Spring 4-20-2010

# A Computer Vision Application to Accurately Estimate Object Distance

Kayton B. Parekh
*Macalester College*, kaytonparekh@gmail.com

# A Computer Vision Application to Accurately Estimate Object Distance

Submitted to the Department of Mathematics, Statistics and Computer Science in partial fulfillment of the requirements for the degree of Bachelor of Arts

*By*
Kayton Bharat Parekh

| | |
|---|---|
| Advisor: | Prof. Susan Fox |
| Second Reader: | Prof. Elizabeth Shoop |
| Third Reader: | Prof. Daniel Kaplan |

Macalester College

Apr 20, 2010

## Abstract

Scientists have been working to create robots that perform manual work for years. However, creating machines that can navigate themselves and respond to their environment has proven to be difficult. One integral task to such research is to estimate the position of objects in the robot's visual field.

In this project we examine an implementation of computer vision depth perception. Our application uses color-based object tracking combined with model-based pose estimation to estimate the depth of specific objects in the view of our Pioneer 2 and Power Wheels robots. We use the Camshift algorithm for color-based object tracking, which uses a color-histogram and gradient density climbing to find the object. We then use the POSIT algorithm, or Pose from Orthographic Scaling with ITerations, to estimate the pose of the object based on a model. Although pose consists of a translation vector and rotation matrix, we are only concerned with the z-component of the translation vector as it is an estimation of depth.

Our results show that distortions caused by the environment result in inaccurate distance estimations. These distortions include reflections of the object off surfaces such as a linoleum floor. However, when no such distortions are present, the application is rather well behaved and returns consistent, accurate depth estimations.

An advantage of the application is that it can be run in real-time for both standard web cameras and more expensive higher quality cameras. It can also be used with stored image files.

## Acknowledgments

I would like to acknowledge the work contributed by Professor Susan Fox, my research supervisor and thesis advisor. I would also like to thank her for keeping me on the right track. I would like to thank Professor Libby Shoop for holding me to such high standards. In addition, I would like to thank Professor Vittorio Addona, Professor Daniel Flath, Professor Thomas Halverson and Professor Tonnis ter Veldhuis for their help in understanding the math notation and concepts used in these algorithms.

# Contents

# List of Figures

# 1 Introduction

Consider a robot that needs to determine how to navigate around an obstacle in its path. The robot first needs to be able to detect the obstacle, determine how far away the object is, and calculate the best path around it. The second of these three tasks, determining how far away the object is, can be accomplished easily with relevant equipment such as a laser range-sensor. However, when a robot relies entirely on its vision, other techniques must be employed.

There are two main tasks to estimating the depth of an object from a single video sequence, tracking the object, and estimating the position of the tracked object. Object tracking is a problem that has been studied by many. Some solutions to object tracking include segmentation [3] [2], motion vector calculations [2], and density gradient estimation [4]. Pose estimation is the problem of determining the translation and rotation of an object in an image with respect to the camera. The translation obtained from pose estimation is the position of the object and pose estimation can therefore be used for the second task involved in depth estimation. Methods for pose estimation include genetic algorithms [5], stereo cameras [2], and algebraic methods with projections [6]. This study uses an adaptive version of density gradient estimation for object tracking with an iterative implementation of the algebraic methods for pose estimation.

In this project we consider the case when a robot has only a single camera and wishes to estimate the distance to an object in its field of vision. The method we use for object tracking is called the Camshift algorithm [1], and relies on color histograms and density gradient estimation. The method for pose estimation we use is called POSIT [6], and uses properties of the camera and a model of the object to algebraically estimate the pose while iteratively improving its estimate. The application we developed can be used with inexpensive web cameras, very expensive high quality cameras, or stored images on the hard disk.

Although the motivation for this research was to improve the accuracy of a Monte Carlo Localizer in the robots, this project is not concerned with the actual integration of the application for this purpose. This goal of this project is to create an application for object depth perception.

The results of this project show that, when there are no distortions in the environment, the error in the reported distance, that is the difference between the estimated and measured distances, is normal. By normal we mean it is Gaussian with mean values close to zero and relatively low standard deviations. Thus, the estimations in these cases gave predictably reliable estimations of depth. However, 32% of the environments in which we collected data showed significant distortions and inaccurate measurements. These distortions mostly consisted of reflections of the object off the floor which confused the color-tracker into tracking the object and its reflection, effectively doubling the perceived size of the object. In order to make use of this application, non-reflective surfaces will need to be placed underneath the objects.

In Section 3 we give a general view of the field of computer vision and what it entails. We also discuss other methods for object tracking and pose estimation, as well as mathematical functions needed to understand the Camshift algorithm. In Section 2 we discuss the robotics system at Macalester College and what motivated this research. We then go on to discuss the Camshift and POSIT algorithms in Section 5. Then in Section 6 we discuss the design and components of our application. In Sections 7 and 8 we discuss the results of our work provide a conclusion. In Section 9 we discuss possible future work. The source code for our application is provided in the appendices.

## 1.1 Aim

The aim of this project is to create an application which can extract range information from color intensity images. We do this by estimating the distance between the camera and a predetermined object in the robot's visual field: an orange cone. Before we can estimate the object's distance, we must first determine where the object is in the image. To answer this question, we need a method to detect and track the object when it appears in the robot's visual field. We then need a method to estimate the position of the object with respect to the camera.

The algorithms we chose to use are the Camshift algorithm and the POSIT algorithm. The Camshift algorithm is color-based object tracker, meaning it tracks objects based on an initial seed of color pixels. The POSIT algorithm uses a model of the object, the location of the object in the image, and intrinsic properties of the camera to iteratively improve its estimate of the object pose (rotation and translation).

# 2 Motivation

The robotics project at Macalester College uses Monte Carlo Localization, which relies on sensor information from a robot to determine where that robot is in a map of the Olin-Rice Science Center at Macalester College. The sonar sensors which are used are unreliable and return inaccurate range information, which created the need for additional sensors to be incorporated into the localizer. The only other sensors onboard the robots were cameras, which led us to investigate computer vision. Because of how Monte Carlo Localization works, it is easier to incorporate depth information into the localizer because it can be treated similar to how the sonar information is treated. Therefore, we chose to look into extracting some sort of range information from the cameras, which led us to investigate estimating the depth of known objects in the visual field of a robot. To explain the motivation behind our research, we describe Monte Carlo Localization and the Macalester robotics system in the next two sections.

## 2.1 Monte Carlo Localization

The Monte Carlo methods are a set of algorithms that rely on repeated random sampling to calculate results. This probabilistic approach is favorable when it is impossible to deterministically calculate an exact result. Monte Carlo Localization (MCL) is an approach to robot localization that uses a statistical particle filter. Robot localization determines the position and orientation of a robot as it moves through a known map of its world. MCL is a particle filter because it randomly generates possible points on the map and then eliminates points with low probability of being the actual position of the robot [7]. An example of this in action is shown in Fig. 1. Each point, or sample, is represented by $x$ and $y$ coordinates, an angle of orientation, and an associated probability. By eliminating samples with low probability, generating new samples based on previous ones, and updating current samples, the MCL algorithm can narrow samples down to only those near the actual location of the robot [8]. An illustration of this process is shown in Fig. 1(a) and Fig. 1(b).

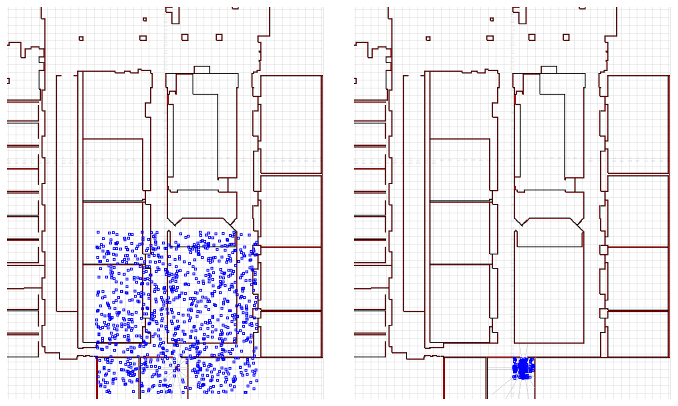

(a) An initial set of samples (b) A set of samples after several iterations

Figure 1: MCL samples converging on robot location

While moving, the robot reports information about its motion: a distance traveled and an angle turned. However, this motion alone cannot be used to accurately estimate the position of the robot. This technique, known as dead reckoning, can only work if the initial location of the robot is known and the motion information is extremely accurate. If the motion information has even a little error, this error will accumulate after a while and result in completely inaccurate estimates of the location. MCL is therefore more useful for most common robots where often the data reported about the motion does not always agree with the actual movement [8].

MCL therefore uses both motion and sensor information together to generate better samples from iteration to iteration. MCL changes the probability of each sample based on the motion and sensor information measured by the robot, and then eliminates low-probability samples [8].

Motion information consists of measurements of the distance moved and the angle turned sent by the robot. When this data is sent from the robot to the localizer, the localizer uses it to update the location of each of the samples. MCL could simply move each sample by the motion reported from the robot, but as mentioned this data is inaccurate. Therefore a better way to update a sample is by randomly selecting a distance and angle from normal distributions centered at the distance and angle values reported from the robot. Therefore, each sample moves randomly with a mean movement equal to the motion values reported by the robot [7].

Sensor information can be in many different forms depending on the types of sensors. For example, robots can have range sensors such as sonar or radar, vision sensors such as a camera or infrared imager, or localizing sensors such as GPS devices. In each of these cases the measured values for each sensor must be compared to the calculated values for each of the samples. Given the sensor measurement, the probability of a sample is altered depending on how close the measured sensor value is to the calculated sensor value for that sample [7]. The localizer used at Macalester College relies on range data from sonar sensors onboard the robots. Therefore, during each iteration, the sonar distances are calculated for each sample and compared to the values reported by the robot.

Between iterations, the localizer eliminates low-probability samples and biases towards samples closer to the actual location of the robot. This is done by randomly selecting samples from the old set such that higher-probability samples are more likely to be selected. These are combined with a small set of new samples generated near the believed location of the robot to create the sample set for the next iteration [8].

Because of the need to calculate sensor information for each sample and compare it to measured values, the easiest way to incorporate vision sensors into the localizer is if the data consists of a kind of range data to a known object in the robot's field of view, such as a cone. We can then calculate the distance between each sample and the nearest known object and compare this distance to the reported value from the robot. Therefore, in this project we create an application to extract the depth of an object using only the camera as a sensor. This depth can then be used to update sample probabilities similar to how the sonar sensor measurements are used currently.

## 2.2 The Macalester Robotics System

The Robotics Lab at Macalester College is headed by Professor Susan Fox. It consists of three robots, two of which are ActivMedia Pioneer 2 robots named Lauren and Humphrey. These have 16 sonar sensors positioned around their bodies and Sony cameras mounted on their backs. A picture of Lauren is shown in Fig. 2(a).

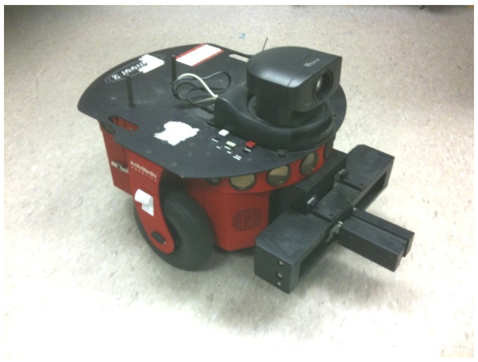
The third robot is a retrofitted Fisher-Price Power Wheels toy car as shown in Fig. 2(b). The toy has been modified with sonar sensors, a linear-actuator for steering, a webcam, and micro-controllers for interface with an onboard laptop. This robot is named Borg, after the late computer scientist Anita Borg. The laptop onboard Borg is a Dell Latitude running Ubuntu, while the computers onboard Humphrey and Lauren are older, slower machines running Red Hat.

Because of the age of the computers onboard Humphrey and Lauren, localization or vision algorithms cannot be run onboard due to computational latency. A desktop in the lab remotely runs these algorithms and interfaces with the mobile robots via wireless networks. Humphrey and Lauren both have wireless antennas and are capable of connecting to wireless computer networks. We therefore send sensor and motion information over the Macalester network to the desktop computer which sends back instructions. The Monte Carlo Localizer is run remotely on this desktop as well as a learning-based path finder written by Professor Susan Fox.

Since the robot controllers are written in Python, but the localizer and our vision application are written in C++, the `popen` Python module is used to spawn separate processes. This allows the Python process to control the subprocesses by reading from and writing to their input and output streams.

Currently, Borg does not run the MCL algorithm nor does she have a map of her world. We use Borg to test this project's vision application and to run simple condition-based decision algorithms. The MCL algorithm is used solely with Humphrey and Lauren. For more information on robotics at Macalester College see Anderson-Sprecher [9] and Stamenova [8].

The current localizer has one major problem, the sonar sensors return inaccurate data. This is because the walls in Olin-Rice, the science building at Macalester College, are reflective, and often the sonar pings are not returned to the sensors. This obviously confuses the MCL algorithm. In order to improve the localizer, we needed a method to incorporate the other available sensor devices: the cameras. If we could use the cameras as range finders and estimate distance based on images, we could use this information in the localizer similar to how we use the sonar data.


(a) Lauren, one of the two Pioneer 2 robots

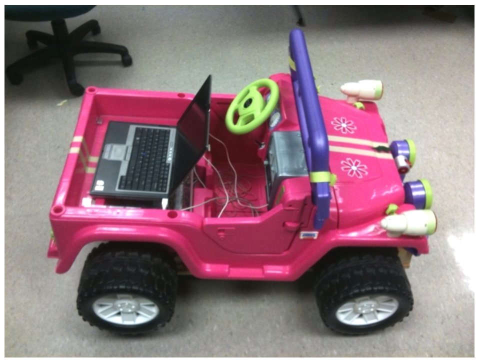
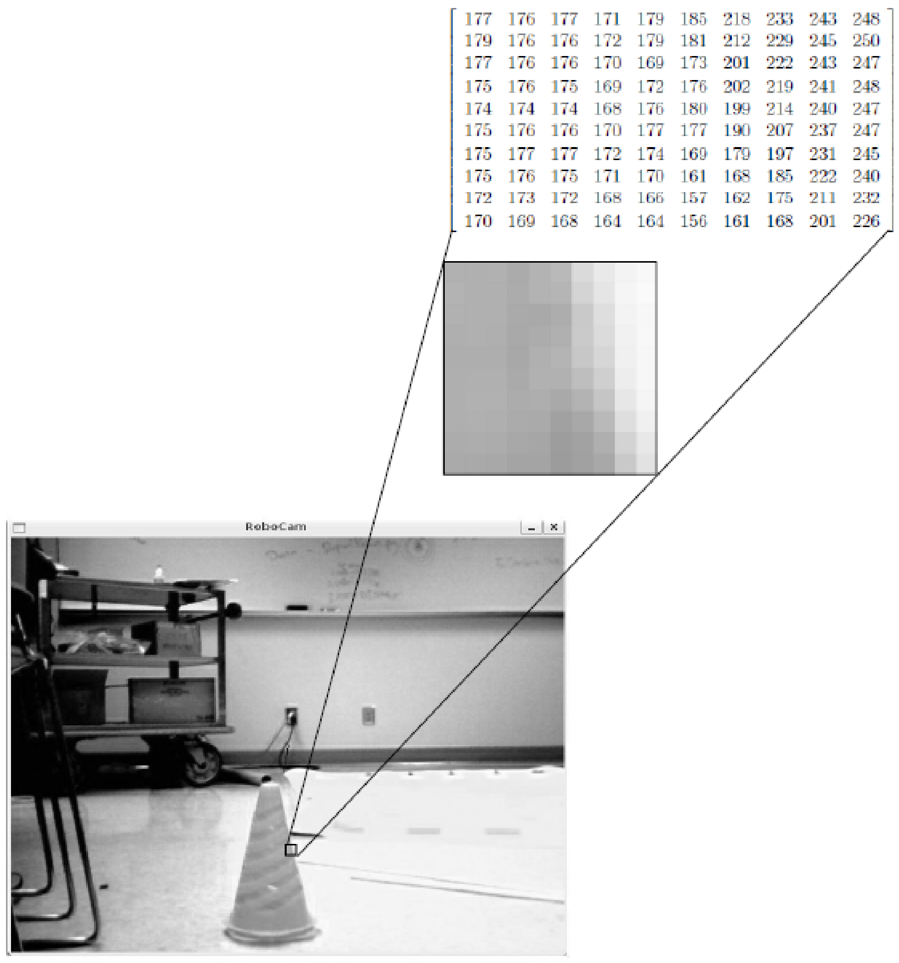

(b) Borg, the retrofitted Power Wheels robot

Figure 2: The two types of robots at Macalester College

## 3 Background

According to Bradski and Kaehler [2], computer vision is the transformation of two-dimensional image data into a decision or a new representation. It is easy to underestimate the difficulty of this problem because vision comes naturally to humans. Humans have a fully integrated system for vision with large reliance on feedback and automatic control. The human eyes automatically adjust to focus and aperture. The human brain is able to identify and select important parts of an image and suppress others. The brain is also able to cross-reference images with years of experience. Computers do not have such an integrated system nor do they have such a vast corpus of past experiences. To computers, images are simply a matrix of numbers, as shown in Fig. 3. The computer's goal is to make some decision about the image, such as "there is an orange cone in the scene", or "There is a person standing 500 centimeters away." To give a sense of the field of computer vision, we discuss some basic topics of digital imaging including vision sensors, image representation, and color spaces. We then discuss a few applications of computer vision.



$$\begin{bmatrix} 177 & 176 & 177 & 171 & 179 & 185 & 218 & 233 & 243 & 248 \\ 179 & 176 & 176 & 172 & 179 & 181 & 212 & 229 & 245 & 250 \\ 177 & 176 & 176 & 170 & 169 & 173 & 201 & 222 & 243 & 247 \\ 175 & 176 & 175 & 169 & 172 & 176 & 202 & 219 & 241 & 248 \\ 174 & 174 & 174 & 168 & 176 & 180 & 199 & 214 & 240 & 247 \\ 175 & 176 & 176 & 170 & 177 & 177 & 190 & 207 & 237 & 247 \\ 175 & 177 & 177 & 172 & 174 & 169 & 179 & 197 & 231 & 245 \\ 175 & 176 & 175 & 171 & 170 & 161 & 168 & 185 & 222 & 240 \\ 172 & 173 & 172 & 168 & 166 & 157 & 162 & 175 & 211 & 232 \\ 170 & 169 & 168 & 164 & 164 & 156 & 161 & 168 & 201 & 226 \end{bmatrix}$$

Figure 3: How a computer represents an image

## 3.1 Vision Sensors

There are many different types of digital vision sensors. Each is used to create an image that contains some information about the scene. Some sensors detect range information while others simply take color images. Some sensors can even take pictures of the inside of the human body. In general, though, images consist of a two-dimensional array of binary data in which each cell in the array makes up a pixel, or picture element. We provide a basic description of range cameras, normal light-sensing cameras, and video cameras here. For more information on types of vision sensors see Shapiro and Stockman [10].

Range cameras use electromagnetic radar beams to detect depth information in the scene. Each pixel of the image therefore represents the depth of the object at that location in the image. Such images contain information about the surface of objects in the scene, and can more accurately represent motion than intensity images [10].

Light-sensing camera sensors consist of a two-dimensional array of light sensors on a chip. These sensors detect the intensity of light that hits them. A major type of camera sensor is the charge-coupled device (CCD). In CCD cameras, the light hits a photoelectric cell producing an amount of charge depending on the intensity of the light. The charge is converted to binary data using a shift register, or CCD [11].

Video cameras are simply normal cameras that record sequences of images, or frames. The standard rate for frame sequencing is 30 fps (frames per second). Humans cannot detect change fast enough to notice the change from frame to frame, and therefore the sequence gives the illusion of motion [10]. For this project, standard color video cameras are used, however our application can track objects in still images as well.

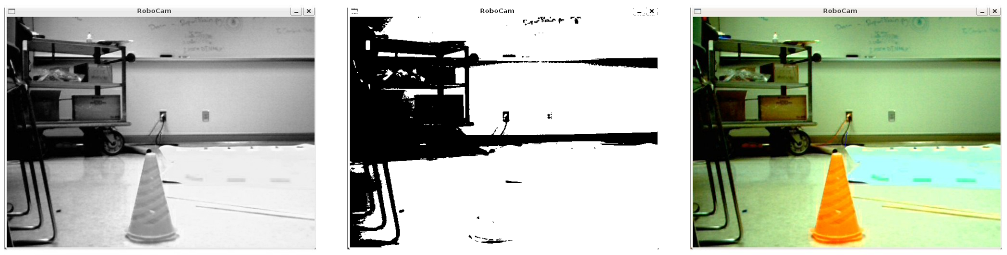
## 3.2 Image representation

Each pixel in an image is represented by binary data. This data can be one value or a combination of a few values. These values are usually represented in a single 8-bit byte of data, meaning they can be between 0 and 255.

In grayscale, an image is a single two-dimensional array of pixels, where each pixel is represented by a single byte. This byte represents the intensity of the pixel. An example of a grayscale image is shown in Fig. 4(a).

Binary images are images where each pixel is either a zero or one. Depending on the encoding, zero could represent the absence of a pixel and one could represent the presence of a pixel. Therefore, the image would be white where there are no pixels and black where there are. An example of a binary image is shown in Fig. 4(b).

Color images contain more information that just intensity or the presence of a pixel. Color images must encode the color of each pixel. There are several color systems for doing this, two of which are discussed in Section 3.3. The standard color system, Red-Green-Blue, represents each pixel by 24-bits (3 bytes). Each byte represents a red, green, or blue value for the pixel. The combination of these three values gives us the color of the pixel. Fig. 4(c) shows an example of an RGB image. Our application uses color images in order to track objects based on their color. However, we convert the color images to grayscale images that represent probability, as will be described in Section 5.

