UNIVERSIDADE FEDERAL DE SANTA MARIA
CENTRO DE TECNOLOGIA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Douglas Tybusch

MARCADOR FIDUCIAL COLORIDO E RECURSIVO PARA REALIDADE AUMENTADA: ALGORITMO DE DETECÇÃO E PROJEÇÃO

Santa Maria, RS
2018

Douglas Tybusch

MARCADOR FIDUCIAL COLORIDO E RECURSIVO PARA REALIDADE AUMENTADA: ALGORITMO DE DETECÇÃO E PROJEÇÃO

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação, Área de Concentração em Computação, da Universidade Federal de Santa Maria (UFSM, RS), como requisito parcial para obtenção do grau de **Mestre em Ciência da Computação**.

ORIENTADOR: Prof. Andrei Piccinini Legg

Santa Maria, RS
2018

Tybusch, Douglas
Marcador Fiducial Colorido e Recursivo para Realidade Aumentada: Algoritmo de Detecção e Projeção / Douglas Tybusch.- 2018.
90 p.; 30 cm

Orientador: Andrei Legg
Dissertação (mestrado) - Universidade Federal de Santa Maria, Centro de Tecnologia, Programa de Pós-Graduação em Ciência da Computação, RS, 2018

1. Marcadores Fiduciais 2. Realidade Aumentada 3. Visão Computacional 4. Marcadores Fiduciais Coloridos 5. Detecção de Cores I. Legg, Andrei II. Título.

Sistema de geração automática de ficha catalográfica da UFSM. Dados fornecidos pelo autor(a). Sob supervisão da Direção da Divisão de Processos Técnicos da Biblioteca Central. Bibliotecária responsável Paula Schoenfeldt Patta CRB 10/1728.

©2018

Todos os direitos autorais reservados a Douglas Tybusch. A reprodução de partes ou do todo deste trabalho só poderá ser feita mediante a citação da fonte.

End. Eletr.: Douglas.Tybusch@outlook.com

Douglas Tybusch

MARCADOR FIDUCIAL COLORIDO E RECURSIVO PARA REALIDADE AUMENTADA: ALGORITMO DE DETECÇÃO E PROJEÇÃO

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação, Área de Concentração em Computação, da Universidade Federal de Santa Maria (UFSM, RS), como requisito parcial para obtenção do grau de **Mestre em Ciência da Computação**.

Aprovado em 21 de fevereiro de 2018:

_____

Andrei Piccinini Legg, Dr. (UFSM)
(Presidente/Orientador)

_____

Osmar Marchi dos Santos, Dr. (UFSM)

_____

Edison Pignaton de Freitas, Dr. (UFRGS) (videoconferência)

Santa Maria, RS
2018

RESUMO

MARCADOR FIDUCIAL COLORIDO E RECURSIVO PARA REALIDADE AUMENTADA: ALGORITMO DE DETECÇÃO E PROJEÇÃO

AUTOR: Douglas Tybusch
ORIENTADOR: Andrei Piccinini Legg

Realidade Aumentada é um conceito tecnológico que existe a diversos anos, mas que vem apresentando uma popularidade crescente nos últimos anos, devido a evolução de dispositivos como *smartphones*, em que seus processadores móveis e câmeras evoluem de maneira constante. Uma das principais áreas de pesquisa em realidade aumentada é a detecção de objetos e elementos, como marcadores fiduciais binários. A detecção destes elementos permite a sobreposição de objetos e a criação de ambientes virtuais, possibilitando a interação entre o cenário real e o virtual para o usuário. Neste trabalho é apresentado um Marcador Fiducial Colorido e Recursivo para Realidade Aumentada, é demonstrada a elaboração de sua estrutura e *layout*, e o desenvolvimento de seu algoritmo de detecção. Como resultado, foram implementados com sucesso a utilização de cores no marcador, e a identificação e geração de identificadores únicos baseados em sua estrutura e sequências de cores. Também foi implementado um *layout* hierárquico que permite a detecção dos marcadores criados em situações de oclusão parcial. O algoritmo proposto apresentou performance e precisão de detecção de forma satisfatória, oferecendo uma precisão similar a algoritmos binários clássicos, porém, apresentando performance até 37% mais rápida.

Palavras-chave: Marcadores Fiduciais. Realidade Aumentada. Detecção de Cores. Visão Computacional. Processamento de Imagens. Marcadores Fiduciais Coloridos.

ABSTRACT

COLORED AND RECURSIVE FIDUCIAL MARKER FOR AUGMENTED REALITY: DETECCTION AND PROJECTION ALGORITHM

AUTHOR: Douglas Tybusch
ADVISOR: Andrei Piccinini Legg

Augmented Reality is a technological concept that has existed for several years, but it has been showing increasing popularity in recent years, due to the evolution of devices such as smartphones, where its mobile processors and cameras are constantly evolving. One of the main areas of Augmented Reality research is the detection of objects and elements, such as binary fiducial markers. The detection of these elements allows the overlay of virtual objects and the creation of virtual environments, allowing the interaction between the real and virtual scene for the user. It is presented in this work, a Colored and Recursive Fiducial Marker for Augmented Reality, it is shown the elaboration of its structure and layout, and also the development of its detection algorithm. As a result, the use of color in the marker has been successfully implemented, as well as the identification and generation of unique identifiers, based on its color structure and sequences. We also implemented a hierarchical layout that allows the detection of markers in situations of partial occlusion. The proposed algorithm presented satisfactory performance and detection accuracy, offering precision similar to classic binary algorithms, however, having performance increased by up to 37% faster.

Keywords: Fiducial Markers. Augmented Reality. Color Detection. Computer Vison. Image Processing. Colored Fiducial Markers.

SUMÁRIO

1 INTRODUÇÃO

Realidade Aumentada (RA) é um conceito tecnológico existente a diversas décadas. Seu objetivo básico é a integração e interação de informações digitais com o ambiente do usuário em tempo real. Estes tipos de informações podem ser fornecidos a partir de diversas formas, como sobreposições gráficas, projeções, áudio ou *feedback* de toque ao usuário (TECHNOLOGIES, 2016).

Apesar de ter sido um conceito elaborado a diversas décadas, e bastante explorado cientificamente, até pouco tempo atrás era uma tecnologia que poucas pessoas tinham acesso, devido a seu alto custo de processamento e mobilidade. Com a evolução dos *smartphones*, este cenário foi mudando, a evolução de processadores móveis e de suas câmeras foi permitindo a utilização mais ampla e popularização desta tecnologia.

Um dos principais tópicos de pesquisa em realidade aumentada é a de detecção de objetos e elementos em imagens (GARRIDO-JURADO et al., 2014). A detecção destes elementos permite a sobreposição de objetos e a criação de ambientes virtuais sobre os elementos detectados, possibilitando uma interação entre o cenário real e o virtual (Figura 1.1).

Figura 1.1 – Exemplo de realidade aumentada utilizando marcadores fiduciais.

Fonte: (SOLIDWORKS, 2018).

O foco deste trabalho é criação e detecção destes elementos, mais especificamente, em marcadores fiduciais. Marcadores fiduciais são elementos de formatos e estruturas que buscam facilitar a sua detecção por um algoritmo de processamento de imagens.

Estes marcadores podem prover informações ao algoritmo, como a sua posição, orientação e escala do objeto em relação a imagem sendo analisada, permitindo a sobreposição de objetos virtuais sobre o marcador detectado.

O desenvolvimento do marcador proposto se justifica pela utilização de cores em seu layout, afim de permitir a criação de um número maior de sub-marcadores em comparação a marcadores binários clássicos, que utilizam apenas as cores preto e branco (contrastes). A detecção de diferentes cores em marcadores é um processo que se mostra dificultoso pela necessidade de maior precisão na determinação de cores, assim como maior processamento em relação a algoritmos clássicos de detecção.

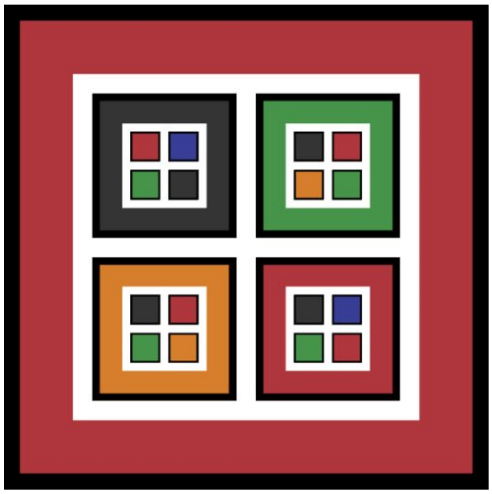
O desenvolvimento deste trabalho também se justifica pela utilização de estruturas hierárquicas, utilizadas principalmente para a detecção do marcador sob diferentes distâncias de detecção e oclusão.

Como principais contribuições deste trabalho, mostra-se a aplicabilidade de marcadores coloridos como alternativa a marcadores binários, tanto em detecção quanto em performance, e também, a utilização de estruturas hierárquicas como processo de filtragem de candidatos, resultando em um aumento de performance total do algoritmo.

1.1 OBJETIVOS

O objetivo principal deste trabalho é a elaboração e apresentação do **Marcador Fiducial Colorido e Recursivo (CRFM)** (TYBUSCH et al., 2017) (Figura 1.2) e o desenvolvimento do algoritmo de detecção do marcador.

Figura 1.2 – Marcador Fiducial Colorido e Recursivo.



Fonte: Próprio autor.

Os objetivos específicos deste trabalho são apresentados a seguir:

- Apresentar e avaliar as principais características do marcador proposto;
- Apresentar e detalhar o algoritmo de detecção proposto:

Algoritmo para detecção de candidatos;

Técnicas para detecção e determinação de cores.

- Avaliação da precisão e performance do algoritmo;
- Comparativos com outros sistemas de marcadores fiduciais propostos.

1.2 ESTRUTURA

O presente trabalho está dividido em 7 capítulos, descritos a seguir.

O Capítulo 2 é apresentado o referencial teórico, exemplificando e descrevendo os principais algoritmos e conceitos utilizados neste trabalho.

No Capítulo 3 são descritos brevemente os principais marcadores fiduciais presentes na literatura, suas características, e evolução dos marcadores como um todo.

No Capítulo 4 é apresentado o marcador proposto, sua estrutura e principais características.

O Capítulo 5 demonstra a implementação do algoritmo de detecção do marcador proposto, e exemplificado o fluxo de detecção do algoritmo.

No Capítulo 6 são apresentados e analisados os testes e resultados de performance e detecção do algoritmo, assim como os comparativos feitos com outros marcadores existentes.

Por fim, no Capítulo 7 poder ser conferidas as principais conclusões quanto ao desenvolvimento deste trabalho, assim como seus resultados e sugestões para trabalhos futuros.

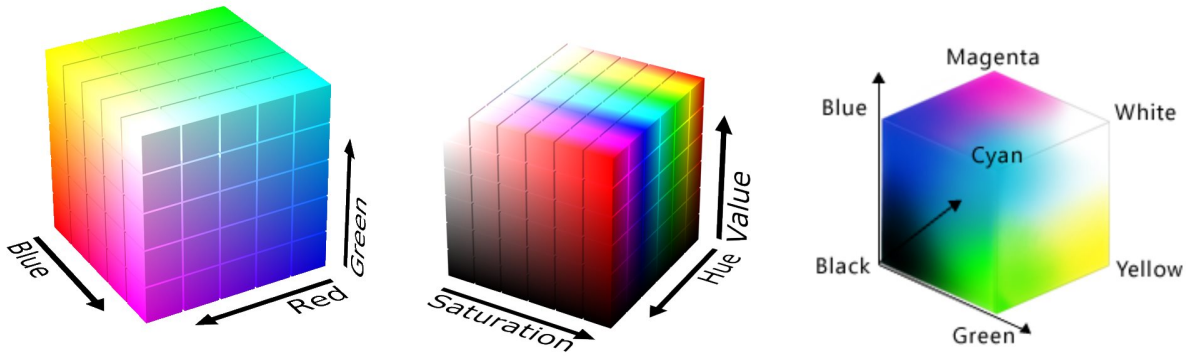
2 REFERENCIAL TEÓRICO

2.1 ESPAÇOS DE CORES

Uma faixa de cores pode ser criada pela combinação de cores primárias de pigmentos, e estas combinações podem definir um espaço de cores em específico. Os espaços de cores também conhecidos como modelos de cores, modelos matemáticos abstratos (ARCSOFT, 2016), utilizados para descrever os intervalos de cores em forma de tuplas, geralmente contendo 3 valores ou componentes de cores, como o RGB, Lab, HSV, CMYK entre outros.

Um espaço de cores pode ser definido como uma elaboração de um sistema de coordenadas cartesianas, onde cada cor no sistema é representada por um único ponto, conforme exemplifica a Figura 2.1.

Figura 2.1 – Exemplos de representações de espaços de cores (RGB, HSV, CMYK).



Fonte: Adaptado (FRANK, 2018) (SHARK, 2018) (MCMASTER, 2018).

Cada espaço de cor pode possuir diferentes vantagens e utilidades. Nesta seção serão demonstradas as principais características dos espaços de cores utilizados neste trabalho.

2.1.1 RGB

O modelo de cores RGB é um modelo de cores aditivo (POYNTON, 2003), em que o vermelho (*Red*), verde (*Green*) e azul (*Blue*) são combinados de forma a produzir novas cores, ou intensidades (Figura 2.2).

Figura 2.2 – Exemplo de combinação aditiva de cores.

Fonte: (COLORIZER, 2018).

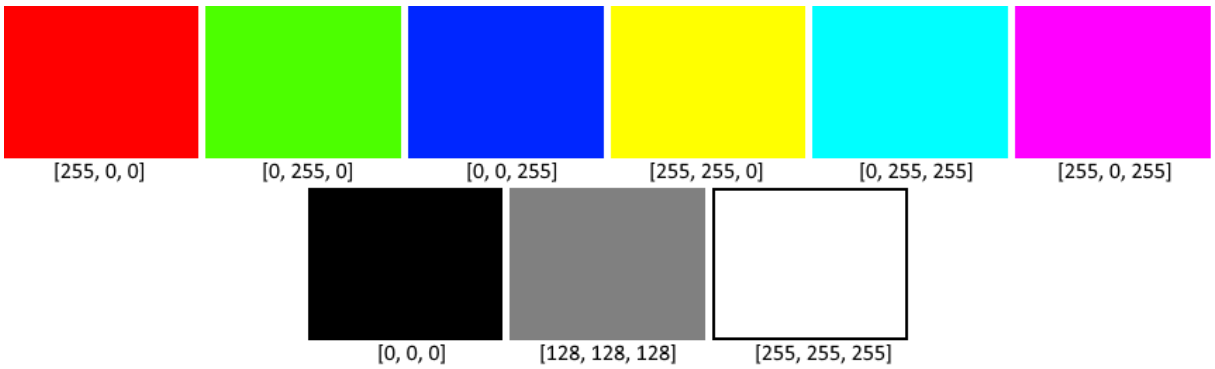
No espaço de cores RGB, um valor RGB é representado pelo uso de suas 3 cores aditivas, em diferentes níveis de intensidade. Sua estrutura básica pode ser definida da seguinte forma:

- **Vermelho (R):** faixa de intensidade entre 0 e 255;
- **Verde (G):** faixa de intensidade entre 0 e 255;
- **Azul (B):** faixa de intensidade entre 0 e 255.

Cada faixa de valores indica a intensidade (ou brilho) da cor representada. A combinação aditiva destes três componentes é utilizada para a geração de novas cores, enquanto que se todos os três valores se mantêm idênticos, cria-se diferentes tons de cinza, variando desde o mais escuro (preto) até o nível mais intenso de brilho (branco), conforme exemplificado na Figura 2.3.

As Figuras 2.4 e 2.5 apresentam os diferentes canais extraídos de uma imagem RGB.

Figura 2.3 – Exemplo de cores e suas representações em valores RGB.



Fonte: Próprio autor.

Figura 2.4 – Canais RGB exibidos isoladamente.

Fonte: Próprio autor.

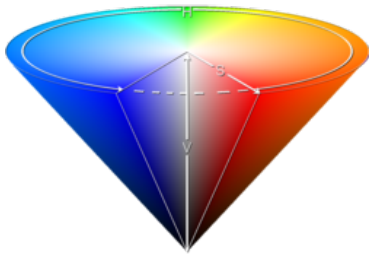
Figura 2.5 – Canais RGB convertidos para escala de cinza.

Fonte: Próprio autor.

2.1.2 HSV

O HSV (*Hue*, *Saturation* e *Value*) (Figura 2.6) é um espaço de cores criado como alternativa ao RGB, com o intuito de melhor se alinhar com o modo em que os humanos percebem as cores (JOBLOVE; GREENBERG, 1978).

Figura 2.6 – Cone de representação dos atributos do espaço de cores HSV.



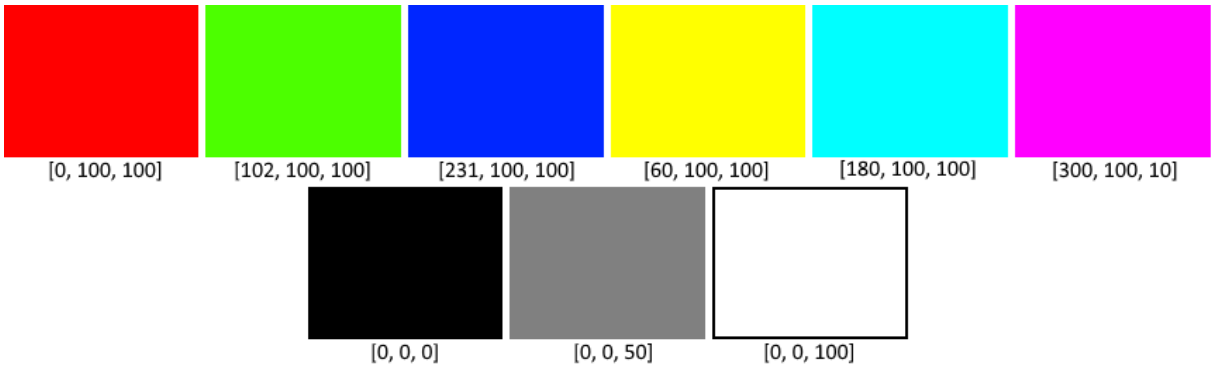
Fonte: (COLORIZER, 2018).

As cores HSV são compostas pelos 3 atributos:

- **Matiz (H):** faixa cilíndrica com valores entre 0 e 360, representam as diversas tonalidades do espectro de cores visível;
- **Saturação (S):** faixa entre 0% e 100%, determina a pureza ou profundidade da cor. (0% representando a cor branca);
- **Valor (V):** ou brilho, possui faixa de intensidade entre 0% e 100%, e representa o brilho da cor (0% representando a cor preta).

A Figura 2.7 exemplifica as diferentes cores e seus respectivos valores HSV. As Figuras 2.8, 2.9 e 2.10 os efeitos que a manipulação dos diferentes canais causam na imagem.

Figura 2.7 – Exemplo de cores e suas representações em valores HSV.



Fonte: Próprio autor.

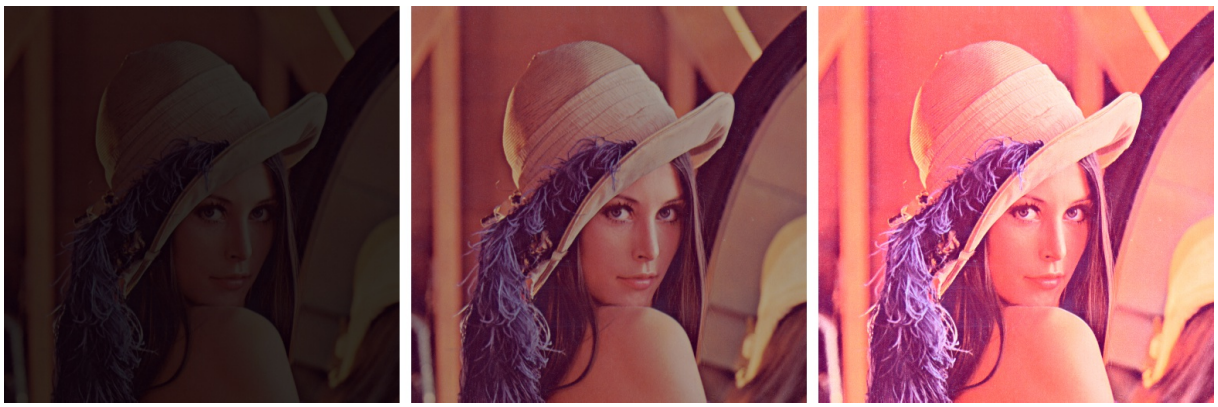
Figura 2.8 – Alteração do canal de matiz em +10, +25 e +95 respectivamente.

Fonte: Próprio autor.

Figura 2.9 – Alteração do canal de saturação em -100%, -30% e +50% respectivamente.

Fonte: Próprio autor.

Figura 2.10 – Alteração do canal de valor/brilho em -80%, -30% e +50% respectivamente.

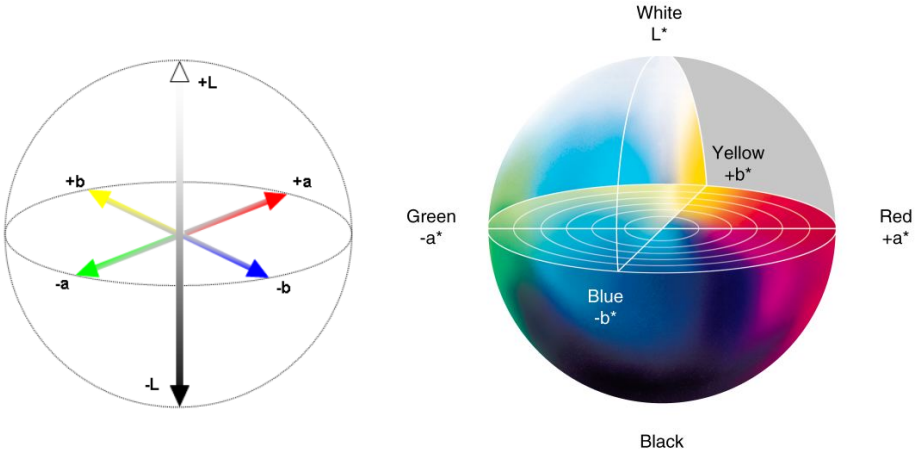


Fonte: Próprio autor.

2.1.3 CIE L*a*b

CIE L*a*b*, mais conhecido como Lab, é um espaço de cores definido pelo *International Color Consortium* (CIE) (MCLAREN, 1976), baseado nos canais de luminância (L), e 2 canais de cores opostas (A e B) (Figura 2.11).

Figura 2.11 – Globos de representação dos componentes do espaço de cores CIE L*a*b*.

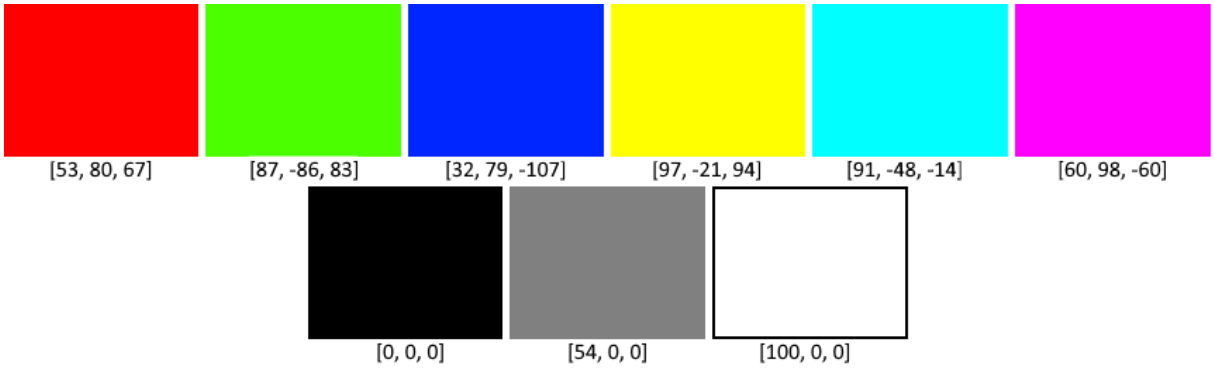


Fonte: Adaptado (GONULDAS; YILMAZ; OZTURK, 2014) (MINOLTA, 2003).

- **Luminância (L):** possui uma faixa com valores entre 0 e 100, com 0 representando o preto, e 100 representando o branco;
- **Canal A (A):** canal de cores com faixa de -128 a +128 (+a indica vermelho e –a indica verde);
- **Canal B (B):** canal de cores com faixa de -128 a +128 (+b indica amarelo e –b indica azul).

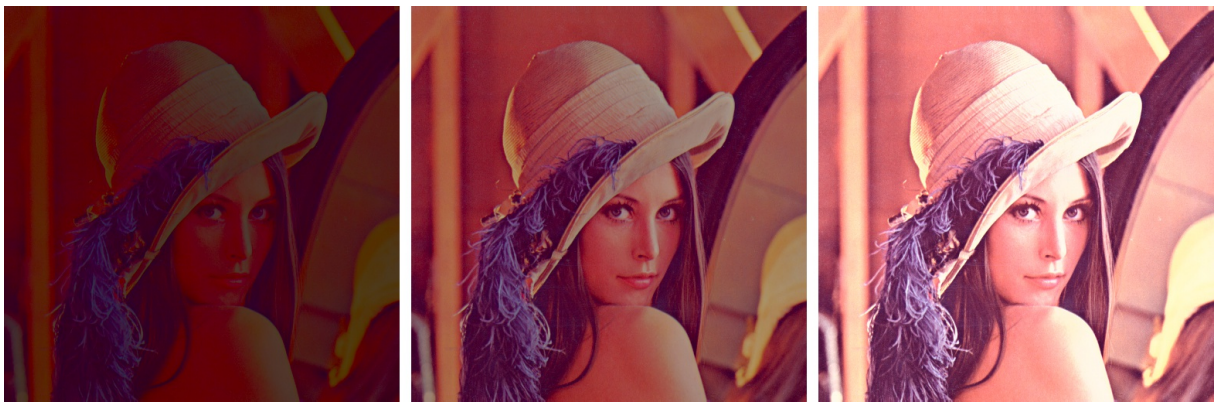
A Figura 2.12 exemplifica as diferentes cores e seus respectivos valores Lab. As Figuras 2.13, 2.14 e 2.15 os efeitos que a manipulação dos diferentes canais causam na imagem.

Figura 2.12 – Exemplo de cores e suas representações em valores Lab.



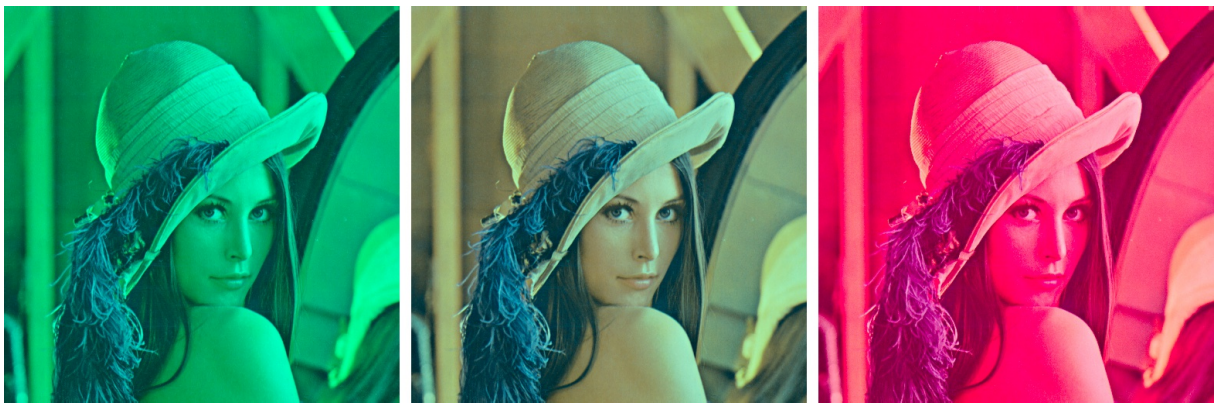
Fonte: Próprio autor.

Figura 2.13 – Alteração do canal de luminosidade em -70%, -30% e +30% respectivamente.



