

PONTIFÍCIA UNIVERSIDADE CATÓLICA DE GOIÁS
ESCOLA DE CIÊNCIAS EXATAS E DA COMPUTAÇÃO
GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO



**PUC
GOIÁS**

AR HAND TRACKING

JONLENES SILVA DE CASTRO

GOIÂNIA
2017

JONLENES SILVA DE CASTRO

AR HAND TRACKING

Trabalho de Conclusão de Curso apresentado à Escola de Ciências Exatas e da Computação, da Pontifícia Universidade Católica de Goiás, como parte dos requisitos para a obtenção do grau de Bacharel em Ciência da Computação.

Orientador: Prof. Me. Anibal Santos Jukemura

GOIÂNIA
2017

JONLENES SILVA DE CASTRO

AR HAND TRACKING

Este Trabalho de Conclusão de Curso foi julgado adequado para obtenção do grau de Bacharel em Ciência da Computação e aprovado em sua forma final pela Escola de Ciências Exatas e da Computação, da Pontifícia Universidade Católica de Goiás, em ____/12/2017.

Profª. Ma. Ludmilla Reis Pinheiro dos Santos
Coordenadora de Trabalho de Conclusão de Curso

Banca examinadora:

Orientador: Prof. Me. Anibal Santos Jukemura

Prof. Me. Fábio Gomes de Assunção

Prof. Me. Max Gontijo De Oliveira

GOIÂNIA
2017

RESUMO

Este trabalho apresenta a construção de um sistema Realidade Aumentada sem Marcadores, com funcionamento em tempo real que utiliza a técnica de tracking da mão do usuário com uma luva de cor preta para posicionar as informações virtuais sobre o ambiente real. Técnicas de Realidade Aumentada dependem da recuperação de informações virtuais através de uma câmera, permitindo a leitura de cada *frame*, estabelecendo uma sequência de imagens para corretamente associar informações 3D às cenas reais. Através da reconstrução *frame-a-frame* da posição da câmera em relação à mão, pode se estabilizar as informações virtuais 3D no topo da mão, permitindo que o usuário inspecione esses objetos virtuais convenientemente de diferentes ângulos de visão. Através da pesquisa realizada, foi possível utilizar algoritmos de reconhecimento baseado nas curvaturas das pontas dos dedos e da utilização do filtro de *Kalman* para suavizar a pose da câmera. A implementação apresentada neste trabalho permitiu recuperar as informações da câmera utilizadas para realizar a inserção dos objetos virtuais no mundo real, gerando resultados com bom índice de robustez com relação a movimentação, alteração de posição da câmera e pequenas alterações de ambientes e iluminação.

Palavras Chave: Realidade Aumentada Sem Marcadores. *Hand Tracking*

ABSTRACT

This essay shows the building process of an Augmented Reality Markerless, with real-time execution, and utilizes a ‘tracking’ technique, where the user can position the virtual information upon the real environment wearing a black glove. Augmented Reality Techniques depend on the recovering of virtual information by means of a camera, allowing the capture of each frame, establishing a sequence of images to accurately associate 3D information to real scenes. Through the frame-by-frame reconstruction of the camera position with respect to the hand, the information can be placed on top of the hand, making possible for the user to parse different perspectives. Through the medium of the research’s results it was possible to make use of the recognition algorithms based on the curvature of the points in the fingers and the Kalman’s filters, to precise the camera’s next moves. The presented implementation in this essay made possible to recover the camera’s previous information to insert the virtual objects in the real world, generating results with a rather good level of quality relative to its movements, some changes on the environment and its lighting.

Key Words: Augmented Reality Markerless. Hand. Tracking.

LISTA DE ILUSTRAÇÕES

1	Rastreamento para RA baseado em marcadores (esquerda) e sem marcadores (direita).	11
2	Aplicação de RA sem marcadores para manutenção de equipamentos.	12
3	Combinação do mundo real com o virtual.	17
4	HeadsUp Displays.	18
5	RA na medicina.	18
6	RA na educação.	19
7	Fluxograma de calibração única e estimativa de <i>pose</i> de câmera em tempo real usando o rastreamento de ponta do dedo.	21
8	Os parâmetros extrínsecos definem a rotação R e translação t , responsáveis por levar o sistema de coordenadas de mundo O para o sistema de coordenadas da câmera C.	25
9	Coordenadas aproximadas de um modelo de mão 3D.	33
10	Etapas da detecção	37
11	Exemplo da utilização da função <i>findContornos</i>	37
12	Aplicação da transformada da distância em uma imagem.	38
13	Aplicação da transformada da distância em uma imagem.	39
14	Processo inicial	46
15	Aplicação dos filtros	46
16	Limiar e transformada da distância	46
17	Contorno da maior região encontrada	47
18	Segmentação inicial	47
19	Resultado apresentado apos todo o processamento	48
20	Resultados	48
21	Resultados	49
22	Resultados	49
23	Resultados	50
24	Resultados	50
25	Resultados	51
26	Resultados	51

LISTA DE ALGORITMOS

1	Fluxo de execução do sistema MAR desenvolvido	32
2	Filtro por intervalo de valores nos canais (1)	35

LISTA DE SIGLAS

RA = <i>Realidade Aumentada</i>	11
PDI = Processamento Digital de Imagens	11
GPS = <i>Global Positioning System</i>	11
CG = Computação Gráfica	11
VC = Visão Computacional	11
IDE = Ambiente de Desenvolvimento Integrado	13
MVS = <i>Microsoft Visual Studio</i>	13
VI = Visualização de Informações	15
RV = Realidade Virtual	16
HUD = <i>HeadsUp Displays</i>	17
SDK = <i>Kit de Desenvolvimento de Software</i>	19
6DoF = <i>Six Degrees-of-Freedom</i>	21
PnP = <i>Perspective-n-Point</i>	25
DLL = <i>Dynamic Linked Library</i>	30
IPL = <i>Image Processing Library</i>	30
API = <i>Application Programming Interface</i>	31

SUMÁRIO

1 INTRODUÇÃO	11
1.1 Motivação e Justificativa	12
1.2 Objetivo	13
1.2.1 Objetivos específicos	13
1.3 Metodologia	13
1.4 Estrutura do trabalho	14
2 REFERENCIAL TEÓRICO	15
2.1 Visualização de Informações	15
2.2 Processamento Digital de Imagens	15
2.3 Visão Computacional	15
2.4 Computação Gráfica	16
2.5 Realidade Aumentada	16
2.5.1 Aplicações	17
2.5.2 Ferramentas	19
2.6 Trabalhos relacionados	20
2.6.1 Realidade Aumentada sem Marcadores Baseado em Areias	20
2.6.2 <i>HandyAR</i>	20
3 SISTEMAS DE REALIDADE AUMENTADA SEM MARCADORES	23
3.1 Parâmetros da câmera	23
3.1.1 Parâmetros <i>intrínsecos</i>	24
3.1.2 Parâmetros <i>extrínsecos</i>	25
3.2 Pose Estimation	25
3.2.1 <i>Perspective-n-Point</i>	26
3.3 Segmentação	26
3.4 Reconhecimento	27
3.5 Rastreamento	27
4 METODOLOGIA E IMPLEMENTAÇÃO	29
4.1 Plataforma de Desenvolvimento	29
4.1.1 Linguagem C++	29
4.1.2 <i>Microsoft Visual Studio</i>	30
4.2 Bibliotecas de Apoio	30
4.2.1 OpenCV	30

4.2.2	OpenGL	31
4.3	Arquitetura do Software	31
4.4	Fluxo da aplicação	32
4.5	Inicialização	33
4.5.1	Carregamento do modelo de mão 3D	33
4.5.2	Carregamento dos parâmetros intrínsecos da câmera	33
4.6	Etapas do Processamento	34
4.6.1	Segmentação	34
4.6.2	Detecção dos dedos	36
4.6.2.1	Etapa 1 - Encontrar os contornos	37
4.6.2.2	Etapa 2 - Encontrar a maior área	38
4.6.2.3	Etapa 3 - Detecção pela curvatura	39
4.6.2.4	Etapa 4 - Ajuste por elipse	40
4.6.2.5	Etapa 5 - Validação de candidatos	41
4.6.3	Rastreamento dos dedos	42
4.6.4	<i>Pose Estimation</i> das extremidades dos dedos	42
4.6.5	Configuração da câmera e renderização	43
5	RESULTADOS	45
5.1	Resultados da segmentação e detecção	45
5.2	Ambientes 1	47
5.3	Ambiente 2	48
6	CONCLUSÕES	52
6.1	Trabalhos futuros	52
REFERÊNCIAS		53
Apêndice A – Módulos		57
A.1	Módulo <i>Capture</i>	57
A.2	Módulo <i>HandRegion</i>	58
A.3	Módulo <i>FingertipProcess</i>	59
A.4	Módulo <i>FingertipTracker</i>	65
A.5	Módulo <i>PoseEstimation</i>	74
A.6	Módulo <i>FingertipPoseEstimation</i>	75
Anexo A – Arquivo utilizados pelo sistema		88
A.1	Arquivo de calibração da câmera	88

A.2 Arquivo com a posição das pontas dos dedos	88
--	----

1 INTRODUÇÃO

A Realidade Aumentada (RA) pode ser definida como o enriquecimento de um ambiente real com objetos virtuais (2). Consequentemente, obtém-se ambientes abstratos que combinam o mundo factual observado pelo usuário com um cenário projetado por um computador e, que acrescente a este mundo novas informações (3). Associando conhecimentos de áreas afins como Computação Gráfica (CG), Processamento Digital de Imagens (PDI) e Visão Computacional (VC), a RA se propõe a expandir as possibilidades e experiências dos usuários (4).

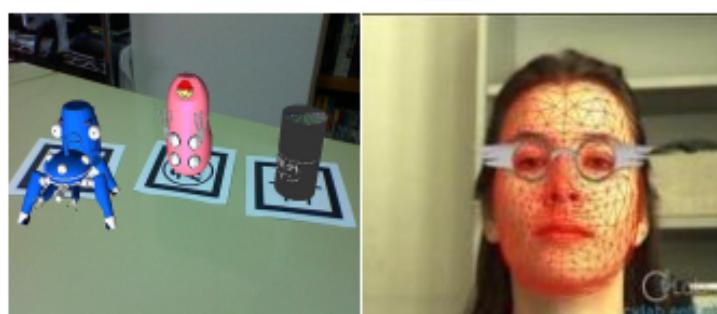
Este processo deve ser feito em tempo real e de forma que os elementos virtuais pareçam fazer parte do ambiente real tridimensional (5). Para isso, são utilizadas técnicas de reconhecimento e rastreamento, responsáveis por identificar características do ambiente factual e determinar o correto posicionamento do objeto virtual em relação à este mundo (5). O reconhecimento em si, visa identificar itens presentes em uma imagem e diferenciá-los das demais informações apresentadas, conforme algum modelo preestabelecido, enquanto o rastreamento objetiva determinar a posição ao longo do tempo do objeto que está sendo rastreado, em uma sequência de imagens (6).

Neste contexto, o rastreamento é uma etapa essencial para os sistemas de RA, que pode utilizar diferentes tipos de sensores, tais como vídeo, *Global Positioning System* (GPS) e rastreadores iniciais (5).

O rastreamento baseado em vídeo, foco deste trabalho, é bastante utilizado devido a requisitos de custo, precisão e robustez (5). Dois tipos de rastreamento com vídeo podem ser citados: o baseado em marcadores e o sem marcadores (5).

O primeiro tipo de rastreamento utiliza, normalmente, marcadores fiduciais utilizados como referência por um sistema de visão computacional, pois tem suas características previamente estabelecidas ao detector, para que este possa realizar a detecção e inferir a sua posição (7). No rastreamento sem marcadores, ao invés de usar elementos artificiais, tais como, marcadores fiduciais, são exploradas características naturais presentes no mundo real (5). Ambos tipos de rastreamento podem ser observados na Figura 1.

Figura 1: Rastreamento para RA baseado em marcadores (esquerda) e sem marcadores (direita).



Fonte: (5).

O rastreamento (*tracking*) de características naturais é utilizada por sistema de Realidade Aumentada sem Marcadores (MAR). Por meio do *tracking* é realizada a correta inserção dos elementos virtuais na cena real, utilizando informações presente na imagem, tais como arestas, texturas ou a própria estrutura da cena, sem a inserção de marcadores no ambiente (5).

Neste contexto, um sistema MAR consiste em reconhecer um objeto apresentado em uma sequência de *frames*, utilizar as técnicas de rastreamento para inferir sua posição ao longo do tempo e posicionar elementos virtuais nas sequências de imagens resultantes para o usuário. O MAR é o sistema que será desenvolvido neste trabalho, conforme justificado a seguir.

1.1 Motivação e Justificativa

Devido as perspectivas de aplicação nas mais diversas áreas, tais como construção civil, aviação e engenharias, as técnicas de MAR têm sido bastante estudadas e discutidas (4), entretanto, atualmente a RA ainda está associada a utilização de marcadores fiduciais (8). Comumente estes estão apresentados em papéis que contém códigos e conseguem sobrepor em si o objeto que será visualizado.

O exemplo apresentado por (5), onde um sistema MAR é desejável, percebe-se o uso de sistemas de suporte à manutenção, como o ilustrado na Figura 2, que sobrepõe instruções e informações à imagem capturada, em tempo real, sobre os procedimentos a serem realizados. Esses tipos de sistemas podem ser utilizados como ferramenta para orientação em tarefas de manutenção e inspeção e, até mesmo para treinamento. O objetivo é prover ao usuário as informações necessárias para a correta realização de ações preventivas ou corretivas através de uma interface gráfica 3D intuitiva. Este sistema de RA representa uma alternativa mais prática ao uso de manuais de papel com centenas de esquemáticos 2D.

Figura 2: Aplicação de RA sem marcadores para manutenção de equipamentos.



Fonte: (5).

Assim como o exemplo apresentado anteriormente, existem diversos sistemas onde a utilização da MAR é desejável, pois a mesma apresenta algumas vantagens, tais como reconhecimento de objetos específicos conhecidos previamente e, simulação de interações entre os mundos

real e virtual (5). Porém, ainda não existem muitas bibliotecas ou ferramentas que permitam a construção de tal sistema (9). Neste contexto, este trabalho propõe a construção de um sistema MAR que utilize o rastreamento da mão do usuário.

Mais especificamente, propõe-se o reconhecimento e o rastreamento da mão em tempo real, para que ela possa ser utilizada como uma alternativa aos marcadores convencionais, aproveitando as vantagens que a MAR oferece para proporcionar maior flexibilidade ao usuário da RA. Sob este aspecto, os usuários poderão inspecionar e manipular objetos de RA em relação a sua própria mão. Este é o objetivo principal deste trabalho, que será explicado a seguir.

1.2 Objetivo

A solução aqui proposta é a implementação de um sistema MAR, que consiste numa biblioteca de código capaz de estimar, em tempo real, a posição e a orientação da mão a partir do vídeo capturado. O propósito desta implementação é averiguar se a técnica proposta pode ser utilizada como mecanismo de interação em ambiente de RA.

Será implementada neste trabalho uma variação da técnica proposta por (10), selecionado devido as suas características de tempo real e o rastreamento da mão em uma sequência de imagem.

1.2.1 Objetivos específicos

Compõe os objetivos específicos:

- Estudar sobre a área de RA, principalmente RA sem marcadores.
- Estudar a cerca da área de processamento digital de imagem, especificamente filtros e segmentação.
- Estudar a área de visão computacional, especialmente reconhecimento e rastreamento de objetos.
- Segmentar, reconhecer e rastrear a mão, para ser utilizada na MAR.
- Estimar a pose câmera a partir da mão detectada.
- Realizar estudos relacionados ao desempenho e à viabilidade do sistema desenvolvido.

1.3 Metodologia

A implementação deste projeto dar-se-á utilizando o Ambiente de Desenvolvimento Integrado (IDE) *Microsoft Visual Studio* (MVS) juntamente com a Linguagem de Programação C++ associado a algumas bibliotecas que auxiliam no desenvolvimento de aplicações de RA.

Será utilizado o método de pesquisa aplicada, que visa gerar conhecimento para aplicações práticas, com a aplicação de leis, teorias e modelos para a solução de problemas, ou seja, uso de conhecimento pré-existente em novas relações e situações (11).

Nesse trabalho será necessário o conhecimento sobre PDI, CG, VC e RA para implementar uma solução que consiga integrar o uso da mão em um sistema MAR. Este software será desenvolvido em etapas, conforme descrito a seguir:

- Implementar o processo de segmentação da mão.
- Implementar o reconhecimento da mão.
- Implementar o rastreamento da mão, considerando ambientes controlados (sala com boa iluminação).
- Implementar o cálculo da posição tridimensional da mão em ambiente virtual.
- Aperfeiçoar o reconhecimento e o rastreamento para ambientes mais abrangentes (ao ar livre, por exemplo).

1.4 Estrutura do trabalho

Este documento está organizado conforme descrito a seguir. O Capítulo 2 apresentará o referencial teórico, onde estarão presentes os conceitos utilizados no decorrer deste trabalho. No Capítulo 3 será apresentado o funcionamento de um sistema MAR, enquanto o Capítulo 4 mostrará o método e a implementação deste trabalho. Os Capítulos 5 e 6, apresentarão os resultados obtidos e a conclusão do desenvolvimento deste trabalho, respectivamente.

2 REFERENCIAL TEÓRICO

Neste capítulo, introduz-se a Visualização de Informações (Seção 2.1), visto que o sistema desenvolvidos tem como objetivo, de modo geral, apresentar uma informação. Neste caso, a informação se trata de objetos virtuais sobrepostos em objetos reais. Dito isso, para que as informações possam ser apresentadas, são necessários alguns conhecimentos de áreas afins, tais como, PDI, VC e CG. Essas áreas estarão descritas nas Seções 2.2, 2.3 e 2.4, respectivamente. Em seguida, a (Seção 2.5), refere-se a RA, suas aplicações e ferramentas, para o desenvolvimento deste tipo de sistema. Finalizando, na (Seção 2.6) serão apresentados os trabalhos relacionados ao MAR, similares ao desenvolvido neste projeto.

2.1 Visualização de Informações

Área da ciência que se destina a construção de representações virtuais de dados abstratos, conhecida como Visualização de Informações (VI), é responsável por realizar a transformação de dados em imagens que possam ser visualizadas pelos seres humanos (12).

A visualização de uma informação, envolve o ato de ver, classificar, armazenar, consultar, filtrar e apresentar as informações relevantes ao usuário. Logo, o estudo sobre este assunto necessita de conhecimentos advindos de outros ramos da ciência. À vista disso, a VI tem grande relação com algumas subáreas da computação, como CG (Seção 2.4) e VC (Seção 2.3), por exemplo (12). No entanto, quando há a necessidade de manipular imagens, como no presente trabalho, torna-se essencial realizar o processamento desta imagem. Neste momento, faz-se indispensável o PDI, que será introduzido na seção seguinte.

2.2 Processamento Digital de Imagens

O PDI é constituído por um conjunto de tarefas, iniciando pela captura da imagem através de um processo de digitalização e pré-processamento, onde podem ser filtrados ruídos e corrigidas distorções provenientes dos sensores, por exemplo. Em seguida, tem-se o processo de análise e identificação de objetos, por meio do qual características são extraídas da imagem como bordas, texturas e movimento (13). A partir dessas características, é realizada a tarefa de classificação, objetivando reconhecer a identidade dos objetos, de responsabilidade da área de VC, como apresentado na seção seguinte.

2.3 Visão Computacional

Visão computacional é a área da ciência que estuda as teorias e métodos para a extração automática de informações contidas em uma imagem. Mais especificamente, é a construção

de descrições explícitas e claras dos objetos de uma imagem e o uso delas para diferentes propósitos (14).

Utilizando as técnicas de VC para monitoramento ou rastreamento de vídeo, é possível realizar a extração dessas informações. Estes métodos, usualmente, se baseiam em duas etapas, o reconhecimento e o rastreamento (15).

Na etapa de reconhecimento detecta-se um objeto, ou pontos de interesse de uma imagem ou *frame*. A detecção pode utilizar técnicas de PDI, apresentada posteriormente na Seção 3.4, como a detecção de bordas utilizada para interpretar as imagens da câmera (15).

Já na etapa de rastreamento, explicada com mais detalhes na Seção 3.5, que consiste no processo de reconhecer um padrão em uma sequência de imagens, busca-se em cada *frame*, a partir dos dados reconhecidos na primeira etapa, atrelar o conhecimento sobre o movimento do objeto que está sendo rastreado de modo a minimizar a busca entre as imagens em uma sequência. Caso não houvesse qualquer conhecimento específico, o processo de rastreamento seria relativamente lento (16).

Essas etapas de reconhecimento e rastreamento são utilizadas por sistemas de RA para encontrar a posição onde os objetos ou informações virtuais serão sobrepostas. Após encontrar esta posição, usa-se a CG, seção que será especificada a seguir, para mostrar a virtualização.

2.4 Computação Gráfica

CG é a área da Computação que estuda a geração, manipulação e análise de imagens, projetadas em um computador (17), a partir de um conjunto de métodos e técnicas que transformam dados em imagem com o auxílio de um dispositivo gráfico (18).