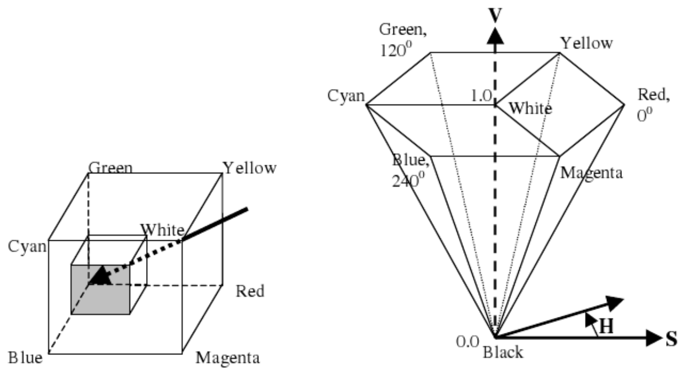


(a) Grayscale Image (b) Binary Image (c) RGB Color Image

Figure 4: An image of a cone in three modes

## 3.3 Color Spaces

The standard basis for color is the Red-Green-Blue (RGB) color space, which can encode 16 million different colors, a number far greater than the number of human-discernible colors. The RGB system uses three bytes to represent color, one for the red value, one for the green value, and one for the blue value. Therefore the encoding of any arbitrary color can be made by a combination of the primary red, green, and blue colors. For example, full red with full green gives yellow $(255, 255, 0)$. Equal proportions of all three colors give different shades of gray $(x, x, x)$ [10]. The graph of this color space is shown in Fig. 5(a). Note that the origin of this color cube corresponds to the color black.



(a) RGB Color Cube (b) HSV Color Hexicone

Figure 5: RGB and HSV Color Spaces, Images used with permission of Bradski [1]

The Hue-Saturation-Value (HSV) color space separates out intensity (value) from color (hue) and saturation. This color space is often also called Hue-Saturation-Intensity (HSI). HSV basically is obtained by projecting the RGB color cube along its principle diagonal. Note that the diagonal arrow in Fig. 5(a) corresponds to the vertical $y$ axis of the HSV color hexicone in Fig. 5(b), but in opposite direction. Fig. 5(b) shows the Hue-Saturation-Value color space. This color space is less susceptible to changes in lighting than the RGB color space [1]. This is because the perceived color of objects in the world can vary greatly due to changes in lighting, however the underlying hue changes less than the actual red, green, and blue values. For this reason, our application uses the HSV color space and RGB images must be converted to HSV before tracking can be applied.

## 3.4 Applications

Computer Vision has many applications. To give a general understanding of the significance of the field, we give descriptions of a few applications in this section.

### 3.4.1 Multimedia Database Search

There are many examples when large collections of digital images or video sequences are useful. For example, the popular video sharing website YouTube, or the image search feature of Google, both need to be able to query a database of multimedia content and find relevant results. Many solutions to this problem use human-entered meta-data to index the images: text that describes the contents of the images. However, content-based database querying has many applications. For example, if a user wants to find images in a database that are similar to a particular image, or the military wants to search a database of satellite images for troop movement, content-bases searching would be useful [12]. The applications of such technology are vast.

### 3.4.2 Computer-Aided Diagnosis

In medicine, images of the human body are used to make diagnoses of disease. Such images are obtained using radiation other than visible light in order to noninvasively capture images of the inside of the human body. Such devices include MRI, or Magnetic Resonance Imaging, Ultrasound, and X-ray images. Current research is being conducted to create computer systems that aid in the diagnosis of diseases by processing such images. Computer vision techniques are often used to aid in data presentation and measurement in medical imaging. However, recent research has started using computer vision for actual diagnosis. Fuentes-Pavon et al. [13] created a system to segment ultrasound images in order to apply diagnostic criteria to discriminate between malignant and benign breast tumors [13].

### 3.4.3 Surveillance Systems

Surveillance systems have started using computer vision to detect suspicious behavior in security surveillance cameras. Researchers have developed technology to detect suspicious activity such as abandoned luggage or the presence of a weapon. Such technology could be used in airports and transport stations to detect possible terrorist activity [14]. When a single surveillance operator is in charge of many cameras it becomes impossible to monitor all activity. Such vision systems have the potential of assisting operators to select which cameras to monitor.

Similarly, Automated Teller Machines (ATM) have been built to scan the human eye to determine identity and detect stolen debit cards [10].

### 3.4.4 Robotic Vehicle Navigation

Robotic Vehicle Navigation is what this project falls under. However, this project deals with small indoor robots rather than full-sized cars. One somewhat famous use of computer vision for vehicle navigation was the DARPA Grand Challenge. The Defense Advanced Research Projects Agency (DARPA) sponsored the Grand Challenge to inspire innovation in unmanned ground vehicle navigation. The Challenge consisted of a 142-mile course through the Mojave desert. The race gave a prize of one million dollars to the team whose robot could successfully navigate the course in the least amount of time. In the first year, no team completed the course. In 2005, the challenge was repeated and Stanford University's "Stanley" robot, shown in Fig. 6, won with a time just under 7 hours. Stanley depends heavily on radar imaging and vision algorithms [15].

### 3.4.5 Game Interfaces

Much research has been conducted toward user interfaces for video games that rely on computer vision. Such systems rely on images to track movements of the operator and translate the movements into gaming controls.

Microsoft Corporation is currently working on a project called Project Na-

Figure 6: Stanley, the robot that won the DARPA grand challenge, Image used with permission of Bradski & Kaehler [2]

tal for their Xbox gaming console, that eliminates the need for hand-held controllers. The gaming console is completely controlled through a vision device that sits on top of the television and tracks movements of the user. Users will be able to drive cars by mimicking the appearance of holding a steering wheel, kick a ball by actually kicking, and browse through items by swiping their hand like turning a page of a book. This interface is scheduled to be released around the end of 2010 [16]. The technology for such a device relies highly on computer vision. The vision sensors in this device are range cameras that have been adapted from Israeli ballistic missile guidance systems [17].

## 3.5 OpenCV

For this project, to avoid recreating algorithms already implemented and available, we use the OpenCV API in our implementation. Computationally optimal implementations of basic versions of Camshift and POSIT exist in OpenCV and our application utilizes them. OpenCV stands for Open Computer Vision, and is an open-source library containing algorithms for image processing and computer vision. The library, available for Linux, Mac OS, and Windows, is written in C and C++. However, current development exists to create libraries for Python, Ruby, MATLAB, and other languages. OpenCV was designed to be computationally optimized in order to support real-time applications. OpenCV also contains a general-purpose Machine Learning Library since vision and machine learning often go hand-in-hand [2].

The major goals of the developers of OpenCV were to prevent researchers from having to reinvent the wheel by giving them the basic algorithms, to provide a common code base for researchers in order to improve portability, and to advance commercial applications of computer vision [2].

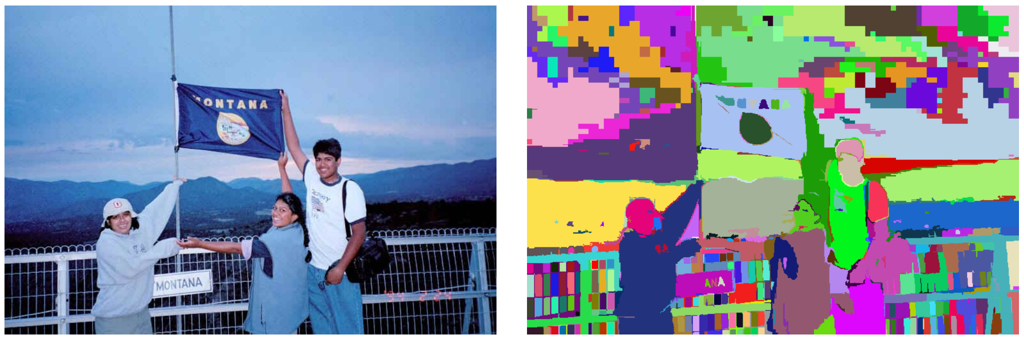
# 4 Related Work

## 4.1 Object Tracking

Object Tracking is a topic within Computer Vision that deals with recognition and tracking of moving objects in a scene. These objects can be known and recognized from a stored model or can be recognized based on features such as shape, color, or texture. Tracking deals with determining the location of the object in the image even if the object changes position between consecutive images such as in a video sequence. Tracking can also deal with prediction of object location based on trajectory. There are several different techniques for object detection and tracking. The techniques we looked at, and discuss here, are segmentation, and optical Flow. The Camshift algorithm, which is what is used in this project, uses another method known as density gradient estimation, which we discuss in Section 5.

### 4.1.1 Segmentation

Segmentation is aimed at partitioning the image into similar regions, or segments. A segmented image is shown in Fig. 7. This can be applied to object tracking in different ways. For example, if the object interior has a unique configuration of segments, then each new image frame is segmented and the tiles belonging to the object are identified. Two methods for image segmentation are Mean Shift and Graph-cuts [18].



(a) The original color image (b) A segmented version of the image

Figure 7: Segmentation by graph-cuts

*Mean Shift Segmentation*

Comaniciu and Meer [19] propose the Mean Shift approach to find clusters of image pixels based on color and location. The algorithm starts with a large number of clusters randomly placed on the image. These clusters are small windows of pixels. Each cluster is iteratively moved along the mean shift vector of their interior pixels. This mean shift vector is discussed more in Section 5.1.3, however it basically finds the centroid of the given cluster window, then re-centers the window over the centroid. The centroid of an image is similar to the center of mass of an object, where image intensities are analogous to physical mass. During this process some clusters are merged. The algorithm finishes when all clusters stop moving or move less than a given threshold. The final positions of the clusters are the segments of the image [18]. This technique is similar to the technique we used for object-tracking because of its hill-climbing approach. However, our approach is based on an application of Mean Shift to object tracking that operates on a search window and does not require computation on the full image.

*Graph-Cuts*

Image segmentation can also be seen as a problem of graph partitioning. In this approach, a graph, $G$, is created such that there is a vertex for each pixel in the image and a weighted edge between each adjacent pixel. The segmentation algorithm then partitions $G$ into disjoint subgraphs by pruning the weighted edges. The weight of each edge is calculated by some similarity between the nodes such as color, brightness, or texture. Two example criteria for pruning the edges are the minimum cut [20] and the normalized cut [21].

Image segmentation requires processing the entire image for each new frame. This can be computationally expensive especially for high-resolution images. Because of this, segmentation is too slow to be run in real-time which is why we rejected it as a solution to our problem.

### 4.1.2 Optical Flow

Another approach to object tracking that we considered is the optical flow method. Unlike our method, optical flow calculates the velocity of specific pixels based on their intensity changes between image frames. Optical flow finds a velocity of each pixel in the frame, this is equivalent to finding the displacement of a given pixel between consecutive image frames. If the velocity for every pixel in the image is calculated, a dense optical flow is created. Calculating dense optical flow is hard, which is why sparse optical flow is sometimes used. Sparse optical flow tracks a smaller subset of the image pixels [2].

*Lucas-Kanade Optical Flow Algorithm*

Lucas and Kanade [22] developed an algorithm for optical flow tracking, known as the Lucas-Kanade (LK) optical flow algorithm. LK works by making three assumptions:

**Brightness Constancy:** A pixel of an object image does not change appearance as the object moves around the scene from frame to frame.

**Temporal Persistence:** An object will move slowly relative to the time interval between frames. This means an object's movement between frames will be small.

**Spatial Coherence:** Neighboring points in the scene are assumed to be on the same surface, have the same motion, and have neighboring image projections.

Eq. 1 simply formalizes the first assumption, that pixel intensity, $I(x, y, t)$, does not change over time.

$$I_x v_x + I_y v_y + I_t = 0 \tag{1}$$

In this equation, $v_x$ and $v_y$ are the components of the optical flow vector $\vec{v}$, and are both unknowns in this equation. The terms $I_x$ and $I_y$ are the spacial derivatives across the first frame, while $I_t$ is the derivative between images over time. For derivation of Eq. 1 see Bradski & Kaehler [2].

Because there are two unknowns in this equation, a unique solution cannot be obtained from measurements of a single pixel. LK therefore uses a small box of pixels around each point and tracks the entire box instead. This produces a system of equations that can be solved for $v_x$ and $v_y$ [2].

This produces a flow vector for a single point. LK calculates the vectors for multiple points on an object and tracks its movement through the scene. LK is fast enough to be used in real-time, but it may not be robust enough for our purposes. LK works nicely for static cameras, but with moving cameras, where the object may move in and out of the visual field, LK may loose its tracked points between frames. In addition to these reasons, we rejected this method mainly because it does not provide any information on the perceived size of the object, which our method for depth estimation needs.

## 4.2 Depth Estimation

3-D Pose is a term referring to the combined translation and rotation of an object. This can be in reference to the initial position and orientation of the object or to some model of the object. In computer vision, pose estimation is often used to determine the position and orientation of an object with respect to the camera. In the following sections we describe how depth is calculated from stereo cameras and a method we looked at for estimating an object's pose using a genetic algorithm. This project uses another method that relies on a model that will be discussed in Section 5.

### 4.2.1 Depth From Stereopsis

The aim of this project is to determine the depth of an object in the robot's visual field. One common way of doing this that we researched is called stereopsis and uses two separate cameras.

Stereopsis is the process that takes two slightly different projections of the world, such as the projections on the retinas of human eyes, and produces a sense of depth. This concept is used to compute the depth of an object using stereo cameras. The basic idea is that the disparity between the location of the object in the left and right images is inversely proportional to the object depth [2]. The object depth is therefore calculated by Eq. 2 where $Z$ is the depth, $f$ is the focal length, $T$ is the distance between the cameras, and $d$ is the disparity. However, in order for this estimate to be accurate the two images must be perfectly rectified and aligned, as well as taken at the exact same moment. This configuration is illustrated in Fig. 8.

$$Z = \frac{fT}{d} \tag{2}$$

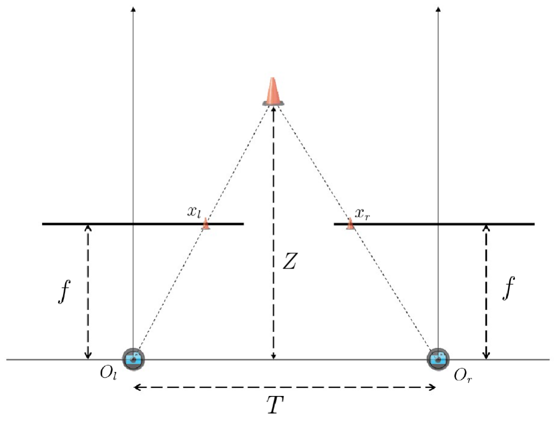


Figure 8: An ideal stereo camera rig for depth perception

This method makes the problem very complex because of the many issues associated with rectifying images of two separate cameras. We rejected this method because we are working with only one camera.

### 4.2.2 Genetic Algorithm Approach

A genetic algorithm (GA) is a optimization technique to solve computation problems that mimics the behavior of biological evolution. Toyama, Shoji, and Miyamichi [23] use a GA to determine the pose of an object based on a known model of the object. The fitness function for their implementation takes a given pose, orients the model in that pose, and projects the model onto a plane to compare with the image. Their method compares four edges from the image and the model projection to determine fitness rather than comparing all edges. According to their results they obtain a good solution in over 5000 iterations [23].

Although the approach by Toyama et al. [23] is similar to ours in that they utilize a model of the object, their approach assumes that edge detection on the image is robust enough to accurately distinguish between object edges and background edges. This is a safe assumption in a controlled environment such as in front of a white screen. However, in the real world edge detection is simply not robust enough. Furthermore, at 5000 iterations, even the smallest population sizes will take prohibitively too much time to be used with a 30 fps video stream. Our approach is able to estimate good approximations of object pose using intrinsic properties of the camera in only a few iterations, as will be described in Section 5.

# 5 Method

The algorithm we used for object tracking is called Camshift, and is based on another algorithm called Mean Shift. The algorithm we use for pose estimation is called POSIT, and is an iterative improvement on the POS algorithm. These algorithms are implemented in the OpenCV library, on which our modified versions are based. In the following sections we describe the methods behind them.

## 5.1 Color-Based Object Tracking

The problem with most conventional color tracking algorithms is that they attempt to deal with irregular object motion due to perspective. That is, objects farther from the camera move slower and appear smaller than objects closer to the camera. The algorithms also try to deal with image noise, distractors, occlusions, and other lighting variations. All of these considerations make other algorithms computationally complex and infeasible for use in a real-time object tracker. The algorithm we use solves these problems using a relatively computationally cheap approach.

The color tracking algorithm we used was first developed by Bradski [1], and was intended as a face tracking algorithm for use in user interfaces. The algorithm is called Camshift, or Continuously Adaptive Mean Shift, and the name comes from the basis algorithm Mean Shift. The Mean Shift algorithm is a robust technique to find the mode, or peak, of probability distributions by climbing density gradients [24]. The mean shift algorithm estimates the gradient using image moments calculated on the search window. The main difference between Mean Shift and Camshift is that Mean Shift operates on a fixed search window size while Camshift resizes its search window at each iteration.

In order to apply Mean Shift to images, we must convert the images to a probability density function. This requires some sort of mapping between pixel intensity and probability. In order to do this we use the hue from the HSV color space. As mentioned, HSV is less susceptible to changes in lighting than RGB. However, if we refer back to the HSV color hexicone in Fig. 5(b), we see that at very low intensity, saturation is also very low, which can cause hue to become very noisy. Pixels can also have inaccurate hues when intensity is very high, such as sunlight. We therefore set very high intensity and very low intensity pixels to zero. To create the probability density function, we then use a histogram of pixel hues from a seed image segment, as described in Section 5.1.1.

### 5.1.1 Calculating the Backproject

The Mean Shift algorithm and subsequently the Camshift algorithm work by climbing the gradients of a probability distribution. Therefore, to apply this to images we must convert the image to a probability distribution. We do this by using a histogram of colors for the target object. This histogram consists of a certain number of color bins (we arbitrarily use 32 bins) each representing a color range. This histogram is created based on the hues of a seed of pixels from an image of the object, which can come from a selection of pixels from a frame in the video sequence or an input file containing pixel color values. Each pixel of the seed is placed into one of the bins based on its hue, which are then normalized, or converted to percentages of the overall seed. Thus the value of any given bin represents the probability that a pixel, whose color would be in that bin, is a pixel on the object. A graphical representation of such a histogram is show in Fig. 9.

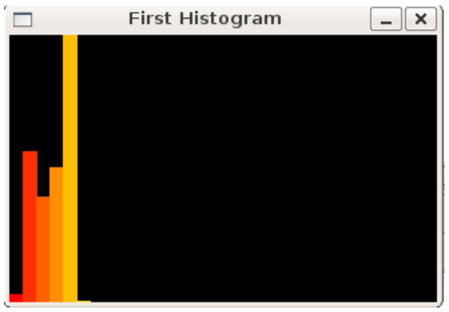
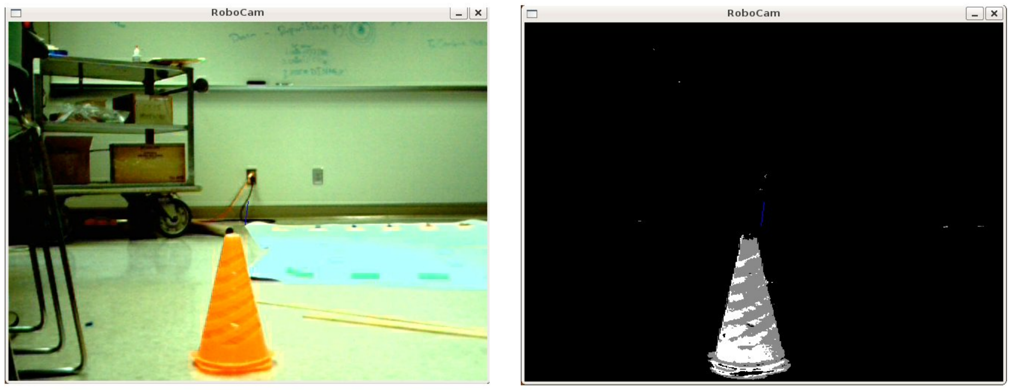


Figure 9: Histogram created from a seed of orange cone pixels

Using the histogram, we can calculate the probability for each pixel in a new frame from the video sequence and thus create a discrete two-dimensional probability function called the backproject. This backproject is stored as a grayscale image with dimensions equal to those of the color image frames. Unlike the color frames, whose pixels consist of color intensity values, the grayscale intensity of each pixel in the backproject is a value between 0 and 255 and represents the probability of the pixel in the same $xy$-coordinate of the image frame [1]. An example of an image and its corresponding backproject, as calculated using the histogram from Fig. 9, are given in Fig. 10.



(a) Orange Cone (b) Backproject of Orange Cone

Figure 10: An image of a cone and its corresponding backproject

### 5.1.2 Moment Functions

Moment Functions are used by the Mean Shift algorithm to estimate the density gradient for an image. They are also used by the Camshift algorithm to resize its search window and calculate the elliptical shape descriptors of the region. These functions are used in statistics, mechanics, and image analysis. In statistics, moments are calculated on a probability density function and represent different properties of the function. The zero-, one-, and two- order moments represent the total probability, the expectation, and the variance respectively. In mechanics, moments calculated for a spatial distribution of mass tell us certain properties of the mass. Similar to statistics, the zero-, one-, and two-order moments tell us the total mass, the centroid position, and the inertia values respectively. In image analysis we can treat the image as a two-dimensional intensity distribution. Furthermore, as described in Section 5.1.1 above, we can convert intensity values to probabilities using a color histogram. We can then take moments of these distributions to tell us certain properties of the image such as the total area, the centroid coordinates, and the orientation. In image analysis we are really only interested in low-order moments because higher-order moments are more sensitive to noise [25].

The general moment generating function is defined as:

$$\Phi_{pq} = \iint \Psi_{pq}(x,y) f(x,y) dx dy \tag{3}$$

where $\Phi_{pq}$ is the moment of degree $(p+q)$, $\Psi_{pq}(x,y)$ is the kernel function in $x$ and $y$, and $f$ is the density distribution. There are many different kernel functions each producing moments of different uses. However, when estimating the gradient of a density function we use geometric moments. Geometric moments are moments with a kernel function of $\Psi_{pq}(x,y) = x^p y^q$, and are denoted by $M_{pq}$ rather than $\Phi_{pq}$. From this point forward, the term moment will be used interchangeably with geometric moment. Furthermore, when working with discrete probability distributions such as the backproject, we use summation to represent integration. Therefore, Eq. 4 shows the geometric moment generating function for images [25]:

$$M_{pq} = \sum_x \sum_y x^p y^q f(x,y) \tag{4}$$

There are several different types of geometric moments. Silhouette moments are calculated from a binary image and thus any pixel on the object has a value of one while pixels outside the object have a value of zero. Boundary moments are moments calculated using only boundary points of an object. Standard moments are moments which have been normalized with respect to scale, translation, and rotation. Range moments are moments calculated on range images where each pixel corresponds to a distance to an object [25].

