Aalto University
School of Science
Master's Programme in Computer, Communication and Information Sciences

Tuomas Savolainen

Fast scale and illumination invariant method for region labeling

Master's Thesis
Espoo, December 23, 2018

Supervisor: Assistant Professor Juho Kannala
Advisor: Assistant Professor Juho Kannala

The document can be stored and made available to the public on the open internet pages of Aalto University. All other rights are reserved.

Aalto University
School of Science

Aalto University
School of Science
Master's Programme in Computer, Communication and Information Sciences

ABSTRACT OF MASTER'S THESIS

Author	Tuomas Savolainen		
Title	Fast scale and illumination invariant method for region labeling		
Major	Computer Science	**Code**	SCI3042
Supervisor	Assistant Professor Juho Kannala		
Advisor	Assistant Professor Juho Kannala		
Date	December 23, 2018	**Pages**	vi + 59

This work describes how to find 3D objects in 2D images. The images may contain various illumination conditions and backgrounds. Furthermore the distance and the rotation of the camera with respect to the object can be arbitrary. The method described in this work provides a way to reduce computation time of the 3D object localization problem by searching only from the regions of the image that include a combination of the most common colors of the object. The accuracy and speed of the implementation is tested on images taken under various illuminations and backgrounds.

Keywords	GPU, parallel programming, machine vision, inertial measurement
Language	English

Aalto-yliopisto
Perustieteiden korkeakoulu
Tietotekniikan maisteriohjelma

DIPLOMITYÖN TIIVISTELMÄ

Tekijä	Tuomas Savolainen		
Työn nimi	Nopea mittakaava- ja valaistusinvariantti metodi alueiden luokitteluun		
Pääaine	Tietotekniikka	**Koodi**	SCI3042
Valvoja	Apulaisprofessori Juho Kannala		
Ohjaaja	Apulaisprofessori Juho Kannala		
Päiväys	23. joulukuuta 2018	**Sivumäärä**	vi + 59

Tämä työ kuvailee miten kolmiulotteisia esineitä voi löytää kaksiulotteisista kuvista. Kuvat voivat sisältää vaihtelevia valaistusolosuhteita ja taustoja. Lisäksi kameran etäisyys ja avaruuskulma suhteessa esineeseen on mielivaltainen. Tässä työssä esitetty menetelmä antaa tavan vähentää kolmiulotteisen esineen löytämisen laskenta-aikaa etsimällä ainoastaan niistä kohdista, joissa on yhdistelmä esineen yleisimpiä värejä. Menetelmän tarkkuus ja nopeus on testattu kuvilla, jotka on otettu erilaisilla valaistuksilla ja taustoilla.

Asiasanat	näytönohjain, rinnakkaisohjelmointi, konenäkö, liikkeen mittaus
Kieli	Englanti

Acknowledgements

I would like to thank all my teachers, friends and family for your continuous support during my studies. Without your help and advice any off this would not have been possible.

Espoo, December 23, 2018

Tuomas Savolainen

Contents

1 Introduction **1**

2 Background **3**

2.1 Light and color . . . 3
2.2 Global illumination . . . 3
2.3 Pinhole camera model and projective space . . . 6
2.4 Frequency space and image compression . . . 8
2.5 Boundary functions and gradient in 2D . . . 10
2.6 Region connectivity . . . 12
2.7 Computation model and processor architecture . . . 13
2.8 Parallel processing and memory . . . 14
2.9 Related work . . . 15

3 Methods **17**

3.1 Defining use case . . . 17
3.2 Collecting test data . . . 19
3.3 Inertial Measurement Unit calibration . . . 20
3.4 Inertial Measurement Unit testing . . . 22
3.5 Setting accuracy and speed targets . . . 23

4 Implementation **25**

4.1 Overview . . . 25
4.2 Transforming colors . . . 26
4.3 Finding borders . . . 26
4.4 Connecting regions . . . 26

4.5 Filtering results 27

5 Results **35**

5.1 Detection accuracy 35

5.2 Processing speed 36

6 Discussion **41**

7 Summary **43**

A Code samples **50**

A.1 Find regions 50

A.2 Normalise colors 52

A.3 Remove edges 52

A.4 Make positive 53

A.5 Find red pixels 54

A.6 Find white pixels 55

A.7 Find red regions 55

A.8 Find white regions 56

A.9 Capture IMU 57

A.10 Capture camera 58

Chapter 1

Introduction

Guiding a person or a robot through an unknown terrain requires simultaneous localization and mapping of the environment. Only recently have commercially available digital cameras, graphical processing units (GPUs) and inertial measurement units (IMUs) reached required functionality, performance and price point for consumer market. Drones, self driving cars and smart phones pave the way for a smarter future. The commercial and academic interest in building context aware devices by combining data with machine learning from cameras, IMUs, lasers and other sensors has grown significantly in the last decade. Enabling developers to create real life applications with all these new devices requires extensive background work on the underlying technologies.

As the manufacturing process reaches the known limits of the quantum physics, we can't simply shrink components to gain performance improvements according to Moore's law. The short term solution is to increase the core count with current state of the art silicon manufacturing and on the software side focus more on parallel algorithms. To address the issues related in building a system capable of real time sensing we focus on a specific, but common problem in visual odometry: tracking objects in time.

Recently M. Rad (2018) and Tremblay et al. (2018) have demonstrated how to determine distance and orientation of arbitrary 3D object in cluttered environments. Our work focuses on reducing the time complexity of the two

most time consuming operations in the state-of-the-art methods. First we propose a method for filtering the image based on the normalized color information: this reduces the search space from complete image to collection of smaller images. Second we investigate the use of IMU for estimating the camera orientation between frames, this should speed up object recognition by restricting the possible orientations of the tracked object with respect to the camera.

We begin by briefly explaining how the adaptability of human visual system is able to interpret shape under various conditions and how this adaptability relates to existing algorithms in digital cameras. Next we present briefly the main changes in computing architectures due to parallelization. We then discuss the difficulty of tracking based on inertial measurement units under general and restricted motion. Then we continue by demonstrating an example program and evaluate it on real world data. Finally we conclude with remarks on future directions and lessons learned during this study.

Chapter 2

Background

2.1 Light and color

Let us begin by exploring what light is and what we mean by color. Physically the visible light is electromagnetic radiation with wavelength in range 400 to 700 $*10^{-9}m$ Seppänen et al. (2005). Wavelengths above or below this range cannot be directly observed by humans, although they can be seen by birds and other animals Wilkie et al. (1998). The trichromatic color system of humans is based on neural responses to three main colors: red, green and blue, see figure 2.2. This sensitivity can be determined by simple test of subjective brightness of monochromatic light at certain wavelength vs white light (monochromasy test), see Trezona (1987). Natural light, or light emitted by the Sun is polychromatic i.e. it consists of multiple wavelengths as can be seen in figure 2.1.

2.2 Global illumination

In order to detect objects from images we need to consider how images are captured. Firstly the image depends on the type of camera used. In general a camera generates a two-dimensional projection of the three-dimensional world. Objects in 3D (scene) are projected to 2D surface (camera sensor). Instead of capturing coordinates (X, Y, Z) of the surface in the scene, a cam-

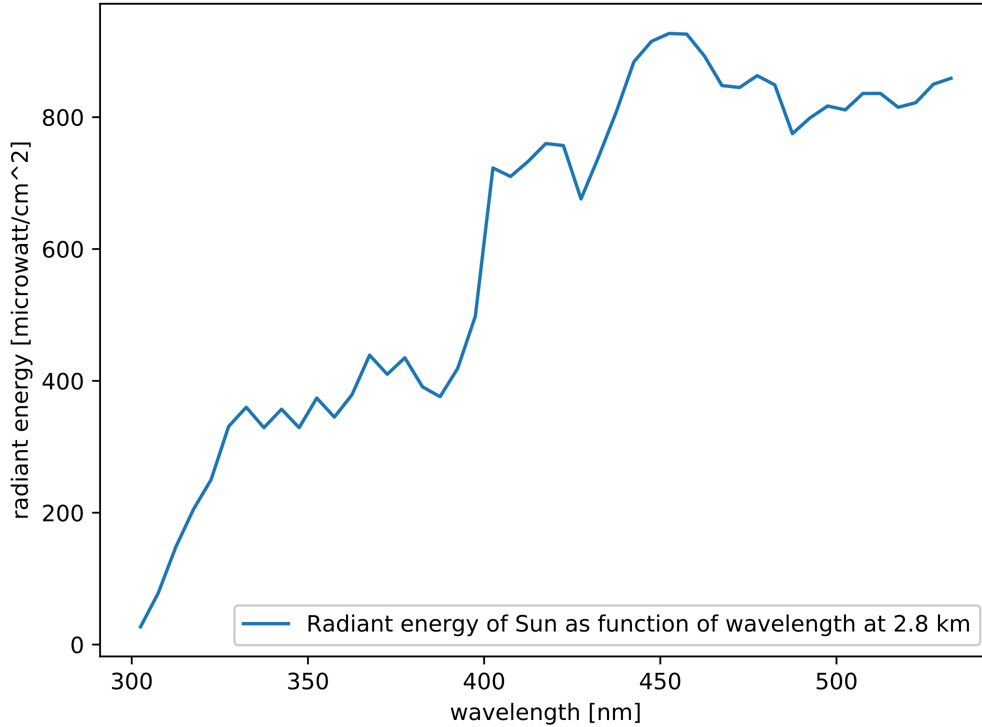

Figure 2.1: Sun's spectrum, see Stair et al. (1954) for details

era is only capable of capturing the light traveling through it lens to it's image sensor. The color of a point in a direction Ψ_o in a scene is determined by the sum of emitted and reflected intensity, for complete description of the rendering equation see Immel et al. (1986):

$$I(\Psi_o)_{total} = I(\Psi_o)_{emitted} + I(\Psi_o)_{reflected} \tag{2.1}$$

A consequence of the equation 2.1 is that the direction, shape and material of every surface in the scene affects every surface in the scene. Hence the name global illumination. In order to distinguish which points in the image correspond to an object that we are tracking we need to know a priori the emittance and reflectivity of the objects surface. In our case the object is a painted soda can which reflects paint colors red and white more than other colors. Another consequence of the equation 2.1 is that any movement will cause change in illumination, whether it's camera, light source or scene object

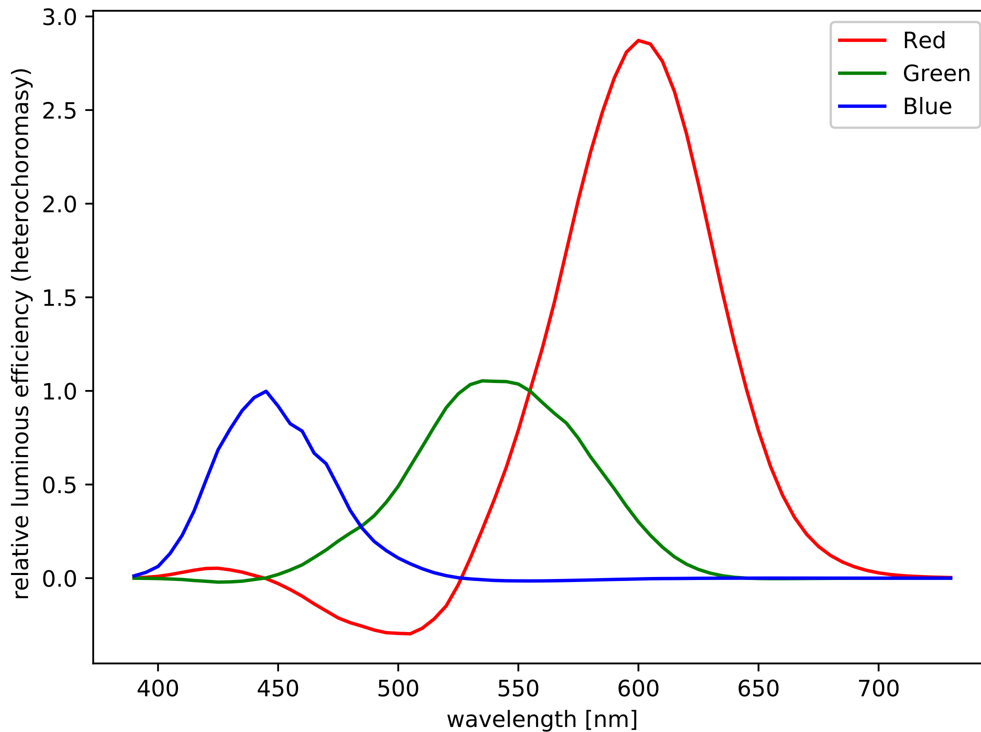

Figure 2.2: Human color vision, see Trezona (1987)

moving. Therefore the total intensity can vary greatly between different images. To solve this problem with variance of illumination intensity, the camera sensors use auto-white balance algorithm. The auto-white balance sets the cameras gain in analog to digital conversion (ADC) to match the brightness of the scene, in photography this process is called *dynamic range* and it is explained in detail in Jourlin and Pinoli (1995). The dynamic range algorithms solve questions such as: what is the range of intensity values, what is the required resolution (i.e. bit depth) of the values, and how fast is the light intensity changing in time? The theory and practice of ADC is beyond the scope of this work and the reader is directed to Pearson (2011) for gentle introduction and Arpaia et al. (1999) for more in-depth review. Most algorithms in digital cameras work on the "gray world" assumption: on average the world is gray Barnard et al. (2002). If we calculate the average intensity of the image, and set it as half of the range of the observed values,

we can assign intensity value to every pixel in the image. Since the intensities are captured in this relative manner, the *absolute* intensity values are hard to estimate from captured image without calibration emitters. If we want to track objects (soda cans) between images independent of whether they are in sunshine or shadow, we need to normalize the intensity between and within images. One way to normalize colors, used in our sample program is the following:

$$(R, G, B) \rightarrow (R/\alpha, G/\alpha, B/\alpha), \alpha = \begin{cases} 1 & \text{if } R = G = B = 0 \\ max(R, G, B) & \text{otherwise} \end{cases} \tag{2.2}$$

i.e. divide each component of the color triplet (R, G, B) by the maximum of the components and if the components are all zero leave it as is. For a review of different methods for color normalization see Healey and Slater (1994). In this normalized color space we can now define red:

$$Red(r, g, b) = \begin{cases} 1 & \text{if } (r = 1) \wedge (r > g \cdot 2) \wedge (r > b \cdot 2) \\ 0 & \text{otherwise} \end{cases} \tag{2.3}$$

and similarly white:

$$White(r, g, b) = \begin{cases} 1 & \text{if } (r = 1) \wedge (g > 0.6) \wedge (b > 0.6) \wedge (r > g) \wedge (r > b) \wedge \\ & (g < b + 0.3) \wedge (b < g + 0.3) \\ 0 & \text{otherwise} \end{cases} \tag{2.4}$$

The use of normalized color spaces enables us to create two sets of boundary functions: one defined by colors and one defined by changes in light intensity. There exists many ways to combine these local color descriptors with local shape descriptors, but even the simplest color information can improve object recognition results, see Fernandez-Maloigne et al. (2013).