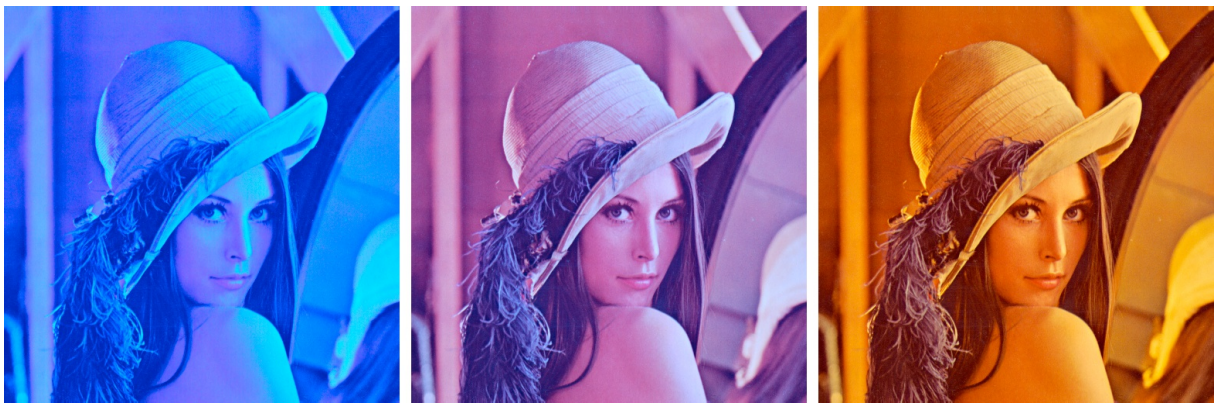
Fonte: Próprio autor.

Figura 2.14 – Alteração do canal A em -70%, -30% e +30% respectivamente.



Fonte: Próprio autor.

Figura 2.15 – Alteração do canal B em -70%, -30% e +30% respectivamente.



Fonte: Próprio autor.

2.1.4 Cálculos para distância e similaridade de cores

Um dos objetivos deste trabalho é o comparativo de similaridade de cores entre uma cor detectada e a cor alvo.

Existem técnicas implementadas de formas simples, e pouco precisas, como o cálculo de distância euclidiana para os espaços de cores RGB (Equação 2.1), HSV (Equação 2.2) e Lab (Equação 2.3, também conhecida como CIE76), em que o resultado apresenta um valor de distância entre ambas as cores comparadas. Quanto mais alto o valor, menos similaridades elas possuem.

$$rgbDistance = \sqrt{(R_2 - R_1)^2 + (G_2 - G_1)^2 + (B_2 - B_1)^2} \tag{2.1}$$

$$hsvDistance = \sqrt{(H_2 - H_1)^2 + (S_2 - S_1)^2 + (V_2 - V_1)^2} \tag{2.2}$$

$$labDistance = \sqrt{(L_2 - L_1)^2 + (a_2 - a_1)^2 + (b_2 - b_1)^2} \tag{2.3}$$

Atualmente, a fórmula que apresenta melhor precisão é a equação CIEDE2000 (SHARMA; WU; DALAL, 2005) desenvolvida para o espaço de cores Lab . Outras fórmulas existem, como CIE76 (MCLAREN, 1976) e CIE94 (MOKRZYCKI; TATOL, 2011) (REINHARD et al., 2008), mas que possuem menor precisão.

A fórmula CIEDE2000 é dada pela Equação 2.4 (SHARMA; WU; DALAL, 2005).

$$\Delta E^*_{00} = \sqrt{\left(\frac{\Delta L'}{k_L S_L}\right)^2 + \left(\frac{\Delta C'}{k_C S_C}\right)^2 + \left(\frac{\Delta H'}{k_H S_H}\right)^2 + R_T \frac{\Delta C'}{k_C S_C}\frac{\Delta H'}{k_H S_H}}, \tag{2.4}$$

em que

$$\Delta L' = L^*_2 - L^*_1 \tag{2.5}$$

$$\bar{L} = \frac{L^*_1 + L^*_2}{2} \quad \bar{C} = \frac{C^*_1 + C^*_2}{2} \tag{2.6}$$

$$a'_1 = a^*_1 + \frac{a^*_1}{2}\left(1 - \sqrt{\frac{\bar{C}^7}{\bar{C}^7 + 25^7}}\right) \quad a'_2 = a^*_2 + \frac{a^*_2}{2}\left(1 - \sqrt{\frac{\bar{C}^7}{\bar{C}^7 + 25^7}}\right) \tag{2.7}$$

$$\bar{C}' = \frac{C'_1 + C'_2}{2} \text{ e } \Delta C' = C'_2 - C'_1 \quad \text{onde } C'_1 = \sqrt{a'^2_1 + b^{*2}_1} \quad C'_2 = \sqrt{a'^2_2 + b^{*2}_2} \tag{2.8}$$

$$h'_1 = \text{atan2}(b^*_1, a'_1) \mod 360^\circ, \quad h'_2 = \text{atan2}(b^*_2, a'_2) \mod 360^\circ \tag{2.9}$$

$$\Delta h' = \begin{cases} h'_2 - h'_1 & |h'_1 - h'_2| \leq 180^\circ \\ h'_2 - h'_1 + 360^\circ & |h'_1 - h'_2| > 180^\circ, h'_2 \leq h'_1 \\ h'_2 - h'_1 - 360^\circ & |h'_1 - h'_2| > 180^\circ, h'_2 > h'_1 \end{cases} \tag{2.10}$$

$$\Delta H' = 2\sqrt{C'_1 C'_2}\sin(\Delta h'/2), \quad \bar{H}' = \begin{cases} (h'_1 + h'_2)/2 & |h'_1 - h'_2| \leq 180^\circ \\ (h'_1 + h'_2 + 360^\circ)/2 & |h'_1 - h'_2| > 180^\circ, h'_1 + h'_2 < 360^\circ \\ (h'_1 + h'_2 - 360^\circ)/2 & |h'_1 - h'_2| > 180^\circ, h'_1 + h'_2 \geq 360^\circ \end{cases} \tag{2.11}$$

$$T = 1 - 0.17\cos(\bar{H}' - 30^\circ) + 0.24\cos(2\bar{H}') + 0.32\cos(3\bar{H}' + 6^\circ) - 0.20\cos(4\bar{H}' - 63^\circ) \tag{2.12}$$

$$S_L = 1 + \frac{0.015\left(\bar{L} - 50\right)^2}{\sqrt{20 + \left(\bar{L} - 50\right)^2}} \tag{2.13}$$

$$S_C = 1 + 0.045\bar{C}' \tag{2.14}$$

$$S_H = 1 + 0.015\bar{C}'T \tag{2.15}$$

$$R_T = -2\sqrt{\frac{\bar{C}'^7}{\bar{C}'^7 + 25^7}}\sin\left[60^\circ \cdot \exp\left(-\left[\frac{\bar{H}' - 275^\circ}{25^\circ}\right]^2\right)\right] \tag{2.16}$$

2.2 OPERAÇÕES DE PROCESSAMENTO DE IMAGENS

2.2.1 Binarização

Binarização é o processo de converter uma imagem baseada em *pixels* (RGB e outros espaços de cores) em uma imagem binária, cujos valores indicam 0 ou 1, preto ou branco.

A binarização geralmente tem o propósito de segmentar as imagens, isto é, dividir ou mudar a representação das imagens, de forma a simplificar sua análise.

Alguns de seus usos, incluem a utilização de imagens binárias como pré-processamento, afim de preparar as imagens para processos posteriores como detecção de contornos (SUZUKI; BE, 1985), ou de bordas (CANNY, 1986).

Outros algoritmos podem utilizar múltiplas imagens binárias, geradas a partir de diferentes valores de limiarização (Figuras 2.16 e 2.17), como entrada para processamento.

Figura 2.16 – Exemplo de binarização em diferentes níveis de *threshold*.

Fonte: Próprio autor.

Figura 2.17 – Exemplo de binarização em diferentes níveis de *threshold*.

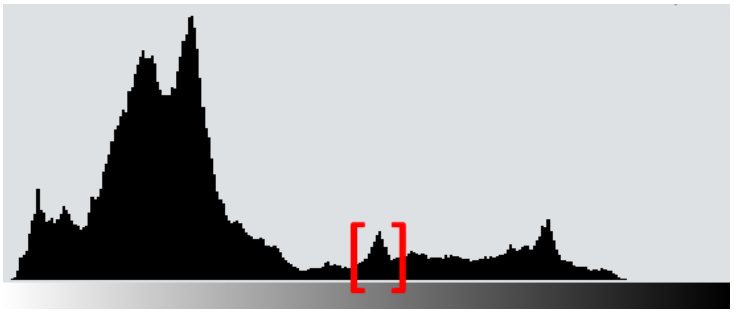
Fonte: Próprio autor.

Outras técnicas envolvem o processo de binarização e limiarização, através da análise de histogramas de imagens, para a filtragem objetos específicos de interesse, conforme Figuras 2.18, 2.19 e 2.20.

Figura 2.18 – Imagem original (esquerda). Imagem convertida para escala de cinza (direita).

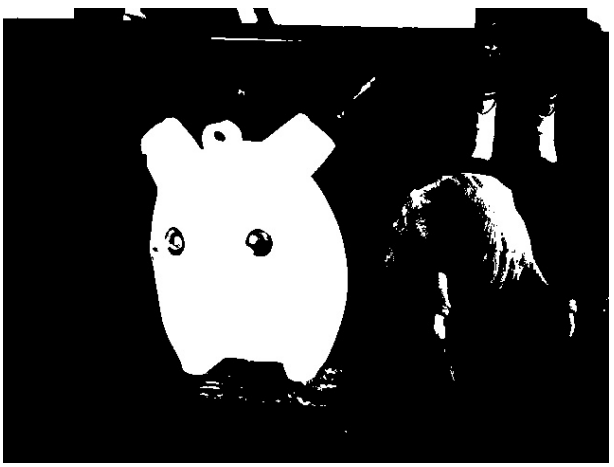
Fonte: Próprio autor.

Figura 2.19 – Análise de histograma da imagem em escala de cinza, e limiarização da área de interesse.



Fonte: Próprio autor.

Figura 2.20 – Imagem binária com a área de interesse filtrada com base nos picos detectados do histograma.



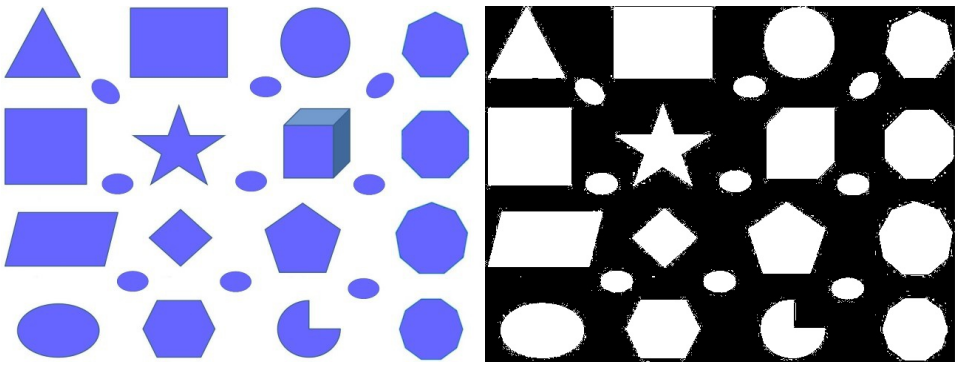
Fonte: Próprio autor.

2.2.2 Detecção de contornos

Algoritmos de detecção de contornos geralmente possuem uma imagem binária como entrada, a qual são aplicadas técnicas para percorrer e detectar linhas presentes nesta imagem. Nos exemplos, foram utilizados o algoritmo proposto por Suzuki-Abe (SUZUKI; BE, 1985) para a detecção de contornos.

Através da detecção de contornos, é possível a realização de diversos tipos de filtros, como por tamanho dos contornos, ou por formas geométricas.

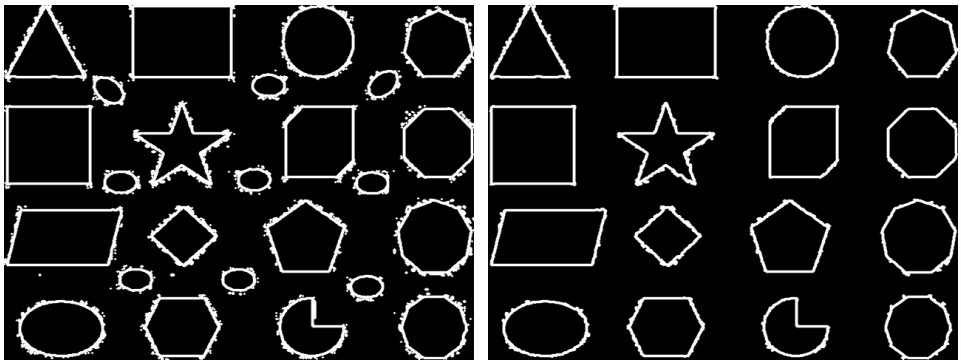
Figura 2.21 – Binarização da imagem alvo.



Fonte: Próprio autor.

Após a binarização de uma imagem (Figura 2.21), é possível processa-la através do algoritmo de detecção de contornos. O algoritmo retorna uma lista de contornos, onde cada elemento de um contorno representa um *pixel* da imagem. É possível filtrar contornos através do tamanho de seu perímetro (quantidade de *pixels*), conforme demonstra a Figura e 2.22.

Figura 2.22 – Contornos detectados (esquerda). Contornos filtrados por perímetro (direita).

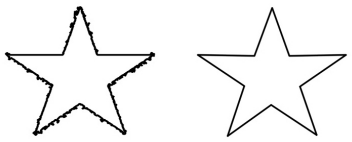


Fonte: Próprio autor.

2.2.3 Aproximação de curvas

O algoritmo Douglas-Peucker (DOUGLAS; PEUCKER, 1973), é um algoritmo de simplificação e aproximação de linhas e curvas. Funciona de forma a selecionar curvas compostas de segmentos de linhas, e encontrar uma curva similar com menos segmentos (pontos), de forma iterativa (Figura 2.23).

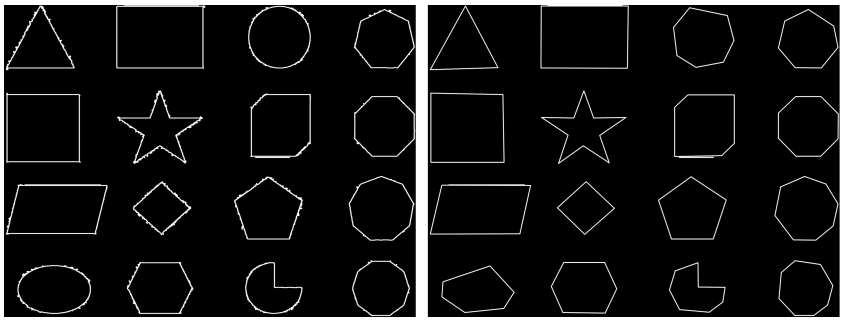
Figura 2.23 – Exemplo de aproximação de curvas utilizando o algoritmo Douglas-Peucker.



Fonte: Próprio autor.

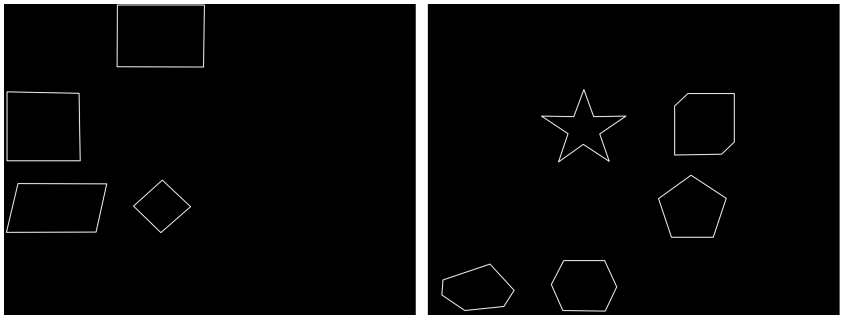
O algoritmo pode ser utilizado para reduzir o número de curvas e/ou ruídos que se formam próximos de formas geométricas que desejam ser detectadas. Tornando assim, uma maneira fácil de filtrar objetos pelo número de curvas após sua aproximação (Figuras 2.24 e 2.25).

Figura 2.24 – Contornos de entrada (esquerda). Contornos aproximados (direita).



Fonte: Próprio autor.

Figura 2.25 – Filtragem de contornos por número de curvas. Esquerda, 4 lados. Direita, 4, 5 e 10 lados.



Fonte: Próprio autor.

2.2.4 Rasterização

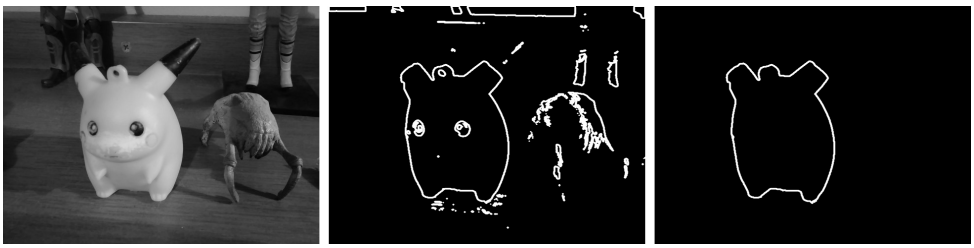
A rasterização é o processo de remover uma imagem descrita em um formato de gráficos vetoriais e convertê-la em uma imagem rasterizada, como imagens RGB.

Figura 2.26 – Exemplo de rasterização.

Fonte: Próprio autor.

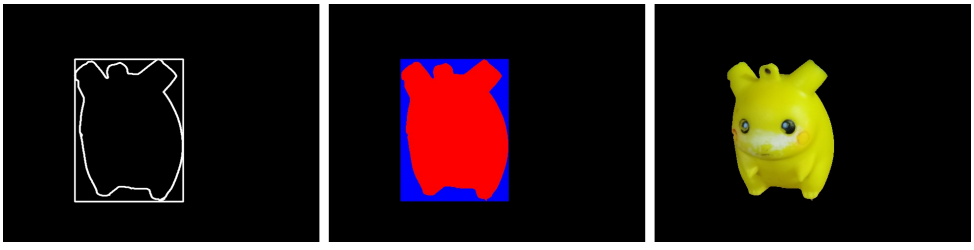
Neste trabalho, a rasterização é utilizada afim de processar elementos de uma imagem de forma isolada (Figura 2.26), baseado no algoritmo Bresenham (BRESENHAM, 1965). O algoritmo funciona de forma a selecionar um contorno (vértices) da área a ser rasterizada, cria-se uma caixa delimitadora sobre a área, e então, itera-se sobre todos os *pixels* presentes na caixa delimitadora, selecionando os que estejam contidos na área desejada. As Figuras 2.27 e 2.28 demonstram o processo de detecção, filtragem e rasterização de um objeto específico.

Figura 2.27 – Seleção da área a ser rasterizada.



Fonte: Próprio autor.

Figura 2.28 – Caixa delimitadora (esquerda). *Pixels* iterados (centro). Área rasterizada (direita).

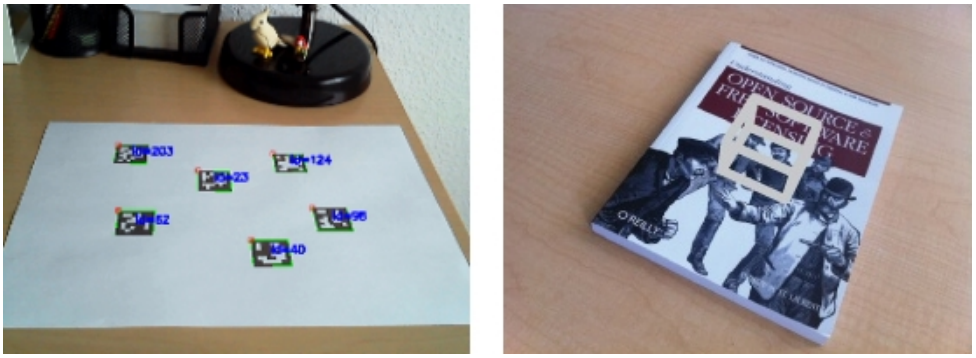


Fonte: Próprio autor.

3 TRABALHOS RELACIONADOS

Na literatura existem dois principais tipos de implementação de algoritmos para realidade aumentada: algoritmos baseados em marcadores fiduciais e algoritmos sem marcadores, exemplificados na Figura 3.1.

Figura 3.1 – Detecção de marcadores binários (esquerda). Detecção por características (direita).



Fonte: (GARRIDO, 2014) (LIN, 2018).

Aplicações de realidade aumentada baseadas em marcadores, geralmente utilizam formatos e padrões previamente conhecidos (como padrões de código de barra ou *QR Code*) para reconhecimento em imagens. Estes padrões são geralmente implementados ao algoritmo de reconhecimento, facilitando a detecção dos padrões-alvo nas imagens, sendo processados de forma mais eficiente.

Implementações sem marcadores são geralmente baseadas na detecção de características de imagens, onde utiliza-se uma imagem (objeto) alvo como modelo, e armazena-se suas características. Após, detecta-se novamente as características das imagens a serem processadas, e comparam-nas com as características armazenadas.

3.1 MARCADORES FIDUCIAIS

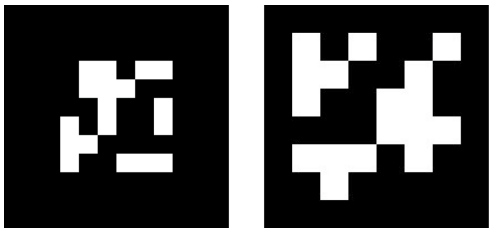
Algoritmos de detecção de marcadores fiduciais utilizam técnicas para encontrar formatos pré-definidos em uma dada imagem. O conhecimento destes formatos pelo algoritmo geralmente resulta em uma melhor performance na detecção dos objetos a serem reconhecidos.

Marcadores fiduciais fornecem dados codificados através de suas diferentes estruturas. Estes dados podem conter informações como a orientação, posição e escala do marcador, seu identificador único e redundância de dados para correção de erros em sua

detecção.

Os tipos de marcadores fiduciais mais utilizados são os marcadores binários. Marcadores binários utilizam *grids* de tamanhos variados, compostos de blocos nas cores preto ou branco, representando bits de informações a serem decodificadas (Figura 3.2).

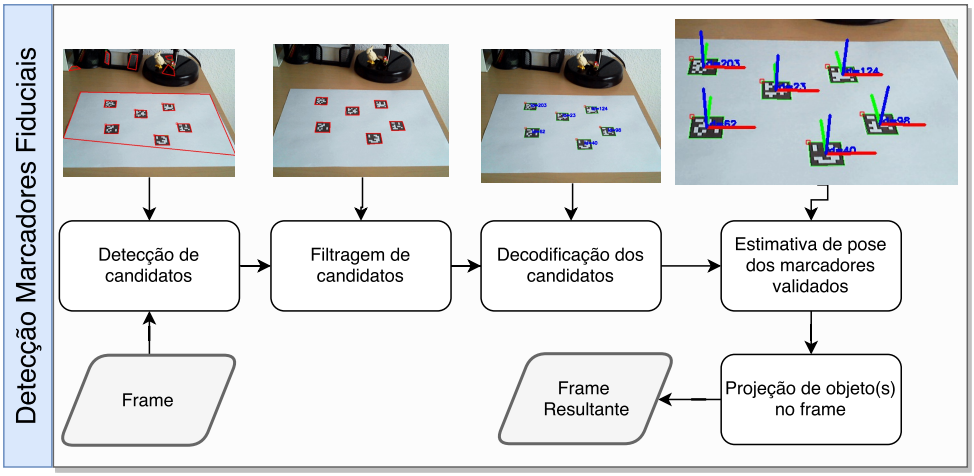
Figura 3.2 – Exemplo de marcadores fiduciais binários.



Fonte: (GARRIDO, 2014).

Marcadores fiduciais podem ser detectados de forma rápida pela combinação de algoritmos de reconhecimento genéricos, como reconhecimento de quadrados, bordas ou outras características específicas do marcador (Figura 3.3). Os *layouts* destes marcadores são criados afim de criar uma maior distinção entre o marcador e o ambiente, facilitando sua detecção através da filtragem de formatos específicos e grandes contrastes.

Figura 3.3 – Diagrama de fluxo para detecção de marcadores fiduciais binários.

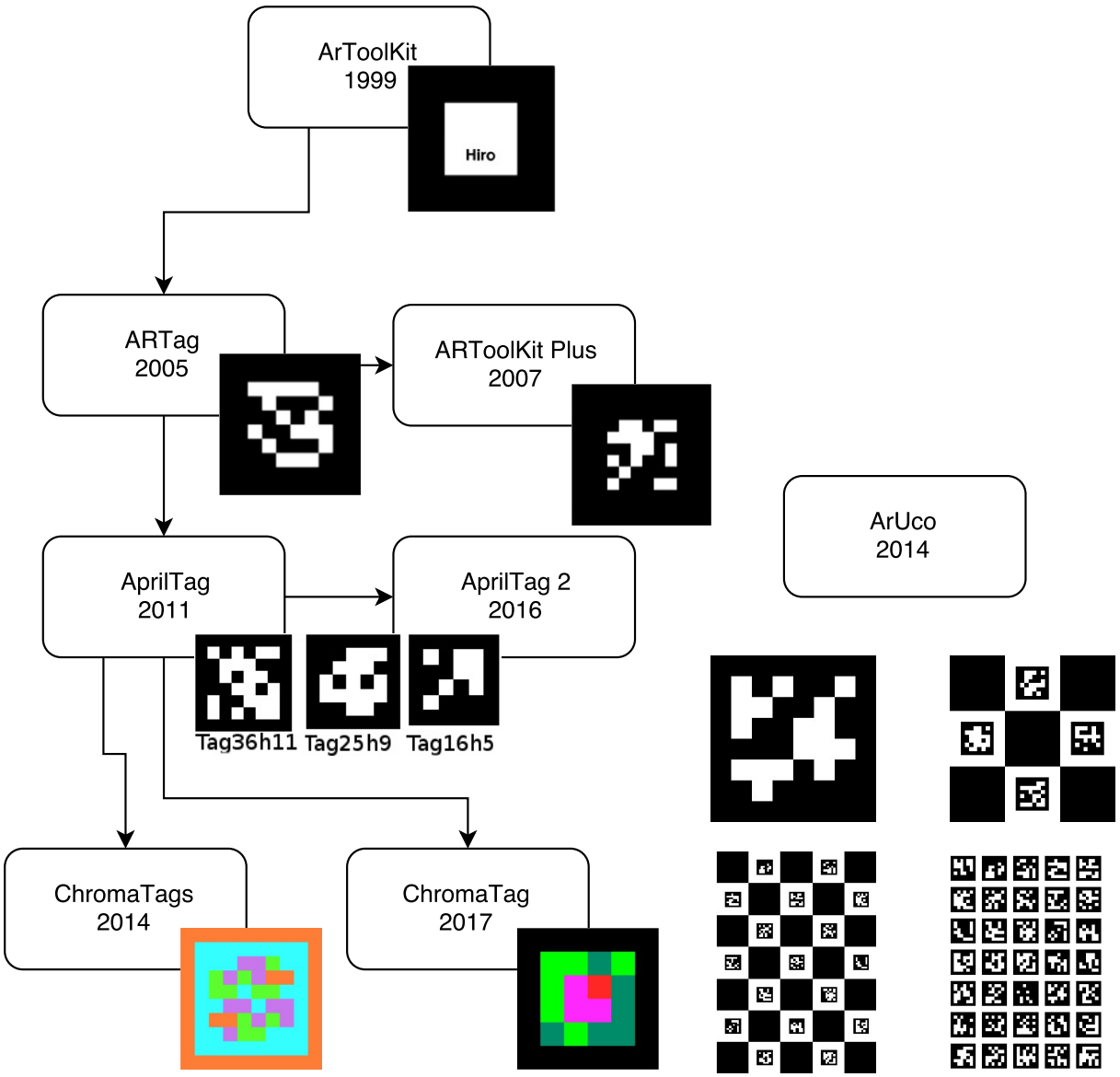


Fonte: Adaptado (GARRIDO, 2014).

3.1.1 Principais marcadores fiduciais

A Figura 3.4 apresenta os principais marcadores binários encontrados na literatura, assim como seus *layouts* e influências.

Figura 3.4 – Listagem e evolução dos principais marcadores fiduciais.



Fonte: Adaptado de ARToolKit (KATO; BILLINGHURST; POUPYREV, 2000), ARTag (FIALA, 2005), ARToolKit Plus (WAGNER; SCHMALSTIEG, 2007), AprilTag (OLSON, 2011), AprilTag 2 (WANG, 2016), ChromaTags (WALTERS, 2015), ChromaTag (DEGOL; BRETL; HOIEM, 2017) e ArUco (GARRIDO-JURADO et al., 2014) (GARRIDO, 2014).

ARToolKit: um dos primeiros, e mais utilizado sistema de marcadores, possui seus marcadores gerados com base em imagens de entrada providas pelo usuário. O sistema inicialmente utiliza técnicas para detecção de bordas afim de determinar candidatos iniciais, e após, algoritmos para correlação de imagens, para correspondência através de *templates* e padrões detectados pelas imagens de entrada.

