UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE CIÊNCIA DA COMPUTAÇÃO – BACHARELADO

CORPUSVR: UM PROTÓTIPO PARA VISUALIZAÇÃO DA ANATOMIA DO CORPO HUMANO

DANILO CONRADO ESKELSEN JUNIOR

BLUMENAU
2018

DANILO CONRADO ESKELSEN JUNIOR

CORPUSVR: UM PROTÓTIPO PARA VISUALIZAÇÃO DA ANATOMIA DO CORPO HUMANO

Trabalho de Conclusão de Curso apresentado ao curso de graduação em Ciência da Computação do Centro de Ciências Exatas e Naturais da Universidade Regional de Blumenau como requisito parcial para a obtenção do grau de Bacharel em Ciência da Computação.

Prof. Aurélio Faustino Hoppe, Mestre - Orientador

BLUMENAU
2018

CORPUSVR: UM PROTÓTIPO PARA VISUALIZAÇÃO DA ANATOMIA DO CORPO HUMANO

Por

DANILO CONRADO ESKELSEN JUNIOR

Trabalho de Conclusão de Curso aprovado para obtenção dos créditos na disciplina de Trabalho de Conclusão de Curso II pela banca examinadora formada por:

Presidente:	Prof. Aurélio Faustino Hoppe, Mestre – Orientador, FURB
Membro:	Prof. Dalton Solano dos Reis, Mestre – FURB
Membro:	Prof. Mauro Marcelo Mattos, Doutor – FURB

Blumenau, 12 de julho de 2018.

Dedico este trabalho a todos os professores do curso, que foram de suma importância na minha vida acadêmica e no desenvolvimento desta monografia.

AGRADECIMENTOS

Aos meus professores, que me guiaram durante esta jornada.

À minha família, que sempre me apoiou e incentivou.

Ao meu orientador em especial, pelo apoio e dedicação durante o desenvolvimento deste trabalho.

You don't have to be great to start, but you have to start to be great.

Zig Ziglar

RESUMO

Este trabalho apresenta o desenvolvimento de um protótipo para visualização da anatomia do sistema esquelético da mão em três dimensões. O protótipo obtém as imagens através da câmera de um dispositivo móvel e a partir disso, as imagens são processadas em tempo real por uma *thread*. Durante a execução da *thread*, a imagem obtida é segmentada através da cor da pele. O resultado da segmentação forma uma imagem binária contendo o maior contorno. Em seguida, é gerada uma lista de pontos convexos do contorno escolhido. Esta lista de pontos convexos é filtrada e em seguida ordenada. Com base na lista de pontos convexos ordenadas juntamente com um algoritmo de Perspective-*n*-Point é estimada a pose da mão. Na sequência, são calculados os ângulos de rotação dos dedos. Depois, são atualizadas as informações da pose da mão e os ângulos de rotação dos dedos em um objeto compartilhado. Este objeto compartilhado é utilizado por outra *thread*, a de renderização do modelo esquelético 3D da mão. Durante a execução desta *thread*, o modelo 3D é desenhado sobre a imagem obtida pela câmera quando a pose da mão é identificada. Por fim, o protótipo foi capaz de identificar e estimar a pose da mão em ambientes controlados com iluminação boa ou moderada, permitindo assim a visualização em tempo real do sistema esquelético de diferentes perspectivas. Tudo isso mantendo uma taxa superior a 15 FPS na maioria dos casos.

Palavras-chave: Sistema esquelético. Visão computacional. Perspective-n-Point. Renderização.

ABSTRACT

This work presents the development of a prototype for visualizing the anatomy of the skeletal system of the hand in three dimensions. The prototype gets the images through the camera of a mobile device and from there, the images are processed in real time by a thread. During the execution of this thread, the obtained image is segmented by the color of the skin. The segmentation result forms a binary image containing the largest contour. Next, a list of convex points of the chosen contour is generated. This list of convex points is filtered and then sorted. Using the ordered list of convex points together with a Perspective-n-Point algorithm, the hand pose is estimated. Next, the rotation angles of the fingers are calculated. Then, the hand pose and the rotation angles of the fingers are updated on a shared object. This shared object is then used by another thread, the rendering of the 3D skeletal model of the hand thread. During the execution of this thread, the 3D model is drawn on top of the image obtained by the camera when the hand pose is identified. Finally, the prototype was able to identify and estimate the hand pose in controlled environments with good or moderate illumination, thus allowing real-time visualization of the skeletal system from different perspectives. All of this while maintaining a rate higher than 15 FPS in most cases.

Key-words: Skeletal system. Computer vision. Perspective-n-point. Rendering.

LISTA DE FIGURAS

Figura 1 – Anatomia do corpo humano 16
Figura 2 – Regiões da mão 18
Figura 3 – Transformação de um vértice 19
Figura 4 – Matriz de translação 19
Figura 5 – Matriz identidade 20
Figura 6 – Matriz de escala 20
Figura 7 – Matrizes de rotação 21
Figura 8 – Câmera matrix 22
Figura 9 – Matriz de parâmetros intrínsecos da câmera 22
Figura 10 – Parâmetros extrínsecos da câmera 23
Figura 11 – Modelo de câmera pinhole 23
Figura 12 – Aplicação Virtuali-Tee 25
Figura 13 – Imagens da anatomia humana 26
Figura 14 – Aplicação Anatomy 4D 27
Figura 15 – Principais telas do Arloon Anatomy 28
Figura 16 – Telas de RA da aplicação 28
Figura 17 – Diagrama de Atividades 31
Figura 18 – Layers do aplicativo 32
Figura 19 – Transformação RGB para Binária 34
Figura 20 – Pontos convexos de um contorno 36
Figura 21 – Triângulos de convexidade 37
Figura 22 – Ordenação dos pontos de defeitos de convexidade 37
Figura 23 – Ângulo interno 38
Figura 24 – Triângulos de convexidade filtrados 40
Figura 25 – Três pontos mais próximos da mão 40
Figura 26 – Localizando o polegar 43
Figura 27 – Lista de pontos de referência 46
Figura 28 – Ângulos de rotação padrão 48
Figura 29 – Métodos abstratos 49
Figura 30 – Modelo 3D da mão 52
Figura 31 – Classificação dos objetos 53

