TRABALHO DE GRADUAÇÃO

INTERFACE HOMEM-MÁQUINA PARA AUTOMAÇÃO PREDIAL BASEADA EM GESTOS

Arthur Jaber Costato

Brasília, 11 de Julho de 2019

ENGENHARIA MECATRÔNICA
UNIVERSIDADE DE BRASÍLIA

UNIVERSIDADE DE BRASILIA
Faculdade de Tecnologia
Curso de Graduação em Engenharia de Controle e Automação

TRABALHO DE GRADUAÇÃO

INTERFACE HOMEM-MÁQUINA PARA AUTOMAÇÃO PREDIAL BASEADA EM GESTOS

Arthur Jaber Costato

Relatório submetido como requisito parcial para obtenção de grau de Engenheiro de Controle e Automação.

Banca Examinadora

Adolfo Bauchspiess, ENE/UnB _____
Orientador

Flávio de Barros Vidal, CIC/UnB _____
Examinador

Bruno Luiggi Macchiavello Espinoza, CIC/ UnB _____
Examinador

Brasília, 11 de Julho de 2019

FICHA CATALOGRÁFICA

COSTATO, ARTHUR JABER.

Interface Homem-Máquina para Automação Predial Baseada em Gestos

[Distrito Federal] 2019.

vii, 49, 297mm (FT/UnB, Engenheiro, Controle e Automação, 2019). Trabalho de Graduação – Universidade de Brasília. Faculdade de Tecnologia.

1. Automação Residencial
2. Visão Computacional
3. Controle por gestos
4. Interface com o usuário intuitiva

I. Mecatrônica/FT/UnB

REFERÊNCIA BIBLIOGRÁFICA

COSTATO, ARTHUR JABER, (2019). Interface Homem-Máquina para Automação Predial Baseada em Gestos. Trabalho de Graduação em Engenharia de Controle e Automação, Publicação FT.TG — *n*◦ 09, Faculdade de Tecnologia, Universidade de Brasília, Brasília, DF, 49p.

CESSÃO DE DIREITOS

AUTOR: Arthur Jaber Costato.

TÍTULO: Interface Homem-Máquina para Automação Predial Baseada em Gestos.

GRAU: Engenheiro de Controle e Automação ANO: 2019

Arthur Jaber Costato.

Universidade de Brasília, Faculdade de Tecnologia

Departamento de Engenharia Elétrica

70297-400 – DF – Brasil.

Resumo

Em automação predial, já existem algumas interfaces inteligentes com o usuário. No entanto, interfaces amigáveis e intuitivas de serem utilizadas ainda estão em desenvolvimento. A mais difundida atualmente é a de comandos por voz. Embora ela seja intuitiva, ainda não é muito amigável, pois não entende comandos muito complexos. Portanto, este trabalho buscou-se encontrar uma nova interface com o usuário que fosse mais intuitiva e fácil de se utilizar, principalmente pela população mais idosa ou com problemas cognitivos. Assim, percebeu-se que o controle por gestos de apontamento é uma interface que atende a estes requisitos. Devido a esforços computacionais envolvidas e confiabilidade das heurísticas necessárias para detectar a direção em que as mãos de uma pessoa estão apontando, decidiu-se confeccionar uma vareta com padrões visuais melhores estruturados para ser detectada. Com ela, o usuário poderá apontar para o objeto que desejará comandar. Um botão é utilizado para acionar. Neste trabalho, desenvolveu-se um algoritmo rápido capaz de identificar tal vareta em diferentes condições de iluminação. Utilizou-se a transformação para o espaço de cores CIE LAB para melhor extração das cores da vareta. Também foram utilizadas duas câmeras sem utilizar reconstrução 3D, através de uma abordagem que correlaciona a projeção da vareta nas duas imagens. Por fim, foram testados diferentes cenários e obteve-se uma média de 1,6 tentativas para encontrar a vareta nas imagens e 1,3 tentativas para encontrar o objeto apontado por ela.

Abstract

Building automation already uses many smart interfaces. However, more user-friendly and intuitive interfaces are still needed. Voice commands are widespread. Although intuitive, it is still not very friendly, as it does not understand complex commands. Therefore, this work looks for a new user interface that is more intuitive and easy to use, especially for the elder population. Control by gestures meets these intuitive, user-friendly requirement. Due to the computational complexities involved and the reliability of the heuristics needed to detect the direction a person's hands is pointing, it was decided to make a rod with structured visual pattern to be detected. Using it, the user can point to the equipment he wants to command. A button is used as action-trigger. A fast algorithm, capable of identifying such a rod in different lighting conditions was developed. CIE LAB color space was used for better color extraction of the rod pattern. In this work two cameras where used without using 3D reconstruction, an approach that matches the projected rod in each image. Finally, it was tested in different scenarios giving an average rod detection of 1.6 trials and an average object detection of 1.3 trials.

Sumário

Lista de Figuras

Lista de Símbolos

3D	Três dimensões;
2D	Duas dimensões;
CAD	Projeto / desenho auxiliado por comutador;
CMOS	Complementary Metal Oxide Semiconductor;
CMYK	Espaço de cores composto por quatro canais: ciano, magenta, amarelo e preto;
CIE	Comissão Internacional em Iluminação, define alguns espaços de cores, como o CIE 1931 XYZ, CIE LAB e o CIE LUV;
HSV	Espaço de cores composto por três canais: tonalidade, saturação e valor;
LARA	Laboratório de Automação e Robótica da Universidade de Brasília;
Matlab	Matrix Laboratory – Software que realiza de forma ótima contas matriciais;
OpenCV	Open Source Computer Vision Library – Biblioteca Aberta de Visão Computacional;
RGB	Espaço de cores composto por três canais: vermelho, verde e azul;
YCrCb	Espaço de cores composto por três canais: iluminação, componente vermelha e componente azul.

Lista de Variáveis

{x, y, z}	Coordenadas globais arbitrárias em unidades de comprimento
$\{x_f, y_f, z_f\}$	Coordenadas no foco da câmera em unidades de comprimento
f	Distância focal da câmera em unidades de comprimento
$\{u, v\}$	Coordenadas na placa CMOS da câmera em unidades de comprimento
$\{i, j\}$	Coordenadas da imagem da câmera em unidades de pixel.
$\{\delta u, \delta v\}$	Comprimento de um píxel na placa CMOS da câmera nas direções $\{u, v\}$
mr_{ij}	Valores dos campos da matriz de rotação das coordenadas globais para coordenadas do foco
$\{Lx_f, Ly_f, Lz_f\}$	Vetor de deslocamento de coordenadas globais para coordenadas do foco
MI_{ij}	Valores dos campos da matriz intrínsecas
$\{\theta_x, \theta_y\}$	Campo de visão da câmera em radianos
$M_{a,b}$	Momento de um contorno na imagem
$\{\bar{i}, \bar{j}\}$	Centro geométrico de um contorno na imagem

Capítulo 1

Introdução

1.1 Motivação

A partir dos anos cinquenta, com o rápido avanço computacional, áreas de automação começaram a ganhar muita força. Em torno dos anos oitenta, começaram a surgir as primeiras implementações de domótica, em controles básicos de climatização, iluminação e segurança em ambientes residenciais [1]. Atualmente, devido a grande acessibilidade das pessoas a dispositivos móveis (como telefones celulares e *tablets*) com grande poder de processamento e com a evolução das redes de internet, a automação residencial vem ganhando força no mercado. No momento, a forma mais comum e mais difundida de automação residencial está no uso de dispositivos móveis para controlar remotamente aparelhos eletroeletrônicos[2]. No entanto, este paradigma está mudando de forma muito rápida.

Segundo [2], existem três gerações de automação residencial: A primeira e maior difundida é a mencionada anteriormente: por controle sem fio via aplicativos em dispositivos móveis. A segunda geração que vem crescendo é a de controle por inteligências artificiais, onde a mais difundida atualmente é a de comandos básicos por voz, como o Alexa da Amazon [3] e o Google Assistente [4]. Já a terceira geração é parecida com a segunda, no entanto utiliza robôs humanoides para interagir com pessoas por meio de áudio e vídeo, como é o caso do Zenbo [5] e do Rovio [6]. Complementando [2], também pode-se acrescentar na segunda geração a automação por gestos [7] [8]. Esta automação ainda encontra-se em pesquisa e algumas aplicações já foram lançadas no mercado, como é o caso do controle de algumas funcionalidades internas de um carro BMW [9] ou para controle de alguns modelos de *drones* [10][11]. Estes controles por gestos, no entanto, são muito rudimentares. No caso de [7], o controle é de um menu em uma tela onde o controle se dá por posições das mãos de uma pessoa posicionada bem em frente a uma câmera, onde cada posição possui um significado específico (preparar / selecionar / voltar / mover menu). Já no caso de [8] o controle é mais direcionado diretamente ao controle do ambiente, no entanto também são movimentos das mãos com significados pré-determinados. O mesmo vale para os controles por gestos do BMW e dos *drones*, todos são gestos ou movimentos com significados pré determinados.

A principal motivação para este trabalho foi a de se criar uma interface com o usuário que fosse intuitiva e fácil de se utilizar. Até o momento, existem dificuldades na operação destes sistemas. Para os da primeira geração, a dificuldade se encontra na necessidade de se realizar uma sequência de procedimentos antes de efetuar o controle, como: desbloquear o dispositivo, acessar o aplicativo, autenticar o usuário e só então controlar. Para algumas pessoas isso é uma grande dificuldade, especialmente para boa parte da população idosa e de pessoas que sofrem de problemas cognitivos. Os dispositivos da segunda e terceira gerações por comando de voz ainda não compreendem frases complexas mas estão se aproximando de uma interface intuitiva e fácil de se utilizar. Já os dispositivos de controle por gestos ainda têm muito o que avançar, pois assim como os da primeira geração, eles não são fáceis de se utilizar por pessoas com deficiências cognitivas, visto que é necessário aprender quais gestos e movimentos executar para cada controle.

Existem pesquisas na área de detecção do gesto de apontamento. Ou seja, pesquisas para identificar que uma pessoa está apontando com as mãos e para qual objeto foi apontado. A pesquisa [12] utiliza várias câmeras estéreo, câmeras omnidirecionais, robô com sensor de distância e *tags* de rádio frequência para coletar a maior quantidade de dados possível para criar um ambiente ubíquo. Já a pesquisa [13] utiliza a tecnologia *Kinect* para realizar uma reconstrução 3D e identificar o gesto de apontamento. Para ambas as pesquisas, no entanto, não foram diretamente aplicadas em automação residencial.

Para este trabalho, buscou-se em realizar uma interface com o usuário através de automação por gesto de apontamento. No entanto, devido ao esforço computacional e da confiabilidade dos algoritmos heurísticos de identificação das mãos de uma pessoa, decidiu-se por dar um passo para trás e construir uma vareta com padrões visuais bem estruturados. Esta vareta, então, será utilizada para apontar, no lugar das mãos. Deste modo, uma pessoa utilizará esta vareta para apontar para o objeto que se deseja acionar (ligar ou desligar).

1.2 Objetivo Geral

Desenvolver uma interface homem-máquina para realizar uma automação predial com o auxílio de uma vareta para apontar para dispositivos eletroeletrônicos a serem comandados.

1.3 Objetivos Específicos

- Selecionar o *hardware* para atingir o objetivo genérico a partir de uma pesquisa do estado da arte.
- Criar um objeto apontador com padrões visuais que sejam: computacionalmente fáceis de se identificar, fácil de uma pessoa operar e que seja de iluminação passiva.
- Criar um programa computacional capaz de identificar a vareta confeccionada em imagens de vídeo em um ambiente com variações de intensidade luminosa.

1.4 Apresentação do Manuscrito

Este trabalho possui cinco capítulos e dois anexos dos códigos criados. O primeiro e atual Capítulo, tratou o que motivou o desenvolvimento deste trabalho, o que busca-se criar, os objetivos que se deseja alcançar e o que se espera encontrar nos próximos capítulos.

O segundo Capítulo tratará da teoria que fundamentará este trabalho explicando de forma concisa algumas equações e algoritmos utilizados ao longo deste projeto. Começará tratando rapidamente sobre as equações de câmera, as principais distorções que podem existir e principalmente como utilizar estas informações para calcular o campo de visão de uma câmera. Depois serão tratados os algoritmos básicos utilizados: operações lógicas entre máscaras, extração de cores, dilatação e erosão, detecção de contornos, momento de imagem e centro geométrico, subtração de fundo e rastreamento. Por fim, se explica o que são espaços de cores e os principais utilizados neste trabalho (HSV e CIE LAB).

O terceiro Capítulo tratará sobre os principais materiais utilizados, bem como a metodologia utilizada para desenvolver este projeto. Ele começa explicando como se detecta um objeto com as imagens de duas câmeras sem realizar o procedimento de reconstrução 3D e como o posicionamento das câmeras influenciam nesta detecção. Além disso, ele detalha o padrão visual utilizado pela vareta e como deixar o algoritmo robusto contra falsos positivos. Em seguida ele trata de como juntou-se os algoritmos discutidos até então para formar o código final. Por fim, este Capítulo trata de como fixou-se as câmeras no ambiente, como mapeou os alvos e quais foram os cenários de teste.

O quarto Capítulo trata dos resultados obtidos. Ele começa tratando de como foi construído a vareta com o padrão visual discutido no Capítulo anterior e como ela foi modificada para se tornar detectável sobre condições de variação de iluminação utilizando o espaço de cores CIE LAB. Em seguida ele trata de como se utilizou as equações de câmera (matriz intrínseca), obtidas de um procedimento de calibração, para calcular o campo de visão das câmeras e com isso como as posicionou no ambiente. Depois ele expõe como foram mapeados os alvos em coordenadas de câmera e o resultado obtido. Por fim, discute-se os resultados qualitativos e quantitativo dos cenários de teste do sistema.

Capítulo 2

Fundamentação Teórica

Neste Capítulo será descrito na seção 2.1 como se dá a projeção da luz emitida por objetos tridimensionais em coordenadas bidimensionais de imagem por meio de matriz intrínseca e extrínseca. Brevemente será tratado as principais distorções que podem ocorrer nas imagens, que para este trabalho as evitou durante a seleção das câmeras. Por fim, nesta seção será descrito como encontrar os ângulos do campo de visão das câmeras baseado na matriz intrínseca delas.

Na seção 2.2, serão tratados de algumas operações básicas que foram utilizadas no método de detecção da vareta. Já a seção 2.3 tratará sobre espaços de cores, pois será o fator que influenciará no tratamento de influências da variação de iluminação do ambiente.

2.1 Equações de câmera

Antes de introduzir o funcionamento do sistema, é importante se ter o entendimento matemático de como as câmeras funcionam. Inicialmente, será descrito como câmeras escuras de orifício se comportam. Em seguida será detalhado como se converte de coordenadas 3D para coordenadas 2D de câmera, introduzindo o conceito de matrizes intrínsecas e extrínsecas de câmera. Depois, brevemente serão apresentados o principal tipo de distorção que pode ocorrer e como tratá-la. Por fim se utilizará destes conhecimentos para definir como encontrar os ângulos de campo de visão da câmera.

2.1.1 Câmera Escura de Orifício

A Figura 2.1 ilustra como uma câmera escura de orifício se comporta. Nela, os feixes de luz emitidos por objetos externos só conseguem transpor um único orifício de entrada. Assim, forma-se uma imagem projetada dos objetos no fundo da câmera. Esta imagem é a projeção bidimensional dos objetos tridimensionais mais próximos da câmera. No entanto, esta projeção está rotacionada em 180º com relação ao eixo normal ao orifício de entrada da luz (“de cabeça para baixo”).

No caso de uma câmera fotográfica, utiliza-se um circuito sensor com tecnologia CMOS (Complementary Metal Oxide Semiconductor) para captar os feixes luminosos (como se fosse o fundo da câmera escura de orifício). Como o ser humano vê mais tons de verde do que azul ou vermelho, estes CMOS possuem mais receptores da coloração verde do que das outras duas. A Figura 2.2 ilustra as duas principais configurações dos receptores.

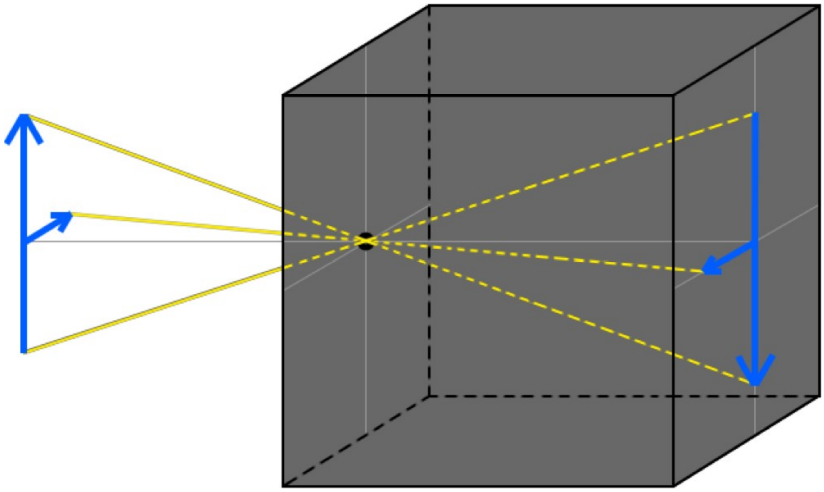

Figura 2.1 – Representação de uma Câmera Escura de Orifício.

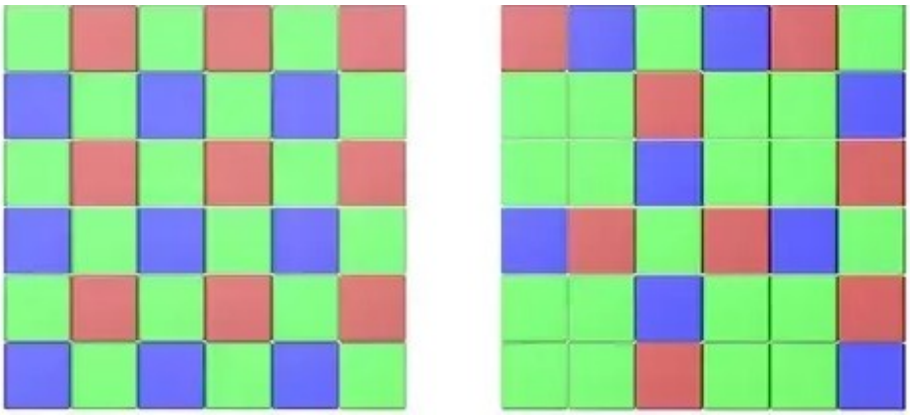

Figura 2.2 – Padrões do CMOS das Câmeras. Padrão Bayer Tradicional (da esquerda) e Padrão X-trans (da direita) [14].

2.1.2 Matriz Intrínseca.

A Figura 2.3 ilustra os principais eixos utilizados para se analisar as equações das câmeras. No fundo, apresentado com um padrão xadrez, representa a placa CMOS da câmera. As linhas azuis que saem de cada receptor (cada casa do padrão xadrez) representam o caminho que a luz percorre para atingir o receptor. Como é de se esperar, estas linhas passam pelo ponto focal (equivalente ao orifício de entrada dos feixes de luz).

O primeiro eixo de coordenadas que se determina é o $\{x_f, y_f, z_f\}$. Sua origem é localizada exatamente no ponto focal. O eixo z_f é normal a placa CMOS, enquanto os eixos x_f e y_f são paralelos às direções horizontal e vertical em que os receptores foram dispostos respectivamente. Onde o eixo z_f intercepta a placa CMOS tem-se o pixel central da imagem (i_0, j_0). Neste ponto, gera-se a origem de um novo eixo (no caso bidimensional) $\{u, v\}$. O eixo u é paralelo ao eixo x_f e com o sentido oposto. Já o eixo v é paralelo ao eixo y_f e com o sentido também oposto.

Por fim, o terceiro eixo importante está localizado no canto superior esquerdo da imagem, representado na Figura pelas 2.3 pelas coordenadas pretas $\{i, j\}$ que são respectivamente paralelas às coordenadas $\{u, v\}$. Diferentemente das outras coordenadas, as coordenadas $\{i, j\}$ são adimensionais (pois se tratam de unidade de pixeis) e não de comprimento.

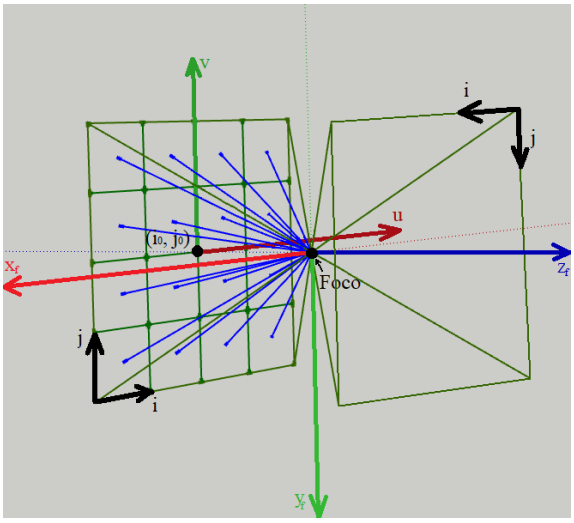

Figura 2.3 – Eixos importantes para análise das equações das câmeras.

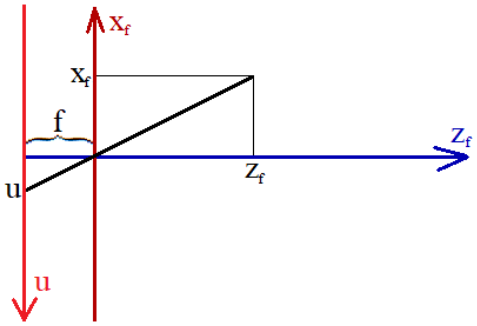

Figura 2.4 – Relação Trigonométrica entre ponto nas coordenadas x e z com a sua projeção na coordenada u.

A Figura 2.4 relaciona as coordenadas x_f e u com relação a distância focal f, distância do foco à placa CMOS. Por relações trigonométricas, pode-se determinar que a razão entre x_f e u é igual à razão entre z_f e f e portanto pode-se chegar na equação

$$u = \frac{f}{z_f} x_f \tag{2.1}$$

A mesma relação pode ser dada de forma análoga para as coordenadas v e y_f, obtendo-se a equação

$$v = \frac{f}{z_f} y_f \tag{2.2}$$

Para se chegar nas coordenadas i e j (coordenadas da matriz referentes à imagem), a partir de u e v, deve-se primeiramente converter de unidades de comprimento para unidades adimensionais de pixel. Utiliza-se as constantes de proporcionalidade δu e δv (comprimento e largura de um fotorreceptor da placa CMOS correspondentes a um pixel) para relacionar i e j com as direções u e v, respectivamente. A origem do eixo $\{u, v\}$ está localizado nas coordenadas (i_0, j_0) e não na origem dos eixos $\{i, j\}$. Desta forma, além de utilizar as constantes de proporcionalidade, deve-se também deslocar as origens. Desta forma, pode-se chegar nas equações 2.3 e 2.4, que correlacionam estas outras duas coordenadas.

$$u = \delta u (i - i_0) \tag{2.3}$$

$$v = \delta v (j - j_0) \tag{2.4}$$

Utilizando-se as equações 2.1 e 2.3, pode-se obter uma relação direta entre i e x_f, como indicado pela equação

$$i = \frac{i_0 z_f + \frac{f}{\delta u} x_f}{z_f} \tag{2.5}$$

O mesmo pode ser feito com as equações 2.2 e 2.4 para obter a relação direta entre j e y_f, como indicado pela equação

$$j = \frac{j_0 z_f + \frac{f}{\delta v} y_f}{z_f} \tag{2.6}$$

Por meio das equações 2.5 e 2.6, pode-se chegar em uma relação matricial

$$\begin{bmatrix} U \\ V \\ S \end{bmatrix} = \begin{bmatrix} \frac{f}{\delta u} & 0 & i_0 \\ 0 & \frac{f}{\delta v} & j_0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_f \\ y_f \\ z_f \end{bmatrix}, \tag{2.7}$$

onde

$$i = \frac{U}{S} \text{ e } j = \frac{V}{S}.$$

Esta relação é chamada de Matriz Intrínseca das câmeras. Ela determina a relação entre as coordenadas de câmera (coordenadas bidimensionais {i, j}) com as coordenadas do foco da câmera (coordenadas tridimensionais $\{x_f, y_f, z_f\}$).

2.1.3 Matriz Extrínseca.

Dado coordenadas globais $\{x, y, z\}$ arbitrariamente definidas, pode-se correlacioná-la com as coordenadas do foco de uma câmera $\{x_f, y_f, z_f\}$ e em seguida com a Matriz Intrínseca da Câmera, pode-se correlacionar com as coordenadas de câmera $\{i_0, j_0\}$. O cálculo que correlaciona as coordenadas globais com as coordenadas do foco é dado pela Matriz Extrínseca da câmera. Ela é gerada por meio da posição e orientação da câmera com relação as coordenadas globais. A relação entre estas duas bases é correlacionada por uma matriz de rotação (matriz que rotaciona as coordenadas $\{x, y$ e $z\}$ para ficarem na mesma orientação de $\{x_f, y_f$ e $z_f\}$) e um vetor de deslocamento (o quão afastado no sentido estão as origens das duas bases, descrita em termos de x_f, y_f e z_f), como indicado pela equação

$$\begin{bmatrix} x_f \\ y_f \\ z_f \end{bmatrix} = \begin{bmatrix} m_{r_{11}} & m_{r_{12}} & m_{r_{13}} \\ m_{r_{21}} & m_{r_{22}} & m_{r_{23}} \\ m_{r_{31}} & m_{r_{32}} & m_{r_{33}} \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} + \begin{bmatrix} L_{x_f} \\ L_{y_f} \\ L_{z_f} \end{bmatrix}$$

Esta equação pode ser reescrita como o produto de apenas uma matriz 3x4 e um vetor 4x1, como indicado pela equação

$$\begin{bmatrix} x_f \\ y_f \\ z_f \end{bmatrix} = \begin{bmatrix} m_{r_{11}} & m_{r_{12}} & m_{r_{13}} & L_{x_f} \\ m_{r_{21}} & m_{r_{22}} & m_{r_{23}} & L_{y_f} \\ m_{r_{31}} & m_{r_{32}} & m_{r_{33}} & L_{z_f} \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}.$$

Sendo esta a equação da Matriz Extrínseca da câmera.

2.1.4 Distorção Radial e Tangencial de Câmera

O modelo até agora proposto considerou as câmeras tradicionais com a aproximação pelas Câmeras Escuras de Orifício. No entanto, existem distorções que podem fazer com que este modelo não seja muito preciso. A principal distorção é denominada de Distorção Radial. Felizmente, há um método muito bem estabelecido no tratamento desta distorção. As Figuras 2.5 e 2.6 apresentam dois tipos de distorções radiais comuns em câmeras.

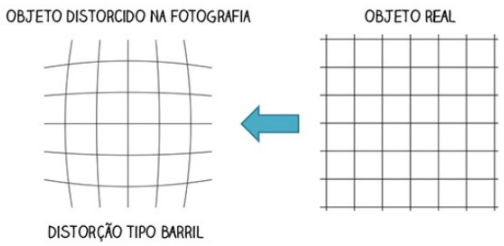

Figura 2.5 – Distorção de imagem do tipo barril [15].

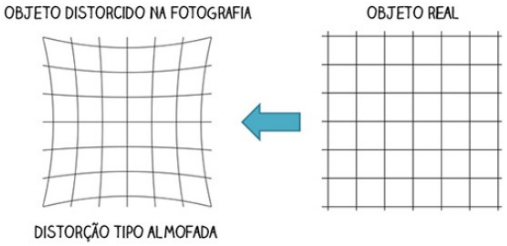

Figura 2.6 – Distorção de imagem do tipo almofada [15].

$$
\begin{aligned}
x_u =& x_d + (x_d - x_c)(K_1 r^2 + K_2 r^4 + \ldots) + \\
& \left(P_1(r^2 + 2(x_d - x_c)^2) + 2P_2(x_d - x_c)(y_d - y_c)\right)(1 + P_3 r^2 + \ldots) \\
y_u =& y_d + (y_d - y_c)(K_1 r^2 + K_2 r^4 + \ldots) + \\
& \left(2P_1(x_d - x_c)(y_d - y_c) + P_2(r^2 + 2(y_d - y_c)^2)\right)(1 + P_3 r^2 + \ldots)
\end{aligned}
$$

where:

(x_u, y_u) = undistorted image point,

(x_d, y_d) = distorted image point,

(x_c, y_c) = centre of distortion,

$K_n = \mathrm{N}^{\mathrm{th}}$ radial distortion coefficient,

$P_n = \mathrm{N}^{\mathrm{th}}$ tangential distortion coefficient,

$r = \sqrt{(x_d - x_c)^2 + (y_d - y_c)^2}$, and

$\ldots$ = an infinite series.

Figura 2.7 – Correção da distorção da câmera [16].

O modelo de Brown–Conrady [16] é apresentado pelas equações da Figura 2.7 e determinam como se faz para se reverter essas distorções. Para isso, é necessário obter os coeficientes de distorção radial e tangencial da imagem. Esses coeficientes podem ser obtidos por um programa de calibração de câmeras. Por exemplo, neste trabalho utilizou-se a biblioteca OpenCV versão 3.4.1 e em seu diretório de instalação na pasta: *\opencv\sources\samples\cpp* existem vários exemplos de como utilizá-la bem como códigos úteis como o: *calibration.cpp* que calcula as matrizes Intrínseca e Extrínseca bem como os coeficientes para reverter a distorção de uma imagem. Um outro código que também pode ser útil (só que não para este trabalho) é o 3*calibration.cpp* que realiza a calibração 3D para câmeras estéreo.

2.1.5 Cálculo do campo de visão da câmera

Obtendo-se a matriz intrínseca de uma câmera através de um programa de calibração, é possível calcular qual seu campo de visão. Pela Figura 2.4, pode-se notar que a razão entre u e f corresponde ao seno do ângulo do pixel na direção u (paralelo a coordenada i e x_f). Assim, para saber o ângulo do campo de visão da câmera, basta saber a posição (u, v) dos fotorreceptores dos pixeis mais extremos da imagem. Para câmeras de resolução 720p (resolução utilizada neste projeto) Um pixel mais extremo estará em $u = \delta u^*(1280 - i_0)$ e $v = \delta v^*(720 - j_0)$. Outro pixel mais extremo estará em $u = \delta u^*(0 - i_0)$ e $v = \delta v^*(0 - j_0)$. Também pode-se considerar o caso ideal em que $(i_0, j_0) = (640, 360)$ (exatamente no pixel central da imagem), neste caso os pixeis mais externos sempre estarão na posição $u = \delta u^*640$ e $v = \delta v^*360$.

