch spikes by up to a factor of ten. More in depth analysis showed that this increase in time happened not in the stitching module itself, but during the copying of the images onto the GPU. This copying is the first request of the GpuStitch module for the GPU resource on each iteration, so we hypothesize that the Stitcher is waiting for the GPU to be released by some other process, likely Windows Explorer which is responsible for updating the screen and which does so using GPU hardware acceleration. Using a second, dedicated GPU should remove these spikes, but we were unable to test this hypothesis this semester, only having access to a single GPU.

6. User's Manuals

This manual is intended to instruct a user how to build the code and required libraries for this project, and then how to use the program once it has been compiled. Several third-party libraries need to be built in order to compile this code, so the user will need to be fairly knowledgeable about building and configuring libraries both from the windows command prompt and through Visual Studio in order to build this program. This manual will make no assumptions about the user's knowledge of computer graphics, however. The user should be able to use this manual with only a surface level understanding of the algorithms and data structures described above in order to use the program.

6.1. Hardware Installation

Cameras used for this program must be compatible with OpenCV. A list of OpenCV-compatible cameras is available at http://opencv.willowgarage.com/wiki/Welcome/OS , but it simply lists cameras which have been specifically tested. Other cameras may work, but we do not know which specific factors determine OpenCV compatibility. In order to use our present functionality for pan/tilt adjustment, the pan/tilt motors for each camera must be accessible through the DirectShow library. DirectShow is a fairly standard interface in the Windows SDK, so most cameras should be compatible with DirectShow in general. However, since DirectShow is specifically a video input library, we doubt that external pan/tilt harnesses will be accessible through DirectShow without extra work.

StitcHD does not make any specific hardware requirements other than OpenCV-compatible cameras and a GPU. While the project can compile without adding GPU support, for optimum performance we recommend installing an NVIDIA graphics card. Because OpenCV's GPU functions and our custom GPU kernel are written in NVIDIA CUDA, other brands of graphics cards are not compatible with our system. We also recommend a high-powered, multi-core processor because of the multi-threaded nature of our application.

6.2. Software Installation

This project depends on several 3rd party libraries in addition to the libraries built in our solution. These libraries must each be installed using the proper configuration, and referenced using the correct environment variables before the StitcHD project can be built.

6.2.1. OpenCV (2.3)

The core of this project utilizes the OpenCV library to analyze and transform the images. OpenCV is primarily a C based library, but OpenCV 2, released in 2009, is written in C++, and adds several new key features. Wrappers have been written for the earlier, C version of the library for C#, Python, Ruby and Java, but these wrappers lack many of the features available in

OpenCV 2.0 and later, features upon which this project depends. So, in order to use OpenCV, this project must be written in C++.

The OpenCV library should be compiled with CUDA and Qt support. If you are using a multi-core processor, the IntelTBB library should be linked in as well. All of the installation mechanics are well-documented on the OpenCV site, as well as instructions for installing the dependent libraries.

NOTE: Make sure you carefully look through the list of CMake options when compiling the library. All of the CUDA variables may not link automatically, and since it takes ~30 minutes to build the library, you may not find your mistake for a while. Look through the CMake output and get rid of any errors you find, and make sure everything is getting compiled that should be. Don't bother compiling with any Python support.

6.2.2. Qt (4.8.1)

The GUI for this project is written in QT, an open-source framework owned by Nokia. It provides many useful cross-platform features beyond the simple visible GUI widgets. We also utilize its event notification system in the background for communication between windows. In fact, the Qt interface is so good that we wish we would have designed the entire project in Qt. It was only near the end of our development cycle that we added the graphical component to our project, after we had already implemented all of the cross-thread communication in the StitcHD .dll project using the cumbersome .NET structures.

6.2.3. CUDA (4.0)

Nvidia's Compute Unified Device Architecture (CUDA) provides an SDK for interfacing with Nvidia GPUs. The OpenCV library can be compiled with support for GPU, referencing the CUDA library. For this project, we decided to write our own CUDA function (called a kernel) to perform the stitching. The SDK provides several extensions to C++. We split up the code written in CUDA into a separate .dll project called GpuStitch so that the project could also be compiled without a GPU or without installing CUDA.