Figura 32 – Pontos inicias dos dedos 55
Figura 33 – Captura de tela.. 57
Figura 34 – Ambientes controlados... 58
Figura 35 – Ambientes com iluminação moderada 58
Figura 36 – Ambientes com baixa iluminação 59
Figura 37 – Ambientes diversos 59
Figura 38 – Pose como os dedos separados... 60
Figura 39 – Pose com os dedos juntos... 60
Figura 40 – Alinhamento do Modelo 3D do sistema esquelético............................ 61
Figura 41 – Tela do aplicativo no final do teste 64
Figura 42 – Gráfico comparativo entre a Versão original vs Versão otimizada 64
Figura 43 – Classificação dos triângulos quanto aos lados 69
Figura 44 – Classificação dos triângulos quanto ao ângulo 69
Figura 45 – Cálculo de perímetro.. 70
Figura 46 – Cálculo de área... 70

LISTA DE QUADROS

Quadro 1 – Callback da câmera....33
Quadro 2 – Loop principal da thread de identificação da mão....33
Quadro 3 – Intervalo HSV....34
Quadro 4 – Segmentação da imagem....34
Quadro 5 – Seleção do contorno....35
Quadro 6 – Pontos convexos de um contorno....35
Quadro 7 – Matriz de defeitos de convexidade....36
Quadro 8 – Filtrando pontos de convexidade....39
Quadro 9 – Procura os 3 pontos mais próximos....42
Quadro 10 – Identificando os índices dos triângulos de convexidade....43
Quadro 11 – Calculando distâncias....43
Quadro 12 – Ordenando lista de pontos de defeito....44
Quadro 13 – Matriz de parâmetros intrínsecos....45
Quadro 14 – Função solvePNP....45
Quadro 15 – Convertendo para matriz 3x3....46
Quadro 16 – Manipulando objeto temporário....47
Quadro 17 – Ângulo de rotação no eixo Z dos dedos....47
Quadro 18 – Atualizando o objeto compartilhado....49
Quadro 19 – Inicialização da matriz de projeção....49
Quadro 20 – Construção da matriz de projeção da câmera....50
Quadro 21 – Método onProjectionChanged....50
Quadro 22 – Inicialização dos objetos da cena....51
Quadro 23 – Carregando o modelo 3D....51
Quadro 24 – Identificação dos objetos carregados....52
Quadro 25 – Etapa final da inicialização....53
Quadro 26 – Etapa inicial da renderização....54
Quadro 27 – Atualização das matrizes ModelView....54
Quadro 28 – Calcula a pose dos dedos....56
Quadro 29 – Atualização da pose do modelo 3D....56
Quadro 30 – Parâmetro de câmera....62
Quadro 31 – Alocação da imagem cache....62

Quadro 32 – Desenho da imagem cache 63
Quadro 33 – Cálculos estatísticos... 63

LISTA DE ABREVIATURAS E SIGLAS

3D - Três Dimensões

API - Application Programming Interface

FPS - Frames Per Second

HSV - Hue, Saturation, Value

P*n*P - Perspective *n* Point

RA - Realidade Aumentada

RF - Requisitos Funcionais

RV – Realidade Virtual

RGB - Red, Green, Blue

RNF - Requisitos Não Funcionais

SNA - Sistema Nervoso Autônomo

SNC - Sistema Nervoso Central

SNP - Sistema Nervoso Periférico

UML - Unified Modeling Language

SUMÁRIO

1 INTRODUÇÃO 14
1.1 OBJETIVOS 15
1.2 ESTRUTURA 15
2 FUNDAMENTAÇÃO TEÓRICA 16
2.1 ANATOMIA DO CORPO HUMANO 16
2.2 MATRIZES DE TRANSFORMAÇÕES DO OPENGL 19
2.3 IDENTIFICAÇÃO DA POSE COM MODELO DE CÂMERA *PINHOLE* 21
2.4 TRABALHOS CORRELATOS 24
2.4.1 VIRTUALI-TEE 24
2.4.2 ANATOMY 4D 25
2.4.3 ARLOON ANATOMY 27
3 DESENVOLVIMENTO DO PROTÓTIPO 29
3.1 REQUISITOS PRINCIPAIS DO PROTÓTIPO PROPOSTO 29
3.2 FERRAMENTAS E TÉCNICAS UTILIZADAS 29
3.3 IMPLEMENTAÇÃO 30
3.4 ANÁLISE DOS RESULTADOS 57
3.4.1 Identificando o contorno de interesse 57
3.4.2 Identificando os principais pontos convexos 59
3.4.3 Renderização do modelo 3D do sistema esquelético com a pose da mão 60
3.4.4 Otimização do módulo de câmera da biblioteca OpenCV 62
4 CONCLUSÕES 65
4.1 EXTENSÕES 66
REFERÊNCIAS 67
APÊNDICE A – GEOMETRIA DO TRIÂNGULO 69

1 INTRODUÇÃO

De acordo com Rebello et al. (2011, p. 1), para ensinar e estudar as estruturas anatômicas, muitas universidades ainda utilizam aulas expositivas, atlas e a dissecação de cadáveres como método de ensino. Rebello et al. (2011, p. 1), também afirmam que por ser uma disciplina em que o aluno precisa visualizar e compreender o funcionamento de diversas estruturas da anatomia humana, em muitos casos o uso destes métodos mesmo que consolidados, acabam não sendo a melhor alternativa e potencialmente comprometem a boa compreensão desta ciência.