ARTag: um dos primeiros a implementar teoria de codificação digital (codificação binária) como base para seu marcador, é uma evolução natural ao ARToolKit em termos de performance de detecção. Sua detecção é baseada em detecção de bordas, e o design de

seu marcador é composto por *grids* de tamanho 6x6, formados por blocos preto e branco, representando sua codificação binária. 10 de seus 36 bits são utilizados para codificação, enquanto que seus bits remanescentes são utilizados para redundância e detecção de erros utilizando distância de *Hamming*.

A performance do ARTag motivou melhorias de performance no ARToolKit, gerando assim, a evolução de seu algoritmo, o **ARToolKit Plus**, tendo dispositivos moveis como alvo de performance, e a também utilização de marcadores fiduciais binários, assim como o ARTag.

O **AprilTag**, outro algoritmo baseado no ARTag, fornece uma detecção mais confiável através de dicionários de marcadores mais robustos, diferentes esquemas de codificação (tamanhos de marcadores), provendo uma detecção robusta sob diferentes condições de iluminação e oclusão, apesar de utilizar um número reduzido de marcadores dependendo do esquema de codificação. **AprilTag 2** oferece um novo detector, mantendo o mesmo dicionário de marcadores do AprilTag. AprilTag 2 oferece melhoras na performance através da decimação de imagens, resultando em grandes ganhos na velocidade de detecção.

ChromaTag é um marcador que utiliza a conversão de marcadores AprilTag para o espaço de cores LAB para a redução de falsos-positivos na detecção de marcadores. Apesar oferecer melhoras na detecção, o custo de conversão de imagens RGB para LAB afeta sua performance drasticamente.

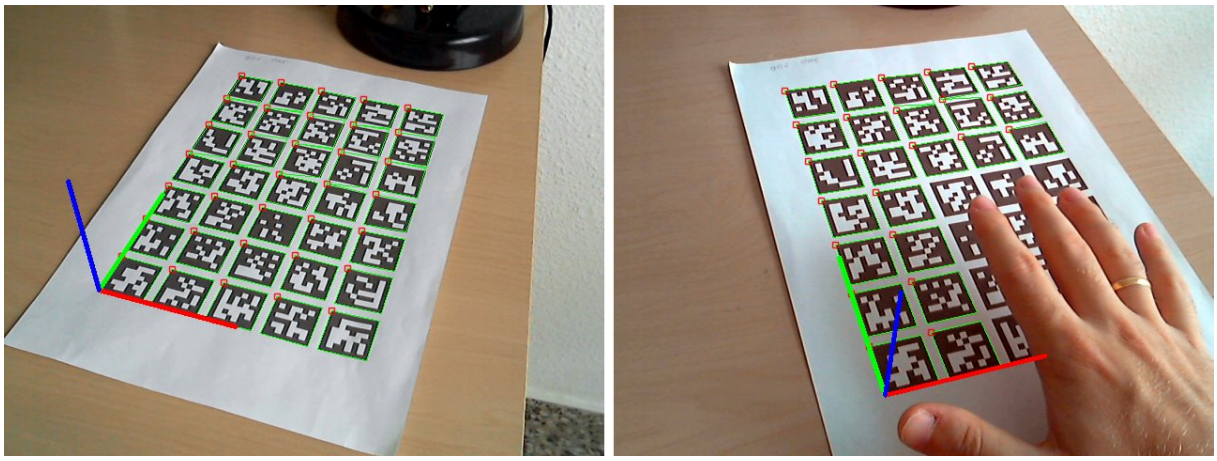
ChromaTags também baseado nos marcadores AprilTag, utiliza tons de vermelho e verde para a detecção inicial de marcadores em uma imagem. Esta técnica se mostrou eficiente devido a baixa ocorrência de contrastes verde e vermelhos na natureza, gerando baixas quantidades de falsos positivos, e candidatos nos passos iniciais de detecção. Apesar de grandes melhorias de performance, o sistema não oferece um sistema robusto para detecção parcial de marcadores, e possui dicionários com quantidades menores de marcadores em relação ao AprilTag.

ArUco (GARRIDO-JURADO et al., 2014) (GARRIDO, 2014) outro marcador binário, e base para elaboração deste trabalho, oferece novas abordagens para marcadores binários. Uma de suas vantagens é a criação customizada de dicionários de marcadores binários, conforme o nível de bits, e redundância de dados ou espessura da borda, selecionados pelo usuário.

Apesar de o marcador não oferecer uma estrutura recursiva em seu *layout*, podem ser utilizados múltiplos marcadores agrupados em formato de tabuleiros (ArUco Boards) (Figura 3.5) para a detecção e tratamento sob oclusão parcial do marcador. Também é possível a criação de múltiplos marcadores agrupados no formato de tabuleiro de xadrez (ChArUco) ou agrupados em forma de diamante (Figura 3.6).

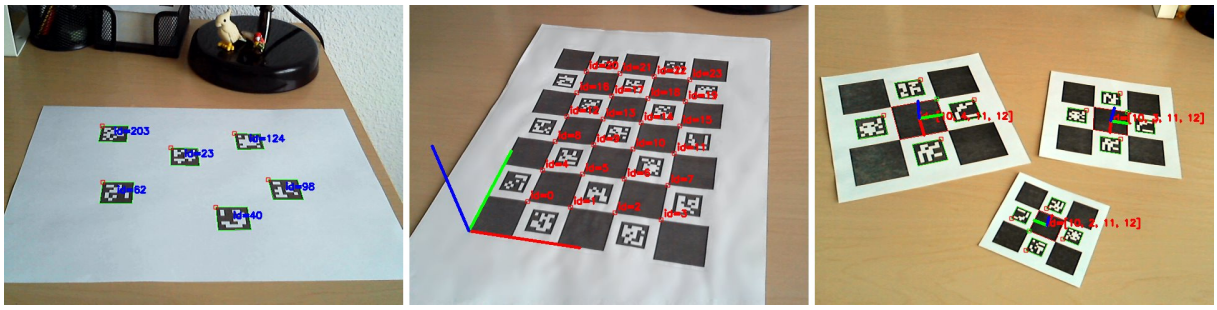
Parte do fluxo do algoritmo de detecção ArUco foi implementado neste trabalho, como estratégia de detecção de contornos e filtragens geométricas.

Figura 3.5 – Marcadores ArUco agrupados em forma de tabuleiro (esquerda). Detecção do tabuleiro como todo, sob oclusão parcial (direita).



Fonte: (GARRIDO, 2014).

Figura 3.6 – Marcadores ArUco (esquerda). ChArUco (centro). Marcadores agrupados em forma de diamante (direita).

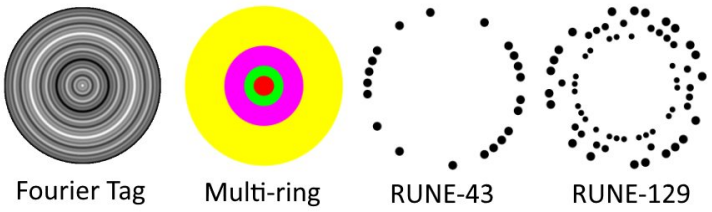


Fonte: (GARRIDO, 2014).

3.1.2 Outros marcadores fiduciais

Existem também marcadores fiduciais baseados em *layouts* circulares, como o **Fourier Tag** (SATTAR et al., 2007) (Figura 3.7), que codifica seus dados do domínio de frequências, tendo como principal objetivo, a degradação graciosa na detecção de seu marcador. Outro marcador circular é o **RUNE-tag** (BERGAMASCO et al., 2011), que possui uma estrutura baseada em pontos, organizados em forma de círculo, oferecendo detecção sob oclusão parcial, e também correção de erros. Marcadores RUNE-tag podem ser compostos de 1 (RUNE-43) ou 3 anéis (RUNE-129), cada anel contendo 43 pontos (Figura 3.7).

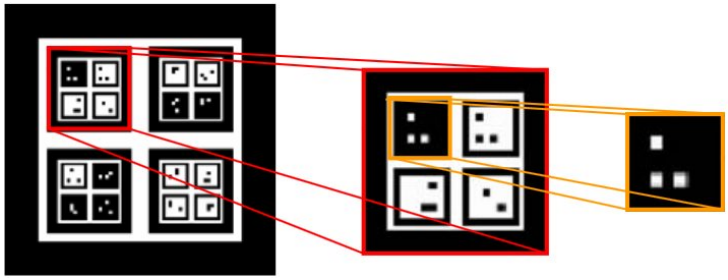
Figura 3.7 – Marcadores circulares.



Fonte: Adaptado (SATTAR et al., 2007) (CHO; NEUMANN, 2001) (BERGAMASCO et al., 2011).

Nested Marker (TATENO; KITAHARA; OHTA, 2006) é um marcador baseado em uma estrutura recursiva binária, contendo 3 níveis para detecção em diferentes distâncias (Figura 3.8). Não existem detalhes sobre a implementação do algoritmo de detecção ou quantidade de marcadores possíveis. Apesar disso, a estrutura recursiva e modelos iniciais do marcador proposto neste trabalho foram baseados no Nested Marker.

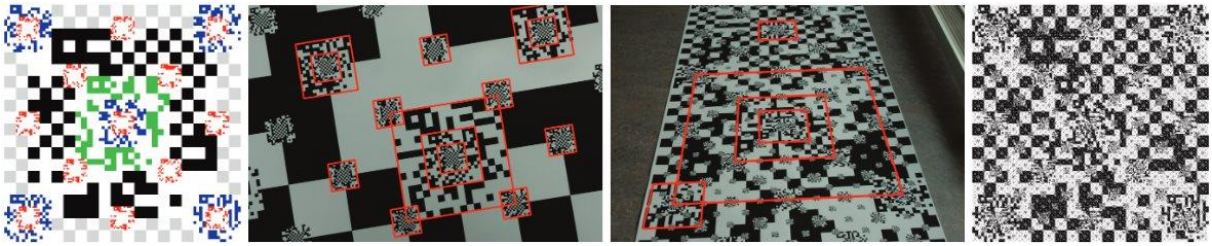
Figura 3.8 – Nested Marker e fragmentação do marcador.



Fonte: (TATENO; KITAHARA; OHTA, 2006).

Fractal Marker Field (HEROUT et al., 2012) (Figura 3.9) é um marcador de estrutura fractal, que tem como principal objetivo resolver limitações de marcadores em relação a sua escala, permitindo uma grande área de detecção devido ao tamanho possível de seus marcadores aninhados. Também foi baseado na estrutura proposta pelo Nested Marker.

Figura 3.9 – Fractal Marker Field.



Fonte: (HEROUT et al., 2012).

3.2 CONSIDERAÇÕES

É possível perceber que existem diversos estudos propondo diferentes tipos e formatos de marcadores. Apesar de existir marcadores baseados em estruturas hierárquicas ou recursivas, o número de estudos ainda é limitado.

Também existem poucos trabalhos utilizando marcadores coloridos, mais especificamente, na implementação da estrutura de seu marcador. Ambos os marcadores ChromaTag (DEGOL; BRETL; HOIEM, 2017) e ChromaTags (WALTERS, 2015), apesar de utilizarem cores na sua detecção, suas codificações ainda são binárias. O marcador Multiring (CHO; NEUMANN, 2001) apesar de abordar uma estrutura hierarquia e utilizar cores em sua estrutura, oferece um número muito baixo de marcadores.

4 MARCADOR PROPOSTO

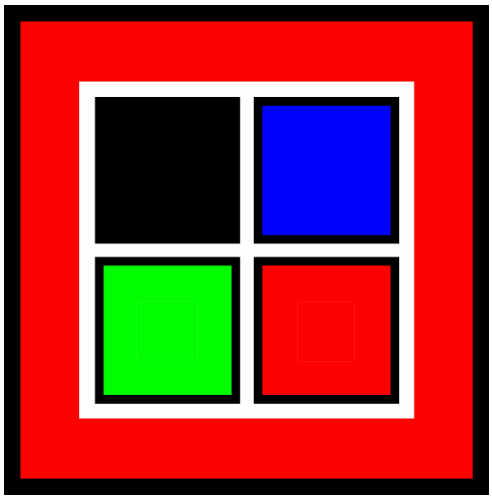
O design do marcador CRFM tem como principais características sua estrutura hierárquica e recursiva. Esta estrutura permite uma detecção otimizada sob diferentes distâncias da câmera, detecção sob situações de oclusão parcial do marcador. Além disso, a utilização de cores fornece um maior número de marcadores distintos para a geração de dicionários.

Este capítulo tem como objetivo apresentar em maiores detalhes o design do marcador proposto pelo algoritmo CRFM, assim como sua estrutura hierárquica, suas principais características e seu processo de geração.

4.1 DESIGN E ESTRUTURA DO MARCADOR

No algoritmo CRFM cada marcador é representado por uma estrutura hierárquica básica (hierarquia), de que cada marcador e sub-marcadores são compostos. Cada hierarquia é composta por cinco blocos coloridos, sendo um bloco principal (o mais externo), e quatro blocos internos, conforme Figura 4.1.

Figura 4.1 – Exemplo de estrutura básica do marcador.



Fonte: Próprio autor.

Cada bloco de um marcador possui uma determinada cor, onde a sequência das cores de um marcador é utilizada de forma a determinar um identificador único (ID) para cada marcador, devido a isto, a sequência de cores que compõe uma dada hierarquia nunca é repetida, garantindo que cada hierarquia possua seu próprio identificador único.

A sequência de cores de uma hierarquia é sempre lida a partir do bloco principal, e do sentido horário a partir do bloco interno inferior-direito, ou seja, se tomarmos como

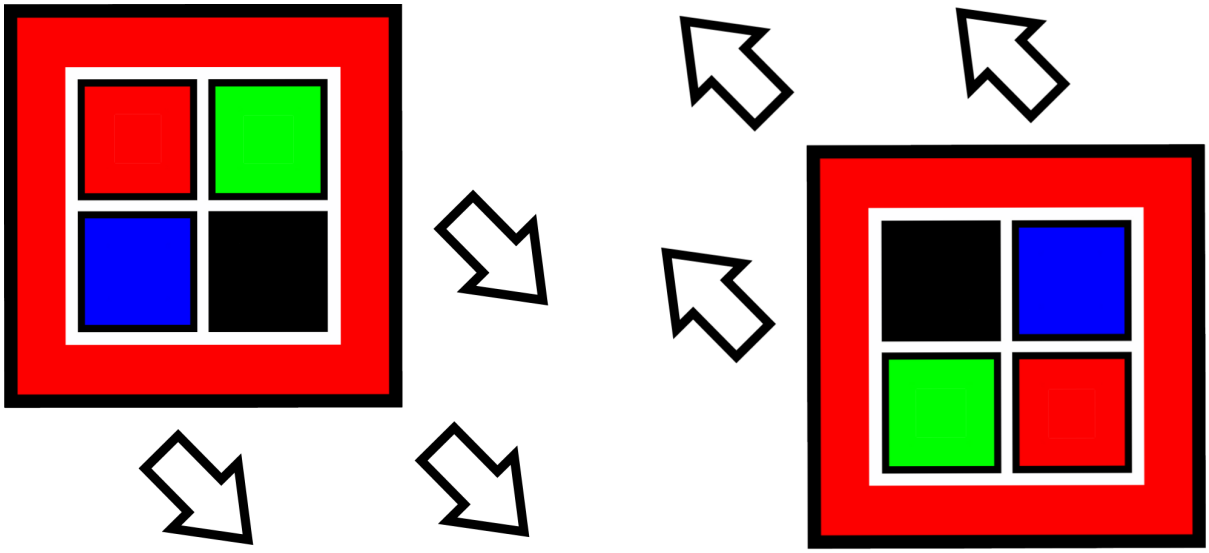
exemplo o marcador apresentado na Figura 4.1 a leitura resultante seria: vermelho, vermelho, verde, preto e azul.

Cada hierarquia pode ser composta de até seis cores: vermelho, verde, azul, amarelo, laranja e preto. Estas cores foram selecionadas seguindo o método empírico, através de testes e avaliações de histogramas de imagens de marcadores previamente impressos, e capturados através de uma câmera, a partir de diferentes ângulos e níveis de iluminação.

Em todas hierarquias, o bloco interno inferior-direito possui sempre a mesma cor que a de seu bloco principal, indicando a orientação do marcador em relação ao ambiente (Figura 4.2), informação utilizada para a projeção correta de objetos sobre o marcador.

Esta característica também gera uma limitação em que a respectiva cor não pode ser repetida em outros blocos internos, afetando o número de marcadores que podem ser gerados.

Figura 4.2 – Exemplo em que as setas representam a orientação do marcador, conforme a sequência de cores.

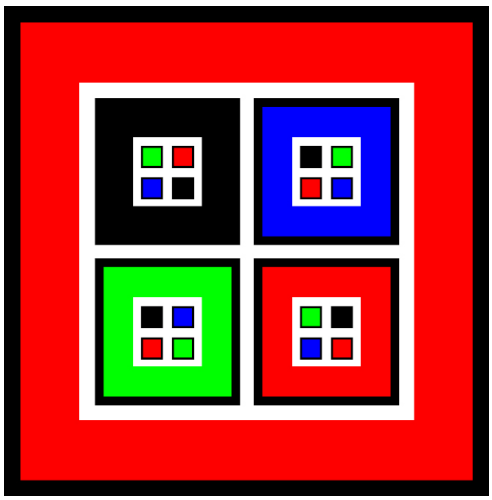


Fonte: Próprio autor.

O marcador CRFM também pode ser utilizado seguindo uma estrutura recursiva, gerando novas camadas de marcadores, onde cada um dos blocos internos de um marcador, pode conter um novo marcador (Figura 4.3). Assim, o bloco interno torna-se o bloco principal em relação ao novo sub-marcador, obedecendo as limitações prévias de cores e sequências para criação de identificadores únicos.

Esta característica pode ser repetida sucessivamente, conforme o número de camadas desejado, o número de cores utilizado para a criação de um dicionário, e o número de marcadores únicos que podem ser criados.

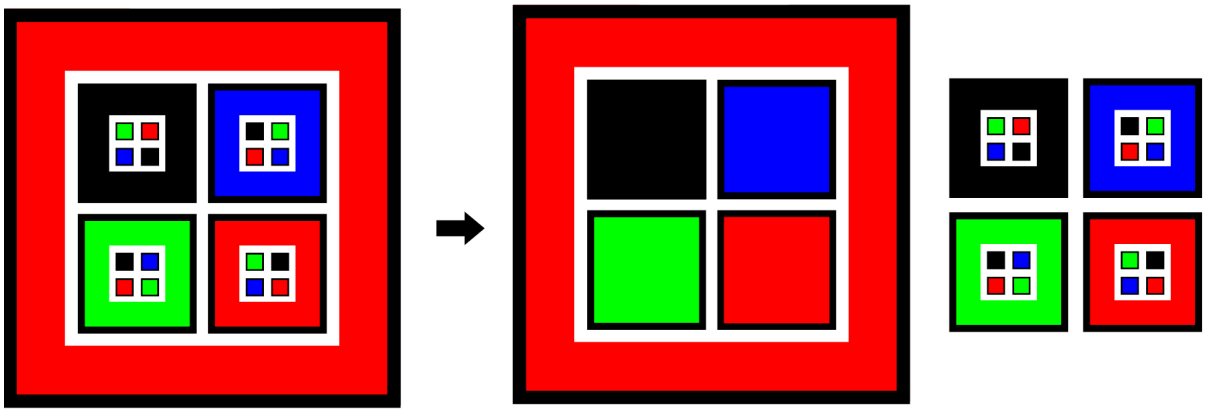
Figura 4.3 – Exemplo de estrutura recursiva contendo duas camadas.



Fonte: Próprio autor.

Figura 4.4 exemplifica como um marcador de duas camadas é decomposto em sub-marcadores, permitindo o processamento de marcadores de forma distinta.

Figura 4.4 – Exemplo de decomposição de um marcador de duas camadas, em seus cinco respectivos sub-marcadores.



Fonte: Próprio autor.

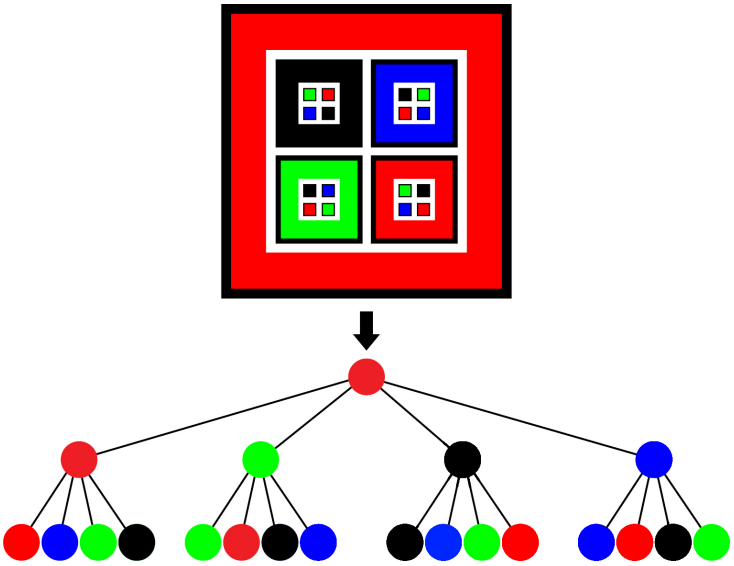
Os blocos dos marcadores CRFM também são constituídos de uma borda preta, e de um espaço em branco em seu redor. Estas características tornam-se necessárias por gerar maior contraste entre os blocos e hierarquias distintas, de forma a facilitar sua detecção conforme o algoritmo para detecção do CRFM.

Os marcadores CRFM também são gerados seguindo um padrão de proporcionalidade entre a espessura das bordas e os espaços em branco, do bloco principal e de seus blocos internos. Estas proporções também foram determinadas através do método empírico, baseado em um modelo de marcador inicial, e adaptado conforme os resultados de testes de detecção realizados.

4.2 DETECÇÃO A PARTIR DE OCLUSÃO PARCIAL

Cada marcador CRFM pode ser representado também como uma arvore n-ária, processo que facilita a visualização e manipulação de dados referentes aos marcadores detectados em uma imagem, conforme mostra a Figura 4.5.

Figura 4.5 – Exemplo de abstração do marcador em formato de arvore n-ária.

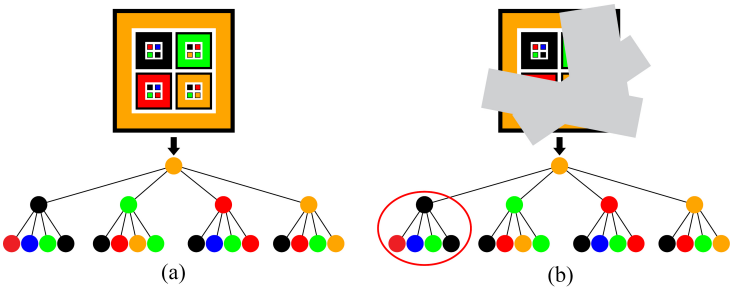


Fonte: Próprio autor.

Devido a utilização desta estrutura recursiva, é possível estimar a pose do marcador como um todo, a partir de detecção de apenas um de seus sub-marcadores. Assim, não é necessário que sejam encontrados todos os sub-marcadores em uma imagem para permitir a projeção de um objeto.

Como resultado, é possível determinar a pose do marcador a partir de situações onde ocorrem oclusões parciais do marcador, conforme exemplificado na Figura 4.6.

Figura 4.6 – Exemplo de detecção do marcador sob oclusão parcial.



Fonte: Próprio autor.

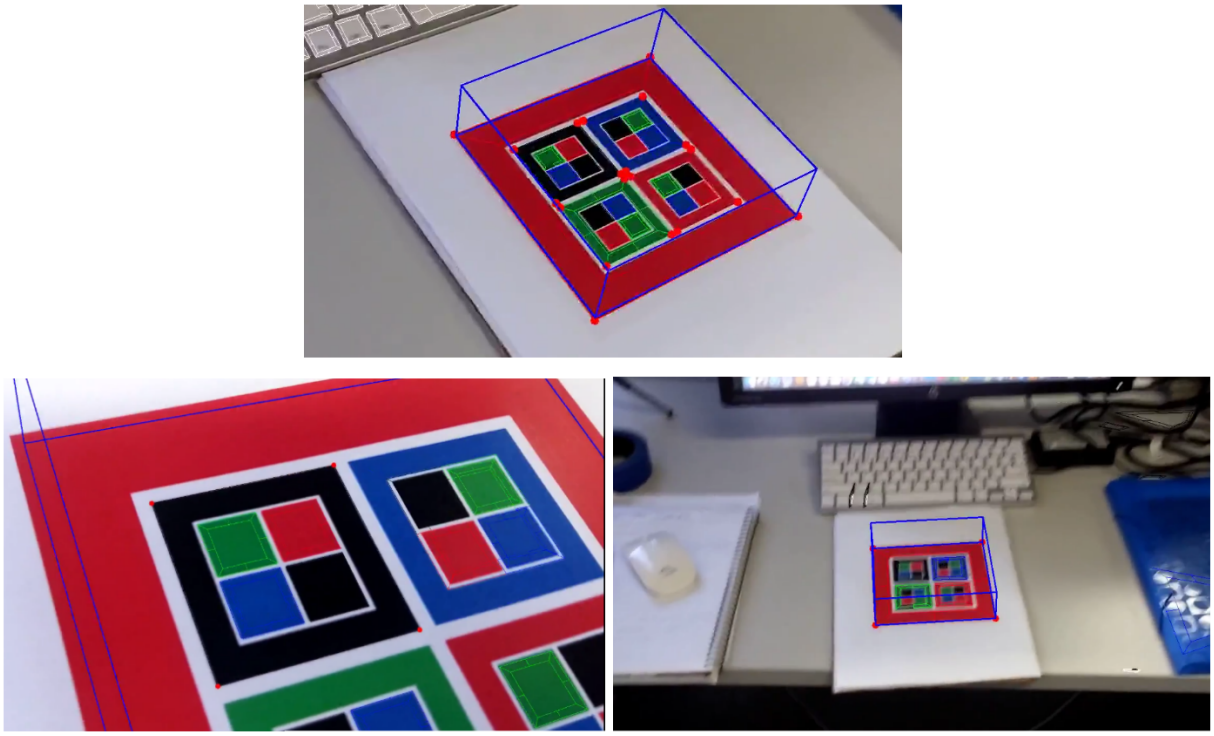
4.3 DETECÇÃO EM DIFERENTES DISTÂNCIAS DA CÂMERA

Outra característica desta estrutura recursiva, é a possibilidade de detecção de um marcador a partir de diferentes distâncias. Exemplo disto, é devido ao fato de marcadores mais externos possuírem um tamanho maior, sendo detectados mais facilmente em situações onde a câmera está a uma distância maior do marcador. A mesma ideia se aplica quando a câmera está em uma posição mais próxima, podendo causar oclusão de marcadores mais externos.

Na Figura 4.7 pode ser visto o funcionamento desta característica, onde é realizada a projeção de arestas azuis que representam um prisma quadrangular, sobre um marcador alvo de duas camadas, constituído de cinco sub-marcadores.

No primeiro caso (posição superior da figura), o prisma quadrangular é projetado conforme a estimativa de pose dos cinco sub-marcadores detectados corretamente. No segundo caso (canto inferior-esquerdo da figura) é realizada a projeção conforme a detecção do sub-marcador superior-esquerdo, o único detectado, devido a oclusão parcial por parte da proximidade da câmera. No terceiro caso (canto inferior-direito da figura), a projeção é realizada com base apenas no sub-marcador mais externo detectado.

Figura 4.7 – Exemplo de detecção em diferentes distâncias da câmera em relação ao marcador.



Fonte: Próprio autor.

4.4 GERAÇÃO DE MARCADORES

Os marcadores devem ser previamente gerados e armazenados como um dicionário antes de serem utilizados no algoritmo CRFM. O algoritmo de geração funciona de forma resumida, conforme listado:

- Gerar todas as sequências possíveis (permutações) de acordo com o número de cores selecionadas, considerando as seguintes regras:
  - O bloco-pai e o primeiro bloco interno (inferior-direito) devem ser da mesma cor
  - Próximos blocos da mesma sequência não devem repetir as cores do bloco-pai e do primeiro bloco interno
- Selecionar uma sequência válida, e remover sequências equivalentes da lista restante;
  - Cada sequência deve ser única (não repetida), então, após a sua inserção, deve-se iterar e remover sequências equivalentes da lista restante.
- Para cada um dos blocos interno da sequência, selecionar uma sequência correspondente (que começa com a mesma cor) para se tornar sua sequência interna.
- Repetir o passo anterior de acordo com o número de hierarquias exigido.