Atualmente, existem bibliotecas de CG amplamente utilizadas, como a OpenGL (Seção 4.2.2). Estas permitem a construção de objetos virtuais primitivos e complexos. Em sistema de RA, estas bibliotecas são utilizadas para compor o desenho e a renderização de objetos virtuais na cena. A apresentação do sistema de RA está sendo realizada na próxima seção.

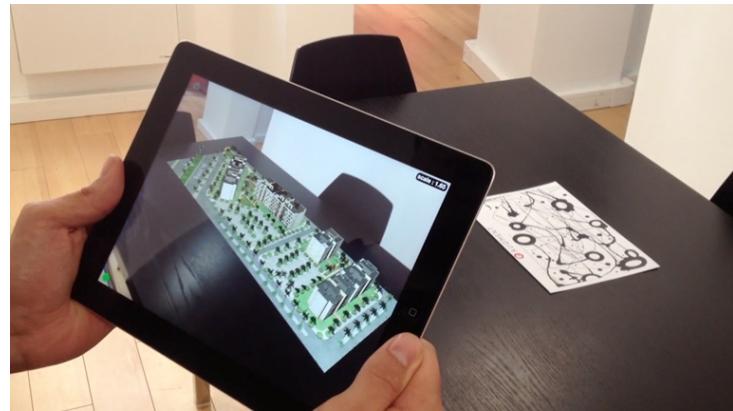
2.5 Realidade Aumentada

A RA é uma variação da Realidade Virtual (RV) que permite a projeção de dados e informações virtuais em objetos do mundo factual. Diferentemente da RV, que transporta o usuário a um ambiente integralmente sintético deixando-o completamente imerso sem visualização do mundo real ao seu redor, a RA apenas aprimora a realidade (15).

Na RA é permitido ao usuário ver o mundo real com objetos virtuais superpostos, de maneira a complementar a realidade e não substitui-la. Ou seja, é apresentado ao usuário um ambiente onde objetos reais e virtuais coexistem no mesmo espaço (19), como mostrado na Figura 3, em

que pode ser visualizada a projeção de uma planta na imagem capturada com a câmera do *tablet*.

Figura 3: Combinação do mundo real com o virtual.



Fonte: (20).

O principal objetivo da RA é adicionar informações e significados a um objeto real para aprimorar a compreensão em relação a determinado assunto. Para isso, ela combina várias tecnologias visando gerar informações digitais na percepção visual (15). De modo a esclarecer como a RA realiza tal procedimento, serão apresentadas algumas de suas aplicações na seguinte (Seção 2.5.1).

2.5.1 Aplicações

A RA teve uma notável evolução ao longo das últimas décadas, principalmente em relação ao seu aspecto tecnológico. Atualmente, existem diversas pesquisas que visam ampliar a utilização da RA, tendo em vista os dispositivos que suportam a execução estes sistemas estarem mais acessíveis, auxiliando para que a tecnologia torne-se mais robusta e preparada para a utilização em outras áreas do conhecimento (15).

No início, estava na área militar a maior representação da aplicação de RA. Os *HeadsUp Displays* (HUD) utilizados por pilotos de jatos militares, mostravam dados digitais relativos ao voo, como, por exemplo, a altitude do avião e sua velocidade (15), demonstrados na Figura 4.

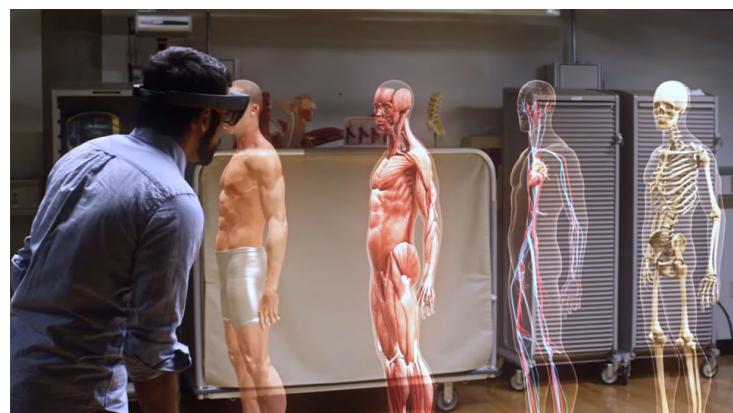
Figura 4: HeadsUp Displays.



Fonte: (21).

A RA tem sido essencial para muitos avanços na medicina, pois através dela, é possível projetar modelos do corpo humano, órgãos e sistemas para estudos mais precisos, como mostrado na Figura 5. Também pode ser utilizada durante os procedimentos cirúrgicos, auxiliando a equipe médica a proceder com mais segurança e êxito, diminuindo, portanto, os riscos de desenvolver complicações (22).

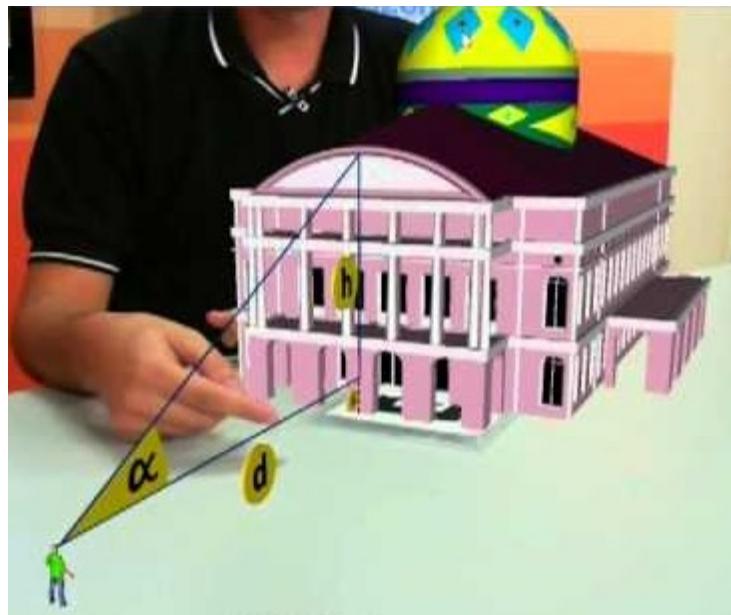
Figura 5: RA na medicina.



Fonte: (23).

Na área educacional a RA é aplicada para estimular e motivar estudantes, permitindo visualizações de objetos que estão distantes ou simulação de experimentos em sala de aula convencional, por exemplo. A RA pode ser aplicada de maneira dinâmica, estimulando a criatividade, demonstrando ser muito assertiva para o desenvolvimento de alunos e professores (22). Um exemplo pode ser visualizado na Figura 6, onde um professor utiliza a RA para ensinar trigonometria a seus alunos.

Figura 6: RA na educação.



Fonte: (24).

Como já explicitado, a RA tem aplicações em diversas áreas e concomitantemente a isso surge a necessidade de ferramentas que auxiliem desenvolvedores a implementar tais aplicações. Na (Seção 2.5.2) serão apresentadas algumas das principais ferramentas existentes para este tipo de aplicação.

2.5.2 *Ferramentas*

Dentre as ferramentas existentes a mais utilizadas para se implementar RA são as que se baseiam nas bibliotecas ARToolKit(19). Este Kit de Desenvolvimento de Software (SDK) permite aos programadores desenvolverem facilmente aplicativos de RA (25).

A ARToolKit é um projeto *Open Source*, multiplataforma, que pode ser utilizado nos sistemas operacionais Windows, Mac OS X, Linux, iOS e Android. As funcionalidades são as mesmas para as diferentes plataformas e o código pode ser facilmente transportado para novas e experimentais plataformas (25). As suas principais características, segundo (25), são:

- Rastreamento robusto.
- Suporte de calibração forte da câmera.
- Acompanhamento simultâneo e suporte à câmera estéreo.
- Suporte a múltiplos idiomas.
- Otimizado para dispositivos móveis.
- Suporte total de *Unity3D* e *OpenSceneGraph*.

Uma das maiores dificuldades no desenvolvimento de uma aplicação de RA é calcular

com precisão o ponto de vista do usuário em tempo real para que as imagens virtuais estejam exatamente alinhadas com os objetos do mundo concreto (26). O *ARToolKit* por meio de técnicas de visão computacional, calcula a posição real da câmera e sua orientação, permitindo ao programador sobrepor seus objetos virtuais. Atualmente ela suporta o marcador quadrado clássico, o código de barras 2D, e o multimarca. (25).

No entanto, apesar das suas características de robustez, ela é uma biblioteca baseada em marcadores, ou seja, é necessário haver um marcador no ambiente para que os objetos sejam posicionados. Para o sistema MAR, não foi encontrada uma biblioteca amplamente utilizada, mas alguns trabalhos similares foram descobertos, e serão descritos a seguir.

2.6 Trabalhos relacionados

No decorrer deste projeto foram encontrados alguns trabalhos semelhantes, sendo os principais o *Markerless Inspection of Augmented Reality Objects Using Fingertip Tracking* (10) onde foi desenvolvido a *HandyAR* (Seção 2.6.2) e o Realidade Aumentada sem Marcadores Baseado em Areias (4) (Seção 2.6.1);

2.6.1 Realidade Aumentada sem Marcadores Baseado em Areias

Este trabalho apresenta um estudo detalhado acerca das técnicas de MAR, focando especialmente nas que se baseiam em arestas. Este tipo de técnica é muito utilizado para o rastreamento de objetos poligonais, e portanto, naturalmente estáveis a mudanças de iluminação (4).

Também é descrita a implementação e avaliação de uma técnica baseada na amostragem de pontos em arestas em um modelo 3D previamente gerado, que se mostra capaz de recuperar as informações da câmera utilizadas para realizar a inserção dos objetos virtuais no mundo real (4).

A técnica implementada se baseou no algoritmo *Moving Edges* para encontrar as correspondências entre os pontos amostrados no objeto virtual e os pontos da imagem real. Foram utilizadas também a extração de arestas invisíveis e o minimizador não-linear *Levenberg-Marquardt* para tornar a técnica robusta a oclusões parciais e auto-oclusões (4).

A implementação feito no trabalho dessa seção, atingiu resultados bastante precisos tanto na utilização de dados reais quanto em cenas sintéticas, com um erro médio de 2 mm, para o caso de teste de maior número de arestas e 15 mm, para o caso de teste contendo menos arestas. Além disto, as aplicações elaboradas obtiveram taxas entre 15 e 30 fps denotando sua eficiência (4).

2.6.2 HandyAR

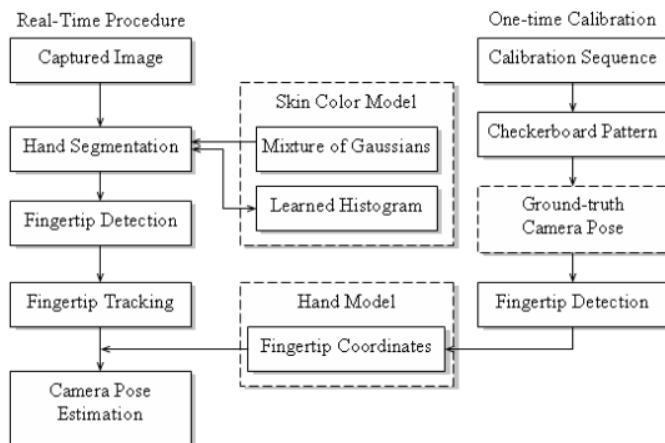
A HandyAR é uma solução baseada em rastreamento sem a utilização de marcadores, a qual apresenta uma interface de usuário que permite inspecionar facilmente objetos de RA,

sendo que ao invés de se utilizar um marcador, foi utilizada a mão humana como um padrão distintivo. Então, foi desenvolvido um algoritmo que define ser robusto em seu trabalho, em tempo real, que reconhece as pontas dos dedos para reconstruir a posição da câmera com Seis Graus de Liberdade (6DOF)¹ em relação à mão estendida do usuário (10).

O método utiliza a mão aberta, semelhante a forma que um marcador de RA estaria sendo manipulado, permitindo interfaces de usuário tangíveis. Com isso, elimina-se a necessidade de marcadores manuais, já que os usuários podem inspecionar e manipular objetos de RA em relação a sua própria mão (10).

Para este fim, foi apresentado um método para rastrear as pontas dos dedos e usá-la para a estimativa de *pose* da câmera. Na Figura 7, é apresentado o fluxo geral do sistema, conforme a seguinte descrição. Em um único passo de calibração off-line, é construído o modelo da mão de um usuário ao ser medida as posições relativas dos dedos estendidos entre si. Posteriormente, as regiões da mão na imagem capturada são segmentadas, e as pontas dos dedos são detectadas e rastreadas, tudo em tempo real. Ao reconhecer e rastrear uma mão estendida em qualquer posição e rotação que possa ser capturada por uma câmera usual, deriva-se uma estimativa de 6DOF para a câmera em relação à mão (10).

Figura 7: Fluxograma de calibração única e estimativa de *pose* de câmera em tempo real usando o rastreamento de ponta do dedo.



Fonte: (10)

Os testes e experimentos indicam que a segmentação da mão é sensível às mudanças na iluminação, mesmo que o modelo de cores aprendido de forma adaptável contribua bastante para a robustez. Nas cenas ao ar livre, a cor da mão pode mudar drasticamente em curtos períodos

¹ Seis Graus de Liberdade (6DoF) que se referem à liberdade de movimento de um corpo rígido em espaço tridimensional. (27).

de tempo e, até mesmo em ambientes controlados, devido ao sombreamento. Além disso, a cor da mão pode ficar saturada de branco, o que dificulta a diferenciação da mesma em relação aos demais objetos presentes na segmentação (10).

A região da mão é o ponto de partida para a detecção dos dedos, então a maior área com cor de pele é considerada como sendo a mão. Isso funciona efetivamente quando a mão for realmente a maior região dentre as segmentadas, entretanto, pode falhar quando uma outra região maior do que a mão entrar na cena, ou seja, a implementação não rastreia com sucesso uma mão quando a exibição mostrar duas mãos de área similar, sobrepostas entre si, por exemplo. Logo, o rastreamento de vários objetos com cor de pele pode ser realizado por um algoritmo mais sofisticado (10).

A *HandyAR* (trabalho apresentado nesta seção) introduz um método de estimativa de *pose* de câmera em tempo real, com 6DOF usando o rastreamento de ponta de dedo. A região da mão é segmentada e rastreada com base em distribuições de cores aprendidas de forma adaptável. As posições dos dedos estão localizadas no contorno da mão ajustando elipses em torno dos segmentos de maior curvatura. Os resultados mostram que a estimativa de *pose* da câmera pode ser efetivamente usada como alternativa ao rastreamento baseado em marcador para implementar uma UI tangível para aplicativos de AR (10).

Com base neste artigo, foi desenvolvido o presente trabalho, tendo a *HandyAR* como base para implementar o sistema MAR aplicado ao projeto proposto.

3 SISTEMAS DE REALIDADE AUMENTADA SEM MARCADORES

Sistemas MAR integram objetos virtuais em um ambiente real 3D em tempo real, melhorando a percepção e interação do usuário com relação ao mundo real. A sucinta diferença presente nos sistemas de RA baseados em marcadores está no método usado para posicionar objetos virtuais, pois neste caso, são utilizadas características naturais presentes na cena real que podem ser usadas como um marcador e rastreadas para posicionar objetos virtuais (5). Isto posto, o presente capítulo apresentará os elementos necessários para a construção do sistema de MAR que foi utilizado neste trabalho.

Quando fala-se de sistemas de RA baseados na captura de vídeo, o processo inicial, normalmente executado uma única vez, é a etapa de calibração da câmera mostrada na Seção 3.1, onde se obtém os parâmetros intrínsecos da mesma. Porém, para que seja feito o correto posicionamento dos objetos virtuais na cena, também é necessário conhecer os parâmetros extrínsecos, sendo que estes podem ser obtidos a partir de algoritmos para estimativa de *pose*, como mostrado na Seção 3.2. No entanto, estes algoritmos devem receber como entrada as coordenadas 3D de um objeto, que podem ser obtidas a partir do objeto real, e as coordenadas 2D do mesmo objeto em uma imagem ou *frame*, sendo que para obter tais coordenadas, deve-se identificar o objeto na imagem.

Para realizar o processo de identificação, inicia-se capturando as imagens a partir de um fluxo de vídeo. Em seguida, a imagem deve ser submetida ao processo de segmentação (Seção 3.3), que resulta em uma imagem binária em preto e branco, com as regiões de interesse que passarão pelo processo de reconhecimento (Seção 3.4) para que sejam fornecidas as coordenadas do objeto alvo do reconhecimento, caso ele esteja presente na imagem. Após ter sido devidamente reconhecido, este objeto estará apto a iniciar o processo de rastreamento (Seção 3.5), para que seja encontrada a posição do objeto identificado em uma sequência de imagens.

Com as coordenadas do objeto real na imagem, pode-se fazer a estimativa de *pose*. Sendo possível, a partir deste momento, renderizar os objetos desejados por intermédio de alguma biblioteca para CG.

Essa é uma escolha possível para a construção do sistema MAR e cada uma das suas etapas serão detalhadas nas próximas seções, sendo descritos também os componentes e itens de configuração para que este processo seja viabilizado. A seguir, inicialmente será apresentada a descrição dos parâmetros da câmera que são fundamentais para estimativa de *pose*.

3.1 Parâmetros da câmera

Câmeras são dispositivos dependentes de instrumentos ópticos que, por suas características de fabricação, podem distorcer a projeção. Na tentativa de corrigir estas distorções, algumas

características deverão ser conhecidas visando a redução destes erros (2). Essas características são conhecidas como parâmetros intrínsecos (Subseção 3.1.1). Existe também um outro grupo de parâmetros, chamados de extrínsecos (Subseção 3.1.2), que denotam a *pose* da câmera.

Visto que a inserção de objetos virtuais em uma cena advém da recuperação dessas informações que descrevem corretamente uma câmera e suas distorções, é necessário obter tais parâmetros. A seguir serão explicados cada um desses grupos de parâmetros e método utilizado para obtê-los.

3.1.1 Parâmetros intrínsecos

Como citado no início deste capítulo, parâmetros intrínsecos são aqueles inerentes à câmera e que não dependem de sua movimentação ao longo do tempo (4). Tais parâmetros dependem exclusivamente das características físicas, tais como: distância focal, ponto principal e distorção das lentes (28). A etapa de calibração pode auxiliar na inferência destes parâmetros.

Existem muitos processos de calibração de câmeras, alguns mais simples e outros mais complexos, dependendo dos requisitos de precisão que a aplicação necessita (2). O ARToolkit disponibiliza um processo de calibração que salva os parâmetros em um arquivo, assim como a OpenCV, que também disponibiliza um processo similar, como especificado em (29).

Ambos os processos encontram os parâmetros necessários para que se possa montar a matriz que define os parâmetros intrínsecos da câmera, também conhecida como matriz de calibração da câmera, que é dada por:

$$k = \begin{bmatrix} \alpha_x & s & u_0 \\ 0 & \alpha_y & v_0 \\ 0 & 0 & 1 \end{bmatrix} \quad (1)$$

onde, conforme (4):

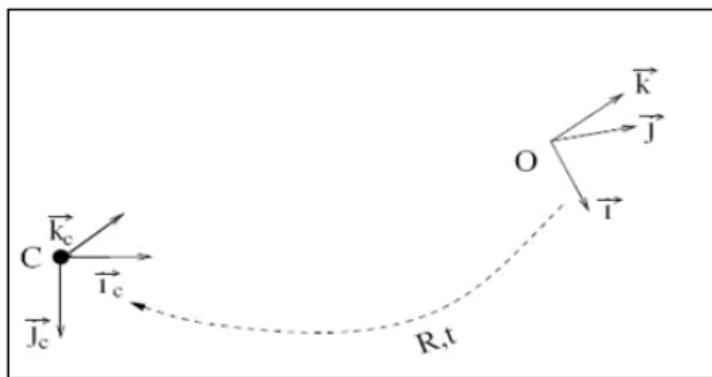
- α_x e α_y são proporcionais à distância focal com pesos k_x e k_y definidos pela quantidade de *pixels* por unidade de distância em cada direção. Assim, α_x e α_y são definidos por $k_x \cdot f$ e $k_y \cdot f$, respectivamente, onde f é a distância focal;
- (u_0, v_0) são as coordenadas correspondentes ao ponto principal da câmera;
- s é o coeficiente de cisalhamento da câmera definindo a relação entre as direções dos eixos do pixel da câmera, sendo nulo nos casos em que as direções são perpendiculares.

Como as características intrínsecas da câmera não tem relação com o seu movimento, diversas técnicas de RA se utilizam do prévio conhecimento acerca da câmera para determinar os parâmetros intrínsecos da mesma, de modo a simplificar ainda mais este processo ao considerar estes parâmetros da câmera como sendo fixos, além de conhecidos (4).

3.1.2 Parâmetros extrínsecos

Ao contrário dos intrínsecos, os parâmetros extrínsecos são aqueles que dependem do movimento da câmera, sendo a sua matriz dada em função de sua localização no espaço, correspondente à rotação e translação que deveriam ser feitas para transformar o sistema de coordenadas de mundo em sistema de coordenadas da câmera, como pode ser visto na Figura 8 (4).

Figura 8: Os parâmetros extrínsecos definem a rotação R e translação t , responsáveis por levar o sistema de coordenadas de mundo O para o sistema de coordenadas da câmera C.