Ramey (2013) garante que os recentes avanços tecnológicos vêm trazendo resultados positivos na área da educação. Para Lacerda (2013), tecnologias como a realidade aumentada são capazes de auxiliar tanto no processo de ensino quanto no de aprendizagem, beneficiando assim professores e alunos. Portanto, a tecnologia na educação é capaz de tornar o processo de aprendizagem mais dinâmico e adjunto para com a realidade dos alunos.

Lacerda (2013, p. 2) aponta que mediante a possibilidade do emprego da tecnologia no âmbito educacional, inicialmente deve ser feita uma análise a fim de determinar quais tecnologias podem satisfazer os objetivos motivacionais da aprendizagem. Além de também avaliar quais tecnologias podem potencializar a eficiência no processo de aquisição de conhecimento.

Segundo Forte (2009), uma tecnologia relativamente recente e que está sendo bastante estudada quanto a sua utilização no campo educacional é a Realidade Aumentada (RA). Esta tecnologia se mostra bastante promissora, oferecendo uma interface moderna e intuitiva. Ela permite que o usuário interaja com um ambiente virtual de forma natural e dinâmica. O uso desta tecnologia possibilita o usuário visualizar projeções de elementos/objetos 3D, além de permitir manipular estes elementos/objetos de forma semelhante de como é feito no mundo real. Tornando o uso desta tecnologia uma experiência bastante imersiva e estimulante.

Para Reardon (2010), o emprego da RA pode ser bastante aproveitável para estudantes de biologia. Eles, poderiam utilizar realidade aumentada, a partir de um dispositivo móvel *smartphone*, para obter informações detalhadas sobre o que estão visualizando enquanto dissecam um réptil durante a aula. Isso indica que aplicações similares voltadas para a visualização da anatomia humana também possam apresentar potencial.

Diante do exposto, este trabalho apresenta o desenvolvimento uma aplicação móvel com caráter educacional utilizando a tecnologia de RA. A aplicação possibilitará a visualização em três dimensões da anatomia do sistema esquelético da mão.

1.1 OBJETIVOS

O objetivo deste trabalho é desenvolver uma aplicação fazendo uso da tecnologia de Realidade Aumentada para visualização de parte da anatomia do corpo humano em tempo real.

Os objetivos específicos são:

a) identificar o contorno da mão a partir da segmentação da imagem com base na cor da pele;
b) estimar a pose da mão a partir dos principais pontos convexos do contorno;
c) renderizar o modelo esquelético 3D utilizando a pose estimada da mão;
d) otimizar o desempenho do módulo da câmera da biblioteca OpenCV.

1.2 ESTRUTURA

Este trabalho está organizado em quatro capítulos. O primeiro capítulo apresenta os objetivos e a motivação para desenvolvimento do trabalho. No segundo capítulo, é tratado da fundamentação teórica do trabalho, demonstrando os principais conceitos e técnicas utilizadas no desenvolvimento do protótipo. No terceiro capítulo são descritos a arquitetura do trabalho através de diagramas, o detalhamento da implementação do protótipo e os resultados obtidos nos testes realizados. Por fim, são apresentadas as conclusões e limitações do trabalho, assim como sugestões para trabalhos futuros.

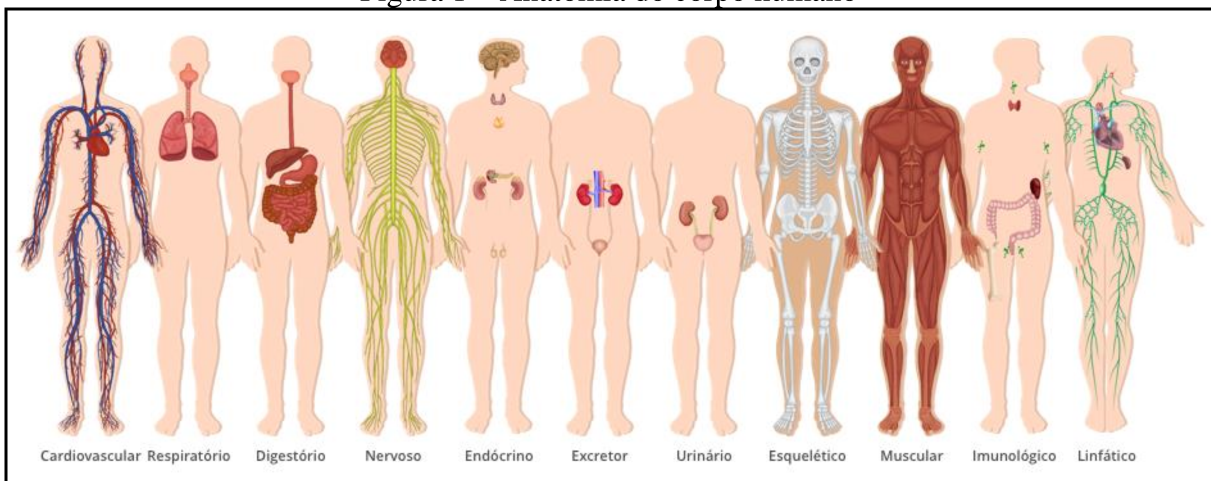
2 FUNDAMENTAÇÃO TEÓRICA

Nesta seção serão abordados os assuntos que deverão fundamentar a aplicação a ser desenvolvida. A seção 2.1 explica a anatomia do corpo humano. A seção 2.2 descreve o funcionamento das matrizes de transformações do OpenGL. A seção 2.3 explica como funciona a identificação da pose com o modelo de câmera *pinhole*. Por último, a seção 2.4 apresenta os trabalhos correlacionados. Além disso, o Apêndice A elucida os conceitos da geometria de triângulo.

