[[1]](#footnote-1)

Multi-Threaded Stereo Vision Positioning

Shiyuan Wu, Andre Lo, Xianglu Shi

*Abstract*— Stereo vision positioning uses images of the same scene from cameras separated by a distance to calculate 3D position of objects. The calculations involve detecting the object in each camera frame and then tracking the position of the object by comparing the frames. When done in real-time, this is useful for applications such as robotics. OpenCV implements many of the mathematical operations for image processing but to achieve better real-time performance we parallelized our use of OpenCV and achieved speedup using only CPU hardware.

# INTRODUCTION

T

HE problem of real-time 3D positioning of an object derived from sensor data is currently a hot topic that has important applications in multiple areas including robotics, automobiles and virtual reality. One solution to this problem is to use two camera in a stereo vision setup and calculate the position of a target object in 3D space. It has already been shown that this technology can be implemented in a single-threaded manner using OpenCV. Our project will present an implementation in a multi-threaded manner.

# Background

## OpenCV

The open source computer vision library, OpenCV, created and maintained by Intel, is designed for computational efficiency and aimed at providing the tools needed to solve real time computer vision problem. It is a free and open source collection of the low-level image-processing functions as well as high-level algorithms such as object segmentation and recognition, stereopsis stereo vision, and motion tracking. Because of its high performance it is widely used as an image processing software.

|  |
| --- |
| Macintosh HD:Users:ZuiZui:Desktop:屏幕快照 2016-11-17 9.57.02 PM.png  Fig. 1: Simplified Stereo Vision System [1] |

## Binocular Stereo Vision Measurement Technology

Nowadays, there are several techniques to infer the depth information using 2D camera sensors. Stereo vision is one of methods is similar to the manner in which human eyes did. Traditional stereo vision system uses two cameras to obtain two differing views on a scene. Ideally, the two cameras are separated by a short distance and displaced horizontally from one another. By comparing the two images, the perception of depth can arises from “disparity”, which is the difference in two images location of corresponding (homologous) points.

To better illustrate how stereo vision works, Fig. 1 shows the diagram of simplified stereo vision setup, where both cameras are perfectly aligned parallel to each other and have the exact same focal length.

|  |
| --- |
| Macintosh HD:Users:ZuiZui:Desktop:屏幕快照 2016-11-17 10.19.10 PM.png  Fig. 2: Geometric Graph for Parallel Cameras [2] |

The variables in Fig. 1 are listed as followed: b is the baseline that is the distance between the two cameras lens centers; f is the focal length of a camera; XA and ZA is the X-axis and optical axis of a camera respectively; P is a 3D point defied by the coordinates X, Y and Z; UL and UR is the projection of the 3D point P in an image acquired by the left and right camera respectively. The distance between those two projected points (UL and UR) is known as disparity, and the distance between 3D point and the stereo vision system is called depth. The basic principle of stereo vision is triangulation, which is the method to determine depth from disparity. In detail, the geometric graph for parallel cameras, shown in Fig. 2, can be used to clarify the principle. From similar triangles, the relationships among variables are given as:

(1)

(2)

(3)

where Y-axis is perpendicular to the page. Thus the location of the 3D point P can be derived from previous equations 1 – 3:

(4)

(5)

(6)

where d is the disparity which equals to , and z is the depth which is inversely proportional to

|  |
| --- |
| Macintosh HD:Users:ZuiZui:Desktop:屏幕快照 2016-12-06 1.57.31 AM.png  Fig. 3: RGB Color Space |

|  |
| --- |
| Macintosh HD:Users:ZuiZui:Desktop:屏幕快照 2016-12-06 2.32.30 AM.png  Fig. 4: HSV Color Space |

disparity. Therefore, two main problems arise for this project. Firstly, we need to figure out the value of focal length f and baseline b by using prior knowledge or camera calibration. In addition, the corresponding points (xl, yl) (xr, yr) for each camera need to be found.

# Image Processing

## Work with Color Spaces

Color can be realized in different ways based on physical, physiological and technical aspects. This forms the basis of different color spaces that computer vision algorithms need to work with.