Just as in physics and statistics, moments of different orders represent different properties of the image's distribution. There are two cases to consider, the case where we calculate moments for the image itself, and the case were we convert the image to a probability distribution. In the first case we are dealing with intensity, in the second we are dealing with probability.

The zero-order moment, $M_{00}$, is defined to be the total intensity or probability for the region of the image. The first order moments, $M_{01}$ and $M_{10}$, are moments about the $y$ and $x$ axes respectively. Just as these moments can be used to find the centroid of a physical object in physics, they can be used to find the centroid of the image region, where the pixels' intensities are analogous to physical mass. The centroid is computed using the functions $x_c = M_{10}/M_{00}$ and $y_c = M_{01}/M_{00}$ [25]:

$$(x_c, y_c) = (\frac{M_{10}}{M_{00}}, \frac{M_{01}}{M_{00}}) \tag{5}$$

A property of the region's centroid that is used in the Mean Shift algorithm, is that the vector from the region's center point to the centroid of the region has the same direction as the gradient vector at the center point. The gradient vector for a density function points in the direction of greatest slope. Therefore, the gradient vector for the backproject will point towards higher probability. Mean Shift uses this fact and climbs the gradient to find the object.

Besides the gradient, Camshift also calculates the elliptical shape descriptors of the region. These are the width, height and angle of an ellipse encompassing the region. In order to calculate these, Camshift needs to compute moments that are independent of position, meaning in reference the region centroid. These moments are called centralized moments and are denoted by $\mu_{pq}$. Eq. 6 shows the centralized moment generating function [25].

$$\mu_{pq} = \sum_{x}\sum_{y}(x - x_c)^p(y - y_c)^q f(x, y) \tag{6}$$

The second order moments, $\mu_{02}$ and $\mu_{20}$, represent the variances about the centroid in both axes. The covariance is given by $\mu_{11}$. Therefore, Eq. 7 shows the covariance matrix [25]:

$$\mathbf{cov} = \begin{bmatrix} \mu_{20} & \mu_{11} \\ \mu_{11} & \mu_{02} \end{bmatrix} \tag{7}$$

The eigenvectors of this matrix correspond to the major and minor axis of an ellipse encompassing the region. The eigenvectors, $\vec{V_1}$ and $\vec{V_2}$ are given in Eq. 8.

$$\vec{V}_{1,2} = \begin{bmatrix} \frac{\mu_{20}-\mu_{02}\pm\sqrt{4\mu_{11}^2+(\mu_{20}-\mu_{02})^2}}{2\mu_{11}} \\ \\ 1 \end{bmatrix} \tag{8}$$

Fig. 11 shows that the tangent function can be used to calculate the angle $\theta$ of the ellipse. The angle of the region's ellipse is the angle of the eigenvector corresponding to the largest eigenvalue.

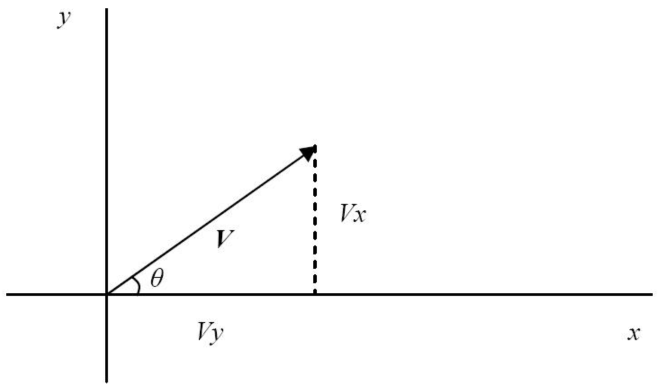


Figure 11: Eigenvector of the covariance matrix

$$\tan\theta = \frac{V_y}{V_x} = \frac{1}{V_x} = \frac{2\mu_{11}}{\mu_{20} - \mu_{02} + \sqrt{4\mu_{11}^2 + (\mu_{20} - \mu_{02})^2}} \tag{9}$$

$$\theta = \arctan\left(\frac{2\mu_{11}}{\mu_{20} - \mu_{02} + \sqrt{4\mu_{11}^2 + (\mu_{20} - \mu_{02})^2}}\right) \tag{10}$$

The eigenvalues of the **cov** matrix are proportional to the squared length of the eigenvector axes. Two different formulas for the eigenvalues exist, one obtained using conventional algebraic eigenvalue computation, the second obtained using positive-definite matrix diagonalization. The eigenvalues obtained from the first method are shown in Eq. 11.

$$\lambda_{1,2} = \frac{(\mu_{20} + \mu_{02}) \pm \sqrt{4\mu_{11}^2 + (\mu_{20} - \mu_{02})^2}}{2} \tag{11}$$

The second method uses positive-definite matrix diagonalization, and is the method used by Camshift. In this method, **cov** is diagonalized into $RDR^T$ where $R$ is the rotation matrix through angle $\theta$ and $D$ is the diagonal matrix with the eigenvalues of **cov** on its diagonal.

$$D = \begin{bmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{bmatrix} \tag{12}$$

$$R = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \tag{13}$$

Therefore, Camshift uses the property of diagonalizable matrices, $(\mathbf{cov} - \lambda I)\mathbf{x} = \mathbf{0}$ to solve for $\lambda$. In this equation, $\mathbf{x}$ is the eigenvector corresponding to $\lambda$ and $I$ is the identity matrix. For simplicity we omit calculations, but provide the formulas obtained [25]:

$$\lambda_1 = \mu_{20} \cos^2 \theta + 2\mu_{11} \cos\theta \sin\theta + \mu_{02} \sin^2 \theta \tag{14}$$

$$\lambda_2 = \mu_{20} \sin^2 \theta - 2\mu_{11} \cos\theta \sin\theta + \mu_{02} \cos^2 \theta \tag{15}$$

Note that these formulas are algebraically identical to those provided in Eq. 11, which can be verified by plugging in values for the unknowns.

Using the eigenvalues, Camshift can obtain elliptical shape descriptors for the image region. Therefore, an ellipse drawn around the region would have major and minor axis magnitudes of $a$ and $b$ respectively and an orientation of $\theta$. The angle $\theta$ can be calculated as already described, but the axes must be calculated using the following formulas [25]:

$$a = 2\sqrt{\frac{\lambda_1}{M_{00}}} \tag{16}$$

$$b = 2\sqrt{\frac{\lambda_2}{M_{00}}} \tag{17}$$

Camshift uses these elliptical shape descriptors to track the object's size and orientation.

### 5.1.3 Mean Shift

The Mean Shift is a nonparametric estimator of density gradient [3]. Mean Shift is used on density functions and nonparametric here refers to the fact that it does not assume any structure of the data. Regardless of the definition, Mean Shift acts as an estimate of the gradient of a density function. As mentioned, the gradient is the tangent vector to the density function in the direction of greatest slope. The iterative process developed by Fukunaga and Hostetler [26] is used to seek the local maximum in the density function. This process, also called the Mean Shift algorithm, involves iteratively shifting a fixed-size search window to the centroid of the data points within the window [3]. In this way Mean Shift can hill-climb to a local mode. This algorithm is proven to converge for discrete data [1]. Therefore, we use the Mean Shift algorithm on the backproject to find the local mode, or peak, in the density function, which translates to the point with the locally highest probability of being the object. The Mean Shift algorithm can thus track objects as they move in two degrees of freedom, translational along the $x$ and $y$ axes [2]. Besides object tracking, Comaniciu and Meer [3] used Mean Shift to segment images by way of data clustering [24], as described above in Section 4.1.1. The Mean Shift procedure is demonstrated by the iterations shown in Fig. 12 [2].

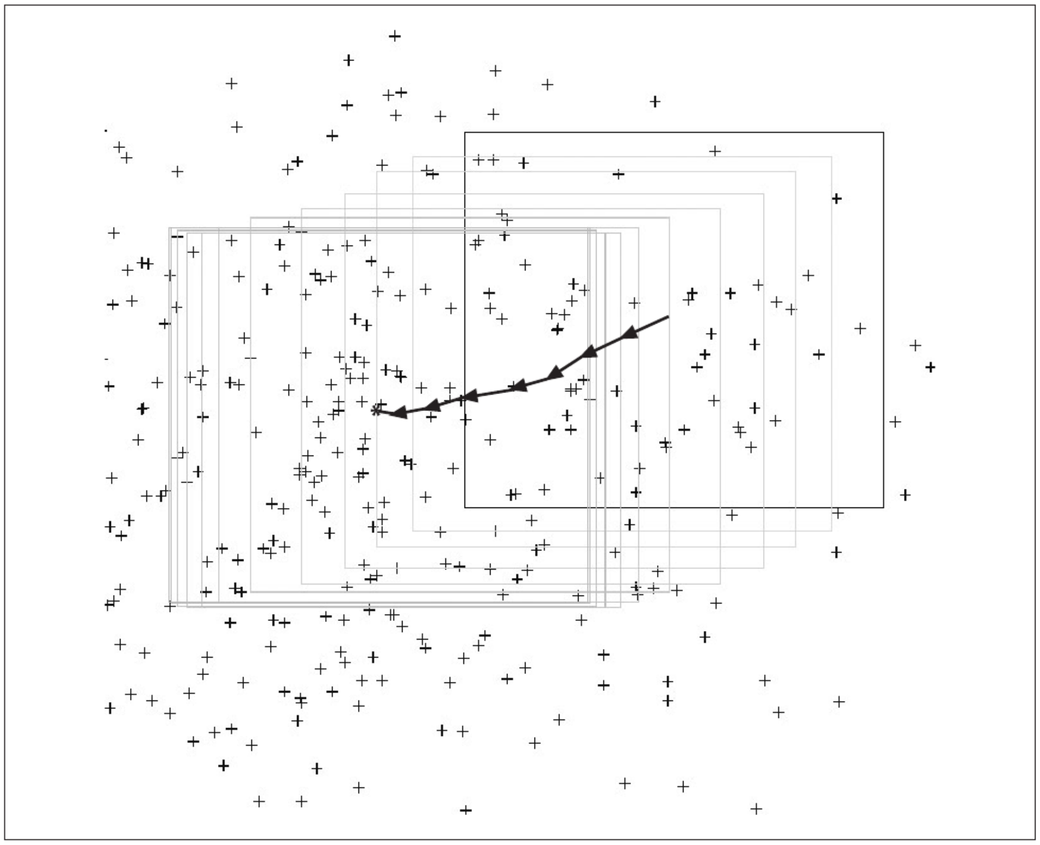


Figure 12: The motion of a Mean Shift search window as it moves over a 2-D data set, Image used with permission of Bradski & Kaehler [2]

In the next two sections we will first describe how Mean Shift estimates density gradient and then the Mean Shift algorithm. For a formal proof of the Mean Shift convergence see Comaniciu and Meer [3].

*Density Gradient Estimation*

Density gradient estimation refers to the use of Mean Shift to estimate the gradient, or direction of maximum upward slope, for a density function, in our case the backproject. We simplify the calculations by using a rectangular search window [2]. Therefore, to obtain the gradient, we calculate centroid of the search window. Recall that the centroid is calculated by the formula from Eq. 5 reproduced here:

$$(x_c, y_c) = (\frac{M_{10}}{M_{00}}, \frac{M_{01}}{M_{00}})$$

By subtracting the coordinates of the search window's current center from the coordinates of the centroid, we obtain a Mean Shift vector. As mentioned in Section 5.1.2, this Mean Shift vector has the same direction as the gradient of the density function at the center of the search window [3].

*The Mean Shift Algorithm*

The Mean Shift procedure is obtained by successively computing the Mean Shift vector and translating the search window through that vector until convergence. This means that the search window will end on the local mode, or maximum. Thus the method is simply climbing the gradient until it reaches the top, and can only go down. When we apply this to the backproject, we get an algorithm to track objects using a fixed window size as shown in algorithm 1 [1].

**Algorithm 1**: MeanShift

```
Input: search window size and position and threshold
trackwindow ← inputed search window;
repeat
    oldtrackwindow ← trackwindow;
    Calculate $M_{00}$, $M_{10}$, $M_{01}$;
    $x_c \leftarrow M_{10}/M_{00}$;
    $y_c \leftarrow M_{01}/M_{00}$;
    $dx \leftarrow x_c -$ trackwindow.center.$x$;
    $dy \leftarrow y_c -$ trackwindow.center.$y$;
    trackwindow.$x$ ← trackwindow.$x + dx$;
    trackwindow.$y$ ← trackwindow.$y + dy$;
until difference between oldtrackwindow and trackwindow $\leq$ threshold ;
```

One can imagine that if a vision system is tracking a moving object between successive image frames, for example a basketball as it is thrown, the object will have moved from its location in the previous frame. If the object is still at least partly within the search window, the hill climbing procedure will move the search window to be centered on the object once again [1]. In this way the search window tracks the object as it moves in the scene of a video sequence.

### 5.1.4 Camshift

Camshift stands for Continuously Adaptive Mean Shift and was first developed as a perceptual user interface by means of face tracking. The name Camshift, Continuously Adaptive Mean Shift, is derived from the fact that it continuously resizes the search window. The Mean Shift algorithm was designed for static distributions and was never intended as an object tracking algorithm. Thus, it does not take into consideration that the object's size will change if it moves along the $z$ axis with respect to the image plane. Camshift improves upon the Mean Shift algorithm with specific considerations for object tracking. For example, Camshift tracks objects in four degrees of freedom as opposed to Mean Shift's two degrees of freedom, and is designed to handle the dynamically changing distributions between consecutive images because it continuously resizes its search window. In addition to the $x$ and $y$ translation that Mean Shift trackers handle, Camshift tracks the area of the object and the roll. The area is a function of the $z$ translation because objects closer to the camera will appear larger than objects farther from it. Additionally, Camshift tracks the roll, or the rotation about the $z$ axis. Finally, Camshift makes the search window somewhat expansive in order to encompass the entirety of an object rather than smaller regions around the Mean Shift location. For example, if tracking faces we want to track the entire face and not fix on the nose or forehead [1].

*Calculating the Search Window Size*

To resize the search window, Camshift uses the area, given by the zero order moment $M_{00}$. The area is dependent on the $z$ coordinate of the object's location, meaning the closer the object is to the camera the larger the area of the object's image. Therefore, Camshift uses a function of $M_{00}$ to determine the search window size for the next iteration. However, Camshift must first convert $M_{00}$ into units of pixels rather than intensity. Since $M_{00}$ is summed from grayscale intensities, converting it to pixels requires dividing by 256.

Since $M_{00}$ corresponds to the area, Camshift can approximate the side length by simply take the square root of the area. In addition, since Camshift was designed with an expansive search window to encompass the nearby connected distribution area, it doubles this side length and obtains [1]:

$$s = 2\sqrt{\frac{M_{00}}{256}} \tag{18}$$

Since Camshift was first developed for face tracking, the search window is resized to match the shape of a human face. Therefore, the search window is set with a width of $s$ and a height of $1.2s$ because the face is somewhat elliptical in shape.

*Calculating the Elliptical Shape Descriptors*

In addition to the $z$ axis, Camshift tracks the roll, or rotation about the $z$ axis. When dealing with a human face, the range of motion represented by roll would be as if the person were to tilt their head from shoulder to shoulder. The roll is simply the angle between the principle axis of an ellipse over the object and the horizontal. This can be calculated using formulas in section 5.1.2, particularly if we look at Eq. 10, reproduced here:

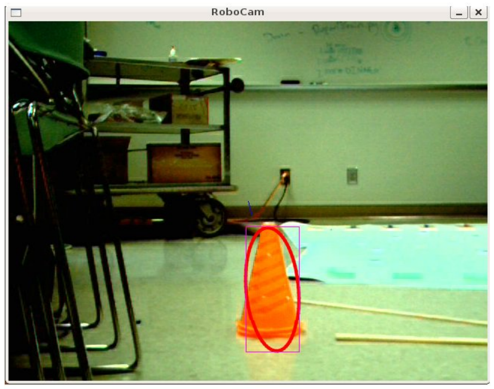
$$\theta = \arctan\left(\frac{2\mu_{11}}{\mu_{20} - \mu_{02} + \sqrt{4\mu_{11}^2 + (\mu_{20} - \mu_{02})^2}}\right)$$

Using this equation we can calculate the roll of the object. We can also use Eqs. 16 and 17 to determine the magnitudes of the major and minor axes of such an ellipse. Again these formulas are reproduced here:

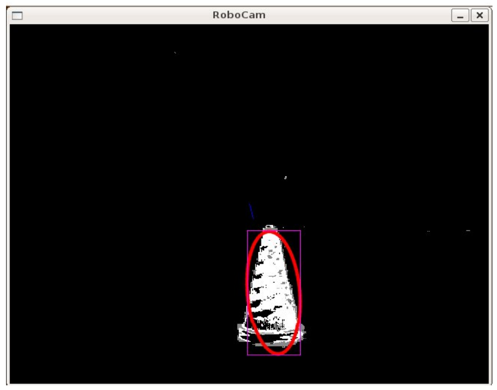
$$a = 2\sqrt{\frac{\lambda_1}{M_{00}}}$$

$$b = 2\sqrt{\frac{\lambda_2}{M_{00}}}$$

With these three elliptical shape descriptors we can draw a track ellipse around the tracked object. This ellipse is more descriptive than the simple search window because it shows the roll of the object as well. The search window is a rectangle and simply shows the location and size of the object. Examples of both are given in Fig. 13.



(a) Tracked Orange Cone



(b) Backproject of Tracked Orange Cone

Figure 13: An image of a cone and its corresponding backproject with overlays of a pink rectangular search window and red ellipse

*The Camshift Algorithm*

The Camshift algorithm invokes the Mean Shift algorithm in order to climb the density gradient. However, Camshift resizes the search window before each consecutive iteration. Algorithm 2 shows the pseudocode for the Camshift algorithm [1].

**Algorithm 2**: Camshift

```
Input: search window size and position
trackwindow ← inputed search window;
repeat
    oldtrackwindow ← trackwindow;
    $M_{00}$ ← MeanShift(trackwindow);
    $s \leftarrow 2\sqrt{M_{00}/256}$;
    trackwindow.width ← $s$;
    trackwindow.height ← $1.2s$;
until difference between oldtrackwindow and trackwindow ≤ threshold ;
```

*Advantages of Camshift Over Other Object Trackers*

There are many advantages to Camshift over other object tracking algorithms. Firstly, Camshift is able to deal with irregular object motion because of the adaptive nature of the search window. To explain, many algorithms have trouble with the fact that objects move at different speeds in the scene depending on their proximity to the camera. Farther objects move slower and appear smaller while closer objects move faster and appear larger. Camshift will resize its window depending on the object size. Therefore, even though the object is small when it is farther from the camera, its slow motion allows Camshift to track it. Also, when objects are close to the camera their motion is rapid, but because they have larger areas Camshift can catch the larger movements. Therefore, Camshift is robust to irregular object motion due to distance from the camera [1].

Furthermore, because Camshift uses a search window that climbs local distribution gradients, it will ignore outliers in the distribution. This also allows it to tend to ignore distractors, or objects with similar hue elsewhere in the scene. Once the Camshift has locked onto an object it will stay with that object unless the motion between two consecutive video frames is too great [1]. However, this also means that if Camshift locks onto a distractor, it will not re-find the object automatically.

In addition, Camshift converges on the mode of the distribution and therefore tends to ignore partial occlusions of the object. Since the search window will be centered on the mode of the distribution, if we block part of the object, the search window will remain over the visible portion of the object because of its proximity to the previous mode. This makes Camshift robust to partial occlusions [1].

Finally, because Camshift computes over pixels only in the search window, it eliminates needless calculations over the entirety of the image, and therefore can run fast enough to be used in real-time.

## 5.2 Model-Based Pose Estimation

The method we use to determine the pose of an object was first developed by DeMenthon and Davis [6]. This method uses geometry and linear algebra techniques to determine the pose of an object based on a model of the object's feature configuration. In other words, the user must define visible features on the object as points in the object's three-dimensional frame of reference. These model points must consist of at least four non-coplanar points on the object. For example, the model and feature points chosen for a cone are shown in Fig. 14. These points are then tracked in the image and a correspondence between feature points and model points is used to calculate pose.

The algorithm, called POSIT, is an iterative improvement on another algorithm called POS, which stands for Pose from Orthographic Scaling [6]. The focal length of the camera, which can be obtained from camera calibration algorithms, is needed to calculate the pose of the object based on a model [2]. POSIT invokes POS over many iterations, each time improving the input parameters to POS based on its previous outputs. This is akin to Newtons's method for finding the square root of a number.

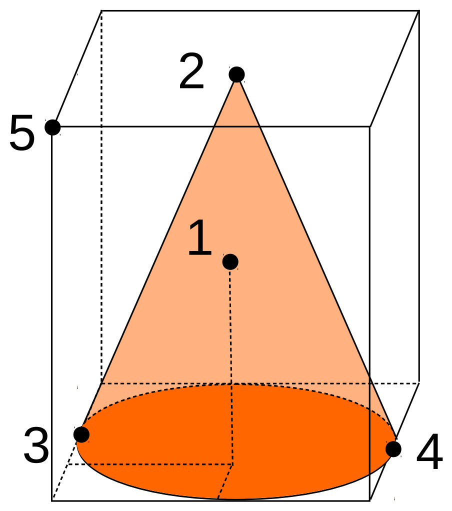


Figure 14: The model of a cone used for pose estimation

### 5.2.1 SOP and Perspective Projection

In order to understand these algorithms, one must understand the difference between perspective projection and Scaled Orthographic Projection (SOP). First, let us describe the pinhole camera model, which has the center of projection, $O$, behind the image plane, $A$, as shown in Fig. 15. Projection, here, refers to the projection of the object points in three-dimensional space onto the two-dimensional image plane. Perspective projection, or true perspective projection, is when the image of $Q_i$, an object model point, is considered to be the intersection between the image plane and the line of sight from $O$ to the object point $Q_i$, denoted $q_i$. SOP ignores the depths of the object points, and projects all object points onto a plane $B$, where $B$ is parallel to the image plane passing through the object point of reference, $Q_0$. Therefore, in SOP, the image of the object point $Q_i$ is $p_i$ rather than $q_i$. The point $p_i$ is the intersection between the image plane and the line of sight from $O$ to $P_i$, where $P_i$ is the projection of $Q_i$ onto plane $B$. This method assumes that the object is flat and that the image is scaled-down version of the flat object [6]. See Fig. 15 for a graphical representation of this.