Como busca-se o campo de visão da câmera, então precisa-se dividir *u* por *f* para encontrar o seno do ângulo do campo de visão. Repare que esta razão nas expressões descritas no parágrafo anterior resultará em, por exemplo, $u/f = (\delta u/f)*(1280 - i_0) = (1280 - MI_{13})/MI_{11}$, onde MI_{11} corresponde ao valor da matriz intrínseca (equação 2.7) da primeira linha e primeira coluna e MI_{13} corresponde ao valor da matriz intrínseca da primeira linha e terceira coluna. Assim, pode-se descobrir o campo de visão da câmera por meio das equações

$$\theta_x = asin\left(\frac{1280 - MI_{13}}{MI_{11}}\right) \qquad \theta_y = asin\left(\frac{720 - MI_{23}}{MI_{22}}\right)$$
$$\theta_x = asin\left(\frac{-MI_{13}}{MI_{11}}\right) \qquad \theta_y = asin\left(\frac{-MI_{23}}{MI_{22}}\right)$$
$$\theta_x = asin\left(\frac{640}{MI_{11}}\right) \qquad \theta_y = asin\left(\frac{360}{MI_{22}}\right).$$

2.2 Operações básicas em imagens

Ao longo do trabalho, testou-se vários algoritmos. Dentre eles, será descrito nesta seção alguns dos utilizados neste projeto: operações lógicas entre máscaras (máscaras são imagens com pixeis binários: pixel preto sendo o nível lógico '0' e branco sendo o nível lógico '1'), extração de cores por limiares (*threshold*) e geração de máscaras, dilatação e erosão, detecção de contornos, momento de imagem e centro geométrico. Além destes, há mais dois que foram utilizados mas foram removidos (motivos da remoção serão tratados no Capítulo 3): subtração de fundo e região de interesse. Outros algoritmos merecem mais detalhamento e serão tratados nas seções 2.3 (conversão entre espaços de cores), 2.4 (identificação do objeto apontado) e 2.6 (detecção da vareta).

2.2.1 Operações lógicas entre máscaras

Como mencionado anteriormente, máscaras são imagens com pixeis binários (preto sendo o nível lógico '0' e branco sendo o nível lógico '1'). Deste modo, operações lógicas podem ser utilizadas entre elas. Para este trabalho, foram utilizadas quatro operações: interseção (E), união (OU), inversão (NÃO) e deslocamento (*shift*).

A interseção e união entre duas máscaras e a inversão de uma máscara se dá pixel-a-pixel (*bitwise*). Ou seja, um pixel na posição (i, j) em uma máscara realizará esta operação com outro pixel na mesma posição (i, j) da outra máscara gerando um pixel na mesma posição (i, j) da máscara resultante. Para a operação de interseção, o valor do pixel gerado será '1' somente se os pixeis das duas máscaras forem também iguais a '1' e será '0' nos casos contrários. Já para a operação de união, o valor do pixel gerado será '0' somente se os pixeis das duas máscaras forem também iguais a '0' e será '1' nos casos contrários. Já a inversão de uma máscara faz com que os pixeis em nível lógico '0' tornem-se '1' e os pixeis em nível lógico '1' tornem-se '0'.

O deslocamento de uma máscara se dá pela cópia de alguns pixeis dela de uma dada posição para outra posição da máscara resultante. Neste projeto, realizou-se o deslocamento para esquerda e o deslocamento para cima. No deslocamento para esquerda, os pixeis da posição (i, j) são copiados para posição $(i–D, j)$, para $i \geq D$ e D um valor inteiro. A máscara resultante possui o mesmo tamanho da máscara original, desta forma, os pixeis mais à direita assumem o valor '0', pixeis entre $(I–D, j)$ e (I, j), onde I é o valor máximo na direção i da imagem. Já o deslocamento para cima é análogo ao deslocamento para esquerda, no entanto são copiados da posição (i, j) para a posição $(i, j–D)$, para $j \geq D$ e os pixeis mais em baixo assumem o valor '0', pixeis entre $(i, J–D)$ e (i, J).

2.2.2 Extração de cores por limiares (ou *threshold*) e geração de máscaras

Uma imagem é formada por pixeis dispostos em uma ou mais matrizes bidimensionais. A quantidade de matrizes dependerá da quantidade de **canais** que esta imagem possui. Por exemplo, imagens no padrão RGB possuem três canais: R (intensidade de vermelho), G (intensidade de verde) e B (intensidade de azul). Já, por exemplo, imagens em tons de cinza só possuem um canal, o tom de cinza. Independentemente do espaço de cores utilizado (e quantidade de canais dos mesmos), o algorítimo de extração por limiares se manterá o mesmo. Para mais informações a respeito de espaços de cores, referir-se à seção 2.3.

Para cada canal da imagem, é definido um intervalo de valores que deseja-se extrair (intervalo definido por duas variáveis: uma com o valor mínimo e outra com o valor máximo). Com este intervalo, gera-se máscaras onde os pixeis com valor dentro deste intervalo receberão o valor lógico '1' e os pixeis com valor fora deste intervalo receberão '0'. A máscara resultante é gerada pela união das máscaras de cada canal da imagem. Também é possível definir a variável com o valor máximo menor do que a com o valor mínimo. Neste caso, realiza-se o mesmo procedimento e em seguida realiza a operação de inversão (NÃO) na máscara desse canal.

2.2.3 Dilatação e Erosão

A erosão é análoga a erosão do solo, onde busca reduzir a espessura de imagens em uma máscara, como ilustra a Figura 2.10. Além disso, a erosão também é utilizada para remover ruídos em uma máscara. Segundo [17], a erosão se dá pelo deslocamento de um *kernel* (uma pequena máscara, ex:3x3) sobre a máscara que se deseja erodir. Um pixel em nível lógico '1' da máscara original só se manterá com este valor se a interseção entre o *kernel* e os pixeis da máscara abaixo dele coincidirem. Caso contrário, o pixel torna-se '0'. A Figura 2.8 ilustra com um exemplo a erosão.

A dilatação é o princípio oposto da erosão, como também demonstrado pela Figura 2.10. Quando o *kernel* coincide com apenas um píxel '1' da máscara original, cria-se novos pixeis na fronteira com a imagem, como ilustrado pela Figura 2.9. Comparando as Figuras 2.8 e 2.9, pode-se notar que a dilatação não é a função inversa da erosão, pois com o mesmo *kernel* e utilizando a máscara resultante da erosão para a dilatação, não obteve-se de volta a mesma máscara original utilizada na erosão. No entanto, pode-se notar que os ruídos que existiam na máscara original foram removidos. Esta é a principal utilização deste algoritmo neste projeto.

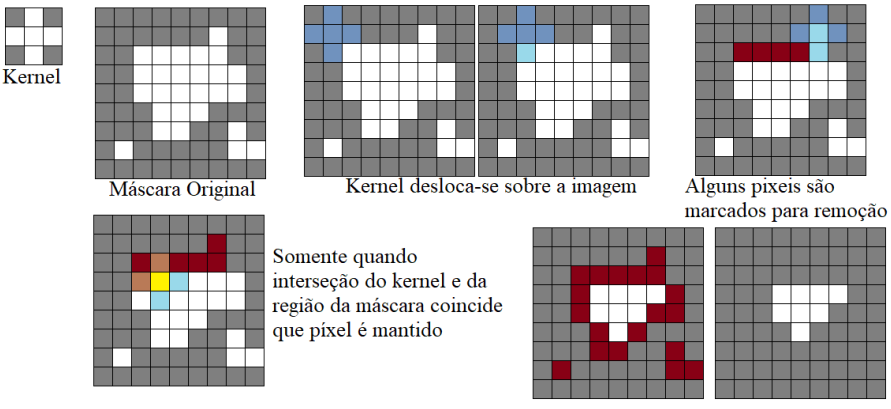

Figura 2.8 – Exemplo de erosão.

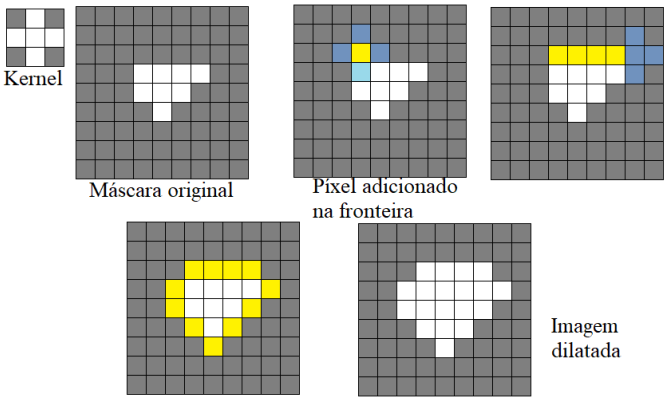

Figura 2.9 – Exemplo de dilatação.

Figura 2.10 – Exemplos de erosão e dilatação do OpenCV[17].

2.2.4 Detecção de Contornos

Existem alguns métodos de detecção de contornos de uma máscara, alguns mais rápidos do que outros. Para este projeto, buscou-se performance, e portanto utilizou o método "*Chain Approach Simple*". Neste método, não se armazena todos os pontos do contorno de uma imagem e sim somente pontos extremos de segmentos de reta verticais, horizontais e diagonais. Além disso, pode existir contornos que são do tipo externo e interno, onde os contornos internos estão dentro de outros contornos, como ilustrado pela Figura 2.11. Para este projeto só foi necessário buscar por contornos externos, o que reduz também muito o custo computacional.

2.2.5 Momento de imagem e centro de geométricos de contornos

O momento de imagem é equivalente ao momento de inércia de corpos físicos. No entanto, para uma objeto, o momento se dá por um somatório e não uma integral, por se tratar de coordenadas discretas dos pixeis. A equação

$$M_{a,b} = \sum_{a} \sum_{b} i^a j^b I(i,j)$$

indica como é calculado, onde *a* e *b* são expoentes que definem o tipo de momento e *I(i,j)* é equivalente a densidade em um cálculo de momento de inércia e é igual a 1 quando o pixel da posição (i, j) foi selecionado para o cálculo do momento e é igual a 0 caso contrário. Com o momento da imagem, calcula-se o centro geométrico dela, como indicado pelas equações

$$\bar{i} = \frac{M_{1,0}}{M_{0,0}} \quad , \quad \bar{j} = \frac{M_{0,1}}{M_{0,0}}$$

Neste projeto, utilizou-se estes cálculos para encontrar os centros geométricos de pontos retornados pela função de detecção de contornos externos. A Figura 2.12 apresenta um exemplo utilizando cálculo de centros geométricos de contornos.

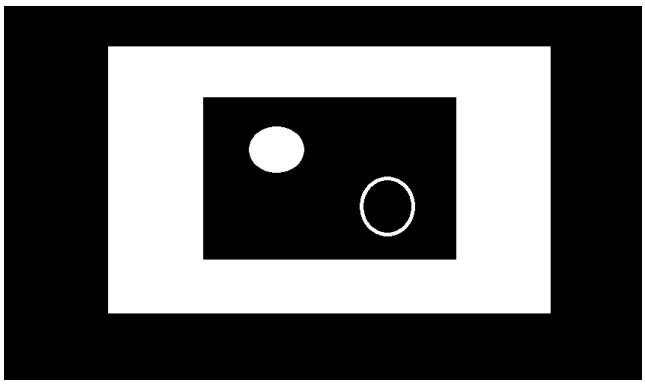

Figura 2.11 – Exemplo de uma figura com contornos internos.

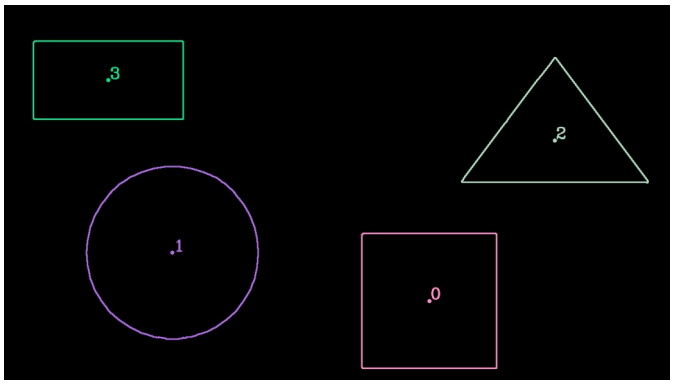

Figura 2.12 – Exemplo de centros geométricos de contornos [18]**.**

2.2.6 Subtração de Fundo

Por se tratar de vídeo, é comum se utilizar subtração de fundo para somente trabalhar com os objetos que se movem pela imagem. A subtração de fundo consiste na subtração da imagem de vídeo com uma variável que armazena a imagem do fundo dela. Com a subtração, cria-se uma máscara onde, para os pixeis cuja diferença seja muito pequena assumem o valor de '0' e para os pixeis cuja diferença seja muito grande assumem um valor de '1'. Em seguida pode-se aplicar erosão e dilatação para tratar esta máscara. Por fim, realiza-se a interseção dela com a imagem de vídeo original. A imagem de fundo armazenada pode ser invariante, mas quando se realizava subtração de fundo neste projeto, ela era estimada utilizando uma média móvel como indicado pela equação

```
fundoEstimado = 0.94*fundoEstimado + 0.06*imagemVídeo.
```

Além disso, não se utilizava todos os quadros do vídeo para estimá-lo e sim um a cada 10 quadros (com uma taxa de amostragem em torno de 30 quadros por segundo resultava em torno de três quadros por segundo). A Figura 2.13 indica a resposta ao degrau da média móvel. Por meio dela, pode-se notar que leva em torno de 20 segundos para se ter uma estimativa de fundo completamente nova.

2.2.7 Região de Interesse e rastreamento (*tracking*)

Em vez de se trabalhar com todos os pixeis da imagem, pode-se decidir por trabalhar com apenas aqueles em uma região da imagem de interesse. No caso deste projeto, os pixeis de interesse são aqueles que contém a vareta. Assim, foi feito um código que rastreava a vareta pela imagem formando uma caixa em seu entorno, tal caixa é a região de interesse da imagem. Deste modo, não precisava trabalhar em toda a imagem e sim somente com os pixeis desta região. O que fazia o custo computacional reduzir.

Existem vários métodos de se desenvolver uma região de interesse, como a *Template-Matching*[20] que compara a região de interesse em imagens em tons de cinza do quadro de vídeo anterior com várias possíveis regiões de interesse no novo quadro e então busca-se minimizar a distância quadrática entre elas para se obter a região com maior similaridade entre elas. Para este projeto, no entanto, este processo seria muito custoso. Então decidiu-se por definir o tamanho da região de interesse como sendo duas vezes o tamanho da vareta e com a vareta centrada nesta região de interesse. Quando este procedimento era realizado, ele era simples e rápido de ser feito, pois ao longo do algorítimo de detecção da vareta, obtém-se a posição dos pixeis extremos dela.

2.3 Espaço de cores

Os espaços de cores são espaços geralmente multidimensionais utilizados para descrever as cores seguindo um determinado critério. Por exemplo, o espaço de cores RGB possui três dimensões, cada uma indicando a intensidade das cores aditivas: vermelha, verde e azul. Contrapondo a este espaço, existe o espaço de cores CMYK que possui quatro coordenadas, cada uma indicando um percentual das cores subtrativas: ciano, magenta e amarelo. A quarta coordenada adicionada a este espaço é o da cor preto (equivalente a subtração das três cores). Também existem espaços de uma coordenada apenas, como é o caso do espaço de níveis de cinza, em que sua única dimensão representa a média ponderada das dimensões RGB (29,9% vermelho, 58,7% verde e 11,4% azul).

O espaço de cores mais conhecido é o espaço RGB. Este espaço é baseado nas células cones da visão humana. Estas células respondem como um filtro passa banda para uma determinada faixa de frequências luminosas e são classificadas em três tipos: *S*, *M*, *L* (de short, medium e large). As células *S* respondem para luzes mais azuladas de baixo comprimento de onda (alta frequência). Já as células *M* respondem para luzes mais esverdeadas de comprimentos de onda medianas. Enquanto as células *L* respondem para luzes mais avermelhadas de comprimentos de onda altos.

A Figura 2.14 apresenta a intensidade média humana das respostas das células cone em função do comprimento de onda. Ela foi obtida na criação do espaço de cores CIE 1931 XYZ medindo-se a resposta de vários participantes e variando o comprimento de onda entre 380 nm a 780 nm (em intervalos de 5 nm). Devido a distribuição das células cones na fóvea, este experimento se deu no arco de 2º do campo de visão. A Figura 2.15 apresenta a acuidade relativa da fóvea do olho esquerdo com relação ao ângulo em graus do campo de visão.

Figura 2.13 – Resposta ao degrau da média móvel da estimativa de fundo.

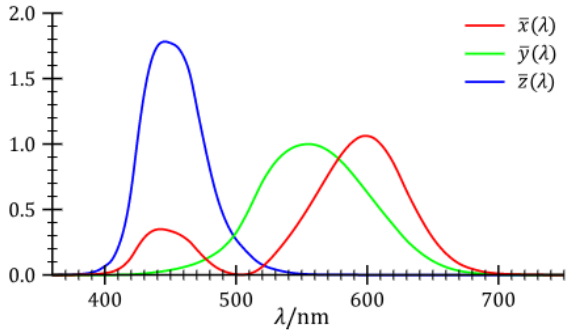

Figura 2.14 – Intensidade média das respostas das células cone [19].

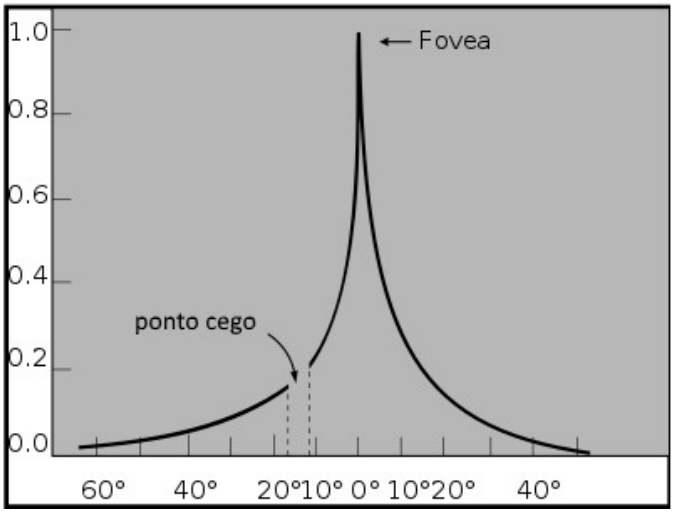

Figura 2.15 – Acuidade relativa da fóvea do olho esquerdo (seção horizontal) em graus [21].

Os televisores coloridos não são capazes de emitir todas as frequências luminosas. Então, de forma a tentar simular estes espectros, estes televisores emitem combinações de intensidade das três cores: vermelha (em torno de 630 nm), verde (em torno de 532 nm) e azul (em torno de 457 nm) [22]. Assim, do que é de interesse da visão computacional, as câmeras filmadoras realizam um processo similar ao das células cones de filtro passa banda para transformar uma determinada frequência lumiosa em um ponto no espaço de cores RGB. Atualmente, para que este ponto seja armazenado em tecnologias digitais, ele é quantizado em um byte para cada dimensão (8 bits binários por dimensão). Ou seja, cada dimensão assume um valor discreto entre 0 e 255 totalizando em 16.8 milhões de cores possíveis.

O espaço de cores RGB não é o mais ideal para se utilizar em extrair cores, pois uma vez que todas as cores são combinações das três cores primárias (vermelha, verde e azul), até mesmo as cores primárias podem possuir valores significativos nas outras dimensões, como ilustra a Figura 2.16. Assim, para extrair uma cor, se faz necessário utilizar uma transformação de espaço para um que possua coordenadas que representam os espectros das cores (geralmente denominada “hue” ou tonalidade das cores). Estes espaços geralmente são formados por círculo de cores em coordenadas polares, onde o ângulo representa o tom. Por exemplo, a Figura 2.17 ilustra o círculo de cores do espaço HSV. Pode-se reparar que cada tom encontra-se devidamente posicionado em uma seção circular.

De acordo com [23], estes círculos de cores estão definidos em quatro grupos: O primeiro é baseado nas cores aditivas, onde as cores vermelha, verde e azul estão aproximadamente igualmente espaçadas (como é o caso do espaço HSV). O segundo é baseado nas cores subtrativas, onde as cores estão dispostas de forma a manter aproximadamente

igualmente espaçadas as cores ciano, magenta, amarelo e preto (geralmente são espaços de cores utilizados em meios artísticos). O terceiro busca manter aproximadamente igualmente espaçadas as cores de tipo oposto: vermelho oposto ao verde e azul oposto ao amarelo (como é o caso do CIE LAB). Já o quarto procura manter aproximadamente igualmente espaçadas as cores de acordo com como elas são visualmente percebidas (como é o caso do CIE LUV).

Figura 2.16 – Cores primárias RGB com quantidades significativas dos outros tons.

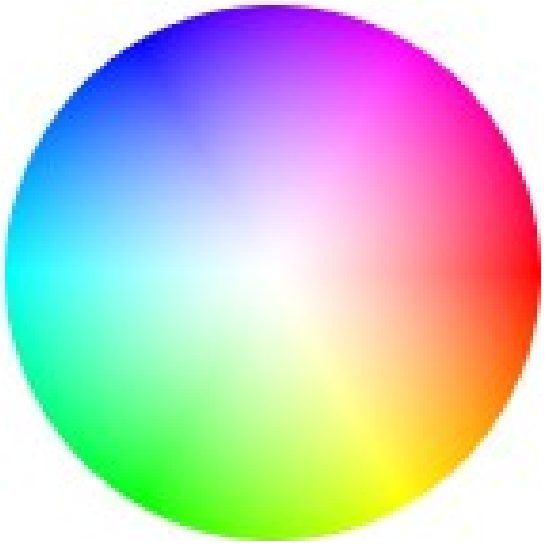

Figura 2.17 – Circulo de cores do espaço de cores HSV.

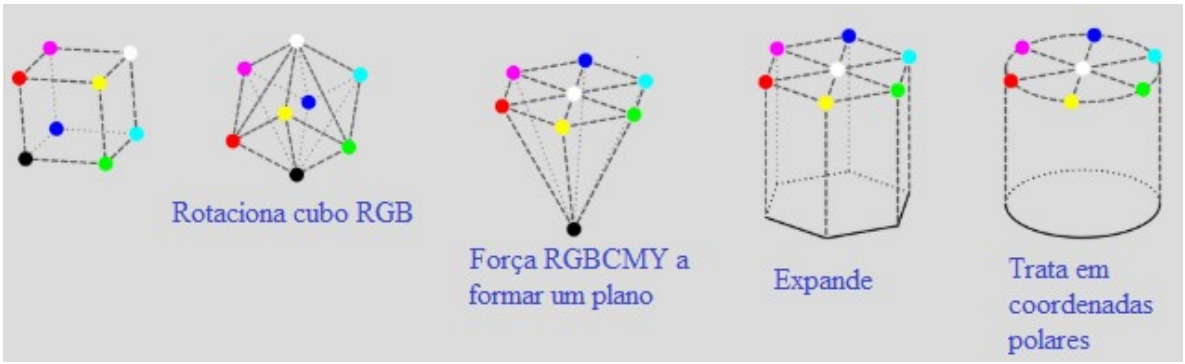

Figura 2.18 – Transformação do espaço RGB para o espaço HSV.

O primeiro espaço de cores que geralmente é escolhido para tratar de extração de cores é o espaço HSV, uma vez que ele gera um círculo de cores baseados nas cores RGB capturadas pela câmera aplicando uma transformação que é computacionalmente rápida (comparada com outras transformações de espaços de cores), como é ilustrado pela Figura 2.18. Os cálculos realizados pela biblioteca OpenCV [24] para realizar esta transformação são dados por

$$
\begin{aligned}
V &\leftarrow max(R,G,B) \\
S &\leftarrow \begin{cases} \frac{V-min(R,G,B)}{V} & \text{if } V \neq 0 \\ 0 & \text{otherwise} \end{cases} \\
H &\leftarrow \begin{cases} 60(G-B)/(V-min(R,G,B)) & \text{if } V = R \\ 120+60(B-R)/(V-min(R,G,B)) & \text{if } V = G \\ 240+60(R-G)/(V-min(R,G,B)) & \text{if } V = B \end{cases}
\end{aligned}
\tag{2.8}
$$

If $H < 0$ then $H \leftarrow H + 360$. On output $0 \leq V \leq 1, 0 \leq S \leq 1, 0 \leq H \leq 360$.

Vários espaços de cores foram testados ao longo deste projeto, mais detalhes serão tratados no Capítulo 3. Para este projeto selecionou-se o espaço de cores CIE LAB. Diferentemente do HSV, o CIE LAB permite que se trabalhe em um ambiente com variações de iluminação. Existe também um espaço chamado HSL, que é bem parecido com o HSV mas que também busca tratar de iluminação. No entanto, como será tratado no Capítulo 3, ele também não foi tão melhor que o HSV.

De acordo com [23], o CIE LAB também é formado por um círculo de cores do terceiro tipo (que mantém mais opostas as cores de tipo oposto: azul-amarelo, vermelho-verde). Uma propriedade importante dele é que ele é perceptivelmente linear. Ou seja, se aplicado uma variação no hue/tonalidade (ângulo) ou no croma (raio) em duas cores, uma pessoa perceberá a mesma "importância" da variação. Esta propriedade para visão computacional significa que pode-se extrair uma cor por meio de uma seção circular deste espaço (um intervalo para a tonalidade e um para o croma).

A desvantagem em se utilizar o CIE LAB no lugar do HSV ou HSL está em seu custo computacional. Onde, como indicado pelas equações 2.9 [24], existe uma função f(t) não linear (com um termo de raiz cúbica) que aumenta o custo computacional desta transformação se comparado por exemplo ao espaço HSV (das equações 2.8) que possui de menor custo.

$$
\begin{aligned}
\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} &\leftarrow \begin{bmatrix} 0.412453 & 0.357580 & 0.180423 \\ 0.212671 & 0.715160 & 0.072169 \\ 0.019334 & 0.119193 & 0.950227 \end{bmatrix} \cdot \begin{bmatrix} R \\ G \\ B \end{bmatrix} \\
X &\leftarrow X/X_n, \text{where} X_n = 0.950456 \\
Z &\leftarrow Z/Z_n, \text{where} Z_n = 1.088754 \\
L &\leftarrow \begin{cases} 116 * Y^{1/3} - 16 & \text{for } Y > 0.008856 \\ 903.3 * Y & \text{for } Y \leq 0.008856 \end{cases} \\
a &\leftarrow 500(f(X) - f(Y)) + 128 \\
b &\leftarrow 200(f(Y) - f(Z)) + 128 \\
L &\leftarrow L * 255/100,\ a \leftarrow a + 128,\ b \leftarrow b + 128
\end{aligned}
\tag{2.9}
$$

where

$$
f(t) = \begin{cases} t^{1/3} & \text{for } t > 0.008856 \\ 7.787t + 16/116 & \text{for } t \leq 0.008856 \end{cases}
$$

Capítulo 3

Materiais e Métodos

Neste Capítulo, será tratado o que foi desenvolvido neste trabalho. Na seção 3.1, será tratado de forma breve quais os materiais utilizados neste projeto. Já na seção 3.2 será tratado os métodos que foram executados. A seção 3.2.1 detalhará como se obteve a informação de qual objeto foi apontado. Já a seção 3.2.2 tratará de como posicionou-se as câmeras de forma a não precisar realizar o procedimento de reconstrução 3D. Na seção 3.2.3 descreve como se desenvolveu o padrão visual da vareta e como tornou o algoritmo robusto para evitar falsos positivos. Na seção 3.2.4 descreve como juntou os algoritmos descritos tanto nas seções de fundamentação teórica como os algoritmos descritos na seção de metodologia para criar de fato o código final deste projeto (incluso no anexo B). A partir da seção 3.2.5, trata-se de como realizou-se experimentações na sala de reuniões do LARA. Na seção 3.2.5, descreve-se como fixou-se as câmeras no laboratório. Já a seção 3.2.6 trata em como se mapeou em coordenadas de câmera os objetos controláveis. Por fim, a seção 3.2.7 trata de como foi os cenários de teste realizados.

3.1 Materiais

Os materiais mais importantes foram as câmeras: duas câmeras com uma boa abertura focal para conseguir enxergar grande parte do ambiente mas sem ser uma abertura muito ampla, pois isso aumentaria as distorções. Além disso, buscou-se câmeras de boa resolução, percebeu-se que 720p (1280 pixeis x 720 pixeis) já garantia uma boa resolução. Mais do que isso poderia acarretar em mais custos computacionais de processamento. Com isso selecionou-se web-cams, pois elas procuram o causar o mínimo de distorções possível (constatado via processo de calibração), seu campo de visão é aceitável (em torno de 60º) e conseguem resoluções de 720p. Inicialmente utilizava-se duas câmeras Logitech C270. Mas devido a problemas de *drivers* delas, foi necessário trocar uma delas por uma Genius FaceCam 1000x, que possui características semelhantes ao da Logitech. Uma das características utilizadas para selecionar estas câmeras no início deste projeto foi também o de se ter um foco fixo, pois seria uma característica importante para reconstrução 3D, atualmente esta característica se tornou pouco relevante por não se realizar o processo de reconstrução 3D.

Além das câmeras, também é importante ressaltar os dispositivos de processamento: um computador e um Arduíno Uno. No início do projeto não se sabia o quão complexo ficaria o algoritmo final, por isso ao longo dele buscou-se simplificá-lo ao máximo. Hoje, percebe-se que um computador possui muito poder de processamento e que este algoritmo poderia ser implementado em plataformas um pouco mais simples, como um Raspberry Pi, tomando cuidado com questões de *driver* de comunicação USB com as câmeras. Como foi utilizado um computador ao longo deste projeto, foi necessário utilizar também um Arduíno Uno para comandar os objetos controláveis. Assim, acrescentou-se ao código do programa uma rotina de comunicação entre o computador e o Arduíno via USB. O Arduíno ao receber um comando do botão envia-o ao computador e o computador por sua vez envia a ele qual porta digital do Arduíno deverá alternar seu estado (porta esta correspondente ao objeto controlável).

