Universidade Federal de Pernambuco

Centro de Tecnologia e Geociências

Departamento de Engenharia Mecânica

Daniele Wanderley Brooman

Identificação de Coordenadas para Manipulação de Braço Robótico por Visão Computacional

Recife

Novembro de 2018

Universidade Federal de Pernambuco
Centro de Tecnologia e Geociências
Departamento de Engenharia Mecânica

Daniele Wanderley Brooman

Identificação de Coordenadas para Manipulação de Braço Robótico por Visão Computacional

Trabalho de Conclusão de Curso apresentado à Universidade Federal de Pernambuco como requisito para a obtenção do grau de Bacharel em Engenharia Mecânica.

Orientador: Pedro Manuel Gonzalez del Foyo
Coorientador: Jacinaldo Balbino de Medeiros Junior

Recife
Novembro de 2018

À memória de Norma Lyra Brooman e Luce Mesquita Wanderley, que foram minhas maiores incentivadoras e hoje são minha maior inspiração.

Agradecimentos

Gostaria de agradecer à minha família, em especial aos meus pais Suzana Mesquita Wanderley e Gregory Alan Brooman, que me deu a base e as ferramentas para chegar até aqui. Ao meu namorado, Eduardo Corte Real Fernandes, por todo o apoio e ajuda durante esses anos. Aos meus amigos, novos e antigos, que me acompanharam e viram minha admiração pela tecnologia crescer com cada descoberta, em especial aos integrantes dos grupos *VExtreme*, *Central Perk* e Maracatronics. Aos amigos e colegas que estiveram comigo em Coventry University e me ajudaram a abrir uma nova porta cheia de possibilidades. Aos professores Pedro Manuel Gonzalez del Foyo, pela constante paciência e incentivo, e Jacinaldo Balbino de Medeiros Junior, por sua ajuda e por dar a ideia que iniciou este projeto.

Por fim, gostaria de agradecer a todos os professores que fizeram parte da minha trajetória e me ensinaram o amor pelo aprendizado.

"Every day you may make progress. Every step may be fruitful. Yet there will stretch out before you an ever-lengthening, ever-ascending, ever-improving path. You know you will never get to the end of the journey. But this, so far from discouraging, only adds to the joy and glory of the climb.- Winston S. Churchill

Resumo

Manipuladores robóticos são utilizados nas mais diversas áreas, desde a industria automotiva até a medicina. O avanço tecnológico vem permitindo novas maneiras para programa-los, porém estas podem requerer treinamentos específicos aos usuários, possuirem alto custo ou as vezes não estarem disponíveis para determinados modelos, limitando seu uso. Este trabalho tem como objetivo desenvolver um código que ajuda o usuário a obter as posições para as quais o manipular deve mover-se, simplificando o processo de programação. O sistema faz uso de visão computacional através de duas *webcams* de baixo custo, um computador para o processamento das imagens e a biblioteca gratuita de código aberto *OpenCV* para visão computacional. Ele utiliza as câmeras para identificar marcadores específicos e calcular suas coordenadas no espaço. A validação do código é realizada através da verificação as coordenadas calculadas pelo sistema de diversos pontos de teste em comparação com suas posições no mundo real e do cálculo do desvio padrão das medições. Foi feita, ainda, uma avaliação da influência que os padrões de calibração e outros fatores têm na calibração das câmeras, que pode afetar na precisão do sistema.

Palavras-chave: Manipulador robótico. Visão computacional. Calibração de câmeras. Obtenção de coordenadas. Programação. *OpenCV*.

Abstract

Robotic manipulators are used in a wide range of areas, from the automotive industry to medicine. The technological advance has allowed new ways to program them, however these may require specific trainings to the users, can be high cost or sometimes are not available for certain models, limiting their use. This work aims to develop a code that helps the user to obtain the positions to which the manipulate should move, simplifying the programming process. The system makes use of computer vision through two low-cost webcams, a computer for image processing, and the free OpenCV library for computer vision. It uses the cameras to identify specific markers and calculate their coordinates in space. Validation of the code is performed by checking the coordinates calculated by the system from several test points compared to their real-world positions and the calculation of the standard deviation of the measurements. An evaluation of the influence that the calibration standards and other factors have on the calibration of the cameras, which can affect the accuracy of the system, was also made.

Keywords: Robotic manipulators. Computer vision. Camera calibration. Coordinates acquisition. Programm. OpenCV.

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

TCP	*Tool Center Point*
CAD	*Computed-aided design*
RGB	*Red, green, blue*
HSV	*Hue, saturatoin, value*
GdL	Grau de liberdade
FPGA	*Field-Programmable Gate Array*
SI	Sistema Internacional de Unidades
RMS	*Root mean square*

Lista de símbolos

f	Distância focal
α	Ângulo em relação ao eixo x
β	Ângulo em relação ao eixo y
γ	Ângulo em relação ao eixo z

Sumário

1 Introdução

A mecanização do trabalho teve início durante a primeira revolução industrial, na Inglaterra do século XVII, com o surgimento de máquinas de fiar e dos teares para produzir algodão, e segue até os dias atuais, nos quais vivenciamos a terceira revolução industrial. O desenvolvimento tecnológico proporcionou uma melhoria na qualidade dos produtos e da condição de trabalho dos operários. A utilização das máquinas para a realização de tarefas repetitivas e desgastantes não eliminou completamente a necessidade do homem, mas permitiu que certas atividades fossem aceleradas.

O aprimoramento de equipamentos eletromecânicos, o desenvolvimento de controladores lógico programáveis e o crescente estudo sobre lógica computacional dos últimos 50 anos permitiram que a indústria aumentasse ainda mais sua eficiência através da automação industrial e a criação de manipuladores (ou braços) robóticos. Estes braços automatizados foram desenvolvidos visando executar tarefas difíceis ou perigosas, que não poderiam (ou deveriam) ser realizadas por um ser humano. Isso foi extremamente benéfico para as indústrias, que aumentaram a produção, agilizaram o processo, minimizaram os desperdícios e impulsionaram as vendas. (CARRARA, 2015)

De acordo com Carrara (2015), a automação industrial atual pode ser classificada em fixa, flexível e programável. A primeira se utiliza de equipamentos desenvolvidos para fins específicos e utilizados em grandes produções em massa. Esses equipamentos executam a mesma operação dezenas de vezes por minuto, como encher e tampar garras, embalar produtos e rechear biscoitos na indústria alimentícia. A automação flexível é utilizada quando o volume de produção não é tão alto e o equipamento pode ser programado para atuar em mais de uma atividade, ou na produção de produtos semelhante. Ela é bastante comum em linhas de montagem automotiva, onde, por exemplo, um mesmo braço robótico pode ser utilizado para prender parafusos em diferentes partes do carro a ser produzido. Por fim, a automação programável é indicada quando há um volume de produção baixa, porém com grande variedade de produtos, como é o caso da fabricação de peças por lote ou encomenda. Desta forma, o robô precisa ser altamente adaptável para produtos diferentes, e com isso sua programação pode se tornar bastante complexa.

A programação de um manipulador robótico consistem, em grande parte, em informar os pontos ou posições para quais o braço deve mover no espaço durante a operação. Os métodos atuais para obtenção dessas coordenadas, embora efetivos, apresentam algumas dificuldades. Em alguns casos, o processo é lento e requerer a utilização do próprio manipulador, de forma que a produção passa um maior tempo parada. Em outros, o sistema necessário é caro ou pode não estar disponível para certos modelos de braços, o

que limita o seu acesso em especial para empresas de menor porte e pesquisadores com pouco recurso.

1.1 Motivação

Braços robóticos podem substituir funcionários em atividades perigosas, altamente repetitivas e/ou desagradáveis. A maioria dos manipuladores é programado através de técnicas de "ensinar e repetir", onde o operador treinador (programador) usualmente utiliza um controle portátil (*robot teach pendant*) ou linhas de código para ensinar a atividade ponto a ponto. Como o programador deve identificar e informar cada um desses pontos, esse ainda é um processo que requer tempo e experiência técnica, além de limitar situações de uso do braço em tempo real onde é necessário que o braço adapte seu posicionamento de acordo com a situação. (BRAHMANI; ROY; ALI, 2013)

A identificação de posições por meio de auxílio visual permite que o braço em questão seja programado de maneira mais simples, reduzindo o nível de treinamento necessário, além de ter a capacidade de ser controlado em tempo real (modo *online*). Dessa forma, este trabalho visa desenvolver um sistema de baixo custo que auxilie e simplifique a programação de manipuladores robóticos. Esse sistema faz uso câmeras de vídeos (*webcams*) para identificar marcadores de cores pré-determinadas e calcular suas coordenadas no espaço para o usuário. O estudo de novos métodos para programação de braços robóticos possibilita uma redução de custos para empresas e pesquisadores que utilizam esse tipo de tecnologia. Para o trabalhador, um novo método que permita que ele trabalhe com movimentos mais naturais e utilizando todo o seu braço ao invés de movimentos repetitivos de dedos e punhos, significa uma melhor condição de trabalho. A possibilidade, ainda, de estar em uma área separada do manipulador mecânico e do material de trabalho traz, em alguns casos, um menor risco a saúde do funcionário que deve realizar a programação.

1.2 Objetivo Geral

Desenvolvimento de um código que permita auxiliar a programação de braços robóticos ao identificar pontos de interesse no espaço através da captação da posição de marcadores por meio de visão computacional.

1.3 Objetivos Específicos

- Especificar uma estrutura de posicionamento das câmeras;
- Tratar as imagens para identificação dos pontos de interesse;

- Calcular as coordenadas dos pontos de interesse em relação à base do braço;
- Permitir o envio das coordenadas dos pontos de interesse para o braço por dois métodos (*online* e *off-line*);
- Validação do código através do cálculo de coordenadas de pontos conhecidos.

1.4 Estrutura

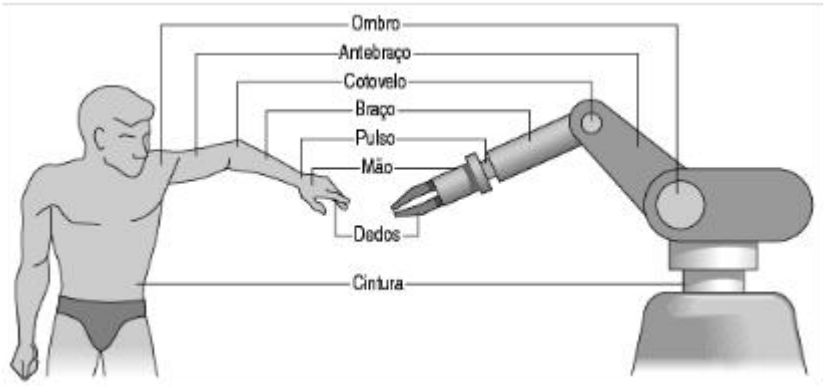
Este trabalho segue a seguinte estrutura:

- Os capítulos dois, três e quatro trazem uma revisão de manipuladores robóticos, de alguns dos métodos para controle e programação usados atualmente ou em processo de pesquisa para manipuladores, e de visão computacional, respectivamente;
- O quinto capítulo apresenta a proposta do projeto;
- O sexto capítulo explica os testes desenvolvidos para verificar a qualidade do programa e a análise dos resultados obtidos.
- O sétimo e último capítulo é composto pela conclusão sobre o trabalho aqui desenvolvido e sugestões para trabalhos futuros.

2 Manipuladores Robóticos

Segundo Rosario (2010), um manipulador robótico é composto por partes mecânicas motorizadas e um computador que controla seus movimentos. O computador é responsável por armazenar o programa que determina o curso que o manipulador precisa seguir, e deve ser capaz de interagir com os atuadores e sensores utilizados. O braço mecânico é projetado com o intuito de realizar diferentes tarefas e repeti-las com um alto nível de precisão. As características antropomórficas, geralmente utilizadas nos braços industriais, permitem uma fácil associação com o membro superior humano, simplificando a visualização e adaptação dos movimentos necessários na linha de produção. A Figura 1 abaixo mostra a analogia entre o mecanismo robótico e o corpo humano.

Figura 1 – Comparação de um braço robótico com o corpo humano.



Fonte: (ROSARIO, 2005)

Dois importantes pontos sobre o funcionamento de um braço robótico são: a percepção do meio pelo robô através de sensores; e a execução do programa de movimentação do mesmo. Os robôs podem, então, ser classificados como de 1ª geração, que não recebe informações do meio, realizando apenas os movimentos pré-programados, ou de 2ª geração, que pode receber informações do meio e realizar movimentos pré-programados de acordo. (ROSARIO, 2010).

2.1 Descrição de posição e orientação

Um manipulador possui corpos rígidos e juntas. Os corpos são ligados uns aos outros por meio das juntas, que permitem o movimento relativo entre eles, formando assim um sistema articulado. Uma das principais características de um mecanismo articulado 3D é sua capacidade de movimento nas seis coordenadas espaciais, que incluem translações

e rotações. As juntas permitem o movimento dos elementos adjacentes, mas de forma restringida. Pelo menos um tipo de deslocamento não é permitido entre dois elementos conectados, para que possam manter sua conexão. (SILVA, 2010)

Os braços podem possuir garra ou ferramenta no seu último elemento, conhecido como o atuador. De maneira geral, este é o elemento de maior importância no manipulador, pois é ele que irá realizar de fato a tarefa para qual o braço foi desenvolvido, como carregar um objeto de um ponto a outro, apertar parafusos, realizar soldas, entre outros.

Na área da robótica é de grande importância a capacidade de localizar os objetos no espaço 3D, sejam eles elementos pertencentes ao robô ou algo com o qual ele precise interagir. Dessa forma, é necessário poder descrever esse objeto através de dois atributos: posição e orientação no espaço. É preciso, ainda, representar essas informações de maneira que seja possível manipula-las matematicamente. É convencional definir um sistema de coordenadas local, conhecido como *frame* (ou quadro), preso aos elementos de interesse, como mostra a Figura 2. Dessa forma, para descrever o elemento, é dado a posição e orientação da sua *frame* em relação a um sistema de coordenadas global, geralmente definida na base do braço. No entanto, qualquer quadro pode servir como referencia para descrever os objetos de interesse, de forma que é comum a utilização de transformações ou mudanças nas descrições de atributos do corpo. (CRAIG, 2005)

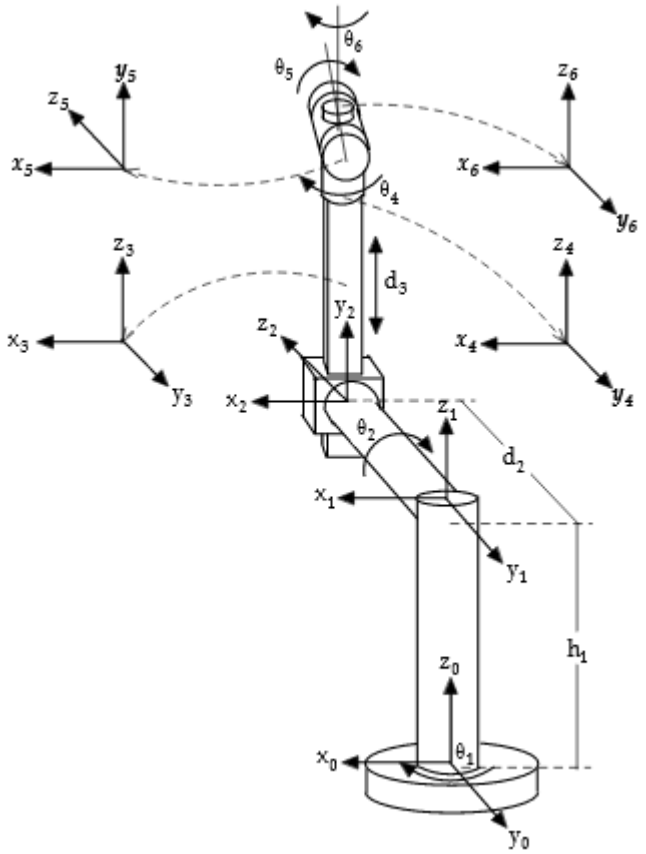
2.2 Cinemática de manipuladores

A cinemática é o estudo analítico do movimento de mecanismos sem levar em consideração a força e o torque que causam tal movimento. Sendo manipuladores robóticos sistemas desenvolvidos especificamente para movimentação, a cinemática do robô é um dos principais aspectos a ser considerado em seu desenho, análise, controle e simulação. A cinemática de um manipulador descreve a posição, velocidade e aceleração de seus elementos, assim como qualquer outra derivação de ordem mais alta que possa ser obtida a partir da pose do robô, sendo esta definida como a posição e orientação de todos os elementos rígidos do manipulador em um dado momento.(SICILIANO; KHATIB, 2008)

A cinemática de um mecanismo pode ser direta ou inversa. A primeira consiste em definir a posição e orientação do elemento final (no caso do braço robótico, o atuador) a partir do ângulo relativo formado em cada junta - uma vez que se presume que o comprimento dos elementos rígidos são conhecidos. A cinemática inversa, por outro lado, é o oposto. Sabendo-se a localização do atuador, é possível calcular a orientação dos demais elementos. As soluções da cinemática direta definem a área de trabalho do robô, ou seja, o conjunto de todos pontos no espaço que o robô é capaz de alcançar.

A cinemática inversa é algo que o corpo humano realiza milhares de vezes ao longo do dia, e é necessário criar um algorítimo para que o braço robótico tenha a mesma

Figura 2 – Exemplo de manipulador com 6 juntas e suas respectivas *frames* para identificação.



Fonte:(CUBERO, 2007)

capacidade. A habilidade de um robô de determinar a pose de seus elementos para obtenção do posicionamento do atuador da forma desejada é de grande importância. Anos atrás, muitos robôs não possuíam este algorítimo, de forma que era necessário mover todo o braço para a posição desejada e gravar a pose final em função da posição de suas juntas. Atualmente, dificilmente se encontrará um braço robótico industrial que não possua essa capacidade, sendo agora possível ao operador apenas informar ao computador do manipulador pra qual ponto deseja que o atuador seja movido. O computador, por sua vez, irá realizar todos os cálculos para tentar determinar a orientação dos elementos rígidos e executar o movimento. As equações necessárias para esse algorítimo, no entanto, são não-lineares, e por isso suas soluções podem ser difíceis de se obter, e algumas vezes até impossíveis. (CRAIG, 2005)

2.3 Programação de robôs

Manipuladores robóticos se diferenciam de outras máquinas da indústria pela sua capacidade de mudança (flexibilidade) permitida por meio da programação. A linguagem de programação de um robô industrial serve como comunicação entre o operador e o manipulador. Através dela, o usuário pode, geralmente, definir um ponto do atuador como o "ponto operacional", conhecido como TCP (do termo em inglês, *Tool Center Point*), e um sistema de coordenadas para referência. (CRAIG, 2005)

Graças a capacidade do robô de calcular sua cinemática reversa, o operador pode definir a movimentação do robô através da localização desejada do TCP em relação ao sistema de coordenadas selecionado. É possível gravar vários pontos ao longo da trajetória ou apenas o inicial e final. O primeiro método é preferível, uma vez que ajuda a evitar que o robô siga percursos desconhecidos, que possam acarretar em acidentes caso haja outros objetos dentro da área de trabalho. O operador muitas vezes também pode definir a velocidade com a qual deseja que o braço se movimente em determinados trechos da trajetória, entre outros critérios, dependendo dos construtores permitidos na linguagem utilizada.

Essa linguagem, no entanto, varia de acordo com vendedor do equipamento. Cada fabricante de braços robóticos possui uma linguagem proprietária que roda em componentes de *hardware* proprietários. Dessa forma, o desenvolvimento de uma plataforma genérica que possa ser utilizada para diversos manipuladores, de marcas diferentes, se torna desejável, porém desafiador. (LEWIS; DAWSON; ABDALLAH, 2004)

2.3.1 Programação *off-line*

Alguns manipuladores possuem ambientes de programação que permitem que o operador escreva o programa sem acessar diretamente o robô, ou seja, de maneira *off-line*. Um ponto positivo desse tipo de ambiente é permitir que se faça alterações em uma linha de montagem sem que esta precise estar parada durante todo o tempo necessário para que a programação seja escrita, permitindo que fábricas automatizadas permaneçam menos tempo paradas. Outra vantagem é poder fazer uso direto dos desenhos computacionais (CADs) do produto a ser manufaturado, o que pode reduzir significativamente o tempo necessário para a programação, uma vez que o operador pode passar informações de posicionamento e orientação da ferramenta com uma maior precisão. (CRAIG, 2005)

A programação offline, em geral, requer que o operador possua um treinamento maior do que para o método tradicional. Além disso, o programador precisa possuir informações precisas sobre as coordenadas das posições que o atuador teve tomar no espaço. Nos casos onde não é possível fazer uso dos CADs e/ou simuladores do manipulador, esse trabalho se torna difícil, pois um erro pode acarretar danos a peças, ferramentas

e ao próprio manipulador. Uma maneira de contornar este problema, no entanto, é o programador informar uma posição próxima (porém segura) do ponto desejado, e, uma vez que passar o programa para o braço, move-lo para corrigir o posicionamento. Apesar desse método não poder ser considerado uma programação completamente *off-line*, ele ainda consome menos tempo do que fazer a programação inteiramente conectado ao robô, significando menos tempo do maquinário parado.

3 Métodos de Controle e Programação de Manipuladores

Formas mais flexíveis de utilização de manipuladores robóticos têm sido introduzidas nos ciclos de manufatura. Manipuladores mecânicos programáveis estão realizando tarefas como solda localizada, pintura a *spray*, manuseio de materiais e montagem de componentes. Para produções de baixo volume, o trabalho manual tem sido, até então, o que apresenta uma melhor relação *custo* x *benefício*. Com o aumento da demanda de produção, no entanto, chega-se a um ponto onde isso muda, e passa-se a dar preferência aos manipuladores robóticos. (BRAHMANI; ROY; ALI, 2013)

Duas outras indústrias, além da automotiva, onde os manipuladores mecânicos têm se mostrado de grande utilidade são a química e alimentícia. Ambas precisam manipular materiais sem riscos de contaminação, seja do próprio produto ou do meio externo. Em casos como esses, é interessante um sistema que permita ao manipulador, dentro de uma área isolada, ser controlado por um operador fora da área de risco de contaminação.

De acordo com a necessidade industrial, diversos métodos de programação vêm sendo desenvolvidos. Atividades que antes se pensava serem impossíveis de serem automatizadas devido ao seu auto nível de não linearidade, estão sendo realizadas por manipuladores autônomos com o auxílio de câmeras de vídeo. Em outros casos, estão sendo utilizados *joysticks* e sensores inerciais para que o operador controle diretamente os braços robóticos. No entanto, em casos onde o mecanismo deve efetuar repetidamente o mesmo exato movimento por vários dias, esses métodos representam gastos desnecessários, e se costuma utilizar a programação ponto a ponto.

