Laser Pointer Interactive System

Alex Taffe, Kan Kawabata, Vincent Velarde

School of Electrical, Computer and Energy Engineering

Abstract

In this paper we present a simple and low-cost method of creating an interactive laser pointer tracking sensing system using laser pointers, a webcam, projector and OpenCV library. This was done by, (1) applying a homography transformation on the camera's perspective to align the camera with the projector output, (2) separating the laser dots from background noise (3) extracting the coordinate of the laser dots from the thresholded image.

Given the above setup we were able to develop a few applications such as basic draw, controlling computer mouse, miniture games, etc. With three laser pointers we were also able to perform position tracking of the user through 3D resection.

1. Introduction

Technology such as smartboards, Nintendo's Wii, and Microsoft's Xbox Kinect use human motion and gestures to create an interactive experience for the user. The ability to fuse technologies with the environment is the first step for true virtual reality and augmented reality. Unfortunately these technologies can be expensive and complex, thus it puts a high skill and money barrier for people to buy and develop for such systems. This becomes a problem as it stagnates innovation and creativity in the industry and discourages potential developers.

The motivation for this project is to create a motion sensing/tracking system with a laser pointer as an alternative to similar technologies that is currently on the market. Our system uses a laser pointer, laptop, webcam, and a projector all of which are either cheap or commonly available to the general public. Our software is based off of Python scripting language and OpenCV library both of which are open-source with great documentation, thus allowing anyone with basic programming skills to create and develop their own ideas.

2. Implementation

In order to achieve our goals our system needs to be able to track multiple laser pointer dots and determine their position relative to the board. There are three stages to completing this task, a homography calibration stage to correct for camera perspective, a stage to threshold camera feed to filter out noise and varied light intensities and detect laser dot, and finally contour detection stage for calculating multiple laser dot positions. Figure 1 shows the initial input and output of the first two stages. Once these steps are completed the binary image will then be used to calculate the coordinates of the laser pointer position using find contour and centroid formula.

2.1. Homograph transformation

The camera calibration was done using a OpenCV's perspective transform function. The Idea is to determine the homography matrix that transforms the camera's perspective of the environment to the head on view of the board.

$$\begin{bmatrix} x' \\ y' \\ z' \\ w' \end{bmatrix} = H * \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \tag{1}$$

In class we shown how to do this using the 8 point algorithm [1] however because the camera captures the projection on the wall (which is co-planer) this transformation can be thought of as a simple 2D mapping problem.

2.2. Robust noise filtering

One of the challenges to the project is finding a robust way of filtering out noise and varied lighting while still detecting the laser dot. Initially we set a global threshold intensity where any pixel with intensity greater than some value would be considered as a laser dot and anything below as background. However this required the user to personally adjust the global thresholding manually as the program has no way of knowing the optimal threshold. Additionally this method is not robust to difference in lighting, a brighter lit area would be more sensitive to noise while a darker area would not detect the laser dots.

To solve this issue, during calibration the camera will capture a couple seconds of background image with no laser dot present, then it will select for each pixel the maximum intensity detected for the duration and use that as a basis for thresholding. This method accounts for difference in lighting as each pixel has their own thresholding value associated with it and it is also automatically done by the system.

2.3. Find Contour and coordinate Estimation

The thresholding produces a black and white image of the detected laser dots. We want to convert this image to a list of coordinates of where the dots are located. To do this we use OpenCV's findContours which takes in an eight bit single channel image and performs the algorithm in [2] and returns a vectors of points that each correspond to a particular laser dot in the camera feed.

Two algorithms are described in [2], one that starts a border following algorithm on borders, once a start condition is fulfilled. The borders are followed in the order they are encountered during a raster scan. Algorithm number two is a modified version of the first, that searches exclusively for outer borders.

Algorithm one starts with a line by line scan of the image from left to right, until the start condition is satisfied. [2] defines a coordinate system where i represents rows and j represents columns with rows (i) increase from top to bottom and columns(j) increase from left to right. The start condition is defined as when and one is encountered during the scan. The border is then classified as either an outer border or a hole border. If a one preceded by a zero then the border is classified as

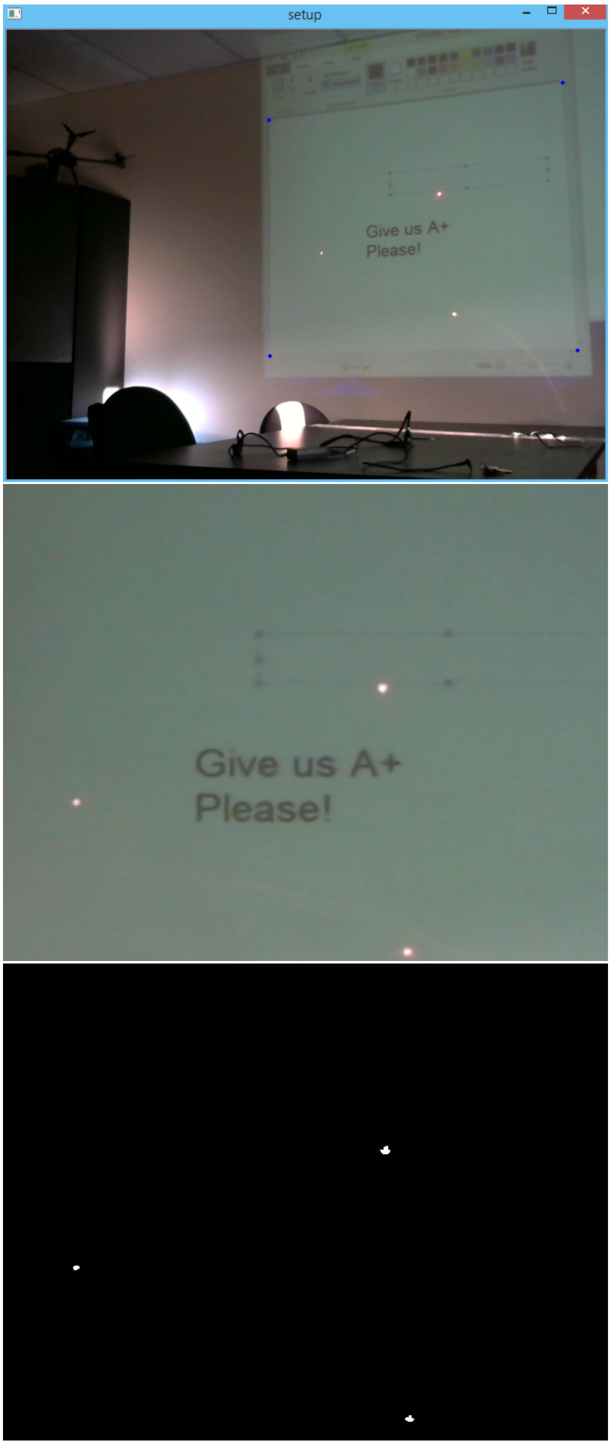


Figure 1: (Top)raw camera feed (middle)perspective transformed (bottom)intensity thresholded

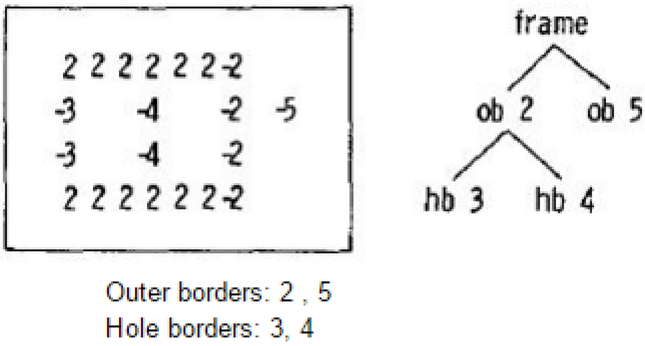


Figure 2: Example contoured binary image [2]

a outer border, otherwise if the one is succeeded by a zero then it is a hole border. Holes are defined as sections of 8- connected zero pixels, whereas the opposite 1-components are classified as 4-connected 1 pixels. If a one is preceded and succeeded by a zero it is classified as an outer border. Each pixels in the current border is then assigned a unique number then the bordered is followed until it ends. If the current pixels is (i,j) then if (i,j + 1) is a zero the pixel is assigned a negative number otherwise it is assigned a positive number. Border following occurs according to the classical method in [3] then the algorithm returns to the start point of the border and continues the raster scan. The parent border of the current border is also recorded and classified as an outer border or a hole border according to the Table 1 in [2].

Algorithm two is the same as algorithm one except hole borders are not recorded only the outermost borders of a section of connected components. The OpenCV library contains other options for detecting blobs in an image the findContours algorithm is low cost in terms of processing power compared to functions in the library like houghCircles for example and performed successfully during the implementation of our project.