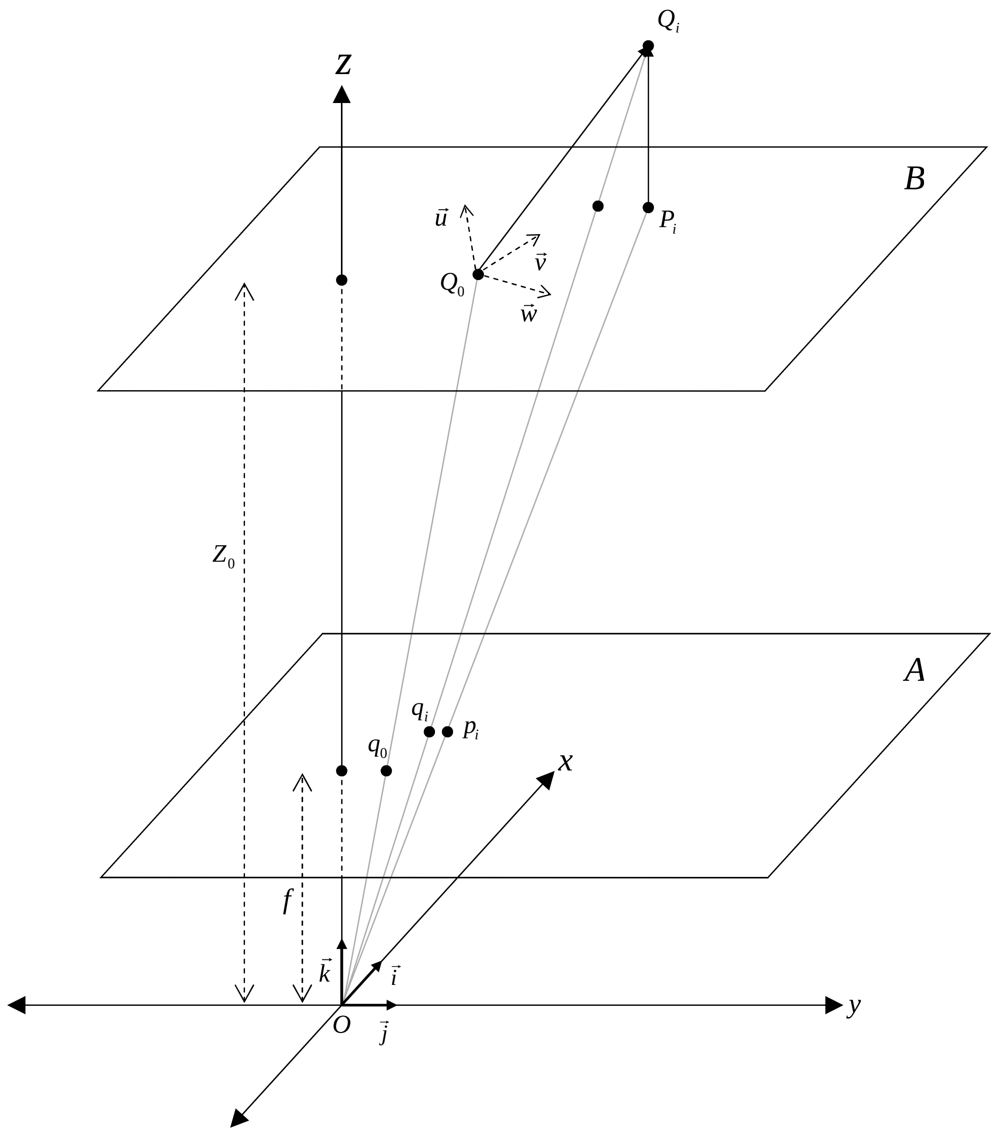


Figure 15: The pinhole camera model

Therefore, in SOP, sometimes called the weak-perspective camera model, the coordinates of point $p_i$ would be [6]:

$$x_i = \frac{f}{Z_0} X_i \tag{19}$$

$$y_i = \frac{f}{Z_0} Y_i \tag{20}$$

Where $Z_0$ is the $z$ coordinate of the point $Q_0$, the object reference point, in the camera's coordinate system. The term $f$ is the camera focal length and $X_i$ and $Y_i$ are the $x$ and $y$ coordinates of the point $Q_i$ in camera coordinates. In perspective projection these same equations would be [27]:

$$x_{i(pp)} = \frac{f}{Z_i} X_i$$

$$y_{i(pp)} = \frac{f}{Z_i} Y_i$$

Notice in the perspective projection equations the focal length $f$ is divided by $Z_i$ rather than $Z_0$ so that the depths of the individual object points are considered. The factor $s = \frac{f}{Z_0}$ is the scaling factor of the projection [6].

### 5.2.2 Perspective Equations

The objective of pose estimation is to determine a translation vector, $\overrightarrow{OQ_0}$, and a rotation matrix to convert points in the camera frame of reference to the object frame of reference. Notice from Fig. 15 that vectors $\vec{i}$, $\vec{j}$, and $\vec{k}$ are the basis vectors for the camera's frame of reference, let us call this vector space $\alpha$. Vectors $\vec{u}$, $\vec{v}$, and $\vec{w}$ are the basis vectors for the object's frame of reference, let us call this vector space $\beta$, and all points $Q_i$ are known in this coordinate space. Therefore the rotation matrix consists of column vectors equal to the unit vectors $\vec{i}_\beta$, $\vec{j}_\beta$, and $\vec{k}_\beta$, where the subscript $\beta$ means in coordinates of the vector space $\beta$, or object coordinates. Notice that $\vec{k}_\beta = \vec{i}_\beta \times \vec{j}_\beta$. Also notice that if the object point of reference, $Q_0$, is chosen to be one of the visible feature points, then its $x$ and $y$ coordinates are obtained by scaling up the coordinates of point $q_0$. Therefore $X_0 = \frac{Z_0}{f} x_0$ and $Y_0 = \frac{Z_0}{f} y_0$; here capital letters represent coordinates of $Q_0$ and lower case letters represent coordinates of $q_0$. Therefore the pose is fully defined by $\vec{i}_\beta$, $\vec{j}_\beta$, and $Z_0$ [6].

The system of equations to solve for these values is [6]:

$$\overrightarrow{Q_0Q_i} \bullet \vec{I} = x_i(1+\varepsilon_i) - x_0 \tag{21}$$

$$\overrightarrow{Q_0Q_i} \bullet \vec{J} = y_i(1+\varepsilon_i) - y_0 \tag{22}$$

Where

$$\vec{I} = \frac{f}{Z_0}\vec{i}_\beta$$

$$\vec{J} = \frac{f}{Z_0}\vec{j}_\beta$$

$$\varepsilon_i = \frac{1}{Z_0}\overrightarrow{Q_0Q_i} \bullet \vec{k}_\beta$$

### 5.2.3 POS

As mentioned, POS stands for Pose from Orthographic Scaling. The main idea is if $\varepsilon_i$ has a value, then the system of equations in Eqs. 21 and 22 can be solved for $\vec{I}$ and $\vec{J}$. This is what POS does, but it uses all of the object model points provided. Therefore, Eqs. 21 and 22 for all model points can be described as two linear systems of equations with $\vec{I}$ and $\vec{J}$ as the unknowns:

$$\mathbf{A}\vec{I} = \vec{x} \tag{23}$$

$$\mathbf{A}\vec{J} = \vec{y} \tag{24}$$

where matrix $\mathbf{A}$ consists of the object coordinates of each of the model points. In addition, given the geometric configuration of the object points $Q_i$, the pseudoinverse of matrix $\mathbf{A}$ can be precomputed as $\mathbf{B} = \left(\mathbf{A}^T\mathbf{A}\right)^{-1}\mathbf{A}^T$. Solving for $\vec{I}$ and $\vec{J}$ results in Eqs. 25 and 26 [6]:

$$\vec{I} = \mathbf{B}\vec{x} \tag{25}$$

$$\vec{J} = \mathbf{B}\vec{y} \tag{26}$$

Once POS has obtained $\vec{I}$ and $\vec{J}$ it will find the scaling factor $s$ by averaging the norms of these two vectors. The vectors $\vec{i}_\beta$ and $\vec{j}_\beta$ are obtained by normalizing $\vec{I}$ and $\vec{J}$, and the vector $\vec{k}_\beta$ is found by the cross product $\vec{i}_\beta \times \vec{j}_\beta$ resulting in the three column vectors of the rotation matrix, $\vec{i}_\beta$, $\vec{j}_\beta$, and $\vec{k}_\beta$. The scaling factor $s$ is used to find the translation vector. Remember that $s = \frac{f}{Z_0}$, therefore $Z_0 = \frac{f}{s}$. From this the other translation vector coordinates are $X_0 = \frac{x_0}{s}$ and $Y_0 = \frac{y_0}{s}$. Therefore the pose is given by:

$$\vec{T} = \overrightarrow{OQ_0} = \begin{bmatrix} X_0 \\ Y_0 \\ Z_0 \end{bmatrix} \text{ and } R = \begin{bmatrix} \vec{i}_\beta & \vec{j}_\beta & \vec{k}_\beta \end{bmatrix}$$

Notice that this pose is only as accurate as the value given to $\varepsilon_i$ [6]. We then use POSIT to improve upon this estimate.

### 5.2.4 POSIT

POSIT stands for POS with ITerations. This is because POSIT simply runs iterations of POS, each time using the resulting pose to calculate more accurate values for $\varepsilon_i$. POSIT starts by assuming $\varepsilon_i = 0$. Recall the formula for $\varepsilon_i$:

$$\varepsilon_i = \frac{1}{Z_0} \overrightarrow{Q_0 Q_i} \bullet \vec{k}_\beta$$

Therefore, after each iteration, $Z_0$ and $\vec{k}_\beta$ are used to calculate the next $\varepsilon_i$. These iterations stop when the change in $\varepsilon_i$ is below a threshold. The pseudocode for POSIT is shown in algorithm 3 [6].

**Algorithm 3**: POSIT

```
Input: Model Points $Q_i(X_i, Y_i, Z_i)$, Image Points $q_i(x_i, y_i)$, and threshold
Output: Translation vector and rotation matrix
$N \leftarrow$number of image points;
for $i \leftarrow 1 \dots N-1$ do $\varepsilon_{i(0)} \leftarrow 0$;
$n \leftarrow 1$;
$\mathbf{A} \leftarrow$ matrix of object coordinates $Q_i$;
$\mathbf{B} \leftarrow \left(\mathbf{A}^T\mathbf{A}\right)^{-1}\mathbf{A}^T$;
repeat
    for $i \leftarrow 0 \dots N-1$ do
        $i$th coordinate of $\vec{x} \leftarrow x_i(1+\varepsilon_{i(n-1)}) - x_0$;
        $i$th coordinate of $\vec{y} \leftarrow y_i(1+\varepsilon_{i(n-1)}) - y_0$;
    end
    $\vec{I} \leftarrow \mathbf{B}\vec{x}$;
    $\vec{J} \leftarrow \mathbf{B}\vec{y}$;
    $s_1 \leftarrow ||\vec{I}||$;
    $s_2 \leftarrow ||\vec{J}||$;
    $s \leftarrow (s_1 + s_2)/2$;
    $\vec{i}_\beta \leftarrow \vec{I}/s_1$;
    $\vec{j}_\beta \leftarrow \vec{J}/s_2$;
    $\vec{k}_\beta \leftarrow \vec{i}_\beta \times \vec{j}_\beta$;
    $Z_0 \leftarrow f/s$;
    for $i \leftarrow 1 \dots N-1$ do $\varepsilon_{i(n)} \leftarrow \frac{1}{Z_0}\overrightarrow{Q_0Q_i} \bullet \vec{k}_\beta$;
    $n \leftarrow n+1$
until $\left|\varepsilon_{i(n)} - \varepsilon_{i(n-1)}\right| < threshold$ ;
$X_0 \leftarrow x_0/s$;
$Y_0 \leftarrow y_0/s$;
$\vec{T} \leftarrow \begin{bmatrix} X_0 \\ Y_0 \\ Z_0 \end{bmatrix}$;
$R \leftarrow \begin{bmatrix} \vec{i}_\beta & \vec{j}_\beta & \vec{k}_\beta \end{bmatrix}$;
return $\vec{T}$ and $R$
```

# 6 Implementation

We built our implementation in C++ with a class called `ColorTracker` invoking the color tracking and pose estimation algorithms. The program that displays the images and handles user I/O is called `Capture`.

## 6.1 The Video Sequence

As mentioned, the computers onboard Humphrey and Lauren, the Pioneer 2 robots, are too slow to perform significant computation onboard. Therefore, just like the MCL algorithm, we needed to run the tracking remotely on the Linux desktop. In order to gain access to the video stream from the onboard cameras, we used Professor Fox's implementation of server-client python scripts communicating over TCP. The server script runs on the robot, capturing images from the camera stream and storing them locally. On the remote machine, a client script pulls the new images from the server. This means the video stream we have is on the hard drive consisting of separate consecutive JPEG images, and is not a webcam device like it is for Borg, the Power Wheels robot. In order to handle both cases, we made two subclasses of the `ColorTracker` object, namely `CameraColorTracker` and `FilesColorTracker`. The only overridden methods in these subclasses are the methods that handle obtaining the next frame of the video sequence. For the `CameraColorTracker` object, the next frame is taken from a video sequence. For the `FilesColorTracker` object, the next frame is taken by incrementing a number in a filename and loading an image from the hard drive using the new filename. Depending on command-line arguments, the `Capture` program knows whether to instantiate the `ColorTracker` as either a `FilesColorTracker` or a `CameraColorTracker`, and due to polymorphism in C++ either can be used when a `ColorTracker` is expected.

## 6.2 The Capture Program

The `Capture` program is what invokes the computer vision methods and displays the images of the video stream. In order to do both of these things simultaneously the program has two threads of operation, the display thread and the I/O thread. The display thread handles invoking the methods from the `ColorTracker` object in the right order. The I/O thread listens for input on standard-in and handles different user input. We use standard-in to control the device and modulate parameters such as reseting the window size or toggling between displaying the backprojects or the image. The inputs of ‘f’ and ‘v’ will cause the program to output the search window information and a vector to the object respectively. This is how a parent process can access the location of an object being tracked in order to make decisions for the robots behavior.

## 6.3 Loading the Histograms

In order to load a histogram to be used to generate the backprojects, we need a seed, or a set of pixels of the object we wish to track. Each pixel in this seed is placed into one of the 32 histogram bins based on its hue. Initially we were using 16 bins, however we found that 32 bins improved the accuracy and robustness of the tracker. The number 32 is arbitrary here and we could use any number of bins. However, adding more bins results in more computation.

The original method we used for seeding the program was to select a region of the displayed video sequence using the mouse. This was handled by the display thread of the `Capture` program. However, we realized that this seed was too small and did not include pixels of the object in different lighting.

One way we improved robustness of the algorithms was to allow the seed to be made from images of the object in many different light settings and with many different backgrounds. For example, we used an orange cone as our object and took pictures of it using the robot’s camera in many different locations around the building, or the robot’s world. We took pictures in dim light, bright light, close to the camera, far from the camera, with brightly lit windows in the background, with walls in the background, etc. This produced a formidable reservoir of images to use. We then needed a way to combine all of these images to be used as a single seed to the vision application. We used a program that Professor Fox developed in MATLAB which loads each image sequentially and prompts the user to select the region of pixels within the object. The program then saves all pixels selected to the single Comma Separated Value (CSV) file, represented as RGB values. When we input this CSV file to the vision application, the application must convert the values to an RGB image, convert the image to an HSV image, then use the image to create the histogram.

## 6.4 Re-finding the Object

The Camshift algorithm uses the search window from the previous frame as the initial search window for the current frame. Therefore, if the program loses the object for some reason, it will not re-find the object unless a part of it appears within the boundaries of the search window. We found that sometimes noise or similarly-hued objects in the background, such as the exit sign in Fig. 16, can steal the search window if the program happens to lose the object. In order to solve this issue we introduced a function that re-finds the object by simply resetting the search window size to that of the entire image. This way, after a few iterations the window will re-converge on the object, given it is the largest of its hue in the scene.

## 6.5 Splitting the Search Window

Another anomaly we observed occurs when two objects of similar hue are in the scene. If the two objects are close enough together the search window will elongate and encompass both objects together as if they were one. This is because the centroid of the two lies between them even though the actual probabilities at the centroid are small. An example of this sitution is shown in Fig. 17. To fix this we implemented a track window splitting function.

The splitting function calculates the total probability for a small region at the center of the search window. If this total probability is lower than a threshold then the window is split in half and the side with the largest probability is taken as the new search window. We split the search window horizontally if the width is less than the height, and vertically if the height is less than the width.The tracker does sometimes jitter on an object if the center probability is low. This is because the search window will be split, but then expand

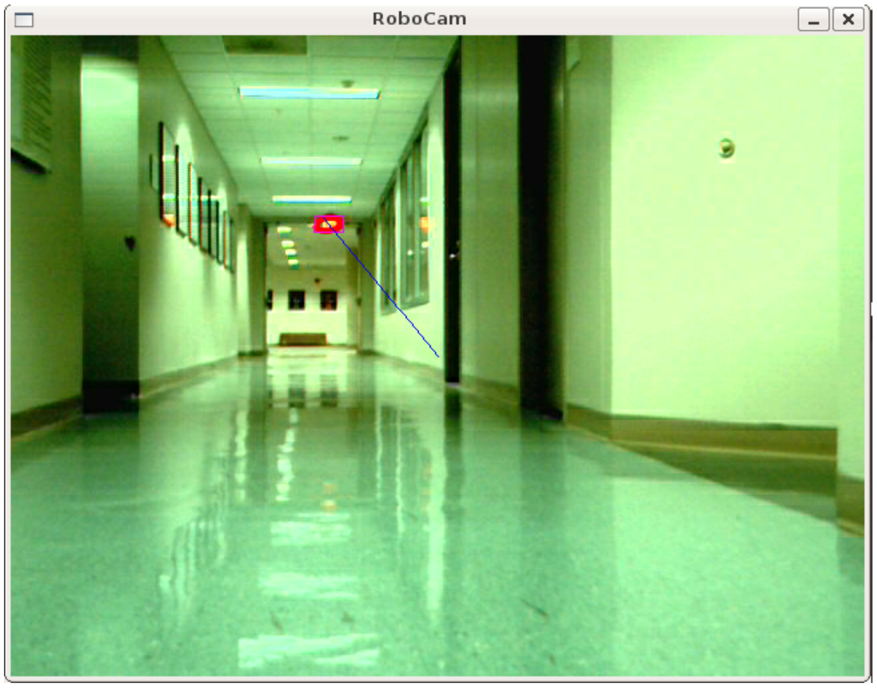


Figure 16: A similarly-hued object in the background steals the search window

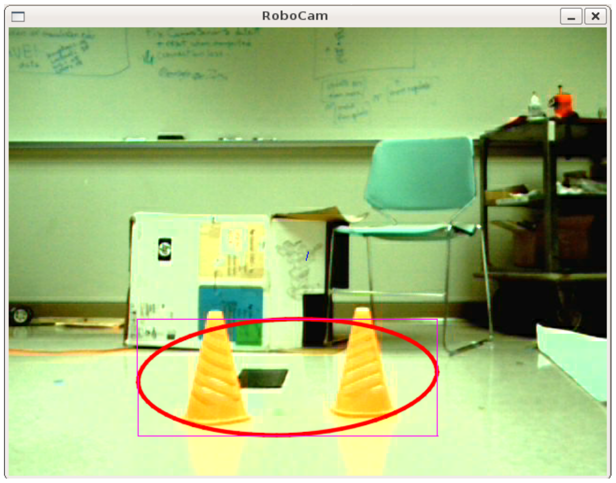


Figure 17: A situation when the search window is stretched over two similarly colored objects

to its original size only to be split again. Despite the occasional jitter, this function works remarkably well and we no longer have the issue of two objects of similar color appearing as a single object.

## 6.6 Tracking Two Objects

We imagined the possible need to track two objects at the same time, such as in Fig. 18. In order to track two objects, two CSV filenames must be inputed, each used to generate a different histogram. These filenames could be different if tracking differently colored objects, or the same. The histograms are used to generate two backprojects on the image frame. The first object is tracked on the first backproject. The search window for the first object is blacked out on the second backproject to avoid tracking the same object in the case that the two histograms are similar. The second backproject is then used to track another object. This is demonstrated in Fig. 19. This could be extended to multiple objects, however the more objects we track the more expensive the operation becomes. We are essentially running Camshift twice per frame. Recall that we need to finish computation in $\frac{1}{fps}$ of a second where $fps$ is the frame rate. For normal video frame rates, we have one 30th of a second to finish both Camshift operations when tracking two objects. When tracking many objects, this limit may cause the program to run in less than real-time.