3.1 Programação direta

A programação direta é principalmente utilizada quando o manipulador deve executar um trabalho repetitivo. Há três modos básicos de se programar robôs industriais conhecidos como *lead through*, *teach method* e programação *off-line*.

O primeiro desses métodos consiste em o operador mover o equipamento manualmente. O controlador do robô grava a posição de cada uma das juntas a intervalos de tempo determinados, repetindo a sequência posteriormente. Esse método pode se tornar bastante complicado a depender do tamanho do manipulador. Alguma vezes, usa-se um modelo em escala apropriada para realizar a programação. Esse método foi muito popular no início do surgimento de robôs industriais, mas praticamente desapareceu nos dias atuais.

(BRITISH AUTOMATION AND ROBOT ASSOCIATION,)

O método de ensino (*teach method*) é o mais comum atualmente. Ele é bastante semelhante ao primeiro método, porém o manipulador agora é movido através de botões ou chaves até a posição desejada. Uma vez que o operador posicione o robô no local desejado, ele deve solicitar que o controlador grave o conjunto de informações das juntas e do atuador. Para um melhor desempenho, aconselha-se que sejam marcado vários pontos ao longo da trajetória desejada, e não apenas o inicial e o final. (ACERVOSABER, 2015)

Por fim, o último método é um processo que não necessita que o usuário esteja no mesmo local que o manipulador propriamente dito para que seja feito o programa, ao contrário dos dois anteriores. Essa característica pode ser bastante útil dentro de uma indústria, uma vez que requer menos tempo em que a linha necessita ficar fora de operação para a troca de código do robô. Principalmente para um pequeno ou médio volume de produto, é possível fazer toda a programação e apenas parar a linha por um curto intervalo de tempo, o suficiente para transferir o programa e reiniciar o sistema. Isso aumenta a viabilidade econômica da implementação. O fato da programação ser feita longe do manipulador também aumenta a segurança para o operador. Com um sistema único de programação para mais de um robô, é necessário um menor custo com treinamento de operadores, pois eles não precisam lidar com as diferenças entre os controladores. Alguns sistemas permitem ainda que o braço robótico seja programado a partir de desenho computacional, onde um modelo do equipamento pode ser utilizado para simulação de trajeto, minimizando o risco de colisões. (NOF, 1999)

3.1.1 Painel de ensino robótico

Atualmente, a maneira mais utilizadas para definir uma nova programação para robôs em fábricas é através do *robot teach pendant*. Ela consiste em utilizar painéis de controle de robôs, como o da Figura 3, para aplicar o segundo método de programação direta apresentado acima. Os primeiros painéis possuíam um alto número de botões, porém poucas funcionalidades. Com o tempo, os *pendants* foram se tornando mais amigáveis para o operador, com o número de botões diminuindo e a quantidade de funções disponíveis aumentando. Atualmente, com a incorporação das telas sensíveis ao toque, a interface homem-máquina evoluiu consideravelmente. (ROBOTIQ COMPANY, 2015)

O painel de acionamento permite que o robô seja programado ponto a ponto, de modo que o operador pode controlar o movimento desejado. O programador deve posicionar o manipulador na posição desejada e salvar a mesma, sendo necessário indicar os pontos iniciais e finais do processo. Informações de funcionalidades e movimentos podem ser adicionados nos pontos necessários. É possível também fazer com o que o robô espere naquela determinada posição por um certo tempo ou até que uma informação externa chegue através de um sensor. (NOF, 1999)

Figura 3 – Controlador de braço robótico (*robot teach pendant*) da marca COMAU.

Fonte: (COMAU,)

3.2 *Joystick*

Joysticks são utilizados tanto para ensinar movimentos ao robô quanto para operações onde o equipamento é manipulado inteiramente por um operador capacitado. Uma das desvantagens do uso do painel de controle é que o programador precisa muitas vezes desviar os olhos do manipulador robótico para identificar o botão correto a ser pressionado. Este problema pode, em grande parte, ser eliminado pelo uso de um *joystick*. As teclas de movimento são montadas diretamente nele, de modo que é possível o acionamento de uma ou mais juntas sem que o controlador precise desviar sua atenção. (ACERVOSABER, 2015)

Os sistemas de *Joysticks* mais simples apresentam apenas um ou dois graus de liberdade (GdL), enquanto os mais sofisticados podem ter seis graus. No entanto, quanto maior a número de graus de liberdade, mais difícil se torna realizar movimentos desacoplados (onde o movimento de uma junta não interfere no da outra). É usual, então, utilizar dois *joysticks* de três GdL cada, que torna o processo mais simples, como o mostrado na Figura 4.

Fora auxiliar na programação de atividades repetitivas, o *joystick* foi introduzido também para controla outros tipos de manipuladores. Na área médica, algumas cirurgias já são realizadas com o auxílio de equipamentos robóticos, como o robô Da Vinci (*The Da Vinci Surgical System*). Como cada procedimento é diferente do outro, o médico controla o braço de modo *online* durante todo ou grande parte do tempo. (NOF, 1999)

O manuseio desse tipo de controlador deve ser como a fluência em uma língua:

Figura 4 – Unidade de controle robótico com dois *joysticks*, da Thompson Automation Systems.

Fonte: (DPA MAGAZINE, 2013)

enquanto é necessário que a pessoa pare para pensar e traduzir o movimento desejado pelo manipulador em ações a serem feitas no *joystick*, ela ainda precisa de treinamento. Mesmo para os controles de apenas três graus de liberdade, pode-se levar um certo tempo para o operador conseguir manipular correta e perfeitamente o braço robótico por meio de *joysticks*, principalmente se dois devem ser utilizados ao mesmo tempo para suprir todos os graus de liberdade do manipulador. Quanto mais complexo e/ou preciso o movimento ou sistema, maior a necessidade de treinar o funcionário para lidar com ele. Isso significa gastos de recursos monetários e de gerenciamento de tempo.

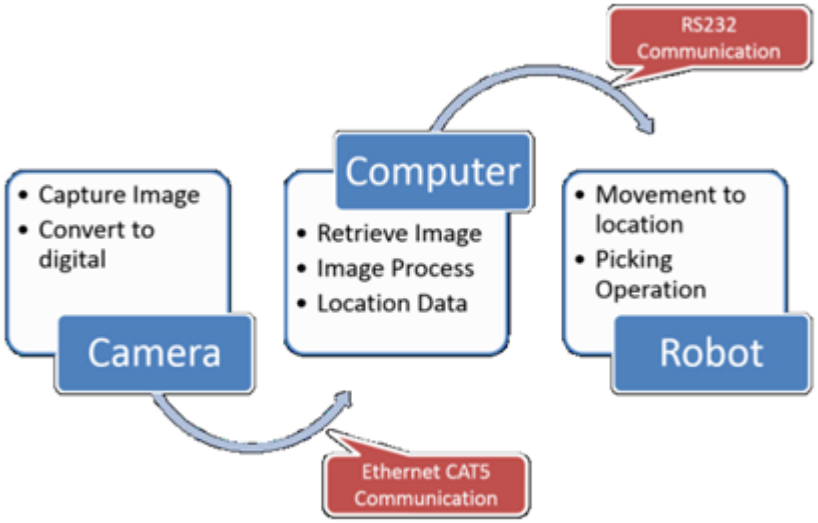
3.3 Sensores visuais

Já há algum tempo que se tenta dar olhos aos manipuladores mecânicos. O uso de câmeras para identificação dos pontos para os quais o braço deve se mover permite uma grande redução do trabalho do operador. Com o objetivo de minimizar o tempo e dinheiro necessário para a programação do manipulador, criaram-se diferentes técnicas de tratamento de imagem e posicionamento de câmeras. Em casos como estes, geralmente, o que importa é o objetivo final, e não a movimentação do manipuladro ser feita de uma determinada maneira (SZABO; GONTEAN, 2015). A Figura 5 mostra a estrutura geral do sistema. Os passos são a captura e conversão da imagem, o processamento da imagem e identificação das informações (localização de objetos) e, por fim, a movimentação e ação do robô para executar a tarefa.

Um dos modos mais comuns na indústria é a colocação da câmera em um local fixo, permitindo a determinação de um ponto de referência global. Após a aplicação de

marcadores, filtros e operadores morfológicos, é possível identificar o objeto alvo assim como a ponta do manipulador. Através do cálculo do centroide dessas peças, é possível fazer com que o manipulador pegue o objeto alvo simplesmente minimizando a distância entre os dois centroides. (CABRE et al., 2013)

Figura 5 – Estrutura geral de sistema de manipulação baseada em câmeras.



Fonte: (SUBRAMANIAM; FAI; MING, 2014)

Outro método utilizado é a colocação da câmera precisamente no elemento final do manipulador mecânico. Isso pode ser uma vantagem quando se foca em objetos dos quais o manipulador se aproxima. Esse sistema não precisa fazer cálculos de cinemática reversa desde que a câmera possa estimar distâncias. Isso significa uma redução na necessidade de processamento. Essa posição de câmera permite uma área de visão muito mais restrita do que o método anterior. No entanto, caso o manipulador robótico possua vários dedos, é possível utilizar mais de uma câmera nas terminações. Dessa forma, é possível reconstruir o objeto em terceira dimensão através de um sistema paralelo que utiliza as imagens capturadas. Esse métodos necessita de um alto poder de processamento, o que pode ser bem difícil em uma plataforma dedicada. No entanto, com o avanço na área de processadores dedicados e *FPGA*s, acredita-se que é apenas uma questão de tempo. (SZABO; GONTEAN, 2015)

Um dos principais problemas ao lidar com a manipulação de braços robóticos a partir de câmeras é o processamento que deve ser feito nas imagens antes que qualquer informação seja retiradas para o sistema. É necessário tratar a imagem capturada para obter a correta identificação dos objetos desejados. A iluminação no local deve ser uniforme e constante, mas nem sempre isso é possível. São utilizados equalizadores de cor, que fazem uso da iluminação e espaço de saturação de cor para minimizar as mudanças no meio. É

necessário então reduzir o ruído da imagem e utilizar filtros computacionais para obter informações dos pixel. Por fim, o objeto alvo é identificado através de grupos de pixel de determinada cor e dimensão (TITARMARE; KATKAR; AGRAWAL, 2014). Geralmente, são utilizados marcadores de cores bem específicas para as peças desejáveis e cores que sejam facilmente eliminadas para demais objetos. Um exemplo é uma bola de cor escura em uma mesa branca.

3.4 Identificação de movimentação humana

Com o avanço no estudo de sensores, se torna cada vez mais viável a identificação de movimentos humanos para auxiliar na programação de robôs. Uma vez que se é capaz de transformar movimentos em informações digitais, é possível desenvolver um sistema que reproduza essa informação fisicamente. A maioria dos dispositivos consegue detectar a movimentação do corpo de maneira não invasiva, sendo segura sua utilização pelos operadores.

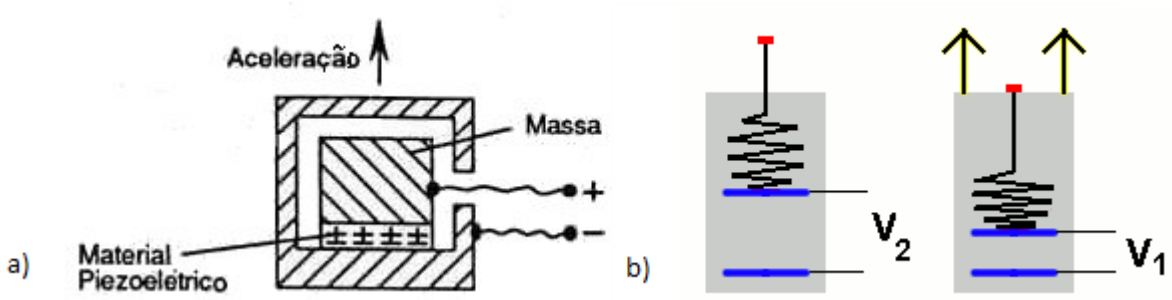
3.4.1 Acelerômetro

O acelerômetro é um sensor inercial capaz de identificar a aceleração do corpo ao qual está preso. Ele se baseia na primeira e segunda leis de Newton, que dizem que um corpo tende a se manter parado até que lhe seja aplicada uma força e que essa força gera uma aceleração. Sabendo-se a aceleração e o peso do corpo, é possível calcular a força aplicada.

Atualmente utilizam-se acelerômetros eletrônicos, que podem ser de vários tipos, sendo os dois mais usuais os de material piezoelétrico e o capacitivo. Ambos contêm uma massa interna a ser acelerada e produzem uma voltagem proporcional à aceleração do objeto estudado, que pode ser medida de modo analógico por um controlador. A diferença entre eles é o modo como é feita a transformação da informação mecânica (aceleração) em eletrônica (voltagem). O primeiro utiliza material piezoelétrico, que tem a propriedade de gerar uma diferença de potencial proporcional a pressão mecânica sob a qual se encontra. Essa pressão é produzida pela movimentação sofrida pelo sensor. O segundo faz uso de capacitores variáveis, geralmente mudando a distância entre as duas placas capacitivas quando a massa interna do acelerômetro é movida (LYNCH et al., 2003). A Figura 6 apresenta dois desenhos que mostram, de maneira simplificada, o princípio de funcionamento de acelerômetros que fazem uso de material piezoelétrico (a) e capacitivo (b). A construção dos modelos comerciais não são exatamente como esses, variando de acordo com o fabricante, a aplicação e a precisão desejada.

Entre suas muitas aplicações, sensores medidores de aceleração podem ser utilizados para reconhecimento de movimentos e gestos. Em vez de se mapear a velocidade ou

Figura 6 – Modelos simplificados de funcionamento de um acelerômetro a) piezoelétrico e b) capacitivo.



Fonte: (SEARA DA CIENCIA,)

transformar a informação no domínio de frequência, os dados são segmentados e organizados no domínio do tempo. Utilizando um sistema de sensoriamento capaz de reconhecer movimentos nos três eixos cartesianos, é possível identificar que gestos foram feitos. Movimentos verticais (para cima e para baixo), horizontais (para um lado ou outro), circulares, entre outros, produzem uma sequencia específica de dados em cada eixo que pode ser facilmente reconhecida. Ao projetar as informações recolhidas nos devidos planos, é possível analisar a mudança de sinal das curvas geradas (sem necessariamente se ater ao formato exato da curva) e, com isso, através de comparações com um banco de dados, selecionar o movimento mais provável de ter sido realizado.(XU; ZHOU; LI, 2012)

Uma vez que o movimento foi identificado pelo controlador, é possível ativar os atuadores de um braço robótico de modo pré-determinado para reproduzir tal movimento. Na comparação com outros métodos de manipulação, este é bastante simples para se trabalhar, pois toda a programação dos atuadores já está incluída no controlador. O operador precisa apenas se preocupar em movimentar seu próprio braço, de uma maneira natural. Assim, não é necessário um especialista, e o funcionário pode controlar o robô de maneira rápida e fácil. (BRAHMANI; ROY; ALI, 2013)

Além da melhoria na operação do equipamento, o acelerômetro traz ainda vantagens por possuir qualidades importantes na área industrial. Ele costuma ser bastante robusto e estanque, ideal para trabalho em locais fabris, além de ser resistente a choque, quedas, umidade, poeira, óleos, entre outros fatores ambientais e de manuseio. Por ser pouco sensível a campos magnéticos, também se enquadra para trabalhos próximos a grandes motores elétricos. Além disso, ele é, em geral, pequeno e leve, de modo que não prejudica a saúde do operador que o tenha preso ao braço. (SEQUEIRA, 2013)

3.4.2 Sensores musculares

Cada vez mais estão surgindo métodos para determinação de características biomecânicas, e uma grande parte disso se deve ao estudo do sistema muscular. A maioria dos sensores, no entanto, só pode identificar uma propriedade em um músculo por vez e sob determinadas condições. Por isso, os testes são de forma geral feitos em laboratório, e alguns de forma invasiva. Os métodos se classificam principalmente entre: medição de força e torque; e velocidade de movimentos ou movimentos de partes específicas. O primeiro grupo tem apresentado problemas técnicos, geralmente solucionados com o auxílio de outros dispositivos, e por isso ainda não demonstram grande vantagem ao serem usados em aplicações gerais. (ĐORđEVIć et al., 2011)

O segundo grupo apresenta uma aplicabilidade muito maior para a indústria, uma vez que permite a identificação de certos movimentos. Assim como no caso do acelerômetro, é possível identificar o tipo de movimento que se deseja realizar através dos sensores musculares. A ativação neurológica das células musculares produzem um potencial elétrico, que pode ser medido. O monitoramento dessa atividade elétrica é conhecida como eletromiografia, e é largamente utilizado para estudos de movimentação e verificação de função muscular durante reabilitação. Pesquisas mais recentes utilizam sensores musculares para controle de próteses móveis e demonstram que é possível diferenciar gestos da mão e dos dedos. (BENKO et al., 2009)

Os estudos agora tentam aumentar a capacidade de um sistema composto por sensores musculares para identificar movimentos do braço em um espaço livre. Previamente, a maioria dos projetos desenvolvidos estavam restringidos a uma superfície de contato, o que não era abrangente o suficiente para aplicações no mundo real. Um outro problema que também precisa ser abordado é a questão da calibragem dos sensores, que precisa ser refeita para cada indivíduo e, as vezes, entre reposicionamento dos sensores na pele. (SAPONAS et al., 2009)

3.4.3 Sensor visual

Reconhecimento de mãos e rosto são um caso especial de problemas que lidam com multi-corpos. Caso a informação desejada seja apenas o movimento da mão como um todo, e não de seu formato exato, uma abordagem de rastreamento clássico pode ser adotada. Grande parte dos estudos tem, até então, tentado solucionar o problema como o rastreamento de múltiplos corpos não rígidos. Com o desenvolvimento de equipamentos como o *Kinect* e outros sensores de três dimensões, técnicas de rastreamento de mãos e corpos em tempo real começou a ser explorado. Esses sistemas permitem minimizar problemas com iluminação e ruídos de imagem. (JIAN; DUERSTOCK; WACHS, 2014)

Tem-se utilizado reconhecimento de gestos e movimentos por câmeras em diversas

aplicações, como vigilância inteligente, melhorando o desempenho de sensores de presença, realidade virtual, para jogos, estúdios visuais e teleconferências, e interação social, entre outros (GRAVILA, 1999). Ainda está em estudo e desenvolvimento a aplicação de sensores visuais para reconhecimento de movimentos humanos dentro da indústria. Esse tipo de sistema tem a capacidade de trazer muitas das qualidades dos sensores de aceleração, porém com o diferencial de que pode identificar distâncias percorridas com maior precisão, além da localização dos elementos no espaço. Isso permite que o operador determine com maior precisão a trajetória desenvolvida pelo manipulador robótico, dando também flexibilidade além dos padrões de movimentos pré tabulados.

Infelizmente, uma grande dificuldade no trabalho com esse tipo de sensor é o tratamento adequado das imagens. A iluminação para a calibragem das cores, no caso de câmeras coloridas, a identificação das informações desejadas em arquivos de imagem de tamanho adequado e a frequência de obtenção de imagens para que o movimento possa ser identificado com uma maior precisão requerem um ambiente estável e um sistema com capacidade de processamento computacional maior do que os controladores robóticos costumam ter.

4 Visão Computacional

"Visão computacional é a ciência de dotar computadores e outras máquinas com visão, ou a habilidade de ver"(LEARNED-MILLER, 2011, p. 2). Um sistema de visão computacional processa as imagens provenientes de câmeras eletrônicas, podendo obter informações assim como o cérebro humano. Comercialmente, é possível a sua utilização na industria, inspecionando produtos, na medicina, para auxilio em diagnósticos, na ciência forense, na identificação automatizada de faces e texturas das iris de uma pessoas, entre outras. Sendo necessário apenas uma câmera, um computador e a interface entre eles, o custo de um sistema de visão computacional caiu consideravelmente com o avanço da tecnologia. Cada vez é mais fácil (e barato) o acesso a computadores com grande capacidade de processamento de imagens e gráficos. (NIXON; AGUADO, 2002)

Atualmente, a aplicação do processamento de imagens se estende por diversa áreas, desde a indústria às mídias sociais. Ele permite que o usuário obtenha informações ou modifique as mesmas. Através de padrões de reconhecimento, um programa computacional pode identificar se há pessoas em uma cena, ou quantos carros passam por um determinado trecho de rua por hora. É possível alterar as imagens, suavizando contornos, eliminando falhas ou até modificar a cor de objetos para realçar e facilitar a identificação por um operador.

Segundo Szeliski (2011), um grande fator de dificuldade na visão computacional é o fato dele ser um "problema inverso", onde se tenta descrever os objetos vistos em uma ou mais imagens e reconstruir suas propriedades como coloração, formato, iluminação, etc. Imagens, em geral, possuem dados insuficientes para solucionar a maior parte dos problemas, por isso pode ser necessário unir essas informações com modelos físicos e/ou probabilísticos para ajudar a eliminar ambiguidades no processamento. Quando uma pessoa vê o retrato de um conhecido, ela é capaz de associar o rosto a um nome. Quando o computador recebe o mesmo retrato, ele não é capaz de por si só localizar um rosto ou definir a quem ele pertence. Antes, é necessário ensinar a ele que formas na imagem podem ser consideradas um rosto, e, a partir daí, que características dentro dessa forma diferencia uma pessoa da outra.

4.1 *OpenCV*

O *OpenCV* é uma biblioteca de código aberto para visão computacional e aprendizado de máquinas contendo mais de 2500 algorítimos otimizados que podem rodar em diferentes sistemas operacionais. Suas funções permitem desde coisas simples como identificar a cor de um objeto a aplicações complexas como identificar cenários e marcadores

para interação com realidade aumentada. Seu objetivo é facilitar o desenvolvimento de aplicações que façam uso de visão computacional através do uso disseminado de uma infraestrutura comum. Dessa forma, ela é utilizada tanto por estudantes como por grandes empresas como *Google* e *Sony*. (OPENCV, 2018)

De acordo com Bradski e Kaehler (2008), a biblioteca permite que milhares de pessoas atuem de maneira mais produtiva na área de visão computacional, proporcionando a alunos e profissionais ferramentas que antes só estavam disponíveis em laboratórios de pesquisa. O *OpenCV* foi desenvolvido visando a eficiência computacional e tem um grande foco em aplicações em tempo-real, o que permite seu uso nas áreas de automação e robótica, principalmente se o usuário tiver acesso a computadores com mais de um núcleo de processamento. Por ter uma licença de código aberto, ela pode ser utilizada em produtos comerciais. No entanto, não é imposto que o fabricante disponibilize seu código ou lucros para o público.