Fonte: (4)

A rotação da câmera é dada por uma matriz 3×3 formada a partir de três vetores correspondentes às rotações dos eixos X , Y e Z , enquanto a translação é definida por uma matriz 3×1 em que cada valor representa o valor da translação ao longo de um desses eixos. Através desta composição obtemos os parâmetros extrínsecos da forma $[R_{3 \times 3}|t_{3 \times 1}]$ que corresponde a uma matriz 3×4 (4). Esses parâmetros podem ser obtidos no processo de *pose estimation*, conforme explicado na seção seguinte.

3.2 Pose Estimation

A *pose estimation* de um objeto em visão computacional é o processo que tem como objetivo obter a orientação da posição de um objeto em imagens digitais. Nesse contexto, calcula-se os três ângulos e as três distâncias no espaço 3D, relativos aos eixos X, Y e Z (28).

Em muitas aplicações de RA os parâmetros intrínsecos não mudam ao longo da sequência de *frames*, conforme mencionando na Seção 3.1.1, visto que a mesma configuração de câmera é usada durante a execução do sistema (5), assim sendo, esses parâmetros podem ser obtidos separadamente e utilizados durante toda execução do sistema. Neste contexto, o problema *Perspective-n-Point* (PnP) usa explicitamente os parâmetros intrínsecos, que precisam ser previamente obtidos, e estima apenas os parâmetros extrínsecos (5), conforme explicado a seguir.

3.2.1 Perspective-n-Point

Dados os parâmetros intrínsecos e algumas correspondências entre pontos 2D da imagem e pontos 3D do modelo, o método para estimativa de *pose*, denominado PnP, é capaz de estimar os parâmetros extrínsecos para um dado *frame*.

O PnP consiste basicamente no problema de estimar a *pose* da câmera $[R_{3x3}|t_{3x1}]$ dadas n correspondências 2D-3D. Dado n ($n \geq 3$) pontos de referência 3D na estrutura do objeto e suas correspondentes projeções 2D, ele determina a orientação e a posição de uma câmera em perspectiva (30).

Várias soluções foram propostas para este problema nas comunidades de VC e RA (5). Porém, conforme (30), não existe uma solução rápida, preferencialmente em tempo real, e globalmente ótima que seja precisa e aplicável a um problema PnP com qualquer número de ponto n ($n \geq 3$).

No caso do sistema MAR desenvolvido neste trabalho, onde tem-se uma quantidade relativamente pequena de pontos, foi utilizada uma implementação disponibilizada pela OpenCV que possui bom desempenho, para resolver este problema.

Como mencionado anteriormente, são necessários os pontos correspondentes na imagem para que se possa fazer a estimativa de *pose*. Esse pontos são obtidos através do reconhecimento e rastreamento do objeto que está sendo utilizado e o processo inicial para fazer essa detecção é a segmentação, como explicado a seguir.

3.3 Segmentação

Uma etapa importante dentro do fluxo de sistemas automáticos de reconhecimento é a de segmentação dos objetos (6). Este processo tem como finalidade partitionar uma imagem em regiões ou objetos distintos, com características ou atributos semelhantes, tais como cor ou proximidade (16).

Normalmente, é o primeiro passo a ser executando no processo de análise de imagens, sendo uma difícil tarefa no processamento de imagem. Esse passo no processamento determina o eventual sucesso ou falha de toda análise (31).

Especificamente, segmentar consiste no isolamento e seleção dos *pixels* de uma imagem que participam da composição de um objeto (32). De modo geral, muitos dos métodos de segmentação consistem em agrupar, independentemente da forma, os *pixels* com mesma propriedade, tais como, cor, intensidade, textura ou continuidade (32).

Esse processo é necessário devido a existência de objetos de interesse em uma imagem. Para isso, isola-se aqueles pixels que não fazem parte desses objetos, obtendo-se uma redução na informação contida na imagem, ou seja, extração de conhecimento (32).

Existe uma grande variedade de técnicas de segmentação na literatura de PDI e VC, e

quaisquer delas que resultem no suporte geométrico dos objetos de interesse em cada quadro do vídeo podem ser adotadas (6). No entanto, essas técnicas se fundamentam em descontinuidade e similaridade (13). Na primeira categoria, a abordagem é partitionar uma imagem baseando-se nas mudanças abruptas no nível de cinza, sendo bastante utilizada para a detecção de pontos isolados, linhas e bordas em uma imagem. Quanto a segunda categoria, encontra-se diversas abordagens, sendo as principais baseadas em limiares (*Thresholding*), crescimento de regiões (*Region Growing*) e aglomeração (*Clustering*) (31).

Como resultado deste processo de segmentação, tem-se um conjunto de objetos, regiões ou contornos extraídos da imagem, sendo que cada um dos *pixels* em uma mesma região é similar em algumas das propriedades mencionadas (33), ou seja, a tarefa de segmentação permite extrair, em cada quadro de um vídeo, *pixels* que possam estar relacionados aos objetos definidos previamente pela imagem de referência. A imagem resultante do processo de segmentação é utilizada como entrada para o processo de reconhecimento, como apresentado a seguir.

3.4 Reconhecimento

Reconhecer significa conhecer de novo, ou seja, é um processo onde existe algum conhecimento prévio (16). Para fazer o reconhecimento é necessário uma base de conhecimento dos objetos a serem reconhecidos, podendo esta base ser implementada diretamente no código, através de um sistema baseado em regras ou pode ser aprendida a partir de um conjunto de amostras dos objetos a serem reconhecidos utilizando técnicas de aprendizado de máquina (16).

O reconhecimento de objetos é uma das principais funções da área de VC (Seção 2.3) e está relacionado diretamente com o reconhecimento de padrões, pois um objeto pode ser definido por mais de um padrão (textura, forma, cor, dimensões, etc) e o reconhecimento individual de cada um destes padrões pode facilitar o reconhecimento do objeto como um todo (16).

Tendo esses padrões definidos e aplicando o reconhecimento, tem-se como resultado as informações geométricas dos objetos como Posição X e Y, caso o padrão tenha sido encontrado na imagem. Essa é uma etapa essencial para um sistema MAR, pois consiste em reconhecer algo que esteja no ambiente. As informações desse objeto que será detectado devem ser passadas ao reconhecedor e este, por sua vez, deve procurar por aquele padrão nesta imagem. Quando detectado com sucesso, o mesmo deve ser repassado para o rastreamento, como será explicado a seguir.

3.5 Rastreamento

O processo de rastreamento visa reconhecer um padrão em uma sequência de imagens, que poderia ser assim simplesmente feito, porém a busca em cada imagem de uma sequência, sem o

uso de qualquer conhecimento específico, é relativamente lenta. Os processos de rastreamento atrelam um conhecimento sobre o movimento do objeto que está sendo rastreado para minimizar a busca entre as imagens em uma sequência (16). Sendo este conhecimento obtido inicialmente na etapa de reconhecimento (Seção 3.4), e repassado para o rastreamento.

Os processos de rastreamento podem ser aplicados em diversas áreas, indo de sistemas de segurança até o uso em sistemas de interface humano-computador. Existem métodos para se prever a posição do objeto *frame a frame*, como os filtros *Kalman*, visto com mais detalhes na Seção 4.6.4 (16). O rastreamento, assim como os filtros para prever posição, são importantes para os sistemas MAR.

Considerando a natureza do rastreamento, as técnicas de MAR que utilizam esse conhecimento prévio pode ser classificada como rastreamento recursivo, onde além do conhecimento passado pelo detector, a *pose* anterior da câmera é utilizada como estimativa para calcular a *pose* atual da mesma.

Caso o processo de rastreamento no sistema MAR seja realizado com sucesso, tem-se a posição dos objetos que estão sendo rastreados na sequência de *frames*. Essa posição corresponde aos pontos 2D dos objetos necessários para que seja realizada a estimativa de *pose*. Logo, esta estimativa poderá ser realizada com sucesso e então o objeto virtual poderá ser posicionado corretamente usando alguma biblioteca de CG (Seção 2.4), como a OpenGL (Seção 4.2.2).

4 METODOLOGIA E IMPLEMENTAÇÃO

Com base nos estudos realizados em capítulos anteriores, foi possível estabelecer a arquitetura e o fluxo do sistema MAR que irá utilizar a mão como marcador. Nesta aplicação, será permitido ao usuário interagir com o mundo virtual, ao utilizar em sua mão uma luva de cor preta. Segundo este contexto, a próxima seção retratará da arquitetura, o fluxo de execução, os algoritmos utilizados e a implementação do sistema utilizados neste trabalho.

Primordialmente é necessário esclarecer que quando inicia-se o desenvolvimento de sistemas, uma das primeiras etapas que deve ser realizada é a escolha da linguagem de programação e da IDE que serão utilizadas. Portanto, na Seção 4.1 será apresentada tanto a linguagem quanto a IDE para a realização deste trabalho.

4.1 Plataforma de Desenvolvimento

A linguagem de programação adotada para a construção deste projeto é a linguagem C++ associada a IDE MVS. Uma linguagem de programação é, basicamente, uma forma padronizada de transferir instruções para um computador, enquanto um IDE é um programa que reúne características e ferramentas de apoio ao desenvolvimento de software, com o objetivo de agilizar este processo (34). Dito isso, nas Seções 4.1.1 e 4.1.2 serão realizadas sucintas explicações acerca da Linguagem C++ e da IDE MVS, respectivamente.

4.1.1 *Linguagem C++*

A Linguagem C++ é considerada uma linguagem de programação de nível médio, baseada na sua antecessora C. Para a criação desta, foram acrescentados elementos de outras linguagens, afim de se criar uma nova, mais poderosa e flexível(35), resultando em uma linguagem que possui uma enorme variedade de bibliotecas, tendo sido incorporadas as preexistentes do C e algumas inéditas. Por fim, incorporou-se a eficiência do C há novas funcionalidades, permitindo a programação em alto e baixo nível (35).

Esta linguagem, com seu suporte a múltiplos paradigmas, atende as necessidades de vários grupos de programadores, pois permite executar funções tanto em baixo quanto em alto nível. Além disso, o C++ também disponibiliza herança múltipla, classes abstratas, métodos estáticos, métodos constantes e membros protegidos, incrementando suporte para orientação à objeto (35).

Neste contexto, esta foi a linguagem escolhida para o desenvolvimento deste trabalho, tendo em vista permitir a programação em baixo nível, e a utilização de funções existentes em alto nível de forma eficiente, com maior desempenho quando comparados com outras linguagens, e também sua programação orientada à objetos. Após a escolha da linguagem, seleciona-se a IDE

que será utilizada, estando o MVS descrito a seguir.

4.1.2 Microsoft Visual Studio

O MVS é uma IDE que permite o desenvolvimento em diversas linguagens diferentes, tais como, C#, VB, C e C++. Juntamente com a IDE também são disponibilizadas algumas APIs, tais como o *DirectX* e o *.NET Framework*. Tanto a IDE, quanto as suas APIs, são exclusivas para o sistema operacional *Microsoft Windows* (26).

Além do suporte completo para a Linguagem de Programação C++, o ambiente MVS também fornece uma plataforma completa para desenvolvimento de vários tipos de aplicação. Essa IDE foi escolhida por disponibilizar um dos melhores e mais completos editores de códigos fonte em C++ e por fornecer maior facilidade para a configuração de bibliotecas de apoio, tais como OpenCV e OpenGL que são introduzidas a seguir.

4.2 Bibliotecas de Apoio

Neste trabalho, foram utilizadas as bibliotecas OpenCV e OpenGL para auxiliar no desenvolvimento do sistema MAR. Essas tecnologias foram de fundamental importância para a construção do sistema, pois elas permitem que os programadores, através de uma interface externa, realize a entrada de informações relacionados ao experimento, transformando-as no resultado esperado para a RA (26). Nas próximas seções serão apresentadas cada uma dessas bibliotecas.

4.2.1 OpenCV

A biblioteca OpenCV, desenvolvida pela Intel, possui mais de 500 funções (16), sendo desenvolvida com o objetivo de tornar a VC mais acessível aos usuários e programadores da área (16). A biblioteca possui código fonte aberto e também disponibiliza os seus executáveis (binários) otimizados para os processadores Intel (16). Ao executar um programa que utilize a OpenCV, é invocado automaticamente uma *Dynamic Linked Library* (DLL) que detecta o tipo de processador e carrega a DLL otimizada para ele (16). Junto com o pacote OpenCV, também é fornecida a biblioteca *Image Processing Library* (IPL), da qual a OpenCV depende parcialmente, além de documentação e um conjunto de códigos exemplos (16).

A biblioteca está dividida em cinco módulos (16): processamento de imagens, análise estrutural, análise de movimento e rastreamento de objetos, reconhecimento de padrões e calibração de câmera e reconstrução 3D. Em seguida, será exibida a OpenGL, que, juntamente com a OpenCV fornece os mecanismos necessários para o desenvolvimento do sistema MAR.

4.2.2 OpenGL

Atualmente, a OpenGL é reconhecida e aceita como sendo a *Application Programming Interface* (API) padrão para desenvolvimento de aplicações gráficas 3D em tempo real (26). Normalmente, quando menciona-se que um programa utiliza a OpenGL, significa que ele está implementado em alguma linguagem de programação e que esta faz chamadas a uma ou mais bibliotecas OpenGL (26).

Por ser portável para diversas plataformas, a OpenGL não possui funções para gerenciamento de janelas, interação com o usuário ou arquivos de entrada/saída e também não existe um formato de arquivo OpenGL para modelos ou ambientes virtuais (26). Neste caso, utiliza-se bibliotecas, para que sejam fornecidas tais funcionalidades, que sejam compatíveis com a OpenGL, como a GLUT (*OpenGL Utility Library*), disponível para a plataforma *Microsoft Windows*.

A biblioteca GLUT fornece várias funções para modelagem, tais como superfícies quádricas e curvas e superfícies *Non Uniform Rational BSplines* (NURBS) (26). Essa biblioteca trabalha em conjunto com a biblioteca *Graphic Library User Interface* (GLUI) implementada na Linguagem C++, sendo provedora da criação de interfaces e controles tais como, botões, caixa de textos e funções específicas. Elas operam em conjunto para o desenvolvimento de aplicações em OpenGL (26).

Após introduzidas as bibliotecas de apoio, a linguagem de programação e a IDE, inicia-se o processo de definição do projeto e da arquitetura, como apresentado na próxima seção.

4.3 Arquitetura do Software

O sistema foi dividido em cinco módulos, sendo cada um deles responsáveis por tarefas específicas. Essa divisão foi escolhida por facilitar o entendimento e desenvolvimento do projeto. Outra característica importante dessa organização em módulos é a abstração, que permitir alterar um módulos sem interferência nos demais.

Estes módulos são representados por uma classe ou conjunto de classes utilizando o paradigma de orientação à objetos. Os principais módulos utilizados são:

- ***Capture***: módulo responsável por prover uma sequência de imagens (ver Apêndice A.1);
- ***HandRegion***: módulo responsável por fazer o pré-processamento e a segmentação das imagens (ver Apêndice A.2);
- ***FingertipProcess***: módulo responsável por encontrar um conjunto de dedos candidatos (ver Apêndice A.3);
- ***FingertipTracker***: módulo responsável por fazer o reconhecimento e o rastreamento dos dedos (ver Apêndice A.4);

- **PoseEstimation:** módulo responsável por trabalhar com os parâmetros da câmera e estimativa de pose (ver Apêndice A.5).

Todos os módulos acima mencionados, foram utilizados na classe *FingertipPoseEstimation* (ver Apêndice A.6) que coordena todo o fluxo do sistema, ou seja, qualquer interação com o sistema deve ser realizada utilizando apenas essa classe, pois a mesma já consome as funcionalidades de todos os outros módulos. A seguir será apresentado o fluxo da aplicação, conforme mencionado anteriormente.

4.4 Fluxo da aplicação

O sistema consome os recursos dos módulos apresentados para integrar o sistema MAR final. Este sistema apresenta o seguinte fluxo de execução, conforme o Algorítimo 1:

Algoritmo 1: Fluxo de execução do sistema MAR desenvolvido

```

begin
    Carrega as coordenadas de um modelo de mão 3D (Seção 4.5.1);
    Carrega os parâmetros intrínsecos da câmera (Seção 4.5.2);
repeat
    Captura o frame (Seção 4.6);
    Faz a segmentação do frame capturado (Seção 4.6.1);
    Busca os pontos candidatos a ponta de dedos;
    if Nenhuma mão foi detectada anteriormente then
        Verifica se os pontos candidatos correspondem as pontas de dedos de uma
        mão (Seção 4.6.2);
    else
        Realiza o rastreamento a partir das pontas dos dedos anteriores e dos novos
        candidatos (Seção 4.6.3);
    end
    Calcula os parâmetros extrínsecos - rotação e translação (Seção 4.6.4);
    Configura o Frustum do OpenGL baseado nos parâmetros intrínsecos da câmera
    (Seção 4.6.5);
    Converte o frame para uma textura do OpenGL (Seção 4.6.5);
    Configura o gluLookAt do OpenGL com os parâmetros extrínsecos (Seção 4.6.5);
    Desenha um objeto virtual (Seção 4.6.5);
    Exibe uma janela OpenGL (Seção 4.6.5);
until usuário finaliza a aplicação;
end

```

As próximas seções explicarão como cada uma das etapas do Algoritmo 1 e dos módulos foram implementadas. A Seção 4.5 começará explicando os métodos de inicialização.

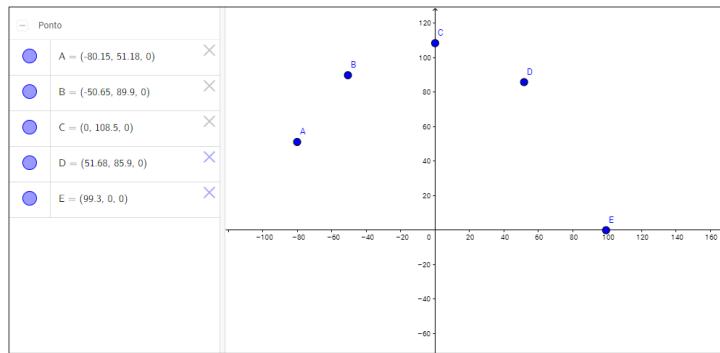
4.5 Inicialização

O processo de inicialização é executado uma única vez, quando o sistema for iniciado, e visa carregar informações que serão utilizadas no decorrer da execução. Os dois métodos inicializados neste sistema são o carregamento das coordenadas de um modelo de mão 3D e o carregamento dos parâmetros intrínsecos da câmera.

4.5.1 Carregamento do modelo de mão 3D

O primeiro processo que deve ser executado no sistema é a chamada do *FingertipPoseEstimation.loadFingertipCoordinates* que recebe o *path* do arquivo que contém as coordenadas de um modelo 3D da mão (ver Figura 9). Essas coordenadas são carregadas do arquivo todo vez que o sistema é inicializado e são utilizadas na *pose estimation*.

Figura 9: Coordenadas aproximadas de um modelo de mão 3D.



Fonte: Elaborado pelo autor.

A Figura 9 ilustra uma representação do modelo de mão carregado pelo sistema, onde o ponto E representa a posição do polegar e os pontos D, C, B e A representam o indicador, médio, anelar e mínimo, respectivamente.

4.5.2 Carregamento dos parâmetros intrínsecos da câmera

No segundo momento, utiliza-se o método *FingertipPoseEstimation.initialize* onde devem ser passados o *path* do arquivo de calibração da câmera (Descrito na Seção 3.1.1) e o tamanho das imagens que serão utilizadas em todo os módulos de processamento. Para este sistema MAR foram utilizados parâmetros intrínsecos fixos, como pode ser visto na Anexo A.1, pois funcionam bem na grande maioria das câmeras. Caso seja desejável ter ainda mais precisão, o usuário poderá calibrar a sua câmera e alterar os parâmetros no arquivo. Um código disponibilizado pela

OpenCV poderá ser utilizado para realizar a calibração da câmera, podendo este ser encontrado em (29). Após realizar a etapa de inicialização, o sistema poderá iniciar as suas etapas de processamento, demostrada na seção seguinte.

4.6 Etapas do Processamento

Para iniciar o processamento, a primeira etapa a ser realizada é a captura do *frame*, obtido a partir do módulo *Capture* que utiliza uma implementação disponibilizada pela OpenCV, a partir da classe *VideoCapture* (36) que lida com fluxo de imagens, seja de um dispositivo de captura ou de um arquivo de vídeo.

Após obter o *frame* atual, este é encaminhado para a módulo *FingertipPoseEstimation* através do método *OnCapture*. Esse método apenas recebe o *frame* e o armazena pra a próxima requisição de processamento. Em seguida, a próxima operação a ser executada é a solicitação para que o *FingertipPoseEstimation* processe o *frame* armazenado, por meio da chamada ao método *OnProcess*.

Um dos principais métodos do sistema é o *OnProcess*, pois ele solicita as tarefas a serem executados pelos outros módulos, sendo a segmentação do *frame* a primeira etapa realizada neste método (ver Seção 3.3). Isso ocorre a partir de uma chamada ao método *getHandRegion* do módulo *HandRegion*, conforme descrito na próxima seção.

4.6.1 Segmentação

Para obter melhor desempenho e maior precisão entre variações de ambientes, a segmentação foi realizada baseando-se em apenas uma cor, sendo o preto esolhido como padrão para este trabalho, podendo ser alterada nos parâmetros da aplicação. Neste contexto, o usuário deverá estar utilizando uma luva da cor preta para que o sistema reconheça sua mão corretamente.