## 6.7 Thresholding

A problem that we ran into is that if the object, our cone, is not present in the scene the vision application picks up the next highest probability pixels. This causes a problem because if all pixels have low probability, but some have slightly higher, those pixels will be wrongly tracked. This means that if there is no cone in the scene, rather than determining so, the application will pick up on noise pixels and track large groups of random noise. In order to fix this we needed a way to depress low probability pixels to zero. We accomplish this using a simple function for thresholding in the OpenCV API. We set the value of pixels with probability less than a predetermined threshold to zero. This helped, however noise is random and no matter

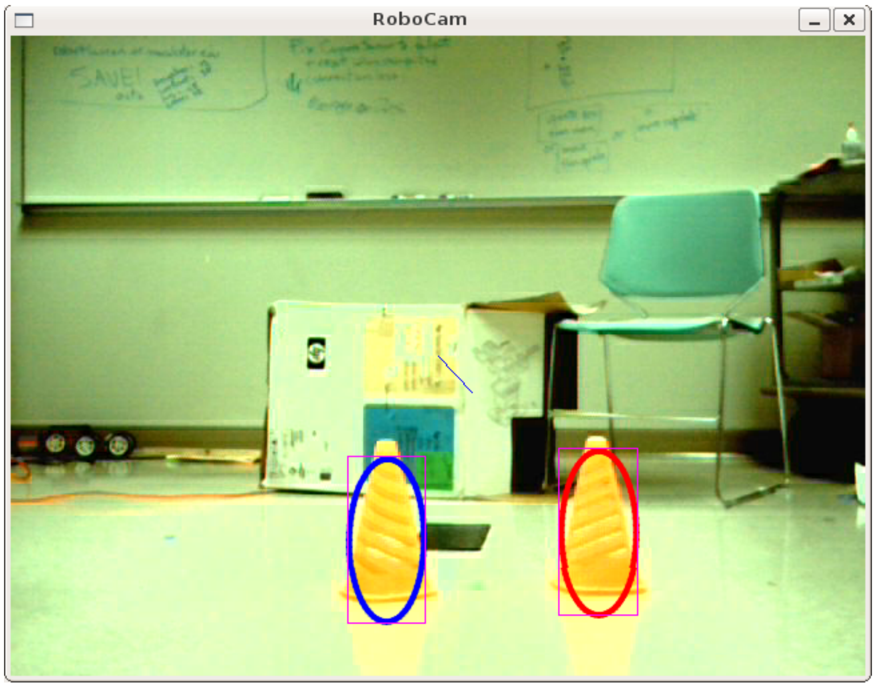
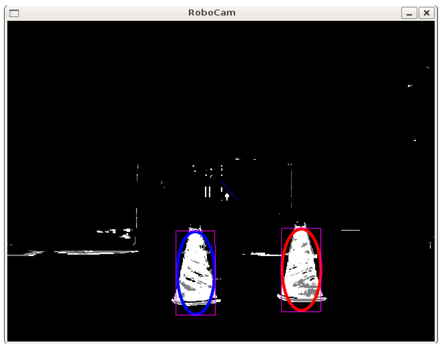
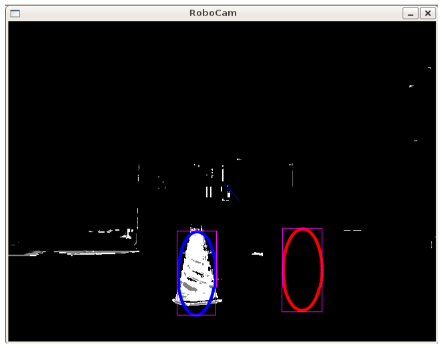


Figure 18: An example of tracking two objects simultaneously



(a) The backproject for the first object



(b) The backproject for the second object

Figure 19: The backprojects when tracking two objects simultaneously

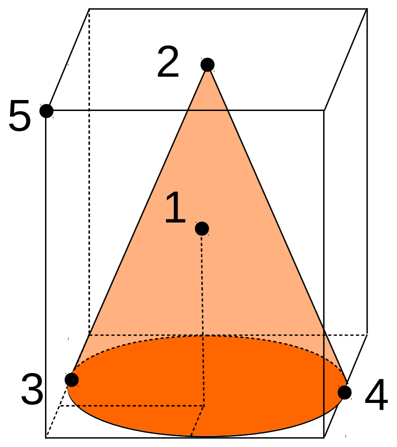
which threshold level we chose it seemed that noise pixels would eventually occur that had probabilities greater than the threshold. To account for this we decided to simply look for the optimal threshold level, one that would decrease the likeliness of high probability noise without causing dimly-lit cones to go unnoticed. We determined this threshold experimentally.

## 6.8 Focal Length

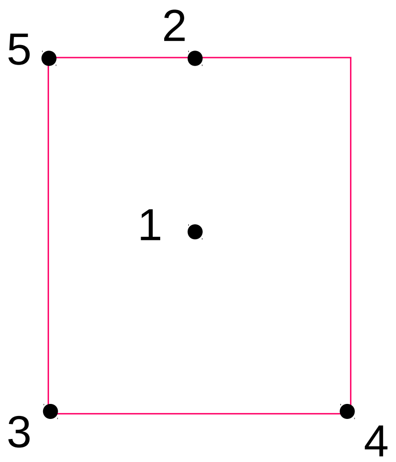
One of the arguments required of the POSIT algorithm is the focal length of the camera. This is used in the POSIT algorithm to determine the pose of the object based on a model of the object. One common technique for determining the focal length is to use camera calibration algorithms [2]. These algorithms return intrinsic properties of the camera. A common technique for camera calibration is to use a chessboard, which is a well known pattern, and rotate and translate the board in front of the camera. Camera calibration algorithms use successive frames of this well known pattern to calculate the camera's intrinsic properties. However, using this technique, we never managed to obtain a useful focal length. So we decided to experimentally determine the best focal length. We measured a distance and placed the object at that distance. We then varied the focal length by small intervals until the distance returned from the POSIT algorithm was within satisfactory error of the actual distance to the object. Although this is somewhat reverse engineered, it works well for our application when the search window accurately represents the object size, as will be shown in Section 7.

## 6.9 Image Points

In order for POSIT to work properly, we use the model of the cone shown in Fig. 20(a). However, POSIT also needs to know the corresponding image feature points for each of these model points. In order to determine the points on the cone we use a somewhat simplified approach. We assume that the search window represents an orthographic projection of the cone onto the image plane. Therefore, we do not account for distortions due to perspective. This will result in inaccurate rotation matrices but the translation vector should be relatively unaffected. The image points of the Camshift search window that we use are shown in Fig. 20(b).



(a) The model points on an orange cone



(b) The search window with feature points marked

Figure 20: The model points and their corresponding feature points

## 6.10 Distance Estimation

Although we are using the pose estimation algorithm POSIT, we are really only interested in the $Z_0$ value calculated. This is the $z$ coordinate of the translation vector returned by POSIT.

Theoretically, if we place cones around the robots' world, and include the positions of these cones in the robots' map of their world, then when a robot sees a cone in their frame of vision they can estimate the distance to that cone. This distance can be used by the MCL algorithm to more accurately alter probabilities of MCL samples similar to how the sonar measurements are used currently.

# 7 Results

## 7.1 Data Collection

We collected data on both types of robots in three different locations in order to vary light and surfaces. The locations were an open atrium, a hallway, and the lab room. The atrium and the lab room both had natural and artificial light. The hallway only had artificial light. The floor of the atrium was carpeted while the other two locations had linoleum.

In each location, for both types of robots, we measured distances in front of the robot and placed orange cones. We then collected data on the distance estimated by the application. We took 1000 data points for each. For the Pioneer 2 robot we used distances of 150, 200, 500, and 1000 centimeters. For the Power Wheels robot we used distances of 200, 300, 500, and 1000 centimeters. The lab room is 5 meters by 5 meters so there was no way to obtain data for 10 meters in that environment. We used different sets of distances because the Power Wheels robot has its camera higher than that of the Pioneer, and therefore it cannot see as close in front of it as the Pioneer.

## 7.2 Distortions

One phenomenon that threw off the system and caused values to be inaccurate were reflective floors. Floors that were tiled with surfaces, such as linoleum, sometimes reflected the image of the cone and distorted the size of the search window. This is shown in Fig. 21. The values returned in such cases were unpredictable and were often much less than the actual distances.

An obvious distortion that can also occur in moving robots is when the object is not completely in the view of the robot. These cases can also cause very inaccurate distance estimations to be reported. Three such cases are shown in Fig. 22.

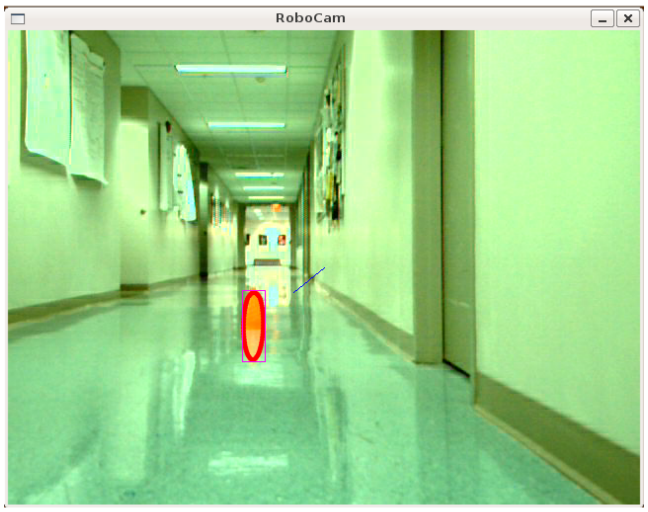
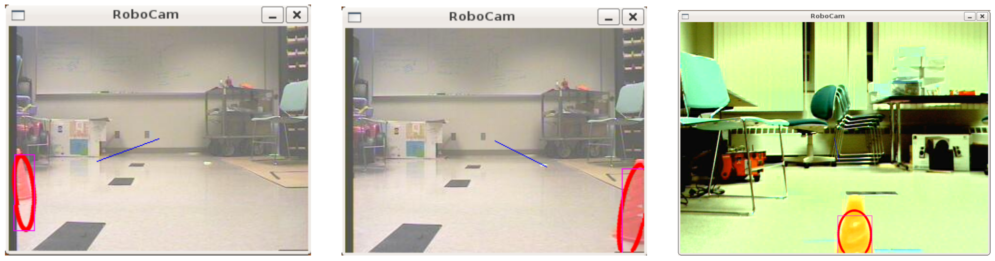


Figure 21: Reflective floors distort the object tracking and create inaccuracies



(a) The cone is too far left of the robot (b) The cone is too far right of the robot (c) The cone is too close to the robot

Figure 22: Distortions due to the object not being completely in the field of vision

## 7.3 Bad Histograms

One thing worth mentioning, is that the system is dependent on the quality of the histogram. This means that the inputed seed needs to include pixels of the image from many different locations in many different light environments. Bad histograms are those with little variation in color, and can cause the object to be lost when it is farther away or in dimmer or brighter light environments. An example of a histogram that has very little variation is shown in Fig. 23. Fig. 24 shows an example of an object tracked with this histogram that is too far away to register enough probability in the backproject.

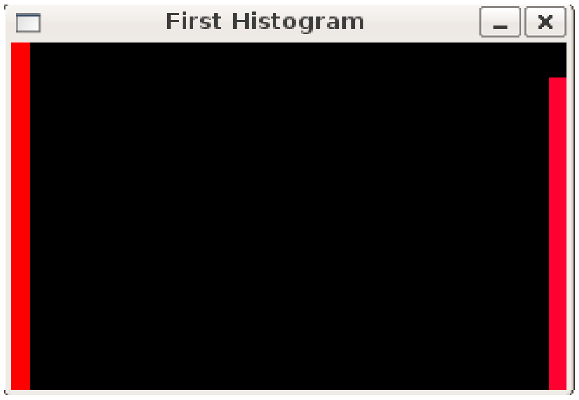


Figure 23: An example of a histogram with little variation

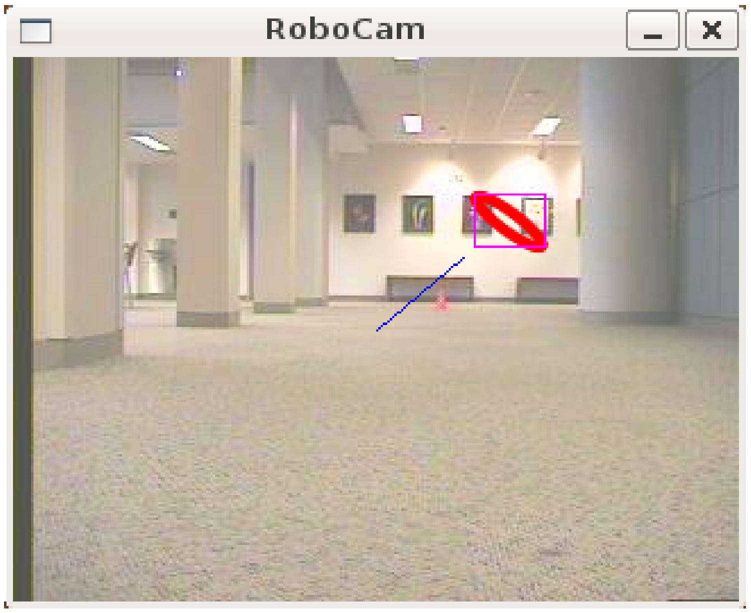


Figure 24: The cone is so far away that its probability in the backproject is less than the brightly lit wall

## 7.4 Cases Without Distortions

Let us now examine the results of one of the cases without any distortion. The case is for the Pioneer robot in the lab. Despite the presence of linoleum floors, the fact that they were white linoleum floors made them less reflective. An example of one frame is shown in Fig. 25. Notice that this frame shows some distortion, which happened infrequently for less-reflective floors. The data histogram for 1000 frames is shown in Fig. 26. On the $x$ axis is the difference between the measured distance and the estimated distance. Along the $y$ axis is the number of data points within that bin. Notice that an envelope of the histogram is a Gaussian distribution with a mean of 0.11 centimeters and a standard deviation of 3.83 centimeters. This shows that for this particular environment the vision application was able to accurately estimate distances with almost normal error.

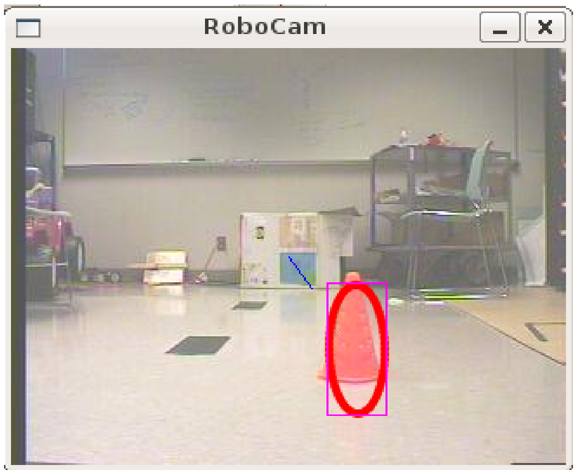


Figure 25: A frame from the lab environment with the cone 150 centimeters away from the Pioneer 2 robot

If we then aggregate all the data collected for both robots, and all distances and locations where distortions did not occur, we again see an almost Gaussian fit with a mean of -8.69 centimeters and a standard deviation of 24.54 centimeters. The histogram for this data is in Fig. 27.

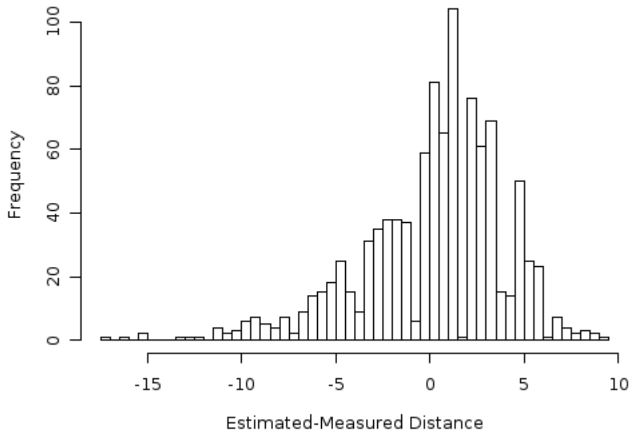


Figure 26: A frequency histogram of error for the lab environment at a distance of 150 centimeters without distortion

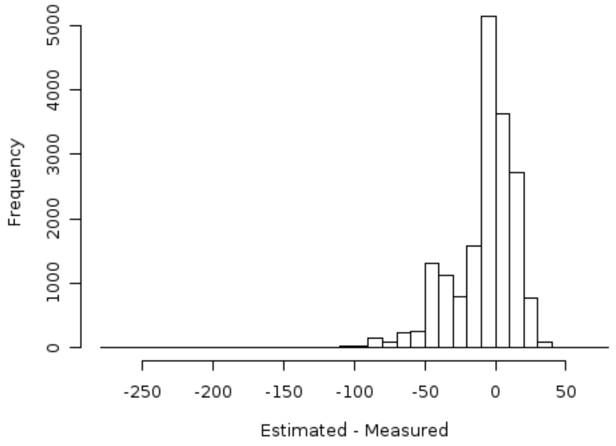


Figure 27: A frequency histogram of error for all environments and distances that did not show distortion

## 7.5 Cases With Distortions

We now examine a case where distortion caused the results to be unreliable. The environment is a hallway with green linoleum flooring. The robot is Borg, the power wheels robot with a standard webcam. The distance in this case is 500 centimeters. Notice from Fig. 28, that the application tracked the cone and its entire reflection, effectively doubling the height of the cone. Therefore we would expect the distance estimations to be shorter than the measured value because the cone appears larger, and thus closer. From all the cases we took data for, 32% of them contained distortion. This number only reflects our choice of environments, however it fairly demonstrates that these distortions are a significant source for error.

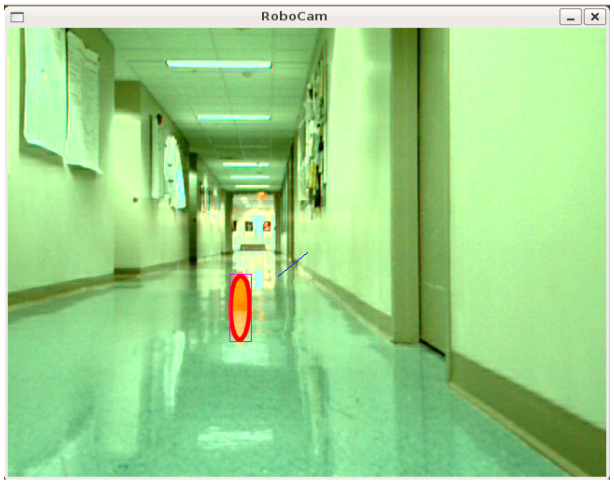


Figure 28: A frame from the hall environment with the cone 500 centimeters away from the Power Wheels robot

A histogram of the data taken for this one case is shown in Fig. 29. Notice that, although the data appears to be somewhat Gaussian, the mean is closer to -122 centimeters. Also, there are fewer histogram bars in this plot meaning that the search window jittered causing the distance estimations to jump.

A histogram of the error for all cases that had distortion is shown in Fig. 30. Notice that although some parts of this data appear Gaussian, values are still very far from zero and jump unreliably. Furthermore, most of the data is

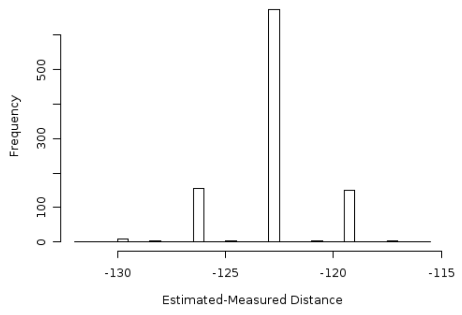


Figure 29: A frequency histogram of error for the hall environment at a distance of 500 centimeters with distortion

less than zero which means that the estimates were more often shorter than the measured value, in many times around 800 centimeters shorter.

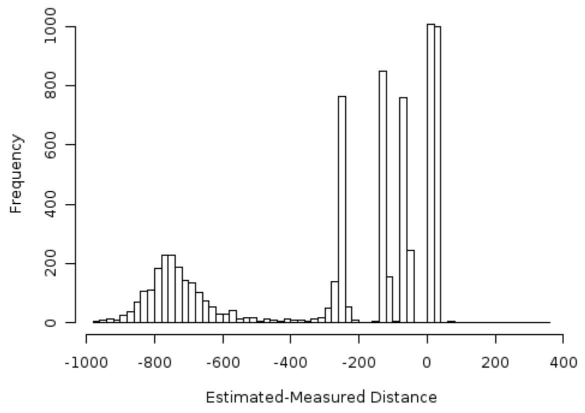


Figure 30: A frequency histogram of error for all environments and distances that showed distortion

## 7.6 All Cases

If we now make a histogram of all data, both with and without distortion, we see that the majority of the data showed good results. The application is able to estimate the distance of the object accurately when there is no distortion. Furthermore, the frequency of accurate results outweighed that of inaccurate results. See Fig. 31 for the histogram of all data.

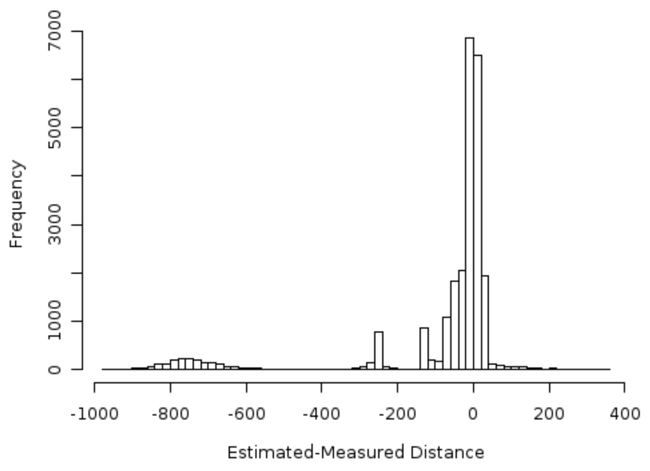


Figure 31: A frequency histogram of error for all environments and distances

# 8 Conclusion

The goal of this project was to create an application for object depth perception. The application we developed uses the Camshift algorithm to track objects based on their color, then passes the location and size of the object to the POSIT algorithm, which estimates the objects pose. Again, Camshift uses density gradient estimation to hill-climb its way to the object. POSIT uses a model of the object, algebraic techniques, and an iterative approach to more accurately estimate the pose of the object.

In order to use the application on video streams with frame rates upto 30 fps, we needed algorithms that could perform on a single frame and could complete their computation in at most a 30th of a second. Other solutions to object tracking and pose estimation, such as segmentation and genetic algorithms, were simply too slow or required multiple images. Thus, we chose Camshift and POSIT over other methods because of their relative speed in computation and the fact that efficient implementations of both are available in OpenCV. Camshift solves many problems associated with object tracking because of its adaptive search window, and POSIT is a fast, accurate algorithm that can be performed on a single image.

From the data we gathered, we have shown that in most of the cases, the application works well and returns accurate estimations of depth. However, when there are reflections or occlusions of the object, the application will return inaccurate data. This is unavoidable because of the nature of color based tracking. These distortions can greatly effect the ability of Camshift to track an object, because they change its perceived size. Furthermore, in order for POSIT to accurately estimate the distance to the object, Camshift must pass it accurate information about the size and location of the object. Therefore, the performance of Camshift has more effect on the overall performance of the system than that of POSIT.

In order to reliably integrate this application into the Monte Carlo Localizer we would need to ensure that no cone will have reflections or occlusions causing its perceived size to be inaccurate. A possible solution to this problem, that involves tracking features rather than color, is discussed in Section 9.2. Another solution may simply be to place non-reflective surfaces underneath the cones when positioning them around the map. We may also need to associate an error or probability with each distance estimation to smooth out the effect of bad estimations.

# 9 Future Work

## 9.1 Accurate Estimation of Focal Length

During our attempts to use camera calibration techniques to accurately estimate the focal length of the cameras, we were never able to obtain useful values. Furthermore, the POSIT function takes a single focal length parameter, while the camera calibration algorithms returned a horizontal and vertical focal length. The POSIT algorithm assumed square pixels when in fact the image pixels were rectangular. Because we are consistent, our false focal length works well. However this could be masking other inaccuracies of the system. Some work to further pursue would be to transform the images into square images, accurately estimate the focal length, and use this focal length to estimate pose.

