

Robotic Vision Systems and Intel's OpenCV Framework

Overview.....	3
Comparison of Robotic Vision Systems.....	3
First Generation: AVRCam, CMUCam I + II.....	3
Second Generation: CMUCam III, Surveyor Blackfin Camera.....	3
Custom x86-Based.....	4
Embedded x86 Computers.....	4
Choosing an x86 Co-Processor.....	4
Using a Laptop.....	4
Assembling Your Own.....	4
Motherboard/Processor.....	4
RAM.....	5
Hard Drive.....	5
Power source.....	5
Choice of Operating System.....	5
Camera.....	6
Introduction to OpenCV.....	6
Overview.....	6
How to get it.....	6
Where To Get More Info.....	6
Basic Concepts of OpenCV.....	6
Structures.....	6
IplImage.....	6
Other structures.....	7
Allocation Convention.....	7
Image Parameters.....	7
OpenCV's Packages.....	7
Getting Started with OpenCV – Acquiring an Image.....	8
Installing OpenCV.....	8
Compilation Requirements.....	8
Basic program structure.....	8
HighGUI Capture.....	8
Capturing an Image.....	9
Displaying Images.....	9
Exiting Gracefully.....	9
Making It Loop.....	9
Full Example Source.....	9
Signal Conditioning.....	10
Flip/Rotate source.....	10
Resolution selection.....	10
Noise Reduction and Image Smoothing.....	10
Image Filters.....	11
Color Thresholding – Fixed and Adaptive.....	11
Color Space Conversion.....	11
Look-Up Table.....	12
Array Arithmetic.....	12
Image Transforms.....	12

Filter2D and Convolution Matrices.....	12
Edge Detection.....	13
Laplacian Operator.....	13
Sobel Operator.....	13
Canny Algorithm.....	14
Erode/Dilate.....	14
Hough Transforms.....	14
Image Analysis.....	14
Image Segmentation.....	14
High-Level Functions.....	15
Writing Your Own Filters – Accessing Pixels.....	15
Example Vision Algorithm Implementation – Robust Colored Ball Tracking.....	16
Debugging OpenCV Programs.....	20
VNC.....	20
Conclusion.....	20
Addendum: RoboRealm.....	21

Overview

The proliferation of amateur robotics has given hobbyists the opportunity to attempt more advanced projects. It soon becomes necessary in these many of these designs to include vision capabilities. There are several products available that provide largely black box-type implementations of basic vision functions such as tracking colored blobs. While these may suffice for simple applications such as chasing a neon-colored ball across a white floor, and may meet the needs of basic projects, there are ways to gain much more functionality with comparatively small cost. This article will provide an overview of how to implement a more capable system using the OpenCV framework.

This article assumes a fairly complete knowledge of the C programming language, and in most cases I will not explain concepts related to the language, which are available in other references. I will, however, make efforts to explain the image processing concepts wherever possible, but some basic knowledge of matrices and/or calculus may be required to understand the concepts fully.

Even though this document centers on OpenCV, I will start with a comparison of the dominant vision systems available to hobbyists to give a sense of which system may be appropriate system various projects.

Comparison of Robotic Vision Systems

First Generation: AVRCam, CMUCam I + II

<http://www.jrobot.net/Projects/AVRCam.html> ; <http://www.cs.cmu.edu/~cmucam/>

\$100

These appeared around 2002 and offered the basic functionality required to track a high contrast colored object. While it was technically possible to customize them, it required buying the dedicated programmer for the microcontroller and wading through source code that was highly optimized in order to get the needed performance out of the low-budget processors. These can now be bought for around \$100 and are a good solution if the developer is working on a smaller budget and doesn't need much beyond basic color blob tracking. These also have the advantage of a very small form factor (less than 2.5 in. cube)

The CMUCam II provided a small upgrade in processing power and added additional functions such as frame differencing (basic motion detection) and color histogramming, but the overall level of capability of the device remains the same.

Second Generation: CMUCam III, Surveyor Blackfin Camera

<http://www.cmucam.org> ; <http://www.surveyor.com>

\$200-\$240

These are an upgrade to the first gen cameras. While maintaining a similarly small form factor, the CMUCam III offers upgraded processing power and the addition of a Lua script interpreter on top of a solid image processing API, which makes it considerably easy for the user to develop custom image processing routines.

The Blackfin Camera from Surveyor offers an incredible amount of processing power for its size through its use of the Blackfin processor from Analog Devices, rated at 500 MHz. The Blackfin Camera also includes a C interpreter that grants much the same functionality as the CMUCam's Lua interface. Surveyor has also released a harness that fits two of these cameras, enabling the use of stereo vision. This is a very capable device.

Custom x86-Based

~\$350-?

Still more power can be gained by tapping the modern x86 family of processors. While requiring more hardware than the other options, they grant developers degrees of image processing not possible through other solutions, including viability of implementing more intelligent and adaptive filters. These systems also provide a more convenient development and debugging environment. This article will describe how to assemble and use this type of system.

Embedded x86 Computers

<http://www.roboard.com/>

\$290

Recently, a product called the RoBoard has been released that provides an x86-compatible (can even reported run Windows) single board computer for robot hobbyists. While I have not used this product, it seems like it could be a viable off-the-shelf alternative to building a system from parts.

Choosing an x86 Co-Processor

The first step is to obtain a system to run the vision code on. There are two main options for this: assembling one from components or making use of a pre-built portable computer.