Outros periféricos são importantes de se ressaltar. A começar com um módulo de rádio frequência (433 MHz) com transmissor e receptor (modelos MX-FS-03V / MX-05V). Este módulo permitiu com que o botão a ser pressionado pudesse ser sem fio. Ele utiliza-se o mesmo protocolo de alguns portões de garagem (no caso um protocolo trinário de codificação: 5 volts, 0 volts e alta impedância). Além deste periférico, também se criou módulos relés para acionar as lâmpadas e o ar-condicionado. A Figura 3.1 ilustra o circuito dos módulos de relés confeccionados. Estes relés devem ser alimentados por uma fonte de 5 V externa (no caso criou-se uma adaptando-se um carregador de telefone celular antigo), pois o Arduíno pode não se capaz de atuar em muitos relés ao mesmo tempo, por isso o módulo contém um transistor para amplificar a corrente.

Também foram criados suportes para as câmeras por um desenho de CAD (Figura 3.2) e impressão 3D. Utilizou-se a calibração de câmeras e o cálculo dos ângulos de campo de visão (descritos no Capítulo 2 seção 2.1.5) para fazer com que estes suportes fixassem as câmeras na orientação desejada.

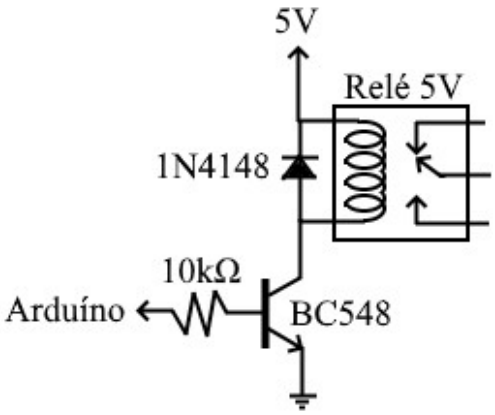

Figura 3.1 – Módulo relé confeccionado.

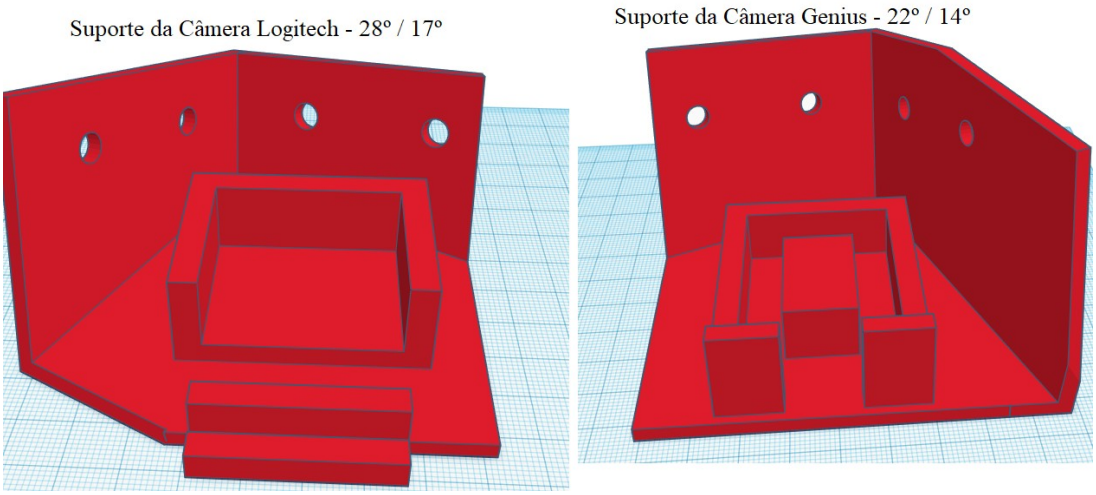

Figura 3.2 – Modelos em CAD dos suportes de câmera.

3.2 Métodos

3.2.1 Obtendo a informação de qual objeto foi apontado

Um objeto pequeno que esteja muito próximo de uma câmera ou grande que esteja muito distante dela pode ser representado pelos mesmos pixeis na imagem, como ilustra a Figura 3.3. Como as coordenadas de câmera são bidimensionais, então perde-se a informação de profundidade. Utiliza-se uma segunda câmera e procura-se dados na imagem da segunda câmera que possam contornar o problema da perda de informação de profundidade. Um dos métodos mais utilizados para isso é a reconstrução 3D [25]. Nela, geralmente se utiliza uma câmera estéreo, como o *Kinect*, para medir a posição de alguns pixeis no espaço tridimensional. Alguns pixeis chaves são identificados na imagem da primeira câmera e procura-se por outros pixeis na imagem da segunda câmera que correspondam aos mesmos pixeis chaves da primeira imagem. Com isso, devido ao efeito de *Paralax* e conhecendo as matrizes intrínseca e extrínseca das câmeras, é possível calcular a posição de objetos em coordenadas tridimensionais.

Uma escolha que foi feita ao longo deste trabalho então foi: deve-se realizar reconstrução 3D? A resposta encontrada foi de que não havia necessidade em fazê-la, pois encontrou-se uma solução mais computacionalmente rápida: percebeu-se que era possível contornar o problema da falta da informação de profundidade de uma reta tridimensional projetada na imagem das duas câmeras.

3.2.1.1 Linhas retas tridimensionais projetadas em duas câmeras

Quando uma pessoa aponta com a vareta, é possível obter em cada imagem uma reta bidimensional que indica a direção apontada. Esta linha nada mais é do que a projeção da reta tridimensional no plano de coordenadas das câmeras. Por mais que uma reta bidimensional em uma imagem possa retratar a projeção de infinitas retas tridimensionais ao variar a profundidade (como tratado pelo exemplo da Figura 3.3), ao se acrescentar a segunda reta bidimensional, pode-se obter a informação que faltava. Como ilustra a Figura 3.4, as possíveis retas vistas pelas câmeras pertencem a um plano no espaço tridimensional. A interseção destes planos corresponde a reta tridimensional real que originou as projeções. Ou seja, com as duas retas bidimensionais, recupera-se a informação de qual reta tridimensional foi projetada nas câmeras, sem necessariamente precisar obter a equação da reta tridimensional para isso.

Figura 3.3 – Objeto menor e mais próximo da câmera (a esquerda) e um objeto maior e mais distante da câmera (a direita).

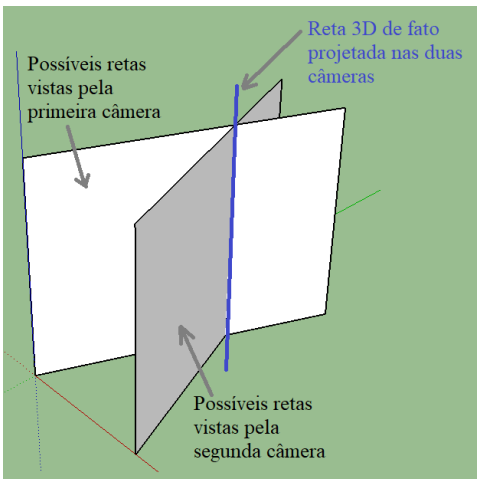

Figura 3.4 – Identificando uma reta 3D com as projeções dela em duas câmeras.

3.2.1.2 Pontos tridimensionais projetados em duas câmeras

Para este projeto, serão mapeados os objetos controláveis/apontáveis como pontos tridimensionais projetados nos planos de coordenadas das câmeras. Assim como no caso anterior, onde retas bidimensionais correspondem a um plano que contém infinitas possibilidades de retas tridimensionais, um ponto bidimensional corresponde a uma reta que contém infinitas possibilidades de projeções de pontos tridimensionais. E assim como no caso anterior, para duas câmeras, existem duas retas que correspondem a projeção deste ponto e, com a interseção delas, obtém-se a informação de qual ponto tridimensional de fato foi projetado. Ou seja, com os dois pontos bidimensionais, recupera-se a informação de qual ponto tridimensional foi projetado nas câmeras, sem necessariamente precisar calcular as coordenadas tridimensionais dele.

3.2.1.3 Identificando o objeto apontado

Como o objeto apontado é representado por um ponto em duas imagens e como a reta que a vareta aponta é representada por uma reta em duas imagens, então percebeu-se que não havia necessidade de se calcular a equação da reta tridimensional da direção apontada. Percebeu-se que poderia trabalhar nas duas coordenadas bidimensionais de câmera sem precisar converter para coordenadas tridimensionais e ainda assim contornaria o problema da falta de informação da profundidade.

Como também o apontar de uma pessoa não é preciso, a reta que ela aponta não necessariamente coincidirá com o alvo que ela busca apontar. Como ilustra a Figura 3.5, as retas verdes indicam as retas que a vareta está de fato apontando e a reta em que a pessoa enxerga o alvo (ponto em azul). Para tratar esta imperfeição, acrescenta-se uma margem de tolerância (representada pelas linhas laranjas). Desta forma, o objeto identificado será aquele que estiver dentro desta margem.

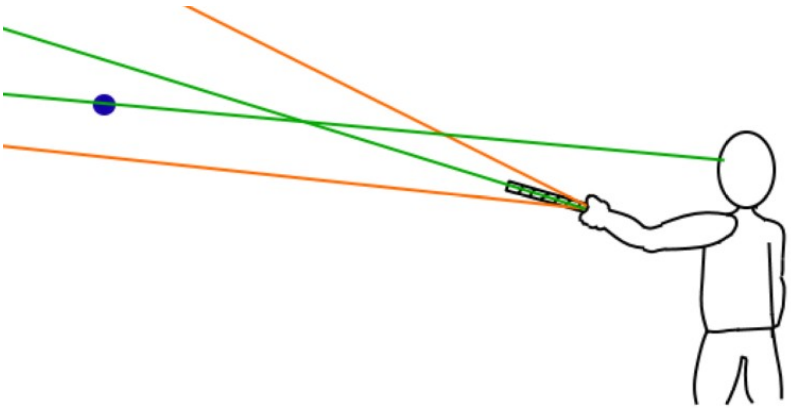

Figura 3.5 – Apontamento não é preciso.

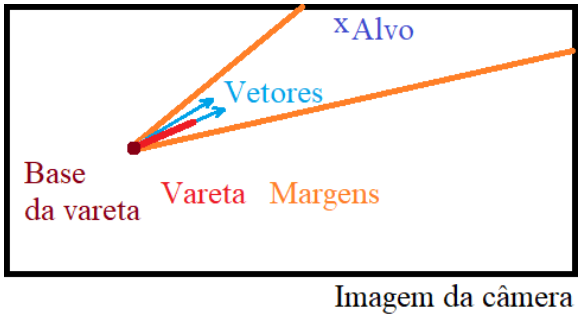

Figura 3.6 – Margens no apontamento em coordenadas de câmera.

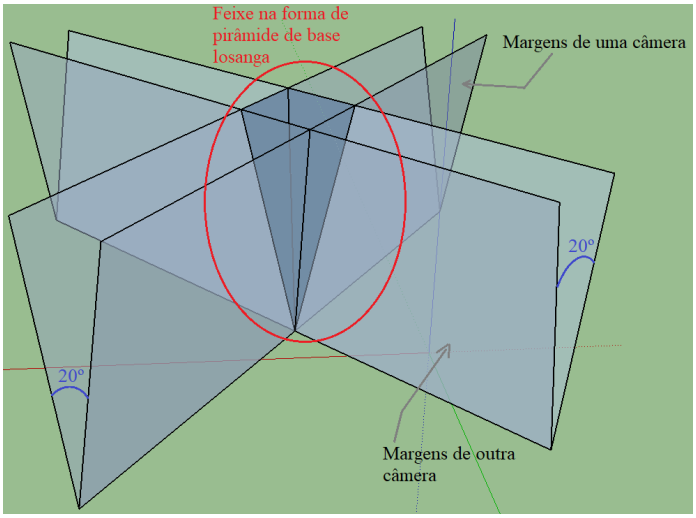

Figura 3.7 – Margens no apontamento no espaço 3D.

Detalhando melhor esta margem: para cada imagem será identificado a reta em que a vareta aponta, além dela, também será identificado o ponto de base da vareta (que da Figura 3.5 é o ponto de interseção das linhas laranjas). Com a base da vareta e a reta criada por ela, determina-se um primeiro vetor, como ilustra a Figura 3.6. Além deste vetor, pode-se criar um outro vetor entre a base da vareta e o alvo, também representado pela Figura 3.6. É possível descobrir a partir de um produto interno o ângulo entre estes vetores, como indica a equação

$$cos(\theta) = \frac{< \overrightarrow{v_1}, \overrightarrow{v_2} >}{||\overrightarrow{v_1}|| * ||\overrightarrow{v_2}||}.$$

Assim, de fato a margem se encontra no valor mínimo que o cosseno obtido pelo produto interno. Para este trabalho, o ângulo máximo de tolerância foi de 10º.

A Figura 3.6 ilustra a representação tridimensional dessas margens. Como tratado na seção 3.2.1.1, uma linha no espaço de bidimensional de câmera corresponde a um plano no espaço tridimensional. Portanto, no espaço tridimensional, estas margens correspondem a um feixe de forma de uma pirâmide de base losanga, onde as laterais desta pirâmide correspondem aos planos formados pelas retas das margens. Dependendo da posição da vareta, os ângulos da base losanga da pirâmide variam. Assim, a base deste feixe não é constante, somente o ângulo entre as marges de uma câmera são constantes e, no caso deste projeto, iguais a 20º.

Repare pela Figura 3.6 que vários objetos podem ser identificados na região entre as margens de uma câmera ou entre as margens de outra câmera. Mas a região de interseção entre elas (a pirâmide de base losanga) possui um volume muito menor e portanto a possibilidade de mais de um objeto estar contido dentro dela é muito menor. Assim, se determina o objeto em que a vareta aponta.

3.2.1.4 Dois ou mais alvos dentro das margens

Se ainda assim ocorrer de dois ou mais alvos estiverem dentro do feixe piramidal formado pelas margens de apontamento da vareta, então deve-se selecionar um deles para ser o alvo em que a pessoa de fato está apontando.

A Figura 3.8 ilustra uma reta tridimensional onde um ponto dela é a base da vareta, outro ponto um alvo e um terceiro ponto, entre os dois primeiros, outro alvo. Ela também ilustra as margens do feixe de apontamento piramidal. Decidiu-se que neste caso, se esta linha estiver dentro do feixe de apontamento, a pessoa apontará para o alvo mais próximo. Se projetado esta reta e alvos em coordenadas de câmera, as distâncias serão alteradas, mas se manterão proporcionais às distâncias originais. Ou seja, se o Alvo 1 está mais perto da base da vareta em coordenadas tridimensionais, então também estará em coordenadas de câmera. Desta forma, para este caso especial, optará em selecionar os alvos mais próximos da base da vareta.

3.2.2 Posicionamento das Câmeras

Pela Figura 3.7, pode-se começar a deduzir o melhor posicionamento das câmeras buscando-se minimizar o volume do feixe piramidal. A Figura 3.9 ilustra o que ocorre com este feixe ao posicionar as câmeras afastadas uma da outra e de forma ortogonais (melhor caso) e quando as posiciona muito próximas uma da outra e quase paralelas (pior caso). Outra informação que pode ser inferida pelo pior caso da Figura 3.9 é que se uma câmera estiver contra a outra, também será formado um feixe com um volume muito grande, o que aumenta a possibilidade de mais de um objeto estar dentro dele.

Portanto, as primeiras regras no posicionamento das câmeras é afastá-las ao máximo e deixá-las o mais ortogonal possível. O problema em deixá-las o mais ortogonal possível se encontra na área útil do ambiente em que o controle poderá operar. Esta área é a região de interseção dos campos de visão das duas câmeras. Ou seja, se uma câmera enxerga a vareta e a outra não, a vareta se encontra fora da área útil do ambiente. Já se a vareta se encontra dentro do campo de visão das duas câmeras, ela está dentro da área útil. Quanto mais ortogonal estiverem as câmeras, menor será esta área, como ilustra a Figura 3.10.

Com base nestas informações, percebeu-se que a melhor forma de se posicionar as câmeras é primeiro afastando-as e em seguida rotacionando-as de forma que o campo de visão tangencie as paredes do ambiente, como ilustra a planta baixa da direita da Figura 3.10. Desta forma maximiza-se a área útil e evita-se um feixe piramidal muito volumoso.

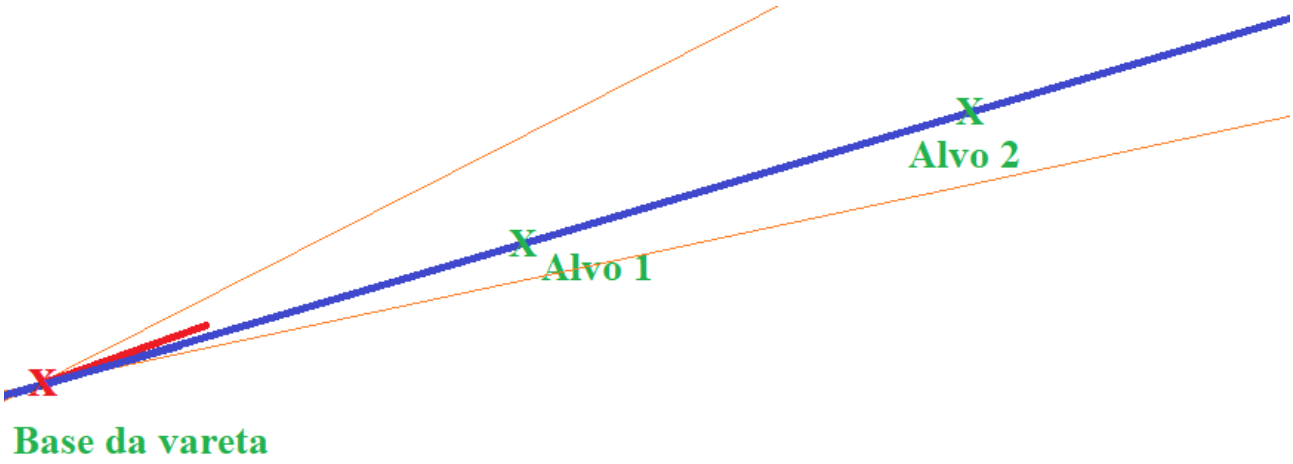

Figura 3.8 – Três pontos em uma reta (3D ou 2D).

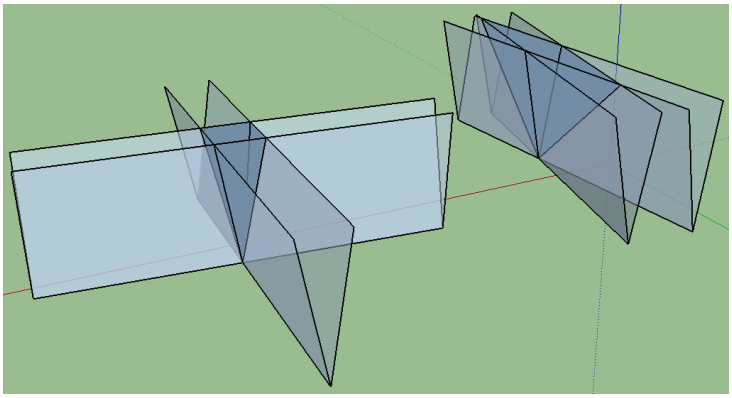

Figura 3.9 – Posicionamento das câmeras buscando minimizar volume do feixe piramidal.

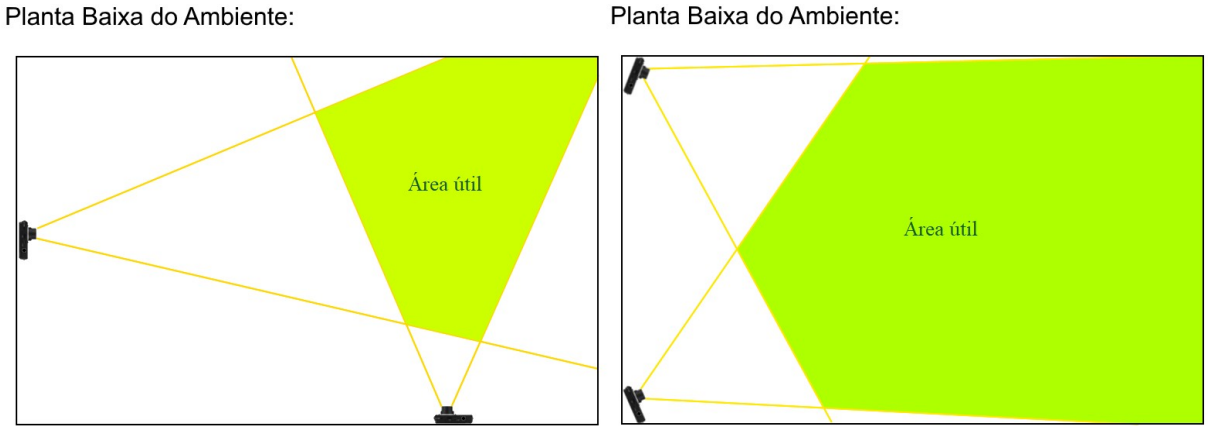

Figura 3.10 – Posicionamento das câmeras buscando maximizar a área útil. (À esquerda: câmeras ortogonais. À direita: câmeras afastadas e com campo de visão tangenciando as paredes).

3.2.3 Confecção e Detecção da Vareta

A primeira característica na escolha do objeto que será utilizado para apontar deve ser a seu esforço computacional versus sua esforço operacional, ou seja, o quão difícil é para um computador de processar versus o quão difícil é de um ser humano operá-lo. Por exemplo, apontar com as mãos é uma das formas mais fáceis de uma pessoa apontar, no entanto é computacionalmente muito custoso fazê-lo (esta foi a forma que idealizou este projeto). Dando um passo para trás, criar um objeto passivo (que reflita luz) com padrões visuais bem definidos ajuda a reduzir muito a complexidade na identificação dele (comparado com a complexidade de se detectar a mão humana). Outros passos para

trás seriam: a criação de um objeto ativo (que emite uma luz bem definida), ou então, utilizar rádio frequência para identificar com exatidão a posição de orientação do objeto. No entanto, para não fugir muito da forma ideal (de apontar com as mãos), decidiu-se manter nessa linha de um objeto passivo com padrões visuais bem definidos.

Existem inúmeras formas de se escolher um padrão para este objeto. Como ele serve para apontar para uma direção, optou-se por ele ter um formato de vareta pequena (em torno de 30 cm). Primeiramente confeccionou-se uma vareta o mais simples possível: somente uma vareta de cor vermelha. A vantagem dela está em sua simplicidade. A desvantagem se encontra em extraí-la da imagem, visto que outros objetos vermelhos também serão extraídos juntos com ela da imagem na forma interferência ou ruído. Assim, torna-se muito complicado detectá-la e também é passível de surgir falsos positivos (identificar outros objetos que não sejam a vareta como sendo ela). Para contornar o problema de falsos positivos, decidiu-se dar uma forma a esta vareta, colocando uma esfera em sua ponta. Por mais que tenha ajudado, ainda não foi o suficiente, pois outros objetos vermelhos continuavam interferindo e foi necessário utilizar algorítimos muito complexos e computacionalmente pesados para reconhecer este padrão morfológico e evitar falsos positivos, que ainda ocorriam (no entanto menos que a vareta simples).

Foi então que surgiu a ideia de alterar o padrão no nível da cor e não da forma. Primeiramente criou-se uma vareta com três esferas de cores diferentes (vermelho, verde e azul) alinhadas e igualmente espaçadas. A vantagem está no fato de ser um padrão muito bem definido e que dificilmente terá problemas de falsos positivos. A desvantagem se encontra no fato de que serão gerados três máscaras e que virão junto a elas outros objetos que possuam estas cores. Portanto, esta vareta é capaz de evitar falsos positivos, no entanto ainda requer um grande poder computacional para isolá-la de outros objetos extraídos juntos.

Por fim, encontrou-se uma forma de se remover grande parte das interferências aproveitando do fato de se utilizar múltiplas cores, e ainda, de uma forma muito simples e com um custo computacional quase que nulo! Primeiramente, criou-se uma vareta com duas cores intercalando elas em um padrão listrado. Em seguida, procurou-se não pelas cores em si, mas pela região de transição entre elas. Para isso, gera-se duas máscaras extraindo as duas cores (como nas situações anteriores). Em seguida, desloca-se uma máscara sobre a outra para esquerda ou para cima e realiza uma operação de interseção lógica entre elas. Deste modo, semente serão extraidos os objetos que possuam uma interseção destas duas cores. A Figura 3.11 ilustra como se dá este procedimento. Já as Figuras 3.12, 3.13 e 3.14 mostram os resultados práticos ao se deslocar para esquerda e fazer a interseção delas. Além deste processo, realizou-se uma operação de erosão e depois de dilatação para filtrar qualquer ruído remanescente. Por fim, utilizou-se detecção de contornos externos e momentos para calcular os centros geométricos destas regiões de interseção (os pontos azuis indicados pela Figura 3.14).

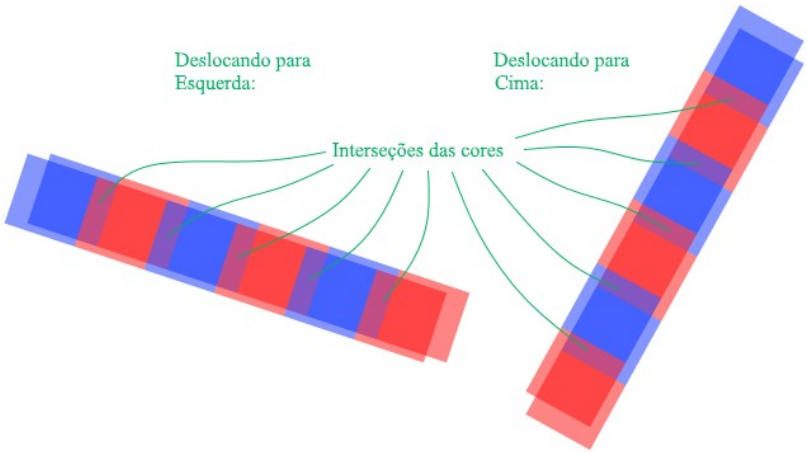

Figura 3.11 – Deslocamento para esquerda e para cima.

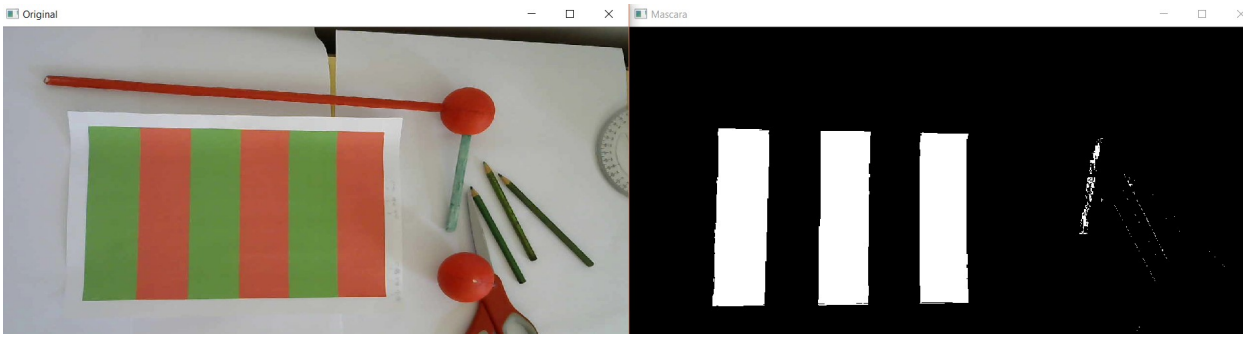

Figura 3.12 – Extraíndo as cores verdes da figura.

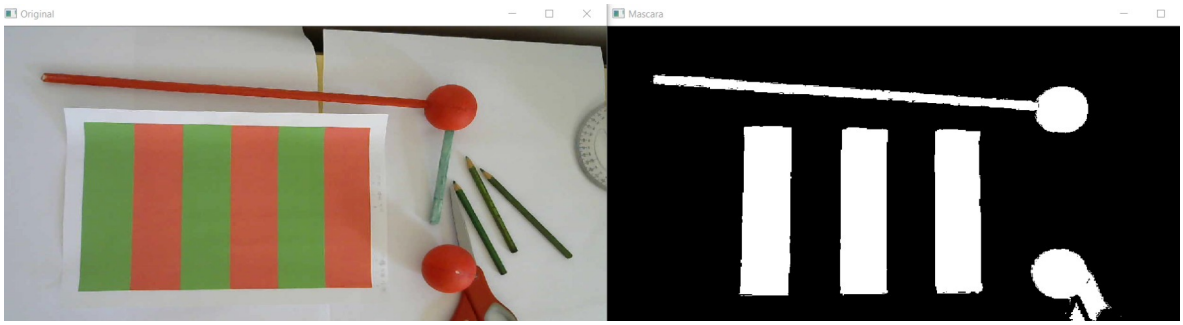

Figura 3.13 – Filtrando as cores vermelhas da figura.

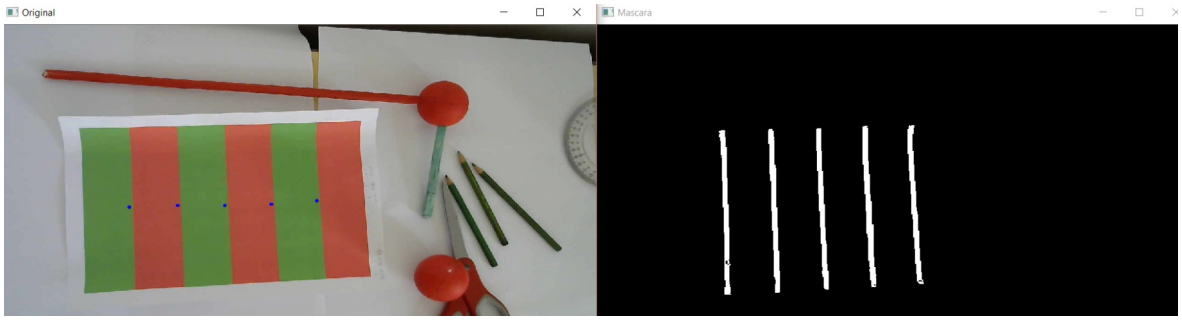

Figura 3.14 – Região de Interseção das cores ao deslocar para esquerda.

A respeito das cores escolhidas e a forma como se extraiu elas da imagem foi tratado na seção 2.2, pois envolve questões de influência da iluminação e requer um Capítulo a parte no seu desenvolvimento. Continuando com a detecção da vareta: encontrar as interseções das cores foi apenas a primeira parte, na qual removeu-se grande parte das

interferências de outros objetos de mesmas cores. No entanto, ainda é possível que existam objetos com as mesmas cores intercaladas e não se pode ter falsos positivos. Portanto um padrão morfológico foi utilizado: a quantidade de listras e o fato de estarem igualmente espaçadas.

