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

Gilseone Rosa de Moraes

# CRFM LIB: UMA FERRAMENTA PARA APLICAÇÕES COM REALIDADE AUMENTADA

Santa Maria, RS
2018

**Gilseone Rosa de Moraes**

# CRFM LIB: UMA FERRAMENTA PARA APLICAÇÕES COM REALIDADE AUMENTADA

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**. Defesa realizada por videoconferência.

ORIENTADOR: Prof. Osmar Marchi dos Santos

Santa Maria, RS
2018

Moraes, Gilseone
CRFM Lib: uma ferramenta para aplicações com Realidade Aumentada / Gilseone Moraes.- 2018.
143 p.; 30 cm

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

1. Realidade Aumentada 2. Marcadores Fiduciais 3. Unity Game Engine 4. Visão Computacional 5. Processamento de Imagens I. Marchi dos Santos, Osmar 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.


Endereço: Rua Dyonélio Machado n. 524, Bairro Camobi
Fone (055) 9 9707 2111; End. Eletr.: gilseonemoraes@gmail.com

**Gilseone Rosa de Moraes**

**CRFM LIB: UMA FERRAMENTA PARA APLICAÇÕES COM REALIDADE AUMENTADA**

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:**

_____

**Osmar Marchi dos Santos, Dr. (UFSM)**
(Presidente/Orientador)

_____

**Andrei Piccinini Legg, Dr. (UFSM)**

_____

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

Santa Maria, RS
2018

## DEDICATÓRIA

*Dedico este trabalho aos meus pais Osmar Pereira de Moraes e Margarida Silva Rosa de Moraes e a minha esposa Francéli Dalberto de Moraes, por todo amor, carinho, compreensão e incentivo, tão essenciais para atingir esta conquista.*

## AGRADECIMENTOS

*Agradeço ao meu orientador, Prof. Dr. Osmar Marchi dos Santos, por seus ensinamentos, paciência, confiança e apoio em todos os momentos difíceis desta caminhada.*

*Aos professores, Dr. Andrei Piccinini Legg e Drª. Simone Regina Ceolin, pela colaboração inestimável para a realização desta pesquisa.*

*Ao professor Dr. Edison Pignaton de Freitas pela disponibilidade e dedicação na participação na banca desse trabalho.*

*Aos meus pais Osmar Pereira de Moraes e Margarida Silva Rosa de Moraes, meus avós paternos Lindolfo Pereira de Moraes e Arlinda Pereira de Moraes (In Memorian), avós maternos Sebastião Crispin da Rosa e Zeneri Tavares da Silva, pelo carinho e apoio, pois sem eles este trabalho e muitos dos outros sonhos não se realizariam.*

*A minha esposa Francéli Dalberto de Moraes pelo apoio incondicional e por estar presente em todos os momentos.*

*Aos professores do curso de Pós-Graduação em Ciência da Computação da Universidade Federal de Santa Maria, pela contribuição com minha formação.*

*Ao colega Douglas Tybusch, pelas contribuições durante o desenvolvimento da pesquisa.*

*A Universidade Federal de Santa Maria pela oportunidade, permitindo o meu aperfeiçoamento pessoal e profissional.*

*Ao CNPq (Conselho Nacional de Desenvolvimento Científico e Tecnológico) pelo apoio financeiro para realização deste trabalho.*

*A todos os amigos pelo incentivo, pelas palavras de apoio e pela compreensão à minha ausência durante os semestres do curso.*

*A todos estes, muito obrigado!*

*Só podemos ver um pouco do futuro,*
*mas o suficiente para perceber*
*que há muito a fazer*

*(Alan Turing)*

# RESUMO

## CRFM LIB: UMA FERRAMENTA PARA APLICAÇÕES COM REALIDADE AUMENTADA

AUTOR: Gilseone Rosa de Moraes
ORIENTADOR: Osmar Marchi dos Santos

A Realidade Aumentada (RA) caracteriza-se pela adição de elementos virtuais a imagens do mundo real, criando uma percepção visual de interação entre elementos virtuais e objetos reais em uma cena visual. Aplicações de Realidade Aumentada, são desenvolvidas visando apresentar formas interativas de visualização. Pesquisas usando RA, vem abrangendo as mais diversas áreas, como medicina, ensino, *design*, manutenção industrial, entretenimento, arquitetura, entre outras. Para a popularização ainda maior de aplicações que utilizam deste conceito tecnológico, são necessárias ferramentas cada vez mais acessíveis que auxiliem desenvolvedores a construírem experiências interativas, usando recursos tecnológicos comuns, como *smartphones* ou *webcams*. Em particular, esta dissertação apresenta o desenvolvimento de uma ferramenta para desenvolvimento de aplicações de Realidade Aumentada para *Unity Game Engine*, usando como alvo de referência o Marcador Fiducial Colorido e Recursivo (*CRFM*). No decorrer do trabalho são descritas as funcionalidades da ferramenta, englobando processos como Geração de Marcadores, Correspondência de Cores e ainda detecção de marcadores com projeção de elementos tridimensionais. Além disso é mostrado o desempenho da ferramenta na detecção dos marcadores CRFM e como a utilização do processo de Correspondência de Cor pode auxiliar na detecção quando diferentes tipos de iluminação são aplicados.

**Palavras-chave:** Realidade Aumentada. Marcadores Fiduciais. *Unity Game Engine*. Visão Computacional. Processamento de Imagens.

# ABSTRACT

## CRFM LIB: A TOOL FOR AUGMENTED REALITY APPLICATIONS

AUTHOR: Gilseone Rosa de Moraes
ADVISOR: Osmar Marchi dos Santos

Augmented Reality (AR) is characterized by the addition of virtual elements to real-world images, creating a visual perception of interaction between virtual elements and real objects into a visual scene. Augmented Reality applications are developed to present interactive forms of visualization. Research using AR covers the most diverse areas, such as medicine, education, design, industrial maintenance, entertainment, architecture, among others. In order to further popularize applications that use this technological concept, increasingly accessible tools are needed in order to help developers build interactive experiences, using common technology resources such as smartphones or webcams. This dissertation presents the development of a tool for developing Augmented Reality applications for the Unity Game Engine, using as a reference target, the Colored and Recursive Fiducial Marker (CRFM). In the course of this work, the tools functionalities are described, encompassing processes such as Markers Generation, Color Matching, and also markers detections and projection of three-dimensional elements. In addition, it is presented the tools performance in relation to the CRFM markers detection, and how the use of the Color Matching process can aid detection when different types of lighting are applied.



**Keywords:** Augmented Reality. Fiducial Markers. Unity Game Engine. Computer Vision. Image Processing.

## LISTA DE FIGURAS

## LISTA DE GRÁFICOS

## LISTA DE ABREVIATURAS E SIGLAS

| | |
|---|---|
| $AMD$ | *Advanced Micro Devices* |
| $API$ | *Application Programming Interface* |
| $AR$ | *Augmented Reality* |
| $ARTag$ | *Augmented Reality Tag* |
| $ARToolkit$ | *Augmented Reality Tool Kit* |
| $AV$ | *Augmented Virtuality* |
| $C\#$ | Linguagem de Programação *C#* |
| C++ | Linguagem de Programação *C++* |
| $CIE$ | *International Commission on Illumination* |
| $CIELab$ | Espaço de Cor *Lightness*, *Green–Red* (a) e *Blue–Yellow* (b) |
| $CIELCh$ | Espaço de Cor *Lightness*, *Green–Red* (a) e *Blue–Yellow* (b) |
| $CIEXYZ$ | Espaço de Cor primário baseado no valores de triestímulos *XYZ* |
| $CM$ | Unidade de Centímetros |
| $CMYK$ | Espaço de Cor *Cyan*, *Magenta*, *Yellow*, *Black* (*Key*) |
| $CRFM$ | *Marcador Fiducial Colorido e Recursivo* |
| $EUA$ | Estados Unidos da América |
| $FMF$ | *Fractal Marker Field* |
| $FPS$ | *Frames Per Second* |
| $GB$ | Unidade e *Gigabyte* |
| $Ghz$ | Unidade de *Gigahertz* |
| $HD$ | *Hard Disk* |
| $HMD$ | *Head Mounted Display* |
| $HSV$ | Espaço de Cor *Hue*, *Saturation* e *Brightness* |
| $IDE$ | *Integrated Development Environment* |
| $iOS$ | *iPhone OS* |
| $LED$ | *Light Emitting Diode* |
| $Lib$ | *Library* |

| | |
|---|---|
| $LUX$ | Unidade de intensidade de iluminação |
| $MacOS$ | *Mac Operation System* |
| $MR$ | *Mixed Reality* |
| $OpenCV$ | *Open Source Computer Vision* |
| $RA$ | Realidade Aumentada |
| $RAM$ | *Random Access Memory* |
| $RGB$ | Espaço de Cor *Red*, *Green* e *Blue* |
| $SDK$ | *Software Development Kit* |
| $SLAM$ | *Simultaneous Localization and Mapping* |
| $SSD$ | *Solid-State Drive* |
| $USB$ | *Universal Serial Bus* |
| $UWP$ | *Universal Windows Plataform* |
| $W$ | *Watts, potência de lâmpada* |
| $XYZ$ | Valores de triestímulos *XYZ* |
| $2D$ | Espaço Bidimensional |
| $3D$ | Espaço Tridimensional |

**SUMÁRIO**

# 1 INTRODUÇÃO

A Realidade Aumentada (RA) é um conceito tecnológico que possibilita a inserção de objetos virtuais no mundo real, mostrada em tempo real ao usuário da aplicação, com o apoio de algum dispositivo tecnológico, de forma que a interface do mundo real é adaptada para visualizar e interagir com objetos virtuais (Figura 1.1) (KIRNER; SISCOUTTO, 2007).

Figura 1.1 – Exemplo de aplicação de Realidade Aumentada.

Fonte: (ARTOOLKIT, 2017).

Considerando o ponto de vista de implementação, é necessário definir que a correta visualização da sobreposição do conteúdo virtual combinada com o ambiente real, pelo usuário, depende da identificação da posição e orientação de determinados elementos presentes na cena real, que irão proporcional referência para inserção posicionamento do conteúdo virtual.

A origem do termo "Realidade Aumentada" remonta a década de 90, mas desde meados dos anos 60 que aplicações que recorrem a algumas das suas características são projetadas. Apesar de ser um conceito tecnológico quem vem sendo pesquisado há pelo menos 50 anos, a quantidade de aplicações com utilidade prática durante muito tempo foi pequena, em parte devido às anteriores limitações tecnológicas e por outro lado devido aos elevados requisitos de *software* e *hardware* que estas requerem (LUO, 2011).

Os avanços da tecnologia de RA têm sido significativos e vêm abrangendo as mais diversas áreas, como medicina, ensino, *design*, manutenção industrial, entretenimento, arquitetura, entre outras (WANG et al., 2014).

A popularização ainda maior de aplicações de Realidade Aumentada, depende de ferramentas que tornem esse conceito tecnológico cada vez mais acessível e aplicável nas mais diversas áreas de conhecimento. É necessário que desenvolvedores tenham acesso a ferramentas que permitam criar experiências interativas utilizando recursos tecnológicos comuns, como *smartphones*, ou até mesmo *webcams*.

A ferramenta *CRFM Lib*, desenvolvida nesta dissertação, possibilita realizar a geração de marcadores *CRFM* com base em regras estruturais. Dicionários de dados são gerados pela ferramenta e utilizados por um algoritmo de detecção que permite projetar elementos tridimensionais sobre marcadores detectados em uma cena.

Para criar aplicações de Realidade Aumentada, *CRFM Lib* provê funcionalidades para definir marcadores que devem ser identificados, baseando-se na aplicação de métodos de Visão Computacional que identificam hierarquias dos marcadores *CRFM* através da análise de áreas de amostragem e reconhecimento de cor de cada região. Isso possibilita a obtenção de posicionamento destes marcadores para realização da projeção dos elementos tridimensionais.

Em situações onde as condições de iluminação interferem de forma negativa na identificação dos marcadores, *CRFM Lib* possibilita a realização de um processo de Correspondência de Cores que avalia as condições como cores de interesse são identificadas na cena. Isso permite que as áreas de amostragem dos marcadores *CRFM* sejam processadas considerando o tipo de iluminação existente na cena onde a aplicação de Realidade Aumentada será executada.

## 1.1 OBJETIVOS

O objetivo principal desta pesquisa é o desenvolvimento de uma ferramenta que seja integrada à *Unity Game Engine* como um *plugin*, compatível inicialmente, com a plataforma *Windows*, de forma que possibilite a criação de aplicações de Realidade Aumentada, utilizando com alvo de referência o Marcador Fiducial Colorido e Recursivo (*CRFM*) (Figura 1.2).

Figura 1.2 – Exemplo de aplicação desenvolvida com *CRFM Lib*.

Fonte: Próprio autor.

### 1.1.1 Objetivos Específicos

Para alcançar o objetivo geral desta pesquisa foi necessário o desenvolvimento das seguintes atividades:

- Realizar um estudo sobre Realidade Aumentada para explorar as possibilidades de aplicação desta tecnologia;
- Realizar estudo sobre motores gráficos de desenvolvimento de jogos, com foco em *Unity3D*;
- Realizar estudo sobre ferramentas para desenvolvimento de Realidade Aumentada, com aprofundamento em pacotes que oferecem compatibilidade com *Unity3D*;
- Realizar estudo sobre a criação de *plugins* integrados à *Unity Game Engine*;
- Realizar estudo sobre Espaços de Cores, conversão entre espaços e métodos de medição de diferença de cor;
- Levantar os requisitos necessários para desenvolvimento da ferramenta, com base nas características do marcador *CRFM*;
- Desenvolver *plugin C++* para processamento das imagens capturadas da cena e detecção dos marcadores *CRFM*;

- Desenvolver a ferramenta para desenvolvimento, integrando o *plugin C++* com a *Unity*;
- Desenvolver método para medir como as cores de interesse são percebidas na cena;
- Avaliar desempenho da ferramenta de detecção de projeção;
- Avaliar a aplicabilidade do método de Correspondência de Cores;
- Avaliar a usabilidade da ferramenta junto a participantes em uma pesquisa.

## 1.2 ESTRUTURA

A presente dissertação consiste de 8 capítulos, estruturados conforme descrito a seguir. O Capítulo 2 apresenta o referencial teórico, onde são descritos conceitos pertinentes para a realização deste trabalho, como Realidade Aumentada e Visão Computacional, abordando algumas das bibliotecas existentes e ainda um estudo sobre Sistemas de Marcadores Fiduciais.

O Capítulo 3 trata sobre ferramentas de desenvolvimento para Realidade Aumentada, conceituando e listando as principais *engines* para desenvolvimento de *games*, onde é possível criar aplicações de RA. Também neste capítulo é mostrado um estudo sobre as principais bibliotecas de Realidade Aumentada presentes no mercado atualmente. No Capítulo 4 é mostrado um estudo sobre cores, seus padrões e operações, abordando espaços de cores, conversões entre espaços e fórmulas para medição de diferença entre cores.

O Capítulo 5 descreve o marcador fiducial *CRFM* que é utilizado como alvo de referência para a ferramenta. No Capítulo 6 é apresentada a proposta da ferramenta *CRFM Lib*, para desenvolvimento de aplicações para Realidade Aumentada, mostrando como é feita a sua integração com a *Unity3D* para realização da detecção dos marcadores e projeção de objetos virtuais com base no alvo descrito.

O Capítulo 7 mostra um conjunto de testes realizados, análise de desempenho da biblioteca proposta e os resultados obtidos. Por fim, no Capítulo 8 são discutidas as considerações finais e os trabalhos futuros.

# 2 REFERENCIAL TEÓRICO

Neste capítulo é apresentada uma revisão sobre temas pertinentes para o desenvolvimento desta dissertação. Inicialmente é apresentada uma revisão sobre a Realidade Aumentada e um breve histórico sobre o tema. Na Seção 2.2 são mostrados conceitos sobre Visão Computacional, bem como trabalhos marcantes para a evolução da área e na Seção 2.3 são abordados trabalhos sobre sistemas de marcadores relacionados, encontrados na literatura.

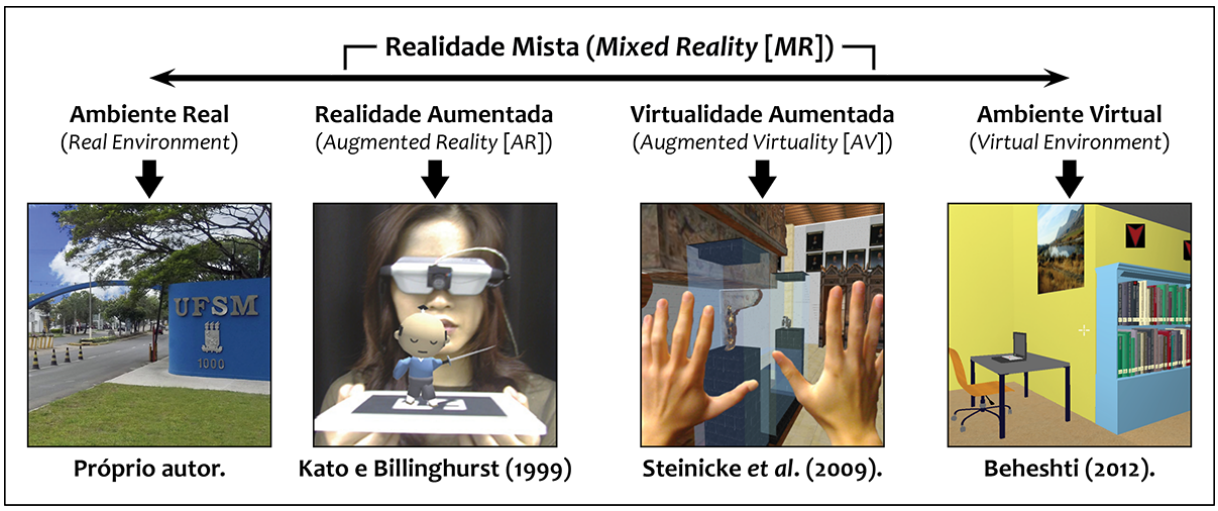
## 2.1 REALIDADE AUMENTADA

A Realidade Aumentada é um amplo campo da tecnologia, que tem buscado criar novos métodos de interação de usuários com os sistemas computacionais. Este campo tecnológico combina diversas áreas de pesquisa, onde o objetivo é mesclar a realidade física, juntamente com gráficos gerados pelo computador.

Kirner e Kirner (2008) definem RA como sendo a inserção de objetos virtuais no ambiente físico, mostrados ao usuário em tempo real, com o apoio de algum dispositivo tecnológico, usando a interface do ambiente real, adaptada para visualizar e manipular os objetos reais e virtuais. Analisando essas definições, a Realidade Aumentada permite a manipulação de elementos virtuais gerados computacionalmente, como, imagens *2D*, gráficos *3D*, sons e vídeos, em imagens de ambientes predominantemente reais.

Milgram et al. (1994), consideram a existência de um "*Continuum* de Realidade-Virtualidade" (Figura 2.1), possuindo vários estados, de forma que entre um Ambiente Real e um Ambiente Virtual podem considerar-se ambientes mistos, em que elementos virtuais se juntam ao mundo real, configurando assim a Realidade Aumentada, e também, onde ambientes virtuais inserem aspectos reais, definindo a Virtualidade Aumentada.

Figura 2.1 – Representação simplificada do *Continuum* de Realidade-Virtualidade.




Fonte: Adaptado de Milgram et al. (1994).

A Figura 2.1 demonstra a diferença entre um ambiente real (onde todos os elementos são reais), Realidade Aumentada (ambiente real aumentado por objetos virtuais, por exemplo aplicações criadas utilizando o sistema de marcadores *ARToolKit* proposto por Kato e Billinghurst (1999)), Virtualidade Aumentada (onde o ambiente virtual é aumentado por objetos reais, como o trabalho proposto por Steinicke et al. (2009)) e Ambiente Virtual (ambiente onde todos os objetos são virtuais, como exemplo pode ser citado o trabalho de Beheshti (2012), que propõe um ambiente virtual para crianças realizarem pesquisas sobre diversos assuntos).

Azuma (1997) define que sistemas de Realidade Aumentada devem possuir as três seguintes características:

1. Combina ambiente real e elementos virtuais;
2. Interatividade em tempo real;
3. O uso de recursos tridimensionais.

De acordo com Kirner e Siscoutto (2007):

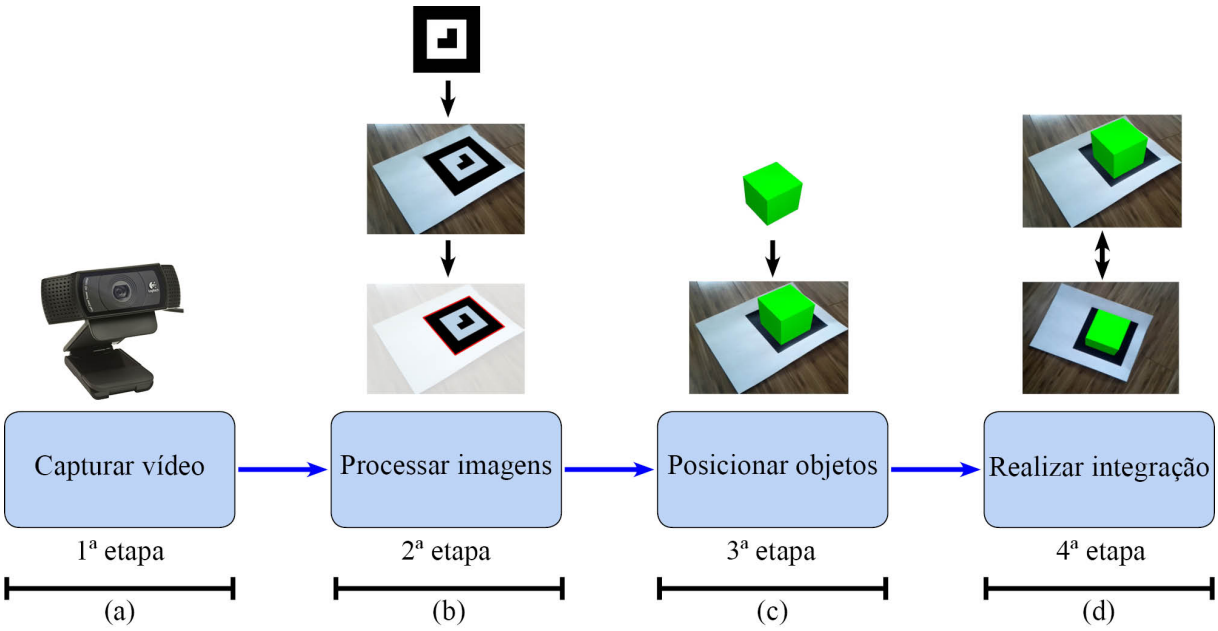
> "Realidade Aumentada é a inserção de objetos virtuais no ambiente físico, mostrada ao usuário, em tempo real, com o apoio de algum dispositivo tecnológico, usando a interface do ambiente real, adaptada para visualizar e manipular os objetos reais e virtuais".

Gravdal (2012) sugere que a Realidade Aumentada possui alta aplicabilidade, podendo ser utilizada para auxiliar usuários em muitas situações da vida real. Tais situações pode ser as mais variadas como:

- **Navegação** - Onde dicas de localização podem ser apresentadas diretamente no campo de visão do usuário;
- **Cirurgias Médicas** - Possibilitando o médico ver diretamente dentro do paciente;
- **Operação de máquinas** - Auxiliar no treinamento de usuários para operação segura de máquinas;
- **Visualização de sistemas ou estruturas dinâmicas** - Mostrar, por exemplo, como determinada estrutura histórica foi construída, com base em modelos tridimensionais, gerados computacionalmente.

Para a experiência de imersão proposta pela Realidade Aumentada, é preciso que exista a união dos objetos virtuais com o mundo real, e visualizados através da tela de algum dispositivo, tal processo pode ser dividido em quatro etapas, conforme descrito por Nickels et al. (2012):

Figura 2.2 – Processo para criação de experiência de imersão proposta pela RA.




Fonte: Adaptado de Nickels et al. (2012).

1. **Obter as informações do mundo real**: Através de dispositivos de entrada, tais como câmeras, sensores, etc (Figura 2.2 (a));
2. **Analisar o mundo real e a posição da câmera**: Neste passo, principalmente através de técnicas de processamento de imagem, coletar informações sobre o posicionamento da câmera e a localização dos elementos de referência onde deverão ser inseridos os modelos virtuais (Figura 2.2 (b));

3. **Gerar os objetos virtuais e suas devidas posições**: Realizar a geração dos objetos virtuais e posicioná-los conforme os parâmetros obtidos através da análise do mundo real e posicionamento da câmera (Figura 2.2 (c));

4. **Integrar o virtual com o real**: Finalmente tudo que foi gerado nos passos anteriores deve ser integrado com a imagem obtida pelo dispositivo de entrada e exibida no dispositivo de saída (Figura 2.2 (d)).

As quatro etapas do processo descrito por Nickels et al. (2012) podem ser visualizadas na Figura 2.2.

### 2.1.1 Histórico da Realidade Aumentada

A Realidade Aumentada é uma tecnologia que não é recente, estando disponível a pelo menos 50 anos, uma vez que surgiu gradualmente a partir da Realidade Virtual, embora o termo como é conhecido atualmente somente tenha sido cunhado no início da década de 90.

Os primeiros passos para a Realidade Aumentada foram dados no final da década de 60, quando *Ivan Sutherland*, construiu o primeiro dispositivo que misturava uma visão de mundo real com elementos 3D gerados por computador (SUTHERLAND, 1968). O dispositivo *Head Mounted Display*, ou *HMD*, criado por *Sutherland*, é considerado como um marco da tecnologia de Realidade Aumentada (Figura 2.3).

Figura 2.3 – Estrutura do *HMD* projetada por *Sutherland*.

Fonte: (SUTHERLAND, 1968)

Nas décadas seguintes, durante os anos 70 e 80, pesquisas foram realizadas pela Força Aérea dos Estados Unidos da América (EUA), como parte do projeto *Super Cockpit*

onde *Tom Furness* desenvolveu uma tela de sobreposição de imagens de alta resolução para pilotos de caça (FURNESS, 1986).

O termo Realidade Aumentada foi cunhado no início dos anos 90 pelos pesquisadores da *Boeing Caudell* e *Mizell* (CAUDELL; MIZELL, 1992). Eles desenvolveram uma configuração para um *HMD*, com o intuito de auxiliar trabalhadores em fábricas de aviões, exibindo instruções sobre onde perfurar buracos para a passagem de fios.

A partir do momento em que monitores portáteis se tornaram comercialmente disponíveis, pesquisadores começaram a olhar para a Realidade Aumentada com o intuito de buscar novas formas de aplicação e cada vez mais interativas.

## 2.2 VISÃO COMPUTACIONAL

As capacidades do sistema de visão humana inspiraram a criação e evolução da Visão Computacional, abordada inicialmente nas décadas de 60 e 70 (DAWSON-HOWE, 2014). A Visão Computacional pode ser definida como sendo o conjunto de algoritmos através dos quais sistemas baseados em computadores podem extrair informações de imagens (TRUCCO; VERRI, 1998) para a resolução de problemas a partir da análise de cenas (DAVIES, 2012), capturadas por câmeras de vídeo, sensores, scanners, entre outros dispositivos.

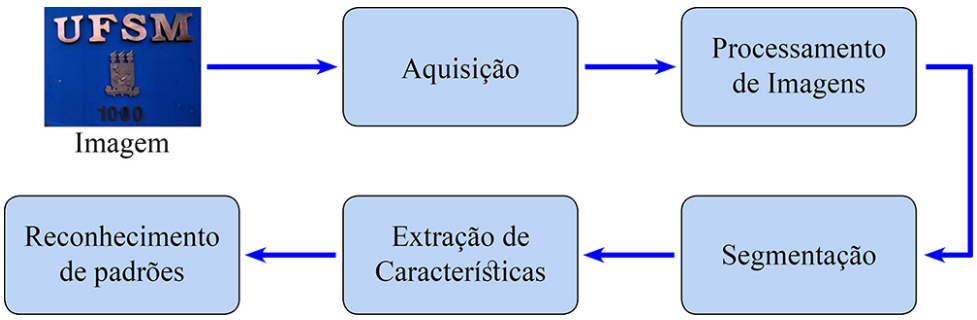
De acordo com Backes e Junior (2016), a análise das informações obtidas com os algoritmos de Visão Computacional permitem reconhecer e manipular os objetos que compõem uma imagem. Genericamente, um sistema de Visão Computacional é constituído de várias fases. São elas:

- **Aquisição** - Nesta fase é realizada a captura das imagens, ou seja, tenta simular a função dos olhos humanos. Os dispositivos que realizam esta ação podem ser *scanners*, filmadoras, máquinas fotográficas, *smartphones*, *webcams*, etc.;
- **Processamento de Imagens** - É a fase que compreende o pré-processamento, realizando operações básicas, como por exemplo rotação de imagens, até operações mais complexas onde a imagem é preparada através da filtragem de cores, aplicação de operações morfológicos, remoção de ruídos, destaque de bordas ou ainda suavização da imagem;
- **Segmentação** - Nesta fase é realizado o particionamento da imagem em regiões de interesse para análise de elementos específicos;
- **Extração de Características** - Esta fase compreende a extração do conjunto de características do objeto de interesse que é processado;

- **Reconhecimento de padrões** - Finalmente, nesta fase, a partir do processamento realizado nas fases anteriores, é realizada a classificação ou agrupamento das imagens, com base no conjunto de características obtidas.

É importante enfatizar que esta é a descrição genérica de um sistema de Visão Computacional, logo, o número de fases pode variar de acordo com o ponto de vista dos autores, bem como algumas fases podem ser supridas dependendo do problema que é proposto. Por exemplo uma imagem pode não passar pela fase de Segmentação, caso a(s) região(ões) de interesse sejam definidas manualmente. A Figura 2.4 apresenta um esquema de um sistema genérico de Visão Computacional.

Figura 2.4 – Esquema genérico de um sistema de Visão Computacional.




Fonte: Adaptado de Backes e Junior (2016).

O conceito de Visão Computacional é muitas vezes confundido com o de Processamento de Imagens, entretanto, este é caracterizado como um processo onde a entrada do sistema é uma imagem e a saída é um conjunto de valores numéricos, que podem ou não compor uma outra imagem.

Segundo Gonzalez, Woods e Eddins (2003), o espectro que vai do processamento de imagens até visão computacional pode ser dividido em três níveis, conforme a Figura 2.5.

Figura 2.5 – Divisão entre Processamento de Imagens e Visão Computacional.

| Nível | Tipo de processamento |
|---|---|
| **Baixo-nível** | Processos que envolvem operações primitivas, tais como a redução de ruído ou melhoria no contraste de uma imagem. |
| **Médio-nível** | Processos que envolvem, operações do tipo segmentação (particionamento da imagem em regiões) ou classificação (reconhecimento dos objetos na imagem). |
| **Alto-nível** | Os processos relacionados com as tarefas de cognição associadas com a visão humana. |

Fonte: Adaptado de Gonzalez, Woods e Eddins (2003).

Os algoritmos de visão computacional existentes são responsáveis por partes específicas do processo de percepção visual. Desse modo, esses algoritmos precisam ser aplicados em conjunto para formar sistemas mais completos que desempenham algum processamento definido.

### 2.2.1 Histórico da área de Visão Computacional

A evolução dos primeiros computadores, à medida que mais recursos como aumento memória e capacidade de processamento se tornaram disponíveis, proporcionou progressos gradativos na área de pesquisa de Visão Computacional (BACKES; JUNIOR, 2016). A seguir são apresentados alguns trabalhos relevantes da evolução da área:

- **Roberts (1963)** - Em sua tese defendida no *Massachusetts Institute of Technology* (*MIT*), foi proposto um dos primeiros detectores de bordas;
- **Haralick, Shanmugam e Dinstein (1973)** - Propuseram as matrizes de co-ocorrência para classificação de texturas;
- **Daugman (1980) e Daugman (1985)** - Apresentou modelos matemáticos bidimensionais para simular comportamento de campos receptivos do córtex visual;
- **Canny (1986)** - Neste trabalho propôs um algoritmo pioneiro para detecção de bordas com alta tolerância a ruídos;
- **Kass, Witkin e Terzopoulos (1988)** - Em seu trabalho propuseram os modelos de contorno ativo;
- **Mallat (1989)** - Propôs a teoria da multirresolução para análises de sinais usando *wavelets*;
- **Mumford e Shah (1989)** - Neste trabalho desenvolveram um algoritmo de segmentação de regiões por meio da minimização de um funcional;
- **Rowley, Baluja e Kanade (1998) e Viola e Jones (2004)** - Em seus trabalhos propuseram algoritmos para processamento de imagens para realização de reconhecimento facial;
- **Lowe (2004)** - Desenvolveu algoritmo para extração de características invariantes de uma imagem que permite realizar correspondência entre objetos em diferentes perspectivas.

Atualmente na área de Visão Computacional destacam-se pesquisas que envolvem algoritmos de alto desempenho para resolução de problemas específicos, como, por exemplo, reconhecimento facial, segmentação de imagens médicas, análise de formas e texturas, navegação de veículos autônomos e ainda algoritmos para rastreamento de objetos em cenas para aplicações que proporcionam experiências de imersão, como a Realidade Aumentada.

### 2.2.2 Bibliotecas de Visão computacional

Bibliotecas de Visão Computacional são usadas ativamente por um grande número de empresas e centros de pesquisa para desenvolvimento de aplicações e estudos que envolvem análise e processamento de imagens. Não seções a seguir são exemplificadas algumas bibliotecas utilizadas para aplicações de Visão Computacional.

#### *2.2.2.1 OpenCV*

*OpenCV*, indiscutivelmente, é a biblioteca de Visão Computacional mais conhecida e a mais utilizada. Seus módulos incluem centenas de funções de Visão, Aprendizado de Máquina e Processamento de Imagens prontas para serem aplicadas. Se uso é feito em aplicações tanto na academia como na indústria (GARCIA et al., 2015) (OPENCV, 2017).

A utilização da *OpenCV* pode ser feita para desenvolver aplicações que podem ser executadas em várias plataformas, como *Windows*, *Linux*, *MacOS*, *Android* e *iOS* (LAGANIERE, 2014).

A biblioteca contém tanto funções de baixo nível para Processamento de Imagens, quanto funções de alto nível para detecção de faces, detecção de pedestres, correspondência de características e rastreamento de objetos. Para o trabalho desenvolvido nesta dissertação, *OpenCV* será a biblioteca utilizada.

#### *2.2.2.2 EmguCV*

A biblioteca *EmguCV* é um *wrapper* (encapsulador) multiplataforma de *OpenCV* para processamento de imagem. Esta biblioteca permite que as funções do *OpenCV* sejam chamadas a partir linguagens compatíveis com o .NET, como *C#*, *VB*, *VC++*, *IronPython* etc. O *wrapper* pode ser compilado no Mono e executado em dispositivos *Windows*, *Linux*, *MacOS*, *iPhone*, *iPad* e *Android* (SHI, 2013).

*EmguCV* fornece uma sintaxe de código amigável para programadores *.NET*, possibilitando que aplicações de Visão Computacional, mesmo sofisticadas, sejam criadas rapidamente. As funcionalidades da biblioteca abrangem, basicamente, as funções que são englobadas pela *OpenCV* (SHI, 2013).

#### *2.2.2.3 SimpleCV*

O *SimpleCV* é uma biblioteca para Visão Computacional usando *Python*. O objetivo principal desta biblioteca é proporcionar que aplicações de Visão Computacional sejam escritas de forma fácil (DEMAAGD et al., 2012).

*SimpleCV* fornece um conjunto de ferramentas para trabalhar com as imagens ou *streaming* de vídeo que vêm de câmeras USB, *WebCams*, *Kinects* ou *smartphones*. A biblioteca está disponível para uso gratuito para as plataformas *MacOS*, *Windows* e *Ubuntu Linux*.

#### *2.2.2.4 VXL*

*VXL* é uma biblioteca de Visão Computacional escrita em *C++*, que implementa vários algoritmos comuns de visão por computador e funcionalidades relacionadas (VXL, 2013). *VXL* envolve principalmente as seguintes bibliotecas.

- ***VGL* (Biblioteca de Geometria)** - Geometria para pontos, curvas e outros objetos elementares em 1, 2 ou 3 dimensões;
- ***VIL* (Biblioteca de processamento de imagem)** - Possibilita carregar, salvar e manipular imagens em muitos formatos comuns de arquivos, incluindo imagens muito grandes;
- ***VNL* (Biblioteca Numérica)** - Recipientes e algoritmos numéricos. Por exemplo: matrizes, vetores, etc.;
- ***VSL* (Biblioteca de Transmissão)** - Entrada e saída de dados;
- ***VBL* (Biblioteca de Modelos)** - Modelos básicos de dados;
- ***VUL* (Biblioteca de utilitários)** - Utilitários da biblioteca.

Além das bibliotecas principais, existem ainda bibliotecas para realização de operações como Processamento de Imagens, Geometria de Câmera, Manipulação de Vídeo,

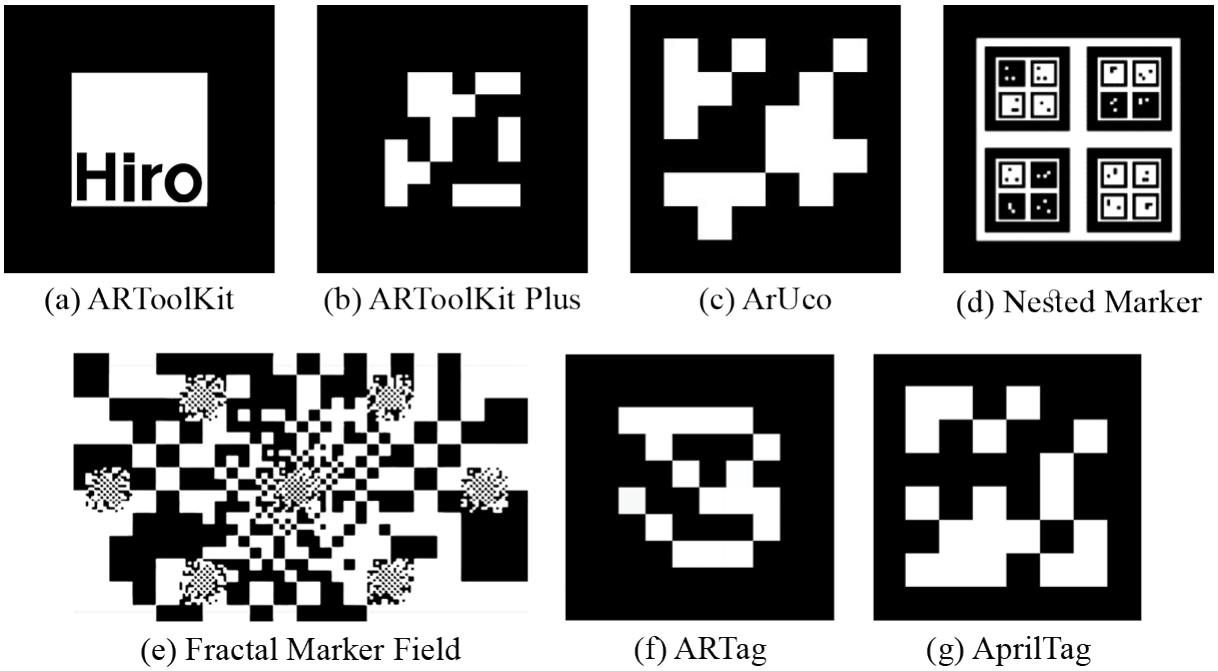
Modelagem de Probabilidade, *Design* de *GUI*, Classificação, Rastreamento de Características, Manipulação de elementos *3D*, entre outras (VXL, 2013).

## 2.3 SISTEMAS DE MARCADORES

Os marcadores utilizados como alvo em cenas de Realidade Aumentada usam principalmente formas geométricas primitivas, que são facilmente detectáveis em imagens, contribuindo para uma rápida identificação do padrão interno do marcador (Figura 2.6). Formas quadradas (KATO; BILLINGHURST, 1999) e circulares (NAIMARK; FOXLIN, 2002) geralmente servem de dica inicial para os algoritmos de identificação de marcadores.

Marcadores quadrados possibilitam que a pose da câmera seja calculada a partir da localização dos seus quatro vértices (SCHWEIGHOFER; PINZ, 2006) (KATO; BILLINGHURST, 1999), por outro lado, utilizando marcadores circulares a pose da câmera pode ser calculada utilizando o contorno do marcador (NAIMARK; FOXLIN, 2002).

Figura 2.6 – Marcadores Fiduciais propostos em trabalhos relacionados.




Fonte: *ARToolKit* (KATO; BILLINGHURST, 1999), *ARToolKitPlus* (WAGNER; SCHMALSTIEG, 2007), *ArUco* (GARRIDO-JURADO et al., 2014), *Nested Marker* (TATENO; KITAHARA; OHTA, 2006), *Fractal Marker Field* (HEROUT et al., 2012), *ARTag* (FIALA, 2005) e *AprilTag* (OLSON, 2011).

Grande parte dos sistemas de marcadores utilizam um padrão de codificação com apenas duas cores (preto e o branco), devido possuírem grande contraste, favorecendo operações de limiar de cores, permitindo maior facilidade para a detecção de contornos e o padrão de codificação interno. Para a borda, geralmente utiliza-se a cor preta, mas o layout interno depende do padrão utilizado pelo sistema de marcadores.