Então, a OpenCV faz a captura dos *frames* no sistema GBR (RGB com componentes invertidos) (37). No entanto, a imagem GBR apresenta desvantagens na realização do processamento, pois em um espaço de cores aditivo, as componentes como luminância e crominância estão distribuídas entre os valores de vermelho, verde e azul, estabelecendo uma interdependência entre as cores básicas para quaisquer variações (37). Para corrigir estes problemas, o sistema de cores foi convertido para o *Hue*, *Saturation* e *Value* (HSV).

A segmentação de objetos pela cor demanda robustez, em relação às variações de iluminação, presença de ruídos, dentre outros, o que pode ser alcançado utilizando-se este formato (37). O HSV separa a matiz (*Hue*) que estabelece o tipo de cor, a saturação (*Saturation*) que está relacionada a pureza da cor e o valor (*Value*) que indica o grau de luminância (37).

Este sistema de cores apresenta vantagens para o PDI, tendo em vista que todas as cores são representadas em uma única grandeza (matiz) (37). Neste caso, a primeira operação executada

no método *getHandRegion* é conversão da imagem para HSV, que é feita utilizando a função *cvtColor* disponibilizada pela OpenCV. Após a conversão, é inicialmente feito um filtro em seus canais, como se segue.

O processo inicial verifica se em cada canal de cada pixel o valor daquele canal se encontra entre o limite inferior e o superior informado, conforme Algoritmo 2. Segundo (37), os valores para realizar o filtro por cores próximas à cor preta são: para o primeiro componente - cor azul - valores de 0 a 180, no segundo - cor verde - valores de 0 a 255 e no último componente - cor vermelha - valores de 0 a 30.

Algoritmo 2: Filtro por intervalo de valores nos canais (1)

Input: *SRC*: imagem original;

LI_I: limite inferior no canal *I*;

LS_I: limite superior no canal *I*;

Output: *DST*: imagem de saída

begin

for cada posição de pixel *I* em *SRC* **do**

//Exemplo para uma matriz com dois canais;

DST(*I*) = *LI₀* ≤ *SRC*(*I*)₀ ≤ *LS₀* ∧ *LI₁* ≤ *SRC*(*I*)₁ ≤ *LS₁*;

end

end

Essa operação foi realizada utilizando a função *inRange* disponibilizada pela OpenCV. Após esse processo é realizada a suavização do *frame* resultante utilizando a função *blur* também concedida pela OpenCV, que tem como objetivo remover possíveis ruídos na imagem.

A suavização ocorrerá na imagem por meio do filtro de caixa normalizado (borrão homogêneo) (38), que ocorre ao se deslizar uma janela (*kernel*) em toda a imagem e calcular cada *pixel* com base no valor do *kernel* e no valor dos *pixels* sobrepostos da imagem original. O *kernel* utilizando no filtro é constituído como se segue:

$$K = \frac{1}{WK \cdot HK} \begin{pmatrix} 1 & 1 & 1 & \dots & 1 & 1 \\ 1 & 1 & 1 & \dots & 1 & 1 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 1 & 1 & 1 & \dots & 1 & 1 \end{pmatrix} \quad (2)$$

onde *K* é o *kernel* e *WK* e *HK* são as quantidades de colunas e linhas do matriz do *kernel*, respectivamente. Na implementação foi utilizado um *kernel* 5x5, pois conforme (38) este demonstrou melhor funcionamento para a suavização e o filtro de cor. Após a suavização, é feita a segmentação utilizando *thresholding*.

Os métodos de *thresholding* são os mais simples para a segmentação de imagens, pois

consistem em dividir os *pixel* da imagem em relação ao seu nível de intensidade (33). Normalmente, utiliza-se um valor fixo de limiar aplicado em toda a imagem. Considerando um limiar L , uma imagem de entrada (*SRC*), a imagem de saída (*DST*) pode ser obtida aplicando a seguinte fórmula para cada *pixel*:

$$DST(x,y) = \begin{cases} MAXVAL, & \text{se } SRC(x,y) > L \\ 0, & \text{caso contrário} \end{cases} \quad (3)$$

Os valores dos limiares ideais para atingir o objetivo da segmentação podem ser calculados com a ajuda dos picos dos histogramas da imagem (39). Para este trabalho foi utilizada a função *threshold* disponibilizada pela OpenCV, com o limiar L de 128 e o *MAXVAL* igual a 255, passando o flag *CV_THRESH_BINARY*. Ou seja, todos os *pixels* da imagem de entrada que possuirem valor maior do que 128, passarão a ter a cor branca (255), enquanto os demais receberão o valor 0. Isso resulta em uma imagem binária, onde a regiões que continham cores próximas de preto são todas brancas e o restante da imagem será preta.

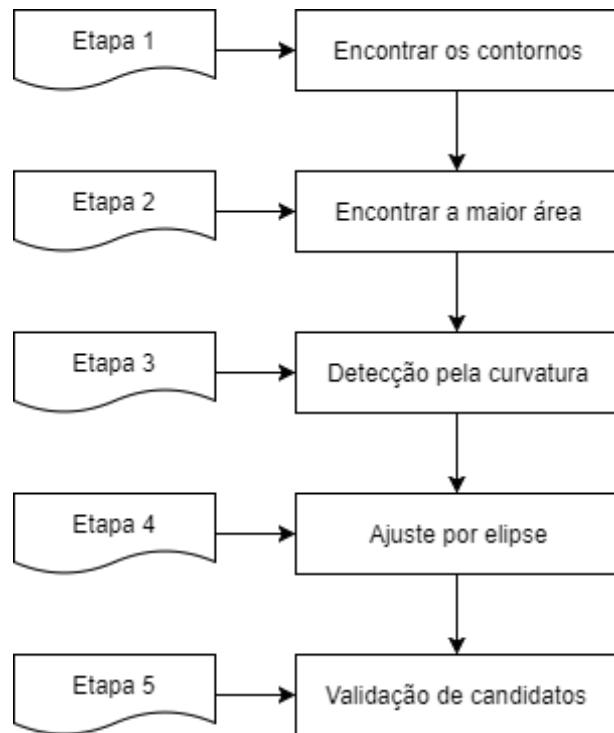
Neste momento, tem-se a imagem segmentada, porém ela pode conter ruídos oriundos de objetos com cores similares às presentes no alvo do rastreamento (37). Torna-se necessário, portanto, aplicar filtros que possam garantir a seleção do objeto desejado.

O primeiro filtro que pode ser aplicado é o morfológico, que realiza as operações de dilatação e erosão (37). A operação de erosão elimina pequenas regiões brancas na imagem, consideradas como ruídos e a dilatação elimina as pequenas regiões pretas dentro das regiões brancas, reduzindo o ruído interno dos objetos identificados (37). Essas duas operações morfológicas foram utilizadas neste trabalho, por intermédio das bibliotecas disponibilizadas pela OpenCV.

4.6.2 Detecção dos dedos

Finalizado o processo de segmentação, obtém-se uma imagem com uma região ou uma série de regiões candidatas, iniciando-se, então, o processo de detecção das pontas dos dedos. Tal processo executa um conjunto de ações, conforme a sequência apresentada na Figura 10.

Figura 10: Etapas da detecção

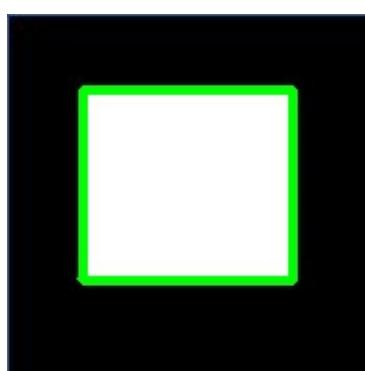


Fonte: Elaborado pelo autor.

4.6.2.1 Etapa 1 - Encontrar os contornos

Para cada região calcula-se o seu contorno a partir da função *findContours* fornecida pela OpenCV. Esta localiza contornos em uma imagem binária ao utilizar o algoritmo de *Suzuki* (40), retornando uma lista de contornos, em que cada contorno é uma lista de pontos. Um exemplo da execução deste algoritmo está demonstrado na Figura 11, onde a parte branca representa uma região qualquer e a parte verde representa os pontos retornados pela função.

Figura 11: Exemplo da utilização da função *findContornos*.



Fonte: (41)

4.6.2.2 Etapa 2 - Encontrar a maior área

Assumindo que a região da mão é maior região encontrada na segmentação, deve ser possível localizar nos contornos obtidos, a região com maior área. Uma maneira para obter tal informação é utilizar os *centroids*¹ dessas regiões, que pode ter seu valor aproximado calculado pela transformada da distância.

Essa transformada calcula a distância aproximada ou precisa de cada *pixel* da imagem binária até o seu *pixel* zero mais próximo, tendo como saída uma matriz com a mesma dimensão da imagem de entrada, contendo essas distâncias (42). Na Figura 12 tem-se uma demonstração da aplicação dessa transformada em uma imagem, onde, à esquerda está a imagem original e à direita a matriz de saída. A função utilizada para fazer essas transformações foi a *distanceTransform* disponibilizada pela OpenCV.

Figura 12: Aplicação da transformada da distância em uma imagem .

0	0	0	0	0	0	0	0
0	1	1	1	1	1	1	0
0	1	1	1	1	1	1	0
0	1	1	1	1	1	1	0
0	1	1	1	1	1	1	0
0	1	1	1	1	1	1	0
0	0	0	0	0	0	0	0

0	0	0	0	0	0	0	0
0	1	1	1	1	1	1	0
0	1	2	2	2	2	1	0
0	1	2	3	3	2	1	0
0	1	2	2	2	2	1	0
0	1	1	1	1	1	1	0
0	0	0	0	0	0	0	0

Fonte: (43).

Após essa matriz ser preenchida, busca-se a posição (linha e coluna) onde o valor da distância é máximo, pois está posição corresponde ao *centroid* da maior região da imagem. Considerando a imagem da direita na Figura 12, a posição que possui a distância máxima é a (4, 4), sendo que para ser realizada esta operação, foi utilizada a função *minMaxLoc* da OpenCV.

De posse do *centroid* da maior região foi realizado o preenchimento da área contornada com uma cor, utilizando a função *drawContours* da OpenCV, e verificando se a posição do *centroid* também foi pintada com a mesma cor utilizada no preenchimento do contorno, o que indicaria que esta posição está contida neste contorno e que este possui a maior área. Um exemplo de preenchimento de contorno pode ser visto na Figura 13, onde os *pixels* em vermelho representam o contorno e os verdes representam o preenchimento do contorno, sendo necessário apenas verificar se a posição do *centroid* possui a cor verde.

¹ Ponto ou coordenada de uma forma geométrica que estabelece o seu centro geométrico ou baricentro

Figura 13: Aplicação da transformada da distância em uma imagem.

0	0	0	0	0	0	0	0
0	1	1	1	1	1	1	0
0	1	1	1	1	1	1	0
0	1	1	1	1	1	1	0
0	1	1	1	1	1	1	0
0	1	1	1	1	1	1	0
0	1	1	1	1	1	1	0
0	0	0	0	0	0	0	0

Fonte: (43).

4.6.2.3 Etapa 3 - Detecção pela curvatura

Após encontrada a região com maior área, inicia-se o processo de detecção dos pontos candidatos a ser extremidades dos dedos. Estes são detectados a partir dos pontos do contorno, utilizando um algorítimo que calcula a curvatura da cabeça da falange distal. Este algoritmo, baseado em (10), computa a curvatura dos *pixels* do contorno afim de detectar os pontos candidatos a ser extremidades dos dedos, pois essas são curvadas.

A curvatura de um ponto no contorno é medida utilizando múltiplos valores de deslocamento², sendo considerado o maior ângulo computado, conforme a seguinte descrição: dado um valor mínimo de deslocamento (*L_MIN*) e um valor máximo (*L_MAX*), é computado o ângulo entre $\overrightarrow{P_i P_{i-l}}$ e $\overrightarrow{P_i P_{i+l}}$, onde *l* tem seu valor variando de *L_MIN* até *L_MAX*. Este ângulo pode ser calculado utilizando o *dot product* da seguinte maneira (10):

$$\theta = \arccos\left(\frac{\overrightarrow{P_i P_{i-l}} \cdot \overrightarrow{P_i P_{i+l}}}{\| \overrightarrow{P_i P_{i-l}} \| \| \overrightarrow{P_i P_{i+l}} \|}\right) \quad (4)$$

onde θ é o ângulo, P_i é o *i*-ésimo ponto no contorno, P_{i-l} e P_{i+l} é o antecessor e o sucessor, respectivamente, com valor de deslocamento *l* e $\| \overrightarrow{P_i P_{i-l}} \|$ e $\| \overrightarrow{P_i P_{i+l}} \|$ representam as magnitudes dos vetores $\overrightarrow{P_i P_{i-l}}$ e $\overrightarrow{P_i P_{i+l}}$, respectivamente.

Para essa aplicação foram utilizados valores de deslocamento de 5 a 25, pois segundo (10) um intervalo para *l* que inclui todos estes números funciona bem. Além do alto valor de curvatura, a direção da curva também deve ser considerada para determinar se o ponto representa a cabeça da falange distal ou um vale entre dois dedos. As direções são indicadas pelo sinal do *cross*

² Deslocamento se refere a distância (em quantidade de pontos) entre pontos computados. Considerando um deslocamento *l*, os pontos a serem utilizados seriam P_{i-l} , P_i e P_{i+l} .

product dos dois vetores, que pode ser calculada da seguinte maneira:

$$\text{cross} = \| P_i P_{i-l} \| \cdot \| P_i P_{i+l} \| \cdot \sin(\theta) \quad (5)$$

onde *cross* representa o valor do *cross product* e $\| P_i P_{i-l} \|$, $\| P_i P_{i+l} \|$ e θ são os mesmos valores da Fórmula 4.

Após computar o ângulo máximo para o ponto i , verifica-se se este ângulo é menor do que o ângulo limite pré-estabelecido. Este limite varia de acordo com o objetivo da aplicação, sendo que neste trabalho foi utilizado ângulos cujo cosseno seja maior do que 0.5, ou seja, ângulos que variam de -60 a +60, ou ainda, ângulos com seu valor absoluto menor que 60 graus, conforme (10).

Se este ângulo além de atender a restrição anterior for também o primeiro a atendê-la, então ele é marcado como sendo um possível início de curvatura. A cada novo ponto consecutivo que atender a restrição do ângulo, será incrementado o contador de pontos conectados, pois para ser considerado uma curvatura, deve-se ter uma quantidade mínima de pontos conectados que possua o ângulo dentro do limite estabelecido.

Para este trabalho foram utilizados 10 pontos, conforme o utilizado por (10). Ou seja, são necessários ao menos 10 pontos consecutivos para a curvatura ser considerada, caso contrário, a curvatura é muito sutil para ser considerada como uma extremidade de dedo, sendo, então, ignorada.

Também deve ser definida a quantidade mínima de pontos que não atendem a restrição do ângulo, limitando o último ponto que a atenda (esses pontos são chamados de pontos de *gap*). Esse parâmetro é importante para garantir que a curva computada tenha realmente terminado naquele ponto e para computar a direção da elipse que será criada visando aumentar a precisão dos pontos representantes das cabeças das falanges distais.

4.6.2.4 Etapa 4 - Ajuste por elipse

Para que seja calculado um ponto preciso da extremidade do dedo, é utilizada uma elipse. Isso é feito ajustando a elipse ao redor da ponta do dedo, usando o ajuste de mínimos quadrados, da função *fitEllipse* disponibilizada pela OpenCV.

Essa função calcula a elipse que melhor se encaixa (em um sentido de mínimos quadrados) no conjunto de pontos 2D, sendo retornado o retângulo rotacionado no qual a elipse está inscrita. Neste contexto, são passados todos os pontos do contorno que estão entre o ponto marcado como início da curvatura até o último ponto de *gap* computado.

Em seguida, calcula-se os pontos de interseção do eixo principal da elipse com sua borda, pois este será a possível ponta do dedo. Segundo (10), o ajuste da elipse aumenta a precisão da

estimativa da *pose* da câmera em comparação com o ponto de maior curvatura.

Esse processo de encontrar as cinco extremidades dos dedos pela curvatura é executado em todas as interações do sistema. No entanto, o algoritmo de detecção das cabeças das falanges distais pode produzir falsos positivos, sendo necessário amenizar este problema. Por isso, estes pontos não são repassados diretamente para a função de rastreamento, evitando-se assim, que, caso os mesmos não estejam corretos o processo de rastreamento falhe.

4.6.2.5 Etapa 5 - Validação de candidatos

Enquanto o conjunto de pontos não for identificado como extremidade dos dedos válidos, estes serão repassados para a função *FingertipTracker.feedFingertipCandidates*. Essa função armazena os pontos encontrados na etapa anterior no decorrer das interações, sendo que de uma iteração para a outra é calculado a correspondência por vizinho mais próximo para saber se estes são os mesmo pontos que já foram detectados anteriormente ou se são novos pontos.

Este algoritmo de correspondência foi implementado no método *FingertipTracker.matchCorrespondencesByNearestNeighbor*, e funciona da seguinte maneira: para cada extremidade de dedo candidata, procura-se dentre os novos pontos aquele mais próximo deste candidato, considerando o limite máximo de distância entre os dois pontos, que no caso dessa aplicação foi definido como sendo 50 pixels. Se essa correspondência for encontrada, então o candidato é atualizado e tem o seu *score*³ aumentado em uma unidade. Os novos pontos para o qual não foi encontrando nenhum candidato correspondente são adicionados na lista de candidatos com o seu *score* zerado, desde que a lista de candidatos não tenha atingido o seu limite máximo, que para este trabalho, foi definida como 20 candidatos, pois, nesse caso, o ponto será descartado.

Após a busca por pontos correspondentes, executa-se o processo de descarte dos candidatos, baseado no seguinte critério: um candidato que não teve nenhum corresponde por uma quantidade de *frames* seguidas (neste trabalho a quantidade é igual a 5 *frames*) ou caso o candidato já tenha armazenado pelo menos a quantidade mínima de *frames* necessária para ser rastreado (para este trabalho esta quantidade chamada de *age* também é igual a 5) e a proporção entre o seu *score* e sua *age* for menor do que 0.5, pois neste caso, apesar do candidato ter *age* suficiente para ser rastreado, ele não teve nenhum correspondente encontrado nos últimos *frames*, isso normalmente significa que a posição da mão foi alterada.

Após este processo, verifica-se se restam ao menos 5 pontos que possuam sua coordenada y maior do que a coordenada do *centroid* da mão⁴ e que possua o *age* mínimo para ser

³ O score define a quantidade de vezes que o mesmo candidato já foi detectado.

⁴ É considerado que os dedos estejam acima da *centroid* e qualquer candidato que não atenda essa restrição não será considerado.

rastreado. Caso afirmativo, esses 5 pontos candidatos são considerados válidos e podem ser repassados diretamente ao método de rastreamento, sendo esta função chamada novamente apenas se o processo de rastreamento, que será explicado na próxima seção, faltar.

Por fim, as pontas dos dedos são ordenadas com base no índice do polegar, que pode ser determinado como a extremidade do dedo mais distante da posição média de todas as outras extremidades. A ordem das cabeças das falanges distais será usada posteriormente no rastreamento e na estimativa de *pose* da câmera, como se segue.

4.6.3 Rastreamento dos dedos

Uma vez que as extremidades dos dedos são detectadas, elas são rastreadas com base na correspondência das extremidades atuais com as previamente rastreadas, ou seja do *frame* imediatamente anterior. Para isso, rastreia-se a trajetória da cabeça da falange distal com um algoritmo de correspondência, baseado em (10), que minimiza o deslocamento de pares de dedos sobre dois *frames*. Também foi utilizado o *centroid* da mão para lidar com grandes movimentos da seguinte maneira: o custo de correspondência é minimizado pela fórmula a seguir.

$$\mathcal{F}_{i+1} = \min \sum_{j=0}^{N-1} \| (\mathcal{F}_{i,j} - \mathcal{C}_i) - (\mathcal{F}_{i+1,j} - \mathcal{C}_{i+1}) \| \quad (6)$$

onde \mathcal{F}_i e \mathcal{F}_{i+1} são conjuntos com N extremidades de dedos nos i-ésimo e i-ésimo + 1 *frames* respectivamente, $\mathcal{F}_{i,j}$ representa a extremidade do dedo com índice j-ésimo em \mathcal{F}_i , e \mathcal{C}_i e \mathcal{C}_{i+1} são os centroides dos *frames* correspondentes.

4.6.4 Pose Estimation das extremidades dos dedos

Usando o modelo da mão carregada na inicialização e uma estimativa dos parâmetros intrínsecos da câmera, pode-se estimar uma *pose* da câmera. O método *FingertipTracker.findExtrinsicCameraParams* utiliza esses dados para calcular os parâmetros extrínsecos da mesma. Foi utilizada a função *solvePnP*, disponibilizada pela OPenCV para calcular a *pose estimation*. Essa função recebe os pontos dos modelos 3D, das extremidades dos dedos na imagem, os parâmetros intrínsecos da câmera e executa uma versão do algoritmo PnP (Seção 3.2.1) para estimar o *rotate* e o *translate* da câmera, respresentando os parâmetros extrínsecos, conforme explicado na Seção 3.1.2.