O uso desta biblioteca permite uma infinidade e projetos, pois ela trata da matemática por trás da visão computacional. Ela é amplamente utilizada para identificar objetos e pode manipular imagens através de filtros com este propósito. Fazendo uso de uma sequencia de imagens ou de imagens de mais de uma câmera, ela permite calcular distâncias, identificar movimentos, modelar objetos 3D e até identificar imagens semelhantes em um banco de dados. Apesar de ter sido escrita inicialmente em linguagem *C/C++*, atualmente ela pode ser utilizada também em outras linguagens como *Python*, *Java*, *Ruby*, entre outras. É possível utiliza-la em diversos sistemas operacionais como no *Windows*, *iOS*, *Linux* e *Android*.

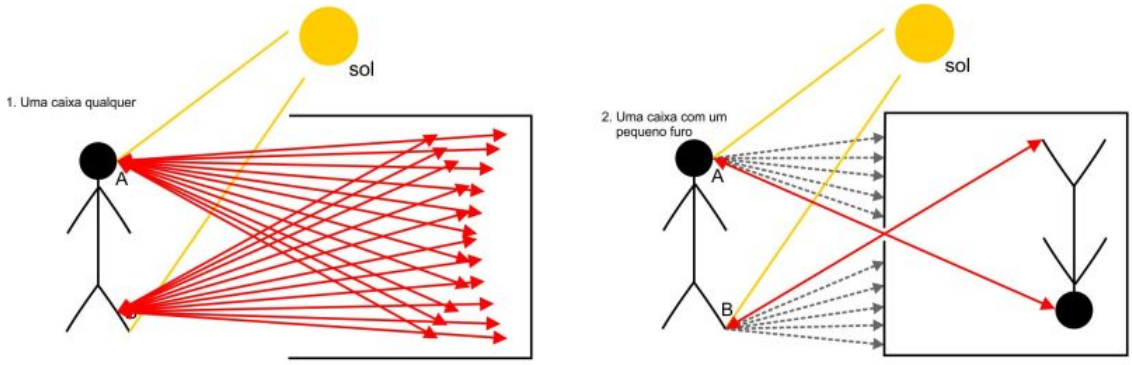
4.2 Câmeras

4.2.1 Modelo *Pinhole*

Uma fotografia é produzida a partir dos raios de luz refletidos nos objetos da cena que chegam ao obturador da câmera, semelhante ao que acontece com o olho humano. Um modelo bastante utilizado por simplificar a geometria da emissão desses raios é o da câmera *pinhole* (também conhecida como caixa escura ou câmera estenopeica). Esse modelo consiste de uma parede imaginária (ou caixa) com um pequeno buraco. A parede (ou lateral da caixa) bloqueia todos os raios de luz exceto os que passam pelo furo, que formam a imagem captada no chamado *plano de imagem* ou *plano de projeção*, como mostra a Figura 7. (BRADSKI; KAEHLER, 2008)

A representação matemática do modelo da caixa escura pode ser vista na Figura 8. A relação de semelhança de triângulos permite trazer o plano de projeção (*image plane*) para um local entre o centro de projeção, que representa o furo na parede no modelo *pinhole*

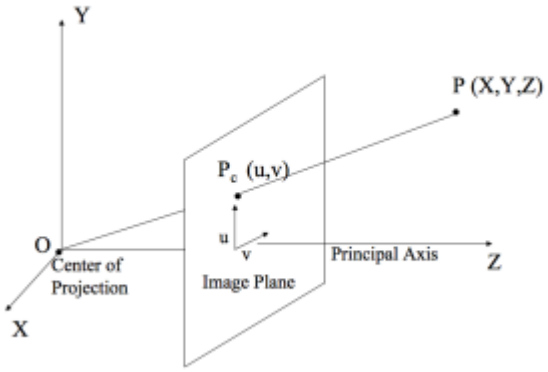
Figura 7 – Demonstração do funcionamento do modelo de câmera *pinhole*.



Fonte: (APRENDA FOTOGRAFIA, 2014)

e o centro da câmera real (*center of projection*), e o ponto (P) do objeto a ser projetado. Nessa representação é fácil notar que o ponto P_c de coordenadas (u,v) na imagem é o ponto de interseção da reta (que passa pelo ponto P do mundo real e o centro de projeção) com o plano de imagem. Como todos os pontos pertencentes a essa reta serão representados no mesmo ponto do plano de imagem, perde-se a informação de profundidade. Dessa forma, com apenas uma imagem do ponto P não é possível saber com exatidão qual são suas coordenadas no mundo real, mas sim a qual reta ele pertence.

Figura 8 – Representação geométrica do modelo de câmera *pinhole*.



Fonte: (PERPETUAL ENIGMA, 2014)

A distância entre o centro de projeção e o plano de projeção na Figura 8 é definido como a distância focal (f) da câmera. O sistema de coordenadas é definido como a origem no centro de projeção, e o plano XY é paralelo ao plano de imagem. O eixo Z é conhecido como eixo principal, e o ponto onde ele intercepta o plano de projeção recebe o nome de ponto principal. Dessa forma, todos os pontos da imagem possuem coordenadas (x, y, f)

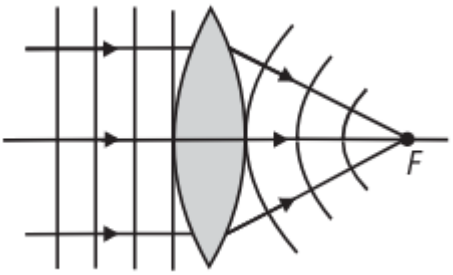
com referência ao centro da câmera. (BRINK, 2017)

4.2.2 Distorções nas lentes

Câmeras estenopeicas não apenas existem e foram usadas por vários anos como este ainda é um dos modos mais simples de se tirar fotografias. No entanto, devido a pequena quantidade de luz que consegue penetrar no furo de alfinete utilizado, o processo de formação da imagem é demasiadamente lento. Sistemas de visão computacional muitas vezes precisam avaliar mudanças em uma cena, seja para contabilizar o número de pessoas que passam, seja para auxiliar na movimentação de um robô. Nesse quesito, câmeras *pinhole* não são as mais eficientes.

Para acelerar a formação da imagem, é necessário uma maior quantidade de luz focada no ponto de projeção. Isso pode ser obtido através da utilização de lentes convexas, que são capazes de coletar luz de uma área maior e converge-la para o foco da lente (Figura 9). A utilização de lentes, no entanto, introduz um novo problema: há distorções nas imagens produzidas. Devido à dificuldade em fabricar uma lente matematicamente exata e em alinhar-la com a imagem, nenhuma lente é perfeita. As duas principais distorções causadas por lentes reais são a distorção radial, resultado do formato da lente, e a distorção tangencial, resultado do alinhamento na câmera. (BRADSKI; KAEHLER, 2008)

Figura 9 – A lente convexa é capaz de convergir luz para seu ponto focal.



Fonte: (PEDROTTI et al.,)

De acordo com Bradski e Kaehler (2008) a distorção radial causa uma curvatura na imagem na proximidade das bordas, fenômeno conhecido como tunelamento ou "olho de peixe". Devido a imperfeições físicas na lente, raios de luz que atingem pontos afastados do centro acabam convergindo para um local mais próximo que o ponto focal. Assim, a distorção é zero no centro óptico da imagem e aumenta ao longo que o ponto se move para a periferia. Para *webcams* simples, costuma-se utilizar dois fatores (k_1 e k_2) para descrever esta relação, e para câmeras com lentes de altas distorções é utilizado também um terceiro terceiro termo (k_3). Assim, a posição radial de um ponto de coordenada (x, y) na imagem

passa a ser:

$$x_{corrigido} = x(1 + k_1 r^2 + k_2 r^4 + k_3 r^6)$$
$$y_{corrigido} = y(1 + k_1 r^2 + k_2 r^4 + k_3 r^6)$$

Enquanto isso, a distorção tangencial pode ser caracterizada por apenas dois parâmetros (p_1 e p_2), totalizando cinco coeficientes de distorção.

$$x_{corrigido} = x + [2p_1 y + p_2(r^2 + 2x^2)]$$
$$y_{corrigido} = y + [p_1(r^2 + 2y^2) + 2p_2 x]$$

4.2.3 Parâmetros intrínsecos

Em um sistema de visão computacional, a unidade de medida na imagem é o pixel. Câmeras digitais de baixo custo costumam produzir imagens com pixel retangulares, e não quadrados. Isso gera duas distâncias focais diferentes, (f_x e f_y), que são a distancia focal f multiplicada pelo tamanho do pixel (s_x x s_y) em cada eixo. Também é improvável que o centro da imagem coincida com o ponto principal da imagem, devido a dificuldade de fabricação. Assim, é indicada a introdução de dois parâmetros (c_x e c_y) para modelar esse deslocamento. Esses quatro parâmetros são conhecidos como os parâmetros intrínsecos da câmera. A localização na imagem (na tela do computador) do ponto P de coordenadas reais (X, Y, Z) pode ser dada pelas duas equações a baixo. (BRADSKI; KAEHLER, 2008)

$$x_{pixel} = f_x(X/Z) + c_x$$
$$y_{pixel} = f_y(Y/Z) + c_y$$

4.2.4 Homografia plana

De acordo com Bradski e Kaehler (2008), homografia planar na visão computacional é a projeção de um plano em outro, como por exemplo a imagem de um objeto bi-dimensional em uma câmera. Considerando um ponto P_o no objeto, a sua imagem P_i na câmera pode ser expressa fazendo uso da homografia H e um fator escalar s. Para tal, é indicado o uso de coordenadas homogêneas em matrizes:

$$P_o = \begin{bmatrix} X & Y & Z & 1 \end{bmatrix}^T$$
$$P_i = \begin{bmatrix} x & y & 1 \end{bmatrix}^T$$
$$P_i = sHP_o$$

A matriz H possui duas partes: a transformada que localiza o plano do objeto no espaço e sua projeção, que inclui os parâmetros intrínsecos da câmera. A primeira parte é definida

como a representação de um plano em um sistema de coordenadas global, de modo que é o efeito de rotações e translações. Mantendo o padrão de representação por coordenadas homogêneas, tem-se $W = \begin{bmatrix} R & t \end{bmatrix}$. A segunda parte será representada pela matriz da câmera M, que contem seus parâmetros intrínsecos. Desse modo, tem-se que:

$$P_i = sHP_o = sMWP_o,$$

Onde

$$M = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$$

Como dito anteriormente, a homografia representa um plano em outro, de modo que o ponto P_o na verdade está restrito a um plano conhecido, e pode ser descrito, sem perdas de generalidade como $P'_o = \begin{bmatrix} X & Y & 0 & 1 \end{bmatrix}^T$. Dessa forma, é possível simplificar a expressão quebrando a matriz de rotação em três ($R = \begin{bmatrix} r_1 & r_2 & r_3 \end{bmatrix}$) e representando P_i em função de P'_o:

$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = sM \begin{bmatrix} r_1 & r_2 & r_3 & t \end{bmatrix} \begin{bmatrix} X \\ Y \\ 0 \\ 1 \end{bmatrix} = sM \begin{bmatrix} r_1 & r_2 & t \end{bmatrix} \begin{bmatrix} X \\ Y \\ 1 \end{bmatrix}$$

O que fará com que H se torne uma matriz 3x3.

$$P_i = sHP'_o \quad onde \quad H = sM \begin{bmatrix} r_1 & r_2 & t \end{bmatrix}$$

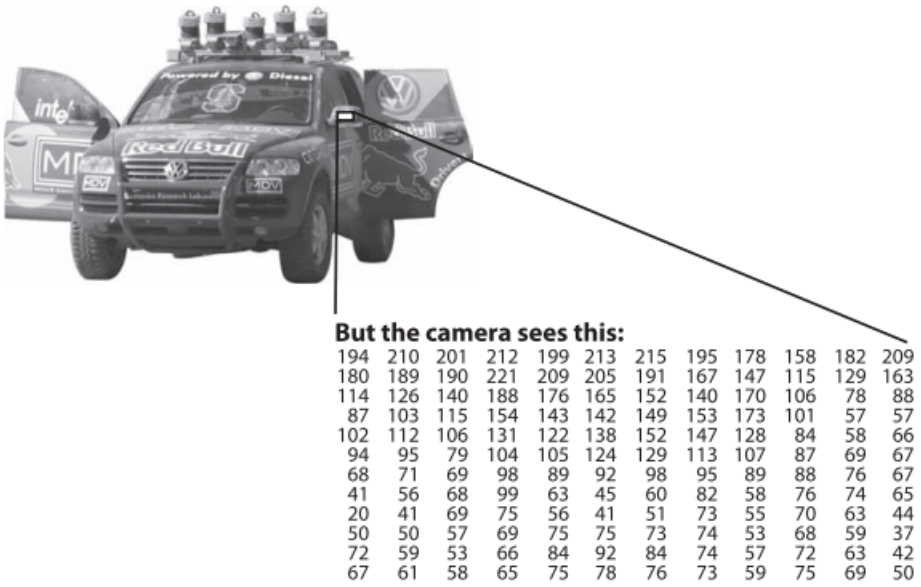
4.3 Imagens no formato digital

Ao acionar uma câmera analógica, a variação da luz que chega da cena ao obturar forma a imagem no filme fotossensível, que pode então ser passada para o papel fotográfico quando a imagem é revelada. No caso das câmeras digitais, uma corrente é gerada proporcionalmente a quantidade de luz incidente em cada ponto dos sensores. Essa corrente é transformada em valores numéricos quando digitalizada. Dessa forma, uma imagem digital é, na verdade, um grande vetor de valores, como é representado na Figura 10. *Softwares* de imagem são responsáveis por interpretar esse vetor de valores e reproduzir a imagem na tela do computador. (NIXON; AGUADO, 2002)

4.3.1 Cores na tela

Quando uma imagem digital é formada, ela possui um cabeçalho de informações para que possa ser interpretada de maneira padronizada por diferentes *softwares*. Dentre essas informações está a quantidade de pixel que formam a imagem, as dimensões da mesma e a quantidade de dados que formam um pixel. Cada tipo de arquivo de imagem

Figura 10 – Uma câmera digital vê a imagem como uma série de números.



Fonte: (BRADSKI; KAEHLER, 2008)

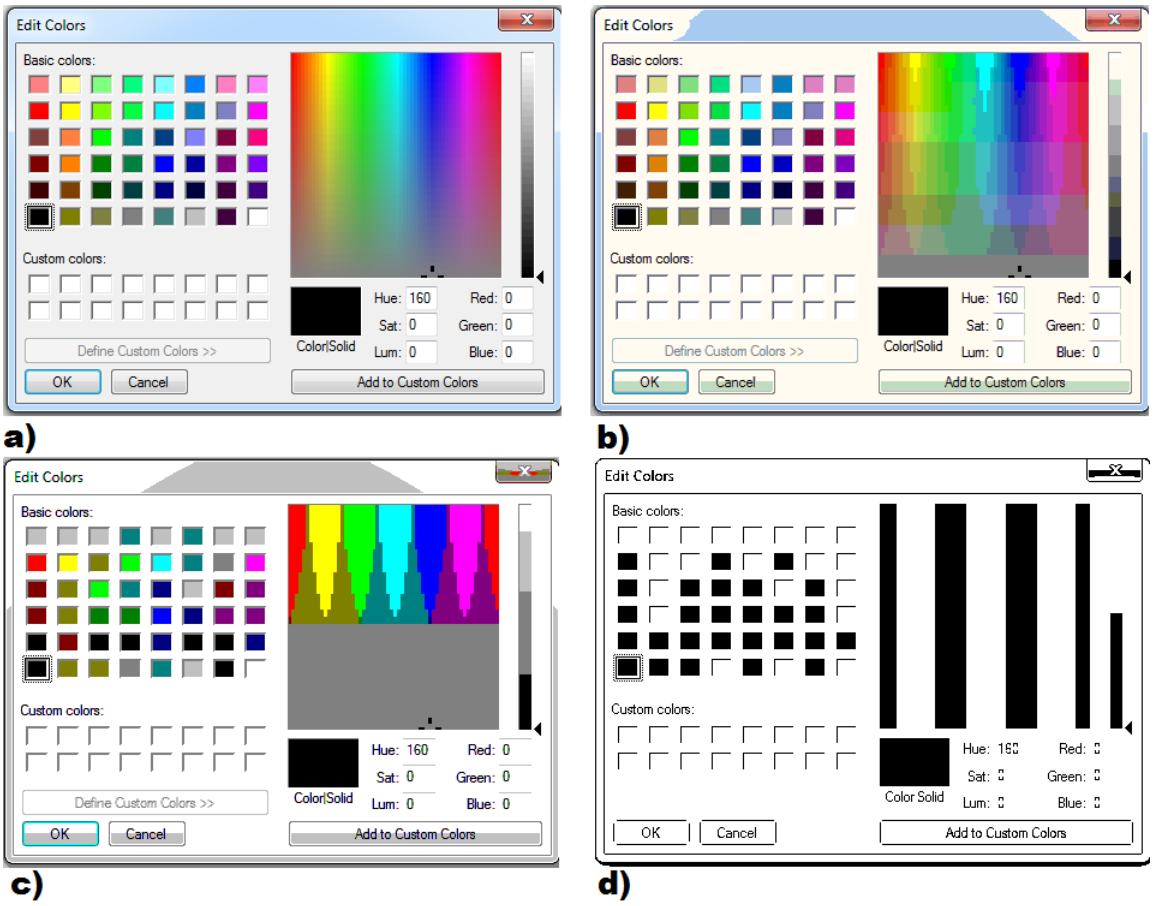
(**.jpeg, *.png, *.bmp,* etc.) possui um padrão de dados, e a cor que será representada em cada um dos pixel da imagem depende desse padrão.

Arquivos do tipo *Bitmap* (**.bmp*) são um ótimo modo de exemplificar a influencia das cores na qualidade da imagem. Esse tipo de arquivo possui quatro padrões de armazenamento de cores para pixel, usando 1, 4, 8 ou 24 bits para cada pixel. Os três primeiros tipos utilizam tabelas de cores com 2, 16 e 256 opções de cores (respectivamente). Os valores nos bits são o índice da cor desejada na tabela. O último padrão utiliza três bytes de informação para cada pixel, informando a quantidade de vermelho, azul e verde que formam a cor desejada nele. A Figura 11 abaixo exemplifica como o armazenamento da informação de cores pode influenciar a qualidade de uma imagem.

A utilização das cores vermelho, azul e verde para representação de cores em telas e monitores é amplamente utilizada, e conhecida como modelo de cor RGB (da sigla em inglês "*red, green, blue*"). Esse modelo é baseado em um sistema cartesiano que contem a cor preta em sua origem e cada um dos eixos representa uma das cores bases, como pode ser visto na Figura 12. Todas as cores existentes podem, assim, ser representadas através de uma decomposição das cores principais. Uma escala de cinza se estende pela diagonal, onde todas as cores têm o mesmo valor, do ponto de origem até a cor branca, que possui os maiores valores de coordenadas do sistema. (GONZALEZ; WOODS, 2008)

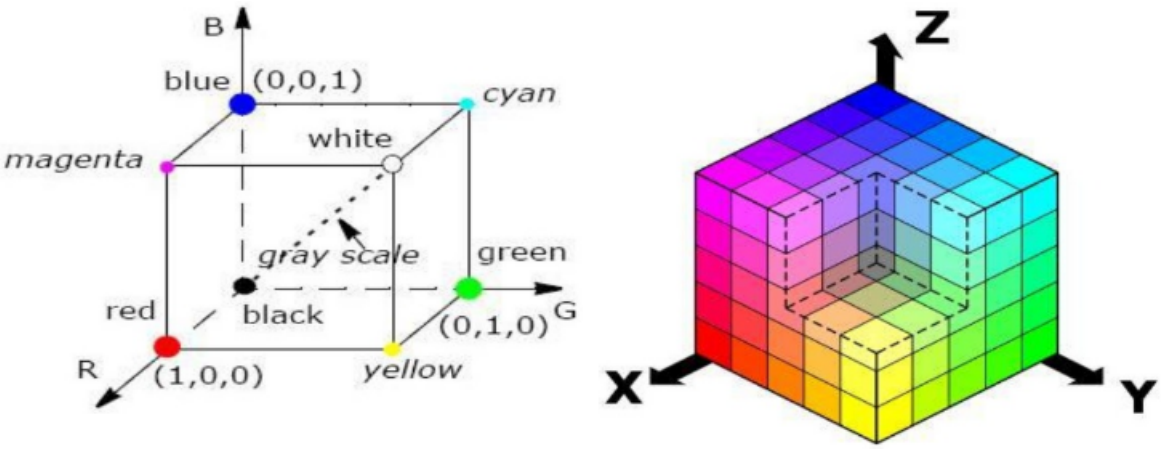
Apesar de ser muito utilizado, o modelo RGB pode apresentar dificuldades quando se deseja identificar objetos de uma determinada cor em uma imagem digital, principalmente em uma sequência de imagens (ou um vídeo) de um local que não tem iluminação constante.

Figura 11 – Editor de cores do *software Paint* salvo em formato *bitmap* com a) 24 bits, b) 8 bit, c) 4 bits e d) 1 bit de informações de cor por pixel.



Fonte: imagem produzida pelo autor.

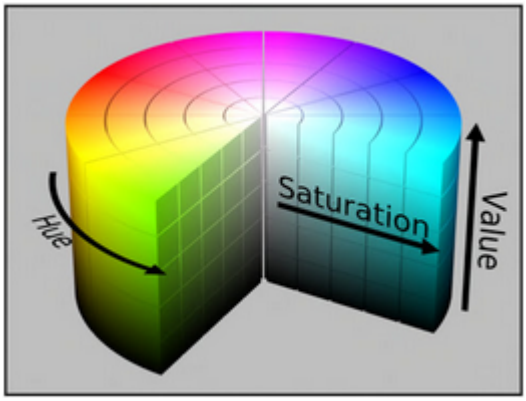
Figura 12 – Modelo de cor RGB baseado em um sistema de coordenadas cartesianas.



Fonte: (PRADHAN, 2013)

Em casos como esse, é preferível a utilização do modelo HSV (*Hue*, *Saturation* e *Value*). Neste, primeiro termo é a tonalidade, e define a cor em si, especificada de maneira geral pela onda visível que a produz (amarelo, azul, violeta, laranja, etc). A saturação, no segundo termo, indica a proporção de cinza e da cor "pura"naquele tom específico. O último termo é o valor, e traduz o que percebemos como luminosidade da cor no objeto. A utilização do HSV também é favorecida pelo fato de que, apesar do modelo RGB ser mais próximo de como o olho humano percebe cores, as pessoas estão mais condicionados a identifica-las usando parâmetros semelhantes ao do modelo HSV, pois a descrição de cores por decomposição não é algo natural para a maior parte das pessoas. O modelo HSV pode ser representando através de um sistema de coordenadas cilíndricas, como mostra a Figura 13. (EASTON, 2010)

Figura 13 – Modelo HSV em representação de coordenadas cilíndricas.