Muitas propostas de sistemas de marcadores fiduciais consistem em utilizar técnicas para segmentar e identificar o padrão codificado dentro das bordas quadradas de marcadores (GARRIDO-JURADO et al., 2014), utilizando algoritmos de identificação de padrão binário ou correlações de imagem. Sistemas mais clássicos, utilizam internamente um padrão binário, intercalando blocos com preto ou branco.

Mesmo esta sendo uma técnica amplamente utilizada, também existem sistemas como *ARToolKit* (KATO; BILLINGHURST, 1999) (Figura 2.6 (a)), que propõe a utilização de técnicas de correlação de imagem.

*ARToolKit* possui uma vasta utilização. Devido a este fato, vários outros sistemas de marcadores também adotam a sua solução, como é o caso de *Nested Marker* (TATENO; KITAHARA; OHTA, 2006) (Figura 2.6 (d)). *Nested Marker* possui sua estrutura disposta em camadas recursivas, onde o marcador localizado na camada superior é utilizado para localização quando posicionado distante da câmera. Na camada inferior existem quatro marcadores menores para identificação quando a câmera está mais próxima.

A abordagem de *Nested Marker* possibilita estimar a pose da câmera mesmo que algum dos submarcadores seja ocluído por algum objeto, pois a estimativa será feita a partir de algum dos outros submarcadores que permanece visível na imagem capturada.

Diferentemente do *ARToolkit* (KATO; BILLINGHURST, 1999), *ARTag* (FIALA, 2005) (Figura 2.6 (f)) usa uma técnica baseada na detecção de borda e codificação dos bits binários. O *design* do marcador é composto por um *grid* de 6x6 blocos internos que intercalam quadrados preto e branco, representando dados binários. Dos 36 bits, 10 são usados para codificação, enquanto os bits restantes são usados para redundância e detecção de erros.

O desempenho do sistema de marcadores ARTag motivou alterações no *ARToolKit*, levando assim a criação de *ARToolKit Plus* (WAGNER; SCHMALSTIEG, 2007), visando o desempenho também em dispositivos móveis.

Outra abordagem baseada em *ARTag*, o algoritmo de *AprilTag* (OLSON, 2011) (WANG; OLSON, 2016) (Figura 2.6 (g)) é também baseado em blocos quadrados intercalados nas cores preta e branca, proporcionando uma detecção robusta em condições variáveis de iluminação e oclusão, no entanto, com um número reduzido de marcadores possíveis, dependendo do esquema de codificação.

Outro sistema de marcadores, ArUco (GARRIDO-JURADO et al., 2014) (Figura 2.6 (c)), propõe métodos para identificação e tratamento de erros em marcadores fiduciais. Além da utilização de forma individual, o sistema ArUco possibilita realizar agrupamento de marcadores para geração de placas, que visam formar um sistema robusto para tratamento de oclusão.

O *Fractal Marker Field* (*FMF*) (HEROUT et al., 2012) (Figura 2.6 (e)) é um sistema de marcadores em estrutura de fractal, que busca resolver limitações dos marcadores quanto a sua escala, possibilitando liberdade de movimento de câmera. O marcador incorpora características de marcadores com códigos matriciais, como o *QRCode*, para

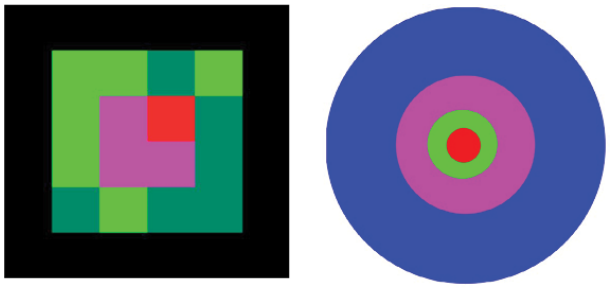
formar um grande número de escalas distintas. Para possibilitar liberdade de movimentos de câmera o *FMF* junta aos códigos matriciais um grande nível de marcadores aninhados, baseado na estrutura proposta pelo *Nested Marker* (TATENO; KITAHARA; OHTA, 2006).

Sistemas de marcadores Fiduciais como *Nested Marker* (TATENO; KITAHARA; OHTA, 2006) e *Fractal Marker Field* propõem soluções para detecção dos marcadores em diferentes distâncias de câmeras, abordando organização dos elementos em camadas recursivas. Por outro lado, ArUco (GARRIDO-JURADO et al., 2014) realiza a criação e placas de marcadores, mantendo o mesmo tamanho de cada elemento da placa e aumentando o número de marcadores na cena, para criar assim um sistema robusto para tratamento de oclusão.

### 2.3.1 Marcadores coloridos

Grande parte dos sistemas de marcadores fiduciais utilizam apenas as cores preta e branca para codificação dos seus bits internos. A escolha destas cores para definição da borda do marcador facilita o processo de identificação e rastreamento do marcador devido a facilidade de execução de operações de limiar de cor. Embora existam alguns trabalhos que exploram a utilização de cores para compor a estrutura de marcadores (Figura 2.7), este não é um assunto muito abordado.

Figura 2.7 – Propostas de marcadores que usam cores em sua estrutura.



(a) ChromaTag (b) Multiring Color

Fonte: *ChromaTag* (DEGOL; BRETL; HOIEM, 2017) e *Multiring Color* (CHO; NEUMANN, 2001).

O sistema de marcadores fiduciais *ChromaTag* (Figura 2.7 (a)) (DEGOL; BRETL; HOIEM, 2017) usa cores no seu *design*. Este sistema utiliza tons verdes e vermelhos para compor sua grade binária. O algoritmo de detecção de *ChromaTag* baseia-se principalmente em encontrar gradientes elevados entre as cores vermelha e verde analisando o canal *a* do espaço de cores *CIELab*, a fim de detectar inicialmente candidatos a marcadores em uma determinada imagem.

Apesar da utilização de cor para compor a estrutura interna do marcador, a borda é composta apenas pela cor preta, para ser processada no canal *L* (luminosidade) para uma localização precisa. A identificação do ID do marcador é obtida a partir do processamento

da informação encontrada no canal *b*.

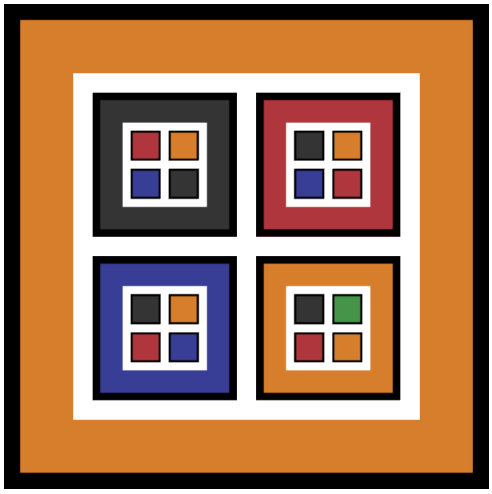
A técnica utilizada para a detecção dos marcadores *ChromaTag* provou ser bastante eficiente devido à baixa ocorrência de contrastes verdes e vermelhos na natureza, dando-lhe menos falsos positivos em candidatos em seu passo inicial. Apesar do desempenho, o algoritmo de ChromaTag não fornece um sistema robusto para a detecção de marcadores parcialmente ocluídos e devido às posições fixas das células no centro do seu design, o tamanho do dicionário de marcadores é pequeno, comparado ao AprilTag (OLSON, 2011), sistema de marcadores a qual *ChromaTag* se baseia.

Apensar do sistema de marcadores *ChromaTag* utilizar cores em sua estrutura, sua codificação ainda á baseada em bits binários, já que os canais *CIELab* são processados separadamente para reconhecimento do seu identificador.

*Multiring Color* (CHO; NEUMANN, 2001) é outro sistema de marcadores que utiliza cor em sua estrutura (Figura 2.7 (b)), possibilitando a geração de alvos de diferente níveis e cores. O primeiro nível do marcador é composto por um círculo central e um anel externo e conforme os níveis aumentam, são adicionados externamente anéis extras.

Marcadores *Multiring Color* é outro sistema que explora a distância da câmera com base nos diferentes níveis de anéis que compõem o marcador. Quando a câmera está próxima, apenas as camadas mais internas e marcadores de pequeno porte são detectados, entretanto, conforme a câmera se distancia, as camadas superiores dos marcadores são então detectadas.

Figura 2.8 – Marcador CRFM utilizado neste trabalho.



Fonte: Próprio autor.

Outro sistema de marcadores que aborda a utilização de cores em sua estrutura, que é utilizado como alvo de referência para a ferramenta *CRFM Lib* proposta, o Marcador Fiducial Colorido e Recursivo (*CRFM*) (Figura 2.8), possibilita diferentes níveis hierárquicos, dando um *design* recursivo ao marcador. *CRFM* tem sua estrutura composta por borda e blocos internos coloridos, que compõem um identificador exclusivo. A organiza-

ção hierárquica do marcador também se caracteriza como uma estratégia para explorar a detecção com diferentes distâncias de câmera. O *design* e estrutura do marcador *CRFM* serão abordados em detalhes no Capítulo 5.

## 2.4 CONSIDERAÇÕES FINAIS SOBRE O CAPÍTULO

Existem vários estudos que propõem marcadores de diferentes tipos e formatos, entretanto poucos destes estudos exploram a utilização de cores, especificamente, na implementação de sua estrutura de marcadores. O Marcador *CRFM*, diferentemente de sistemas clássicos de marcadores, possui hierarquia composta por elementos coloridos. Além disso, na literatura, marcadores hierárquicos ou recursivos não parecem ser um tópico altamente explorado.

# 3 FERRAMENTAS PARA REALIDADE AUMENTADA

Existem diversas ferramentas que auxiliam na criação de aplicações para Realidade Aumentada, proporcionando funcionalidades que agilizam processo de desenvolvimento. Este capítulo aborda as *game engines* e bibliotecas usadas para criação de jogos e também para construção de aplicações de RA. As bibliotecas listadas foram escolhidas pelo fato de que possuem versões para integração com a *Unity Game Engine*, relacionando-se assim com o foco geral deste trabalho.

## 3.1 *ENGINES* DE DESENVOLVIMENTO

O processo de desenvolvimento de aplicações envolve vários estágios. Especificamente, o desenvolvimento de aplicações para Realidade Aumentada pode receber apoio de uma base tecnológica para a sua produção, através da utilização de *game engines* para desenvolvimento, apesar de existirem outras possibilidades para produção deste tipo de aplicação.

Uma *game engine* auxilia o desenvolvedor no processo de desenvolvimento de jogos, provendo ferramentas visuais e componentes reutilizáveis (SCHWAB, 2009). Também é possível a utilização das *engines* para criação de aplicações que oferecem experiências de Realidade Aumentada. A figura 3.1 apresenta as *games engines* que serão tratados nesta seção e as principais plataformas em que são compatíveis.

Figura 3.1 – Compatibilidade das *Game Engines*.

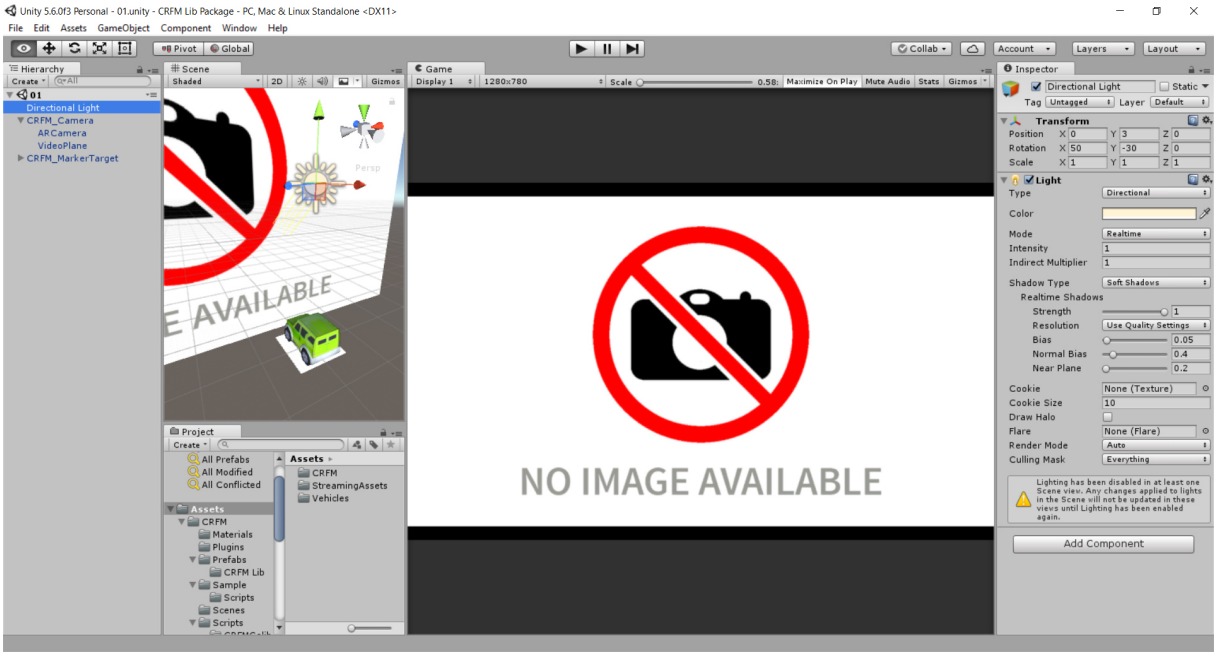
| | CryEngine | Unreal Engine | Unity3D |
|---|---|---|---|
| **Windows** | Sim | Sim | Sim |
| **GNU/Linux** | Não | Sim | Sim |
| **Android** | Não | Sim | Sim |
| **iOS** | Não | Sim | Sim |
| **XBox One** | Sim | Sim | Sim |
| **Playstation 4** | Sim | Sim | Sim |
| **Oculos Rift** | Sim | Sim | Sim |

Fonte: Próprio autor.

### 3.1.1 *Unity Game Engine*

*Unity Game Engine* (Figura 3.2), é um motor gráfico criado pela *Unity Technologies*, que possibilita o desenvolvimento de jogos, sendo possível projetar aplicações para as mais diversas plataformas (Figura 3.1). Além da portabilidade, a Unity possui uma grande quantidade de ferramentas que visam facilitar e otimizar o desenvolvimento das aplicações (BLACKMAN, 2014).

Figura 3.2 – Interface da *Unity Game Engine*.




Fonte: Próprio autor.

A programação das funcionalidades das aplicações no *Unity* é construída em *Mono*, (uma implementação *Open Source* do *.NET Framework*). Os programadores podem usar as linguagens de programação *JavaScript* ou *C#* para criar as ações do jogo (THORN, 2017).

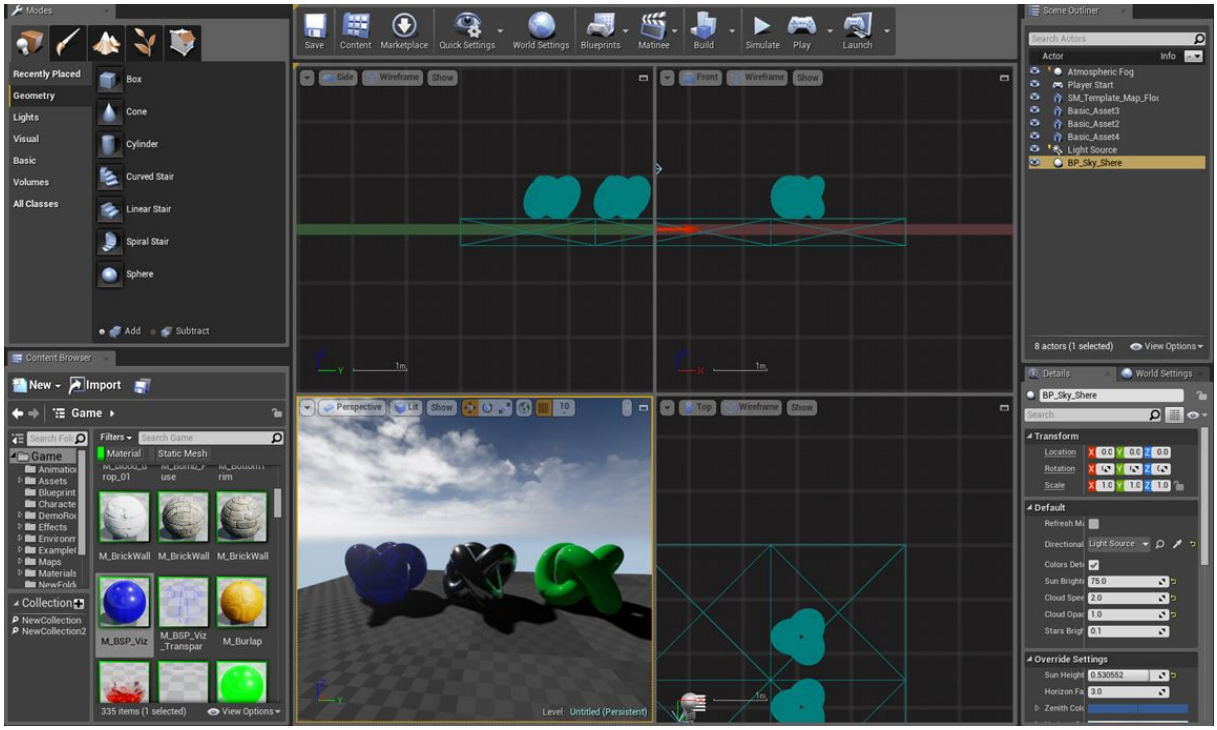
A *Unity* destaca-se com uma das principais *game engines* para desenvolvimento de jogos. A popularidade do motor gráfico está relacionada com o seu conjunto de ferramentas que auxiliam o processo de desenvolvimento. No *Unity* são encontrados recursos para criação de funções gráficas básicas até operações mais complexas para aplicação de física aos objetos, adição de trilhas sonoras e animações de personagens (JACKSON, 2014) (HOCKING, 2015).

*Unity* foi escolhida para ser a *game engine* que recebe a integração da ferramenta desenvolvida neste trabalho pela sua popularidade, facilidade de utilização e ainda pela variedade de ferramentas que fornecem agilidade durante o processo de desenvolvimento.

### 3.1.2 *Unreal Engine*

*Unreal Engine* (Figura 3.3) é um dos principais *softwares* para desenvolvimento da indústria de jogos 3D (UNREALENGINE, 2017). Em 2009 a *Epic Games*, empresa criadora da *engine*, liberou o acesso de forma gratuita às suas principais funcionalidades, disponibilizando o *Unreal Development Kit* (*UDK* da *Unreal Engine*) para qualquer desenvolvedor que quisesse utilizar o motor gráfico.

Figura 3.3 – Interface da *Unreal Engine*.



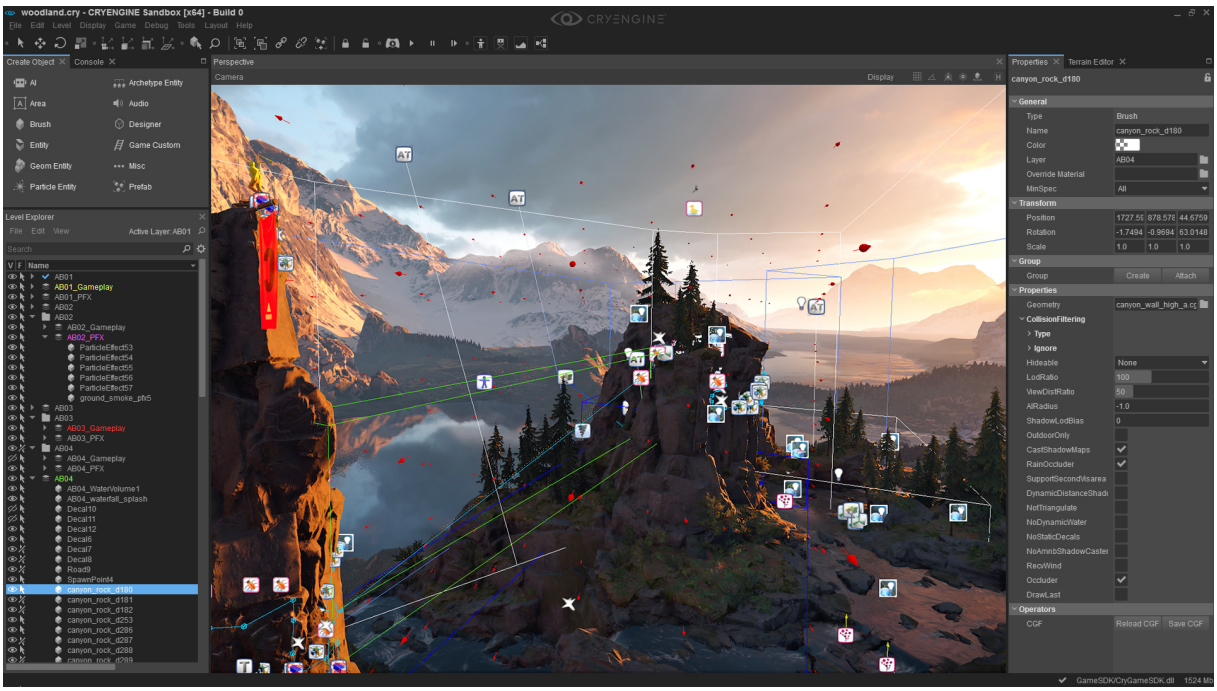

Fonte: (UNREALENGINE, 2017)

Embora a *Unreal* ter sido desenvolvida principalmente a criação de jogos em primeira pessoa, ele já foi utilizado com sucesso em projetos de vários outros gêneros. A linguagem de programação usada para escrita de código na *Unreal Engine* é *C++* e as aplicações desenvolvidas apresentam um elevado grau de portabilidade para as mais variadas plataformas (DORAN, 2015) (SHERIF; WHITTLE, 2016).

### 3.1.3 *CryENGINE*

*CryENGINE* (Figura 3.4) é um poderoso motor gráfico para o desenvolvimento de games em 3D criado pela empresa *Crytec* (CRYENGINE, 2017). A exemplo de outras *engines* de desenvolvimento, *CryENGINE* foi liberada gratuitamente para estudantes e para desenvolvedores independentes. Entretanto os projetos desenvolvidos com a ferramenta podem ser distribuídos somente se nada for cobrado pelo jogo.

Figura 3.4 – Interface da *CryENGINE*.



Fonte: (CRYENGINE, 2017)

*CryENGINE* é um dos mais poderosos motores *3D* em tempo real. O editor usado com o *CryENGINE* é conhecido como o *CryENGINE Sandbox*, e contém um conjunto completo de ferramentas e sub-editores para o desenvolvimento de jogos (GUNDLACH; MARTIN, 2014). *CryENGINE Sandbox* pode ser definido como uma ferramenta de composição de jogos que atua de forma semelhante a qualquer software de composição de vídeo digital, como *Sony Vegas*, *Adobe Premiere* e até mesmo o *Windows Movie Maker*. No entanto, ao invés de inserir clipes de vídeo e áudio, o desenvolvedor insere arte, *design* e código que eventualmente se juntarão para a criação de um jogo (TRACY; REINDELL, 2012).

## 3.2 BIBLIOTECAS DE REALIDADE AUMENTADA

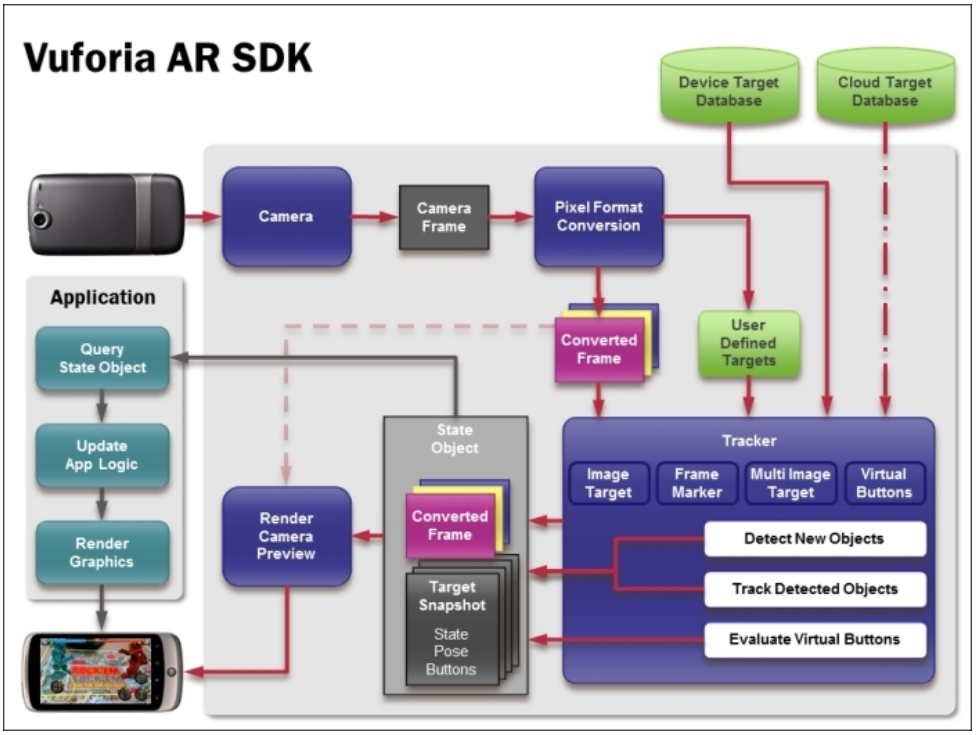
Atualmente, diversas ferramentas estão disponíveis com o objetivo de auxiliar no desenvolvimento de aplicações em Realidade Aumentada. As bibliotecas listadas nesta seção, possuem pacotes disponíveis para integração com *engines* de desenvolvimento de jogos, especificamente com a *Unity Game Engine* que é um dos focos deste trabalho.

### 3.2.1 *Vuforia SDK*

Vuforia é uma biblioteca que possibilita diferentes formas de implementação de aplicações para Realidade Aumentada, entre as quais se encontra a aquisição de imagens baseando-se em diferentes tipos alvos que podem ser imagens planas ou até objetos 3D. Inclui ainda configurações para realizar identificação e rastreamento de múltiplos alvos em uma mesma cena (VUFORIA, 2017).

Para criação de alvos rastreáveis podem ser utilizados marcadores naturais não aparentes (*markerless*), sendo possível utilizar qualquer imagem ou objeto como referência para o registro no sistema, como por exemplo rótulos de produtos, fotografias, objetos *3D*, entre outros (VUFORIA, 2017) (GRUBERT; GRASSET, 2013) (CUSHNAN; HABBAK, 2013). *Vuforia SDK* inclui facilidades que permite a integração com o *software Unity Game Engine*, fornecidas através de um pacote de ferramentas (*package*).

Figura 3.5 – Digrama com a arquitetura do *Vuforia SDK* em um ambiente de aplicação.




Fonte: (GRUBERT; GRASSET, 2013).

A arquitetura de uma aplicação de Realidade Aumentada desenvolvida usando o *Vuforia SDK* apresentada na Figura 3.5, faz uso de um dispositivo de entrada (geralmente uma câmera de vídeo) para capturar as imagens de uma cena. As imagens capturadas são processadas por um algoritmo de Visão Computacional para realizar a identificação e rastreamento de marcadores dos alvos dispostos na cena. O resultado deste processamento é armazenado em um objeto de estado, que é utilizado para a renderização do vídeo resultante com projeção do objeto virtuais tridimensionais no mundo real de acordo com o alvo de referência (GRUBERT; GRASSET, 2013) (CUSHNAN; HABBAK, 2013).

Vuforia é um *SDK* proprietário, criado pela *Qualcomm* e adquirido pela *PTC* em 2015 (PTC, 2015) (KREPS; FLETCHER; GRIFFITHS, 2016). A negociação realizada não afetou o tipo de licença como a biblioteca é distribuída, mantendo-a disponível gratuitamente para utilização de projetos. Uma limitação existente na versão gratuita de *Vuforia*, é relacionada a quantidade de reconhecimentos dos marcadores configurados. *Vuforia SDK* está disponível para as plataformas *iOS*, *Android* e *UWP* (VUFORIA, 2017).

Dentre as vantagens do *Vuforia SDK*, destacam-se:

- ***Cloud Recognition*** - Possibilita a utilização de bases de dados remotas, onde a aplicação busca os dados em tempo real;
- ***Virtual Buttons*** - A manipulação de botões virtuais com objetos reais utilizados como marcadores proporciona interatividade nas aplicações;
- ***Extended Tracking*** - Vuforia utiliza recursos do ambiente para melhorar o desempenho de rastreamento e manter a projeção do objeto mesmo quando o alvo não está mais à vista;
- ***Licença*** - Possui licenças gratuitas, mesmo para aplicações comerciais, entretanto com quantidades limitadas de identificações de alvos de referência.

### 3.2.2 *Wikitude SDK*

O conjunto de ferramentas para desenvolvimento *Wikitude SDK* possibilita o desenvolvimento de Realidade Aumentada baseada de alvos bidimensionais e objetos tridimensionais tolerantes a oclusão, mesmo quando a câmera perde o alvo que está sendo rastreado. O reconhecimento dos alvos pode ser feito tanto diretamente no dispositivo quanto em serviços na nuvem em tempo real. Outra característica da biblioteca é a possibilidade de criação de aplicações de RA baseados em geolocalização (WIKITUDE, 2017).

*Wikitude SDK* possibilita a mapeamento de ambientes em exibição de objetos virtuais livre de marcadores, baseando-se na tecnologia de *SLAM*. Esta tecnologia de Visão Computacional possibilita que um dispositivo rastreie a sua posição enquanto mapeia o ambiente para transmitir informações espaciais para o dispositivo móvel (KIM; KIM; YANG, 2008). As informações obtidas com a aplicação de *SLAM* podem ser usadas não apenas para Realidade Aumentada e Mista, mas também para Realidade Virtual.

A licença de utilização do *Wikitude* é exclusivamente comercial, possuindo diferentes versões que variam de acordo com o tipo de funcionalidades disponibilizadas. *Wikitude* é suportado nas plataformas móveis *Android* e *iOS*, oferecendo suporte para *Unity 3D* além de outros *frameworks* como *Cordova* e *Xamarin* (WIKITUDE, 2017).

### 3.2.3 *EasyAR*

*EasyAR* é suportado pelas plataformas *Android*, *iOS*, *UWP*, *Windows* e *MacOS* (EASYAR, 2017). Entre os recursos que são suportados por *EasyAR* estão a utilização de imagens como alvos de referência, percepção de ambiente, suporte de conteúdo e integração com *Unity Game Engine*.

*EasyAR*, diferentemente de outros *SDKs* utiliza um sistema de armazenamento de marcadores na nuvem para superar limitações referentes a quantidade de marcadores suportados no dispositivo que executa a aplicação RA.

A versão básica da biblioteca é gratuita para a criação de aplicações, entretanto existe a versão Pro, com recursos específicos disponíveis, como suporte a *SLAM* (Localização e Mapeamento Simultâneo), rastreamento de objetos *3D*, detecção e rastreamento simultâneo de múltiplos de alvos e ainda a gravação de tela da aplicação (EASYAR, 2017).

### 3.2.4 *Kudan SDK*

*Kudan* é uma empresa fundada em 2011 para disponibilização da biblioteca de Realidade Aumentada *Kudan SDK*. O conjunto de ferramentas é suportado apenas pelas plataformas móveis para *Android* e *iOS* possuindo, ainda um pacote de integração à *Unity3D* (KUDAN, 2017).

O rastreamento de alvos realizado pelo *Kudan* é realizado apenas com base em ferramentas locais, não possibilitando reconhecimento na nuvem. *Kudan* não define limite de quanto ao número de alvos rastreáveis podem ser utilizados, desta forma o único limite existente é o limite físico do dispositivo (KUDAN, 2017). Além disso alvos podem ser definidos em tempo de execução, proporcionando alternativas para personalização das aplicações desenvolvidas.

*Kudan SDK* possibilita é a aplicação de uma técnica de *SLAM* bastante robusta, utilizando uma única câmera.

### 3.2.5 *ARToolKit SDK*

*ARToolKit SDK* foi desenvolvido com base no sistema de marcadores proposto por (KATO; BILLINGHURST, 1999) é amplamente utilizado no desenvolvimento de aplicações de RA. Para identificação e rastreamento esta biblioteca utiliza marcadores quadrados, baseados tanto em código binário como correlação de imagens.

Até o ano de 2015, a biblioteca era mantida pela *ARToolworks*, até sua aquisição pela *DAQRI* (KREPS; FLETCHER; GRIFFITHS, 2016). Após a aquisição realizada, a

licença de *ARToolKit* foi transformada pela *Open Source*, sendo disponibilizados gratuitamente os recursos disponíveis anteriormente apenas na versão comercial da biblioteca.

A versão de *ARToolKit* para *Unity* é distribuído como um pacote (*package*) para a *engine* que integra as ferramentas de desenvolvimento de aplicações de Realidade Aumentada do *ARToolKit SDK* com as características gráficas de desenvolvimento de games do *Unity* (ARTOOLKIT, 2017). Neste pacote estão contidos o *plugin*, *scripts* e um conjunto de recursos necessários para realizar a integração do *ARToolKit* com *Unity*.

Existem limitações relacionadas com o fato de que as imagens utilizadas para alvos de rastreamento por *ARToolkit* para a experiência de RA, não serem totalmente personalizáveis, como as outras bibliotecas listados, limitado o seu uso a imagens com padrões geométricos quadrados. *ARToolKit* para *Unity* é multiplataforma possui código livre para modificações e é gratuito para uso no desenvolvimento de RA (ARTOOLKIT, 2017).

### 3.2.6 Considerações sobre as bibliotecas de Realidade Aumentada

Muitos conjuntos de ferramentas para criação de aplicações de RA estão disponíveis atualmente. Apesar do intuito destas bibliotecas ser a criação de experiências de Realidade Aumentada, cada *SDK* possui funcionalidades específicas, necessitando uma avaliação sobre as características de acordo com um conjunto de critérios:

- **Tipo de Licença** - Diferentes tipos de licenças são oferecidos pelos *SDKs*, sendo a licença comercial a mais comum, com a disponibilização de uma versão básica para realização de testes, mas este não possuindo todas as funcionalidades do *kit* completo. Existe também bibliotecas de Realidade Aumentada *Open Source*, para o qual os desenvolvedores podem contribuir e adicionar mais funções;

- **Plataformas Suportadas** - A grande maioria das bibliotecas de RA são suportadas pelas plataformas móveis mais utilizadas como *Android* e *iOS*, entretanto alguns *SDKs* também são compatíveis com *Windows*, *Universal Windows Platform (UWP)*, *MacOS* e *Linux*;

- **Suporte para Óculos Inteligentes** - Grande parte das aplicações de Realidade Aumentada funciona através de *smartphones*, onde os usuários visualizam os elementos virtuais interagindo com o ambiente real, pela tela dos seus dispositivos. Mesmo esta sendo a forma mais popular de consumo de aplicações de RA, existe uma maneira diferente de visualizar a sobreposição dos objetos virtuais no mundo real que são óculos inteligentes. Esta forma de visualização permite experiências interativa de Realidade Aumentada, sem que o usuário necessite ficar segurando o dispositivo para visualização;

- **Suporte para Unity** - Como o *Unity3D* é um dos motores de jogos mais avançados e bastante popular, ele é usado para criação de jogos para diferentes plataformas, sendo também capaz de criar aplicações de Realidade Aumentada, desta forma a compatibilidade do *SDK* com o *Unity3D* é um critério interessante a ser avaliado;

- **Reconhecimento na Nuvem** - Alguns *kits* de desenvolvimento de Realidade Aumentada oferecem suporte para reconhecimento na nuvem. Esta funcionalidade torna-se útil quando a aplicação precisa reconhecer um grande número de marcadores diferentes. Com este recurso, os marcadores são armazenados na nuvem, não ocupando espaço no dispositivo;

- **Reconhecimento Local** - Quando o objetivo é criar uma aplicação de Realidade Aumentada simples, com poucos marcadores para ser reconhecidos e rastreados, este recurso pode ser usado, desta forma não é necessário possuir conexão com a *Internet* para executar o serviço de reconhecimento dos alvos;

- **Rastreamento de objetos 3D** - Atualmente as principais plataformas de Realidade Aumentada suportam o rastreamento de objetos tridimensionais. A possibilidade de reconhecimento e rastreio de objetos *3D* como, caixas, brinquedos, veículos (entre muitos outros), amplia as possibilidades de aplicações para RA;

- **Geolocalização** - Este tipo de critério permite a criação de aplicações baseadas em localização geográfica, possibilitando adicionar pontos virtuais de interesse que podem ser identificados pela aplicação, como por exemplos, restaurantes, lojas, escolas, universidades, oficinas ou até mesmo postos de gasolina próximos geograficamente;

- **SLAM** - É uma técnica de localização e mapeamento simultâneos. Esta tecnologia permite que aplicações mapeiem um ambiente e acompanhem os próximos movimentos que serão realizados pelo dispositivo. Este tipo de funcionalidade possibilita criar mapas de navegação em ambientes internos;

- **Documentação** - Um ponto que deve ser observado é se a biblioteca possui uma documentação organizada e de fácil acesso, para auxiliar desenvolvedores a conhecer e entender funcionalidades específicas do *SDK*;

- **Marcadores Binários** - Diferentes técnicas de rastreamento de alvos são aplicadas pelos vários *kits* de desenvolvimento, mas a possibilidade de utilização de marcadores bidimensionais com padrão de blocos binários deve ser analisada, caso existam dificuldades causadas pela utilização de uma grande quantidade de alvos baseados em técnicas de correlação de imagem, que precisam ser processados utilizando técnicas custosas para identificação e rastreio destes alvos.

Com base nos critérios listados acima, na Figura 3.6 mostra uma de comparação das cinco principais bibliotecas para desenvolvimento de aplicações de Realidade Aumentada.

Figura 3.6 – Comparação entre as bibliotecas de Realidade Aumentada.

| | Vuforia | Wikitude | EasyAR | Kudan | ARToolKit |
|---|---|---|---|---|---|
| **Tipo de Licença** | *Free*, Comercial | Comercial | *Free*, Comercial | *Free*, Comercial | *Free Open Source* |
| **Plataformas Suportadas** | Android, iOS, UWP | Android, iOS | Android, iOS, UWP, macOS | Android, iOS | Android, iOS, Linux, Windows, macOS |
| **Suporte para Óculos Inteligentes** | Sim | Sim | Não | Não | Sim |
| **Suporte para Unity** | Sim | Sim | Sim | Sim | Sim |
| **Reconhecimento na nuvem** | Sim | Sim | Sim | Não | Não |
| **Reconhecimento Local** | Sim | Sim | Sim | Sim | Sim |
| **Reconhecimento de objetos 3D** | Sim | Sim | Sim | Sim | Não |
| **Geolocalização** | Não | Sim | Não | Não | Não |
| **SLAM** | Não | Sim | Sim | Sim | Não |
| **Documentação** | Excelente | Excelente | Excelente | Excelente | Excelente |
| **Marcadores Binários** | Não | Não | Não | Não | Sim |

Fonte: Próprio autor.

Os cinco kits de desenvolvimento listados são sistemas amplamente utilizados em aplicações para experiências em Realidade Aumentada. Um dos pontos em todos se destacam é pela documentação ser bastante completa auxiliando no processo de desenvolvimento.

*ARToolKit* possui código fonte aberto, possibilitando que desenvolvedores estudem a estrutura do projeto e desenvolvam melhorias para a biblioteca, já demais solução são

proprietárias. Com exceção do *Wikitude*, os outros *SDKs* oferecem versões básicas para utilização em projetos, com funcionalidades limitadas, entretanto *ARToolKit* oferece a versão completa, com todas as funcionalidades disponíveis para utilização.

Outro destaque positivo para este conjunto de bibliotecas é a possibilidade de desenvolvimento utilizando *Unity*, onde todas oferecem um pacote para integração com a *game engine*. Outro ponto que também se destaca é suporte para plataformas *mobile*, até mesmo algumas delas, como *Wikitude* e *Kudan* oferecem suporte exclusivo para estas plataformas. Para óculos inteligentes apenas *Vuforia*, *Wikitude* e *ARToolkit* oferecem compatibilidade. Em contraponto à vasta possibilidade das plataformas mobile, somente *ARToolKit* possui suporte para a plataforma *Linux* e *Windows*.

Para reconhecimento de alvos locais, todas as bibliotecas oferecem esta funcionalidade, que pode até mesmo ser considerada básica para *SDKs* de Realidade Aumentada. Entretanto, *Vuforia*, *Wikitude*, *EasyAR* e *Kudan* oferecem possibilidades para outras formas de reconhecimento de alvos. Estes quatro possibilitam reconhecimento de objetos tridimensionais, já o reconhecimento de alvos na nuvem é feito apenas por *Vuforia*, *Wikitude* e *EasyAR*.

Aplicações que utilizam localização geográfica para experiências de RA podem ser desenvolvidas utilizando apenas *Wikitude*. Marcadores fiduciais bidimensionais com padrão de blocos binários é suportado apenas por *ARToolKit*.

Dentre os *SDKs* listados, somente *Vuforia* e *ARToolKit* não utilizam a técnica de *SLAM* para mapeamento de ambiente com o movimento do dispositivo. Desta forma *Wikitude*, *EasyAR* e *Kudan* são alternativas interessantes quando a experiência de RA é baseada na movimentação do dispositivo e do usuário por um ambiente interno.