2.1 ANATOMIA DO CORPO HUMANO

Segundo TodaMateria (2018a), o corpo humano é composto pelos seguintes sistemas: cardiovascular, respiratório, digestório, nervoso, sensorial, endócrino, excretor, urinário, reprodutor, esquelético, muscular, imunológico, linfático e tegumentar, como é possível conferir na Figura 1. Cada um destes sistemas envolve órgãos e possuem uma função específica no funcionamento do corpo humano.

Figura 1 – Anatomia do corpo humano



Fonte: TodaMateria (2018a).

O sistema cardiovascular envolve um conjunto de órgãos com a responsabilidade de levar oxigênio e nutrientes para as partes do corpo. O principal órgão desse sistema é o coração, no qual bombeia o sangue, de modo que com ele possa chegar até as extremidades do corpo e voltar para o coração (TODAMATERIA, 2018b). Já o sistema respiratório é responsável pela troca de gases ligados ao processo de respiração das células. O conjunto de órgãos facilita a captação do oxigênio do ar e a eliminação do gás carbônico (CO2), produzido pelo organismo, para o meio ambiente. O sistema respiratório é composto pelas vias respiratórias e pelos pulmões (TODAMATERIA, 2018c).

O sistema digestório é responsável por fazer a ingestão, digestão, absorção e eliminação dos alimentos através de processos mecânicos e químicos. Os principais órgãos do sistema digestório são: boca, faringe, esôfago, estomago, intestino delgado e intestino grosso (TODAMATERIA, 2018d). Por outro lado, o sistema nervoso é o mediador das atividades corporais e se subdivide em três segmentos: Sistema Nervoso Central (SNC), Sistema Nervoso Periférico (SNP) e o Sistema Nervoso Autônomo (SNA). O SNC exerce funções como: pensamento, visão, audição, tato, paladar, fala, escrita, memória, raciocínio, inteligência, imaginação, e controla os movimentos voluntários. O SNP tem como papel interligar o sistema nervoso central ao resto do corpo e transmitir impulsos nervosos. Já o SNA atua conjuntamente com o SNC para estimular o funcionamento de ações involuntárias como batimentos cardíacos, dilatamento da pupila, etc (TODAMATERIA, 2018e).

O sistema endócrino é composto por um conjunto de glândulas responsáveis pela elaboração dos hormônios, no quais são lançados na corrente sanguínea e percorrem o corpo até chegar aos órgãos-alvo. Dentre as glândulas que fazem parte do sistema endócrino estão: hipófise, tireoide, paratireoides, timo, suprarrenais, pâncreas e glândulas sexuais (TODAMATERIA, 2018f). Já o sistema excretor tem o papel de eliminar resíduos das células, produzidos durante o processo de metabolismo. Sendo assim, o sistema excretor é o principal responsável pelo equilíbrio da composição química no ambiente interno (TODAMATERIA, 2018g). Por outro lado, o sistema urinário é responsável pela produção e eliminação da urina, filtrando as impurezas da corrente sanguínea. O sistema urinário é composto por dois rins e as vias urinárias (ureteres, bexiga, uretra) (TODAMATERIA, 2018h).

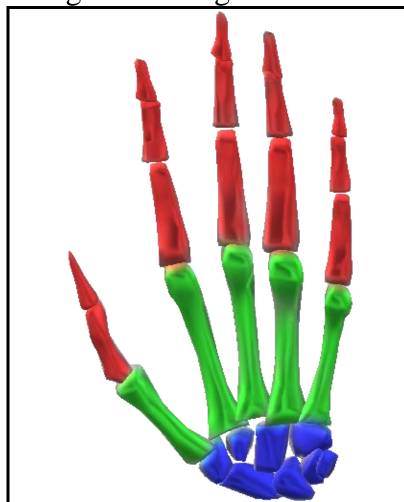
Segundo TodaMateria (2018i), o sistema reprodutor masculino possui órgãos internos e externos, que passam por um processo de amadurecimento lento (puberdade). Os órgãos que fazem parte deste sistema são: testículos, epidídimos, canal deferente, vesículas seminais, próstata, uretra e pênis. Estes órgãos têm como função eliminar a urina produzida pelo sistema urinário, além de produzir e transportar o sêmen até o exterior. Já o sistema reprodutor feminino é o sistema responsável pela reprodução humana. Os órgãos que fazem parte deste sistema são: ovários, tubas, uterinas, útero e vagina. Os ovários são dois órgãos ovais que produzem células sexuais femininas (óvulos). Tubas uterinas são dois tubos que unem o ovário ao útero, permitindo a passagem do óvulo. O útero é um órgão muscular oco de grande elasticidade, responsável por acomodar o embrião durante a fase de gravidez. Por fim, a vagina é um canal que faz a comunicação do útero com o meio excretor (TODAMATERIA, 2018j).

O sistema muscular é composto por diversos músculos do corpo humano pelos quais são controlados pelo sistema nervoso. Suas principais funções são: estabilidade, movimentação, aquecimento, sustentação e auxiliar na circulação sanguínea. Os músculos deste sistema são agrupados nos seguintes grupos musculares: músculos da cabeça e do pescoço, músculos do tórax e do abdômen, músculos dos membros superiores e músculos dos membros inferiores (TODAMATERIA, 2018k).