Once the coordinates are determined we can then find the centroid using the centroid formula,

$$\bar{x} = \frac{\sum_{x=x_{min}}^{x_{max}} xP_x(x)}{\sum_{x=x_{min}}^{x_{max}} P_x(x)}, \quad \bar{y} = \frac{\sum_{y=y_{min}}^{y_{max}} yP_y(y)}{\sum_{y=y_{min}}^{y_{max}} P_y(y)} \tag{2}$$

Where $P_x(x)$ and $P_y(y)$ are the sum of pixels in that column or row where laser dot is detected.

3. Applications and Tech Demos

Once the system can track laser dots and find their coordinates, the applications are mostly determined by the developer's own creativity and imagination.

For this project we created a variety of tech demos (see youtube playlist [4] and code) that shows the possible application this system can have. These include:

- Basic Draw Function - ability to draw on canvas (see figure 3)
- Mouse Function - ability to move and click computer mouse with laser dot
- Maze Demo - game to move through a maze without hitting walls

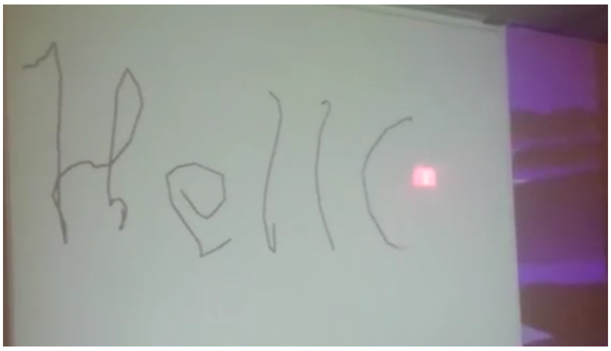


Figure 3: screen shot of basic draw

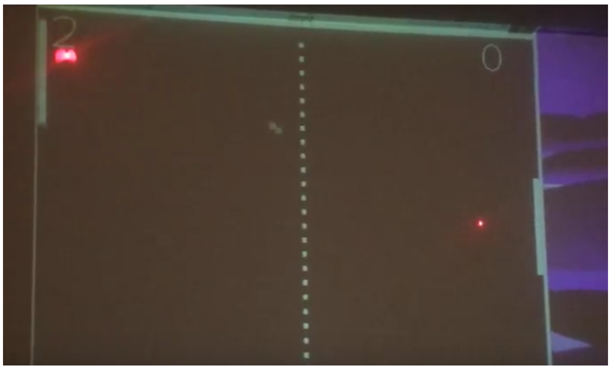
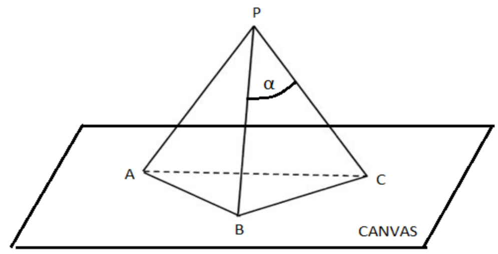
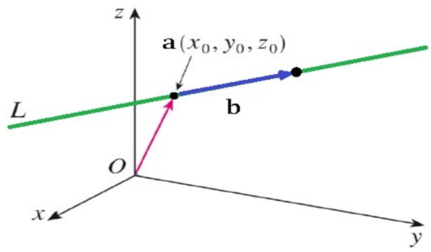


Figure 4: screen shot of pong game



$$\cos(\alpha) = \left(\frac{AP^2+BP^2-AB^2}{2*AP*BP}\right) \quad \cos(\alpha) = \left(\frac{AP^2+CP^2-AC^2}{2*AP*CP}\right) \quad \cos(\alpha) = \left(\frac{BP^2+CP^2-BC^2}{2*CP*BP}\right)$$

Figure 5: Geometry of resection problem



$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} x_1 \\ y_1 \\ z_1 \end{bmatrix} + \begin{bmatrix} u_1 \\ v_1 \\ w_1 \end{bmatrix} t_1 \quad \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} x_2 \\ y_2 \\ z_2 \end{bmatrix} + \begin{bmatrix} u_2 \\ v_2 \\ w_2 \end{bmatrix} t_2 \quad \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} x_3 \\ y_3 \\ z_3 \end{bmatrix} + \begin{bmatrix} u_3 \\ v_3 \\ w_3 \end{bmatrix} t_3$$

Figure 6: Parametric Equation of Line in 3D

- Target Shooting Demo - game to shoot at targets displayed
- Pong game - control the paddles with laser dots (see figure 4)
- Position Tracking Demo - see next section

4. Position Estimation with 3D Re-sectioning

An unconstrained object in 3D motion has six degrees of freedom: three translational and three rotational. Given the projections of three lines on a canvas all which intersect a common source, is it possible to find the position of the source? In finding the solution to this problem, it was uncovered that a similar problem, called the Three-Dimension Resection Problem (3DR) had been solved well in literature. The 3DR problem is to determine the lengths of a tetrahedral given three angles between the sides and the lengths opposite of the angles. The solution, though well documented, is not trivial and with multiple solutions. It is common instead to try to numerically solve the proper solution. Our problem, that of finding the position of the source of three lasers given their intersection with a plane, is somewhat more involved than just solving the 3DR problem. What follows is a breakdown of the procedure including approaches to deal with practical issues that arise.

4.1. Solving for the Lengths

If we examine the geometry that the three lasers make with the canvas, the shape that forms is an irregular tetrahedral. If we consider what we know, the end points of the lasers and the angles between the lasers, the problem of solving for the lengths of the lasers is exactly solving the 3DR problem. Fortunately, the geometry is rather straight forward as the problem boils down to solving the three equations defined by the law of cosines. The issue is this problem has no explicit solution that is trivial, and those explicit solutions which exist, they do not guarantee unique solutions. Here we opt to solve this problem numerically.

4.2. Solving for the Direction Vectors

To obtain the source coordinates, we are not just satisfied knowing the lengths of the lasers, but also their directions. With this knowledge we can directly calculate the source position. To solve for the direction vectors knowing the lengths of the lasers, the end points of the lasers, and finally knowing that the lasers have a common source, we can set up three equations vector equations defined by the equation of a line, restricting the lines to pass through one of the three lasers end points A,B,C as well as the common source P.

What we obtain are nine algebraic equations with 9 unknowns corresponding to the three entries of the three direction vectors we are interested in. Here t_1,t_2,t_3 represent the lengths of the lasers found, x_n,y_n,z_n the coordinates of the laser points on the canvas, and finally u_n,v_n,w_n the unit direction vectors each laser.

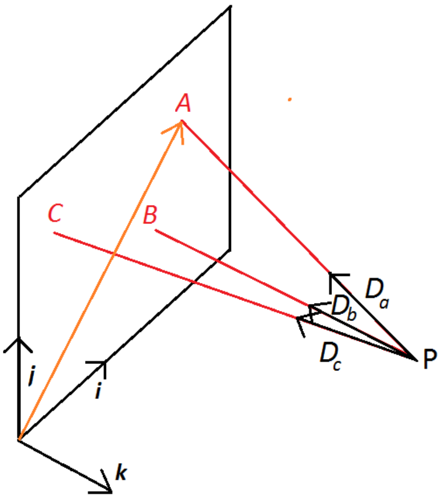


Figure 7: Directional vector and length visualization

$$\begin{bmatrix} x_2-x_1\\ y_2-y_1\\ z_2-z_1\\ x_3-x_1\\ y_3-y_1\\ z_3-z_1\\ x_2-x_3\\ y_2-y_3\\ z_2-z_3 \end{bmatrix} = \begin{bmatrix} t_1 & 0 & 0 & -t_2 & 0 & 0 & 0 & 0 & 0\\ 0 & t_1 & 0 & 0 & -t_2 & 0 & 0 & 0 & 0\\ 0 & 0 & t_1 & 0 & 0 & -t_2 & 0 & 0 & 0\\ t_1 & 0 & 0 & 0 & 0 & 0 & -t_3 & 0 & 0\\ 0 & t_1 & 0 & 0 & 0 & 0 & 0 & -t_3 & 0\\ 0 & 0 & t_1 & 0 & 0 & 0 & 0 & 0 & -t_3\\ 0 & 0 & 0 & -t_2 & 0 & 0 & t_3 & 0 & 0\\ 0 & 0 & 0 & 0 & -t_2 & 0 & 0 & t_3 & 0\\ 0 & 0 & 0 & 0 & 0 & -t_2 & 0 & 0 & t_3 \end{bmatrix} \begin{bmatrix} u_1\\ v_1\\ w_1\\ u_2\\ v_2\\ w_2\\ u_3\\ v_3\\ w_3 \end{bmatrix} \tag{3}$$

$$\begin{bmatrix} x_2-x_1\\ y_2-y_1\\ z_2-z_1\\ x_3-x_1\\ y_3-y_1\\ z_3-z_1\\ x_2-x_3\\ y_2-y_3\\ z_2-z_3 \end{bmatrix} = \begin{bmatrix} t_1 & 0 & 0 & -t_2 & 0 & 0 & 0 & 0 & 0\\ 0 & t_1 & 0 & 0 & -t_2 & 0 & 0 & 0 & 0\\ 0 & 0 & t_1 & 0 & 0 & -t_2 & 0 & 0 & 0\\ t_1 & 0 & 0 & 0 & 0 & 0 & -t_3 & 0 & 0\\ 0 & t_1 & 0 & 0 & 0 & 0 & 0 & -t_3 & 0\\ 0 & 0 & t_1 & 0 & 0 & 0 & 0 & 0 & -t_3\\ 0 & 0 & 0 & -t_2 & 0 & 0 & t_3 & 0 & 0\\ 0 & 0 & 0 & 0 & -t_2 & 0 & 0 & t_3 & 0\\ 0 & 0 & 0 & 0 & 0 & -t_2 & 0 & 0 & t_3 \end{bmatrix} \begin{bmatrix} X_1/\|V_1\|\\ Y_1/\|V_1\|\\ Z_1/\|V_1\|\\ X_2/\|V_2\|\\ Y_2/\|V_2\|\\ Z_2/\|V_2\|\\ X_3/\|V_3\|\\ Y_3/\|V_3\|\\ Z_3/\|V_3\| \end{bmatrix} \tag{4}$$