As varetas criadas ao longo do projeto a partir deste ponto foram confeccionadas com papel de cartolina branca e tinta acrílica fosca. Elas possuem um diâmetro em torno de 3 cm e seis listras de 6 cm (três de cada cor), totalizando 36 cm do comprimento total da vareta. Desta forma, o padrão morfológico são de cinco interseções de cores dispostas em linha reta e igualmente espaçadas. Além disso, repare que na Figura 3.14 os centros geométricos (representados por pontos azuis) estão levemente deslocados para esquerda e estão em cima das regiões coloridas. Ou seja, pode-se atribuir a estes pontos a característica de qual cor eles estão em cima. Deste modo, pode-se utilizar esta característica para um terceiro padrão: estas cores se alternam. A Figura 3.15 ilustra os possíveis casos em que a vareta pode estar orientada (deslocando para esquerda ou para cima). Mais a frente utilizará esta Figura para se determinar o sentido em que ela está apontado, como ilustrado pela Figura 3.17.

Mas antes de se encontrar o sentido, deve-se encontrar quais centros geométricos pertencem a vareta e quais são de outros objetos externos que foram extraidos juntos. Para isso, será utilizado o exemplo ilustrado pela Figura 3.16 para explicar como se dá o algorítimo de detecção da vareta.

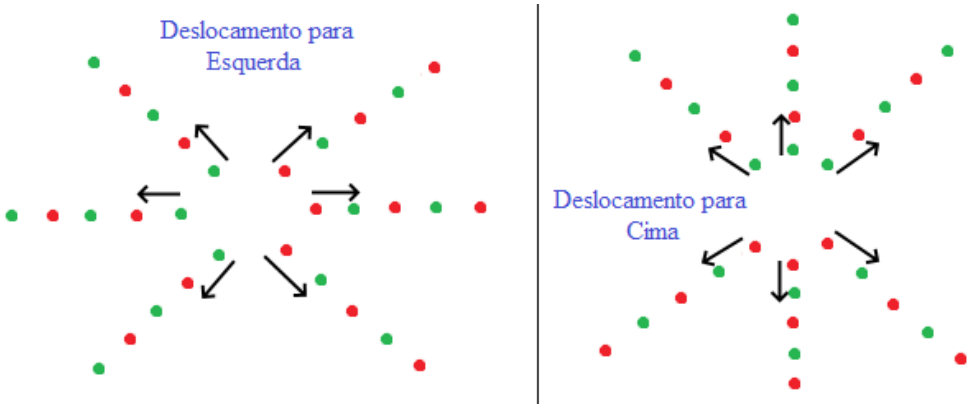

Figura 3.15 – Padrão da vareta e sentido de apontamento da vareta.

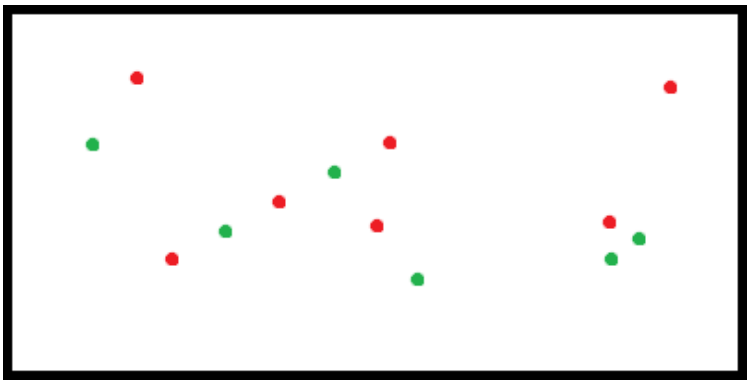

Figura 3.16 – Exemplo de um possível caso, representando os centros geométricos das regiões de interseção de cores e com a característica de qual cor estes pontos estão em cima.

As próximas etapas do algorítimo buscam encontrar: a base da vareta, a direção e o sentido em que ela está apontando. Onde a base da vareta é o ponto de centro geométricos correspondente a interseção do vermelho mais extremo da vareta com o verde próximo a ele, como ilustra a Figura 3.17.

A primeira etapa deste algorítimo é em se ordenar estes pontos em duas listas (de pontos verdes e de pontos vermelhos). A ordem se dará pela distância destes pontos com o ponto de base da vareta da leitura anterior. Se este ponto base não existir (nos casos de primeira leitura), então é utilizado a origem da imagem (no canto superior esquerdo dela). A Figura 3.18 ilustra como se dá a ordem dos pontos do exemplo da Figura 3.16 no caso de ordenando com a distância com a origem da imagem.

A próxima etapa utiliza-se de um pouco de força bruta: para cada ponto vermelho, forma-se pares com cada um dos pontos verdes. Com estes pares, estima-se a posição de outros dois pontos. Esta estimativa é dada pelas distâncias entre os pontos nos eixos x e y da imagem, como ilustra pelos pontos laranjas nas Figuras 3.19 e 3.20. Para estes dois pontos estimados, procura em suas respectivas listas se existem pontos nelas próximos a eles (dado uma pequena margem de tolerância). Como estas listas estão ordenadas por distância (com a base da vareta ou com a origem da imagem), então não precisa percorrer toda a lista procurando por pontos e sim até que distância de um ponto da lista fique maior do que a distância do ponto estimado (mais uma margem de tolerância). No caso deste exemplo, os pontos estimados não correspondem a nenhum ponto das listas, então forma-se um novo par mantendo o vermelho e passando para o próximo ponto verde, seguindo com este padrão até percorrer ambas as listas.

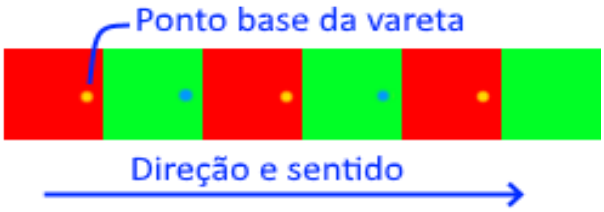

Figura 3.17 – Ponto base da vareta, direção e sentido.

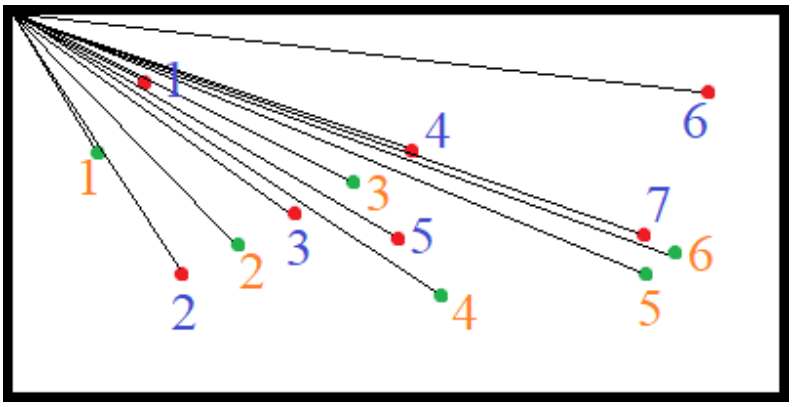

Figura 3.18 – Ordem dos pontos com relação a distância com a origem.

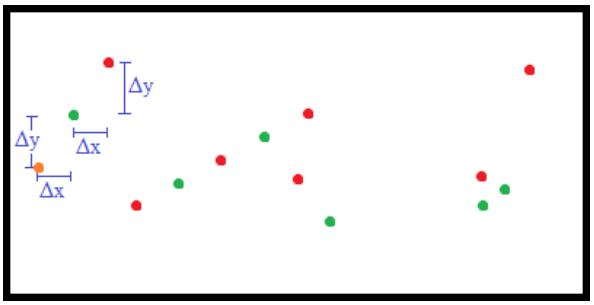

Figura 3.19 – Estimativa de posição de um outro ponto vermelho (representado pelo ponto laranja).

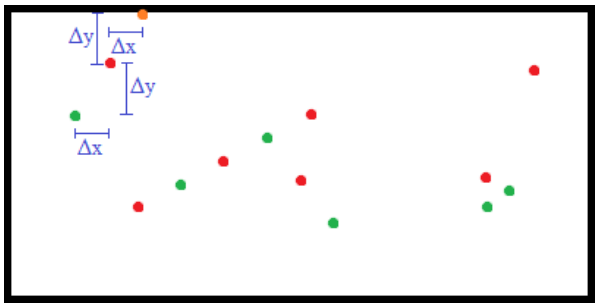

Figura 3.20 – Estimativa de posição de um outro ponto verde (representado pelo ponto laranja).

Note pela Figura 3.21 que quando se utiliza o primeiro ponto vermelho e o terceiro ponto verde, foi possível estimar a posição de um novo ponto vermelho, mas não é possível estimar a posição de um ponto verde, pois ele se

encontraria fora da imagem. Neste caso, nem precisa percorrer a lista de pontos verdes procurando pela existência de um ponto próximo, pois é evidente que não existirá.

Muito bem, a vareta de fato só será encontrada quando parear o segundo ponto vermelho com o segundo ponto verde, como ilustra a Figura 3.22. Repare que a estimativa de um ponto vermelho coincide com um ponto vermelho da lista, já a estimativa de um ponto verde não coincide com nenhum ponto da outra lista. Uma vez em que isso ocorre (um ponto estimado é encontrado), realiza-se a terceira etapa deste algorítimo, que é procurar por mais pontos na linha reta destes pontos já encontrados. O algorítimo é parecido com o anterior, onde se estima pontos e procura nas listas por outros próximos aos estimados. No entanto, continuará procurando pontos na linha reta até que não sejam mais encontrados nenhum.

Se forem encontrados um total de seis pontos com cores alternadas em uma linha reta ou menos de cinco, esta etapa do algorítimo retorna para a etapa anterior com uma indicação de falha, pois a vareta possui cinco cores. Já se forem encontrados exatamente cinco pontos, então estes pontos são tratados como um possível candidato a ser a vareta.

O algorítimo, no entanto, não para por ai. Ele retorna para a segunda etapa e continua procurando por mais candidatos a vareta. Se dois objetos distintos na imagem forem formados por cinco pontos em linha reta e com cores alternadas, o algorítimo retorna uma falha indicando que duas varetas foram encontradas (sendo que deveria ter apenas uma). Se chegar até o final da segunda etapa com apenas uma vareta identificada, então ele segue para a quarta etapa, que identifica a direção e o sentido da vareta.

A quarta e última etapa busca o sentido da vareta. Para isso, volte na Figura 3.15 e repare na quantidade de pontos verdes e vermelhos. Deslocando para esquerda, se houver mais pontos verdes do que vermelho, a vareta está apontando para esquerda, caso contrário, ela está apontando para direita. Já no caso de deslocamento para cima, a vareta está apontando para cima quando houver mais pontos verdes que vermelhos, caso contrário, ela está apontando para baixo.

Alguns detalhes importantes que não foram tratados são: Para o exemplo em que vinhamos trabalhando, será encontrada um candidato a vareta. Quando for fazer novos pares (na segunda etapa), eles não podem ser feitos com pontos da vareta, pois se forem, a terceira etapa encontrará novamente os mesmos pontos e o algorítimo como um todo responderá que duas varetas foram encontradas, o que seria falso (a Figura 3.23 ilustra esta situação). O mesmo tratamento é feito para objetos com seis ou mais pontos encontrados, mesmo que não sejam candidatos a vareta. Neste caso, isso é apenas para reduzir o esforço computacional.

A Figura 3.24 ilustra duas varetas, uma com cinco interseções e outra com mais de cinco. Quando a vareta é identificada, marcou-se os seus centros geométricos. Repare que na Figura 3.24 a vareta foi encontrada, a pesar da existência de um outro objeto com o mesmo padrão que ela, só com mais cores. A Figura 3.25 ilustra as mesmas duas varetas, no entanto, uma sobre a outra. Repare que mesmo assim a vareta foi identificada. Já as Figuras 3.26 e 3.27 ilustram a direção e sentido obtidos.

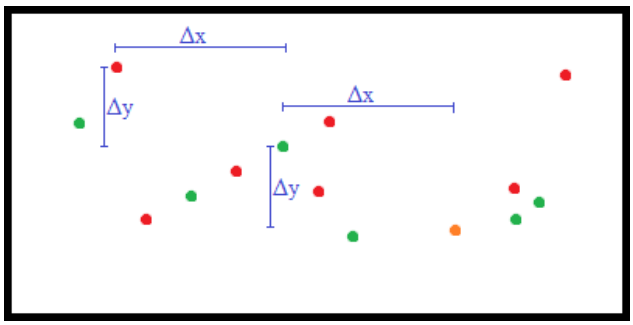

Figura 3.21 – Estimando novos pontos com o primeiro ponto vermelho e o terceiro ponto verde só é capaz de estimar um ponto vermelho e nenhum ponto verde.

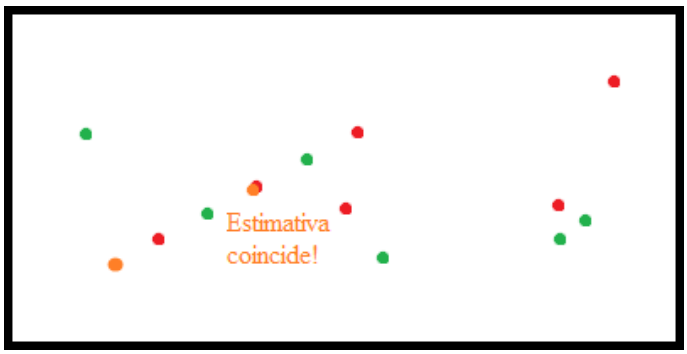

Figura 3.22 – Estimativa de um ponto que coincide com um ponto da lista.

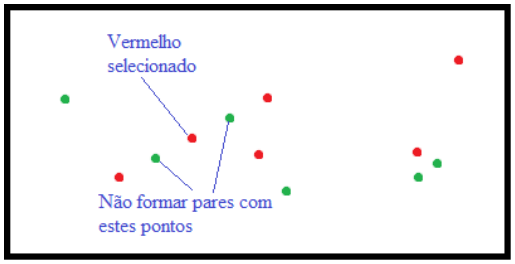

Figura 3.23 – Não formar pares com pontos da vareta já encontrados.

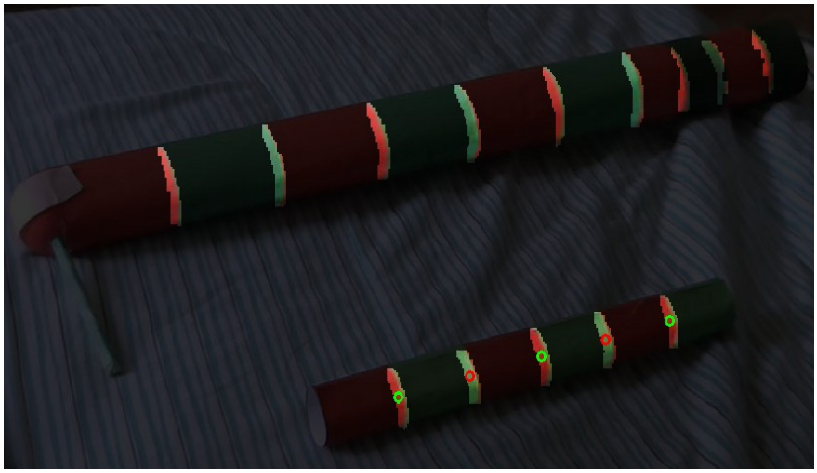

Figura 3.24 – Dois objetos, sendo o de baixo a vareta.

Figura 3.25 – Vareta em cima de um objeto com mais de cinco interseções.

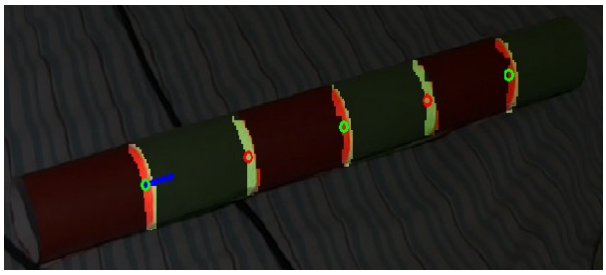

Figura 3.26 – Vareta apontando para direita.

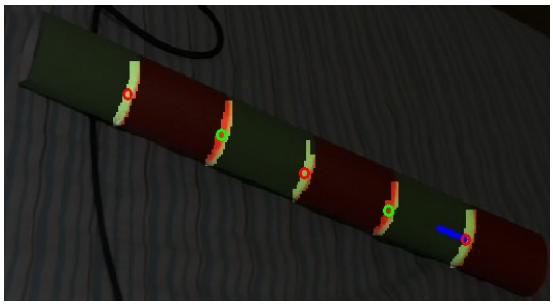

Figura 3.27 – Vareta apontando para esquerda.

3.2.4 Algoritmos

Uma pessoa apontará com a vareta para um dos objetos controláveis, no caso deste projeto são duas lâmpadas e um ar-condicionado na sala de reuniões do laboratório LARA (SG11 da Universidade de Brasília). No que a pessoa apertar um botão, o estado do objeto em que ela estiver apontando alterará de ligado/desligado. Na seção 3.1.1 tratará de como estava o algoritmo, uma versão mais próxima de um apontamento com as mãos livres (sem a vareta). Já a seção 3.1.2 tratará de como ele está atualmente, mais simplificado e portanto mais rápido. Só então será tratado os materiais e métodos.

3.2.4.1 Versão antiga

Antes, o algoritmo executava ininterruptamente a rotina ilustrada pela rede de Petri da Figura 3.28. Nela, para todos os quadros dos vídeos das câmeras, procurava ou rastreava a vareta. Com isso enquanto procurava a vareta, também subtraia-se o fundo, para reduzir a quantidade de objetos da imagem e facilitar assim a procura da vareta. Já quando a vareta era encontrada, utilizava o processo de rastreamento descrito na seção 2.1.7 e deixava de subtrair o fundo. Somente quando a vareta era vista pelas duas câmeras e quando apertava o botão que procurava identificar o objeto apontado e comandá-lo.

Esta versão é interessante para um caso futuro em que se deseje modificar este projeto para encontrar as mãos das pessoas no ambiente e a direção apontada por elas. Porque neste caso com as mãos livres, não existirá um botão a ser apertado e portanto deverá ficar constantemente realizando o rastreamento dela até que ela aponte para um objeto controlável.

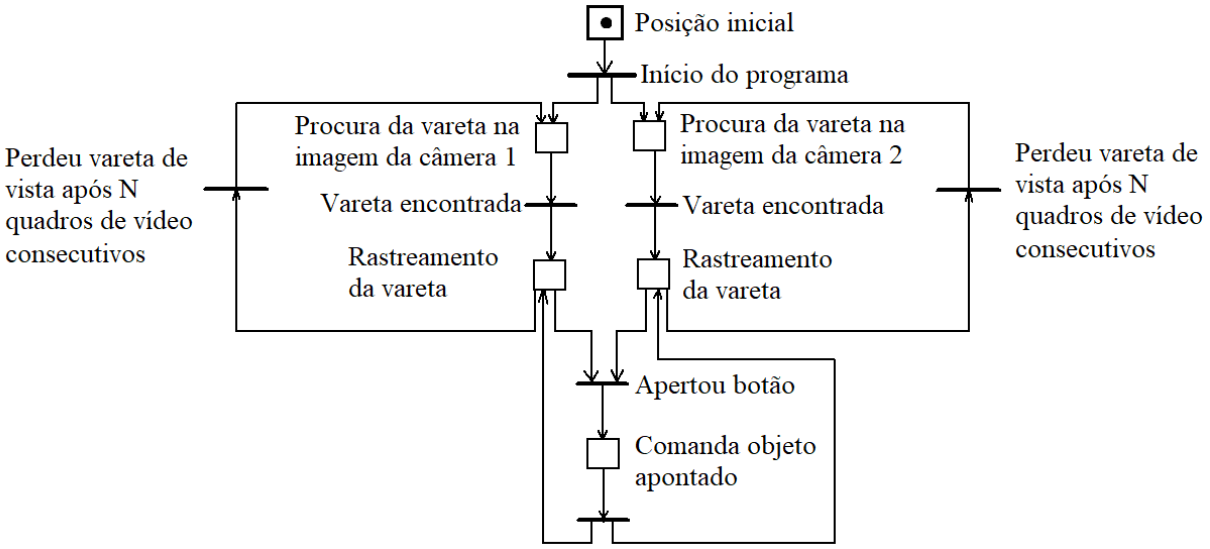

Figura 3.28 – Rede de Petri do algoritmo antigo.

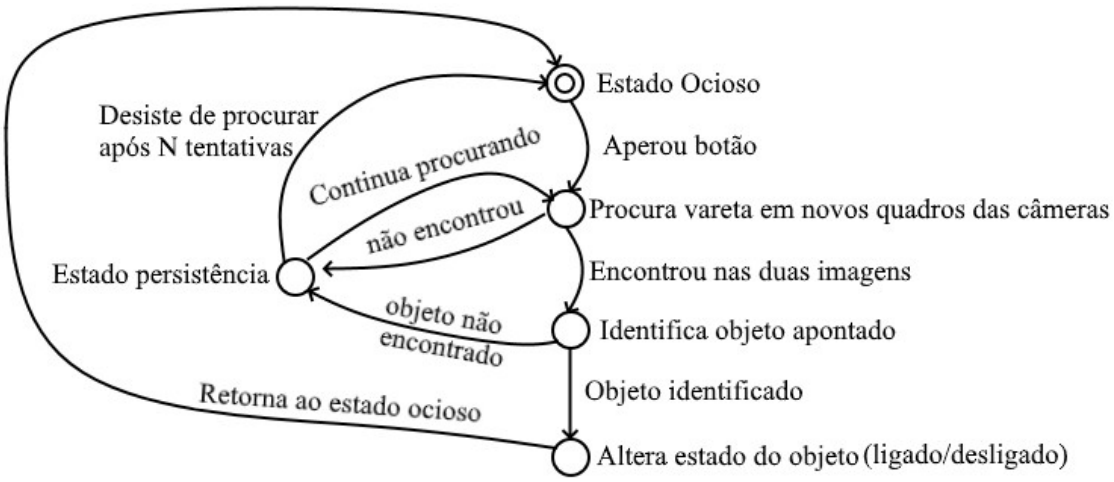

Figura 3.29 – Espaço de estados do novo algoritmo.

3.2.4.2 Nova versão

Como o algoritmo deste projeto busca procurar uma vareta comandada pelo apertar de um botão, então decidiu-se alterar a ordem dos eventos retratados pela Figura 3.28. Agora, o algoritmo permanecerá em estado ocioso até que o botão seja apertado. Só quando ele é apertado que procura-se pela vareta nas duas imagens, seguindo o algoritmo descrito na seção 3.2.3. Desta forma, não há necessidade de se capturar todos os quadros de vídeo, e sim somente os quadros após o aperto do botão. Assim, não é mais necessário realizar subtração de fundo e rastreamento da vareta com região de interesse.

Esta modificação fez com que o algoritmo ficasse ainda mais rápido e a remoção da subtração de fundo e região de interesse não aumentou muito o tempo de processamento. A Figura 3.29 apresenta o espaço de estados do novo algoritmo. A descrição detalhada dos algoritmos de procura da vareta e de identificação do objeto apontado encontram-se nas seções 3.2.3 e 3.2.1, respectivamente.

3.2.5 Fixação das câmeras no ambiente

Fixou-se as câmeras de acordo com a forma apresentada pela Figura 3.10 (planta baixa da direita). Fixou-se as câmeras nos cantos superiores das paredes buscando com que seus campos de visão tangenciassem as paredes. Elas foram posicionadas o mais alto possível, somente abaixo do nível de alguns objetos do teto da sala. Além disso, inclinou-as um pouco para baixo de forma que seus campos de visão também tangenciassem o teto (de fato um pouco mais para baixo do que a tangente do teto). A fixação se deu pelos modelos de CAD impressos apresentados pela Figura 3.2. A Figura 3.30 mostra a foto das câmeras posicionadas na parede.

Uma decisão contraindicada tomada no posicionamento das câmeras foi colocá-las contra a luz solar externa (contra as janelas). Decidiu-se por elas nesta posição, no entanto, pois um dos cenários de teste foi contra a iluminação externa. No mais, a sala possui um bom sistema de cortinas capaz de bloquear a maior parte da iluminação externa, o que reduz muito o prejuízo causado por ela.

3.2.6 Mapeando objetos controláveis

Como visto na seção 3.2.1, os objetos controláveis são mapeados por dois pontos, um em cada imagem de câmera. Além disso, essa seção também demonstrou que linhas tridimensionais são projetadas como linhas bidimensionais. Utilizando-se deste princípio, fixou-se um barbante trançado (de maior espessura) nos alvos de forma que, ao esticá-lo, formasse linhas nas imagens das duas câmeras. Criou-se um programa capaz de receber estas imagens e traduzir em suas equações de retas (utilizando extração de cores e algoritmo de Hough para retas). No entanto, este programa não se demonstrou muito preciso e então realizou-se este procedimento de forma manual.

Para cada alvo, obteve-se várias imagens, variando-se o ponto de fixação do barbante trançado. Para cada imagem, utilizou-se uma ferramenta de edição de imagens para adquirir a posição dos pixeis mais extremos possíveis de ser identificados da reta formada pelo barbante (alguns pixeis de algumas imagens foram mais difíceis de se adquirir devido a condições de iluminação e distância/foco da câmera). Mapeando-se cada par de pixeis, criou-se um programa em *Matlab* para gerar imagens das linhas formadas por eles. Assim, selecionou-se as melhores retas para encontrar o ponto de interseção entre elas.

Figura 3.30 – Fixação das câmeras no ambiente.

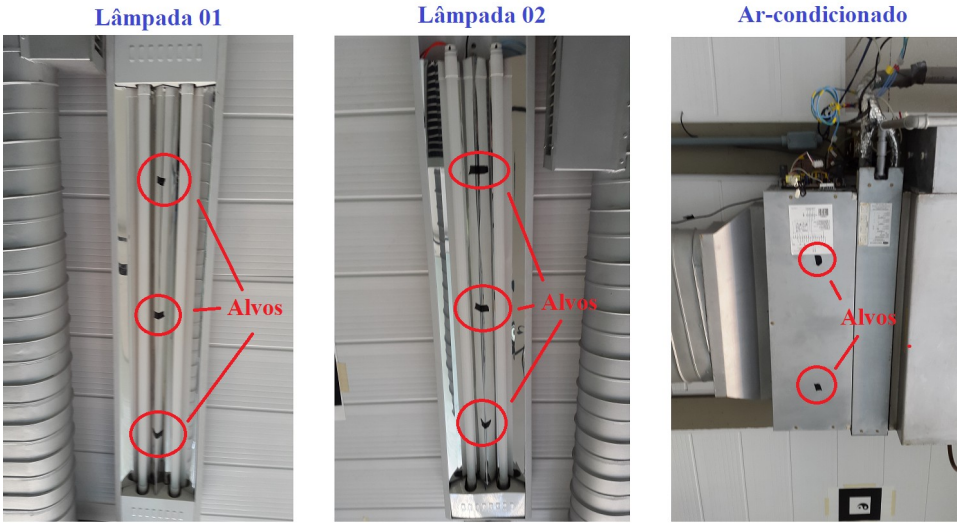

Figura 3.31 – Alvos das lâmpadas e do ar-condicionado.

Cada objeto controlável poderia ser indicado por apenas um alvo. No entanto, como são objetos compridos, decidiu-se definir cada lâmpada por três alvos e o ar-condicionado por dois alvos, como ilustra a Figura 3.31. Modificou-se o código do programa para aceitar com que alvos de um mesmo objetos sejam reconhecidos como o "mesmo". Porque dois alvos diferentes dentro do mesmo feixe retorna o alvo mais próximo, como descrito no Capítulo 2 seção 2.4.3. Esta modificação foi na adição de um número identificador (*id*) para os alvos e fazendo este identificador ser o mesmo para os alvos de um mesmo objeto. No caso utilizou-se a porta digital do Arduíno que comanda estes objetos como o número de identificação. Assim, a rotina que identifica o objeto apontado retorna a porta do Arduíno do objeto apontado ao invés do alvo como um todo (uma estrutura de dados com outras informações a mais a respeito dele).

3.2.7 Cenários de teste

Foram testados seis cenários. O primeiro utilizou o barbante utilizado para mapear os alvos para servir de guia para a vareta apontar sempre para um ponto fixo. No caso os pontos que foram fixados foram: entre os alvos 1 e 2 da lâmpada 1, entre os alvos 2 e 3 da lâmpada 1, entre os alvos 1 e 2 da lâmpada 2, entre os alvos 2 e 3 da lâmpada 2 e entre os alvos do ar-condicionado. Desta forma, garantiu-se que a vareta sempre apontaria para estes pontos, pois ela é um cilindro oco e passou o babante por dentro dela e a fixou nele com fita adesiva. Como são alvos internos, garantiu-se que apontaria para os objetos que eles representam. Então, variou-se a posição da vareta no ambiente para cada ponto fixo para validar o mapeamento dos alvos.

Os demais cenários foram menores. O segundo cenário testou com o ambiente muito escuro (com um pouco de iluminação externa refletida). O terceiro cenário utilizou uma lanterna com capacidade de regular sua intensidade luminosa, utilizou ela no máximo e no mínimo. O quarto cenário abriu-se as cortinas e voltou-se as câmeras contra o Sol. O quinto cenário buscou testar mover a vareta após apertar o botão, mirando de um ponto fora dos alvos controláveis e movendo em direção aos objetos controláveis. Já o sexto cenário fixou a vareta em pontos arbitrários e também a utilizou para apontar de forma livre.

Por fim realizou-se um teste quantitativo apontando 101 vezes para os alvos e identificando quantas tentativas foram necessárias para o programa identificar a vareta e quantas forma necessárias para ele identificar o objeto apontado.

Capítulo 4

Análise dos Resultados

Este capítulo tratará dos resultados obtidos neste projeto. O primeiro resultado importante foi o das etapas da criação da vareta, em especial como se tratou o problema de um ambiente externo sujeito a variações de iluminação. Nesta etapa, decobriu-se que o espaço de cores CIE LAB melhor trata este problema, mas ele não é tão ótimo, pois deve-se selecionar cores muito específicas, claras mas com grande saturação para funcionar. Em seguida, informou-se os resultados obtidos do cálculo do campo de visão das câmeras e como o utilizou para criar os suportes delas e as posicionou na sala. Depois, tratou-se de como mapeou os alvos dos objetos controláveis, obtendo-se as coordenadas de câmera respectivas aos alvos. Em seguida, descreve-se dos resultados obtidos testando-se os cenários qualitativos, tratando principalmente dos resultados negativos obtidos neles. Por fim, encontra-se a média e desvio padrão da quantidade de tentativas que o programa leva para encontrar a vareta nas imagens e para encontrar o alvo apontado.