De maneira geral, as diferenças entre a ferramenta proposta neste trabalho e as ferramentas para RA listadas nesta seção, se devem ao fato de que os alvos de referência para detecção utilizados por *CRFM Lib* são compostos por um sistema de marcadores próprio, que é caracterizado por uma estrutura hierárquica composta por elementos coloridos. A ferramenta proposta oferece ainda desde a o processo de geração, bem como análise de cor conforme percebida no ambiente, até a realização de detecção dos marcadores e projeção dos elementos virtuais nas posições obtidas com processamento das informações da cena.

# 4 PADRÕES E OPERAÇÕES COM CORES

Um espaço de cores facilita a especificação de cores respeitando um padrão de representação aceito pela comunidade científica. Operações com espaços de cores, como conversões por exemplo, são comumente usadas para garantir que determinados procedimentos sejam realizados em um mesmo espaço de cor específico.

O processo de detecção de marcadores *CRFM*, é baseado em operações de conversão entre espaços de cores para identificação das cores nos blocos que compõem a hierarquia do marcador. A identificação das cores de áreas de amostragem é feita através da medição de diferença entre cores no espaço *CIELab*, enquanto a imagens de entrada estão em *RGB*, existindo uma necessidade existam conversões.

Este capítulo aborda a representação de espaços de cores, tratando sobre medição de diferença entre cores e realização de conversão entre diferentes espaços de cores.

## 4.1 DEFINIÇÃO DE COR

A cor é uma grandeza descrita através da percepção atrelada às células fotossensíveis do olho humano e, à representação formada através do cérebro (GIROLAMI et al., 2013). Cor é a porção do espectro visível da luz que é refletida de volta a partir de uma superfície. Todas as superfícies absorvem parte da luz. Superfícies pretas absorvem toda a luz, entretanto, as brancas refletem-na.

A percepção da cor é um fenômeno que depende da composição do objeto e a iluminação no local, além de estar ligada às propriedades dos olhos e cérebro que captam e processam a cena percebida e, ainda, os ângulos de visão e iluminação existentes (WU; SUN, 2013).

Independente das características que envolvem a percepção da cor, é necessária uma formulação de um modelo matemático para realizar a representação da cor, que permita sua mensuração e manipulação. A formação deste modelo matemático é denominada Espaço de Cor e será abordado na seção seguinte.

## 4.2 ESPAÇOS DE CORES

Um espaço de cor é uma organização específica de cores, caracterizando-se como um modelo matemático para descrever uma cor a partir de fórmulas (KUEHNI, 2003). O objetivo dos espaços de cores é auxiliar o processo de descrição da cor, entre pessoas,

máquinas ou *softwares*.

Conforme (GUIMARãES, 2004), um espaço de cores tem por objetivo:

> "... incluir todas as cores, ao menos em forma teórica, em um modelo topológico, prevendo uma posição específica para cada uma delas e propondo alguma lógica que determine a organização total.

De maneira geral, um espaço de cores é a definição de um sistema de coordenadas e um subespaço dentro deste, de modo que uma cor é representada através de um (único) ponto nesse subespaço (GONZALEZ; WOODS, 2006).

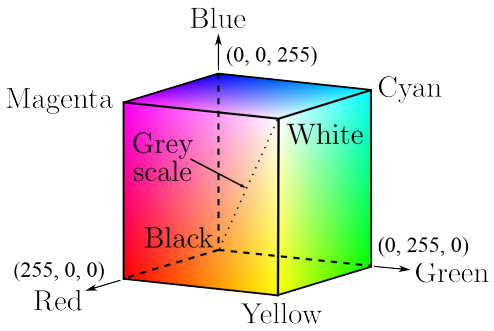
Um espaço de cores pode ser arbitrário, com cores específicas atribuídas a um conjunto de amostras de cores físicas nomeadas ou numeradas, tal como com o sistema Pantone, ou matematicamente estruturado, como *RGB*, *CIELab* e *HSV*, por exemplo.

### 4.2.1 Espaço de cores *RGB*

O espaço de cores *RGB*, composto pelas cores Vermelho (*Red*), Verde (*Green*) e Azul (*Blue*) é baseado em coordenadas cartesianas e pode ser representado por um cubo definido como Cubo de Cores *RGB* (AZEVEDO; CONCI, 2003) (GONZALEZ; WOODS, 2006) (Figura 4.1).

As três cores são primárias e aditivas, logo, os componentes individuais são adicionados, e com isso formam outras cores (Figura 4.2). Sua origem começa na cor preta, e todas as outras cores são derivadas adicionando determinadas quantidades de cores primárias.

Figura 4.1 – Cubo de Cores *RGB*.



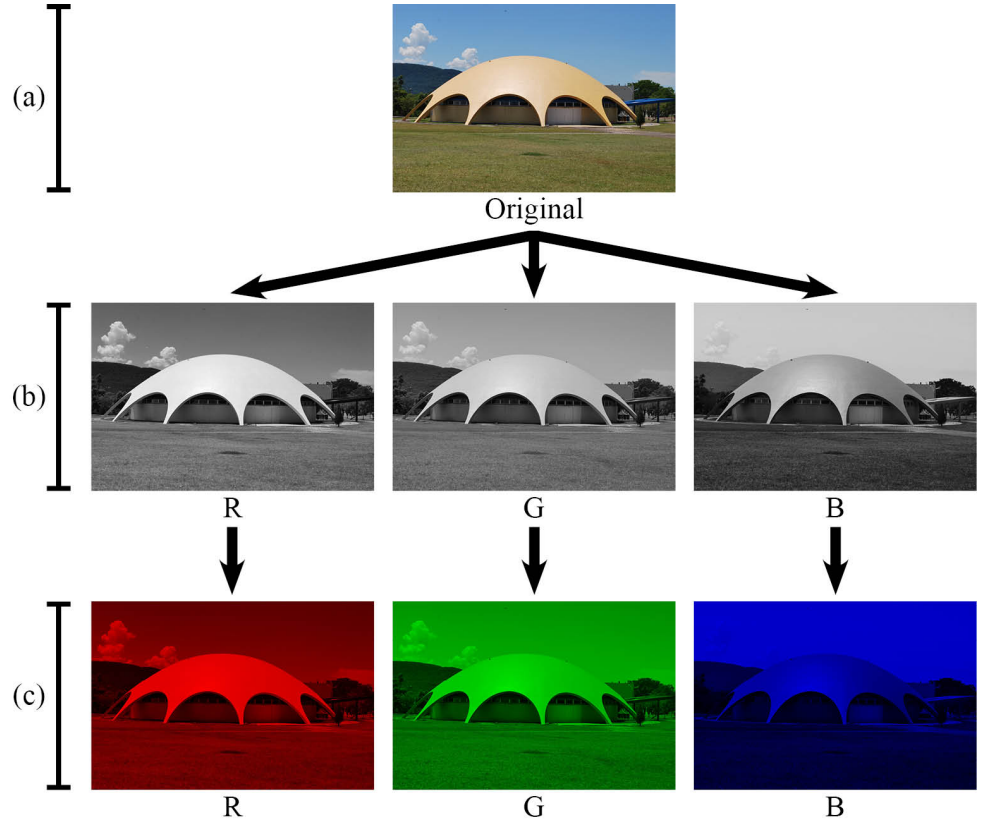

Fonte: (ARAMESH et al., 2016).

As cores do Espaço de Cores *RGB* são mapeadas no Cubo de cores *RGB*, desta forma, a cor preta tem zero intensidades em vermelho, verde ou azul, por isso tem as coordenadas (0, 0, 0). No canto oposto do cubo de cores, o branco tem intensidades máximas de cada cor, ou seja, (255, 255, 255). Vermelho de intensidade total, com zero componentes verdes ou azuis, também está posicionado em um canto do cubo no local

(255, 0, 0). Amarelo, que é combinação de vermelho e verde, está posicionado em (255, 255, 0) (GONZALEZ; WOODS, 2006).

A Figura 4.2 exibe uma imagem em (a), seus canais RGB são convertidos para escala de cinza em (b) e exibidos isoladamente em (c).

Figura 4.2 – Separação dos canais RGB.




Fonte: Próprio autor.

As cores ciano e magenta, que são combinações de (verde e azul) e (vermelho e azul), respectivamente, estão em (0, 255, 255) e (255, 0, 255) (Figura 4.3). Finalmente, o ponto médio do valor de cinza está no centro exato do cubo no local (128, 128, 128) (KLETTE, 2014). Todas as outras cores podem ser descritas especificando suas respectivas coordenadas dentro deste cubo.

Figura 4.3 – Coordenadas cartesianas localizadas no Cubo de Cores *RGB*.

| | R | G | B |
|---|---|---|---|
| Vermelho | 255 | 0 | 0 |
| Verde | 0 | 255 | 0 |
| Azul | 0 | 0 | 255 |
| Branco | 255 | 255 | 255 |
| Preto | 0 | 0 | 0 |
| Ciano | 0 | 255 | 255 |
| Amarelo | 255 | 255 | 0 |
| Magenta | 255 | 0 | 255 |

Fonte: Próprio autor.

O Espaço de cores *RGB* é a escolha mais provável para gráficos computacionais, pois as telas coloridas usam combinações de Vermelho, Verde e Azul para criar a cor desejada. Embora monitores e outros *displays* de dispositivos sejam capazes de representar milhões cores, existem vários outros sistemas que possuem limitação de cores representadas, como é o caso de imagens da *Web*.

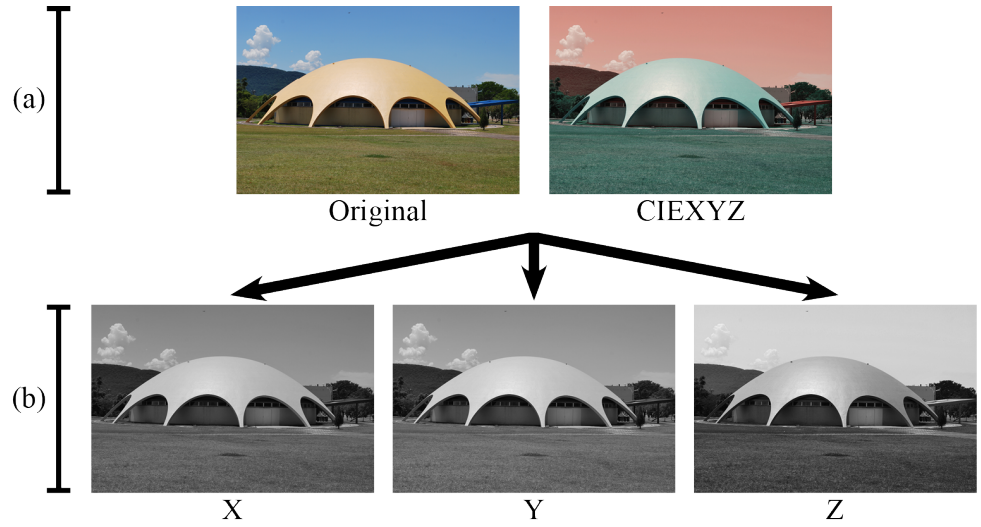
### 4.2.2 Espaço de cores *CIEXYZ*

Em 1931, a *Commission Internationale de l'Eclairage* (*CIE*) padronizou um espaço de cores primário chamado *CIEXYZ*. Segundo (MINOLTA, 2003), o modelo *CIEXYZ* é um espaço de cores baseado nos valores de triestímulos *XYZ* e é o ponto de partida para as especificações dos sistemas de cores *CIE*.

Neste espaço, um conjunto de cores correspondem a três funções que estão relacionadas com os cones vermelhos, verdes e azuis nos olhos, sendo chamado de observador padrão (HUNT; POINTER, 2011). Para o Espaço de Cores *CIEXYZ*, o *Y* representa a leveza, já o *X* e *Z* correspondem a dois componentes virtuais que se parecem com uma curva sensível dos cones vermelho e azul.

A Figura 4.4 exibe uma imagem (a) e seus canais XYZ são exibidos isoladamente em (b).

Figura 4.4 – Separação dos canais *CIEXYZ*.




Fonte: Próprio autor.

Segundo (WU; SUN, 2013), devido ao fato de *CIEXYZ* não representar uma graduação uniforme de cor, posteriormente o Espaço de Cor *CIELab* foi criado e amplamente utilizado em instrumentos de medição de cor. O espaço *CIELab* foi originado a partir de transformações não lineares de *CIEXYZ*.

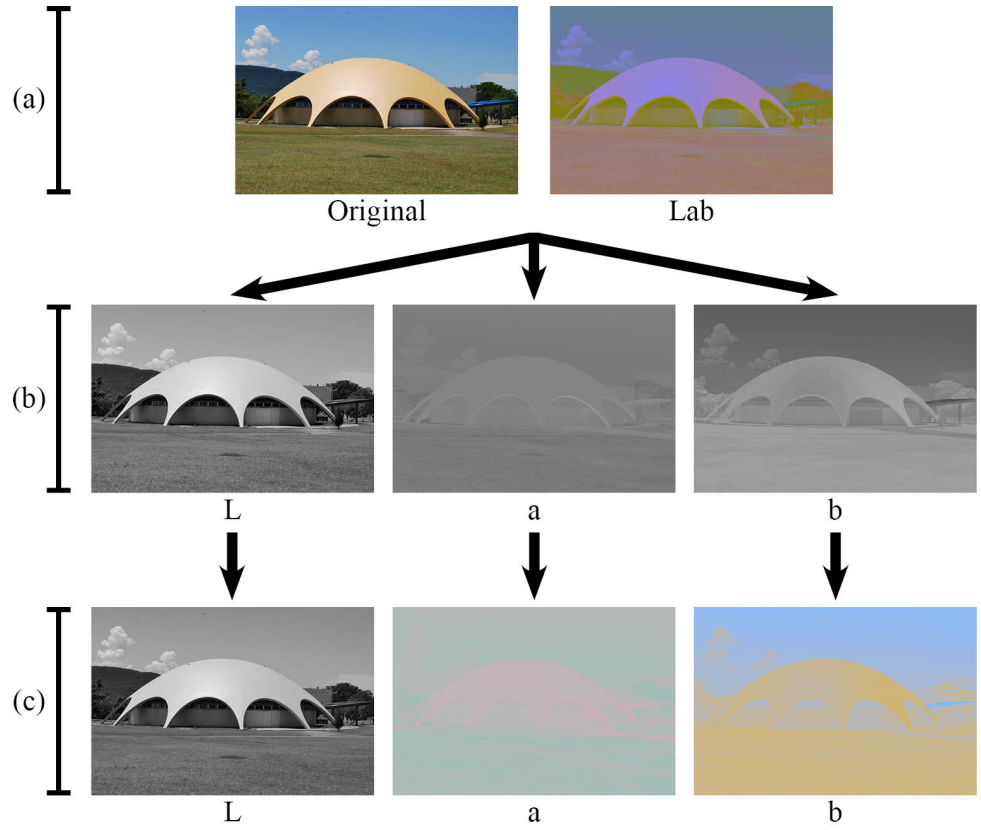
O Espaço de Cores *CIEXYZ*, é independente de dispositivo, desta forma, cada cor primária (*X*, *Y*, *Z*) é sempre constante, ao contrário de *RGB* que varia de acordo com cada dispositivo individual (monitor, *scanner*, câmera, etc.).

### 4.2.3 Espaço de cores *CIELab*

O espaço de cores *CIELab* descreve matematicamente, em três canais, todas as cores percebíveis (Figura 4.5): um canal que representa a luminosidade da cor (*L**) (que vai de 0 (preto) até 100 (branco)), e os parâmetros *a** (que vai do verde, se negativo, até o vermelho, quando positivo) e *b** (do azul, se negativo, até o amarelo, se positivo) são os componentes cromáticos, e variam entre -120 e +120 (GIROLAMI et al., 2013) (Figura 4.6).

A Figura 4.5 exibe uma imagem (a), seus canais *CIELab* são convertidos para tons de cinza (b) e exibidos isoladamente (c).

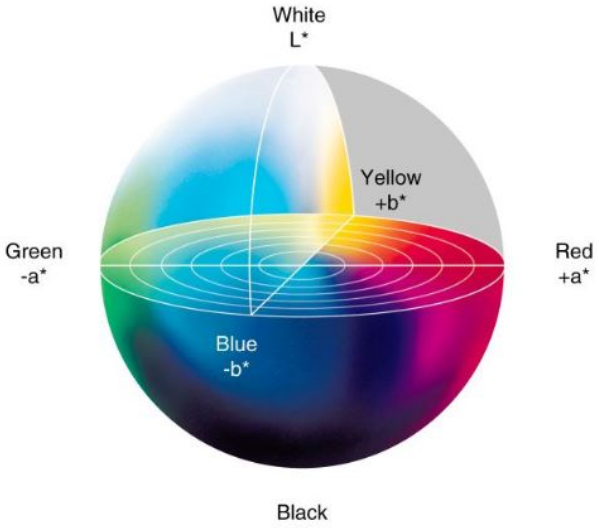
Figura 4.5 – Separação dos canais *CIELab*.




Fonte: Próprio autor.

Outra forma de representação do espaço *CIELab* é pelo uso de coordenadas cilíndricas de luminosidade (*L*), croma (*C*) e tonalidade (*h*), relacionados diretamente com as coordenadas *Munsell* (JUDD; WYSZECKI, 1975) (Figura 4.7).

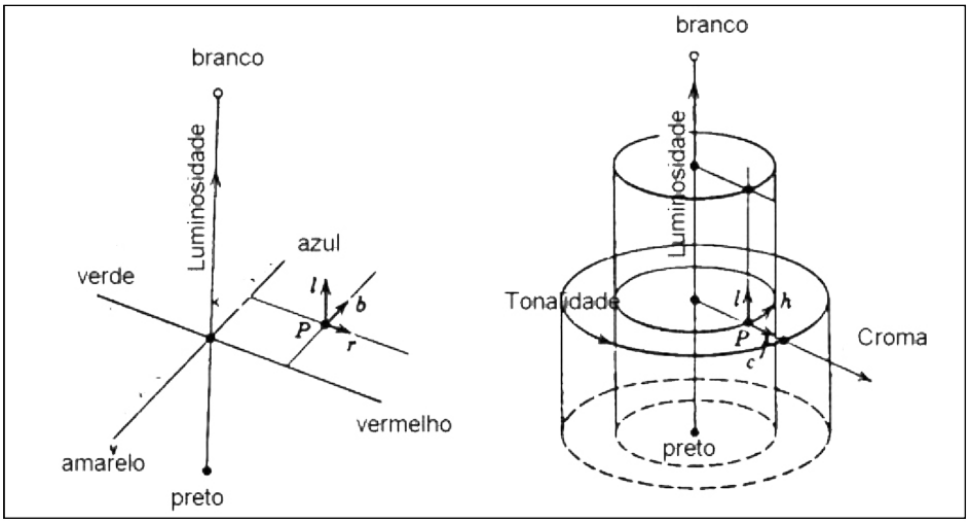
Figura 4.6 – Representação do espaço de cor *CIELAb*.




Fonte: Adaptado de Minolta (2003)

Um importante atributo do espaço de cores *CIELab* é a sua independência do dispositivo, significando que as cores são definidas independentemente do dispositivo onde são exibidas. Essa precisão e portabilidade o tornam adequado em várias indústrias diferentes, como impressão, automotivas, têxteis e plásticos.

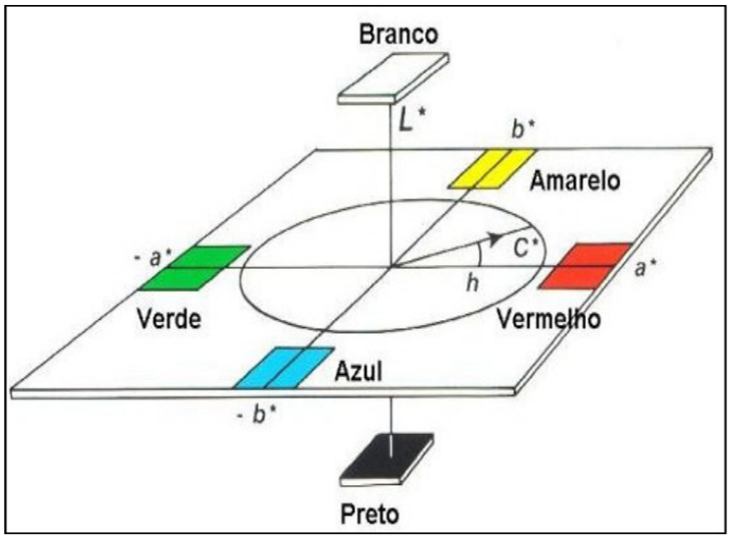
Figura 4.7 – Coordenadas cartesianas do espaço *CIELab* (e) e *CIELCh* (d).




Fonte: Adaptado de Judd e Wyszecki (1975)

A Figura 4.8 apresenta o significado geométrico do conjunto de coordenadas do espaço *CIELab*.

Figura 4.8 – O significado geométrico das coordenadas *CIELab* e *CIELCh*.




Fonte: Adaptado de Mokrzycki e Tatol (2011)

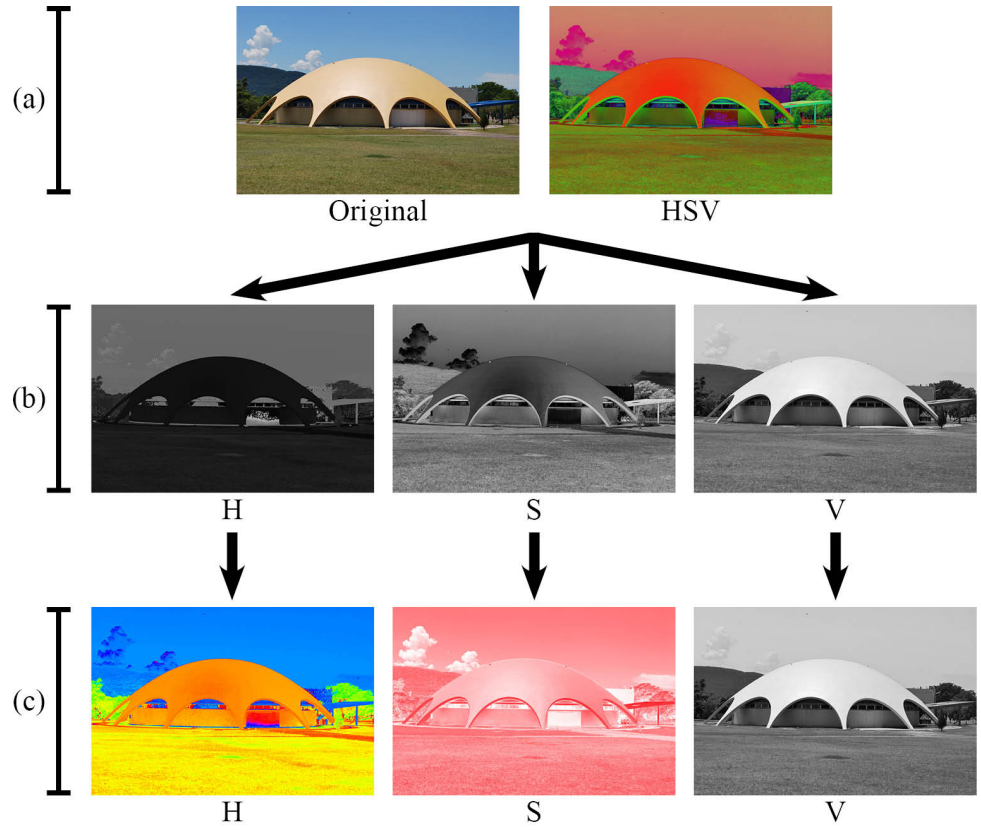
Embora o espaço de cores *CIELab* seja a representação mais exata da cor, não é o mais usado. A cor, no espaço de cores *CIELab*, geralmente é convertida em espaços de cores menos precisos, como *RGB* e *CYMK*, porque monitores de computador e impressoras usam estes espaços para representar imagens.

### 4.2.4 Espaço de cores *HSV*

O modelo *HSV*, diferentemente do *RGB*, separa a cromaticidade dos valores de intensidade e vivacidade de cor, se aproximando mais da maneira como o ser humano percebe as cores (GONZALEZ; WOODS, 2006). *HSV* é a abreviatura para os três componentes *hue* (matiz), *saturation* (saturação) e *value* (valor) (Figura 4.9).

A Figura 4.5 exibe uma imagem (a), seus canais HSV separados em (b) e decompostos em (c).

Figura 4.9 – Separação dos canais *HSV*.

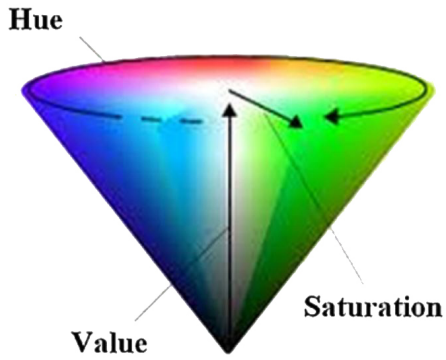



Fonte: Próprio autor.

A definição de cada cor no espaço de cores *HSV* é feita abaixo, conforme descrito por Gonzalez e Woods (2006):

- **Matiz (tonalidade)**: Verifica o tipo de cor, abrangendo todas as cores do espectro. Atinge valores de 0 a 360, mas para algumas aplicações, esse valor é normalizado de 0 a 100%.
- **Saturação**: Também chamado de "pureza". Quanto menor esse valor, mais com tom de cinza aparecerá a imagem. Quanto maior o valor, mais "pura" é a imagem. Atinge valores de 0 a 100%.
- **Valor (brilho)**: Descreve uma noção acromática de intensidade, que pode ser relacionada com o brilho. Atinge valores de 0 a 100%.

Figura 4.10 – Espaço de Cor *HSV*.




Fonte: (CHARISIS et al., 2012).

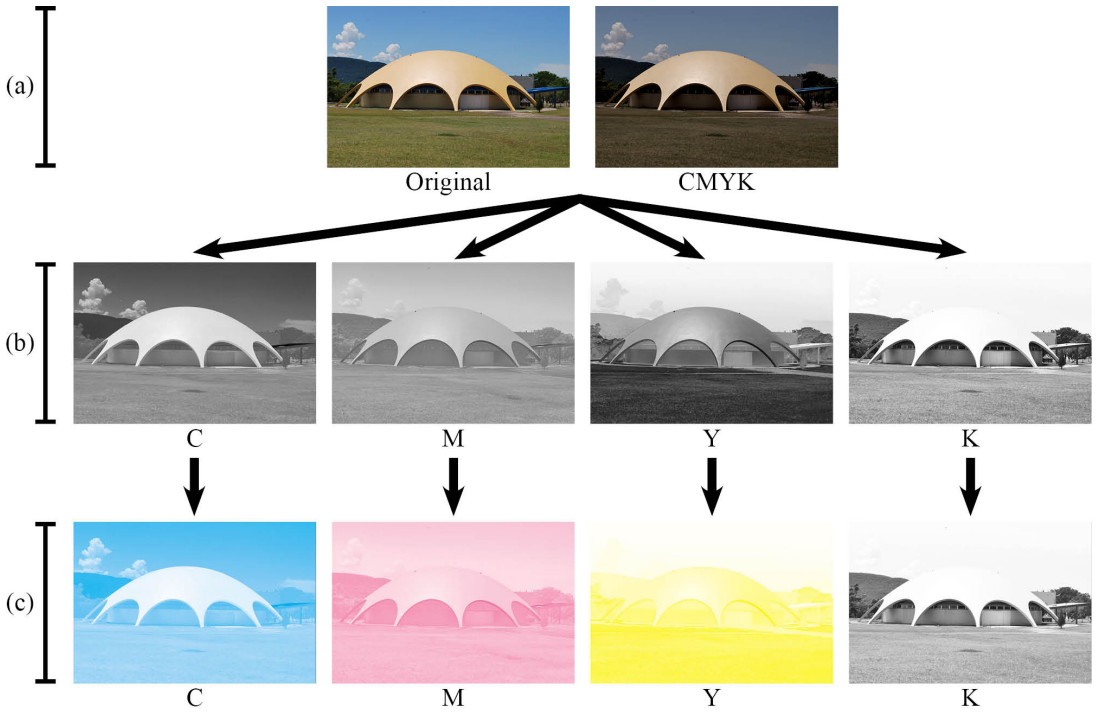
Conforme a representação do modelo, exibida na Figura 4.10, o componente *H* pode ser descrito como um ângulo, com domínio entre [0, $2\pi$]. A Saturação pode ser compreendida como a distância radial (a partir do centro) do cone que representa o sistema *HSV*, sendo que assume valores entre [0, 1]. O componente de intensidade pode ser compreendido como o eixo vertical do cone e possui valores pertencentes ao intervalo [0, 1] (SURAL; QIAN; PRAMANIK, 2002).

### 4.2.5 Espaço de cores *CMYK*

O modelo de cores *CMYK* é um espaço de cores subtrativas geralmente utilizado para descrever o processo de impressão colorida (GONZALEZ; WOODS, 2006). O modelo baseia-se na combinação dos componentes Ciano (*Cyan*), Magenta (*Magenta*), Amarelo (*Yellow*) e Preto (*Black*) para criar diferentes cores (Figura 4.11). Para impressoras que utilizam este espaço de cores, independente do espaço de cores do arquivo digital, será realizada uma conversão para *CMYK* para ser realizada a impressão (GONZALEZ; WOODS, 2006).

A Figura 4.5 exibe uma imagem (a), seus canais CMYK separados em (b) e decompostos em (c).

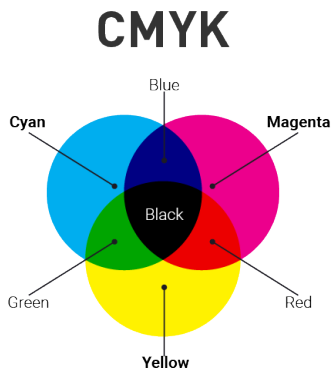
Figura 4.11 – Separação dos canais *CMYK*.




Fonte: Próprio autor.

A combinação de duas cores do espaço de cores *CMYK* cria uma nova que que é relativamente pura ou saturada. Por exemplo, pode-se fazer o vermelho combinando o magenta e o amarelo, que absorvem as luzes verde e azul, respectivamente (Figura 4.12).

Figura 4.12 – Espaço de Cor *CMYK*.




Fonte: Adaptado de Rosi et al. (2016).

A Figura 4.12 ilustra as possíveis combinações das três componentes primitivas subtrativas completamente saturadas. Nota-se que considerando combinações por pares de componentes, tem-se as cores aditivas e considerando a combinação do terno de componentes tem-se, teoricamente, a cor preta pura.

Os valores para cada um dos canais variam de 0% até 100%, para a composição

da imagem. Quanto maior for a percentagem de cada uma das cores que a compõe mais escura será a imagem final. A cor preta está presente neste modelo devido ao fato da mistura de Ciano, Magenta e Amarelo não produzirem a cor preta pura. As cores subtrativas, geralmente são utilizadas para impressões, o que necessita de corantes que normalmente possuem impurezas. Esta constatação conclui que a intenção de impressão da cor preta pela combinação das três componentes subtrativas gera, na maioria dos casos, uma cor próxima à cor preta. Para contornar tal situação o modelo de cores *CMYK* foi criado, possuindo o componente *K* (*black*), que misturado com os componentes *C*, *M* e *Y* possibilita a criação da cor preta pura.

## 4.3 MEDIÇÃO DE DIFERENÇA DE COR

A diferença ou a distância entre duas cores é uma métrica definida que permite uma análise quantificada da relação entre duas cores. A quantificação dessa propriedade é de grande importância para que diferenças entre cores sejam encontradas quando ocorrer uma avaliação utilizando algum método de medição, mesmo que as cores avaliadas sejam visualmente idênticas.

Para aplicações em Realidade Aumentada, que avaliam a cor de elementos distribuídos na cena, como marcadores, onde a cor é influenciada por fatores, como tipo de iluminação, por exemplo, se a cor de um dos marcadores rastreados não corresponder com um padrão, a técnica de identificação e rastreamento poderá ser comprometida.

A medição da diferença entre cores pode ser aplicada em um dos passos do processo de calibração de câmeras, utilizada em aplicações de Realidade Aumentada, aplicando seus conceitos para encontrar a relação das cores que a câmera captura da cena, com cores alvo que são esperadas. Desta forma, com uma relação estabelecida, é possível aplicar o resultado da medição da diferença de cor em algoritmos de correção de cor e compensação de iluminação.

Para começar o processo de medição da diferença entre cores, a cor de amostra e a cor alvo devem ser medidas e os valores de cada medida são salvos. Estes valores são aplicados em uma comparação numérica entre as cores, indicando inconsistências nas coordenadas de cores absolutas.

### 4.3.1 Fórmulas de diferença de cor no espaço de cores *RGB*

O espaço de cores *RGB*, se for considerado com um espaço Euclidiano, então as medidas de distâncias conhecidas podem ser adotadas para o cálculo de uma diferença de cor (KOSCHAN; ABIDI, 2008), tanto para as quantidades de valores de cor padronizadas

por intensidade, quanto como para os valores não padronizados.

A Equação 4.1 mostra como é feita a padronização de cor por intensidade:

$$R_o = \frac{R_i}{R_i + G_i + B_i}, \quad G_o = \frac{G_i}{R_i + G_i + B_i}, \quad and \quad B_i = \frac{B_i}{R_i + G_i + B_i}, \tag{4.1}$$

em que $R_i$, $G_i$ e $B_i$ são os valores de entrada e $R_o$, $G_o$ e $B_o$ os valores com a padronização de cor por intensidade já aplicada.

A medida da distância de cor pode ser usada tanto para os valores proporcionais da cor ($R_i$, $G_i$, $B_i$), padronizados por intensidade e para os valores não padronizados. No espaço de cores RGB, a distância euclidiana entre dois vetores de cores $F_1$ e $F_1$ representam o ângulo entre os dois vetores (Figura 4.13).

$$d = \sqrt{(R_2 - R_1)^2 + (G_2 - G_1)^2 + (B_2 - B_1)^2}, \tag{4.2}$$

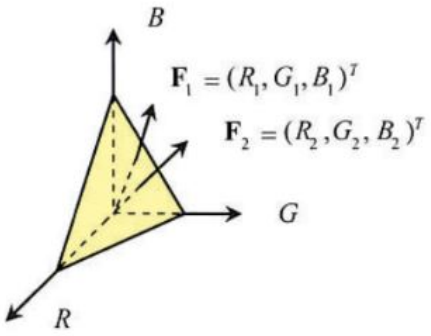
Como, para este cálculo, o resultado deve ser computacionalmente simples, também é aceitável remover a raiz quadrada e simplesmente usar:

$$d^2 = (R_2 - R_1)^2 + (G_2 - G_1)^2 + (B_2 - B_1)^2 \tag{4.3}$$

em que *d* corresponde a distância calculada para as cores definidas pelas retas $F_1$ ($R_1$, $G_1$, $B_1$) e $F_2$ ($R_2$, $G_2$, $B_2$).

Essas medidas de distância são amplamente utilizadas no espaço de cores *RGB*, devido este espaço de cores ser uma maneira conveniente de representação de cores. No entanto, a percepção humana identifica as cores de uma maneira diferente de como as cores são representadas no espaço de cores *RGB*, logo não existe conexão entre as medidas de distância Euclidiana e a percepção de cor humana. O que aparenta cores idênticas para o olho humano, pode resultar em uma distância Euclidiana que é maior que a distância Euclidiana de cores que se parecem diferentes quando comparados no espaço de cores *RGB*.

Figura 4.13 – Representação de dois vetores de cores $F_1$ e $F_2$ no espaço *RGB*.

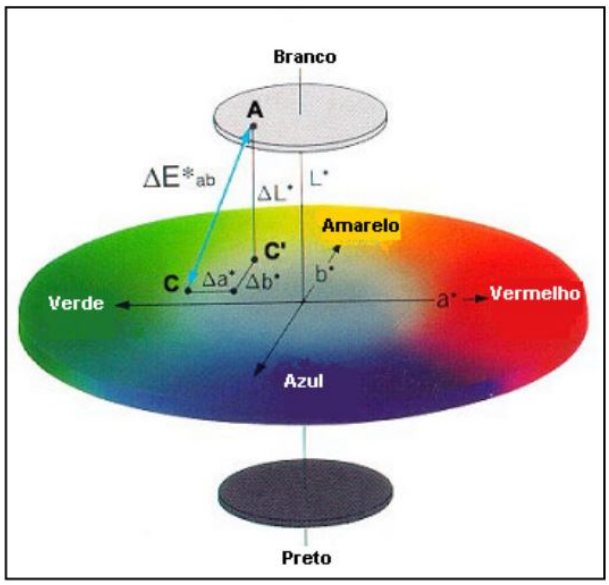



Fonte: Adaptado de Koschan e Abidi (2008)

### 4.3.2 Fórmulas de diferença de cor no espaço de cores *CIELab*

No espaço *CIELab* é possível quantificar as diferenças de cor através do cálculo de *Delta E* ($\Delta E*$) (Figura 4.14).

Figura 4.14 – Diagrama de cálculo de $\Delta E*$ no diagrama *CIELAB*.




Fonte: Adaptado de Minolta (2003)

*Delta E* é uma métrica para entender como o olho humano percebe a diferença de cor. O termo *delta* vem da matemática, ou seja, mudança em uma variável ou função. De acordo com Mokrzycki e Tatol (2011), em uma escala típica, o valor $\Delta E*$ variará de 0 a 100, conforme mostrado na Figura 4.15.

Figura 4.15 – Valores gerais de percepção com base em $\Delta E*$.

| Delta E | Percepção |
|---|---|
| **0 < ΔE < 1** | A diferença é imperceptível |
| **1 < ΔE < 2** | A diferença é notada apenas por um observador experiente |
| **2 < ΔE < 3.5** | A diferença também é notada por um observador inexperiente |
| **3.5 < ΔE ≤ 5** | A diferença é claramente perceptível |
| **5 < ΔE** | Dá a impressão de que são duas cores diferentes |

Fonte: Adaptado de Mokrzycki e Tatol (2011)

Admitindo os valores definidos na Figura 4.15 como um guia geral; é possível obter um valor $\Delta E*$ abaixo de 1.0 para duas cores que aparecem diferentes. Este é o caso das

fórmulas *CIE76* e *CIE94*, nas quais a saturação não é considerada ou não é ponderada adequadamente (MOKRZYCKI; TATOL, 2011).

Devido a inconsistências entre os algoritmos, o significado exato do $\Delta E$ muda ligeiramente dependendo da fórmula utilizada. Desta forma é necessário pensar em $\Delta E$ menos como uma resposta definitiva e, em vez disso, uma métrica útil para se candidatar a um caso de uso específico.

#### *4.3.2.1 Delta E 76*

Em 1976 o *CIE* apresentou o espaço de cores *CIELab*, dando origem a primeira fórmula $\Delta E$, também chamada de $\Delta E^*_{ab}$, para cálculo de diferença entre cores (MCLAREN, 1976). Esta relação é possível devido ao fato do espaço de cores *CIELab* ter sido criado como uma matriz *3D* de pontos de cor, logo uma abordagem lógica seria encontrar a diferença entre duas cores apenas medindo a distância entre dois pontos no espaço *3D* (MOKRZYCKI; TATOL, 2011).

Segundo McLAREN (1976), a diferença, $\Delta E^*_{76}$, entre uma cor de referência $(L^*_1, a^*_1, b^*_1)$ e outra cor $(L^*_2, a^*_2, b^*_2)$ é calculada por:

$$\Delta E^*_{ab} \quad = \quad [(\Delta L^*)^2 \quad + \quad (\Delta a^*)^2 \quad + \quad (\Delta b^*)^2]^{1/2}, \tag{4.4}$$

logo,

$$\Delta E^*_{ab} = \sqrt{(L^*_2 - L^*_1)^2 + (a^*_2 - a^*_1)^2 + (b^*_2 - b^*_1)^2} \tag{4.5}$$

Figura 4.16 – Azul escuro e o vermelho escuro no espaço de cores *CIELab*.

Fonte: Próprio autor.

A saturação é um grande problema para $\Delta E^*_{76}$. Como exemplo, a Figura 4.16 mostra dois blocos que representam as cores azul escuro e vermelho escuro no espaço de cores *CIELab*. Utilizando a fórmula $\Delta E$ para cálculo da diferença de cor, será relatado como resultado $\Delta E^*_{76} = 5$ (diferença claramente perceptível), quando, por definição, a diferença deveria estar no intervalo $\Delta E \sim 1 - 2$ (perceptível apenas por um observador experiente).

#### *4.3.2.2 Delta E 94*

Em 1994, a fórmula original de $\Delta E$ foi melhorada. A nova fórmula apresentada passaria a levar em consideração certos fatores de ponderação para cada valor de luminância, croma e matiz. Esta melhoria também introduziu a capacidade de adicionar um modificador de acordo com a caso de uso do cálculo da diferença de cores (MOKRZYCKI; TATOL, 2011) (REINHARD et al., 2008).