NOTE: OpenCV 2.3 does not work with CUDA 4.1. If you want to upgrade OpenCV to 2.4 or CUDA to 4.1, you should upgrade both.

6.2.4. Boost (1.49)

Boost is used for a small portion of this program as well. It handles the file I/O operations for writing timing data, videos and pictures that are produced by the program. This module could be quickly rewritten without the need for Boost, perhaps by replacing it with suitable Qt functions, but we used it before adding Qt to the project, simply to avoid the cumbersome .NET functions for file I/O.

6.2.5. DirectShow

DirectShow is a library included in the standard Windows SDK which is used to control video input devices such as webcams. DirectShow can be used for a variety of functions, from frame capture to changing attributes like focus, zoom, and pan/tilt. For frame capture, OpenCV provides an easier way to interface with compatible cameras, so we only use DirectShow to adjust the built-in pan/tilt motors of our cameras.

The code we use to access the DirectShow functions was developed by the SkyView senior design team, based on some sample code retrieved from a defunct version of the Logitech website. Theoretically, this code should be compatible with any DirectShow compatible cameras, except for the function *reset_mechanical_pan_tilt.* This function uses pan and tilt values which are specific to the Orbit AF and will probably not reset other cameras correctly.

Originally, we had hoped to use DirectShow as a way to allow cameras to share the same settings for focus, exposure, etc., or to just turn off automatic adjustment. We believe that this may be possible, but we were unable to implement this feature in time.

6.2.6. Platform Requirements

Each of the different libraries used has its own platform requirements, though most of them are fairly platform independent. OpenCV works on all platforms (Windows, OSX and Linux; 32-bit and 64-bit). Actually all of these libraries have full platform except for Qt and DirectShow. Qt is nearly platform independent, but they have placed Windows 7 64-bit systems on their "Tier 2 Platforms" list. And DirectShow is only usable in Microsoft operating systems. In addition, we have used several Windows-specific function calls in the StitcHD project for inter-thread communication. So, this project is constrained to Windows 32-bit operating systems. We have run it on Windows Server 2008 SP1 with 32-bit libraries, and on Windows 7 SP1 with 32-bit libraries.

If you were to extend this project to other platforms, like Linux or OSX, you should use Qt's abstractions of threads that are platform-independent in the StitcHD project, and you should remove the code which controls the movement of the cameras, or else find a way to do it on other platforms using tools other than DirectShow. Another consideration you would need to make is the camera selection. OpenCV supports some cameras natively, but many it does not. Some cameras work on some platforms and not on others. We know that our Orbit AF cameras combined effortlessly with OpenCV, but that is not guaranteed for all cameras.

6.2.7. Visual Studio

This project requires Visual Studio 2010. The VC10 compiler is necessary for the GpuStitch library, which is compiled with CUDA's nmake compiler, to link with the rest of the C++ code. In addition, the CUDA plug-in and the Qt plug-in should be installed to tell Visual Studio how to compile the various projects in the solution.

6.3. Operating Instructions

Once all the necessary libraries have been installed, open the StitcHD executable to begin using the system. The program should open up the main menu, as seen earlier in the Implementation section. From the main menu, you can set parameters which cannot be changed while the program is stitching. These include the number of desired cameras, which cameras are inverted, and what resolution the cameras should be set at. Beware of setting the resolution too low as well as setting it too high. A very low resolution results in very few keypoints being detected and very erratic stitching results, while a very high resolution will run frustratingly slowly.

After setting these parameters, we recommend aligning the cameras in View Cameras mode. Although our system is designed to warp and shift frames into a similar perspective, this works best when the cameras are lined up. Specifically, this means that cameras which are above and below each other should have the same relative x-axis position, and cameras which are left and right of each other should have the same relative y-axis position.