Supondo que possa haver erros no rastreamento dos dedos, é vantajoso suavizar a estimativa de *pose* da câmera usando um filtro *Kalman* (Ver Montanari (44) para mais detalhes sobre o funcionamento deste filtro), modelando a posição e orientação da câmera.

O algoritmo de filtro de *Kalman* envolve dois passos: predição e correção. O primeiro passo usa estados anteriores para prever o estado atual. A segunda etapa usa a medida atual, como a localização do objeto, para corrigir o estado (45). Esses dados do filtros de *Kalman* mantém o

rastreamento dos dedos mesmos que eles não possam ser detectados por alguns *frames*.

Na implementação desse sistema foi definido o estado x do filtro de *Kalman* como sendo:

$$x = \begin{bmatrix} t \\ r \\ v_t \\ v_r \end{bmatrix} \quad (7)$$

onde t é um 3-vetor com o *translate* da câmera, r é um *quaternion* para rotação, e v_t e v_r é velocidade de *translate* e *rotate*, respectivamente. A matriz de transição de estado A_{14x14} é então definida como sendo:

$$A = \begin{bmatrix} I_{7x7} & D(\Delta t)_{7x7} \\ 0 & I_{7x7} \end{bmatrix} \quad (8)$$

onde I é uma matriz identidade e $D(\Delta t)$ é uma matriz diagonal com os valores de Δt (período de tempo entre os *frames* capturados). A *measurement* y é modelada diretamente como sendo:

$$y = \begin{bmatrix} t \\ r \end{bmatrix} \quad (9)$$

onde t e r são os mesmos que em (x) . A matriz de observação H e a matriz de direção G são definidas como sendo:

$$H = \begin{bmatrix} I_{7x7} & 0 \end{bmatrix}_{7x14} \quad (10)$$

$$G = \begin{bmatrix} 0 \\ I_{7x7} \end{bmatrix}_{14x7} \quad (11)$$

mapeando diretamente o vetor de estado para o vetor de *measurement*. Ao usar o filtro de *Kalman* conforme definido acima, a estimativa da *pose* da câmera é suavizada, segundo (10).

4.6.5 Configuração da câmera e renderização

Após calcular o *pose estimation*, tem-se todos os dados necessários para configurar uma câmera virtual e fazer a renderização de objetos. Neste trabalho foi utilizada a biblioteca OpenCV (Seção 4.2.2) para realizar tais operações.

Para a correta apresentação dos dados desejados, alguns parâmetros da OpenGL precisam ser configurados a cada novo *frame*, tais como, a projeção e a posição do observador. A matriz de projeção foi configurada na função *FingertipPoseEstimation.setOpenGLFrustum* utilizando

a função *glFrustum* da própria OpenGL. Esta função multiplica a matriz atual por uma matriz de perspectiva que é montada utilizando os parâmetros intrínsecos da câmera.

A posição do observador foi configurada no método *FingertipPoseEstimation.setOpenGLModelView* utilizando a função *gluLookAt*, disponibilizada pela GLUT e utilizando os parâmetros extrínsecos calculados no processo de *pose estimation*. Ambos métodos foram desenvolvidos na HandyAR e estão sendo utilizados nesta aplicação. Para mais detalhes sobre o cálculo feito em suas implementações, visite (10)

Por fim, para inspecionar um objeto de RA sobre a mão em diferentes ângulos de visão, o usuário pode girar ou mover a mão arbitrariamente. Com base na simetria das mãos esquerda e direita, o modelo da mão que é construído a partir da mão esquerda, também pode ser usado para a mão direita.

5 RESULTADOS

Para que fosse realizada a mensuração da taxa de *frames* por segundo (fps), foram utilizadas algumas funções da biblioteca OpenCV, que como resultado mostraram que o sistema processa a uma taxa média de 28 fps, aproximadamente. Esta taxa diminui quando tem-se um longo período de tempo sem nenhuma mão rastreada, sendo esta ainda superior a 20 fps. Essa diminuição se justifica devido ao processo de detecção demandar mais processamento do que o rastreamento, conforme pode ser observado na Seção 4.6.2.

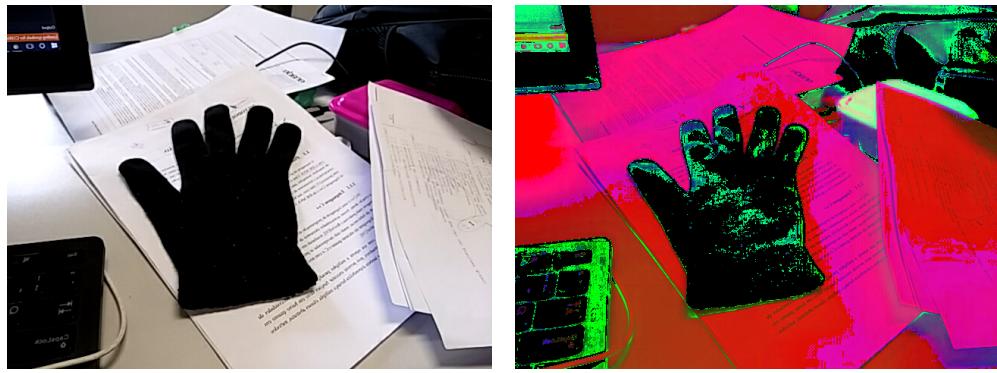
Segundo (10), a taxa mínima necessária para um sistema ser considerado de tempo real, é de 15 fps. Neste contexto o sistema desenvolvido conseguiu atingir o requisito de processamento em tempo real, pois a taxa de processamento mais baixa obtida foi de 20 fps. Com isso, as próximas seções mostrarão os resultados obtidos no processo de segmentação, detecção e rastreamento. A Seção 5.1 inicia apresentando os resultados obtidos após a execução de cada uma das principais tarefas da segmentação presente no Capítulo 4, ao ser utilizado um *frame* como exemplo.

5.1 Resultados da segmentação e detecção

Nesta seção será mostrada uma imagem capturada após a aplicação de determinadas operações no *frame* capturado, sendo que todas as operações a seguir foram explicadas no Capítulo 4.

O primeiro passo é a captura do *frame*, mostrado na Figura 14a. Seguido da conversão desse *frame* capturado para o sistema de cores HSV, conforme apresentado na Figura 14b. Posteriormente é aplicado o filtro pela cor preta (Figura 15a) e ao se aplicar o filtro *blur*, obtém-se uma imagem suavizada (com menos ruídos), conforme Figura 15b. Com a imagem suavizada, é aplicado o limiar que consegue descartar algumas informações irrelevantes da imagem e montar uma imagem binária, demonstrada na Figura 16a. Após obter a imagem binária, é aplicada a transformada da distância para procurar o *centroid* da maior região, o resultado dessa transformada pode ser observado na Figura 16b. Ao ser calculado o *centroid*, passa-se pelo processo de localizar a região com maior área, demostrado na Figura 17

Figura 14: Processo inicial

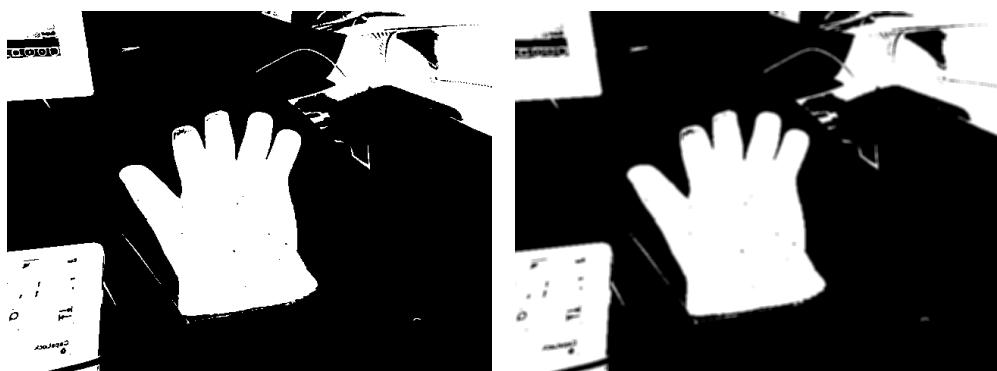


(a) *Frame* capturado.

(b) *Frame* convertido para o sistema de cores HSV.

Fonte: Elaborado pelo autor.

Figura 15: Aplicação dos filtros



(a) Filtro realizado nos canais pela cor preta. (b) Remoção de ruídos com o filtro de suavização *blur*.

Fonte: Elaborado pelo autor.

Figura 16: Limiar e transformada da distância



(a) Aplicando o limiar.

(b) Resultado da transformada da distância.

Fonte: Elaborado pelo autor.

Figura 17: Contorno da maior região encontrada



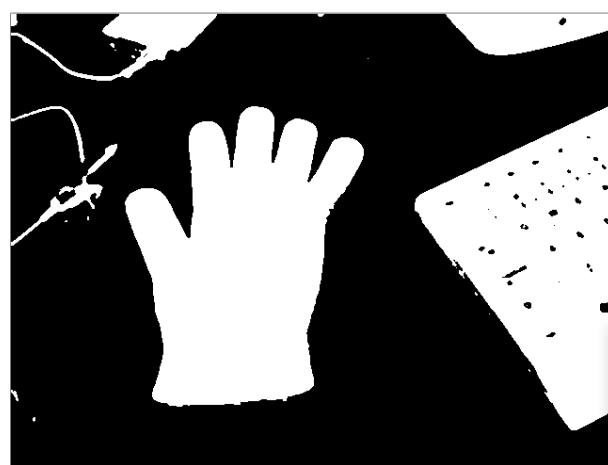
Fonte: Elaborado pelo autor.

Por fim, tem-se apenas a região da mão na imagem resultado, que por sua vez, será repassada para a detecção e futuramente pelo rastreamento. O resultado do rastreamento é mostrado nos testes a seguir, que foram feitos em variados ambientes.

5.2 Ambientes 1

O primeiro teste aqui demonstrado foi realizado em um ambiente com a presença de vários outros objetos pretos. Este teste realizado em uma sala com iluminação excessiva (lâmpadas de *led*), o que dificulta a detecção da cor, e apenas com a luva, deixando os dedos parcialmente justapostos. A Figura 18, mostra a segmentação inicial onde estão presentes todos os objetos pretos encontrados.

Figura 18: Segmentação inicial



Fonte: Elaborado pelo autor.

No entanto, a mão é o maior objeto dentre os detectados, o que faz com que ela consiga ser detectada, como pode ser visto na Figura 19

Figura 19: Resultado apresentado apos todo o processamento



Fonte: Elaborado pelo autor.

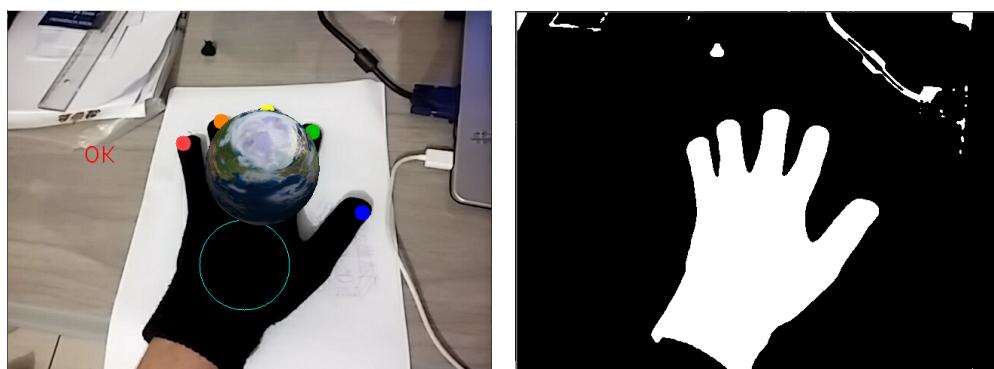
Como pôde ser visto, a detecção foi realizada, porém as elipses não foram detectadas precisamente devido a justaposição dos dedos, mas as pontas dos dedos foram detectadas corretamente e o objeto virtual foi posicionado como o esperado.

5.3 Ambiente 2

O segundo teste foi realizado durante a noite em um ambiente com vários outros itens de cores diversas. Este ambiente possuía iluminação ruim, com a presença de sombras e reflexos, como pode ser observado nas próximas figuras. A câmera utilizada era de baixa qualidade e possui resolução de 640x480.

A Figura 20 mostra o resultado da segmentação e da projeção do objeto virtual estando a mão em cima de um papel, voltada para baixo, sendo possível verificar que o rastreamento foi realizado com sucesso.

Figura 20: Resultados



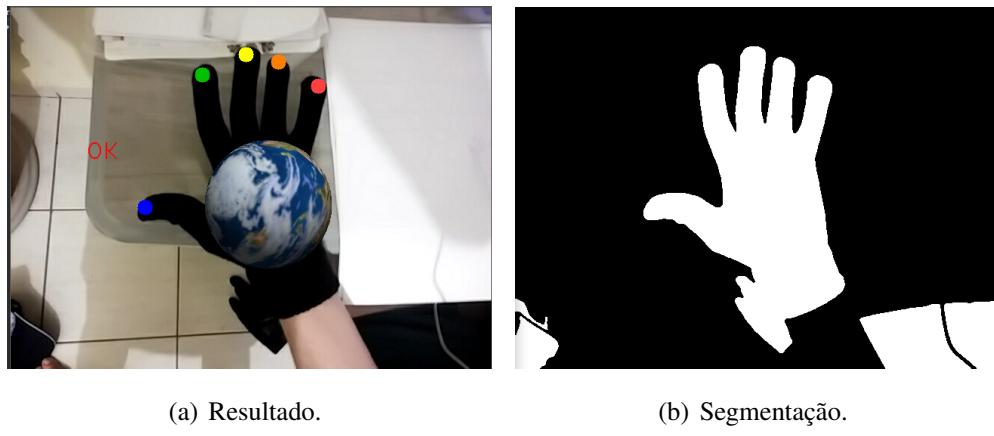
(a) Resultado.

(b) Segmentação.

Fonte: Elaborado pelo autor.

A próxima figura, 21, possui duas características diferentes das anteriores. Primeiro, a mão está voltada para cima, mostrando que o rastreamento também funciona corretamente com a mão desse modo, a outra diferença é que a mão não está totalmente sobre a mesa, estando prestes alguns objetos pretos, como pode ser visualizado no chão.

Figura 21: Resultados



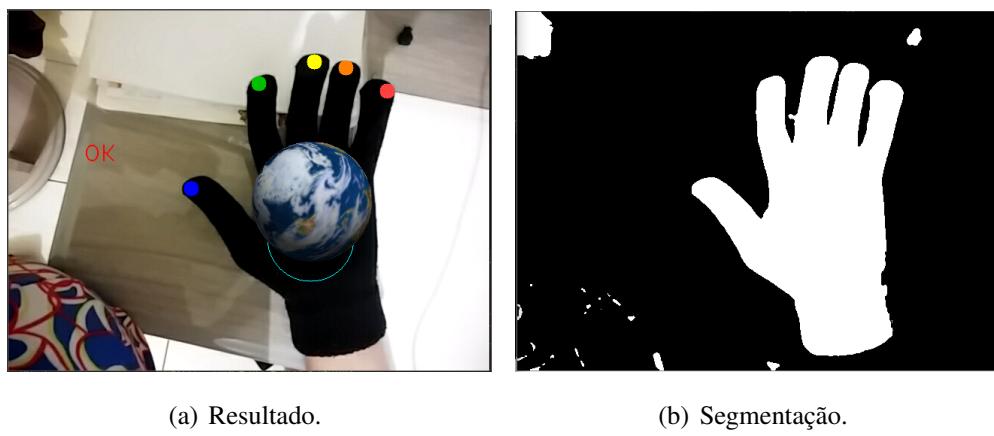
(a) Resultado.

(b) Segmentação.

Fonte: Elaborado pelo autor.

Na Figura 22 encontra-se a mesma situação da figura anterior, porém o outro objeto preto está no canto superior esquerdo.

Figura 22: Resultados



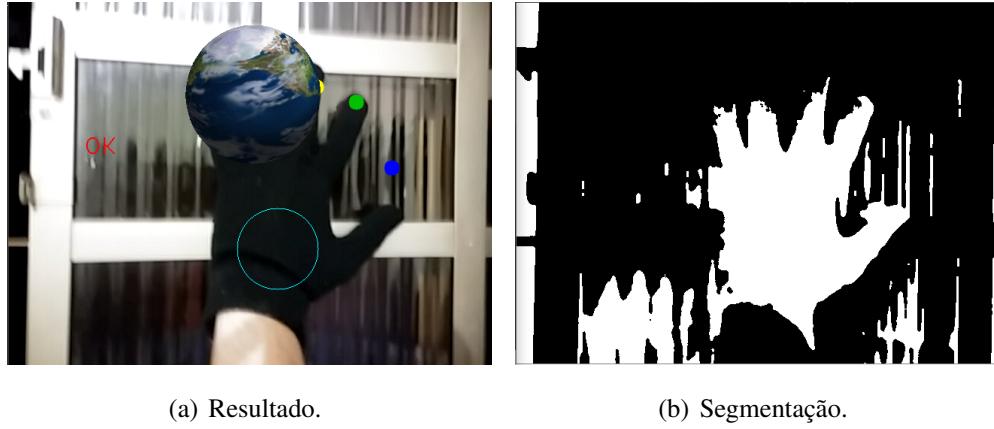
(a) Resultado.

(b) Segmentação.

Fonte: Elaborado pelo autor.

A figura seguinte, 23, visa mostrar o comportamento do sistema quando temos a presença de muito reflexo, como pode ser visto no vidro da janela. A mão foi posicionada em frente ao vidro da janela apontada para uma região sem luz do outro lado. É possível perceber que a segmentação ficou parcialmente comprometida, porém o rastreamento ainda conseguiu ser realizado.

Figura 23: Resultados



Fonte: Elaborado pelo autor.

A Figura 24 apresenta a situação onde tem-se um objeto preto tão grande quanto a mão, que neste caso os cabelos. O rastreamento só foi realizado com sucesso quando uma parte da região do cabelo foi coberta.

Figura 24: Resultados



Fonte: Elaborado pelo autor.

Na Figura 25 tem-se a mesma situação da figura anterior, entretanto, a mão foi posicionada mais próximo da câmera, tornando-se maior do que a a região da cabeça. É possível perceber que o polegar não foi detectado na posição correta devido ao fato de o olho também possuir curvaturas similares a dos dedos, no entanto o rastreamento ainda conseguiu ser realizado com sucesso.

Figura 25: Resultados



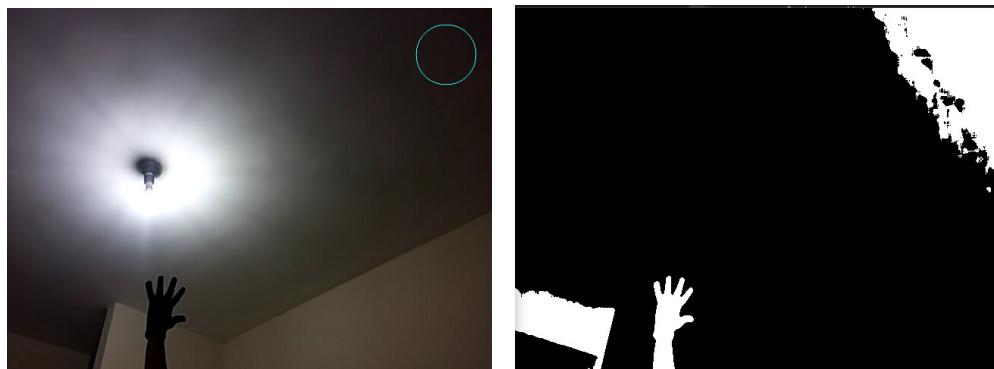
(a) Resultado.

(b) Segmentação.

Fonte: Elaborado pelo autor.

Por fim, tem-se um exemplo onde o rastreamento não foi realizado com sucesso, Figura 26. Isso ocorre devido ao fato de a luz estar mal distribuída, deixando algum ponto escuro dentro da sala. Então, mesmo a mão tendo sido segmentada corretamente, ela não será reconhecida, pois o sistema não tem reconhecimento por modelo.

Figura 26: Resultados



(a) Resultado.

(b) Segmentação.

Fonte: Elaborado pelo autor.

6 CONCLUSÕES

A técnica de rastreamento 3D baseada nas extremidades dos dedos foi implementada com êxito e apresentou uma boa taxa de quadros por segundo, assim como um bom índice de robustez em relação a movimentação, alteração de posição da câmera e pequenas alterações de ambientes e iluminação.

No entanto, quando o rastreamento não está sendo realizado, houve uma diminuição significativa na taxa de fps, o que indica a necessidade de melhoria no algoritmo que executa a tentativa de detecção, apesar da taxa de fps ainda ter demonstrado ser superior ao limite mínimo necessário para ser considerado um sistema em tempo real.

Porém, independentemente destes resultados, a técnica ainda se mostra instável em situações onde são realizados movimentos abruptos, uma vez que os dados armazenados no processo de rastreamento são dispensados a medida em que se tem uma grande sequência de *frames* sem nenhuma detecção realizada. Isso implica que deve ser realizado todo o processo de detecção quando a mão voltar à cena. Essa instabilidade também ocorre em grandes variações de iluminação.

O rastreamento também não foi realizado com sucesso quando há, na detecção inicial, um objeto de cor preta com área maior do que a da mão. Contudo, casos onde a mão já estava sendo rastreada, e o objeto maior não entrou em contato com a região da mão, o rastreamento continuou sendo realizado com sucesso.