2.3 Pinhole camera model and projective space

The pinhole camera model is the simplest projective camera model. In pinhole camera all rays travel from the scene through one point in front of the

camera and get projected onto the cameras sensor. The projection matrix P describes how world coordinates $[X, Y, Z]$ are projected to image coordinates $[x, y]$. Notice that the projection in general is not one to one mapping. In fact only the closest object along the ray from the camera is visible. The camera produces only finite sample of the illumination in the scene:

$$\{x \in [-x_0, x_0] \, | x_0 \in \mathbb{Z}_+\}, \{y \in [-y_0, y_0] \, | y_0 \in \mathbb{Z}_+\} \tag{2.5}$$

,where $2x_0$ is the width and $2y_o$ the height of the camera sensor in pixels. The right handed coordinate system's origin is in the lower left corner of the camera sensor, with the point (x_y, y_o) located at center of the the image respectively. The Projective matrix is camera intrinsic matrix K multiplied by extrinsic matrix E

$$P = KE \tag{2.6}$$

In order to avoid nonlinear equations in projective transformations homogeneous coordinates are used to form a system of linear equations. A point in $\mathbb{R}^3$ can be written with homogeneous coordinates by adding one dimension:

$$\left\{(x, y, z, h), | \; h \neq 0, (x, y, z) \in \mathbf{E^3}\right\} \tag{2.7}$$

For simplicity $h = 1$ is usually preferred.

The intrinsic camera matrix defined according to Sturm et al. (2011) assuming square pixels is:

$$K = \begin{bmatrix} f & 0 & x_o \\ 0 & f & y_o \\ 0 & 0 & 1 \end{bmatrix} \tag{2.8}$$

, where the point (x_y, y_o) is located at center of the image as before and f is the focal length of the camera. The extrinsic matrix can be written with inverse of rotation matrix R and translation vector C describing the camera position and rotation in world coordinates.

$$E = [R \mid t] = \begin{bmatrix} R^T & -R^T\overline{C} \\ 0 & 1 \end{bmatrix} \tag{2.9}$$

The Tait–Bryan chained rotation matrix can be defined with rotations along x-axis, y-axis and z-axis as:

$$R = R_x R_y R_z = \begin{bmatrix} 1 & 0 & 0 \\ 0 & cos(\phi) & -sin(\phi) \\ 0 & sin(\phi) & cos(\phi) \end{bmatrix} \begin{bmatrix} cos(\theta) & 0 & sin(\theta) \\ 0 & 1 & 0 \\ -sin(\theta) & 0 & cos(\theta) \end{bmatrix} \begin{bmatrix} cos(\psi) & -sin(\psi) & 0 \\ sin(\psi) & cos(\psi) & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{2.10}$$

We can then get the screen coordinates (x, y) from projected coordinates (x', y', w) by:

$$\begin{bmatrix} x' \\ y' \\ w \end{bmatrix} = KE \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} \Rightarrow \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x'/w \\ y'/w \end{bmatrix}, w \neq 0 \tag{2.11}$$

2.4 Frequency space and image compression

Fourier analysis is a common tool in signal analysis for transforming 1D signal from time space to frequency space. Signals can be decomposed to a sum of orthogonal components in higher dimensions in a similar manner. In image analysis we are specifically interested in discrete two dimensional Fourier transform, defined according to Burger and Burge (2016) as:

$$G(m, n) = \frac{1}{\sqrt{MN}} \sum_{u=0}^{M-1} \sum_{v=0}^{N-1} g(u, v) e^{i2\pi \frac{mu}{M}} e^{i2\pi \frac{nv}{N}} \tag{2.12}$$

, where $g(u, v)$ is the intensity at $row = u, column = v$, M is the number of rows and N is the number of columns. Related and highly useful transform, discrete cosine transform is used in JPEG images according to Wallace (1991) as:

$$G(m, n) = C(u)C(v) \left[\sum_{x=0}^{7} \sum_{y=0}^{7} g(u, v) cos(\frac{(2x + 1)u\pi}{16}) cos(\frac{(2x + 1)v\pi}{16}) \right] \tag{2.13}$$

,where

$$C(u), C(v) = \begin{cases} 1/\sqrt{2} & \text{if } u, v = 0 \\ 1 & \text{otherwise} \end{cases} \tag{2.14}$$

According to Wallace (1991) this 8x8 block compression concentrates most of the signal to lower spatial frequencies. Thus JPEG acts as a low-pass filter on the image. The use of JPEG has three benefits:

1. Most camera pipelines output JPEG's at no additional cost or delay
2. JPEG decoders are considerably faster than PNG decoders for the same resolution
3. JPEG compression acts as a low-pass filter and smooths the image. For reference see figures 2.3 and 2.4.

Figure 2.3: Original png image

Figure 2.4: The difference of png and jpeg image computed in frequency space and projected back to image space. Notice how the parts of the image with high frequency content such as edges undergo most change.

2.5 Boundary functions and gradient in 2D

In order to track objects we need some way of distinguishing them from the background. If we consider each color channel (red,green,blue) separately, we can calculate how fast each color changes in both vertical and horizontal directions simply by calculating a gradient. The gradient tells us how fast the color is decreasing or increasing. Assuming that the background has different color as the object in front the color changes rapidly on the boundaries. Correspondingly the color should change slowly inside a homogeneously colored object. Since we don't know the actual background a priori we can't assign a threshold value to the boundaries between the object and the background. On the other hand we know that the values should change slowly inside our

object. And given that all image segmentation methods try to cluster the pixels of the image in spatial domain, as pointed out by Pal and Pal (1993), we should try to remove small discontinuities inside the object by removing edges, see appendix A.3. Another way to view this is that we wish to remove high frequency noise from the image and in order to do it we need to calculate the location of this noise with the gradient. The Prewit gradient operators used in our sample program are defined according to Haralick (1987) as:

$$vertical = 1/6 \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix}, horizontal = 1/6 \begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix} \tag{2.15}$$

The vertical borders are obtained by applying the convolution operation on image's red, green and blue channels separately, the calculations for red channel are shown:

$$\frac{1}{6}\begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix} * \begin{bmatrix} r_{0,0} & \cdots & r_{0,j-1} \\ \vdots & \ddots & \vdots \\ r_{i-1,0} & \cdots & r_{i-1,j-1} \end{bmatrix} = \begin{bmatrix} h_{1,1} & \cdots & h_{1,j-2} \\ \vdots & \ddots & \vdots \\ h_{i-2,1} & \cdots & h_{i-2,j-2} \end{bmatrix} = \mathbf{G_{redX}} \tag{2.16}$$

Similarly for horizontal borders:

$$\frac{1}{6}\begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix} * \begin{bmatrix} r_{0,0} & \cdots & r_{0,j-1} \\ \vdots & \ddots & \vdots \\ r_{i-1,0} & \cdots & r_{i-1,j-1} \end{bmatrix} = \begin{bmatrix} v_{1,1} & \cdots & v_{1,j-2} \\ \vdots & \ddots & \vdots \\ v_{i-2,1} & \cdots & v_{i-2,j-2} \end{bmatrix} = \mathbf{G_{redY}} \tag{2.17}$$

The magnitude of the gradient is simply:

$$\mathbf{G} = \sqrt{\mathbf{G^2_{redX}} + \mathbf{G^2_{redY}}} = \begin{bmatrix} \sqrt{h^2_{1,1} + v^2_{1,1}} & \cdots & \sqrt{h^2_{1,j-2} + v^2_{1,j-2}} \\ \vdots & \ddots & \vdots \\ \sqrt{h^2_{i-2,1} + v^2_{i-2,1}} & \cdots & \sqrt{h^2_{i-2,j-2} + v^2_{i-2,j-2}} \end{bmatrix} \tag{2.18}$$

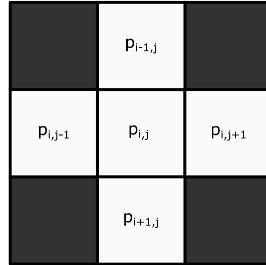

Figure 2.5: Pixel $p_{i,j}$ and it's neighbors

2.6 Region connectivity

Using the thresholds of red and white in equations 2.3 and 2.4 we can construct two binary images B_r and B_w for red and white pixels respectively. Given a binary image B we can start to label different regions. We wish to create symbol image S from B in the following manner:
Define region R_k as a set with label k, so that pixel $p_{i,j}$ belongs to set R_k if the pixel in symbol image S has value k:

$$\{p_{i,j} \in R_k | S(p_{i,j}) = k\} \tag{2.19}$$

further require that all regions are disjoint sets, since no pixel can belong to two regions:

$$R_k \cap R_m = \{\} \tag{2.20}$$

Within any region all it's pixels are connected to each other. The four-connectivity is defined in the following way: take a pixel $p_{i,j}$ in a binary image. If the pixel value is zero (0) it's does not belong to any region, if it is one (1) it is part of a region. If the pixel $p_{i,j}$ has non-zero pixel above ($p_{i-1,j}$), below ($p_{i+1,j}$), left ($p_{i,j-1}$) or right ($p_{i,j+1}$) it is connected to that pixel, see figure 2.5. Two pixels that are connected have the same label and they belong to the same region. We a use simple one scan connected labeling technique to create a Symbol image S from a binary image B. The method is similar to AbuBaker et al. (2007).

2.7 Computation model and processor architecture

For over fifty years after the invention of the integrated circuit, in the year 1947, the number of transistors per square meter increased according to Moore's law, see Schaller (1997). By the end of the 90's the researchers noticed however that the memory speed was lagging behind CPU speed, see Wulf and McKee (1995). This phenomenon was aptly named *memory wall*: the new instructions can't be red into the processor as fast as it consumes them, thus creating a barrier for achieving peak performance. The superscalar or vector processing was proven to increase computation throughput in the super computers, but the adoption of instruction level parallelism was not widespread outside scientific computation community, see Rau and Fisher (1993). Despite the fact that the manufacturing process is reaching the electron wall limit in the near future see Zhirnov et al. (2003), the number of transistors per square meter has been keeping up with Moore's law very accurately. The power density is described by the following formula, see Bergman et al. (2008):

$$P = C\rho f V^2 \tag{2.21}$$

where: P is power density, C is total capacitance, f is the clock frequency and V the is voltage respectively. Requiring higher clock frequencies also requires higher operating voltages to enable faster charge and discharge of capacitance, thus high clock-rates increase the power density dramatically, see Brodtkorb et al. (2010). Therefore modern architectures feature lower voltage levels and smaller frequencies, but higher core counts, see Bergman et al. (2008). At the same time as the vector processing capabilities of the CPUs have increased the GPUs have become a major platform for general purpose computing. Some companies have gone one step further and started creating custom hardware such as Tensor Processing Unit for efficient 8-bit integer matrix multiply, see Jouppi et al. (2018). Because of the memory wall problem the most import thing in computing remains keeping all processors fed with new instructions, see Bauer et al. (2011).

On a modern platform the total execution time of the program is determined by the serial code's ratio to the parallel code according to Amdahl's law, see Amdahl (1967). Transition from the serial code to the parallel code requires knowledge of the parallel architectures and algorithms. The gain in performance comes at a cost: in order to get the most performance per watt out of the new computing platforms the programmer needs to be aware of the low level implementation details.