## 9.2 Feature Tracking Algorithms

One other shortcoming of our method is how we match feature points to model points. We assume that the Camshift search window exactly encloses the cone which it does not always do. One way to improve the system might be to use visible features of the object, meaning features on the object that cause recognizable pixel intensity variation, such as a corner. This would also help solve the problems of distortions because the pose would be based on only visible feature points and would ignore color reflection off the floor.

One possible way to detect suitable features would be to use SIFT features that are trackable in the vicinity of the search window. SIFT, first developed by Lowe [28], stands for Scale-Invariant Feature Transform. SIFT features, as their name suggests, are invariant to scale and rotation. Therefore SIFT features are relatively robust in detection [28]. If we used model points for our cone that correspond to SIFT features, we might get a better estimation of pose from POSIT. One possible limitation of this is that there may not be enough SIFT features on a cone to obtain four non-coplanar points. We may have to choose a different object, while ensuring it is still highly contrastive to its environment.

## 9.3 Kalman Filter

A possible technique we can use to improve the measurements from the application is called a Kalman Filter, and is a mathematical technique to estimate the state of a linear dynamic system. The Kalman Filter is named after Rudolf E. Kalman, because of his seminal paper on the subject in 1960 [29]. The purpose of the Kalman Filter is to use observed measurements over time that contain noise and inaccuracies to produce values that tend closer to the true values of the measurements [30].

The Kalman filter combines the system's dynamics model, the known control inputs to the system, and sensor measurements to estimate the system's state. The system's dynamics model comes from properties such as physical laws of motion or other known properties of the system that can be used to predict the next value [2].

If the Kalman filter were to be used to improve our application, the system's state would be the location and size of the object being tracked. The system's dynamics model would use the fact that the location and size of the object should change proportionally to its speed and direction. Finally, the sensor measurements would be the location and size of the Camshift search window.

## 9.4 Integration of Vision Application into MCL

Finally, we need to integrate the vision application into the Monte Carlo Localizer, and test the improvement gained from the added sensors. This would require knowing the exact position of cones in the map of Olin-Rice, which we would measure manually. The basic idea is that the distance returned from the vision application would be sent to MCL, which would then look for those samples that have a cone in front of them at a similar distance.

Because the application does not return 100% accurate data, we would also want to incorporate some error estimation with the distances values. This would allow the localizer to smooth out any false distance estimations passed to it.

# A Source Code for the Capture Program

```
/****************************************************************************\
 *  Capture.h:  header of Capture.cpp:                                        *
 *  program to capture images from either a default webcam or                 *
 *  from files, and run several OpenCV algorithms on those images in order to *
 *  track objects by color and estimate their pose.                           *
 *  Written by Kayton Parekh and Susan Fox in Summer 2009                     *
\****************************************************************************/

#ifndef CAPTURE_H_
#define CAPTURE_H_

#include <cv.h>
#include <highgui.h>
#include <math.h>
#include <pthread.h>
#include <cstring>
#include <iostream>
#include <stdlib.h>
#include <dirent.h>
#include <fstream>
#include "ColorTracker.h"
#include "CameraColorTracker.h"
#include "FilesColorTracker.h"

#define SRC_CAMERA     5
#define SRC_FILES     10
using namespace std;

// Variables marked by **shared** below are shared by both threads and
// control MUST be mediated through a lock.

// Locks
pthread_mutex_t colorTrackerLock = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t stopLock = PTHREAD_MUTEX_INITIALIZER;

// Global variables
bool globStop = false;// **shared** sends when to quit b/w threads
bool verbose = false; // when to produce verbose output
bool logFlag = false; // if we want to log avi files
bool useMouseSelection = false;
int selectObject = 0;// when the mouse is selecting an object
int trackObject = 0;    // when an object is being tracked
int picSource = SRC_CAMERA;  // if we are using camera or files

// Used by Hough lines to store important lines
int lineTotal = 10;
int lineCount = 0;
CvPoint** importantLines = 0;

// Used to keep track of the folder and fileName
char picsFolder[100];
char picsFileName[100];

// Variables below are shared by both threads
```

```
char wndName[] = "Edge";
char tbarName[] = "Threshold";
int edgeThresh = 1;
int imageOrigin = 0;
CvPoint origin;
ColorTracker* colorTracker; // **shared**

// Variable used by display function
CvRect selection;

//END GLOBAL VARIABLES

// ===========================================================================
// DECLARATIONS OF FUNCTIONS

// edge detector functions
IplImage* edgeDetect( IplImage* image);
IplImage* lineDetect( IplImage* src);
bool betweenAngles(float theta, float lowBound, float highBound);
void setupHough();
void cleanupHough();

// I/O Functions
void* ioThread( void *ptr);
void updateSaveName(char *fileName, char* folder, char* wholepath);
void printVerbose(const string str);

// Display Functions
void* displayThread( void *ptr);
void onMouse( int event, int x, int y, int flags, void* param );

// END FUNCTION DECLARATIONS


#endif /* CAPTURE_H_ */

/***************************************************************************\
 *  Capture.cpp:  a program to capture images from either a default webcam or *
 *  from files, and run several OpenCV algorithms on those images in order to *
 *  track objects by color, and estimate their pose.                          *
 *  Written by Kayton Parekh and Susan Fox, Summer 2009                       *
\***************************************************************************/
#include "Capture.h"

/* ===========================================================================
 * This section contains the main function */
// ---------------------------------------------------------------------------

int main(int argc, char** argv ) {
    char csvFileName1[100], csvFileName2[100];
    bool loaded1 = true, loaded2 = true; // load success flags

    //argument flags
    bool doFile = false, seenFirst = false, seenSecond = false;

    for ( int i = 1; i < argc; i++ ) {
        // user entered -v on command-line
        if ( strcmp(argv[i], "-v") == 0 ) {
            verbose = true;
        }
        // user entered -f on command-line
        else if ( strcmp(argv[i], "-f") == 0 ) {
            doFile = true;
        }
        // user entered -fv OR -VF on command-line
        else if ( strcmp(argv[i], "-fv") == 0
                || strcmp(argv[i], "-vf") == 0 ) {
            doFile = true;
            verbose = true;
        }
        // this is the first csv filename we've seen
        else if ( !seenFirst ) {
            seenFirst = true;
            strcpy(csvFileName1, argv[i]);
        }
        // this is the second csv filename we've seen
        else if ( !seenSecond ) {
            seenSecond = true;
            strcpy(csvFileName2, argv[i]);
        }
        else {
            cerr << "Invalid or too much command-line input!" << endl;
            exit(0);
        }
    }

    // Set up the right kind of color tracker based on command line input
    if ( doFile ) {
        picSource = SRC_FILES;
        colorTracker = new FilesColorTracker();
    }
    else {
        colorTracker = new CameraColorTracker();
    }

    // Set up histogram(s) and related variables
    if ( !seenFirst ) {  // If no csv files input
        useMouseSelection = true;
    }
    else {              // one or more csv files were input
        loaded1 = colorTracker->loadHist(csvFileName1, OBJECT_FIRST);
        colorTracker->showHistImage(OBJECT_FIRST);
        if (! seenSecond ) {   // exactly one csv file was input
            colorTracker->resetWindow(OBJECT_FIRST);
        }
        else {            // two csv files were input
            colorTracker->trackTwoObjects(true);
            loaded2 = colorTracker->loadHist(csvFileName2, OBJECT_SECOND);
            if( strcmp( csvFileName1, csvFileName2 ) ) {
                colorTracker->showHistImage(OBJECT_SECOND);
            }
            colorTracker->resetWindow(OBJECT_BOTH);
        }
    }
    //if loading failed print error and exit
    // doesnt work because loading will be true no matter what
    // cant get try-catch working right.
    if(!loaded1){
        cerr << "ERROR: could not load first csv file" << endl;
    }
    if(!loaded2){
        cerr << "ERROR: could not load second csv file" << endl;
    }
    if(!loaded1 || !loaded2){
        exit(0);
    }

    // user input to determine if we log or not
    printVerbose("Log?(y/n): ");
    char log[10];
    cin >> log;
    if( strcmp(log, "y")==0 || strcmp(log,"Y")==0 ){
        logFlag = true;
    }

    // create the two threads
    pthread_t thread1, thread2;
    int threadval1, threadval2;

    threadval1 = pthread_create(&thread1, NULL, displayThread, NULL);
    threadval2 = pthread_create(&thread2, NULL, ioThread, NULL);

    // wait for the threads to end
    pthread_join( thread1, NULL);
    pthread_join( thread2, NULL);

    // Release the capture device housekeeping
    delete colorTracker;
    cerr << "finished main()" << endl;
    return(1);
}

/* ========================================================================
 * This section contains code for the input-output thread that
 * communicates through standard io */

// ------------------------------------------------------------------------
/* This function ignores its input, but it uses three
 * global variables, which need to be locked before using:
 * globStop: a boolean, that may be set to true by either thread
 * capture: the camera capture device
 * frame:  the current picture taken by the camera */
void* ioThread( void *ptr) {
    char wholeName[100];
    char folder[80] = "/home/macalester/Desktop/Capture/pictures/";
    char filenm[20] = "foo0000.jpg";
    bool stop;

    //LOCK: ------------------------------------
    pthread_mutex_lock( &stopLock );
```

```
stop = globStop;
pthread_mutex_unlock( &stopLock );
//UNLOCK: ---------------------------------

while( !stop ) {
    char userInput;
    if(cin.peek() != EOF){
        cin >> userInput;
        // q to quit
        if (userInput == 'q' || userInput == 'Q' || userInput == EOF) {
            //LOCK: -----------------------------------
            pthread_mutex_lock( &stopLock );
            globStop = true;
            pthread_mutex_unlock( &stopLock );
            //UNLOCK: ---------------------------------
        }
        // p to take a picture
        else if (userInput == 'p' || userInput == 'P') {
            updateSaveName(filenm, folder, wholeName);
            //LOCK: -----------------------------------
            pthread_mutex_lock( &colorTrackerLock );
            int retVal = cvSaveImage(
                    wholeName,
                    colorTracker->getCurFrame()
            );
            pthread_mutex_unlock( &colorTrackerLock );
            //UNLOCK: ---------------------------------
            if (!retVal) {
                cerr << "Could not save: " << wholeName << endl;
                cout << "NOFILE" << endl;
            }
            else {
                cout << wholeName << endl;
            }
        }
        // f to get the object info
        else if (userInput == 'f' || userInput == 'F') {
            string info;
            //LOCK: ------------------------------------
            pthread_mutex_lock( &colorTrackerLock );
            colorTracker->getObjectInfo( &info );
            pthread_mutex_unlock( &colorTrackerLock );
            //UNLOCK: ---------------------------------
            cout << info << endl;
        }
        // v to get the object vector
        else if (userInput == 'v' || userInput == 'V') {
            string info;
            //LOCK: ------------------------------------
            pthread_mutex_lock( &colorTrackerLock );
            colorTracker->getObjectVector( &info );
            pthread_mutex_unlock( &colorTrackerLock );
            //UNLOCK: ---------------------------------
            cout << info << endl;
        }
        // x to reset both object windows
        else if (userInput == 'x' || userInput == 'X') {
```

```
    //LOCK: ----------------------------------
    pthread_mutex_lock( &colorTrackerLock );
    //need code to find object
    colorTracker->resetWindow(OBJECT_BOTH);
    pthread_mutex_unlock( &colorTrackerLock );
    //UNLOCK: --------------------------------
    printVerbose("finding both");
}
// 1 to reset the 1st object window
else if (userInput == '1'){
    //LOCK: ----------------------------------
    pthread_mutex_lock( &colorTrackerLock );
    colorTracker->resetWindow(OBJECT_FIRST);
    pthread_mutex_unlock( &colorTrackerLock );
    //UNLOCK: --------------------------------
    printVerbose("finding 1st(red)");
}
//2 to reset the 2nd object window
else if (userInput == '2'){
    //LOCK: ------------------------------------
    pthread_mutex_lock( &colorTrackerLock );
    colorTracker->resetWindow(OBJECT_SECOND);
    pthread_mutex_unlock( &colorTrackerLock );
    //UNLOCK: ----------------------------------
    printVerbose("finding 2nd(blue)");
}
// + to increase the threshold on the first backproject
else if (userInput == '+'){
    //LOCK: ----------------------------------
    pthread_mutex_lock( &colorTrackerLock );
    colorTracker->incBackProjectThresh(OBJECT_FIRST);
    pthread_mutex_unlock( &colorTrackerLock );
    //UNLOCK: --------------------------------
}
// - to decrease the threshold on the first backproject
else if (userInput == '-'){
    //LOCK: ----------------------------------
    pthread_mutex_lock( &colorTrackerLock );
    colorTracker->decBackProjectThresh(OBJECT_FIRST);
    pthread_mutex_unlock( &colorTrackerLock );
    //UNLOCK: --------------------------------
}
// m to increase the threshold on the second backproject
else if (userInput == 'm' || userInput == 'M'){
    //LOCK: ----------------------------------
    pthread_mutex_lock( &colorTrackerLock );
    colorTracker->incBackProjectThresh(OBJECT_SECOND);
    pthread_mutex_unlock( &colorTrackerLock );
    //UNLOCK: --------------------------------
}
// l to increase the threshold on the second backproject
else if (userInput == 'l' || userInput == 'L'){
    //LOCK: ----------------------------------
    pthread_mutex_lock( &colorTrackerLock );
    colorTracker->decBackProjectThresh(OBJECT_SECOND);
    pthread_mutex_unlock( &colorTrackerLock );
    //UNLOCK: --------------------------------
```

```
        }
        // b to toggle between displaying backprojects or original image
        else if (userInput == 'b' || userInput == 'B' ){
            //LOCK: ------------------------------------------
            pthread_mutex_lock( &colorTrackerLock );
            colorTracker->toggleBackProjectMode();
            pthread_mutex_unlock( &colorTrackerLock );
            //UNLOCK: ----------------------------------------
        }
        // n to update frame in colorTracker,
        // only valid if getting frames from files.
        else if (picSource == SRC_FILES &&
                (userInput == 'n' || userInput == 'N')) {
            //LOCK: ----------------------------------
            pthread_mutex_lock( &colorTrackerLock );
            colorTracker->updateFrame();
            colorTracker->resetPoseAverage();
            pthread_mutex_unlock( &colorTrackerLock );
            //UNLOCK: --------------------------------
        }
        else {
            cerr << "Input was:  " << userInput << endl;
            cout <<  "Valid commands are:" << endl;
            cout <<  "\tq   to quit" << endl;
            cout <<  "\tp   to take a picture and save to a file" << endl;
            cout <<  "\tf   to find current object's location" << endl;
            cout <<  "\tx   to attempt to refind both objects" << endl;
            cout <<  "\t1   to attempt to refind first(red) object"
            << endl;
            cout <<  "\t2   to attempt to refind second(blue) object"
            << endl;
            cout <<  "\t+   to raise threshold of first(red) object"
            << endl;
            cout <<  "\t-   to raise threshold of first(red) object"
            << endl;
            cout <<  "\tm   to raise threshold of second(blue) object"
            << endl;
            cout <<  "\tl   to lower threshold of second(blue) object"
            << endl;
            cout <<  "\tb   to toggle between backProjects" << endl;
            if(picSource == SRC_FILES){
                cout <<  "\tn   to load next frame from files" << endl;
            }
        }
    }
    //LOCK: --------------------------------
    pthread_mutex_lock( &stopLock );
    stop = globStop;
    pthread_mutex_unlock( &stopLock );
    //UNLOCK: ------------------------------

} // end while

//LOCK: --------------------------------
pthread_mutex_lock( &stopLock );
globStop = true;
pthread_mutex_unlock( &stopLock );
```

```
    //UNLOCK: -------------------------------

    cerr << "shutdown IO" << endl;
    return ptr;
}   // end function

// -------------------------------------------------------------------------
/* decides what to name the file for the saving picture
 * Takes in strings fileName , folder, and wholepath.
 * counts how many files are in the folder and changes the filename
 * accordingly, then puts 'folder/filename' into wholepath.
 */
void updateSaveName(char *fileName, char* folder, char* wholepath) {
    DIR *pdir;
    struct dirent *pent;
    int count = 0;
    pdir = opendir(folder);
    if (!pdir) {
        cerr << "opendir failed for pictures\n" << endl;
        return;
    }
    do{
        pent = readdir(pdir);
        if(pent)
            count++;
    }while (pent);
    // we decrement2 so we don't count the files . and .. in our count
    count -= 2;
    closedir(pdir);

    int h, t, r;

    if (count < 1000) {
        h = count / 100;
        r = count % 100;
        t = r / 10;
        r = r % 10;

        fileName[4] = '0' + h;
        fileName[5] = '0' + t;
        fileName[6] = '0' + r;
    }
    else {
        fileName[4] = 'X';
        fileName[5] = 'X';
        fileName[6] = 'X';
    }
    strcpy(wholepath, folder);
    strcat(wholepath, fileName);
}

/* -------------------------------------------------------------------------
 * Printer function that only prints to stdout if set to verbose(-v)
 */
void printVerbose(const string str){
    if (verbose) {
        cout << str <<  endl;
    }
}


/* ==========================================================================
 * This section contains code for the display thread that shows a live
 * image in a window on the screen. Also handles communication with the
 * color tracking object.
 */
// --------------------------------------------------------------------------
void* displayThread( void *ptr) {

    bool stop;
    //IplImage* edgeDetectImg = NULL;
    IplImage* colorTrackedImg = NULL;
    //IplImage* newFrame = NULL;
    CvVideoWriter* videoWriter = NULL;
    char aviFileName[80];
    IplImage* frame;
    IplImage* frameCopy;

    // Create a window in which the captured images will be presented
    cvNamedWindow( "RoboCam", 1 );
    // determine name of avi file if we are logging
    if(logFlag)
    {
        // decide what to Name the avi file for the recording
        DIR *pdir;
        struct dirent *pent;
        int count = 0;
        pdir = opendir("/home/macalester/Desktop/Capture/captured");
        if (!pdir) {
            cerr << "opendir failed\n" << endl;
            return(0);
        }
        pent = readdir(pdir);
        while (pent) {
            count++;
            pent = readdir(pdir);
        }
        // we decrement 2 so we don't count the files . and ..
        count -= 2;
        closedir(pdir);
        sprintf(
                aviFileName,
                "/home/macalester/Desktop/Capture/captured/capture%d.mpg",
                count
        );
        printVerbose(aviFileName);
    }
    if(useMouseSelection){
        cvSetMouseCallback( "RoboCam", onMouse, 0 );
    }
    //LOCK: -----------------------------------
    pthread_mutex_lock( &stopLock );
    stop = globStop;
    //UNLOCK: ---------------------------------
```

```
pthread_mutex_unlock( &stopLock );
//setupHough();
// Show the image captured from the camera in the window and repeat
while( !stop ) {
    //LOCK: ---------------------------------------------------
    pthread_mutex_lock( &colorTrackerLock );
    if(picSource==SRC_CAMERA){
        colorTracker->updateFrame();
        colorTracker->resetPoseAverage();
    }
    frame = colorTracker->getCurFrame();
    pthread_mutex_unlock( &colorTrackerLock );
    //UNLOCK: -------------------------------------------------
    if( !frame ) {
        cerr << "ERROR: frame is null..." << endl;
        break;
    }
    if(!frameCopy){
        frameCopy = cvCreateImage( cvGetSize(frame), 8, 3 );
    }
    if(!useMouseSelection){
        trackObject = 1;
    }
    if(logFlag){
        if (!videoWriter) {
            videoWriter  = cvCreateVideoWriter(
                    aviFileName,
                    CV_FOURCC('M', 'P', '4', '2'),
                    30,
                    cvSize(frame->width,
                            frame->height)
            );
        }
    }
    if( trackObject ){
        if( trackObject < 0 ){
            //LOCK: ---------------------------------------------------
            pthread_mutex_lock( &colorTrackerLock );
            colorTracker->loadHistFromMouse(selection);
            colorTracker->showHistImage(OBJECT_FIRST);
            pthread_mutex_unlock( &colorTrackerLock );
            //UNLOCK: -------------------------------------------------
            trackObject = 1;
        }
        //call color track
        //LOCK: ---------------------------------------------------
        pthread_mutex_lock( &colorTrackerLock );
        colorTrackedImg = colorTracker->colorTrack();
        colorTracker->estimatePose(colorTrackedImg);
        pthread_mutex_unlock( &colorTrackerLock );
        //UNLOCK: -------------------------------------------------
        //edgeDetectImg = edgeDetect(frame);
        //cvOr(newFrame, colorTrackedImg, edgeDetectImg);


    }
    else{
            colorTrackedImg = frame;
        }
        if( selectObject && selection.width > 0 && selection.height > 0 )
        {
            cvCopy(frame, frameCopy, 0);
            colorTrackedImg = frameCopy;
            cvSetImageROI( colorTrackedImg, selection );
            cvXorS( colorTrackedImg, cvScalarAll(255), colorTrackedImg, 0 );
            cvResetImageROI( colorTrackedImg );
        }
        if(logFlag){
            cvWriteFrame( videoWriter,colorTrackedImg);
        }
        cvShowImage( "RoboCam", colorTrackedImg );
        // The following looks for key-presses in the window displaying
        // the image
        char c = cvWaitKey(10);
        if (c == 27 || c == 'q' || c == 'Q') {
            cout << c <<endl;
            break;
        }
        if(c == 'w' || c == 'W'){
            pthread_mutex_lock( &colorTrackerLock );
            colorTracker->incFocalLength(5);
            pthread_mutex_unlock( &colorTrackerLock );
        }
        if(c == 's' || c == 'S'){
            pthread_mutex_lock( &colorTrackerLock );
            colorTracker->decFocalLength(5);
            pthread_mutex_unlock( &colorTrackerLock );
        }
        //LOCK: ---------------------------------
        pthread_mutex_lock( &stopLock );
        stop = globStop;
        pthread_mutex_unlock( &stopLock );
        //UNLOCK: -------------------------------
    } // end while
    cvReleaseImage(&frameCopy);
    //LOCK: -----------------------------------
    pthread_mutex_lock( &stopLock );
    globStop = true;
    pthread_mutex_unlock( &stopLock );
    //UNLOCK: ---------------------------------
    cleanupHough();
    if(logFlag)
        cvReleaseVideoWriter( &videoWriter );
    cvDestroyWindow("RoboCam");
    cerr << "shutdown display" << endl;
    return ptr;
} // end function


// ---------------------------------------------------------------------------
/* onMouse: a call-back for mouse events in the main window, it
 * looks for the user to select a rectangular region, and then
 * uses that as the target colors for the CAMSHIFT algorithm,
 * and as the starting window for the target object.
 */
void onMouse( int event, int x, int y, int flags, void* param )
{
    //LOCK: ------------------------------------
    pthread_mutex_lock( &colorTrackerLock );
    IplImage* frame = colorTracker->getCurFrame();
    pthread_mutex_unlock( &colorTrackerLock );
    //UNLOCK: ----------------------------------

    /*    if( !image )selection
        return;
     */
    if( imageOrigin )
        y = frame->height - y;

    if( selectObject )
    {
        selection.x = MIN(x,origin.x);
        selection.y = MIN(y,origin.y);
        selection.width = selection.x + CV_IABS(x - origin.x);
        selection.height = selection.y + CV_IABS(y - origin.y);
        selection.x = MAX( selection.x, 0 );
        selection.y = MAX( selection.y, 0 );
        selection.width = MIN( selection.width, frame->width );
        selection.height = MIN( selection.height, frame->height );
        selection.width -= selection.x;
        selection.height -= selection.y;
    }

    switch( event )
    {
    case CV_EVENT_LBUTTONDOWN:
        origin = cvPoint(x,y);
        selection = cvRect(x,y,0,0);
        selectObject = 1;
        trackObject = 0;
        break;
    case CV_EVENT_LBUTTONUP:
        selectObject = 0;
        if( selection.width > 0 && selection.height > 0 )
            trackObject = -1;
        break;
    }
}



// ------------------------------------------------------------------------------
/* Helper to compute whether a given angle is between two boundaries
 * The input angles are given in degrees
 */
bool betweenAngles(float theta, float lowBound, float highBound) {
    return (theta > lowBound && theta < highBound);
}

// ------------------------------------------------------------------------------
```

```
void setupHough() {
    importantLines = new CvPoint*[lineTotal];
    for (int i = 0; i < lineTotal; i++) {
        importantLines[i] = new CvPoint[2];

    }
}

// ---------------------------------------------------------------------------
void cleanupHough() {
    for (int i = 0; i < lineTotal; i++) {
        delete importantLines[i];
    }
    delete importantLines;
}
```