Dada uma cor de referência $(L_1^*, a_1^*, b_1^*)$ e outra cor $(L_2^*, a_2^*, b_2^*)$, a diferença entre as cores é definida por:

$$\Delta E_{94}^* = \sqrt{\left(\frac{\Delta L^*}{k_L S_L}\right)^2 + \left(\frac{\Delta C_{ab}^*}{k_C S_C}\right)^2 + \left(\frac{\Delta H_{ab}^*}{k_H S_H}\right)^2}, \tag{4.6}$$

em que, a diferença de brilho é definida por $\Delta L^*$:

$$\Delta L^* = L_1^* - L_2^* \tag{4.7}$$

A saturação é definida por $\Delta C_{ab}^*$:

$$\Delta C_{ab}^* = C_1^* - C_2^* \tag{4.8}$$

$$C_1^* = \sqrt{a_1^{*2} + b_1^{*2}} \tag{4.9}$$

$$C_2^* = \sqrt{a_2^{*2} + b_2^{*2}} \tag{4.10}$$

O valor da sombra para ambas as amostras comparadas é definido por $\Delta H_{ab}^*$:

$$\Delta H_{ab}^* = \sqrt{{\Delta E_{ab}^*}^2 - \Delta L^{*2} - {\Delta C_{ab}^*}^2} = \sqrt{\Delta a^{*2} + \Delta b^{*2} - {\Delta C_{ab}^*}^2} \tag{4.11}$$

$$\Delta a^* = a_1^* - a_2^* \tag{4.12}$$

$$S_L = 1 \tag{4.13}$$

$$S_C = 1 + K_1 C_1^* \tag{4.14}$$

$$S_H = 1 + K_2 C_1^* \tag{4.15}$$

Os valores definidos por $k_C$ e $k_H$ possuem valor padrão 1 e os fatores de ponderação $k_L$, $K_1$ e $K_2$ dependem da aplicação, conforme a Figura 4.17:

Figura 4.17 – Valores para os fatores de ponderação, de acordo com a aplicação.

| | Artes Gráficas | Têxteis |
|---|---|---|
| $k_L$ | 1 | 2 |
| $K_1$ | 0.045 | 0.048 |
| $K_2$ | 0.015 | 0.014 |

Fonte: Adaptado de Mokrzycki e Tatol (2011).

Este método de medição de diferença de cores, introduziu uma conversão do valor *CIELab* para o espaço de cores *CIELCh* (REINHARD et al., 2008). Os dois modelos de cores diferem em que *CIELCh* representa a tonalidade como um ângulo em vez de pontos infinitos de cor.

#### *4.3.2.3 Delta E de 2000*

O desenvolvimento de uma melhor fórmula para cálculo de diferença de cor não parou com a publicação de *CIE94* ($\Delta E^*_{94}$). Testes realizado mostraram que o *CIE94* ainda possuía grandes erros nas cores azuis saturadas e para cores quase neutras (CIE, 142-2001). Em 2001, a *CIE* recomendou uma fórmula melhorada de diferença de cor, chamada *CIEDE2000* ($\Delta E^*_{00}$).

Como *CIE94* não resolveu adequadamente questões de uniformidade perceptiva, a CIE acrescentou correções a *CIEDE2000* (SHARMA; WU; DALAL, 2005):

- Um termo de rotação de tonalidade (*RT*), para lidar com as regiões problemáticas em azul;
- Compensação para cores neutras (os valores iniciados nas diferenças *L*C*h*);
- Compensação pela luminosidade (*SL*);
- Compensação por croma (*SC*);
- Compensação pela tonalidade (*SH*).

Segundo (SHARMA; WU; DALAL, 2005), dado um par de valores de cor no espaço *CIELAB* $L_1, a_1, b_1$ e $L_2, a_2, b_2$, a diferença entre as cores é pela fórmula *CIEDE2000* é

definida por:

$$\Delta E^*_{00}(L_1, a_1, b_1; L_2, a_2, b_2) = \Delta E^{12}_{00} = \Delta E^*_{00} \tag{4.16}$$

Segundo Mokrzycki e Tatol (2011) processo de cálculo da diferença de cor é realizado seguindo as seguintes equações:

$$\Delta E^*_{00} = \sqrt{\left(\frac{\Delta L'}{k_L S_L}\right)^2 + \left(\frac{\Delta C'}{k_C S_C}\right)^2 + \left(\frac{\Delta H'}{k_H S_H}\right)^2 + R_T \frac{\Delta C'}{k_C S_C}\frac{\Delta H'}{k_H S_H}}, \tag{4.17}$$

em que

$$\Delta L' = L^*_2 - L^*_1 \tag{4.18}$$

$$\bar{L} = \frac{L^*_1 + L^*_2}{2} \quad \bar{C} = \frac{C^*_1 + C^*_2}{2} \tag{4.19}$$

$$a'_1 = a^*_1 + \frac{a^*_1}{2}\left(1 - \sqrt{\frac{\bar{C}^7}{\bar{C}^7 + 25^7}}\right) \quad a'_2 = a^*_2 + \frac{a^*_2}{2}\left(1 - \sqrt{\frac{\bar{C}^7}{\bar{C}^7 + 25^7}}\right) \tag{4.20}$$

$$\bar{C}' = \frac{C'_1 + C'_2}{2} \text{ e } \Delta C' = C'_2 - C'_1 \quad \text{em que } C'_1 = \sqrt{a_1'^2 + b_1^{*2}} \quad C'_2 = \sqrt{a_2'^2 + b_2^{*2}} \tag{4.21}$$

$$h'_1 = \text{atan2}(b^*_1, a'_1) \mod 360^\circ, \quad h'_2 = \text{atan2}(b^*_2, a'_2) \mod 360^\circ \tag{4.22}$$

As especificações de cores são dadas em 0 a 360 graus, então é necessário algum ajuste. A tangente inversa ($atan2 = tan^{-1}$) é indeterminada se ambos $a^{'}$ e $b$ são zero (o que também significa que o $C^{'}$ correspondente é zero); nesse caso, o ângulo de matiz deve ser ajustado para zero (SHARMA; WU; DALAL, 2005).

$$\Delta h' = \begin{cases} h'_2 - h'_1 & |h'_1 - h'_2| \leq 180^\circ \\ h'_2 - h'_1 + 360^\circ & |h'_1 - h'_2| > 180^\circ, h'_2 \leq h'_1 \\ h'_2 - h'_1 - 360^\circ & |h'_1 - h'_2| > 180^\circ, h'_2 > h'_1 \end{cases} \tag{4.23}$$

$$\Delta H' = 2\sqrt{C'_1 C'_2}\sin(\Delta h'/2), \quad \bar{H}' = \begin{cases} (h'_1 + h'_2)/2 & |h'_1 - h'_2| \leq 180^\circ \\ (h'_1 + h'_2 + 360^\circ)/2 & |h'_1 - h'_2| > 180^\circ, h'_1 + h'_2 < 360^\circ \\ (h'_1 + h'_2 - 360^\circ)/2 & |h'_1 - h'_2| > 180^\circ, h'_1 + h'_2 \geq 360^\circ \end{cases} \tag{4.24}$$

$$T = 1 - 0.17\cos(\bar{H}' - 30^\circ) + 0.24\cos(2\bar{H}') + 0.32\cos(3\bar{H}' + 6^\circ) - 0.20\cos(4\bar{H}' - 63^\circ) \tag{4.25}$$

$$S_L = 1 + \frac{0.015\left(\bar{L} - 50\right)^2}{\sqrt{20 + \left(\bar{L} - 50\right)^2}} \tag{4.26}$$

$$S_C = 1 + 0.045\bar{C}' \tag{4.27}$$

$$S_H = 1 + 0.015\bar{C}'T \tag{4.28}$$

$$R_T = -2\sqrt{\frac{\bar{C}'^7}{\bar{C}'^7 + 25^7}}\sin\left[60^\circ \cdot \exp\left(-\left[\frac{\bar{H}' - 275^\circ}{25^\circ}\right]^2\right)\right] \tag{4.29}$$

## 4.4 CONVERSÃO DE ESPAÇO DE COR

CRFM Lib realiza operações de conversões em espaços de cores no processo de correspondência de cores, que é abordado na Seção 6.3. As conversões de Espaços de Cores são necessárias para traduzir a representação de cores de um modelo matemático para o outro.

### 4.4.1 Conversão de *RGB* para *CMYK*

Os valores *R*, *G* e *B* são divididos por 255 para alterar o intervalo de 0 a 255 para 0 a 1:

$$R' = \frac{R}{255}, G' = \frac{G}{255}, B' = \frac{B}{255} \tag{4.30}$$

$$m_{max} = max(R', G', B') \tag{4.31}$$

Para obter cada um dos valores de *CMYK*:

- **Black**: Calculada a partir das cores Vermelha ($R'$), Verde ($G'$) e Azul ($B'$);

$$K = 1 - m_{max} \tag{4.32}$$

- **Ciano** - Calculada a partir das cores vermelha ($R'$) e Preta ($K$):

$$C = \frac{(1 - R' - K)}{(1 - K)} \tag{4.33}$$

- **Magenta** - Calculada a partir das cores Verde ($G'$) e Preta ($K$);

$$M = \frac{(1 - G' - K)}{(1 - K)} \tag{4.34}$$

- **Amarelo** - Calculada a partir das cores Azul ($B'$) e Preta ($K$);

$$Y = \frac{(1 - B' - K)}{(1 - K)} \tag{4.35}$$

A Figura 4.18 apresenta alguns exemplos valores *RGB* de cores e o respectivo valor *CMYK*.

Figura 4.18 – Cores com valores em *RGB* e valor correspondente em *CMYK*.

| | RGB | CMYK |
|---|---|---|
| **Preto** | (52, 52, 52) | (0, 0, 0, 0.796) |
| **Branco** | (243, 243, 242) | (0, 0, 0.004, 0.047) |
| **Vermelho** | (175, 54, 60) | (0, 0.691, 0.657, 0.313) |
| **Verde** | (70, 148, 73) | (0.527, 0, 0.506, 0.419) |
| **Azul** | (56, 61, 150) | (0.626, 0.593, 0, 0.411) |
| **Laranja** | (214, 126, 44) | (0, 0.411, 0.794, 0.160) |

Fonte: Próprio autor.

### 4.4.2 Conversão de *CMYK* para *RGB*

Os valores *R*, *G* e *B* são dados na faixa de 0 a 255. Para obter cada um destes valores:

- **Vermelha** - Calculada a partir das cores Ciano (*C*) e Preta (*K*);

$$R = 255 \times (1 - C) \times (1 - K) \tag{4.36}$$

- **Verde** - Calculada a partir das cores Magenta (*C*) e Preta (*K*);

$$G = 255 \times (1 - M) \times (1 - K) \tag{4.37}$$

- **Azul** - Calculada a partir das cores Amarelo (*C*) e Preta (*K*).

$$G = 255 \times (1 - Y) \times (1 - K) \tag{4.38}$$

A Figura 4.19 apresenta alguns exemplos valores *CMYK* de cores e o respectivo valor *RGB*.

Figura 4.19 – Cores com valores em *CMYK* e valor correspondente em *RGB*.

| | CMYK | RGB |
|---|---|---|
| Preto | (0, 0, 0, 0.796) | (52, 52, 52) |
| Branco | (0, 0, 0.004, 0.047) | (243, 243, 242) |
| Vermelho | (0, 0.691, 0.657, 0.313) | (175, 54, 60) |
| Verde | (0.527, 0, 0.506, 0.419) | (70, 148, 73) |
| Azul | (0.626, 0.593, 0, 0.411) | (56, 61, 150) |
| Laranja | (0, 0.411, 0.794, 0.160) | (214, 126, 44) |

Fonte: Próprio autor.

### 4.4.3 Conversão de *RGB* para *HSV*

Inicialmente os valores no Espaço de Cores *RGB*, entre 0 e 255 são recuperados em uma escala de 0 a 1. Para isso, os valores devem ser divididos por 255:

$$R' = \frac{R}{255}, G' = \frac{G}{255}, B' = \frac{B}{255} \tag{4.39}$$

$$m_{max} = max(R', G', B') \tag{4.40}$$

$$m_{min} = min(R', G', B') \tag{4.41}$$

$$\Delta = m_{max} - m_{min} \tag{4.42}$$

O cálculo do valor de *H* é realizado com base no maior valor $R'$, $G'$ e $B'$. Os dois valores menores são subtraídos e divididos pela diferença entre o maior e o menor, para então normalizar a tonalidade adicionando 0, 2 ou 4.

$$H = \begin{cases} 0°, se\ \Delta = 0 \\ \frac{G'-B'}{\Delta}, se\ m_{max} = R' \\ \frac{B'-R'}{\Delta} + 2, se\ m_{max} = G' \\ \frac{R'-G'}{\Delta} + 4, se\ m_{max} = B' \end{cases} \tag{4.43}$$

$$H' = H \times 255 \tag{4.44}$$

A saturação, *S*, é a diferença entre os maiores e os menores valores dos canais de cores, divididos pelo brilho, *V*. Se o valor de *V* for 0, então a saturação resultante também será 0.

$$S = \begin{cases} 0, se\ V = 0 \\ \frac{\Delta}{v}, se\ m_{max} \neq 0 \end{cases} \tag{4.45}$$

$$S' = S \times 255 \tag{4.46}$$

O brilho, *V*, é baseado no canal de cores mais brilhante.

$$V = m_{max} \tag{4.47}$$

$$V' = V \times 255 \tag{4.48}$$

A Figura 4.20 apresenta alguns exemplos valores *RGB* de cores e o respectivo valor *HSV*.

Figura 4.20 – Cores com valores em *RGB* e valor correspondente em *HSV*.

| | RGB | HSV |
|---|---|---|
| Preto | (52, 52, 52) | (0, 0.0, 20.4) |
| Branco | (243, 243, 242) | (60, 0.4, 95.3) |
| Vermelho | (175, 54, 60) | (357, 69.1, 68.6) |
| Verde | (70, 148, 73) | (122, 52.7, 58.0) |
| Azul | (56, 61, 150) | (237, 62.7, 58.8) |
| Laranja | (214, 126, 44) | (29, 79.4, 83.9) |

Fonte: Próprio autor.

### 4.4.4 Conversão de *HSV* para *RGB*

Inicialmente são obtidos os valores da tonalidade (*H*), a saturação (*S*) e o brilho (*V*), onde *H* está em uma escala entre 0 a 6 inclusive, e *S* e *V* numa escala de 0 a 1.

$$H = \begin{cases} Indefinido, se\ H' \\ \left(\frac{H'}{255} mod\ 6\right) + 6, se\ H' < 0 \\ \frac{H'}{255} mod\ 6, para\ outros\ casos \end{cases} \tag{4.49}$$

$$S = \frac{S'}{255}, V = \frac{V'}{255} \tag{4.50}$$

O passo seguinte é a obtenção dos valores de $\alpha$, $\beta$, $\gamma$ para então analisar a sua correspondência com vermelho, verde ou azul:

$$\alpha = V \times (1 - S) \tag{4.51}$$

$$\beta = \begin{cases} Indefinido,\ se\ H\ for\ indefinido \\ V \times (1 - (H - [H]) \times S),\ para\ outros\ casos \end{cases} \tag{4.52}$$

$$\gamma = \begin{cases} Indefinido,\ se\ H\ for\ indefinido \\ V \times (1 - (1 - (H - [H])) \times S),\ para\ outros\ casos \end{cases} \tag{4.53}$$

$$(R', G', B') = \begin{cases} (V, V, V), \ se\, V \ for\ indefinido \\ (V, \gamma, \alpha), \ se\, 0 \leq H > 1 \\ (\beta, V, \alpha), \ se\, 1 \leq H > 2 \\ (\alpha, V, \gamma), \ se\, 2 \leq H > 3 \\ (\alpha, \beta, V), \ se\, 3 \leq H > 4 \\ (\gamma, \alpha, V), \ se\, 4 \leq H > 5 \\ (V, \alpha, \beta), \ se\, 5 \leq H > 6 \end{cases} \tag{4.54}$$

$$(R, G, B) = (R \times 255, G \times 255, B \times 255) \tag{4.55}$$

A Figura 4.21 apresenta alguns exemplos valores *HSV* de cores e o respectivo valor *RGB*.

Figura 4.21 – Cores com valores em *HSV* e valor correspondente em *RGB*.

| | HSV | RGB |
|---|---|---|
| Preto | (0, 0.0, 20.4) | (52, 52, 52) |
| Branco | (60, 0.4, 95.3) | (243, 243, 242) |
| Vermelho | (357, 69.1, 68.6) | (175, 54, 60) |
| Verde | (122, 52.7, 58.0) | (70, 148, 73) |
| Azul | (237, 62.7, 58.8) | (56, 61, 150) |
| Laranja | (29, 79.4, 83.9) | (214, 126, 44) |

Fonte: Próprio autor.

### 4.4.5 Conversão de *RGB* para *CIELab*

A conversão de do Espaço de Cores *RGB* para *CIELAB* é realizada em duas fases. Na primeira é feita a conversão de *RGB* para *CIEXYZ* e na segunda de *CIEXYZ* para *CIELAB*.

- **Primeira fase** - Conversão de *RGB* para *CIEXYZ*: *RGB* $\rightarrow$ *CIEXYZ*

$$R' = \frac{R}{255}, G' = \frac{G}{255}, B' = \frac{B}{255}, \tag{4.56}$$

em que os valores *RGB* são redimensionados para 0 a 1.

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} 0.412453 & 0.357580 & 0.180423 \\ 0.212671 & 0.715160 & 0.072169 \\ 0.019334 & 0.119193 & 0.950227 \end{bmatrix} \times \begin{bmatrix} R' \\ G' \\ B' \end{bmatrix}, \tag{4.57}$$

em que os valores *CIEXYZ* são dimensionados para 0 a 100.

Desta forma,

$$X' = (0.412452 \times R' + 0.357580 \times G' + 0.180423 \times B') \tag{4.58}$$

$$Y' = (0.212671 \times R' + 0.715160 \times G' + 0.072169 \times B') \tag{4.59}$$

$$Z' = (0.019334 \times R' + 0.119193 \times G' + 0.950227 \times B'), \tag{4.60}$$

logo,

$$X = X' \times 100,\ Y = Y' \times 100\ e\ Z = Z' \times 100 \tag{4.61}$$

- **Segunda fase** - Conversão de *CIEXYZ* para *CIELab CIEXYZ* $\rightarrow$ *CIELab*

$$a^* = 500\ \left[ f\left(\frac{X}{X_n}\right) - f\left(\frac{Y}{Y_n}\right) \right] \tag{4.62}$$

$$b^* = 200\ \left[ f\left(\frac{Y}{Y_n}\right) - f\left(\frac{Z}{Z_n}\right) \right], \tag{4.63}$$

em que,

$$\begin{cases} f(r) = r^{\frac{1}{3}}, & se\ r\ >\ 0.008856 \\ f(r) = 7.787r + \frac{16}{116}, & se\ r\ \leq\ 0.008856 \end{cases} \tag{4.64}$$

*Y* $\rightarrow$ *L*

$$\begin{cases} L^* = 116\left(\frac{Y}{Y_n}\right)^{\frac{1}{3}} - 16, & se\ \frac{Y}{Y_n} >\ 0.008856 \\ L^* = 903.3\left(\frac{Y}{Y_n}\right), & se\ \frac{Y}{Y_n} \leq\ 0.008856 \end{cases} \tag{4.65}$$

A Figura 4.22 apresenta alguns exemplos valores *RGB* de cores e o respectivo valor *CIELab*.

Figura 4.22 – Cores com valores em *RGB* e valor correspondente em *CIELab*.

| | RGB | CIELab |
|---|---|---|
| Preto | (52, 52, 52) | (21.704, 0.001, -0.003) |
| Branco | (243, 243, 242) | (95.816, -0.171, 0.470) |
| Vermelho | (175, 54, 60) | (41.340, 49.312, 24.654) |
| Verde | (70, 148, 73) | (55.034, -40.137, 32.294) |
| Azul | (56, 61, 150) | (30.352, 26.441, -49.674) |
| Laranja | (214, 126, 44) | (61.132, 28.108, 56.131) |

Fonte: Próprio autor.

### 4.4.6 Conversão de *CIELab* para *RGB*

A conversão de do Espaço de Cores *CIELAB* para *RGB* é realizada em duas fases. Na primeira é feita a conversão de *CIELAB* para *CIEXYZ* e na segunda de *CIEXYZ* para *RGB*.

- **Primeira fase** - Conversão de *CIELab* para *CIEXYZ* $CIELab \rightarrow CIEXYZ$

$$X = f^{-1}\left(f\left(\frac{Y}{Y_n}\right) + \frac{a^*}{50}\right) X_n, \tag{4.66}$$

em que,

$$\begin{cases} f^{-1}(r) = r^3, \ se \ \frac{L^*+16}{116} > 0.206893 \\ f^{-1} = \frac{r-\frac{16}{116}}{7.787}, \ se \ \frac{L^*+16}{116} \leq 0.206893 \end{cases} \tag{4.67}$$

$L \rightarrow Y$

$$\begin{cases} Y = Y_n \left(\frac{L^*+16}{116}\right)^3, \ se \ L > 7.999625 \\ Y = Y_n \left(\frac{L^*+16}{116}\right), \ se \ L \leq 7.999625 \end{cases} \tag{4.68}$$

- **Segunda fase** - Conversão de *CIEXYZ* para *RGB* $CIEXYZ \rightarrow RGB$

$$X' = \frac{X}{100}, Y' = \frac{Y}{100}, Z' = \frac{Z}{100}, \tag{4.69}$$

em que os valores *CIEXYZ* são redimensionados para 0 a 1.

$$\begin{bmatrix} R' \\ G' \\ B' \end{bmatrix} = \begin{bmatrix} 3.240479 & -1.53715 & -0.498535 \\ -0.969256 & 1.875991 & 0.041556 \\ 0.055648 & -0.204043 & 1.057311 \end{bmatrix} \times \begin{bmatrix} X' \\ Y' \\ Z' \end{bmatrix}, \tag{4.70}$$

em que os valores *RGB* são dimensionados entre 0 a 255.

Desta forma,

$$R' = (3.240479X' - 1.537150 * Y' - 0.498535 * Z') \tag{4.71}$$

$$G' = (-0.969256 * X' + 1.875992 * Y' + 0.041556 * Z') \tag{4.72}$$

$$B' = (0.055648 * X' - 0.204043 * Y' + 1.057311 * Z'), \tag{4.73}$$

logo,

$$R = R' \times 255, \;\; G = G' \times 255 \;\; e \; B = B' \times 255 \tag{4.74}$$

A Figura 4.23 apresenta alguns exemplos valores *CIELab* de cores e o respectivo valor *RGB*.

Figura 4.23 – Cores com valores em *CIELab* e valor correspondente em *RGB*.

| | CIELab | RGB |
|---|---|---|
| **Preto** | (21.704, 0.001, -0.003) | (52, 52, 52) |
| **Branco** | (95.816, -0.171, 0.470) | (243, 243, 242) |
| **Vermelho** | (41.340, 49.312, 24.654) | (175, 54, 60) |
| **Verde** | (55.034, -40.137, 32.294) | (70, 148, 73) |
| **Azul** | (30.352, 26.441, -49.674) | (56, 61, 150) |
| **Laranja** | (61.132, 28.108, 56.131) | (214, 126, 44) |

Fonte: Próprio autor.

## 4.5 CONSIDERAÇÕES FINAIS SOBRE O CAPÍTULO

Conversões entre espaços de cores são realizadas quando existe a necessidade que determinadas operações sejam feitas em um espaço de cor específico, mas as informações da cor, que representam os dados de entrada, estão em um espaço de cores diferente. Desta forma, é necessário aplicar as fórmulas de conversão abordadas neste capítulo.

O Algoritmo de detecção de marcadores CRFM, por exemplo, utiliza operações de conversão entre espaços de cores durante o processamento das áreas de amostragem para descoberta da cor de cada bloco que compõe a hierarquia do marcador. As imagens são obtidas da câmera no espaço RGB, mas o processo de análise de cor utiliza é baseado na medição da distância de cor espaço de cores CIELab. Desta forma operações de conversão são necessárias. O capítulo seguinte abordará o Marcador CRFM, processo de descoberta de cor da sua hierarquia e as características da sua estrutura.

# 5 MARCADOR CRFM

O *Marcador Fiducial Colorido e Recursivo* (*CRFM*) é um sistema de marcadores fiduciais, que utiliza cores para compor a sua estrutura. A principal característica destes marcadores é a sua característica hierárquica, que possibilita a existência de diferentes níveis com outras hierarquias internas, dando um *design* recursivo ao marcador (TYBUSCH et al., 2017).

A estrutura do marcador, desenvolvida em conjunto, é uma abordagem importante pela utilização de cores para sua composição, visto que este tema, bem como marcadores fiduciais hierárquicos ou recursivos, parecem não ser tópicos muito explorados.

Neste capítulo serão apresentadas em detalhes as características do marcador *CRFM*, assim como composição da sua estrutura e a evolução do seu *design* desde os primeiros modelos gerados.

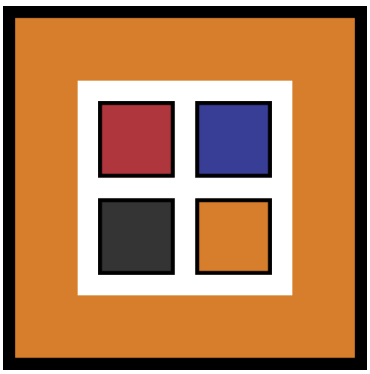
## 5.1 *DESIGN* E ESTRUTURA DO MARCADOR *CRFM*

A estrutura do *CRFM* é composta por borda e blocos internos coloridos que, juntos, compõem um identificador exclusivo (Figura 5.1). A relação entre o bloco principal e os blocos internos é chamada de hierarquia. Dependendo do número de níveis que o marcador possuir, os blocos internos também podem possuir uma hierarquia formando submarcadores, dando um *design* recursivo ao marcador, possibilitando a sua identificação em diferentes distâncias.

Para composição das hierarquias do marcador, podem ser usadas combinações entre as cinco cores selecionadas: vermelho, verde, azul, laranja e preto. Estas cores foram selecionadas 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.

A relação entre as cores que compõem uma hierarquia nunca é repetida, independentemente do número de níveis que o marcador possuir. Em qualquer hierarquia, o bloco inferior direito deve ter a mesma cor do seu bloco principal, indicando a sua orientação (Figura 5.1), possibilitando assim obter parâmetros para realizar a projeção de objetos sobre o marcador. A sequência de cores de uma hierarquia é sempre lida a partir do bloco inferior direito, identificador de posição, desta forma, para a hierarquia principal, mostrada na Figura 5.1 a sequência de cores definida é: laranja, laranja, preto, vermelho e azul.

Figura 5.1 – Marcador *CRFM* de um nível e *grid* 2x2.



Fonte: Próprio autor.

Para marcadores *CRFM* a cor da borda do bloco principal e a sequência de cores dos blocos internos não deve coincidir com as cores definidas para algum outro marcador, independentemente do nível. Isso irá possibilitar que, mesmo em casos de oclusão, localizando apenas uma hierarquia existente no marcador, seja possível reconhecer o seu identificador.

Marcadores *CRFM* com mais do que um nível apresenta uma estrutura recursiva, onde blocos internos também possuem hierarquias, formando submarcadores, de forma que sejam obedecidas regras estruturais sobre sequência de cores para composição dos identificadores únicos (Figura 5.2). A característica recursiva do marcador permite que seja repetida sucessivamente a criação de níveis internos do marcador, entretanto o número de níveis fica limitado a quantidade de combinações de cores que garantem hierarquias com identificadores únicos. Este assunto será aprofundado na Seção 6.2 que tratará sobre a geração de marcadores *CRFM* na biblioteca *CRFM Lib*.

Figura 5.2 – Marcador *CRFM* de um nível e *grid* 2x2.

Fonte: Próprio autor.

O processamento executado pelo algoritmo de identificação procura pela maior hi-

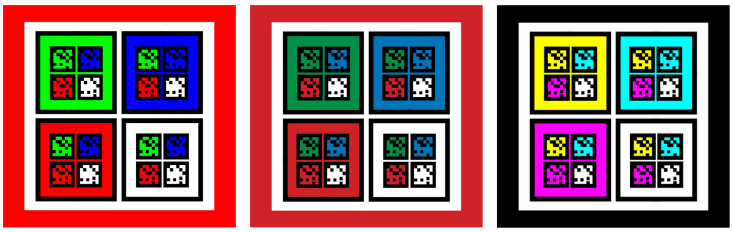
erarquia visível de cada marcador. Isso possibilita realizar uma identificação de acordo com a distância da câmera em relação ao marcador. Logo, se observado o marcador de distâncias maiores, o marcador da hierarquia superior é utilizado, mas se observado de distâncias pequenas, onde o marcador da camada superior não pode ser visualizado por completo, serão então utilizados os marcadores menores, da camada inferior, para identificação.

### 5.1.1 Evolução do *design* do marcador

O marcador *CRFM* passou por um processo de evolução até a definição do seu *design* final. Diversas alterações foram realizadas com o objetivo de melhorar o desempenho do algoritmo de detecção. No total, os marcadores, passaram por sete modificações em sua estrutura, alterando além das cores, o *layout* das hierarquias. A evolução do marcador é apresentada na listagem a seguir:

- **Primeiro *Design*** - O primeiro *design* (Figura 5.3) criado para marcadores CRFM foi fortemente baseado na estrutura do marcador *Nested Marker* (TATENO; KITAHARA; OHTA, 2006). Cada marcador existente na estrutura inicial proposta, iria usar o sistema de identificação de *ARToolkit* (KATO; BILLINGHURST, 1999), com a diferença que usaria também a cor do marcador para a composição do seu identificador. Este *design* foi alterado para um modelo em que a quantidade de elementos em cada camada fosse igual, onde sua estrutura deveria ser composta por blocos;

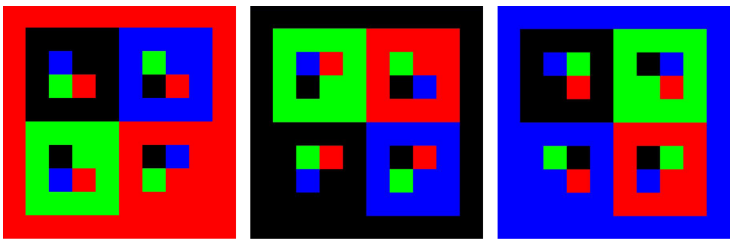
Figura 5.3 – Primeiro *design* criado para o marcador CRFM.



Fonte: Próprio autor.

- **Segundo *Design*** - O segundo *design* (Figura 5.4) criado eliminou a proposta de utilização do sistema de identificação *ARToolkit* (KATO; BILLINGHURST, 1999). Neste *design* o marcador seria composto por blocos coloridos, baseados em peças de *LEGO*. Este modelo foi alterado devido a dificuldades da sua detecção e separação das hierarquias do marcador, já que não haviam divisões entre os blocos internos;

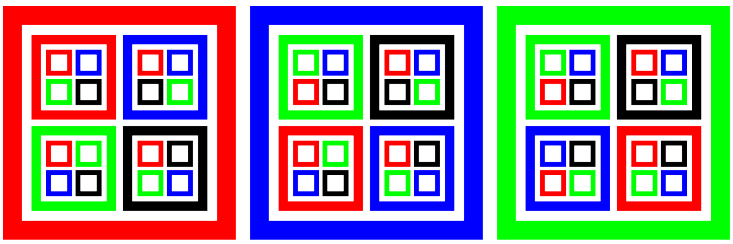
Figura 5.4 – Segundo *design* criado para o marcador *CRFM*.



Fonte: Próprio autor.

- **Terceiro *Design*** - O terceiro *design* criado (Figura 5.5), inseriu diferentes espaços entre os blocos que faziam parte da composição do marcador. Basicamente, este modelo pode ser entendido como sendo composto apenas pelas bordas, logo para identificar uma hierarquia basta identificar a borda externa e as bordas dos seus blocos internos. Este modelo foi modificado após encontrada a necessidade de as bordas possuírem maiores espessuras;

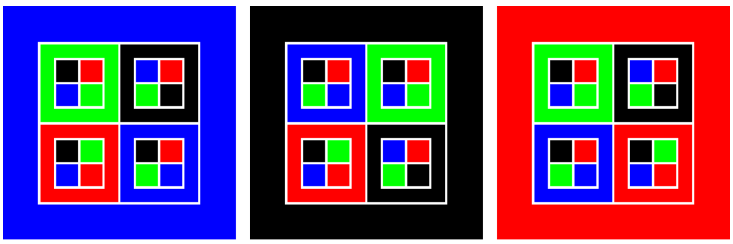
Figura 5.5 – Terceiro *design* criado para o marcador CRFM.



Fonte: Próprio autor.

- **Quarto *Design*** - O quarto *design* (Figura 5.6), introduziu o conceito do bloco identificador de posição, para localizar as rotações aplicadas ao marcador. Neste modelo os blocos mais internos eram preenchidos com a mesma cor da borda e os espaços entre os blocos foram diminuídos, aplicando o mesmo espaçamento, independentemente do nível onde as hierarquias estavam localizadas. Este modelo precisou ser modificado, devido apresentar dificuldades para identificação em distâncias maiores, devido aos espaços entre os blocos serem muito pequenos, principalmente em situações onde fosse aplicado algum ângulo de inclinação ao marcador;

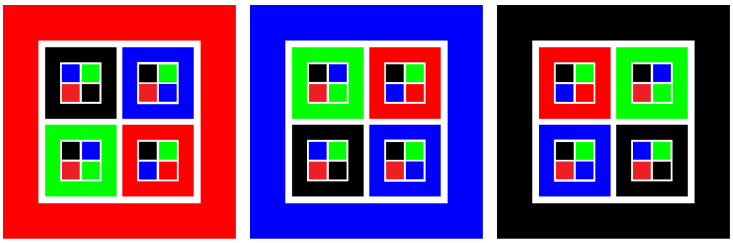
Figura 5.6 – Quarto *design* criado para o marcador CRFM.



Fonte: Próprio autor.

- **Quinto *Design*** - O quinto *design* (Figura 5.7), basicamente realizou uma modificação dos espaçamentos dos blocos do marcador proposto no quarto modelo. Os espaçamentos entre blocos e espessura das bordas passaram a ser proporcionais, com o objetivo de a percepção ser a mesma de acordo com a distância entre a câmera e o marcador. Neste modelo foi definida a utilização de quatro cores para composição da estrutura: vermelho, verde, azul e preto. Em situações com iluminação intensa sobre o marcador, as bordas de alguns blocos apresentaram erros de identificação, sendo necessário, desta forma, adicionar uma borda para cada bloco, na cor preta que apresenta bastante contraste para as operações de limiar de cor;

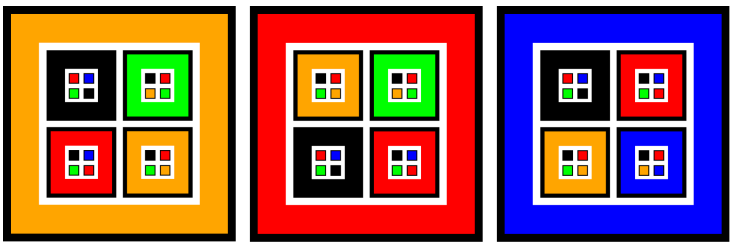
Figura 5.7 – Quinto *design* criado para o marcador CRFM.



Fonte: Próprio autor.

- **Sexto *Design*** - O sexto *design* proposto (Figura 5.8), inseriu uma borda na cor preta em cada bloco que fazia parte do marcador, visando melhorar a identificação do marcador, mesmo em distâncias maiores. Esta estratégia foi adotada devido ao fato dos modelos de marcadores anteriores apresentarem uma mistura das cores nas regiões entre blocos vizinhos, ocasionando erros de identificação da cor dos elementos. Este *design* foi modificado para ajustar os espaços entre blocos na camada mais interna do marcador, e ainda, as cores precisaram ser ajustadas para a impressão;

Figura 5.8 – Sexto *design* criado para o marcador CRFM.



Fonte: Próprio autor.

- **Sétimo *Design*** - O sétimo *design* (Figura 5.9), basicamente usa as proporções propostas no sexto design, fazendo alguns ajustes das espessuras das bordas dos blocos internos do marcador. Também, neste design, foram ajustadas as cores para a impressão, visando diminuir a diferença entre a cor gerada e a cor impressa que é analisada pelo algoritmo de detecção.

Figura 5.9 – Sétimo *design* criado para o marcador CRFM.

Fonte: Próprio autor.

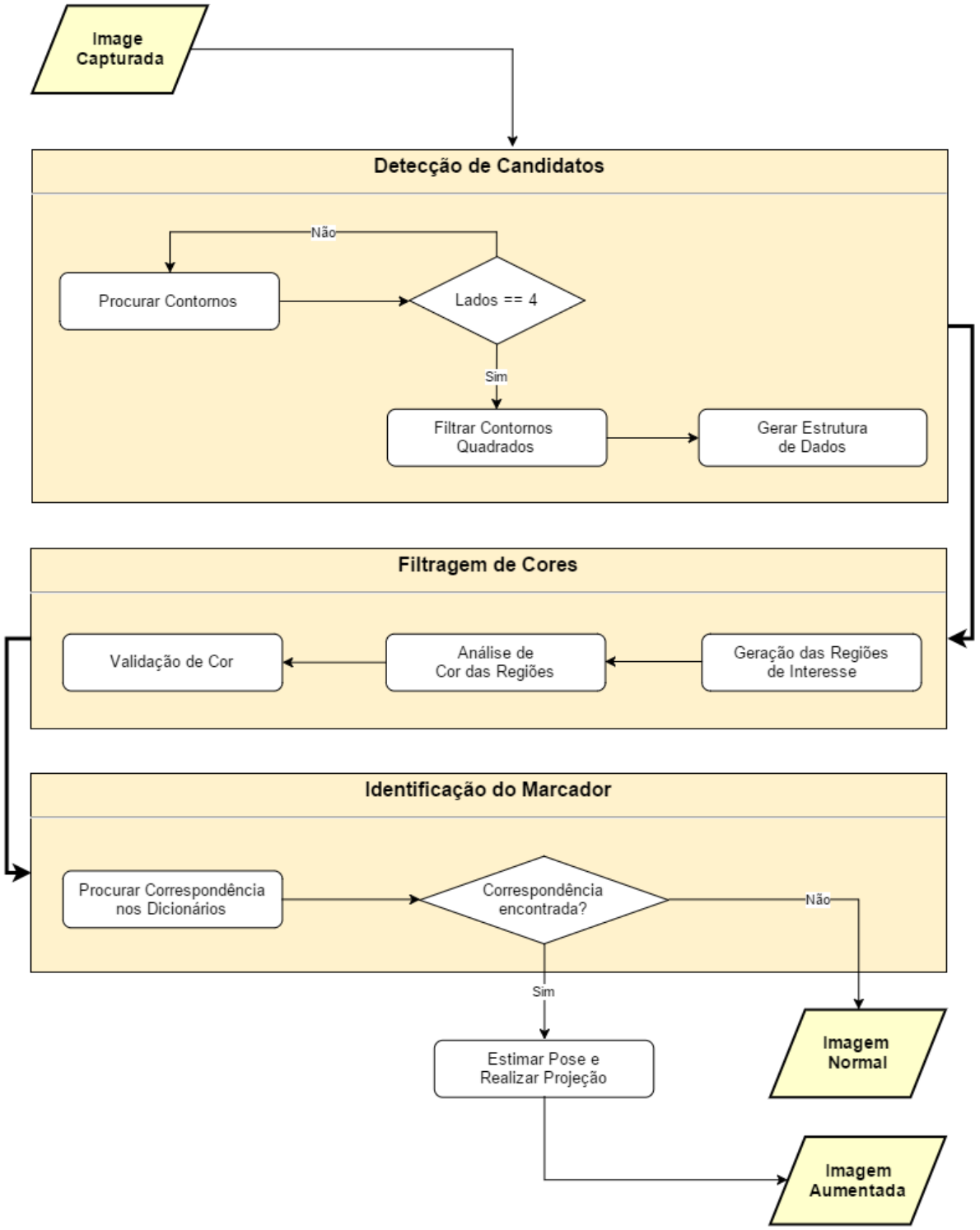
## 5.2 DETECÇÃO DE MARCADORES

O processo de detecção de marcadores CRFM inicia pala capturas da imagem pela *webcam*. Cada imagem capturada é processada para extrair informações que possam representar contornos de possíveis marcadores. O algoritmo de detecção de candidatos a marcadores de CRFM é uma versão modificada do algoritmo de detecção proposto pelo sistema de marcadores ArUco (GARRIDO-JURADO et al., 2014).

Os candidatos a marcadores encontrados no início do processo de análise das imagens capturadas passam por uma filtragem de contornos de acordo com o seu perímetro e seu tipo de contorno, para eliminar candidatos muito pequenos e que não possuam realmente quatro contornos, representado assim formas não retangulares.

A execução do algoritmo de detecção de marcadores *CRFM*, apresentado na Figura 5.10, segue um processo em três etapas principais:

Figura 5.10 – Fluxo de execução do algoritmo de detecção de marcadores CRFM.




Fonte: Próprio autor.