2.8 Parallel processing and memory

In order to determine the labels of the image at the same rate as the images are captured we need to adapt the program to use the parallel computing platform. When reading from and writing to the memory we need to consider two things: first what is the bandwidth of the transaction (in MB/s) and two what is the delay from data transforms (in ms). For example reading an image file from the Hard Disk Drive (HDD) to the Random Access Memory (RAM) as an array of colors is limited by the bandwidth of HDD read speed and has a delay associated with the conversion of the packed image format to an array of bytes. There exists many ways to improve the disk IO-speed among them using Redundant Array of the Independent Disks (RAID), flash based memory such as Solid State Drives (SSD), hidden caches etc. which are explained in detail by Micheloni et al. (2013). The choice of programming language has also a significant impact on the performance. Interpreted languages such as Python are great for prototyping and testing. Unfortunately the dynamic resource allocation can become a major bottleneck in a parallel program. The convenient abstractions of Python hide away the details of memory management at the cost of memory and speed. The problem becomes even more profound when using GPUs that require explicit memory transfers with proper alignment for achieving peak performance. We used Python with SciPy library for writing the initial version of the program, see Jones et al. (2001–) for details on SciPy. After achieving reasonable accuracy we then continued by rewriting the entire program in C++ with Simple Fast Multimedia Library (SFML), see Haller and Hansson (2013) for further

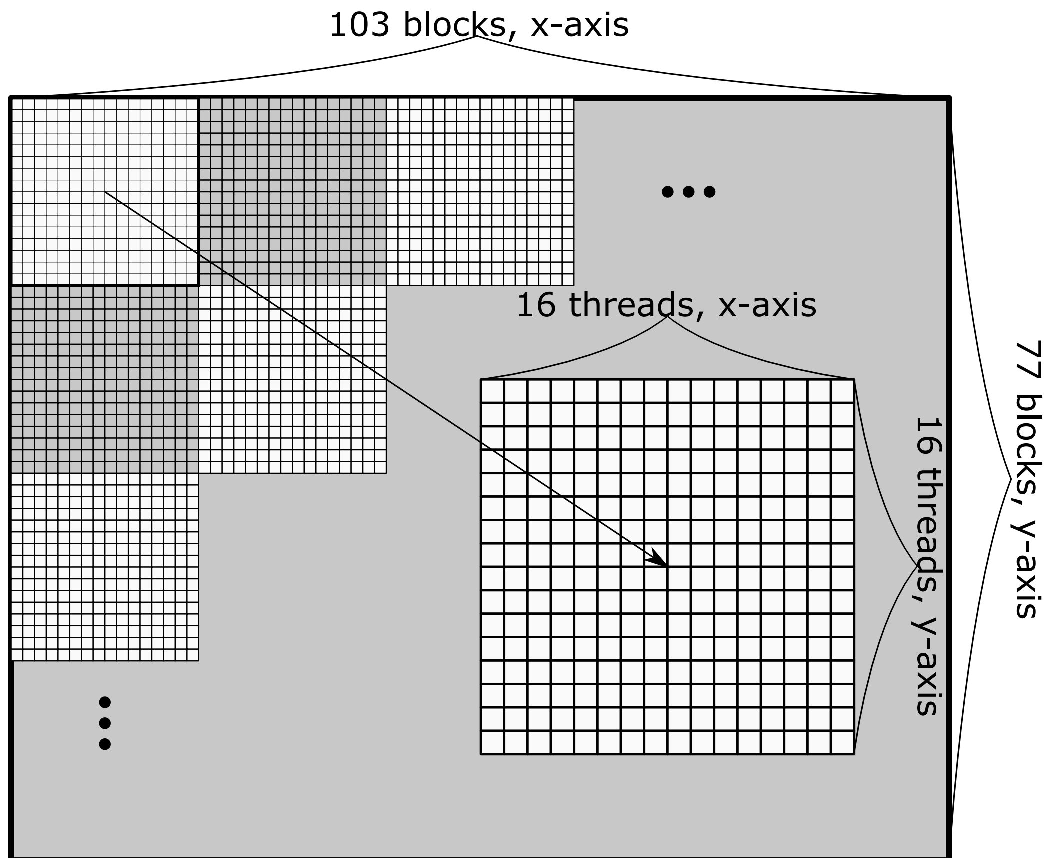

Figure 2.6: Compute grid consists of 103 blocks along x-axis and 77 blocks along y-axis. Each block consists of 16 threads in x-direction and 16 threads in y-direction.

details on SFML. Finally we parallelized the program with CUDA and optimized it for throughput (fps). Most of the workload was done on the compute grid depicted in figure 2.6. The limits of the disk IO, CPU, memory and GPU were analyzed for the final version.

2.9 Related work

Visual odometry is used in many different fields. Robots use visual odometry to complement inertial tracking in simultaneous localization and mapping. Virtual reality and augmented reality applications rely extensively on track-

ing the movement of the user with respect to a static object. Recently the use of neural networks for image segmentation Shelhamer et al. (2017), object recognition Rad and Lepetit (2017) and finally 3D pose estimation M. Rad (2018) has been extensively studied. Building on the advances of neural networks for solving the subproblems it was demonstrated that tracking objects position and orientation with respect to monocular RGB camera is possible with reasonable accuracy. The state-of-the-art of method uses a combination of synthetic and photo-realistic data for training a deep neural network Tremblay et al. (2018). While Tremblay et al. (2018) achieves impressive geometrical accuracy it can't operate in real-time on high-resolution images. The most time consuming task in the pose estimation is according to Rad and Lepetit (2017) the segmentation of the image. Our work specifically tries to reduce the segmentation time by reducing the original large image to a set of small candidate images. Furthermore when only the camera is moving and not the tracked object itself the inertial data should provide a reasonable estimate for change in objects orientation with respect to the camera. This should lead to further time savings in the pose prediction.

Another closely related task to ours is tracking objects in long image sequences. In long image sequences it is desirable to both determine if the target is in the image and what is the 2D bounding box of the target. Valmadre et al. (2018) presents a collection of video sequences spanning a total of 14 hours with various backgrounds for evaluating long term tracking. Valmadre et al. (2018) define true positive as finding the bounding box which overlaps with the ground truth. Unlike Valmadre et al. (2018) our method however does not require initial bounding box for the object to be tracked, rather the algorithm creates multiple probable bounding boxes for each image. Combining our work, state of the art 3D pose estimation and temporal techniques from long term video tracking could provide means for real-time high-resolution tracking.

Chapter 3

Methods

3.1 Defining use case

There are many ways to create 3D reconstructions of a static environment. Most methods use a combination of laser distance measurements coupled with image data from digital cameras. When estimating a scene from the cameras alone the scene can be constructed up to a scaling factor. With or without lasers the location and orientation of the camera must be know with respect to the inertial frame of reference (world coordinates). The simplest and most laborious way is to directly measure the camera angles and locations using inclinometers and measuring tape. The second approach is to use automatic or annotated point correspondences between images in order to determine the camera pose and location. The more involved and computationally expensive operation is to fuse the data from both image and laser data and minimize the re-projection error from estimated projection matrices. Since Inertial Measurement Units (IMUs) are cheap and ubiquitous on modern devices, many algorithms use them in addition to lasers.

We present a method that uses a combination of calibration object, IMU, camera and GPU for fast visual odometry. The object we choose for calibration should be easily available, "distinguishable" from it's background and have relatively simple 3D-geometry. Simple soda can will work fine for demonstration. Given a center-point of a soda can it's borders can be de-

termined by gradient or sobel operators. With borders of a cylinder we can compute it's relative distance and orientation with respect to camera. If we know a priori the height of the can we can also determine the scale of the projective transform and thus also the distance to the can. In the general case of four or more 3D to 2D point correspondences the projective transform can be efficiently computed with perspective-n-point algorithm, see Lepetit et al. (2008). For further details on how to obtain the pose and distance in the case of a cylinder see Huang et al. (1996). Another way to obtain distance of the can with unknown height is to use the IMU's data.
From IMU (Invensense MPU9255) we get magnetic field, acceleration and rotation along x-axis, y-axis and z-axis. Taking the tailor series of the acceleration function near zero we get:

$$a(t) = a(0) + \frac{a'(0)}{1!}t + \frac{a''(0)}{2!}t^2 \dots \tag{3.1}$$

Integrating the first term:

$$\iint a(0)dt = \int (a(0)t + v(0))dt = \frac{1}{2}a(0)t^2 + v(0)t + s(0) \tag{3.2}$$

$$\iint_0^{\Delta t} a(0)dt = \frac{1}{2}a(0)(\Delta t)^2 + v(0)(\Delta t) + s(0) \tag{3.3}$$

And the higher order terms:

$$\iint (\frac{a'(0)}{1!}t + \frac{a''(0)}{2!}t^2 \dots)dt = \frac{1}{6}a'(0)t^3 + \frac{1}{12}a'(0)t^4 + \dots \tag{3.4}$$

If the sampling rate is high enough, the Δt is small and we can neglect the higher order terms:

$$\Delta t \ll 1 \Rightarrow \frac{1}{6}a'(0)(\Delta t)^3 + \frac{1}{12}a'(0)(\Delta t)^4 + \dots \approx 0 \tag{3.5}$$

Therefore the displacement vector is approximately:

$$\vec{d} = \frac{1}{2}\vec{a}(0)(\Delta t)^2 + \vec{v}(0)(\Delta t) + s(\vec{0}) \tag{3.6}$$

We could similarly directly integrate the angular velocity to get estimate of angular change.

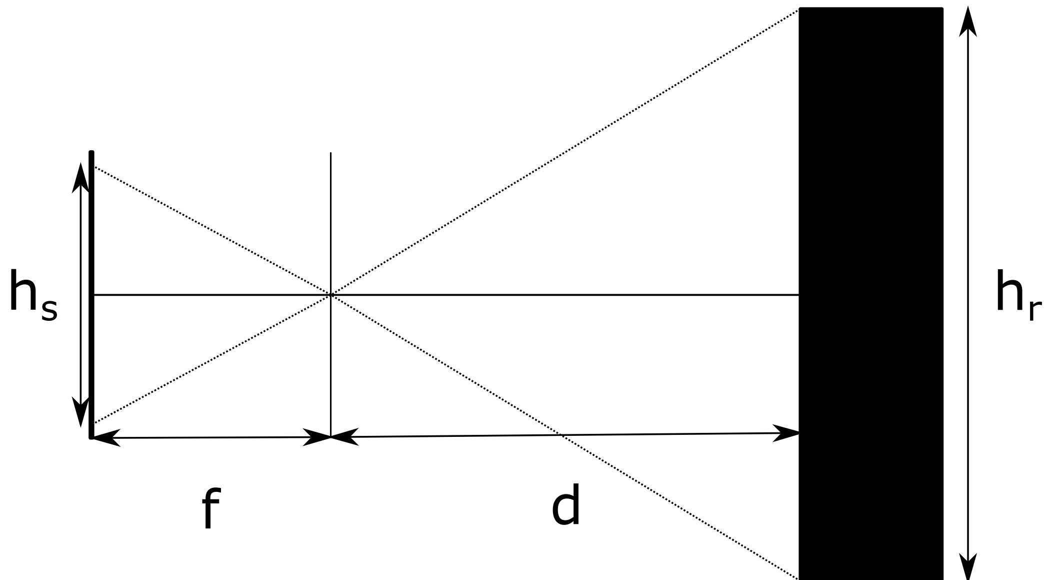

Figure 3.1: The distance d can be calculated from similar triangles

Using a calibrated camera without cropping and a known focal length we can get the distance to the soda can from similar triangles, see figure 3.1:

$$\frac{f}{h_s} = \frac{d}{h_r} \tag{3.7}$$

where: f is focal length, h_s is the height of the can on the sensor, d is the distance to the can and h_r is the foreknown height of the can. Since in general the can is not on the camera axis and perpendicular to it, we need to correct the formula for perspective distortion.

3.2 Collecting test data

The initial program was created in Python using a 284 images of soda cans with various orientations and backgrounds. Once the program was finalized it was tested on 90 out of sample images collected on a different camera. The 90 test images were captured with Sony IMX219 8-mega-pixel camera on Raspberry Pi 3B without cropping using 1640 (*width*) x 1232 (*height*) resolution. The images were taken by first using auto-white-balance to determine exposure settings and then taking 10 images with the same setting

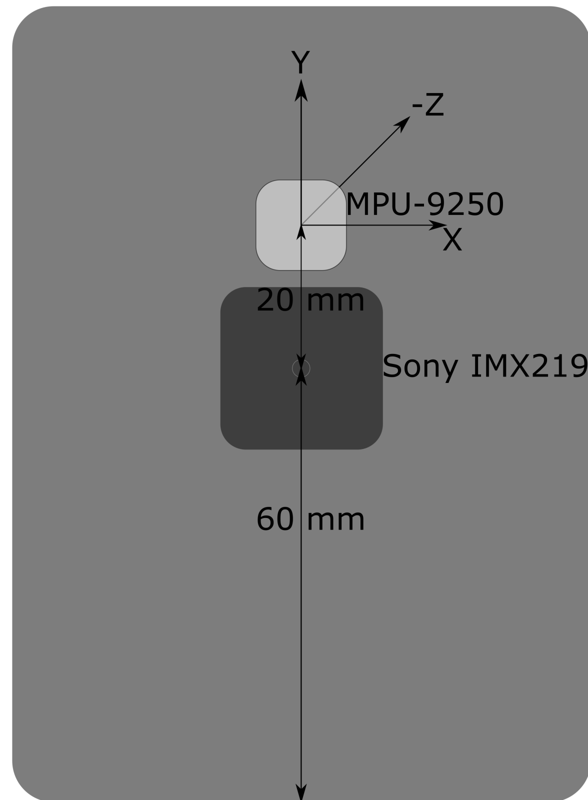

Figure 3.2: IMU and camera setup with camera coordinate frame

with simultaneous IMU capture see appendix A.9 and A.10.