# B Source Code for the ColorTracker Class

## ColorTracker

### ColorTracker.h

```
/*****************************************************************************\
 *         ColorTracker.h                                                    *
 *         Color-based tracking using CAMSHIFT algorithm, modified           *
 *         from the camshiftdemo.c that came with opencv                     *
 *         Written by Kayton Parekh and Susan Fox in Summer 2009             *
\*****************************************************************************/

#include <cv.h>
#include <highgui.h>
#include <vector>
#include <iostream>
#include <sstream>
#include <fstream>

#define OBJECT_FIRST    1
#define OBJECT_SECOND   2
#define OBJECT_BOTH     3

using namespace std;

extern void printVerbose(const string str);
#ifndef COLORTRACKER_H_
#define COLORTRACKER_H_

class ColorTracker{

protected:
    IplImage *frame;
    virtual void setupQueryFrame();
    void setupColorTrack();
    void setupPosit(double _focalLength);

private:
    CvConnectedComp trackComp, trackComp2;
    CvRect trackWindow, trackWindow2, selection;
    CvBox2D trackBox, trackBox2;
    IplImage *hsv, *hue, *mask;
    IplImage *backProject, *histImg,  *backProject2, *histImg2;
    CvHistogram *hist,  *hist2;
    int hDims;
    float hrangesArr[2];
    float* hranges;
    int vmin, vmax, smin;
    int  backProjectThresh1, backProjectThresh2;
    bool trackTwoObj;
    int lostObjectFlag; // **shared**  when it's lost the object
    int backProjectMode; // **shared** if we want to see backProject
    IplImage* image;
    float sum[3];
    int total;

    CvPOSITObject *positObject;
    std::vector<CvPoint3D32f> objectPoints;
    CvMatr32f rotationMatrix;
    CvVect32f translationVector;
    CvTermCriteria terminationCriteria;
    double focalLength;



    int needResetWindow();

    bool needSplitWindow(IplImage *thisBackProject,
            CvBox2D thisTrackBox,
            CvRect thisTrackWindow,
            int minProbThresh);

    // -----------------------------------------------------------------------
    // splitTrackWindow tries to split the track window in two either
    // vertically or horizontally and keeps the one with the larger area
    void splitTrackWindow(CvRect* trackWindow,
            CvBox2D* trackBox,
            IplImage* backProject );

    // -----------------------------------------------------------------------
    // hsv2rgb takes a floating point value that represents the hue
    // part of a pixel, and converts it to its rgp equivalent, with some
    // fixed saturation and brightness
    // taken straight from opencv camshiftdemo code
    CvScalar hsv2rgb( float hue );


    // -----------------------------------------------------------------------
    /* trackSecondObject, when called, blackens out first object found
     * and then finds a second object to be tracked
     */
    void trackSecondObject(IplImage* image);

public:
    ColorTracker();
    virtual ~ColorTracker();

    // ----------------------------------------------------------------------
    /* loadHist takes a fileName of a .csv file containing rgb values for
     * a object to be tracked. x is 1 or 2 corresponding to first or second
     * object. it then converts this data into a histogram that can be
     * used by openCV. x can be either OBJECT_FIRST or OBJECT_SECOND to
     * to determine which object this histogram will correspond to.
     */
    bool loadHist(char* fileName, int x = OBJECT_FIRST);
    void loadHistFromMouse(CvRect selection);
    void showHistImage(int object);
    void getObjectInfo(string* retVal);
    void trackTwoObjects(bool x);
    void resetWindow(int lostObject);
    virtual void updateFrame();
    IplImage* getCurFrame();
    void incBackProjectThresh(int object);
    void decBackProjectThresh(int object);
    void estimatePose(IplImage* frame);
    void getObjectVector( string* retPtr );
    void incFocalLength(int x);
    void decFocalLength(int x);
    void resetPoseAverage();

    // ----------------------------------------------------------------------
    // colorTrack takes an image and performs the CamShift algorithm on
    // it.  NOTE that it depends on global variables hist and trackWindow
    // (among others) retaining their values from one call to the next...
    IplImage* colorTrack();
    void toggleBackProjectMode();

};

#endif /* COLORTRACKER_H_ */
```

## ColorTracker.cpp

```
/******************************************************************************\
 *      ColorTracker.cpp                                                      *
 *      Color-based tracking object that uses the CAMSHIFT algorithm          *
 *      Also Implements Pose estimation using POSIT algorithm                 *
 *      Written by Kayton Parekh and Susan Fox in Summer 2009                 *
\******************************************************************************/

#include "ColorTracker.h"

// ----------------------------------------------------------------------------
// Constructor that simply assigns initial values to the global variables
ColorTracker::ColorTracker(){
    hDims = 32;
    hrangesArr[0] = 0;
    hrangesArr[1] = 180;
    hranges = hrangesArr;
    vmin = 230;
    vmax = 256;
    smin = 30;
    backProjectThresh1 = 100;
    backProjectThresh2 = 100;
    trackTwoObj = false;
    lostObjectFlag = 3; // when it's lost the object
    backProjectMode = 0; // if we want to see backProject
}
// ---------------------------------------------------------------------------
// requires a valid frame to run thus must call updateFrame to ensure this
// is used to set up global variables that will be needed for colortracking.
void ColorTracker::setupColorTrack() {
    updateFrame();
    histImg = cvCreateImage( cvSize(320,200), 8, 3 );
    histImg2 = cvCreateImage( cvSize(320,200), 8, 3 );
    hist = cvCreateHist(1, &hDims, CV_HIST_ARRAY, &hranges, 1);
    hist2 = cvCreateHist(1, &hDims, CV_HIST_ARRAY, &hranges, 1);
    hsv = cvCreateImage( cvGetSize(frame), 8, 3 );
    hue = cvCreateImage( cvGetSize(frame), 8, 1 );
    mask = cvCreateImage( cvGetSize(frame), 8, 1 );
    image = cvCreateImage(cvGetSize(frame), 8, 3);
    backProject = cvCreateImage( cvGetSize(frame), 8, 1 );
    backProject2 = cvCreateImage( cvGetSize(frame), 8, 1 );
}

/*
 * deconstructor simply releases several images, histograms\
 * and windows possibly held
 */
ColorTracker::~ColorTracker(){
    cvReleaseImage( &hsv );
    cvReleaseImage( &hue );
    cvReleaseImage( &mask );
    cvReleaseImage( &backProject );
    cvReleaseImage( &backProject2 );
    cvReleaseImage( &image );
    cvReleaseImage( &histImg );
    cvReleaseImage( &histImg2 );
    cvReleaseHist( &hist );
    cvReleaseHist( &hist2 );
    cvDestroyWindow( "First Histogram" );
    cvDestroyWindow( "Second Histogram" );
    cvReleasePOSITObject( &positObject );
}

/*
 * should be overridden by child class,
 * this method should contain the code to load
 * the next frame, either from files or from the capture object.
 */
void ColorTracker::updateFrame(){
}
```

```
// simple getter for the current frame
IplImage* ColorTracker::getCurFrame(){
    return frame;
}

// should be overriddern by child class, this method should contain the code to
// setup the objects needed to obtain frames, i.e. capture object
void ColorTracker::setupQueryFrame(){
}

/*
 * sets the value of retPtr to a string that is the location of the tracked
 * objects, if only one object is being tracked returns zeros for data on
 * second object
 */
void ColorTracker::getObjectInfo(string* retPtr){
    stringstream ss (stringstream::in | stringstream::out);
    ss << trackBox.center.x      << " ";
    ss << trackBox.center.y      << " ";
    ss << trackBox.size.width    << " ";
    ss << trackBox.size.height   << " ";
    ss << trackBox.angle         << " ";
    ss << frame->width           << " ";
    ss << frame->height          << " ";
    ss << trackBox2.center.x     << " ";
    ss << trackBox2.center.y     << " ";
    ss << trackBox2.size.width   << " ";
    ss << trackBox2.size.height  << " ";
    ss << trackBox2.angle;
    retPtr->assign( ss.str() );
}

// ---------------------------------------------------------------------------
/* loadHist takes a fileName of a .csv file containing rgb values for
 * an object to be tracked. x is OBJECT_FIRST or OBJECT_SECOND corresponding
 * to first or second objects. it then converts this data into a histogram that
 * can be used by openCV.
 */
bool ColorTracker::loadHist(char* fileName, int obj){
    ifstream csvFile;
    int r, g, b; //rgb values
    char comma; // useless holder for commas between r,g,b values
    int cnt = 0;
    IplImage *colorDataHsv, *colorDataHue, *colorDataMask, *colorData;

    // need try-catch IOExceptions, if fail then return false
    // try catch not working

    // count the number of data lines
    csvFile.open(fileName);
    csvFile >> vmin; // first line of csv file should be the vmin
    while (!csvFile.eof()) { // count how many sets of rgb values their are
        csvFile >> r >> comma >> g >> comma >> b;  //r,g,b
        cnt++;
    }
    csvFile.close();
    // create image that is one pixel high and "cnt" pixels wide
    colorData = cvCreateImage(cvSize(cnt, 1), 8, 3);

    // take each data line and convert it to a pixel in colorData
    csvFile.open(fileName);
    csvFile >> vmin;
    for (int i = 0; i < cnt; i++) {
        csvFile >> r >> comma >> g >> comma >> b;
        //cout << i << " " << r << " " << g << " " << b << endl;
        cvSet2D(colorData, 0, i, cvScalar(b, g, r, 0));
    }
    csvFile.close();

    //prepare to create histogram
    float maxVal = 0.f;
    colorDataHsv = cvCreateImage( cvGetSize(colorData), 8, 3 );
    colorDataHue = cvCreateImage( cvGetSize(colorData), 8, 1 );
    colorDataMask = cvCreateImage( cvGetSize(colorData), 8, 1 );

    cvCvtColor( colorData, colorDataHsv, CV_BGR2HSV );
    cvInRangeS( colorDataHsv, cvScalar(0,smin,MIN(vmin,vmax),0),
            cvScalar(180,256,MAX(vmin,vmax),0), colorDataMask );
    cvSplit( colorDataHsv, colorDataHue, 0, 0, 0 );

    //create histogram and assign it to hist or hist2 depending of value of obj
    if(obj == OBJECT_FIRST){
        cvCalcHist( &colorDataHue, hist, 0, colorDataMask );
        cvGetMinMaxHistValue( hist, 0, &maxVal, 0, 0 );
        cvConvertScale(
                hist->bins,
                hist->bins,
                (maxVal ? 255. / maxVal : 0.),
                0
        );
    }else if(obj == OBJECT_SECOND){
        cvCalcHist( &colorDataHue, hist2, 0, colorDataMask );
        cvGetMinMaxHistValue( hist2, 0, &maxVal, 0, 0 );
        cvConvertScale(
                hist2->bins,
                hist2->bins,
                (maxVal ? 255. / maxVal : 0.),
                0
        );
    }

    //release images used it the process
    cvReleaseImage(&colorDataHsv);
    cvReleaseImage(&colorDataHue);
    cvReleaseImage(&colorDataMask);
    cvReleaseImage(&colorData);

    return true; // we return true in the hope that someday we will have
    // mastered the use of try-catch statements in C++ and instead
    // return false if IOExceptions are caught.
}

// this method is used to load the histogram from a mouse selection rather
// than a csv file
void ColorTracker::loadHistFromMouse(CvRect selection){
    cvCvtColor( frame, hsv, CV_BGR2HSV );
    cvInRangeS( hsv, cvScalar(0,smin,MIN(vmin,vmax),0),
            cvScalar(180,256,MAX(vmin,vmax),0), mask );
    cvSplit( hsv, hue, 0, 0, 0 );
    float maxVal = 0.f;

    cvSetImageROI( hue, selection );
    cvSetImageROI( mask, selection );
    cvCalcHist( &hue, hist, 0, mask );
    cvGetMinMaxHistValue( hist, 0, &maxVal, 0, 0 );
    cvConvertScale(
            hist->bins,
            hist->bins,
            (maxVal ? 255. / maxVal : 0.),
            0
    );
    cvResetImageROI( hue );
    cvResetImageROI( mask );
    trackWindow = selection;
}

// simple setter for the trackTwoObjects flag, which determines if we are
void ColorTracker::trackTwoObjects(bool x){
    trackTwoObj = x;
}

// this method resets one or both of the trackwindows to the size of
// the entire window depending of the value of lostObject.
void ColorTracker::resetWindow(int lostObject){
    if (!lostObject){
        return;
    }
    if (lostObject == OBJECT_FIRST || lostObject == OBJECT_BOTH){
        trackWindow.x = 0;
        trackWindow.y = 0;
        trackWindow.width = frame->width;
        trackWindow.height = frame->height;
    }
    if(lostObject == OBJECT_SECOND || lostObject == OBJECT_BOTH){
        trackWindow2.x = 0;
        trackWindow2.y = 0;
        trackWindow2.width = frame->width;
        trackWindow2.height = frame->height;
    }
}

// ---------------------------------------------------------------------------
// colorTrack takes an image and performs the CamShift algorithm on
// it.  NOTE that it depends on global variables hist and trackWindow
// (among others) retaining their values from one call to the next...
// if trackTwoObjects then colorTrack will make a call to trackSecondObject()
IplImage* ColorTracker::colorTrack() {
    //make a copy of the frame that is in hsv mode
    image->origin = frame->origin;
    cvCopy(frame, image, 0);
```

```
cvCvtColor( image, hsv, CV_BGR2HSV );

// calculate mask based on smin, vmin, and vmax
cvInRangeS( hsv, cvScalar(0,smin,MIN(vmin,vmax),0),
        cvScalar(180,256,MAX(vmin,vmax),0), mask );
cvSplit( hsv, hue, 0, 0, 0 );
// calculate the backproject based on hue and hist
cvCalcBackProject( &hue, backProject, hist );
cvAnd( backProject, mask, backProject, 0 );

// help eliminate some noise by setting a thresh on backProject
cvThreshold(backProject, backProject,
        backProjectThresh1, 255, CV_THRESH_TOZERO);

// run the camshift
cvCamShift(
        backProject,
        trackWindow,
        cvTermCriteria(
                CV_TERMCRIT_EPS | CV_TERMCRIT_ITER,
                10,
                1
        ),
        &trackComp,
        &trackBox
);
trackWindow = trackComp.rect;

// see if we need to split the track window of the second object
if ( needSplitWindow(backProject, trackBox, trackWindow, 10) )
{
    printVerbose( "Splitting first trackWindow" );
    splitTrackWindow(&trackWindow, &trackBox, backProject);
}

//display backProjects if in that mode
if( backProjectMode == OBJECT_FIRST)
{
    cvCvtColor( backProject, image, CV_GRAY2BGR );
}
else if (backProjectMode == OBJECT_SECOND && trackTwoObj)
{
    cvCvtColor( backProject2, image, CV_GRAY2BGR );
}

// dont know what this does but got it from camshiftdemo code
if( !image->origin ) {
    trackBox.angle = -trackBox.angle;
    trackBox2.angle = -trackBox2.angle;
}

// if we are tracking two objects, call trackSecondObject
if(trackTwoObj)
{
    trackSecondObject(image);
}
```

```
    // now dray the trackWindow and
    // trackBox(oval shaped not really a box onto the image)
    cvEllipseBox( image, trackBox, CV_RGB(255,0,0), 3, CV_AA, 0 );
    cvRectangle (
            image,
            cvPoint(trackWindow.x,trackWindow.y),
            cvPoint(trackWindow.x+trackWindow.width,
                    trackWindow.y+trackWindow.height),
                    CV_RGB(255,0,255)
    );
    // determine if we have lost an object and
    // if so reset that objects trackWindow
    int x = needResetWindow();
    if(x){
        resetWindow(x);
    }

    return image;
}

// this method looks to see if either or both of the track objects are too small
// (meaning we have lost that object in the tracking) and returns integer
// corresponding to the result.
int ColorTracker::needResetWindow(){
    int x = 0;
    if( trackBox.size.width < 5 || trackBox.size.height < 5){
        x = OBJECT_FIRST;
        printVerbose("Reseting FIRST trackWindow...");

    }
    if(trackTwoObj){
        if( trackBox2.size.width < 5 || trackBox2.size.height < 5){
            if(x == OBJECT_FIRST)
                x = OBJECT_BOTH;
            else
                x = OBJECT_SECOND;

            printVerbose("Reseting SECOND trackWindow...");
        }
    }
    return x;
}

// this method creates a window to show the
// histogram of the first, second or both objects
void ColorTracker::showHistImage(int object){
    if(object == OBJECT_BOTH){
        showHistImage(OBJECT_FIRST);
        showHistImage(OBJECT_SECOND);

    }else{
        IplImage* histImage;
        CvHistogram* histogram;
        char windowName[25];
        int yVal;

        if(object == OBJECT_FIRST){
            strcpy(windowName,"First Histogram");
            yVal = 1;
            histImage = histImg;
            histogram = hist;
        }
        else if(object == OBJECT_SECOND){
            strcpy(windowName,"Second Histogram");
            yVal = 255;
            histImage = histImg2;
            histogram = hist2;
        }

        cvNamedWindow( windowName, 1 );
        cvMoveWindow( windowName, 650, yVal);
        // Initialize the histogram image
        cvZero( histImage );
        // Set the displayed width each bin of the histogram has
        int binWidth = histImage->width / hDims;
        // Add rectangles to the histimg that reflect the size of the bins
        // and the average color for each bin
        for( int i = 0; i < hDims; i++ )
        {
            int val = cvRound(
                    cvGetReal1D(histogram->bins,i) * histImage->height/255
            );

            CvScalar color = hsv2rgb(i * (180.f / hDims));
            cvRectangle( histImage,
                    cvPoint(i * binWidth, histImage->height),
                    cvPoint((i + 1) * binWidth, histImage->height - val),
                    color, CV_FILLED, 8, 0 );
        }
        //show the histImage
        cvShowImage( windowName, histImage );
    }
}

// this method is used to increment the
// backproject thresholds from outside the class.
void ColorTracker::incBackProjectThresh(int object){
    if(object == OBJECT_FIRST || object == OBJECT_BOTH){
        backProjectThresh1 += 5;
        backProjectThresh1 = ( backProjectThresh1 > 255) ? 255
                : backProjectThresh1;

        cout<< "thresh1: " << backProjectThresh1 << endl;
    }
    if(object == OBJECT_SECOND || object == OBJECT_BOTH){
        backProjectThresh2 += 5;
        backProjectThresh2 = ( backProjectThresh2 > 255) ? 255
                : backProjectThresh2;
        cout<< "thresh2: " << backProjectThresh2 << endl;
    }
}

// this method is used to decrement the
// backproject thresholds from outside the class.
```

```
void ColorTracker::decBackProjectThresh(int object){
    if(object == OBJECT_FIRST || object == OBJECT_BOTH){
        backProjectThresh1 -= 5;
        backProjectThresh1 = ( backProjectThresh1 < 0) ? 0
                : backProjectThresh1;

        cout<< "thresh1: " << backProjectThresh1 << endl;
    }
    if(object == OBJECT_SECOND || object == OBJECT_BOTH){
        backProjectThresh2 -= 5;
        backProjectThresh2 = ( backProjectThresh2 < 0) ? 0
                : backProjectThresh2;

        cout<< "thresh2: " << backProjectThresh2 << endl;
    }
}



// ---------------------------------------------------------------------------
/* trackSecondObject, when called, blackens out first object found
 * and then finds a second object to be tracked
 */
void ColorTracker::trackSecondObject(IplImage* image) {
    //this next part if we are tracking different histogram
    cvCalcBackProject( &hue, backProject2, hist2 );
    cvAnd( backProject2, mask, backProject2, 0 );
    cvThreshold(
            backProject2,
            backProject2,
            backProjectThresh2,
            255,
            CV_THRESH_TOZERO
    );
    // Show the backProject pixels in orange
    //cvSet(image, cvScalar(0, 255, 255, 0), backProject2);

    // now black out the track box1 and try to track another object

    // find coordinates of top left and bottom right
    // points of a rectangle to black out but
    // make it slightly bigger than the first trackWindow to avoid error
    float tmpX1 = trackWindow.x - trackWindow.width / 2;
    float tmpY1 = trackWindow.y - trackWindow.height / 2;
    float tmpX2 = trackWindow.x + 1.5 * trackWindow.width;
    float tmpY2 = trackWindow.y + 1.5 * trackWindow.height;

    // test if we have gone out of bounds of the image and correct if so
    tmpX1 = (tmpX1 < 0) ? 0 : tmpX1;
    tmpY1 = (tmpY1 < 0) ? 0 : tmpY1;
    tmpX2 = (tmpX2 > frame->width) ? frame->width : tmpX2;
    tmpY2 = (tmpY2 > frame->height) ? frame->height : tmpY2;

    // black out the area on the second objects backproject
    cvRectangle (
            backProject2,
            cvPoint(tmpX1,tmpY1),
            cvPoint(tmpX2,tmpY2),
            cvScalar(0,0,0,0),
            CV_FILLED
    );
    //run cam shift for this second object
    cvCamShift(
            backProject2,
            trackWindow2,
            cvTermCriteria( CV_TERMCRIT_EPS | CV_TERMCRIT_ITER,
                    10,
                    1
            ),
            &trackComp2,
            &trackBox2
    );
    trackWindow2 = trackComp2.rect;

    // see if we need to split the track window of the second object
    if ( needSplitWindow(backProject2, trackBox2, trackWindow2, 10) ) {
        printVerbose( "Probability fits on second box  " );
        splitTrackWindow(&trackWindow2, &trackBox2, backProject2);
    }

    // draw the trackWindow and trackBox onto the image
    cvEllipseBox( image, trackBox2, CV_RGB(0,0,255), 3, CV_AA, 0 );
    cvRectangle (
            image,
            cvPoint(trackWindow2.x,trackWindow2.y),
            cvPoint(trackWindow2.x+trackWindow2.width,
                    trackWindow2.y+trackWindow2.height),
                    CV_RGB(255,0,255)
    );
}

// ---------------------------------------------------------------------------
/* Compute the average probability for a 5x5 region around
 * the center of the trackWindow, to see if the center is
 * not on a matching color.  If it is not on a matching
 * color, then the track window might be streched around two different
 * objects */
bool ColorTracker::needSplitWindow(IplImage *thisBackProject,
        CvBox2D thisTrackBox,
        CvRect thisTrackWindow,
        int minProbThresh) {
    int cnt = 0;
    double prob = 0;
    float yPos, xPos;
    float centerX = thisTrackBox.center.x;
    float centerY = thisTrackBox.center.y;
    //    float height = thisTrackBox.size.height;
    //    float width = thisTrackBox.size.width;
    //    float dW=0;
    //    float dH=0;
    //
    //    // this following conditional is to determine the test region depending
    //    // on the orientation of the window
    //    if(height < width){
```

```
    //        dW = 2;
    //        dH = (int)(height)/2;;
    //    }else if(height >= width){
    //        dW = (int)(width)/2;;
    //        dH = 2;
    //    }

    // nested for loops to calulate avg prob at center of trackBox
    for (int dx = -2; dx < 3; dx++) {
        for (int dy = -2; dy < 3; dy++) {
            yPos = centerY + dy;
            xPos = centerX + dx;
            if (    yPos >= 0
                    && yPos < frame->height
                    && xPos >= 0
                    && xPos < frame->width    ) {
                prob = prob + cvGetReal2D(thisBackProject,
                        centerY + dy,
                        centerX + dx);
                cnt++;
            }
        }
    }
    prob = prob / cnt;

    // if center pixel is not the probability we want...
    if (prob < minProbThresh &&
            (thisTrackBox.size.width >= 5 || thisTrackBox.size.height >= 5)
    )
    {
        return true;
    }
    else{
        return false;
    }
}


// ---------------------------------------------------------------------------
// splitTrackWindow tries to split the track window in two either
// vertically or horizontally and keeps the one with the larger area
void ColorTracker::splitTrackWindow(CvRect* trackWindow,
        CvBox2D* trackBox,
        IplImage* backProject ) {
    float angle = trackBox->angle;
    CvRect trackA;
    CvRect trackB;
    CvBox2D trackBoxA, trackBoxB;
    CvConnectedComp trackCompA, trackCompB;
    char aString[6];
    char bString[6];

    trackA.x = trackWindow->x;
    trackA.y = trackWindow->y;

    if (abs((int)angle) < 45) { // trackBox is more or less horizontal
        printVerbose( "Splitting horizontal track window");
        strcpy(aString, "left");
        strcpy(bString, "right");
        trackA.width = max(1, trackWindow->width / 2);
        trackA.height = trackWindow->height;
        trackB.x = trackWindow->x + (trackWindow->width / 2);
        trackB.y = trackWindow->y;
        trackB.width = max(1, trackWindow->width / 2);
        trackB.height = trackWindow->height;
    }
    else {
        printVerbose( "Splitting vertical track window");
        strcpy(aString, "upper");
        strcpy(bString, "lower");
        trackA.width = trackWindow->width ;
        trackA.height = max(1,trackWindow->height / 2);
        trackB.x = trackWindow->x;
        trackB.y = trackWindow->y + (trackWindow->height / 2);
        trackB.width = trackWindow->width;
        trackB.height = max(1, trackWindow->height / 2);
    }

    cvCamShift(
            backProject,
            trackA,
            cvTermCriteria(
                    CV_TERMCRIT_EPS | CV_TERMCRIT_ITER,
                    10,
                    1
            ),
            &trackCompA,
            &trackBoxA
    );
    trackA = trackCompA.rect;
    cvCamShift(
            backProject,
            trackB,
            cvTermCriteria(
                    CV_TERMCRIT_EPS | CV_TERMCRIT_ITER,
                    10,
                    1
            ),
            &trackCompB,
            &trackBoxB
    );
    trackB = trackCompB.rect;

    /* pick the larger trackBox and set trackWindow to be
     the corresponding window*/
    float areaA = trackA.width * trackA.height;
    float areaB = trackB.width * trackB.height;
    // print out which half was kept:
    if (areaA == areaB) {
        printVerbose("SAME AREAS");
    }
    char tmp1[30] = "    Keeping ";
    char tmp2[10] = " window";
    if (areaA > areaB) {
        strcat(tmp1,aString);
        strcat(tmp1,tmp2);
        printVerbose(tmp1);
        *trackWindow = trackA;
        *trackBox = trackBoxA;
    }
    else {
        strcat(tmp1,bString);
        strcat(tmp1,tmp2);
        printVerbose(tmp1);
        *trackWindow = trackB;
        *trackBox = trackBoxB;
    }
}

// ---------------------------------------------------------------------------
/* hsv2rgb takes a floating point value that represents the hue
 * part of a pixel, and converts it to its rgp equivalent, with some
 * fixed saturation and brightness
 */
CvScalar ColorTracker::hsv2rgb( float hue )
{
    int rgb[3], p, sector;
    static const int sectorData[][3]=
    {{0,2,1}, {1,2,0}, {1,0,2}, {2,0,1}, {2,1,0}, {0,1,2}};
    hue *= 0.033333333333333333333333333333333f;
    sector = cvFloor(hue);
    p = cvRound(255*(hue - sector));
    p ^= sector & 1 ? 255 : 0;

    rgb[sectorData[sector][0]] = 255;
    rgb[sectorData[sector][1]] = 0;
    rgb[sectorData[sector][2]] = p;

    return cvScalar(rgb[2], rgb[1], rgb[0],0);
}

/* used to toggle the backProjectMode which is used to determine whether to
 * display the backproject or hide it.
 * Because OBJECT_FIRST and OBJECT_SECOND are defined to 1 and 2 respectively
 * using modulo arithmetic works nicely.
 */
void ColorTracker::toggleBackProjectMode(){
    int modulo;
    if(trackTwoObj){
        modulo = 3;
    }
    else{
        modulo = 2;
    }
    backProjectMode = (backProjectMode + 1) % modulo;
}

void ColorTracker::setupPosit(double _focalLength){
    objectPoints.push_back( cvPoint3D32f( 0.0f, 0.0f, 0.0f )        );
    objectPoints.push_back( cvPoint3D32f( 0.0f, 11.5f, 3.75f )     );
    objectPoints.push_back( cvPoint3D32f( -7.5f, -11.5f, 3.75f    )     );
    objectPoints.push_back( cvPoint3D32f( 7.5f, -11.5f, 3.75f )       );
    objectPoints.push_back( cvPoint3D32f( -7.5f, 11.5f, -3.75f )       );

    positObject = cvCreatePOSITObject(
            &objectPoints[0],
            static_cast<int>(objectPoints.size()) );
    rotationMatrix = new float[9];
    translationVector = new float[3];
    terminationCriteria = cvTermCriteria(
            CV_TERMCRIT_EPS | CV_TERMCRIT_ITER,
            100,
            1.0e-4f
    );
    focalLength = _focalLength;
    sum[0] = 0.0f;
    sum[1] = 0.0f;
    sum[2] = 0.0f;
    total = 0.0;

}
void ColorTracker::resetPoseAverage(){
    sum[0] = 0.0f;
    sum[1] = 0.0f;
    sum[2] = 0.0f;
    total = 0.0;
}

void ColorTracker::estimatePose(IplImage* image){
    float centreX = 0.5 * image->width;
    float centreY = 0.5 * image->height;
    float trackWinTransX = trackWindow.x - centreX;
    float trackWinTransY = centreY - trackWindow.y;
    float objWidth = trackWindow.width;
    float objHeight = trackWindow.height;

    std::vector<CvPoint2D32f> imagePoints;

    imagePoints.push_back(cvPoint2D32f(
            trackWinTransX + 0.5 * objWidth,
            trackWinTransY - 0.5 * objHeight
    ));
    imagePoints.push_back(cvPoint2D32f(
            trackWinTransX + 0.5 * objWidth,
            trackWinTransY
    ));
    imagePoints.push_back(cvPoint2D32f(
            trackWinTransX,
            trackWinTransY - objHeight
    ));
    imagePoints.push_back(cvPoint2D32f(
            trackWinTransX + objWidth,
            trackWinTransY - objHeight
    ));
    imagePoints.push_back(cvPoint2D32f(
            trackWinTransX,
            trackWinTransY
    ));

    cvPOSIT(
            positObject,
            &imagePoints[0],
            focalLength,
            terminationCriteria,
            rotationMatrix,
            translationVector
    );
    sum[0]+= translationVector[0];
    sum[1]+= translationVector[1];
    sum[2]+= translationVector[2];
    total++;
    cvLine(    image,
            cvPoint(centreX,centreY),
            cvPoint(centreX +translationVector[0], // wrong scale: centre is
                    centreY -translationVector[1]),// in pixels & vector is cm.
                    cvScalar(255,0,0)
    );


}

/* sets the value of retPtr to a string that is the
 * vector location of the first tracked object
 */
void ColorTracker::getObjectVector( string* retPtr ){
    if(total == 0){
        estimatePose(image);
    }
    stringstream ss (stringstream::in | stringstream::out);
    ss << sum[0]/total << " ";
    ss << sum[1]/total << " ";
    ss << sum[2]/total;
    retPtr->assign( ss.str() );
}

// the following are for debugging purposes

void ColorTracker::incFocalLength(int x){
    focalLength += x;
    cerr << "Flen: " << focalLength << endl;
}

void ColorTracker::decFocalLength(int x){
    focalLength -=x;
    cerr << "Flen: " << focalLength << endl;
}
```