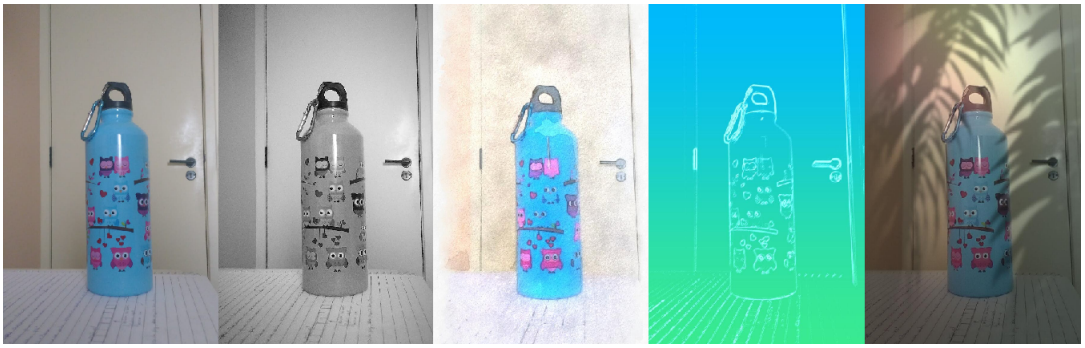
Fonte: (KAPUR, 2017)

4.3.2 Processamento de imagens

Por se tratar de vetores numéricos, imagens podem ser trabalhadas e alteradas matematicamente. É possível modificar seu tamanho introduzindo novos pixel para amplia-la. Eles receberão valores semelhantes ou intermediários aos pixel ao seu redor para uma transição suave. O processo inverso também é possível para reduzir o tamanho da figura. Um filtro de imagem, por sua vez, ira comparar os valores no vetor com algum padrão estabelecido e modificar-los de acordo com o resultado. Um filtro que suavize contornos, por exemplo, pode usar uma diferença brusca de cor para identificar as marges dos objetos e substituir os valores dos pixel da fronteira por valores intermediários. É possível também adicionar elementos gráficos as imagens através de uma máscara que irá alterar ou substituir grupos de pixel em locais específicos do vetor. O numero de acessos a mídias sociais através de celulares trouxe um aumentos significativo na quantidade de aplicativos com *features* onde é possível fazer modificações nas imagens como as descritas.

A Figura 14 abaixo traz alguns exemplos de filtros do aplicativo *Instagram*, que permite esse tipo de alteração nas imagens.

Figura 14 – Exemplos de filtros de imagem do aplicativo *Instagram*.



Fonte: imagem produzida pelo autor.

Um filtro de cor pode selecionar apenas os objetos da cena que estão dentro do intervalo estabelecido e apagar todo o resto da imagem. A mudança das cores de uma imagem pode acontecer também de maneira relativamente simples. Digamos que se deseja dar um tom avermelhado a foto. Basta acrescentar uma constante às posições que representam o vermelho em cada pixel. Caso o usuário deseje transformar a imagem em uma fotografia preta e branca, basta apenas substituir por um pixel branco todos aqueles cuja cor estiverem acima do limite indicado e por preto os que estiverem abaixo.

5 Proposta: Programação Auxiliada por Visão Computacional

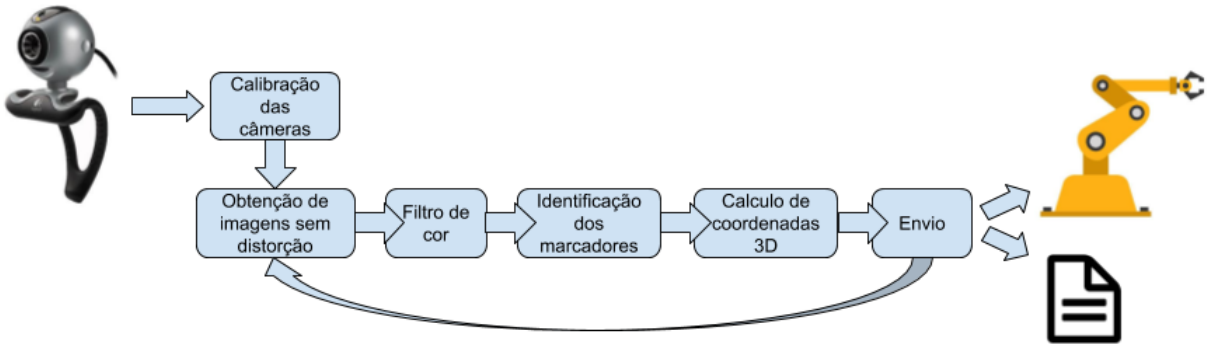
Este projeto tem como objetivo a criação de um programa de baixo custo e utilização simplificada para auxiliar na programação de braços robóticos com base na reprodução de um movimento humano. Dessa forma, foi selecionado o método de visão computacional através do uso de *webcams*, devido a sua acessibilidade e a facilidade do uso de dados obtidos, e marcadores de cores específicas para captação do movimento do operador. Para completar o sistema, foi optado pelo uso da biblioteca de código aberto *OpenCV* (*Open Source Computer Vision Library*), gratuita para uso tanto acadêmico quanto comercial. Esta biblioteca permite a manipulação e processamento de imagens computacionais, com foco na eficiência de processamento e aplicações em tempo real.

Como a parte mais importante da programação de um braço robótico consiste em conhecer a localização e o posicionamento do atuador presente no fim do manipulador, não é necessário reconhecer a movimentação de todo o braço do operador, apenas de um ponto que represente o TCP. Foi utilizada uma luva com os marcadores nos dedos para a identificação, de forma que o programa desenvolvido seja capaz de encontrar a posição no espaço da mão do operador e sua orientação em relação aos eixos do sistema de coordenada. Ele pode detectar ainda a distância entre os marcadores, que pode ser utilizada caso o atuador seja uma garra ou seja desejável introduzir alguma informação complementar ao funcionamento da ferramenta utilizada, como ligar ou desligar uma parafusadora.

O sistema desenvolvido segue a arquitetura apresentada na Figura 15 abaixo. Na primeira etapa ocorre a calibração das câmeras, que será utilizada para remover distorções das imagens. Em seguida, as imagens passam por um filtro de cor para a identificação dos objetos de interesse. É obtido um ponto médio representante de cada objeto nas imagens, e com eles são calculadas as posições desses objetos no mundo real. Por último, é montado um vetor com as informações, que pode ser enviado diretamente ao robô (em tempo real) ou a um arquivo para uso futuro.

Cada ponto da arquitetura é realizado por uma ou mais etapas no programa desenvolvido, que serão detalhadas abaixo. As funções do programa foram organizadas em classes, de forma a generalizar e simplificar o uso futuro. Antes disso, porém, o primeiro passo do projeto foi a seleção do número e posicionamento da(s) câmera(s) a ser utilizado, pois este influenciaria em como seria realizada a identificação das coordenadas dos marcadores.

Figura 15 – Arquitetura do sistema desenvolvido.



Fonte: imagem produzida pelo autor.

5.1 Número de câmeras e posicionamento

As imagens produzidas pelas *webcams* comerciais possuem apenas duas dimensões, de modo que as informações de profundidade da cena captada (e seus objetos) são perdidas. Para a recuperação desses dados, é preciso adicionar informações ao sistema, o que pode ser feito de maneira simples através do uso de uma segunda câmera que irá captar a mesma cena, porém de outra perspectiva. Fazendo uso de ambas as imagens é possível identificar a profundidade de diversos elementos através de cálculos trigonométricos.

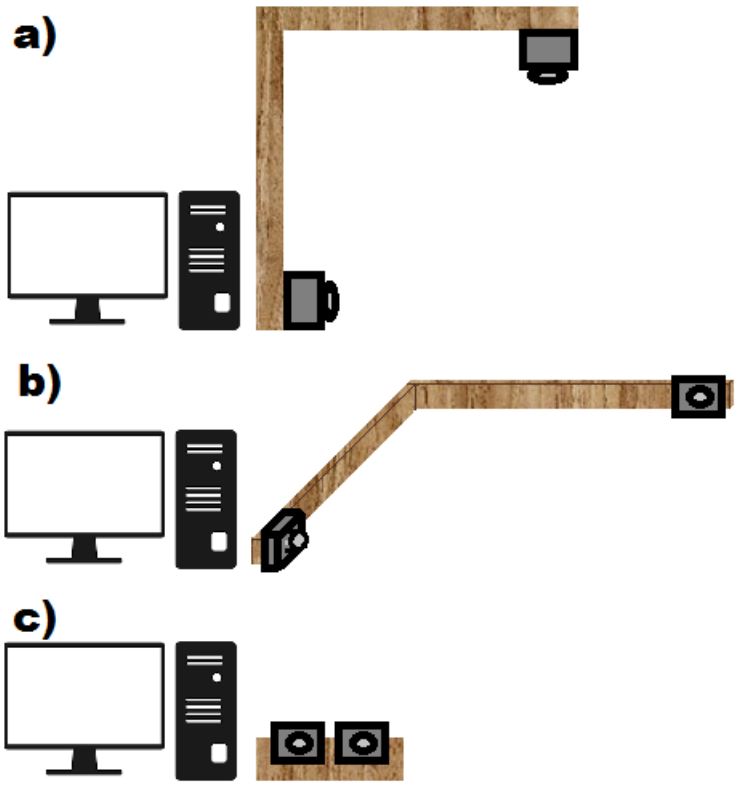
Ao longo do desenvolvimento deste projeto, foram considerados três posicionamentos diferentes para essas duas câmeras. A primeira configuração constituía de uma câmera frontal, que iria captar o movimento no plano XY, e uma câmera superior, perpendicular a primeira, que identificaria o posicionamento (x, z) dos marcadores (Figura 16.a). Este *setup*, no entanto, apresentava uma limitação na área de trabalho disponível. Devido a baixa amplitude visual das câmeras, seria necessária um estrutura muito alta para conter a área desejada para a movimentação do braço do operador.

Uma segunda montagem foi idealizada, onde a câmera superior seria substituída por uma câmera lateral (que captaria o movimento no plano YZ) para simplificar a estrutura necessárias (Figura 16.b). Essa configuração, apesar de possuir uma construção mais simples, ainda apresentava a mesma limitação da área de trabalho em relação ao seu tamanho. Além disso, ela também apresentou uma maior dificuldade de acesso a área de trabalho pelo operador se fosse desejado/necessário que nenhuma outra parte do seu corpo, que não a mão e o braço, estivesse no campo de visão das câmeras.

A terceira, e última, montagem (Figura 16.c) é baseada nas câmeras estereoscópicas. Nessa configuração, a segunda câmera não está mais perpendicular a primeira, como nos modelos anteriores, mas sim ao seu lado. Isso causou uma imensa simplificação da estrutura necessárias para acomodar as câmeras além de uma maior flexibilidade no tamanho da

área de trabalho, uma vez que basta que o operador se posicione mais perto ou mais afastado da estrutura para diminuir ou aumentar (respectivamente) a área útil captada nas imagens.

Figura 16 – Configurações de posicionamento de câmeras consideradas para o projeto.



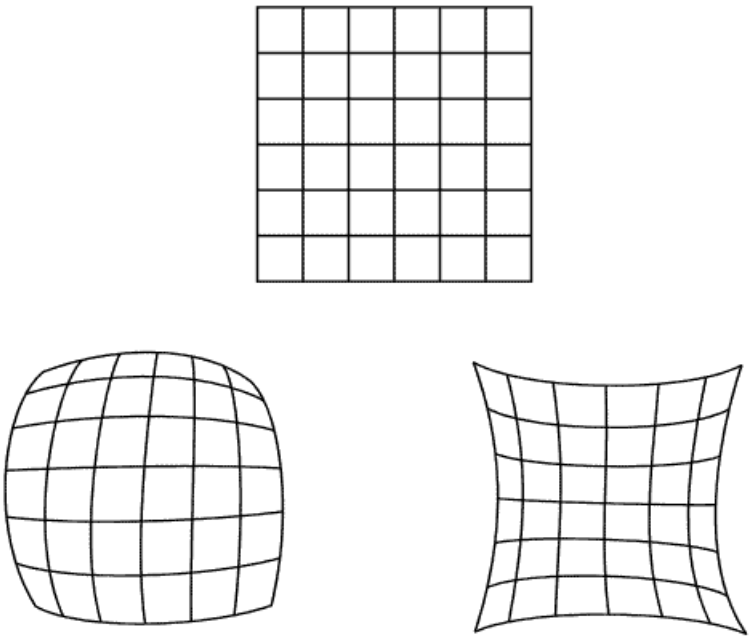
Fonte: imagem produzida pelo autor.

5.2 Calibração das câmeras

As imagens produzidas por câmeras podem aparecer distorcidas, como exemplifica a Figura 17, por conta das lentes utilizadas e das dificuldades de fabricação. Essa distorção pode causar uma identificação de posicionamento de elementos da imagem que não condiz com a situação real. Dessa forma, é de grande importância realizar a calibração da câmera, que obtém seus parâmetros causadores de distorção, e, a partir daí, trabalhar a imagem para que ela se aproxime o máximo possível da cena real.

A calibração é feita a partir da utilização de um objeto de estrutura conhecida contendo pontos bem definidos que possam ser identificados por filtros digitais. São feitas capturas da imagem desse elemento em diferentes posições e ângulos. Com um número suficiente de pontos e imagens, é possível calcular os parâmetros desejados. O objeto pode ser tridimensional, usando arestas e cantos como marcadores, ou pode ser uma figura

Figura 17 – Exemplo de distorções radiais positiva e negativa.



Fonte: (NEY DIAS - ÓPTICA OFTáLMICA, 2014)

bidimensional. Algo bastante utilizado é uma imagem (geralmente preta e branca) de um tabuleiro de xadrez de casas quadradas. O contraste das cores permite uma fácil identificação das interseções entre quadrados, que são usadas como pontos de marcação. Como o tabuleiro é plano e de dimensões padronizadas, é possível identificar a angulação em relação a câmera assim como a diferença entre onde os pontos deveriam estar e onde de fato estão na imagem. Isso permite, ainda, que ao final do processo o sistema tenha a informação da relação *distância entre pixel/distância no mundo real.*(BRADSKI; KAEHLER, 2008)

Para efetuar a calibração, é necessário encontrar os quatro parâmetros intrínsecos da câmera (f_x, f_y, c_x e c_y) e os cinco de distorção da lente (os radiais k_1, k_2 e k_3 e os tangenciais p_1 e p_2). O primeiro grupo está relacionado à posição da câmera e dos objetos visíveis por ela no espaço, enquanto que o segundo interfere em como o padrão de pontos é distorcido na imagem. O processo para calcular os parâmetros é realizado em três etapas.

Na primeira etapa, como demonstrado por Bradski e Kaehler (2008), são identificadas as equações homográficas (H) de cada imagem salva do tabuleiro em relação a câmera. Ignorando-se por enquanto a distorção causada pelas lentes, cada equação é igual a matriz intrínseca da câmera (M) multiplicada pelas duas primeiras colunas da matriz de rotação e o vetor de translação.

$$H = [h_1 \quad h_2 \quad h_3] = sM[r_1 \quad r_2 \quad t]$$

De modo que, fazendo $s = 1/\lambda$:

$$\begin{aligned} h_1 = sMr_1 \quad ou \quad r_1 = \lambda M^{-1} h_1 \\ h_2 = sMr_2 \quad ou \quad r_2 = \lambda M^{-1} h_2 \\ h_3 = sMt \quad ou \quad t = \lambda M^{-1} h_3 \end{aligned}$$

Fazendo uso do fato que os vetores de rotação possuem o mesmo tamanho e são ortogonais entre sim, é possível obter que:

$$\begin{aligned} h_1^T M^{-T} M^{-1} h_2 = 0 \\ h_1^T M^{-T} M^{-1} h_1 = h_2^T M^{-T} M^{-1} h \end{aligned}$$

Definindo B como $B = M^T M^{-1}$:

$$B = \begin{bmatrix} B_{11} & B_{12} & B_{13} \\ B_{21} & B_{22} & B_{23} \\ B_{31} & B_{32} & B_{33} \end{bmatrix} = \begin{bmatrix} \frac{1}{f_x^2} & 0 & \frac{-c_x}{f_x^2} \\ 0 & \frac{1}{f_y^2} & \frac{-c_y}{f_y^2} \\ \frac{-c_x}{f_x^2} & \frac{-c_y}{f_y^2} & \frac{c_x^2}{f_x^2} + \frac{c_y^2}{f_y^2} + 1 \end{bmatrix}$$

E com algumas operações algébricas é possível encontras as equações que definem os parâmetros intrínsecos da câmera:

$$\begin{aligned} f_x = \sqrt{\lambda / B_{11}} \\ f_y = \sqrt{\lambda B_{11} / (B_{11} B_{22} - B_{12}^2}} \\ c_x = -B_{13} f_x^2 / \lambda \\ c_y = (B_{12} B_{13} - B_{11} B_{23}) / (B_{11} B_{22} - B_{12}^2) \end{aligned}$$

onde:

$$\lambda = B_{33} - (B_{13}^2 + c_y (B_{12} B_{13} - B_{11} B_{23})) / B_{11}$$

Os parâmetros extrínsecos de rotação e translação podem ser então calculados a partir das equações:

$$\begin{aligned} r_1 = \lambda M^{-1} h_1 \\ r_2 = \lambda M^{-1} h_3 \\ r_3 = r_1 \times r_2 \end{aligned}$$

$$t = \lambda M^{-1} h_3$$

Uma vez que esses valores foram estipulados, os parâmetros relativos a distorção das lentes podem ser calculados. A diferença dos pontos em uma imagem gerada por um modelo pinhole da câmera e pelo seu modelo real (com lente) pode ser descrita como:

$$\begin{bmatrix} x_p \\ y_p \end{bmatrix} = \begin{bmatrix} f_x X^W / Z^W + c_x \\ f_y X^W / Z^W + c_y \end{bmatrix}$$

Substituindo-se os resultados da calibração sem distorção:

$$\begin{bmatrix} x_p \\ y_p \end{bmatrix} = (1 + k_1 r^2 + k_2 r^4 + k_3 r^6) \begin{bmatrix} x_d \\ y_d \end{bmatrix} + \begin{bmatrix} 2p_1 x_d y_d + p_2(r^2 + 2x_d^2) \\ p_1(r^2 + 2y_d^2) + 2p_2 x_d y_d \end{bmatrix}$$

Uma grande quantidade dessas equações são coletadas e resolvidas para determinar os parâmetros de distorção. Com esse processo finalizado, é possível re-estimar os parâmetros intrínsecos e extrínsecos da câmera. A biblioteca *OpenCV* possui funções que realizam todo o trabalho matemático descrito acima.

5.2.1 Calibração Estéreo

Na calibração estéreo, é necessário encontrar, além dos parâmetros das câmeras, a relação geométrica entre elas no espaço. Isso é feito através da matriz rotação R e do vetor de translação T entre as duas câmeras. Sendo possível definir um ponto (P_o) no espaço em relação a cada uma das câmera como $P_d = R_d P_o + T_d$ para a câmera da direita e $P_e = R_e P_o + T_d$ para a da esquerda, é possível utilizar esses pontos para expressar a relação entre as câmeras. Considerando o sistema de coordenadas com origem na câmera da esquerda, o ponto P_e na imagem da câmera da direita pode ser expresso nesse sistema como $P_e = R^T(P_d - T)$. Como os termos R_d, T_d, R_e e T_e são os parâmetros extrínsecos das câmeras da direita e da esquerda, respectivamente, que podem ser calculados durante o processo de calibração com uso do tabuleiro de xadrez, é de interesse representar R e T em função deles. (BRADSKI; KAEHLER, 2008)

$$R = R_d (R_e)^T$$

$$T = T_d - R T_e$$

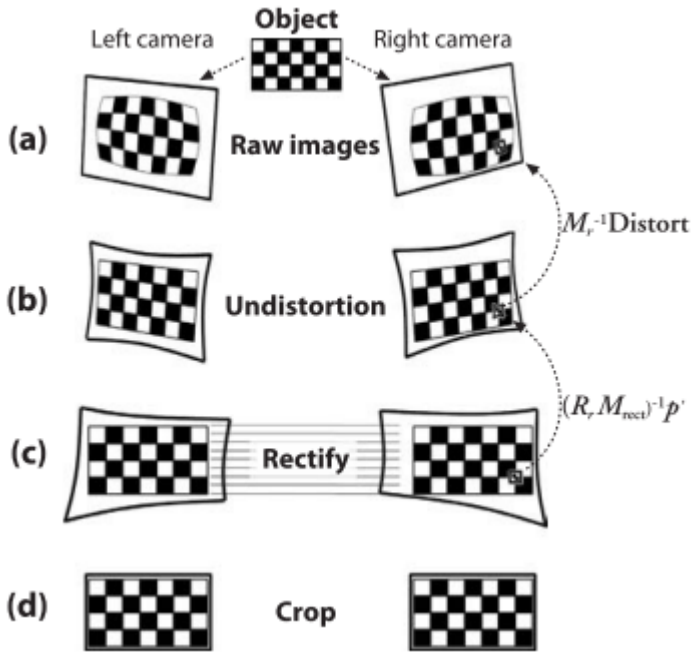
O resultado da calibração estéreo coloca as duas câmeras no mesmo plano, de modo que o encontro entre os eixos focais acontece no infinito. Isso irá auxiliar no momento de identificar a coordenada de um objeto no espaço a partir das imagens das câmeras.

5.2.2 Retificação Estéreo

A retificação estéreo serve para alinhar as imagens das câmeras, principalmente após a calibração. Isso permite que um ponto do espaço seja representado por pontos que estão a mesma altura em ambas as imagens. Esse processo é importante pois simplifica consideravelmente o cálculo que permite identificar as coordenadas de um objeto no mundo real a partir das imagens.

Como pode ser visto na Figura 18, o processo acontece em quatro etapas. A obtenção das imagens a partir das câmera acontece primeiro. Essas imagens podem apresentar diferentes graus de distorção radial e tangencial (a). Essas distorções devem ser retiradas de modo que as imagens apresentadas sejam uma representação mais fiel do mundo real (b). Em seguida é feita a retificação (c), que irá alinhar as imagens. Esse processo é feito primeiro encontrando os parâmetros de retificação e depois realizando o remapeamento dos pixel, onde cada pixel da nova imagem será computado de acordo com os valores do pixel (ou região) que ele representa da imagem original com base nos parâmetros calculados. Realizar o remapeamento desse modo evita "buracos"na imagem final. Por último as imagens são cortadas para dar ênfase as áreas que são visíveis por ambas as câmeras (d). (BRADSKI; KAEHLER, 2008)

Figura 18 – Processo de retificação de câmeras estéreo.



Fonte: (BRADSKI; KAEHLER, 2008)

De acordo com Bradski e Kaehler (2008), a biblioteca *OpenCV* possui dois métodos para realizar a retificação das imagens. O primeiro retifica imagens de câmeras estéreo

não calibradas, e pode ser utilizado sem grandes perdas em situações onde as câmeras possuem parâmetros (intrínsecos e de distorção) semelhante e se encontram com um bom alinhamento frontal. O segundo método utiliza as parâmetros de rotação e translação encontrados na calibração do par estéreo, tendendo a ter um resultado melhor, de forma que, sendo possível a calibração das câmeras, é preferível seu uso. Visando a flexibilidade e o baixo custo do projeto aqui desenvolvido (que não necessariamente terá câmeras com parâmetros semelhantes) e a facilidade de realizar a calibração através do uso da biblioteca selecionada, o segundo método foi escolhido para ser implementado.