Using a Laptop

The recent release of Intel's Atom processor line has opened up a whole new selection of portable computer systems. While I have not tested these devices for use in vision systems, they seem to fit the requirements very nicely. The “net book” form factor offers an appropriate system in a package of a reasonable size for robotics. The integrated screen in laptop-style computers is also beneficial as it allows one to monitor the system as it is working. (See the section on VNC for other options)

Even if a pre-built system is chosen for a project, read the following sections to ensure the components are adequate. Inexpensive netbooks may look attractive, but my experience is many of them aren't powerful enough.

Assembling Your Own

Building a system from components, on the other hand, allows one to customize the system and can also save money.

Note: It is assumed that the reader can assemble the computer from components, and as such the following sections will comment on part selection, and leave assembly to the discretion of the reader and the instructions included with most computer components.

Motherboard/Processor

\$70-

When choosing a processor, the overall theme is to get as fast a processor as possible. This may seem obvious, but the processor is the most important component in a vision system (perhaps next to the camera itself), as it is the one running the vision algorithms, and in order for the system to be useful, the frame rate has to stay reasonably high. My experience is the clock speed should be at least 1 GHz.

When choosing a motherboard, “mini-ITX,” “nano-ITX,” or even the newer (and smaller) “pico-ITX” from Via offer small form factors of a size appropriate for robotics. Systems of a similar size from Intel

and AMD are usually labeled “Micro-ATX,” but the Micro-ATX encompasses systems on both 244x244mm and 171x171mm scales, and I found the larger size to be a bit large, so check for a size measurement.

I can make a specific recommendation for the Intel D201GLY motherboard, which includes a 1.2 GHz Celeron processor soldered on. While this eliminates the option of upgrading the processor in the future, no other system can match its \$70 price tag.

RAM

<http://www.frys.com>

\$15-

At least 1 GB of memory should be used: the RAM should be able to hold system's memory without the use of a pagefile (I recommend turning paging off), as the vision process will be significantly slowed if it has to page memory back and forth to the hard drive.

Hard Drive

<http://www.addonics.com>

\$25

<http://www.frys.com>

\$20-

The parts suggestions made in this guide are under the assumption that the vision system will be used on a medium-sized mobile robot. Obviously, if the robot will be a permanent installation, larger systems can be used. The same guidelines go for hard drives: for a fixed robot, a standard magnetic platter hard drive may be used, but for mobile robots that have a chance of encountering rougher terrain, a solid state hard drive is probably better suited.

A 4 GB CompactFlash card combined with a CF to IDE adapter such as those available from Addonics is a very easy and cost-effective way to construct such a drive.

Power source

<http://www.mini-box.com>

Two choices exist for powering your co-processor: either using a dedicated battery or powering it off of the robot's main battery. While an isolated battery provides a clean source of power for the co-processor, it is also extra weight that the robot has to carry around.

If the robot has a battery that can support it, an alternative is to condition the power from the robot's main power source to power the co-processor. This conditioning is necessary because the robot's motors will cause spikes that can be dangerous to the computer. I've used the M2-ATX power supply from www.mini-box.com, and it has worked flawlessly, even through 80A power surges (running off a 12v sealed lead-acid battery).

Choice of Operating System

The operating system run on a vision system should provide the drivers and subsystems necessary for interfacing with the camera and other hardware devices, while providing as little overhead as possible. The best operating system for this, in my opinion, is Windows 2000, but Windows XP can also be viable if the unnecessary glitz and features are disabled. Windows Vista, realistically, is unreasonable. Some may make the suggestion of using Linux, and it would be a very good one, except I've had troubles with Video4Linux distorting the colors coming from all of several webcams that I tried. If this issue can be resolved, then Linux would be a powerful option.

Any operating system used should be trimmed of excess services in order to make the system as streamlined as possible; in most cases, a robot does not need anti-virus.

Camera

\$25-

Most modern webcams offer at least VGA resolution, and should suffice for any vision application. Firewire camcorders, etc., can also be used, but are overkill in most cases.

Introduction to OpenCV

Overview

OpenCV is an image processing framework originally developed by Intel Corp. and now maintained by Willow Garage and the open source community. It has been optimized for x86 processors, although third parties have ported it to both the STI Cell processor and the PlayStation Portable device, possibly among others.

OpenCV has been used by developers at all levels, and was one of the enabling technologies behind Stanford University's winning entry into the DARPA Grand Challenge in 2005.

How to get it

Packages for both Windows and Linux are available through the project's Sourceforge page, listed under Where To Get More Info. The source can also be compiled on Macs; the project's Wiki has more information on this.

Where To Get More Info

Sourceforge project site: <http://sourceforge.net/projects/opencvlibrary/>

Main Wiki hosted on Willow Garage: <http://opencv.willowgarage.com/>

Bradski and Kaehler, [Learning OpenCV](#), O'Reilly Media.

Basic Concepts of OpenCV

OpenCV follows the imperative programming model of the C programming language. Data structures such as images or data arrays are initialized, then passed through a series of functions that modify them. This does a decent job of mimicking the dataflow paradigm that signal (image) processing applications take on. The following sections will first introduce the construction and destruction of the basic data types, then describe the toolkit of filters that can be applied to these data.

Structures

For the most part, the raw data is hidden from the user in OpenCV, encapsulated in data structures.