At first glance these equations look linear, and they are, however we must restrict the vectors found to be unit vectors, scaled by the previous found lengths. We place this restraint by dividing each vector by its norm. In doing so the system becomes nonlinear, and again a rather nasty set of equations to solve for. How do we then solve for the unit direction entries? Our solution is to solve them numerically.

$$\begin{aligned} V_1 = [X_1\ Y_1\ Z_1], V_2 = [X_2\ Y_2\ Z_2], \quad V_3 = [X_3\ Y_3\ Z_3] \\ D_a = \frac{V_1}{\|\|V_1\|\|}, D_b = \frac{V_2}{\|\|V_2\|\|}, \quad D_c = \frac{V_3}{\|\|V_3\|\|} \end{aligned} \tag{5}$$

4.3. Solving for Position

Now that we have obtained the lengths and direction vectors of the lasers and that we know the coordinates of the laser end points A,B,C we can now obtain the source position simply by vector subtraction.

$$P_{avg} = \frac{(A - D_a t_1) + (B - D_b t_2) + (C - D_c t_3)}{3} \tag{6}$$

4.4. Solving for Orientation (Roll,Pitch,Yaw)

If we consider an arbitrary rotation matrix, there are 9 unknowns. These entries are determined by the angles, axis, and order of the consecutive rotations in sequence. For roll, pitch , yaw the order is a 3-2-1 rotation starting with yaw, pitch, then roll. The rotation matrix has the following form:

$$R(\phi,\theta,\psi) = \begin{bmatrix} c\psi c\theta & c\psi s\phi s\theta - cs\psi & s\phi s\psi + c\phi c\psi s\theta \\ c\theta s\psi & c\phi c\, psi + ss & c\phi s\psi s\theta - c\psi s\phi \\ -s\theta & c\theta s\phi & c\phi c\theta) \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix} \tag{7}$$

$$\begin{aligned} \phi &= \tan^{-1}(r_{21}/r_{11}) \\ \theta_1 &= \tan^{-1}(-r_{31}/\sqrt(r_{32}^2 + r_{33}^2)), \qquad \theta \in (-\pi/2, \pi/2) \\ \theta_2 &= \tan^{-1}(-r_{31}/\sqrt(r_{32}^2 + r_{33}^2)), \qquad \theta \in (\pi/2, 3\pi/2) \\ \psi &= \tan^{-1}(r_{32}/r_{33}) \end{aligned} \tag{8}$$

Yaw, pitch, and roll angles can be solved by examining elements. It is desirable to define angles in terms of tan2, which keeps track of the quadrant. There are two different solutions for pitch given the expected quadrant it should be in. For our application, we expect pitch to be between -pi/2 and pi/2.

Next, we need to determine how to find the rotation matrix for each update. This is done by considering an initial orientation with the initial direction vectors of the lasers VA_o,VB_o,VC_o and the current direction vectors of the lasers VA,VB,VC to determine the rotation matrix R which acts to map the two.

$$\begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix} \begin{bmatrix} DA_{xo} \\ DA_{yo} \\ DA_{zo} \end{bmatrix} = \begin{bmatrix} DA_x \\ DA_y \\ DA_z \end{bmatrix} \tag{9}$$

$$\begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix} \begin{bmatrix} DB_{xo} \\ DB_{yo} \\ DB_{zo} \end{bmatrix} = \begin{bmatrix} DB_x \\ DB_y \\ DB_z \end{bmatrix} \tag{10}$$

$$\begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix} \begin{bmatrix} DC_{xo} \\ DC_{yo} \\ DC_{zo} \end{bmatrix} = \begin{bmatrix} DC_x \\ DC_y \\ DC_z \end{bmatrix} \tag{11}$$

Here we have 9 equations in 9 unknowns, as such we can solve this linear system. Here, because of error introduced into the system, we opt to find the nearest solution, in case one does not exist. The system can be rewritten to solve for the components of R.

$$\begin{bmatrix} DA_x \\ DA_y \\ DA_z \\ DB_x \\ DB_y \\ DB_z \\ DC_x \\ DC_y \\ DC_z \end{bmatrix}_b = \begin{bmatrix} DA_{xo} & 0 & 0 & DA_{yo} & 0 & 0 & DA_{zo} & 0 & 0 \\ 0 & DA_{xo} & 0 & 0 & DA_{yo} & 0 & 0 & DA_{zo} & 0 \\ 0 & 0 & DA_{xo} & 0 & 0 & DA_{yo} & 0 & 0 & DA_{zo} \\ DB_{xo} & 0 & 0 & DB_{yo} & 0 & 0 & DB_{zo} & 0 & 0 \\ 0 & DB_{xo} & 0 & 0 & DB_{yo} & 0 & 0 & DB_{zo} & 0 \\ 0 & 0 & DB_{xo} & 0 & 0 & DB_{yo} & 0 & 0 & DB_{zo} \\ DC_{xo} & 0 & 0 & DC_{yo} & 0 & 0 & DC_{zo} & 0 & 0 \\ 0 & DC_{xo} & 0 & 0 & DC_{yo} & 0 & 0 & DC_{zo} & 0 \\ 0 & 0 & DC_{xo} & 0 & 0 & DC_{yo} & 0 & 0 & DC_{zo} \end{bmatrix}_A \begin{bmatrix} r_{11} \\ r_{21} \\ r_{31} \\ r_{12} \\ r_{22} \\ r_{23} \\ r_{13} \\ r_{23} \\ r_{33} \end{bmatrix}_x \tag{12}$$

The above equation can then be solved using the least squares formula,

$$x = (A^T A)^{-1} A^T b \tag{13}$$

4.5. Methods of Implementation

MATLAB and Pythons fsolve In solving the lengths and directions we applied a numeric solver which requires an initial value to process. Unfortunately providing just any initial value will not guarantee the correct value. There are two simple approaches which attempt to avoid the wrong solution. The first approach is to simply use the previous solutions for lengths and directions. This works assuming that the both translation and rotation are slow. Approach 1:

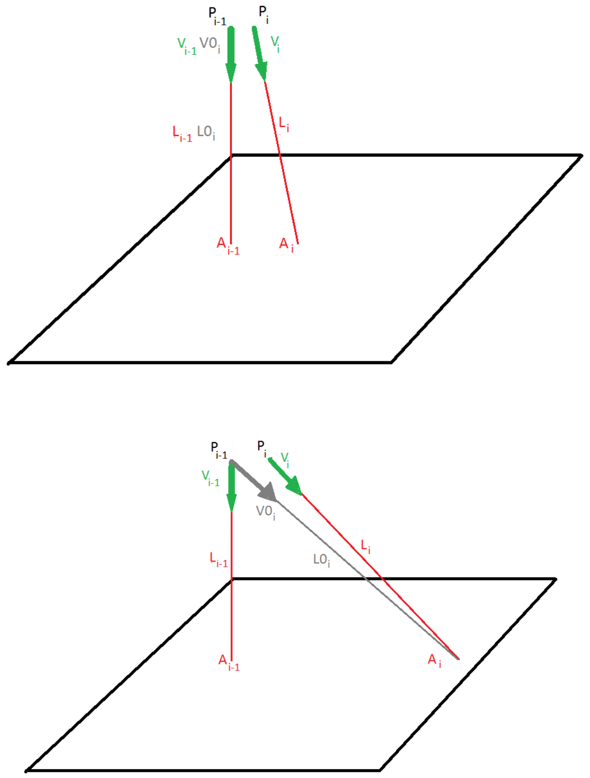


Figure 8: Approach 1 visualization (above) approach 2 visualization (below)

$$Lo_i = L_{i-1}, \qquad Vo_i = V_{i-1} \tag{14}$$

While it may be reasonable to limit the translation, limiting the rotation is simply not realistic. It may be difficult for one to translate a great distance between updates, but it is very feasible to change orientation with the flick of a wrist. For this operation we instead use the previous position estimate and the new known laser end points to estimate the direction and length of the lasers.