5.3 Filtro de cor para identificação dos objetos

Uma vez que se tem as imagens sem distorções, é necessário identificar os marcadores de cores definidas pelo usuário. As imagens produzidas pelas câmeras possuem estrutura de dados baseados no modelo RGB de cores, porém este modelo não é o mais indicado quando se deseja identificar objetos em locais onde não se pode garantir a uniformidade da luz.

Para tornar o código mais robusto, o processo de filtragem por cor ocorre em duas etapas. Primeiro, há uma transformação das informações da imagens do sistema RGB para o sistema HSV. Isso irá permitir que o processo ignore parte do efeito da luz na composição da cor dos elementos. Em seguida as imagens passam por uma máscara de filtro, que irá comparar as informações das cores de cada pixel com valores máximos e mínimos estabelecidos e colocar o resultado em uma nova imagem de mesmo tamanho, porém preta e branca. Caso o pixel esteja dentro desses intervalos, ele será representado na nova imagem como sendo da cor branca (recebendo o valor 1). Caso o pixel esteja fora de algum dos intervalos, seu pixel correspondente na nova imagem será preto (indicado por ter valor igual a 0).

As cores a serem filtradas podem ser configuradas através da calibração de cor. Esse processo é bastante simples e consiste em alterar os limites de tonalidade, saturação e valor usados no filtro até que a imagem final apresente apenas o objeto de interesse na cor branca. O programa desenvolvido neste projeto permite que o usuário realize a calibração das cores utilizadas nos marcadores para identificação do movimento antes de iniciar o processo de calibração das câmeras.

Após a criação das novas imagens filtradas, o programa realiza etapas de erosão e dilatação nas imagens. De acordo com Bradski e Kaehler (2008), o processo de dilatação tende a suavizar contornos côncavos. Ela atua como uma máscara, geralmente de formato quadrado ou circular, com um ponto central. Essa máscara passa por toda a imagem verificando um grupo de pixel por vez. Caso o valor total deste grupo seja maior que um limiar pré-determinado, o pixel no ponto central é substituído por um pixel com este valor

(neste caso particular, o valor é 1). A erosão, por sua vez, é o processo inverso. Ela verifica se o valor total é abaixo de um certo nível, substituindo, neste caso, o pixel central por um de cor preta, e tende a suavizar protusões na imagem. Essas etapas ajudam principalmente a retirar ruídos das imagens produzidas.

5.4 Sistema de coordenadas

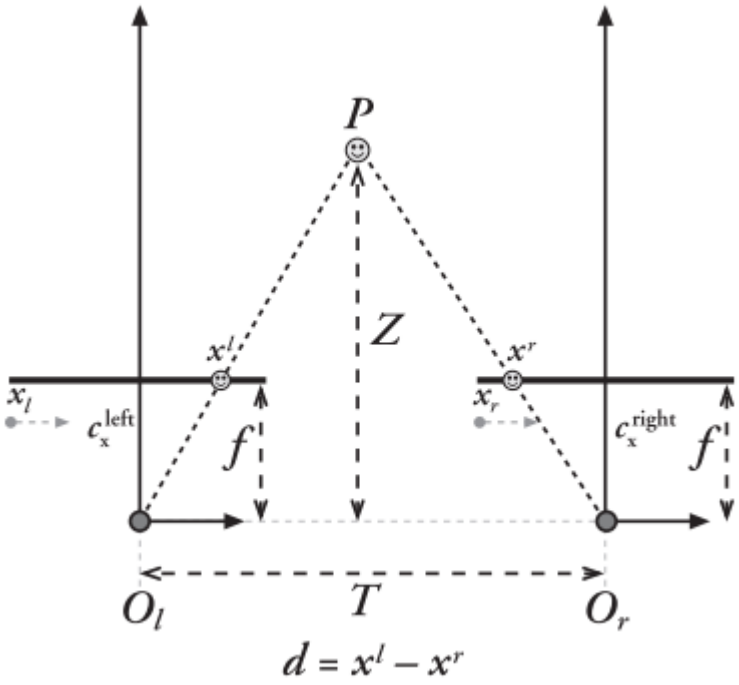
Os marcadores possuem dimensões maiores do que um pixel, porém, para facilitar a utilização, convém converter a área deles a um único ponto. Isso é feito calculando-se o momento nos eixos x e y e dividindo pela área total do marcador para encontrar o centroide da figura. Esse dados são encontrados facilmente, uma vez que cada pixel ocupado pelo marcador terá valor 1, e os demais valor 0 após a passagem pelo filtro de cor. Análogo ao cálculo do momento mecânico de um elemento físico, como por exemplo em uma viga, o momento m_{ij} em cada eixo é calculado pelo somatório do valor de cada pixel multiplicado pela sua coordenada naquele eixo. A área do marcador (m_{00}) é simplesmente a soma de todos os valores do vetor da imagem.

$$m_{ij} = \sum(imagem(x, y) \cdot x^i \cdot y^j)$$

Após a identificação dos centroides, cada imagem possui um conjunto de pontos representando os marcadores. Como as imagens de cada par estão alinhadas após o processo de retificação, os pontos representando o mesmo marcador na câmera da esquerda e da direita possuem a mesma coordenadas y. Fazendo uso dessa informação, é possível calcular as coordenadas no mundo real do marcador através de uma simples triangulação com base na disparidade entre as coordenadas x dos pontos, como pode ser vista na Figura 19 abaixo. O ponto em cada imagem representa uma linha no espaço na qual o marcador pode estar. O encontro dessas duas linhas determina o local real do objeto. As distâncias focais são conhecidas após o processo de calibração, assim com os distância entre câmeras.

A biblioteca *OpenCV* possui uma função chamada *perspectiveTransform()* que executa esse cálculo ao receber as coordenadas do ponto desejado na câmera da esquerda, a diferença (d) entre as coordenadas x das duas imagens e a relação entre profundidade e disparidade obtida na retificação (BRADSKI; KAEHLER, 2008). Essa função retorna a posição do objeto no espaço em relação ao centro da câmera à esquerda na montagem. Esse, porém, não é o sistema de coordenadas desejado, já que é usual que braços robóticos possuam a origem de seu sistema de coordenadas na sua base, e não no operador a sua frente. Dessa forma, um sistema com origem na câmera exige uma mudança de coordenada para que o braço possa utilizar os valores adquiridos diretamente do *software*, sendo necessário estabelecer uma nova origem.

Figura 19 – É possível determinar a posição do objeto a partir de semelhança de triângulos.



Fonte: (BRADSKI; KAEHLER, 2008)

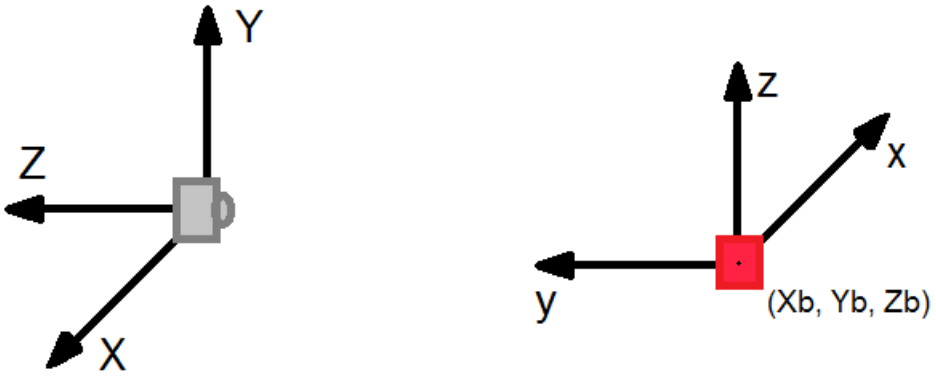
5.4.1 Cálculo da pose do atuador

Para permitir uma maior flexibilidade ao sistema, o programa desenvolvido permite que o usuário determine o novo centro do sistema de coordenadas antes de iniciar a movimentação desejada. O usuário deverá posicionar um marcador no ponto que deseja que represente a base do braço robótico em sua área de trabalho. Essa posição pode ser salva e, ao iniciar a gravação (ou envio) do movimento, o *software* realizará a mudança da coordenada usando o ponto salvo como nova origem. Dessa forma, todas as informações produzidas pelo sistema para o usuário estarão em relação a base estabelecida. Uma vez que durante o processo de calibração se conhecia os tamanhos reais entre os pontos do padrão de calibração, essas posições são calculadas em milímetros (no SI).

A Figura 20 mostra o sistema de coordenadas originário na câmera (coordenadas globais) e o objeto (em vermelho) que indica a posição da base do manipulador na área de trabalho (que dará a origem das coordenadas locais). Como pode ser visto, o sistema na câmera possui o eixo X na direção horizontal positivo para a direita e o eixo Y na direção vertical e positivo para cima. Por se tratar de um sistema dextrogiro, o eixo Z, perpendicular a ambos, é positivo no sentido entrando na câmera. Por outro lado, o sistema a ser definido como do manipulador robótico possui o plano XY paralelo ao chão, sendo x positivo para a direita do manipulador e y para a frente. Por se tratar também de um sistema dextrogiro, o eixo z deve ter direção vertical, sendo positivo para cima. Dessa

forma, para fazer a transformação de uma coordenada em outra, é necessária tanto uma translação do centro quanto uma rotação dos eixos.

Figura 20 – Sistemas de coordenadas baseados na câmera e no objeto representando a base do braço.



Fonte: imagem produzida pelo autor.

A matriz de rotação da transformada de coordenadas no espaço 3D é expressa em função dos senos e cossenos dos ângulos entre os eixos quando os sistemas possuem a mesma origem. Como é possível observar, neste caso os ângulos formados são ângulos retos, o que simplifica consideravelmente as equações, uma vez que elas se reduzirão a zeros e uns. A translação da origem é feita subtraindo-se a origem do sistema local da posição do ponto nas coordenadas globais. Assim, considerando que o novo sistema tem origem nas coordenadas (X_b, Y_b, Z_b), um ponto M pode ser expresso nas coordenadas do braço como:

$$\begin{bmatrix} x_m \\ y_m \\ z_m \end{bmatrix} = \begin{bmatrix} -1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} X_m \\ Y_m \\ Z_m \end{bmatrix} - \begin{bmatrix} -X_b \\ Z_b \\ Y_b \end{bmatrix}$$

Dessa forma, a posição de cada marcador em relação ao local definido como a base do manipulador na área de trabalho pode ser calculada a partir de suas coordenadas em relação a câmera através das seguintes equações:

$$x_m = -X_m + X_b$$

$$y_m = Z_m - Z_b$$

$$z_m = Y_m - Y_b$$

Tendo-se as coordenadas no espaço dos dois marcadores ($m1$ e $m2$), é possível traçar um seguimento de reta entre eles. O TCP é calculado, então, como o ponto médio

deste seguimento, dado por:

$$TCP_x = \frac{m1_x + m2_x}{2}$$
$$TCP_y = \frac{m1_y + m2_y}{2}$$
$$TCP_z = \frac{m1_z + m2_z}{2}$$

A distância d entre os marcadores, que pode ser utilizada para abrir ou fechar uma garra, acionar ou desligar um ferramenta, é calculada como o comprimento do segmento no espaço tridimensional. Sendo:

$$d_x = m2_x - m1_x$$
$$d_y = m2_y - m1_y$$
$$d_z = m2_z - m1_z$$

Tem-se então:

$$d = \sqrt{d_x^2 + d_y^2 + d_z^2}$$

Os ângulos (α, β, γ) que este segmento de reta forma com cada um dos eixos do sistema local (x, y e z respectivamente) definem a angulação da ferramenta do braço em relação aos mesmos. Esses ângulos são calculados fazendo a projeção do segmento em cada um dos planos do sistema de coordenadas, de forma que:

$$\alpha = acos(\frac{d_x}{A})$$
$$\beta = acos(\frac{d_y}{A})$$
$$\gamma = acos(\frac{d_z}{B})$$

Onde:

$$A = \sqrt{d_x^2 + d_y^2} \quad e \quad B = \sqrt{d_x^2 + d_z^2}$$

Uma vez que o usuário obteve os dados desejados através das respectivas funções, fica a seu encargo o método de utilização dos mesmos. O programa permite, ainda, que os dados sejam escritos na tela, que pode ser usado para testes rápidos e simulações.

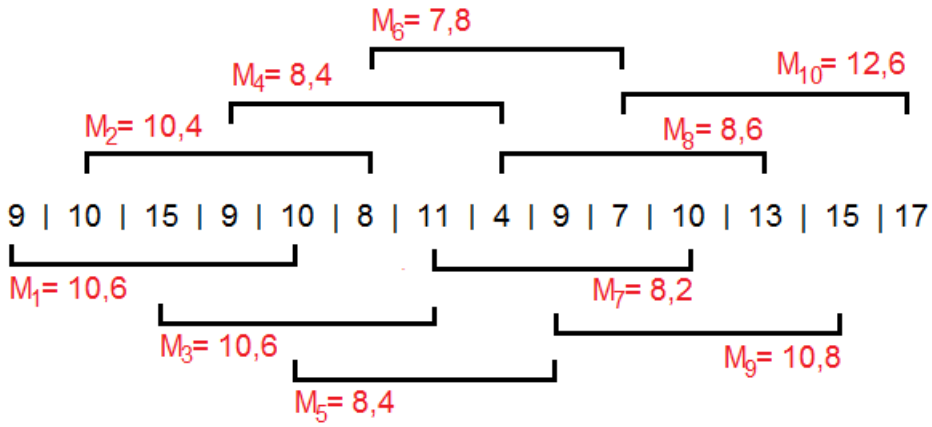
5.4.2 Media Móvel

Após alguns testes, notou-se uma grande frequência na variação das coordenadas dos pontos dos marcadores, mesmo com o objeto estacionário. Isso se deve por conta da qualidade das *webcams* utilizadas, que geram imagens com bastante ruído. É possível notar pequenas variações nas cores dos pixels das imagens produzidas, o que pode alterar a

área total identificada como pertencente a cada marcador, alterando assim o cálculo do centroide. Este problema é minimizado pelas etapas de dilatação e erosão e por uma boa escolha nos limites do filtro de cor, porém não completamente erradicado. Para contornar este problema, foi introduzida uma média móvel após o cálculo das coordenadas dos pontos.

Uma média móvel é exatamente o que o próprio nome descreve. A média é calculada somando uma determinada quantidade de termos e dividindo o total por esta quantidade. Por exemplo, uma média de 5 termos terá o total da soma dos termos dividido por 5. Estes termos, no entanto, são constantemente atualizados, de forma que a média sempre é formada pelos últimos valores adquiridos pelo sistema. A Figura 21 exemplifica o processo. A média móvel é amplamente utilizada na computação e na eletrônica, além do setor financeiro, justamente por ser uma forma prática de minimizar alterações bruscas ou de alta frequência. As alterações, positivas ou negativas, são diluídas na média, uma vez que a diferença de valor será dividida entre todos os termos.

Figura 21 – Exemplo de média móvel.



Fonte: imagem produzida pelo autor.

A escolha do tamanho da média é um fator de grande importância. Um valor baixo não permitirá uma boa diluição dos picos registrados, pois estes serão divididos por uma quantidade menor de termos. Um valor alto, no entanto, pode gerar uma suavização acentuada dos resultados, o que acarreta em um atraso de resposta. Uma vez que a mudança no valor dos termos pode indicar a movimentação da mão do operador (e não apenas o ruído causado pela alteração da área dos marcadores), um atraso elevado pode prejudicar a funcionalidade do sistema e não é desejável no projeto aqui desenvolvido. Dessa forma, o tamanho da média utilizado foi escolhido de maneira empírica através de alguns testes, para que houvesse um balanço entre a retirada da alta frequência e o tempo de resposta.

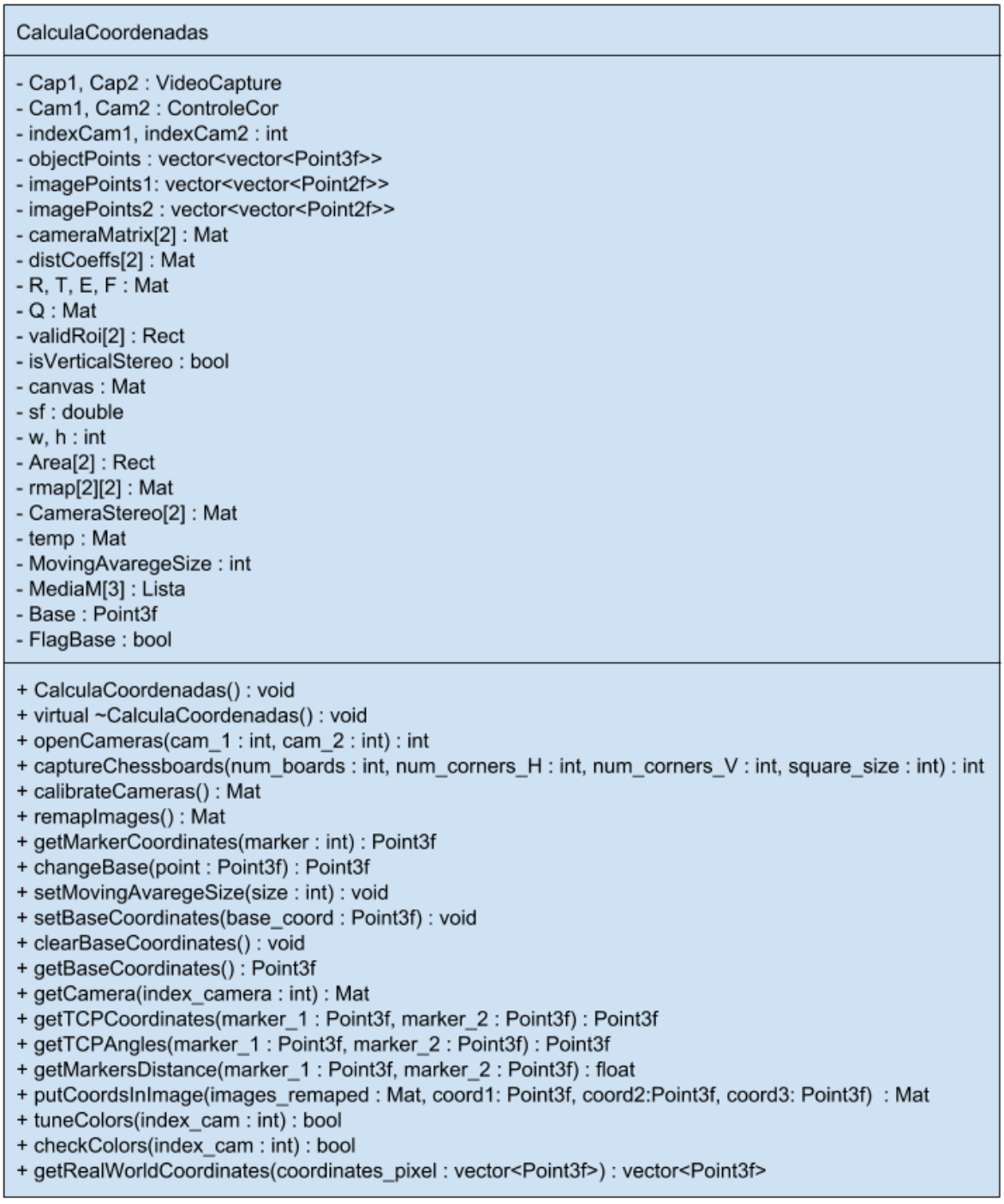
5.5 Arquitetura do código

Como mencionado, o código desenvolvido (que pode ser encontrado no anexo deste trabalho) foi organizado em funções, e estas organizadas em classes. A principal classe é a *CalculaCoordendas*, apresentada no diagrama de classes da Figura 22, que realiza todo o trabalho de calibração, retificação, remapeamento, verificação e calibração de cores dos marcadores, identificação de marcadores, cálculo de coordenadas (com mudança de base) e de pose do atuador, além de implementar a média móvel. A ideia é que esta seja a classe com a qual o usuário irá interagir para criar seu próprio programa de aquisição de coordenadas. A classe possui ainda algumas funções que podem auxiliar o usuário com informações e funcionalidades extras, como a escrita das coordenadas na imagem das câmeras e uma função que retorna as coordenadas selecionadas para a base do braço.

A classe *CalculaCoordenadas* possui objetos da classe *ControleCor*, que é quem realiza de fatos as ações relacionadas à identificação dos objetos através de cores nas imagens. Ela realiza tanto a filtragem de cor quanto calcula o centroide do objeto identificado na filtragemm. Seu diagrama pode ser visto na Figura 23. Ambas as classes mencionadas fazem uso extensivo das funções da biblioteca *OpenCV*. A ideia é que o usuário não precise lidar diretamente com as funções da biblioteca para adquirir as coordenadas que deseja, o que iria demandar um longo período de tempo de estudo. A biblioteca, por possuir diversos colaboradores, possui uma grande quantidade de funções, que podem sofrer alterações entre uma versão e outra. Apesar de possuir extensa documentação, essa documentação é em sua grande parte feita de maneira pouco didática e por usuários da biblioteca, de modo que é necessária a leitura de um grande volume de material para criar um programa como o aqui desenvolvido. Através do uso das classes *CalculaCoordenadas* e *ControleCor*, o usuário que deseja especificamente a identificação das coordenadas dos marcadores pode fazer uso do seu tempo de maneira mais eficiente.

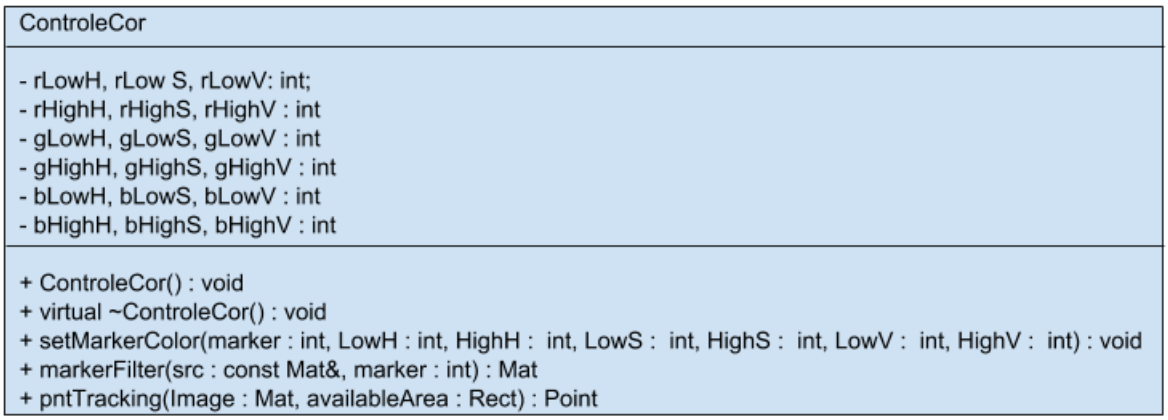
Por fim, foram criadas duas classes chamadas de *Lista* e *Celula*, utilizadas na estruturação de dados no formato de listas encadeadas. Essas estruturas foram utilizadas para o controle das médias móveis utilizadas para suavizar o efeito do ruído nas imagens. As classes são responsáveis por criar e manipular os elementos e informações contidos nessas listas, e seus diagramas podem ser vistos na Figura 24.