## FilesColorTracker

### FilesColorTracker.h

```
/***************************************************************************\
 *        FilesColorTracker.h: header for FilesColorTracker.cpp             *
 *        Written by Kayton Parekh and Susan Fox in Summer 2009             *
\***************************************************************************/


#ifndef FILESCOLORTRACKER_H_
#define FILESCOLORTRACKER_H_

#include "ColorTracker.h"
#define SONY_FOCAL_LENGTH    410.0

class FilesColorTracker: public ColorTracker {

public:
    FilesColorTracker();
    virtual ~FilesColorTracker();
    void updateFrame();

private:
    char folder[80];
    char filename[50];
    char wholename[100];
    int imgCnt;

    void setupQueryFrame();
    void updateLoadName(char *name, int num);

};

#endif /* FILESCOLORTRACKER_H_ */
```

### FilesColorTracker.cpp

```
/***************************************************************************\
 *        FilesColorTracker.cpp: child of ColorTracker.cpp                  *
 *        color tracker object that takes frames from files on Hard Drive   *
 *        Written by Kayton Parekh and Susan Fox in Summer 2009             *
\***************************************************************************/


#include "FilesColorTracker.h"
// FilesColorTracker is a child class of ColorTracker

FilesColorTracker::FilesColorTracker() {
    setupQueryFrame();
    setupColorTrack();
    setupPosit(SONY_FOCAL_LENGTH);
}
```

```
FilesColorTracker::~FilesColorTracker() {}

// setup query frame sets the file path and name
void FilesColorTracker::setupQueryFrame(){
    imgCnt = 0;
    strcpy(filename,"foo0000.jpeg");
    strcpy(folder,"/home/kparekh/Desktop/RupartPyrobot/camera/PICS/");
}

/* update frame gets the next picture file from the folder path
 * is coded to simply increment the number in file name and try
 * load that file. if files go foo0000.jpeg to foo0002.jpeg this
 * will try to load foo0001.jpeg and will fail.
 */
void FilesColorTracker::updateFrame(){
    updateLoadName(filename, imgCnt++);
    strcpy(wholename, folder);
    strcat(wholename, filename);
    cvReleaseImage(&frame);
    frame = cvLoadImage( wholename, 1 );
    int i;
    for(i=0; i < 20 && !frame; i++){
        cerr <<"FilesColorTracker.cpp ERROR: no file ";
        cerr << filename << " found" <<endl;
        updateLoadName(filename, imgCnt++);
        strcpy(wholename, folder);
        strcat(wholename, filename);
        frame = cvLoadImage( wholename, 1 );
    }
    if(!frame && i == 20){
        cerr << "ERROR: 20 bad filenames" << endl;
        exit(0);
    }
    //cerr<<"loaded"<<endl;
}

/* Takes in a filename string and an integer
 * and incorporates the integer into the string
 * need to find a better way to do this!!
 */
void FilesColorTracker::updateLoadName(char *name, int num) {
    int th, h, t, r;
    if (num < 10000) {
        th = num / 1000;
        r = num % 1000;
        h = r / 100;
        r = r % 100;
        t = r / 10;
        r = r % 10;

        name[3] = '0' + th;
        name[4] = '0' + h;
        name[5] = '0' + t;
        name[6] = '0' + r;
    }
    else {
        name[3] = 'X';
        name[4] = 'X';
        name[5] = 'X';
        name[6] = 'X';
    }
}
```

## CameraColorTracker

### CameraColorTracker.h

```
/******************************************************************************\
 *        CameraColorTracker.h: header for CameraColorTracker.cpp           *
 *        Written by Kayton Parekh and Susan Fox in Summer 2009             *
\******************************************************************************/

#ifndef CAMERACOLORTRACKER_H_
#define CAMERACOLORTRACKER_H_

#include "ColorTracker.h"
#define PHILIPS_FOCAL_LENGTH    1300.0

class CameraColorTracker: public ColorTracker {

public:
    CameraColorTracker();
    virtual ~CameraColorTracker();
    void updateFrame();

private:
    CvCapture *capture;
    void setupQueryFrame();

};

#endif /* CAMERACOLORTRACKER_H_ */
```

### CameraColorTracker.cpp

```
/******************************************************************************\
 *        CameraColorTracker.cpp: child of ColorTracker.cpp                 *
 *        color tracker object that takes frames from a camera video stream  *
 *        Written by Kayton Parekh and Susan Fox in Summer 2009             *
\******************************************************************************/

#include "CameraColorTracker.h"
// CameraColorTracker is a child class of ColorTracker

CameraColorTracker::CameraColorTracker() {
    setupQueryFrame();
    setupColorTrack();
    setupPosit(PHILIPS_FOCAL_LENGTH);
}

CameraColorTracker::~CameraColorTracker() {
    cvReleaseCapture( &capture );
}

// setup Query Frame creates the capture object to get frames from the camera
void CameraColorTracker::setupQueryFrame(){
    // connect to any available camera
    capture = cvCreateCameraCapture( CV_CAP_ANY );
    if( !capture ) {
        cout << "ERROR: capture is NULL" << endl;
        exit(0);
    }

    // Set resolution on the camera
    cvSetCaptureProperty(capture, CV_CAP_PROP_FRAME_WIDTH, 640);
    cvSetCaptureProperty(capture, CV_CAP_PROP_FRAME_HEIGHT, 480);
}

// updateFrame gets the next frame from the capture object
void CameraColorTracker::updateFrame(){
    frame = cvQueryFrame( capture );
}
```

# References

[1] G. Bradski, "Computer vision face tracking for use in a perceptual user interface," *Intel Technology Journal*, no. Q2, 1998. [Online]. Available: download.intel.com/technology/itj/q21998/pdf/camshift.pdf

[2] G. Bradski and A. Kaehler, *Learning OpenCV: Computer Vision with the OpenCV Library*, 1st ed. O'Reilly Media, Sep. 2008.

[3] D. Comaniciu and P. Meer, "Mean shift analysis and applications," in *Computer Vision, 1999. The Proceedings of the Seventh IEEE International Conference on*, vol. 2, 1999, pp. 1197–1203 vol.2.

[4] D. Comaniciu, V. Ramesh, and P. Meer, "Kernel-based object tracking," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 25, no. 5, pp. 564–577, 2003.

[5] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 2nd ed. Prentice Hall, Dec. 2002.

[6] D. DeMenthon and L. S. Davis, "Model-Based object pose in 25 lines of code," in *ECCV '92: Proceedings of the Second European Conference on Computer Vision.* London, UK: Springer-Verlag, 1992.

[7] S. Thrun, W. Burgard, and D. Fox, *Probabilistic Robotics.* The MIT Press, Sep. 2005.

[8] S. Stamenova, "Solving the maze: Robot localization using the monte carlo localization algorithm and shape context," Undergraduate Thesis, Macalester College, Apr. 2009.

[9] P. Anderson-Sprecher, "Probabilistic robot localization using visual landmarks," Undergraduate Thesis, Macalester College, May 2006.

[10] L. G. Shapiro and G. C. Stockman, *Computer Vision.* Prentice Hall, Feb. 2001.

[11] J. R. Janesick, *Scientific charge-coupled devices.* SPIE Press, 2001.

[12] D. A. Forsyth and J. Ponce, *Computer Vision: A Modern Approach*, US

ed ed. Prentice Hall, Aug. 2002.

[13] R. Fuentes-Pavon, M. Aleman-Flores, L. Alvarez-Leon, P. Aleman-Flores, and J. Santana-Montesdeoca, "Computer vision techniques for breast tumor ultrasound analysis," *Breast Journal*, vol. 14, no. 5, pp. 483–486, 2008.

[14] A. Tighe, "UK develops 'intelligent CCTV'," http://news.bbc.co.uk/2/hi/uk_news/8561367.stm, Mar. 2010.

[15] S. Thrun, M. Montemerlo, H. Dahlkamp, D. Stavens, A. Aron, J. Diebel, P. Fong, J. Gale, M. Halpenny, G. Hoffmann, K. Lau, C. Oakley, M. Palatucci, V. Pratt, P. Stang, S. Strohband, C. Dupont, L. Jendrossek, C. Koelen, C. Markey, C. Rummel, J. van Niekerk, E. Jensen, P. Alessandrini, G. Bradski, B. Davies, S. Ettinger, A. Kaehler, A. Nefian, and P. Mahoney, "Stanley: The robot that won the DARPA grand challenge," *Journal of Field Robotics*, vol. 23, no. 9, pp. 661–692, 2006. [Online]. Available: http://dx.doi.org/10.1002/rob.20147

[16] M. Corporation, "Project natal," http://www.xbox.com/en-us/live/projectnatal/, 2010. [Online]. Available: http://www.xbox.com/en-us/live/projectnatal/

[17] B. Johnson, "Newly asked questions: How exactly does microsoft's project natal work?" http://www.guardian.co.uk/technology/gamesblog/2009/jun/04/microsoft-project-natal-xbox, Jun. 2009. [Online]. Available: http://www.guardian.co.uk/technology/gamesblog/2009/jun/04/microsoft-project-natal-xbox

[18] A. Yilmaz, O. Javed, and M. Shah, "Object tracking: A survey," *ACM Computer Surveys*, vol. 38, no. 4, Dec. 2006.

[19] D. Comaniciu and P. Meer, "Mean shift: a robust approach toward feature space analysis," in *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 24, 2002, pp. 603–619.

[20] Z. Wu, "An optimal graph theoretic approach to data clustering: Theory and its application to image segmentation," *IEEE Transactions On*

*Pattern Analysis and Machine Intelligence*, vol. 15, no. 11, p. 1101, 1993.

[21] J. Shi, *Normalized cuts and image segmentation.* Berkeley Calif.: University of California Berkeley Computer Science Division, 1997.

[22] B. D. Lucas and T. Kanade, “An iterative image registration technique with an application to stereo vision,” 1981, pp. 674—679.

[23] F. Toyama, K. Shoji, and J. Miyamichi, “Model-Based pose estimation using genetic algorithm,” in *Pattern Recognition, International Conference on*, vol. 1. Los Alamitos, CA, USA: IEEE Computer Society, 1998, p. 198.

[24] Y. Cheng, “Mean shift, mode seeking, and clustering,” *IEEE transactions on pattern analysis and machine intelligence*, vol. 17, no. 8, p. 790, 1995.

[25] R. Mukundan and K. R. Ramakrishnan, *Moment functions in image analysis : theory and applications.* Singapore ; River Edge NJ: World Scientific, 1998.

[26] K. Fukunaga and L. Hostetler, “The estimation of the gradient of a density function, with applications in pattern recognition,” *IEEE Transactions on Information Theory*, vol. 21, no. 1, pp. 32–40, 1975.

[27] Trucco and A. Verri, *Introductory Techniques for 3-D Computer Vision.* Prentice Hall, Mar. 1998.

[28] D. Lowe, “Distinctive image features from Scale-Invariant keypoints,” *International Journal of Computer Vision*, vol. 60, no. 2, pp. 91–110, Nov. 2004.

[29] R. Kalman, “A new approach to linear filtering and prediction problems,” *Transactions of the ASME – Journal of Basic Engineering*, no. 82 (Series D), pp. 45, 35, 1960. [Online]. Available: http://www.cs.unc.edu/ welch/kalman/media/pdf/Kalman1960.pdf

[30] G. Welch and G. Bishop, “An introduction to the kalman filter,” University of North Carolina at Chapel Hill, Chapel Hill, NC 27599-3175, Tech. Rep. TR 95-041, Jul. 2006. [Online]. Available: http://www.cs.unc.edu/ welch/kalman/kalmanIntro.html