4.1 Criação da Vareta

Fez-se a primeira vareta com duas cores: verde e vermelho e utilizou-se o espaço HSV para extrair estas cores selecionadas. Percebeu-se que, quando a iluminação é constante, este espaço funcionou perfeitamente para encontrar estas cores. No entanto, ao se variar a iluminação local, tanto os níveis S e V (de saturação e de valor) variavam consideravelmente. Uma opção para contornar este problema seria ignorar estas dimensões e utilizar somente a dimensão H (de "*hue*" / tonalidade). Entretanto, percebeu-se que utilizar apenas uma dimensão acarretava em muita perda de informação o que fez com que outras cores diferentes das selecionadas fossem interpretadas como as cores desejadas, como por exemplo: cor de pele e cores marrons foram interpretadas como o vermelho e alguns tons de amarelos e azuis foram interpretadas como o verde. Por este trabalho se tratar de um projeto de automação residencial, não se pode deixar com que a variação de iluminação interfira na detecção das cores. O espaço de cores HSV portanto mostrou-se falho neste quesito.

Um espaço com transformação muito similar ao do HSV que tem propósito de contornar o problema da iluminação é o HSL. Entretanto, realizando o mesmo procedimento utilizado com o HSV, percebeu-se que houve uma melhora com o HSL, mas ela não foi tão significativa a ponto de solucionar o problema.

Portanto foi necessário encontrar um novo espaço de cores que pudesse tratar o problema causado pela variação de iluminação. Para isso, comparou-se os espaços de cores variando a iluminação entre alguns dos espaços que o OpenCV é capaz de transformar a partir do RGB (com exceção do espaço de tons de cinza): YCrCb, CIE XYZ com ponto branco (iluminação) em D65, CIE LAB e CIE LUV. Desta forma, determinou-se que o espaço CIE LAB sofreu menor influência com a variação de iluminação.

A Figura 4.1 compara os resultados do CIELAB com o HSV ao extrair as cores vermelho e verde da vareta variando a iluminação entre escura e clara. Para obter estes resultados, utilizou-se 600 quadros de vídeo (20 segundos de filmagem), pois mesmo com a imagem estática (nenhum objeto movendo-se nela), pode-se notar variação nas cores

capturadas. Então criou-se uma região de interesse na imagem para selecionar somente os pixeis correspondentes às cores examinadas. Para cada pixel desta região de interesse, armazenou-se o seu valor máximo e o valor mínimo que ele obteve ao longo dos 600 quadros (valores máximos e mínimos das seis coordenadas: *H*, *S*, *V*, *L*, *a*, *b* para cada pixel da região de interesse). Desta forma, para cada cor (verde e vermelho), criou-se doze listas que foram ordenadas de forma crescente com os valores máximos e mínimos dos seis níveis (*H*, *S*, *V*, *L*, *a*, *b*). Reparou-se que o menor valor da lista de valores mínimos e o maior valor da lista de máximos geralmente estavam muito fora dos valores esperados de ser obtidos. Então, para o valor mínimo, selecionou-se o valor que estava na posição da lista a 10% do início dela (por exemplo, se a lista possui 1000 valores ordenados de forma crescente, selecionou-se o valor da posição 100). Já para o valor máximo, selecionou-se o valor que estava na posição 90% do início da lista. Desta forma, obteve-se valores mais de acordo com o esperado.

A primeira informação que pode-se obter destes gráficos é que para o HSV e para uma iluminação constante, seja ela clara ou escura, os níveis *H* e *S* estão em faixas estreitas. Ou seja, poderia utilizar os níveis *H* e *S* para extrair estas cores (somente quando a iluminação é constante!). Já o nível *V* possui faixas mais largas (especialmente para ambientes mais iluminados), o que não é tão útil e portanto pode até ser desconsiderado na hora de extrair. Agora, quando a iluminação varia de escuro para claro, a saturação cai consideravelmente e o valor sobe consideravelmente. Desta forma, quando há variação de iluminação, somente o nível *H* possui uma boa informação, mantendo-se em uma faixa mais constante. Mesmo que não se descarte os outros níveis, como eles se encontram em faixas muito largas, a perda de informação é muito grande e outras cores indesejadas são extraidas junto com as selecionadas.

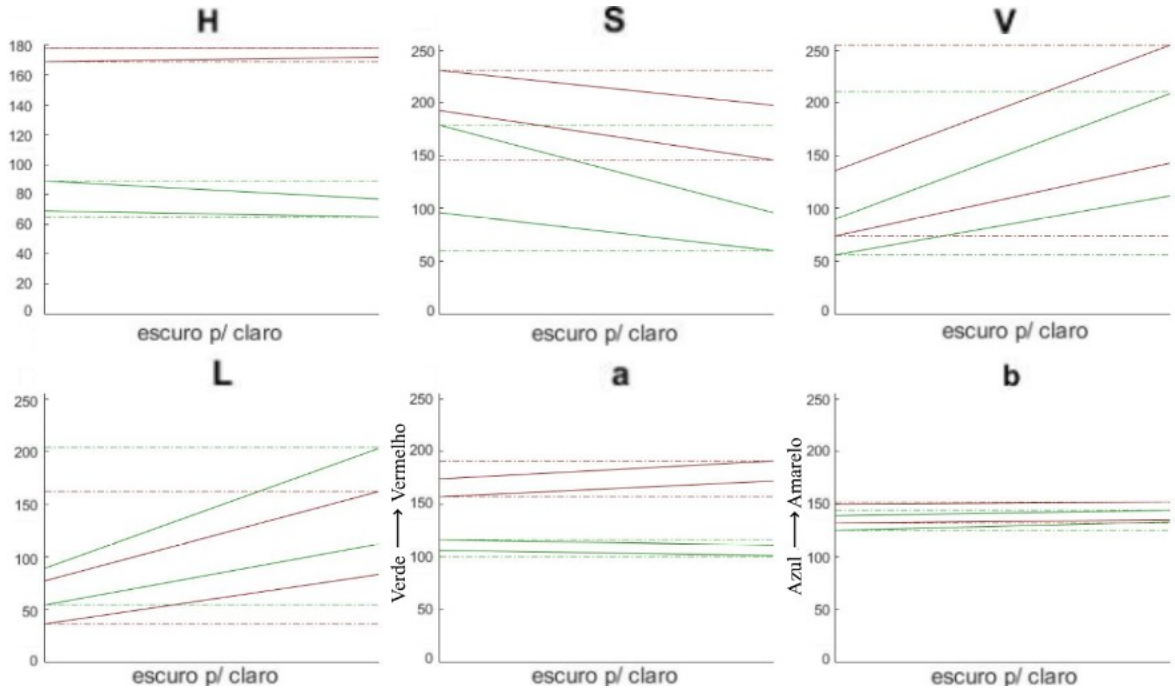

Figura 4.1 – Comparação dos espaços de cores HSV e CIE LAB.

Observação: Existe uma falha que foi notada nos valores máximos de *H* para a cor vermelha, pois o vermelho é representado por duas faixas de valores esperados: entre ~165 e 180 e entre 0 e ~15. Portanto, os valores "máximos" para a cor vermelha de fato estão em torno de 15 e não de 180. Mesmo assim, este erro não atrapalhará no desenvolvimento da análise.

Agora, para o CIE LAB, as faixas *a* e *b* se mantiveram estreitas e quase constantes e somente a faixa *L* variou com a variação de iluminação, como esperado. Assim, descartando-se o nível *L*, a perda de informação é minimizada se comparado ao HSV. No entanto, o CIE LAB também não é perfeito. Para as cores selecionadas da primeira vareta feita (vermelho e verde) também foram extraidas outras cores indesejadas junto a elas (no entanto muito menos do que utilizando o HSV). Para o vermelho, veio também um pouco de marrom e para o verde, veio também um pouco de amarelo, como pode-se ver na Figura 4.2**,** onde o vermelho da vareta se encontrava no canto inferior direito (mais de fora do círculo) e o verde no canto inferior esquerdo (dentro do círculo).

Para se combater este problema, existe uma outra transformação para o CIE LCH, que corresponde ao mesmo espaço de cores que o CIE LAB, mas tratado em coordenadas cilíndricas e não em coordenadas retangulares. Onde *C* corresponde ao *croma* (coordenada radial, equivalente a saturação do HSV) e o *H* ao "*hue*" / tonalidade (como no HSV). No entanto, a biblioteca utilizada neste trabalho não realiza transformação para o CIE LCH, somente para o CIE LAB. Então, por este motivo criou-se um programa que gerava o plano AB e que permitia variar o *L* (utilizado para gerar a Figura 4.2) e verificou-se visualmente que em seções circulares, as tonalidades variam menos do que em seções retangulares, uma vez que, como tratado anteriormente por [23], o CIE LAB é um círculo de cores do terceiro tipo (que mantém mais opostas as cores de tipo oposto: azul-amarelo, vermelho-verde). Ainda assim, percebeu-se que há tons que variam menos do que outros dentro de seções circulares. Deste modo, procurou-se novas cores para se construir uma nova vareta.

Como o CIE LAB está em coordenadas retangulares, para se utilizar da propriedade do CIE LCH, de menor influência de outros tons em seções circulares, decidiu-se que as novas cores da vareta deveriam se encontrar em seções retangulares que se aproximam de seções circulares. Ou seja, para os ângulos (*H*) em 0º, 90º, 180º ou 270º. Portanto, as cores possíveis seriam: azul (parte de cima dos gráficos da Figura 4.2), rosa (à direita), amarelo (em baixo) ou "verde-água" (à esquerda). Percebeu-se também que variando o *L*, as cores menos influenciadas por outros tons eram a azul e a rosa. Portanto, criou-se uma segunda vareta com estas cores.

Uma observação: a vareta anterior, verde e vermelha, foi feita com tinta acrílica fosca, pois foi a melhor forma de se manter um tom constante. Tentou-se realizar uma impressão colorida e utilizar lápis e giz colorido e não tiveram um tom constante. Para a vareta azul e rosa, não se conseguiu atingir exatamente os tons desejados com tinta, no entanto, preferiu-se utilizá-la assim do que ter uma vareta com tons inconstantes.

Assim, criou-se a vareta azul e rosa que teve resultados muito satisfatórios com a questão de iluminação e que não se confundiu muito com outras cores (muito menos do que para os outros espaços de cores). Mesmo assim, ainda que se possa ter tons que se confundam, como foi visto na seção 3.2.3 ("Confecção e Detecção da Vareta"): o fato de se ter duas cores e se procurar pela interseção entre elas faz com que muitas interferências desapareçam.

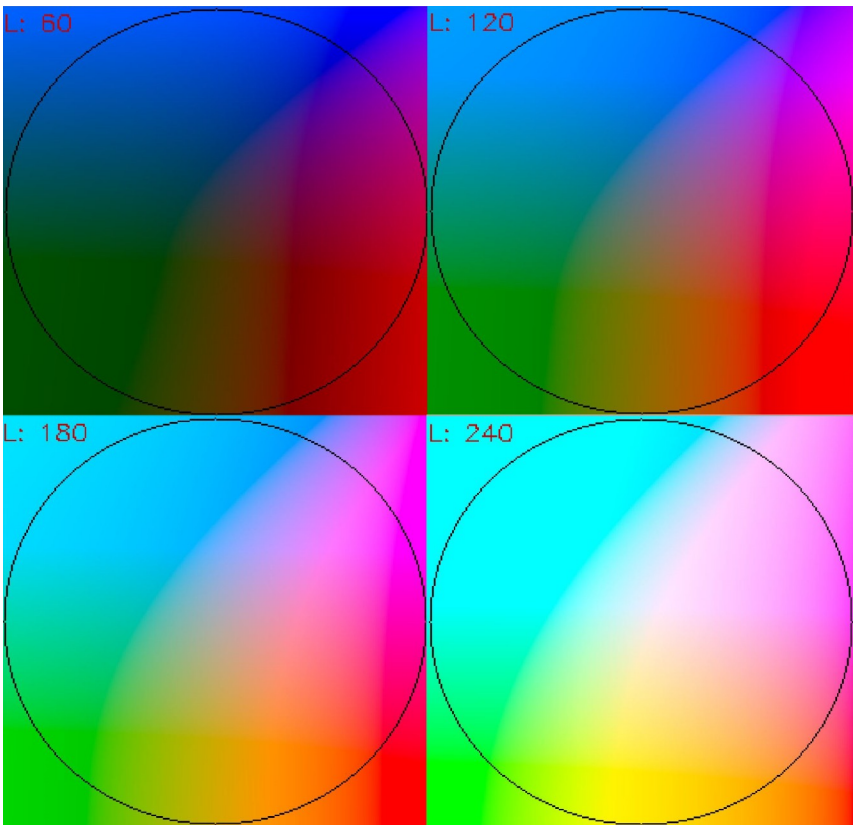

Figura 4.2 – CIE LAB para algumas faixas de iluminação. (Eixo A da esquerda para direita, eixo B de cima para baixo e origem no centro).

4.2 Campo de visão das câmeras

Como já dito no Capítulo anterior os suportes de câmera projetados em um programa de CAD utilizaram o cálculo de calibração das câmeras para descobrir o ângulo do campo de visão. O anexo A contém o código em *Matlab* para realizar estes cálculos, bem como os valores obtidos das matrizes intrínsecas das duas câmeras. Nele também foi colocado na forma de comentários o valor retornado dos ângulos calculados. O campo de visão calculado da câmera Logitech, ± 28º, é maior do que o da câmera Genius, ± 22º. Além destes ângulos (da direção x), também calculou-se os na direção y, para poder inclinar as câmeras para baixo. Foi obtido 15º para Logitech e 12º para a Genius. Desta forma inclinou a Logitech em 17º e a Genius em 14º, para mirar um pouco mais para baixo do que o nível do teto. Assim como o previsto, o campo útil de visão das câmeras (interseção entre seus campos de visão) foi uma área considerável, como poderá ser visto nas próximas seções, a partir da Figura 4.22.

Além dos ângulos do campo de visão, a forma de fixação das câmeras são diferentes, assim, para a criação dos suportes, utilizou-se um paquímetro para se criar um suporte bem justos, com precisão de 0,1 milímetro. Desta forma é possível fixar as câmeras e restringir suas articulações na orientação desejada e também é possível removê-las e depois recolocá-las na mesma posição e orientação. Desta forma, não se prejudica no mapeamento dos alvos remover e recolocar as câmeras em seus suportes. Imprimiu-se estes suportes com uma densidade de 30% para que fossem bem resistentes, contudo, suas paredes ficaram um pouco finas e não é dá receio em ficar forçando muito.

4.3 Mapeando Alvos

A Figura 4.3 ilustra as seis imagens obtidas para mapear o alvo 1 da lâmpada 1. As próximas 16 Figuras apresentam o resultado das retas mapeadas para todos os alvos (três por lâmpada e dois do ar-condicionado) para as duas câmeras (Figuras de 4.4 a 4.19). Cada reta de cada uma destas Figuras foi obtida por uma imagem diferente, alterando a posição do barbante, como exemplificado pela Figura 4.3. Os pontos vermelhos são os pontos extraídos das imagens para formar as retas. Como as coordenadas de câmera possuem o eixo *y* de cabeça para baixo, estas Figuras possuem o eixo y invertido, pois o *Matlab* plota com o eixo *y* apontando para cima. Então as interseções das retas para as lâmpadas e o ar-condicionado estarão na região inferior da imagem. Como as câmeras não os enxerga, estes pontos serão mapeados para fora das imagens, mas isso não é um problema, pois pode-se estender o plano de câmera para limites fora de seu campo de visão. Desenhou-se um retângulo indicando a região deste plano correspondente ao campo de visão das câmeras.

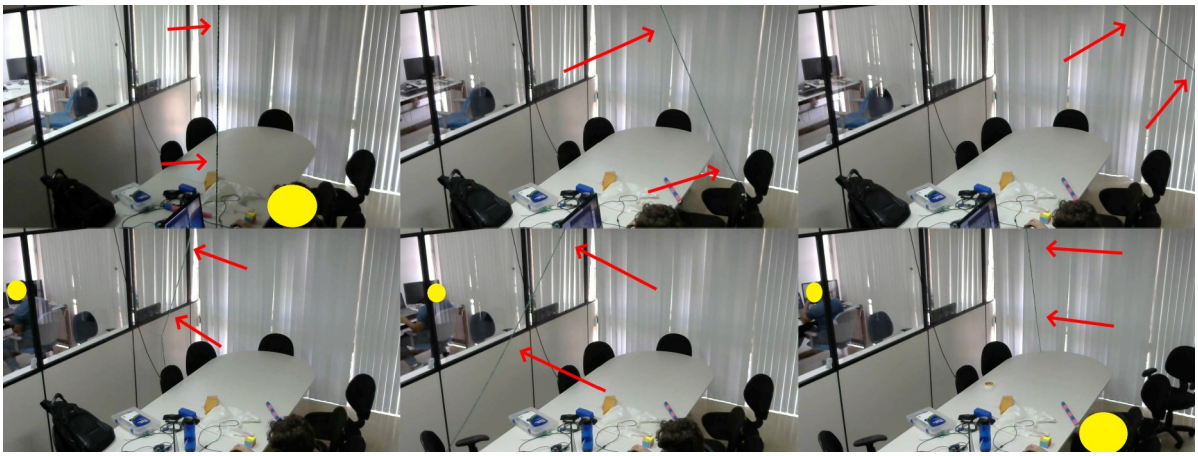

Figura 4.3 –Posicionando o barbante para mapear o alvo 1 da câmera 1.

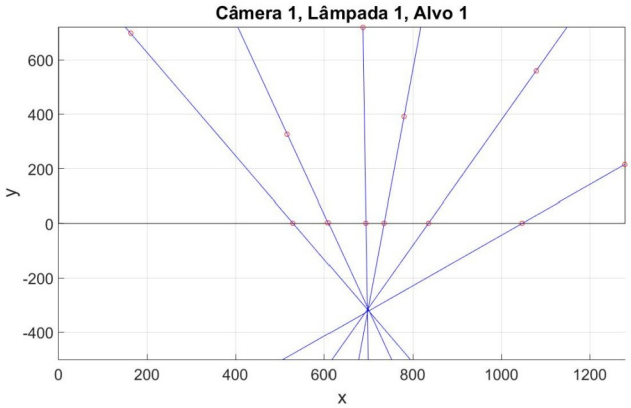

Figura 4.4 – Câmera 1, Lâmpada 1, Alvo 1.

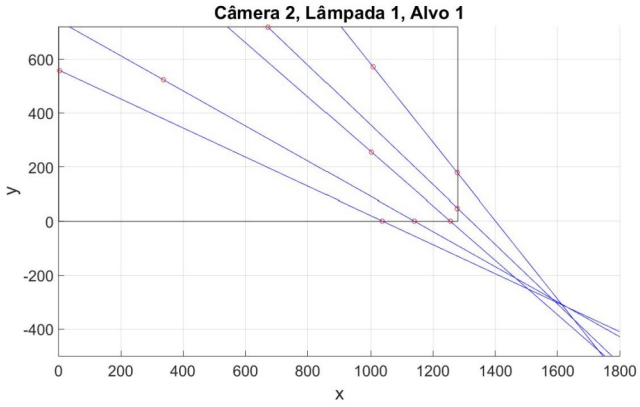

Figura 4.5 – Câmera 2, Lâmpada 1, Alvo 1.

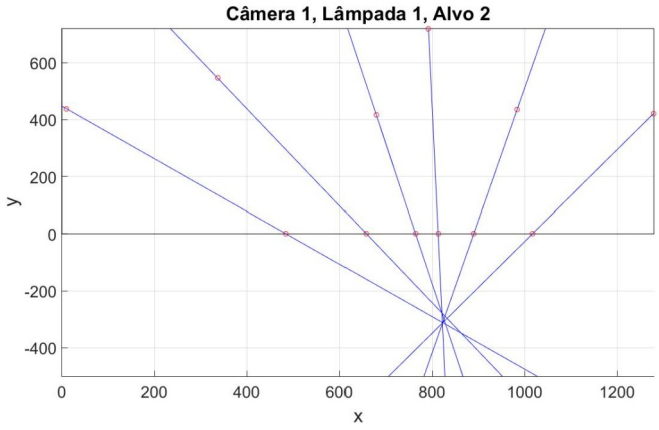

Figura 4.6 – Câmera 1, Lâmpada 1, Alvo 2.

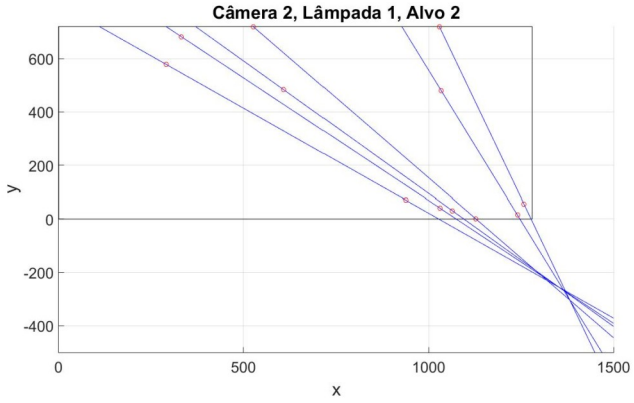

Figura 4.7 – Câmera 2, Lâmpada 1, Alvo 2.

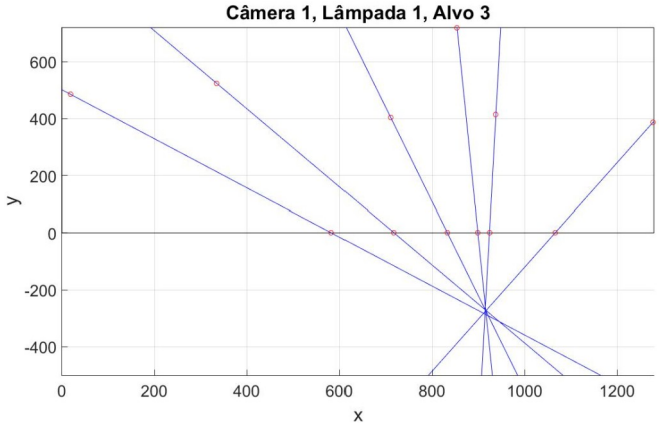

Figura 4.8 – Câmera 1, Lâmpada 1, Alvo 3.

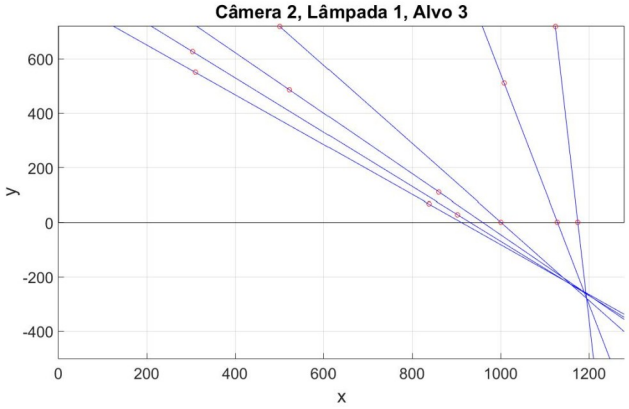

Figura 4.9 – Câmera 2, Lâmpada 1, Alvo 3.

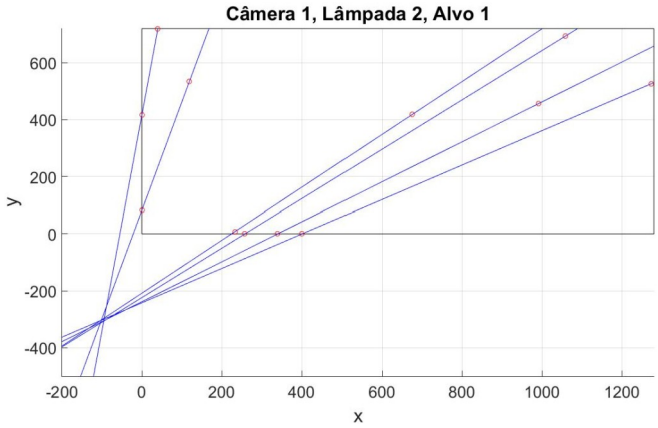

Figura 4.10 – Câmera 1, Lâmpada 2, Alvo 1.

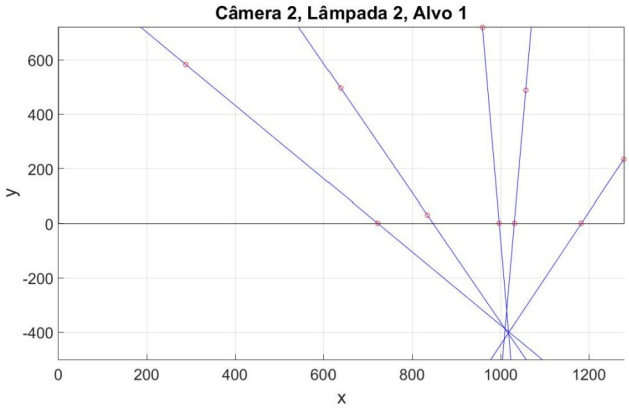

Figura 4.11 – Câmera 2, Lâmpada 2, Alvo 1.

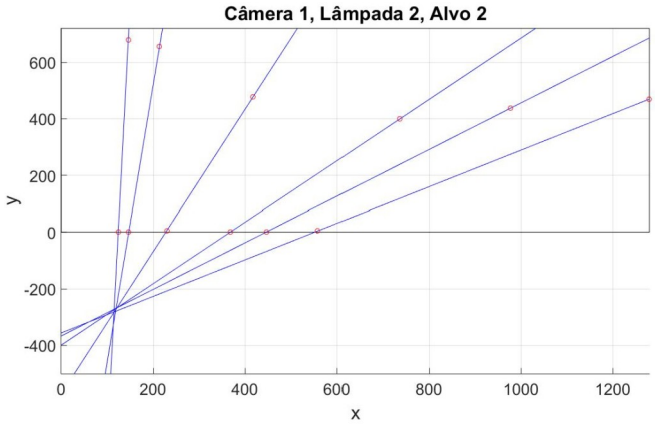

Figura 4.12 – Câmera 1, Lâmpada 2, Alvo 2.

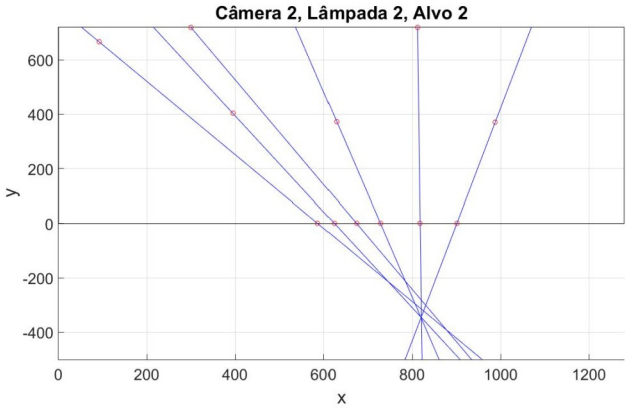

Figura 4.13 – Câmera 2, Lâmpada 2, Alvo 2.

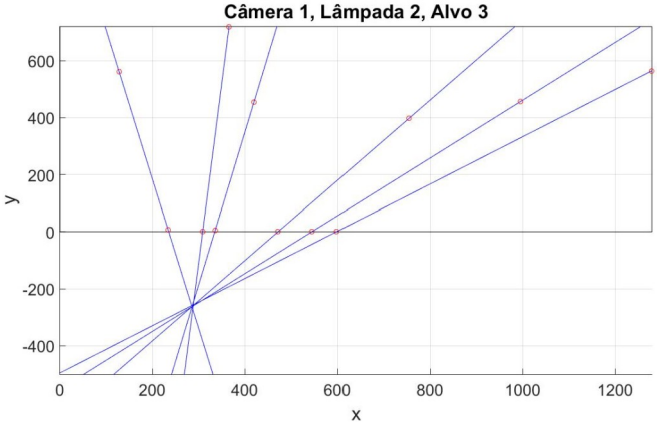

Figura 4.14 – Câmera 1, Lâmpada 2, Alvo 3.

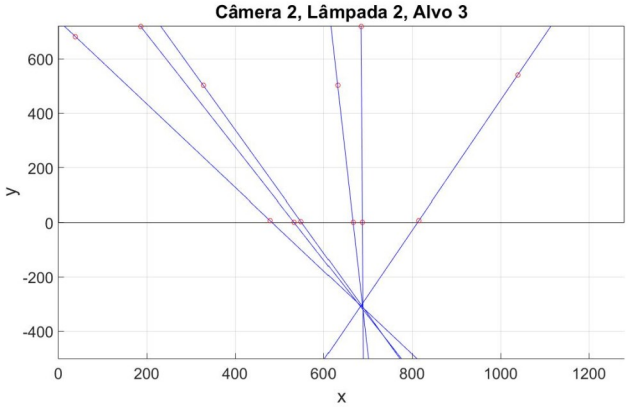

Figura 4.15 – Câmera 2, Lâmpada 2, Alvo 3.

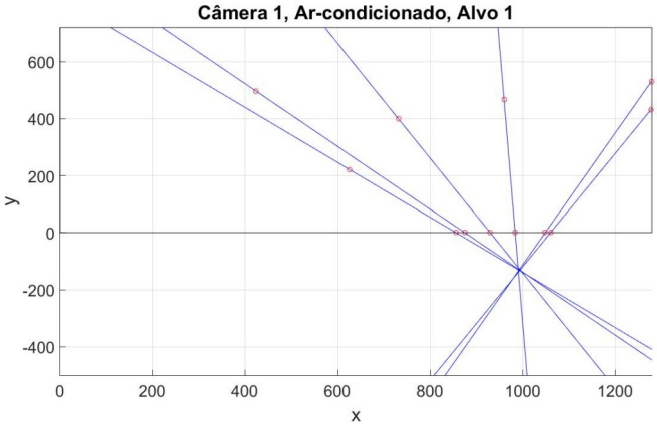

Figura 4.16 – Câmera 1, Ar-condicionado, Alvo 1.

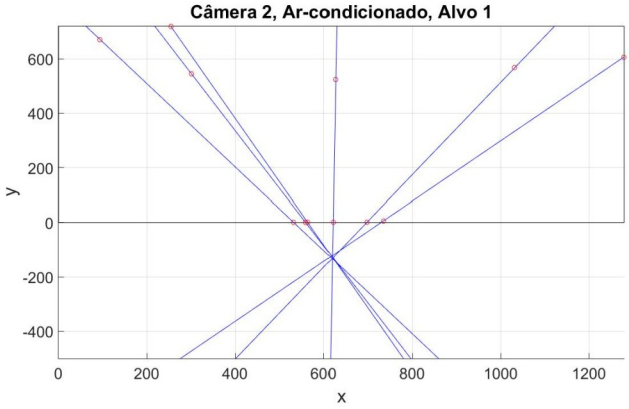

Figura 4.17 – Câmera 2, Ar-condicionado, Alvo 1.