O sistema linfático tem função remover fluidos dos tecidos, absorver ácidos graxos e encaminhá-los para o sistema circulatório e produzir células imunes. Os principais órgãos deste sistema são: baço, linfonodos, macrófagos e linfócitos (TODAMATERIA, 2018l). Nesse contexto, o sistema tegumentar é responsável por proteger o corpo contra possíveis agentes externos, regular a temperatura do corpo e garantir a sensibilidade através de nervos. O principal órgão deste sistema é a pele (TODAMATERIA, 2018m).

O sistema esquelético é formado por ossos, cartilagens, ligamentos e tendões. Sua principal função é sustentar, proteger, dar forma e atuar em conjunto com o sistema muscular para permitir o movimento. O esqueleto humano pode ser dividido em: esqueleto apendicular e esqueleto axial. O esqueleto apendicular é constituído pelos ossos dos membros superiores e inferiores. Já o esqueleto axial é constituído pelo crânio, a caixa torácica e a coluna vertebral (TODAMATERIA, 2018n). No segmento terminal do membro superior ficam os ossos da região que compreendem a mão. Ao total são 27 diferentes ossos que compõe a mão, juntamente com os músculos e articulações, que em conjunto permitem a realização de movimentos. Em relação a estrutura óssea, a mão possui as seguintes subdivisões: carpo, metacarpo e falange (TODAMATERIA, 2018o). No modelo da Figura 2 são destacadas as regiões do carpo, metacarpo e falange com as cores azul, verde e vermelho, respectivamente.

Figura 2 – Regiões da mão



Fonte: elaborado pelo autor.

Segundo TodaMateria (2018o), a região do carpo possui 8 ossos que compreendem região do pulso. A região do metacarpo é constituída por 5 ossos similares que se articulam com os ossos do carpo e com as falanges. Já a região da falange é constituída por 14 ossos que compreendem os dedos da mão e se articulam com os ossos do metacarpo. Cada dedo possui três falanges, com exceção do polegar que possui 2 falanges. O principal movimento realizado pela mão é a ação de pinça, graças ao polegar opositor. Este movimento permite que sejam realizados trabalhos mais delicados e com maior precisão, como por exemplo, escrita, construção de ferramentas, desenhos e entre outros.

2.2 MATRIZES DE TRANSFORMAÇÕES DO OPENGL

Na área de computação gráfica é muito comum o uso de matrizes para representar transformações em um sistema de coordenadas. O tipo de matriz mais utilizado para o processamento gráfico são matrizes de tamanho 4x4. Estas matrizes de tamanho 4x4 quando multiplicadas por um vértice (nesta ordem, necessariamente) geram um vértice transformado (OPENGL-TUTORIAL, 2017). Na Figura 3 é ilustrada a transformação de um vértice.

Figura 3 – Transformação de um vértice

$$\begin{bmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{bmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} a \cdot x + b \cdot y + c \cdot z + d \cdot 1 \\ e \cdot x + f \cdot y + g \cdot z + h \cdot 1 \\ i \cdot x + j \cdot y + k \cdot z + l \cdot 1 \\ m \cdot x + n \cdot y + o \cdot z + p \cdot 1 \end{pmatrix}$$

Fonte: Overvoorde (2017).

Segundo OpenGL-Tutorial (2017), no OpenGL as matrizes de translação são os tipos de matrizes de transformações mais simples. Como o próprio nome sugere, estas matrizes são responsáveis em aplicar o efeito de translação em um dado vértice. Na Figura 4 é demonstrada a transformação de um vértice com uma matriz de translação.

Figura 4 – Matriz de translação

$$\begin{bmatrix} 1 & 0 & 0 & X \\ 0 & 1 & 0 & Y \\ 0 & 0 & 1 & Z \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x + X \cdot 1 \\ y + Y \cdot 1 \\ z + Z \cdot 1 \\ 1 \end{pmatrix}$$

Fonte: Overvoorde (2017).

A matriz identidade quando usada no OpenGL como uma matriz de transformação, não realiza nenhuma transformação (OPENGL-TUTORIAL, 2017). Como é demonstrado na Figura 5, ao multiplicar a matriz identidade por um vértice, o resultado obtido é o próprio vértice.

Figura 5 – Matriz identidade

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \cdot x + 0 \cdot y + 0 \cdot z + 0 \cdot 1 \\ 0 \cdot x + 1 \cdot y + 0 \cdot z + 0 \cdot 1 \\ 0 \cdot x + 0 \cdot y + 1 \cdot z + 0 \cdot 1 \\ 0 \cdot x + 0 \cdot y + 0 \cdot z + 1 \cdot 1 \end{pmatrix} = \begin{pmatrix} 1 \cdot x \\ 1 \cdot y \\ 1 \cdot z \\ 1 \cdot 1 \end{pmatrix}$$

Fonte: Overvoorde (2017).

As matrizes de escala no OpenGL também são simples. Quando usadas como matriz de transformação, são responsáveis por aumentar a escala de um vértice (OPENGL-TUTORIAL, 2017). Na Figura 6 é ilustrada a transformação de um vértice com uma matriz de escala, onde as variáveis SX, SY e SZ representam os fatores de escala nos eixos X, Y e Z respectivamente.

Figura 6 – Matriz de escala

$$\begin{bmatrix} SX & 0 & 0 & 0 \\ 0 & SY & 0 & 0 \\ 0 & 0 & SZ & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} SX \cdot x \\ SY \cdot y \\ SZ \cdot z \\ 1 \end{pmatrix}$$

Fonte: Overvoorde (2017).