### BGR (Blue, Green, Red) Color Space

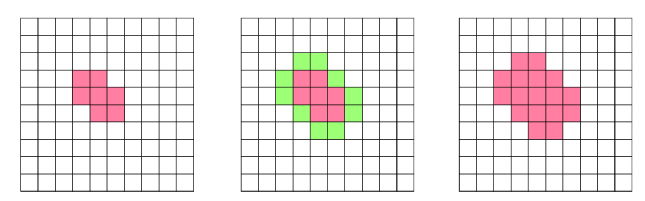
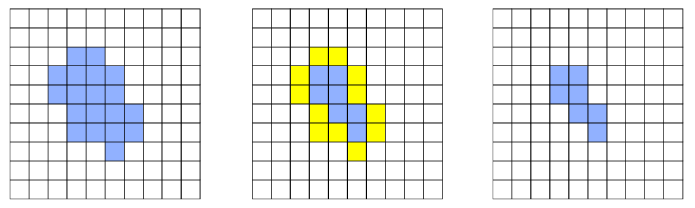
BGR color space, shown in Fig. 3, consists three kinds of colors, blue, green and red, which can be mixed in various ways to reproduce a broad array of colors. BGR mainly used in computer system. Images and videos captured by OpenCV usually is 8-bit BGR format. However, BGR color space is not a good option for color image processing due to machine dependency and integration of chrominance and luminance data [3]. To facilitate working around this problem, it requires converting the BGR color space to a different space that is more suitable for processing.

Fig. 5: Erosion Operation. The yellow pixels go to the background [15]

Fig. 6: Dilation Operation. The green pixels come to the foreground [15]

### HSV (Hue, Saturation, Value) Color Space

HSV color space, shown in Fig. 4, contains three ingredients, Hue, Saturation, and Value and is also often called HSB (B for brightness). Hue represents color type that range 0-360 degree. Saturation refers to the vividness of color, which is the amount of white mixed with respective color in range of 0%-100%. Value is the brightness of color, which is the amount black mixed with respective color in range of 0%-100%. In OpenCV, value range for Hue, Saturation and Value are 0-179, 0-255 and 0-255 respectively [4]. The separation between luminance part and chrominance data make HSV color space can provide good performance in image processing. Thus, HSV color space is the most suitable color space in computer-vision and image investigation for element recognition or image segmentation [3]. So, in the project application, the color space of original image of the video needs to be converted from BGR to HSV image.

## Mathematical Morphological Operations

Mathematical morphological operations are a wide range of operators that contributes to image processing based on shapes. Morphological operators apply a structuring element (also known as kernel) to an input image and creating an output image of same size. The operators are particularly useful for the analysis of binary images and have a wide range of uses, i.e. image noise removal, enhancement and segmentation. The basic morphological operators are dilation and erosion.

### Dilation

Dilation operation can be considered as a swelling procedure. In dilation operation, an image A convolutes with some kernel B. There is an anchor point located the center of the 3x3 kernel. The basic effect of the operator on a binary image is to gradually enlarge the boundaries of region of foreground pixels. Scanning the kernel B over the image, we compute the maximal pixel value overlapped by B and replace the image pixel in the anchor point position with that maximal value. As Fig. 5 shows, the background pixel is changed to foreground when dilation is applied.

### Erosion

Fig. 7: Basic OpenCV image processing workflow from camera video feed

Erosion operation can be viewed as a shrinking procedure. The basic effect of the operator on a binary image is to gradually erase away the boundaries of region of foreground pixels. What this operation does is to compute a local minimal pixel value overlapped by B and replace the image pixel under the anchor point with that minimal value. As Fig. 6 shows, the foreground pixel is changed to background when erosion is applied.

# Implementation

The implementation is set up on 64 bit Windows 10 operating System with Intel® Core™ i5-6200U @2.30GHz CPU processor and 8GB RAM. Initially the plan was to use 2 Sony PlayStation Eye web cameras to handle the video feed. There is existing multi-camera driver for commercial usage called CL-Eye Platform Driver [5]. However, it does not support Windows 10 Operating System. Therefore, we will leave dual camera driver involvement for future work. For this project, we switch our plan to use pre-recorded video clip to act as video feed and perform our single-thread and multi-threaded simulation based on that.

## Base OpenCV Image Processing

|  |
| --- |
| C:\Users\Shiyuan\AppData\Local\Microsoft\Windows\INetCacheContent.Word\Setup.jpg  Fig. 8: Stereo video capture infrastructure layout |

The backbone of image capture and process is from Kyle Hounslow [6] which provides a demonstration on how to transfer video captured image to HSV value and then use the threshold adjustor to find object in specific HSV value range. Fig. 7 shows the process data flow. Video frames from camera are translated from RGB value to HSV value using OpenCV function “cvtColor”. Then by adjusting the threshold object bars we can find perfect value range to filter down the specific green object to white and all the rest to black background in Threshold domain. For the green object in Figure 4.1 we can see the adjust value range is: H(67-84), S(56-256), V(96-256). We will ask the program to find all object within this range during real time video streaming feed and thus we can keep track of the project.