3.3 Inertial Measurement Unit calibration

The dynamics of the camera movement in the scene are described by differential equations. If we attach the camera to a system with known dynamics such as a specific car or a simple pendulum, we know before hand the approximate shape of these equations from Newtonian physics. If we knew exactly the initial conditions i.e. the camera pose and location at $t = 0$ and the dynamics of the system we could also define exactly the trajectory of the camera in time. In practice we always have some error in initial conditions, model of dynamics and the inertial measurements. The purpose of IMU calibration is to minimize the initial and accumulated error of the camera trajectory.

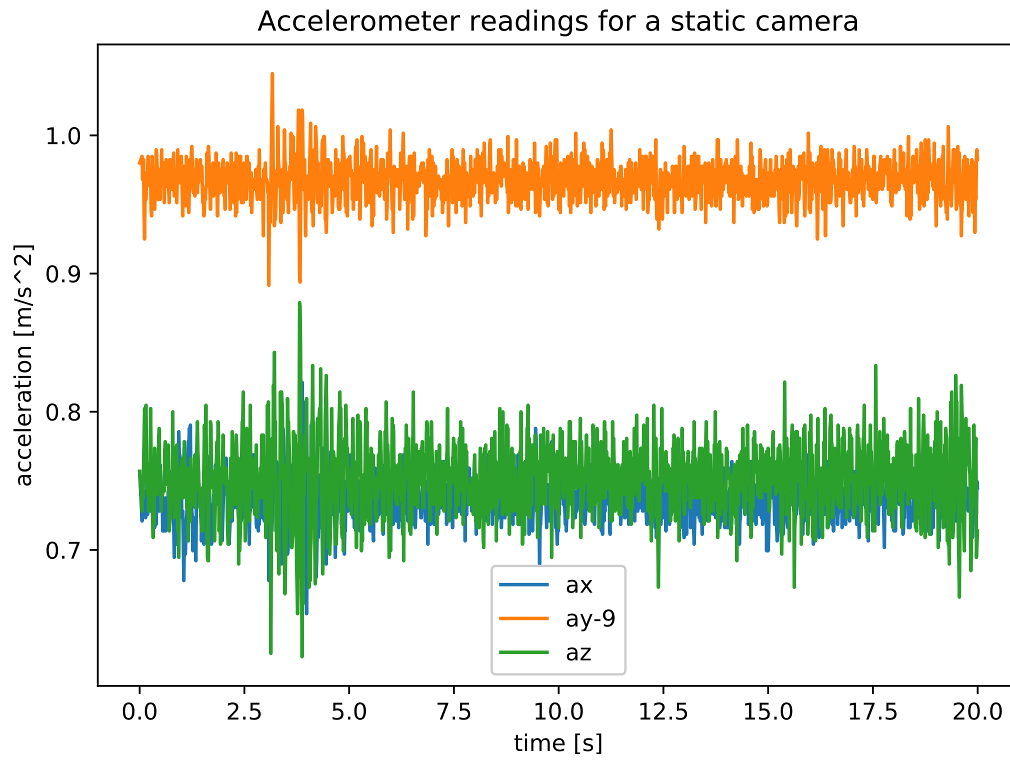

Figure 3.3: Static IMU and measurement noise

A static IMU standing on a ground will ideally show the direction and magnitude of the acceleration due to gravity. Errors due to thermal noise, quantification inaccuracies and other sources will introduce white noise to the measurements as can be seen in figure 3.3. For more detailed analysis of various error terms see Unsal and Demirbas (2012). Since acceleration is integrated twice to obtain trajectory, see equation 3.6, the random walk due to the inherent errors in acceleration measurement increases quadratically in time. In our experimental capture system the image capture rate, was unfortunately 0.33 fps and therefore it was not feasible to use IMU for actual filtering. We will instead settle for explaining the limitations and requirements for using IMU in future applications.

3.4 Inertial Measurement Unit testing

In order to test the IMU calibration we devised a test using a simple pendulum. The dynamics of simple pendulum with small initial angle are described by simple harmonic motion, see Giancoli (2016):

$$\theta(t) = \theta_0 cos(\omega t) \tag{3.8}$$

,where θ is angle as a function of time t, ω is angular velocity and θ_0 is initial displacement angle. This equation assumes that the pendulum is released from rest at a small angle θ_0 see figure 3.4. Therefore in the beginning t is zero and θ is θ_0. After time equal to $\frac{\pi}{2\omega}$ has passed the angle θ is zero and the angular velocity $\dot{\theta}$ is at it's maximum, see figure 3.5.

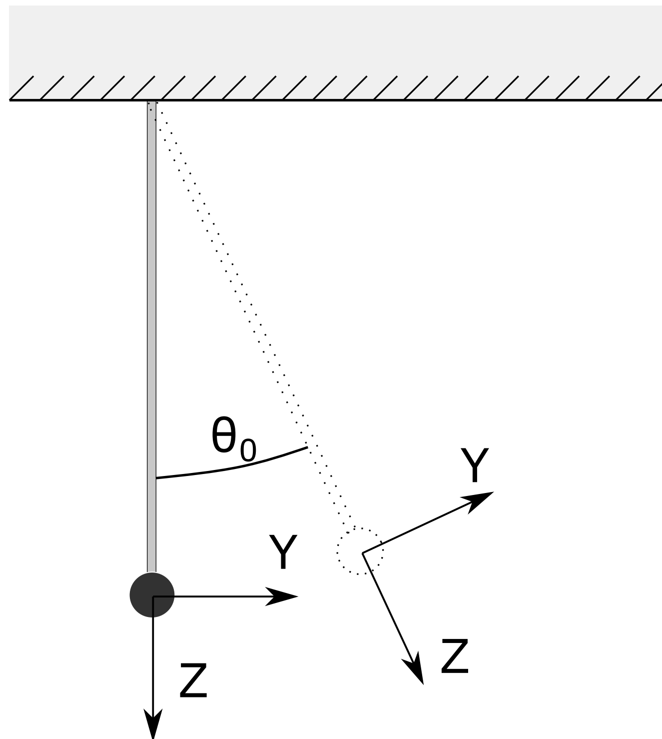

Figure 3.4: Simple pendulum at rest and with initial displacement angle $\theta_0(dotted)$

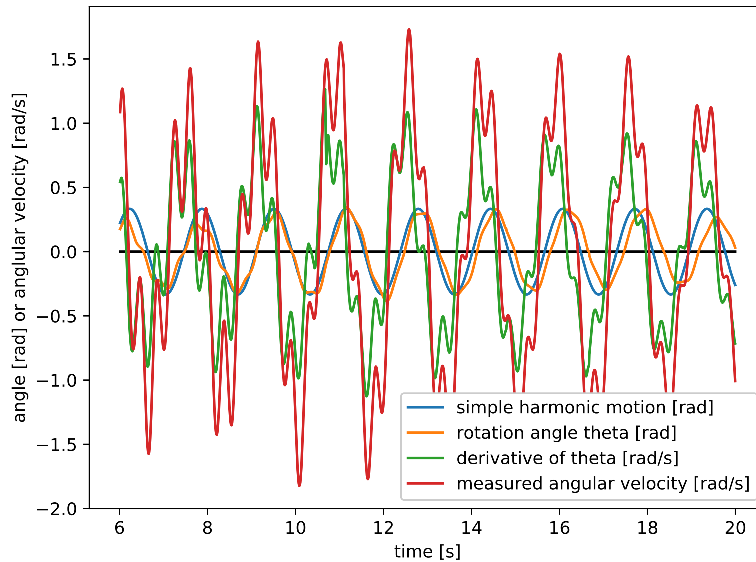

Figure 3.5: Simple pendulum measurements

3.5 Setting accuracy and speed targets

Modern mobile cameras are capable of capturing images with high frame rates and resolutions. As mentioned before, analyzing the video and sensor data in real time would enable sought after applications such as simultaneous localization and mapping (SLAM) and augmented reality (AR). The low price of mobile cameras has enabled adding them to cars, bikes, drones etc. Real time processing on these remote systems imposes several design challenges on power consumption and network connectivity. Currently the processing of visual data on the mobile device is still too resource intensive and it's better to resort to offloading it to cloud, see Noreikis et al. (2017).
However the OpenGLES and OpenCL are both including more and more general purpose programming interfaces while device manufacturers provide even more powerful mobile GPU's. In the near future it is possible to compute at least some of the visual computation on the device on it self. Therefore we decided to investigate how the current accelerometer and camera data could be processed on high performance CPUs and GPUs. In this work we present simple yet practical way of utilizing GPU, camera and IMU to ob-

tain first estimates of the tracked objects. The idea is to first prefilter the data on the edge device and then send it to the cloud for further analysis (image recognition, 3D construction etc.). In order to match the frame rate of the digital cameras we decided that the target system should run over 30 fps on modern multi-core desktop processor such as Intel® Core™ i7-7700 3.6Ghz with modern GPU such as Nvidia GeForce GTX 1070. The accuracy of detecting a target in a single image was set at $80 - 90\%$, with a few false positives.

Chapter 4

Implementation

4.1 Overview

The initial version of the program was developed using Python with SciPy library, see Jones et al. (2001–). This enabled fast visualization and testing of different approaches. When the target was found in 95% of images in the 284 'training' images we started to rewrite it in C++. The C++ program was an order of magnitude faster than the Python implementation, but still worked only at approximately 2fps. Next the most time consuming parts of the code were parallelized and rewritten using CUDA to enable GPU acceleration. With removal of unnecessary memory allocations and expensive transfers between CPU and GPU a frame rate of over 30 fps was achieved. Finally the parallel implementation was tested on 90 out of sample images.

The program takes a number of image files, finds the targets in them (soda cans) and writes the locations and the shapes of these cans to a file. The program flow is shown in figure 4.1. We start with an image containing a soda can. The first step is to normalize the image colors according to equation 2.2, for code sample see appendix A.2. Next we remove the edges obtained from the image using Prewit gradient operator defined in equation 2.15, for code sample see appendix A.3. After that we normalize the image again according to equation 2.2 and make all values positive, see appendix

A.4 . After that we generate two binary images: one for pixels that are red and another for white pixels see A.5 and A.6. The binary images are then scaled down by finding the continuous blocks in the red and white binary images, see appendix A.7 and A.8. The scaled down images are then labeled according to 4-way connectivity as defined in section 2.6, see appendix A.1 for code. If there is white regions that are near the red regions we have a match. Finally we calculate the midpoint and the size of the red region in pixels and save it to file.

4.2 Transforming colors

As described in the section 2.2 it is necessary to normalize colors in order to handle scenes with different illuminations. The image color normalization is defined in equation 2.2 at page 6. For reference see how input image 0 in 4.2 is normalized to 4.3 and similarly input image 1 in figure 4.4 is normalized to 4.5.

4.3 Finding borders

After removing the edges from the image (see flow chart in 4.1), we normalize the image again in order to distinguish between colors. For reference see how the images 0 and 1 are normalized in figures 4.6 and 4.7.

4.4 Connecting regions

Tracking of the object is based on the objects colors. The red and white pixels used in tracking the soda can are defined according to equations 2.3 and 2.4. If enough red or white pixels are close to each other they form red and white blocks. The masks used to compute these blocks are show in figure 4.12 for red blocks and in figure 4.13 for white blocks.

4.5 Filtering results

Theoretically the best way to determine the accuracy of the tracking would be to re-project the image of the target into the captured image and compare it pixels by pixel. Unfortunately this kind of technique has several challenges. Most notably determining the global illumination is hard and very resource intensive task, if we were to project the object into the scene we would also have to project the illumination of the scene on the object. Secondly, since the rotation of the target is arbitrary at least in one dimension (around it's vertical axis), comparing the label to all of the possible rotations would be time consuming. Thirdly the measured location of the object is just an approximation and even a small deviation from the ideal position in the scene will change the pixel values thus making the comparison hard. One possibility would be to track the borders of the target and compare them to the target. On the other hand *if we knew accurately* were the borders of the can are, we could directly compute it's location and orientation according to Huang et al. (1996).
Therefore we want to discuss the option of using the IMU for checking if the rate of change in region size and IMU readings are consistent in time. The IMU library (RTIMU, Richards-Tech) uses Spherical Linear Quaternion interpolation or SLERP for short, see Dam et al. (1998). According to SPORTILLO (2015) the RTQF fusion algorithm is used for pose estimation. In RTQF the pose is interpolated between the predicted state from gyroscope reading and measured state from gyroscope and accelerometer. A default value of 0.02 was used in interpolation, which means that most of the quaternion comes from the gyroscope reading. Because the magnetometer readings vary by location and time, see Courtillot and Le Mouel (1988) it is necessary to calibrate magnetometer.

