Rewrite of the Vision Tracking System, To Benefit the Accuracy and Speed of the nVidia Tegra©

Nicholas Burrell
Software Team

Febuary - March, 2016

Contents

1 Introduction

1.1 Before We Begin

Thank you for taking the time to read the documentation for our systems new rewrite. The old system had a ***LOT*** of issues, which we can see below. They range from simple things like inconsistent variable naming schemes and tabbing, to more extreme things like memory management and even neglecting an entire subsystem of the Tegra©, the CUDA© Cores.
Some examples of poor writting include...

```
1  template <class T>
2  T getMin (vector <T> input) {
3    size_t i;
4    T minimum = 0;
5    T minimum_out = 0;
6    if (input.size() == 0)
7      return 360;
8    if (input.size() == 1)
9      return input[0];
10   else{
11     minimum = 360;
12     for (i = 0; i < input.size(); i++) {
13       if (minimum > abs(input[i])) {
14         minimum = abs(input[i]);
15         minimum_out = input[i];
16       }
17     }
18   }
19   return minimum_out;
20 }
```

Listing 1: Poor Coding Part 1

The first issue with this example, is it is written for reusability, while it doesn't need to be. That is *not* and issue, but for the sake of simplicity, it can and *will* be rewritten. The second being the hard coded value. Line 11, to be precise. If we were to write this code to be reusable, we should just have the value be some arbitrarily large number, say, 600 billion. Speaking of bad practices...

```

float angle;
  int numOfBadFrames = 0;
    bool die = false;
  string filename("snapshot");
  string suffix(".png");
  int i_snap(0), iter(0);
  auto index = 0;
    vector < vector <Point> > contours;
    vector<Rect > boundRect;
  vector <KeyPoint> keypoints;
  vector <Point> centers;
  //vector <float > abs_angle;
  Mat frame(Size(480, 360), CV_8UC3, Scalar(0));
  bool bIsConnected = false;
  int sockfd, new_fd;  // listen on sock_fd, new connection on new_fd
  struct addrinfo hints, *servinfo, *p;
  struct sockaddr_storage cli_addr; // connector's address information
  socklen_t sin_size;
  struct sigaction sa;
  int yes=1;
  char s[INET6_ADDRSTRLEN];
  int rv;
  char buf[256];
  int numchars = 0;
```

Listing 2: Poor Coding Part 2

The naming convention has *no* convention. It appears as though the code was copied off of the internet. Granted, a small portion of it was. (Mostly the sockets) Along with this, the tabbing is atrocious. This code is *begging* to be rewritten.

1.2 General Improvements

Before we delve into what will be rewritten, lets compare and contrast some of the issues with solutions.
The first issue is the writing style. Relatively easy fix. The next issue, is speed. The fact that the camera in the rear of the robot can go at *whatever* framerate we need at the time, means my code should perform that fast as well. Parallel processing would improve the speed *drastically*, as would the GPU. Some of these will require some things like named pipes, or more advanced things such as CUDA$^{©}$ mathematics.

2 New Features And Improvements

2.1 Improvements (In Greater Detail)

One of the things I have noticed with the system in the past, is that if there is too much ambient IR or other kinds lighting, (e.g. the Towers LEDs), it has caused issues. The way to fix this light correction. This would take some more interesting algorithms to accomplish, such as sequential frame analysis and comparison to find differences in the light, and increasing or decreasing the color threshold to accommodate the current conditions.
Another interesting feature being integrated is a custom made networking API. A client/server socket wrapper. Really simple to use. Simply call the client or server constructor, and then either send or receive, respectively.

2.2 New Features

There are not nearly as many new features as their are improvements and bug fixes. Completely revamped algorithms, boosting mathmatics with CUDA$^{©}$ and assembly (Mostly GAS and NASM) for smaller operations (getMax() and getMin()).

2.3 Improvements Continued

Continuing on the topic of original features, we have a few areas in need of some *serious* TLC. They fall under 3 categories, corresponding to 3 subsections of this section.