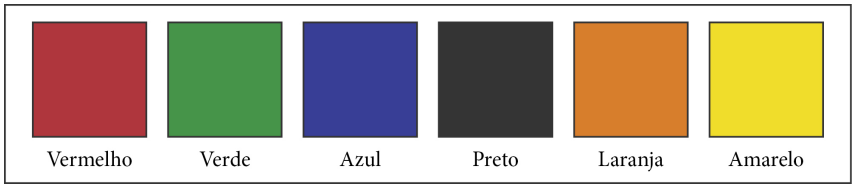
Após a geração de sequências de cores, as cores são armazenadas em vetores representando os dicionários de marcadores, e utilizados posteriormente para a impressão de marcadores, ou como referência para o marcador-alvo no processo de detecção do CRFM, método explicado em detalhes no Capítulo 5.

4.4.1 Número de marcadores

O número de possíveis marcadores a serem gerados, é afetado por cinco fatores limitantes:

- **O número de cores possíveis para um dicionário (Figura 4.8);**

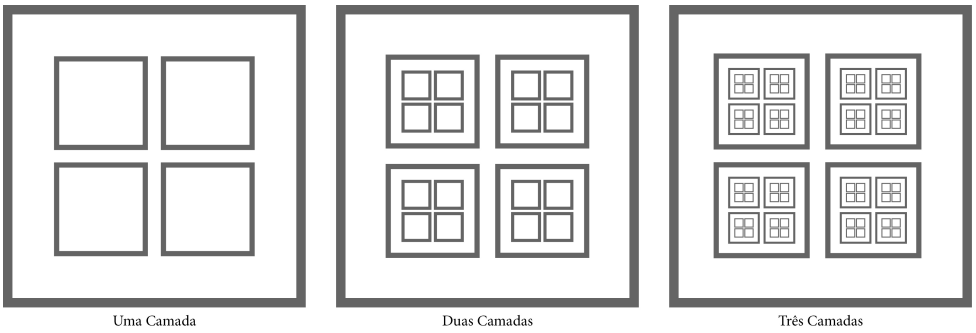
Figura 4.8 – Cores utilizadas para geração dos marcadores.



Fonte: Próprio autor.

- **O número de camadas para cada marcador de um dicionário (Figura 4.9)**;

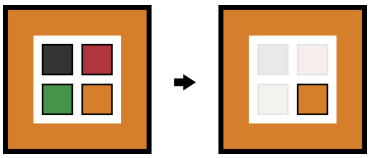
Figura 4.9 – Camadas para o marcador.



Fonte: Próprio autor.

- **O bloco interno inferior-direito de uma hierarquia deve repetir a cor do bloco principal (Figura 4.10);**

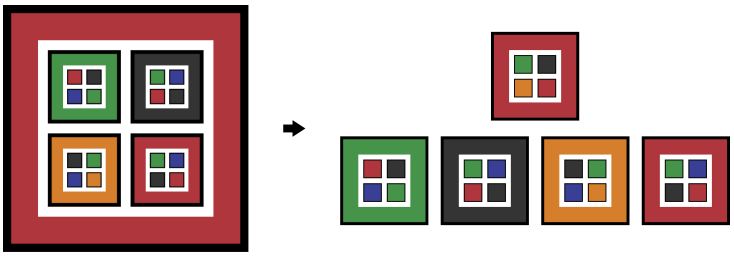
Figura 4.10 – Hierarquia interna sem repetição de sequência.



Fonte: Próprio autor.

- **Outros blocos internos (da mesma hierarquia) não podem repetir a cor do bloco principal e do inferior-direito (Figura 4.11);**

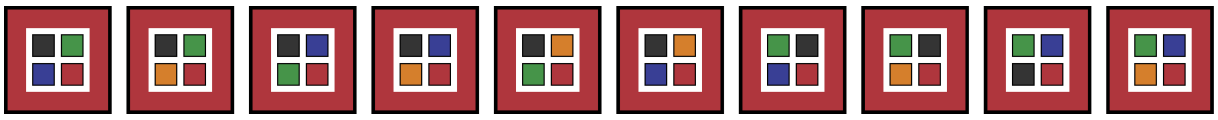
Figura 4.11 – Composição de uma hierarquia de duas camadas.



Fonte: Próprio autor.

- **As sequências de cores de uma hierarquia devem ser distintas, não podendo ser repetidas em um mesmo dicionário (Figura 4.12).**

Figura 4.12 – Exemplo de marcadores sem repetição de sequência.



Fonte: Próprio autor.

Cada marcador é caracterizado pela pelas cores que o compõe e pela ordem dos elementos internos. A composição das hierarquias de marcadores, independente da quantidade de níveis, é composta por hierarquias de um nível.

A quantidade de cores disponíveis para a geração de um dicionário, é um fator que impacta de maneira exponencial a quantidade de hierarquias possíveis. A Equação 4.1 define a geração das combinações básicas possíveis para formação de hierarquias que podem ser usadas para a criação de marcadores com quantidades maiores de níveis.

$$QtdArranjos = \frac{N!}{(N-(V*H))!}, \tag{4.1}$$

onde, *N* representa a quantidade de cores que formam o conjunto. Os valores de V e H representam os elementos verticais (*V*) e horizontais (*H*) que compõem o *grid* interno do marcador.

O Quadro 4.1 mostra a quantidade de hierarquias de um nível possíveis de acordo com a quantidade de cores. Através da análise destas informações é possível perceber que quando o número de cores é menor que 4, torna-se inviável a utilização da estrutura recursiva do CRFM, e que não seria possível a utilização deste formato de marcador em uma situação onde se utilizasse de cores binárias (preto e branco).

Quadro 4.1 – Relação do número de camadas de um marcador para o total de submarcadores necessários por camada.

NumeroCores	2	3	4	5	6	7	8
QtdArranjos	*	*	24	120	360	840	1680

Fonte: Próprio autor.

O número de camadas é um fator de grande impacto, pois este valor afeta de maneira exponencial, de acordo com a função de recorrência. A quantidade hierarquias (marcadores de um nível) exigidos para a geração de um único marcador é definido pela Equação 4.2. Considerando 5 cores disponíveis para compor a estrutura do marcador com *grid* interno de 2x2, a quantidade de hierarquias possíveis pode ser visualizada no Quadro 4.2.

$$QtdHierarquias = \sum_{i=1, q_{Ant}=1}^{i \leq L} q_{Ant},\ de\ forma\ que\ q_{Ant} = (q_{Ant} * (H * V)) + 1 \tag{4.2}$$

Quadro 4.2 – Relação do número de camadas de um marcador para o total de submarcadores necessários por camada.

NumeroCamadas	0	1	2	3	4	5	6	7
QtdHierarquias	*	1	5	21	85	341	1365	5461

Fonte: Próprio autor.

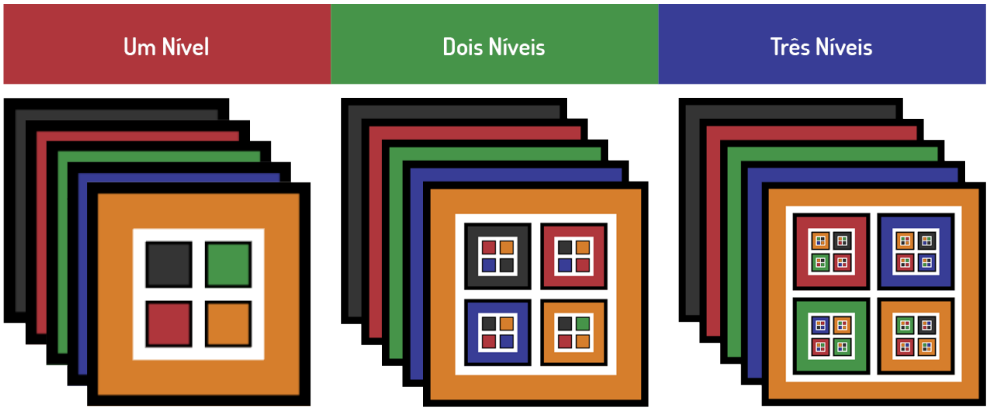
Para a composição de cada marcador de um nível, por exemplo, com tamanho de *grid* 2x2 são utilizadas quatro cores para formar os 5 elementos que compõem cada marcador. Quatro destes elementos representam o *grid* interno e um quinto elemento é usado para representar a borda, possuindo a mesma cor do elemento direcional do marcador. Desta forma, para essa configuração de marcadores, utilizando as 5 cores selecionadas, são possíveis 120 marcadores.

A Equação 4.3 onde, apresenta o cálculo para definição da quantidade de hierarquias necessárias para compor um único marcador, de acordo com a quantidade de níveis, somando os marcadores dos níveis inferiores até o nível mais externos. Desta forma, para encontrar o número de marcadores possíveis de acordo com a quantidade de níveis e tamanho do *grid*, basta aplicar a equação:

$$QtdMarcadores = \frac{QtdArranjos}{QtdHierarquias} \tag{4.3}$$

Por fim, a Figura 4.13 exibe o número final de marcadores que podem ser gerados e a quantidade de hierarquias de um nível necessárias.

Figura 4.13 – Número de marcadores que podem ser gerados.



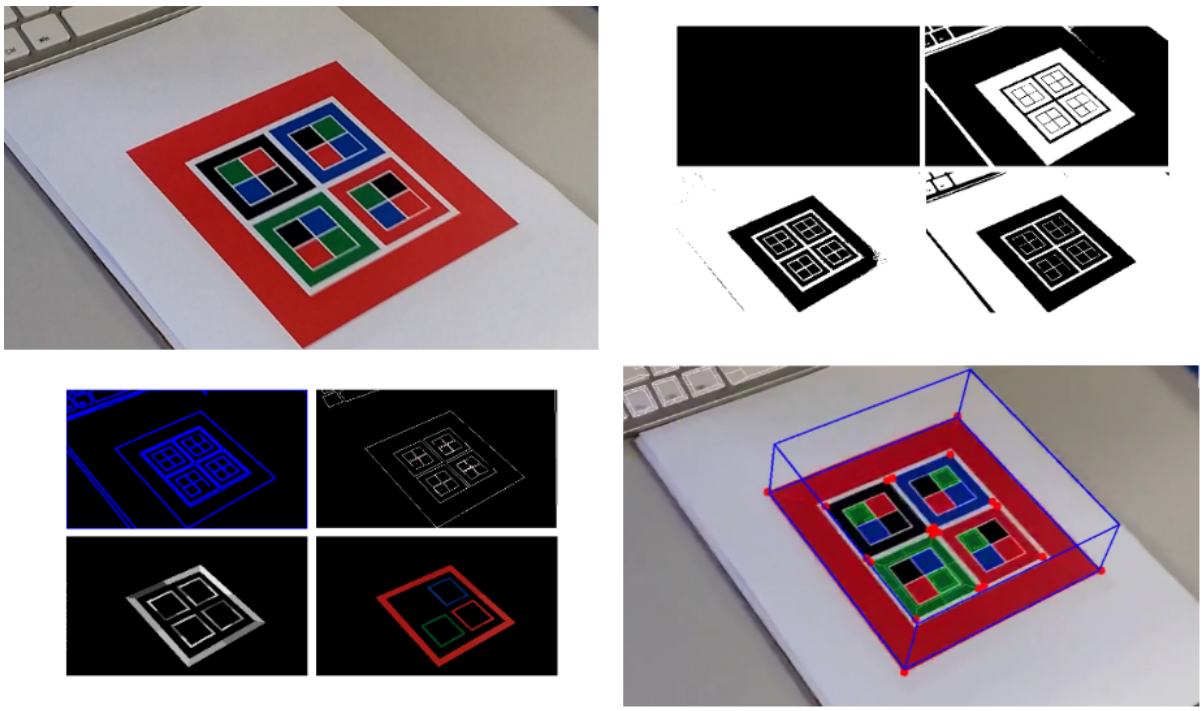
	Um Nível	Dois Níveis	Três Níveis
Quantidade de marcadores de um nível necessários	-	5 marcadores	21 marcadores
Quantidade de marcadores possíveis	120 marcadores possíveis	24 marcadores	5 marcadores

Fonte: Próprio autor.

5 ALGORITMO DE DETECÇÃO

Este capítulo busca apresentar e detalhar passo-a-passo o funcionamento e objetivo dos principais processos do algoritmo de detecção CRFM, desde a captura de um quadro de imagem, o processamento e análise desta imagem, e a exibição final com projeção, no caso de detecção conforme Figura 5.1.

Figura 5.1 – Exemplos de etapas de processamento até o quadro final resultante.



Fonte: Próprio autor.

O algoritmo tem início a partir do recebimento de uma imagem, que pode ser capturada a partir de uma *webcam*, vídeo ou arquivos de imagens. Esta imagem é então processada através de diferente funções, afim de extrair informações que possam determinar candidatos (áreas e quadriláteros presentes na imagem) de possíveis marcadores, que serão analisados de forma mais detalhada (filtragem de cores e outros sub-processos). A partir dos resultados destes processos, é possível determinar se um ou mais marcadores foram detectados de forma correta, calcular a estimativa de pose dos marcadores, e então realizar a projeção de objetos na cena (imagem) de acordo com os marcadores detectados.

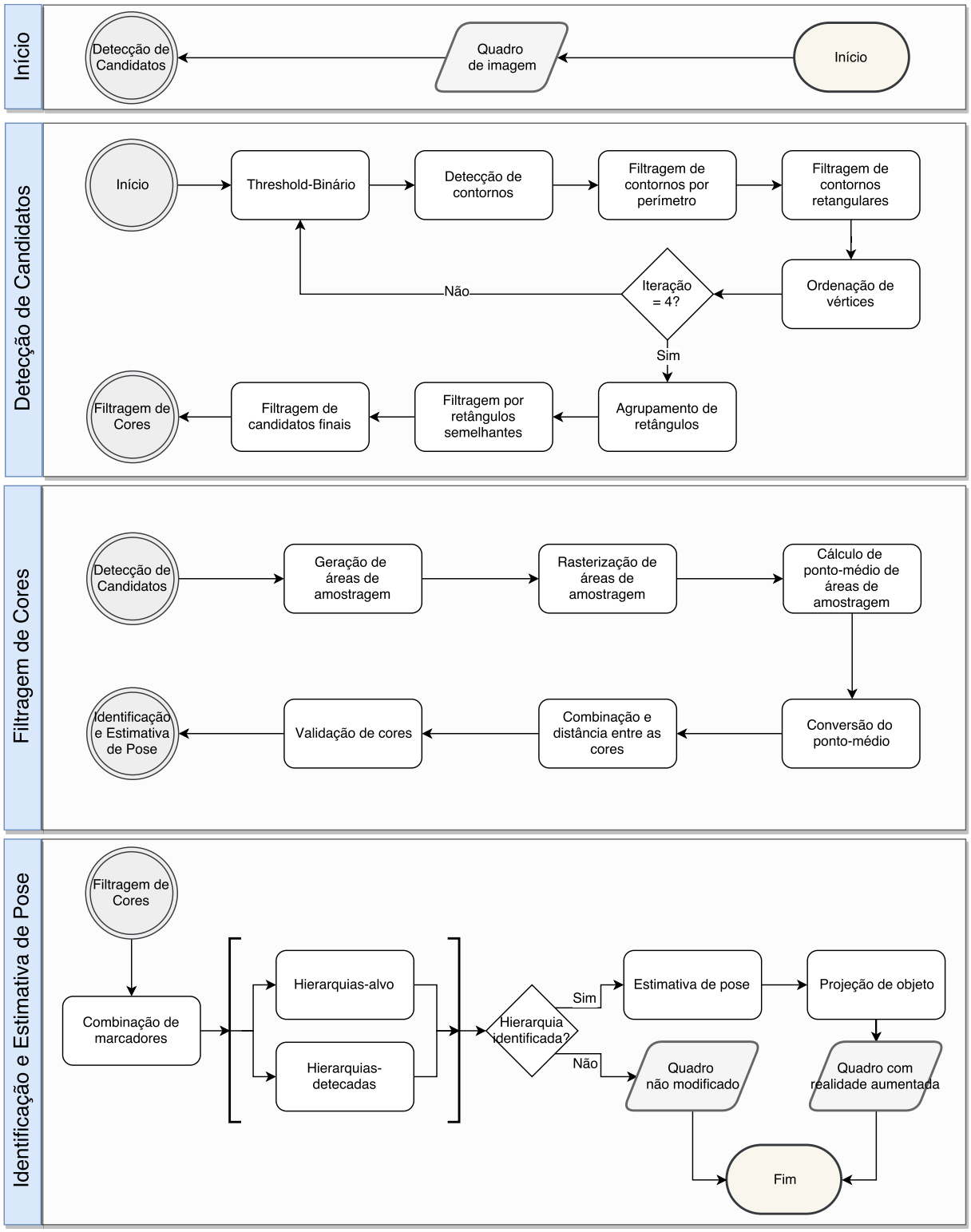
O fluxo de detecção (Figura 5.2) do algoritmo pode ser separado em três principais processos:

- Detecção de candidatos;
- Filtragem de cores;

- Identificação e estimativa de pose.

Estes processos (e seus sub-processos) serão analisados com maiores detalhes e exemplificados nas seções subsequentes deste capítulo.

Figura 5.2 – Diagrama de fluxo simplificado do algoritmo de detecção CRFM.

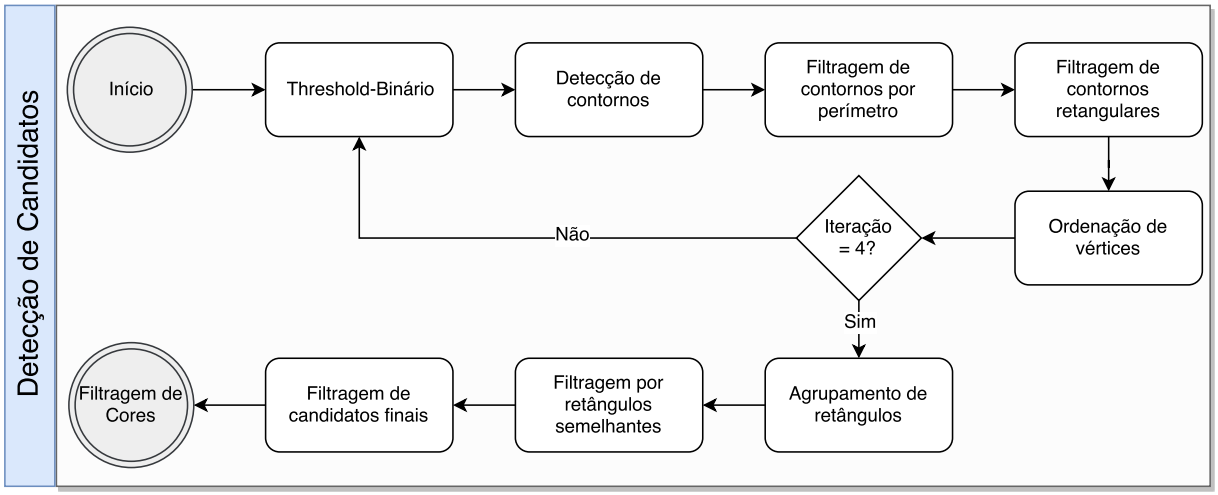


Fonte: Próprio autor.

5.1 DETECÇÃO DE CONTORNOS E CANDIDATOS

O algoritmo para detecção de candidatos consiste de uma versão brevemente modificada do algoritmo de detecção ArUco, e pode ser resumido nos seguintes passos (Figura 5.3):

Figura 5.3 – Diagrama de fluxo do processo de detecção de candidatos.



Fonte: Próprio autor.

Esta etapa consiste da detecção e filtragem de contornos da imagem, com o propósito de filtrar o maior número de contornos possíveis, de forma a manter apenas contornos em formato de quadrilaterais, e que sejam pertencentes a alguma hierarquia. Esta etapa é critica para o alvo de performance em tempo real do algoritmo, pois os contornos eliminados nesta etapa, são contornos que não serão processados na etapa seguinte do algoritmo, a etapa de filtragem de cores.

Uma das principais diferenças desta etapa para o algoritmo ArUco, é o *threshold*-binário com faixas de *threshold* customizadas, e o passo de filtragem de contornos por hierarquia, processo que reduz de maneira significativa o número de candidatos para a etapa posterior.

5.1.1 *Threshold*-binário

Este processo consiste em aplicar uma técnica de *threshold*-binário sobre a imagem sendo analisada. Este processo ocorre de forma paralelizada, e são geradas quatro imagens binárias, processadas a partir de diferentes faixas de *threshold*. Estas faixas foram determinadas de maneira empírica, com o objetivo de detectar os contrastes entre as hierarquias do marcador CRFM.

O número de imagens binárias a serem geradas afeta diretamente a performance do algoritmo, pois elas são processadas individualmente nos próximos passos. Utiliza-se um número par de imagens a serem paralelizadas, de acordo o número de núcleos comumente encontrados em processadores atuais.

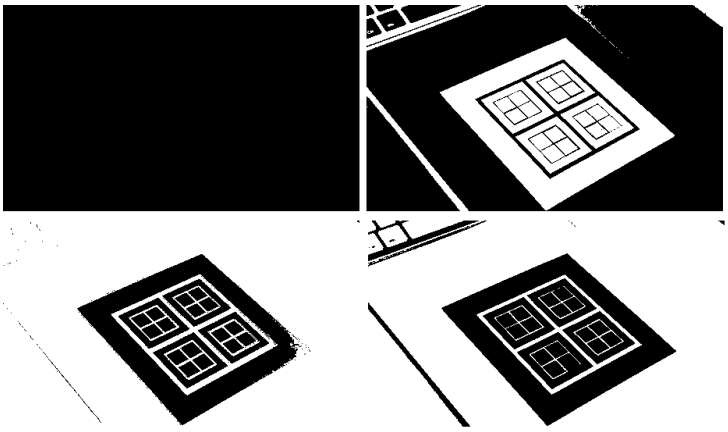
As faixas de *threshold* são utilizadas como valores BGR, e implementadas conforme Figura 5.4. Como resultado desta etapa, obtemos 4 imagens binárias, conforme Figura 5.5.

Figura 5.4 – Código-fonte: *threshold*-binário.

```
1  void ThresholdBinario(const cv::Mat& imgIn, Threshold type, cv::Mat& imgOut){
2     switch (type){
3        case Threshold::LOW:
4           inRange(imgIn, Scalar(0, 0, 0), Scalar(125, 125, 215), imgOut);
5        case Threshold::MEDIUM:
6           inRange(imgIn, Scalar(40, 40, 40), Scalar(220, 220, 220), imgOut);
7        case Threshold::HIGH:
8           inRange(imgIn, Scalar(100, 100, 100), Scalar(255, 255, 255), imgOut);
9        case Threshold::HIGHEST:
10          inRange(imgIn, Scalar(100, 200, 200), Scalar(190, 255, 255), imgOut);
11    }
12 }
```

Fonte: Próprio autor.

Figura 5.5 – Imagens binárias geradas a partir das diferentes faixas de *threshold*.



Fonte: Próprio autor.

5.1.2 Detecção de contornos

Nesta etapa, é utilizado o algoritmo de detecção de contornos Suzuki-Abe em cada uma das imagens geradas no passo anterior. O algoritmo retorna uma lista de contornos, onde cada contorno é representado por outra lista de pontos cartesianos. Estes pontos indicam a posição de cada *pixel* pertencente ao contorno em relação a imagem.

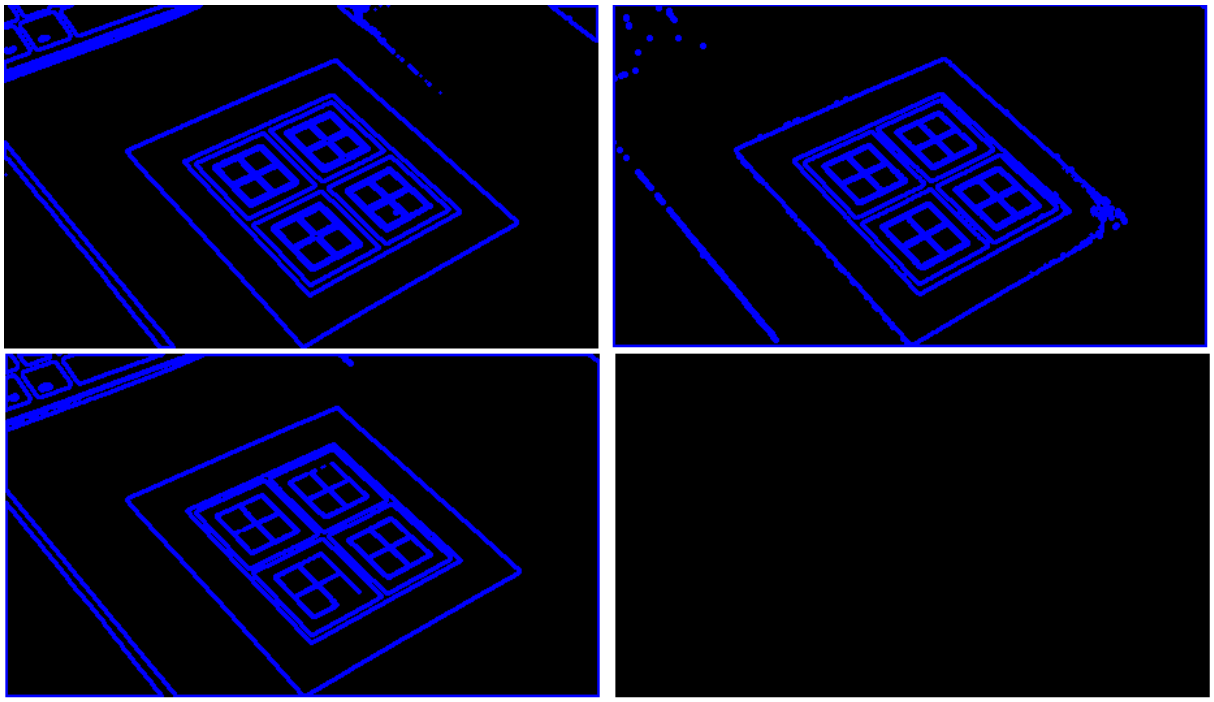
Figura 5.6 – Código-fonte: busca de contornos.

```
1 void BuscarContornos(const Mat& imgIn, vector<vector<Point>>& contours)
2 {
3    cv::findContours(imgIn, contours, RETR_TREE, CHAIN_APPROX_NONE);
4 }
```

Fonte: Próprio autor.

A Figura 5.7 apresenta em azul os contornos detectados a partir de cada uma das imagens binárias, e a Figura 5.6 seu código para execução.

Figura 5.7 – Contornos detectados com o algoritmo Suzuki-Abe.



Fonte: Próprio autor.

5.1.3 Filtragem de contornos por perímetro

Este passo tem como objetivo eliminar contornos que não possuam um tamanho mínimo de perímetro (Figura 5.8). Para determinar o valor mínimo de perímetro é utilizada uma porcentagem do tamanho da imagem original sendo analisada. O tamanho do perímetro de cada contorno é estabelecido pela quantidade de elementos de seu vetor (número de *pixels*).

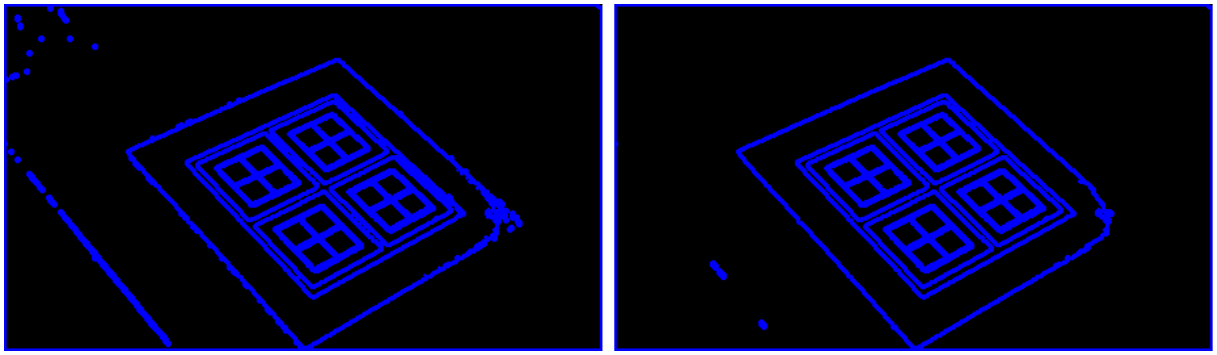
Figura 5.8 – Código-fonte: filtragem de contornos por perímetro.

```
1  void FiltragemContornos_Perimetro(const Mat& imgSrc, vector<vector<Point>>& contours)
2  {
3     // Calcula tamanho mínimo
4     double minMarkerPerimeterRate = 0.02;
5     unsigned int minPerimeterPixels =
6         (unsigned int)(minMarkerPerimeterRate * max(imgSrc.cols, imgSrc.rows));
7
8     // Remove contornos fora do tamanho
9     for (int i = 0; i< contours.size(); i++)
10    {
11       if (contours[i].size() < minPerimeterPixels)
12       {
13          contours.erase(contours.begin() + i);
14          i--;
15       }
16    }
17 }
```

Fonte: Próprio autor.

A Figura 5.9 mostra um comparativo dos contornos removidos que não atingem o tamanho mínimo de perímetro.