IplImage

IplImage is the main data type of OpenCV, and it represents an image. It contains the actual pixel data of the image, as well as metadata about the image, such as size, bit-depth, number of color channels, etc. The Ipl prefix comes from the structure's origins in the Intel Image Processing Library; most structures in OpenCV have a Cv prefix.

Other structures

Other distinct structures in OpenCV include:

CvMat represents an arbitrary-sized matrix, such as can be used for creating convolution functions (more on these later), or by the various matrix algebra functions included in the library.

CvSeq represents a linked list of objects. Cast a CvSeq * to the payload type to gain access to the current element; the next element is contained in ->next. For example, a CvSeq of IplImages called `sequence` would be used thus: to access the current image, use `(IplImage *) sequence`, to get the next element in `sequence`, call `sequence->next`, which can then be cast to an IplImage, etc. A CvSeq can hold any type; refer the documentation of specific functions.

CvMemStorage is generally allocated by the user to provide heap space for a function to work in, and then deallocated after the results of that function are no longer needed.

CvArr can be thought of as the “parent” type for IplImage, CvMat, CvSeq, etc (even though there is no Object Oriented-like inheritance in C). Often functions are declared as taking CvArr values as parameters, so read the documentation as to what type of value(s) the function can take.

CvScalar, which represents up to 4 scalar values, and as such is often used for passing color values, which can be created by the `CV_RGB(int, int, int)` macro, in addition to single-scalar values, i.e. returning number of objects found by a pattern matching function. Unlike the previously listed structures, CvScalars usually can be allocated as stack space, and are passed by value rather than by reference.

The remaining structures are predominantly used to hold information about the previously given structures. They include CvSize, CvPoint, etc; the names are fairly self explanatory. These are almost always passed by value. Most of these “helper” structures have corresponding macros to assist in initialization called `cvSize`, `cvPoint`, etc (notice the lower-case “cv”).

Allocation Convention

As noted in the previous section, many of the structures in OpenCV are heap-allocated and passed by reference. OpenCV provides its own set of memory management functions, usually called `cvCreate<datatype>`, for example `cvCreateMat` or `cvCreateImage`. The corresponding freeing functions are called `cvRelease<datatype>`, and expect a reference to the pointer variable. For example, basic image creation and deletion would progress as follows:

```
IplImage *image = cvCreateImage(cvSize(300, 300), IPL_DEPTH_8U, 3);  
cvReleaseImage(&image);
```

Image Parameters

Three parameters provide a general characterization of an image: Image size is the width and height of the image in pixels, bit depth controls how many bits are used to represent each of the color channels of one of the image's pixels (i.e. 8 bit means 8 bits per color channel per pixel), number of channels is how many color channels are used to represent each pixel: grayscale images use one channel, RGB images use three channels; if the image has transparency, a fourth channel is required. These are also the three parameters accepted by `cvCreateImage`.

OpenCV's Packages

OpenCV is divided into 5 different packages:

CxCore contains “utility”-type functions, such as the basic memory management functions, basic mathematic operators, basic overlay drawing functions, etc.

Cv contains the main body of the image processing functionality.

HighGUI contains a library that makes displaying images to the user easy and painless, and also includes functions for capturing video from cameras and video files.

These three will be the focus of this tutorial. The remaining two packages are CvAux, which includes more image analysis functions that haven't been merged into the main Cv package, and the Machine Learning package, which includes prebuilt Bayesian classifiers, neural network builders, etc.

Getting Started with OpenCV – Acquiring an Image

This section will get you started with OpenCV, and show how to do basic image capture from a camera and display the image to the user.

Installing OpenCV

Instructions for installing OpenCV on various platforms is available at <http://opencv.willowgarage.com>

Compilation Requirements

Each of OpenCV's aforementioned packages has its own header file and library file.

On Windows, header files are usually located at "C:\Program Files\OpenCV\cxcore\include" "C:\Program Files\OpenCV\cv\include" or "C:\Program Files\OpenCV\otherlibs\highgui" and libraries in "C:\Program Files\OpenCV\lib"

On Linux, header files are usually in /usr/include/opencv and libraries in /usr/lib. Many installations of OpenCV also support pkg-config.

Basic program structure

Our basic program structure will consist of first opening a camera capture and a HighGUI window, then entering the main loop, which will capture an image, then display it to the user. Lastly, we will need to release the window and capture.

HighGUI Capture

To open a capture from the first camera found on the system, call the `cvCaptureFromCAM` with its single parameter set to `CV_CAP_ANY`. To specify a specific camera, pass a non-negative integer. For the sake of example, we will make use of the default camera, so:

```
CvCapture *capture = cvCaptureFromCAM(CV_CAP_ANY);
```

Then check to make sure the capture was successfully opened:

```
if (capture==NULL)
{
    fprintf(stderr, "Capture is null\n");
    return -1;
}
```

To capture from a video file instead, replace `cvCaptureFromCAM` with `cvCaptureFromFile` and pass the name of the video file. Note: OpenCV will only capture from a limited number of file formats,

so you may need to convert the video file before it can be used.