1. Mathematics
2. Networking
3. Image Processing

We will cover each area in the order above.

2.3.1 Mathematics

Overview In the mathematics department, we are *completely* redoing some of the algorithms. Examples such as:

```
cv::Point getMean(cv::Point pt1, cv::Point pt2) {
  return Point(((pt1.x + pt2.x)/2), ((pt1.y + pt2.y)/2));
}

int getProxToPoint (cv::Point pt1, int midline) {
  //int midline = (frame.cols / 2);
  return (midline - pt1.x);

}

float fGetAnglefromPixel ( float _fMidLine, int _iFOV, float _fX) {
  cout << _fX << endl;
  float _fRatioOfXToFrame = _fX/(_fMidLine);
  cout << _fRatioOfXToFrame << endl;
  float _fAngle = _fRatioOfXToFrame * (_iFOV/2);
  cout << _fAngle;
  if (_fX < _fMidLine) {
    _fAngle = (((_iFOV/2) - _fAngle)* -1);
  }
  else {
    _fAngle = (_fAngle - (_iFOV/2));
  }
  cout << _fAngle << endl;
  return _fAngle;
}
```

Listing 3: Basic Mathematical Algorithms

These are *very* basic mathematical functions. The last one can be put as simply as

$$|\theta| = \frac{x}{m} * \frac{FOV}{2} \tag{1}$$

where m is the x at the middle of the image, the FOV in our case being ≈ 67. Of course, there is additional logic involved, including how to tell if you are to the left or the right, but that can be done by doing the following.

```
if (x < midline) return -x;
else return x;
```

Listing 4: Additional Logic Required

The other two are exceptionally easy formulas.
getProxToPoint (where m is the midline of the image)

$$m - x \tag{2}$$

And for getMean (Assuming X and Y are inputs).

$$x = \frac{(x_1+x_2)}{2} \tag{3}$$

$$y = \left(\frac{y_1+y_2}{2}\right) \tag{4}$$

Optimizing Putting the preceding code into assembly would be relatively simple. As inline assembly, the code would look like the following.

```
struct Point{
  uint32 x, //for the sake of being used; is just unsigned int
  uint32 y //same as above
  };
  inline Point getMean(Point pt1, Point pt2) {
    uint32 x3 = 0;
    uint32 y3 = 0;
    Point pt3;

    #ifdef __i386__
    asm ("mov bx, $2\n"
        "add %1, %2\n"
        "mov eax, %1\n"
        "div ax\n"
        "mov %0, %1"
        : "=x"(pt3.x): "x1"(pt1.x), "x2"(pt2.x) : );

    asm ("mov bx, $2\n"
        "add %1, %2\n"
        "mov eax, %1\n"
        "div ax\n"
        "mov %0, %1"
        : "=y3"(pt3.y): "y1"(pt1.y), "y2"(pt2.y) : );

    #endif
    return pt3;

  }
```

Listing 5: Basic Mathematical Algorithms Continued; Inline Assembly

It appears as though this is a sizeable chunk of code; however, if you take into account the speed boost by doing the commands in *raw* (almost) machine code (1:1 correspondence), it is blazingly fast. Doing the math, it would take $\frac{22}{Hz}$ ***SECONDS***! (Hz being CPU speed in GHz) 22 operations, or approximately 44

bytes *per instance*. Reason enough for it to be inline. The issue with the inline assembly, is portability. At the moment, if I were to compile this code on an i686 platform (or x86_64 with *very* little modification) it would most likely run as expected; however, on an ARM© based platform, say a Cortex© A9 (the very same processor inside of the nVidia© Jetson TK1) it would complain about register names. Code should be written to be portable, unless it is *not* explicitly needed. The __i686__ is only there so the user can test the code on a PC, for debugging reasons, *not* release (Possibly and encompassing #ifdef DEBUG).
There is a large list of other functions that we could cover, but you should assume that they will be addressed in the final product. On to the next topic.