### 5.2.1 Detecção de candidatos

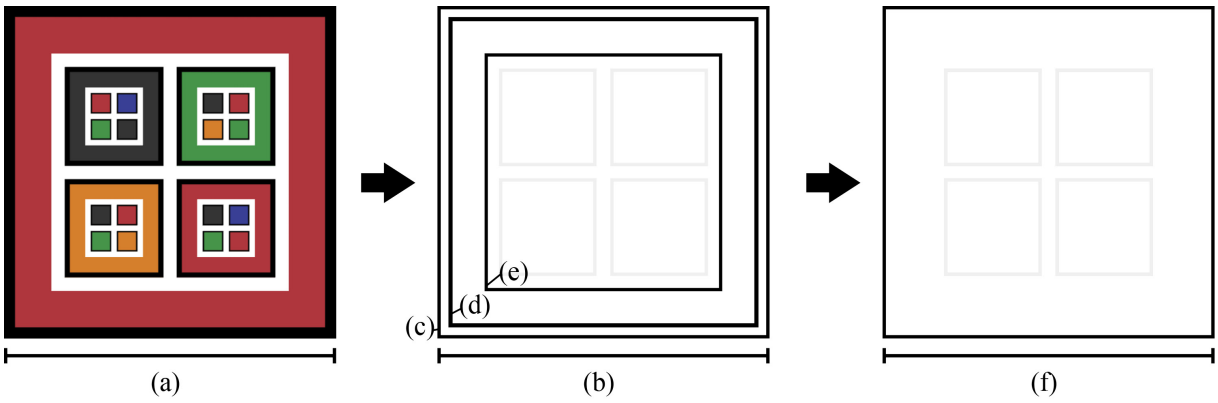
Inicialmente é realizada a localização dos contornos de elementos que representam candidatos a marcador utilizando o algoritmo proposto por Suzuki e be (1985). Também nesta etapa é aplicado o algoritmo de Douglas e Peucker (1973) para reduzir o número de vértices e então filtrar apenas os elementos que possuem contornos quadrados contendo apenas quatro vértices.

Após definidos os candidatos que possuam formas retangulares, outros filtros são também aplicados. O objetivo é a remoção de quadriláteros com perímetro muito pequeno, que acabam não sendo conclusivos para uma análise de cor, devido ao seu tamanho. Também nesta fase, são removidos contornos muito semelhantes que possam ocasionar erros na detecção. Estes erros, por exemplo, podem representar os contornos externo e interno de uma mesma borda, sendo necessário remover quaisquer contornos internos desta, mantendo apenas o externo.

Ainda nesta fase da detecção, os vértices do quadrilátero são organizados para garantir que o primeiro vértice esteja localizado no primeiro quadrante, o segundo vértice no segundo quadrante e assim sucessivamente, ordenando os índices em sentido horário a partir do primeiro vértice, visto que de acordo com o tipo de rotação aplicado ao marcador pode acontecer que algum dos vértices esteja invertido, sendo necessário a realização de um ajuste.

A Figura 5.11 exibe contornos detectados de um marcador CRFM. Em (a) é mostrado o marcador inicial que é utilizado para processamento e em (b) os contornos detectados. Na figura são destacados somente os contornos da borda principal do marcador, visto que nesse passo do algoritmo a filtragem realiza a remoção dos contornos (d) e (e), devido aos critérios de proximidade, mantendo assim, somente o contorno principal da borda do marcador (c), conforme pode ser visualizado em (f).

Figura 5.11 – Contornos detectados em um marcador *CRFM*.

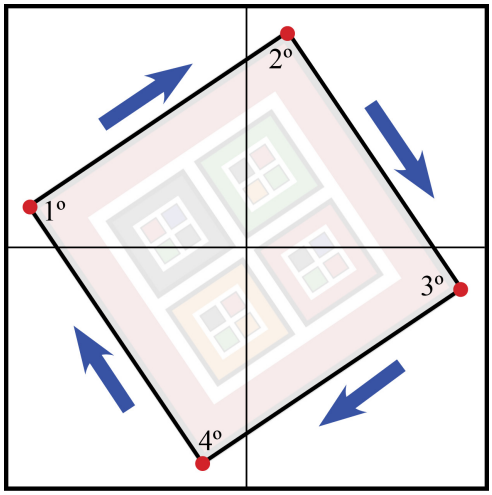



Fonte: Próprio autor.

A última etapa do processo de detecção dos marcadores candidatos, consistem em

manter apenas os quadriláteros que possuam organização em hierarquia, desta forma a partir da maior hierarquia visível realizar a análise de cor e possibilitar então o reconhecimento do marcador.

Figura 5.12 – Contornos detectados em um marcador *CRFM*.




Fonte: Próprio autor.

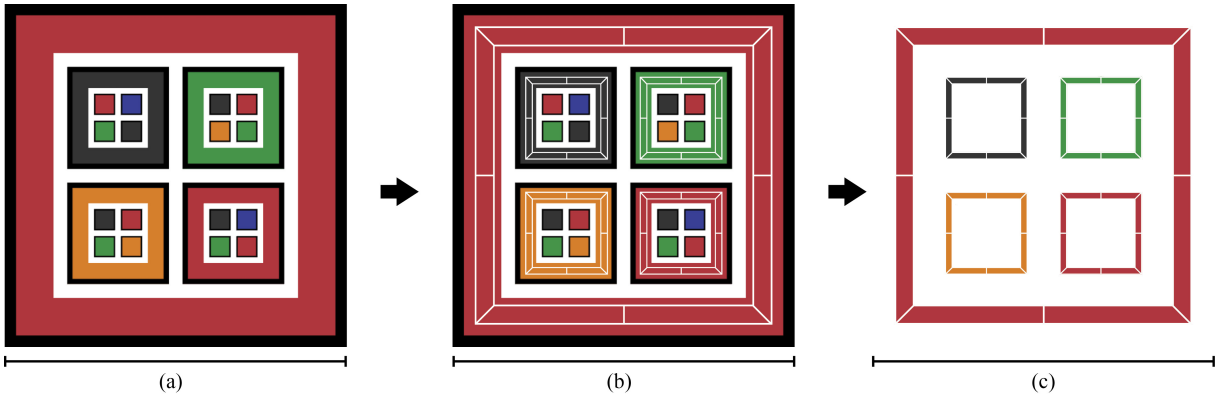
Os filtros aplicados nesta fase inicial da detecção têm o objetivo garantir que cada borda, de cada bloco que compõe o marcador, seja representada por apenas um quadrilátero, com os seus vértices ordenados, de acordo com a sua localização em cada quadrante, conforme exibido na Figura 5.12.

### 5.2.2 Filtragem de cores

Nesta etapa os candidatos são organizados em estruturas de dados para processamento e descoberta das cores por meio da análise dos *pixels* das áreas de amostragem de cada um dos quadriláteros detectados na imagem processada.

Como o marcador CRFM possui uma estreita borda preta na parte externa de cada bloco do marcador, é necessário realizar um deslocamento para que esta região não seja processada. A Figura 5.13 exibe em (a) o marcador processado, as áreas de amostragem demarcadas, com divisão em oito regiões, com deslocamento que remove a borda preta do bloco em (b) e em (c) as apenas as oito áreas de amostragem que serão analisadas.

Figura 5.13 – Áreas de amostragem definidas.



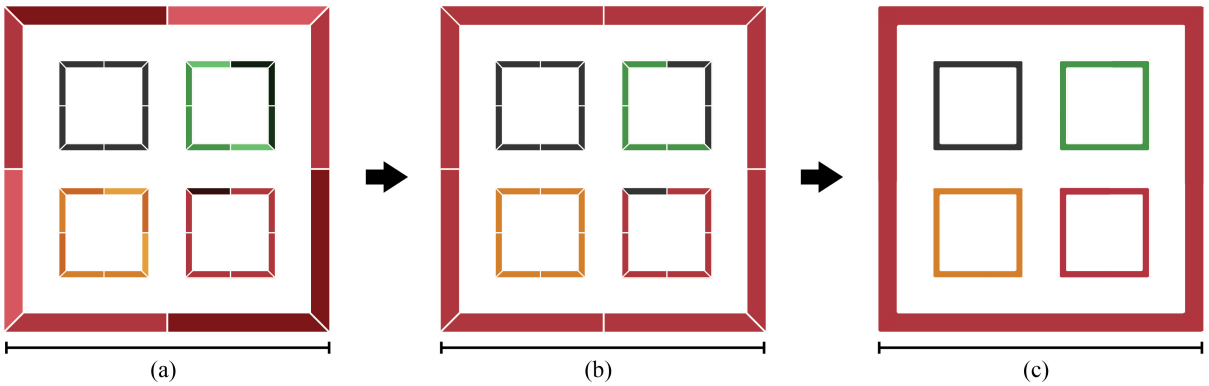

Fonte: Próprio autor.

Após as definições das áreas de amostragem é realizado o cálculo do valor médio RGB de cada uma das 8 áreas de amostragem definidas para cada quadrilátero (Figura 5.14 (a)). As cores alvo podem ser cores obtidas com o processo de correspondência de cores, que obtém a forma como a cor é medida de acordo com as condições da cena, definidos na seção 6.3. O valor RGB médio de cada área é então convertido para o espaço de cores *CIELab* e calculada a distância de cores através da fórmula $\Delta E^*_{2000}$, definida na Seção 4.3.2.3 do Capítulo sobre Padrões e Operações com Cores.

Através da aplicação da fórmula $\Delta E^*_{2000}$, utilizando o algoritmo adaptado de Fiumara (2017), é possível verificar a cor que representa a menor distância entre o valor médio calculado da região e cada uma das cores alvo, que podem compor o marcador. A alvo que resultar na menor distância é definida como sendo a cor mais similar à região comparada (Figura 5.14 (b)).

Com a definição da cor para cada uma das oito áreas de amostragem do quadrilátero é necessário definir a cor final do quadrilátero. Será eleita como a cor que representa o quadrilátero, aquela cor alvo que for identificada pelo menos metade do número das áreas de amostragem, ou seja, em quadro áreas (Figura 5.14 (c)). Caso nenhuma cor seja localizada pelo menos quatro vezes, então o quadrilátero tem sua cor definida como indeterminada, evitando assim detecções imprecisas.

Figura 5.14 – Definição das cores das áreas de amostragem.




Fonte: Próprio autor.

As áreas de amostragem definidas nesta região podem ser comparadas às cores definidas no processo de correspondência de cores da ferramenta *CRFM Lib*, estabelecido na Seção 6.3.

### 5.2.3 Identificação de marcadores

Após a filtragem e definição das cores dos elementos do marcador, é realizada a sua identificação, buscando correspondências entre o dicionário do marcador alvo e as sequências de cores detectadas nas imagens processadas. Dicionários de marcadores são abordados na Seção 6.2.1.

Caso exista uma correspondência entre o marcador alvo e o marcador localizado na cena, pode ser realizada a projeção de objetos virtuais. A realização da projeção necessita que sejam identificadas as informações referentes a orientação dos marcadores alvo relacionados, com base na imagem da cena capturada, para criar valores de rotação e translação para o objeto que deverá ser aumentado.

Para estimativa da pose da câmera é usado o método dos mínimos quadrados de *Levenberg-Marquardt* (LEVENBERG, 1944) (MARQUARDT, 1963), definido na função *cv::solvePNP()*, da *OpenCV* (OPENCV, 2017). A utilização deste método, mapeia pontos *3D* no mundo, para pontos *2D* localizados na imagem capturada e desta forma obtêm os parâmetros relacionados a rotação e translação.

A projeção de objetos virtuais é abordada na Seção 6.4.2, no capítulo sobre a ferramenta *CRFM Lib*, mostrando como são aplicados os parâmetros mapeados.

# 6 *CRFM LIB FOR UNITY*

Neste trabalho foi desenvolvida uma ferramenta para criação de aplicações de Realidade Aumentada, utilizando os marcadores CRFM. O objetivo da ferramenta é servir como uma ferramenta integrada ao Unity, que irá prover funcionalidades para reconhecimento de marcadores fiduciais e realizar projeção de objetos tridimensionais com base nas informações obtidas pela identificação dos alvos rastreáveis.

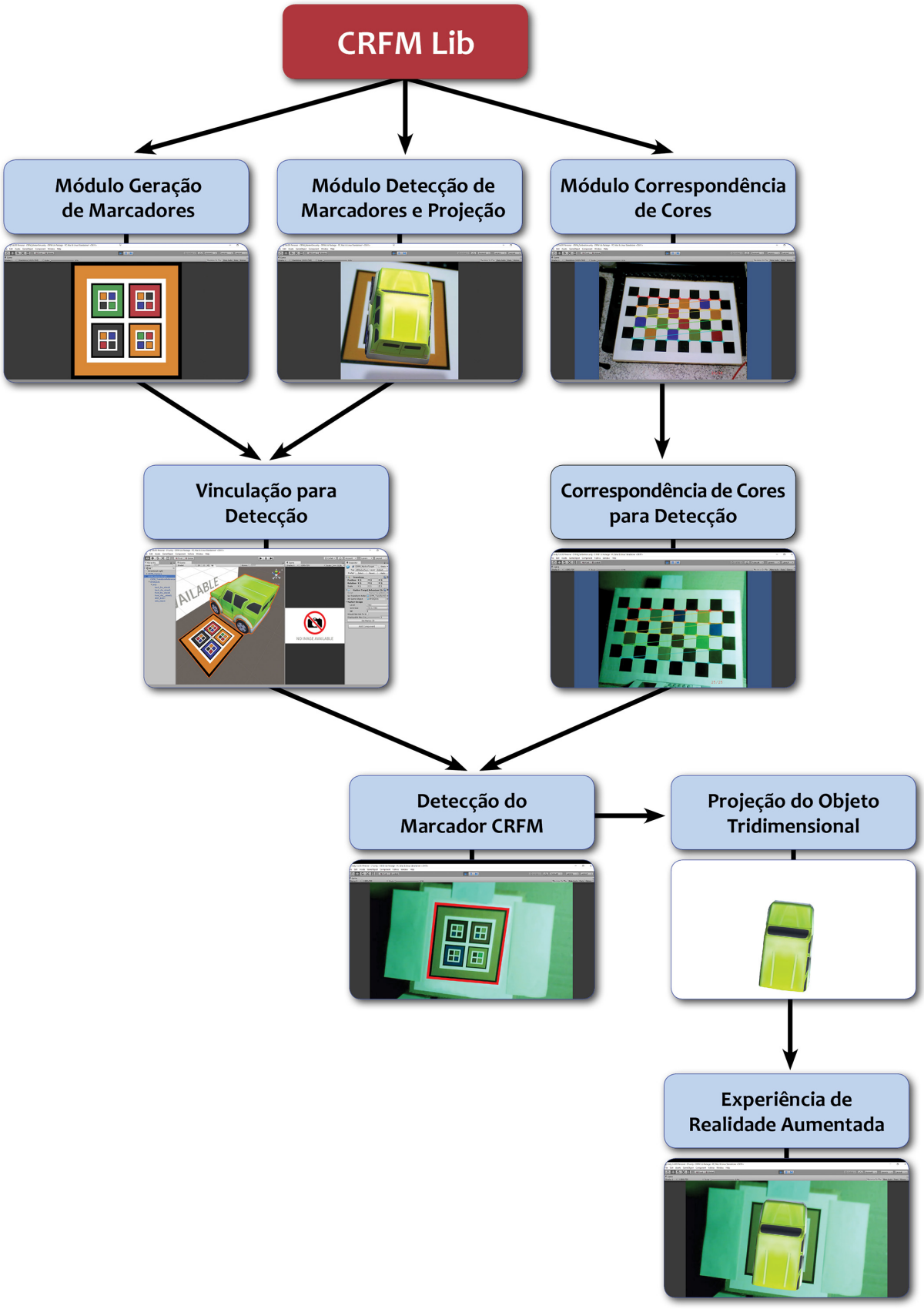
A ferramenta *CRFM Lib* é estruturada em três módulos, conforme mostrado na Figura 6.1. O primeiro módulo possibilita realizar a geração dos marcadores *CRFM* com base em suas regras estruturais e com isso construir um dicionário de marcadores. Estes marcadores poderão ser vinculados para realização da sua detecção na cena. O segundo módulo possibilita a realização de uma análise das cores de uma placa de correspondência, para identificar como as cores de interesse da aplicação são percebidas na cena, de acordo com as variações das condições de iluminação. E, finalmente, o terceiro módulo provê ferramentas para a criação de aplicações de Realidade Aumentada, com identificação e rastreamento dos marcadores *CRFM* gerados no primeiro módulo, através do processamento e identificação das cores da estrutura interna do marcador. Considerando as condições das cores percebidas na cena obtidas no segundo módulo. A relação entre os módulos da ferramenta é mostrada na Figura 6.1.

Para criação da ferramenta é realizada uma integração da biblioteca de Visão Computacional *OpenCV* com *Unity Game Engine*. *OpenCV* possui versões para utilização com *C++*, *C*, *Python* e *Java*, entretanto não possui, de forma nativa, uma versão disponível para as linguagens utilizadas na *Unity* com as arquiteturas *Mono* e *.NET Framework*. Desta forma, para utilizar as funcionalidades de *OpenCV* na *Unity* é necessário implementar mecanismos para acessar funções nativas escritas em *C++*, através das linguagens suportadas pela *Unity*.

Existem implementações disponíveis para utilização dos algoritmos de Visão Computacional fornecidos pela *OpenCV*, mas de forma paga, como é o caso da *OpenCVForUnity* (ENOXSOFTWARE, 2017), onde um pacote de ferramentas para a Game Engine é adquirido, contendo as principais funções da biblioteca. Para este pacote de ferramentas as principais funções da biblioteca são reescritas em *C#* e disponibilizadas para venda na loja de pacotes da *Unity*.

Devido a necessidade de compatibilidades entre *OpenCV* e *Unity* é necessária a criação de *plugins* que permitam interoperabilidade entre as linguagens. Para *CRFM Lib* foi desenvolvido um *plugin* escrito em *C++*, que possibilita a utilização das funcionalidades de *OpenCV* necessárias para a criação de aplicações de RA na *Unity* utilizando o marcador *CRFM* como alvo na cena.

Figura 6.1 – Relação entre os módulos da ferramenta CRFM Lib.




Fonte: Próprio autor.

## 6.1 INTEROPERABILIDADE ENTRE *C#* E *C++*

Para criar interoperabilidade entre o plugin *C++* e *C#*, utilizada em projetos da *Unity Game Engine*, devem ser escritos os métodos, dentro do projeto *C++*, que realizarão as operações de processamento de imagens a partir da *OpenCV*. Para isso, todos os métodos que podem ser chamados de dentro da *Unity* precisam estar explicitamente expostos, através de definições na assinatura de cada um destes métodos.

Normalmente, o compilador *C++* irá esconder os nomes dos métodos ao construir a *dll*. Portanto, é necessário usar um estilo de assinatura para deixar os nomes dos métodos disponíveis exatamente como foram escritos. Todos os métodos *C++* que serão expostos a um aplicativo gerenciado precisarão utilizar uma sintaxe que evidencie a forma como estarão visíveis.

Figura 6.2 – Código-fonte *C++*: Pseudocódigo da assinatura de um método exposto.

```
1 extern "C" inline __declspec(dllexport)
2     tipoRetorno __stdcall nomeMetodo(tipoParametro parametro)
3 {
4     ...
5 }
```

Fonte: Próprio autor.

A Figura 6.2 mostra o exemplo de pseudocódigo de uma assinatura de método que é exposto para ser chamado a partir de outras aplicações, através da referência à sua *dll* gerada.

Após a escrita de todos os métodos que devem ser expostos, é necessário realizar a construção da *dll*, através da compilação do projeto C++, no *Microsoft Visual Studio*. Após a construção, todas os arquivos *.dll* resultantes precisam ser copiadas para a pasta de *plugins* do projeto Unity, juntamente com todos os arquivos .dll e .lib da *OpenCV*.

No projeto *Unity* deverá ser criada uma classe *C#* estática para referenciar todos os métodos *C++* explicitamente expostos através da definição da assinatura de cada método. É necessário observar que tipo de retorno, nome do método e tipos de parâmetros, contidos na classe estática, devem corresponder aos métodos *C++*. O atributo *DllImport* leva o nome do arquivo da *.dll* referenciado, conforme pode ser visualizado na Figura 6.3.

Figura 6.3 – Código-fonte *C#*: Referência na classe *C#* aos métodos *C++* expostos.

```
internal static class OpenCVInterop
{
    [DllImport("NomeDll", CharSet = CharSet.Unicode, EntryPoint = "nomeMetodo",
        CallingConvention = CallingConvention.Cdecl)]
    public static extern tipoRetorno nomeMetodo(tipoParametro parametro);
}
```

Fonte: Próprio autor.

## 6.2 MÓDULO DE GERAÇÃO DE MARCADORES

Os marcadores *CRFM* são pré-gerados em uma aplicação desenvolvida (Figura 6.12) e então usados como dicionários dentro do algoritmo de identificação. O algoritmo de geração funciona seguindo os passos:

1. Gerar todas as sequências possíveis (arranjos) de acordo com o número de cores selecionadas, seguindo algumas regras de geração;
    - O bloco pai e o primeiro bloco interno (inferior direita) devem possuir a mesma cor;
    - Outros blocos da mesma sequência não devem repetir as cores do bloco pai e do bloco inferior direito;
    - Não é realizada nenhuma repetição de sequência pai-filho, mesmo em níveis diferentes;

Figura 6.4 – Código-fonte *C#*: Geração dos arranjos de possíveis de um nível.

```
//Enumerador que representa as cores utilizadas para compor o marcador
[Serializable]
public enum ColorBWRGB : int
{
   [StringValue("BLACK")]
   BLACK = 1,

   [StringValue("RED")]
   RED = 2,

   [StringValue("GREEN")]
   GREEN = 3,

   [StringValue("BLUE")]
   BLUE = 4,

   [StringValue("ORANGE")]
   ORANGE = 5,
}
```

Fonte: Próprio autor.

A Figura 6.4 mostra os enumeradores que representam as cores que são utilizadas para geração dos arranjos possíveis de marcadores de um nível, que são utilizados para geração de outros níveis de marcadores *CRFM*, conforme exibido na Figura 6.5. A geração dos arranjos possíveis é realizada no método *GenerateArrangements()*, utilizando uma lista com as cores de interesse.

Figura 6.5 – Código-fonte *C#*: Geração dos arranjos de possíveis de um nível.

```
//Gera uma lista com todos os arranjos possíveis
private List<ColorNodeGen> GenerateArrangements()
{
   //Cria uma lista com todas as cores do enumerador
   List<ColorBWRGB> list = EnumToList<ColorBWRGB>();

   //Cria uma lista para preencher com os arranjos possíveis
   List<ColorNodeGen> listColorNodeGen = new List<ColorNodeGen>();

   for (int i = 0; i < list.Count; i++)
   {
      for (int j = 0; j < list.Count; j++)
      {
         for (int k = 0; k < list.Count; k++)
         {
            for (int l = 0; l < list.Count; l++)
            {
               if (i == j || i == k || i == l || j == k || j == l || k == l)
                  continue;
               listColorNodeGen.Add(
                       new ColorNodeGen(
                           list[i],
                           list[j],
                           list[k],
                           list[l],
                           list[i]
                        )
                    );
            }
         }
      }
   }
}
```

Fonte: Próprio autor.

2. Selecionar cada sequência válida aplicando-a na geração de cada marcador em cada nível e removendo as sequências equivalentes da lista de arranjos restantes;

   - Cada sequência deve ser única e não repetida, portanto, após a inserção, deve-se iterar e remover sequências equivalentes da lista restante.

3. Gerar o dicionário equivalente do marcador;

   As Figuras 6.6, 6.7, 6.8 exibem exemplos de dicionários de marcadores gerados

neste módulo. Dicionários de marcadores serão abordados em detalhes na Seção 6.2.1.

Figura 6.6 – Código-fonte *C#*: Exemplo de dicionário com 1 nível.

```
private static ColorBWRGB[,] DICT_CRFM_LVL_1 = new ColorBWRGB[,]
{
   {
      ColorBWRGB.BLACK,
      ColorBWRGB.BLACK,
      ColorBWRGB.BLUE,
      ColorBWRGB.RED,
      ColorBWRGB.GREEN
   },
   ...
};
```

Fonte: Próprio autor.

Figura 6.7 – Código-fonte *C#*: Exemplo de dicionário com 2 níveis.

```
private static ColorBWRGB[,] DICT_CRFM_LVL_2 = new ColorBWRGB[,]
{
   {
      ColorBWRGB.BLACK, ColorBWRGB.BLACK,
      ColorBWRGB.BLACK, ColorBWRGB.ORANGE,
      ColorBWRGB.RED, ColorBWRGB.GREEN,
      ColorBWRGB.BLUE, ColorBWRGB.BLUE,
      ColorBWRGB.GREEN, ColorBWRGB.BLACK,
      ColorBWRGB.RED, ColorBWRGB.RED,
      ColorBWRGB.RED, ColorBWRGB.BLUE,
      ColorBWRGB.BLACK, ColorBWRGB.GREEN,
      ColorBWRGB.GREEN, ColorBWRGB.GREEN,
      ColorBWRGB.BLUE, ColorBWRGB.BLACK,
      ColorBWRGB.RED
   },
   ...
};
```

Fonte: Próprio autor.

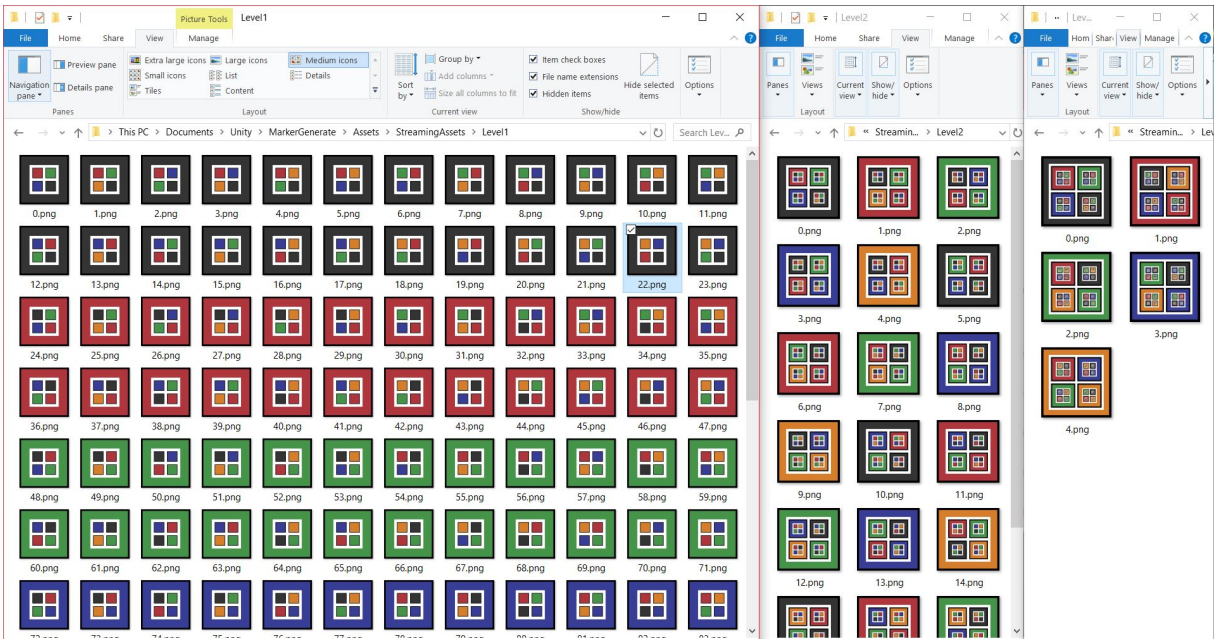
Figura 6.8 – Código-fonte *C#*: Exemplo de dicionário com 3 níveis.

```
private static ColorBWRGB[,] DICT_CRFM_LVL_3 = new ColorBWRGB[,]
{
   {
      ColorBWRGB.BLACK, ColorBWRGB.BLACK,
      ColorBWRGB.BLACK, ColorBWRGB.BLACK,
      ColorBWRGB.BLUE, ColorBWRGB.RED,
      ColorBWRGB.ORANGE, ColorBWRGB.ORANGE,
      ColorBWRGB.ORANGE, ColorBWRGB.GREEN,
      ColorBWRGB.BLACK, ColorBWRGB.RED,
      ColorBWRGB.RED, ColorBWRGB.RED,
      ColorBWRGB.GREEN, ColorBWRGB.BLACK,
      ColorBWRGB.ORANGE, ColorBWRGB.GREEN,
      ColorBWRGB.GREEN, ColorBWRGB.RED,
      ColorBWRGB.BLACK, ColorBWRGB.ORANGE,
      ColorBWRGB.BLUE, ColorBWRGB.BLUE,
      ColorBWRGB.BLUE, ColorBWRGB.ORANGE,
      ColorBWRGB.BLACK, ColorBWRGB.GREEN,
      ColorBWRGB.GREEN, ColorBWRGB.GREEN,
      ColorBWRGB.ORANGE, ColorBWRGB.BLACK,
      ColorBWRGB.BLUE, ColorBWRGB.BLACK,
      ColorBWRGB.BLACK, ColorBWRGB.GREEN,
      ColorBWRGB.RED, ColorBWRGB.ORANGE,
      ColorBWRGB.RED, ColorBWRGB.RED,
      ColorBWRGB.ORANGE, ColorBWRGB.BLACK,
      ColorBWRGB.BLUE, ColorBWRGB.GREEN,
      ColorBWRGB.GREEN, ColorBWRGB.GREEN,
      ColorBWRGB.RED, ColorBWRGB.BLACK,
      ColorBWRGB.BLUE, ColorBWRGB.BLUE,
      ColorBWRGB.BLUE, ColorBWRGB.RED,
      ColorBWRGB.BLACK, ColorBWRGB.GREEN,
      ColorBWRGB.BLACK, ColorBWRGB.BLACK,
      ColorBWRGB.ORANGE, ColorBWRGB.RED,
      ColorBWRGB.BLUE, ColorBWRGB.RED,
      ColorBWRGB.RED, ColorBWRGB.GREEN,
      ColorBWRGB.BLACK, ColorBWRGB.BLUE,
      ColorBWRGB.RED, ColorBWRGB.RED,
      ColorBWRGB.RED, ColorBWRGB.ORANGE,
      ColorBWRGB.BLACK, ColorBWRGB.GREEN,
      ColorBWRGB.BLUE, ColorBWRGB.BLUE,
      ColorBWRGB.ORANGE, ColorBWRGB.BLACK,
      ColorBWRGB.RED, ColorBWRGB.BLACK,
      ColorBWRGB.BLACK, ColorBWRGB.GREEN,
      ColorBWRGB.RED, ColorBWRGB.BLUE,
      ColorBWRGB.GREEN, ColorBWRGB.GREEN,
      ColorBWRGB.ORANGE, ColorBWRGB.BLACK,
      ColorBWRGB.RED
   },
   ...
};
```

Fonte: Próprio autor.

4. Gerar imagem do marcador;

Figura 6.9 – Definição das cores das áreas de amostragem .




Fonte: Próprio autor.

A Figura 6.9 mostra marcadores de 1, 2 e 3 níveis gerados no módulo, já na Figura 6.10 é mostrado o método *SaveMarkers()* utilizado para a geração das imagens dos marcadores.

Figura 6.10 – Código-fonte *C#*: Geração das imagens dos marcadores.

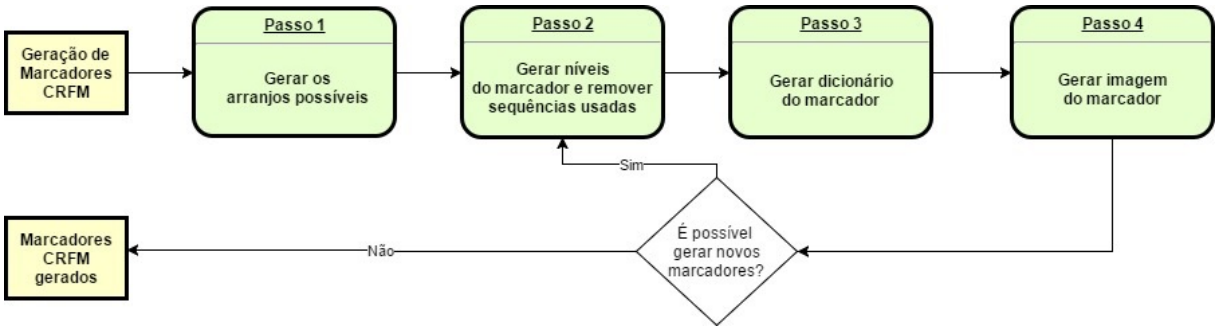
```
//Salvar todos os marcadores gerados como imagens
private void SaveMarkers()
{
   var children = goCanvas.GetComponentsInChildren<Transform>();
   //Inativa todos os gameobject gerados no painel Hierarchy
   foreach (var c in children)
   {
      if (c.parent != null)
         if (c.parent.gameObject == goCanvas.gameObject)
            c.gameObject.SetActive(false);
   }
   /*
      Inicia uma corrotina para ativar cada gameobject que
      corresponde a um marcador gerado e salvar a sua imagem
   */
   StartCoroutine(ChangeStateGameObjects(children, true, 0.3f));
}

//Ativa cada gameobject que corresponde a um marcador e salva a imagem
IEnumerator ChangeStateGameObjects(Transform[] children, bool state, float seconds)
{
   int iSpriteCount = 0;
   foreach (var c in children)
   {
      if (c.parent != null)
      {
         if (c.parent.gameObject == goCanvas.gameObject)
         {
            c.gameObject.SetActive(state);
            if (state)
               SaveImage(iSpriteCount++);
            yield return new WaitForSeconds(seconds);
         }
      }
   }
   yield return null;
}

//Salvar as imagens dos marcadores gerados
private void SaveImage(int idMarker)
{
   string path = Application.dataPath + "/StreamingAssets/Level" +
                 Convert.ToInt16(enumLevels) + "/";
   try
   {
      //Cria o diretório se não existe
      if (!Directory.Exists(path)) Directory.CreateDirectory(path);
   }
   catch (IOException ex)
   {
      Debug.Log("Erro ao criar pasta. " + ex.Message);
   }
   //Salva a imagem do marcador
   Application.CaptureScreenshot(String.Format("{0}{1}.png", path, idMarker));
}
```

Fonte: Próprio autor.

5. Repetir os passos 2 até 4, de acordo com o número marcadores possíveis.

O diagrama apresentado na Figura 6.11 mostra os passos listados que são seguidos para geração de marcadores *CRFM*.

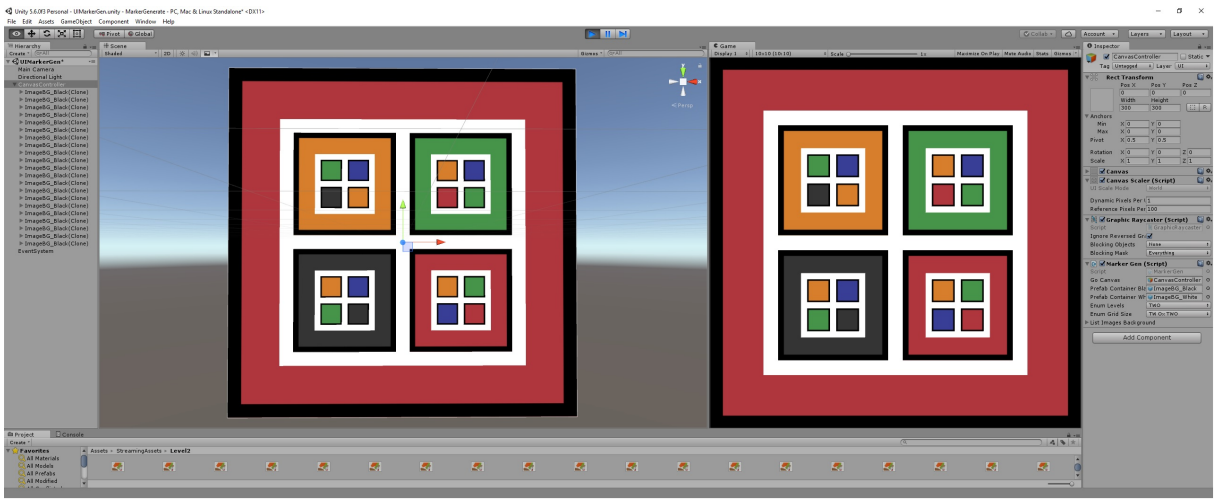
Figura 6.11 – Fluxo da geração do *CRFM*.




Fonte: Próprio autor.

Os marcadores gerados na *Unity Game Engine* (Figura 6.12) são salvos nos diretórios estruturados do projeto, onde poderão ser escolhidos como elementos de referência no módulo de detecção e rastreamento do *CRFM Lib*, podendo ainda ser impressos em papel branco comum para utilização como alvos nas aplicações de Realidade Aumentada construídas com a biblioteca.

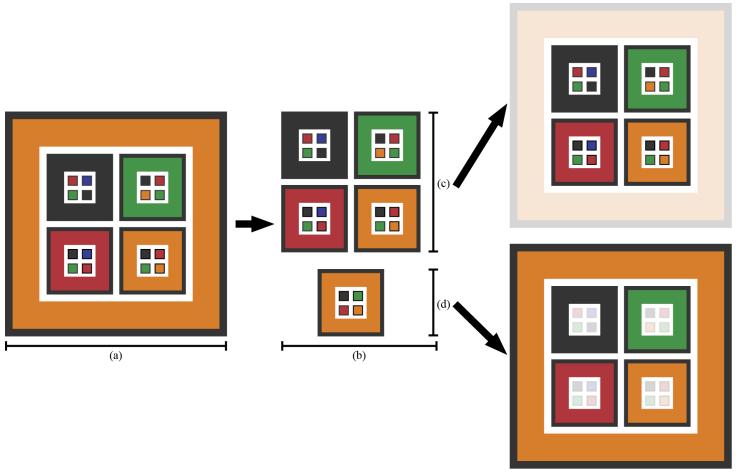
Figura 6.12 – Geração dos marcadores *CRFM* na *Unity Game Engine*.



Fonte: Próprio autor.

A Figura 6.13 exibe a composição de um marcador *CRFM* com dois níveis de hierarquia (a) e tamanho de *grid* de 2x2, com as regras de geração aplicadas. Em (b) são exibidos os 5 marcadores de um nível que compõem a sua estrutura, onde quatro destes formam o *grid* interno (c) e um representa o arranjo considerando apenas a borda principal do marcador e as bordas dos elementos do *grid* interno (d).

Figura 6.13 – Composição do marcador *CRFM*.

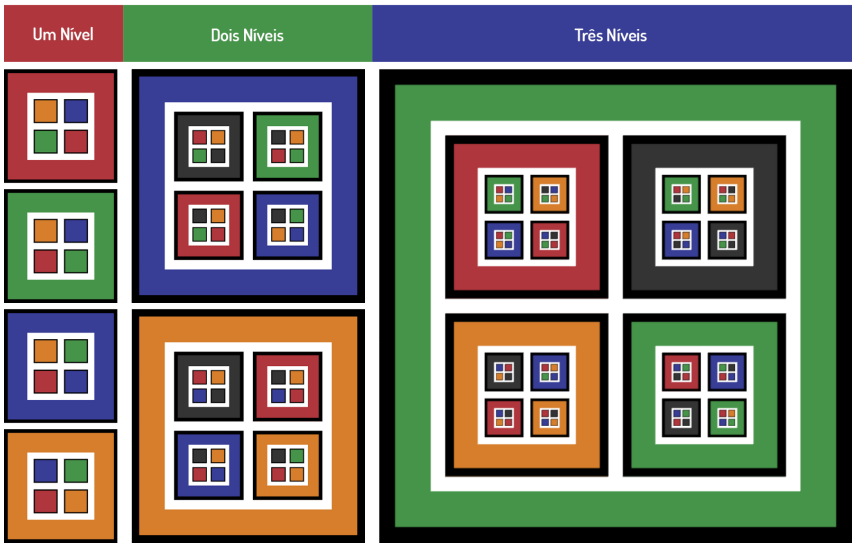



Fonte: Próprio autor.

### 6.2.1 Dicionários de Marcadores

A configuração de um marcador *CRFM* é definida com base com o tamanho do seu *grid* interno e a quantidade de níveis de sua estrutura. A partir da geração dos marcadores, é construído um dicionário de marcadores específicos de acordo com a quantidade de níveis para ser utilizado pelo algoritmo de detecção. Para este trabalho foram gerados três dicionários, para marcadores de um nível, dois níveis e três níveis. Exemplos de marcadores de cada nível podem ser visualizados na Figura 6.14. Os marcadores gerados também ficam disponíveis em arquivos de imagem para impressão.

Figura 6.14 – Exemplos de marcadores gerados.

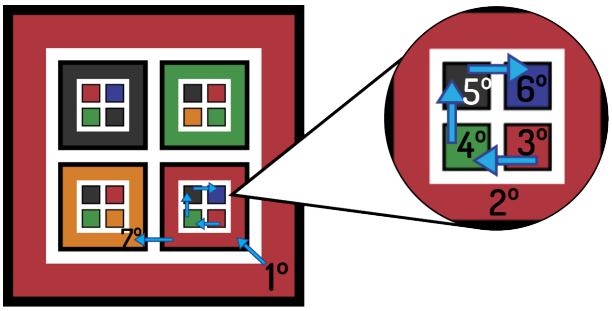



Fonte: Próprio autor.

O módulo de detecção do *CRFM Lib* irá se basear nos dicionários gerados para identificar o cada marcador exposto na cena. Por isso existe uma padronização de valores

que correspondem a estrutura recursiva do marcador (Figura 6.16).

Figura 6.15 – Sentido de geração de um dicionário de marcador.

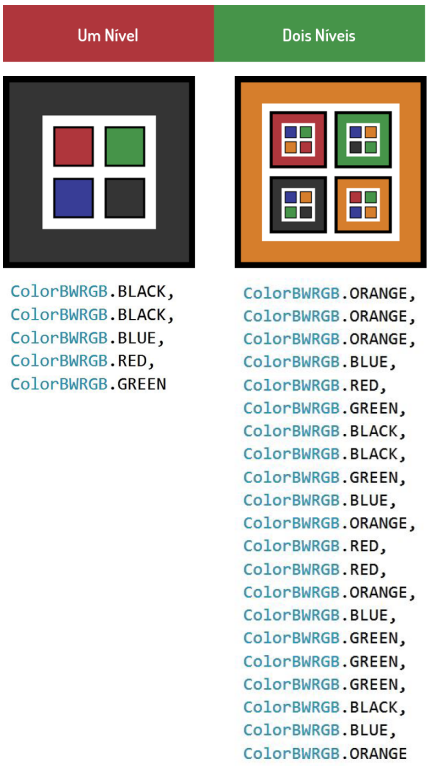



Fonte: Próprio autor.