6.1 Trabalhos futuros

Com base nas limitações apresentadas anteriormente e em potencial, o projeto pode ser melhorado e expandido das seguintes formas:

- Otimização dos algoritmos utilizado para aumentar a taxa de fps.
- Aprimoramento da robustez em relação a variações de iluminação (uso de algoritmos adaptativos, por exemplo).
- Melhoramento da etapa de reconhecimento ser possível realizar a detecção, mesmo a mão não sendo a maior região da imagem.
- Implementar o reconhecimento de gestos e sinas para que a técnica seja utilizada na interação homem-máquina.

REFERÊNCIAS

- 1 TEAM, O. D. **Image Filtering**. 2014. Disponível em: <<https://docs.opencv.org/2.4/modules/imgproc/doc/filtering.html?highlight=filter2d#filter2d>>. Acesso em: 12 out. 2017.
- 2 KIRNER, C.; SISCOUTTO, R. **Realidade Virtual e Aumentada: Conceitos, Projeto e Aplicações**. [S.I.]: IX Symposium on Virtual and Augmented Reality, 2014.
- 3 FARIAS, T.; FERNANDES, B. Incentivo à leitura com a utilização de livros em realidade aumentada. **CONF-IRM**, p. 355–366, 2013. Disponível em: <<http://aisel.aisnet.org/confirm2013/57>>.
- 4 SIMÕES, F. P. Realidade aumentada sem marcadores baseada em arestas, um estudo de caso. **Trabalho de Graduação, Universidade Federal de Pernambuco, Recife**, 2008.
- 5 LIMA, J. Paulo Silva do M. Realidade aumentada sem marcadores multiplataforma utilizando rastreamento baseado em modelo. Universidade Federal de Pernambuco, 2010.
- 6 GRACIANO, A. B. V. **Rastreamento de objetos baseado em reconhecimento estrutural de padrões**. Dissertação (Mestrado) — Universidade de São Paulo, mar 2007. Disponível em: <<http://www.teses.usp.br/teses/disponiveis/45/45134/tde-05112009-204609/publico/MestradoAnaGraciano.pdf>> Acesso em: 31 maio. 2017.
- 7 SAPORITO, S. de S. **PROPOSTA DE UM MARCADOR FIDUCIAL E ALGORITMO PARA ESTIMATIVA DE COORDENADAS ESPACIAIS**. 2017. TCC (Graduação) - Engenharia de Computação, Universidade Federal do Rio de Janeiro, Rio de Janeiro, Brasil. Disponível em: <<http://monografias.poli.ufrj.br/monografias/monopolio10019576.pdf>>. Acesso de 05 out. 2017.
- 8 KIRNER, C. **Realidade Virtual e Aumentada**. 2017. Disponível em <<http://www.realidadevirtual.com.br>> Accessado em 14-10-2017.
- 9 OLIVEIRA, D.; PEREIRA, L.; SILVA, R. L. d. S. da. Uma arquitetura de software para bibliotecas de realidade aumentada baseada em marcadores naturais. **Revista de Informática Teórica e Aplicada**, v. 24, n. 1, p. 41–58.
- 10 LEE, T.; HOLLERER, T. Handy ar: Markerless inspection of augmented reality objects using fingertip tracking. In: **2007 11th IEEE International Symposium on Wearable Computers**. [S.I.: s.n.], 2007. p. 83–90. ISSN 1550-4816.
- 11 WAZLAWICK, R. S. **Metodologia de Pesquisa para Ciência da Computação**. [S.I.]: Rio de Janeiro: Elsevier, 2007.
- 12 NASCIMENTO, H.; FERREIRA, C. Uma introdução à visualização de informações - doi 10.5216/vis.v9i2.19844. **Visualidades**, v. 9, n. 2, 2012. Disponível em: <<https://www.revistas.ufg.br/VISUAL/article/view/19844>>. Acesso em: 16 out. 2017.
- 13 MELO, N. **Abordagens do processo de Segmentação: Limiarização, Orientada a Regiões e Baseada em Bordas**. 2011. Disponível em: <<http://www.dsc.ufcg.edu.br/pet/jornal-setembro2011/materias/recapitulando.html>>. Acesso em: 12 out. 2017.

- 14 RIOS, L. R. S. **Visão Computacional**. [2010]. Disponível em: <[http://homes.dcc.ufba.br/~luizromario/Apresenta%C3%A7%C3%A3o%20de%20IA/Artigo%20\(final\).pdf](http://homes.dcc.ufba.br/~luizromario/Apresenta%C3%A7%C3%A3o%20de%20IA/Artigo%20(final).pdf)> Acesso em: 26 jul. 2017.
- 15 ANAMI, B. M. **BOAS PRÁTICAS DE REALIDADE AUMENTADA APLICADA À EDUCAÇÃO**. 2013. TCC (Graduação) - Curso de Ciência da Computação, Universidade Estadual de Londrina, Londrina, Brazil.
- 16 MARENCONI, M.; STRINGHINI, D. Tutorial: Introdução à visão computacional usando opencv. **Rita, Farroupilha**, v. 1, n. 16, p. 125–160, 2009. Disponível em: <<https://ai2-s2-pdfs.s3.amazonaws.com/a0a6/301a239519f117ca6c0398d5230ff15c67ff.pdf>>. Acesso em: 27 jul. 2017.
- 17 MANSOUR, I. H.; COHEN, M. Introdução à computação gráfica. **RITA**, v. 13, n. 2, p. 43–68, 2006.
- 18 BATISTA, N. C. Introdução à computação gráfica.
- 19 JUKEMURA, A. S. **A Realidade Aumentada Aplicada ao Gerenciamento de Redes de Computadores**. Dissertação (Mestrado) — Universidade Federal de Goiás, Goiânia, 2007.
- 20 IDEIAS, A. S. **Realidade Aumentada Na Arquitetura**. 2016. Disponível em: <<http://arquitetesuasideias.com.br/2016/08/29/realidade-aumentada-na-arquitetura/>> Acesso em: 15 out. 2017.
- 21 AR, J. do. **VEJA EM PRIMEIRA PESSOA O QUE ACONTECE NO HUD DO BOEING 737**. 2014. Disponível em: <<https://jornaldoar.com/2014/11/veja-em-primeira-pessoa-o-que-acontece-no-hud-do-boeing-737/>> Acesso em: 15 out. 2017.
- 22 RODINO, M. **QUAIS AS MELHORES APLICAÇÕES DA REALIDADE AUMENTADA?** 2017. Disponível em: <<http://www.flexinterativa.com.br/blogflex/quais-as-melhores-aplicacoes-de-realidade-aumentada>>. Acesso em: 26 jul 2017.
- 23 COGNITIVO, J. **Veja como os óculos de realidade aumentada da Microsoft podem mudar a medicina**. 2015. Disponível em: <<https://www.menosfios.com/oculos-de-realidade-aumentada-da-microsoft-podem-mudar-a-medicina/>> Acesso em: 15 out. 2017.
- 24 AMANDA. **Detalhes leva inovação e tecnologia para a sala de aula**. [201–]. Disponível em: <<http://www.detalheseducacao.com.br/blog/realidade-aumentada-2/agencia-detalhes-leva-inovacao-e-tecnologia-para-a-sala-de-aula/>> Acesso em: 15 out. 2017.
- 25 ART TOOLKIT. **Sobre o ARToolKit**. 2017. Disponível em: <<https://archive.artoolkit.org/aboutartoolkit>>. Acesso em: 27 jul. 2017.
- 26 ARAÚJO, J. C. d. et al. Uso de realidade virtual e aumentada na visualização do fluxo do campo magnético de um motor de indução monofásico. Universidade Federal de Uberlândia, 2008.
- 27 TSUMAKI, Y. et al. Design of a compact 6-dof haptic interface. In: IEEE. **Robotics and Automation, 1998. Proceedings. 1998 IEEE International Conference on**. [S.I.], 1998. v. 3, p. 2580–2585.

- 28 MALLICK, S. **Head Pose Estimation using OpenCV and Dlib.** 2016. Disponível em: <<https://www.learnopencv.com/head-pose-estimation-using-opencv-and-dlib/>>. Acesso em: 05 jul. 2017.
- 29 OPENCV. **Camera calibration With OpenCV.** 2017. Disponível em: <https://docs.opencv.org/2.4/doc/tutorials/calib3d/camera_calibration/camera_calibration.html>. Acesso em: 05 nov. 2017.
- 30 ZHENG, Y. et al. Revisiting the pnp problem: A fast, general and optimal solution. In: **Proceedings of the IEEE International Conference on Computer Vision.** [S.l.: s.n.], 2013. p. 2344–2351.
- 31 NEVES, S. C. M.; PELAES, E. G.; SINAIS, L. d. P. de. Estudo e implementação de técnicas de segmentação de imagens. **Revista Virtual de Iniciação Acadêmica da UFPA-Universidade Federal do Pará–Departamento de Engenharia Elétrica e de Computação**, v. 1, n. 2, 2008.
- 32 FONSECA, M. S. da. **Segmentação de Imagem: Segmentação Limiar em Tons de Cinza.** 2017. Disponível em: <<http://www2.ic.uff.br/~aconci/limiarizacao.htm>>. Acesso em: 08 out. 2017.
- 33 KAUR, D.; KAUR, Y. Various image segmentation techniques: A review. **IJCSCMC**, v. 3, n. 5, p. 809–814, 2014. Disponível em: <<http://ijcsmc.com/docs/papers/May2014/V3I5201499a84.pdf>>. Acesso em: 12 out. 2017.
- 34 NOVAES, R. **O que é e para que serve IDE?** 2014. Disponível em: <<http://www.psafe.com/blog/o-que-serve-ide/>>. Acesso em: 14 nov. 2017.
- 35 PACIEVITCH, Y. **C++.** 2016. Disponível em: <<https://www.infoescola.com/informatica/cpp/>>. Acesso em: 14 nov. 2017.
- 36 OPENCV. **VideoCapture Class Reference.** 2017. Disponível em: <https://docs.opencv.org/3.2.0/d8/dfe/classcv_1_1VideoCapture.html>. Acesso em: 05 nov. 2017.
- 37 SILVA, A. S. et al. Object tracking by color and active contour models segmentation. **IEEE Latin America Transactions**, IEEE, v. 14, n. 3, p. 1488–1493, 2016.
- 38 FERNANDO, S. **OpenCV Tutorial C++.** 2017. Disponível em: <<http://opencv-srf.blogspot.com.br/2013/10/smooth-images.html>>. Acesso em: 15 nov. 2017.
- 39 MENOTTI, D. **Reconhecimento de Padrões.** Ouro Preto: [s.n.], 2011. 33 slides, color. Disponível em: <www.decom.ufop.br/menotti/rp102/slides/02-Segmentacao.ppt>. Acesso em: 12 out. 2012.
- 40 OPENCV. **Structural Analysis and Shape Descriptors.** 2017. Disponível em: <https://docs.opencv.org/2.4/modules/imgproc/doc/structural_analysis_and_shape_descriptors.html>. Acesso em: 15 nov. 2017.
- 41 WILLOW, E. **Contours : Getting Started.** 2016. Disponível em: <<http://opencv24-python-tutorials.readthedocs.io.html>>. Acesso em: 15 nov. 2017.

- 42 OPENCV. **Miscellaneous Image Transformations**. 2017. Disponível em: <https://docs.opencv.org/2.4/modules/imgproc/doc/miscellaneous_transformations.html>. Acesso em: 15 nov. 2017.
- 43 FISHER, R. et al. **Distance Transform**. 2003. Disponível em: <<https://homepages.inf.ed.ac.uk/rbf/HIPR2/distance.htm>>. Acesso em: 15 nov. 2017.
- 44 MONTANARI, R. **Detecção e classificação de objetos em imagens para rastreamento de veículos**. Tese (Doutorado) — Universidade de São Paulo, 2016.
- 45 MATLAB. **Kalman Filter**. 2014. Disponível em: <<https://www.mathworks.com/help/vision/ref/vision.kalmanfilter-class.html>>. Acesso em: 15 nov. 2017.

APÊNDICE A – Módulos

Essa apêndice contém todos os módulos que forma desenvolvido no decorrer deste trabalho.

A.1 Módulo *Capture*

```
#pragma once

#include <opencv2\opencv.hpp>

using namespace cv;

#define CAPTURE_SIZE_WIDTH 640
#define CAPTURE_SIZE_HEIGHT 480

class Capture
{
public:
    Capture();
    ~Capture();

    bool initialize(int index);
    void terminate();

    Mat &nextFrame();

    int getFrameWidth() const;
    int getFrameHeight() const;

private:
    VideoCapture cap;
    Mat frame;
};

#include "Capture.h"

Capture::Capture()
{
}

bool Capture::initialize(int index)
{
    //Abra a camera do indice passado
    cap.open(index);

    //Verifica se conseguiu abrir a camera
    if (!cap.isOpened())
        return false;

    //Definindo a altura e a largura
    cap.set(CV_CAP_PROP_FRAME_WIDTH, CAPTURE_SIZE_WIDTH);
    cap.set(CV_CAP_PROP_FRAME_HEIGHT, CAPTURE_SIZE_HEIGHT);

    return true;
}
```

```

Mat &Capture :: nextFrame()
{
    cap >> frame;

    if (frame.empty())
        throw("Frame vazio.");

    // Invertendo a imagem, pois a opencv e opengl armazena imagem de forma
    // oposta
    flip(frame, frame, 0);

    return frame;
}

int Capture :: getFrameWidth() const
{
    return CAPTURE_SIZE_WIDTH;
}

int Capture :: getFrameHeigth() const
{
    return CAPTURE_SIZE_HEIGHT;
}

```

A.2 Módulo HandRegion

```

#pragma once

#include <opencv2\opencv.hpp>

#include "ArException.h"

using namespace cv;
using namespace std;

class HandRegion
{
public:
    HandRegion();

    Mat getHandRegion(Mat &srcImage);
};

#include "HandRegion.h"

HandRegion :: HandRegion()
{
}

Mat HandRegion :: getHandRegion(Mat &srcImage)
{
    Mat image;

    try {
        //Converte para o esquema HSV
        cvtColor(srcImage, image, COLOR_BGR2HSV);

```

```

//Faz filtro pela cor preta
inRange(image, Scalar(0, 0, 0), Scalar(180, 255, 30), image);

//suaviza a imagem usando um kernel (ver kernel na documentação da
opencv)
blur(image, image, Size(5, 5));

// Aplica um limiar de nÃvel fixo para cada elemento da matriz
// Converte a imagem para binÃaria.
threshold(image, image, 128, 255, CV_THRESH_BINARY);

//-----
// Morphology Operation (Opening and Closing)

// Faz uma erosão seguida de uma dilatação – Remove ruído cv
morphologyEx(image, image, CV_MOP_OPEN, 0);

//Faz uma dilatação seguida de uma erosão –
// fecha pequenos orifícios dentro dos objetos
morphologyEx(image, image, CV_MOP_CLOSE, 0);
//-----

}

catch (exception &e) {
    throw ArException("HandRegion", "getHandRegion", e);
}

return image;
}

```

A.3 Módulo *FingertipProcess*

```

#pragma once

#include <opencv2\opencv.hpp>
#include <vector>

#include "ArException.h"

using namespace cv;
using namespace std;

#define MAX_END_POINT          1000

#define MIN_CURVATURE_STEP     10
#define MAX_CURVATURE_STEP     50
#define FINGERTIP_ANGLE_THRESHOLD 0.5
#define GAP_POINT_THRESHOLD     10
#define CONNECTED_POINT_THRESHOLD 10

class FingertipProcess
{
public:
    FingertipProcess();

    void reset();
}

```

```

void initialize(Size size);
int findFingerTipCandidatesByCurvature(Mat handRegion);

public:

void getMaxDistPoint(Mat &mat, Mat *output);
void clearVet();

// Tamanho das imagens que serão processadas
Size _size;

// Pontos finais (Candidatos a ponto de dedo)
vector<Point2f> _vetEndPoints;
vector<float> _vetEndPointDist;
vector<float> _vetEndPointScore;

// Armazena um componente conectado para a mão
Mat _ccContourImage;

// Transformação da distância
Point _maxDistPoint;
double _maxDistValue;
Mat _distHandImage; // DTI
Mat _distImage; // TDTI

// Ellipses
vector<RotatedRect> _ellipses;
};

#include "FingertipProcess.h"

FingertipProcess :: FingertipProcess()
{
}

void FingertipProcess :: reset()
{
    _vetEndPoints . clear();
    _vetEndPointDist . clear();
    _vetEndPointScore . clear();
    _ellipses . clear();
}

void FingertipProcess :: initialize(Size size)
{
    _size = size;
    _ccContourImage = Mat::zeros(_size, CV_8UC1);
    _distImage = Mat::zeros(_size, IPL_DEPTH_32F);
    _distHandImage = Mat::zeros(_size, IPL_DEPTH_32F);
}

//
// Aplica a transformada da distância e busca o ponto com distância máxima
//
void FingertipProcess :: getMaxDistPoint(Mat &mat, Mat *output)
{
    // Função da opencv que calcula a transformada da distância
    distanceTransform(mat, *output, CV_DIST_L2, CV_DIST_MASK_3);
}

```

```

// Esse ponto, consequentemente será o centroid da mão
_maxDistValue = 0;
_maxDistPoint = cvPoint(0, 0);

// Pega o valor maximo de _pDistImage2 e o ponto que tem esse valor
// máximo
minMaxLoc(*output, NULL, &_maxDistValue, NULL, &_maxDistPoint);

// Ajustando os limites
if (_maxDistPoint.x == 0) _maxDistPoint.x++;
if (_maxDistPoint.y == 0) _maxDistPoint.y++;
if (_maxDistPoint.x >= _size.width - 1) _maxDistPoint.x--;
if (_maxDistPoint.y >= _size.height - 1) _maxDistPoint.y--;
}

int FingertipProcess :: findFingerTipCandidatesByCurvature(Mat handRegion)
{
    reset();

    if (handRegion.empty())
        return 0;

    // Se não tem pontos finais
    if (!vetEndPoints.size()) {
        // Encontro o ponto com distância máxima da borda
        getMaxDistPoint(handRegion, &_distImage);
        // Util::savePrint("8_dist_trans", _distImage);
    }

    //
    // Encontrar contornos para obter um único componente conectado para a
    // mão que possui o ponto de distância máxima nele.
    //
    vector< vector<Point> > contours;
    int i;

    // Localiza contornos em uma imagem binária.
    findContours(handRegion, contours, CV_RETR_EXTERNAL,
                CV_CHAIN_APPROX_SIMPLE, Point(0, 0));

    // Verifico se foi encontrado algum contorno
    if (contours.empty()) return 0;

    // Limpando a matriz que armazenará os contornos
    _ccContourImage.setTo(Scalar::all(0));

    // Percorre todos os contornos encontrados -
    // Um contorno é determinado por um conjunto de pontos
    for (i = 0; i < int(contours.size()); ++i) {

        // Pego o contorno atual
        auto contour = contours[i];

        // Se o contorno tem mais de 100 pontos
        if (contour.size() <= 100) continue;

```

```

// Desenha o contorno e faz o preenchimento - CUIDADO
drawContours(_ccContourImage, contours, i, Scalar(255), CV_FILLED);

// Verifico se o ponto maximo (centro da mão) foi "pintado" por este
contorno
if (int(_ccContourImage.at<uchar>(_maxDistPoint)) == 255) {

    // Limpando os contornos, para redesenhar apenas o contorno atual
    _ccContourImage.setTo(Scalar::all(0));

    // Desenha o contorno atual novamente
    drawContours(_ccContourImage, contours, i, Scalar(255), CV_FILLED)
        ;

    // Calcula o ponto de maior distancia dentro da imagem
    getMaxDistPoint(_ccContourImage, &_distHandImage);

    break;
}

// Se não achou contorno
if (i == int(contours.size()) || contours[i].size() < 100)
    return 0;

//
// Encontrar pontos máximos curvatura local
//

// Variaveis utilizadas
int nPointsConnected = 0;
int nPointsGap = 0;
double minAngle = 0;
int minPointIndex = -1;
double mediumIndex = 0;
double sumAngle = 0;
double meanPointX = 0;
double meanPointY = 0;
int min_curvature_step = int(_maxDistValue * 0.2);
int max_curvature_step = int(_maxDistValue * 1.0);
if (min_curvature_step < MIN_CURVATURE_STEP)
    min_curvature_step = MIN_CURVATURE_STEP;
if (max_curvature_step > MAX_CURVATURE_STEP)
    max_curvature_step = MAX_CURVATURE_STEP;
int startIndex, endIndex;

// Contorno escolhido
vector< Point > &contour = contours[i];
int countPoints = int(contour.size());

// Percorrendo os pontos novamente
for (i = 0; i < countPoints; ++i)
{
    //
    // For different scale, calculate curvature
    // _maxDistValue is approximately our bound.
    //