Capturing an Image

To capture an image from a capture (whether from a camera or video file), call the `cvQueryFrame` function and pass it the capture object. This will return an `IplImage *` with the video frame. It is very important to note, however, that this returned image object is reused for each captured frame and thus **should not be freed** like other images.

```
IplImage *frame = cvQueryFrame(capture);
```

Displaying Images

To display images to the user, first a window has to be created. This is done by calling the `cvNamedWindow` function with the desired name of the window. Then images can be displayed in that window by calling `cvShowImage` with the name of the window and the image to be displayed. Lastly, a window is released by calling `cvDestroyWindow` with the name of the window.

```
cvNamedWindow("Frame");
```

```
cvShowImage("Frame", frame);
```

```
cvDestroyWindow("Frame");
```

Exiting Gracefully

Even though the captured image should not be released, we still have to release the capture by calling `cvReleaseCapture` (&capture);

Making It Loop

The HighGUI package provides a function called `cvWaitKey` which will wait for an specified period of time for a key to be pressed. If a key is pressed, it will return that key, otherwise it returns a 0. In our case, we want to continue in the loop of capturing and displaying frames until the user presses 'q,' so we'll check each time through the loop by calling

```
(cvWaitKey(10) & 0xff) != 'q'
```

To see if the user has pressed the letter 'q.'

Full Example Source

```
#include "cxcore.h"
#include "cv.h"
#include "highgui.h"

int main(int argc, char *argv[])
{
    CvCapture *capture = cvCaptureFromCAM(CV_CAP_ANY);
    if (capture==NULL)
    {
        printf("Capture is null\n");
        return -1;
    }

    cvNamedWindow("Frame");
```

```
while((cvWaitKey(10) & 0xff) != 'q')
{
    IplImage *frame = cvQueryFrame(capture);

    cvShowImage("Frame", frame);
}

cvDestroyWindow("Frame");
cvReleaseCapture(&capture);
}
```

The following four sections are not meant to provide an exhaustive documentation of OpenCV's processing functions, but more to give a sense of how they would be used. The purpose of this tutorial is not to repeat documentation that is already freely available, but to give a guide on how to create vision processing algorithms using OpenCV.

Signal Conditioning

The first step in image processing once an image has been captured is to condition it into a relatively usable form.

Flip/Rotate source

Depending on the camera's drivers and the mounting for the camera, it may be necessary to reorient the image coming from the camera. The `cvFlip` function from CxCore will do a vertical flip, horizontal flip, or both.

Another common problem, especially on Windows systems, is often the order of the pixels read in from the camera will be reversed: the image is typically stored in OpenCV as Blue, Green, Red (BGR), but some cameras will give images stored as Red, Green, Blue (RGB). The `cvCvtColor` function can be used to rectify this, using the `CV_RGB2BGR` conversion constant.

Resolution selection

An important factor to consider, especially in systems (like robots often are) where near-real time performance is desired, is controlling the amount of data being processed versus processing power available. An easy way to keep this ratio under control when it comes to vision is by picking an appropriate resolution. In general, however, the higher resolution that can reasonably be processed the better, as a higher resolution will yield a more accurate result. Often, cameras will have proprietary interfaces to control their resolution, but if none of these are available, the OpenCV `cvResize` function can be used. The destination resolution is controlled by the size of the image passed in as the destination image; create an image using `cvCreateImage` of the desired size.

Noise Reduction and Image Smoothing

The images coming from most cameras will contain a significant amount of noise, even if it is not readily apparent to a human observer. In general, most vision algorithms work by performing some sort of image segmentation (dividing the image into distinct regions), often based on color, and noise in the image can cause bad artifacts.

`cvSmooth` will perform several different types of smoothing on the image, but for most cases, I have found the most effective type to be median smoothing. This method determines the value of each smoothed pixel by finding the median value of the `param1` x `param2` rectangle of pixels around it (called the filter's "window"). This generally results in an image with harder divisions between color patches.

The two other common types of smoothing are Gaussian smoothing and blurring. Blurring is a simple averaging of the pixels within the window, while Gaussian smoothing is an average as well, but one that is weighted more heavily toward the pixels at the center of the window (around the original pixel).

In addition to picking the right smoothing type, the other crucial aspect when using smoothing is to pick the right window size. Picking a too-large window will cause loss of detail (while too small won't be effective), so start with 5x5 or 3x3 and increase if necessary until as many artifacts are removed as possible without corrupting the desired sections of the image.

Image Filters

Once a useful image has been obtained, the next general step is to isolate those portions of the image which are easily identifiable as being of interest. The most basic method used for this step is color thresholding, or bound checking on pixel values. If the object(s) of interest is not uniquely colored, other methods will have to be found to identify it.

Color Thresholding – Fixed and Adaptive

OpenCV provides two thresholding functions, `cvThreshold` and `cvAdaptiveThreshold`. `cvThreshold` is the standard fixed thresholding function: using thresholding mode `CV_THRESH_BINARY` will return an image consisting of pixels equal to either `max_value` or 0 depending on whether the pixel meets the threshold. Such an image is called a binary image, as it contains only two distinct values. `cvAdaptiveThreshold` generates filters an image using an adaptively-calculated threshold, meaning it will generate a threshold value for a given pixel by finding an average of the surrounding pixels. Generally, `cvThreshold` is the first option to try, as it provides more predictable results, but `cvAdaptiveThreshold` can be useful when the filter has to respond to images with a very broad range of brightnesses or contrasts.

Both functions work on single-channel arrays (i.e. monochrome images) only, so to threshold the typical RGB image, first use the `cvSplit` function to separate the image into three monochromatic images, then recombine the resulting binary images using `cvAnd`, resulting in a single binary image that is the intersection of all three thresholded images.

If both upper and lower thresholds are desired when using a fixed threshold, the `cvInRangeS` function may also be of use.