To do this, arrange the four (or fewer) camera windows on the screen in the correct relative positions and with the edges of the windows lining up. Next, using the pan/tilt buttons in each camera window, move the camera views away from each other to the desired level of separation between cameras. (Be sure to leave enough overlap for the stitching calculations to be accurate.) Finally, choose a reference point in each pair of frames that you can see in both camera views. Adjust the pan/tilt until they line up.

After the cameras have been adjusted to the desired positions and overlap, close the View Cameras windows and run the program from the MainWindow. This will bring up the SettingsWindow and the DisplayStitcHD window, where the stitched video will be displayed. While the VideoStitcher is running, you can observe the frame rate at the top of the DisplayStitcHD window and observe changes to the latencies of the two main modules as well as view changes in the stitching quality as you change the parameters given in the SettingsWindow. The settings filed under the "Stitcher settings" group affect how the images are blended together, while the settings under the "Parameter calculating settings" group are more difficult to understand initially. Checking the "Show Matching Keypoints" box should help to explain what is happening behind the scenes as the homographies are calculated.

Assuming that the cameras have been aligned properly, the default settings should give visually pleasing results at reasonable speeds.

7. Course debriefing

Our main avenue of quick and effective communication was the GroupMe text function that provides a number for you all to text that is basically a group chat. We also had a website, but it ended up being underutilized due to the convenience of Google Docs. All of our documents, including presentations, reports, as well as the weekly agendas, were uploaded to Google Docs and shared with the other two group members. This ended up being the most convenient way to keep everyone up to date about project information. When writing the reports, since coordination to meet in person was difficult, we would all just sign in to the Google Doc and work on it at the same time. We are able to all edit simultaneously, as well as utilize the chat function in Google Docs to discuss any questions we may have.

Our team was able to meet approximately twice a week throughout the semester. These meetings were the most vital in getting our project done because we were able to discuss questions, confusion, and inconsistencies more thoroughly in person. It also allowed for group brainstorming sessions, which, in the end, progressed our project the quickest. However, we also had individual assignments. Heather was in charge of communicating with NASA, creating the GUI, and creating the camera mount. Sam's assignment was calibrating the cameras, and the homography calculation module. Luke always kept up with the latencies of each module to make sure we knew where we needed to improve, and he created the stitching module. All parts of the project were successfully completed.

The only thing we would have changed if we were to do the project again would be a greater amount of face to face communication in the early stages of the semester. We noticed the problem and fixed it about a third of the way into the semester. If we had collaborated with more ease in the beginning, our project may have progressed quicker.

Since our team had the special privilege of having NASA as our client, we had the extra proponent of communicating with them on a regular basis. The main consistent collaboration we

had was a bi-weekly Telecom where we discussed our progress as well as brought up any questions we had about the desired product. We also were sure to e-mail NASA all presentations and reports that we wrote up for class to keep them updated. We visited Johnson Space Center mid-way through the semester to present our Critical Design Report, meet the NASA employees face to face, and toured their lab for a better understanding as to where all of this technology is going. This helped us greatly to have a better vision for our project, and directed it in a way that we progressed much faster than before. They were very helpful and encouraging, and eager to answer any of our questions to the best of their ability. When we presented, they did not hesitate to compliment what they felt necessary, which provided a giant boost to our overall morale. Our experience in working with NASA employees was a very positive one, and we are very glad we got the opportunity to do so.

The only ethical and safety concern we encountered was the possibility of filming people without permission if our system were to be used for surveillance. This, however, was not the primary purpose of our project, and this ethical concern has already been solved since surveillance is so common today.

Our project was tested in a variety of situations. We rolled around campus and stitched video in many different scenarios. These included the library, out of a second story window to view traffic, as well as a few others. This was to ensure that our project could work in a variety of situations. If we were to do this project again, we probably would have at some point tried to film/stitch the night sky. This is important since our target scenery has been space all along, but we did not manage to test this.