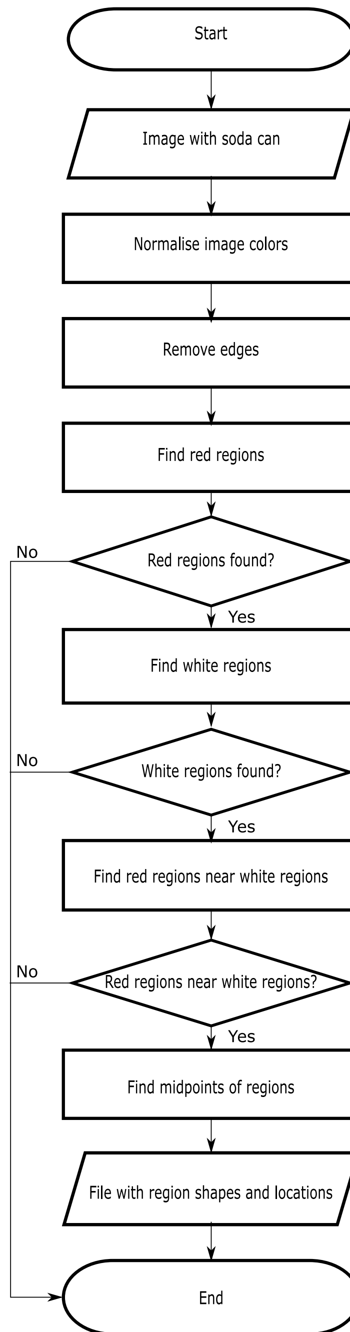

Figure 4.1: Program flow

Figure 4.2: Captured image 0

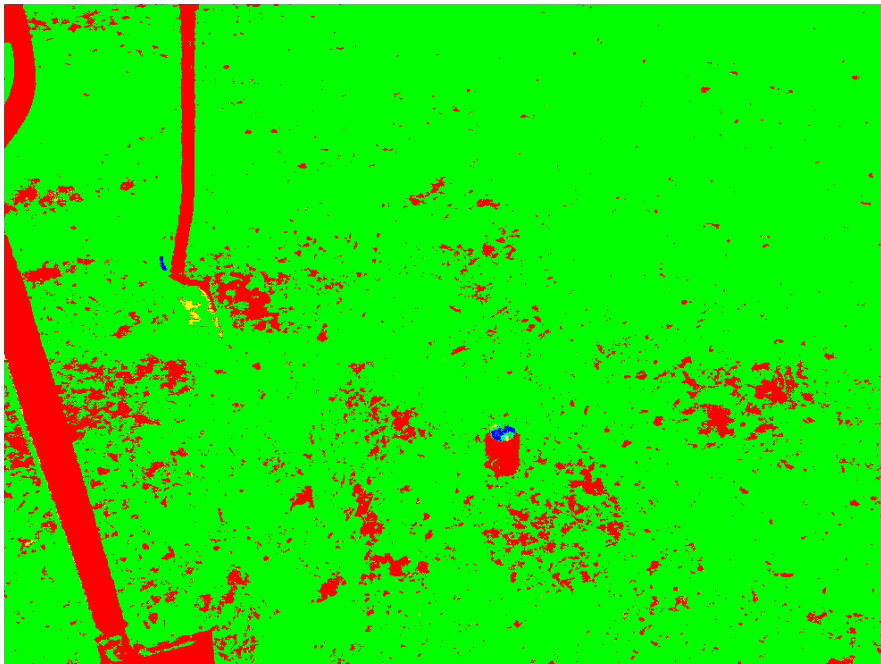

Figure 4.3: Normalized image 0

Figure 4.4: Captured image 1

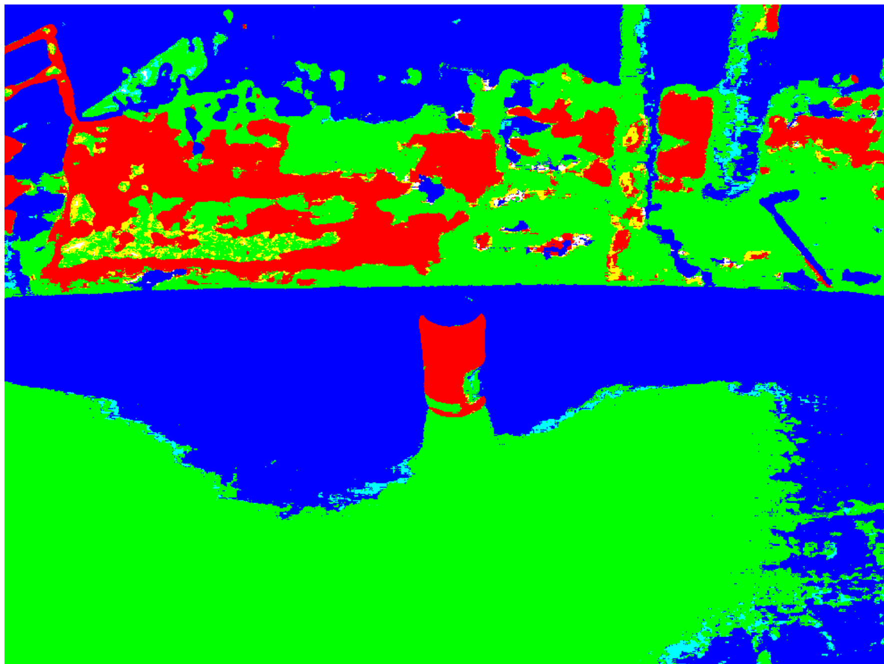

Figure 4.5: Normalized image 1

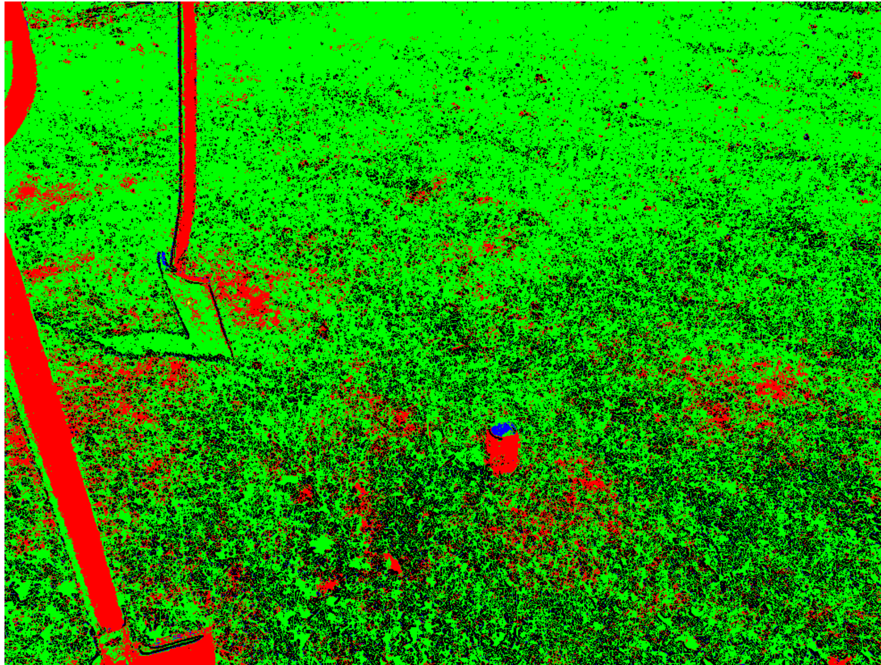

Figure 4.6: Edge removed and normalized image 0

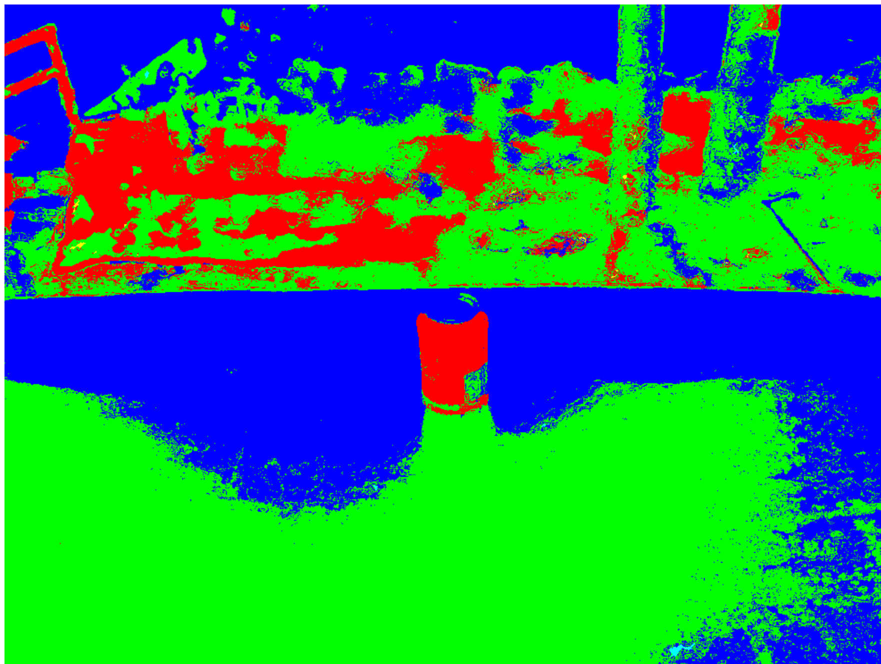

Figure 4.7: Edge removed and normalized image 1

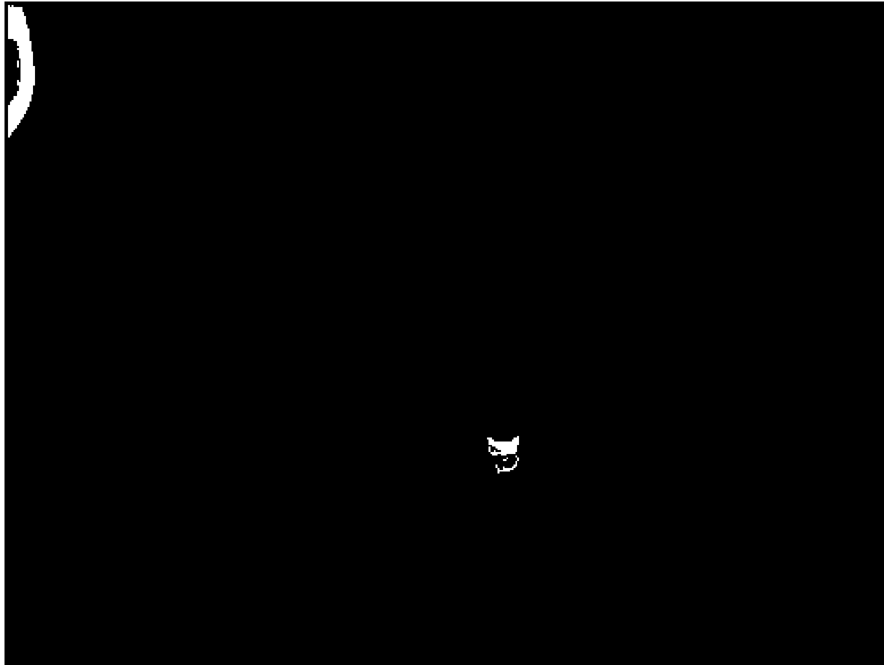

Figure 4.8: Red pixels of image 0

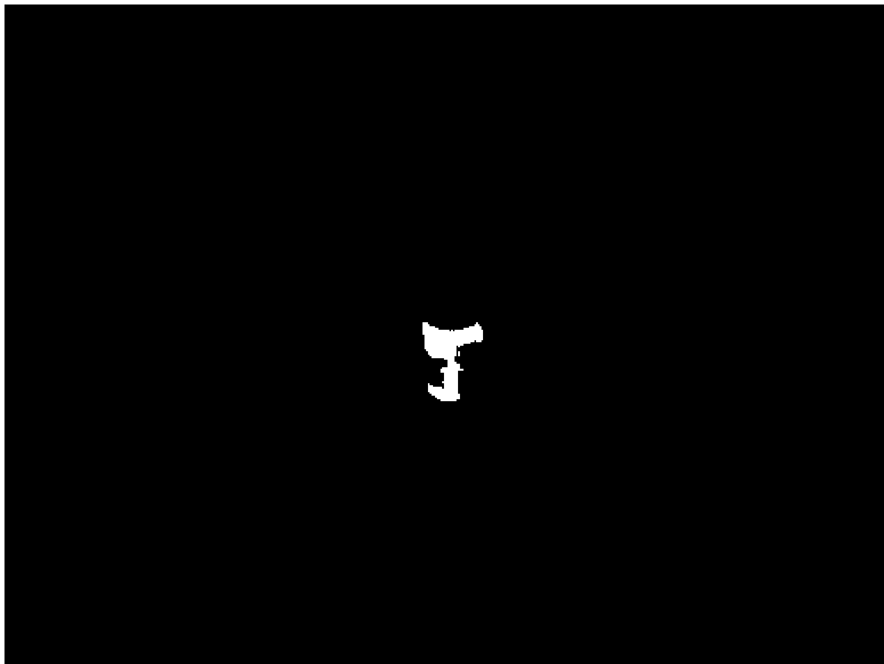

Figure 4.9: Red pixels of image 1

Figure 4.10: White pixels of image 0

Figure 4.11: White pixels of image 1

Figure 4.12: Red region

Figure 4.13: White region

Chapter 5

Results

5.1 Detection accuracy

The accuracy of the region labeling was tested on the ninety test images described in section 3.1. The algorithm found the target in 78 images out of the 90 test images. These 78 images contained also false positives due to other red regions in these images, see example image 2.3 and image with detected regions (marked with blue color) in 5.2. Observations: most false positives are present in the scenes where the soda can is surrounded with white and red patterns as expected see $wh0$ - $wh9$ in table 5.1. Tilting the can also has severe effect on detection accuracy as can be seen in $sf0$ - $sf9$ in table 5.1. Another key observation is that the reddish regions i.e. regions that have $R > G$ and $R > B$ are treated just as 'red' and thus different shades of red are indistinguishable from each other.
One possible improvement would be to consider the re-projection error ϵ from projecting planar soda can logo on the detected surfaces. Another simpler approach would be to detect ellipses and circles above the region resulting from the top of the can. Unfortunately even detecting ellipses of a arbitrary shape and size in a general image proves to be a non-trivial task. The brute-force approach of searching the whole Hough Transform space of elliptical space is also rather time consuming, see Davies (1989). Parallel version using GPU with compressed parameter space would be considerably faster,

Figure 5.1: The nine scenes used to take the 90 test images, 10 images were taken at each scene

but we didn't ourselves explore that option further, for reference see Ito et al. (2011).