Segundo Overvoorde (2017), as matrizes de rotação do OpenGL são responsáveis por aplicar o efeito de rotação. Existem 3 tipos de matrizes de rotação neste caso, onde cada uma é responsável por rotacionar o vértice em um eixo especifico. Na Figura 7 é representado o produto da transformação de um vértice com as matrizes de rotação.

Figura 7 – Matrizes de rotação

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x \\ \cos\theta \cdot y - \sin\theta \cdot z \\ \sin\theta \cdot y + \cos\theta \cdot z \\ 1 \end{pmatrix}$$

(a) Matriz de rotação do eixo X

$$\begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} \cos\theta \cdot x + \sin\theta \cdot z \\ y \\ -\sin\theta \cdot x + \cos\theta \cdot z \\ 1 \end{pmatrix}$$

(b) Matriz de rotação do eixo Y

$$\begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} \cos\theta \cdot x - \sin\theta \cdot y \\ \sin\theta \cdot x + \cos\theta \cdot y \\ z \\ 1 \end{pmatrix}$$

(c) Matriz de rotação do eixo Z

Fonte: Overvoorde (2017).

Os conceitos sobre matrizes de transformações do OpenGL se tornam relevantes durante os principais processos do protótipo. Durante o processo de identificação da mão, por exemplo, a pose da mão é calculada e então é gerada a sua matriz de transformação, também conhecida como *ModelView*. Já no processo de renderização do modelo 3D do protótipo, as matrizes de transformações dos dedos são calculadas a partir de outras matrizes de transformações.

2.3 IDENTIFICAÇÃO DA POSE COM MODELO DE CÂMERA PINHOLE

Segundo DigitalRune (2016), a combinação de posição e orientação de um objeto também é conhecida como pose. De acordo com Grabner, Roth e Lepetit (2018), identificar a pose de um objeto a partir de uma imagem é de extrema importância para compreensão de uma cena em três dimensões(3D), principalmente aplicações de realidade aumentada e tarefas como manipulação de objetos virtuais. O uso desta técnica em combinação com realidade aumentada pode render inúmeras possibilidades. Como por exemplo, manipular um objeto

virtual através de gestos realizados com a mão, ou ainda sobrepor objetos do mundo real com objetos virtuais em três dimensões.

Para identificar a pose de um objeto é comum o uso da solução para o problema Perspective-*n*-Point (P*n*P). P*n*P é o problema de estimar a pose de uma câmera calibrada dado um conjunto de *n* pontos em 3D de um espaço e seus pontos em duas dimensões (2D) correspondentes em uma imagem. Uma solução comum utilizada para resolver o problema onde o conjunto de pontos *n* é igual a 3, é chamada de P3P. Além desta solução, existem outras soluções para o caso de *n* for maior do que 3, como Efficient P*n*P(EP*n*P) por exemplo (OPENCV, 2018a).

Os algoritmos mais comuns usados para resolver o problema P*n*P fazem uso do modelo de câmera estenopeica ou *pinhole*, em inglês (OPENCV, 2018a). Segundo MathWorks (2018), neste modelo de câmera, a cena visualizada é formada pela projeção de pontos em 3D (*World points*) no plano da imagem (*Image points*) utilizando a matriz de transformação de perspectiva (*Camera matrix*). Na Figura 8 a Câmera *matrix* é representada pela letra P. Portanto, a matriz P é formada pelo produto dos parâmetros intrínsecos e extrínsecos da câmera.

Figura 8 – Câmera matrix

$$w\,[x\ y\ 1] = [X\ Y\ Z\ 1]\,P$$

Scale factor, Image points, World points

$$P = \begin{bmatrix} R \\ t \end{bmatrix} K$$

Camera matrix, Extrinsics Rotation and translation, Intrinsic matrix

Fonte: MathWorks (2018).

Os parâmetros intrínsecos do modelo de câmera pinhole incluem *focal length* (Fx e Fy), *optical center* ou *principal point* (Cx e Cy) e coeficiente skew (s), como é ilustrado na Figura 9.

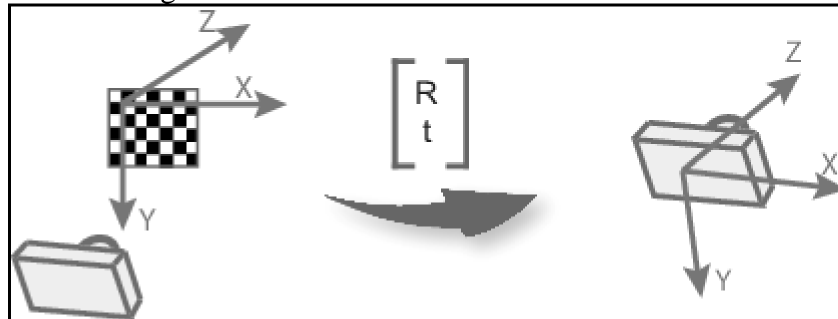
Figura 9 – Matriz de parâmetros intrínsecos da câmera

$$\begin{vmatrix} Fx & 0 & 0 \\ s & Fy & 0 \\ Cx & Cy & 1 \end{vmatrix}$$

Fonte: MathWorks (2018).

Já os parâmetros extrínsecos da câmera representam as matrizes de rotação e translação da câmera, como é representado na Figura 10.