Para formar um dicionário que corresponde a estrutura do marcador, os valores que identificam cada cor são adicionar a um *array* seguindo o sentido horário. A representação inicia pela borda mais externa, seguindo pelo bloco identificador de posição, até a camada mais interna do marcador, conforme mostrado na Figura 6.15.

A Figura 6.16 mostra marcadores *CRFM* e o seu dicionário correspondente.

Figura 6.16 – Exemplo de itens de dicionários de marcadores gerados.




Fonte: Próprio autor.

### 6.2.2 Quantidade de marcadores possíveis

Devido ao fato de cada marcador ser composto por uma borda e um *grid* com elementos internos, é necessário que um dos elementos internos seja o identificador de posição do marcador, recebendo a mesma cor da borda, para que o algoritmo de identificação, reconheça o marcador mesmo com algum tipo de rotação aplicada.

A existência direta da relação da cor do elemento identificador de posição com a borda do marcador, possibilita que geração considere apenas a quantidade de elementos do *grid* interno, definido pela aplicação da fórmula de Arranjo Simples, onde cada marcador é caracterizado pela natureza e pela ordem dos elementos envolvidos.

A equação 6.1 define a geração básica das combinações possíveis para formação de marcadores de um nível que podem ser usados para formação dos marcadores com quantidades maiores de níveis.

$$A = \frac{N!}{(N-P)!},\ de\ forma\ que\ P = V * H, \tag{6.1}$$

em que, *N* é a quantidade de cores que formam o conjunto e *P* é quantidade de cores que deverá formar cada agrupamento, calculado com base na quantidade de elementos verticais (*V*) e horizontais (*H*) que compõem o *grid* do marcador. Logo, para o *CRFM*, *P* pode ser tratado como o tamanho do *grid* interno do marcador.

A equação 6.2 mostra como é calculada a quantidade de marcadores de um nível são necessários para compor marcadores de níveis maiores:

$$q = \sum_{i=1,q_{Ant}=1}^{i<L} q_{Ant},\ de\ forma\ que\ q_{Ant} = (q_{Ant} * P) + 1, \tag{6.2}$$

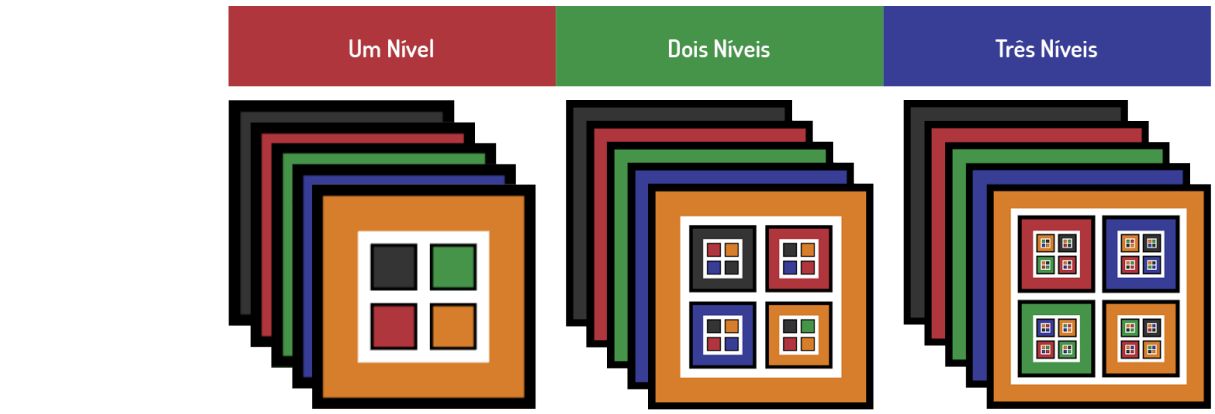
em que, o cálculo é realizado somando os marcadores dos níveis inferiores até o nível mais externo representado por *L*. Desta forma, para encontrar o número de marcadores possíveis de acordo com a quantidade de níveis (*L*) e tamanho do *grid* (*P*) basta aplicar a equação 6.3:

$$Q = \frac{\frac{N!}{(N-P)!}}{\sum_{i=1,q_{Ant}=1}^{i<L} q_{Ant},\ de\ forma\ que\ q_{Ant} = (q_{Ant} * P) + 1} \tag{6.3}$$

Para a composição de cada marcador de um nível, por exemplo, com tamanho de grid 2x2 são utilizadas quatro cores para formar os cinco 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 cinco cores selecionadas, são possíveis 120 marcadores.

A Figura 6.17 define a quantidade de marcadores possíveis de acordo com a quantidade de níveis configurada para cada dicionário de marcadores, utilizando cinco cores.

Figura 6.17 – Número de marcadores possíveis.




| | 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.

É necessário considerar a quantidade de níveis para a criação dos marcadores, pois o número de marcadores diminui, para cada nível adicional. Cada nível utiliza arranjos de marcadores de um nível, para a sua configuração, desta forma, para um dicionário de marcadores de dois níveis e tamanho de *grid* 2x2, por exemplo, são usados quatro marcadores de um nível para compor o nível com os blocos mais internos. Entretanto, é necessário observar que as bordas destes blocos internos e a borda principal do marcador formam um quinto marcador de um nível, resultando, desta forma, na necessidade da existência de cinco marcadores de um nível para compor a estrutura do alvo.

O módulo de Geração de Marcadores utiliza cinco cores para criar as combinações de elementos que compõem a estrutura do marcador. Este número se deve à dificuldade de identificação das cores em determinadas condições de iluminação. Como, por exemplo, ciano, que pode ser confundido com azul em uma cena pouco iluminada. Desta forma, a quantidade de níveis abordados no módulo fica limitada a três, entretanto, com a utilização de uma quantidade maior de cores, é possível a geração de marcadores com mais níveis.

## 6.3 MÓDULO DE CORRESPONDÊNCIA DE CORES

Nenhuma câmera interpreta as cores da mesma forma que outra, desta forma isso representa problema para a aplicação de detecção do marcador CRFM. Logo, o processo de correspondência de cores que será realizado, irá criar de referência das cores de interesse, usadas para definição do marcador CRFM, de acordo com as condições variadas

de cada cena processada.

Cada marcador gerado, quando visualizado diretamente em um *display* de algum dispositivo, os tons de cores apresentados, são diferentes daqueles que são visualizados na mesma imagem quando impressa. A Figura 6.18 apresenta de forma clara a diferença entre um marcador gerado com as cores alvo sem nenhuma alteração (A) e um marcador que foi impresso, sofrendo alteração das tonalidades das cores (B).

Figura 6.18 – Comparação entre marcador gerado e impresso.

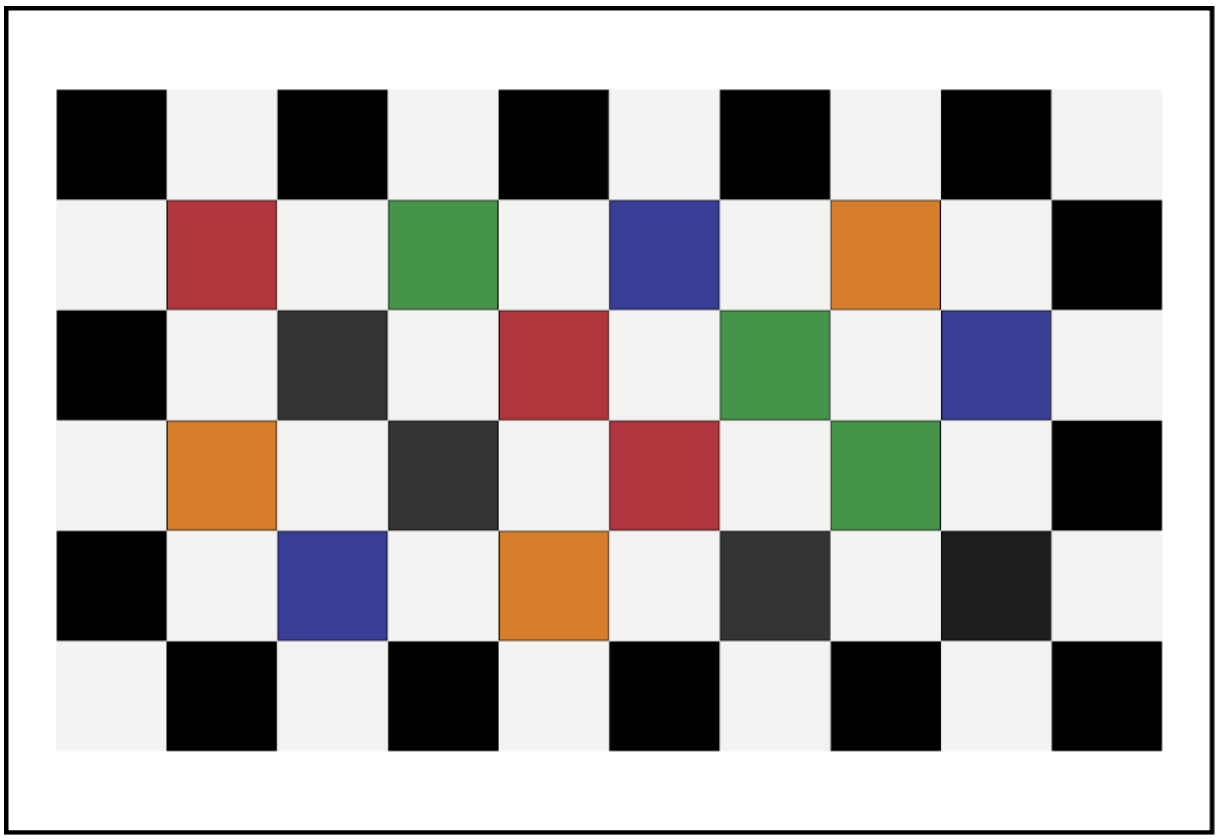

Fonte: Próprio autor.

Cada modelo de câmera, quando captura uma imagem, poderá interpretar as cores de uma forma diferente, sendo afetada pelas condições de iluminação do ambiente onde a captura da imagem está sendo realizada. O objeto fotografado possuirá uma determinada cor, mas, o sensor da câmera, de acordo com o tipo de iluminação irá registrá-la um pouco diferente.

Quando a imagem capturada é processada, para realizar a detecção um marcador, as suas cores já estão bem diferentes das cores do marcador original que foi gerado. A impressão do marcador é onde pode ocorrer a maior quantidade de alteração das cores originais do marcador, já que precisam ser levadas em consideração fatores como tipo papel e tipo de impressão (*laser* ou jato de tinta). Além disso, é necessária uma escolha apropriada dos valores para os tons de cores usados para configuração do marcador, com o objetivo que exista a menor diferença possível entre a cor do marcador gerado e a sua versão impressa.

Devido a variação das cores alvo, de acordo com variáveis relacionadas a impressão e ao tipo de iluminação na cena onde dos marcadores *CRFM* serão detectados, é necessário que exista um processo de faça a correspondência das cores alvo com as cores na forma que são capturas em cada cena, para possibilitar reconhecimento mais preciso pelo algoritmo de identificação do *CRFM*.

Figura 6.19 – Placa de correspondência de cores.



Fonte: Próprio autor.

Para realização da correspondência de cores, foi desenvolvida uma placa, semelhante à placa *chessboard* clássica de calibração de câmeras, no entanto o seu *grid* interno é colorido. A placa de correspondência desenvolvida possui tamanho de *grid* 5x7, de forma que cada quadrado interno da placa possui uma das cores que de interesse para detecção do marcador *CRFM*, seguindo a sequência: vermelho, verde, azul, laranja e preto. Estas cores são preenchidas sequencialmente na placa, desta forma cada cor aparece três vezes no *grid* (Figura 6.19).

Devido ao fato do *grid* interno da placa possuir 16 quadrados e a sequência das cores poderia preencher 15 destes, aplicando a sequência das cores por três vezes, um quadrado ficaria sem uma cor para preenche-lo, logo, foi optado por preenchê-lo com a cor preta, repetindo a cor do quadrado imediatamente anterior.

Para realização da correspondência da cor branca, são usados os quadrados que ficam localizados entre os coloridos (Figura 6.19). Assim, estas regiões são preenchias com um valor não totalmente puro da cor branca, para ser também impresso, e, da mesma forma que as demais cores, avaliada a variação que sofre de acordo com as condições da cena.

Figura 6.20 – Código-fonte *C++*: Identificação da Placa de Correspondência.

```
//Método para identificar a Placa de Correspondência
extern "C" inline __declspec(dllexport) void __stdcall
                            findCorrespondenceBoard(Mat &view)
{
   Size boardSize(5, 7);
   vector<Point2f> pointBuf;
   vector<vector<Point2f>> squaresTemp;

   bool found = findChessboardCorners(view,
                               boardSize,
                               pointBuf,
                               CV_CALIB_CB_ADAPTIVE_THRESH |
                                  CV_CALIB_CB_FAST_CHECK |
                                  CV_CALIB_CB_NORMALIZE_IMAGE);
   Mat viewGray;
   //Converte a imagem para tons de cinza
   cvtColor(view, viewGray, CV_BGR2GRAY);
   cornerSubPix(viewGray,
             pointBuf,
             Size(11, 11),
             Size(-1, -1),
             TermCriteria(CV_TERMCRIT_EPS + CV_TERMCRIT_ITER, 30, 0.1));

   //Adicionar todos os quadrados da placa em um vetor
   for (int j = 0; j < pointBuf.size() - (boardSize.width + 1); j++)
   {
      vector< Point2f > currentSquare;
      currentSquare.push_back(pointBuf[j]);
      currentSquare.push_back(pointBuf[j + 1]);
      currentSquare.push_back(pointBuf[j + 1 + boardSize.width]);
      currentSquare.push_back(pointBuf[j + boardSize.width]);
      squaresTemp.push_back(currentSquare);
   }
   vector<Scalar> foundColors;
   vector<Scalar> foundColorsWhite;
   for (int j = 0; j < squaresTemp.size(); j++)
   {
      //Verifica se os vértices pertecem a um quadrado na mesma linha
      if (distanceBtwPoints(squaresTemp[j][1], squaresTemp[j][2]) * 2 >
         distanceBtwPoints(squaresTemp[j][0], squaresTemp[j][1]))
      {
         Scalar color;
         processRegion(squaresTemp[j], view, color);
         //Quadrados não brancos
         if (j % 2 == 0)
            foundColors.push_back(color);
         //Quadrados brancos
         else
            foundColorsWhite.push_back(color);
      }
   }
   //Busca a correspondência entre as cores alvo e as detectadas na placa
   ColorMatching foundColorMatching = getColorMatching(foundColors)
}
```

Fonte: Próprio autor.

Figura 6.21 – Código-fonte *C++*: Deslocando de borda para cálculo do valor *RGB*.

```
//Processa a região para identificar a sua cor
inline void processRegion(const vector<Point2f> &verticesIn,
        const Mat &img, Scalar &color) const
{
   vector<Point2f> vecticesOut
   cv::Mat canonicalMarkerImage;
   std::vector<cv::Point2f> m_markerCorners2d;
   Size markerSize(100, 100);

   //Calcula o centro do quadrado
   Point2f centroid = calculateCentroid(verticesIn);

   Point2f pointP0 = verticesIn[0];
   Point2f pointP1 = verticesIn[1];
   Point2f pointP2 = verticesIn[2];
   Point2f pointP3 = verticesIn[3];

   Point2f centroidP0, centroidP1, centroidP2, centroidP3;
   centroidP0 = centroidP1 = centroidP2 = centroidP3 = centroid;

   //Faz deslocamento dos vértices para eliminar as bordas do quadrado
   for (size_t i = 0; i < (int)PercentEvaluation::PERCENT25; i++)
   {
      centroidP0 = Point2f((pointP0.x+centroidP0.x)/2, (pointP0.y+centroidP0.y)/2);
      centroidP1 = Point2f((pointP1.x+centroidP1.x)/2, (pointP1.y+centroidP1.y)/2);
      centroidP2 = Point2f((pointP2.x+centroidP2.x)/2, (pointP2.y+centroidP2.y)/2);
      centroidP3 = Point2f((pointP3.x+centroidP3.x)/2, (pointP3.y+centroidP3.y)/2);
   }

   for (size_t i = 0; i < (int)PercentEvaluation::PERCENT50; i++)
   {
      pointP0 = Point2f((pointP0.x+centroidP0.x)/2, (pointP0.y+centroidP0.y)/2);
      pointP1 = Point2f((pointP1.x+centroidP1.x)/2, (pointP1.y+centroidP1.y)/2);
      pointP2 = Point2f((pointP2.x+centroidP2.x)/2, (pointP2.y+centroidP2.y)/2);
      pointP3 = Point2f((pointP3.x+centroidP3.x)/2, (pointP3.y+centroidP3.y)/2);
   }

   vecticesOut.push_back(pointP0);
   vecticesOut.push_back(pointP1);
   vecticesOut.push_back(pointP2);
   vecticesOut.push_back(pointP3);

   m_markerCorners2d.push_back(Point2f(0, 0));
   m_markerCorners2d.push_back(Point2f(markerSize.width-1, 0));
   m_markerCorners2d.push_back(Point2f(markerSize.width-1, markerSize.height-1));
   m_markerCorners2d.push_back(Point2f(0, markerSize.height - 1));

   cv::Mat markerTransform = getPerspectiveTransform(vecticesOut, m_markerCorners2d);
   cv::warpPerspective(img, canonicalMarkerImage, markerTransform, markerSize);

   //Calcula a cor média RGB da região
   getPolygonsColor_BWRGB(canonicalMarkerImage, color);
}
```

Fonte: Próprio autor.

Basicamente o processo de correspondência de cores inicia pela aplicação do algoritmo de identificação de *chessboard* da biblioteca *OpenCV*, que está definido na função *cv::findChessboardCorners()* (Figura 6.20), em uma cena capturada por uma câmera, com a placa de correspondência sendo exposta à câmera, sob as mesmas condições de iluminação do ambiente que o marcador *CRFM* será exposto. Esta função irá preencher uma lista com a localização de todos os contornos internos da placa.

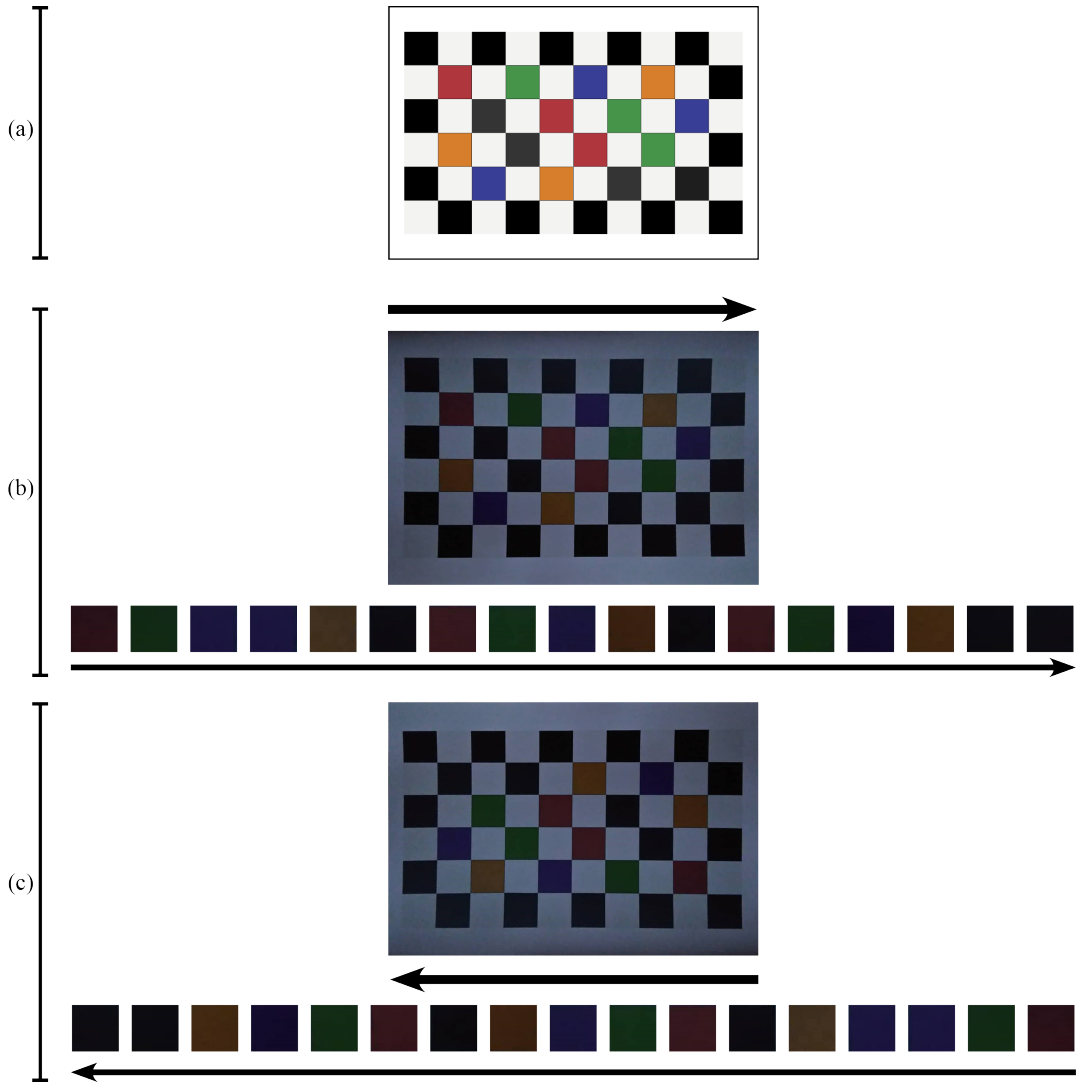
Com base na lista de contornos processados, os quadrados internos da placa são analisados para identificação do valor médio da cor *RGB* de cada região. Devido ao fato de os quadrados internos possuírem uma borda preta, para facilitar a operação *threshold* do algoritmo de detecção de *chessboard*, é necessário fazer um deslocamento de borda, para o interior da região, para eliminar a região mais externa de cada quadrado. A Figura 6.21 mostra o código-fonte para realizar o deslocamento de borda da região e calcular o valor médio RGB, já a Figura 6.22 mostra o resultado do deslocamento de borda, representado pelo que está limitado pelo pontilhado em branco e a região externa que é desconsiderada pelo algoritmo.

Figura 6.22 – Área processada de um quadrado que compõe o *chessboard*.

Fonte: Próprio autor.

Para o cálculo do valor média da cor branca, são usados todos os espaços entre os quadrados coloridos da placa. Este valor é analisado diretamente com a sua cor alvo correspondente. Para as demais cores de interesse, os seus quadrados coloridos estão em posições fixas dentro da estrutura do *chessboard*, desta forma é necessário realizar um tratamento para identificar erros de correspondência, como em situações em que a placa é girada, por exemplo.

Figura 6.23 – Exemplo de aplicação de correspondência de cores.




Fonte: Próprio autor.

Em ocorrências em que a placa está com alguma rotação aplicada, ela é processada para identificação das cores de cada posição e a partir disso criado um *array* com todos estes valores. Cada posição deste *array* é comparada com a sua posição correspondente de um outro *array* de cores alvo, que são as cores que são esperadas na cena (Figura 6.23 (a)). A fórmula $\Delta E_{2000}$ é aplicada e contabilizando o erro entre as cores de cada posição. Desta forma o *array* com as cores detectadas na placa é processado nos dois sentidos, a partir da primeira posição (Figura 6.23 (b)) e também a partir da última posição (Figura 6.23 (c)). Assim, quando o seu processamento que começa pela última posição do *array* resultar em um valor total de erro superior ao valor encontrado no sentido oposto, significa que a placa possui rotação, logo, o *array* é invertido para que as suas corres fiquem as posições corretas. A Figura 6.24 exibe como é identificada se alguma rotação está aplicada a placa, e se necessário realiza a correção para criar a correspondência entre as cores alvo e cores detectadas.

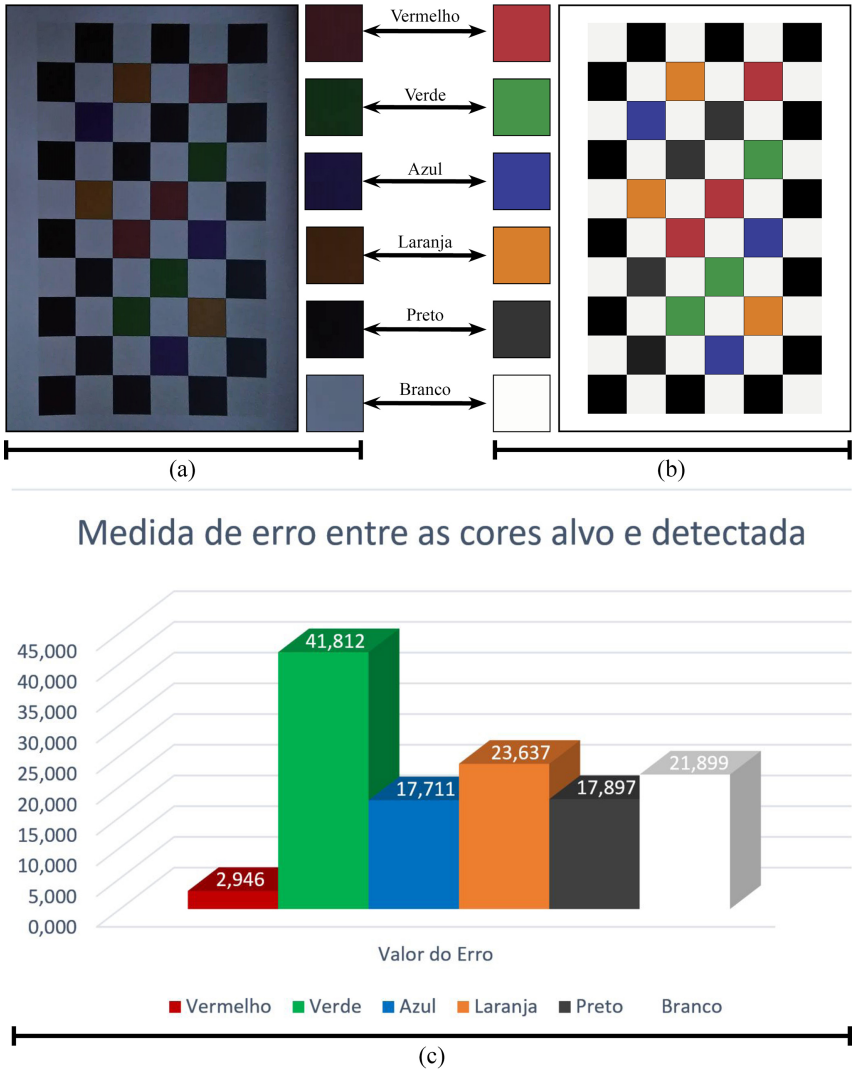
Figura 6.24 – Código-fonte *C++*: Criação da correspondência entre cores.

```
//Estrutura que representará os vetore de cores alvo e e cores detectadas
struct ColorMatching {
   vector<Scalar> detectedColors;
   vector<Scalar> targetColors;
};

//Busca a correspondência entre as cores alvo e as detectadas na placa
inline ColorMatching getColorMatching(const vector<Scalar>& foundColors) const
{
   double error1 = 0, error2 = 0;
   vector<vector<Point2f>> tempPoints;
   vector<Scalar> tempVector;

   ColorMatching foundColorMatching;

   //Adiciona as cores alvo à estrutura
   getTargetColors(foundColorMatching.targetColors);

   //Copia as cores detectadas na placa à estrutura
   copyVector(foundColors, foundColorMatching.detectedColors);

   //Calcula a diferença de cor em um sentido
   error1 = differenceCIE2000(tempVector, foundColorMatching.targetColors);

   //Inverte o vetor
   flipVector(tempVector);

   //Calcula a diferença de cor no outro sentido
   error2 = differenceCIE2000(tempVector, foundColorMatching.targetColors);

   /*
      Se error1 for maior que error2 é devido alguma rotação aplicada a placa,
      logo o vetor de cores detectadas na cena precisa ser invertido
   */
   if (error1 > error2)
   {
      flipVector(foundColorMatching.detectedColors);
   }

   return foundColorMatching;
}
```

Fonte: Próprio autor.

A Figura 6.25 mostra a placa com as cores alvo (a) e outras duas placas com as cores detectadas na cena (b e c), para criar uma correspondência de acordo com as condições de iluminação da imagem captura. O gráfico (c) apresenta os valores da diferença entre as cores alvo e detectada, aplicando a fórmula $\Delta E_{2000}$.

Figura 6.25 – Exemplo de aplicação de correspondência de cores.




Fonte: Próprio autor.

Para cada cor é calculado o valor médio em *RGB* e convertido para *CIELab*. Este valor é então usado na detecção como a representação da cor alvo na cena onde foi aplicado o processo de correspondência de cores. Para cada região de cor do marcador *CRFM* que for analisada, será realizada a comparação com o valor *CIELab* da cor alvo sem alteração e também com a cor obtida pela correspondência, servindo para uma detecção mais precisa de cada cor que compõe a estrutura do marcador.

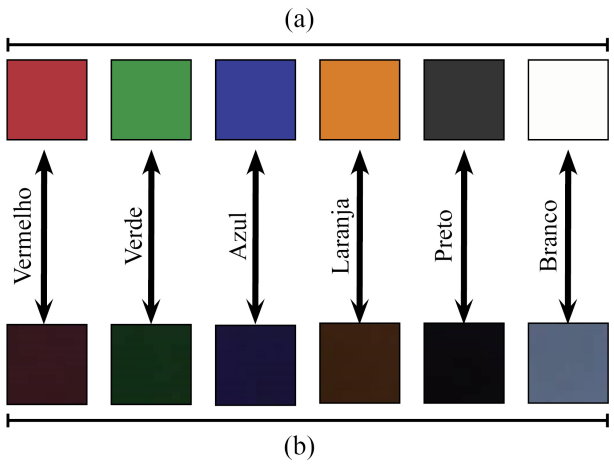
É preciso destacar que a placa de marcadores após impressa já irá sofre alteração nas cores, sendo agravado pelas condições de iluminação de cada cena. O gráfico (Figura 6.25 (c)) mostra o quanto esta diferença pode ser grande, sendo necessário medir como

cada cor de interesse é percebida de acordo com as condições da cena, para possibilitar uma detecção precisa do marcador *CRFM*.

### 6.3.1 Aplicação da correspondência de cores no algoritmo de detecção

Os valores *CIELab* de cada uma das cores de interesse, obtidos com a aplicação do processo de correspondência de cores, é enviado para o algoritmo de detecção de marcadores *CRFM*, para que estas cores sejam as cores alvos que são processadas na fase de filtragem de cores do algoritmo de identificação, conforme a Seção 5.2.2, que aborda a filtragem de cores.

Figura 6.26 – Correspondência de cores alvo e cores identificadas na cena.




Fonte: Próprio autor.

A Figura 6.26 mostra as cores alvo iniciais (b) e exemplos de cores identificadas de acordo com as condições de iluminação da cena (b). No algoritmo de filtragem de cores, os valores obtidos com o processo de correspondência são aplicados para obter as cores de cada uma das áreas de amostragem do quadrilátero. A utilização das cores conforme são encontradas em cada cena, tem o objetivo de diminuir as diferenças encontradas com a aplicação da fórmula $\Delta E^*_{2000}$, obtidas com utilizando o algoritmo adaptado de Fiumara (2017).

## 6.4 MÓDULO DE DETECÇÃO DE MARCADORES E PROJEÇÃO

A biblioteca *CRFM Lib*, integrada ao *Unity3D*, proporciona mecanismos para a criação de aplicações que fornecem experiências em Realidade Aumentada. Estes mecanismos são componentes construídos que transmitem informações com o *plugin OpenCV*,

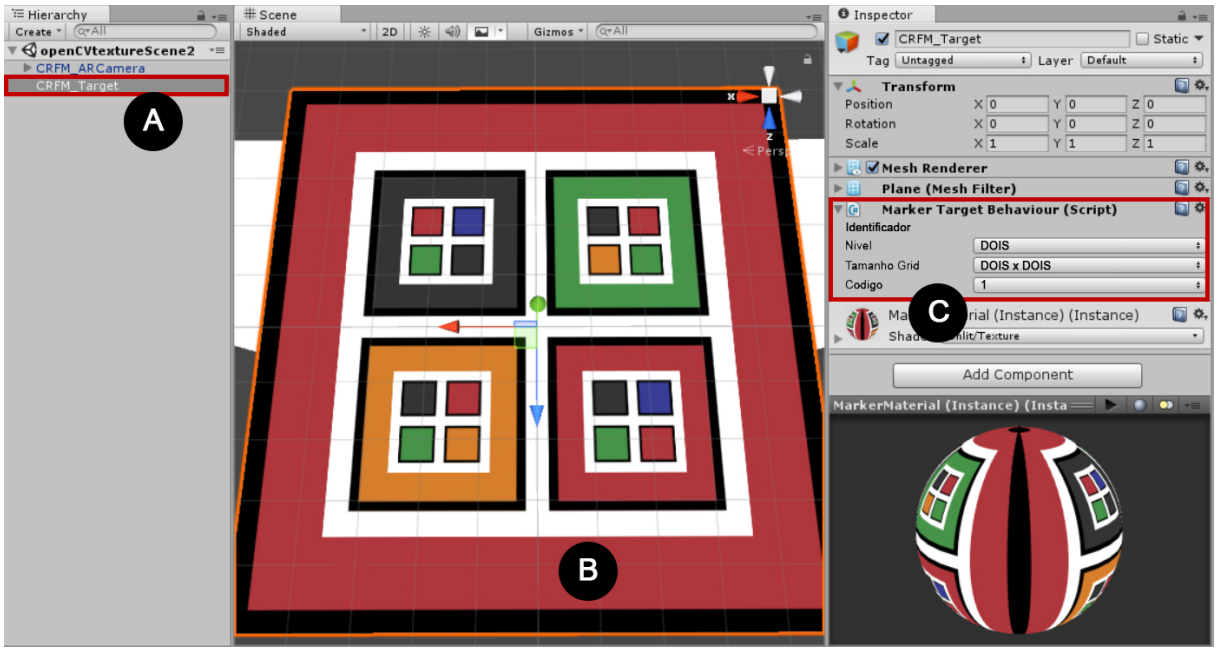
para a criação de correspondência de cor, usada na detecção dos marcadores, e parâmetros para projeção, quando os objetos virtuais interagem com o marcador *CRFM* alvo na cena.

O algoritmo de *CRFM* identifica os marcadores com base na maior hierarquia visível do alvo encontrado na cena. Os dados da estimativa da pose são obtidos pelo *Plugin Unity* e aplicados para compor a rotação e translação do objeto aumentado na cena.

### 6.4.1 Detecção de marcadores *CFRM*

O identificador de um marcador CRFM é composto pelo seu nível, tamanho de *grid* e um código. Estes dados (Figura 6.27 (C)) estão vinculados a *GameObjects* alvo (Figura 6.27 (A) e (B)) que serão usados para posicionamento dentro da cena do *Unity3D*. O identificador do marcador é enviado para processamento no *plugin C++* no início da execução da aplicação. A partir disso, marcadores que possuem o identificador definido, passam a ser rastreados na cena.

Figura 6.27 – Vinculação de um identificador de marcador *CRFM* no *Unity3D*.




Fonte: Próprio autor.

### 6.4.2 Projeção de objetos tridimensionais

Após a detecção dos marcadores *CRFM* dispostos na cena, pode ser realizada a projeção de elementos virtuais com base no posicionamento do marcador CRFM disposto

na cena. Para a realização da projeção é necessário que sejam identificadas as informações referentes a orientação do marcador alvo relacionado, com base na imagem da cena capturada, para criar valores de rotação e translação para o objeto que deverá ser aumentado.

Figura 6.28 – Código-fonte *C++*: Cálculo da matriz de rotação e vetor de translação.

```
1  Mat cameraMatrix;
2  Mat distCoeffs;
3  vector<Point3d> objectPoints;
4  vector<Point2d> imagePoints;
5
6  Mat rotationVector, rotationVectorAux;
7  Mat translationVector, translationVectorAux;
8  Mat rotationMatrix;
9
10 cv::solvePnP(objectPoints,
11    imagePoints,
12    cameraMatrix,
13    distCoeffs,
14    rotationVectorAux,
15    translationVectorAux);
16
17 rotationVectorAux.convertTo(rotationVector, CV_32F);
18 translationVectorAux.convertTo(translationVector, CV_32F);
19
20 cv::Rodrigues(rotationVector, rotationMatrix);
```

Fonte: Próprio autor.

O código *C++* exibido na Figura 6.28, mostra como são calculados os valores de rotação e translação que são usados pelo *Unity3D* para orientação dos objetos tridimensionais projetados. Ao lidar com espaços tridimensionais a representação da rotação geralmente é feita por matrizes 3 por 3, desta forma, para preparar os valores para o *Unity3D* é aplicado o método *cv::Rodrigues()*, de *OpenCV*, que converte o vetor de rotação para matriz, e vice-versa.

Figura 6.29 – Código-fonte *C#*: Definição da matriz de transformação.

```
Matrix4x4 transformationMatrixAux = new Matrix4x4();

transformationMatrixAux.SetRow(0, new Vector4((float)rotationMatrix.row0.c0,
                                    (float)rotationMatrix.row0.c1,
                                    (float)rotationMatrix.row0.c2,
                                    (float)translationVector.c0));

transformationMatrixAux.SetRow(1, new Vector4((float)rotationMatrix.row1.c0,
                                    (float)rotationMatrix.row1.c1,
                                    (float)rotationMatrix.row1.c2,
                                    (float)translationVector.c1));

transformationMatrixAux.SetRow(2, new Vector4((float)rotationMatrix.row2.c0,
                                    (float)rotationMatrix.row2.c1,
                                    (float)rotationMatrix.row2.c2,
                                    (float)translationVector.c2));

transformationMatrixAux.SetRow(3, new Vector4(0, 0, 0, 1));
```

Fonte: Próprio autor.

Os valores da matriz de rotação e do vetor de translação são obtidos no processamento das informações de cada marcador reconhecido na cena, processados pelos *plugin C++*. Estes valores são recebidos no *Unity3D* e aplicados em uma matriz 4 por 4, para criar uma matriz de transformação. Esta matriz com quatro linhas e quatro colunas é preenchida conforme mostrada na Figura 6.29 com código *C#*, de forma que a quarta coluna é preenchida com o vetor de translação e as demais colunas são compostas pela matriz de rotação.

Figura 6.30 – Código-fonte *C#*: Ajuste na matriz de transformação.

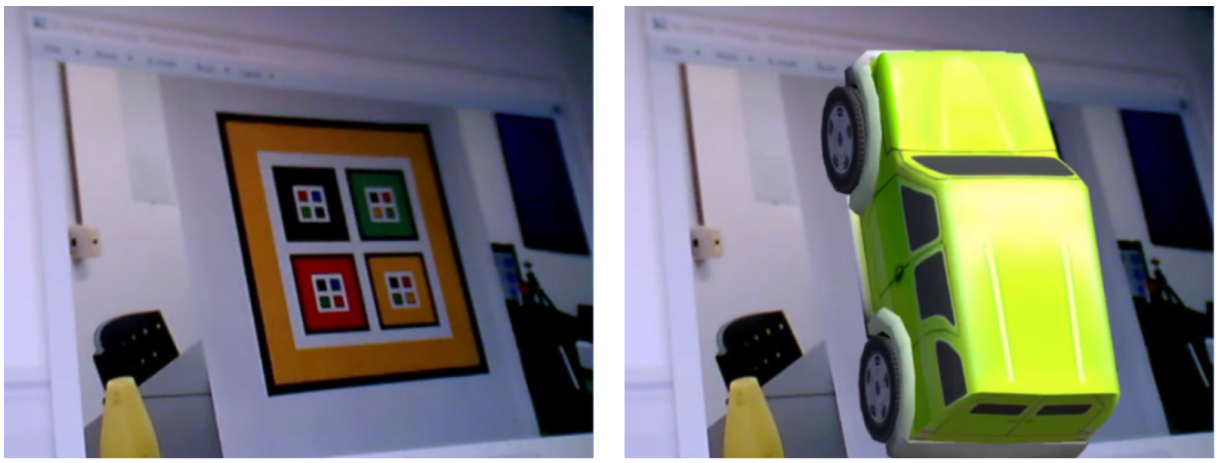
```
Matrix4x4 invertYM = Matrix4x4.TRS(Vector3.zero,
                                   Quaternion.identity,
                                   new Vector3(1, -1, 1));

Matrix4x4 invertZM = Matrix4x4.TRS(Vector3.zero,
                                   Quaternion.identity,
                                   new Vector3(1, 1, -1));

Matrix4x4 transformationMatrix = ARCamera.transform.localToWorldMatrix * invertYM *
                                    transformationMatrixAux * invertZM;
```

Fonte: Próprio autor.