8. Budget

- LG Super-Multi DVD Burner 22X DVD+R CD-ROM Black SATA Model: $15.99
- ASUS Sabertooth X58 LGA 1366 SATA Intel Motherboard: $194.99
- 12GB (3 x 4GB) DDR3 SDRAM Desktop Memory: $69.99
- 2 Crucial M4 2.5" 64GB SATA III MLC Internal Solid State Drive: (2 x $88.99 = $178.99
- EVGA GeForce GTX 560 2GB Video Card: $279.99
- Intel Core i7-950 Bloomfield 3.06GHz LGA 1366 Quad-Core Processor: $269.99
- Black Aluminum ATX Full Tower Computer Case: $179.99
- OCZ ZS Series 750W 80PLUS Bronze High Performance Power Supply: $89.99
- 4 Logitech Quickcam Orbit AF webcams: ($130 x 4 = $520)
- 2 PlayStation Eye Prototype Cameras: ($40 x 2 = $80)

We did not end up using the 2 Prototype Cameras because they arrived too late, but we do not think they would have been very helpful in hindsight. The rest of the purchases were not only vital, but there is a known direct correlation in quality of stitching output and quality of hardware. Therefore, if more expensive hardware were to be purchased, the output of our project would improve. More importantly, if better cameras were used, the image quality would increase rapidly. The pan/tilt functions on our chosen cameras caused many difficulties, and using USB cameras is not the most efficient choice. With this in mind, it is hard to guesstimate a price for mass production. The two main uses for our project currently would be for surveillance

or to see outside of a windowless room (spacecraft). The price will completely depend on the desired quality of camera and hardware, and the desired outcome.

9. Appendices

9.1. Alpha Blending

Most of the options for blending between images are outlined in Szeliski's article, "Image Stitching and Alignment: A Tutorial," under sections 6.2: Pixel Selection and Weighting, and 6.3: Blending. There are varying levels of sophistication in the way that the images are blending, but we needed to choose blending techniques which require only simple calculations that can be done for each pixel independently and done quickly.

The simplest choice programmatically is to choose a hierarchy of images, such that in each overlap region, only one image is considered. The result is an image that looks like a bunch of overlapped pictures.

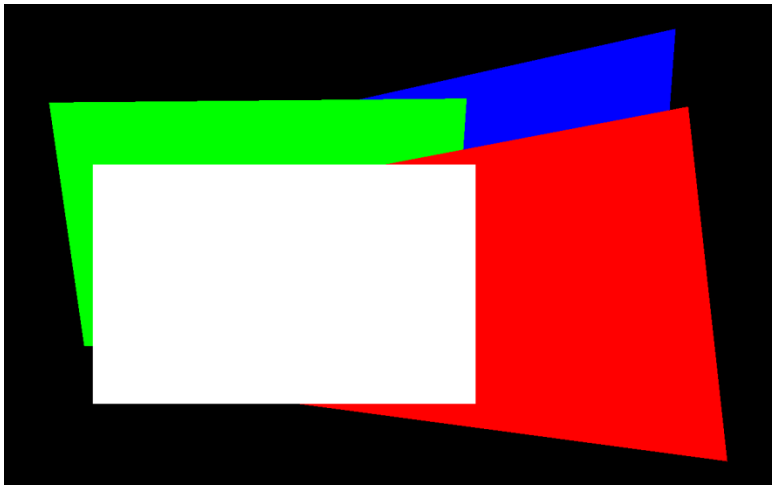

Figure 16: Overlapped Blending

This is equivalent to no blending at all. Each edge is a hard transition from one frame to another. Since the homography perspective calculation is not a true representation of the transformation between the two images, as explained in the section regarding the stitching pipeline, and since the homography calculated changes over time due to minute variations in the images, these edges will not be smooth. Straight lines will have discontinuities at the edges, and color differences between the images will make these transitions glaringly obvious.

The simplest way to blend between the two images is to take the average pixel value from each frame in the overlapping regions.