Approach 2:

$$Lo_i = L_{i-1}, \qquad Vo_i = \frac{A_i - P_{i-1}}{||A_i - P_{i-1}||} \tag{15}$$

Simulations of both cases within MATLAB and Python show that the second approach is more robust to fast orientations changes which are more likely to occur than quick displacements. Both the lengths of the lasers and their direction vectors were solved using MATLABs and Pythons fsolve(). Computation time using MATLAB was on the order of .1 seconds compared to Pythons .001 seconds.

4.6. Initialization

Both solving the lengths and directions of the lasers use a numeric solver which requires the prior position for guessing an initial value. Also the angles between each of the lasers are required. To obtain roll, pitch, and yaw, the initial direction vectors of the lasers are required. All of these values can be found during an initialization procedure upon start up. Initial Position: Here we need to know a good guess for position in order to estimate the lengths and directions of the lasers as mentioned earlier. Here we opt to simply ask the user to stand at the left-lower hand corner of the screen corresponding to the origin, and then stand an equal distance from the board compared to the height of the board. For example, if the board is 2m tall, the user would stand 2m away from the lower left hand corner. Because this position is known, this is used to initialize position.

$$P_o = \begin{bmatrix} 0 \\ 0 \\ -2 \end{bmatrix} \text{(meters)} \tag{16}$$

4.7. Initial angles

In order to obtain estimates of our orientation, we must compare the unit direction vectors of the three lasers at a given point in time to an initial, zeroed angle, orientation. There are two ways to do this. The first way is to simply predefine an zero orientation and a corresponding set of laser directions. This is only feasible if you know the angles between the lasers. For example lets assume laser A is pointing in the negative K-axis, laser B is made by rotating laser A by an angle AB about the J-axis, and laser C is defined by the restrictions on angle AC and BC. The difficulty is in calculating the direction of laser C, because the axis of rotation is not known trivially. The system is constrained, so a relation can be found, however for the purpose of this project, we opted to use the initial direction vectors to define the zeroed angles. This is done by using the initial position and the measured end laser points, and back solving the direction vectors.

$$\begin{bmatrix} L_{Ao} \\ L_{Bo} \\ L_{Co} \end{bmatrix} = \begin{bmatrix} ||A_o - P_o|| \\ ||B_o - P_o|| \\ ||C_o - P_o|| \end{bmatrix} \tag{17}$$

$$D_{Ao} = \frac{A_o - P_o}{L_{AO}}, \quad D_{Bo} = \frac{B_o - P_o}{L_{BO}}, \quad D_{Co} = \frac{C_o - P_o}{L_{CO}} \tag{18}$$

4.8. Defining angles between lasers

So far we have assumed that we know the angles between the lasers, however if the lasers are not mounted precision, the angles may not be known well. In our case the mounting hardware for the lasers were not rigid as to allow for movement of the lasers to ensure that they intersected a point in space. So every time we tested, the lasers angles may have changed. To get around this issue, we simply back calculated the angles assuming that the initial position was known, as described above.

$$\begin{bmatrix} \alpha_{AB} \\ \alpha_{AC} \\ \alpha_{BC} \end{bmatrix} = \begin{bmatrix} \cos^{-1}(\frac{AP^2+BP^2-AB^2}{2AP*BP}) \\ \cos^{-1}(\frac{AP^2+CP^2-AC^2}{2AP*CP}) \\ \cos^{-1}(\frac{BP^2+CP^2-BC^2}{2BP*CP}) \end{bmatrix}, \quad \begin{bmatrix} AP \\ BP \\ CP \end{bmatrix} = \begin{bmatrix} LA_o \\ LB_o \\ LC_o \end{bmatrix} \tag{19}$$

In the likely case that the angles found are note equal, the program must be able to track which laser points belong to which laser. In our application we used OpenCVs contour function, which separates blobs, however it does not keep track of which blobs are which, but instead applies a kernel from the top left to bottom right. The implication of this is that the lasers eventually swap order and the position estimation fails. A simple solution to resolve this is to assume that given a correct set of prior laser points tags, that the correct tags for the next iteration will result minimizing the displacements. This was found to resolve the swapping issue.

$$\begin{bmatrix} A_i \\ B_i \\ C_i \end{bmatrix} = \begin{bmatrix} P_j \\ P_k \\ P_l \end{bmatrix} \tag{20}$$

where j, k, l satisfy,

$$\underset{j,k,l}{\operatorname{argmin}}(||A_{i-1} - P_j|| + ||B_{i-1} - P_k|| + ||C_{i-1} - P_l||) \tag{21}$$

for j,k,l = 1,2,3 and $j \neq k \neq l$.

4.9. Simulation

Two different simulations were carried out, one in MATLAB to visualize the results, and one ported to Python which was the platform used for the project. The MATLAB simulation proved to be a very valuable tool in evaluating the validity of the numeric solutions, though computation time proved to be slow on the order of a tenth of a second. Here it was discovered that using the prior position plus the new laser positions to estimate the new laser lengths and directions 10 proved to be better than just using the past directions and lengths 9.

After the methods passed testing in MATLAB, the program was transferred to Python showing nearly exacting results in much less time. The process operates as fast as 100Hz.

4.10. Test Results

The Position Estimation Program runs as one of the demos in the Laser Board Program. To begin the user selects the Laser Position Demo from the main GUI. Next the user is asked to stand at the bottom left corner of the screen at a distance equal to that of the height of the screen, wall while keeping the lasers on the board as close to the bottom left corner as possible. Once the user has done this they can press reset to initialize and run the position and orientation estimation. The user is given the estimated position in xyz coordinates and the orientation in yaw, pitch, and roll angles. Also the user is presented with a circle of varying location and size which is a visualization of their estimated position. The final results of the tracking demonstrate that position and orientation can be found with reasonable accuracy and speed. The user receives accurate position despite rotation the handles and receives accurate angle despite a change in location. It has been found, however, that estimates can be off if the user does not take care to ensure the initialization is accurate. Also the user must ensure the in fact the lasers meet at a point. If the points do not, then the solution becomes less accurate. That is the geometry of the lasers must make a tetrahedral.

5. Conclusion

We were successfully able to develop a low budget laser pointer interactive system using multiple laser pointers, a webcam and a projector. The paper went over the process required to determine the coordinate of multiple laser dots, these includes camera view homography transformation, noise attenuation, localized color intensity thresholding, and contour finding/coordinate calculation.

Using these components we were able to successfully demonstrate some simple application of our system such as basically drawing, mouse control, and maze completion games.

The system can also be used to perform complex actions such as position and orientation tracking using 3D re-sectioning of three laser pointers.

Figure 12: A screen shot of position and orientation tracking demo [4]

6. References

[1] R. I. Hartley, "In defense of the eight-point algorithm," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 19, no. 6, pp. 580–593, 1997.

[2] S. Suzuki *et al.*, "Topological structural analysis of digitized binary images by border following," *Computer Vision, Graphics, and Image Processing*, vol. 30, no. 1, pp. 32–46, 1985.

[3] A. Rosenfeld and A. C. Kak, *Digital picture processing*. Elsevier, 2014, vol. 1.

[4] K. Kawabata. Laserboard project. Youtube. [Online]. Available: https://www.youtube.com/watch?v=n__QOrV0HoM&list=UU_X92xQyqnSpMQr1m20O_Fw

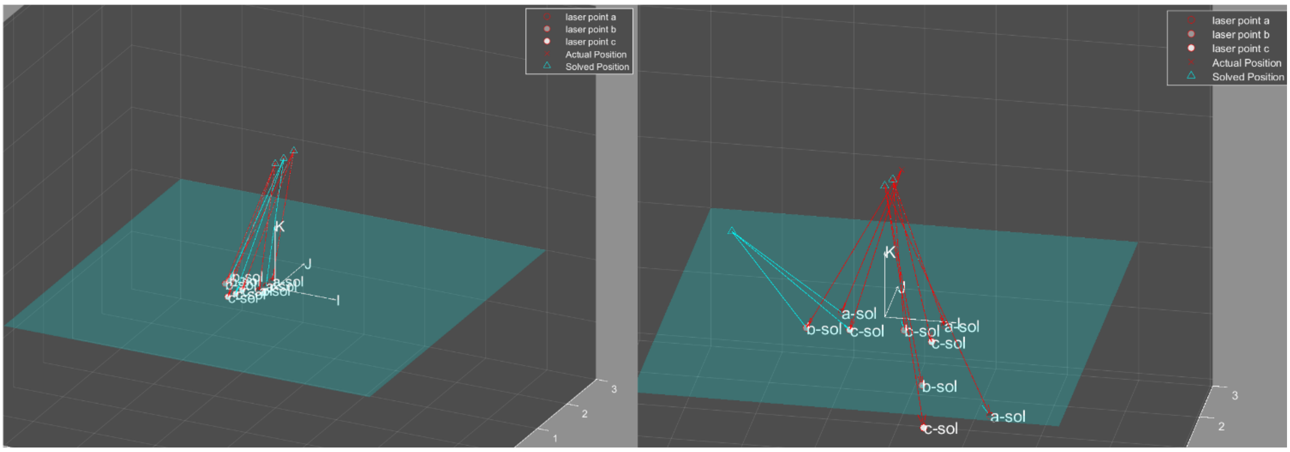


Figure 9: Estimation based on prior lengths and direction vectors small changes in orientation(Left) large changes in orientation (Right)