A definição da orientação do objeto tridimensional que será aumentado, é feito com base na posição da câmera na cena do *Unity3D*. Logo, para obter a matriz de transformação ajustada, deve ser multiplicada a matriz de posicionamento de câmera na cena, invertendo seu eixo *Y*, pela matriz de transformação já obtida, invertendo seu eixo *Z* (Figura 6.30). A matriz resultante é então aplicada para rotação e translação do objeto tridimensi-

onal no *Unity3D* em relação a câmera na cena, conforme pode ser visualizado na Figura 6.31.

Figura 6.31 – Projeção realizada no CRFM Lib.



(a) (b)

Fonte: Próprio autor.

A Figura 6.31 (a) exibe uma imagem capturada e em (b) a mesma imagem processada com projeção feita no *CRFM Lib*.

### 6.4.3 Projeto *CRFM Lib* na *Unity Game Engine*

As funcionalidades necessárias, que possibilitam *CRFM Lib* ser usada como uma ferramenta para desenvolvimento de aplicações de Realidade Aumentada, são construídas e adicionadas a *prefabs*. Estes, são definidos como modelos compostos por *GameObjects*, que guardam comportamentos pré-programados e componentes configurados (HOCKING, 2015), permitindo que funcionalidades específicas sejam compartilhadas por várias aplicações que usam o mesmo modelo pré-fabricado.

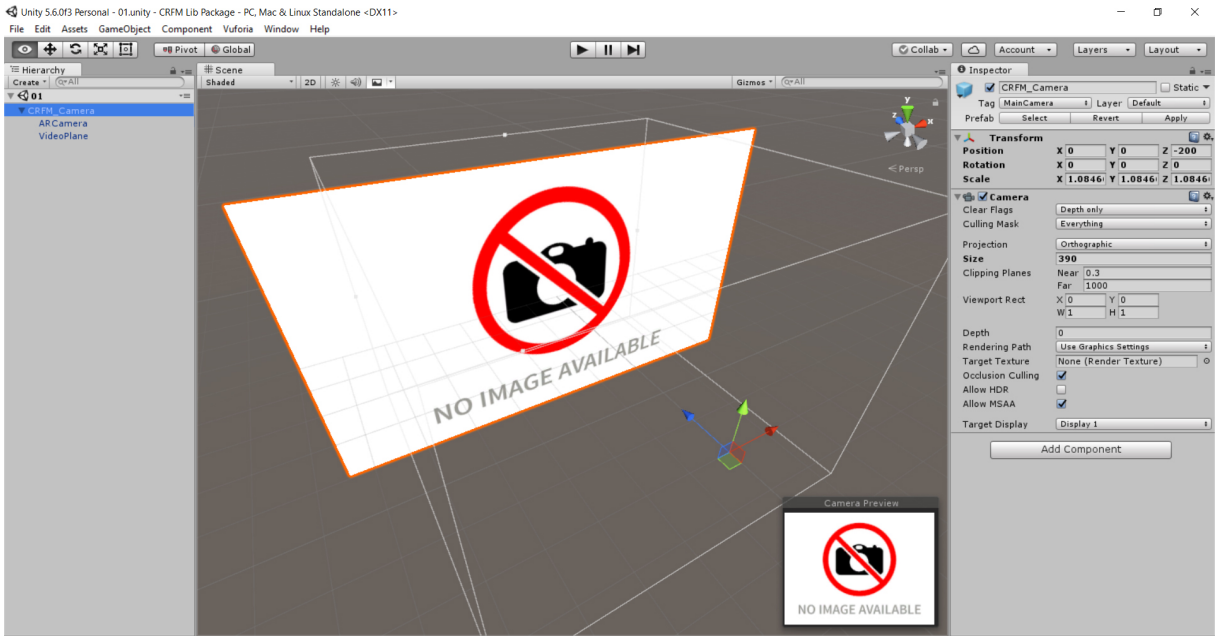
Para o *CRFM* as funcionalidades de captura de imagem, detecção de marcadores e projeção de elementos tridimensionais são definidos em dois *prefabs*, buscando proporcionar agilidade no desenvolvimento de aplicações de RA.

#### *6.4.3.1 Prefab para gerenciamento de Câmera*

Neste *prefab* estão centralizadas as principais funcionalidades da aplicação, como o gerenciamento de câmera, a realização da captura das imagens e sua exibição da tela do dispositivo. Através de utilização do *prefab CRFM_Camera*, podem ser configuradas

a resolução da imagem obtida pela câmera, e também a sua taxa de quadros capturados por segundo (*FPS*), conforme é exibido na Figura 6.32.

Figura 6.32 – *Prefab CRFM_Camera*.




Fonte: Próprio autor.

A Figura 6.33 mostra o método para definição da taxa de quadros na *Unity Game Engine*, quando é iniciada a execução da aplicação, com base nos valores definidos no *prefab*.

Figura 6.33 – Código-fonte *C#*: Alteração da taxa de quadros na *Unity Game Engine*.

```
1  void Awake()
2  {
3      //Alterar o frame rate no Unity
4      StartCoroutine(SetFrameRate(imFps));
5  }
6
7  IEnumerator SetFrameRate(int frameRate)
8  {
9      yield return new WaitForSeconds(1);
10     QualitySettings.vSyncCount = 0;
11     Application.targetFrameRate = frameRate;
12 }
```

Fonte: Próprio autor.

Também neste *prefab* estão definidas as regras para projeções, baseadas nos alvos reconhecidos na cena. Todos os marcadores alvos que estão configurados para serem identificados, são gerenciados por *CRFM_Camera*. Através desta referência a posição dos alvos em relação a câmera, na cena *Unity*, são conhecidos e usados para projeção dos objetos quando cada marcador for detectado, conforme demonstrado na Figura 6.34.

Figura 6.34 – Código-fonte *C#*: Método *Update()* para detecção do marcador *CRFM*.

```
void Update()
{
   //Verifica se existem CRFM_MarkerTargets adicionados na cena
   if (markerTargets.Length > 0)
   {
      //Processa o algoritmo de detecção do marcador
      //Passa todos os identificadores adicionados na cena
      imgColorData = OpenCVInterop.ProcessColoredAlgorithm(_listIdentifiers);
      //Converte o array de bytes para o formato Texture2D
      texColor.LoadRawTextureData(imgColorData);
      //Aplica a image capturada no plano de exibição
      texColor.Apply();
      //Percorre todos os CRFM_MarkerTarget existentes na cena
      foreach (MarkerTargetBehaviour settings in markerTargets)
      {
         //Verifica o status da detecção de cada CRFM_MarkerTarget existente na cena
         settings.SetDetectionState(
                  OpenCVInterop.GetStateDetectionMarker(
                              Convert.ToInt16(settings.MarkerDesign.Level),
                              Convert.ToInt16(settings.MarkerDesign.GridSize),
                              settings.MarkerDesign.Id));
         //Se caso o identificador associado ao CRFM_MarkerTarget foi detectado

         if (settings.getDetectionState())
         {
            settings.FrameCount = 0;
            //Calcula a matriz de rotação e translação
            RTMatrix rtMatrix = OpenCVInterop.GetRTMatrix();
            //Encontra o centro do marcador, na imagem
            CenterPosition center = OpenCVInterop.GetCenterPosition();
            //Encontra as distâncias internas do marcador, na imagem
            DistanceSize distance = OpenCVInterop.GetDistanceSize();
            //Realiza a projeção com base nas informações obtidas
            settings.AugmentetObject(ARCamera, rtMatrix, center,
                  distance, relationPlaneImage, PlaneSize);
         }
         else
         {
            if (settings.FrameCount == settings.FrameRepeatMax)
            {
               settings.FrameCount = 0;
               //Após repetir a projeção por X frames, desabilita o objeto 3D
               settings.setAllARGameObjectsDisable();
            }
            settings.FrameCount++;
         }
      }
   }
}
```

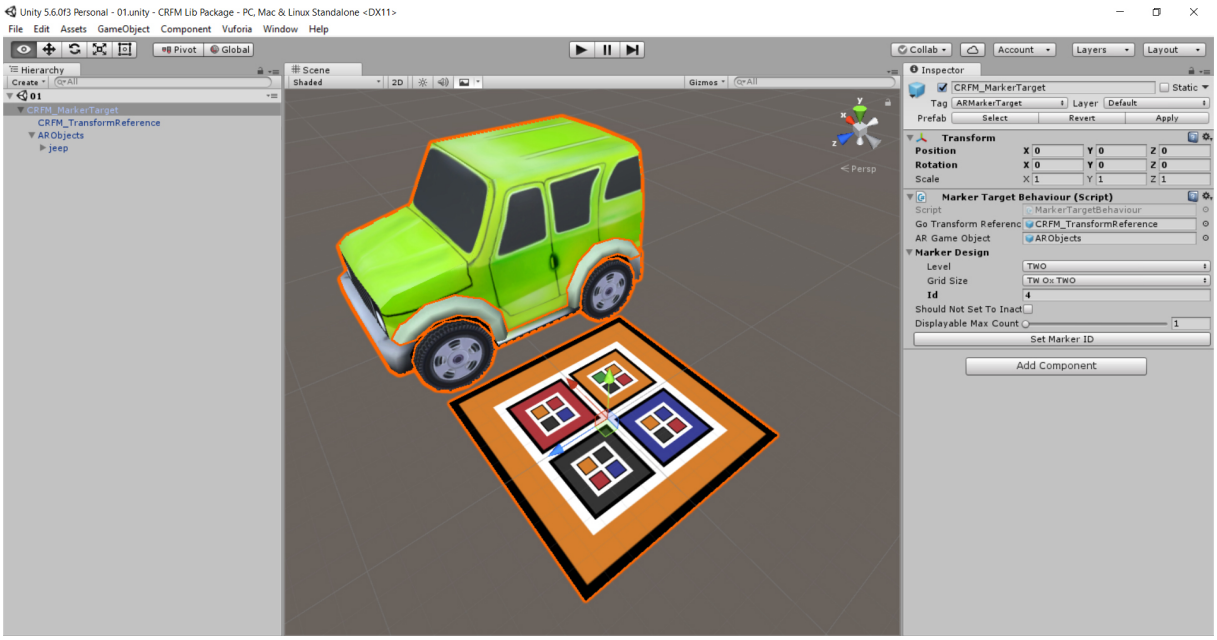
Fonte: Próprio autor.

### *6.4.3.2 Prefab para gerenciamento de marcadores alvo*

Diversos marcadores podem ser identificados em uma cena, mas para determinadas aplicações, somente marcadores com identificadores específicos são requeridos. Desta forma é necessário que sejam configurados os marcadores alvos que precisam ser procurados.

O *prefab CRFM_MarkerTarget* permite que um marcador seja configurado para ser procurado em uma cena, desta forma *CRFM Lib* irá adquirir os parâmetros para projeção somente do marcador especificado. Neste *prefab* são definidos o Identificador do marcador, composto pelo seu nível, tamanho de grid e código, bem como o conteúdo que será aumentado na cena e uma referência de posição para a sua projeção.

Figura 6.35 – *Prefab CRFM_MarkerTarget*.




Fonte: Próprio autor.

Cada *CRFM_MarkerTarget* configurado para detecção, é referenciado pelo *prefab CRFM_Camera*, desta forma, quando localizado na cena, o identificador do alvo definido, é realizada a projeção com base na matriz de transformação obtida do marcador e a referência de posição do alvo definida no *prefab*.

A Figura 6.36 mostra os métodos para cálculo da matriz de transformação ajustada, utilizando a matriz de transformação do marcador na imagem processada e a matriz de transformação da câmera dentro da cena *Unity*, por fim invertendo seus eixos para ajustar a projeção dos objetos.

Figura 6.36 – Código-fonte *C#*: Preenchimento da matriz de transformação.

```
//Obtém a matriz de transformação ajustada invertando os eixos Y e Z
private void GetARM(Matrix4x4 cameraLocalToWorldMatrix, RTMatrix rtMatrix)
{
    SetTransformation(rtMatrix);
    ARM = cameraLocalToWorldMatrix * invertYM * Transformation * invertZM;
}

private void SetTransformation(RTMatrix rtMatrix)
{
    Transformation.SetRow(0, new Vector4((float)rtMatrix.row0.c0,
                                         (float)rtMatrix.row0.c1,
                                         (float)rtMatrix.row0.c2,
                                         (float)rtMatrix.row0.c3));
    Transformation.SetRow(1, new Vector4((float)rtMatrix.row1.c0,
                                         (float)rtMatrix.row1.c1,
                                         (float)rtMatrix.row1.c2,
                                         (float)rtMatrix.row1.c3));
    Transformation.SetRow(2, new Vector4((float)rtMatrix.row2.c0,
                                         (float)rtMatrix.row2.c1,
                                         (float)rtMatrix.row2.c2,
                                         (float)rtMatrix.row2.c3));
    Transformation.SetRow(3, new Vector4((float)rtMatrix.row3.c0,
                                         (float)rtMatrix.row3.c1,
                                         (float)rtMatrix.row3.c2,
                                         (float)rtMatrix.row3.c3));
}
```

Fonte: Próprio autor.

A Figura 6.37 mostra o método para aumentar os objetos tridimensionais com base nos parâmetros obtidos no *Plugin C++*.

Figura 6.37 – Código-fonte *C#*: Método para aumentar objeto *3D* na cena.

```
public void AugmentObject(Camera arCamera,
    GameObject arGameObject,
    RTMatrix rtMatrix,
    CenterPosition center,
    DistanceSize distance,
    Vector2 relationPlaneImage,
    Vector2 planeSize)
{

    //Calcula a matriz de transformação
    GetARM(arCamera.transform.localToWorldMatrix, rtMatrix);

    var matrix4X4 = ARM;

    //Aplica a matriz de transformação
    ARUtils.SetTransformFromMatrix(arGameObject.transform, ref matrix4X4);

    //Ajusta a posição do objeto aumentado
    arGameObject.transform.position = new Vector3(
                ((float)center.x * relationPlaneImage.x) - planeSize.x / 2,
                ((float)center.y * relationPlaneImage.y) - planeSize.y / 2,
                arGameObject.transform.position.z);

    /*
     Ajusta a escala do objeto aumentado, com base
     nas distâncias calculas do marcador na imagem processada
    */
    arGameObject.transform.localScale = new Vector3(
                (float)distance.dDistanceV,
                (float)distance.dDistanceV,
                (float)distance.dDistanceV);
}
```

Fonte: Próprio autor.

## 6.5 CONSIDERAÇÕES FINAIS SOBRE O CAPÍTULO

Neste capítulo, foi apresentado o desenvolvimento da ferramenta *CRFM Lib*, proposta nesta dissertação, onde o *plugin C++* para identificação do marcador *CRFM* é integrado ao *Unity3D* para criação de aplicações em Realidade Aumentada. É importante salientar que esta ferramenta é suportada apenas pela plataforma *Windows*. Testes de desempenho da ferramenta foram desenvolvidos, conforme apresentado no Capítulo 7.

# 7 VALIDAÇÃO E TESTES

Neste capítulo, é apresentada uma avaliação da ferramenta sob diferentes aspectos. A próxima seção aborda o *hardware* usado para a realização dos testes. A Seção 7.2 apresenta uma análise da utilização da Placa de Correspondência de Cores, mostrando a variação das cores em diferentes tipos de iluminação. Na Seção 7.3 são mostrados testes sobre usabilidade, referente ao uso da ferramenta, em integração ao *Unity*, para desenvolvimento de uma aplicação. Também é avaliado o desempenho da ferramenta durante a execução. Esta análise tem como objetivo demonstrar a viabilidade e eficácia da ferramenta desenvolvida.

## 7.1 *HARDWARE UTILIZADADO PARA OS TESTES*

Todas as imagens no conjunto de dados foram capturadas a partir de uma câmera USB Novatek NY99140 capturando uma resolução de 1280x720.

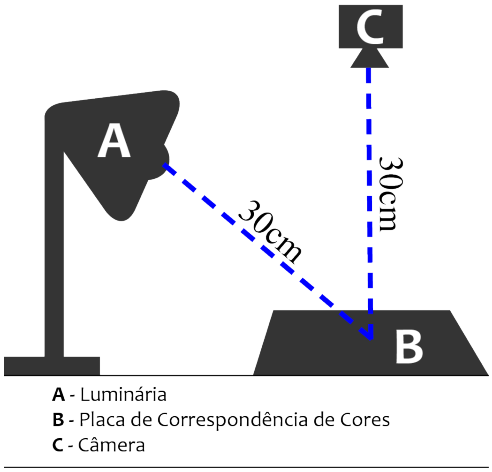
Os experimentos de avaliação foram executados em um *notebook Dell Inspiron* 15 série 5000 (5547) com as seguintes caraterísticas:

- Processador: *Intel Core i7–4510U 3.1Ghz*;
- Memória: *8GB de RAM DDR3*;
- Armazenamento: *220GB SSD*;
- Placa de vídeo: *AMD Radeon HD 2GB 64bits*;
- Sistema Operacional: *Windows 10 Pro*.

## 7.2 UTILIZAÇÃO DA PLACA DE CORRESPONDÊNCIA DE CORES

A Placa de Correspondência de Cores foi testada em diferentes condições de iluminação, para avaliar a sua aplicabilidade em medições de variação das cores alvo em diferentes condições de iluminação. Para esta análise foram simulados ambientes com diferentes cores de iluminações que poderiam interferir na identificação do marcador *CRFM*.

Figura 7.1 – Distâncias aplicadas para composição do ambiente de testes.

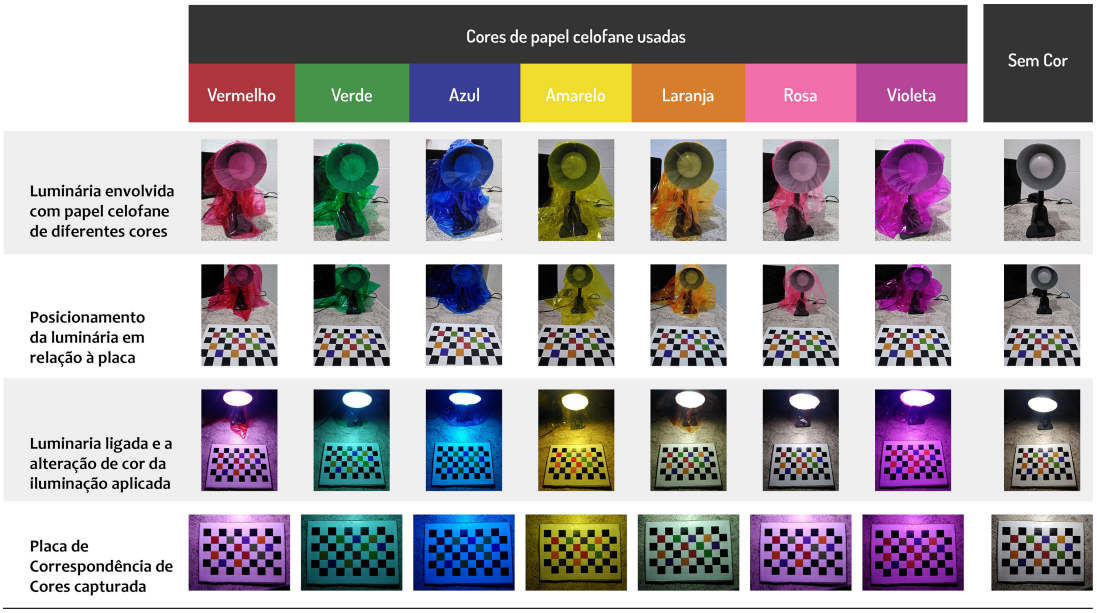



Fonte: Próprio autor.

Para simular a variação de cores de iluminação, foi utilizada uma luminária com lâmpada de *LED* de *10W*, envolvida com folhas de papel celofane de variadas cores. A luminária foi posicionada lateralmente a 30cm do centro da placa. Para cada condição de iluminação estabelecida, a placa foi capturada usando o mesmo posicionamento de câmera, com distância de também 30cm, do centro da placa. As distâncias aplicadas entre os objetos são mostradas na Figura 7.1.

A Figura 7.2 mostra como o ambiente de testes foi preparado, e as respectivas imagens capturadas para cada tipo de iluminação aplicado.

Figura 7.2 – Cores usadas para preparação do ambiente de testes.




Fonte: Próprio autor.

A Figura 7.3 exibe as variáveis envolvidas na execução deste teste.

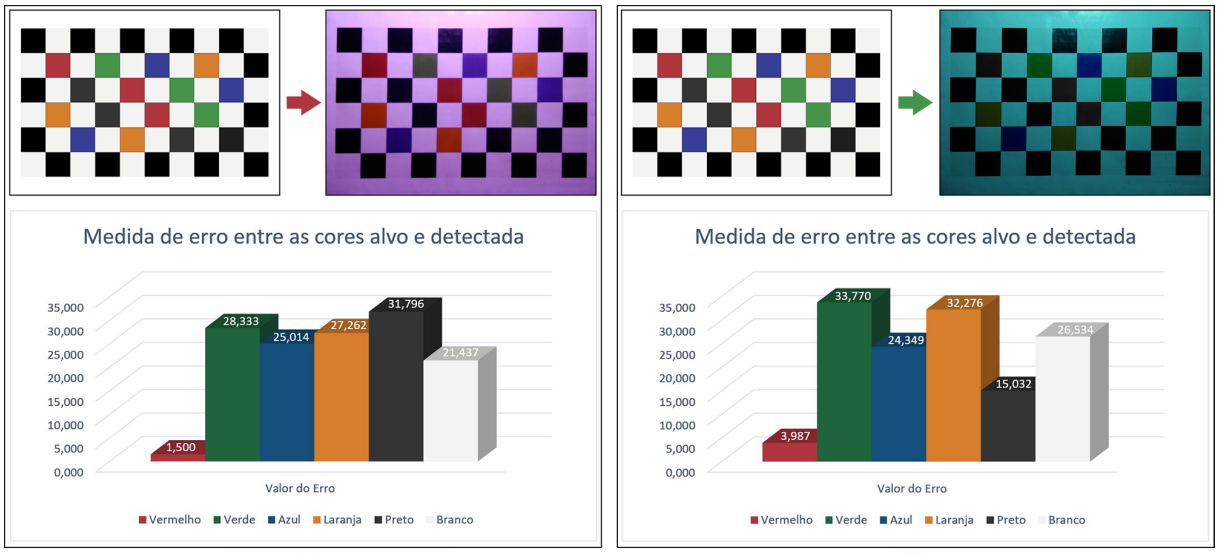
Figura 7.3 – Variáveis envolvidas no teste de aplicação da Placa de Correspondência.

| Variáveis Envolvidas | |
|---|---|
| **Distância entre câmera e placa** | 30cm entre a câmera e o centro da placa.<br>A câmera é posicionada sobre a placa. |
| **Distância entre luminária e placa** | 30cm entre a luminária e o centro da placa.<br>A luminária é posicionada lateralmente a placa. |
| **Cores aplicadas na iluminação** | 7 cores de papel celofane foram aplicadas:<br>vermelho, verde, azul, amarelo, laraja, rosa, violeta. |

Fonte: Próprio autor.

As imagens capturadas foram submetidas ao método de medição de distância de cor, usando a placa de correspondência, oito imagens, onde, em sete delas é utilizado papel celofane para simular alterações da cor da iluminação aplicada. Quando todos os resultados da medição foram gerados, realizou-se o processo de análise destes através de gráficos.

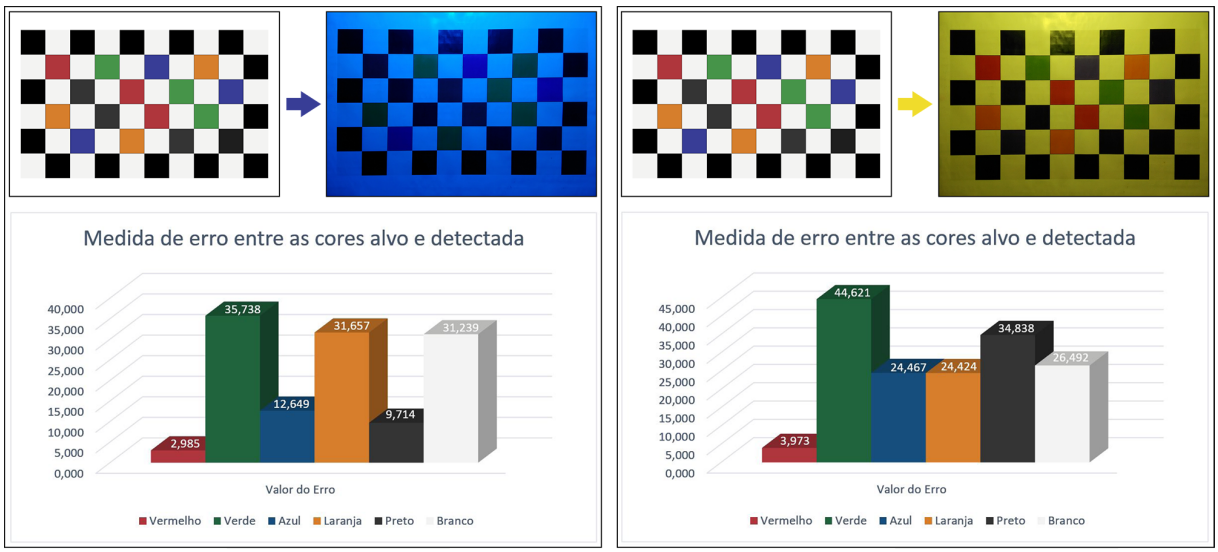
Figura 7.4 – Variação das cores sob iluminação nas cores vermelha (a) e verde (b).




(a) (b)

Fonte: Próprio autor.

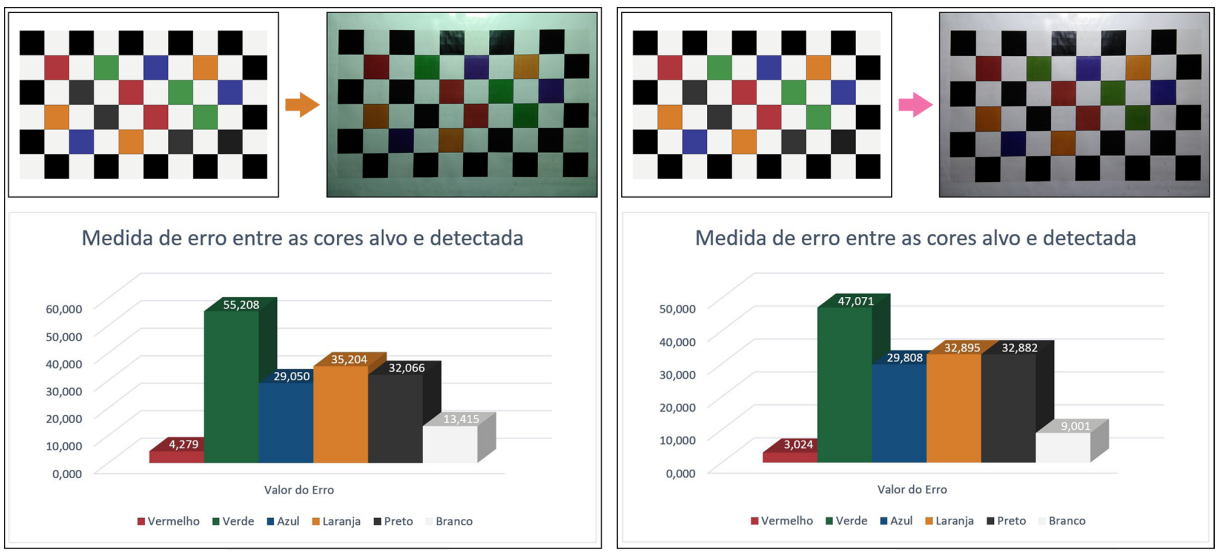
Figura 7.5 – Variação das cores sob iluminação nas cores azul (a) e amarela (b).




(a) (b)

Fonte: Próprio autor.

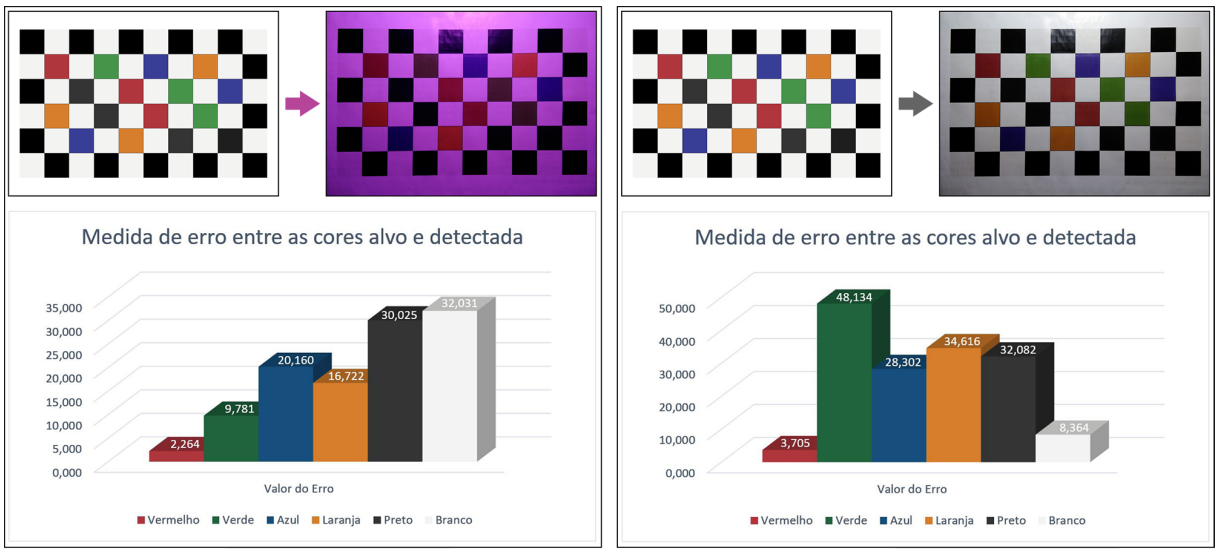
Figura 7.6 – Variação das cores sob iluminação nas cores laranja (a) e rosa (b).




(a) (b)

Fonte: Próprio autor.

Figura 7.7 – Variação das cores sob iluminação na cor violeta (a) e sem de cor (b).




(a) (b)

Fonte: Próprio autor.

Gráficos mostram a distância de cores detectadas sob as condições de iluminação simuladas, podem ser visualizados nas Figuras 7.4, 7.5, 7.6 e 7.7. Em uma análise dos gráficos gerados, pode-se notar que dentre as cores que compõem a Placa de Correspondência, que a cor vermelha é a que se mantém com menos alteração quando comparados aos cenários de iluminação criados. Por outro lado, a cor verde é a que sofreu maior variação em relação aos testes realizados.

## 7.3 APLICAÇÃO DESENVOLVIDA COM *CRFM LIB*

Para avaliação da biblioteca *CRFM Lib* foi desenvolvida uma aplicação de Realidade Aumentada utilizando a *Unity Game Engine*. A aplicação consiste em projetar sobre o marcador *CRFM* um modelo tridimensional de um veículo.

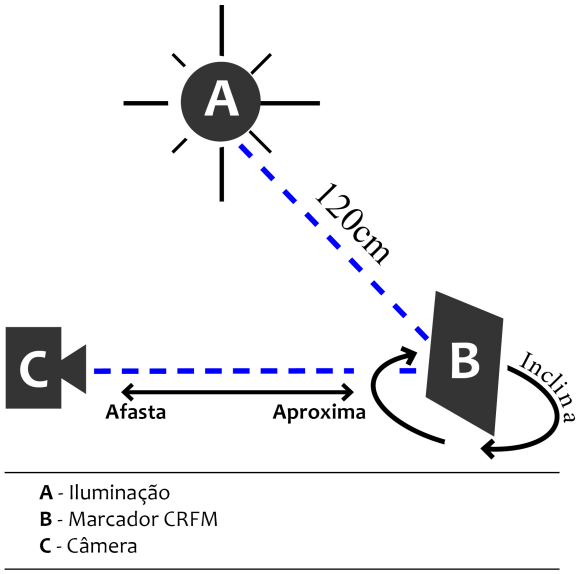
A avaliação do desempenho de detecção de marcadores *CRFM* foi realizada utilizando um marcador de 8,5cm de lado, impresso em papel comum. Os testes foram realizados com a câmera fixa e baseados na execução da aplicação diretamente na *Unity*.

### 7.3.1 Testes de Detecção sem Movimentação de Câmera

Inicialmente foi avaliado o desempenho da detecção para diferentes distâncias e com diferenças ângulos de inclinação aplicados ao marcador. Para este teste não foi aplicado o método de Correspondência de Cor.

As distâncias aplicadas entre os objetos são mostradas na Figura 7.8, já Figura 7.9 exibe o ambiente preparado para a realização dos testes.

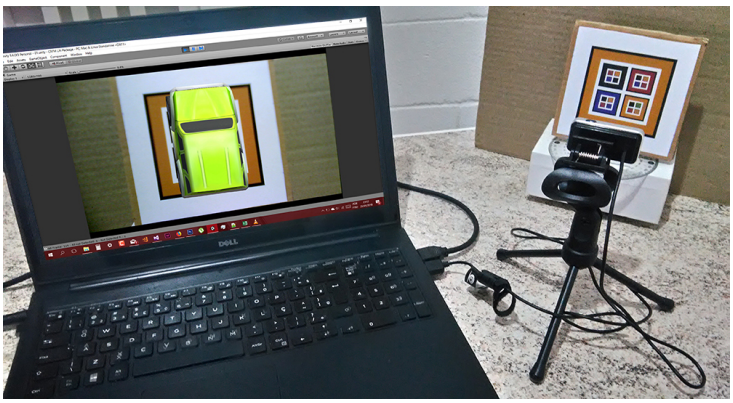
Figura 7.8 – Distâncias aplicadas para composição do ambiente de teste.




Fonte: Próprio autor.

As condições de iluminação foram mantidas iguais para todos os posicionamentos de câmera aplicados. Todos os testes foram realizados com a câmera fixada em um tripé, em ambiente fechado, com uma medição média de luminância (lux) de 200.

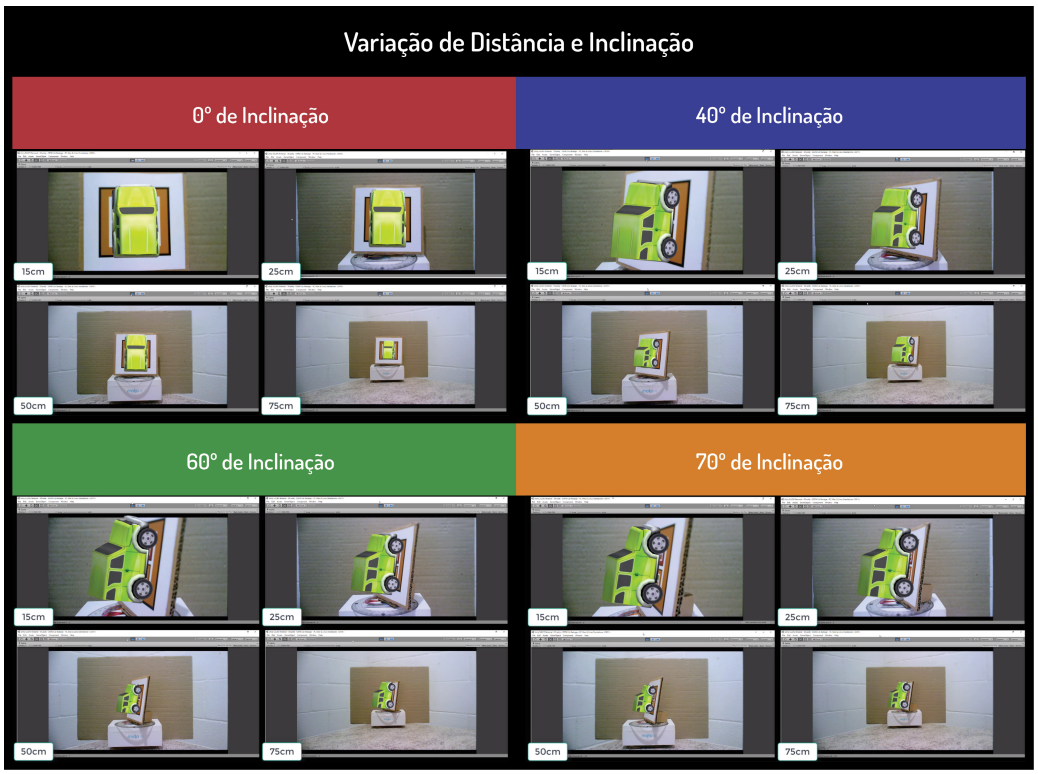
Figura 7.9 – Organização do cenário para realização dos testes.



Fonte: Próprio autor.

Foram capturados 420 *frames*, em que continham o marcador CRFM, variando a distância da câmera em 15cm, 25cm, 50cm e 75cm. Além disso, para cada distância, foram aplicados 4 ângulos diferentes, 0° (sem inclinação), 40°, 60°, 70°, conforme é mostrado na Figura 7.10.

Figura 7.10 – Variação de distância e inclinação dos testes de detecção realizados.




Fonte: Próprio autor.

A Figura 7.11 exibe as variáveis envolvidas na execução deste teste.

Figura 7.11 – Variáveis envolvidas na avaliação do percentual de detecção.

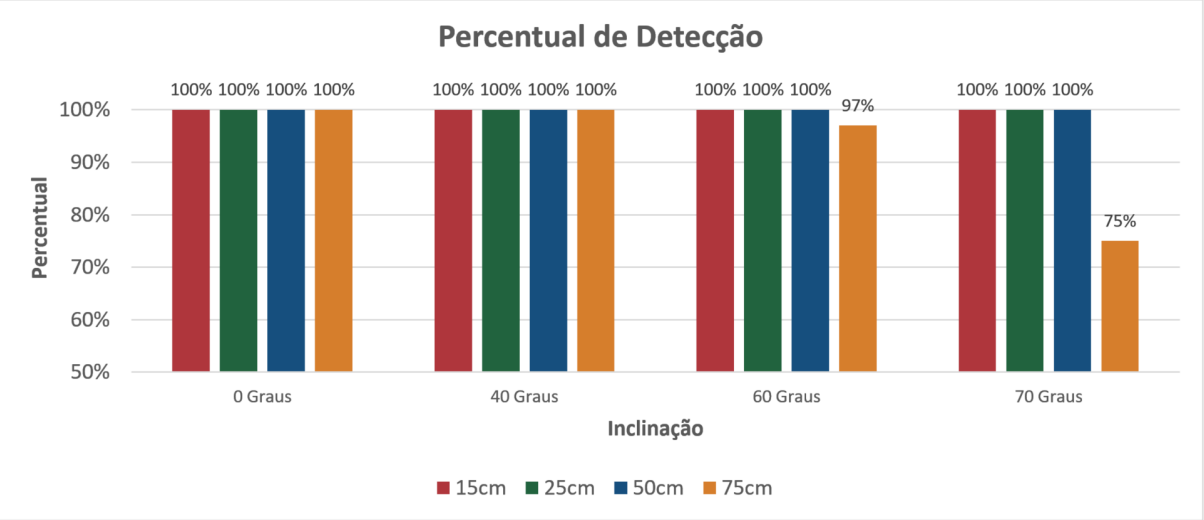
| Variáveis Envolvidas | |
|---|---|
| **Distância entre iluminação e marcador** | 120cm entre a luminária e o centro do marcador. A luminária é posicionada lateralmente ao marcador. |
| **Distância entre câmera e marcador** | Para cada ângulo de inclinação são aplicadas as distâncias: 15cm, 25cm, 50cm e 75cm. |
| **Ângulo de inclinação aplicado ao marcador** | Para cada distância entre marcador e câmera são aplicados os ângulos de inclinação: 0° (sem inclinação), 40°, 60°, 70°. |

Fonte: Próprio autor.

Para cada variação de inclinação e distância, foram capturados 420 *frames*, destes, os primeiros 60 foram descartados, devido a adaptação de iluminação pelo sensor da câmera, com os 360 *frames* restantes sendo processados. O Gráfico 7.1 mostra o percentual de detecção do marcador CRFM no cenário estabelecido. Para as amostras analisadas, somente nos casos onde a inclinação aplicada era de 70° não ocorreu a identificação do marcador em alguns *frames*, sendo possível verificar ainda que, quanto maior a distância e inclinação do marcador, maior a taxa de falhas na detecção, desta forma, para inclinações

superiores a 70° a detecção do marcador começa a ser inviável. Um vídeo demonstrando a detecção com as variadas distâncias e inclinações foi disponibilizado na *internet* para visualização (CRFMLIB, 2018).

Gráfico 7.1 – Percentual de detecção sem movimentação da câmera.




Fonte: Próprio autor.

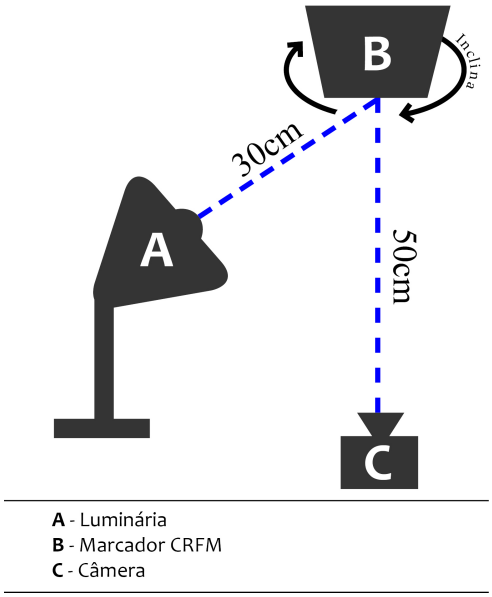
A falha na detecção do marcador CRFM pode ser atribuída ao fato de que, em maiores inclinações a borda branca de separação de blocos e hierarquias possui sua visibilidade reduzida, bem como as bordas de cada hierarquia, reduzindo assim as regiões de interesse para análise de cor.