```

```

double minK = 0;
double minDir = 0;
double count = 0;

for (int k = min_curvature_step; k <= max_curvature_step; k++)
{
    int iPrev = (i - k + countPoints) % countPoints;
    int iNext = (i + k) % countPoints;

    // Compute Curvature
    int pp1_x = contour[iPrev].x - contour[i].x;
    int pp1_y = contour[iPrev].y - contour[i].y;
    int pp2_x = contour[iNext].x - contour[i].x;
    int pp2_y = contour[iNext].y - contour[i].y;

    // dot product ( cosine )
    float cross = (pp1_x*pp2_x + pp1_y*pp2_y)
        / (sqrt((float)pp1_x*pp1_x + pp1_y*pp1_y) *
        sqrt((float)pp2_x*pp2_x + pp2_y*pp2_y));
    // cross product ( sign of z-component )
    double dir = pp1_x*pp2_y - pp1_y*pp2_x;

    if (minK <= cross)
    {
        minK = cross;
        minDir = dir;
    }
}

// Tomar os pontos com base na curvatura e direção
if (minK > FINGERTIP_ANGLE_THRESHOLD && minDir > 0)
{
    nPointsGap = 0;
    if (nPointsConnected == 0) startIndex = i;
    nPointsConnected++;
    if (minAngle < minK)
    {
        minAngle = minK;
        minPointIndex = i;
    }
    mediumIndex += (minK * i);
    meanPointX += (minK * contour[i].x);
    meanPointY += (minK * contour[i].y);
    sumAngle += minK;
}

else {

    nPointsGap++;
    if (nPointsGap >= GAP_POINT_THRESHOLD && nPointsConnected >
        CONNECTED_POINT_THRESHOLD)
    {
        // check image boundary
        if (contour[minPointIndex].x > 10 &&
            contour[minPointIndex].x < _size.width - 10 &&
            contour[minPointIndex].y > 10 &&
            contour[minPointIndex].y < _size.height - 10)
        {
            mediumIndex /= sumAngle;
        }
    }
}

```

```

meanPointX /= sumAngle;
meanPointY /= sumAngle;

// Armazenando o ponto da ponta de dedo
_vetEndPoints.push_back( Point2f(float(meanPointX), float(meanPointY) ) );

endIndex = i;

// Adapta-se a uma elipse em torno de um conjunto de pontos 2D
_ellipses.push_back( fitEllipse(vector<Point>(contour.begin() + startIndex, contour.begin() + endIndex)) );

// Encontre o ponto final ao longo do primeiro eixo
double minDist = 1000000;
for (int dAngle = 90; dAngle < 360; dAngle += 180) {

    // Pegando a ultima elipse
    RotatedRect ellipse = _ellipses[ _ellipses.size() - 1];

    // Coordenadas do centro da elipse
    double x1 = ellipse.center.x;
    double y1 = ellipse.center.y;

    if (dAngle == 0 || dAngle == 180)
    {
        x1 += cos( (ellipse.angle + dAngle) * CV_PI / 180 ) *
              ellipse.size.width * 0.5;
        y1 += sin( (ellipse.angle + dAngle) * CV_PI / 180 ) *
              ellipse.size.width * 0.5;
    }
    else
    {
        x1 += cos( (ellipse.angle + dAngle) * CV_PI / 180 ) *
              ellipse.size.height * 0.5;
        y1 += sin( (ellipse.angle + dAngle) * CV_PI / 180 ) *
              ellipse.size.height * 0.5;
    }
    double dist = pow(meanPointX - x1, 2) + pow(meanPointY - y1, 2);

    if (minDist > dist) minDist = dist;
}

_vetEndPointDist.push_back(10);
_vetEndPointScore.push_back(1);

if (int(_vetEndPoints.size()) == MAX_END_POINT)
    break;

}
}

if (nPointsGap >= GAP_POINT_THRESHOLD)
{

```

```

    nPointsConnected = 0;
    minPointIndex = -1;
    minAngle = 0;
    mediumIndex = 0;
    meanPointX = 0;
    meanPointY = 0;
    sumAngle = 0;
}
}

return int(_vetEndPoints.size());
}

```

A.4 Módulo *FingertipTracker*

```

#pragma once

#include <opencv2\opencv.hpp>
#include <algorithm>
#include <fstream>

#include "FingertipCandidate.h"
#include "FingerTip.h"

#include "ArException.h"

using namespace cv;
using namespace std;

// Numero máximo de candidato a dedo
#define NUM_MAX_CANDIDATES 20
// Distancia maxima entre o candidato
// perdido e um novo candidato para ser procurar padrão
#define THRESHOLD_CLOSEST_DIST 50
// Quantidade de vezes que o ponto pode ser
// perdido e continuar na lista de candidatos
#define THRESHOLD_LOST_TRACKING 5
// Quantidade de vezes que deve ser encontrado
// se manter no vetor de candidatos
#define MIN_AGE_TO_TRACK 5
// Quantidade de vezes que deve ser encontrado
//na imagem para ser detectado
#define THRESHOLD_AGE_TO_DETECT 10

#define NUM_FINGERTIP 5

#define MODE_FINGERTIP_NONE 0
#define MODE_FINGERTIP_FIRST_DETECTED 1
#define MODE_FINGERTIP_TRACKING 2
#define MODE_FINGERTIP_LOST_TRACKING 3

class FingertipTracker
{
public:
    FingertipTracker();
}
```

```

~FingertipTracker();

bool feedFingertipCandidates( vector<Point2f> &points ,
    vector<float> &distValue ,
    Point currCentroid);
bool trackFingertips( vector<Point2f> &points ,
    vector<float> &distValue ,
    Point prevCentroid ,
    Point currCentroid);
bool findExtrinsicCameraParams( Mat &cameraIntrinsic ,
    Mat &cameraDistortion ,
    Mat &fingerRotation ,
    Mat &fingerTranslation);

void reset();

bool loadFingertipCoordinates( string filename);

bool getFlipOrder() { return _fingertipFlipOrder; }
FingerTip * getFingertip( int index);

protected:
    void matchCorrespondencesByNearestNeighbor( vector<Point2f> &points ,
        vector<float> &distValue);

private:
    //
    // Tracking Mode
    //
    int          _nMode;

    //
    // Candidates
    //
    vector<FingertipCandidate> _vetCandidates;

    //
    // Fingertips
    //
    bool          _detected;
    bool          _fingertipFlipOrder;
    vector<FingerTip> _vetFingertip;
    vector<bool>   _vetTracked;

    //
    // Fingertip Coordinates
    //
    bool          _fingertipCoordinates;
    vector< Point3f > _vetFingertipCoordinates;
    float         _numCoordinateSamples;

};

#include "FingertipTracker.h"
#include "Util.h"

```

```

FingertipTracker :: FingertipTracker()
{
}

FingertipTracker ::~ FingertipTracker()
{
}

void FingertipTracker :: reset()
{
    _nMode = MODE_FINGERTIP_NONE;
    _detected = false;

    _vetCandidates . clear();
    _vetFingertip . clear();
}

FingerTip * FingertipTracker :: getFingertip(int index)
{
    if (index >= int(_vetFingertip . size ())) return 0;

    return &(_vetFingertip [index]);
}

<//
// Correspondencia por vizinho mais próximo
//
void FingertipTracker :: matchCorrespondencesByNearestNeighbor(vector<Point2f
    > &points ,
    vector<float> &distValue)
{
    int nPoints = int(points . size ());
    vector<bool> matched(nPoints , false);

    for (int i = 0; i < int(_vetCandidates . size ()); i++)
    {
        //
        // Encontre o ponto mais próximo de cada ponto candidato
        //
        float minDist = THRESHOLD_CLOSEST_DIST;
        int minJ = -1;
        for (int j = 0; j < nPoints; j++)
        {
            float dist = sqrt(
                pow((float)points [j ].x - _vetCandidates [i ]._point.x, 2) +
                pow((float)points [j ].y - _vetCandidates [i ]._point.y, 2));
            if (!matched[j ] && dist < minDist)
            {
                minDist = dist;
                minJ = j ;
            }
        }

        if (minJ >= 0)
        {
            //
        }
    }
}

```

```

// Se existe atualiza a pontuação e a posição
//
matched[minJ] = true;

_vetCandidates[i]._velocity.first = points[minJ].x -
    _vetCandidates[i]._point.x;
_vetCandidates[i]._velocity.second = points[minJ].y -
    _vetCandidates[i]._point.y;

_vetCandidates[i]._point.x = points[minJ].x;
_vetCandidates[i]._point.y = points[minJ].y;

_vetCandidates[i]._dist = distValue[minJ];

_vetCandidates[i]._score++;
_vetCandidates[i]._lost = 0;

}

}

// Percorro todos os pontos
for (int i = 0; i < nPoints; i++)
{
    // Se não foi encontrado o padrão e ainda não tiver no máximo
    // possivel de candidatos
    if (int(_vetCandidates.size()) < NUM_MAX_CANDIDATES && matched[i] ==
        false)
        _vetCandidates.push_back(
            FingertipCandidate(points[i], 0, 0, 0, pair<float , float> (0.0f ,
                0.0f), distValue[i]));
}

}

bool FingertipTracker::feedFingertipCandidates(vector<Point2f> &points ,
                                                vector<float> &distValue ,
                                                Point currCentroid)
{
    //

    // Basicamente incrementa o valor perdido. Quando encontrar, o valor
    // será decrementado
    //
    for (int i = 0; i < int(_vetCandidates.size()); i++)
        _vetCandidates[i]._lost++;

    //

    // Correspondencia por vizinho mais próximo
    //
    matchCorrespondencesByNearestNeighbor(points , distValue);

    //

    // Apago os pontos candidatos por valor perdido e pontuação
    //
    for (int i = 0; i < int(_vetCandidates.size()); i++)
    {
        if (_vetCandidates[i]._lost > THRESHOLD_LOST_TRACKING ||

```

```

_vetCandidates[i]._age > MIN_AGE_TO_TRACK &&
(float)_vetCandidates[i]._score / _vetCandidates[i]._age < 0.5)
{
    // Removo o ponto
    _vetCandidates.erase( _vetCandidates.begin() + i );
    i--;
}
}

//
// Incrementa a idade
//
for (int i = 0; i < int(_vetCandidates.size()); i++)
    _vetCandidates[i]._age++;

//
// Faço o modelo Fingertip para a ponta dos dedos detectada
//
_detected = false;
_vetFingertip.clear();

// Verifica se detectou 5 pontas de dedos
for (int i = 0; i < int(_vetCandidates.size()); i++)
{
    if (_vetCandidates[i]._age > THRESHOLD_AGE_TO_DETECT &&
        _vetCandidates[i]._point.y > currCentroid.y)
    {
        //
        // Detected !
        //
        _vetFingertip.push_back(FingerTip(i, _vetCandidates[i]._point,
                                         _vetCandidates[i]._dist));

        if (int(_vetFingertip.size()) == NUM_FINGERTIP)
        {
            _detected = true;
            break;
        }
    }
}

//
// Quando detectado, atribuir o número do índice da ponta do dedo
//
if (_detected) {

    //Faz a ordenação dos dedos (por x-axis)
    sort(_vetFingertip.begin(), _vetFingertip.end());

    //
    // Determinar o polegar com base na distância até o ponto médio
    //

    // Calcula o ponto médio
    float mean_x = 0;
    float mean_y = 0;
}

```

```

for (auto item : _vetFingertip) {
    mean_x += item._point.x;
    mean_y += item._point.y;
}

mean_x /= float(_vetFingertip.size());
mean_y /= float(_vetFingertip.size());

// Pega o dedo mais distante
float maxDist = 0;
int maxDistIndex = 0;
for (int i = 0; i < int(_vetFingertip.size()); ++i) {

    auto item = _vetFingertip[i];
    float dist = sqrt( pow(mean_x - item._point.x, 2) + pow(mean_y -
        item._point.y, 2) );
    if (dist > maxDist) {
        maxDist = dist;
        maxDistIndex = i;
    }
}

// Se as distâncias são negativas, então inverte os pontos.
_fingertipFlipOrder = (maxDistIndex == 0);

if (_fingertipFlipOrder) reverse(_vetFingertip.begin(),
    _vetFingertip.end());
}

//
// Update Mode
//
if (_detected)
{
    if (_nMode == MODE_FINGERTIP_NONE)
        _nMode = MODE_FINGERTIP_FIRST_DETECTED;
    if (_nMode == MODE_FINGERTIP_LOST_TRACKING)
        _nMode = MODE_FINGERTIP_TRACKING;
}

return _detected;
}

bool FingertipTracker::trackFingertips (vector<Point2f> &points ,
    vector<float> &distValue ,
    Point prevCentroid , Point currCentroid)
{
    try {

        //
        // Init
        //
        _vetTracked.assign(NUM_FINGERTIP, false);

        //
        // Check
        //
        if (int(points.size()) < NUM_FINGERTIP) {

```

```

    reset();
    _fmode = MODE_FINGERTIP_LOST_TRACKING;
    return _detected;
}

// 
// Correspondência de pontos e pontas dos dedos
//
int index[5] = { 0 };
int minIndex[5] = { 0 };
float dist[5] = { 0 };
float minCost = 1000000;
int nPoints = int(points.size());

for (int pt0 = 0; pt0 < nPoints; pt0++) // first
{
    fingertip
    {
        index[0] = pt0 % nPoints;
        dist[0] = sqrt(
            pow((float)points[index[0]].x - currCentroid.x - _vetFingertip
                [0]._point.x + prevCentroid.x, 2) +
            pow((float)points[index[0]].y - currCentroid.y - _vetFingertip
                [0]._point.y + prevCentroid.y, 2));

        for (int pt1 = 1; pt1 <= nPoints - 4; pt1++) // second
            fingertip
            {
                index[1] = _fingertipFlipOrder ? Util::mod(pt0 - pt1, nPoints)
                    : (pt0 + pt1) % nPoints;
                dist[1] = sqrt(
                    pow((float)points[index[1]].x - currCentroid.x -
                        _vetFingertip[1]._point.x + prevCentroid.x, 2) +
                    pow((float)points[index[1]].y - currCentroid.y -
                        _vetFingertip[1]._point.y + prevCentroid.y, 2));

                for (int pt2 = 1; pt2 <= nPoints - pt1 - 3; pt2++) // third
                    fingertip
                    {
                        index[2] = _fingertipFlipOrder ? Util::mod(pt0 - pt1 - pt2, nPoints)
                            : (pt0 + pt1 + pt2) % nPoints;
                        dist[2] = sqrt(
                            pow((float)points[index[2]].x - currCentroid.x -
                                _vetFingertip[2]._point.x + prevCentroid.x, 2) +
                            pow((float)points[index[2]].y - currCentroid.y -
                                _vetFingertip[2]._point.y + prevCentroid.y, 2));

                        for (int pt3 = 1; pt3 <= nPoints - pt1 - pt2 - 2; pt3++) // fourth
                            fingertip
                            {
                                index[3] = _fingertipFlipOrder ? Util::mod(pt0 - pt1 -
                                    pt2 - pt3, nPoints) : (pt0 + pt1 + pt2 + pt3) %
                                    nPoints;
                                dist[3] = sqrt(
                                    pow((float)points[index[3]].x - currCentroid.x -
                                        _vetFingertip[3]._point.x + prevCentroid.x, 2) +
                                    pow((float)points[index[3]].y - currCentroid.y -
                                        _vetFingertip[3]._point.y + prevCentroid.y, 2));
                            }
                    }
                }
}

```

```

        for (int pt4 = 1; pt4 <= nPoints - pt1 - pt2 - pt3 - 1;
             pt4++)// fifth fingertip
    {
        index[4] = _fingertipFlipOrder ? Util::mod(pt0 - pt1 -
            pt2 - pt3 - pt4, nPoints) : (pt0 + pt1 + pt2 + pt3
            + pt4) % nPoints;
        dist[4] = sqrt(
            pow((float)points[index[4]].x - currCentroid.x -
                _vetFingertip[4]._point.x + prevCentroid.x, 2) +
            pow((float)points[index[4]].y - currCentroid.y -
                _vetFingertip[4]._point.y + prevCentroid.y, 2));
        // Compute the matching cost
        // float matchingCost = 0;
        for (int i = 0; i < 5; i++)
        {
            matchingCost += dist[i];
        }

        // Find minimum cost matching
        //
        if (matchingCost < minCost)
        {
            minCost = matchingCost;
            for (int i = 0; i < 5; i++)
            {
                minIndex[i] = index[i];
            }
        }
    }

}

// Update Tracking
//
for (int i = 0; i < 5; i++)
{
    _vetFingertip[i]._point.x = points[minIndex[i]].x;
    _vetFingertip[i]._point.y = points[minIndex[i]].y;
    _vetFingertip[i]._dist = distValue[minIndex[i]];

    _vetTracked[i] = true;
}

_nMode = MODE_FINGERTIP_TRACKING;
}
catch (Exception &e) {
    throw ArException("FingertipTracker", "trackFingertips", e);
}
return _detected;
}

```

```

bool FingertipTracker::findExtrinsicCameraParams(Mat &cameraIntrinsic ,
    Mat &cameraDistortion ,
    Mat &fingerRotation ,
    Mat &fingerTranslation )
{
    bool poseEstimated = false ;

    try {

        //
// Check availability
//
        if (int(_vetFingertip.size()) != NUM_FINGERTIP ||  

            _fingertipCoordinates == false)
            return false ;

        // Preparando o vector com os pontos
        vector< Point2f > ptos;
        for (auto item : _vetFingertip)
            ptos.push_back(item._point);

        //
// Call OpenCV function
//

        // Essa função entroca 6 valores, 3 de rotação e 3 de tradução, Dado
um modelo 2d uma imagem.
        poseEstimated = solvePnP(_vetFingertipCoordinates ,
            ptos ,
            cameraIntrinsic ,
            cameraDistortion ,
            fingerRotation ,
            fingerTranslation ,
            false ,
            CV_ITERATIVE) ;

        ptos.clear();

    }
    catch (Exception const &e) {
        throw ArException("FingertipTracker", "findExtrinsicCameraParams", e)
        ;
    }

    return poseEstimated;
}

bool FingertipTracker::loadFingertipCoordinates(string filename)
{
    //
// Carrega as coordenadas das pontas dos dedos
//
    fstream file(filename.c_str(), fstream::in | fstream::out);

    if (!file.is_open())
        return false;
}

```

```

_vetFingertipCoordinates . assign (NUM_FINGERTIP, Point3d () );

for (int i = 0; i < NUM_FINGERTIP; i++)
{
    file >> _vetFingertipCoordinates [i].x;
    file >> _vetFingertipCoordinates [i].y;
    file >> _vetFingertipCoordinates [i].z;
}

file . close ();

_fingertipCoordinates = true ;

return _fingertipCoordinates ;
}

```

A.5 Módulo *PoseEstimation*

```

#pragma once

#include <iostream>
#include <fstream>
#include <vector>
#include <opencv2\opencv.hpp>

using namespace std;
using namespace cv;

class PoseEstimation
{
public:
    PoseEstimation();

    bool readIntrinsicParameters ( string fileName);

public:
    //
    // Intrinsic Camera Parameters
    //
    bool _intrinsic;
    Mat _cameraIntrinsic;
    Mat _cameraDistortion;

};

#include "PoseEstimation.h"

#define atd at<double>

PoseEstimation :: PoseEstimation ()
{
    _cameraIntrinsic = Mat::zeros (3, 3, DataType<double>::type);
    _cameraDistortion = Mat::zeros (4, 1, DataType<double>::type);
}

bool PoseEstimation :: readIntrinsicParameters ( string fileName)
{
    fstream file (fileName . c_str (), fstream :: in | fstream :: out);

```

```

if (!file.is_open())
    return false;

_cameraIntrinsic.atd(2, 2) = 1;

file >> _cameraIntrinsic.atd(0, 0);
file >> _cameraIntrinsic.atd(1, 1);
file >> _cameraIntrinsic.atd(0, 2);
file >> _cameraIntrinsic.atd(1, 2);

for (int i = 0; i < 4; ++i)
    file >> _cameraDistortion.atd(i);
file.close();

_intrinsic = true;

return _intrinsic;
}

```

A.6 Módulo *FingertipPoseEstimation*

```

#pragma once

#include <opencv2\opencv.hpp>
#include <GL\glut.h>

#include <iomanip>

#include "Quaternion.h"

#include "HandRegion.h"
#include "FingertipProcess.h"
#include "FingertipTracker.h"
#include "PoseEstimation.h"

using namespace cv;

class FingertipPoseEstimation
{
public:
    FingertipPoseEstimation();
    ~FingertipPoseEstimation();

    bool initialize(string calibFilename, Size size);
    void terminate();
    void reset();

    void OnCapture(Mat frame);
    void OnProcess();
    void OnDisplay();

    bool loadFingertipCoordinates(string filename);

    // Interfaçao da openCL para OpenGL
    void setOpenGLFrustum(); // Frustum de visualização
    void setOpenGLModelView(); // Posição do observador
}

```

```

private:
    //
    // Tamanho das imagens que serÃ£o processadas
    //
    Size _size;

    //
    // Imagens
    //
    Mat _image;
    Mat _grayImage;

    //
    // Modulos de processamento
    //
    HandRegion _handRegion;
    FingertipProcess _fingerTipProcess;
    FingertipTracker _fingertipTracker;
    PoseEstimation _poseEstimation;

    //
    // Centro da mÃ©dia
    //
    Point _prevCentroid;
    Point _currCentroid;

    //
    // Running Mode
    //
    bool _fingertipDetected;

    //
    // Pose Estimation by Fingertips
    //
    bool _poseEstimatedByFingertips;
    bool _validPose;
    Mat _matFingerRotation;
    Mat _matFingerRotation3by3;
    Mat _matFingerTranslation;
    Mat _matFingerQuaternion;

    //
    // Camera Transform
    //
    Mat _matCameraCenterT;
    Mat _matCameraCenterR;

    //
    // OpenGL View Frustum
    //
    GLuint _nCameraTexID;

    //
    // OpenGL ModelView Matrix
    //
    float _modelView[16];