## Environment Setup

We use OpenCV version 3.1.0 and integrate it with Visual Studio Community 2015 running on 64 bit Windows 10 operating system. OpenCV configuration was performed referring to tutorial [7].

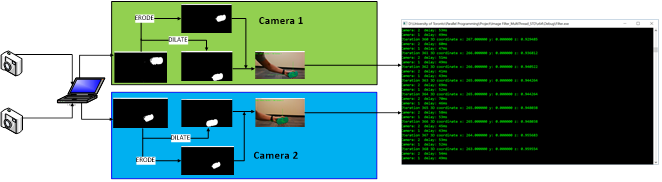
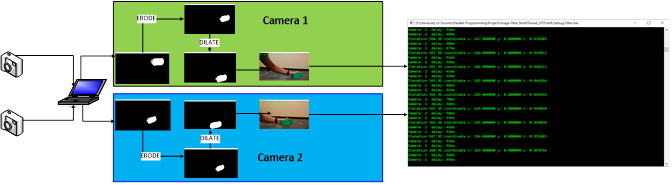
After that, we configure our stereo capture infrastructure as Fig. 8. Two Blackberry Classic smartphones with 2MP 2.31mm fix focal front cameras are used as the stereo cameras. The distance between two cameras is 140mm. There are several fixed distance points acting as simple calibration points which will be used to perform Z-index mapping. As discussed in above sections, the distance from the green object to the cameras is calculated per following equation (6) where b is the distance between 2 cameras and f is the camera focal length, xl and xr are the x-axis location of the object in two camera views. Here we notice that the scale of b and f are in mm while xl and xr are in pixels. Therefore, the z value derived from this equation will be mm\*mm/pixel which is not a scalable value. Hence we build the simple calibration system to build a distance map between the derived z value to a distance value in milometers. Due to time constraint, camera calibration was not performed and thus we can notice significant distortion from the captured video which will lead to inaccurate when the object appears at the edge of camera vision. We will leave camera calibration part to future work.

Fig. 9: DWORD WINAPI Implementation

Fig. 10: Std:Thread Implementation

## Single thread implementation

Single thread implementation is show in Figure 4.3. All image processing on the 2 camera video feeds are performed in sequence. Once both camera feeds are transferred into HSV domain and filtered by the threshold range, the camera first takes video feed frame from camera 1 and perform erosion [8]. All white points with pixel size of less than 10 by 10 pixel rectangles are treated as noise and are deleted from the threshold domain. After that dilation is performed to those points which are larger than 16 by 16 pixels and the outcome is intensity bumps and holes of the filtered image being wiped out so that the image is clearer for tracking [8]. After that, erosion and dilation is done on video feed frame of camera 2. Then tracking is performed to the threshold domain camera 1 and 2 frame in sequence using function “trackFilteredObject” and x, y value on both camera visions are captured. With the x value, together with the camera distance b and focal length f, we can calculate the distance between the object and the camera.

## Parallel implementation

We implement our pipelining parallel version in two different approaches: Microsoft DWORD WINAPI and Std::Thread. WINAPI is an C API for Microsoft Windows system while Std::Thread is new C++ standard library [9]. Std::Thread has been commonly agreed to be better for portability and easier to use. This is also proved from our experiment as shown in Fig. 9. We have been able to perform additional parallelization on erode and dilate functions using Std::Thread and received better performance as a result.