### 7.3.2 Testes de Detecção com diferentes tipos de iluminação

Para este teste foi verifica a aplicabilidade do método de Correspondência de Cor para melhorar a taxa de detecção do marcador. Foram simulados diferentes tipos de iluminação. Para simular a variação de cores de iluminação, foi utilizada uma luminária com lâmpada de LED de 10W, envolvida com folhas de papel celofane de variadas cores.

A luminária foi posicionada lateralmente a 30cm de distância do centro do marcador. Para cada condição de iluminação estabelecida, o marcador foi processado pela aplicação em execução na *Unity* usando o mesmo posicionamento de câmera, com distância de 50cm do centro do marcador. As distâncias aplicadas entre os elementos na cena são mostradas na Figura 7.12, já a Figura 7.13 exibe o ambiente preparado para a realização dos testes.

Figura 7.12 – Distâncias aplicadas para composição do ambiente de teste.




Fonte: Próprio autor.

Para cada variação de inclinação e distância, foram capturados 420 *frames* contendo o marcador *CRFM*, aplicando ângulos de 0° (sem inclinação), 40°, 60°, 70°, a uma distância fixa de 50cm entre a câmera e o marcador impresso. Os primeiros 60 foram descartados, para adaptação de iluminação pelo sensor da câmera, e os 360 *frames* restantes foram processados.

Figura 7.13 – Organização do cenário para realização dos testes.

Fonte: Próprio autor.

A Figura 7.14 exibe as variáveis envolvidas na execução deste teste.

Figura 7.14 – Variáveis envolvidas na avaliação sob diferentes iluminações.

| Variáveis Envolvidas | |
|---|---|
| **Distância entre iluminação e marcador** | 30cm entre a luminária e o centro do marcador.<br>A luminária é posicionada lateralmente ao marcador. |
| **Distância entre câmera e marcador** | 50cm entre a câmera e o centro do marcador.<br>A câmera é posicionada em frente ao marcador. |
| **Ângulo de inclinação aplicado ao marcador** | São aplicados os ângulos de inclinação:<br>0° (sem inclinação), 40°, 60°, 70°. |
| **Cores aplicadas na iluminação** | 7 cores de papel celofane foram aplicadas:<br>vermelho, verde, azul, amarelo, laraja, rosa, violeta. |

Fonte: Próprio autor.

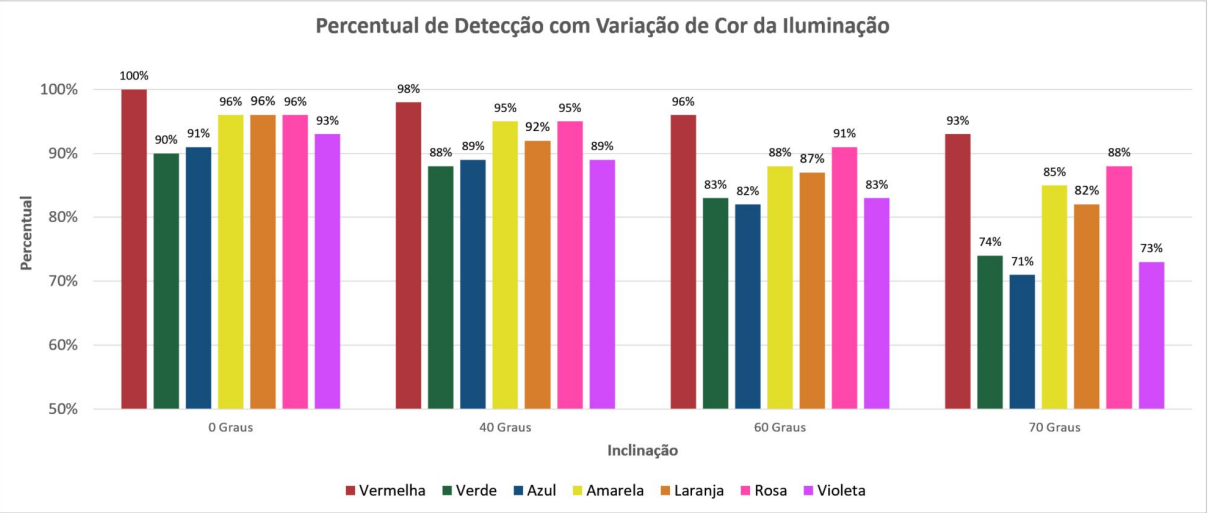
Os resultados da avaliação são discutidos a seguir.

#### *7.3.2.1 Testes sem aplicação de Correspondência de Cor*

Nesta avaliação não foi aplicado o método de Correspondência de Cor, desta forma pode ser verificado quando diferentes tipos de iluminação existentes na cena influenciam na identificação do marcador CRFM. Com base na execução deste teste, pretende-se mostrar como determinadas cores de iluminação interferem de forma negativa no percentual de reconhecimento do algoritmo de detecção. Após medidas as taxas de detecção, foram refeitos os mesmos testes, porém, aplicando a Correspondência de Cor, com o objetivo de comparar as taxas de detecção encontradas.

O Gráfico 7.2 mostra o percentual de detecção do marcador CRFM no cenário estabelecido com diferentes tipos de iluminação aplicadas. Para as amostras analisadas, é possível verificar que as cores de iluminação verde, azul e violeta ocasionam maiores percentuais de erros, sendo possível verificar ainda que, quanto maior a inclinação do marcador, maior passa a ser a taxa de falhas na detecção.

Gráfico 7.2 – Percentual de detecção antes de aplicar Correspondência de Cor.




Fonte: Próprio autor.

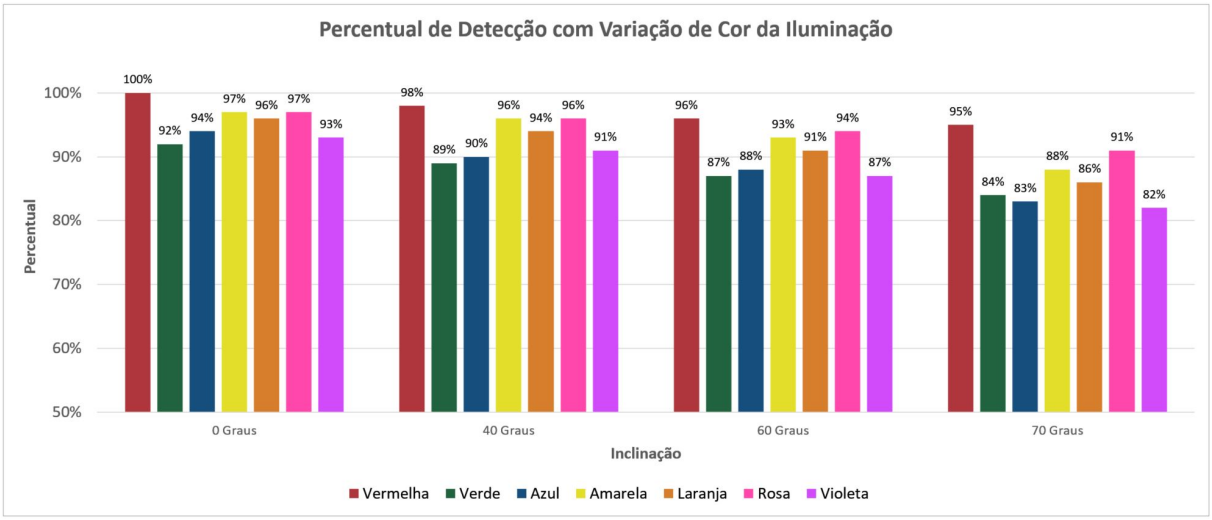
As falhas no reconhecimento do marcador CRFM podem ser atribuídas ao fato de que, algumas cores de iluminação aplicadas afetam mais as cores do marcador do que outras, como é o caso das cores verde, azul e violeta, que ocasionam as piores taxas de detecção, independente da inclinação aplicada. Por outro lado, a cor vermelha é a que menos prejudica a detecção do marcador, independente da inclinação aplicada.

#### *7.3.2.2 Testes com aplicação de Correspondência de Cor*

O mesmo conjunto de testes anterior foi repetido, aplicado desta vez o método de Correspondência de Cor, para cada tipo de iluminação utilizada, detectando a Placa de Correspondência na cena. O objetivo da aplicação deste teste é mostrar que a utilização do processo de Correspondência de Cor pode melhorar os percentuais de identificação do marcador. O Gráfico 7.2 demonstra o percentual de detecção no cenário estabelecido.

Realizando um comparativo dos resultados apresentados nos Gráficos 7.2 e 7.3, é possível verificar que, após a aplicação do método de Correspondência de Cor, ocorreu uma melhora nos percentuais de detecção para todas as cores de iluminação aplicadas, entretanto, as cores verde, azul e violeta, que já haviam demonstrado ser mais sensíveis, pelo cenário criado, demonstraram um percentual maior de melhora na detecção.

Gráfico 7.3 – Percentual de detecção após aplicar Correspondência de Cor.




Fonte: Próprio autor.

Aliado ao processo de Correspondência de Cor, pode ser usada uma estratégia de repetição de projeção em *frames* onde não ocorre detecção do marcador. De forma que, para um determinado número de *frames* subsequentes à detecção do marcador *CRFM*, a projeção seja repetida para melhorar a experiência de interação na aplicação de Realidade Aumentada.

Através desta abordagem, realizando a repetição de projeção para 5 *frames*, por exemplo, subsequentes à detecção do marcador, é possível que neste intervalo o alvo seja reconhecido novamente, possibilitando que o usuário da aplicação não perceba que o marcador deixou de ser reconhecido.

### 7.3.3 Testes de usabilidade

A biblioteca *CRFM Lib* foi apresentada a 10 participantes, entre estudantes de pós-graduação em Ciência da Computação e a programadores que atuam no desenvolvimento de *software*. Entre os participantes da avaliação, todos informaram já conhecer e/ou ter utilizado aplicações de Realidade Aumentada, 60% informaram possuir experiência com o desenvolvimento de jogos, 40% já utilizaram alguma biblioteca para desenvolvimento de aplicações de RA na Unity, destacando o *Vuforia SDK*, entretanto, 10% relataram não conhecer ou possuir apenas conhecimentos superficiais sobre Unity3D.

Explicou-se aos participantes da avaliação a finalidade da biblioteca proposta. Individualmente, cada programador, com auxílio de um tutor, pode utilizar o Unity3D com a ferramenta integrada, para criação de uma aplicação de Realidade Aumentada simples, podendo assim analisar a usabilidade da ferramenta, até mesmo para quem não possuía

experiência com a utilização da *game engine*.

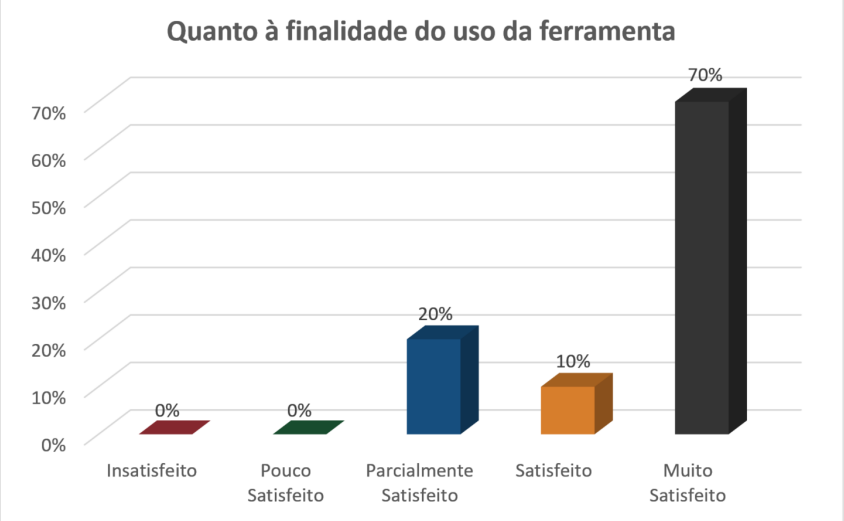
O objetivo do tutor nesta atividade é atuar como um instrutor de utilização da ferramenta *CRFM Lib*, explicando os passos para criação da aplicação. O teste também poderia ser realizado sem a presença do tutor, mas neste caso o participante precisaria ter a disposição a documentação da ferramenta que orientasse os passos de execução com a sequência de desenvolvimento definida.

Após a realização do experimento, cada participante respondeu a um questionário de avaliação da biblioteca *CRFM Lib*.

#### *7.3.3.1 Quanto à finalidade do uso da ferramenta*

Conforme pode ser visualizado no Gráfico 7.4, observa-se que a maioria dos participantes da avaliação consideram-se satisfeitos com a finalidade da biblioteca *CRFM Lib*. Alguns dos participantes comentaram que, para criação de aplicações, é necessário certo conhecimento sobre *Unity Game Engine*.

Gráfico 7.4 – Finalidade de utilização.



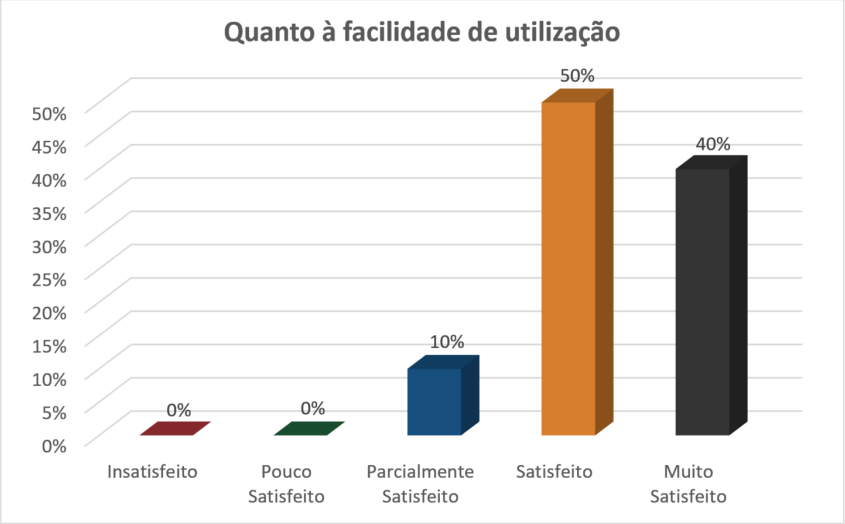

Fonte: Próprio autor.

#### *7.3.3.2 Quanto à facilidade de utilização*

No Gráfico 7.5, pode ser observado que a maioria dos participantes se sentiram satisfeitos quanto a finalidade de utilização da *CRFM Lib*, entretanto 10% demonstraram um parecer neutro quanto a sua análise da ferramenta, sentindo-se parcialmente satisfeitos,

embora exista a possibilidade desta avaliação estar relacionada ao fato do participante não possuir conhecimento sobre Unity3D ou ferramentas para RA.

Gráfico 7.5 – facilidade de utilização.

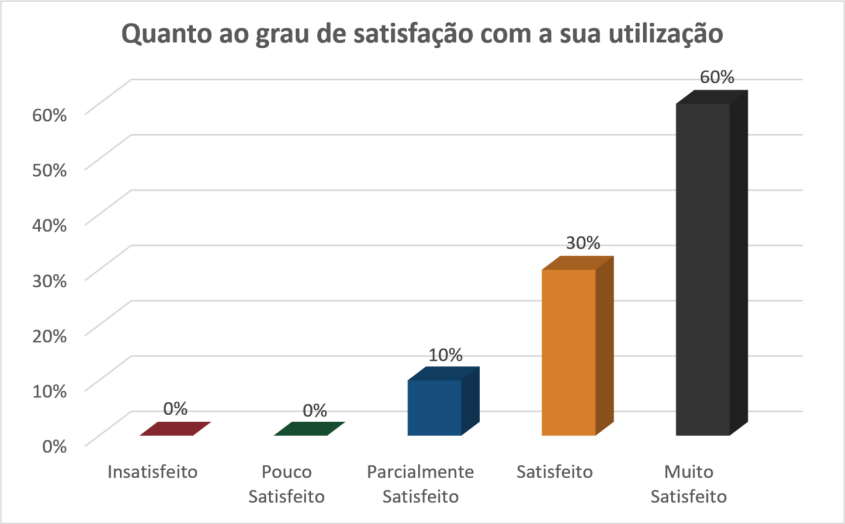



Fonte: Próprio autor.

#### *7.3.3.3 Quanto ao grau de satisfação com a sua utilização*

Observa-se, que no Gráfico 7.6, a grande maioria dos usuários avaliaram como estando satisfeitos com a utilização da ferramenta. Alguns participantes sugeriram que a biblioteca também deveria suportar a criação de aplicações para plataformas *mobile*. Tal sugestão é destacada como trabalho futuro na Seção 8.1.

Gráfico 7.6 – Satisfação de utilização.

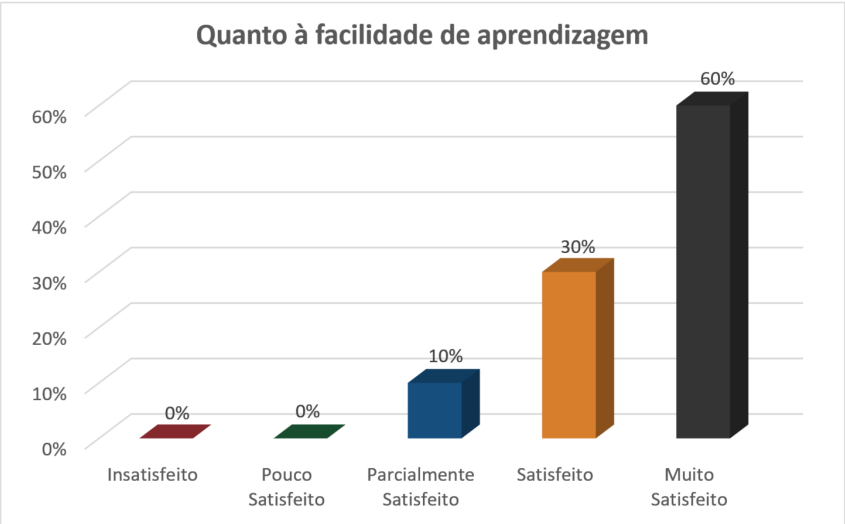



Fonte: Próprio autor.

### *7.3.3.4 Quanto à facilidade de aprendizagem*

O Gráfico 7.7 demonstra que a maior parte dos usuários ficaram bastante satisfeitos em relação a aprendizagem. Estes participantes ainda destacaram a facilidade de entendimento e agilidade proporcionada para criação de uma aplicação de RA no *Unity*, mesmo que esta seja bastante simples, com um baixo grau de complexidade.

Gráfico 7.7 – Facilidade de aprendizagem.

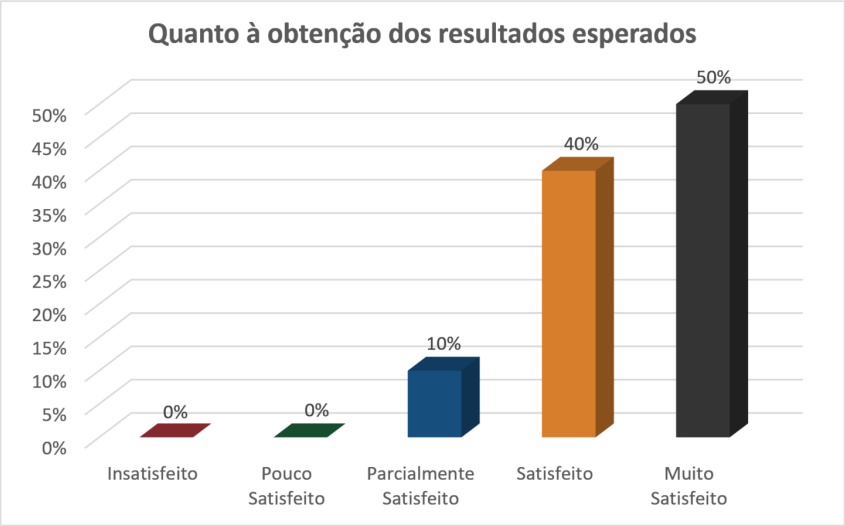



Fonte: Próprio autor.

#### *7.3.3.5 Quanto à obtenção dos resultados esperados*

Através da observação do Gráfico 7.8, é possível verificar que a maioria dos participantes da avaliação consideram que os resultados alcançados com a utilização de *CRFM Lib* foram satisfatórios.

Gráfico 7.8 – Obtenção dos resultados esperados.




Fonte: Próprio autor.

## 7.4 CONSIDERAÇÕES FINAIS SOBRE O CAPÍTULO

Os resultados apresentados neste capítulo demonstraram a possibilidade de utilização da ferramenta *CRFM Lib* no desenvolvimento de aplicações de Realidade Aumentada. A partir da análise gráfica dos percentuais de detecção, é possível verificar a possibilidade de detecção de marcadores CRFM até mesmo em situações de grande rotação aplicada, como 70°, conforme demonstrado na avaliação apresentada.

O percentual de detecções aparenta chegar em um limite onde o percentual de reconhecimento ainda é aceitável. Desta forma foi verificado que quanto maior a distância aplicada entre marcador e a câmera, pior será o desempenho de detecção em situações de rotação aplicadas superior a 60°.

Com base nos resultados obtidos com a medição dos percentuais de detecção em diferentes cores de iluminação aplicada, foi possível verificar que a aplicação do método de Correspondência de Cor, para medir a forma como a cor é encontrada na cena, conseguiu melhorar os percentuais de detecção, inclusive das cores que mais sofreram influência pelo tipo de iluminação simuladas neste trabalho.

Por fim, a análise do questionário aplicado indica que os participantes da avaliação

demonstraram satisfação com a utilização da ferramenta. Os participantes concordam que a utilização da ferramenta desenvolvida facilita o processo de desenvolvimento de aplicações de Realidade Aumentada na *Unity Game Engine*. Apesar dos resultados positivos obtidos com a avaliação aplicada aos participantes, foram sugeridas por estes, algumas melhorias, como por exemplo, a possibilidade de compatibilidade da aplicação desenvolvida com a ferramenta, também para plataformas *mobile*.

# 8 CONCLUSÃO

As diferentes tecnologias disponíveis para implementação de sistemas de Realidade Aumentada têm possibilitado a sua difusão nas mais variadas áreas de conhecimento, seja a nível de desenvolvimento de aplicações complexas utilizadas em processos específicos, seja a nível de disseminação da sua utilização por parte do usuário comum. A variedade de ferramentas disponíveis para implementar aplicações de RA demonstra a crescente popularidade desta área.

A agilidade proporcionada por estas ferramentas, em específico as que possibilitam integração com a *Unity Game Engine* para desenvolvimento é um dos principais fatores que podem garantir a popularidade cada vez maior da Realidade Aumentada. Desta forma, cada vez mais, novas ferramentas surgirão para possibilitar a criação deste tipo de aplicação, de forma cada vez mais fácil.

No contexto de ferramentas que podem ser integradas à *Unity*, CRFM Lib é projetada para o desenvolvimento de aplicações de RA, utilizando como alvo de referência para reconhecimento na cena, o marcador fiducial CRFM, com suporte a plataforma *Windows*. O sentido de oportunidade deste trabalho incide sobre a possibilidade de providenciar, através da integração com a *Unity Game Engine*, uma ferramenta para a criação de aplicações, focada na análise das cores e verificar como são percebidas na cena, para identificação dos marcadores.

Ao utilizar cores para composição das hierarquias dos marcadores, são enfrentados problemas que marcadores binários que usam apenas as cores preto e branco para formar seus *bits* internos não enfrentam, uma vez que a variação das condições de iluminação tem maior impacto na identificação de marcadores coloridos, de forma que em condições extremas de iluminação a precisão da avaliação pode ser pior do que os marcadores de binários tradicionais. Neste sentido foi proposto um processo de Correspondência de Cor, para medir como a cor é percebida na cena. Esse método, obtém inicialmente os valores das cores de interesse e aplica no algoritmo de identificação, visando diminuir a diferença entre as cores alvo e as cores conforme são percebidas na cena.

## 8.1 TRABALHOS FUTUROS

De maneira geral, pretende-se continuar a explorar as possibilidades com a integração da ferramenta junto a *Unity Game Engine*. Inicialmente sugere-se a realização de estudos sobre a arquitetura de plataformas *mobile* com o objetivo de adaptar a ferramenta, possibilitando construir aplicações no *Unity3D* também para estas plataformas. Assim, possibilitando avaliar a eficiência do algoritmo de detecção e projeção também em

plataformas *mobile*.

Outra implementação sugerida para trabalho futuro é adaptar a ferramenta *CRFM Lib* para possibilitar que a aplicação criada também seja compatível com óculos inteligentes de Realidade Aumentada, possibilitando assim uma forma de visualização que permita experiências interativa de Realidade Aumentada, sem que o usuário necessite ficar segurando o dispositivo para visualização.

Como neste trabalho a projeção é realizada a cada *frame* em que o marcador *CRFM* é detectado, também se sugere como trabalho futuro, a implementação de um algoritmo de *tracking* para possibilitar maior estabilidade na projeção dos objetos tridimensionais.

## REFERÊNCIAS BIBLIOGRÁFICAS


ARAMESH, R. et al. Hydrodynamics and particle mixing/segregation measurements in an industrial gas phase olefin polymerization reactor using image processing technique and cfd-pbm model. **Measurement**, v. 83, p. 106 – 122, 2016. ISSN 0263-2241. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0263224116000476>.

ARTOOLKIT. **ARToolKit Documentation**. 2017. (Accessado em 12/11/2017). Disponível em: <https://www.artoolkit.org/documentation/>.

AZEVEDO, E.; CONCI, A. **Computação Gráfica: Teoria e Prática**. Rio de Janeiro: Elsevier, 2003. ISBN 9788535212525.

AZUMA, R. A survey of augmented reality. **Presence: Teleoperators and virtual environments**, MIT Press, v. 6, n. 4, p. 355–385, 1997.

BACKES, A. R.; JUNIOR, J. J. de M. S. **Introdução à Visão Computacional Usando MATLAB**. 1st. ed. [S.l.]: Alta Books Editora, 2016. ISBN 978-85-508-0023-3.

BEHESHTI, J. Virtual environments for children and teens. In: EICHENBERG, C. (Ed.). **Virtual Reality in Psychological, Medical and Pedagogical Applications**. Rijeka: InTech, 2012. cap. 13. Disponível em: <http://dx.doi.org/10.5772/51628>.

BLACKMAN, S. **Unity for Absolute Beginners**. 1st edition. ed. [S.l.]: Apress, 2014. ISBN 978-1-4302-6779-9.

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.

CAUDELL, T.; MIZELL, D. W. Augmented reality: An application of heads-up display technology to manual manufacturing processes. In: IEEE. **System Sciences, 1992. Proceedings of the Twenty-Fifth Hawaii International Conference on**. [S.l.], 1992. v. 2, p. 659–669.

CHARISIS, V. S. et al. Capsule endoscopy image analysis using texture information from various colour models. **Computer Methods and Programs in Biomedicine**, v. 107, n. 1, p. 61 – 74, 2012. ISSN 0169-2607. Advances in Biomedical Engineering and Computing: the MEDICON conference case. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0169260711002689>.

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>.

CIE. Improvement to industrial colour-difference evaluation. **Vienna: Central Bureau of the CIE**, 142–2001.

CRFMLIB. **CRFMLib For Unity Video Samples**. 2018. Disponível em: <https://www.youtube.com/watch?v=Rm1pT-7ajY8&list=PLvPvzovuVUkm1WWka7Xj4OghAnQUDSSMk>.

CRYENGINE. **CryENGINE website**. 2017. (Accessado em 25/11/2017). Disponível em: <https://www.cryengine.com/>.

CUSHNAN, D.; HABBAK, H. E. **Developing AR Games for iOS and Android**. [S.l.]: Packt Publishing, 2013. ISBN 978-1-78328-003-2.

DAUGMAN, J. G. Two-dimensional spectral analysis of cortical receptive field profiles. **Vision Research**, v. 20, n. 10, p. 847 – 856, 1980. ISSN 0042-6989. Disponível em: <http://www.sciencedirect.com/science/article/pii/0042698980900656>.

____. Uncertainty relation for resolution in space, spatial frequency, and orientation optimized by two-dimensional visual cortical filters. **J. Opt. Soc. Am. A**, OSA, v. 2, n. 7, p. 1160–1169, Jul 1985. Disponível em: <http://josaa.osa.org/abstract.cfm?URI=josaa-2-7-1160>.

DAVIES, E. R. **Computer and machine vision: theory, algorithms, practicalities**. [S.l.]: Academic Press, 2012. ISBN 9780123869081.

DAWSON-HOWE, K. **A Practical Introduction to Computer Vision with OpenCV**. 1st. ed. [S.l.]: Wiley Publishing, 2014. ISBN 1118848454, 9781118848456.

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

DEMAAGD, K. et al. **Practical Computer Vision with SimpleCV: The Simple Way to Make Technology See**. [S.l.]: O'Reilly Media, Inc., 2012. ISBN 1449320368, 9781449320362.

DORAN, J. P. **Unreal Engine Game Development Cookbook**. [S.l.]: Packt Publishing, 2015. ISBN 978-1-78439-816-3.

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.

EASYAR. **EasyAR Documentation**. 2017. (Accessado em 13/11/2017). Disponível em: <https://www.easyar.com/doc/>.

ENOXSOFTWARE. **OpenCV for Unity**. 2017. (Accessado em 21/11/2017). Disponível em: <https://www.assetstore.unity3d.com/en/#!/content/21088>.

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. **Algoritmo CIE Delta E 2000**. 2017. Disponível em: <https://github.com/gfiumara/CIEDE2000>.

FURNESS, I. T. A. The super cockpit and its human factors challenges. **Proceedings of the Human Factors Society Annual Meeting**, v. 30, n. 1, p. 48–52, 1986. Disponível em: <https://doi.org/10.1177/154193128603000112>.

GARCIA, G. B. et al. **Learning Image Processing with OpenCV**. [S.l.]: Packt Publishing, 2015. ISBN 1783287659, 9781783287659.

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>.

GIROLAMI, A. et al. Measurement of meat color using a computer vision system. **Meat Science**, v. 93, n. 1, p. 111 – 118, 2013. ISSN 0309-1740. Disponível em: <http://www.sciencedirect.com/science/article/pii/S030917401200277X>.

GONZALEZ, R. C.; WOODS, R. E. **Digital Image Processing (3rd Edition)**. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 2006. ISBN 013168728X.

GONZALEZ, R. C.; WOODS, R. E.; EDDINS, S. L. **Digital Image Processing Using MATLAB**. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 2003. ISBN 0130085197.

GRAVDAL, E. Augmented reality and object tracking for mobile devices. Institutt for teknisk kybernetikk, 2012.

GRUBERT, J.; GRASSET, R. **Augmented Reality for Android Application Development**. [S.l.]: Packt Publishing, 2013. ISBN 978-1-78216-855-3.

GUIMARãES, L. **A cor como informação: a construção biofísica, linguistica e cultural da simbologia das cores**. 3nd. ed. [S.l.]: Annablume, 2004. ISBN 9788574191683.

GUNDLACH, S.; MARTIN, M. K. **Mastering CryENGINE**. [S.l.]: Packt Publishing, 2014. ISBN 1783550252, 9781783550258.

HARALICK, R. M.; SHANMUGAM, K.; DINSTEIN, I. Textural features for image classification. **IEEE Transactions on Systems, Man, and Cybernetics**, SMC-3, n. 6, p. 610–621, Nov 1973. ISSN 0018-9472.

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>.

HOCKING, J. **Unity in Action**. [S.l.]: PManning Publications, 2015. ISBN 9781617292323.

HUNT, R. W. G.; POINTER, M. R. **Measuring Colour**. [S.l.]: Wiley, 2011. ISBN 978-1-119-97537-3.

JACKSON, S. **Mastering Unity 2D Game Development**. [S.l.]: Packt Publishing, 2014. ISBN 1849697345, 9781849697347.

JUDD, D. B.; WYSZECKI, G. Color in business, science and industry. Wiley, 1975.

KASS, M.; WITKIN, A.; TERZOPOULOS, D. Snakes: Active contour models. **International Journal of Computer Vision**, v. 1, n. 4, p. 321–331, Jan 1988. ISSN 1573-1405. Disponível em: <https://doi.org/10.1007/BF00133570>.

KATO, H.; BILLINGHURST, M. Marker tracking and hmd calibration for a video-based augmented reality conferencing system. In: IEEE. **Augmented Reality, 1999.(IWAR'99) Proceedings. 2nd IEEE and ACM International Workshop on**. [S.l.], 1999. p. 85–94.

KIM, S. H.; KIM, J. G.; YANG, T. K. Autonomous slam technique by integrating grid and topology map. In: **2008 International Conference on Smart Manufacturing Application**. [S.l.: s.n.], 2008. p. 413–418.

KIRNER, C.; KIRNER, T. G. Virtual reality and augmented reality applied to simulation visualization. In: **Simulation and Modeling: Current Technologies and Applications**. [S.l.]: IGI Global, 2008. p. 391–419.

KIRNER, C.; SISCOUTTO, R. **Realidade Virtual e Aumentada: Conceitos, Projeto e Aplicações**. 1nd. ed. [S.l.]: Editora SBC – Sociedade Brasileira de Computação, 2007. ISBN 85-7669-108-6.

KLETTE, R. **Concise Computer Vision: An Introduction into Theory and Algorithms**. [S.l.]: Springer Publishing Company, Incorporated, 2014. ISBN 1447163192, 9781447163190.

KOSCHAN, A.; ABIDI, M. A. **Digital Color Image Processing**. New York, NY, USA: Wiley-Interscience, 2008. ISBN 0470147083, 9780470147085.

KREPS, D.; FLETCHER, G.; GRIFFITHS, M. **Technology and Intimacy: Choice or Coercion : 12th IFIP TC 9 International Conference on Human Choice and Computers, HCC12 2016, Salford, UK, September 7-9, 2016**. [S.l.]: Springer, 2016. ISSN 1868-4238. ISBN 978-3-319-44804-6.

KUDAN. **The Kudan Developer Hub**. 2017. (Accessado em 18/07/2017). Disponível em: <https://kudan.readme.io/>.

KUEHNI, R. G. **Color Space and Its Divisions: Color Order from Antiquity to the Present**. [S.l.]: Wiley, 2003. ISBN 9780471326700.

LAGANIERE, R. **OpenCV Computer Vision Application Programming Cookbook**. 2nd. ed. [S.l.]: Packt Publishing, 2014. ISBN 1782161481, 9781782161486.

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.

LOWE, D. G. Distinctive image features from scale-invariant keypoints. **Int. J. Comput. Vision**, Kluwer Academic Publishers, Hingham, MA, USA, v. 60, n. 2, p. 91–110, nov. 2004. ISSN 0920-5691. Disponível em: <https://doi.org/10.1023/B:VISI.0000029664.99615.94>.

LUO, X. The cloud-mobile convergence paradigm for augmented reality. **Augmented Reality - Some Emerging Application Areas**, InTech, p. 33–58, 2011.

MALLAT, S. G. A theory for multiresolution signal decomposition: The wavelet representation. **IEEE Trans. Pattern Anal. Mach. Intell.**, IEEE Computer Society, Washington, DC, USA, v. 11, n. 7, p. 674–693, jul. 1989. ISSN 0162-8828. Disponível em: <http://dx.doi.org/10.1109/34.192463>.

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>.

MILGRAM, P. et al. Augmented reality: A class of displays on the reality-virtuality continuum. In: . [S.l.: s.n.], 1994. p. 282–292.

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.

MUMFORD, D.; SHAH, J. Optimal approximation by piecewise smooth function and associated variational problems. v. 42, 07 1989.

NAIMARK, L.; FOXLIN, E. Circular data matrix fiducial system and robust image processing for a wearable vision-inertial self-tracker. In: **Proceedings of the 1st International Symposium on Mixed and Augmented Reality**. Washington, DC, USA: IEEE Computer Society, 2002. (ISMAR '02), p. 27–. ISBN 0-7695-1781-1. Disponível em: <http://dl.acm.org/citation.cfm?id=850976.854961>.

NICKELS, S. et al. Proteinscanar-an augmented reality web application for high school education in biomolecular life sciences. In: IEEE. **Information Visualisation (IV), 2012 16th International Conference on**. [S.l.], 2012. p. 578–583.

OLSON, E. AprilTag: A robust and flexible visual fiducial system. In: **Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)**. [S.l.]: IEEE, 2011. p. 3400–3407.

OPENCV. **Open Source Computer Vision Library**. 2017. Disponível em: <https://github.com/itseez/opencv>.

PTC. **PTC to Acquire Augmented Reality Leader Vuforia From Qualcomm**. 2015. (Accessado em 12/05/2017). Disponível em: <https://www.ptc.com/en/news/2015/ptc-to-acquire-vuforia-from-qualcomm>.

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

ROBERTS, L. G. Machine perception of three-dimensional solids. Massachusetts Institute of Technology, 01 1963. Disponível em: <http://www.packet.cc/files/mach-per-3D-solids.html>.

ROSI, T. et al. What are we looking at when we say magenta? quantitative measurements of rgb and cmyk colours with a homemade spectrophotometer. **European Journal of Physics**, v. 37, n. 6, p. 065301, 2016. Disponível em: <http://stacks.iop.org/0143-0807/37/i=6/a=065301>.

ROWLEY, H. A.; BALUJA, S.; KANADE, T. Neural network-based face detection. **IEEE Transactions on Pattern Analysis and Machine Intelligence**, v. 20, n. 1, p. 23–38, Jan 1998. ISSN 0162-8828.

SCHWAB, B. **Ai Game Engine Programming**. [S.l.]: Course Technology, 2009. ISBN 978-1-5845-0572-3.

SCHWEIGHOFER, G.; PINZ, A. Robust pose estimation from a planar target. **IEEE Trans. Pattern Anal. Mach. Intell.**, IEEE Computer Society, Washington, DC, USA, v. 28, n. 12, p. 2024–2030, dez. 2006. ISSN 0162-8828. Disponível em: <http://dx.doi.org/10.1109/TPAMI.2006.252>.

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>.

SHERIF, W.; WHITTLE, S. **Unreal Engine 4 Scripting with C++ Cookbook**. [S.l.]: Packt Publishing, 2016. ISBN 978-1-78588-554-9.

SHI, S. **Emgu CV Essentials**. [S.l.]: Packt Publishing, 2013. ISBN 1783559527, 9781783559527.

STEINICKE, F. et al. Poster: A virtual body for augmented virtuality by chroma-keying of egocentric videos. In: **Proceedings of the IEEE Symposium on 3D User Interfaces (3DUI) (Poster Presentation)**. IEEE Press, 2009. p. 125–126. Disponível em: <http://basilic.informatik.uni-hamburg.de/Publications/2009/SBRH09a>.

SURAL, S.; QIAN, G.; PRAMANIK, S. Segmentation and histogram generation using the hsv color space for image retrieval. In: **Proceedings. International Conference on Image Processing**. [S.l.: s.n.], 2002. v. 2, p. II–589–II–592 vol.2. ISSN 1522-4880.

SUTHERLAND, I. A head-mounted three dimensional display. In: ACM. **Proceedings of the December 9-11, 1968, fall joint computer conference, part I**. [S.l.], 1968. p. 757–764.

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>.

THORN, A. **Mastering Unity 5.x**. [S.l.]: Packt Publishing, 2017. ISBN 9781785880742.

TRACY, S.; REINDELL, P. **CryENGINE 3 Game Development: Beginner's Guide**. [S.l.]: Packt Publishing, 2012. ISBN 9781849692007.

TRUCCO, E.; VERRI, A. **Introductory techniques for 3-D computer vision.** [S.l.: s.n.], 1998. ISBN 978-0-13-261108-4.

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.

UNREALENGINE. **Unreal Engine website**. 2017. (Accessado em 21/11/2017). Disponível em: <https://www.unrealengine.com/>.

VIOLA, P.; JONES, M. J. Robust real-time face detection. **International Journal of Computer Vision**, v. 57, n. 2, p. 137–154, May 2004. ISSN 1573-1405. Disponível em: <https://doi.org/10.1023/B:VISI.0000013087.49260.fb>.

VUFORIA. **Vuforia Developer Library**. 2017. (Accessado em 13/11/2017). Disponível em: <https://library.vuforia.com/>.

VXL. **VXL Developers**. 2013. (Accessado em 20/10/2017). Disponível em: <https://public.kitware.com/vxl/doc/release/books/core/book.html>.

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

WANG, J.; OLSON, E. AprilTag 2: Efficient and robust fiducial detection. In: **Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)**. [S.l.: s.n.], 2016.

WANG, X. et al. Integrating augmented reality with building information modeling: Onsite construction process controlling for liquefied natural gas industry. **Automation in Construction**, v. 40, p. 96 – 105, 2014. ISSN 0926-5805. Disponível em: <http://www.sciencedirect.com/science/article/pii/S092658051300215X>.

WIKITUDE. **Wikitude Documentation**. 2017. (Accessado em 18/07/2017). Disponível em: <https://www.wikitude.com/documentation/>.

WU, D.; SUN, D.-W. Colour measurements by computer vision for food quality control – a review. **Trends in Food Science & Technology**, v. 29, n. 1, p. 5 – 20, 2013. ISSN 0924-2244. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0924224412001835>.