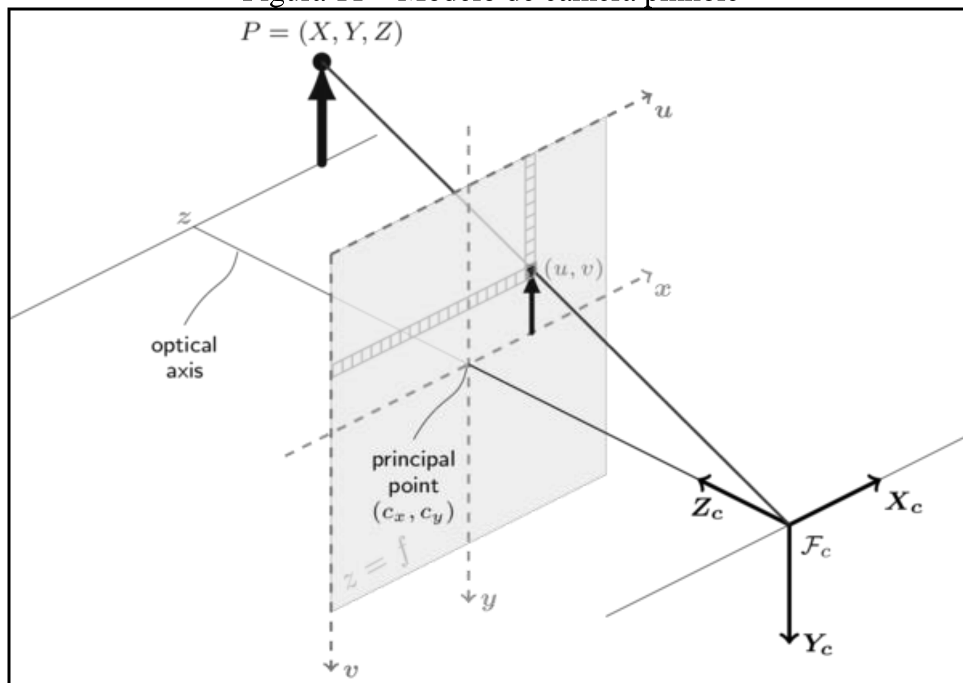
Figura 10 – Parâmetros extrínsecos da câmera



Fonte: MathWorks (2018).

O modelo de câmera *pinhole* é ilustrado na Figura 11 onde: as coordenadas de P(X, Y, Z) representam *world point*, u e v *image points*, e por fim, o sistema de coordenadas em preto (Xc, Yc e Zc) representa os parâmetros extrínsecos da câmera (OPENCV, 2018b).

Figura 11 – Modelo de câmera pinhole



Fonte: OpenCV(2018b).

O modelo de câmera *pinhole* se torna relevante no desenvolvimento do protótipo durante o processo de identificação da mão. Durante este processo, a pose da mão é calculada com o auxílio de uma função que soluciona o problema P*n*P. Nesta função é necessário informar como um dos parâmetros de entrada uma matriz 3x3 de parâmetros intrínsecos da câmera. Como resultado a função retorna os parâmetros extrínsecos da câmera, ou seja, as matrizes de rotação e translação.

2.4 TRABALHOS CORRELATOS

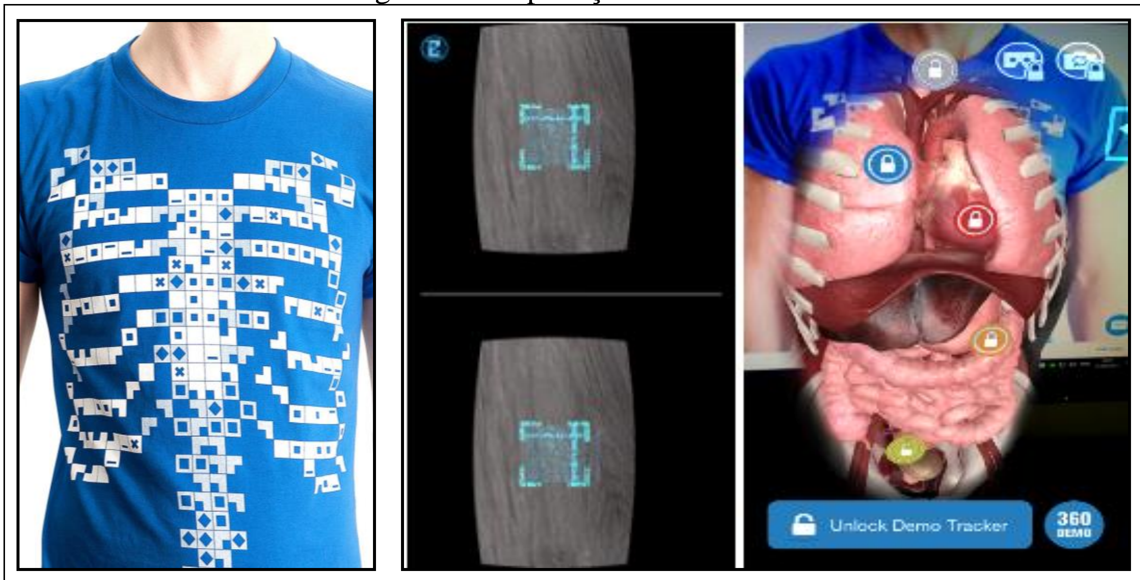
Nesta seção serão apresentadas três aplicações que utilizam RA, cuja a finalidade está relacionada ao que será desenvolvido neste trabalho. A seção 2.4.1 apresenta a aplicação Virtuali-Tee (CURISCOPE, 2016). Ela faz uso de uma camisa especial com marcador para assim visualizar e interagir com os sistemas fisiológicos do corpo humano. Na seção 2.4.2 é descrito uma aplicação que permite ao usuário visualizar o interior do corpo humano e do coração fazendo uso de apenas cartões de RA (DAQRI, 2014). Por fim, a seção 2.4.3 apresenta uma aplicação que além de exibir todos os sistemas fisiológicos do corpo humano, inclui uma série de questionários que são respondidos interagindo com os modelos 3D das partes do corpo humano (ARLOON, 2015).