|  |
| --- |
| Fig. 11: Flow chart of the 3 threading approaches |

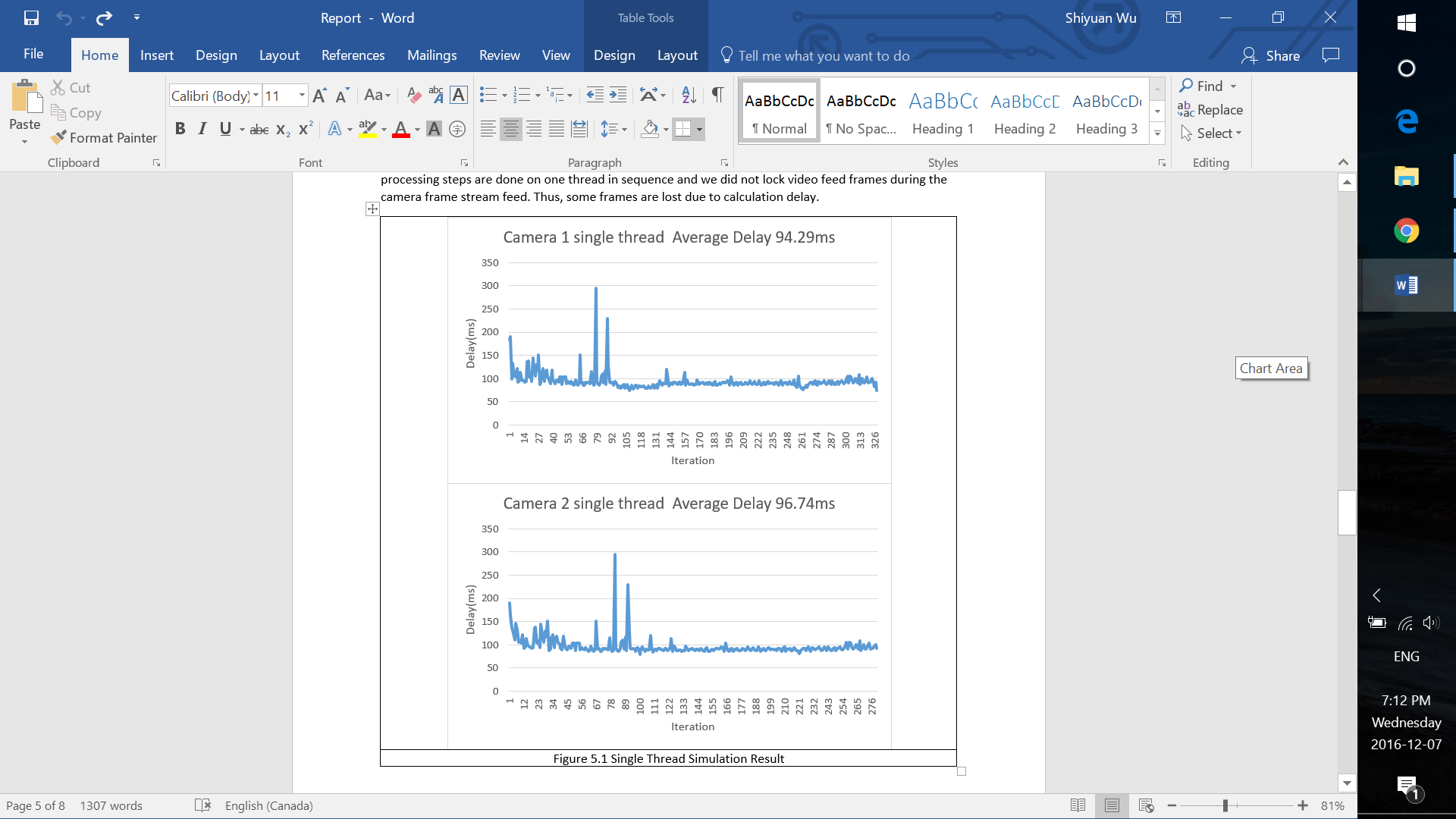
### Windows DWORD WINAPI

The WINAPI implementation is shown in Fig. 10. We created two threads to handle the video stream of camera 1 and camera 2 separately. Additional thread is created to perform calculation on z value once x,y of both image frames are ready. In other words, we created 3 threads in total: Thread 1 handles object tracking on Camera1, Thread 2 handles object tracking on Camera2 and Thread 3 handles the 3D calculation and reconstruction. The dependencies between the camera threads and the calculation threads are handled as following. We use iteration variables and store tracked x and y coordinates into buffer array and the calculation thread has its local loop variable. Each iteration it first check the two camera buffer arrays and see if according coordinate is already buffered, if yes perform the calculation otherwise it will wait for next round. We noticed Erode and Dilate on filtered object can also be parallelized, however we did not find approach to implement this using WINAPI. We did, however, fulfill this using the Std::Thread approach.

|  |  |
| --- | --- |
|  |  |
| B. WINAPI Multi-Threading |
|  |
| 1. Single Thread | C. std::thread Multi-Threading |
| *Figure 11 Flow chart of 3 approaches* | |