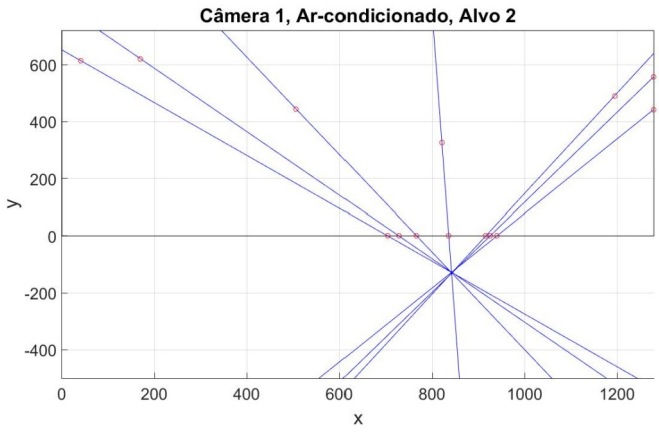

Figura 4.18 – Câmera 1, Ar-condicionado, Alvo 2.

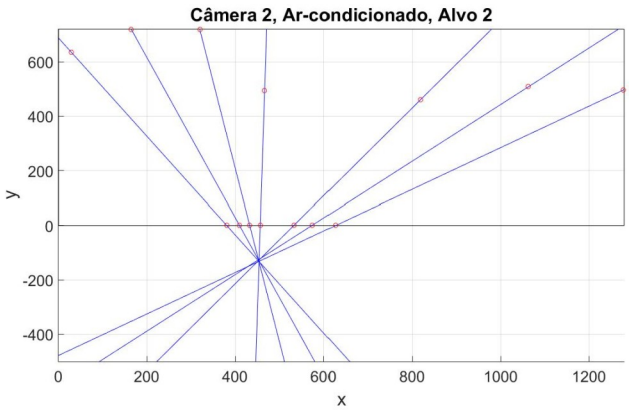

Figura 4.19 – Câmera 2, Ar-condicionado, Alvo 2.

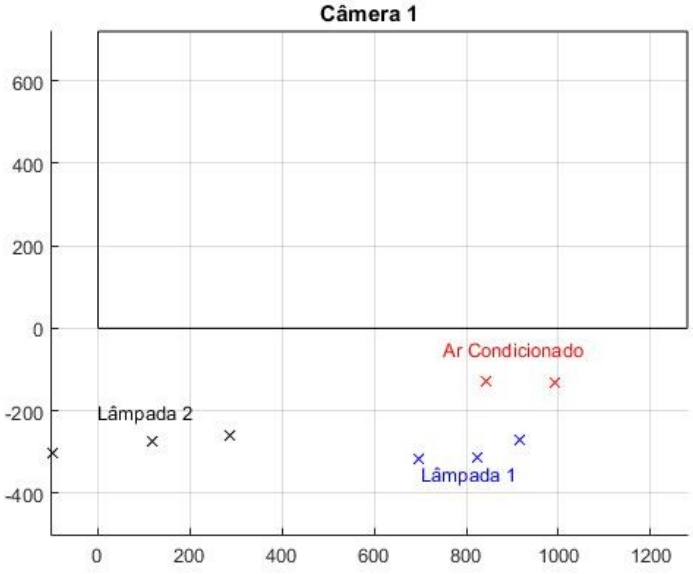

Figura 4.20 – Alvos da Câmera 1.

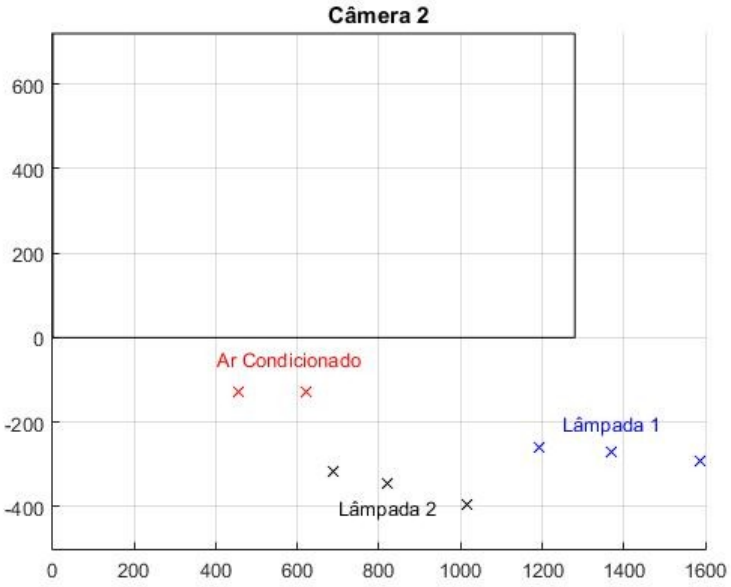

Figura 4.21 – Alvos da Câmera 2.

Repare, das Figuras 4.4 a 4.19, que algumas das retas ficaram muito fora do ponto de interseção. Então para estes casos, removeu-se estas retas do cálculo do ponto de interseção. Com isso, as Figuras 4.20 e 4.21 ilustram os pontos de interseção de todos os alvos mapeados nos planos das duas câmeras. Com estes dados, pode-se extrair que a utilização de apenas uma câmera seria um problema, visto que os alvos da lâmpada 1 estão muito próximos aos do ar-condicionado para o plano da câmera 1, mas estão bem distantes para o plano da câmera 2. Isso indica que se só houvesse a câmera 1, não seria possível distinguir entre qual deles foi apontado.

4.4 Testando Cenários

Agora que os alvos foram mapeados, testou-se o primeiro cenário, descrito no Capítulo 3 seção 3.3.3 Nele fixou-se o barbante nas interseções dos alvos e passou ele por dentro da vareta, como ilustra a Figura 4.22. Então variando a direção em que a vareta aponta, buscou-se os pontos em que ela falharia. Um primeiro ponto de falha foi com a vareta muito inclinada, como ilustra a Figura 4.23. Nela, uma câmera conseguiu detectar a vareta, enquanto a outra não foi capaz. Outra falha foi quando posicionou-se a vareta contra uma brecha de luz saindo das cortinas, ilustrado pela Figura 4.24. Neste caso, também não foi possível detectar a vareta.

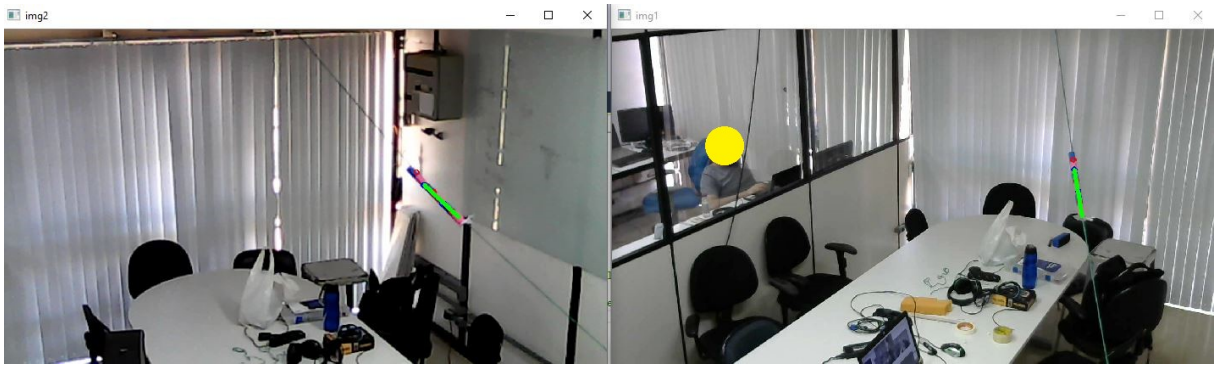

Figura 4.22 – Fixação da vareta no barbante e direção apontada por ela.

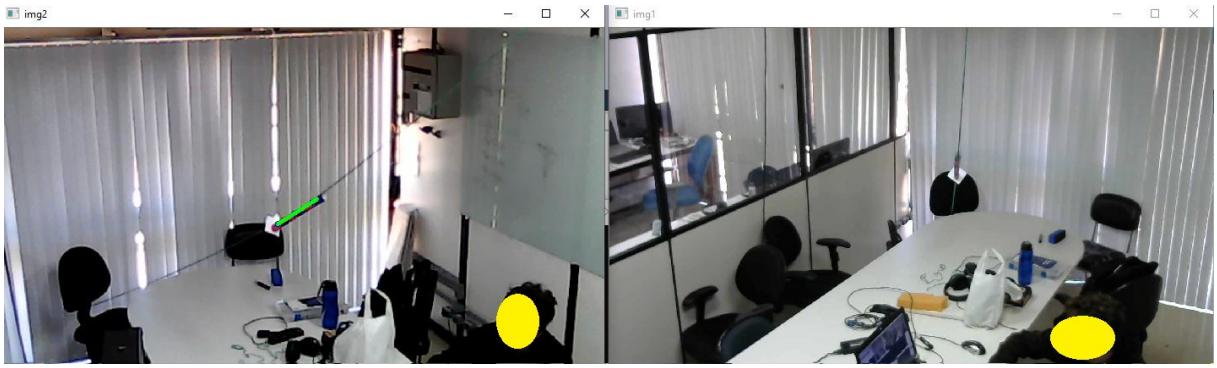

Figura 4.23 – Falha: vareta muito inclinada.

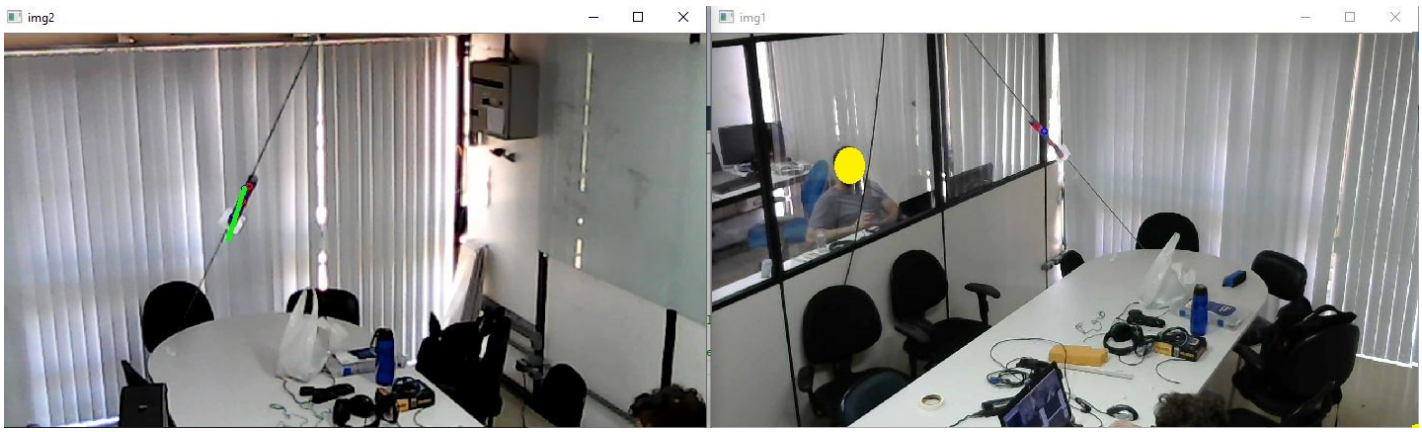

Figura 4.24 – Falha: vareta contra brecha de luz.

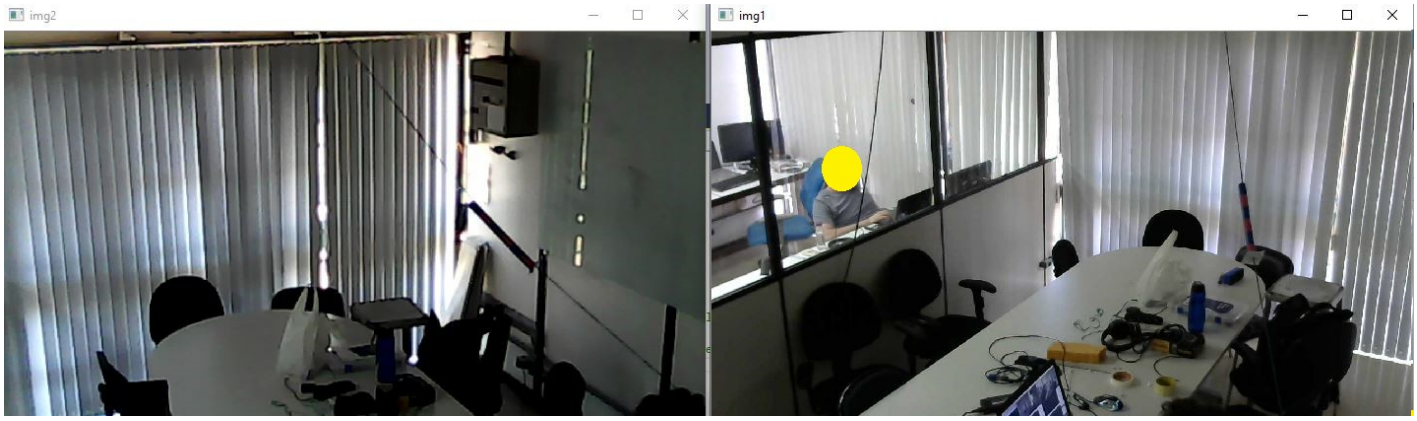

Figura 4.25 – Ambiente muito escuro.

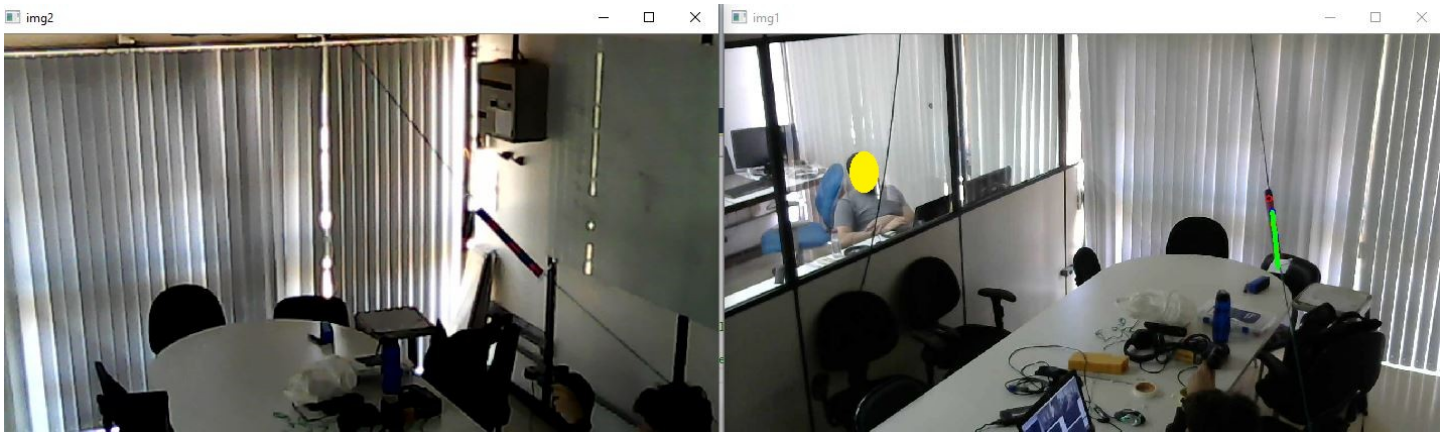

Figura 4.26 – Ambiente muito escuro com lanterna no mínimo.

O segundo cenário foi colocando o ambiente muito escuro, como ilustra a Figura 4.25. De perto, percebeu-se que a cor rosa da vareta conseguia ser extraida. No entanto, a cor azul escolhida para pintar a vareta acabou ficando muito escura. Ela era parcialmente extraida, mas sombras do ambiente eram extraidas junto a ela. De longe, foi pior para o rosa, onde nem todos os pixeis necessários foram extraidos. Portanto, criou-se um terceiro cenário, utilizando uma lanterna com regulação de intensidade luminosa. Com ela no máximo, foi possível enxergar a vareta e funcionou perfeitamente, como no primeiro cenário. Já com ela no mínimo funcionou mas com falhas, como ilustra a Figura 4.26.

O quarto cenário abriu-se as cortinas e expôs as câmeras contra a luz solar. Este foi o pior cenário, visto que a maior parte das imagens ficou ofuscada pela interferência da luz. Não obteve-se sucesso em apontar para os alvos com a vareta neste cenário.

O quinto cenário foi bem interessante. Apontou com a vareta para pontos fora dos alvos dos objetos controláveis e em seguida foi movendo-a em direção ao objeto que se deseja controlar. Para o ar-condicionado, escolheu-se um ponto mais próximo da janela, assim seria também um ponto afastado das duas lâmpadas também. Para as lâmpadas, escolheu-se pontos na direção oposta da outra lâmpada (que não se quer controlar) e que também são bem afastados do ar-condicionado. Pela rotina ilustrada pela Figura 3.29, uma das características do programa é a de continuar procurando pela vareta enquanto não foi identificado o objeto apontado após N tentativas. Escolheu-se uma quantidade razoável, que não fosse tão pequena, mas nem tão grande também, para que possibilitasse este movimento. Na maioria das tentativas o tempo escolhido foi bom e o objeto apontado foi identificado. No entanto, algumas poucas vezes o tempo foi pouco e não se identificou o objeto.

O sexto cenário foi o mais complexo deles. Nele, fixou-se a vareta em pontos arbitrários com o barbante, bem como a utilizou de forma livre. Os pontos arbitrários selecionados foram fora dos alvos. Na maioria deles, a vareta não identificou os alvos. Mas a complexidade foi que também dependendo da orientação e distância da vareta a estes pontos, de vez em quando acusava-se que a vareta apontava para um dos alvos cadastrados. Percebeu-se que quando a vareta estava mais afastada ou mais inclinada, estes falsos positivos ocorriam. De fato, não são falhas, visto que existem as margens de tolerância, que formam no espaço tridimensional um feixe na forma de pirâmide de base losanga. Quanto mais distante a vareta, maior fica a área deste feixe e portanto maior são as chances dele interceptar um alvo. A respeito da inclinação da vareta, os alvos acusados por falso positivo estavam voltados para direção desta inclinação, o que leva a entender que estes falsos positivos foram causados pelas margens de tolerância. Um caso especial foi fixar em um ponto entre todos os alvos (quase equidistante a eles). Na maioria das orientações da vareta, nenhum alvo foi identificado, somente foi quando inclinou a vareta na direção do ar-condicionado. Neste caso, obteve-se um falso positivo.

Dentro do sexto cenário, também testou-se um cenário mais prático, mais próximo de um cenário real, com a vareta livre (sem o barbante). Apontou-se para os alvos de forma natural e obteve-se sucesso na maioria dos casos. Percebeu-se que quando se está posicionado de baixo do alvo ou nas proximidades desta posição, é mais fácil de se apontar. Já quando se está mais distante desta posição (de baixo do alvo), a tendência é de se acreditar apontar para o alvo, enquanto de fato se está apontando mais para cima dele, criando falsos negativos eventuais, como representado pela Figura 3.5. Assim, percebeu-se que é melhor apontar de forma que a linha formada por seu campo de visão (entre seu olho e o alvo) fique próxima e quase paralela com a linha da vareta. Uma das formas é olhando por dentro da vareta (por ela ser um objeto cilíndrico oco). Outro modo é como se estivesse apontando com um revólver, olhando tangenciando as paredes da vareta.

Por fim, realizou-se um teste quantitativo onde apontou-se para os alvos 102 vezes em diferentes posições com as luzes acesas. Destas medidas, apenas uma foi um falso negativo (não encontrou a vareta) devido a vareta estar posicionada contra uma brecha de luz. As demais foram verdadeiros positivos (encontrou a vareta e o objeto apontado como gostaria). Não houveram falsos positivos (de encontrar a vareta quando não devia). Para as 101 medidas verdadeiro positivo, mediu-se a quantidade de tentativas que o programa levou para identificar a vareta nas duas imagens simultaneamente e para identificar o objeto apontado por elas. A Tabela 4.1 indica a quantidade de tentativas que foram necessárias para identificar a vareta. Já a Tabela 4.2 indica a quantidade de tentativas para identificar o objeto apontado.

Tabela 4.1 – Quantidade de tentativas para identificar a vareta

Quantidade de Tentativas	Quantidade de Ocorrências
01	85
02	4
03	6
05	1
07	1
08	1
09	1
10	1
12	1

Tabela 4.2 – Quantidade de tentativas para identificar o objeto apontado.

Quantidade de Tentativas	Quantidade de Ocorrências
01	88
02	5
03	3
04	3
05	1
12	1

Por meio das Tabelas 4.1 e 4.2 pode-se determinar que em média encontrou-se a vareta após 1,604 tentativas, com um desvio padrão de 1,883 tentativas e encontrou o objeto apontado em média após 1,347 tentativas, com um desvio padrão de 1,293 tentativas. De todas estas tentativas, acertou-se todos os objetos apontados (com exceção da tentativa que deu falso negativo). Neste projeto utilizou-se 20 tentativas antes do programa desistir, o que foi mais do que suficiente para todos os casos, visto que o máximo de tentativas necessárias foi de 12, tanto para identificar a vareta como para identificar o objeto apontado.

Capítulo 5

Conclusão

Este trabalho em princípio buscou desenvolver uma interface mais amigável e intuitiva com o usuário. Dentre as possíveis soluções, percebeu-se que uma interface por gesto de apontamento é muito amigável e intuitiva. Como o custo computacional é grande com cálculos heurísticos para identificar as mãos das pessoas e como tais algoritmos estão sujeitos a maiores possibilidades de falhas (falsos positivos ou negativos), seguiu um caminho o mais próximo deste substituindo o apontamento com as mãos por um com uma vareta com padrões visuais bem definidos, mas mantendo a iluminação passiva da vareta.

Dentre as principais dificuldades deste trabalho, a iluminação foi uma das maiores, onde foi resolvida utilizando-se um espaço de cores mais tolerante a variações de iluminação, o CIE LAB. Percebeu-se contudo que ele também não é perfeito e que os tons mais próximos dos eixos *A* e *B,* de valores de maior *croma* e tons mais claros são melhores identificados dado variação de iluminação.

A solução de deslocar uma máscara sobre a outra e realizar a interseção entre elas para encontrar as regiões de interseção entre as cores da vareta permitiu com que se obtivesse um método rápido de se remover quase todas as interferências da imagem e ruídos. Para ruídos remanescentes, a erosão ajudou a removê-los. Assim, percebeu-se que um padrão de cores funcionou melhor que padrões morfológicos para encontrar a vareta. No entanto, os padrões morfológicos dela (cores intercaladas em seis faixas) permitiram distinguir a vareta de outros objetos que possam ser extraidos junto a ela, evitando falsos positivos.

Neste trabalho, percebeu-se também que não era necessário a realização de reconstrução 3D, pois foi possível realizar a identificação dos objetos controláveis apontados por meio das coordenadas de câmera. Para isso, foi necessário identificar a melhor posição para instalar as câmeras e determinou-se que elas deveriam ficar o mais afastadas e ortogonais possível. Contudo, a ortogonalidade delas reduzia a área útil da interseção dos campos de visão delas. Portanto, deve-se também reduzir a ortogonalidade entre elas a fim de se maximizar a área útil do campo de visão.

Os testes realizados foram bem satisfatórios. Houve problemas dependendo da orientação da vareta, iluminações contrárias às câmeras e de ambiente muito escuro. Também houve alguns falsos positivos devido ao feixe das margens de tolerância da vareta. No mais, o projeto operou como esperado e desejado.

5.1 Modificações e Trabalhos posteriores

Uma modificação que pode ser feita está nas cores da vareta. Foi um grande trabalho ajustá-las de forma a colocá-las sobre os eixos *A* ou *B* (do CIE LAB) para se aproximar do CIE LCH. Ainda assim, a cor azul ficou escura, o que atrapalhou em um ambiente muito escuro. A cor rosa ficou muito boa e funcionou bem para variações de iluminação. Portanto, é necessário testar outros tons mais claros para substituir este azul, que funcionou para variações de iluminação, mas não em um ambiente muito escuro. No entanto, ele não pode ser claro demais pois uma das varetas

feitas anteriormente tinha um tom verde azulado bem claro e com baixa saturação. Este tom foi um dos piores obtidos, pois se confundia com cinzas e sombras do ambiente. Percebeu-se nesta tentativa que deveria evitar tons muito próximos da origem do CIE LAB (valores baixos de *croma*), para evitar interferências com sombras e tons de cinza.

Trabalhos posteriores podem seguir duas linhas: a primeira é em se continuar trabalhando com varetas com padrões visuais baseados no desenvolvido deste trabalho ou buscar trabalhar com as mãos livres utilizando algum algoritmo heurístico para identificar as mãos das pessoas em um ambiente. Se continuar trabalhando com varetas, recomenta-se adaptar a rotina apresentada pela Figura 3.29. Já se for trabalhar com as mãos livres, recomenda-se adaptar o algoritmo apresentado pela Figura 3.28.

Sugere-se algumas ideias de trabalhos futuros: a primeira é em seu utilizar mais de uma vareta em um ambiente, precisando substituir o botão por alguma outra forma de comando. Uma segunda ideia está em se adaptar a rotina que permite identificar a vareta em movimento para mover objetos no espaço (como, por exemplo, mover a posição de uma cortina ou uma ponte rolante em um ambiente industrial). Outra opção é em se adicionar mais botões ou outras formas de comando para a vareta comandar objetos mais complexos, como, por exemplo, o controle da intensidade luminosa por um *dimmer*, temperatura do ar-condicionado, volume de uma música ou então tom de cor de uma lâmpada *Led* RGB.

Para a primeira ideia, sugere-se utilizar um mecanismo para indicar as varetas, como de *TAGs* visuais e associá-los a um botão com sua respectiva codificação. Ou então, pode-se associar um movimento da vareta ou de um segundo objeto para indicar o comando. Para a segunda ideia sugere-se adaptar o botão para casos de “clicar e segurar”, assim, enquanto o botão estiver pressionado, deverá continuar com o movimento. Já para a terceira ideia, sugere-se adicionar mais botões (forma mais fácil de implementar, contudo mais difícil de se adaptar para controle com as mãos livres) ou associar um movimento ou outro objeto para efetuar este controle.

Referencias Bibliográficas

[1] **Anna Adami.** "Domótica". Disponível em: https://www.infoescola.com/tecnologia/domotica/. [Consultado em: 13/06/2019].

[2] **Li, Rita Yi Man; Li, Herru Ching Yu; Mak, Cho Kei; Tang, Tony Beiqi (2016)**. "Sustainable Smart Home and Home Automation: Big Data Analytics Approach". International Journal of Smart Home. Vol. 10. No. 8: pp. 177–198. DOI: 10.14257/ijsh.2016.10.8.18.

[3] **Amazon**. "Amazon Echo - Now Available". Disponível em: https://www.youtube.com/watch?v=FQn6aFQ_wBQU. [Consultado em: 13/06/2019].

[4] **Google**. "Google Assistente". Disponível em: https://play.google.com/store/apps/details?id=com.google.android.apps.googleassistant&hl=pt_BR. [Consultado em: 13/06/2019].

[5] **Asus** "Zenbo". Disponível em: https://zenbo.asus.com/. [Consultado em: 13/06/2016].

[6] **Robo Realm**. "WowWee Rovio". Disponível em: http://www.roborealm.com/help/WowWee_Rovio.php. [Consultado em: 13/06/2019].

[7] **Dong-Luong Dinh; Jeong Tai Kim; Tae-SeongKim**. "Hand Gesture Recognition and Interface via a Depth Imaging Sensor for Smart Home Appliances". Energy Procedia. Volume 62, 2014, Paginas: 576-582. DOI: 10.1016/j.egypro.2014.12.419.

[8] **Marcin Denkowski; Krzysztof Dmitruk; Łukasz Sadkowski**. "Building Automation Control System driven by Gestures". IFAC-PapersOnLine 48-4 (2015) 246–251. DOI: 10.1016/j.ifacol.2015.07.041.

[9] **BMW USA**. "Gesture Controls | BMW Genius How-To". Disponível em: https://www.youtube.com/watch?v=wqvAPskg_k0. [Consultado em: 20/05/2019].

[10] **Andrea Sanna, Fabrizio Lamberti, Gianluca Paravati, Federico Manuri**. "A Kinect-based natural interface for quadrotor control". Entertainment Computing Volume 4, No. 3, Agosto de 2013, Páginas 179-186. DOI: 10.1016/j.entcom.2013.01.001.

[11] **Brett Dvoretz**. "The 10 Best Gesture Controlled Drones". Disponível em:https://wiki.ezvid.com/best-gesture-controlled-drones. [Consultado em: 20/05/2019].

[12] **Nur Safwati Mohd Nor; Ngo Lam Trung; Yoshio Maeda; Makoto Mizukawa**. "Tracking and Detection of Pointing Gesture in 3D Space". 2012 9th International Conference on Ubiquitous Robots and Ambient Intelligence (URAI). DOI: 10.1109/URAI.2012.6462983.

[13] **Yuhui Lai; Chen Wang; Yanan Li; Shuzhi Sam Ge; Deqing Huang**. "3D pointing gesture recognition for human-robot interaction". 2016 Chinese Control and Decision Conference (CCDC). DOI: 10.1109/CCDC.2016.7531881.

[14] Disponível em: https://www.quora.com/The-CMOS-camera-sensor-is-one-of-the-most-common-types-in-digital-cameras-Whats-another-camera-sensor-and-how-well-does-it-compare-to-CMOS. [Consultado em 2018].

[15] **Câmera Neon**. "Distorção Radial em Fotografias". Disponível em: http://www.cameraneon.com/tecnicas/distorcao-radial-em-fotografias/ [Consultado em: 02/07/2019].

[16] **Jason P. de Villiers; F. Wilhelm Leuschner; Ronelle Geldenhuys**. "Centi-pixel accurate real-time inverse distortion correction". Disponível em: http://researchspace.csir.co.za/dspace/bitstream/handle/10204/3168/De%20Villiers_2008.pdf;jsessionid=AFC8E9264616D2F1616150D01E7C7B11?sequence=1. [Consultado em: 13/06/2019].