Figura 22 – Diagrama da classe CalculaCoordenadas.



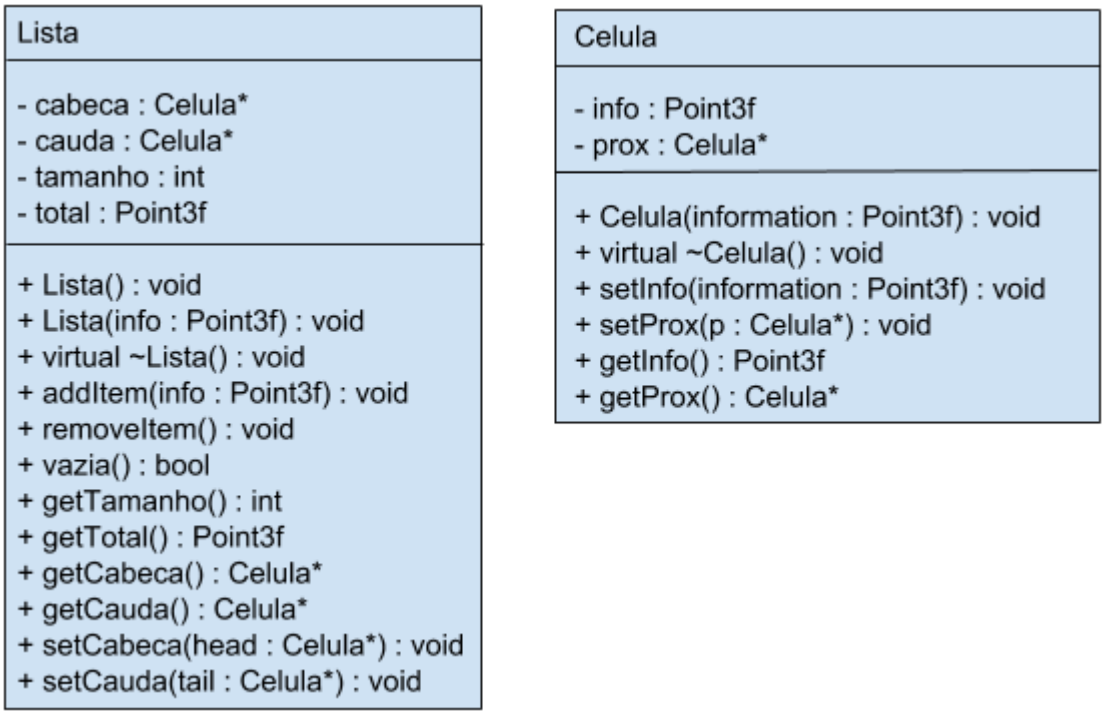
Fonte: imagem produzida pelo autor.

Figura 23 – Diagrama da classe ControleCor.



Fonte: imagem produzida pelo autor.

Figura 24 – Diagrama das classes Lista e Celula.



Fonte: imagem produzida pelo autor.

6 Testes e Análise de Resultados

Para identificar a precisão das coordenadas calculadas e a influência que a calibração tem sobre elas, foram realizados dois testes. O primeiro, Teste do Desvio Padrão, verificou a precisão do sistema proposto servindo também para validar os métodos utilizados para retirada de ruído. O segundo teste, Teste dos Padrões, teve como objetivo analisar a influência dos padrões de calibração no sistema, sendo utilizados padrões no formato de tabuleiro de xadrez.

6.1 Testes

Ambos os testes foram realizados através do cálculo de coordenadas de marcadores posicionados em locais pré-estabelecidos. Para garantir a consistência no posicionamento, esses locais foram mapeados em uma folha fixada à mesa de testes. Devido a limitação física da mesa, a área plana de trabalho delimitada para colocação dos marcadores foi de 26cm por 34cm, com o centro localizado a aproximadamente 70cm do par de câmeras.

O Teste do Desvio Padrão consistiu em obter 15 vezes as coordenadas (x, y, z) de 3 marcadores com posições fixas em relação a uma base estabelecida. Este teste foi realizado 3 vezes, cada vez após uma nova calibração, para verificar a consistência dos resultados. Todas as calibrações foram realizadas com o mesmo padrão de calibração e o mesmo número de poses para calibração.

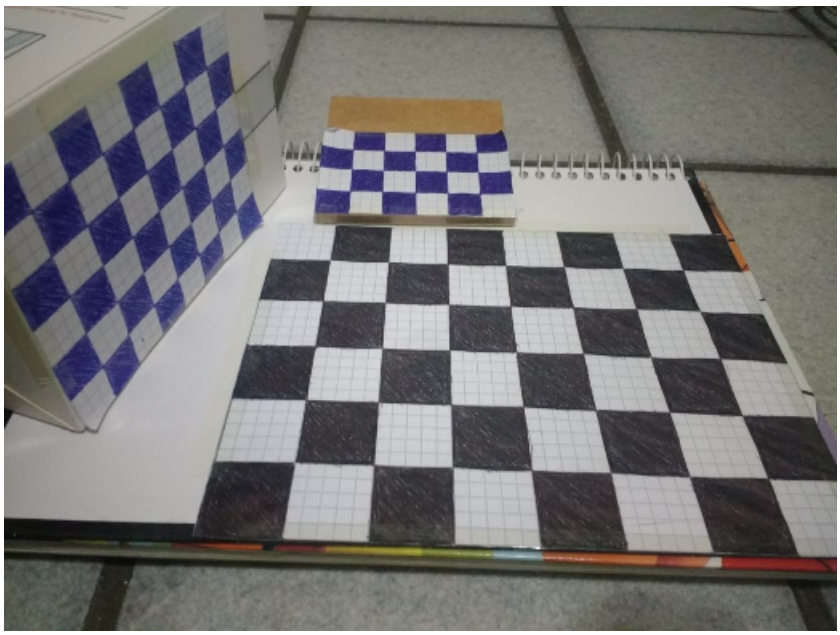
O Teste dos Padrões, por sua vez, realizou uma série de calibrações utilizando 5 padrões diferentes (alterando tamanho e quantidade de quadrados) e 10 poses do tabuleiro em cada calibração. A tabela 1 apresenta a largura dos quadrados de cada padrão, assim como a quantidade de quadrados e de pontos úteis para a calibração. O primeiro padrão foi o utilizado durante o desenvolvimento do programa e durante o Teste de Desvio Padrão. A escolha dos padrões 2 e 3 teve como objetivo avaliar a influência do número de pontos (e da quantidade de equações que podem ser obtidas) no processo, enquanto que com os padrões 4 e 5 visou-se identificar a influência do tamanho dos quadrados do padrão. A Figura 25 apresenta os padrões 1, 2 e 5. Cada padrão foi utilizado 5 vezes, para garantir a consistência das informações obtidas.

Para avaliar a qualidade da calibração realizada, foram determinados 9 pontos na área de trabalho para posicionamento de marcadores (Figura 26). Além do cálculo das coordenadas, foram analisados também fatores que podem influenciar a qualidade da calibração. Para tal, foi feita uma avaliação visual da imagem produzida após a retificação e a verificação do erro RMS da re-projeção baseada nos parâmetros obtidos

Tabela 1 – Padrões de calibração utilizados

Nº	Largura dos quadrados	Quantidade de quadrados	Quantidades de pontos
1	15mm	6 x 8	35
2	15mm	4 x 6	15
3	15mm	8 x 10	63
4	10mm	6 x 8	35
5	25mm	6 x 8	35

Figura 25 – Padrões 1 (esquerda), 2 (topo) e 5 (direita) utilizados para calibração nos testes.



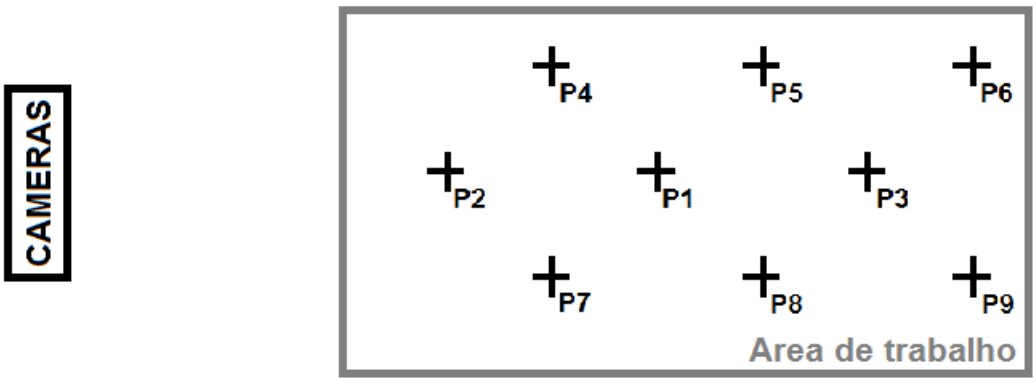
Fonte: imagem produzida pelo autor.

na calibração. Esse erro pode ser obtido através da própria função *stereoCalibrate()* da biblioteca *OpenCV*, utilizada na função *calibrateCameras()* da classe *CalculaCoordendas* para obter os parâmetros intrínsecos, extrínsecos e de distorção das câmeras. Ele é calculado a partir da média dos quadrados da diferença entre a posição em que um determinado ponto se encontra na imagem e a posição em que este mesmo ponto deveria estar na imagem segundo o modelo calculado pelo programa. Um valor elevado de erro RMS indica uma discrepância que pode afetar consideravelmente a retificação das imagens e com isso o cálculo das coordenadas.

6.2 Análise de Resultados

Após a obtenção das 15 medições em cada calibração (rodada) do Teste de Desvio Padrão, foi calculado o desvio padrão de cada coordenada (x, y, z) de cada marcador. Uma

Figura 26 – Posição dos pontos de referência para teste de calibração.



Fonte: imagem produzida pelo autor.

Tabela 2 – Resultados do Teste de Desvio Padrão

	Calibração 1	Calibração 2	Calibração 3
Marcador 1 (x)	0,006361	0,073204	0,088674
Marcador 1 (y)	0,025443	0,953341	1,080041
Marcador 1 (z)	0,006311	0,084408	0,19357
Marcador 2 (x)	0,001721	0,019772	0,096575
Marcador 2 (y)	0,007234	0,440038	1,070905
Marcador 2 (z)	0,001724	0,05212	0,244869
Marcador 3 (x)	0,060928	0,094198	0,260375
Marcador 3 (y)	0,262866	0,974778	1,598326
Marcador 4 (z)	0,063292	0,146904	0,229042

análise dos resultados mostrou que o sistema apresenta uma menor precisão no cálculo da coordenada Y dos marcadores. No entanto, os valores de desvio obtidos foram baixos, sendo o mais alto de apenas 1,6mm. Isso mostra que os métodos de erosão e dilatação nas imagens filtradas e da média móvel no valor calculado das coordenadas são efetivos. A tabela 2 mostra os desvios padrões (em milímetros) das coordenadas dos marcadores em cada rodada do teste.

O principal parâmetro para a avaliação do Teste dos Padrões foi o erro RMS obtido em cada calibração, seguido pela avaliação visual das imagens resultantes do processo. A tabela 3 apresenta o erro em cada calibração realizada com os padrões estabelecidos. Em alguns casos a calibração não foi bem sucedida, de forma que as imagens sofreram uma grave distorção ao invés de serem corrigidas. Nesses casos (assinalados em vermelho na tabela 3), não foi possível completar o teste, pois o sistema não era capaz de identificar o marcador. Em outros casos, o erro de calibração indicado foi pequeno, porém o programa não foi capaz de calcular corretamento a posição do marcador, acarretando em coordenadas (0, 0, 0) como saída do sistema. Nos demais casos, as calibrações foram consideradas bem sucedidas, e foi possível fazer algumas observações.

Tabela 3 – Erro RMS das calibrações.

Nº Calibração	Padrão 1	Padrão 2	Padrão 3	Padrão 4	Padrão 5
1	0,5404	0,4204	0,5874	14 9593	52,4043
2	0,4442	0,4612	0,4933	21,2082	0,5931
3	0,6508	0,3364	0,4881	0,5416	0,5978
4	0,8903	0,9001	0,6094	0,6018	35,1551
5	21,2110	0,4965	0,6225	0,6201	35,1890

Foi possível identificar que o processo de definir o sistema de coordenadas nas câmeras não é perfeito, principalmente em relação a posição de origem dos eixos x e z. Essa posição é estimada a partir dos parâmetros calculados durante a calibração, e por isso é afetada pela mesma. No entanto, quanto utilizada a funcionalidade do código proposto de determinar uma nova base para o sistema de coordenadas, as distâncias entre os pontos se mostraram consistente, apresentando um erro médio de $\pm$5mm no cálculo quando comparado com as distâncias reais na área de trabalho. Além disso, foi possível observar que o sistema apresentou um bom resultado para pontos a mais de 65cm de distância das câmeras quando a calibração foi realizada de maneira correta.

A partir dos testes foi observado que, apesar de o tamanho do padrão e dos quadrados do tabuleiro não afetarem diretamente a resolução do sistema de coordenadas, eles influenciam a calibração de outras maneiras que podem afetar o mesmo. Durante os testes com os padrões 3 e 5, fisicamente maiores que os demais, as câmeras tiveram dificuldade em identificar os pontos de interseção entre os quadrados do tabuleiro, principalmente em regiões mais próximas. Isso acarretou em problemas sérios de distorção em três dos testes do padrão 5, enquanto que os outros dois testes apresentaram inconsistências. Neles, o sistema foi capaz de identificar corretamente alguns dos pontos mais distantes, porém os demais apresentaram valores completamente diferentes dos esperados. É indicada, assim, a utilização de padrões menores em casos de uma área de trabalho reduzida, uma vez que é mais fácil posiciona-los de forma correta.

Vale salientar ainda dois fatores que se mostraram de grande importância para a obtenção de bons resultados. O primeiro é a quantidade de luz incidente nas câmeras. Foi observado que um alto nível de luminosidade é prejudicial a calibração das câmeras utilizadas. Cerca de 10 testes precisaram ser refeitos, pois a claridade acarretou em erros altíssimos, que produziram imagens onde não era possível identificar os marcadores devido as distorções introduzidas.

O segundo fator é a necessidade de movimentar o tabuleiro utilizado por toda a área de trabalho desejada. É importante ainda que a angulação entre o tabuleiro e as câmeras seja modificada ao longo que ele é transladado, para evitar equações semelhantes umas as outras e que não acrescentam ao processo do cálculo dos parâmetros. Em casos onde o tabuleiro não foi movido de forma correta, ou não foi possível posiciona-lo de

maneira adequada devido ao seu tamanho, a área onde o sistema manteve sua resolução foi significativamente reduzida. A área mais próxima das câmeras foi a mais afetada por esse fator.

7 Conclusão

O avanço da tecnologia tem agregado um grande crescimento na área de manipuladores robóticos, e novos métodos para programação e utilização destes estão sendo explorados. O tempo necessário para determinar o posicionamento correto do manipulador sem auxílio de informações proveniente de CADs ou simuladores pode tornar o processo de controle e programação demasiadamente longo. Por outro lado, a utilização de visão computacional tem se tornado cada vez mais viável com a melhoria dos computadores e a facilidade de acesso a meios de processamento de imagens, permitindo que esta seja utilizada com uma frequência e qualidade cada vez maior na área da robótica. Isso dá abertura para a utilização de câmeras no processo de controle de manipuladores.

Com o objetivo de auxiliar o processo de programação de braços robóticos, foi desenvolvido neste trabalho um sistema que permite ao usuário a obtenção das coordenadas no mundo real de pontos de interesse para a movimentação do atuador em relação a base do manipulador através de câmeras de vídeo. O código testado foi capaz de calcular a posição de marcadores na área de trabalho com uma acurácia de ± 5 e um desvio padrão menor que 2mm. Todo o projeto foi desenvolvido visando um baixo custo, fazendo uso de *webcams* baratas e uma biblioteca de processamento de imagens gratuita. Isso demonstra que ele pode ter sua precisão melhorada através da obtenção de câmeras de vídeo de uma qualidade mais alta, mas que também pode ser utilizado de forma satisfatória por outros usuários com poucos recursos financeiros, como é o caso de muitos estudantes de pesquisa.

Os testes realizados demonstraram algumas das limitações do sistema, como a dificuldade em identificar corretamente a localização de marcadores posicionados muito próximos as câmeras. No entanto, a maioria dos problemas encontrados durante o processo puderam ser contornados através do uso de filtros, iluminação adequada e boas práticas de calibração. De forma geral, o desempenho do projeto foi considerado satisfatório, com uma precisão boa o suficiente para permitir sua utilização como forma de agilizar o processo de programação ou até substitui-lo, a depender das necessidades do usuário.

7.1 Trabalhos futuros

É indicado como trabalho futuro a implementação deste projeto para a utilização em braços robóticos reais, avaliando sua usabilidade e o efeito que o mesmo tem no dia de trabalho do operador. Seria também interessante validar o processo de capacitação de usuários para utilizar este tipo de sistema, em especial para casos onde se deseja um controle do braço integralmente em tempo real, e não apenas sua programação.

Referências

ACERVOSABER. *Robotica*. 2015. Disponível em: <http://www.acervosaber.com.br/trabalhos/eletronica1/robotica.php>. Acesso em: 29 nov. 2015.

APRENDA FOTOGRAFIA. *Como Funciona a Câmera 1 - Introdução*. 2014. Disponível em: <https://aprendafotografia.org/como-funciona-camera-fotografica/>. Acesso em: 14 out. 2018.

BENKO, H. et al. Enhancing input on and above the interactive surface with muscle sensing. *Proceedings of the ACM International Conference on Interactive Tabletops and Surfaces*, Banff, p. 93–100, 2009.

BRADSKI, G.; KAEHLER, A. *Learning OpenCV*. Sebastpool: O'Reilly Media, Inc, 2008. ISBN 978-0-596-51613-0.

BRAHMANI, K.; ROY, K. S.; ALI, M. Arm 7 based robotic arm control by eletronic gesture recognition unit using mems. *International Journal of Engineering Trends and Technology*, v. 4, n. 4, p. 1245–1248, 2013. ISSN 2231-5381.

BRINK, W. *Computer Vision*. Stellenbosch: [s.n.], 2017.

BRITISH AUTOMATION AND ROBOT ASSOCIATION. *Robot programming methods*. Disponível em: <http://www.bara.org.uk/robots/robot-programming-methods.html>. Acesso em: 13 out. 2018.

CABRE, T. P. et al. Project-baes learning example: controlling an educational robotic arm with computer vision. *IEEE Revista Iberoamericana de Tecnologias del Aprendizaje*, v. 8, n. 3, p. 135–142, 2013. ISSN 1932-8540.

CARRARA, V. *Introdução a Robótica Industrial*. São José dos Campos, 2015.

COMAU. *Teach Pendant Comau*. Disponível em: <https://www.comau.com/en/our-competences/robotics/teach-pendant-tp5>. Acesso em: 13 out. 2018.

CRAIG, J. J. *Introduction to Robotics. Mechanics and Control*. Upper Saddle River: Prentice Hall, 2005. ISBN 0-13-123629-6.

CUBERO, S. *Industrial Robotics: Theory, Modelling and Control*. Mammendorf: pIV pro literatur Verlag Robert Mayer Scholz, 2007. ISBN 3-86611-285-8.

DPA MAGAZINE. *Dual-purpose robot joystick control*. 2013. Disponível em: <http://www.dpaonthenet.net/article/59673/Dual-purpose-robot-joystick-control.aspx>. Acesso em: 13 out. 2018.

EASTON, R. L. *Fundamentals of Image Processing*. [S.l.: s.n.], 2010.

GONZALEZ, R. C.; WOODS, R. E. *Digital Image Processing*. Upper Saddle River: Pearson Education, Inc, 2008. ISBN 0-13-168728-X.

GRAVILA, D. M. The visual analysis of human movement: a survey. *Computer Vision and Image Understanding*, v. 73, n. 1, p. 81–98, 1999.

JIAN, H.; DUERSTOCK, B. S.; WACHS, J. P. A machine vision-based gestural interface for people with upper extremity physical impairments. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, v. 44, n. 55, p. 630–641, 2014. ISSN 2168-2216.

KAPUR, S. *Computer Vision with Phython 3*. Birmingham: Packt Publishing Ltd, 2017. ISBN 978-1-78829-976-3.

LEARNED-MILLER, E. G. *Introduction to Computer Vision*. Amherst: [s.n.], 2011.

LEWIS, F. L.; DAWSON, D. M.; ABDALLAH, C. T. *Robot Manipulator Control. Theory and Practice*. New York: Marcel Dekker, Inc, 2004. ISBN 0-8247-4072-6.

LYNCH, J. P. et al. Design of piezoresistive mems-based accelerometer for integration with wireless sensing unit for structural monitoring. *Journal of Aerospace Engineering*, v. 16, n. 3, p. 108–114, 2003. ISSN 0893-1321.

NEY DIAS - ÓPTICA OFTáLMICA. *Lentes*. 2014. Disponível em: <https://sites.google.com/site/neydiasopticaoftalmica/lentes>. Acesso em: 4 nov. 2018.

NIXON, M. S.; AGUADO, A. S. *Feature Extraction and Image Processing*. Oxford: Butterworth-Heinemann, 2002. ISBN 0-7506-5078-8.

NOF, S. Y. *Handbook of industrial robotics*. New York: John Wiley and Sons, Inc, 1999.

OPENCV. *About*. 2018. Disponível em: <https://opencv.org/about.html>. Acesso em: 20 nov. 2018.

PEDROTTI, L. S. et al. *Fundamentals of Photonics*. [S.l.].

PERPETUAL ENIGMA. *Understanding Camera Calibration*. 2014. Disponível em: <https://prateekvjoshi.com/2014/05/31/understanding-camera-calibration/>. Acesso em: 14 nov. 2018.

PRADHAN, K. *Color Base Image Processing. Tracking and Automation*. 2013. Disponível em: <https://www.slideshare.net/KamalPradhan1/color-based-image-processing-tracking-and-automation-using-matlab>. Acesso em: 16 nov. 2018.

ROBOTIQ COMPANY. *The future of the robot teach pendant*. 2015. Disponível em: <http://blog.robotiq.com/future-of-robot-teach-pendant-robot-programming>. Acesso em: 13 out. 2018.