### Std:Thread

Std::Thread implementation is shown in Fig. 9. We can notice the main difference between this approach and WINAPI is that the erode and dilate workflow on both cameras are further parallelized. During the implementation stage we notice the Std::Thread library is much easier for the developer to use compare with WINAPI. It requires much less coding and does not require developer to concern on address and garbage collection. Therefore, to achieve better speedup, we run image enhancement and noise control in parallel. There is argument that Erode and Dilate should be done in sequence because they are depending on each other. However, because our image tracking happens on final threshold domain where all thresholds are finalized, thus even if erode and dilate violates dependency requirements there will be no downstream impact to threshold domain. The experimental results did not show visible exceptions caused by this additional parallelization



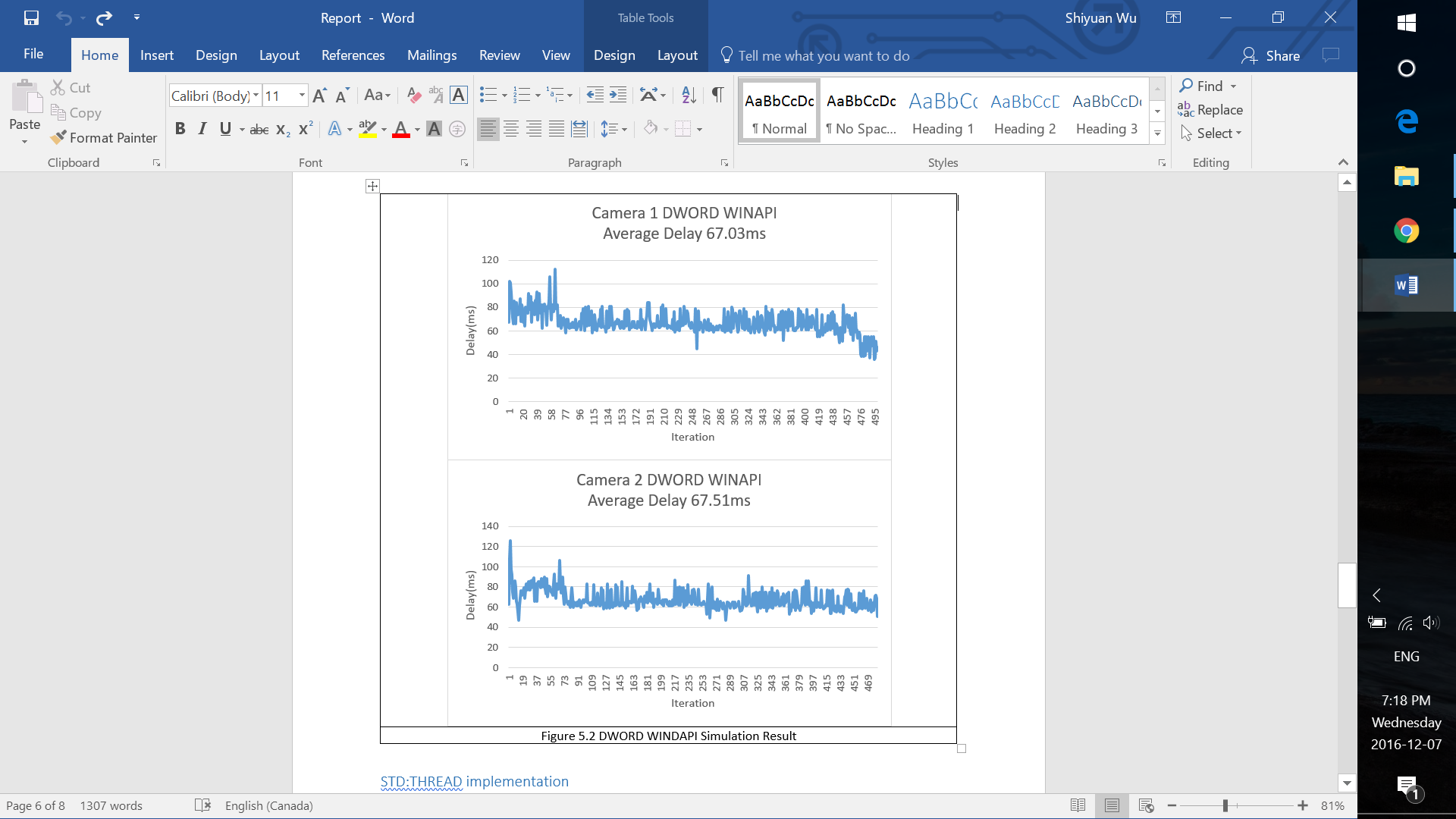
*Fig. 12: Single threaded simulation result*

To summarize, the three implementations’ flow chart is demonstrated in Fig. 11.