5.2 Processing speed

The purpose of the preprocessing is reduction in the execution time of the final image processing pipeline, with negligible effect on the accuracy. Therefore, as we discussed in the section 2.8 it is necessary to do the preprocessing at rate of over 30 fps. We tested the performance of the program by running it 100 times in a loop. For a single process the average frame rate was calculated simply as the number of frames divided by the execution time. For multiple processes the workload was divided between the processes and some of the processes finished processing before others. In the case of multiple processes the execution time was calculated by the slowest process. We also

Figure 5.2: Detected regions of the image in figure 2.3, the regions are marked with blue color

compared the execution times when using uncompressed png file format and jpeg file format with compression setting at 80%. On a multi-core system, see table 5.2 the fps varied between 28 and 92 for 1 process and 8 processes respectively, see table 5.3 and figure 5.3.

Most of the time was spent on average in the sequential parts of the c++ code, as can be seen in figure 5.4. Loading image to the RAM amounted to considerable portion of the time budget even when using jpeg file format, some time savings could be achieved if the image stream from the camera were forwarded directly to the GPU and decoded there. Further time savings could be achieved if the image loading and the image processing were done in parallel.

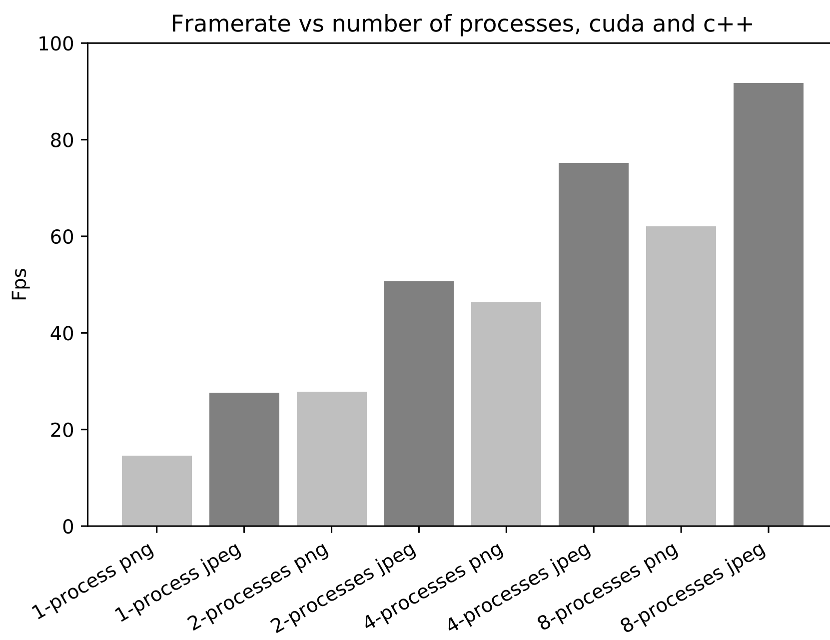

Figure 5.3: Increasing the number of processes to solve the problem increases fps, but makes it quickly IO-bound. All images are 1640 (width) x 1232 (height)

image	true	false	image	true	false	image	true	false
as0	1	2	du0	1	15	sf0	1	1
as1	1	6	du1	1	13	sf1	0	0
as2	1	5	du2	1	14	sf2	0	2
as3	1	5	du3	1	15	sf3	1	2
as4	1	6	du4	1	14	sf4	0	2
as5	1	6	du5	1	14	sf5	0	0
as6	1	6	du6	1	12	sf6	0	2
as7	1	11	du7	1	14	sf7	0	0
as8	1	11	du8	1	15	sf8	0	1
as9	1	11	du9	1	15	sf9	0	1
ca0	1	0	fu0	1	1	us0	1	0
ca1	1	0	fu1	1	1	us1	1	0
ca2	0	0	fu2	1	1	us2	1	0
ca3	1	0	fu3	1	1	us3	1	0
ca4	1	0	fu4	1	1	us4	1	1
ca5	1	0	fu5	1	1	us5	1	3
ca6	1	0	fu6	1	1	us6	1	1
ca7	1	0	fu7	1	1	us7	1	1
ca8	1	0	fu8	0	0	us8	1	1
ca9	1	0	fu9	0	0	us9	1	1
df0	1	14	ro0	1	11	wh0	1	21
df1	1	15	ro1	1	8	wh1	1	20
df2	1	14	ro2	1	8	wh2	1	21
df3	1	11	ro3	1	7	wh3	1	19
df4	1	11	ro4	1	7	wh4	1	19
df5	1	11	ro5	1	1	wh5	1	21
df6	1	11	ro6	1	0	wh6	1	21
df7	1	8	ro7	1	1	wh7	0	19
df8	1	10	ro8	1	2	wh8	1	19
df9	1	9	ro9	1	1	wh9	0	20

Table 5.1: True and false positives in 90 test images

CPU	Intel Core i7-7700, 3.6GHz 8-core
GPU	NVIDIA GeForce GTX 1070, VRAM 8191MB
RAM	HyperX Fury DDR4, 4 x 4GB 2.4GHz
HDD	Sandisk HyperX Savage SSD 480GB, 560/530 MB/s

Table 5.2: Desktop system configuration

number of processes	uncompressed png fps	compressed jpeg fps
1	14.5	27.6
2	27.8	50.6
4	46.3	75.2
8	62.0	91.7

Table 5.3: Program performance, all images are 1640 (width) x 1232 (height)

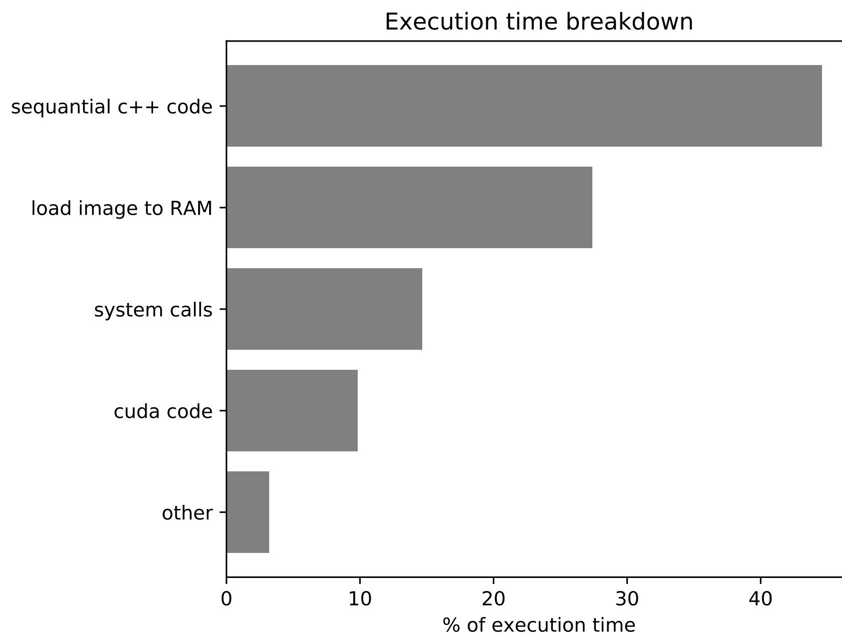

Figure 5.4: Execution time breakdown of the program with jpeg file format

Chapter 6

Discussion

In this work we described a simple method for tracking colored objects. The most important feature of the preprocessing in our work was to reduce the computational load of the tracking in order to enable real-time functionality. While our method produced quite many false positives in environments where the background was similarly colored it managed to find the correct targets relatively often. Thus the speed of the preprocessing should be adequate compensation for it's inaccuracy.

The use of IMU for additional constraints on the camera trajectory were also explored. Unfortunately we could not obtain uncompressed images from the camera's video port because of the access restrictions to the camera's frame buffer. Instead we had to revert to starting two process at the same time: one taking uncropped still frames at maximum frame rate and another recording the IMU readings. This invalidated our assumption of small time elapsed between captured images and rendered the use of IMU data mostly useless in the context of non-restricted movement. It would be interesting to test IMU filtering with high frame rate cameras.

We did not pay much attention to the shape of the objects, since current methods using neural networks can accurately determine the pose of the object by comparing it to the projections of the object's 3D model, see M. Rad (2018). The recent work by Tremblay et al. (2018) has demonstrated that is possible to do near real-time recognition of multiple objects. As the cameras,

sensors and the mobile computing platforms continue to evolve there will be many interesting ways to solve the tracking and navigations problems.

Chapter 7

Summary

The theoretical and practical challenges of tracking objects with sensor fusion in real environments were presented in this thesis. We briefly explained the hardware and the software used to sense the physical environment. While mapping robustly the entire three dimensional environment in real-time remains one of the unsolved challenges, the new computing platforms, sensors and, to increasing extent, the machine learning are pushing the boundary of how the smart devices sense our surroundings.

By focusing on object tracking and it's implementation with connected component labeling we showcased how some of the resource intensive visual tracking load can be reduced by preprocessing the images on the graphics cards, thus leaving more time for the post-processing such as recognition and interpretation.

The possibility to utilize the inertial measurement unit was also investigated. It was noted that using IMU still requires constrained environments and proper calibration to be practically useful. In the future we would like to explore which parts of the environment sensing could be learned instead of being explicitly set, for instance could the program learn the proper normalization of the colors on-the-fly? Finally, could you combine our method for image preprocessing with the state-of-the-art 3D pose detection into a real-time tracker suitable for mobile devices, robotics and augmented reality?

References

A. AbuBaker, R. Qahwaji, S. Ipson, and M. Saleh. One scan connected component labeling technique. *Signal Processing and Communications, 2007. ICSPC 2007. IEEE International Conference on*, pages 1283–1286. IEEE, 2007. DOI: 10.1109/ICSPC.2007.4728561.

G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. *Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485. ACM, 1967. DOI: 10.1145/1465482.1465560.

P. Arpaia, P. Daponte, and L. Michaeli. Influence of the architecture on adc error modeling. *IEEE Transactions on Instrumentation and Measurement*, 48(5):956–966, 1999. DOI: 10.1109/19.799654.

K. Barnard, V. Cardei, and B. Funt. A comparison of computational color constancy algorithms. i: Methodology and experiments with synthesized data. *IEEE transactions on Image Processing*, 11(9):972–984, 2002. DOI: 10.1109/TIP.2002.802531.

M. Bauer, H. Cook, and B. Khailany. Cudadma: optimizing gpu memory bandwidth via warp specialization. *Proceedings of 2011 international conference for high performance computing, networking, storage and analysis*, page 12. ACM, 2011. ISBN: 978-1-4503-0771-0.

K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, K. Hill, J. Hiller, et al. Exascale computing study:

Technology challenges in achieving exascale systems. Technical report, Defense Advanced Research Projects Agency Information Processing Techniques Office (DARPA IPTO), Tech. Rep, 2008. contract: FA8650-07-C-7724.

A. R. Brodtkorb, C. Dyken, T. R. Hagen, J. M. Hjelmervik, and O. O. Storaasli. State-of-the-art in heterogeneous computing. *Scientific Programming*, 18(1):1–33, 2010. DOI: 10.3233/SPR-2009-0296.

W. Burger and M. J. Burge. *Digital image processing: an algorithmic introduction using Java*. Springer, 2016. ISBN: 978-1-4471-6684-9.

V. Courtillot and J. L. Le Mouel. Time variations of the earth's magnetic field: from daily to secular. *Annual Review of Earth and Planetary Sciences*, 16(1):389–476, 1988. 10.1146/annurev.ea.16.050188.002133.

E. B. Dam, M. Koch, and M. Lillholm. Quaternions, interpolation and animation. Technical report, Department of Computer Science University of Copenhagen, 1998. DIKU-TR-98/5.

E. Davies. Finding ellipses using the generalised hough transform. *Pattern Recognition Letters*, 9(2):87–96, 1989. DOI: 10.1016/0167-8655(89)90041-X.