ROSARIO, J. M. *Princípios da Mecatrônica*. São Paulo: Prentice Hall, 2005.

ROSARIO, J. M. *Robótica Industrial I: Modelagem, utilização e programação*. São Paulo: Baraúna, 2010.

SAPONAS, T. S. et al. Enabling always-available input with muscle-computer interfaces. *Proceedings of the 22nd annual ACM symposium on User interface software and technology*, Victoria, p. 197–176, 2009.

SEARA DA CIENCIA. *Tipos mais comuns de acelerômetros*. Disponível em: <http://www.seara.ufc.br/tintim/tecnologia/acelerometro/acelerometro01.htm>. Acesso em: 13 out. 2018.

SEQUEIRA, C. Sensores para medições de vibrações mecânicas - acelerômetro. *Manutenção*, Lisboa, n. 16, p. 4–6, 2013.

SICILIANO, B.; KHATIB, O. *Springer Handbook of Robotics*. Berlin: Springer, 2008. ISBN 978-3-540-23957-4.

SILVA, J. M. B. *Mecanismos Articulados*. Recife: Ed. Universitária da UFPE, 2010. 198 p. ISBN 978-85-7315-864-9.

SUBRAMANIAM, Y.; FAI, Y. C.; MING, E. S. L. Edible bird nest processing using machine vision and robotic arm. *Jurnal Teknologi*, p. 85–88, 2014.

SZABO, R.; GONTEAN, A. Robotic arm control algorithm based on stereo vision using roborealm vision. *Advances in Electrical and Computer Engineering*, v. 15, n. 2, p. 65–74, 2015. ISSN 1582-7445.

SZELISKI, R. *Computer Vision: Algorithms and Applications*. London: Springer-Verlag London, 2011. ISBN 978-1-84882-935-0.

TITARMARE, J. C.; KATKAR, M. D.; AGRAWAL, A. J. Robotic arm - the ball catcher. *International Journal of Engineering Trends and Technology*, v. 8, n. 9, p. 503–505, 2014. ISSN 2231-5381.

XU, R.; ZHOU, S.; LI, W. J. Mems accelerometer based nonspecific-user hand gesture recognition. *IEEE Sensors Journal*, v. 12, n. 5, p. 1166–1173, 2012. ISSN 1530-437X.

ĐORđEVIć, S. et al. Mc sensor - a novel method for measurement of muscle tension. *Sensors*, v. 11, p. 9411–9425, 2011. ISSN 9411-9425.

ANEXO A – Código princial

```
#include <iostream>
#include <opencv2/core/core.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <opencv2/videoio/videoio.hpp>
#include <opencv2/imgproc/imgproc.hpp>
#include <opencv2/calib3d/calib3d.hpp>
#include "ControleCor.h"
#include "Lista.h"
#include "CalculaCoordenadas.h"

#include <stdio.h>
#include <stdlib.h>

#define VERMELHO 0
#define VERDE 1
#define AZUL 2

#define TAMANHO_MEDIA 10
#define CAM1  0
#define CAM2  1

using namespace std;
using namespace cv;

CalculaCoordenadas Braco;

int main(int argc, char **argv) {

   Mat  imgTemporaria;

   //Abre cameras
   if(!Braco.openCameras(CAM1, CAM2))
      return -1;

   //Coloca imagem com menu na tela
   imgTemporaria = Braco.getCamera(CAM1);
   Mat usuario = Mat::zeros(imgTemporaria.size(), CV_8UC3);
```

```
putText(usuario, "1- Verificar marcadores.", Point(50, 100),
    FONT_HERSHEY_COMPLEX_SMALL, 1 , Scalar(0, 0, 255));
putText(usuario, "2- Calibrar marcadores.", Point(50, 130),
    FONT_HERSHEY_COMPLEX_SMALL, 1 , Scalar(0, 0, 255));
putText(usuario, "Esc - Calcular coordendas", Point(50, 160),
    FONT_HERSHEY_COMPLEX_SMALL, 1 , Scalar(0, 0, 255));

while(1)
{
   imshow("Menu", usuario);
   int key = waitKey(0);
   if(key == '1')     //Verifica cores dos marcadores
   {
      Braco.checkColors(CAM1);
      Braco.checkColors(CAM2);
   }
   else if(key == '2')   //Calibra cores dos marcadores
   {
      Braco.tuneColors(CAM1);
      Braco.tuneColors(CAM2);
   }
   if(key == 27)      //Fecha menu e segue o programa
   {
      destroyAllWindows();
      break;
   }
}

//Variaveis para calibracao com Chessboard
int NumTabuleiros = 10;  //Quantidade de imagens
int NumVerticesHor = 5;  //Quantidade de vertices na horizontal
int NumVerticesVer = 3;  //Quantidade e vertices na vertical
int TamanhoQuadrado = 15; //Tamanho dos quadrados em mm

//Realiza a captura das imagens do tabuleiro utilizado como padrao de
    calibracao
if(Braco.captureChessboards(NumTabuleiros, NumVerticesHor, NumVerticesVer,
    TamanhoQuadrado))
   Braco.calibrateCameras();
else
   return -1;    //Se algum erro durante a captura (ou nao pegar todas as
       imagens), encerra programa
```

```
cout << "Comecar a fazer o remap com a retificacao" << endl;

//Variaveis para calculo das coordenadas
vector<Point3f> Ponto3D;
vector<Point3f> Ponto3DReal;
Point3f TCPcoord;
Point3f TCPangulo;
float AberturaGarra;

char Comando[9];
int i = 0;
bool flag_TCP;

Mat canvas;

Ponto3D.resize(3);
Ponto3DReal.resize(3);

//Define tamanho da media movel
Braco.setMovingAvaregeSize(TAMANHO_MEDIA);

while(1)
{
   //Realiza o remapeamento das imagens para retirar distorcoes
   canvas = Braco.remapImages();

   //Pega coordenadas dos marcadores
   Ponto3D[VERMELHO] = Braco.getMarkerCoordinates(VERMELHO);
   Ponto3D[VERDE] = Braco.getMarkerCoordinates(VERDE);
   Ponto3D[AZUL] = Braco.getMarkerCoordinates(AZUL);

   //Pega coordenadas dos marcadores no mundo real com base no mapa de
       disparidade
   Ponto3DReal = Braco.getRealWorldCoordinates(Ponto3D);

   //Realiza a mudanca de base se a base tiver sido definida
   Ponto3DReal[VERMELHO] = Braco.changeBase(Ponto3DReal[VERMELHO]);
   Ponto3DReal[VERDE] = Braco.changeBase(Ponto3DReal[VERDE]);
   Ponto3DReal[AZUL] = Braco.changeBase(Ponto3DReal[AZUL]);

   //Calcula informacoes da TCP
```

```
    TCPcoord = Braco.getTCPCoordinates(Ponto3DReal[VERMELHO],
        Ponto3DReal[AZUL]);
    TCPangulo = Braco.getTCPAngles(Ponto3DReal[VERMELHO], Ponto3DReal[AZUL]);
    AberturaGarra = Braco.getMarkersDistance(Ponto3DReal[VERMELHO],
        Ponto3DReal[AZUL]);

    //Coloca dados calculados na imagem das cameras
    if(flag_TCP)
       Braco.putCoordsInImage(canvas, TCPcoord, TCPangulo,
           Ponto3DReal[VERDE]);
    else
       Braco.putCoordsInImage(canvas, Ponto3DReal[VERMELHO],
           Ponto3DReal[VERDE], Ponto3DReal[AZUL]);

    imshow("Rectified", canvas); //atualiza imagem das cameras
    int key = waitKey(1);
    if(key==27)              //Encerra o programa
       return 0;
    else if(key == 'b')        //Define marcador VERDE como base
       Braco.setBaseCoordinates(Ponto3DReal[VERDE]);
    else if(key == 'r')        //Limpa coordenda da base
       Braco.clearBaseCoordinates();
    else if(key == 't')        //Coloca na imagem coordenadas da TCP
       flag_TCP = true;
    else if(key == 'c')        //Coloca na imagem coordenadas dos marcadores
       flag_TCP = false;
    else if(key==' ')       //Monta vetor para envio de informacoes
    {
       Comando[i++] = 170;        //cabecalho 0xAA
       Comando[i++] = 187;        //cabecalho 0xBB
       Comando[i++] = TCPcoord.x;  //coordenada x
       Comando[i++] = TCPcoord.y;  //coordenada y
       Comando[i++] = TCPcoord.z;  //coordenada z
       Comando[i++] = TCPangulo.x;  //angulo alpha
       Comando[i++] = TCPangulo.y;   //angulo beta
       Comando[i++] = TCPangulo.z;   //angulo gama
       Comando[i++] = AberturaGarra; //distancia entre marcadores

       i = 0;
    }
}
return 1;
```

```
}
```

ANEXO B – Classe CalculaCoordendas

```
#ifndef CALCULACOORDENADAS_H_
#define CALCULACOORDENADAS_H_

#include "ControleCor.h"
#include "Lista.h"

class CalculaCoordenadas
{
private:
   VideoCapture Cap1, Cap2;
   ControleCor Cam1, Cam2;
   int indexCam1, indexCam2;
   vector<vector<Point3f> > objectPoints;
   vector<vector<Point2f> > imagePoints1;
   vector<vector<Point2f> > imagePoints2;
   Mat cameraMatrix[2];
   Mat distCoeffs[2];
   Mat R, T, E, F;
   Mat Q;
   Rect validRoi[2];
   bool isVerticalStereo;
   Mat canvas;
   double sf;
   int w, h;
   Rect Area[2];
   Mat rmap[2][2];
   Mat CameraStereo[2];
   Mat temp;
   int MovingAvaregeSize;
   Lista MediaM[3];
   Point3f Base;
   bool FlagBase;

public:
   CalculaCoordenadas();
   virtual ~CalculaCoordenadas();
```

```
    int openCameras(int cam_1, int cam_2);   //Abre cameras indicadas pelo
        usuario
    int captureChessboards(int num_boards, int num_corners_H, int
        num_corners_V, int square_size); //Pega imagens para calibrar as cameras
    Mat calibrateCameras(void);              //Calibra as cameras e define
        matriz remap
    Mat remapImages(void);                //Re-mapeia as imagens das cameras
        para eliminar distorcoes
    Point3f getMarkerCoordinates(int marker);  //Retorna a coordenada no mundo
        real do marcador selecionado
    vector<Point3f> getRealWorldCoordinates(vector<Point3f> coordinates_pixel);
         //Calcula coordenadas no mundo real
    Point3f changeBase(Point3f point);       //realiza mudanca de base
    void setMovingAvaregeSize(int size);     //define tamanho da media movel
    void setBaseCoordinates(Point3f base_coord); //define coordenadas da nova
        base
    void clearBaseCoordinates(void);      //apaga coordenadas da nova base
    Point3f getBaseCoordinates(void);        //pega coordenadas da nova base
    Mat getCamera(int index_camera);      //captura imagem da camera
    Point3f getTCPCoordinates(Point3f marker_1, Point3f marker_2);  //cacula
        coordenadas do TCP de acordo com marcadores
    Point3f getTCPAngles(Point3f marker_1, Point3f marker_2);   //cacula
        angulos do TCP de acordo com marcadores
    float getMarkersDistance(Point3f marker_1, Point3f marker_2);  //calcula
        distancia entre marcadores
    Mat putCoordsInImage(Mat images_remaped, Point3f coord1, Point3f coord2,
        Point3f coord3); //Escreve coordendas na imagem
    bool tuneColors(int index_cam);          //Calibracao de cor dos marcadores
    bool checkColors(int index_cam);      //Verificacao de cores dos marcadores
};

#endif /* CALCULACOORDENADAS_H_ */
```

```
#include "CalculaCoordenadas.h"
#include <iostream>

#include <opencv2/core/core.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <opencv2/videoio/videoio.hpp>
#include <opencv2/imgproc/imgproc.hpp>
#include <opencv2/calib3d/calib3d.hpp>
```

```
#include "ControleCor.h"
#include "Lista.h"

\#include <stdio.h>
\#include <stdlib.h>
using namespace std;
using namespace cv;

CalculaCoordenadas::CalculaCoordenadas() {
   MovingAvaregeSize = 0;
   FlagBase = false;
   indexCam1 = 0;
   indexCam2 = 0;
}

CalculaCoordenadas::~CalculaCoordenadas() {
   // TODO Auto-generated destructor stub
}

/*
 * Abre as cameras de indices "cam_1" e "cam_2" no computador e armazena esses
     indices para referencia futura
 */
int CalculaCoordenadas::openCameras(int cam_1, int cam_2)
{
   Cap1.open(cam_1);                                    //Abre primeira camera
       (Cam_1)
   Cap2.open(cam_2);                                    //Abre segunda camera
       (Cam_2)

   if( !Cap1.isOpened() )                               //Verifica se Cam_1 OK
   {
      cout << "Could not initialize cap1"<< endl;
      return 0;
   }

   if( !Cap2.isOpened() )                               //Verifica se Cam_2 OK
   {
      cout << "Could not initialize cap2"<< endl;
      return 0;
   }
```

```
   indexCam1 = cam_1;
   indexCam2 = cam_2;
   return 1;
}

/*
 * Armazena as coordenadas dos pontos dos tabuleiros utilizados para a
     calibracao das cameras
 * O usuario deve informar a quantidade de imagens de tabuleiros para a
     calibracao, a quantidade
 * de quinas internas na horizontal e na vertical e o tamanho (em mm) da
     lateral dos quadrados do
 * tabuleiro. O usuario deve utilizar a barra de espaco para capturar a imagem
     do tabuleiro e "esc"
 * para cancelar a funcao
 */
int CalculaCoordenadas::captureChessboards(int num_boards, int num_corners_H,
    int num_corners_V, int square_size)
{
   int numSquares = num_corners_H * num_corners_V;
   Size board_sz = Size(num_corners_H, num_corners_V);
   vector<Point2f> corners_1;
   vector<Point2f> corners_2;
   int successes=0;
   Mat image_1, image_2;
   Mat gray_image_1, gray_image_2;

   //Preenchendo vetor de objetos
   vector<Point3f> obj;
   for(int j=0;j<numSquares;j++)
   {
      obj.push_back(Point3f((j/num_corners_H)*square_size,
          (j%num_corners_H)*square_size, 0.0f));
   }

   //loop de aquisicao
   while(successes < num_boards)
   {
      Cap1 >> image_1;
      Cap2 >> image_2;

      //fez copia de imagem em preto e branco para procurar tabuleiro
```

```
cvtColor(image_1, gray_image_1, CV_BGR2GRAY);
cvtColor(image_2, gray_image_2, CV_BGR2GRAY);

//procura tabuleiro e salva pontos
bool found_1 = findChessboardCorners(image_1, board_sz, corners_1,
    CV_CALIB_CB_ADAPTIVE_THRESH | CV_CALIB_CB_FILTER_QUADS);
bool found_2 = findChessboardCorners(image_2, board_sz, corners_2,
    CV_CALIB_CB_ADAPTIVE_THRESH | CV_CALIB_CB_FILTER_QUADS);

if(found_1 && found_2)
{
   //refina os pontos encontrados e desenha o tabuleiro na imagem
   cornerSubPix(gray_image_1, corners_1, Size(11, 11), Size(-1, -1),
      TermCriteria(CV_TERMCRIT_EPS | CV_TERMCRIT_ITER, 30, 0.1));
   drawChessboardCorners(gray_image_1, board_sz, corners_1, found_1);
   cornerSubPix(gray_image_2, corners_2, Size(11, 11), Size(-1, -1),
      TermCriteria(CV_TERMCRIT_EPS | CV_TERMCRIT_ITER, 30, 0.1));
   drawChessboardCorners(gray_image_2, board_sz, corners_2, found_2);
}

//atualiza imagens nas janelas
imshow("Cam1_gray", gray_image_1);
imshow("Cam2_gray", gray_image_2);

int key = waitKey(1);
if(key==27)
{
   return 0;
}
if((key==' ') && (found_1 != 0) && (found_2 != 0))
{
   imagePoints1.push_back(corners_1);
   imagePoints2.push_back(corners_2);
   objectPoints.push_back(obj);

   printf("Snap stored!");

   successes++;
   cout <<"Sucessos: "<< successes <<endl;
   if(successes>=num_boards)
   {
      //Fecha as janelas
```

```
            destroyWindow("Cam1_gray");
            destroyWindow("Cam2_gray");
            break;
          }
        }
    }
    return 1;
}

/*
 * Realiza a calibracao das cameras e a retificacao, informa o erro RMS da
     calibracao e o erro do alinhamento
 * vertical entre as cameras. Retorna ao usuario a matriz Q, que define a
     relacao entre disparidade e profundidade
 */
Mat CalculaCoordenadas::calibrateCameras(void)
{
    Mat image_1, image_2;
    Mat R1, R2, P1, P2;

    Cap1 >> image_1;
    Cap2 >> image_2;

    //Calculando matrix das cameras
    cameraMatrix[0] =
        initCameraMatrix2D(objectPoints,imagePoints1,image_1.size(),0);
    cameraMatrix[1] =
        initCameraMatrix2D(objectPoints,imagePoints2,image_2.size(),0);

    cout << "Comecou stereoCalibrate" << endl;
    //calibracao estereo
    double rms = stereoCalibrate(objectPoints, imagePoints1, imagePoints2,
        cameraMatrix[0], distCoeffs[0], cameraMatrix[1], distCoeffs[1],
          image_1.size(), R, T, E, F, CALIB_FIX_ASPECT_RATIO +
              CALIB_ZERO_TANGENT_DIST + CALIB_USE_INTRINSIC_GUESS +
              CALIB_SAME_FOCAL_LENGTH +
          CALIB_RATIONAL_MODEL + CALIB_FIX_K3 + CALIB_FIX_K4 + CALIB_FIX_K5,
              TermCriteria(TermCriteria::COUNT+TermCriteria::EPS, 100, 1e-5) );

    cout << "Finalizado com erro RMS = " << rms << endl;

    cout << "Comecando stereoRectify" << endl;
```

```
    stereoRectify(cameraMatrix[0], distCoeffs[0], cameraMatrix[1],
        distCoeffs[1], image_1.size(), R, T, R1, R2, P1, P2, Q,
               CALIB_ZERO_DISPARITY, 1, image_1.size(), &validRoi[0],
                   &validRoi[1]);

    cout << "Terminou stereoRectify" << endl;

    isVerticalStereo = fabs(P2.at<double>(1, 3)) > fabs(P2.at<double>(0, 3));

    cout << "Comecando unitUndistortRectifyMap" << endl;

    initUndistortRectifyMap(cameraMatrix[0], distCoeffs[0], R1, P1,
        image_1.size(), CV_16SC2, rmap[0][0], rmap[0][1]);
    initUndistortRectifyMap(cameraMatrix[1], distCoeffs[1], R2, P2,
        image_1.size(), CV_16SC2, rmap[1][0], rmap[1][1]);

    cout << "Terminou unitUndistortRectifyMap" << endl;

    cout << "Verificando se e estereo na vertical" << endl;

    if( !isVerticalStereo )
    {
        sf = 600./MAX(image_1.size().width, image_1.size().height);
        w = cvRound(image_1.size().width*sf);
        h = cvRound(image_1.size().height*sf);
        canvas.create(h, w*2, CV_8UC3);
        cout << "nao stereo vertical. sf:" << sf << endl;
    }
    else
    {
        sf = 300./MAX(image_1.size().width, image_1.size().height);
        w = cvRound(image_1.size().width*sf);
        h = cvRound(image_1.size().height*sf);
        canvas.create(h*2, w, CV_8UC3);
        cout << " stereo vertical. sf:" << sf << endl;
    }

    return Q;
}

/*
```

```
 * Remapeia as imagens das cameras retirando as distorcoes e realizando o
     alinhamento vertical das duas.
 * Retorna ao usuario a matriz Canvas que contem a imagem das duas cameras
     remapeada
 */
Mat CalculaCoordenadas::remapImages(void)
{
   Mat image_1, image_2;
   Mat rimg_1, cimg_1, rimg_2, cimg_2;
   Mat canvasPart;

   Cap1 >> image_1;
   Cap2 >> image_2;

   remap(image_1, rimg_1, rmap[0][0], rmap[0][1], INTER_LINEAR);
   remap(image_2, rimg_2, rmap[1][0], rmap[1][1], INTER_LINEAR);

   for(int k = 0; k< 2; k++)
   {
      canvasPart = !isVerticalStereo ? canvas(Rect(w*k, 0, w, h)) :
          canvas(Rect(0, h*k, w, h));
      resize(k==0 ? rimg_1 : rimg_2, canvasPart, canvasPart.size(), 0, 0,
          INTER_AREA);
      Rect vroi(cvRound(validRoi[k].x*sf), cvRound(validRoi[k].y*sf),
            cvRound(validRoi[k].width*sf), cvRound(validRoi[k].height*sf));
      rectangle(canvasPart, vroi, Scalar(0,0,255), 3, 8);

      CameraStereo[k] = canvasPart;
      Area[k] = vroi;
   }

   if( !isVerticalStereo )
      for( int j = 0; j < canvas.rows; j += 16 )
         line(canvas, Point(0, j), Point(canvas.cols, j), Scalar(0, 255, 0), 1,
             8);
   else
      for( int j = 0; j < canvas.cols; j += 16 )
         line(canvas, Point(j, 0), Point(j, canvas.rows), Scalar(0, 255, 0), 1,
             8);

   return canvas;
}
```

```
/*
 * Identifica as coordenadas em pixel o marcador indicado pelo usuario
    (marker) e retorna um Point3f com
 * as coordenadas do ponto para a segunda camera a distancia dele para o ponto
    da primeira camera (disparidade).
 * Esse Point3f deve ser utilizado no vetor de entrada da funcao
    getRealWorldCoordinates.
 * Caso o usuario tenha definido um tamanho para a media movel aplicada ao
    ponto de forma
 * a suaviazar a mudanca de valores obtidos
 */
Point3f CalculaCoordenadas::getMarkerCoordinates(int marker)
{
   Mat imageHVS_1, imageHVS_2;
   Mat temp_1, temp_2;
   Point3f Point3D;
   Point Pnt_1, Pnt_2;

   //passa as imagens para HSV
   cvtColor(CameraStereo[0], imageHVS_1, COLOR_BGR2HSV);
   cvtColor(CameraStereo[1], imageHVS_2, COLOR_BGR2HSV);

   //Filtra as imagens
   switch(marker)
   {
      case 0:
         temp_1 = Cam1.markerFilter(imageHVS_1, 0);
         temp_2 = Cam2.markerFilter(imageHVS_2, 0);
         break;

      case 1:
         temp_1 = Cam1.markerFilter(imageHVS_1, 1);
         temp_2 = Cam2.markerFilter(imageHVS_2, 1);
         break;

      case 2:
         temp_1 = Cam1.markerFilter(imageHVS_1, 2);
         temp_2 = Cam2.markerFilter(imageHVS_2, 2);
         break;
   }
```

```
   //Identifica as coordenadas em pixel das imagens filtradas
   Pnt_1 = Cam1.pntTracking(temp_1, Area[0]);
   Pnt_2 = Cam2.pntTracking(temp_2, Area[1]);

   //Monta point de retorno
   Point3D = Point3f(Pnt_2.x, Pnt_2.y, Pnt_2.x - Pnt_1.x);

   if(MovingAvaregeSize > 0)
   {
      MediaM[marker].addItem(Point3D);
      Point3f Media = MediaM[marker].getTotal();
      int Tamanho_Lista = MediaM[marker].getTamanho();
      Point3D = Point3f(Media.x/Tamanho_Lista, Media.y/Tamanho_Lista,
          Media.z/Tamanho_Lista);
      if(Tamanho_Lista == MovingAvaregeSize)
         MediaM[marker].removeItem();
   }

   return Point3D;
}

/*
 * Retorna ao usuario vetor de Point3f com as coordenadas no mundo real dos
     Point3f informados por ele
 */
vector<Point3f> CalculaCoordenadas::getRealWorldCoordinates(vector<Point3f>
    coordinates_pixel)
{
   vector<Point3f> coordinates_real;
   perspectiveTransform(coordinates_pixel, coordinates_real, Q);

   return coordinates_real;
}

/*
 * Define o tamanho da media movel
*/
void CalculaCoordenadas::setMovingAvaregeSize(int size)
{
   MovingAvaregeSize = size;
}
```

```
/*
 * Define coordenadas da base.
 */
void CalculaCoordenadas::setBaseCoordinates(Point3f base_coord)
{
   Base.x = base_coord.x;
   Base.y = base_coord.y;
   Base.z = base_coord.z;
   FlagBase = true;
}

/*
 * Reseta coordenadas da base
 */
void CalculaCoordenadas::clearBaseCoordinates(void)
{
   Base = Point3f(0, 0, 0);
   FlagBase = false;
}

/*
 * Retorna coordendas da base
 */
Point3f CalculaCoordenadas::getBaseCoordinates(void)
{
   return Base;
}

/*
 * Retorna matriz com imagem da camera do indice informado pelo usuario.
 * Atencao: deve ser usada apenas depois que as cameras forem abertas
 */
Mat CalculaCoordenadas::getCamera(int index_camera)
{
   if(index_camera == indexCam1)
      Cap1.read(temp);
   else if(index_camera == indexCam2)
      Cap2.read(temp);

   return temp;
}
```

```
/*
 * O usuario informa a imagem e 3 Points de coordendas a serem escritas na
     imagem. Caso deseje menos pontos,
 * basta colocar o(s) outro(s) como Point3f(0, 0, 0)
 */
Mat CalculaCoordenadas::putCoordsInImage(Mat images_remaped, Point3f coord1,
    Point3f coord2, Point3f coord3)
{
   Mat canvas_aux = images_remaped;

   if(coord1 != Point3f(0, 0, 0))
   {
      ostringstream coordPntRx, coordPntRy, coordPntRz;
      coordPntRx << "X =" << coord1.x;
      string posCordR = coordPntRx.str();
      putText(canvas_aux, posCordR.c_str(), Point(100,100),
          FONT_HERSHEY_COMPLEX_SMALL, 1 , Scalar(0, 0, 255));
      coordPntRy << "Y =" << coord1.y;
      posCordR = coordPntRy.str();
      putText(canvas_aux, posCordR.c_str(), Point(100,130),
          FONT_HERSHEY_COMPLEX_SMALL, 1 , Scalar(0, 0, 255));
      coordPntRz << "Z =" << coord1.z;
      posCordR = coordPntRz.str();
      putText(canvas_aux, posCordR.c_str(), Point(100,160),
          FONT_HERSHEY_COMPLEX_SMALL, 1 , Scalar(0, 0, 255));
   }

   if(coord2 != Point3f(0, 0, 0))
   {
      ostringstream coordPntGx, coordPntGy, coordPntGz;
      coordPntGx << "X =" <<coord2.x;
      string posCordG = coordPntGx.str();
      putText(canvas_aux, posCordG.c_str(), Point(100,200),
          FONT_HERSHEY_COMPLEX_SMALL, 1 , Scalar(0, 255, 0));
      coordPntGy << "Y =" << coord2.y;
      posCordG = coordPntGy.str();
      putText(canvas_aux, posCordG.c_str(), Point(100,230),
          FONT_HERSHEY_COMPLEX_SMALL, 1 , Scalar(0, 255, 0));
      coordPntGz << "Z =" << coord2.z;
      posCordG = coordPntGz.str();
      putText(canvas_aux, posCordG.c_str(), Point(100,260),
          FONT_HERSHEY_COMPLEX_SMALL, 1 , Scalar(0, 255, 0));
```

```
    }

    if(coord3 != Point3f(0, 0, 0))
    {
        ostringstream coordPntBx, coordPntBy, coordPntBz;
        coordPntBx << "X =" << coord3.x;
        string posCordB = coordPntBx.str();
        putText(canvas_aux, posCordB.c_str(), Point(100,300),
            FONT_HERSHEY_COMPLEX_SMALL, 1 , Scalar(255, 0, 0));
        coordPntBy << "Y =" << coord3.y;
        posCordB = coordPntBy.str();
        putText(canvas_aux, posCordB.c_str(), Point(100,330),
            FONT_HERSHEY_COMPLEX_SMALL, 1 , Scalar(255, 0, 0));
        coordPntBz << "Z =" << coord3.z;
        posCordB = coordPntBz.str();
        putText(canvas_aux, posCordB.c_str(), Point(100,360),
            FONT_HERSHEY_COMPLEX_SMALL, 1 , Scalar(255, 0, 0));
    }

    return canvas_aux;
}

/*
 * Retorna coordenadas do ponto medio entre os 2 marcadores informados pelo
     usuario
 */
Point3f CalculaCoordenadas::getTCPCoordinates(Point3f marker_1, Point3f
    marker_2)
{
    Point3f PontoMedio;

    PontoMedio.x = (marker_1.x + marker_2.x)/2;
    PontoMedio.y = (marker_1.y + marker_2.y)/2;
    PontoMedio.z = (marker_1.z + marker_2.z)/2;

    return PontoMedio;
}

/*
 * Retorna angulo entre a reta formada pelos 2 marcadores informados pelo
     usuario e os 3 eixos de coordendas
 */
```

```
Point3f CalculaCoordenadas::getTCPAngles(Point3f marker_1, Point3f marker_2)
{
   float x, y, z, A, B;
   Point3f angles;

   x = marker_1.x - marker_2.x;
   y = marker_1.y - marker_2.y;
   z = marker_1.z - marker_2.z;
   A = sqrt((x*x) + (y*y));
   B = sqrt((x*x) + (z*z));

   angles.x = acos(x/A);
   angles.y = acos(y/A);
   angles.z = acos(z/B);

   return angles;
}

/*
 * Retorna distancia entre os dois marcadores informados pelo usuario
 */
float CalculaCoordenadas::getMarkersDistance(Point3f marker_1, Point3f
    marker_2)
{
   float x, y, z, dist;
   x = marker_1.x - marker_2.x;
   y = marker_1.y - marker_2.y;
   z = marker_1.z - marker_2.z;

   dist = sqrt((x*x) + (y*y) + (z*z));

   return dist;
}

/*
 * Realiza a mudanca de base para que foi definida pelo usuario. Retorna as
     coordendas do ponto informado
 * na nova base
 */
Point3f CalculaCoordenadas::changeBase(Point3f point)
{
   if(FlagBase)
```

```
   {
      Point3f NovoPonto;

      NovoPonto.x = Base.x - point.x;
      NovoPonto.y = point.z - Base.z;
      NovoPonto.z = point.y - Base.y;

      return NovoPonto;
   }

   return point;
}

/*
 * Calibra as cores dos marcadores. O usuario deve definir os valores de HSV
    no controle e digitar o numero
 * do marcador correspondente ao finalizar (0, 1 ou 2)
 */
bool CalculaCoordenadas::tuneColors(int index_cam)  //Calibracao de Cor
{
   Mat frame, imageHVS;
   bool Flag_M0, Flag_M1, Flag_M2;
   Mat color;

   Flag_M0 = false;
   Flag_M1 = false;
   Flag_M2 = false;

   //Cria janela de controle de valores
   namedWindow("HSV Control", CV_WINDOW_AUTOSIZE);
   int iLowH = 0;
   int iHighH = 179;

   int iLowS = 0;
   int iHighS = 255;

   int iLowV = 0;
   int iHighV = 255;

   cvCreateTrackbar("LowH", "HSV Control", &iLowH, 179); //Hue (0 - 179)
   cvCreateTrackbar("HighH", "HSV Control", &iHighH, 179);
```

```
cvCreateTrackbar("LowS", "HSV Control", &iLowS, 255); //Saturation (0 - 255)
cvCreateTrackbar("HighS", "HSV Control", &iHighS, 255);

cvCreateTrackbar("LowV", "HSV Control", &iLowV, 255); //Value (0 - 255)
cvCreateTrackbar("HighV", "HSV Control", &iHighV, 255);

//Enquanto faltar marcador para calibrar
while((Flag_M0==false) || (Flag_M1==false) || (Flag_M2==false))
{
   //captura imagem da camera
   if(index_cam == indexCam1)
   {
      if(!Cap1.read(frame))
      break;
   }
   else if(index_cam == indexCam2)
   {
      if(!Cap2.read(frame))
      break;
   }
   else
      return false;

   //filtra a imagem com o valor do controle
   cvtColor(frame, imageHVS, COLOR_BGR2HSV);
   inRange(imageHVS, Scalar(iLowH, iLowS, iLowV), Scalar(iHighH, iHighS,
       iHighV), color);
   erode(color, color, getStructuringElement(MORPH_ELLIPSE, Size(5, 5)) );
   dilate( color, color, getStructuringElement(MORPH_ELLIPSE, Size(5, 5)) );

   dilate( color, color, getStructuringElement(MORPH_ELLIPSE, Size(5, 5)) );
   erode(color, color, getStructuringElement(MORPH_ELLIPSE, Size(5, 5)) );

   //mostra resultado da filtragem
   imshow("Camera", color);

   int key = waitKey(1);
   if(key==27)      //Se usuario apertar "esc", sai da calibracao
   {
      destroyAllWindows();
      return false;
   }
```

```
      else if((key == '0') || (key == '1') || (key == '2'))  //Se usuario
          identificar o marcador, salva os dados
      {
         if(key == '0')
            Flag_M0 = true;
         else if(key == '1')
            Flag_M1 = true;
         else if(key == '2')
            Flag_M2 = true;

         int Marker = key - 48;

         if(index_cam == indexCam1)
            Cam1.setMarkerColor(Marker, iLowH, iHighH, iLowS, iHighS, iLowV,
                iHighV);
         else if(index_cam == indexCam2)
            Cam2.setMarkerColor(Marker, iLowH, iHighH, iLowS, iHighS, iLowV,
                iHighV);

         cout << "Marcador " << key << ":" << iLowH << " , " << iHighH << " , "
             << iLowS << " , " << iHighS << " , " << iLowV << " , " << iHighV
             << endl;
         Flag_M0 = true;
      }
   }
   destroyAllWindows();
   return true;
}

/*
 * Abre janelas com filtro das 3 cores selecionadas para o marcador para que o
     usuario verifique se e necessaria
 * nova calibracao de cor
 */
bool CalculaCoordenadas::checkColors(int index_cam)
{
   Mat frame, imageHVS, marker0, marker1, marker2;

   while(1)
   {
      //captura imagem da camera
      if(index_cam == indexCam1)
```

```
{
   if(!Cap1.read(frame))
   break;
}
else if(index_cam == indexCam2)
{
   if(!Cap2.read(frame))
   break;
}
else
   return false;

//Passa imagem para HSV e filtra para os marcadores
cvtColor(frame, imageHVS, COLOR_BGR2HSV);
if(index_cam == indexCam1)
{
   marker0 = Cam1.markerFilter(imageHVS, 0);            //MAT da Cam1
       filtrada por cor vermelho
   marker1 = Cam1.markerFilter(imageHVS, 1);            //MAT da Cam1
       filtrada por cor azul
   marker2 = Cam1.markerFilter(imageHVS, 2);            //MAT da Cam1
       filtrada por cor verde
}
else if(index_cam == indexCam2)
{
   marker0 = Cam2.markerFilter(imageHVS, 0);            //MAT da Cam2
       filtrada por cor vermelho
   marker1 = Cam2.markerFilter(imageHVS, 1);            //MAT da Cam2
       filtrada por cor azul
   marker2 = Cam2.markerFilter(imageHVS, 2);            //MAT da Cam2
       filtrada por cor verde
}

//mostra as janelas com as imagens filtradas
imshow("Marker 0", marker0);
imshow("Marker 1", marker1);
imshow("Marker 2 ", marker2);
int key = waitKey(1);
if(key==27)   //se usuario apertar "esc", sai da verificacao
{
   break;
}
```

```
    }

    destroyAllWindows();
    return true;
}
```