Figura 5.9 – Filtragem de contornos por perímetro mínimo.



Fonte: Próprio autor.

5.1.4 Filtragem por aproximação de curvas

O principal objetivo desta etapa é filtrar e manter apenas os contornos que tenham um formato aproximado de quadrilátero, ou seja, que possuam apenas quatro curvas (Figuras 5.10 e 5.11).

Esta etapa busca filtrar os contornos através de uma técnica de aproximação de curvas utilizando o algoritmo Douglas-Peucker. O algoritmo utiliza um valor máximo para tolerância da variação de curvas dos contornos, indicando quando um *pixel* deve ou não ser aproximado. Este valor é baseado em uma porcentagem do tamanho do contorno sendo analisado.

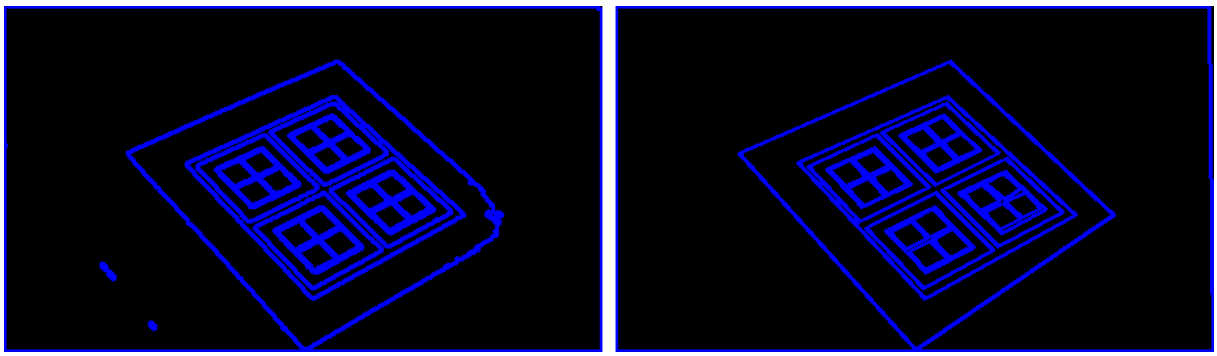
A função de aproximação de curvas recebe o contorno a ser analisado e o valor de precisão. A partir desta função, é gerado então um novo contorno com seus *pixels* (curvas) aproximados. Após, é feito então uma filtragem baseada em seu novo número de curvas, onde são removidos todos os contornos que possuam um valor diferente de quatro.

Figura 5.10 – Código-fonte: aproximação e filtragem de curvas.

```
1  void FiltragemContornos_AproximacaoDeCurvas(vector<vector<Point>>& contours)
2  {
3     double accuracyRate = 0.03;
4
5     for (int i = 0; i < contours.size(); i++)
6     {
7        // Gera o novo contorno aproximado
8        vector< Point > approxCurve;
9        approxPolyDP(contours[i], approxCurve,
10           double(contours[i].size()) * accuracyRate, true);
11
12       // Verifica se possui quatro lados
13       if (approxCurve.size() != 4)
14       {
15          contours.erase(contours.begin() + i);
16          i--;
17          continue;
18       }
19
20       // Substitui no vetor pelo contorno aproximado
21       contours[i].resize(4);
22       for (int j = 0; j < 4; j++)
23       {
24          contours[i][j] = Point2f((float)approxCurve[j].x, (float)approxCurve[j].y);
25       }
26    }
27 }
```

Fonte: Próprio autor.

Figura 5.11 – Exemplo filtragem por aproximação de curvas.



Fonte: Próprio autor.

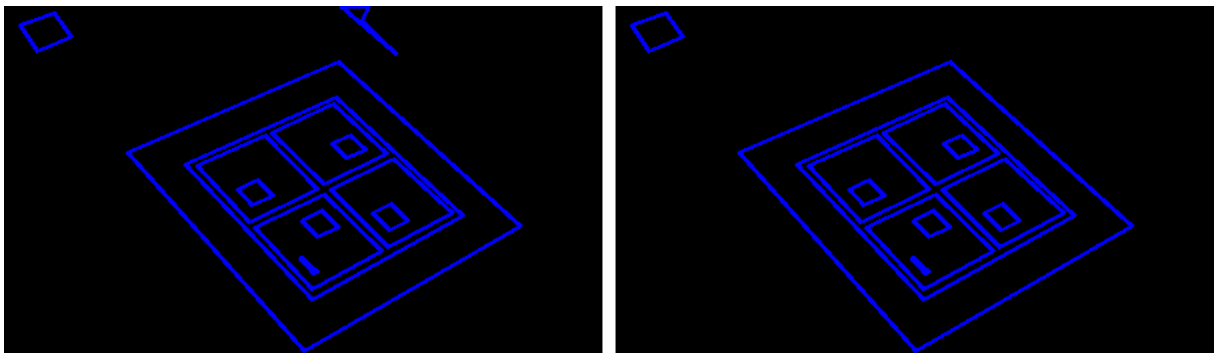
Também nesta mesma etapa, é feito o processo de filtragem de contornos convexos, removendo todos os contornos previamente aproximados que não sejam convexos (Figuras 5.12 e 5.13).

Figura 5.12 – Código-fonte: filtragem de contornos convexos.

```
void FiltragemContornos_Convexos(vector<vector<Point>>& contours)
{
   for (int i = 0; i < contours.size(); i++)
   {
      // Verifica se contorno é convexo
      if (!isContourConvex(contours[i]))
      {
         contours.erase(contours.begin() + i);
         i--;
      }
   }
}
```

Fonte: Próprio autor.

Figura 5.13 – Exemplo de filtragem por remoção de quadriláteros não convexos.

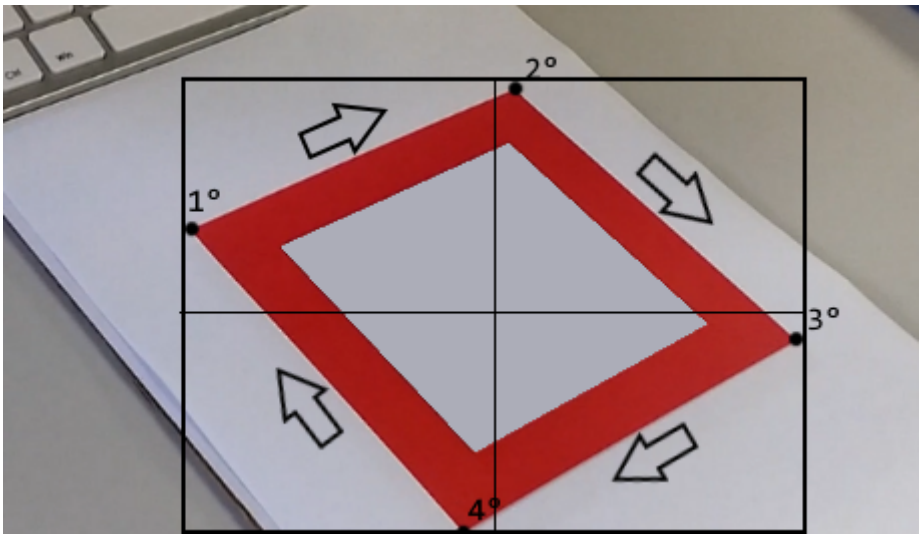


Fonte: Próprio autor.

5.1.5 Ordenação de vértices dos quadriláteros

Este próximo passo consiste da ordenação dos vértices dos quadriláteros gerados. A ordenação final dos vértices (Figura 5.14) deve obedecer a seguinte ordem: índices ordenados em sentido horário, começando a partir do vértice superior-esquerdo em relação a caixa delimitadora mínima do bloco principal.

Figura 5.14 – Ordenação dos vértices em relação a sua caixa delimitadora do bloco principal de uma hierarquia.



Fonte: Próprio autor.

O primeiro passo desta etapa é a ordenação dos vértices de cada quadrilátero para a ordem em sentido horário, conforme demonstrado na Figura 5.15.

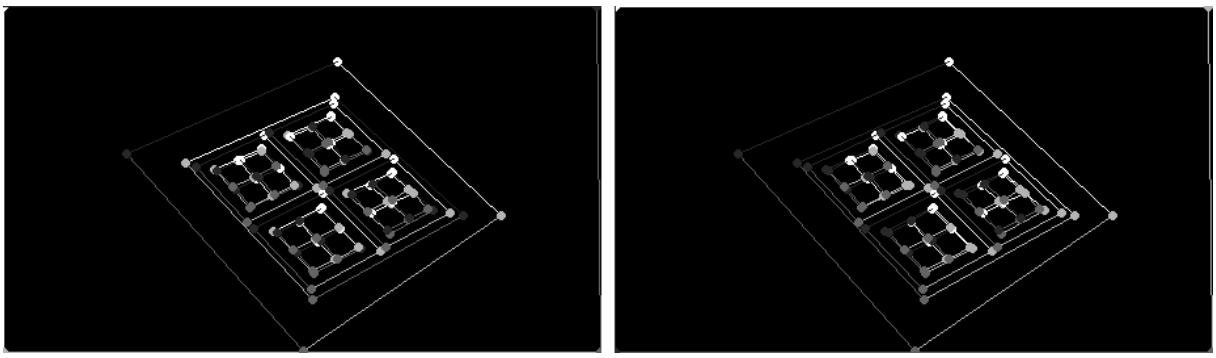
Figura 5.15 – Código-fonte: Ordenação de vértices em sentido horário.

```
void ReodernarVertices_SentidoHorario(vector<vector<Point>>& contours)
{
   for (unsigned int i = 0; i < contours.size(); i++)
   {
      double dx1 = contours[i][1].x - contours[i][0].x;
      double dy1 = contours[i][1].y - contours[i][0].y;
      double dx2 = contours[i][2].x - contours[i][0].x;
      double dy2 = contours[i][2].y - contours[i][0].y;
      double crossProduct = (dx1 * dy2) - (dy1 * dx2);

      // Se não está em sentido horário
      if (crossProduct < 0.0)
      {
         swap(contours[i][1], contours[i][3]);
      }
   }
}
```

Fonte: Próprio autor.

A Figura 5.16 exibe um comparativo de ordenação dos vértices para o sentido horário. A ordem dos vértices pode ser deduzida a partir da intensidade das linhas de ligação entre eles.

Figura 5.16 – Ordenação de vértices para o sentido horário.



Fonte: Próprio autor.

Em seguida, é feita a ordenação dos índices dos vértices (Figura 5.17). Para isto, utiliza-se o ponto superior-esquerdo da caixa delimitadora em relação ao quadrilátero, e altera-se o índice dos vértices de maneira cíclica, até que o primeiro vértice da lista (Figura 5.18), seja o mais próximo do canto superior-esquerdo da caixa delimitadora.

Figura 5.17 – Código-fonte: ordenação de vértices em relação a caixa delimitadora.

```
void ReordernarVertices_Posicoes(vector<vector<Point>>& contours) {
   for (unsigned int i = 0; i < contours.size(); i++){
      float minX = contours[i][0].x, minY = contours[i][0].y;

      // Busca o canto superior-esquerdo
      for (size_t j = 0; j < contours[i].size(); j++){
         if (contours[i][j].x < minX)
            minX = contours[i][j].x;

         if (contours[i][j].y < minY)
            minY = contours[i][j].y;
      }

      // Rotaciona a ordem dos vértices até que o primeiro vértice
      // se torne o vértice mais próximo do canto superior-direito
      int posTopLeft =
          _buscarVerticeMaisProximoDePosicao(contours[i], Point2f(minX, minY), 0);

      if (posTopLeft != 0)
         for (int t = posTopLeft; t < 4 && t != 0; t++){
            contours[i].insert(contours[i].begin(), contours[i][3]);
            contours[i].pop_back();
         }
   }
}
```

Fonte: Próprio autor.

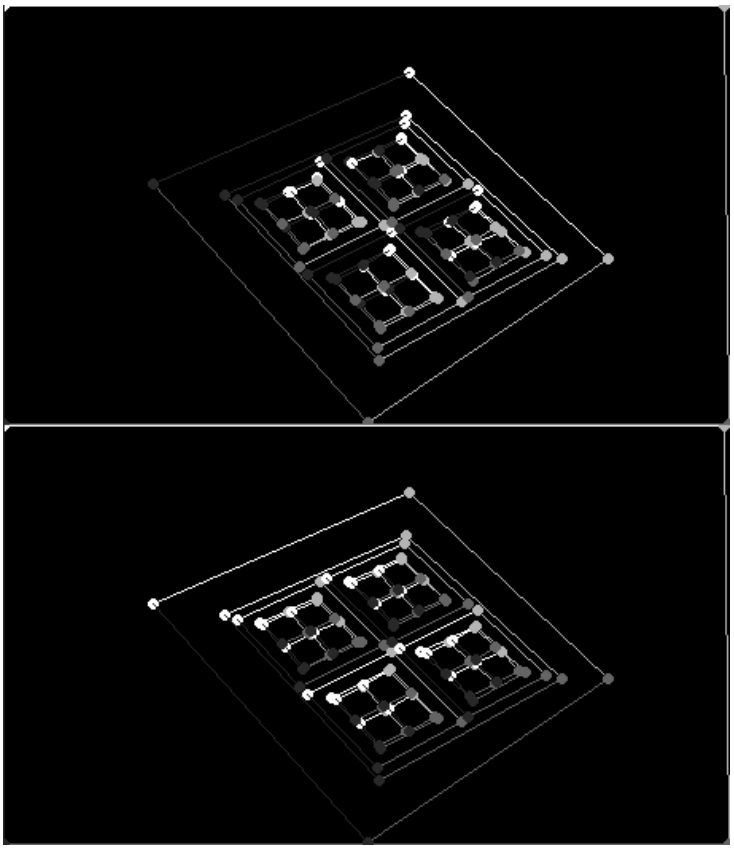
Figura 5.18 – Código-fonte: ordenação de vértices em relação a caixa delimitadora (continuação).

```
int _buscarVerticeMaisProximoDePosicao(vector<Point> vertices, Point2f target, int startIndex)
{
   int sum = 0;
   int index = startIndex;
   sum += (int)abs(vertices[startIndex].x - target.x);
   sum += (int)abs(vertices[startIndex].y - target.y);

   // Selecionar o vértice / índice de menor distância para a posição alvo
   for (size_t i = startIndex; i < vertices.size(); i++){
      if (abs(vertices[i].x - target.x) + abs(vertices[i].y - target.y) < sum)
      {
         sum = (int)abs(vertices[i].x - target.x) + (int)abs(vertices[i].y - target.y);
         index = i;
      }
   }

   return index;
}
```

Fonte: Próprio autor.

A Figura 5.19 demonstra o resultado final deste processo.

Figura 5.19 – Quadriláteros com vértices ordenados corretamente.



Fonte: Próprio autor.

5.1.6 Agrupamento dos quadriláteros

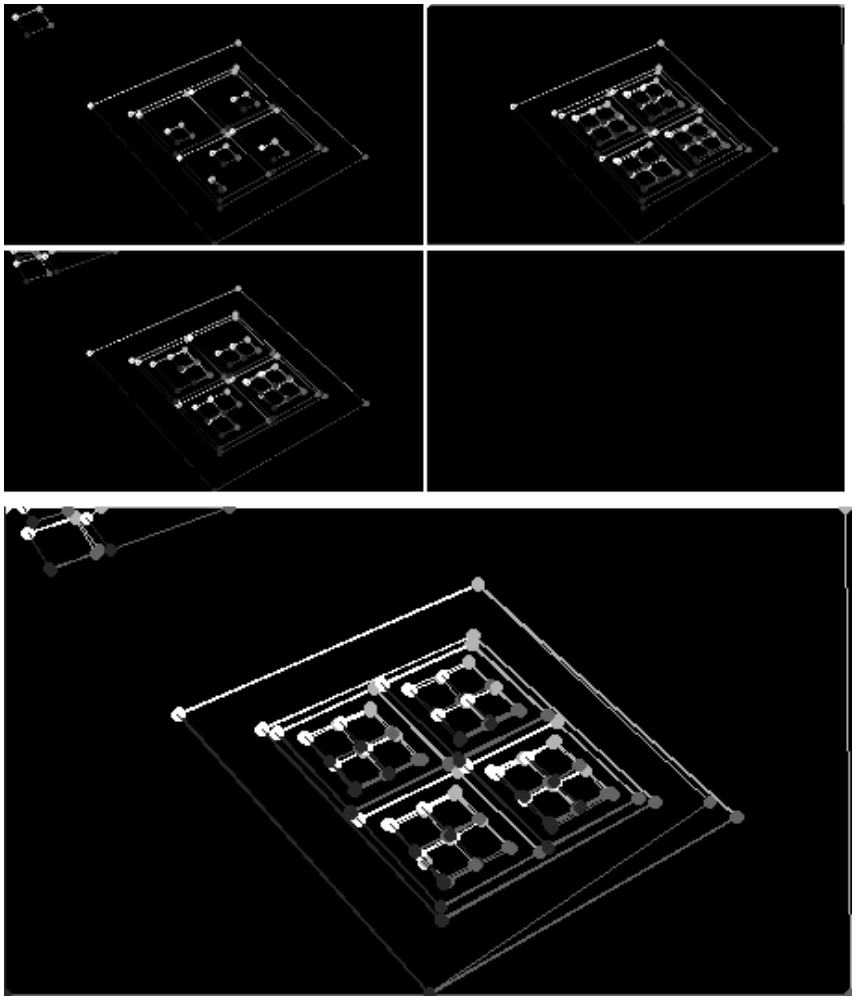
Até este passo, todos os processos eram repetidos de forma paralelizada sobre cada uma das imagens binárias geradas. Este passo consiste do agrupamento dos quadriláteros detectados em um único vetor (Figuras 5.20 e 5.21), afim de compara-los posteriormente.

Figura 5.20 – Código-fonte: agrupamento de contornos.

```
1  // Agrupamento dos vetores de cada imagem binária
2  vector<vector<Point>> contours;
3  for (int i = 0; i < 4; i++)
4     for (int j = 0; j < vecContours[i].size(); j++)
5        contours.push_back(vecContours[i][j]);
```

Fonte: Próprio autor.

Figura 5.21 – Agrupamento de contornos.



Fonte: Próprio autor.

5.1.7 Filtragem de quadriláteros semelhantes

Após o agrupamento dos quadriláteros, é necessário realizar a filtragem dos quadriláteros que possuam área e centroide semelhantes, mantendo apenas o quadrilátero maior entre os que tenham um certo nível de similaridade. O primeiro passo deste processo envolve calcular a posição do centroide e o valor de área de para cada um dos quadriláteros (Figura 5.22). Em seguida, são ordenados conforme o tamanho de sua área (Figura 5.23) utilizando o método bolha de ordenação.

Figura 5.22 – Código-fonte: transformação de contorno para quadrilátero;

```
inline void CalcularAreaECentroideContornos(vector<vector<Point>>& contours,
    vector<Quadrilateral>& vecQuads){
   // Durante a inserção no vetor, transforma em quadrilateral ,
   // calcula a área e centroide pelo construtor
   for (unsigned i = 0; i < contours.size(); i++)
      vecQuads.push_back(Quadrilateral(contours[i]));
}
```

Fonte: Próprio autor.

Figura 5.23 – Código-fonte: ordenação de quadriláteros por tamanho de área.

```
inline void OrdenarListaDeContornosPorArea(vector<Quadrilateral>& vecQuads){
   // Ordenar lista de contornos por área utilizando método bolha
   for (unsigned int c = 0; c < vecQuads.size(); c++)
      for (unsigned int d = 0; d < vecQuads.size() - 1; d++)
         if (vecQuads[d].area < vecQuads[d + 1].area)
            swap(vecQuads[d], vecQuads[d + 1]);
}
```

Fonte: Próprio autor.

A similaridade dos quadriláteros é baseada nos atributos calculados anteriormente: posição do centroide dos quadriláteros, e tamanho de área dos quadriláteros.

Cada quadrilátero é comparado com os quadriláteros restantes da lista, e são removidos os que atendam ambas as regras:

- Possuir uma área entre 55% e 100% do quadrilátero sendo comparado;
- Possuir seu centroide a uma distância inferior a distância mínima estabelecida em relação ao quadrilátero sendo comparado.

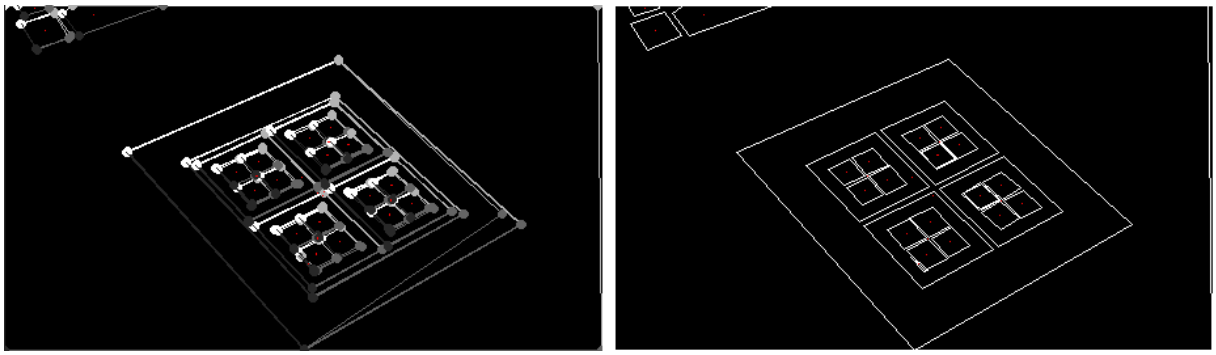
Este passo garante que sejam removidos os quadriláteros de posições e áreas semelhantes, mas de forma que mantenha os quadriláteros filhos que possam vir a formar uma hierarquia similar a do marcador CRFM. O algoritmo de filtragem pode ser conferido conforme Figura 5.24, e o resultado final do processo, conforme Figura 5.25.

Figura 5.24 – Código-fonte: filtragem de contornos semelhantes.

```
inline void CompararERemoverContornosEquivalentesEmAreaECentroide(
    vector<Quadrilateral>& vecQuads){
   // Comparar e remover contornos equivalentes em area e centroide
   float maxPercentArea = 55;
   float minCentroidDistancePercentage = 12;

   for (unsigned int i = 0; i < vecQuads.size(); i++) {
      for (unsigned int j = i + 1; j < vecQuads.size(); j++) {
         Quadrilateral q1 = vecQuads[i];
         Quadrilateral q2 = vecQuads[j];

         // Filtragem por área
         double tot = (q1.area + q2.area);
         double diff = (q1.area - q2.area);
         double percent = ((diff / (tot)) * 100);
         if (!BETWEEN(percent, 0, maxPercentArea)) break;

         // Filtragem por distância de centróide
         double avgDist, dx, dy;
         dx = (q1.vertices[2].x - q1.vertices[0].x);
         dy = (q1.vertices[2].y - q1.vertices[0].y);
         avgDist = sqrt(dx*dx + dy * dy);
         dx = (q1.vertices[3].x - q1.vertices[1].x);
         dy = (q1.vertices[3].y - q1.vertices[1].y);
         avgDist = (avgDist + sqrt(dx*dx + dy * dy)) / 2;

         float minCentroidDistance = (avgDist / 100) * minCentroidDistancePercentage;
         if ((abs(q1.centroid.x - q2.centroid.x) < minCentroidDistance &&
             abs(q1.centroid.y - q2.centroid.y) < minCentroidDistance)){
            vecQuads.erase(vecQuads.begin() + j);
            j--;
         }
      }
   }
}
```

Fonte: Próprio autor.

Figura 5.25 – Exemplo de filtragem de quadriláteros semelhantes.



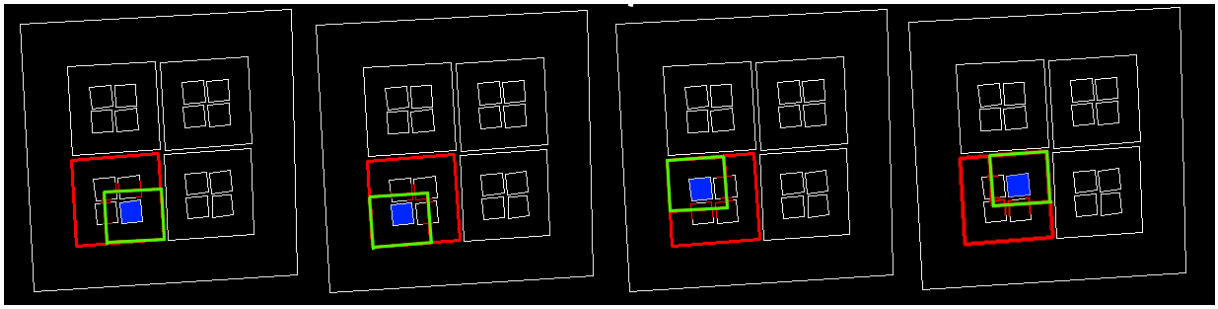
Fonte: Próprio autor.

5.1.8 Filtragem de candidatos finais

Esta última etapa do processo de detecção de candidatos, tem o objetivo de filtrar e manter apenas os quadriláteros que contenham e/ou façam parte de uma hierarquia.

Para cada quadrilátero, são denominadas quatro áreas de interesse. Estas áreas de interesse são comparadas com outros quadriláteros que possam vir a ser tornar blocos internos da hierarquia do quadrilátero sendo analisado. Os quadriláteros que tenham área e centroide compatível com as áreas de interesse são então selecionados como blocos internos do quadrilátero. A Figura 5.26 exemplifica o processo realizado.

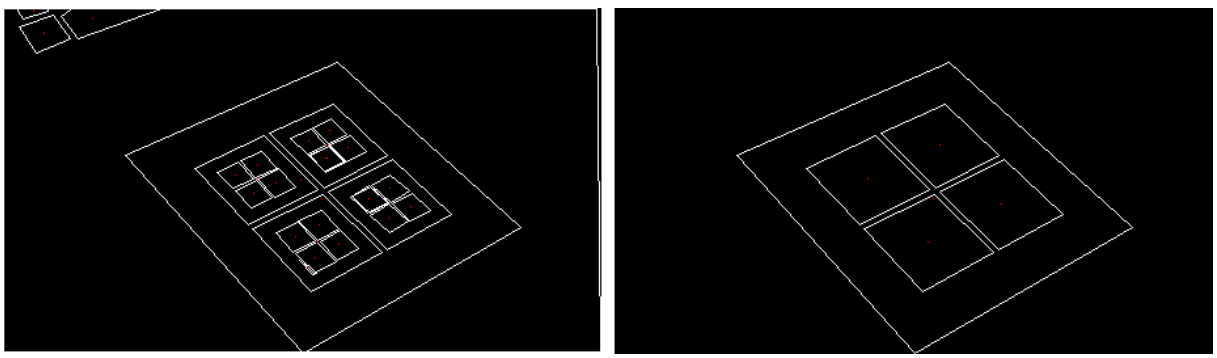
Figura 5.26 – Quadrilátero sendo analisado (vermelho). Áreas de interesse geradas (verde). Respectivos quadriláteros detectados como blocos internos (azul).



Fonte: Próprio autor.

Após a seleção de blocos internos para todos os quadriláteros, são então filtrados e mantidos apenas os quadriláteros que formam uma hierarquia completa. Ou seja, possuam um bloco internado detectado corretamente para cada posição de interesse. A Figura 5.27 demonstra apenas os quadriláteros em que foram detectados uma hierarquia completa, em um total de 5 hierarquias detectadas corretamente. Figuras 5.28 e 5.29 demonstram a implementação do processo.

Figura 5.27 – Filtragem final de blocos com hierarquia completa detectada.



Fonte: Próprio autor.

Figura 5.28 – Código-fonte: geração de áreas de interesse e busca quadriláteros internos.

```
inline void BuscarFilhosCandidatos(vector<Quadrilateral>& vecQuads){
   // Para cada quadrilátero, busca os possíveis filhos
   for (size_t i = 0; i < vecQuads.size(); i++){
      // Determina a área dos possíveis filhos
      vector<Point> v = vecQuads[i].vertices, t0, t1, t2, t3;
      // Bottom-left
      t0.push_back(Point((v[2].x + v[0].x) / 2, (v[2].y + v[0].y) / 2));
      t0.push_back(Point((v[2].x + v[1].x) / 2, (v[2].y + v[1].y) / 2));
      t0.push_back(Point(v[2].x, v[2].y));
      t0.push_back(Point((v[2].x + v[3].x) / 2, (v[2].y + v[3].y) / 2));
      // Repete para as áreas t1 Bottom-right, t2 Top-left e t3 Top-right
      // *código removido para exemplificação*

      // Armazena quadriláteros alvo que se encaixam em alguma das posições
      for (size_t j = i + 1; j < vecQuads.size(); j++){
         float areaMin = (float)((vecQuads[i].area / 100) * 3);
         float areaMax = (float)((vecQuads[i].area / 100) * 13);

         if (!BETWEEN(vecQuads[j].area, areaMin, areaMax)) continue;

         if (_centroidePertenceAoAlvo(t0, vecQuads[j].centroid) > 0)
            vecQuads[i].childrenBottomRight.push_back(vecQuads[j]);
         else if (_centroidePertenceAoAlvo(t1, vecQuads[j].centroid) > 0)
            vecQuads[i].childrenBottomLeft.push_back(vecQuads[j]);
         else if (_centroidePertenceAoAlvo(t2, vecQuads[j].centroid) > 0)
            vecQuads[i].childrenTopLeft.push_back(vecQuads[j]);
         else if (_centroidePertenceAoAlvo(t3, vecQuads[j].centroid) > 0)
            vecQuads[i].childrenTopRight.push_back(vecQuads[j]);
      }
   }
}
```

Fonte: Próprio autor.