[17] **OpenCV**. "Morphological Transformations". Disponível em: https://docs.opencv.org/3.0-beta/doc/py_tutorials/py_imgproc/py_morphological_ops/py_morphological_ops.html. Acesso em: 15/06/2019

[18] Disponível em: https://answers.opencv.org/question/74482/what-does-humoments-tell-me/. [Consultado em: 02/07/2019].

[19] Disponível em: https://biology.stackexchange.com/questions/51870/can-red-cone-cells-actually-see-much-red-light. [Consultado em: 02/07/2019].

[20] **Flavio B. Vidal; Victor H. Casanova Alcalde**. "Window-Matching Techniques with Kalman Filtering for an Improved Object Visual Tracking". Proceedings of the IEEE Conference on Automation Science and Engineering, páginas 829 – 834, USA, 2007. DOI: 10.1109/COASE.2007.4341822.

[21] Disponível em: https://en.wikipedia.org/wiki/Fovea_centralis#Angular_size_of_foveal_cones. [Consultado em: 02/07/2019].

[22] **Yishi Weng, Yuning Zhang, Jingyi Cui, Ao Liu, Zhongwen Shen, Xiaohua Li, and Baoping Wang**. "Liquid-crystal-based polarization volume grating applied for full-color waveguide displays". Vol. 43, Issue 23, pp. 5773-5776 (2018). DOI: 10.1364/OL.43.005773.

[23] **Kurt nassau**. "Color for Science, Art and Technology.". Volume 1 primeira edição. Páginas 20 a 22. ISBN: 0 444 89846 8.

[24] **OpenCV**. "Color conversions". Disponível em: https://docs.opencv.org/3.1.0/de/d25/imgproc_color_conversions.Html. [Consultado em: 20/05/2019]

[25] **Cristiano Jacques Miosso Rodrigues; Mirele de Almeida Mencari.; Adolfo Bauchspiess.** "Sistema de Visão Estéreo para Formas Polimétricas". Brasília, Março de 1999.

Anexo A

Código em Matlab para cálculo do campo de visão das câmeras.

```
clear; close; clc;

% Matriz Intrinseca da câmera da Logitech:
matIntr = [ 1.3550573161218874e+03, 0, 6.2969420574894900e+02;
    0, 1.3550573161218874e+03, 3.2769726754546991e+02;
    0, 0, 1];

angX(1) = 180*asin(matIntr(1,3)/matIntr(1,1))/pi;          % 27.69°
angX(2) = 180*asin((1280-matIntr(1,3))/matIntr(1,1))/pi;   % 28.68°
angX(3) = 180*asin(640/matIntr(1,1))/pi;                   % 28.18°
mediaAngX = mean(angX);                                    % 28.19°
angY(1) = 180*asin(matIntr(2,3)/matIntr(2,2))/pi;          % 13.99°
angY(2) = 180*asin((720-matIntr(2,3))/matIntr(2,2))/pi;    % 16,83°
angY(3) = 180*asin(360/matIntr(2,2))/pi;                   % 15,41°
mediaAngY = mean(angY);                                    % 15,41°

%%
clear; close; clc;
% Matriz Intrinseca da câmera Genius:
matIntr = [ 1.7051363413114120e+03, 0., 6.3854940767960034e+02;
    0, 1.7051363413114120e+03, 3.3782862954455231e+02;
    0, 0, 1];

angX(1) = 180*asin(matIntr(1,3)/matIntr(1,1))/pi;          % 21.9925°
angX(2) = 180*asin((1280-matIntr(1,3))/matIntr(1,1))/pi;   % 22.0977°
angX(3) = 180*asin(640/matIntr(1,1))/pi;                   % 22.0451°
mediaAngX = mean(angX);                                    % 22.0451°
angY(1) = 180*asin(matIntr(2,3)/matIntr(2,2))/pi;          % 11.4273°
angY(2) = 180*asin((720-matIntr(2,3))/matIntr(2,2))/pi;    % 12.9517°
angY(3) = 180*asin(360/matIntr(2,2))/pi;                   % 12.1884°
mediaAngY = mean(angY);                                    % 12.1891°
```

Anexo B

Arquivos que compõem o código do programa.

arqInclusao.h

```
#ifndef ARQINCLUSAO
#define ARQINCLUSAO

#define DESLOCAMENTO 10  // O quanto deslocará uma imagem sobre a outra em pixeis

#define EROSAO1 3        // Erosão e dilatação da máscara resultante
#define DILATACAO1 3
#define EROSAO2 5        // Erosão e dilatação da máscara resultante
#define DILATACAO2 5

#define QUANT_MIN_PTS_CONTORNOS 4 // Depois de encontrados contornos, a quantidade minima de pontos
que ele deve ter

#define TOLERANCIA CLUSTERS 20          // Distância em pixeis na direção x ou y que ponto estimado
tem de ponto verdadeiro
#define TOLERANCIA_CLUSTERS_SQRT2 28    // a distância anterior vezes sqrt(2)
#define QUANT_CORES_VARETA 5            // A quantidade de cores que possui a vareta

#define COS_ANG 9848.07753012208 // vezes 10000 // cosseno do ângulo entre vareta e alvo apontado

#define QUANT_FRAMES 20  // Quantidade de frames utilizados para procurar a vareta

#define MEU_FPS 30.0
#define _1_MEU_FPS 1.0/MEU_FPS

// Bibliotécas OpenCV
#include<opencv2/core/core.hpp>
#include<opencv2/highgui/highgui.hpp>
#include<opencv2/imgproc/imgproc.hpp>

// Bibliotecas Cpp
#include<iostream>
#include <fstream>
//#include <vector>
#include <string>
#include<conio.h>
//#include<stdio.h>
#include <time.h>
//#include<stack>
//#include<math.h>

// Minhas Bibliotécas
//#include "backgroundSubtraction.h"
//#include "hsv_handler.h"
#include "filtraCores.h"
#include "Vareta.h"
#include "manipulaImagem.h"
#include "manipulaArquivos.h"
#include "objetoApontado.h"
#include "SerialPort.h"
#endif
```

Principal.cpp

```
#include "arqInclusao.h"

unsigned char execucaoParte2(cv::Mat& imagem, cv::Point& vetorPosicaoVareta, cv::Point&
vetorDirecaoVareta){
	cv::Mat imagemLab;
	cv::Mat maskPink;
	cv::Mat maskBlue;
	cv::Rect ROI = cv::Rect(0, 0, 1280, 720);
	unsigned char varetaEncontrada = 0;

	cv::cvtColor(imagem, imagemLab, CV_BGR2Lab);

	// Deslocando para esquerda:
	mascaraIntersecaoLab(imagemLab, maskPink, maskBlue, 0); // Já faz erosão e dilatação
	direcaoApontada(ROI, maskPink, maskBlue, vetorPosicaoVareta, vetorDirecaoVareta, 0,
varetaEncontrada);

	//mostra(maskPink, "Pink");
	//mostra(maskBlue, "Blue");

	// Deslocando para cima:
	if(!varetaEncontrada){
		mascaraIntersecaoLab(imagemLab, maskPink, maskBlue, 1); // Já faz erosão e dilatação
		direcaoApontada(ROI, maskPink, maskBlue, vetorPosicaoVareta, vetorDirecaoVareta, 1,
varetaEncontrada);
	}

	//mostra(maskPink, "Pink");
	//mostra(maskBlue, "Blue");

	imagemLab.release();
	maskPink.release();
	maskBlue.release();

	return varetaEncontrada;
}

unsigned char execucao(cv::Mat& imagem1, cv::Mat& imagem2, std::vector<alvo>& alvos1,
std::vector<alvo>& alvos2) {
	cv::Point vetorPosicaoVareta1 = cv::Point(0, 0);
	cv::Point vetorPosicaoVareta2 = cv::Point(0, 0);

	cv::Point vetorDirecaoVareta1 = cv::Point(0, 0);
	cv::Point vetorDirecaoVareta2 = cv::Point(0, 0);

	unsigned char varetasEncontradas = 0; // bit0: imagem1, bit1: imagem2

	varetasEncontradas = execucaoParte2(imagem1, vetorPosicaoVareta1, vetorDirecaoVareta1);
	varetasEncontradas |= execucaoParte2(imagem2, vetorPosicaoVareta2, vetorDirecaoVareta2) << 1;

	int portaArduino;
	if (varetasEncontradas == 3) {
		portaArduino = (int)objetoApontado(alvos1, vetorPosicaoVareta1, vetorDirecaoVareta1,
alvos2, vetorPosicaoVareta2, vetorDirecaoVareta2, varetasEncontradas);
		std::cout << portaArduino << "\n";
		if (portaArduino == 255) varetasEncontradas = 0;
		// TODO: passar numero da porta para o Arduíno
	}

	// Código opcional:
	if(varetasEncontradas & 1){
		/*for (int i = 0; i < alvos1.size(); ++i) {
			cv::line(imagem1, vetorPosicaoVareta1, alvos1[i].pt, cv::Scalar(0, 0, 255), 3);
		}*/
		cv::line(imagem1, vetorPosicaoVareta1, vetorPosicaoVareta1 + vetorDirecaoVareta1,
cv::Scalar(0, 255, 0), 8);
	}
	if(varetasEncontradas & 2){
		/*for (int i = 0; i < alvos2.size(); ++i) {
			cv::line(imagem2, vetorPosicaoVareta2, alvos2[i].pt, cv::Scalar(0, 0, 255), 3);
		}*/
		cv::line(imagem2, vetorPosicaoVareta2, vetorPosicaoVareta2 + vetorDirecaoVareta2,
cv::Scalar(0, 255, 0), 8);
	}
```

```
	// Código opcional:
	mostra(imagem1, "img1");
	mostra(imagem2, "img2");

	return varetasEncontradas;
}

int main() {
	cv::Mat imagem1, imagem2; // Imagem do Vídeo será armazenada nesta matriz

	cv::VideoCapture capWebimagem1(1); // usar 0 para câmera do PC,
	cv::VideoCapture capWebimagem2(2); // 1 e 2 para câmeras USB

	// Resolução das câmeras:
	capWebimagem1.set(CV_CAP_PROP_FRAME_WIDTH, 1280); // 640 (default)
	capWebimagem1.set(CV_CAP_PROP_FRAME_HEIGHT, 720); // 480 (default)
	capWebimagem2.set(CV_CAP_PROP_FRAME_WIDTH, 1280); // 640 (default)
	capWebimagem2.set(CV_CAP_PROP_FRAME_HEIGHT, 720); // 480 (default)
		// std::printf("%d x %d", imagem1.cols, imagem1.rows); // 1280 x 720

	// Se não detectar as câmeras:
	if (capWebimagem1.isOpened() == false) {
		std::cout << "ERROR: Nao foi possivel acessar a camera 1...\n\n";
		_getch();
		return 0;
	}
	if (capWebimagem2.isOpened() == false) {
		std::cout << "ERROR: Nao foi possivel acessar a camera 2...\n\n";
		_getch();
		return 0;
	}

	// Carrega Alvos
	// tipo alvo foi definido em manipulaArquivos.h
	std::vector<alvo> alvos1, alvos2;
	if(carregaAlvos(alvos1, "alvosLara1.txt")) return 0;
	if(carregaAlvos(alvos2, "alvosLara2.txt")) return 0;

	std::cout << "Alvos1:\n";
	for (int i = 0; i < alvos1.size(); ++i) {
		std::cout << alvos1[i].pt.x << " , " << alvos1[i].pt.y << "\n";
	}

	std::cout << "Alvos2:\n";
	for (int i = 0; i < alvos2.size(); ++i) {
		std::cout << alvos2[i].pt.x << " , " << alvos2[i].pt.y << "\n";
	}

	// Enquanto não prescinar esc e câmeras estiverem aberta
	char teclaPressionada = 0;
	char botaoPressionado = 0;

	cv::Mat imagemPreta = cv::Mat(10, 10, CV_8UC3, CV_RGB(0, 0, 0));
	mostra(imagemPreta, "Main");
	unsigned char varetaNaoEncontradaCont = 0;

	unsigned char contadorPulaFrames;

	while (teclaPressionada != 27 && capWebimagem1.isOpened() && capWebimagem2.isOpened()) {
		if (teclaPressionada == 13) {
			//std::cout << " > > > > > > > > > > pressionou Enter < < < < < < < < < <\n";
			if (botaoPressionado == 0) {
				contadorPulaFrames = 0;
			}
			botaoPressionado = 1;
			varetaNaoEncontradaCont = 0;
		}
		if(botaoPressionado && teclaPressionada != 13){ // espera soltar o botão para executar
(para não executar por "infinitos" loops
			//botaoPressionado = 0;
			++contadorPulaFrames;

			bool frameLido = capWebimagem1.read(imagem1);
			if (!frameLido || imagem1.empty()) {
				std::cout << "ERROR: Erro ao tentar ler frame da camera 1\n";
				continue;
			}
```

```
                frameLido = capWebimagem2.read(imagem2);
                if (!frameLido || imagem2.empty()) {
                     std::cout << "ERROR: Erro ao tentar ler frame da camera 2\n";
                     continue;
                }

                //mostra(imagem1, "Imagem Original 1");
                //mostra(imagem2, "Imagem Original 2");
                if(contadorPulaFrames > 5) // pula 5 frames pois a camera da Logitech continua com
falhas...
                if (execucao(imagem1, imagem2, alvos1, alvos2) == 3) { // vareta foi encontrada
                     botaoPressionado = 0;
                }else{
                     ++varetaNaoEncontradaCont;
                     if (varetaNaoEncontradaCont >= QUANT_FRAMES) { // Continua procurando por
QUANT_FRAMES frames
                          botaoPressionado = 0;
                          std::cout << "Vareta nao encontrada\n";
                     }
                }
           }

           teclaPressionada = cv::waitKey(5); // Dá um delay de 5ms, para alguém prescinar esc
      }

      return 0;
}
```

alvosLara1.txt

"Lampada 1 Alvo 1" [697 , -316] - 3

"Lampada 1 Alvo 2" [822 , -312] - 3

"Lampada 1 Alvo 3" [916 , -272] - 3

"Lampada 2 Alvo 1" [-98 , -304] - 4

"Lampada 2 Alvo 2" [117 , -275] - 4

"Lampada 2 Alvo 3" [287 , -259] - 4

"AC Alvo 1" [993 , -131] - 5

"AC Alvo 2" [842 , -129] – 5

alvosLara2.txt

"Lampada 1 Alvo 1" [1587 , -292] - 3

"Lampada 1 Alvo 2" [1368 , -269] - 3

"Lampada 1 Alvo 3" [1192 , -258] - 3

"Lampada 2 Alvo 1" [1015 , -396] - 4

"Lampada 2 Alvo 2" [820 , -346] - 4

"Lampada 2 Alvo 3" [688 , -316] - 4

"AC Alvo 1" [620 , -128] - 5

"AC Alvo 2" [454 , -129] - 5

filtraCores.cpp

```
#include "arqInclusao.h"

// [ Funções de filtro ]  - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

/** Entrar com uma imagem em BGR / HSV / (CIE)Lab / ...
a,b,c sao os canais da imagem (ex: p/ HSV: a = H, b = S, c = V)
Se entrar com iMin < iMax, então pega-se dentro deste intervalo iMin < i < iMax (onde "i" é "b" ou
"c")
Se entrar com iMin > iMax, então pega-se fora deste intervalo i < iMin ou i > iMax **/
void filtraBC(cv::Mat& imagemABC_, cv::Mat& mask, int bMin, int bMax, int cMin, int cMax) {
     // Variáveis locais
     cv::Mat imagemABC = imagemABC_.clone();
     cv::Mat maskB; cv::Mat maskC;

     unsigned char flag = 0; int aux;

     // Se necessário, inverte iMin com iMax e seta flag
     if (bMin > bMax) {
          flag += 1;
          aux = bMin; bMin = bMax; bMax = aux;
     }
     if (cMin > cMax) {
          flag += 2;
          aux = cMin; cMin = cMax; cMax = aux;
     }

     // Extrai canais B e C
     cv::extractChannel(imagemABC, maskB, 1);
     cv::extractChannel(imagemABC, maskC, 2);
     imagemABC.release();

     // Cria máscaras
     cv::inRange(maskB, bMin, bMax, maskB);
     cv::inRange(maskC, cMin, cMax, maskC);

     // Inverte máscaras, se necessário
     if (flag & 1) {
          maskB = ~maskB;
     }
     if (flag & 2) {
          maskC = ~maskC;
     }

     // Interseção das máscaras
     mask = maskB & maskC;
     maskB.release();
     maskC.release();
}
```

manipulaArquivos.cpp

```
#include "arqInclusao.h"

unsigned char armazenaAlvo(std::string linha, alvo& novoAlvo){
     unsigned char estado = 0;
     char caractere;
     unsigned char negativo = 0;
     for (std::string::iterator it=linha.begin(); it!=linha.end(); ++it){
          caractere = *it;

          switch(estado){
               case 0:
                    if(caractere == ' ' || caractere == '\t') break;
                    if(caractere == '\"'){++estado; novoAlvo.nome.clear(); break;}
                    return 1;
               case 1:
                    if(caractere == '\"'){++estado; break;}
                    novoAlvo.nome.append({caractere});
                    break;
               case 2:
                    if(caractere == ' ' || caractere == '\t') break;
                    if(caractere == '['){negativo = 0; ++estado; break;}
                    return 1;
               case 3:
                    if(caractere == ' ' || caractere == '\t') break;
                    if(caractere == '-') {negativo = 1; break;}
                    if(caractere >= '0' && caractere <= '9'){
                         novoAlvo.pt.x = caractere - '0';
                         ++estado;
                         break;
                    }
                    return 1;
               case 4:
                    if(caractere >= '0' && caractere <= '9'){
                         novoAlvo.pt.x *= 10;
                         novoAlvo.pt.x += caractere - '0';
                         break;
                    }
                    else if(caractere == ' ' || caractere == '\t'){++estado; if(negativo)
{novoAlvo.pt.x *= -1; negativo = 0;} break;}
                    else if(caractere == ',' || caractere == ';'){estado += 2; if(negativo)
{novoAlvo.pt.x *= -1; negativo = 0;} break;}
                    return 1;
               case 5:
                    if(caractere == ' ' || caractere == '\t') break;
                    if(caractere == ',' || caractere == ';'){++estado; break;}
                    return 1;
               case 6:
                    if(caractere == ' ' || caractere == '\t') break;
                    if(caractere == '-') {negativo = 1; break;}
                    if(caractere >= '0' && caractere <= '9'){
                         novoAlvo.pt.y = caractere - '0';
                         ++estado;
                         break;
                    }
                    return 1;
               case 7:
                    if(caractere >= '0' && caractere <= '9'){
                         novoAlvo.pt.y *= 10;
                         novoAlvo.pt.y += caractere - '0';
                         break;
                    }
                    else if(caractere == ' ' || caractere == '\t'){++estado; if(negativo)
{novoAlvo.pt.y *= -1; negativo = 0;} break;}
                    else if(caractere == ']'){estado += 2; if(negativo) {novoAlvo.pt.y *= -1;
negativo = 0;} break;}
                    return 1;
               case 8:
                    if(caractere == ' ' || caractere == '\t') break;
                    if(caractere == ']'){++estado; break;}
                    return 1;
               case 9:
                    if(negativo) novoAlvo.pt.y *= -1;
                    if(caractere == ' ' || caractere == '\t') break;
                    if(caractere == '-'){++estado; break;}
                    return 1;
```

```
                case 10:
                     if(caractere == ' ' || caractere == '\t') break;
                     if(caractere >= '0' && caractere <= '9'){
                          novoAlvo.portaArduino = caractere - '0';
                          ++estado;
                          break;
                     }
                     return 1;
                case 11:
                     if(caractere >= '0' && caractere <= '9'){
                          novoAlvo.pt.y *= 10;
                          novoAlvo.pt.y += caractere - '0';
                          break;
                     }
                     ++estado;
                     break;
           }
           if(estado == 12) break;
      }
      if(estado < 11) return 1; // não leu tudo que precisava
      return 0; // tudo ok
}

unsigned char carregaAlvos(std::vector<alvo>& alvos, const char* nomeArquivo){
      std::ifstream arquivo;
      arquivo.open(nomeArquivo);

      std::string linha;
      unsigned char linhaCont = 0;
      alvo novoAlvo;
      if(arquivo.is_open()){
           while(std::getline(arquivo, linha)){
                ++linhaCont;
                if(armazenaAlvo(linha, novoAlvo)){
                     std::cout << "Verifique linha: " << (int) linhaCont << "!\nPadrao: \"Nome\" [x,
y] - portaArduino Comentarios\n";
                     getchar();
                     return 1;
                }
                alvos.push_back(novoAlvo);
           }
      }else{
           std::cout << "Nao foi possivel abrir " << nomeArquivo << "!\n";
           getchar();
           return 1;
      }
      arquivo.close();

      return 0;
}
```

manipulaArquivos.h

```
typedef struct alvo{
      cv::Point pt;
      std::string nome;
      unsigned char portaArduino; // Também serve como ID do alvo
      float distancia;
}alvo;

unsigned char carregaAlvos(std::vector<alvo>& alvos, const char* nomeArquivo);
```

manipulaImagem.cpp

```
#include "arqInclusao.h"

void mostra(cv::Mat& imagem, const char* titulo) {
	cv::namedWindow(titulo, CV_WINDOW_NORMAL);
	cv::imshow(titulo, imagem);
}

void clareiaImagemComMascara(cv::Mat& imagem, cv::Mat& mask) {
	cv::Mat imagemANDmask = imagem.clone();
	cv::Mat imagemAND_NOTmask = imagem.clone();
	cv::bitwise_and(imagemANDmask, 1, imagemANDmask, ~mask); // Só sobra partes da imagem com
interseção com a máscara
	cv::bitwise_and(imagemAND_NOTmask, 1, imagemAND_NOTmask, mask); // Só sobra partes da imagem
sem interseção com a máscara

	//cv::cvReleaseImage(&imagem);
	imagem = 1.2*imagemANDmask + .2*imagemAND_NOTmask;
	imagemANDmask.release();
	imagemAND_NOTmask.release();
}

/** Escreve texto na imagem na posição x, y com escala (1 é padrão) e com cores b, g, r **/
void escreveTexto(cv::Mat& imagem, cv::String txt, int x, int y, double escala, int b, int g, int r)
{
	cv::putText(imagem, txt, cv::Point(x, y), cv::FONT_HERSHEY_SIMPLEX, escala, cv::Scalar(b, g,
r));
}

/** Escreve texto e valor na imagem na posição x, y com escala (1 é padrão) e com cores b, g, r **/
void escreveTexto(cv::Mat& imagem, cv::String txt, double valor, int x, int y, double escala, int b,
int g, int r) {
	std::ostringstream strs;
	strs << valor; // converte valor para string
	txt += strs.str(); // concatena texto com valor
	cv::putText(imagem, txt, cv::Point(x, y), cv::FONT_HERSHEY_SIMPLEX, escala, cv::Scalar(b, g,
r));
}

/** Desenha um retangulo entre os pontos (x1,y1), (x2,y2) com cores b, g, r **/
void desenhaRetangulo(cv::Mat& imagem, int x1, int y1, int x2, int y2, int b, int g, int r) {
	cv::rectangle(imagem, cv::Point(x1, y1), cv::Point(x2, y2), cv::Scalar(b, g, r));
}

/** Dilata e erode as máscaras **/
void dilatacaoErosaoMascara(cv::Mat& mask, int tamErosao, int tamDilatacao) {
	cv::Mat kernel1 = cv::getStructuringElement(cv::MORPH_RECT, cv::Size(tamErosao, tamErosao));
	cv::Mat kernel2 = cv::getStructuringElement(cv::MORPH_RECT, cv::Size(tamDilatacao,
tamDilatacao));
	cv::erode(mask, mask, kernel1);
	cv::dilate(mask, mask, kernel2);

	kernel1.release();
	kernel2.release();
}

void dilatacaoMascara(cv::Mat& mask, int tamDilatacao) {
	cv::Mat kernel = cv::getStructuringElement(cv::MORPH_RECT, cv::Size(tamDilatacao,
tamDilatacao));
	cv::dilate(mask, mask, kernel);
	kernel.release();
}
```

objetoApontado.cpp

```
#include "arqInclusao.h"

void objetosApontados(std::vector<alvo>& alvos, cv::Point& vetorPosicao, cv::Point& vetorDirecao,
std::vector<alvo>& alvosApontados){
     cv::Point direcaoAlvo;
     double cosAng;

     int listaSize = alvos.size();
     float distancia;
     for(int i = 0; i < listaSize; ++i){
          direcaoAlvo = alvos[i].pt - vetorPosicao;
          distancia = distanciaPts(alvos[i].pt, vetorPosicao); // distanciaPts() está definida em
Vareta.cpp
          direcaoAlvo *= 100.0; // Vetor direção tem magnitude 100
          direcaoAlvo /= distancia; // mas tem de deixálo unitário "antes"
          cosAng = vetorDirecao.x * direcaoAlvo.x + vetorDirecao.y * direcaoAlvo.y; // de verdade
10000 vezes o cosseno (pois ambos os vetores tem magnitude 100)
          if(cosAng >= COS_ANG){ // +- 10°
               alvos[i].distancia = distancia;
               alvosApontados.push_back(alvos[i]);
          }
     }
}

void intersecaoAlvos(std::vector<alvo>& alvosApontados1, std::vector<alvo>& alvosApontados2){
     int lista2Size = alvosApontados2.size();
     unsigned char encontrouIntersecao;
     for(int i = 0; i < alvosApontados1.size(); ++i){
          encontrouIntersecao = 0;
          for(int j = 0; j < lista2Size; ++j){
               if(alvosApontados1[i].portaArduino == alvosApontados2[j].portaArduino){
                    encontrouIntersecao = 1;
                    break;
               }
          }
          if(!encontrouIntersecao){ // Se não encontrou, remove de alvosApontados1
               alvosApontados1.erase(alvosApontados1.begin() + i);
               --i;
          }
     }
}

unsigned char alvoMaisProximo(std::vector<alvo> alvosApontados){
     if(alvosApontados.size() == 0)
          return 255; // não encontrou alvo

     alvo menorDistancia = alvosApontados[0];
     int listaSize = alvosApontados.size();
     for(int i = 1; i < listaSize; ++i){
          if(alvosApontados[i].distancia < menorDistancia.distancia){
               menorDistancia = alvosApontados[i];
          }
     }
     return menorDistancia.portaArduino;
}

unsigned char objetoApontado(std::vector<alvo>& alvos1, cv::Point& vetorPosicao1, cv::Point&
vetorDirecao1,
                                   std::vector<alvo>& alvos2, cv::Point& vetorPosicao2, cv::Point&
vetorDirecao2,
                                   unsigned char varetasEncontradas){
     std::vector<alvo> alvosApontados1, alvosApontados2;

     if((varetasEncontradas & 3) != 3) return 255;
     objetosApontados(alvos1, vetorPosicao1, vetorDirecao1, alvosApontados1);
     objetosApontados(alvos2, vetorPosicao2, vetorDirecao2, alvosApontados2);
     intersecaoAlvos(alvosApontados1, alvosApontados2); // retorna interseção em alvosApontados1
     return alvoMaisProximo(alvosApontados1);
}
```