C. Fernandez-Maloigne, F. Robert-Inacio, and L. Macaire. Digital color imaging, 2013. ISBN: 9781118561966.

D. C. Giancoli. *Physics: principles with applications*. Pearson, 2016. ISBN: 9781292066851.

J. Haller and H. V. Hansson. *SFML Game Development*. Packt Publishing Ltd, 2013. ISBN: 9781849696845.

R. M. Haralick. Digital step edges from zero crossing of second directional derivatives. In: M. A. Fischler and Firschein (eds.), *Readings in Computer Vision*, pages 216–226. Elsevier, 1987. ISBN 978-0-08-051581-6.

G. Healey and D. Slater. Global color constancy: recognition of objects by use of illumination-invariant properties of color distributions. *JOSA A*, 11 (11):3003–3010, 1994. DOI:10.1364/JOSAA.11.003003.

J.-B. Huang, Z. Chen, T.-L. Chia, et al. Pose determination of a cylinder using reprojection transformation. *Pattern recognition letters*, 17(10): 1089–1099, 1996. DOI: 10.1016/0167-8655(96)00061-X.

D. S. Immel, M. F. Cohen, and D. P. Greenberg. A radiosity method for non-diffuse environments. *Acm Siggraph Computer Graphics*, volume 20, pages 133–142. ACM, 1986. DOI: 10.1145/15922.15901.

Y. Ito, K. Ogawa, and K. Nakano. Fast ellipse detection algorithm using hough transform on the gpu. *Networking and Computing (ICNC), 2011 Second International Conference on*, pages 313–319. IEEE, 2011. DOI: 10.1109/ICNC.2011.61.

E. Jones, T. Oliphant, P. Peterson, et al. SciPy: Open source scientific tools for Python, 2001–. URL `http://www.scipy.org/`. [Online; accessed March 24, 2018].

N. Jouppi, C. Young, N. Patil, and D. Patterson. Motivation for and evaluation of the first tensor processing unit. *IEEE Micro*, 38(3):10–19, 2018. DOI: 10.1109/MM.2018.032271057.

M. Jourlin and J.-C. Pinoli. Image dynamic range enhancement and stabilization in the context of the logarithmic image processing model. *Signal processing*, 41(2):225–237, 1995. DOI: 10.1016/0165-1684(94)00102-6.

V. Lepetit, F. Moreno-Noguer, and P. Fua. Epnp: An accurate o(n) solution to the pnp problem. *International Journal of Computer Vision*, 81(2): 155–166, 2008. ISSN 1573-1405. DOI: 10.1007/s11263-008-0152-6.

V. L. M. Rad, M. Oberweger. Feature mapping for learning fast and accurate 3d pose inference from synthetic images. *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4663–4672. IEEE, 2018. arXiv preprint arXiv:1712.03904v2.

R. Micheloni, A. Marelli, and K. Eshghi. *Inside solid state drives (SSDs)*. Springer, 2013. ISBN: 978-94-007-5145-3.

M. Noreikis, Y. Xiao, and A. Ylä-Jääski. Seenav: Seamless and energy-efficient indoor navigation using augmented reality. *Proceedings of the on Thematic Workshops of ACM Multimedia 2017*, pages 186–193. ACM, 2017. DOI: 10.1145/3126686.3126733.

N. R. Pal and S. K. Pal. A review on image segmentation techniques. *Pattern recognition*, 26(9):1277–1294, 1993. DOI: 10.1016/0031-3203(93)90135-J.

C. Pearson. High-speed, analog-to-digital converter basics. Technical report, Texas Instruments, January 2011. Application Report: SLAA510.

M. Rad and V. Lepetit. Bb8: A scalable, accurate, robust to partial occlusion method for predicting the 3d poses of challenging objects without using depth. *2017 IEEE International Conference on Computer Vision (ICCV)*, pages 3848–3856. IEEE, 2017. DOI: 10.1109/ICCV.2017.413.

B. R. Rau and J. A. Fisher. Instruction-level parallel processing: history, overview, and perspective. *The Journal of Supercomputing*, 7(1-2):9–50, 1993. DOI: 10.1007/BF01205181.

R. R. Schaller. Moore's law: past, present and future. *IEEE spectrum*, 34 (6):52–59, 1997. DOI: 10.1109/6.591665.

R. Seppänen, M. Kervinen, I. Parkkila, L. Karkela, and P. Meriläinen. Maol-taulukot: Matematiikka, fysiikka, kemia. *Helsinki: Kustannusosakey-htiö Otava*, 2005. ISBN: 978-951-1-20607-1.

E. Shelhamer, J. Long, and T. Darrell. Fully convolutional networks for semantic segmentation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 39(4):640–651, 2017. DOI: 10.1109/TPAMI.2016.2572683.

D. SPORTILLO. Addressing the problem of interaction in fully immersive virtual environments: from raw sensor data to effective devices. Master's

thesis, Università di Pisa, Scuola Superiore Sant'Anna, Pisa, Italy, 2015. 108 pages.

R. Stair, R. G. Johnston, and T. C. Bagg. Spectral distribution of energy from the sun. *Journal of Research of the National Bureau of Standards*, 53(2):113–119, 1954. ISSN: 0160-1741.

P. Sturm, S. Ramalingam, J.-P. Tardif, S. Gasparini, J. Barreto, et al. Camera models and fundamental concepts used in geometric computer vision. *Foundations and Trends® in Computer Graphics and Vision*, 6(1-2):1–183, 2011. DOI: 10.1561/0600000023.

J. Tremblay, T. To, B. Sundaralingam, Y. Xiang, D. Fox, and S. Birchfield. Deep object pose estimation for semantic robotic grasping of household objects. *arXiv preprint arXiv:1809.10790*, 2018.

P. Trezona. Individual observer data for the 1955 stiles–burch 2° pilot investigation. *JOSA A*, 4(4):769–782, 1987. DOI: 10.1364/JOSAA.4.000769.

D. Unsal and K. Demirbas. Estimation of deterministic and stochastic imu error parameters. *Position Location and Navigation Symposium (PLANS), 2012 IEEE/ION*, pages 862–868. IEEE, 2012. DOI: 10.1109/PLANS.2012.6236828.

J. Valmadre, L. Bertinetto, J. F. Henriques, R. Tao, A. Vedaldi, A. Smeulders, P. Torr, and E. Gavves. Long-term tracking in the wild: A benchmark. *arXiv preprint arXiv:1803.09502*, 2018.

G. K. Wallace. The jpeg still picture compression standard. *Communications of the ACM*, 34(4):30–44, 1991. DOI: 10.1145/103085.103089.

E. S. Wilkie, P. M. Vissers, D. Debipriya, J. W. Derip, K. J. Bowmaker, and M. D. Hunt. The molecular basis for uv vision in birds: spectral characteristics, cdna sequence and retinal localization of the uv-sensitive visual pigment of the budgerigar (melopsittacus undulatus). *Biochemical Journal*, 330(1):541–547, 1998. ISSN: 0264-6021.

W. A. Wulf and S. A. McKee. Hitting the memory wall: implications of the obvious. *ACM SIGARCH computer architecture news*, 23(1):20–24, 1995. DOI: 10.1145/216585.216588.

V. V. Zhirnov, R. K. Cavin, J. A. Hutchby, and G. I. Bourianoff. Limits to binary logic switch scaling-a gedanken model. *Proceedings of the IEEE*, 91 (11):1934–1939, 2003. DOI: 10.1109/JPROC.2003.818324.

Appendix A

Code samples

A.1 Find regions

```
std::vector<std::pair<uint32_t, uint32_t>> findr(uint32_t* regi)
{

for (int i = 0; i < N_QUARTERCOLS; ++i)
{
  regi[i] = 0;
}
//set left column to zero
for (int i = 0; i < N_QUARTERROWS; ++i)
{
  regi[i*N_QUARTERCOLS] = 0;
}
//set last row to zero
for (int i = 0; i < N_QUARTERCOLS; ++i)
{
  regi[N_QUARTERCOLS*(N_QUARTERROWS - 1) + i] = 0;
}
//set right column to zero
for (int i = 0; i <  N_QUARTERROWS; ++i)
{
  regi[i*N_QUARTERCOLS + (N_QUARTERCOLS - 1)] = 0;
}

uint32_t count = 0;
uint32_t name = 1;
std::vector<std::pair<uint32_t, uint32_t>> my_stack;
std::vector<std::pair<uint32_t, uint32_t>> sums;
bool found_region = false;
uint32_t stx = 0;
uint32_t sty = 0;
std::pair<uint32_t, uint32_t> location;
```

```
for (int i = 1; i < N_QUARTERROWS - 1; ++i)
{
  for (int j = 1; j < N_QUARTERCOLS - 1; ++j)
  {
    count = 0;
    if (regi[i*N_QUARTERCOLS + j] == 1)
    {
      name += 1;
      count = 1;
      regi[i*N_QUARTERCOLS + j] = name;
      my_stack.push_back(std::make_pair(i, j));
    }
    stx = i;
    sty = j;
    while (my_stack.size() > 0)
    {
      location = my_stack.back(); //get
      my_stack.pop_back(); //  and remove last element from stack
      stx = location.first;
      sty = location.second;
      //go right
      if (regi[stx*N_QUARTERCOLS + sty + 1] == 1)
      {
        regi[stx*N_QUARTERCOLS + sty + 1] = name;
        count += 1;
        my_stack.push_back(std::make_pair(stx, sty + 1));
      }
      //go up
      if (regi[(stx - 1)*N_QUARTERCOLS + sty] == 1)
      {
        regi[(stx - 1)*N_QUARTERCOLS + sty] = name;
        count += 1;
        my_stack.push_back(std::make_pair(stx - 1, sty));
      }
      //go left
      if (regi[stx*N_QUARTERCOLS + sty - 1] == 1)
      {
        regi[stx*N_QUARTERCOLS + sty - 1] = name;
        count += 1;
        my_stack.push_back(std::make_pair(stx, sty - 1));
      }
      //go down
      if (regi[(stx + 1)*N_QUARTERCOLS + sty] == 1)
      {
        regi[(stx + 1)*N_QUARTERCOLS + sty] = name;
        count += 1;
        my_stack.push_back(std::make_pair(stx + 1, sty));
      }
    }
    //end of region
    //save to sums
    if (count > 0)
    {
      sums.push_back(std::make_pair(name, count));
    }
  }
}

return sums;

}
```

A.2 Normalise colors

```
__global__ void
normalise2D(float* dest_img, float* source_img, int n_rows, int n_cols, size_t d_pitch, size_t s_pitch)
{

  int t_idx = blockIdx.x*blockDim.x + threadIdx.x;
  int t_idy = blockIdx.y*blockDim.y + threadIdx.y;

  if ((t_idx > 0) && (t_idx < N_COLS - 1) && (t_idy > 0) && (t_idy < N_ROWS - 1))
  {
    float4 *row_dest = (float4*)((char *)dest_img + t_idy * d_pitch);

    //source base address is equal to
    //offset in bytes from base address = (char *)
    //row number * row in bytes = t_idy * s_pitch
    //this is converted to float4 type for accessing elements with: (float4*)
    float4 *row_source = (float4*)((char *)source_img + t_idy * s_pitch);


    //get max value:
    float max_v = 0.0;
    max_v = fmaxf(fmaxf(row_source[t_idx].x, row_source[t_idx].y), row_source[t_idx].z);
    if (max_v <= 0.0)
    {
      max_v = 1.0;
    }


    row_dest[t_idx].x = row_source[t_idx].x / max_v;
    row_dest[t_idx].y = row_source[t_idx].y / max_v;
    row_dest[t_idx].z = row_source[t_idx].z / max_v;
    //keep alpha channel the same
    row_dest[t_idx].w = row_source[t_idx].w /255.0;

  }

}
```

A.3 Remove edges

```
__global__ void
removeWhiteBorders2D(float* dest_img, float* source_img, int n_rows, int n_cols, size_t d_pitch, size_t s_pitch)
{

  int t_idx = blockIdx.x*blockDim.x + threadIdx.x;
  int t_idy = blockIdx.y*blockDim.y + threadIdx.y;

  if ((t_idx > 0) && (t_idx < N_COLS - 1) && (t_idy > 0) && (t_idy < N_ROWS - 1))
  {
    float4 *row_dest = (float4*)((char *)dest_img + t_idy * d_pitch);

    //source base address is equal to
    //offset in bytes from base address = (char *)
    //row number * row in bytes = t_idy * s_pitch
    //this is converted to float4 type for accessing elements with: (float4*)
    float4 *row_source = (float4*)((char *)source_img + t_idy * s_pitch);

    //remember to reset alpha channel!

    float4 horizontal_e;
```

```
    horizontal_e.x = 0.0f;
    horizontal_e.y = 0.0f;
    horizontal_e.z = 0.0f;
    horizontal_e.w = 0.0f;

    row_source = (float4*)((char *)source_img + (t_idy-1) * s_pitch); // row above
    horizontal_e -= row_source[t_idx - 1];
    horizontal_e -= row_source[t_idx];
    horizontal_e -= row_source[t_idx + 1];


    row_source = (float4*)((char *)source_img + (t_idy + 1) * s_pitch); // row below
    horizontal_e += row_source[t_idx - 1];
    horizontal_e += row_source[t_idx];
    horizontal_e += row_source[t_idx + 1];

    horizontal_e = horizontal_e*0.16666666666f;

    float4 vertical_e;
    vertical_e.x = 0.0f;
    vertical_e.y = 0.0f;
    vertical_e.z = 0.0f;
    vertical_e.w = 0.0f;

    row_source = (float4*)((char *)source_img + (t_idy - 1) * s_pitch); // row above
    vertical_e -= row_source[t_idx - 1];
    vertical_e += row_source[t_idx + 1];
    row_source = (float4*)((char *)source_img + (t_idy) * s_pitch); // current row
    vertical_e -= row_source[t_idx - 1];
    vertical_e += row_source[t_idx + 1];
    row_source = (float4*)((char *)source_img + (t_idy + 1) * s_pitch); // row below
    vertical_e -= row_source[t_idx - 1];
    vertical_e += row_source[t_idx + 1];
    vertical_e = vertical_e*0.16666666666f;


    row_source = (float4*)((char *)source_img + (t_idy - 1) * s_pitch); // current row
    //-= sqrt(horizontal_e*horizontal_e + vertical_e*vertical_e);
    row_dest[t_idx].x = row_source[t_idx].x - hypotf(vertical_e.x, horizontal_e.x);
    row_dest[t_idx].y = row_source[t_idx].y - hypotf(vertical_e.y, horizontal_e.y);
    row_dest[t_idx].z = row_source[t_idx].z - hypotf(vertical_e.z, horizontal_e.z);
    row_dest[t_idx].w = 1.0;

  }

}
```