Figura 5.29 – Código-fonte: seleção final de quadriláteros que formam uma hierarquia completa.

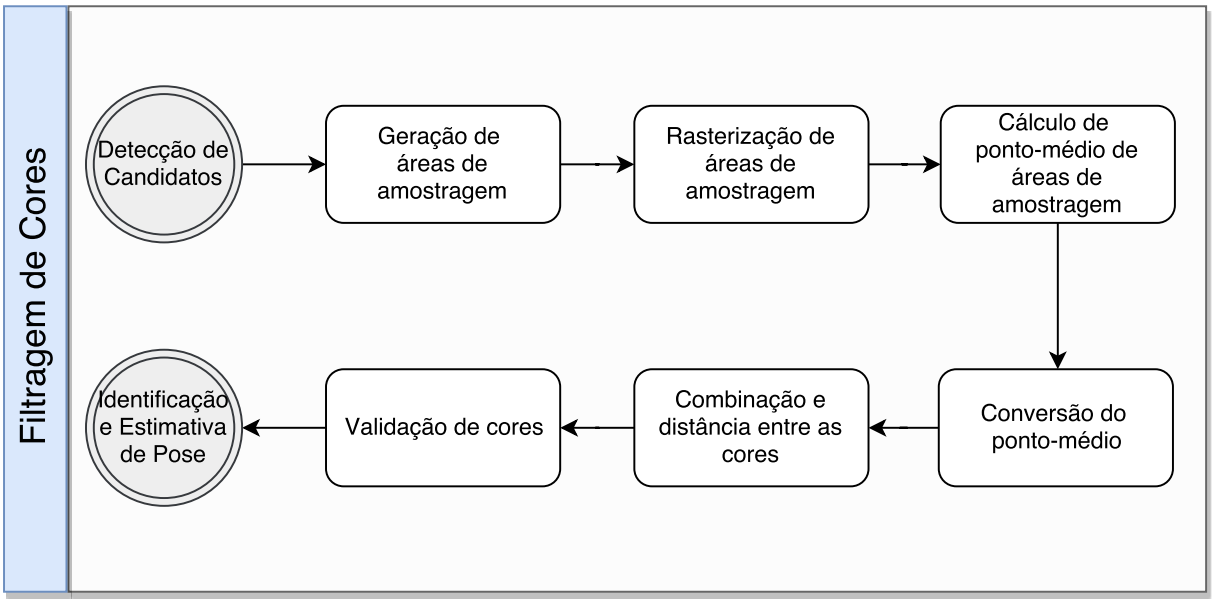
```
inline void GerarVetorFinal(const vector<Quadrilateral>& vecQuads,
    vector<Quadrilateral>& vecQuadsFinal){
   // Gera o vetor com os candidatos finais,
   // ou seja, os candidatos que possuem um filho em cada posição
   for (int i = 0; i < vecQuads.size(); i++){
      if  (vecQuads[i].childrenTopLeft.size() > 0 &&
          vecQuads[i].childrenTopRight.size() > 0 &&
          vecQuads[i].childrenBottomLeft.size() > 0 &&
         vecQuads[i].childrenBottomRight.size() > 0)
      {
         vecQuadsFinal.push_back(vecQuads[i]);
      }
   }
}
```

Fonte: Próprio autor.

5.2 FILTRAGEM DE CORES

Após os contornos candidatos serem filtrados, apenas contornos em formato de quadriláteros e pertencentes a alguma hierarquia devem permanecer. Então, para cada quadrilátero, é necessário determinar sua respectiva cor. Para isso, são analisados os *pixels* contidos dentro de suas áreas. Estes passos são descritos a seguir, conforme demonstra a Figura 5.30;

Figura 5.30 – Diagrama de fluxo do processo de filtragem de cores.



Fonte: Próprio autor.

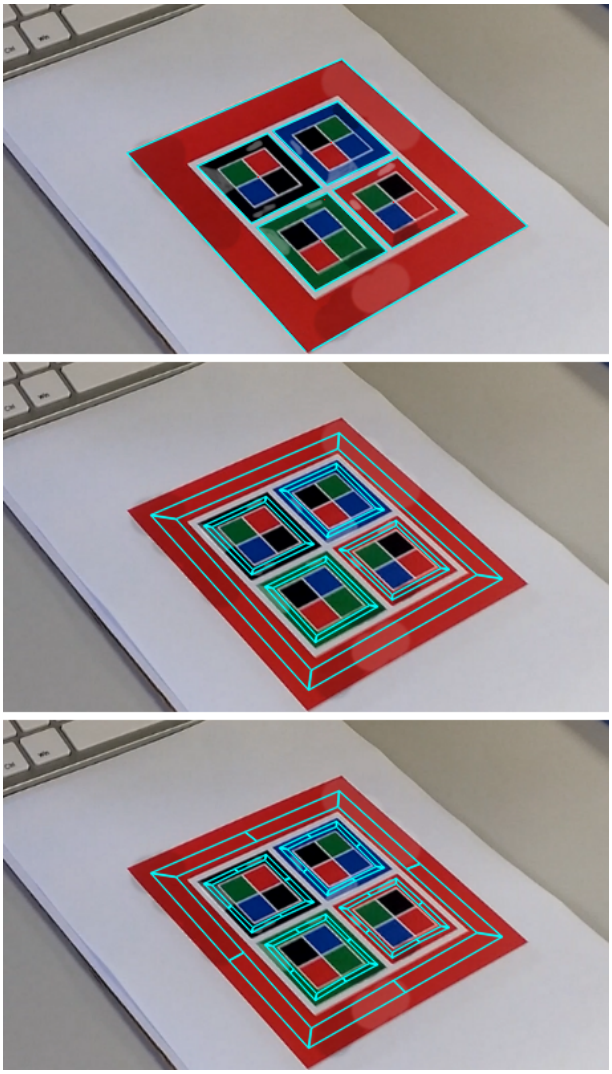
5.2.1 Geração de áreas de amostragem

A fim de reduzir o processamento requerido pelo algoritmo, é definida uma área reduzida do quadrilátero a ser analisado. A geração desta área de amostragem tem seu cálculo baseado em cada um dos lados do quadrilátero detectado.

Após as áreas de amostragem serem determinadas, elas são então convertidas em oito quadriláteros correspondentes. Aumentando o número de quadriláteros, e determinando a cor de cada um separadamente, é possível reduzir o número de erros e falsos positivos na detecção de cores. Estes erros são geralmente causados por mudanças da iluminação, reflexos e sombras sobre o marcador.

A Figura 5.31 exemplifica o processo realizado. Figuras 5.32 e 5.33 demonstram a implementação do processo.

Figura 5.31 – Exemplo de áreas de amostragem de um marcador.



Fonte: Próprio autor.

Figura 5.32 – Código-fonte: geração de áreas de amostragem.

```
inline void GeracaoVerticesAmostragem(const vector<Point> &vIn,
    vector<vector<Point>>& vOut)
{
   Point x = vIn[3] - vIn[0], d1 = vIn[2] - vIn[0], d2 = vIn[1] - vIn[3];

   float cross = d1.x * d2.y - d1.y  *d2.x;
   if (abs(cross) < 1e-8)
      return;

   double t1 = (x.x * d2.y - x.y * d2.x) / cross;
   Point centroid = vIn[0] + d1 * t1;
   Point P0 = vIn[0], P1 = vIn[1], P2 = vIn[2],  P3 = vIn[3];

   Point centroidP0, centroidP1, centroidP2, centroidP3;
   centroidP0 = centroidP1 = centroidP2 = centroidP3 = centroid;

   int percent = 2; // = 25%
   for (int i = 0; i < (int)percent; i++)
   {
      centroidP0 = Point((P0.x + centroidP0.x) / 2, (P0.y + centroidP0.y) / 2);
      centroidP1 = Point((P1.x + centroidP1.x) / 2, (P1.y + centroidP1.y) / 2);
      centroidP2 = Point((P2.x + centroidP2.x) / 2, (P2.y + centroidP2.y) / 2);
      centroidP3 = Point((P3.x + centroidP3.x) / 2, (P3.y + centroidP3.y) / 2);
   }

   for (int i = 0; i < (int)PercentEvaluation::PERCENT50; i++)
   {
      P0 = Point((P0.x + centroidP0.x) / 2, (P0.y + centroidP0.y) / 2);
      P1 = Point((P1.x + centroidP1.x) / 2, (P1.y + centroidP1.y) / 2);
      P2 = Point((P2.x + centroidP2.x) / 2, (P2.y + centroidP2.y) / 2);
      P3 = Point((P3.x + centroidP3.x) / 2, (P3.y + centroidP3.y) / 2);
   }

   vOut.clear();
   vOut.push_back(vector<Point>());
   vOut[0].push_back(P0);   vOut[0].push_back(P1);
   vOut[0].push_back(centroidP1);   vOut[0].push_back(centroidP0);

   vOut.push_back(vector<Point>());
   vOut[1].push_back(centroidP1);   vOut[1].push_back(P1);
   vOut[1].push_back(P2);   vOut[1].push_back(centroidP2);

   vOut.push_back(vector<Point>());
   vOut[2].push_back(centroidP3);   vOut[2].push_back(centroidP2);
   vOut[2].push_back(P2);   vOut[2].push_back(P3);

   vOut.push_back(vector<Point>());
   vOut[3].push_back(P0);   vOut[3].push_back(centroidP0);
   vOut[3].push_back(centroidP3); vOut[3].push_back(P3);
```

Fonte: Próprio autor.

Figura 5.33 – Código-fonte: geração de áreas de amostragem (continuação).

```
   // Transforma em 8 retangulos
   for (int j = 0; j < 4; j++)
   {
      vector<Point> quad = vOut[j], t1, t2;

      if (j % 2 == 0)
      {
         t1.push_back(Point(quad[0].x, quad[0].y));
         t1.push_back(Point((quad[0].x + quad[1].x) / 2,
             (quad[0].y + quad[1].y) / 2));
         t1.push_back(Point((quad[2].x + quad[3].x) / 2,
             (quad[2].y + quad[3].y) / 2));
         t1.push_back(Point(quad[3].x, quad[3].y));

         t2.push_back(Point((quad[0].x + quad[1].x) / 2,
             (quad[0].y + quad[1].y) / 2));
         t2.push_back(Point(quad[1].x, quad[1].y));
         t2.push_back(Point(quad[2].x, quad[2].y));
         t2.push_back(Point((quad[2].x + quad[3].x) / 2,
             (quad[2].y + quad[3].y) / 2));
      }
      else
      {
         t1.push_back(Point(quad[0].x, quad[0].y));
         t1.push_back(Point(quad[1].x, quad[1].y));
         t1.push_back(Point((quad[1].x + quad[2].x) / 2,
             (quad[1].y + quad[2].y) / 2));
         t1.push_back(Point((quad[0].x + quad[3].x) / 2,
             (quad[0].y + quad[3].y) / 2));

         t2.push_back(Point((quad[0].x + quad[3].x) / 2,
             (quad[0].y + quad[3].y) / 2));
         t2.push_back(Point((quad[1].x + quad[2].x) / 2,
             (quad[1].y + quad[2].y) / 2));
         t2.push_back(Point(quad[2].x, quad[2].y));
         t2.push_back(Point(quad[3].x, quad[3].y));
      }

      vOut.push_back(t1);
      vOut.push_back(t2);
   }

   vOut.erase(vOut.begin());
   vOut.erase(vOut.begin());
   vOut.erase(vOut.begin());
   vOut.erase(vOut.begin());
}
```

Fonte: Próprio autor.

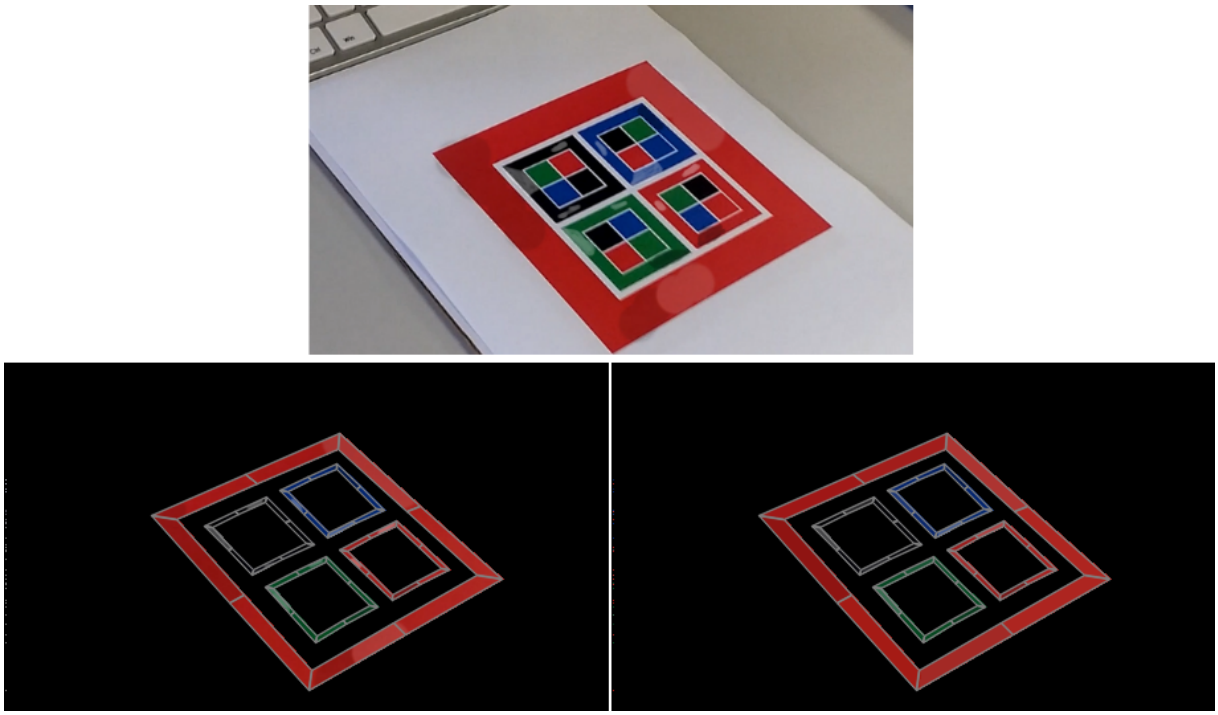
5.2.2 Rasterização das áreas de amostragem

Como os quadriláteros podem ser posicionados sob qualquer orientação na imagem, é necessário realizar um processo de rasterização nas áreas selecionadas. O processo de rasterização foi implementado utilizando o algoritmo Bresenham (BRESENHAM, 1965), que gera uma lista de linhas, onde cada linha contém um intervalo de pixels que pertencem ao quadrilátero alvo. Estas listas de intervalos são utilizadas para iterar apenas sobre os *pixels* que constituem as áreas de amostragem.

5.2.3 Geração de um valor RGB-médio para as áreas de amostragem

Este próximo passo consiste da iteração sobre os *pixels* da área de amostragem rasterizada, e o cálculo valor médio RGB da respectiva região (Figuras 5.34 e 5.35).

Figura 5.34 – (A) Imagem de entrada modificada para realçar efeitos do processo. (B) Áreas rasterizadas da imagem. (C) Áreas rasterizadas exibindo o valor-médio RGB.



Fonte: Próprio autor.

Figura 5.35 – Código-fonte: Rasterização e cálculo do valor médio RGB.

```
inline RGB CalcularCorRGBMedia(const Mat& input, const vector<Point_<int>>& vertices)
{
   // Geração das linhas contendo os intervalos que o quadrilátero pertence
   int HEIGHT_OFFSET;
   vector<long[2]> ContourX = scanPolygonArea(vertices, HEIGHT_OFFSET);

   unsigned int countBlue = 0, countGreen = 0, countRed = 0, countPixels = 0;;

   if (ContourX[ContourX.size() - 1][0] == 0)
      ContourX[ContourX.size() - 1][1] = 0;

   // Iteração sobre as linhas
   for (unsigned int y = 0; y < ContourX.size(); y++)
   {
      if (ContourX[y][1] >= ContourX[y][0])
      {
         long x = ContourX[y][0];
         long len = 1 + ContourX[y][1] - ContourX[y][0];

         while (len--)
         {
            int realY = y + HEIGHT_OFFSET;
            x++;

            if (realY >= input.rows || x >= input.cols)
               continue;

            // Calculo do valor médio RGB
            countPixels++;
            countBlue += input.at<Vec3b>(Point(x, realY))[0];
            countGreen += input.at<Vec3b>(Point(x, realY))[1];
            countRed += input.at<Vec3b>(Point(x, realY))[2];
         }
      }
   }

   RGB pixelMedio;
   pixelMedio.blue = countBlue / countPixels;
   pixelMedio.green = countGreen / countPixels;
   pixelMedio.red = countRed / countPixels;

   return pixelMedio;
}
```

Fonte: Próprio autor.

5.2.4 Conversão para Lab

Após calcular o valor médio RGB, é feito então a conversão dos valores médios de cada área de amostragem, para um *pixel* equivalente em formato Lab. A utilização desta técnica, se dá pelo fato de que manipular *pixels* no formato Lab, ser um forma mais robusta de estimar uma cor em especifico.

Devido a conversão de *pixels* RGB para Lab (ou outros espaços de cores) ser um processo custoso, não são convertidos todos os *pixels* de uma área de amostragem (ou da imagem), mas sim, apenas o valor médio da área analisada.

5.2.5 Combinação e cálculo de distância de cores Lab

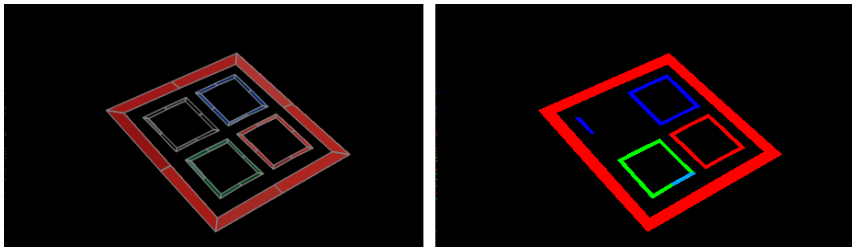
Após a conversão para espaço Lab, é feito o cálculo de distância da cor Lab sendo avaliada, com as cores alvo. As cores alvo utilizadas neste exemplo, são os valores HSV das cores do CRFM, com seu valor de brilho reduzido em 35% (imagens capturadas raramente apresentam o valor puro da cor alvo) e convertidos para Lab, conforme Figura 5.36. As cores alvo também podem ser parametrizadas afim de melhorar a detecção conforme valores calibrados para diferentes câmeras que venham a ser utilizadas.

Figura 5.36 – Exemplo de cores-alvo.

Fonte: Próprio autor.

O cálculo de distância é feito seguindo a fórmula Delta e 2000, utilizando o código adaptado por Fiumara (2015). É calculada a distância do valor Lab da área analisada para cada uma das cores alvo. A cor alvo que resultar em menor distância, é a cor mais similar com a da região comparada (Figuras 5.37 e 5.38).

Figura 5.37 – Exemplo de seleção de cores conforme o cálculo de distância.



Fonte: Próprio autor.

Figura 5.38 – Código-fonte: cálculo de distância de cores Lab.

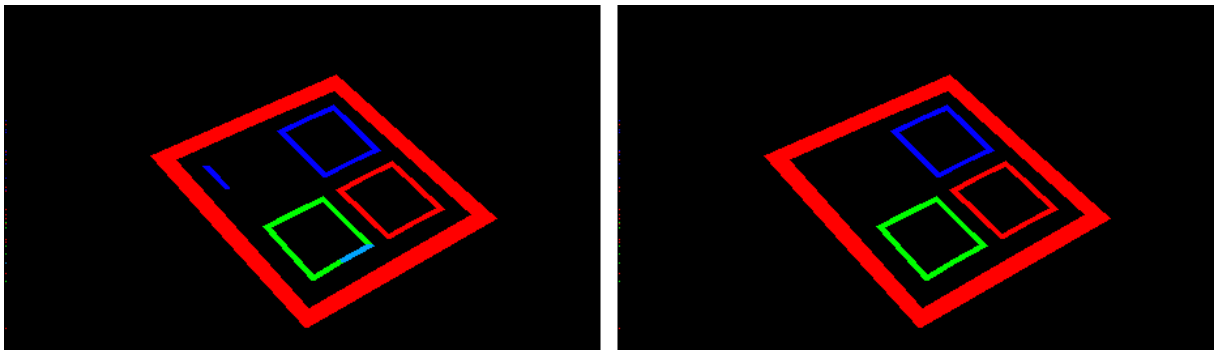
```
inline ColorBWRGB cvtPixel_LAB2BWRGB(Mat3b& pixel)
{
    // Adaptação das faixas de valores LAB OpenCV para LAB original
   CIEDE2000::LAB lab;
   lab = {
      (100.0000 * (pixel.at<Vec3b>(0, 0)[0] / 255.0000)),
      pixel.at<Vec3b>(0, 0)[1] - 128,
      pixel.at<Vec3b>(0, 0)[2] - 128
   };

    // Para cada cor, calcula a distância do valor médio com a cor alvo
   double black = 0, red = 0, blue = 0, green = 0, yellow = 0, orange = 0;
   black += CIEDE2000::CIEDE2000(lab, CIEDE2000::LAB{ 0, 0, 0 });
   blue += CIEDE2000::CIEDE2000(lab, CIEDE2000::LAB{ 20.96, 60.6, -82.53 });
   green += CIEDE2000::CIEDE2000(lab, CIEDE2000::LAB{ 63.38, -65.95, 63.65 });
   orange += CIEDE2000::CIEDE2000(lab, CIEDE2000::LAB{ 53.75, 17.92, 60.53 });
   red += CIEDE2000::CIEDE2000(lab, CIEDE2000::LAB{ 36.97, 61.3, 51.38 });
   yellow += CIEDE2000::CIEDE2000(lab, CIEDE2000::LAB{ 70.57, -16.49, 72.29 });

    // Seleciona a cor que resultou em menor distância
   if (yellow < red && yellow < orange && yellow < green &&
       yellow < blue && yellow < black)
      return ColorBWRGB::YELLOW;

    if (red < yellow && red < orange && red < green && red < blue &&
        red < black)
      return ColorBWRGB::RED;

   if (orange < yellow && orange < red && orange < green && orange < blue &&
       orange < black)
      return ColorBWRGB::ORANGE;

   if (green < yellow && green < red && green < orange && green < blue &&
       green < black)
      return ColorBWRGB::GREEN;

   if (blue < yellow && blue < red && blue < orange && blue < green &&
       blue < black)
      return ColorBWRGB::BLUE;

   if (black < yellow && black < red && black < orange && black < green &&
       black < blue)
      return ColorBWRGB::BLACK;

   return ColorBWRGB::INVALID;
}
```

Fonte: Próprio autor.

5.2.6 Cor final do quadrilátero

Após determinar a cor final de cada área de amostragem, é preciso determinar a cor final do quadrilátero (Figuras 5.39 e 5.40). Para isto, é necessário selecionar a cor predominante das 8 áreas de amostragem. Esta cor deve ser a mesma em pelo menos metade do número de áreas de amostragem (quatro). Caso contrário, o quadrilátero tem sua cor definida como indeterminada, afim de evitar detecções imprecisas.

Figura 5.39 – Determinação da cor predominante dos quadriláteros.



Fonte: Próprio autor.

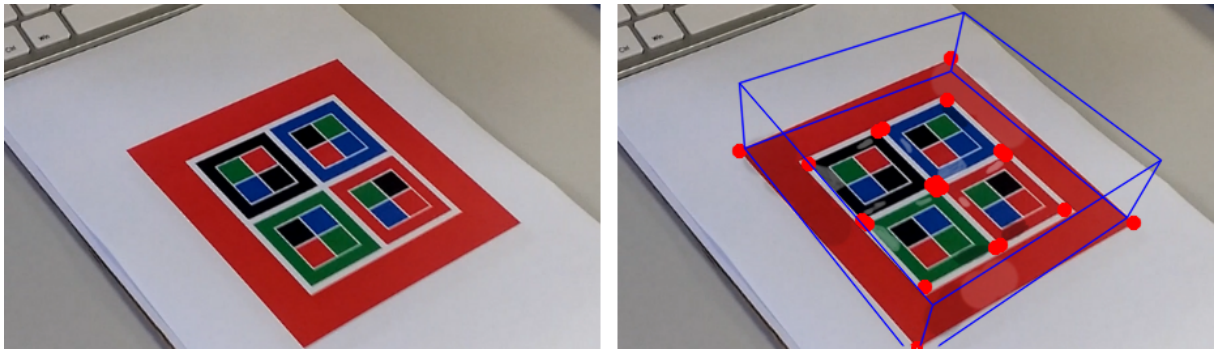
Figura 5.40 – Código-fonte: determinação da cor predominante de um quadrilátero

```
inline ColorBWRGB findMostCommon_(vector<ColorBWRGB>& vecColor){
   std::vector<ColorBWRGB> pop; int popular_cnt = 0;
   vector<ColorBWRGB>::const_iterator first = vecColor.begin(), last = vecColor.end();

   for (vector<ColorBWRGB>::const_iterator it = first; it != last; ++it){
      int temp_cnt = 0;
      for (vector<ColorBWRGB>::const_iterator it2 = it + 1; it2 != last; ++it2)
         if (*it == *it2) ++temp_cnt;
      if (temp_cnt)
         if (temp_cnt > popular_cnt){
            popular_cnt = temp_cnt;
            pop.clear();
            pop.push_back(*it);
         }
         else if (temp_cnt == popular_cnt) pop.push_back(*it);
   }
   if (pop.size() == 1) {
      int cont = 0;
      for (unsigned int i = 0; i < vecColor.size(); i++)
         if (vecColor[i] == pop.front()) cont++;
      if (cont >= 5) return pop[0];
   }
   else if (pop.size() != 1) return ColorBWRGB::INVALID;
   return ColorBWRGB::INVALID;
}
```

Fonte: Próprio autor.

5.3 IDENTIFICAÇÃO DE MARCADORES E PROJEÇÃO

Figura 5.41 – Projeção de um prisma quadrangular sobre um marcador detectado.

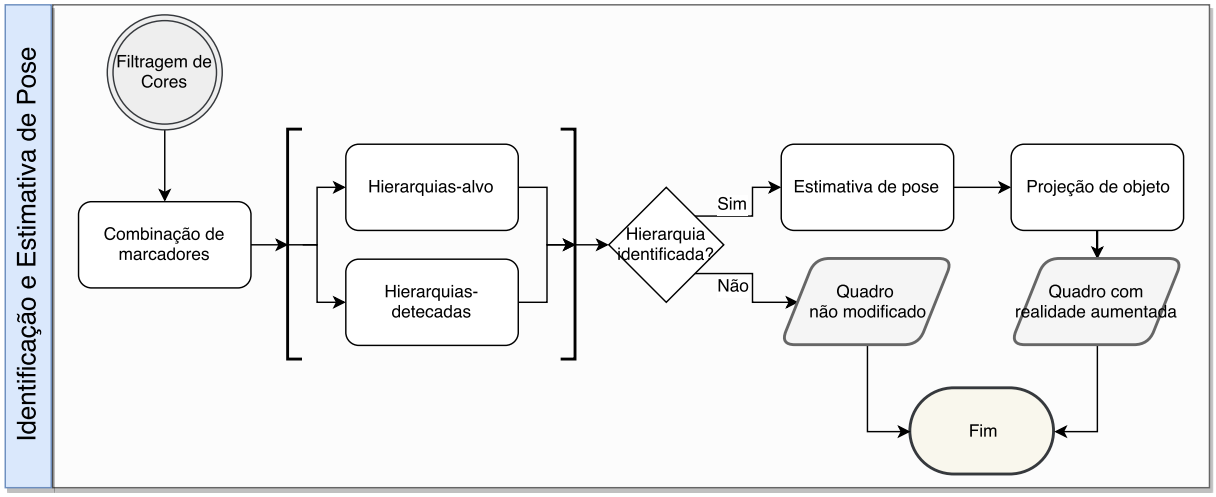


Fonte: Próprio autor.