Color Space Conversion

A useful method for achieving more desirable results from color thresholding is to first convert the image to another color space. A color space such as HSL (Hue, Saturation, Luminosity) can sometimes result in a better-designed filter, as it allows the characteristics of color hue, saturation, and luminosity (brightness) to be addressed separately, i.e. allowing a more forgiving filter for luminosity if a large range of input brightnesses is anticipated.

Look-Up Table

A different type of threshold can also be made by using `cvLUT`, the look-up table function. Similar to look-up tables already utilized by many teams in applications such as distance sensor-based autonomous routines, this function could allow the programmer to decide on the basis of specific values whether or not a certain color should be allowed through the threshold. As before, this should most likely be used to try to obtain a binary image.

Array Arithmetic

OpenCV provides built-in functions for both array-array and array-scalar arithmetic, which can be used as necessary. Array-scalar functions are denoted by a capital S suffix.

Image Transforms

Once color thresholding and other basic filtering has been performed, the next level of complexity is to examine the morphology (shapes) in the image.

Most of this stage is accomplished by using image transforms. Image transforms work by converting from one *image space* to another, for example between the base image and the image that represents the edges in that base image. In the following descriptions, *source space* means pixels from the source image, similarly, *destination space* means pixels that are in the destination image. The term of *space* comes from the mathematical basis of the transforms, and can be thought of as representing one type of data; this paper will introduce functions to move to “edge space,” and *Hough space*, which can be used to find circles or lines in an image using functions from OpenCV, among others. One can also include image smoothing, introduced previously, as an image transform.

It should be noted that all the following functions operate on single-channel images, so either a filtering step will have to be done first, or `cvSplit` will have to be used to process each color channel separately.

Filter2D and Convolution Matrices

The basic building block of many image transforms is the convolution matrix (called a convolution kernel in OpenCV). A convolution matrix of size $n \times n$ describe image transforms such that a pixel at position (x,y) in the destination image equals a weighted sum of the $n \times n$ square of pixels in the source image centered at (x,y) , with the weights given in the convolution matrix. For example, to create a simple blurring transform, one could use the following 3×3 convolution matrix:

$$\begin{bmatrix} 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \end{bmatrix}$$

This would mean that each pixel in the destination image would be the average of each 3×3 pixels in the source image. If one wanted slightly less blurring, one could give more emphasis to the original pixel value using the following matrix:

$$\begin{bmatrix} 1/12 & 1/12 & 1/12 \\ 1/12 & 1/3 & 1/12 \\ 1/12 & 1/12 & 1/12 \end{bmatrix}$$

As another example, a standard edge detection matrix is:

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & -8 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

This works because if all nine pixels are the same, the resulting pixel will be zero, representing no edge. However, if one or more of the surrounding pixels are significantly different from the center pixel, such as would happen if there is an abrupt change occurring at this point in the image, the resulting pixel will be non-zero.

This particular transform is an example of a Laplacian transform, and is one of three edge detection algorithms built into OpenCV.

Edge Detection

The general concept behind edge detection is to scan the image for places of sudden change (in color), which usually denote a division. There are several different methods that can be used, the key is to choose the one that performs best on the types of images that will be analyzed by the vision algorithm.

Laplacian Operator

`cvLaplace` is an implementation of the Laplacian operator. For those with a calculus background, it can be thought of as a 2-dimensional 2nd derivative. This works for finding edges, because if one thinks about the first derivative as the rate of change of the color through the image, a constant first derivative value would denote a smooth gradient of color. Peaks in the gradient of the image would denote the places of largest change, and the Laplacian of the image can be used to find these points. Because the Laplacian is based off of the 2nd derivative, it is called a 2nd-order edge detection algorithm.

The `aperture_size` setting has a similar effect size parameters of `cvSmooth`, controlling how large an area of the image is used to calculate the derivative.

As an interesting aside, specifying aperture size 1 creates a transform that can be represented

by the convolution matrix $\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$, which can be derived by taking the definition of

the Laplacian operator as the two dimensional second derivative: considering the elements non-zero *center*, *up*, *down*, *left*, *right*, the first derivative of (amount of change around) *center*

along the *x* and *y* dimensions is: $\begin{bmatrix} (center - up) \\ (center - left) & (right - center) \\ (down - center) \end{bmatrix}$. The derivative of this

(i.e. the 2nd derivative), yields $(down - center) - (center - up) = down - 2center + up$ in the *y* dimension, and $(right - center) - (left - center) = right - 2center + left$ in the *x* dimension.

Summing these together yields $right + down - 4center + left + up$ which can be represented by the convolution matrix given. The sample edge detection convolution given in the previous section can be got in the same manner, by adding in the derivatives along the two diagonals.

Sobel Operator

The Sobel operator is a 1st-order edge detector (although OpenCV allows you to specify higher orders, which essentially applies the Sobel operator multiple times), meaning instead of looking for peaks in the image gradient, it calculates the “first derivative” of the image, giving the change in intensity. By

thresholding the resulting image, one can find areas of large change, which usually correspond to edges in the image.

Finding the right settings for `xorder`, `yorder`, and `aperture_size` may require some experimenting. I would advise as with other settings previously mentioned to start with smaller numbers and then if those don't work try larger ones. If high `order` settings still do not achieve the desired results, this is probably the wrong algorithm to use. `aperture_size` has the same effect as in `cvSobel`.