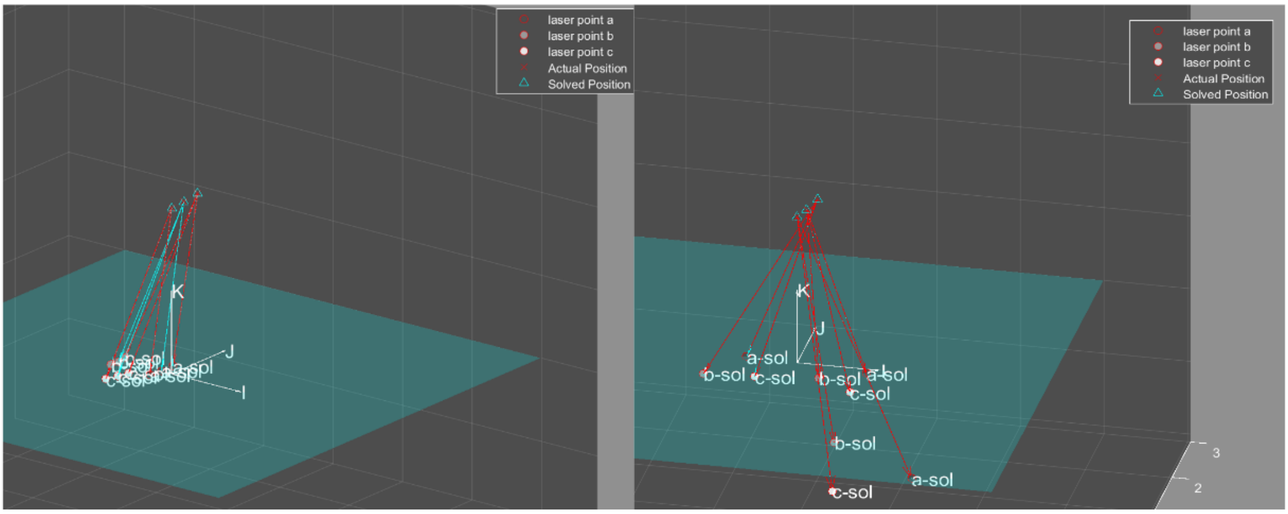


Figure 10: Estimation based on prior position and new lasers endpoints small changes in orientation (Left) large changes in orientation (Right)