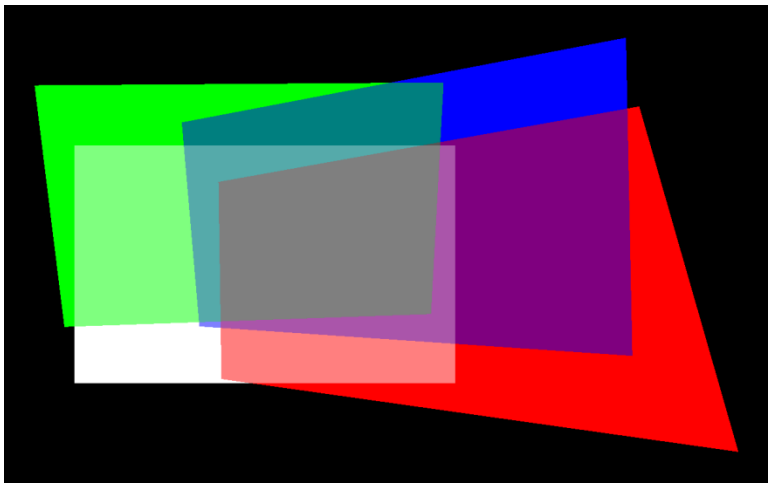

Figure 17: Average Blending

With average blending, each edge becomes a softer transition. As you progress from image 1 to image 2, there is an intermediate transition region where the blending is happening. However, with simple average blending, you begin to see ghosting effects, or artifacts, in the image. These occur when the same object appears in both images, but the object exists in a different position in one image than it does in another. This is often due to parallax error when one object is closer to the cameras than the rest of the objects. But these can also occur simply because the transformation is imperfect.

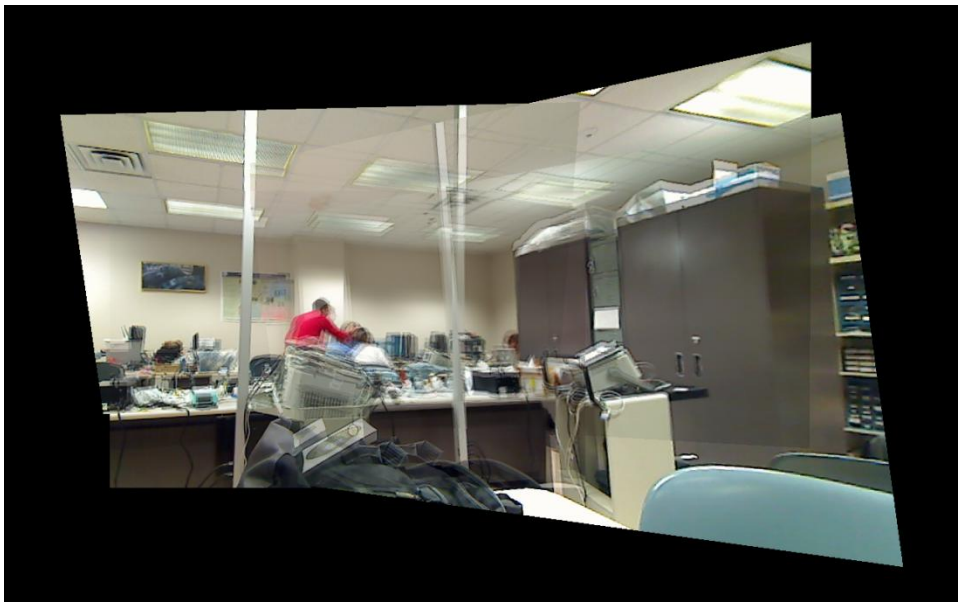

Figure 18: Average Blending

In the figure above, the central column exhibits this ghosting effect. There appear to be two copies of the same object. In this case, the ghosting is largely due to the column being closer in space to the camera array than the background of the image. We want a solution that minimizes ghosting effects while still leaving a large enough blend region that the transitions between the images doesn't appear overly harsh.

The premise behind our linear and exponential blending techniques is to assign a weight to each image that overlaps on a particular pixel. The output BGR value is the weighted average of a pixel in the overlapping image (or, an average of pixels if we're using bilinear interpolation). This multiplier is calculated as a function of the distance of a pixel from the center of its image. This gives us the desirable quality that the overlap region centers on the midpoint between two images. Mathematically, the overlap regions mark out the Voronoi regions around the midpoints of each image.