Canny Algorithm

While the Sobel and Laplace operators will produce gray scale images, the Canny algorithm will result in a binary image. This is because Canny is actually a method to filter the results of the Sobel operator that provides better results than a simple threshold. It judges each potential line based on the knowledge that firstly, larger differences in the value of the image are more likely to be edges, and then that detected edges with longer length are also more likely to be true edges.

The `aperture_size` setting controls the same setting in the Sobel operator. The larger of the two threshold settings controls the cutoff for the first stage of the algorithm, in which line segments are selected based on value difference in the image (this is similar to simply thresholding the result of `cvSobel`), while the smaller threshold controls which lines will be traced in the second stage of the algorithm, when lines are found based on length. To provide the best results, the larger threshold should be adjusted until only line segments are selected which are absolutely assured to be true lines, then the smaller threshold should be set so that the necessary lines are filled in.

Erode/Dilate

The `cvErode` and `cvDilate` functions are conceptually simple. Erode will reduce the size of blobs of pixels in the image, and Dilate will increase the size of such blobs, either adding or subtracting pixels from around the perimeter of the blob. For vision processing, they are useful because they can either accentuate or eliminate smaller blocks of pixels in the image. In addition, first applying Dilate and then Erode can cause adjacent blobs of pixels to become connected, while application in the reverse order can cause them to disconnect, without changing the general size of the blobs.

Hough Transforms

Hough space is an image space that describes the probability that certain shapes exist at different locations in an the image. OpenCV contains two functions making use Hough transforms, that can identify instances of straight lines (`cvHoughLines2`) or circles (`cvHoughCircles`) within an image. Both functions require the input image to be grayscale. For further documentation, refer to the example given at the end of this article, or the OpenCV documentation.

Image Analysis

Additional image processing can be done by examining the structure of the image.

Image Segmentation

Sometimes after a certain amount of filtering has been performed, a few separate possible results may be produced, each of which should be examined individually and comparison can be done to determine which is the best match. This will especially be the case if the robot is working in an environment with several of the same object present, and it will have to decide which of the targets to focus on. The usual

way to separate the results is to use image segmentation, which separates the image into blocks of contiguous pixels. Generally, the image should be smoothed before these algorithms take longer if there are more segments in the image and will count stray pixels of noise as segments; a median filter or similar is recommended, as it will keep the image as binary instead of creating gray values.

In OpenCV, the usual way of finding image segments is through contour finding, which can be thought of as a form of edge detection, but instead of generating another image with the edges marked, the contours are polygon approximations of the edges in the image. `cvFindContours` will extract the contours of the input binary image into a `CvSeq` list, the elements of which can either be passed to `cvDrawContours` to draw each segment to its own image, or there are additional functions that can provide information on the shape of each contour, which are well described in the Contour Processing Functions and Computational Geometry sections of the OpenCV manual (<http://opencv.willowgarage.com/wiki/CvReference>). For an example of using `cvFindContours` and `cvDrawContours`, see the example at the end of the article.

High-Level Functions

This paper is meant only as a summary of the more commonly used functions in OpenCV. There are many more algorithms included, such as for stereo cameras and learning-based object detection (classifiers for different objects can be created based off of a many pictures of that object).

Writing Your Own Filters – Accessing Pixels

In some situations, you may wish to use filtering functionality that is not built into OpenCV, or may be clumsy to implement using only OpenCV functions. In these cases, filters can be custom-coded by accessing the pixels in the image directly. The `IplImage` structure is a C structure like any other, and the definition is available in OpenCV's header files (`cxtypes.h`).

The important field for the purposes of accessing the raw image is called `imageData`, which consists of an array of the image's pixels listed left to right, top to bottom. The field is declared as a character pointer, but should be cast to the appropriate type depending on the definition of the image. For example, to access an unsigned 8-bit image (the most common type of image):

```
unsigned char* pixelData = (unsigned char*)(image->imageData);
```

unsigned 16-bit images are unsigned shorts, etc. The other aspect to keep in mind is that different channels are stored separately, so each pixel in an RGB image will take three elements in the pixel array, while a grayscale image will only take one element per pixel. Thus, we can create a general form for accessing a specific pixel value:

```
pixelData[(j*image->width+i)*image->nChannels+k]
```

for the k^{th} channel of the pixel at (i,j) .

Usually, filters iterate over all the pixels in the image, so a template for one type of custom filter could be:

```
int i, j;
for(j=0; j<image->height; j++)
{
    for(i=0; i<image->width; i++)
    {
```

```

int ind = (j*image->width+i)*image->nChannels;
// 1st channel (usually blue)
pixelData[ind+0] = filter(pixelData[ind+0]);
// 2nd channel (usually green)
pixelData[ind+1] = filter(pixelData[ind+1]);
// 3rd channel (usually red)
pixelData[ind+2] = filter(pixelData[ind+2]);
//...
}
}

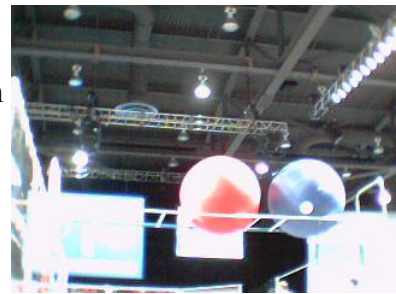
```

But, obviously, each filter will be different.