```

```

// Kalman filter for the camera pose
//  $x(t+1) = A x(t) + G w(t)$ 
//  $y(t) = H x(t) + v(t)$ 
//
KalmanFilter *_kalmanFilter;
Mat _pKalmanMat_A; // State Transition Matrix
Mat _pKalmanMat_G; // Driving Matrix
Mat _pKalmanMat_H; // Observation Matrix
Mat _pKalmanMat_W; // Process Noise Matrix

//
// Processing Time Profile
//
fstream _fpTime;
int64 _PrevTickCount;

// Metodos privados
void zerosMatCameraParam();
};

#include "FingertipPoseEstimation.h"

#define atd at<double>

FingertipPoseEstimation::FingertipPoseEstimation()
{
    _matFingerRotation = Mat::zeros(3, 1, DataType<double>::type);
    _matFingerTranslation = Mat::zeros(3, 1, DataType<double>::type);
    _matFingerRotation3by3 = Mat::zeros(3, 3, DataType<double>::type);
    _matFingerQuaternion = Mat::zeros(4, 1, DataType<double>::type);

    _matCameraCenterT = Mat::zeros(3, 1, DataType<double>::type);
    _matCameraCenterR = Mat::zeros(3, 3, DataType<double>::type);

    _kalmanFilter = 0;

    _fpTime.open("C:\\\\Users\\\\asus\\\\Desktop\\\\time.txt", fstream::app);
}

bool FingertipPoseEstimation::initialize(string calibFilename, Size size)
{
    // Armazenando o size
    _size = size;

    // Inicializando os modulos
    _fingerTipProcess.initialize(_size);

    if (_poseEstimation.readIntrinsicParameters(calibFilename) == false)
        return false;

    //
    // Texture for Video Frame
    //
}

```

```

glGenTextures(1, &_nCameraTexID);
glBindTexture(GL_TEXTURE_2D, _nCameraTexID);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
// Set texture clamping method
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP); //MY
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP); //MY
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, 1024, 1024, 0, GL_RGB,
             GL_UNSIGNED_BYTE, 0);

//
// Kalman filter
//
_kalmanFilter = 0;

_pKalmanMat_A = Mat::zeros(14, 14, DataType<double>::type);
setIdentity(_pKalmanMat_A);
for (int i = 0; i < 7; i++)
    _pKalmanMat_A.atd(i, i + 7) = 1;

_pKalmanMat_G = Mat::zeros(14, 7, DataType<double>::type);
for (int i = 0; i < 7; i++)
    _pKalmanMat_A.atd(i + 7, i) = 1;

_pKalmanMat_H = Mat::zeros(7, 14, DataType<double>::type);
for (int i = 0; i < 7; i++)
    _pKalmanMat_A.atd(i, i) = 1;

_pKalmanMat_W = Mat::zeros(14, 1, DataType<double>::type);

return true;
}

void FingertipPoseEstimation::terminate()
{
    if (_kalmanFilter) {
        delete _kalmanFilter;
        _kalmanFilter = 0;
    }
}

void FingertipPoseEstimation::reset()
{
    _fingertipDetected = false;
    _fingertipTracker.reset();

    // Reset Kalman filter
    if (_kalmanFilter)
    {
        delete _kalmanFilter;
        _kalmanFilter = 0;
    }
}

void FingertipPoseEstimation::OnCapture(Mat frame)

```

```

{
    _image = frame;
}

void FingertipPoseEstimation :: OnProcess()
{
    // Segmentação
    Mat handImg = _handRegion.getHandRegion(_image);

    //
    // Find Fingertip Candidates
    //
    if (_fingertipDetected == false)
        _fingerTipProcess.reset(); // Limpa os vetores

    // Retorna a quantidade de pontos de dedos encontradas
    // (Encontra no máximo 5 candidatos de ser ponta de dedo)
    int nFingertipCandidates = _fingerTipProcess.
        findFingerTipCandidatesByCurvature(handImg);

    //
    // Pega Centroid
    //
    _prevCentroid = _currCentroid;
    _currCentroid = _fingerTipProcess._maxDistPoint;

    //
    // Detectar / rastrear ponto de dedos
    //
    if (_fingertipDetected == false)
    {

        // Ainda não foi detectado. Tenta detectar 5 pontas nos dedos.
        // Detecta aos pontos dos dedos
        _fingertipDetected =
            _fingertipTracker.feedFingertipCandidates(_fingerTipProcess.
                _vetEndPoints,
                _fingerTipProcess._vetEndPointDist,
                _currCentroid);

    }
    else
    {
        // Detected.

        // Predict by Kalman filter
        if (_kalmanFilter)
            Mat prediction = _kalmanFilter->predict();

        // Rastreamento dos dedos
        _fingertipDetected = _fingertipTracker.trackFingertips(
            _fingerTipProcess._vetEndPoints,
            _fingerTipProcess._vetEndPointDist,
            _prevCentroid, _currCentroid);
    }
}

```

```

// Zerando as matrizes: _matFingerRotation , _matFingerTranslation e
// _matFingerQuaternion
if (_fingertipDetected == false)
    zerosMatCameraParam();

//
// Calcula a posição estimada para os dedos – Pose Estimation from
// Fingertips
//
bool fPrevPoseEstimated = _poseEstimatedByFingertips;
bool fForceOrientation = false;
_poseEstimatedByFingertips = false;

if (_fingertipDetected)
{
    // Armazena os parâmetros atuais da camera
    float rotation_old[3];
    float translation_old[3];
    float quaternion_old[4];
    float center_old[3];

    for (int i = 0; i < 3; i++)
    {
        rotation_old[i] = _matFingerRotation.atd(i);
        translation_old[i] = _matFingerTranslation.atd(i);
        quaternion_old[i] = _matFingerQuaternion.atd(i);
        center_old[i] = _matCameraCenterT.atd(i);
    }
    quaternion_old[3] = _matFingerQuaternion.atd(3);

    // Zerando as matrizes: _matFingerRotation , _matFingerTranslation e
    // _matFingerQuaternion
    zerosMatCameraParam();

    // cout << _matFingerRotation << endl; DEBUG

    // Busca a rotação e o transalate da mão
    _poseEstimatedByFingertips = _fingertipTracker.
        findExtrinsicCameraParams(_poseEstimation._cameraIntrinsic ,
        _poseEstimation._cameraDistortion ,
        _matFingerRotation ,
        _matFingerTranslation);

    //
    // Check Camera Center Range
    //
    Rodrigues(_matFingerRotation , _matFingerRotation3by3);

    // Converte matriz para quaternion
    Quaternion::matrix2Quaternion(_matFingerRotation3by3 ,
        _matFingerQuaternion);

    // Encontra o inverso ou pseudo-inverso de uma matriz.
    invert(_matFingerRotation3by3 , _matCameraCenterR);
    _matCameraCenterT = _matCameraCenterR * _matFingerTranslation;
}

```

```

// Alterando a escala
_matCameraCenterT.convertTo(_matCameraCenterT, -1, -1);
cout << "\n\n_CameraCenterTMat cvScale\n" << _matCameraCenterT <<
endl;

// Calcula a diferen a de Quaternion
float diffQuaternion = 0;
for (int i = 0; i < 4; i++)
    diffQuaternion += pow(_matFingerQuaternion.atd(i) - quaternion_old
        [i], 2);
diffQuaternion = sqrt(diffQuaternion);

// Verifica se a camera  o valida
if (abs(_matCameraCenterT.atd(2)) > 2000.0 ||
    (_fingertipTracker.getFlipOrder() == false && _matCameraCenterT.
        atd(2) > 0) ||
    (_fingertipTracker.getFlipOrder() == true && _matCameraCenterT.atd
        (2) < 0) ||
    (center_old[0] || center_old[1] || center_old[2]) &&
    sqrt(pow(_matCameraCenterT.atd(0) - center_old[0], 2) +
    pow(_matCameraCenterT.atd(1) - center_old[1], 2) +
    pow(_matCameraCenterT.atd(2) - center_old[2], 2)) > 200)
{
    printf("Invalid Camera (by Fingertip) (%5.2f, %5.2f, %5.2f)\n",
        _matCameraCenterT.atd(0), _matCameraCenterT.atd(1),
        _matCameraCenterT.atd(2));
    _poseEstimatedByFingertips = false;
}
else
    printf("OK Camera (by Fingertip) (%5.2f, %5.2f, %5.2f) diffQ =
        %5.2f\n", _matCameraCenterT.atd(0), _matCameraCenterT.atd(1),
        _matCameraCenterT.atd(2), diffQuaternion);

printf("T(%5.2f %5.2f %5.2f) Q(%5.2f %5.2f %5.2f %5.2f)\n",
    _matFingerTranslation.atd(0), _matFingerTranslation.atd(1),
    _matFingerTranslation.atd(2),
    _matFingerQuaternion.atd(0), _matFingerQuaternion.atd(1),
    _matFingerQuaternion.atd(2), _matFingerQuaternion.atd(3));

// Quando a orienta o muda completamente (por exemplo, -0,97 ->
// 0,98),
// for sar a orienta o sem filtragem do kalman
if ((center_old[0] || center_old[1] || center_old[2]) &&
    diffQuaternion > 1.0 && _kalmanFilter)
{
    fForceOrientation = true;
    for (int i = 0; i < 4; i++)
    {
        _kalmanFilter->statePre.at<float>(i + 3) = _matFingerQuaternion
            .atd(i);
        _kalmanFilter->statePre.at<float>(i + 10) = 0;
    }
}

// Voltando os par metros antigo
if (_poseEstimatedByFingertips == false)

```

```

{
{
    for (int i = 0; i < 3; i++)
    {
        _matFingerRotation.atd(i) = rotation_old[i];
        _matFingerTranslation.atd(i) = translation_old[i];
        _matFingerQuaternion.atd(i) = quaternion_old[i];
    }
    _matFingerQuaternion.atd(3) = quaternion_old[3];
    /* */
    printf("forced to be old pose.\n");
}
}
else
{
    //
    // Init Kalman filter if necessary
    if (!_kalmanFilter)
    {
        _kalmanFilter = new KalmanFilter(14, 7, 0);

        for (int i = 0; i < 7; i++)
            _pKalmanMat_A.atd(i, i + 7) = 1;

        // Setando a matriz _pKalmanMat_A para a transitionMatrix
        _pKalmanMat_A.copyTo(_kalmanFilter->transitionMatrix);

        // Setando identidade
        setIdentity(_kalmanFilter->measurementMatrix, cvRealScalar(1));

        // noise covariances
        _kalmanFilter->processNoiseCov.setTo(Scalar::all(0));
        _kalmanFilter->measurementNoiseCov.setTo(Scalar::all(0));

        for (int i = 0; i < 3; i++)
        {
            // translation part
            _kalmanFilter->processNoiseCov.at<float>(i, i) = 10;
            _kalmanFilter->processNoiseCov.at<float>(i + 7, i + 7) = 10;
            _kalmanFilter->measurementNoiseCov.at<float>(i, i) = 30;
        }
        for (int i = 0; i < 4; i++)
        {
            // rotation part
            _kalmanFilter->processNoiseCov.at<float>(i + 3, i + 3) =
                0.2;
            _kalmanFilter->processNoiseCov.at<float>(i + 10, i + 10) =
                0.2;
            _kalmanFilter->measurementNoiseCov.at<float>(i + 3, i + 3) =
                0.5;
        }

        // Setando identidade
        setIdentity(_kalmanFilter->errorCovPost, cvRealScalar(1));

        // Zerando statePost
        _kalmanFilter->statePost.setTo(Scalar::all(0));
    }
}

```

```

for (int i = 0; i < 3; i++)
    _kalmanFilter->statePost.at<float>(i) =
        _matFingerTranslation.atd(i);

for (int i = 0; i < 4; i++)
    _kalmanFilter->statePost.at<float>(i + 3) =
        _matFingerQuaternion.atd(i);
}

else
{
    // Kalman correct
    float measurement[7];
    Mat measurementMat(7, 1, CV_32FC1, measurement);

    // Salvando o translation
    for (int i = 0; i < 3; i++)
        measurement[i] = _matFingerTranslation.atd(i);

    // Salvando o Quaternion
    for (int i = 0; i < 4; i++)
        measurement[i + 3] = _matFingerQuaternion.atd(i);

    // update A's delta T
    //for (int i = 0; i < 7; i++)
    //    cvSetReal2D(_pKalmanMat_A, i, i + 7, _deltaT);

    // Copiando
    _pKalmanMat_A.copyTo(_kalmanFilter->transitionMatrix);
    //cvCopy(_pKalmanMat_A, _pKalman->transition_matrix);

    // correct kalman filter state
    _kalmanFilter->correct(measurementMat);
}

// Copy the corrected state
for (int i = 0; i < 3; i++)
    _matFingerTranslation.atd(i) = _kalmanFilter->statePost.at<float>(i);

for (int i = 0; i < 4; i++)
    _matFingerQuaternion.atd(i) = _kalmanFilter->statePost.at<float>(i + 3);
}

else
{
    // Reset Kalman filter
    if (_kalmanFilter)
    {
        delete _kalmanFilter;
        _kalmanFilter = 0;
    }
}

_validPose = _poseEstimatedByFingertips || (_fingertipDetected &&
    fPrevPoseEstimated);

```

```

printf("pose: %s\n", _validPose ? "valid" : "invalid");

//
// Compute Camera Center
//

printf("T(%5.2f %5.2f %5.2f) Q(%5.2f %5.2f %5.2f %5.2f)\n",
       _matFingerTranslation.atd(0), _matFingerTranslation.atd(1),
       _matFingerTranslation.atd(2), _matFingerQuaternion.atd(0),
       _matFingerQuaternion.atd(1), _matFingerQuaternion.atd(2),
       _matFingerQuaternion.atd(3));

// Converte matiz para quaternion (!)
Quaternion::quaternion2Matrix(_matFingerQuaternion,
                               _matFingerRotation3by3);

// Converte uma matriz de rotação em um vetor de rotação ou vice
// -versa.
Rodrigues(_matFingerRotation3by3, _matFingerRotation);

invert(_matFingerRotation3by3, _matCameraCenterR);

// _cameraCenterT = _cameraCenterR * _vetFingerTranslation
_matCameraCenterT = _matCameraCenterR * _matFingerTranslation;

// Alterando a escala
_matCameraCenterT.convertTo(_matCameraCenterT, -1, -1);

// Mostrando os parâmetros da camera
printf("Camera(by Fingertips) = (%5.2f, %5.2f, %5.2f)\n",
       _matCameraCenterT.atd(0), _matCameraCenterT.atd(1), _matCameraCenterT
       .atd(2));
}

//
// Faz desenhos usando a posição da mão e dos dedos
//
void FingertipPoseEstimation::OnDisplay()
{
    // Se os dedos não foi detectado
    if (_image.empty())
        return;

    //
    // Fingertips ( Candidates )
    //
    Scalar color[] = {
        { Scalar( 64,      64,      255 ) },    // red
        { Scalar( 0,      128,      255 ) },    // orange
        { Scalar( 0,      255,      255 ) },    // yellow
        { Scalar( 0,      192,       0 ) },    // green
        { Scalar( 255,      0,       0 ) }     // blue
    };

    //
    // Desenhando círculos nas pontas dos dedos
    for (int i = 0; i < NUM_FINGERTIP; i++)
    {
        FingerTip *fingerTip = _fingertipTracker.getFingertip(i);

```

```

    if (fingerTip)
        circle(_image, fingerTip->_point, fingerTip->_dist, color[i], -1);
}

// Desenhando as elipses no dedo
for (int i = 0; i < int(_fingerTipProcess._vetEndPoints.size()); i++)
{
    Point2f center = _fingerTipProcess._ellipses[i].center;
    Size2f size = cvSize(_fingerTipProcess._ellipses[i].size.width * 0.5,
                        _fingerTipProcess._ellipses[i].size.height * 0.5);

    if (size.width > 0 && size.height > 0 && size.width < _size.width &&
        size.height < _size.height)
        // ellipse(_image, _fingerTipProcess._ellipses[i], Scalar(0, 255,
        // 0), 1, CV_AA);
        ellipse(_image, center, size,
                _fingerTipProcess._ellipses[i].angle,
                0, 360, Scalar(0, 255, 0));
}

//
// Max Distance Point + Region of Interest
//
circle(_image, _fingerTipProcess._maxDistPoint,
       _fingerTipProcess._maxDistValue * 0.7,
       Scalar(255, 255, 0), 1, 8, 0);

if (_validPose)
    putText(_image, "OK", Point(100, 300), FONT_HERSHEY_DUPLEX, 1, Scalar
(0, 0, 255));

}

//MÃ©todo original da HandyAR
void FingertipPoseEstimation::setOpenGLFrustrum()
{
    if (_image.empty())
        return;

    // set viewing frustrum to match camera FOV (ref: ARTag's 3
    // d_augmentations.cpp)
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    double camera_fx = _poseEstimation._cameraIntrinsic.atd(0, 0);
    double camera_fy = _poseEstimation._cameraIntrinsic.atd(1, 1);
    double camera_ox = _poseEstimation._cameraIntrinsic.atd(0, 2);
    double camera_oy = _poseEstimation._cameraIntrinsic.atd(1, 2);
    double camera_opengl_dRight = (double)( _size.width - camera_ox) / (
        double)(camera_fx);
    double camera_opengl_dLeft = -camera_ox / camera_fx;
    double camera_opengl_dTop = (double)( _size.height - camera_oy) / (
        double)(camera_fy);
    double camera_opengl_dBottom = -camera_oy / camera_fy;

    // if (_validPose)
        glFrustum(camera_opengl_dLeft, camera_opengl_dRight,

```

```

    camera_opengl_dBottom , camera_opengl_dTop , 1.0 , 102500.0);

camera_opengl_dLeft *= 10240;
camera_opengl_dRight *= 10240;
camera_opengl_dBottom *= 10240;
camera_opengl_dTop *= 10240;

// convert image R and B channel
cvtColor(_image , _image , CV_RGB2BGR);

// draw the camera frame
glBindTexture(GL_TEXTURE_2D, _nCameraTexID);
glTexSubImage2D(GL_TEXTURE_2D, 0, 0, 0, _size.width, _size.height,
    GL_RGB, GL_UNSIGNED_BYTE, _image.data);

glClearColor(0, 0, 0, 0);

// draw a quad for the background video
glMatrixMode(GL_TEXTURE);
glLoadIdentity();
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glDisable(GL_LIGHTING);

glEnable(GL_TEXTURE_2D);
glBindTexture(GL_TEXTURE_2D, _nCameraTexID);

glBegin(GL_QUADS);
glColor3f(1.0f, 1.0f, 1.0f);
//draw camera texture, set with offset to aim only at cam_width x
cam_height upper left bit
glTexCoord2f(0.0f, 0.0f);
glVertex3f(camera_opengl_dLeft, camera_opengl_dBottom, -10240);
glTexCoord2f(0.0f, (_float)_size.height / 1024.0);
glVertex3f(camera_opengl_dLeft, camera_opengl_dTop, -10240);
glTexCoord2f((_float)_size.width / 1024.0, (_float)_size.height / 1024.0);
glVertex3f(camera_opengl_dRight, camera_opengl_dTop, -10240);
glTexCoord2f((_float)_size.width / 1024.0, 0.0f);
glVertex3f(camera_opengl_dRight, camera_opengl_dBottom, -10240);
glEnd();

glDisable(GL_TEXTURE_2D);
glEnable(GL_LIGHTING);
}

//MÃ©todo original da HandyAR
void FingertipPoseEstimation::setOpenGLModelView()
{
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();

    gluLookAt(
        _matCameraCenterT.atd(0),
        _matCameraCenterT.atd(1),
        -_matCameraCenterT.atd(2),
        _matCameraCenterT.atd(0) + _matFingerRotation3by3.atd(2, 0),
        _matCameraCenterT.atd(1) + _matFingerRotation3by3.atd(2, 1),
        -(_matCameraCenterT.atd(2) + _matFingerRotation3by3.atd(2, 2)),

```

```
_matFingerRotation3by3.atd(1, 0),
_matFingerRotation3by3.atd(1, 1),
-_matFingerRotation3by3.atd(1, 2));

glGetFloatv(GL_MODELVIEW_MATRIX, _modelView);
}

bool FingertipPoseEstimation::loadFingertipCoordinates(string filename)
{
    return _fingertipTracker.loadFingertipCoordinates(filename);
}

void FingertipPoseEstimation::zerosMatCameraParam()
{
    _matFingerRotation.setTo(Scalar::all(0));
    _matFingerTranslation.setTo(Scalar::all(0));
    _matFingerQuaternion.setTo(Scalar::all(0));
}
```

ANEXO A – Arquivo utilizados pelo sistema**A.1 Arquivo de calibração da câmera**

664.006287 673.420898 293.043488 247.477997 0.254980 -0.569536 0.013796 -0.011997

A.2 Arquivo com a posição das pontas dos dedos

-80.153412 51.177849 0.000000 -50.652924 89.900345 0.000000 0.000000 108.495193
0.000000 51.680298 85.897430 0.000000 99.296082 0.000000 0.000000