ANEXO C – Classe ControleCor

```
#ifndef CONTROLECOR_H_
#define CONTROLECOR_H_

#include <iostream>
#include <opencv2/core/core.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <opencv2/videoio/videoio.hpp>
#include <opencv2/imgproc/imgproc.hpp>
#include <opencv2/calib3d/calib3d.hpp>

#include <stdio.h>
#include <stdlib.h>

using namespace std;
using namespace cv;

class ControleCor
{
   private:
      int rLowH, rLowS, rLowV, rHighH, rHighS, rHighV;
      int gLowH, gLowS, gLowV, gHighH, gHighS, gHighV;
        int bLowH, bLowS, bLowV, bHighH, bHighS, bHighV;

   public:
      ControleCor();
      virtual ~ControleCor();

      void setMarkerColor(int marker, int LowH, int HighH, int LowS, int
         HighS, int LowV, int HighV);  //define cor HSV do marcador
      Mat markerFilter(const Mat& src, int marker);    //Filtra imagem em
         relacao a cor do marcador
      Point pntTracking(Mat Image, Rect availableArea);  //Identifica
         centroide da area da imagem (filtrada)
};

#endif /* CONTROLECOR_H_ */
```

```
#include "ControleCor.h"
#include <opencv2/core/core.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <opencv2/videoio/videoio.hpp>
#include <opencv2/imgproc/imgproc.hpp>
#include <opencv2/calib3d/calib3d.hpp>

using namespace std;
using namespace cv;

ControleCor::ControleCor() {
   // TODO Auto-generated constructor stub
   rLowH = 15;
   rHighH = 50;
   rLowS = 111;
   rHighS = 255;
   rLowV = 80;
   rHighV = 233;

   gLowH = 60;
   gHighH = 102;
   gLowS = 74;
   gHighS = 255;
   gLowV = 13;
   gHighV = 255;

   bLowH = 103;
   bHighH = 139;
   bLowS = 149;
   bHighS = 235;
   bLowV = 36;
   bHighV = 241;
}

ControleCor::~ControleCor() {
   // TODO Auto-generated destructor stub
}

/*
 * Define os valores HSV maximos e minimos do marcador indicado pelo usuario
 */
void ControleCor::setMarkerColor(int marker, int LowH, int HighH, int LowS,
```

```
    int HighS, int LowV, int HighV)
{
   switch(marker)
   {
      case 0:
         rLowH = LowH;
         rHighH = HighH;
         rLowS = LowS;
         rHighS = HighS;
         rLowV = LowV;
         rHighV = HighV;
         break;

      case 1:
         gLowH = LowH;
         gHighH = HighH;
         gLowS = LowS;
         gHighS = HighS;
         gLowV = LowV;
         gHighV = HighV;
         break;

      case 2:
         bLowH = LowH;
         bHighH = HighH;
         bLowS = LowS;
         bHighS = HighS;
         bLowV = LowV;
         bHighV = HighV;
         break;
   }
}

/*
 * Filtra a imagem src de acordo com os valores definidos para o marcador
     indicado pelo usuario e
 * retorna imagem filtrada (elemento da cor indicada estara branco e o
     restante preto)
 */
Mat ControleCor::markerFilter(const Mat& src, int marker)
{
   Mat filtered;
```

```
int LowH, LowS, LowV, HighH, HighS, HighV;

switch(marker)
{
   case 0:
      LowH = rLowH;
      LowS = rLowS;
      LowV  = rLowV;
      HighH = rHighH;
      HighS = rHighS;
      HighV = rHighV;
      break;

   case 1:
      LowH = gLowH;
      LowS = gLowS;
      LowV = gLowV;
      HighH = gHighH;
      HighS = gHighS;
      HighV = gHighV;
      break;

   case 2:
      LowH = bLowH;
      LowS = bLowS;
      LowV = bLowV;
      HighH = bHighH;
      HighS = bHighS;
      HighV = bHighV;
      break;
}

inRange(src, Scalar(LowH, LowS, LowV), Scalar(HighH, HighS, HighV),
    filtered);
erode(filtered, filtered, getStructuringElement(MORPH_ELLIPSE, Size(5, 5))
    );
dilate( filtered, filtered, getStructuringElement(MORPH_ELLIPSE, Size(5,
    5)) );

dilate( filtered, filtered, getStructuringElement(MORPH_ELLIPSE, Size(5,
    5)) );
erode(filtered, filtered, getStructuringElement(MORPH_ELLIPSE, Size(5, 5))
```

```
      );

   return filtered;
}

/*
 * Identifica a coordenada do pixel do centroide da imagem de entrada dentro
     da area definida (availableArea)
 */
Point ControleCor::pntTracking(Mat Image, Rect availableArea)
{
   Point Point_Track;
   Moments Moments = moments(Image);

   double M01 = Moments.m01;
   double M10 = Moments.m10;
   double Area = Moments.m00;

   if(Area > 10000)
   {
      int posX = M10/Area;
      int posY = M01/Area;

      if((posX >= 0) && (posY >= 0) && (posX >= availableArea.x) && (posY >=
          availableArea.y) &&
            (posX <= availableArea.width) && (posY <= availableArea.height))
      {
         Point_Track = Point(posX, posY);
      }
      else
      {
         Point_Track = Point(0, 0);
      }
   }
   return Point_Track;
}
```

ANEXO D – Classe Lista

```
#ifndef LISTA_H_
#define LISTA_H_

#include "Celula.h"

class Lista
{
private:
   Celula *cabeca;
   Celula *cauda;
   int tamanho;
   Point3f total;

public:
   Lista();
   Lista(Point3f info);
   virtual ~Lista();

   void addItem(Point3f info);        //adiciona item
   void removeItem(void);           //retira o primeiro item da lista
   bool vazia(void);                  //verifica se esta vazia
   int getTamanho(void);              //retorna tamanho real da lista
   Point3f getTotal(void);          //retorna a soma total dos dados da lista
   Celula* getCabeca(void);           //retorna a celula no inicio da lista
   Celula* getCauda(void);            //retorna a celula no fim da lista
   void setCabeca(Celula* head);      //define a celula no inicio da lista
   void setCauda(Celula* tail);       //define a celula no fim da lista
};

#endif /* LISTA_H_ */
```

```
#include "Lista.h"
#include "Celula.h"

#include <iostream>
#include <stdio.h>
using namespace std;
```

```
using namespace cv;

Lista::Lista() {
   cabeca = NULL;
   cauda = NULL;
   tamanho = 0;
   total.x = 0;
   total.y = 0;
   total.z = 0;

}

Lista::Lista(Point3f info)
{
   cabeca = new Celula(info);
   cauda = cabeca;
   tamanho = 1;
   total.x = info.x;
   total.y = info.y;
   total.z = info.z;
}

Lista::~Lista() {
   // TODO Auto-generated destructor stub
}

/*
 * Adiciona item ao fim da lista
 */
void Lista::addItem(Point3f info)
{
    Celula* nova_celula = new Celula(info);

   if(vazia())
   {
      cabeca = nova_celula;
      cauda = cabeca;
      tamanho = 1;
   }
   else
   {
      cauda->setProx(nova_celula);
```

```
      cauda = nova_celula;
      tamanho++;
   }

   total.x = total.x + info.x;
   total.y = total.y + info.y;
   total.z = total.z + info.z;
}

/*
 * Retira primeiro item da lista
 */
void Lista::removeItem(void)
{
   Celula *pnt_aux;
   Point3f info = cabeca->getInfo();
   total.x = total.x - info.x;
   total.y = total.y - info.y;
   total.z = total.z - info.z;

   pnt_aux = cabeca;
   cabeca = cabeca->getProx();
   free(pnt_aux);
   tamanho--;
}

/*
 * Retorna verdadeiro se a lista estiver vazia
 */
bool Lista::vazia(void)
{
     return(cabeca == NULL);
}

/*
 * /Retorna o tamanho real da lista
 */
int Lista::getTamanho(void)
{
    return tamanho;
}
```

```
/*
 * Retorna a soma total dos dados da lista
 */
Point3f Lista::getTotal(void)
{
   return total;
}

/*
 * Retorna a cabeca da lista
 */
Celula* Lista::getCabeca(void)
{
     return cabeca;
}

/*
 * Retorna a cauda da lista
 */
Celula* Lista::getCauda(void)
{
     return cauda;
}

/*
 * Define a cabeca da lista
 */
void Lista::setCabeca(Celula* head)
{
     cabeca = head;
}

/*
 * Define a cauda da lista
 */
void Lista::setCauda(Celula* tail)
{
     cauda = tail;
}
```

ANEXO E – Classe Celula

```
#ifndef CELULA_H_
#define CELULA_H_

#include <opencv2/calib3d/calib3d.hpp>

#include <stdio.h>
#include <stdlib.h>

using namespace std;
using namespace cv;

class Celula
{
private:
   Point3f info;
   Celula *prox;

public:
   Celula(Point3f information);
   virtual ~Celula();

   void setInfo(Point3f information);  //define informacao da celula
   void setProx(Celula *p);         //define proxima celula da lista
   Point3f getInfo(void);           //pega informacao da celula
   Celula* getProx(void);           //pega proxima celula da lista
};

#endif /* CELULA_H_ */
```

```
#include "Celula.h"

#include <opencv2/calib3d/calib3d.hpp>

#include <stdio.h>
#include <stdlib.h>

using namespace std;
```

```
using namespace cv;

Celula::Celula(Point3f information)
{
   info = information;
   prox = NULL;
}

Celula::~Celula() {
   // TODO Auto-generated destructor stub
}

/*
 * Define o valor da informacao contida na celula
 */
void Celula::setInfo(Point3f information)
{
   info = information;
}

/*
 * Define o ponteiro da proxima celula na lista
 */
void Celula::setProx(Celula *p)
{
   prox = p;
}

/*
 * Retorna o valor da informacao contida na celula
 */
Point3f Celula::getInfo(void)
{
   return info;
}

/*
 * Retorna o pronteiro da proxima celula na lista
 */
Celula* Celula::getProx(void)
{
   return prox;
```

```
}
```