One way to assign the weights for each image is linearly. As the pixel moves away from the center of its image, the weight decreases linearly.

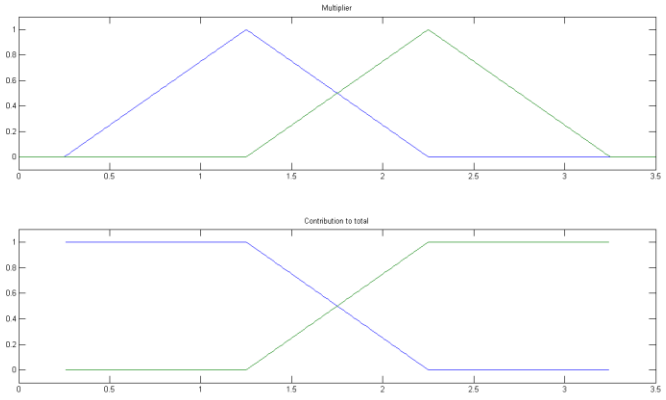

Figure 19: Linear Blending - Math

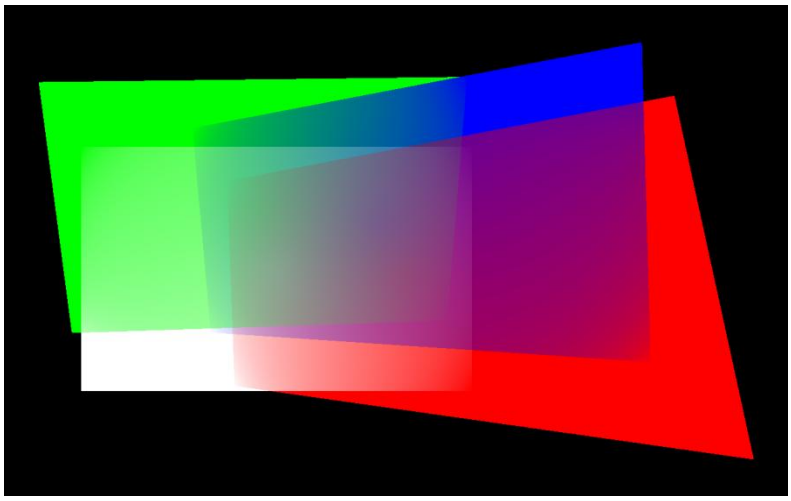

Figure 20: Linear Blending - Visual

This works better than average blending, but actually not much better. What we would like to do is choose the weights such that for each pixel closer to one image than another, the difference in weights increases by a factor of *x*, where *x* is an adjustable parameter. This would give us control over the width of the blending region. This can be done using exponential widths, where as the pixel is removed further from the center of its image, the weight decreases exponentially.