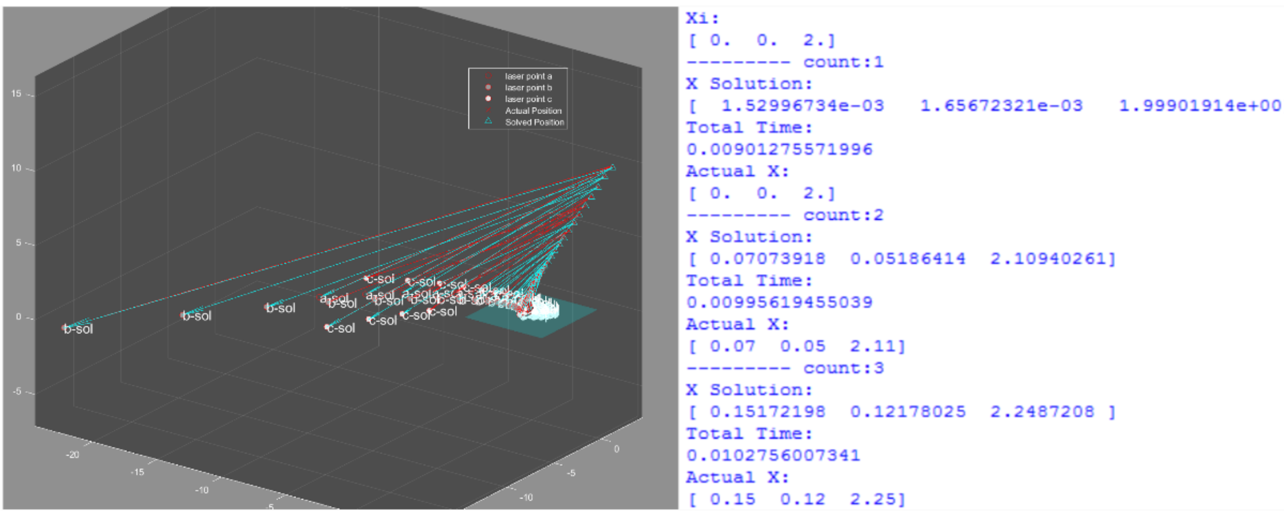


Figure 11: Iterative simulation comparison in MATLAB and Python

7. Source Code

Main code

```python
from __future__ import division
import cv2
import numpy as np
import time
import win32api
import win32con
# import win32gui
import multiprocessing
import os
from LaserPosOrientEstimator import LaserPosOrientEstimator


class LaserBoard:
    def __init__(self, height, width, src=0):
        self.vid = cv2.VideoCapture(src)
        self.board_h = height
        self.board_w = width
        self.canvas = np.zeros((self.board_h, self.board_w), dtype=np.uint8)
        self.canvas_pos = []
        self.H = []
        self.canvas_bg = np.zeros((self.board_h, self.board_w, 3), dtype=np.uint8)
        self.canvas_thresh_c = 1.1  # increase the threshold by constant
        self.q_frame = multiprocessing.Queue()
        self.q_key = multiprocessing.Queue()
        self.show_thread = multiprocessing.Process(target=show_loop,
                                                   args=(self.q_frame, self.q_key, [self.board_h, self.board_w]))
        self.min_dot_size = 10
        self.lpoe = LaserPosOrientEstimator()

    def laser_board_run(self):
        self.show_thread.start()
        self.calibration_setup()
        while 1:
            os.system('cls')
            print 'Choose program to run'
            print '1. basic draw'
            print '2. laser mouse'
            print '3. target shooting'
            print '4. position tracking demo'
            print '5. maze demo'
            print '6. pong game'
            print '7. camera/tracking test'
            print '8. recalibrate'
            print '9. quit'
            choice = raw_input()

            if choice == '1':
                self.basic_draw()
            elif choice == '2':
                self.mouse_fun()
            elif choice == '3':
                self.target_shoot()
            elif choice == '4':
                self.pos_tracking_demo()
            elif choice == '5':
                self.maze_demo()
            elif choice == '6':
                self.pong_demo()
            elif choice == '7':
                self.camera_view()
            elif choice == '8':
                self.calibration_setup()
            elif choice == '9':
                self.release()

            self.canvas = np.zeros((self.board_h, self.board_w), dtype=np.uint8)

    def detector(self, image):
        _, contours, _ = cv2.findContours(image, cv2.RETR_LIST, cv2.CHAIN_APPROX_NONE)
        coord = []
```

```python
        for cnt in contours:
            if len(cnt) > self.min_dot_size:
                temp = [cv2.moments(cnt)[x] for x in ['m10', 'm01', 'm00']]
                if temp[2] != 0:
                    coord.append((int(temp[0]/temp[2]), int(temp[1]/temp[2])))
        return coord

    def basic_draw(self):
        self.canvas = np.zeros([self.board_h, self.board_w], dtype=np.uint8)
        prev_pos = []
        while 1:
            ret, view = self.vid.read()
            board = cv2.warpPerspective(view, self.H, (self.board_w, self.board_h))
            dt_view = self.find_dots(board)
            key_points = self.detector(dt_view)
            if key_points:
                if prev_pos:
                    cv2.line(self.canvas, tuple(key_points[0]), tuple(prev_pos), 255, 2)
                    prev_pos = key_points[0]
                else:
                    prev_pos = key_points[0]
                    cv2.line(self.canvas, tuple(key_points[0]), tuple(prev_pos), 255, 2)
            else:
                prev_pos = []

            self.q_frame.put(255 - self.canvas)
            if not self.q_key.empty():
                keypress = self.q_key.get_nowait()
                if keypress == ord('q'):
                    cv2.destroyWindow('setup')
                    self.q_frame.put(255 - np.zeros([self.board_h, self.board_w], dtype=np.uint8))
                    return
                elif keypress == ord('r'):
                    self.canvas = np.zeros([self.board_h, self.board_w], dtype=np.uint8)
                    print 'canvas cleared'
            cv2.waitKey(1)
            cv2.imshow('setup', self.canvas_bg)

    def mouse_fun(self):
        anchor_pos = np.array((0, 0))
        start_time = time.clock()
        while 1:
            ret, view = self.vid.read()

            if not self.q_key.empty():
                keypress = self.q_key.get_nowait()
                if keypress == ord('q'):
                    self.q_frame.put(np.zeros([self.board_h, self.board_w], dtype=np.uint8))
                    return
                elif keypress == ord('r'):
                    self.canvas = np.zeros([self.board_h, self.board_w], dtype=np.uint8)
                    print('canvas cleared')

            board = cv2.warpPerspective(view, self.H, (self.board_w, self.board_h))
            dt_view = self.find_dots(board)

            key_points = self.detector(dt_view)
            if len(key_points) > 0:
                pos = np.array((int(key_points[0][0]) + win32api.GetSystemMetrics(0), int(key_points[0][1]) + 30))
                win32api.SetCursorPos(pos)
                if np.linalg.norm(pos - anchor_pos) > 30:
                    anchor_pos = pos
                    start_time = time.clock()
                else:
                    if time.clock() - start_time > 2:
                        win32api.mouse_event(win32con.MOUSEEVENTF_RIGHTDOWN, pos[0], pos[1], 0, 0)
                        win32api.mouse_event(win32con.MOUSEEVENTF_RIGHTUP, pos[0], pos[1], 0, 0)
                        start_time = time.clock()

    def target_shoot(self):
        points = 0
        start_time = time.clock()
```

```
        target = ((np.random.rand(1, 1) * .9 + .05) * self.board_w,
                  (np.random.rand(1, 1) * .9 + .05) * self.board_h)
        while 1:
            ret, view = self.vid.read()
            if not self.q_key.empty():
                keypress = self.q_key.get_nowait()
                if keypress == ord('q'):
                    self.q_frame.put(np.zeros([self.board_h, self.board_w], dtype=np.uint8))
                    return
                elif keypress == ord('r'):
                    target = ((np.random.rand(1, 1) * .9 + .05) * self.board_w,
                              (np.random.rand(1, 1) * .9 + .05) * self.board_h)
                    points = 0

            board = cv2.warpPerspective(view, self.H, (self.board_w, self.board_h))

            dt_view = self.find_dots(board)

            target_range = np.zeros((self.board_h, self.board_w), dtype=np.uint8)
            cv2.circle(target_range, target, 10, 255, -1)

            if cv2.bitwise_and(dt_view, target_range).any():
                target = ((np.random.rand(1, 1) * .9 + .05) * self.board_w,
                          (np.random.rand(1, 1) * .9 + .05) * self.board_h)
                points += 1

            cv2.putText(target_range, str(points) + ' targets shot', (10, 20), cv2.FONT_HERSHEY_PLAIN, 2, 255, 2)
            cv2.putText(target_range, str(int(time.clock() - start_time)) +
                        ' seconds', (10, 40), cv2.FONT_HERSHEY_PLAIN, 2, 255, 2)
            self.q_frame.put(target_range)
            # cv2.imshow('setup', dt_view)

    def pos_tracking_demo(self):
        scale = np.array([200, -200])
        start = False
        dialog1 = 'please position a window width away'
        dialog2 = 'from the bottom left corner and press r'
        offset = np.array([self.board_w / 2, self.board_h / 2])
        while 1:

            ret, view = self.vid.read()
            board = cv2.warpPerspective(view, self.H, (self.board_w, self.board_h))
            dt_view = self.find_dots(board)
            key_points = self.detector(dt_view)

            if not self.q_key.empty():
                keypress = self.q_key.get_nowait()
                if keypress == ord('q'):
                    self.q_frame.put(np.zeros([self.board_h, self.board_w], dtype=np.uint8))
                    return
                elif keypress == ord('r'):
                    start = False
                    screen_height = 2
                    print 'please position the foci point one meter from the origin and press r'
                    while 1:
                        ret, view = self.vid.read()
                        board = cv2.warpPerspective(view, self.H, (self.board_w, self.board_h))
                        dt_view = self.find_dots(board)
                        key_points = self.detector(dt_view)
                        if len(key_points) == 3:
                            self.lpoe.calibrate_angles(key_points[0], key_points[1],
                                                       key_points[2], screen_height, self.board_h)
                            start = True
                            print "started detection"
                            break

            if start and (len(key_points) == 3):
                est_pos, order = self.lpoe.getPos(key_points[0], key_points[1], key_points[2])
                est_orient = self.lpoe.getRPY()
                dialog1 = '[     X,       Y,       Z],[    Roll,    Pitch,    Yaw]:'
                dialog2 = str(np.round(est_pos, 2)) + ', ' + str(np.round(est_orient, 2))
```

```
                cv2.circle(dt_view, tuple((np.multiply(np.array(est_pos[0:2]), scale) + offset).astype(int)),
                           int(abs(est_pos[2])*5), 255, -1)
                cv2.putText(dt_view, 'A', key_points[order[0]], cv2.FONT_HERSHEY_PLAIN, 2, 255, 2)
                cv2.putText(dt_view, 'B', key_points[order[1]], cv2.FONT_HERSHEY_PLAIN, 2, 255, 2)
                cv2.putText(dt_view, 'C', key_points[order[2]], cv2.FONT_HERSHEY_PLAIN, 2, 255, 2)

            cv2.putText(dt_view, dialog1, (10, 20), cv2.FONT_HERSHEY_PLAIN, 2, 255, 2)
            cv2.putText(dt_view, dialog2, (10, 45), cv2.FONT_HERSHEY_PLAIN, 2, 255, 2)

            self.q_frame.put(dt_view)

    def maze_demo(self):
        maze_map = cv2.resize(cv2.imread('maze.png'), (self.board_w, self.board_h))
        state = 0
        start_time = 0
        end_time = 0
        prev_pos = []

        while 1:
            maze = maze_map.copy()
            ret, view = self.vid.read()
            if not self.q_key.empty():
                keypress = self.q_key.get_nowait()
                if keypress == ord('q'):
                    self.q_frame.put(np.zeros([self.board_h, self.board_w], dtype=np.uint8))
                    return
                elif keypress == ord('r'):
                    self.canvas = np.zeros([self.board_h, self.board_w], dtype=np.uint8)
                    print('canvas cleared')

            board = cv2.warpPerspective(view, self.H, (self.board_w, self.board_h))
            dt_view = self.find_dots(board)
            pos_color = set(maze_map[dt_view != 0, 2])

            if 0 not in pos_color:
                if 200 in pos_color and state == 0:
                    state = 1
                    start_time = time.clock()
                elif 100 in pos_color and state == 1:
                    state = 2
                    end_time = time.clock() - start_time
            else:
                state = 0
                start_time = 0

            if state == 1:
                cv2.putText(maze, "{0:.2f}".format(time.clock() - start_time) + ' seconds',
                            (10, 20), cv2.FONT_HERSHEY_PLAIN, 2, 0, 2)
                key_points = self.detector(dt_view)
                if key_points:
                    if prev_pos:
                        cv2.line(self.canvas, tuple(key_points[0]), tuple(prev_pos), 255, 2)
                        prev_pos = key_points[0]
                    else:
                        prev_pos = key_points[0]
            elif state == 2:
                if end_time < 6:
                    cv2.putText(maze, 'you have finished the rat race in ' + "{0:.2f}".format(end_time) +
                                ' seconds', (10, 20), cv2.FONT_HERSHEY_PLAIN, 2, 0, 2)
                else:
                    cv2.putText(maze, 'You could do better', (10, 20),
                                cv2.FONT_HERSHEY_PLAIN, 2, 0, 2)
            else:
                cv2.putText(maze, 'Start Light Gray, Go to Dark Grey, Avoid Walls', (10, 20),
                            cv2.FONT_HERSHEY_PLAIN, 2, 0, 2)
                prev_pos = []
            self.q_frame.put(maze)

    def pong_demo(self):
        paddle_width = 80