# Result Analysis

1. *Single Tread Implementation*

Simulation result is shown in Fig. 12. Camera 1 received average delay of 94.29ms and Camera 2 received average of 96.74ms. We noticed using same video clip as video source the overall iteration number is lower than the multi-threaded approaches. And we suggest it is related to the fact that all processing steps are done on one thread in sequence and we did not lock video feed frames during the camera frame stream feed. Thus, some frames are lost due to calculation delay.



*Fig. 13: DWORD WINAPI multithreaded simulation result*

## DWORD WINAPI Implementation

The simulation result is shown in Fig. 13. We can observer by parallelization on Camera 1 and Camera 2 image processing, the delay of Camera 1 is reduced to 67.03ms and that of Camera 2 is reduced to 67.51ms. With same video feed clip, we see the iteration count of multi-threaded approach is almost 500 which exceeds the number of single thread implementation.

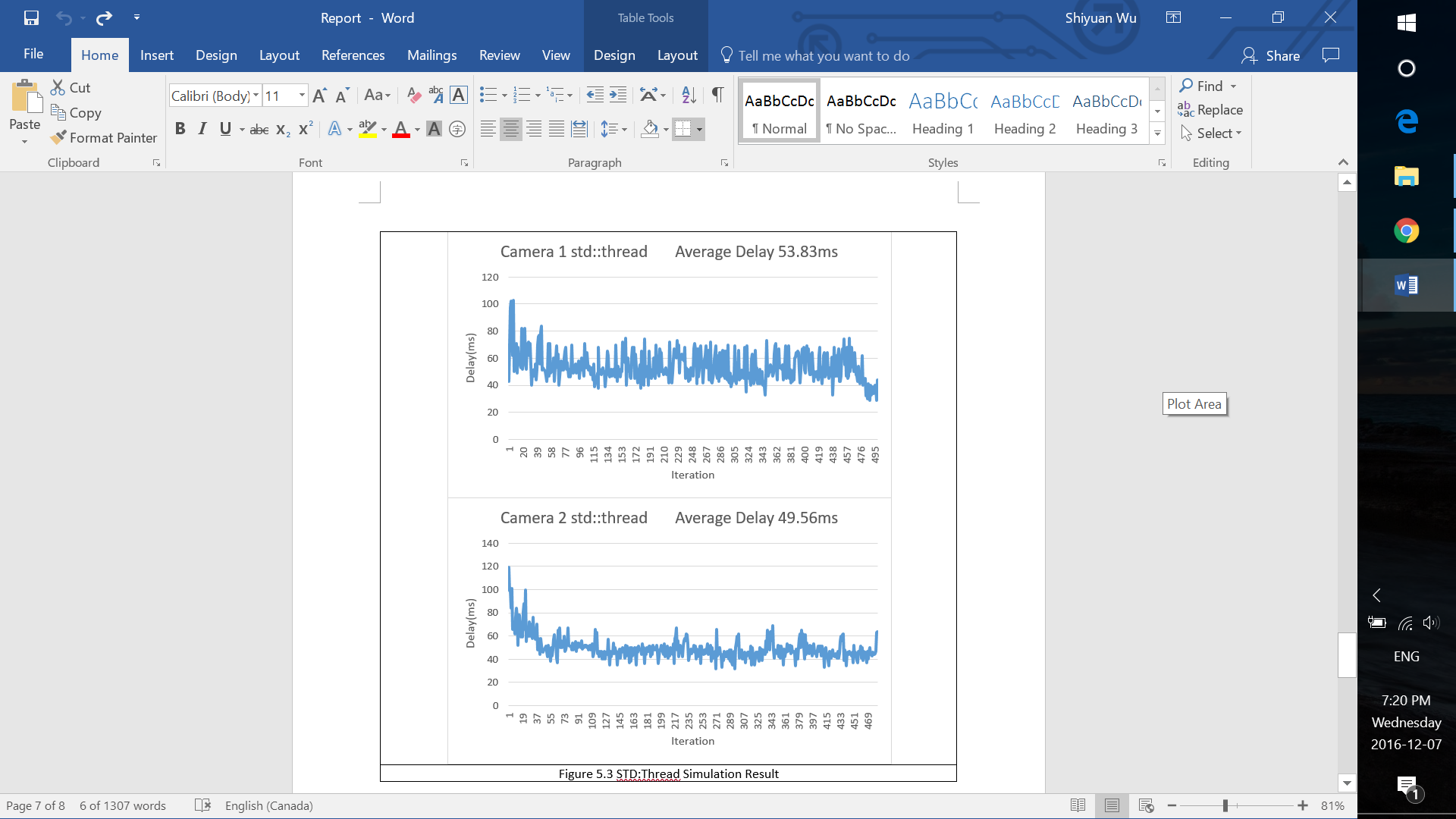
## Std:Thread implementation

The simulation result is shown is Fig. 14. In addition to parallelization on both cameras’ feed and processing, we further parallel image enhancement part on both cameras which are erode and dilate. The result shows processing delay is further reduced to 53.83ms in Camera 1 and 49.56ms in Camera 2.

# Related Work

As a well-known problem within the computer vision domain, typical models and methods for depth/disparity extraction in 2D-to-3D conversion has been investigated for decades. In this section, we present some existing techniques in depth extraction. One method is depth extraction from geometric constraints. In [10], a DepthFinder system is developed for in-vehicle surveillance to assist the driver when driving a car. This system can find the distances from objects to the vehicle through a monocular vision model using a camera that is mounted at the front or the side of a vehicle. With the prior knowledge of camera setting, such as focal length, moving distance of the vehicle is employed as the baseline for depth estimation. Another important method for depth estimation is focus/defocus analysis, also referred to as blur analysis. This method requires only a single camera to calculate the depth in a visual scene by modeling the effect of varying camera’s focal parameters on the image. Focal length can be obtained as depth based on the amount of blur in the image by employing inverse filtering [11]. However, blur is not available for general case, because expect focus length, blur can be produced by many factors, such as lens aberration, atmospheric interference, and motion.

# Conclusions and Future Work



*Fig. 14: Std::Thread multithreaded simulation result*

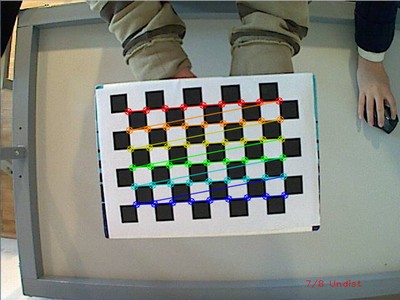