Vareta.cpp

```
#include "arqInclusao.h"

//*********************************************************************//
//*                                                                   *//
//*   Função para gerar máscaras de interseção das cores rosa e azul  *//
//*                                                                   *//
//*********************************************************************//

/** Cira mascaras com pontos de interseção das cores rosa e azul da imagem
     Uma das máscaras representam a parte rosa da interseção, já a  outra a parte azul**/
void mascaraIntersecaoLab(cv::Mat& imagemLab, cv::Mat& maskPink, cv::Mat& maskBlue, unsigned char
direcaoDeslocamento) {
     // Filtra Cores - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
     cv::Mat maskP;
     cv::Mat maskB;
     filtraBC(imagemLab, maskP, 145, 190, 100, 145);
     filtraBC(imagemLab, maskB, 120, 160, 60, 113);

     dilatacaoErosaoMascara(maskP, EROSAO1, DILATACAO1);
     dilatacaoErosaoMascara(maskP, EROSAO2, DILATACAO2);
     dilatacaoErosaoMascara(maskB, EROSAO1, DILATACAO1);
     dilatacaoErosaoMascara(maskB, EROSAO2, DILATACAO2);

     cv::Mat maskPD;     // pink deslocado
     cv::Mat maskBD;     // blue deslocado

     // Desloca mascaras para esquerda - - - - - - - - - - - - - - - - - - - - - - - - - -
     if(direcaoDeslocamento == 0){
          // desloca mascara rosa para a esquerda
          maskPD = cv::Mat::zeros(maskP.size(), maskP.type());
          maskP(cv::Rect(DESLOCAMENTO, 0, maskP.cols - DESLOCAMENTO,
maskP.rows)).copyTo(maskPD(cv::Rect(0, 0, maskP.cols - DESLOCAMENTO, maskP.rows)));

          // desloca mascara azul para a esquerda
          maskBD = cv::Mat::zeros(maskB.size(), maskB.type());
          maskB(cv::Rect(DESLOCAMENTO, 0, maskB.cols - DESLOCAMENTO,
maskB.rows)).copyTo(maskBD(cv::Rect(0, 0, maskB.cols - DESLOCAMENTO, maskB.rows)));

          // Detecta interseção (AND) e NÃO MAIS as une (OR) - - - - - - - - - - - - - - - -
          cv::bitwise_and(maskPD, maskB, maskBlue, maskB); // Inputs: maskPD, maskB, Output:
maskBlue, ~maskB diz para focar apenas onde tem bits azuls
          cv::bitwise_and(maskBD, maskP, maskPink, maskP);
     // Desloca mascaras para cima - - - - - - - - - - - - - - - - - - - - - - - - - - - -
     }else{
          // desloca mascara rosa para cima
          maskPD = cv::Mat::zeros(maskP.size(), maskP.type());
          maskP(cv::Rect(0, DESLOCAMENTO, maskP.cols, maskP.rows -
DESLOCAMENTO)).copyTo(maskPD(cv::Rect(0, 0, maskP.cols, maskP.rows - DESLOCAMENTO)));

          // desloca mascara azul para cima
          maskBD = cv::Mat::zeros(maskB.size(), maskB.type());
          maskB(cv::Rect(0, DESLOCAMENTO, maskB.cols, maskB.rows -
DESLOCAMENTO)).copyTo(maskBD(cv::Rect(0, 0, maskB.cols, maskB.rows - DESLOCAMENTO)));

          // Detecta interseção (AND) //e NÃO MAIS as une (OR) - - - - - - - - - - - - - - -
          cv::bitwise_and(maskPD, maskB, maskBlue, maskB);
          cv::bitwise_and(maskBD, maskP, maskPink, maskP);
     }

     dilatacaoMascara(maskPink, 5);
     dilatacaoMascara(maskBlue, 5);

     maskP.release();
     maskB.release();
     maskPD.release();
     maskBD.release();
}

//*********************************************************************//
//*                                                                   *//
//*   Funções auxiliares para determinar a posição da vareta          *//
//*                                                                   *//
//*********************************************************************//

/** Recebe as mascaras das interseções e retorna os clusters delas **/
```

```
void geraClusters(cv::Mat& maskPink, cv::Mat& maskBlue, std::vector<cv::Point>& centroClustersP,
std::vector<cv::Point>& centroClustersB){
     // Encontra contornos das mascaras  - - - - - - - - - - - - - - - - - - - - - - - - - - - -
     std::vector<std::vector<cv::Point>> contornosP;
     std::vector<std::vector<cv::Point>> contornosB;
     std::vector<cv::Vec4i> hierarchyP;
     std::vector<cv::Vec4i> hierarchyB;

     cv::findContours(maskPink, contornosP, hierarchyP, CV_RETR_EXTERNAL, CV_CHAIN_APPROX_SIMPLE,
cv::Point(0, 0)); // Somente contornos externos
     cv::findContours(maskBlue, contornosB, hierarchyB, CV_RETR_EXTERNAL, CV_CHAIN_APPROX_SIMPLE,
cv::Point(0, 0)); // Somente contornos externos

     hierarchyP.clear(); // Hierarquias são desnecessárias
     hierarchyB.clear(); // Hierarquias são desnecessárias
     if (contornosP.size() < 1 || contornosB.size() < 1) return; // se não encontrou ao menos 1
contorno

     // Encontra contornos das mascaras  - - - - - - - - - - - - - - - - - - - - - - - - - - - -
     std::vector<cv::Moments> muP(contornosP.size()); // Momentos
     std::vector<cv::Moments> muB(contornosB.size()); // Momentos
     //std::vector<cv::Point> mcP(contornosB.size()); // Centro geométrico
     //std::vector<cv::Point> mcB(contornosB.size()); // Centro geométrico

     int tamAux = contornosP.size();
     for (int i = 0; i < tamAux; ++i) {
          muP[i] = cv::moments(contornosP[i], false);
     }
     tamAux = contornosB.size();
     for (int i = 0; i < tamAux; ++i) {
          muB[i] = cv::moments(contornosB[i], false);
     }

     contornosP.clear();
     contornosB.clear();

     tamAux = muP.size();
     for (int i = 0; i < tamAux; ++i) {
          //mcP[i] = cv::Point(muP[i].m10 / muP[i].m00, muP[i].m01 / muP[i].m00);
          centroClustersP.push_back(cv::Point(muP[i].m10 / muP[i].m00, muP[i].m01 / muP[i].m00));
     }
     tamAux = muB.size();
     for (int i = 0; i < tamAux; ++i) {
          //mcB[i] = cv::Point(muB[i].m10 / muB[i].m00, muB[i].m01 / muB[i].m00);
          centroClustersB.push_back(cv::Point(muB[i].m10 / muB[i].m00, muB[i].m01 / muB[i].m00));
     }

     muP.clear();
     muB.clear();
}

// Código de merge sort adaptado do disponível em:
https://pt.wikipedia.org/wiki/Merge_sort#Implementa%C3%A7%C3%A3o_em_C++
// Acesso em 27/03/2019
void mergePointDist(std::vector<pointDist>& vetor, int ini, int meio, int fim,
std::vector<pointDist>& vetorAux) {
    int esq = ini;
    int dir = meio;
    for (int i = ini; i < fim; ++i) {
        if ((esq < meio) and ((dir >= fim) or (vetor[esq].dist < vetor[dir].dist))) {
            vetorAux[i] = vetor[esq];
            ++esq;
        }
        else {
            vetorAux[i] = vetor[dir];
            ++dir;
        }
    }
    //copiando o vetor de volta
    for (int i = ini; i < fim; ++i) {
        vetor[i] = vetorAux[i];
    }
}

void mergeSortPointDist(std::vector<pointDist>& vetor, int inicio, int fim, std::vector<pointDist>&
vetorAux) {
    if ((fim - inicio) < 2) return;
```

```
    int meio = ((inicio + fim)/2);
    mergeSortPointDist(vetor, inicio, meio, vetorAux);
    mergeSortPointDist(vetor, meio, fim, vetorAux);
    mergePointDist(vetor, inicio, meio, fim, vetorAux);
}

// Função do merge sort que o usuario realmente chama
void mergeSortPointDist(std::vector<pointDist>& vetor, int tamanho) {
    //criando vetor auxiliar
     std::vector<pointDist> vetorAux(tamanho);
    mergeSortPointDist(vetor, 0, tamanho, vetorAux);
}

double distanciaPts(cv::Point& pt1, cv::Point& pt2){
     double cat1 = pt1.x - pt2.x;
     double cat2 = pt1.y - pt2.y;
     return sqrt(cat1*cat1 + cat2*cat2);
}

void ordenaClusters(cv::Point& vetorPosicaoROI, std::vector<cv::Point>& centroClusters,
std::vector<pointDist>& lista){
     int tamCentroClusters = centroClusters.size();
     for(int i = 0; i < tamCentroClusters; ++i){
          lista[i].pt = centroClusters[i];
          lista[i].dist = distanciaPts(centroClusters[i], vetorPosicaoROI);
     }
     mergeSortPointDist(lista, lista.size());
}

unsigned char verificaSeDentroROI(cv::Point& ptEstimado, cv::Rect& ROI){
     if(ptEstimado.x < -TOLERANCIA_CLUSTERS) return 0;
     if(ptEstimado.y < -TOLERANCIA_CLUSTERS) return 0;
     if(ptEstimado.x > ROI.width + TOLERANCIA_CLUSTERS) return 0;
     if(ptEstimado.y > ROI.height + TOLERANCIA_CLUSTERS) return 0;
     return 1;
}

unsigned char estimaProxPt(cv::Point& vetorPosicaoROI, cv::Point& pt1, cv::Point& pt2, cv::Rect&
ROI, pointDist& ptEstimado){
     // Estima posição do próximo ponto da vareta
     ptEstimado.pt = pt1 + 2*(pt2-pt1);
     // Distância com relação ao vetor posição da vareta. Distância esta utilizada para ordenar
clusters
     ptEstimado.dist = distanciaPts(ptEstimado.pt, vetorPosicaoROI);
     // Se estiver fora da ROI, retorna 0
     return verificaSeDentroROI(ptEstimado.pt, ROI);
}

int procuraPtEstimado(pointDist& ptEstimado, std::vector<pointDist> lista){
     int listaSize = lista.size();
     for(int i = 0; i < listaSize; ++i){
          // Não precisa percorrer toda a lista (só até ultrapassar distância)
          if(lista[i].dist > ptEstimado.dist + TOLERANCIA_CLUSTERS_SQRT2)
               break;
          if(std::abs(lista[i].pt.x - ptEstimado.pt.x) < TOLERANCIA_CLUSTERS
               && std::abs(lista[i].pt.y - ptEstimado.pt.y) < TOLERANCIA_CLUSTERS)
               return i;
     }
     return -1;
}

void copiaLista(std::vector<pointDist>& listaFrom, std::vector<pointDist>& listaTo) {
     listaTo.clear();
     int listaSize = listaFrom.size();
     for (int i = 0; i < listaSize; ++i) {
          listaTo.push_back(listaFrom[i]);
     }
}

unsigned char ptEhDaVareta(pointDist& pt, std::vector<pointDist>& ptsVareta){
     int listaSize = ptsVareta.size();
     for(int i = 0; i < listaSize; ++i){
          if(ptsVareta[i].dist > pt.dist) return 0;
          if(ptsVareta[i].pt.x == pt.pt.x
               && ptsVareta[i].pt.y == pt.pt.y) return 1;
     }
```

```
     return 0;
}

//*********************************//
//*                               *//
//*   Funções para determinar ROI   *//
//*                               *//
//*********************************//

void alteraROI(cv::Rect& ROI, int x, int y, int width, int height){
     ROI.x = x;
     ROI.y = y;
     ROI.width = width;
     ROI.height = height;
}

void trackVareta(cv::Point& pt1, cv::Point& pt2, cv::Rect& ROI){
     // pt1: pt mais a esquerda ou mais em cima
     // pt2: pt mais a direita ou mais em baixo

     int xMin, xMax, yMin, yMax, distX, distY;

     if(pt1.x < pt2.x){
          xMin = pt1.x + ROI.x;
          xMax = pt2.x + ROI.x;
     }else{
          xMin = pt2.x + ROI.x;
          xMax = pt1.x + ROI.x;
     }
     if(pt1.y < pt2.y){
          yMin = pt1.y + ROI.y;
          yMax = pt2.y + ROI.y;
     }else{
          yMin = pt2.y + ROI.y;
          yMax = pt1.y + ROI.y;
     }

     distX = xMax - xMin;
     distY = yMax - yMin;

     if(distX > 75){
          xMin -= distX >> 1; // distX dividido por 2
          xMax = distX << 1;  // distX e distY vezes 2
     }else{
          xMin -= 100;
          xMax =  200;
     }
     if(distY > 75){
          yMin -= distY >> 1; // distY dividido por 2
          yMax = distY << 1;  // distY e distY vezes 2
     }else{
          yMin -= 100;
          yMax = 200;
     }

     if (xMin < 0) xMin = 0;
     if (yMin < 0) yMin = 0;

     if (xMin + xMax > 1280)xMax = 1280 - xMin;
     if (yMin + yMax > 720)  yMax = 720 - yMin;

     alteraROI(ROI, xMin, yMin, xMax, yMax);
}

//*************************************************//
//*                                               *//
//*   Funções para determinar a posição da vareta   *//
//*                                               *//
//*************************************************//

/** Uma vez encontrados três pontos de cores alternadas em linha reta e equidistantes,
     Esta função procura pelos demais 2 da vareta (ou se encontrar 6+ descarta opção de ser vareta,
para QUANT_CORES_VARETA == 5) **/
int procuraMaisPtsVareta(std::vector<pointDist>& listaP, std::vector<pointDist>& listaB, unsigned
char& varetaEncontrada,
                         cv::Point vetorPosicaoROI, cv::Rect& ROI,
```

```
                              std::vector<pointDist>& ptsEncontradosP, std::vector<pointDist>&
ptsEncontradosB,
                              std::vector<pointDist>& ptsCandidatosVaretaP, std::vector<pointDist>&
ptsCandidatosVaretaB){
     cv::Point deslocamentoBP = ptsEncontradosP[0].pt - ptsEncontradosB[0].pt; // Estimativa de
quanto deslocar para encontrar novos pontos
                                                                                                //
Aponta do azulSelecionado para o rosaSelecionado

     pointDist ptDir, ptContDir; // Pontos extremos da vareta na direção do deslocamentoBP e na
direção contrária a deslocamentoBP
     unsigned char contQuantPts = 2; // contador de quantidade de pontos encontrados

     unsigned char flagCtrl = 3; // Procurar para o sentido do deslocamentoBP (+1) e no sentido
contrário (+2)
                                      // Se não encontrar nestes sentidos (-1 e/ou -2)
                                      // Se estes bits == 0, então sai da função (verificando se
vareta foi ou não encontrada)
                                      // Também sai se encontrar 6+ pontos (para
QUANT_CORES_VARETA == 5)
     if(ptsEncontradosP.size() == 2) {flagCtrl |= 4; ++contQuantPts;} // Se a função anterior
encontrou um rosa
     if(ptsEncontradosB.size() == 2) {flagCtrl |= 8; ++contQuantPts;} // Se a função anterior
encontrou um azul
     // Se flagCtrl & 8 == 8, então a próxima cor a se procurar na direção ao deslocamentoBP é rosa
     // Se flagCtrl & 4 == 4, então a próxima cor a se procurar na direção contrária ao
deslocamentoBP é azul

     // Faz média dos intervalos entre cores e armazena em deslocamentoBP  - - - - - - - - - -
     if(flagCtrl & 4){
          // Novo pt rosa com pt azulSelecionado
          deslocamentoBP += ptsEncontradosB[0].pt - ptsEncontradosP[1].pt;
          if(flagCtrl & 8){
               // Novo pt azul com pt rosaSelecionado
               deslocamentoBP = (deslocamentoBP + ptsEncontradosB[1].pt - ptsEncontradosP[0].pt) /
3;
               ptDir = ptsEncontradosB[1]; // novo pt azul
          }else{
               deslocamentoBP /= 2;
               ptDir = ptsEncontradosP[0]; // pt azulSelecionado
          }
          ptContDir = ptsEncontradosP[1]; // novo pt rosa
     }else{
          // Novo pt azul com pt rosaSelecionado
          deslocamentoBP = (deslocamentoBP + ptsEncontradosB[1].pt - ptsEncontradosP[0].pt) / 2;
          ptDir = ptsEncontradosB[1];   // novo pt azul
          ptContDir = ptsEncontradosB[0]; // pt rosaSelecionado
     }

     // Procura por 5 ou 6+ cores na linha reta  - - - - - - - - - - - - - - - - - - - - - - - -
     int posEncontrada;
     while(1){
          // Estima posição do novo ponto na direção de deslocamentoBP
          if(flagCtrl & 1){
               ptDir.pt += deslocamentoBP;
               ptDir.dist = distanciaPts(ptDir.pt, vetorPosicaoROI);
               // Se não estiver dentro da ROI
               if(!verificaSeDentroROI(ptDir.pt, ROI)){
                    flagCtrl &= 0b11111110; // Não procurar na direção de deslocamentoBP
               }
          }
          // Procura na direção de deslocamentoBP
          if(flagCtrl & 1){
               // Procura por um rosa
               if(flagCtrl & 8){
                    flagCtrl &= 0b11110111; // Alterna para procurar azul na próxima iteração
                    posEncontrada = procuraPtEstimado(ptDir, listaP);
                    if(posEncontrada != -1){
                         // Se encontrar, adiciona na lista de ptsEncontradosP
                         ptsEncontradosP.push_back(listaP[posEncontrada]);
                         ++contQuantPts;
                    }else{
                         // Se não encontrar
                         flagCtrl &= 0b11111110; // Não procurar na direção de deslocamentoBP
                    }
               // Procura por um azul
               }else{
```

```
                    flagCtrl |= 8; // Alterna para procurar rosa na próxima iteração
                    posEncontrada = procuraPtEstimado(ptDir, listaB);
                    if(posEncontrada != -1){
                         // Se encontrar, adiciona na lista de ptsEncontradosB
                         ptsEncontradosB.push_back(listaB[posEncontrada]);
                         ++contQuantPts;
                    }else{
                         // Se não encontrar
                         flagCtrl &= 0b11111110; // Não procurar na direção de deslocamentoBP
                    }
               }
          }
          // Estima posição do novo ponto na direção contrária de deslocamentoBP
          if(flagCtrl & 2){
               ptContDir.pt -= deslocamentoBP;
               ptContDir.dist = distanciaPts(ptContDir.pt, vetorPosicaoROI);
               if(!verificaSeDentroROI(ptContDir.pt, ROI)){
                    // Se não estiver dentro da ROI
                    flagCtrl &= 0b11111101; // Não procurar na direção contrária de deslocamentoBP
               }
          }
          // Procura na direção contrária de deslocamentoBP
          if(flagCtrl & 2){
               // Procura por um azul
               if(flagCtrl & 4){
                    flagCtrl &= 0b11111011; // Alterna para procurar rosa na próxima iteração
                    posEncontrada = procuraPtEstimado(ptContDir, listaB);
                    if(posEncontrada != -1){
                         // Se encontrar, adiciona na lista de ptsEncontradosB
                         ptsEncontradosB.push_back(listaB[posEncontrada]);
                         ++contQuantPts;
                    }else{
                         // Se não encontrar
                         flagCtrl &= 0b11111101; // Não procurar na direção contrária de
deslocamentoBP
                    }
               // Procura por um rosa
               }else{
                    flagCtrl |= 4; // Alterna para procurar azul na próxima iteração
                    posEncontrada = procuraPtEstimado(ptContDir, listaP);
                    if(posEncontrada != -1){
                         // Se encontrar, adiciona na lista de ptsEncontradosP
                         ptsEncontradosP.push_back(listaP[posEncontrada]);
                         ++contQuantPts;
                    }else{
                         // Se não encontrar
                         flagCtrl &= 0b11111101; // Não procurar na direção contrária de
deslocamentoBP
                    }
               }
          }
          // Se objeto tem 6+ cores, não é a vareta (para QUANT_CORES_VARETA == 5)
          if(contQuantPts > QUANT_CORES_VARETA){
               break;
          }
          // Não encontrou mais cores nos 2 sentidos (ptDir e ptContDir)
          if(!(flagCtrl & 3)){
               break;
          }
     }

     // Ordena resultados
     mergeSortPointDist(ptsEncontradosP, ptsEncontradosP.size());
     mergeSortPointDist(ptsEncontradosB, ptsEncontradosB.size());
     // Se encontrou a vareta
     if(contQuantPts == QUANT_CORES_VARETA){
          if(varetaEncontrada) return -1; // 2 "varetas" encontradas
          varetaEncontrada = 1;
          copiaLista(ptsEncontradosP, ptsCandidatosVaretaP);
          copiaLista(ptsEncontradosB, ptsCandidatosVaretaB);
          return 1;
     }else{
          return 0;
     }
}
```

```
/** Com os clusters ordenados em listaP e listaB, esta função procura pela vareta dentre eles e, se
encontrar, armazena em
     ptsCandidatosVaretaP e ptsCandidatosVaretaB e muda flag de varetaEncontrada para 1 **/
void procuraVareta(std::vector<pointDist>& listaP, std::vector<pointDist>& listaB, unsigned char&
varetaEncontrada, cv::Point vetorPosicaoROI, cv::Rect& ROI,
                              std::vector<pointDist>& ptsCandidatosVaretaP, std::vector<pointDist>&
ptsCandidatosVaretaB){
     varetaEncontrada = 0;

     int posListaP = 0; // contadores que indicarão
     int posListaB = 0; // cluster corrente nas listaP e listaB.

     pointDist* rosaSelecionado; // clusters correntes
     pointDist* azulSelecionado; // das listaP e listaB.

     pointDist ptEstimado; // ponto auxiliar com estimativa de próximo ponto da vareta
     unsigned char flagPtsEncontrados = 0; // flag para identificar que ponto foi encontrado
     int posEncontrada; // variável auxiliar que guarda temporariamente a posição do ponto
encontrada

     std::vector<pointDist> ptsEncontradosP; // lista temporária que guarda
     std::vector<pointDist> ptsEncontradosB; // pontos encontrados (possível vareta)

     // loop que avançará posListaP e, quando chegar ao valor maximo,
     // avançará em 1 a posListaB e resetará em 0 a posListaP até ambos chegarem ao fim
     while(1){
          rosaSelecionado = &(listaP[posListaP]);
          azulSelecionado = &(listaB[posListaB]);
          ptsEncontradosP.clear();
          ptsEncontradosB.clear();
          flagPtsEncontrados = 0;
          // Estima próximo ponto azul e determina se ele se encontra dentro da ROI
          if(estimaProxPt(vetorPosicaoROI, azulSelecionado->pt, rosaSelecionado->pt, ROI,
ptEstimado)){
               // Procura em listaB pelo ponto estimado
               posEncontrada = procuraPtEstimado(ptEstimado, listaB);
               if(posEncontrada != -1){ // Se encontrar na listaB o ponto estimado
                    ptsEncontradosB.push_back(*azulSelecionado);
                    ptsEncontradosB.push_back(listaB[posEncontrada]);
                    flagPtsEncontrados = 1;
               }
          }
          // Estima próximo ponto rosa e determina se ele se encontra dentro da ROI
          if(estimaProxPt(vetorPosicaoROI, rosaSelecionado->pt, azulSelecionado->pt, ROI,
ptEstimado)){
               // Procura em listaP pelo ponto estimado
               posEncontrada = procuraPtEstimado(ptEstimado, listaP);
               if(flagPtsEncontrados){ // Se encontrou um pt azul anteriormente
                    ptsEncontradosP.push_back(*rosaSelecionado);
                    if(posEncontrada != -1){ // Se encontrou um rosa também
                         ptsEncontradosP.push_back(listaP[posEncontrada]);
                    }
               }else if(posEncontrada != -1){ // Se não encontrou um ponto azul mas encontrou um
rosa
                    ptsEncontradosB.push_back(*azulSelecionado);
                    ptsEncontradosP.push_back(*rosaSelecionado);
                    ptsEncontradosP.push_back(listaP[posEncontrada]);
                    flagPtsEncontrados = 1;
               }
          // Se não conseguiu estimar próximo ponto rosa, mas encontrou um ponto azul anteriormente
          }else if(flagPtsEncontrados){
               ptsEncontradosP.push_back(*rosaSelecionado);
          }

          if(flagPtsEncontrados){
               // Procura por mais pontos na linha reta (se encontrar exatamente 5 com cores
alternadas, são os candidatos aos pontos da vareta)
               // Já se encontrar 6+, então não são os candidatos
               // Esta função, se encontrar a vareta, inicializará ptsCandidatosVaretaP,
ptsCandidatosVaretaB e fará varetaEncontrada = 1
               // Se encontrar 2 "varetas", esta função retornará -1
               if(procuraMaisPtsVareta(listaP, listaB, varetaEncontrada, vetorPosicaoROI, ROI,
ptsEncontradosP,  ptsEncontradosB,
                              ptsCandidatosVaretaP, ptsCandidatosVaretaB) == -1){
                    // Mais de uma "vareta" foi encontrada
                    varetaEncontrada = 0;
                    return;
```

```
                }
            }

            // Avança contadores
            do{
                if(posListaP < listaP.size()-1){
                    posListaP++;
                }else{
                    posListaP = 0;
                    if(posListaB < listaB.size()-1){
                        posListaB++;
                    }
                    else {
                        return;
                    }
                }
            // Para não comparar com 2 pontos já encontrados da vareta (ou com ponto de um objeto com
6+ cores)
            }while((flagPtsEncontrados && ptEhDaVareta(listaP[posListaP], ptsEncontradosP)
                && ptEhDaVareta(listaB[posListaB], ptsEncontradosB)) ||
                    (varetaEncontrada && ptEhDaVareta(listaP[posListaP], ptsCandidatosVaretaP)
                && ptEhDaVareta(listaB[posListaB], ptsCandidatosVaretaB)));
        }
}

/** Determina os vetores posição e direção da vareta **/
void geraVetoresPosicaoDirecao(std::vector<pointDist>& ptsVaretaP, std::vector<pointDist>&
ptsVaretaB, cv::Rect& ROI, cv::Point& pt1, cv::Point& pt2, cv::Point& vetorPosicao, cv::Point&
vetorDirecao, unsigned char direcaoDeslocamento){
    // Encontra pontos extremos - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
    int listaSize;
    pt1 = ptsVaretaP[0].pt; // pt mais a esquerda ou mais em cima
    pt2 = ptsVaretaP[0].pt; // pt mais a direita ou mais em baixo
    // Se deslocou para cima (compara y's)
    if(direcaoDeslocamento){
        listaSize = ptsVaretaP.size();
        for(int i = 1; i < listaSize; ++i){
            if(pt1.y > ptsVaretaP[i].pt.y){
                pt1 = ptsVaretaP[i].pt;
            }
            if(pt2.y < ptsVaretaP[i].pt.y){
                pt1 = ptsVaretaP[i].pt;
            }
        }
        listaSize = ptsVaretaB.size();
        for(int i = 0; i < listaSize; ++i){
            if(pt1.y > ptsVaretaB[i].pt.y){
                pt1 = ptsVaretaB[i].pt;
            }
            if(pt2.y < ptsVaretaB[i].pt.y){
                pt1 = ptsVaretaB[i].pt;
            }
        }
    // Se deslocou para esquerda (compara x's)
    }else{
        listaSize = ptsVaretaP.size();
        for(int i = 1; i < listaSize; ++i){
            if(pt1.x > ptsVaretaP[i].pt.x){
                pt1 = ptsVaretaP[i].pt;
            }
            if(pt2.x < ptsVaretaP[i].pt.x){
                pt2 = ptsVaretaP[i].pt;
            }
        }
        listaSize = ptsVaretaB.size();
        for(int i = 0; i < listaSize; ++i){
            if(pt1.x > ptsVaretaB[i].pt.x){
                pt1 = ptsVaretaB[i].pt;
            }
            if(pt2.x < ptsVaretaB[i].pt.x){
                pt2 = ptsVaretaB[i].pt;
            }
        }
    }

    // Determina o ponto de base e sentido da vareta  - - - - - - - - - - - - - - - - - - -
```

```
     cv::Point ROIPoint(ROI.x, ROI.y); // Vetor posição da origem da ROI em coordenadas da imagem
original

     // Se tiver 3 pontos rosas e 2 azuis (apontando para direita ou para baixo)
     if(ptsVaretaP.size() > ptsVaretaB.size()){
          vetorPosicao = pt1 + ROIPoint;
          vetorDirecao = pt2 - pt1;
     // Se tiver 3 pontos azuis e 3 rosas (apontando para esquera ou para cima)
     }else{
          vetorPosicao = pt2 + ROIPoint;
          vetorDirecao = pt1 - pt2;
     }
     // Vetor direção com magnitude 100
     vetorDirecao *= 100.0 / distanciaPts(pt1, pt2);
}

/** Recebe as máscaras e retorna vetores de posição e direção da vareta (se a encontrar). Também
altera a ROI **/
void direcaoApontada(cv::Rect& ROI, cv::Mat& maskPink, cv::Mat& maskBlue, cv::Point& vetorPosicao,
cv::Point& vetorDirecao, unsigned char direcaoDeslocamento, unsigned char& varetaEncontrada) {
     // Encontra os centros dos clusters pink e blue - - - - - - - - - - - - - - - - - - - - - -
     std::vector<cv::Point> centroClustersP;
     std::vector<cv::Point> centroClustersB;
     geraClusters(maskPink, maskBlue, centroClustersP, centroClustersB);
     if(centroClustersB.size() == 0 || centroClustersP.size() == 0){
          varetaEncontrada = 0;
          return;
     }

     // Ordena clusters  - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
     // Clusters serão ordenados por proximidade ao último vetor posição encontrado.
     // vetorPosicao se encontra em coordenadas de câmera, enquanto centroClusters se encontram em
coordenadas da ROI.
     // Portanto realiza a seguinte conversão:
     cv::Point vetorPosicaoROI = vetorPosicao - cv::Point(ROI.x, ROI.y); // Posição em coordenadas
da ROI do vetor posição
     std::vector<pointDist> listaP(centroClustersP.size());
     std::vector<pointDist> listaB(centroClustersB.size());
     ordenaClusters(vetorPosicaoROI, centroClustersP, listaP);
     ordenaClusters(vetorPosicaoROI, centroClustersB, listaB);
     centroClustersP.clear();
     centroClustersB.clear();

     // Procura pela vareta  - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
     std::vector<pointDist> ptsVaretaP, ptsVaretaB; // os 5 pontos da vareta
     procuraVareta(listaP, listaB, varetaEncontrada, vetorPosicaoROI, ROI, ptsVaretaP, ptsVaretaB);

     // Altera ROI, vetorPosicao, vetorDirecao - - - - - - - - - - - - - - - - - - - - - - - - -
     if(varetaEncontrada){
          cv::Point pt1; // pt mais a esquerda ou mais em cima
          cv::Point pt2; // pt mais a direita ou mais em baixo
          geraVetoresPosicaoDirecao(ptsVaretaP, ptsVaretaB, ROI, pt1, pt2, vetorPosicao,
vetorDirecao, direcaoDeslocamento);

          trackVareta(pt1, pt2, ROI);
     }
}
```

SerialPort.cpp (Criado por: Manash Kumar Mandal**)**

```
// Fonte deste arquivo: https://blog.manash.me/serial-communication-with-an-arduino-using-c-on-windows-d08710186498
// Criado por: Manash Kumar Mandal
// Versão: 22 de maio de 2016
// Licenciamento: MIT

#include "arqInclusao.h"

SerialPort::SerialPort(char *portName)
{
	this->connected = false;

	this->handler = CreateFileA(static_cast<LPCSTR>(portName),
		GENERIC_READ | GENERIC_WRITE,
		0,
		NULL,
		OPEN_EXISTING,
		FILE_ATTRIBUTE_NORMAL,
		NULL);
	if (this->handler == INVALID_HANDLE_VALUE) {
		if (GetLastError() == ERROR_FILE_NOT_FOUND) {
			printf("ERROR: Handle was not attached. Reason: %s not available\n", portName);
		}
		else
		{
			printf("ERROR!!!");
		}
	}
	else {
		DCB dcbSerialParameters = { 0 };

		if (!GetCommState(this->handler, &dcbSerialParameters)) {
			printf("failed to get current serial parameters");
		}
		else {
			dcbSerialParameters.BaudRate = CBR_9600;
			dcbSerialParameters.ByteSize = 8;
			dcbSerialParameters.StopBits = ONESTOPBIT;
			dcbSerialParameters.Parity = NOPARITY;
			dcbSerialParameters.fDtrControl = DTR_CONTROL_ENABLE;

			if (!SetCommState(handler, &dcbSerialParameters))
			{
				printf("ALERT: could not set Serial port parameters\n");
			}
			else {
				this->connected = true;
				PurgeComm(this->handler, PURGE_RXCLEAR | PURGE_TXCLEAR);
				Sleep(ARDUINO_WAIT_TIME);
			}
		}
	}
}

SerialPort::~SerialPort()
{
	if (this->connected) {
		this->connected = false;
		CloseHandle(this->handler);
	}
}

int SerialPort::readSerialPort(char *buffer, unsigned int buf_size)
{
	DWORD bytesRead;
	unsigned int toRead = 0;

	ClearCommError(this->handler, &this->errors, &this->status);

	if (this->status.cbInQue > 0) {
		if (this->status.cbInQue > buf_size) {
			toRead = buf_size;
		}
		else toRead = this->status.cbInQue;
	}
```

```
     if (ReadFile(this->handler, buffer, toRead, &bytesRead, NULL)) {
          buffer[bytesRead] = '\0';
          return bytesRead;
     }

     return 0;
}

bool SerialPort::writeSerialPort(char *buffer, unsigned int buf_size)
{
     DWORD bytesSend;

     if (!WriteFile(this->handler, (void*)buffer, buf_size, &bytesSend, 0)) {
          ClearCommError(this->handler, &this->errors, &this->status);
          return false;
     }
     else return true;
}

bool SerialPort::isConnected()
{
     return this->connected;
}
```