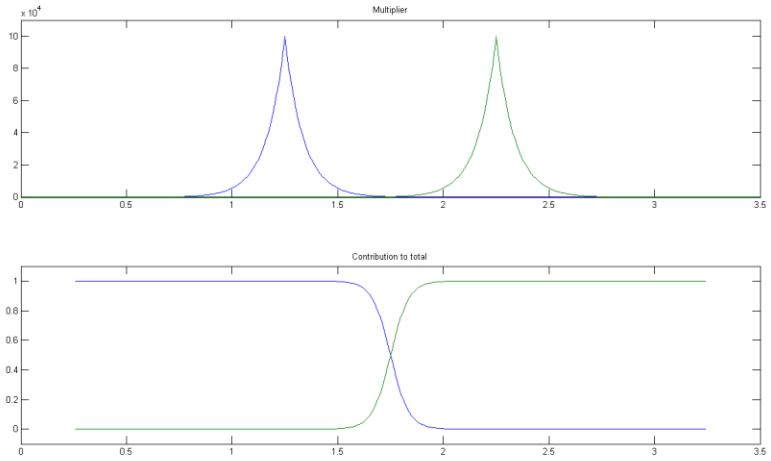

Figure 21: Exponential Blending - Math

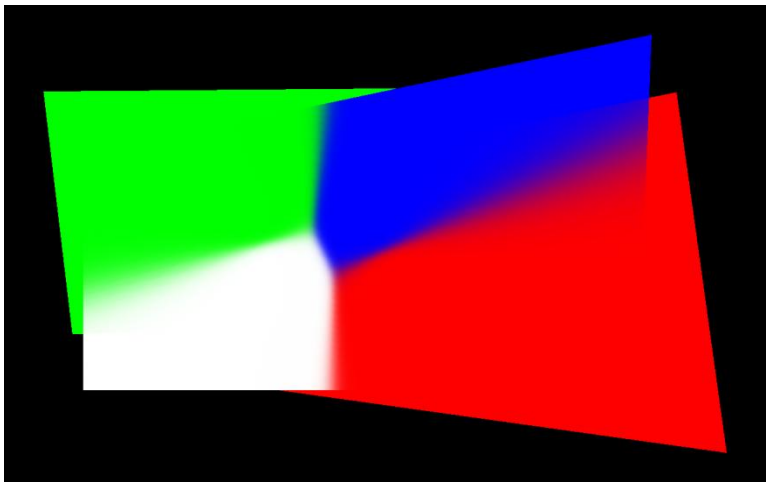

Figure 22: Exponential Blending - Visual

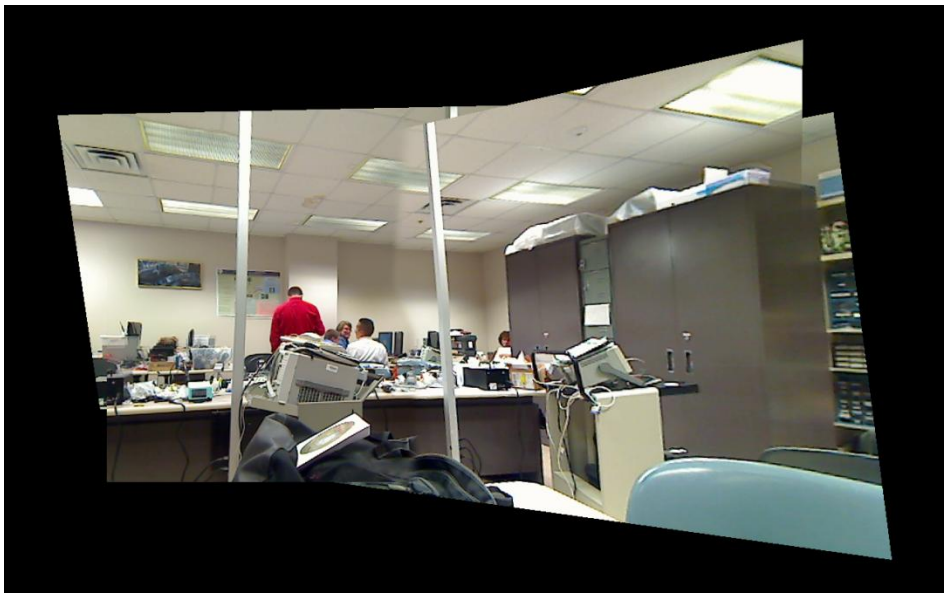

Figure 23: Exponential Blending - Example

As you can plainly see in the above example, the overlap region is quite small, leaving very little room for artifacts to appear, but the transition regions are very obvious, showing the color differences between the cameras, and the perspective transformation inaccuracies in the top of the image.

There is a practical limit on the strength of this blending due to floating point numbers, since floats can only represent positive numbers as low as 1e-5 and up to 1e37. For doubles, this increases to from 1e-9 to 1e37. So, for floats, the maximum strength exponential blending can only adjust the weights from 1e-5 to 1e37 / 255, since we are multiplying the BGR values by these weights, and the BGR values range from 0 to 255. We have provided a slider in the GUI so that you can adjust this parameter for yourself. You will find that for certain situations a low level of exponential blending is best, and at other times you will want hard, well-defined seams.