```

```
        def hit(pad_l, pad_r, ball):
            if ball[0] < 10:
                rel_pos = pad_l[1] - ball[1]
                if abs(rel_pos) > paddle_width:
                    return None
            elif self.board_w - 10 < ball[0]:
                rel_pos = pad_r[1] - ball[1]
                if abs(rel_pos) > paddle_width:
                    return None
            else:
                return None

            ball_vel[1] += int(-rel_pos*20/paddle_width)
            if ball_vel[1] > 30:
                ball_vel[1] = 30
            ball_vel[0] = -ball_vel[0]
            return rel_pos

        pong_map = cv2.resize(cv2.imread('pong_map.png', 0), (self.board_w, self.board_h))
        begin = False
        vel = [15, 10]
        score = [0, 0]
        ball_pos = np.array([int(self.board_w / 2), int(self.board_h / 2)])
        paddle_l = [10, int(self.board_h / 2)]
        paddle_r = [self.board_w - 10, int(self.board_h / 2)]
        while 1:
            ret, view = self.vid.read()
            if not self.q_key.empty():
                keypress = self.q_key.get_nowait()
                if keypress == ord('q'):
                    self.q_frame.put(np.zeros([self.board_h, self.board_w], dtype=np.uint8))
                    return
                elif keypress == ord('r'):
                    score = [0, 0]
                    begin = False
                    ball_pos = np.array([int(self.board_w / 2), int(self.board_h / 2)])
                    ball_vel = np.array(vel)
                    print('canvas cleared')

            if begin:
                ball_pos = ball_pos + ball_vel
                if hit(paddle_l, paddle_r, ball_pos):
                    pass
                elif ball_pos[0] > self.board_w:
                    begin = False
                    score[0] += 1
                    ball_pos = np.array([int(self.board_w / 2), int(self.board_h / 2)])
                    ball_vel = np.array(vel)
                elif ball_pos[0] < 0:
                    begin = False
                    score[1] += 1
                    ball_pos = np.array([int(self.board_w / 2), int(self.board_h / 2)])
                    ball_vel = np.array(vel)
                    ball_vel[0] = -ball_vel[0]
                elif ball_pos[1] > self.board_h or ball_pos[1] < 0:
                    ball_vel[1] = -ball_vel[1]
            else:
                ball_pos = np.array([int(self.board_w / 2), int(self.board_h / 2)])
                ball_vel = np.array(vel)

            board = cv2.warpPerspective(view, self.H, (self.board_w, self.board_h))
            dt_view = self.find_dots(board)
            key_points = self.detector(dt_view)
            if len(key_points) > 0:
                if len(key_points) == 2:
                    begin = True
                for key_point in key_points:
                    if key_point[0] < self.board_w/2:
                        paddle_l[1] = np.array(key_point[1])
                    else:
                        paddle_r[1] = np.array(key_point[1])

            canvas = cv2.bitwise_or(pong_map.copy(), dt_view)
```

```python
            cv2.rectangle(canvas, tuple(paddle_l + np.array([-10, -paddle_width])),
                          tuple(paddle_l + np.array([0, paddle_width])), 255, -1)
            cv2.rectangle(canvas, tuple(paddle_r + np.array([0, -paddle_width])),
                          tuple(paddle_r + np.array([10, paddle_width])), 255, -1)
            cv2.rectangle(canvas, tuple(ball_pos + np.array([-5, -5])), tuple(ball_pos + np.array([5, 5])), 255, -1)
            cv2.putText(canvas, str(score[0]), (10, 60), cv2.FONT_HERSHEY_PLAIN, 5, 255, 2)
            cv2.putText(canvas, str(score[1]), (self.board_w - 100, 60), cv2.FONT_HERSHEY_PLAIN, 5, 255, 2)
            self.q_frame.put(canvas)

    def camera_view(self):
        while 1:
            ret, view = self.vid.read()
            if not self.q_key.empty():
                keypress = self.q_key.get_nowait()
                if keypress == ord('q'):
                    self.q_frame.put(np.zeros([self.board_h, self.board_w], dtype=np.uint8))
                    return
                elif keypress == ord('r'):
                    self.canvas = np.zeros([self.board_h, self.board_w], dtype=np.uint8)
                    print('canvas cleared')

            board = cv2.warpPerspective(view, self.H, (self.board_w, self.board_h))

            dt_view = self.find_dots(board)
            key_points = self.detector(dt_view)
            print len(key_points)
            if len(key_points) > 0:
                for i in key_points:
                    cv2.circle(dt_view, i, 10, (100, 0, 0), -1)
            self.q_frame.put(255-dt_view)

    def calibration_setup(self):
        cv2.namedWindow('setup')
        self.q_frame.put(255 - np.zeros([self.board_h, self.board_w], dtype=np.uint8))
        self.position_setup()
        self.color_setup()
        cv2.destroyWindow('setup')

    def position_setup(self):
        calibration_var = [False, np.zeros([4, 2]), 0]

        def corners_clicked(event, x, y, _, calibration_stats):
            if event == cv2.EVENT_LBUTTONDOWN:
                if not calibration_stats[0]:
                    calibration_stats[1][calibration_stats[2], :] = [x, y]
                    calibration_stats[2] += 1
                    if calibration_stats[2] == 4:
                        calibration_stats[0] = True

        cv2.setMouseCallback('setup', corners_clicked, calibration_var)
        print 'calibrating screen corners please click corners from top left going clockwise'
        while 1:
            ret, view = self.vid.read()
            keypress = cv2.waitKey(1) & 0xFF
            if keypress == ord('e'):
                if calibration_var[0]:
                    self.H = cv2.getPerspectiveTransform(np.float32(calibration_var[1]), np.float32(
                        [[0, 0], [self.board_w, 0], [self.board_w, self.board_h], [0, self.board_h]]))
                    print 'position calibration complete'
                    self.canvas_pos = calibration_var[1]
                    return
                else:
                    print 'not enough corners selected'
            elif keypress == ord('r'):
                calibration_var = [False, np.zeros([4, 2]), 0]
                cv2.setMouseCallback('setup', corners_clicked, calibration_var)
                print 'corners reset'
            elif keypress == ord('q'):
                self.release()

```

```
            for i in calibration_var[1]:
                cv2.circle(view, tuple(int(x) for x in i), 2, (255, 0, 0), -1)

            cv2.imshow('setup', view)

    def color_setup(self):
        def nothing():
            pass
        cv2.createTrackbar('threshold', 'setup', 0, 200, nothing)
        cv2.setTrackbarPos('threshold', 'setup', 110)
        cv2.resizeWindow('setup', self.board_w, self.board_h)
        temp_canvas_bg = np.zeros((self.board_h, self.board_w, 3), dtype=np.uint8)

        max_frames_grabbed = 0
        while 1:
            max_frames_grabbed += 20
            print 'estimating max background intensity'
            for i in range(1, max_frames_grabbed):
                ret, frame = self.vid.read()
                frame = cv2.warpPerspective(frame, self.H, (self.board_w, self.board_h))
                temp_canvas_bg = np.maximum(frame, temp_canvas_bg)
                time.sleep(.1)

            print 'please choose optimal thresholding value (remove noise but keep laser dots)'
            while 1:
                ret, frame = self.vid.read()
                frame = cv2.warpPerspective(frame, self.H, (self.board_w, self.board_h))

                temp_thresh = cv2.getTrackbarPos('threshold', 'setup')/100

                self.canvas_bg = (temp_canvas_bg.astype(np.float64) * temp_thresh)
                self.canvas_bg[self.canvas_bg > 255] = 250

                dt_view = self.find_dots(frame)
                keypress = cv2.waitKey(1) & 0xFF
                if keypress == ord('e'):
                    print 'color calibration complete'
                    self.canvas_thresh_c = temp_thresh
                    self.canvas_bg = temp_canvas_bg
                    cv2.destroyWindow('test')
                    return
                elif keypress == ord('r'):
                    temp_canvas_bg = np.zeros((self.board_h, self.board_w, 3), dtype=np.uint8)
                    print 'color range reset'
                    break
                elif keypress == ord('q'):
                    self.release()
                cv2.imshow('test', self.canvas_bg)
                cv2.imshow('setup', dt_view)

    def find_dots(self, view):
        return (view > self.canvas_bg).any(2) * np.uint8(255)

    def release(self):
        self.vid.release()
        if self.show_thread:
            self.show_thread.terminate()
        cv2.destroyAllWindows()
        quit()


def show_loop(q_frame, q_key, dim):
    cv2.namedWindow('Board')
    from_queue = []
    while 1:
        keypress = cv2.waitKey(1) & 0xFF

        if keypress != 255:
            q_key.put(keypress)
        if not q_frame.empty():
            from_queue = q_frame.get_nowait()
            cv2.imshow('Board', from_queue)
        else:
```

```
            if len(from_queue) == 0:
                cv2.imshow('Board', 255 - np.zeros(dim, dtype=np.uint8))

if __name__ == "__main__":
    res = 200
    lb = LaserBoard(3 * res, 4 * res, 0)
    # lb = LaserBoard(2,1)
    lb.laser_board_run()
```