Este último processo consiste de buscar correspondências entre os marcadores e sequências de cores previamente detectadas, com um dicionário de marcadores alvo. Após encontrar as correspondências é feita a estimativa de pose através do algoritmo Levenberg-Marquardt (MARQUARDT, 1963) (LEVENBERG, 1944) e a projeção de um prisma quadrangular sobre o marcador detectado (Figura 5.41).

A Figura 5.42 demonstra o fluxo de processos a serem descritos a seguir.

Figura 5.42 – Diagrama de fluxo do processo de identificação e estimativa de pose.



Fonte: Próprio autor.

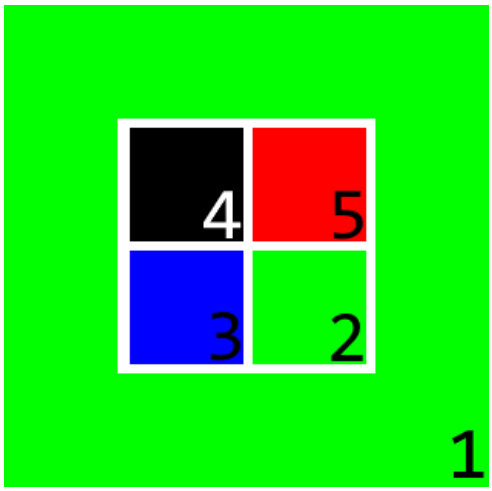
5.3.1 Dicionários de marcadores

Dicionários de marcadores são estruturas de dados fixos previamente determinados, e armazenadas para consulta pelo algoritmo CRFM. Cada dicionário contém uma descrição dos marcadores gerados, conforme seu número de cores e de níveis de hierarquia do dicionário.

Os dicionários seguem uma estrutura onde cada lista de cores representa um marcador e suas hierarquias, e cada cor representa um bloco do marcador alvo.

As cores são armazenadas nos dicionários conforme a ordem das Figuras 5.43, 5.44, 5.45 e 5.46.

Figura 5.43 – Ordem de leitura para armazenamento de marcadores de um nível.



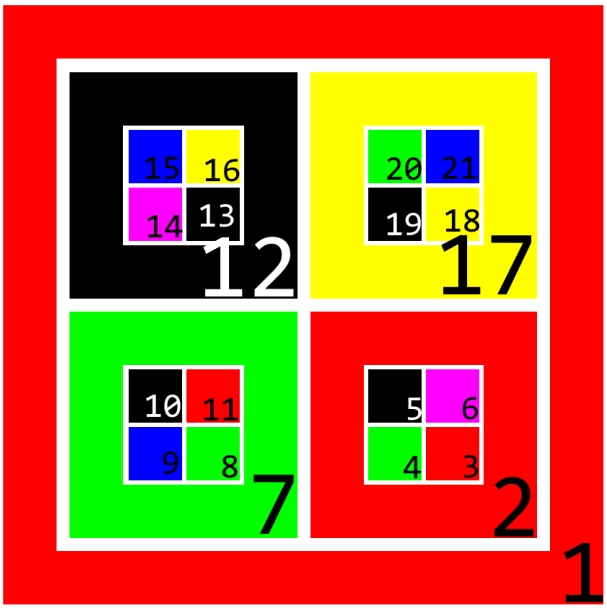
Fonte: Próprio autor.

Figura 5.44 – Código-fonte: exemplo de armazenamento de marcador de 1 nível.

```
static ColorBWRGB DICT_CRFM_LVL_1[][5] =
{
   {
      ColorBWRGB::GREEN,      // 01
      ColorBWRGB::GREEN,      // 02
      ColorBWRGB::BLUE,       // 03
      ColorBWRGB::BLACK,      // 04
      ColorBWRGB::RED         // 05
   }
    // restante dos marcadores são armazenados
    // nos próximos elementos do array
}
```

Fonte: Próprio autor.

Figura 5.45 – Ordem de leitura para armazenamento de marcadores de dois niveis.



Fonte: Próprio autor.

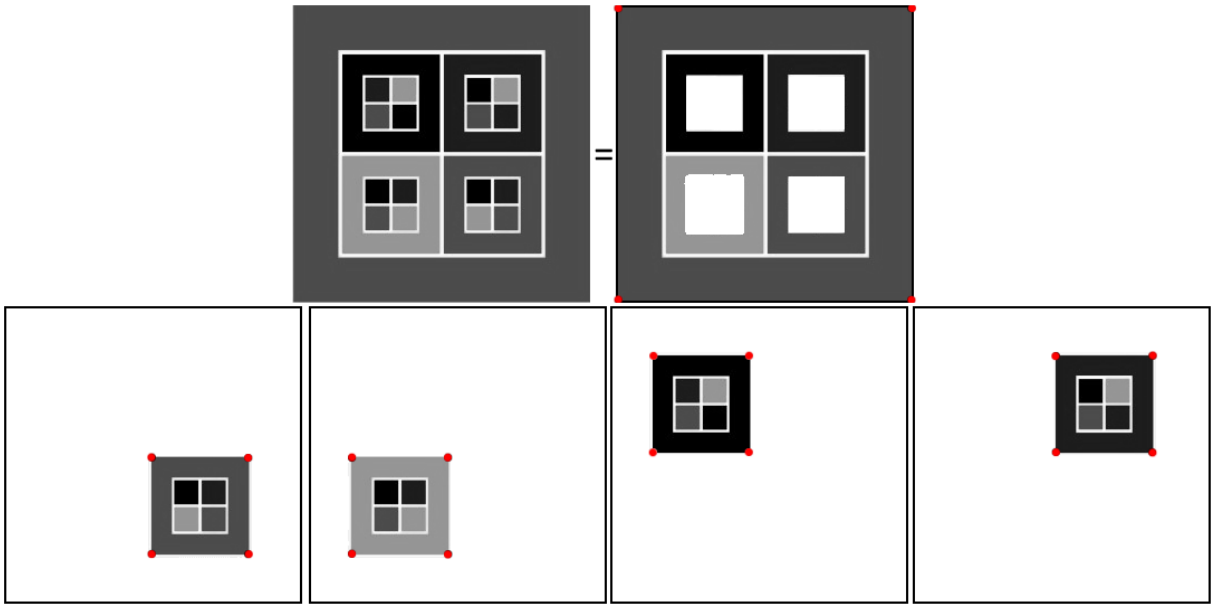
Figura 5.46 – Código-fonte: exemplo de armazenamento de marcador de 2 níveis

```
static ColorBWRGB DICT_CRFM_LVL_2[][21] =
{
   {
      ColorBWRGB::RED,        // 01
      ColorBWRGB::RED,        // 02
      ColorBWRGB::RED,        // 03
      ColorBWRGB::GREEN,      // 04
      ColorBWRGB::BLACK,      // 05
      ColorBWRGB::MAGENTA,    // 06
      ColorBWRGB::GREEN,      // 07
      ColorBWRGB::GREEN,      // 08
      ColorBWRGB::BLUE,       // 09
      ColorBWRGB::BLACK,      // 10
      ColorBWRGB::RED,        // 11
      ColorBWRGB::BLACK,      // 12
      ColorBWRGB::BLACK,      // 13
      ColorBWRGB::MAGENTA,    // 14
      ColorBWRGB::BLUE,       // 15
      ColorBWRGB::YELLOW,     // 16
      ColorBWRGB::YELLOW,     // 17
      ColorBWRGB::YELLOW,     // 18
      ColorBWRGB::BLACK,      // 19
      ColorBWRGB::GREEN,      // 20
      ColorBWRGB::BLUE        // 21
   }
    // restante dos marcadores são armazenados
    // nos próximos elementos do array
}
```

Fonte: Próprio autor.

Para os diferentes níveis de marcadores, foi criado também um modelo contendo a posição referente dos vértices de cada hierarquia de um marcador. A Figura 5.47 demonstra a fragmentação de um marcador de 2 níveis em 5 hierarquias, e as posições armazenadas de suas hierarquias em vermelho. Este modelo é utilizado para correspondência da posição (Figura 5.48) de hierarquias detectadas em imagens, afim de calcular sua estimativa de pose.

Figura 5.47 – Posições das hierarquias de um marcador de 2 níveis.



Fonte: Próprio autor.

Figura 5.48 – Código-fonte: dicionário de posições das hierarquias de um marcador de 2 níveis.

```
const Point MODEL_REF_LVL_2[][4] =
{
    // WHOLE MARKER
    { Point(000),       Point(677),         Point(677, 677),    Point(000, 677) },

    // BOTTOM RIGHT
    { Point(343, 343),  Point(565, 343),    Point(565, 565),    Point(343, 565) },

    // BOTTOM LEFT
    { Point(112, 343),  Point(333, 343),    Point(333, 565),   Point(112, 565) },

    // TOP LEFT
    { Point(112, 112),  Point(333, 112),   Point(333, 333),   Point(112, 333) },

    // TOP RIGHT
    { Point(344, 112),  Point(565, 112),    Point(565, 333),    Point(344, 333) }
};
```

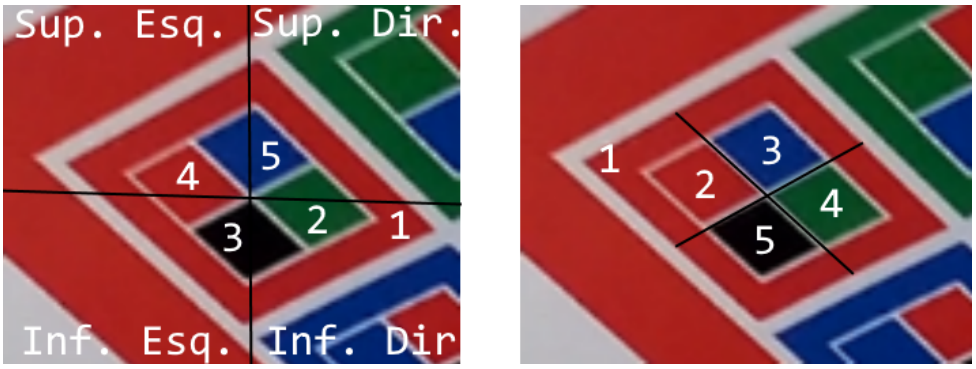
Fonte: Próprio autor.

5.3.2 Ordenação da sequência de cores pelo bloco de orientação

Após determinar a cor final de todos os quadriláteros detectados na imagem, e organiza-los em hierarquias, é preciso organiza-los de forma que se identifique a ordem da sequência de cores de cada hierarquia. Como um marcador pode ser detectado a partir de diferentes ângulos de visualização, é necessário então, determinar a orientação correta de cada hierarquia.

Até o passo atual, os blocos internos de uma hierarquia são ordenados em relação a sua posição global em relação a imagem (Figura 5.49 esquerda), sem levar em conta o bloco de orientação. É necessário ordena-los em relação ao bloco de orientação da hierarquia (Figura 5.49 direita).

Figura 5.49 – Blocos internos organizados em relação a sua posição global da imagem (esquerda). Blocos internos organizados em relação ao bloco de orientação da hierarquia (direita).



Fonte: Próprio autor.

Para a ordenação correta da sequência dos blocos de uma hierarquia, o índice dos blocos filhos são alterados de forma cíclica, até que o primeiro elemento da lista seja o de mesma cor que o bloco principal da hierarquia. Após, é inserido a cor do bloco principal no começo da lista, e armazenado o número de alterações para uso posterior (Figura 5.50).

Figura 5.50 – Código-fonte: ordenação de sequência de cores dos blocos internos.

```
1  inline void Quadrilateral::setOrder(){
2      // Alteração cíclica: último elemento se torna o primeiro
3      // Ex: 1032 -> 2103 -> 3210
4      while (vecCoresFilhos[0] != this->color) {
5        vecCoresFilhos.insert(vecCoresFilhos.begin(), vecCoresFilhos[3]);
6        vecCoresFilhos.pop_back();
7        this->qtdIteracoes++;
8     }
9     vecCoresFilhos.insert(vecCoresFilhos.begin(), this->color);
10 }
```

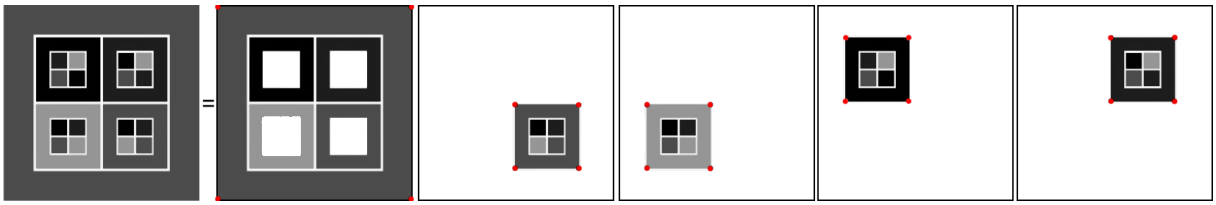
Fonte: Próprio autor.

5.3.3 Correspondência dos marcadores-alvo

Após estabelecer a sequência de cores das hierarquias detectadas, é feita a correspondência de marcadores, entre um dicionário-alvo, e as sequências de cores detectadas (Figura 5.54).

Este processo tem início a partir da seleção de um marcador contido no dicionário. Após, é feita a fragmentação do marcador em suas diferentes hierarquias (Figura 5.51). Cada uma dessas hierarquias-alvo correspondem a uma sequência de cores e posições (vértices do bloco principal da hierarquia) em relação ao marcador alvo selecionado.

Figura 5.51 – Fragmentação de um marcador-alvo conforme suas hierarquias.



Fonte: Próprio autor.

Após, para cada hierarquia-alvo gerada, é verificado sua possível correspondência com uma das hierarquias detectadas (Figura 5.52).

Figura 5.52 – Código-fonte: fragmentação do marcador alvo.

```
switch (dictionary)
{
    // Para um marcador de DOIS niveis, são gerados 5 hierarquias-alvo
    case MarkerDictionary::LVL_2:
        // Pega sequencia de cores (vecCoresAlvo)
        // da primeira hierarquia-alvo (Position::WholeMarker)
        SequenceAtPosition(dictionary, markerId, Position::WholeMarker, vecCoresAlvo);

        // Procura e/ou gera correspondências entre hierarquia-alvo (vecCoresAlvo)
        // e as hierarquias detectadas (vecQuads)
        GeraCorrespondencias(vecCoresAlvo, Position::WholeMarker, vecQuads);

        // Repete para as próximas hierarquias-alvo
        SequenceAtPosition(dictionary, markerId, Position::BottomRight, vecCoresAlvo);
        GeraCorrespondencias(vecCoresAlvo, Position::BottomRight, vecQuads);
        SequenceAtPosition(dictionary, markerId, Position::BottomLeft, vecCoresAlvo);
        GeraCorrespondencias(vecCoresAlvo, Position::BottomLeft, vecQuads);
        SequenceAtPosition(dictionary, markerId, Position::TopLeft, vecCoresAlvo);
        GeraCorrespondencias(vecCoresAlvo, Position::TopLeft, vecQuads);
        SequenceAtPosition(dictionary, markerId, Position::TopRight, vecCoresAlvo);
        GeraCorrespondencias(vecCoresAlvo, Position::TopRight, vecQuads);
        break;
}
```

Fonte: Próprio autor.

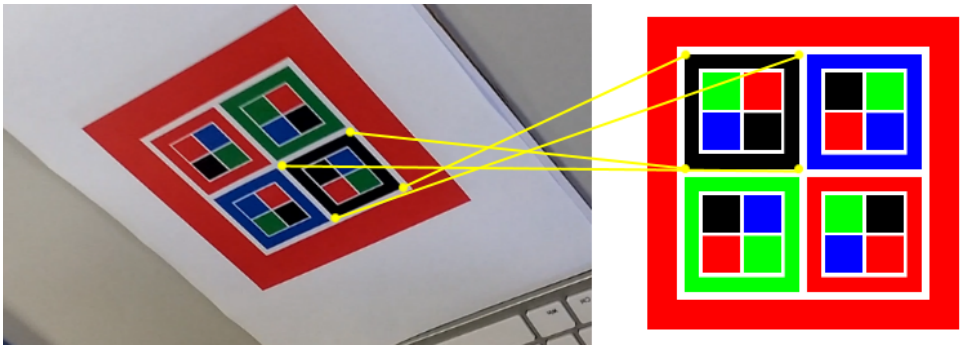
Para cada correspondência de sequência de cores encontrada, é feita também a correspondência dos vértices do quadrilátero detectado (image_points), com os vértices do modelo de marcador previamente armazenado (model_points) (Figura 5.53).

Figura 5.53 – Código-fonte: correspondência de sequência de cores e vértices.

```
inline void GeraCorrespondencias(const vector<ColorBWRGB>& vecCoresAlvo,
    Position pos, const vector<Quadrilateral*>& vecQuads){
   for (unsigned int i = 0; i < vecQuads.size(); i++)
      if (vecCoresAlvo[0] == vecQuads[i]->vecCoresFilhos[0] &&
         vecCoresAlvo[1] == vecQuads[i]->vecCoresFilhos[1] &&
         vecCoresAlvo[2] == vecQuads[i]->vecCoresFilhos[2] &&
         vecCoresAlvo[3] == vecQuads[i]->vecCoresFilhos[3] &&
         vecCoresAlvo[4] == vecQuads[i]->vecCoresFilhos[4])
      {
         image_points.push_back(
             vecQuads[i]->vertices[(0 + vecQuads[i]->nodoOrientacaoHierarquia) % 4]);
         image_points.push_back(
             vecQuads[i]->vertices[(1 + vecQuads[i]->nodoOrientacaoHierarquia) % 4]);
         image_points.push_back(
             vecQuads[i]->vertices[(2 + vecQuads[i]->nodoOrientacaoHierarquia) % 4]);
         image_points.push_back(
             vecQuads[i]->vertices[(3 + vecQuads[i]->nodoOrientacaoHierarquia) % 4]);

         int offSet = 0;
         if ((int)pos == 1)
            offSet = 1;
         if ((int)pos >= 1)
            offSet = (((int)pos - 1) * 5) + 1;

         model_points.push_back(MODEL_REF_LVL_2[offSet][0]);
         model_points.push_back(MODEL_REF_LVL_2[offSet][1]);
         model_points.push_back(MODEL_REF_LVL_2[offSet][2]);
         model_points.push_back(MODEL_REF_LVL_2[offSet][3]);
      }
}
```

Fonte: Próprio autor.

Figura 5.54 – Exemplo de correspondência de posição de vértices.

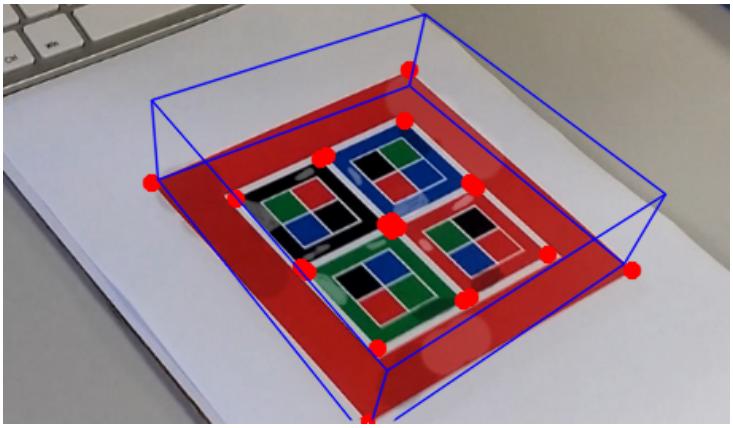


Fonte: Próprio autor.

5.4 ESTIMATIVA DE POSE E PROJEÇÃO

Após a realização das correspondências, é possível iniciar o processo de projeção (Figura 5.55) a partir dos dados filtrados pelos processos anteriores.

Figura 5.55 – Projeção de um prisma quadrangular sobre o marcador detectado. Pontos vermelhos representam as hierarquias detectadas.



Fonte: Próprio autor.

A primeira etapa para a projeção envolve a utilização do algoritmo Levenberg-Marquardt (MARQUARDT, 1963) (LEVENBERG, 1944) para geração dos vetores de rotação e translação. No exemplo gerado (Figura 5.56), são utilizados parâmetros para calibração de câmera nulos, como a matriz da câmera (comprimento focal e centro óptico) e coeficientes de distorção, para reprodutibilidade em outros equipamentos.

A partir dos vértices de correspondência entre a imagem capturada (image_points) e o modelo (model_points), é possível calcular os vetores de rotação e translação, para a projeção de um objeto.

Figura 5.56 – Código-fonte: estimativa de pose dos pontos detectados.

```
1  // Parâmetros de projeção
2  Mat camera_matrix = (cv::Mat_<double>(3, 3) << src.cols, 0, src.cols / 2, 0,
3      src.cols, src.rows / 2, 0, 0, 1);
4  Mat dist_coeffs = cv::Mat::zeros(4, 1, cv::DataType<double>::type);
5  Mat rotation_vector; // será preenchido pelo SolveProjection
6  Mat translation_vector; // será preenchido pelo SolveProjection
7
8  // Levenberg-Marquardt
9  solvePnP(model_points, image_points, camera_matrix, dist_coeffs,
10     rotation_vector, translation_vector);
```

Fonte: Próprio autor.

Após este processo, é então mapeada as coordenadas do objeto a ser projetado. Estas coordenadas são recalculadas de acordo com os vetores de rotação e translação

gerados, e projetadas na imagem com sua perspectiva corrigida (Figura 5.57).

Figura 5.57 – Código-fonte: projeção de um prisma quadrangular sobre o marcador.

```
inline void ProjectAndPaint(cv::Mat & img, cv::Mat &rotation_vector,
    cv::Mat &translation_vector, cv::Mat &camera_matrix, cv::Mat &dist_coeffs){
   if (image_points.size() == 0)
      return;
    // Imprime um circulo nos vértices da hierarquia detectada
   for (unsigned int i = 0; i < image_points.size(); i++)
      circle(img, image_points[i], 3, Scalar(0, 0, 255), 3, LINE_AA);
   // Coordenadas 3d do prisma
   vector<Point3d> rectangle_end_point3D;
   vector<Point2d> rectangle_end_point2D;

   rectangle_end_point3D.push_back(Point3d(0, 0, 0));
   rectangle_end_point3D.push_back(Point3d(677, 0, 0));
   rectangle_end_point3D.push_back(Point3d(677, 677, 0));
   rectangle_end_point3D.push_back(Point3d(0, 677, 0));
   rectangle_end_point3D.push_back(Point3d(0, 0, -200));
   rectangle_end_point3D.push_back(Point3d(677, 0, -200));
   rectangle_end_point3D.push_back(Point3d(677, 677, -200));
   rectangle_end_point3D.push_back(Point3d(0, 677, -200));
   // Calcula as novas coodenadas do prisma
   projectPoints(rectangle_end_point3D, rotation_vector, translation_vector,
       camera_matrix, dist_coeffs, rectangle_end_point2D);

   // Imprime as linhas ligando os vértices do prisma
   line(img, rectangle_end_point2D[0], rectangle_end_point2D[1],
       cv::Scalar(255, 0, 0), 1, LINE_AA);
   line(img, rectangle_end_point2D[1], rectangle_end_point2D[2],
       cv::Scalar(255, 0, 0), 1, LINE_AA);
   line(img, rectangle_end_point2D[2], rectangle_end_point2D[3],
       cv::Scalar(255, 0, 0), 1, LINE_AA);
   line(img, rectangle_end_point2D[3], rectangle_end_point2D[0],
       cv::Scalar(255, 0, 0), 1, LINE_AA);

   line(img, rectangle_end_point2D[4], rectangle_end_point2D[5],
       cv::Scalar(255, 0, 0), 1, LINE_AA);
   line(img, rectangle_end_point2D[5], rectangle_end_point2D[6],
       cv::Scalar(255, 0, 0), 1, LINE_AA);
   line(img, rectangle_end_point2D[6], rectangle_end_point2D[7],
       cv::Scalar(255, 0, 0), 1, LINE_AA);
   line(img, rectangle_end_point2D[7], rectangle_end_point2D[4],
       cv::Scalar(255, 0, 0), 1, LINE_AA);

   line(img, rectangle_end_point2D[0], rectangle_end_point2D[4],
       cv::Scalar(255, 0, 0), 1, LINE_AA);
   line(img, rectangle_end_point2D[1], rectangle_end_point2D[5],
       cv::Scalar(255, 0, 0), 1, LINE_AA);
   line(img, rectangle_end_point2D[2], rectangle_end_point2D[6],
       cv::Scalar(255, 0, 0), 1, LINE_AA);
   line(img, rectangle_end_point2D[3], rectangle_end_point2D[7],
       cv::Scalar(255, 0, 0), 1, LINE_AA);
}
```

Fonte: Próprio autor.

6 RESULTADOS

Nas seções sequentes deste capítulo, são analisados os testes e resultados de performance e detecção de marcadores realizados com o algoritmo de detecção CRFM.

Os testes deste capítulo foram divididos em três principais seções, descritas a seguir.

Na Seção 6.3 busca-se analisar os diferentes aspectos e processos que afetam a performance do algoritmo CRFM, tais como performance por processo e sub-processos, afim de verificar os processos que mais afetam o algoritmo em termos de performance, além de comparativos de performance com outros métodos de detecção considerados para a implementação do algoritmo.

Na Seção 6.5 é realizado e analisado um comparativo de performance com o marcador e algoritmo de detecção ArUco, demonstrando seus resultados de performance em milissegundos (ms) utilizando um data-set de 32 imagens para cada marcador.

E por fim, na Seção 6.6 é realizado um comparativo de precisão de detecção, entre o marcador CRFM e ArUco. São demonstrados os resultados da detecção do data-set, conforme a distância e ângulo de rotação do marcador.

6.1 HARDWARE UTILIZADO PARA A REALIZAÇÃO DE TESTES

A seguir, são apresentadas as configurações do computador e aparelhos utilizados para a realização dos testes:

- Configurações do computador:

  Laptop Dell Inspiron 7359 2-in-1;

  Processador: Intel® Core™ i7-6500U;

  Memória: 8GB RAM DDR3 1600mhz;

  Armazenamento: Samsung 850 PRO 256GB;

  Placa de Vídeo: Intel® HD Graphics 520;

  Sistema Operacional: Windows 10 Education Versão 1709.

- Aparelhos utilizados para a captura de imagens:

  Webcam Logitech C920 HD Pro;

  LG G3 D855.

6.2 METODOLOGIA DE TESTES

Os testes foram realizados utilizando ambiente de programação Visual Studio 2017, aplicativos compilados utilizando a biblioteca OpenCV versão 3.3.0, e compilador Visual C++ 12.0 (Visual C++ 2013).

Cada teste foi realizado 50 vezes, ignorando-se os resultados das primeiras 25 iterações, e levando em consideração apenas os valores médios, mínimos e máximos das 25 últimas iterações. Esta técnica foi utilizada afim de rejeitar a variação inicial de performance presente nas primeiras iterações, conforme valores determinados empiricamente.

Nos testes foram utilizados marcadores CRFM de 2 níveis, e marcadores ArUco Board de tamanhos 3x3 e 4x4 impressos em folhas A4 comuns.

6.3 ANÁLISE DE PERFORMANCE DO ALGORITMO CRFM

Nesta seção é analisada de maneira geral a performance do algoritmo, e os principais processos que afetam seu tempo de processamento. Afim de analisar o escalonamento do algoritmo em diversas resoluções, foi utiliza a imagem capturada conforme Figura 6.1, nas resoluções de 1280x720, 1920x1080 e 3840x2160 (resolução original). A imagem foi preparada afim conter diversos tipos de formas geométricas que possam afetar a performance do algoritmo de maneiras diversas.

Figura 6.1 – Imagem utilizada para testes (captura feita com LG G3 D855).

Fonte: Próprio autor.

O Quadro 6.1 demonstra a performance do algoritmo a partir de diferentes resoluções. Um alvo de performance (ms) desejável para interação em tempo real conforme experimentos empíricos, demonstra ser em torno do valor de 33.3ms, ou 30 quadros por segundo. Este valor foi atingido pelo algoritmo nas resoluções de 1920x1080 e 1280x720, enquanto que na resolução de 3840x2160 (Ultra HD) o algoritmo apresenta uma baixa performance, de 12,7 quadros por segundo.

Quadro 6.1 – Tempo médio (ms) do algoritmo de detecção CRFM.

	1280x720	**1920x1080**	**3840x2160**
Tempo (ms)	22.872	33.883	78.698

Fonte: Próprio autor.

Quadro 6.2 apresenta os valores de performance no formato de quadros por segundo (FPS).

Quadro 6.2 – Taxa de quadros por segundo (FPS) do algoritmo de detecção CRFM.

	1280x720	**1920x1080**	**3840x2160**
Taxa de Quadros por segundo (FPS)	43.7	29.5	12.7

Fonte: Próprio autor.

Conforme demonstra o Quadro 6.3, o processo de maior impacto no algoritmo é o de Filtragem de Candidatos, enquanto que os processos remanescentes (Filtragem de Cores e Estimativa de Pose) afetam de maneira muito menor a performance do algoritmo de detecção.