The object of this project was to explore how real-time stereo positioning can be implemented to take advantage of multi-core CPUs. Since the actual image processing operations of erode, dilate, and object tracking has already been efficiently implemented by OpenCV, our work focuses on parallelizing the feeding of data into the OpenCV APIs and post-processing of data output from OpenCV APIs into position coordinates. Our final 3-thread parallel implementation achieved a 1.4x

|  |
| --- |
|  |

speedup and our 5-thread (with 2 additional erode/ dilate sub-threads) parallel implementation achieved a 1.8x speedup. Given that the Intel CPU we used has 2 hardware cores, in theory we should only need 2 threads to achieve a 2x speedup. However, we see that we needed 5 threads to achieve even a 1.8x speedup due to the relative imbalance of the workload between the heavier image processing threads and lighter position calculation thread. This imbalance is tied to the nature of the stereo positioning algorithm. Rather than try to tune the performance of each thread, the easier and more effective solution is to spawn another thread to fill in the gaps in CPU time.

In the future, this project can be extended to implement camera calibration features which is useful for real-world positioning application where accuracy is important. The purpose of camera calibration is correct for image distortion caused by an imperfect camera lens and to determine the relation between the camera’s pixels and the distances in the real word [12]. Calibration is done by capturing a set of test images such as a checkerboard pattern shown in Fig. 15 into to OpenCV calibration functions and producing a re-projection error matrix to correct for distortion. Therefore, the image capture can re-use the existing parallelized code we have implemented but the correction calculation will need to be implemented and parallelized. This calibration is required for every unique setup of stereo camera hardware so therefore there is plenty of motivation to optimize it.

In addition, the project can also be extended to implement live camera feed capturing to closely model use-cases such as object detection for drones or autonomous vehicle systems [13]. From our experience in this project, 2 additional threads can be spawned to capture 2 camera sensor inputs to produce frames which can then be consumed by the threads calling OpenCV APIs. Depending on the latencies between camera capture and image processing, this may require some buffering of the image frames and semaphores between the producer and consumer threads to synchronize in a classic consumer-producer problem scheme. A stereo positioning system with this feature should be well suited to CPUs in the future given that the trend towards increasing CPU cores even in embedded applications would match well with the extra threads for camera capture.