Example Vision Algorithm Implementation – Robust Colored Ball Tracking

The 2008 FIRST Robotics Competition game involved the robots manipulating large colored yoga balls. I decided to work at trying to get my team's robot to chase the balls autonomously. Here's an overview of the process I went through to derive the vision algorithm I used:

Signal conditioning: Since I was looking for fairly large objects, and real time performance was necessary, I selected to use a 320 x 240 frame size. After testing, I found I still achieved good results even when I didn't smooth until after the first stage of processing, so I decided not to smooth the initial image.



Identifying key distinguishing factors of the targets: relatively large, high contrast color (blue or red), circular shape

General theory: start with characteristics that are the easiest to filter for, so the computer does the least work. Easiest of the distinguishing characteristics is the target's color.



Since the robot needed to work in a wide variety of lighting conditions, thresholding on RGB values may not be the best, especially as the colors get very close to white or black. Using the HSL color space allowed me to filter specifically on the hue of each pixel, without worrying about how light or saturated each pixel is, which vary the most with changing lighting.

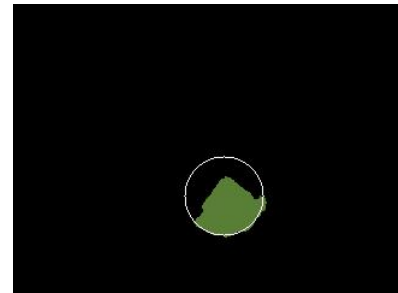
I didn't smooth the image earlier, but found it best to do so at this point. Since I'm dealing with larger targets, I can then smooth the image fairly harshly to eliminate stray pixels and smaller blobs. Since I'm going to segment the image, median smoothing is by far the best smoothing method to use for this case, since it provides hard edges instead of blurred edges which would be interpreted as a bunch of smaller blobs by the segmentation algorithm.



I was then left with an image with several large segments. I used image segmentation to separate them so I could analyze each individually. I filtered the result even further by setting a minimum number of pixels and discarding any of the segments with a smaller area than that.

The remaining, and perhaps most important, characteristic of the targets are their circularity, so I Hough transformed each of the blobs to evaluate how circular that segment is. Whichever is the most circular is the one the algorithm is most confident is actually a target.

Each of the blobs were filtered separately because the Hough transform only considers the outlines of the shapes it's filtering, so if several blobs were arranged in a circle, it might generate a false positive.



Final code:

```
#include <cxcore.h>
#include <cv.h>
#include <highgui.h>

#define RED 1
#define BLUE 0

// test if a pixel is red
bool redTest(int h, int s, int l)
{
    return ((h>330) || (h<35)) && (s>20) && (l>15) && (l < 95);
}
// test if a pixel is blue
bool blueTest(int h, int s, int l)
{
    return ((h>180) && (h<300)) && (s>10) && (l>30) && (l < 95);
}

int main(int argc, char* argv[])
{
    if (argc!=3)
    {
        printf("Usage: OpenCVTracker.exe CAM|<video file> RED|BLUE\n");
        return -1;
    }

    // parse color command line parameter
    int color = strcmp(argv[2], "BLUE")==0 ? BLUE : RED;;

    // open capture on either video file or camera and test if successful
    CvCapture* capture = strcmp(argv[1], "CAM")==0 ?
        cvCaptureFromCAM( CV_CAP_ANY ) :
        cvCaptureFromAVI( argv[1] );

    if( !capture )
    {
        fprintf( stderr, "ERROR: capture is NULL \n" );
        getchar();
        return -1;
    }

    // create display windows
    cvNamedWindow( "frame", CV_WINDOW_AUTOSIZE );
    cvNamedWindow( "threshold", CV_WINDOW_AUTOSIZE );
    cvNamedWindow( "smooth", CV_WINDOW_AUTOSIZE );
    cvNamedWindow( "circles", CV_WINDOW_AUTOSIZE );

    // create image buffers
    IplImage* frame = cvCreateImage( cvSize(320,240), 8, 3 );
```

```

IplImage* imageHSL = cvCreateImage( cvGetSize(frame), 8, 3 );
IplImage* binary   = cvCreateImage( cvGetSize(frame), 8, 1 );
IplImage* smooth   = cvCreateImage( cvGetSize(frame), 8, 1 );
IplImage* imageCircles = cvCreateImage( cvGetSize(frame), 8, 3 );
IplImage* imageBlobs = cvCreateImage( cvGetSize(frame), 8, 1 );

// allocate memory for image segmentation and Hough transform
CvMemStorage* storBlob = cvCreateMemStorage(0);

// the last detected radius, used to approximate the current radius
int r = 10;

// while we haven't been told to stop (includes 50ms delay)
while( (cvWaitKey(50) & 255) != 27 )
{
    // read a frame from the capture
    IplImage* framecap = cvQueryFrame( capture );
    if( !framecap )
    {
        fprintf( stderr, "ERROR: frame is null...\n" );
        getchar();
        break;
    }

    // resize to the desired resolution
    cvResize(framecap, frame);
    // convert to HSL color space
    cvCvtColor(frame, imageHSL, CV_RGB2HLS);

    // custom thresholding filter for HSL colorspace
    for (int i = 0; i < frame->width; i++)
    {
        for(int j = 0; j < frame->height; j++)
        {
            // extract the color components for each pixel
            unsigned char* sourceData = (unsigned char*)(imageHSL->imageData);
            int h = sourceData[(imageHSL->width*j+i)*3];
            int l = sourceData[(imageHSL->width*j+i)*3+1];
            int s = sourceData[(imageHSL->width*j+i)*3+2];
            int hue = 240 - (h*2) % 360;
            int sat = s * 100 / 255;
            int lum = l * 100 / 255;

            // call the appropriate thresholding function
            bool thisRed = color==RED ? redTest(hue,sat,lum) :
                                blueTest(hue,sat,lum);

            // write the result
            unsigned char* destData = (unsigned char*)(binary->imageData);
            destData[binary->widthStep*j+i] = thisRed ? 255 : 0;
        }
    }

    // smooth the image to get rid of stray pixels
    cvSmooth( binary, smooth, CV_MEDIAN, 9, 9 );

    // clear output image
    cvZero(imageCircles);
}