2.3.2 Networking

The networking system has been wholly overhauled. We now have a class based client/server wrapper based off of Berkley Sockets. The header file network.hpp, is found below.

```
class Client {
public:
  Client(char* _szIP, char* _szPort, int _iSockType);
  virtual ~Client();

  bool bRecv(); //TCP
  bool bSendTo(char* _szBuf); //UDP

  bool bNewIP(char* _szIP);
protected:


private:

  int iSockfd;
  int iNumBytes;
  int iRV;
  int iSockType;

  const char* szPORT;
  char* szIP;
  char* szBuf;
  char szS[INET6_ADDRSTRLEN];

  struct addrinfo aiHints, *aiServInfo, *aiP;

};
```

Listing 6: The Client Class

Overview As you can see, this looks *a lot* better than that original snippet we looked at (see Listings 1 and 2). It is neatly written, with consistent naming, and is written using libc, so it is fully portable on any Unix© based platform (e.g. Mac and Linux). This class will even theoretically function on the cRio and roboRio, so integrations is already done. In the following paragraphs, I will outline the actual functionality of each member function, excluding bNewIp, as you can figure out what it does easily.

Constructor and Destructor The constructor and destructor of this class is *very* straight forward. You call the constructor, pass in the server IP, port, and socket type, and then either send or receive (For a more detailed description, wait and see the dOxygen documentation).

Receive and Send To The bRecv function looks at that moment for any incoming data on the assigned socket. It has a default buffer of 512 bytes, but this can be expanded in future iterations of the API. bSendTo is a little different. It is a UDP talker, meaning it sends data to a listener, which is considered a server. You simply pass in the *character pointer* buffer, not a string, and it will send. Pretty simple.

```
class Server {
public:

  Server(char* _szPort, int _iSockType, int _iBacklog, unsigned int _iMaxClients, bool
    _bIsPersistant);
  virtual ~Server();

  bool bBroadcast(char* _szBuf); //Only use if your using TCP sockets
  bool bListen(); //Only use if your using UDP sockets


protected:


private:

  int iSockfd;
  int iNewfd;
  int iYes = 1;
  int iRV;
  int iSockType;
  int iBacklog;
  int iNumBytes;

  bool bIsPersistant;

  uint32 iMaxClients;

  char szS[INET6_ADDRSTRLEN];
  char* szBuf;

  const char* szPORT;

  struct addrinfo aiHints, *aiServInfo, *aiP;

  socklen_t slAddrLen, slSinSize;

  struct sigaction sa;

  struct sockaddr_storage saCliAddr;

};
```

Listing 7: The Server Class

Overview The server class acts in a similar vain to the client class. It has a constructor, a virtual destructor (for any child class you make, so you can rewrite without fear), and a broadcast and listen function. As of the writing of this documentation, only TCP and UDP have been integrated into either class.

Constructor and Destructor The constructor is almost the exact same, except you don't require an IP, as it is your (either DHCP or static) IP. The backlog is the maximum amount of buffer packets. Maximum clients is obvious. The last option, persistence, tells the socket whether you will keep looping through broadcast on a separate thread, constantly feeding out data, or stop after you reach the maximum amount of clients.

Broadcast and Listen Act similarly to the receive and send to functions of the client. Broadcast spams a message out to all those willing to listen:. to a point. Listen, just waits until you get a packet from a client, telling you to do something. Essentially a trigger.

2.3.3 Image Processing

Overview Now comes the fun part. The actual processing of the image. To go to the most basic functions, we have a method to do color-based thresholding on an image. The new method will use similar math to the cv::inRange function, checking if the color is within the given HSV range. The code will be optimized to perform on the GPU as well, so we can parallelize with relative ease.

3 Summary

We Just covered a lot of the more technical side of of the internal workings of the code. But wait! There's more! The system is constantly evolving, adapting to how we need it in years to come. The current system is a framework for years to come, so we don't start from square one *every single year*

3.1 Are We Finished?

Appendices