Quadro 6.3 – Tempo de médio (ms) por processo.

Processo	**1280x720**	**1920x1080**	**3840x2160**
Filtragem de Candidatos	18.263	29.281	69.668
Filtragem de Cores	2.099	2.253	3.220
Estimativa de pose	2.338	1.829	1.993

Fonte: Próprio autor.

6.3.1 Performance do algoritmo por sub-processos

Os Quadros 6.4, 6.5 e 6.6 demonstram a performance do algoritmo a partir de cada um de seus sub-processos.

É possível perceber que o sub-processo de maior impacto é o de binarização e filtragem de candidatos, que mesmo apesar de serem operações paralelizadas, ainda representam grande parte do processamento necessário pelo algoritmo como um todo.

Na filtragem de Cores, é possível perceber que o Cálculo de Valor Médio e de Distância, apresentam-se como uma boa alternativa a conversão para outros espaços de cores quanto a performance.

Quadro 6.4 – Tempo médio (ms) por sub-processos da Filtragem de Candidatos.

Filtragem de Candidatos	**1280x720**	**1920x1080**	**3840x2160**
Binarização e Filtragem de Contornos	9.810	18.399	59.398
Cálculo de Área e Centroide	0.506	0.564	0.562
Ordenação por Tam. de Área	4.535	6.328	6.109
Remoção de Quads. Similares	0.297	0.347	0.350
Determinação de Hierarquias.	2.812	3.452	2.613
Geração do Vetor Final	0.005925	0.005135	0.004345

Fonte: Próprio autor.

Quadro 6.5 – Tempo médio (ms) por sub-processos da Filtragem de Cores.

Filtragem de Cores	**1280x720**	**1920x1080**	**3840x2160**
Geração de Vértices de Amostragem	0.329	0.291	0.282
Cálculo do Valor Médio	0.446	0.672	1.935
Cálculo de Distância CIEDE2000	1.349	1.456	1.296
Determinação da Cor Final	0.013827	0.022913	0.012246

Fonte: Próprio autor.

Quadro 6.6 – Tempo médio (ms) por sub-processos da Estimativa de Pose.

Estimativa de Pose	**1280x720**	**1920x1080**	**3840x2160**
Organização de Seq. de Cores	0.015012	0.013827	0.011851
Correspondências de Marcadores	1.212	0.647	0.880
Projeção	1.297	1.093	1.358

Fonte: Próprio autor.

6.3.2 Análise de conversão de espaços de cores

Uma das ideias originais para a implementação do algoritmo, era a conversão do espaço de cores das imagens recebidas (geralmente capturadas em RGB) para outros espaços de cores, mais apropriados para a determinação de cores em específico.

Os Quadros 6.7, 6.8 e 6.9 demonstram a performance na conversão de imagens inteiras, conforme sua resolução, do espaço de cores RGB para Lab e HSV.

É possível perceber que estes processos resultariam em um aumento significativo na performance do algoritmo, de maneira geral, seu tempo de execução aumentaria em mais 30%.

Quadro 6.7 – Conversões de espaços de cores em resolução de 1280x720.

	Médio (ms)	**Mínimo (ms)**	**Máximo (ms)**
Conversão RGB→Lab	6.267	5.068	7.909
Conversão RGB→HSV	4.968	4.814	5.308

Fonte: Próprio autor.

Quadro 6.8 – Conversões de espaços de cores em resolução de 1920x1080.

	Médio (ms)	**Mínimo (ms)**	**Máximo (ms)**
Conversão RGB→Lab	7.425	7.416	7.449
Conversão RGB→HSV	10.186	10.043	10.501

Fonte: Próprio autor.

Quadro 6.9 – Conversões de espaços de cores em resolução de 3840x2160.

	Médio (ms)	**Mínimo (ms)**	**Máximo (ms)**
Conversão RGB→Lab	29.549725	29.293804	29.937359
Conversão RGB→HSV	39.732512	39.449253	39.830882

Fonte: Próprio autor.

6.3.3 Performance de cálculo do valor médio RGB

Os Quadros 6.10, 6.11 e 6.12 mostram como a performance seria afetada no caso de conversão das áreas analisadas para o espaço de cores Lab, e o cálculo de distância.

É possível concluir que o processo de Filtragem de Cores teria sua performance afetada de maneira significativa com a conversão de cores.

Quadro 6.10 – Tempos de conversão do Valor Médio em resolução de 1280x720.

	Médio (ms)	**Mínimo (ms)**	**Máximo (ms)**
Cálculo do Valor Médio RGB	0.793	0.645	1.353
Cálculo do Valor Médio RGB e Distância do Valor Médio	1.645	1.594	1.839
Conversão RGB→Lab, Valor Médio e Distância do Valor Médio	5.305	5.208	5.511

Fonte: Próprio autor.

Quadro 6.11 – Tempos de conversão do Valor Médio em resolução de 1920x1080.

	Médio (ms)	**Mínimo (ms)**	**Máximo (ms)**
Cálculo do Valor Médio RGB	1.399	1.379	1.436
Cálculo do Valor Médio RGB e Distância do Valor Médio	3.505	3.446	3.546
Conversão RGB→Lab, Valor Médio e Distância do Valor Médio	11.030	10.934	11.262

Fonte: Próprio autor.

Quadro 6.12 – Tempos de conversão do Valor Médio em resolução de 3840x2160.

	Médio (ms)	**Mínimo (ms)**	**Máximo (ms)**
Cálculo do Valor Médio RGB	5.579	5.456	5.737
Cálculo do Valor Médio RGB e Distância do Valor Médio	13.948	13.835	14.183
Conversão RGB→Lab, Valor Médio e Distância do Valor Médio	44.951	43.189	48.096

Fonte: Próprio autor.

6.3.4 Performance de conversão de cores em relação a rasterização

Os testes anteriores tratam da conversão da imagem como um todo. Como experimento (Quadros 6.13, 6.14 e 6.15), foi realizada a conversão dos *pixels* das imagens de maneira individual, afim de simular a performance no caso de rasterização das áreas de amostragem.

Como resultado, pode-se especular que é mais favorável realizar a conversão das imagens como um todo, e após realizar os processos de rasterização, do que realizar o processo de rasterização e conversão individual posterior dos *pixels* da imagem.

Quadro 6.13 – Tempos de rasterização em resolução de 1280x720.

	Médio (ms)	**Mínimo (ms)**	**Máximo (ms)**
Cálculo do Valor Médio RGB	0.793	0.645	1.353
Conversão RGB→Lab	6.267	5.068	7.909
Conversão RGB→Lab pixel a pixel	1424.397	1398.067	1519.081
Conversão RGB→HSV	4.968	4.814	5.308
Conversão RGB→HSV pixel a pixel	991.457	965.742	1028.974

Fonte: Próprio autor.

Quadro 6.14 – Tempos de rasterização em resolução de 1920x1080.

	Médio (ms)	**Mínimo (ms)**	**Máximo (ms)**
Cálculo do Valor Médio RGB	1.399	1.379	1.436
Conversão RGB→Lab	7.425	7.416	7.449
Conversão RGB→Lab pixel a pixel	3177.064	3160.735	3198.760
Conversão RGB→HSV	10.186	10.043	10.501
Conversão RGB→HSV pixel a pixel	1960.395	1957.240	1964.122

Fonte: Próprio autor.

Quadro 6.15 – Tempos de rasterização em resolução de 3840x2140.

	Médio (ms)	**Mínimo (ms)**	**Máximo (ms)**
Cálculo do Valor Médio RGB	5.579	5.456	5.737
Conversão RGB→Lab	29.549	29.293	29.937
Conversão RGB→Lab pixel a pixel	12556.899	12530.408	12593.084
Conversão RGB→HSV	39.732	39.449	39.830
Conversão RGB→HSV pixel a pixel	7786.871	7777.763	7802.463

Fonte: Próprio autor.

6.3.5 Performance do cálculo de distância de cores

Afim de analisar a performance do cálculo de distância de cores CIEDE2000, foram realizados comparativos de performance em relação a utilização de um valor médio por área e sua distância, com o cálculo de distância realizada sobre cada *pixel* da imagem.

Os resultados podem ser conferidos conforme os Quadros 6.16, 6.17 e 6.18.

Quadro 6.16 – Tempos do cálculo de distância de cores em resolução de 1280x720.

	Médio (ms)	**Mínimo (ms)**	**Máximo (ms)**
Cálculo do Valor Médio RGB	0.793	0.645	1.353
Cálculo do Valor Médio RGB e Distância do Valor Médio	1.645	1.594	1.839
Conversão RGB→Lab	6.267	5.068	7.909
Conversão RGB→Lab e Distância sobre cada pixel	0627.635	612.525	640.924

Fonte: Próprio autor.

Quadro 6.17 – Tempos do cálculo de distância de cores em resolução de 1920x1080.

	Médio (ms)	**Mínimo (ms)**	**Máximo (ms)**
Cálculo do Valor Médio RGB	1.399	1.379	1.436
Cálculo do Valor Médio RGB e Distância do Valor Médio	3.505	3.446	3.546
Conversão RGB→Lab	7.425	7.416	7.449
Conversão RGB→Lab e Distância sobre cada pixel	1310.709	1296.060	1361.018

Fonte: Próprio autor.

Quadro 6.18 – Tempos do cálculo de distância de cores em resolução de 3840x2160.

	Médio (ms)	**Mínimo (ms)**	**Máximo (ms)**
Cálculo do Valor Médio RGB	5.579	5.456	5.737
Cálculo do Valor Médio RGB e Distância do Valor Médio	13.948	13.835	14.183
Conversão RGB→Lab	29.549	29.293	29.937
Conversão RGB→Lab e Distância sobre cada pixel	5174.021	5.699	5180.266

Fonte: Próprio autor.

6.4 CONJUNTOS DE IMAGENS PARA TESTES DE PERFORMANCE E DETECÇÃO

As Figuras 6.2, 6.3 e 6.4 representam o conjunto de imagens utilizados para os testes sub-sequentes deste capítulo.

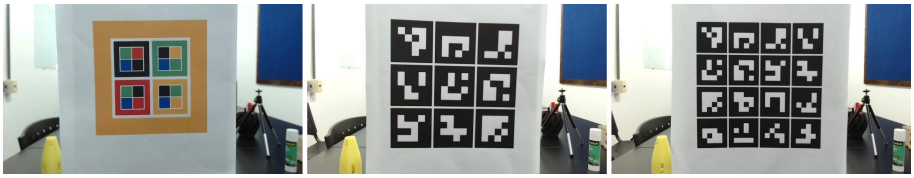
O conjunto de imagens é composto de 96 imagens. Sendo 32 imagens para cada tipo de marcador (ArUco 3x3, 4x4 e CRFM).

Foi capturada uma imagem de cada marcador, posicionados sobre as distâncias de 25, 50, 75, 100, 125, 150, 175 e 200cm. Para cada distância, foi capturada também, uma imagem do marcador rotacionado pelos ângulos de 0°, 45°, 55° e 70°.

Foram posicionados objetos de diferentes tamanhos e formatos no ambiente de testes, com o propósito de criar um cenário diverso, afim de simular diferentes situações de detecção.

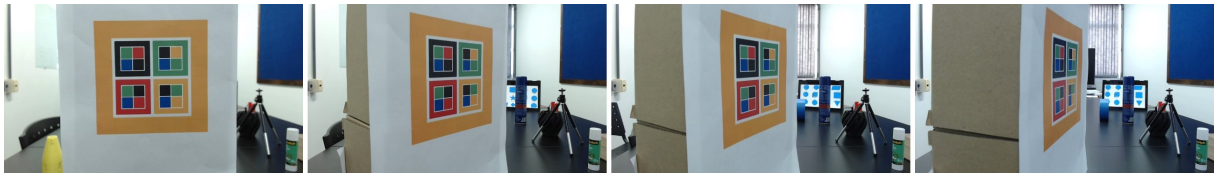
Os conjuntos de imagens e resultados podem ser conferidos na internet (TYBUSCH, 2018).

Figura 6.2 – Marcadores utilizados no *data set*.



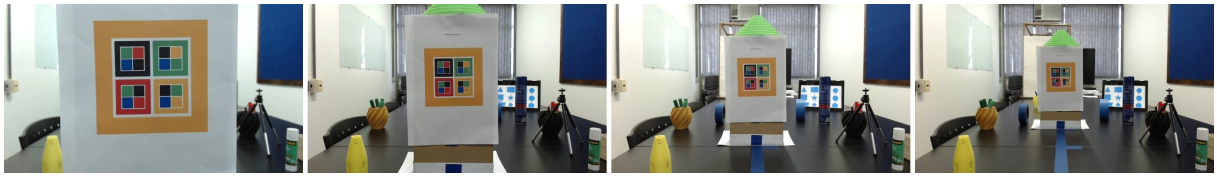
Fonte: Próprio autor.

Figura 6.3 – Amostra de imagens nos ângulos de 0°, 45°, 55° e 70°.



Fonte: Próprio autor.

Figura 6.4 – Amostra de imagens nas distâncias de 25, 50, 75 e 100cm.



Fonte: Próprio autor.

6.5 COMPARATIVO DE PERFORMANCE COM MARCADORES ARUCO E CRFM

O Quadro 6.19 exibe os tempos médios, mínimos e máximos de performance de cada algoritmo em relação ao seu conjunto de imagens capturadas. Pelos resultados, é possível verificar que o algoritmo CRFM apresenta uma performance satisfatória em relação ao algoritmo de detecção ArUco, superando-o com uma performance 30.80% mais rápida em relação ao marcador ArUco 3x3 e 37.02% em relação ao marcador ArUco 4x4.

Quadro 6.19 – Tempo de detecção médio (ms) por tipo de marcador.

Tipo de Marcador	**Médio (ms)**	**Mínimo (ms)**	**Máximo (ms)**
CRFM	23.677	19.706	29.047
ArUco 3x3	30.801	28.242	32.584
ArUco 4x4	32.442	28.996	40.334

Fonte: Próprio autor.

Também é possível perceber no Quadro 6.19, que o marcador CRFM possui uma faixa de performance mais variável em relação ao ArUco, devido a seus valores médios de máximo e mínimo variando, embora seu valor máximo nunca exceda ao valor médio do ArUco.

O motivo desta variância é devido ao processo de Filtragem de Cores, e está diretamente relacionado a distância em que o marcador está da câmera ao ser capturado. Quanto mais próximo da câmera, maior é a área de cores, e tamanho da hierarquia a ser processada, gerando esta discrepância de performance.

Estes valores podem ser conferidos conforme Quadro 6.20.

Quadro 6.20 – Tempo de detecção médio (ms) em relação à distância do marcador.

Distância	**ArUco 3X3**	**Aruco 4x4**	**CRFM**
25cm	28.790	29.664	24.767
50cm	30.428	32.694	26.501
75cm	31.127	33.215	26.439
100cm	31.910	34.385	24.513
125cm	31.439	33.874	22.473
150cm	30.969	31.773	21.679
175cm	30.982	31.748	21.555
200cm	30.764	32.187	21.489

Fonte: Próprio autor.

6.6 COMPARATIVO DE DETECÇÃO COM OUTROS ALGORITMOS

Nesta seção são analisados os resultados de detecção dos algoritmos CRFM e ArUco. Figura 6.5 exibe um exemplo de detecção e projeção CRFM efetuada com sucesso no ambiente de testes.

Figura 6.5 – Exemplo de detecção e projeção.

Fonte: Próprio autor.

O Quadro 6.21 indica as imagens que foram detectadas (✓) ou não (×), conforme o algoritmo, a distância de captura da câmera em relação ao marcador, e o ângulo em que o marcador foi posicionado.

Quadro 6.21 – Quadro indicativo da detecção e projeção em relação a distância (cm) e ângulo por tipo de marcador.

	CRFM				**ArUco 3X3**				**ArUco 4X4**			
	0°	45°	55°	70°	0°	45°	55°	70°	0°	45°	55°	70°
25cm	✓	✓	✓	×	✓	✓	✓	✓	✓	✓	✓	✓
50cm	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
75cm	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
100cm	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
125cm	✓	✓	✓	×	✓	✓	✓	✓	✓	✓	✓	×
150cm	✓	✓	✓	×	✓	✓	✓	✓	✓	✓	✓	×
175cm	✓	✓	✓	×	✓	✓	✓	×	✓	✓	✓	×
200cm	✓	✓	✓	×	✓	✓	✓	×	✓	✓	×	×

Fonte: Próprio autor.

É possível concluir que o algoritmo CRFM demonstra uma detecção satisfatória em relação ao algoritmo ArUco, obtendo valores iguais de detecção em relação ao marcador ArUco 4x4, embora não obtenha um resultado tão satisfatório em relação ao marcador ArUco 3x3.

O Quadro 6.22 exibe o percentual de detecção compilado para cada marcador.

Quadro 6.22 – Porcentagem de detecção por tipo de marcador.

Marcador	**Porcentagem de Detecção**	**Imagens detectadas**
CRFM	84.38%	27/32
ArUco 3x3	93.75%	30/32
ArUco 4x4	84.38%	27/32

Fonte: Próprio autor.

7 CONCLUSÃO

Marcadores fiduciais binários foram e continuam sendo uma área de pesquisa por diversos anos, gerando algoritmos de detecção precisos e eficientes em termos de detecção e performance. Poucas propostas de algoritmos buscaram a inclusão de cores em seus marcadores, ou a criação de estruturas recursivas.

Neste trabalho, foi apresentado um Marcador Colorido e Recursivo para Realidade Aumentada e seu algoritmo de detecção. Foram implementados com sucesso a utilização de cores no marcador, e a identificação e geração de identificadores únicos, conforme sua estrutura e sequência de cores. Também foi implementado um *layout* hierárquico que permite a detecção dos marcadores em situações de oclusão parcial.

O algoritmo CRFM apresentou performance e precisão de detecção de forma satisfatória, oferecendo uma performance até 37% mais rápida em relação ao algoritmo ArUco, e precisão similar.

Uma das principais características do algoritmo proposto, a utilização de hierarquias, foi utilizada também como processo de filtragem de quadriláteros, resultando em grande parte do ganho de desempenho do algoritmo, rejeitando candidatos que não fazem parte de uma hierarquia, fazendo com suas respectivas áreas não sejam analisadas nos processos seguintes.

Também foi demonstrada a aplicabilidade e utilização de cores em marcadores fiduciais, permitindo a geração de maiores números de marcadores, estruturas e layouts não possíveis com apenas as cores preto e branco, presente em algoritmos clássicos de detecção binária.

Apesar das vantagens do marcador CRFM, outras áreas podem ser estudadas para a evolução do algoritmo e marcadores fiduciais utilizando cores, tais como:

- Métodos para a inclusão de cores e novas técnicas para a determinação de cores;
- Melhorias no processo de detecção e filtragem de contornos;
- Busca de candidatos iniciais baseada no algoritmo proposto por ChromaTag para maior performance;
- Implementação de algoritmos binários clássicos (como ArUco e ArUco Boards) utilizando cores em seus blocos;
- Utilização de algoritmo de ordenação de melhor performance no processo de ordenação de quadriláteros por área.

REFERÊNCIAS BIBLIOGRÁFICAS

ARCSOFT. **What is Color Space**. 2016. <http://www.arcsoft.com/topics/photostudio-darkroom/what-is-color-space.html>. (Accessado em 01/02/2018). Disponível em: <http://www.arcsoft.com/topics/photostudio-darkroom/what-is-color-space.html>.

BERGAMASCO, F. et al. Rune-tag: A high accuracy fiducial marker with strong occlusion resilience. In: **Proceedings of the 2011 IEEE Conference on Computer Vision and Pattern Recognition**. Washington, DC, USA: IEEE Computer Society, 2011. (CVPR '11), p. 113–120. ISBN 978-1-4577-0394-2. Disponível em: <http://dx.doi.org/10.1109/CVPR.2011.5995544>.

BRESENHAM, J. E. Algorithm for computer control of a digital plotter. **IBM Syst. J.**, IBM Corp., Riverton, NJ, USA, v. 4, n. 1, p. 25–30, mar. 1965. ISSN 0018-8670. Disponível em: <http://dx.doi.org/10.1147/sj.41.0025>.

CANNY, J. A computational approach to edge detection. **IEEE Transactions on Pattern Analysis and Machine Intelligence**, PAMI-8, n. 6, p. 679–698, Nov 1986. ISSN 0162-8828.

CHO, Y.; NEUMANN, U. Multiring fiducial systems for scalable fiducial-tracking augmented reality. **Presence: Teleoper. Virtual Environ.**, MIT Press, Cambridge, MA, USA, v. 10, n. 6, p. 599–612, dez. 2001. ISSN 1054-7460. Disponível em: <http://dx.doi.org/10.1162/105474601753272853>.

COLORIZER. **Colorizer**. 2018. <http://colorizer.org/>. (Accessado em 01/02/2019). Disponível em: <http://colorizer.org/>.

DEGOL, J.; BRETL, T.; HOIEM, D. Chromatag: A colored marker and fast detection algorithm. In: **ICCV**. [S.l.: s.n.], 2017.

DOUGLAS, D. H.; PEUCKER, T. K. Algorithms for the reduction of the number of points required to represent a digitized line or its caricature. In: **The canadian cartographer**. [S.l.]: Cartographica: The International Journal for Geographic Information and Geovisualization, 1973.

FIALA, M. Artag, a fiducial marker system using digital techniques. In: **Proceedings of the 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05) - Volume 2 - Volume 02**. Washington, DC, USA: IEEE Computer Society, 2005. (CVPR '05), p. 590–596. ISBN 0-7695-2372-2. Disponível em: <http://dx.doi.org/10.1109/CVPR.2005.74>.

FIUMARA, G. **CIEDE2000.h**. 2015. <https://github.com/gfiumara/CIEDE2000/blob/master/CIEDE2000.h>. (Accessado em 01/02/2018). Disponível em: <https://github.com/gfiumara/CIEDE2000/blob/master/CIEDE2000.h>.

FRANK, H. **RGBcolorsolidcube.png**. 2018. <https://commons.wikimedia.org/wiki/File:RGB_Cube_Show_lowgamma_cutout_b.png>. (Accessado em 01/02/2018). Disponível em: <https://commons.wikimedia.org/wiki/File:RGB_Cube_Show_lowgamma_cutout_b.png>.

GARRIDO-JURADO, S. et al. Automatic generation and detection of highly reliable fiducial markers under occlusion. **Pattern Recogn.**, Elsevier Science Inc., New York, NY, USA, v. 47, n. 6, p. 2280–2292, jun. 2014. ISSN 0031-3203. Disponível em: <https://doi.org/10.1016/j.patcog.2014.01.005>.

GARRIDO, S. **ArUco**. 2014. <ArUcomarkerdetection(arucomodule)>. (Accessado em 01/02/2018). Disponível em: <ArUcomarkerdetection(arucomodule)>.

GONULDAS, F.; YıLMAZ, K.; OZTURK, C. The effect of repeated firings on the color change and surface roughness of dental ceramics. v. 6, p. 309–16, 08 2014.

HEROUT, A. et al. Fractal marker fields: No more scale limitations for fiduciary markers. In: **Proceedings of the 2012 IEEE International Symposium on Mixed and Augmented Reality (ISMAR)**. Washington, DC, USA: IEEE Computer Society, 2012. (ISMAR '12), p. 285–286. ISBN 978-1-4673-4660-3. Disponível em: <http://dx.doi.org/10.1109/ISMAR.2012.6402576>.

JOBLOVE, G. H.; GREENBERG, D. Color spaces for computer graphics. **SIGGRAPH Comput. Graph.**, ACM, New York, NY, USA, v. 12, n. 3, p. 20–25, ago. 1978. ISSN 0097-8930. Disponível em: <http://doi.acm.org/10.1145/965139.807362>.

KATO, H.; BILLINGHURST, M.; POUPYREV, I. Artoolkit version 2.33. **Human Interface Lab, Universidade de Washington**, 2000.

LEVENBERG, K. A method for the solution of certain non-linear problems in least squares. **Quarterly of applied mathematics**, v. 2, n. 2, p. 164–168, 1944.

LIN, W. **Real-time augmented reality based on planar homography**. 2018. <http://blog.sciencenet.cn/blog-465130-382127.html>. (Accessado em 01/02/2018). Disponível em: <http://blog.sciencenet.cn/blog-465130-382127.html>.

MARQUARDT, D. W. An algorithm for least-squares estimation of nonlinear parameters. **Journal of the Society for Industrial and Applied Mathematics**, v. 11, n. 2, p. 431–441, 1963. Disponível em: <https://doi.org/10.1137/0111030>.

MCLAREN, K. Xiii—the development of the cie 1976 (l* a* b*) uniform colour space and colour-difference formula. **Journal of the Society of Dyers and Colourists**, Blackwell Publishing Ltd, v. 92, n. 9, p. 338–341, 1976. ISSN 1478-4408. Disponível em: <http://dx.doi.org/10.1111/j.1478-4408.1976.tb03301.x>.

MCMASTER. **McMaster**. 2018. <http://wiki.cas.mcmaster.ca/index.php/File:Color.png>. (Accessado em 01/02/2018). Disponível em: <http://wiki.cas.mcmaster.ca/index.php/File:Color.png>.

MINOLTA, K. Precise color communication: Color control from perception to instrumentation. Konica Minolta, Inc, 2003.

MOKRZYCKI, W.; TATOL, M. Color difference delta e - a survey. **Machine Graphics and Vision**, v. 20, n. 4, p. 383–411, 2011.

OLSON, E. AprilTag: A robust and flexible visual fiducial system. IEEE, p. 3400–3407, May 2011.

POYNTON, C. **Digital Video and HDTV Algorithms and Interfaces**. 1. ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2003. ISBN 1558607927.

REINHARD, E. et al. **Color Imaging: Fundamentals and Applications**. 1nd. ed. [S.l.]: A K Peters, 2008. ISBN 978-1-56881-344-8.

SATTAR, J. et al. Fourier tags: Smoothly degradable fiducial markers for use in human-robot interaction. In: **Computer and Robot Vision, 2007. CRV '07. Fourth Canadian Conference on**. [S.l.: s.n.], 2007. p. 165–174.

SHARK. **HSV color solid cube**. 2018. <https://commons.wikimedia.org/wiki/File:HSV_color_solid_cube.png>. (Accessado em 01/02/2018). Disponível em: <https://commons.wikimedia.org/wiki/File:HSV_color_solid_cube.png>.

SHARMA, G.; WU, W.; DALAL, E. N. The ciede2000 color-difference formula: Implementation notes, supplementary test data, and mathematical observations. **Color Research & Application**, Wiley Subscription Services, Inc., A Wiley Company, v. 30, n. 1, p. 21–30, 2005. ISSN 1520-6378. Disponível em: <http://dx.doi.org/10.1002/col.20070>.

SOLIDWORKS. **SolidWorks**. 2018. <http://blogs.solidworks.com/solidworksblog/2013/02/augmented-reality-in-edrawings.html>. (Accessado em 01/02/2019). Disponível em: <http://blogs.solidworks.com/solidworksblog/2013/02/augmented-reality-in-edrawings.html>.

SUZUKI, S.; BE, K. Topological structural analysis of digitized binary images by border following. **Computer Vision, Graphics, and Image Processing**, v. 30, n. 1, p. 32 – 46, 1985. ISSN 0734-189X. Disponível em: <http://www.sciencedirect.com/science/article/pii/0734189X85900167>.

TATENO, K.; KITAHARA, I.; OHTA, Y. A nested marker for augmented reality. In: **ACM SIGGRAPH 2006 Sketches**. New York, NY, USA: ACM, 2006. (SIGGRAPH '06). ISBN 1-59593-364-6. Disponível em: <http://doi.acm.org/10.1145/1179849.1180039>.

TECHNOLOGIES, R. **Introduction to Augmented Reality (AR)**. 2016. <http://www.realitytechnologies.com/augmented-reality>. (Accessado em 01/02/2018). Disponível em: <http://www.realitytechnologies.com/augmented-reality>.

TYBUSCH, D. **Data-set Samples**. 2018. <https://github.com/tehort/CRFM_Data_set_2018>. (Accessado em 01/02/2018). Disponível em: <https://github.com/tehort/CRFM_Data_set_2018>.

TYBUSCH, D. et al. Color-based and recursive fiducial marker for augmented reality. In: **2017 30th SIBGRAPI Conference on Graphics, Patterns and Images (SIBGRAPI)**. [S.l.: s.n.], 2017. p. 254–261.

WAGNER, D.; SCHMALSTIEG, D. **Artoolkitplus for pose tracking on mobile devices**. [S.l.]: na, 2007.

WALTERS, A. **ChromaTags: An Accurate, Robust, and Fast Visual Fiducial System**. 2015. <https://austingwalters.com/chromatags/>. (Accessado em 01/02/2018). Disponível em: <https://austingwalters.com/chromatags/>.

WANG, E. O. J. AprilTag 2: Efficient and robust fiducial detection. October 2016.