```

```

// segment the image
CvSeq* contour = 0;
cvFindContours( smooth, storBlob, &contour, sizeof(CvContour),
               CV_RETR_EXTERNAL, CV_CHAIN_APPROX_SIMPLE );

// the current best-matched circle out of all the blobs
int bestX = 0;
int bestY = 0;
int bestRadius = 0;

// iterate through all the image segments
for( ; contour != 0; contour = contour->h_next )
{
    // check if this blob fits size requirements
    CvRect rect = ((CvContour*)contour)->rect;
    if (rect.width < r || rect.height < r) continue;

    // create an image with only this segment
    cvZero(imageBlobs);
    cvDrawContours( imageBlobs, contour, CV_RGB(255,255,255),
                  CV_RGB(255,255,255), -1, CV_FILLED, 8 );

    // Hough transform this blob
    CvSeq* circles =
        cvHoughCircles( imageBlobs, storBlob, CV_HOUGH_GRADIENT, 2,
                       imageBlobs->height/4, 200, 20 );

    // if a circle was found
    if ( 0 < circles->total )
    {
        // keep only the largest circle
        float* p = (float*)cvGetSeqElem( circles, 0 );
        if (p[2] > bestRadius)
        {
            bestX = p[0];
            bestY = p[1];
            bestRadius = p[2];
        }
    }

    // draw this segment onto the output image with a random color
    CvScalar drawcolor = CV_RGB( rand()&255, rand()&255, rand()&255 );
    cvDrawContours( imageCircles, contour, drawcolor,
                  drawcolor, -1, CV_FILLED, 8 );
}

// keeps the detected x and y position from one frame to the next
static int x = bestX;
static int y = bestY;
static int lastX = bestX;

// if this circle is close to the last one we found
if ((lastX - bestX) < frame->width / 3)
{
    // store it as the current position of the target
    x = bestX;
    y = bestY;
    r = bestRadius;
}

```

```

// keep track of the last detected circle
lastX = bestX;

// write out result of detection
// in the original code, this wrote to the serial port that was
// connected to the robot's microcontroller
printf("X: %d Y: %d R: %d \n", x, y, r);
cvCircle( imageCircles, cvPoint(x,y), r, CV_RGB(255,255,255), 1, 8, 0 );

// show the various stages of processing for debugging purposes
cvShowImage( "frame", frame );
cvShowImage( "threshold", binary );
cvShowImage( "smooth", smooth);
cvShowImage( "circles", imageCircles);

// reset memory for the next frame
cvClearMemStorage(storBlob);
}

// release all memory
// don't release the capture frame image!

cvReleaseImage(&frame);
cvReleaseImage(&imageHSL);
cvReleaseImage(&imageBlobs);
cvReleaseImage(&imageCircles);
cvReleaseImage(&binary);
cvReleaseImage(&smooth);

cvReleaseMemStorage(&storBlob);

cvReleaseCapture( &capture );

// close windows
cvDestroyWindow( "frame" );
cvDestroyWindow( "threshold" );
cvDestroyWindow( "smooth" );
cvDestroyWindow( "circles" );

return 0;
}

```

One final optimization included is a check to make sure that the target detected this time is close to the previously detected target, which helped to eliminate stray false positives.

You can download this code and example input videos from <http://andrew.cmu.edu/~rcahoon/opencv/>

Debugging OpenCV Programs

Because the data being analyzed is images, the best way to tell what's going on is to look at the images being processed. As described in the “Displaying Images” paragraph of the Getting Started section, OpenCV can be used to display images to the screen. Similarly to how one would at variable watches to a standard program, windows can be added to an OpenCV program to see what the images look like in various stages of being processed. Additional feedback to detected location of targets in images, etc, can be gained by using the drawing functions such as `cvRectangle`, `cvCircle`, and `cvLine` to add markers onto the images.

VNC

This capability can be extended further by enabling the vision coprocessor with WiFi and using VNC or another remote desktop to view the coprocessor's display remotely, allowing the program to be debugged while the robot is running.

Conclusion

The OpenCV framework has been used in all levels of application, from hobby applications like what has been described here, to bringing home the gold medal for Stanford's autonomous car in the DARPA Grand Challenge. It holds a lot of potential to bring higher-end functionality to amateur robotics projects, as for under \$350 dollars (including the cost of Windows and a webcam), one can have access to an extensive library of advanced vision functions.

Addendum: RoboRealm

There is a software program for Windows (full version for a nominal fee, but with a 30 day free trial) called RoboRealm (<http://www.roborealm.com>) that offers many of the same filters as OpenCV, but allows experimenting with them using a GUI that allows the filters to be hotswapped. While the filters are not as customizable as in OpenCV, I've found this program to be useful in rapid prototyping situations, as it can help to test out different filters quickly without the burden of syntax issues and having to recompile the program after every change.