*Fig. 15: A checkerboard pattern is fed as test images to calculate the correction matrix used to compensate for lens distortion [12]*

Lastly, the erode and dilate sub-threads are created and destroyed on every iteration inside the image processing threads. In the future, we can explore measuring the time of creating and joining threads relative to the entire processing time to evaluate whether the erode and dilate sub-threads may be better served by a thread pool. This would be entirely dependent on how quickly the operating system used can create threads and how quickly the hardware and OpenCV can do image processing*.*

# References

|  |  |
| --- | --- |
| [1] | National Instruments, "3D Imaging with NI LabVIEW," 24 Aug 2016. [Online]. Available: http://www.ni.com/white-paper/14103/en/#toc3. [Accessed 8 Dec 2016]. |
| [2] | University of Washington, "Stereo and 3D Vison," [Online]. Available: https://courses.cs.washington.edu/courses/cse455/09wi/Lects/lect16.pdf. [Accessed 8 Dec 2016]. |
| [3] | G. a. S. G.Saravanan, "Real Time Implementation of RGB to HSV/HSI/HSL and Its Reverse Color Space Models," in *International Conference on Communication and Signal Processing*, India, 2016. |
| [4] | S. Fernando, "Color Detection & Object Tracking," 1 Sep 2010. [Online]. Available: http://opencv-srf.blogspot.ca/2010/09/object-detection-using-color-seperation.html. [Accessed 8 Dec 2016]. |
| [5] | Code Laboratories, "CL- About CL Eye Platform - Code Laboratories," [Online]. Available: https://codelaboratories.com/products/eye/driver/. [Accessed 4 Dec 2916]. |
| [6] | K. Hounslow, "Tutorial: Real-Time Object Tracking Using Opencv," 11 Mar 2013. [Online]. Available: https://www.youtube.com/watch?v=bSeFrPrqZ2A. [Accessed 4 Dec 2015]. |
| [7] | C. Dahms, "Opencv 3 Windows 10 Installation Tutorial - Part 1 - C++," 1 Jan 2016. [Online]. Available: https://www.youtube.com/watch?v=7SM5OD2pZKY. [Accessed 4 Dec 2016]. |
| [8] | OpenCV, "Eroding And Dilating — Opencv 2.4.13.1 Documentation," 10 Oct 2016. [Online]. Available: http://docs.opencv.org/2.4/doc/tutorials/imgproc/erosion\_dilatation/erosion\_dilatation.html. [Accessed 4 Dec 2016]. |
| [9] | Stack Overflow, "C++11 std::thread vs windows CreateThread," 10 Aug 2015. [Online]. Available: http://stackoverflow.com/questions/25221247/c11-stdthread-vs-windows-createthread. [Accessed 4 Dec 2016]. |
| [10] | Y. Murphey and J. Chen, "A real-time depth detection system using monocular vision," *Proc. SSGRR Conf.,* 2000. |
| [11] | J. Ens and P. Lawrence, "An investigation of methods for determining depth from focus," *IEEE Trans. Pattern Anal. Mach. Intell.,* Vols. vol. 15, no.2, p. 97–108. |
| [12] | OpenCV, "Camera calibration With OpenCV," 27 Apr 2016. [Online]. Available: http://docs.opencv.org/2.4/doc/tutorials/calib3d/camera\_calibration/camera\_calibration.html. [Accessed 5 Nov 2016]. |
| [13] | MIT Technology Review, "Tesla Announces New Sensors and Puts the Brakes on Autopilot," 20 Oct 2016. [Online]. Available: https://www.technologyreview.com/s/602703/tesla-announces-new-sensors-and-puts-the-brakes-on-autopilot/. [Accessed 5 Nov 2016]. |
| [14] | OpenCV, "Depth Map From Stereo Images," 18 Dec 2015. [Online]. Available: http://docs.opencv.org/3.1.0/dd/d53/tutorial\_py\_depthmap.html. [Accessed 4 Dec 2016]. |
| [15] | University of Utah, "Morphological Operators," 23 Feb 2012. [Online]. Available: http://www.coe.utah.edu/~cs4640/slides/Lecture11.pdf. [Accessed 9 Dec 2016]. |

1. [↑](#footnote-ref-1)