Laser position and orientation estimator class

```
from __future__ import division
import matplotlib.pyplot as plt
import numpy as np
from scipy.optimize import fsolve
from scipy.optimize import broyden1
import time
from numpy.linalg import inv
from numpy.linalg import norm


class LaserPosOrientEstimator:
    def __init__(self, angles=(15, 14, 13)):
        self.angles = np.array(angles)
        self.prev_pos = np.array([0, 0, 2])
        self.screen_height = 2
        self.board_h = 0
        self.orient_mat = []
        self.L = []
        self.AB = []
        self.BC = []
        self.CA = []
        self.A = []
        self.B = []
        self.C = []
        self.Ao = []
        self.Bo = []
        self.Co = []
        self.VA = []
        self.VB = []
        self.VC = []

    def calibrate_angles(self, A, B, C, screen_height, boardH):
        self.board_h = boardH
        self.screen_height = screen_height
        self.prev_pos = np.array([0, 0, 2])
        self.A, self.B, self.C = self.convert_keypoints(A, B, C)
        self.AB, self.BC, self.CA = norm(self.B - self.A), norm(self.C - self.B), norm(self.C - self.A)
        X = np.array([norm(self.prev_pos - self.A), norm(self.prev_pos - self.B), norm(self.prev_pos - self.C)])
        self.angles[0] = np.degrees(np.arccos((np.square(X[0]) + np.square(X[1]) -
                                               np.square(self.AB))/(2*X[0]*X[1])))
        self.angles[1] = np.degrees(np.arccos((np.square(X[0]) + np.square(X[2]) -
                                               np.square(self.CA))/(2*X[0]*X[2])))
        self.angles[2] = np.degrees(np.arccos((np.square(X[1]) + np.square(X[2]) -
                                               np.square(self.BC))/(2*X[1]*X[2])))
        self.Ao, self.Bo, self.Co = self.A, self.B, self.C

        LA = self.A - self.prev_pos
        LB = self.B - self.prev_pos
        LC = self.C - self.prev_pos

        self.orient_mat = np.matrix([[LA[0], LA[1], LA[2], 0,     0,     0,     0,     0,     0],
             [0,     0,     0,     LA[0], LA[1], LA[2], 0,     0,     0],
             [0,     0,     0,     0,     0,     0,     LA[0], LA[1], LA[2]],
             [LB[0], LB[1], LB[2], 0,     0,     0,     0,     0,     0],
             [0,     0,     0,     LB[0], LB[1], LB[2], 0,     0,     0],
             [0,     0,     0,     0,     0,     0,     LB[0], LB[1], LB[2]],
             [LC[0], LC[1], LC[2], 0,     0,     0,     0,     0,     0],
             [0,     0,     0,     LC[0], LC[1], LC[2], 0,     0,     0],
             [0,     0,     0,     0,     0,     0,     LC[0], LC[1], LC[2]]])

        print self.angles

```

```python
    # Solves the Resection problem for lengths
    def length_f(self, X):

        f = np.zeros(3)
        f[0] = 2*X[0]*X[1]*np.cos(np.radians(self.angles[0])) - np.square(X[0]) - np.square(X[1]) + np.square(self.AB)
        f[1] = 2*X[0]*X[2]*np.cos(np.radians(self.angles[1])) - np.square(X[0]) - np.square(X[2]) + np.square(self.CA)
        f[2] = 2*X[1]*X[2]*np.cos(np.radians(self.angles[2])) - np.square(X[1]) - np.square(X[2]) + np.square(self.BC)
        return f

    # Solves the Directions Vectors of each laser
    def vec_f(self, X):
        f = np.zeros(9)

        X1norm = norm(X[0:3])
        X2norm = norm(X[3:6])
        X3norm = norm(X[6:9])
        f[0] = -(self.L[0]*(X[0]/X1norm) - self.L[1]*(X[3]/X2norm)) + (self.A[0] - self.B[0])
        f[1] = -(self.L[0]*(X[1]/X1norm) - self.L[1]*(X[4]/X2norm)) + (self.A[1] - self.B[1])
        f[2] = -(self.L[0]*(X[2]/X1norm) - self.L[1]*(X[5]/X2norm))
        f[3] = -(self.L[0]*(X[0]/X1norm) - self.L[2]*(X[6]/X3norm)) + (self.A[0] - self.C[0])
        f[4] = -(self.L[0]*(X[1]/X1norm) - self.L[2]*(X[7]/X3norm)) + (self.A[1] - self.C[1])
        f[5] = -(self.L[0]*(X[2]/X1norm) - self.L[2]*(X[8]/X3norm))
        f[6] = -(self.L[2]*(X[6]/X3norm) - self.L[1]*(X[3]/X2norm)) + (self.C[0] - self.B[0])
        f[7] = -(self.L[2]*(X[7]/X3norm) - self.L[1]*(X[4]/X2norm)) + (self.C[1] - self.B[1])
        f[8] = -(self.L[2]*(X[8]/X3norm) - self.L[1]*(X[5]/X2norm))

        return f

    # The main function, solves for position given three lasers
    def getPos(self, A, B, C):
        order = []
        self.A, self.B, self.C = self.convert_keypoints(A, B, C)

        # Estimate A,B,C by minimizing the distance
        D1 = norm(self.Ao-self.A) + norm(self.Bo-self.B) + norm(self.Co-self.C)
        D2 = norm(self.Ao-self.A) + norm(self.Bo-self.C) + norm(self.Co-self.B)
        D3 = norm(self.Ao-self.B) + norm(self.Bo-self.A) + norm(self.Co-self.C)
        D4 = norm(self.Ao-self.B) + norm(self.Bo-self.C) + norm(self.Co-self.A)
        D5 = norm(self.Ao-self.C) + norm(self.Bo-self.A) + norm(self.Co-self.B)
        D6 = norm(self.Ao-self.C) + norm(self.Bo-self.B) + norm(self.Co-self.A)
        Darray = D1, D2, D3, D4, D5, D6
        Dindex = np.argmin(Darray)
        if Dindex == 0:
            self.A, self.B, self.C = self.A, self.B, self.C
            order = [0, 1, 2]
        elif Dindex == 1:
            self.A, self.B, self.C = self.A, self.C, self.B
            order = [0, 2, 1]
        elif Dindex == 2:
            self.A, self.B, self.C = self.B, self.A, self.C
            order = [1, 0, 2]
        elif Dindex == 3:
            self.A, self.B, self.C = self.B, self.C, self.A
            order = [1, 2, 0]
        elif Dindex == 4:
            self.A, self.B, self.C = self.C, self.A, self.B
            order = [2, 0, 1]
        elif Dindex == 5:
            self.A, self.B, self.C = self.C, self.B, self.A
            order = [2, 1, 0]

        self.Ao, self.Bo, self.Co = self.A, self.B, self.C

        # Form Estimation of Laser Lengths
        L0 = np.array([norm(self.A - self.prev_pos), norm(self.B - self.prev_pos), norm(self.C - self.prev_pos)])

        # Form Estimation of Laser Directional Vectors
        V0 = np.array([(self.A - self.prev_pos)/L0[0], (self.B - self.prev_pos)/L0[1], (self.C - self.prev_pos)/L0[2]])
```

```python

        # Get Distance Between Points
        self.AB, self.BC, self.CA = norm(self.B - self.A), norm(self.C - self.B), norm(self.C - self.A)

        # Solve Resection for Length of Lasers
        self.L = fsolve(self.length_f, L0)

        # Get Direction Vectors of Lasers
        V = fsolve(self.vec_f, V0)

        # Ensure Directions are Normalized
        V[0:3] = V[0:3]/norm(V[0:3])
        V[3:6] = V[3:6]/norm(V[3:6])
        V[6:9] = V[6:9]/norm(V[6:9])

        # Estimated Lasers
        self.VA = V[0:3]*self.L[0]
        self.VB = V[3:6]*self.L[1]
        self.VC = V[6:9]*self.L[2]

        # Average Position Estimation
        X = ((self.A - self.VA) + (self.B - self.VB) + (self.C - self.VC))/3

        # outputs
        self.prev_pos = X
        return X, order

    def getRPY(self):
        DV = np.array([self.VA*(1/norm(self.VA)), self.VB*(1/norm(self.VB)), self.VC*(1/norm(self.VC))]).flatten()
        r_mat = np.dot(np.linalg.pinv(self.orient_mat), DV).reshape(3,3)

        yaw1 = np.arctan2(r_mat[1, 0], r_mat[0, 0])
        pitch1 = np.arctan2(-r_mat[2, 0], np.sqrt(r_mat[2,1]*r_mat[2, 1] + r_mat[2, 2]*r_mat[2, 2]))
        pitch2 = np.arctan2(r_mat[2, 0], np.sqrt(r_mat[2, 1]*r_mat[2, 1] + r_mat[2, 2]*r_mat[2, 2]))
        roll1 = -np.arctan2(r_mat[2, 1], r_mat[2, 2])
        roll2 = np.arctan2(r_mat[2, 1], r_mat[2, 2])

        self.roll = roll1*180/np.pi
        self.pitch = pitch1*180/np.pi
        self.yaw = yaw1*180/np.pi

        return self.roll, self.pitch, self.yaw

    def convert_keypoints(self, A, B, C):
        A = np.array([A[0], self.board_h - A[1], 0])*(self.screen_height/self.board_h)
        B = np.array([B[0], self.board_h - B[1], 0])*(self.screen_height/self.board_h)
        C = np.array([C[0], self.board_h - C[1], 0])*(self.screen_height/self.board_h)
        return A,B,C
if __name__ == '__main__':
    a = LaserPosEstimator()
```