A.4 Make positive

```
__global__ void makePositive2D(float* dest_img, float* source_img, int n_rows, int n_cols, size_t d_pitch, size_t

  //this would have been nicer looking without manual loop unrolling
  int t_idx = blockIdx.x*blockDim.x + threadIdx.x;
  int t_idy = blockIdx.y*blockDim.y + threadIdx.y;

  if ((t_idx > 0) && (t_idx < N_COLS - 1) && (t_idy > 0) && (t_idy < N_ROWS - 1))
  {
    float4 *row_dest = (float4*)((char *)dest_img + t_idy * d_pitch);

    //source base address is equal to
    //offset in bytes from base address = (char *)
    //row number * row in bytes = t_idy * s_pitch
    //this is converted to float4 type for accessing elements with: (float4*)
    float4 *row_source = (float4*)((char *)source_img + t_idy * s_pitch);
```

```
    //remember to reset alpha channel!

    if (row_source[t_idx].x < 0.0)
    {
      row_dest[t_idx].x = 0.0;
    }
    else
    {
      row_dest[t_idx].x = row_source[t_idx].x;
    }

    if (row_source[t_idx].y < 0.0)
    {
      row_dest[t_idx].y = 0.0;
    }
    else
    {
      row_dest[t_idx].y = row_source[t_idx].y;
    }

    if (row_source[t_idx].z < 0.0)
    {
      row_dest[t_idx].z = 0.0;
    }
    else
    {
      row_dest[t_idx].z = row_source[t_idx].z;
    }

    row_dest[t_idx].w = 1.0;

  }

}
```

A.5 Find red pixels

```
__global__ void
findRed2D(float* dest_img, float* source_img, int n_rows, int n_cols, size_t d_pitch, size_t s_pitch)
{

  int t_idx = blockIdx.x*blockDim.x + threadIdx.x;
  int t_idy = blockIdx.y*blockDim.y + threadIdx.y;

  if ((t_idx > 0) && (t_idx < N_COLS - 1) && (t_idy > 0) && (t_idy < N_ROWS - 1))
  {
    float *row_dest = (float*)((char *)dest_img + t_idy * d_pitch);

    //source base address is equal to
    //offset in bytes from base address = (char *)
    //row number * row in bytes = t_idy * s_pitch
    //this is converted to float4 type for accessing elements with: (float4*)
    float4 *row_source = (float4*)((char *)source_img + t_idy * s_pitch);


    float r = row_source[t_idx].x;
    float g = row_source[t_idx].y;
    float b = row_source[t_idx].z;


    if ((r == 1.0) && (r > 2.0*g) && (r > 2.0*b))
    {
      row_dest[t_idx] = 1.0;
```

```
    }
    else
    {
      row_dest[t_idx] = 0.0;
    }


  }

}
```

A.6 Find white pixels

```
__global__ void
findWhite2D(float* dest_img, float* source_img, int n_rows, int n_cols, size_t d_pitch, size_t s_pitch)
{

  int t_idx = blockIdx.x*blockDim.x + threadIdx.x;
  int t_idy = blockIdx.y*blockDim.y + threadIdx.y;

  if ((t_idx > 0) && (t_idx < N_COLS - 1) && (t_idy > 0) && (t_idy < N_ROWS - 1))
  {
    float *row_dest = (float*)((char *)dest_img + t_idy * d_pitch);

    //source base address is equal to
    //offset in bytes from base address = (char *)
    //row number * row in bytes = t_idy * s_pitch
    //this is converted to float4 type for accessing elements with: (float4*)
    float4 *row_source = (float4*)((char *)source_img + t_idy * s_pitch);


    float r = row_source[t_idx].x;
    float g = row_source[t_idx].y;
    float b = row_source[t_idx].z;


    if ((r == 1.0) && (g > 0.6) && (b > 0.6) && (g < b + 0.3) && (b < g + 0.3) && (r > g) && (r > b))
    {
      row_dest[t_idx] = 1.0;
    }
    else
    {
      row_dest[t_idx] = 0.0;
    }


  }

}
```

A.7 Find red regions

```
__global__ void
findRedBlocks2D(float* dest_img, float* source_img, int n_rows, int n_cols, size_t d_pitch, size_t s_pitch)
{

  int t_idx = (blockIdx.x*blockDim.x + threadIdx.x)*4;
  int t_idy = (blockIdx.y*blockDim.y + threadIdx.y)*4;
```

```
  //now t_idx % 4 = 0 and t_idy % 4 = 0

  if ((t_idx > 1) && (t_idx < N_COLS - 2) && (t_idy > 2) && (t_idy < N_ROWS - 3))
  {
    float *row_dest = (float*)((char *)dest_img + (int(t_idy/4)) * d_pitch);

    //source base address is equal to
    //offset in bytes from base address = (char *)
    //row number * row in bytes = t_idy * s_pitch
    //this is converted to float type for accessing elements with: (float*)
    float *row_source = (float*)((char *)source_img + t_idy * s_pitch);

    if ((row_source[t_idx - 2] != 0.0) && (row_source[t_idx - 1] != 0.0)&&
    (row_source[t_idx] != 0.0)&&(row_source[t_idx+1] != 0.0) && (row_source[t_idx + 2] != 0.0))
    {
      row_source = (float*)((char *)source_img + (t_idy - 3) * s_pitch);
      float above3 = row_source[t_idx];
      row_source = (float*)((char *)source_img + (t_idy - 2) * s_pitch);
      float above2 = row_source[t_idx];
      row_source = (float*)((char *)source_img + (t_idy - 1) * s_pitch);
      float above = row_source[t_idx];
      row_source = (float*)((char *)source_img + (t_idy + 1) * s_pitch);
      float below = row_source[t_idx];
      row_source = (float*)((char *)source_img + (t_idy + 2) * s_pitch);
      float below2 = row_source[t_idx];
      row_source = (float*)((char *)source_img + (t_idy + 3) * s_pitch);
      float below3 = row_source[t_idx];
      if ((above3 != 0.0) && (above2 != 0.0)&&(above != 0.0)
      && (below != 0.0)&&(below2 != 0.0)&&(below3 != 0.0))
      {
        row_dest[int(t_idx/4)] = 1.0;
      }
      else
      {
        row_dest[int(t_idx/4)] = 0.0;
      }
    }
    else
    {
      row_dest[int(t_idx/4)] = 0.0;
    }



  }

}
```

A.8 Find white regions

```
__global__ void
findWhiteBlocks2D(float* dest_img, float* source_img, int n_rows, int n_cols, size_t d_pitch, size_t s_pitch)
{

  int t_idx = (blockIdx.x*blockDim.x + threadIdx.x) * 4;
  int t_idy = (blockIdx.y*blockDim.y + threadIdx.y) * 4;
  //now t_idx % 4 = 0 and t_idy % 4 = 0

  if ((t_idx > 2) && (t_idx < N_COLS - 2) && (t_idy > 3) && (t_idy < N_ROWS - 3))
  {
    float *row_dest = (float*)((char *)dest_img + (int(t_idy / 4)) * d_pitch);

    //source base address is equal to
    //offset in bytes from base address = (char *)
```

```
    //row number * row in bytes = t_idy * s_pitch
    //this is converted to float type for accessing elements with: (float*)
    float *row_source = (float*)((char *)source_img + t_idy * s_pitch);


    int col0 = 0;
    int col1 = 0;
    int col2 = 0;
    int col3 = 0;
    int col4 = 0;
    for (int i = 0; i < 7; i++)
    {
      row_source = (float*)((char *)source_img + (t_idy + i - 3) * s_pitch);
      if (row_source[t_idx - 2])
      {
        col0 |= (1 << i);
      }
      if (row_source[t_idx - 1])
      {
        col1 |= (1 << i);
      }
      if (row_source[t_idx])
      {
        col2 |= (1 << i);
      }
      if (row_source[t_idx + 1])
      {
        col3 |= (1 << i);
      }
      if (row_source[t_idx + 2])
      {
        col4 |= (1 << i);
      }
    }
    if ((col0&col1) && (col1&col2) && (col2&col3) && (col3&col4))
    {
      row_dest[int(t_idx / 4)] = 1.0;
    }
    else
    {
      row_dest[int(t_idx / 4)] = 0.0;
    }

  }

}
```

A.9 Capture IMU

```
import RTIMU
import time
import math
import arrow
import numpy as np
SETTINGS_FILE = "RTIMULib"


s = RTIMU.Settings("RTIMULib")
imu = RTIMU.RTIMU(s)

print("IMU Name: " + imu.IMUName())

if (not imu.IMUInit()):
    print("IMU Init Failed")
```

```
    exit()
else:
    print("IMU Init Succeeded")

# this is a good time to set any fusion parameters

imu.setSlerpPower(0.02)
imu.setGyroEnable(True)
imu.setAccelEnable(True)
imu.setCompassEnable(True)

poll_interval = imu.IMUGetPollInterval()
print("Recommended Poll Interval: %dmS\n" % poll_interval)
time.sleep(2)
base_name = "gettingcloserIMU"
print("saving to files:"+base_name+"_frame_"+str(0).zfill(6)+".png")
#take image
#capture motion data betweeen shots
ix = 0
start_time = str(arrow.utcnow()).replace(":", "-").replace(".", "-")+".txt"
with open("motion_dump_imu_"+start_time, "w") as out_file:
    time1 = arrow.utcnow()
    time2 = arrow.utcnow()
    dt = time2-time1
    motion_data = []
    while dt.seconds < 20:
      if imu.IMURead():
        # x, y, z = imu.getFusionData()
        # print("%f %f %f" % (x,y,z))
        data = imu.getIMUData()
        acc = data["accel"]
        ax = acc[0]*9.81
        ay = acc[1]*9.81
        az = acc[2]*9.81
        fusionPose = data["fusionPose"]
        roll = math.degrees(fusionPose[0])
        pitch = math.degrees(fusionPose[1])
        yaw = math.degrees(fusionPose[2])
        motion_data.append([ax,ay,az,roll, pitch, yaw, str(dt.seconds)+"."+str(dt.microseconds).zfill(6)])
        #print("r: %f p: %f y: %f" % (math.degrees(fusionPose[0]),
        #     math.degrees(fusionPose[1]), math.degrees(fusionPose[2])))
        #time.sleep(poll_interval*1.0/1000.0)
        time2 = arrow.utcnow()
        dt = time2-time1

    for item in motion_data:
        out_file.write(str(item[0])+";"+str(item[1])+";"+str(item[2])+";"+str(item[3])+";"+str(item[4])+";"+str(
    ix += 1
```

A.10 Capture camera

```
import time
import math
from picamera import PiCamera
import picamera
import picamera.array
from fractions import Fraction
import arrow
from skimage import io as spio
import numpy as np


with picamera.PiCamera() as camera:
    camera.resolution = (1640,1232)
```

```
camera.rotation = 180
time.sleep(2)
base_name = "gettingcloser"
print("saving_to_files:"+base_name+"_frame_"+str(0).zfill(6)+".png")
camera.shutter_speed = camera.exposure_speed
camera.exposure_mode = "off"
camera.awb_gains = (Fraction(0,1), Fraction(0,1))
burst_capture = []
#take image
#capture motion data betweeen shots
ix = 0
output = np.empty((1664*1232*3,),dtype=np.uint8)
start_time = str(arrow.utcnow()).replace(":", "-").replace(".", "-")+".txt"
with open("motion_dump_"+start_time, "w") as out_file:
    while ix < 10:
        camera.capture(output, "rgb")
        base_img = output.reshape((1232,1664,3))
        base_img = base_img[:1232, :1640, :]
        fname = base_name+"_frame_"+str(ix).zfill(6)+".jpg"
        spio.imsave(fname, base_img)
        out_file.write(fname+";"+arrow.utcnow().isoformat()+"\n")
        ix += 1
```