2.4.1 VIRTUALI-TEE

O Virtuali-Tee é uma aplicação de RA que permite ao usuário visualizar uma projeção 3D animada do interior do corpo humano (CURISCOPE, 2016). Na animação é possível conferir o sistema respiratório, circulatório, digestivo, urinário e esquelético simultaneamente, com a possibilidade de isolar a visualização para apenas um desses sistemas fisiológicos. A aplicação requer o uso de uma camisa especial, com o marcador na região do torso e um dispositivo móvel (smartphone ou tablet) com o sistema operacional Android ou iOS. Também é possível fazer o uso da câmera frontal do dispositivo e o uso de óculos de Realidade Virtual (RV) com o smartphone (Google Cardboard, por exemplo).

Na Figura 12 é possível conferir algumas das telas disponíveis na aplicação. A primeira imagem representa a camisa que possui o marcador que irá mostrar a animação. Na segunda tela da figura o aplicativo está rodando em modo de RV com imagens estereoscópicas. Já na terceira tela da figura é apresentado a animação do interior do corpo humano.

Figura 12 – Aplicação Virtuali-Tee



Fonte: Curiscope (2016).

Para exibir a animação do sistema fisiológico a aplicação faz uso do marcador na camisa, no qual é identificado pela aplicação ao apontar a câmera. O marcador serve como um ponto de referência para aplicação projetar a animação. Assim, ao reconhecer o marcador, a animação começa a ser projetada. Ao movimentar a câmera é possível experimentar diferentes pontos de vista do interior do corpo humano e interagir com os botões presentes na animação. Cada botão está localizado em uma área específica do sistema fisiológico (sistema respiratório, sistema circulatório, etc) e ao pressioná-lo a animação exibida passa a ser exclusiva desse sistema.

Segundo Curiscope (2016) o Virtuali-Tee proporciona uma experiência diferenciada para o aprendizado do corpo humano, trazendo animações em tempo real de seu funcionamento, é ideal para ser usada por professores em sala de aula, a fim de tornar o ensino mais descontraído. A aplicação é comercializada por R$ 13,99 na Google Play e a camisa por €20 no site oficial Curiscope (2016).

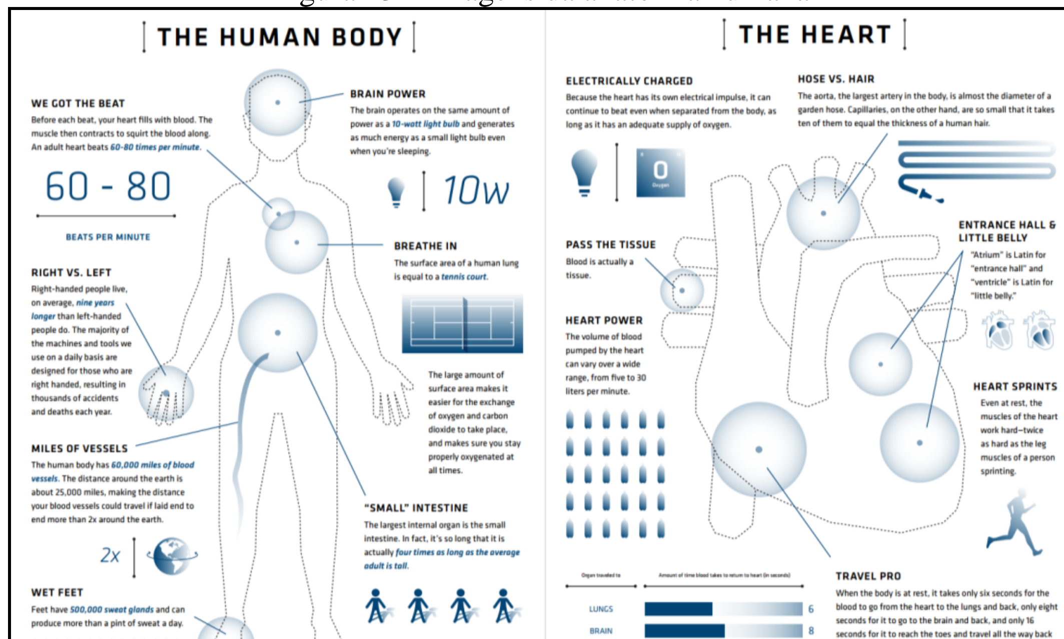
2.4.2 ANATOMY 4D

O Anatomy 4D é uma aplicação de RA disponível para as plataformas Android e iOS. A aplicação permite que o usuário visualize o interior do corpo humano ou do coração, oferecendo também a possibilidade de alternar o sexo do corpo. A partir dela também é possível enfatizar a visualização de um sistema fisiológico individualmente, como por exemplo, focar a visualização apenas para o sistema respiratório, ou nervoso, ou esquelético. Movimentando a câmera do dispositivo móvel para próximo da imagem impressa é possível

analisar um órgão, ou parte do corpo humano e suas particularidades, em detalhes (DAQRI, 2014).

Para fazer uso do aplicativo é necessário um dispositivo móvel (smartphone ou tablet) com câmera e ter impresso as imagens disponibilizadas pela aplicação em uma superfície plana. A aplicação dispõe de 2 imagens para serem impressas, uma delas possui o contorno de um corpo e a outra, de um coração humano, conforme mostra a Figura 13.

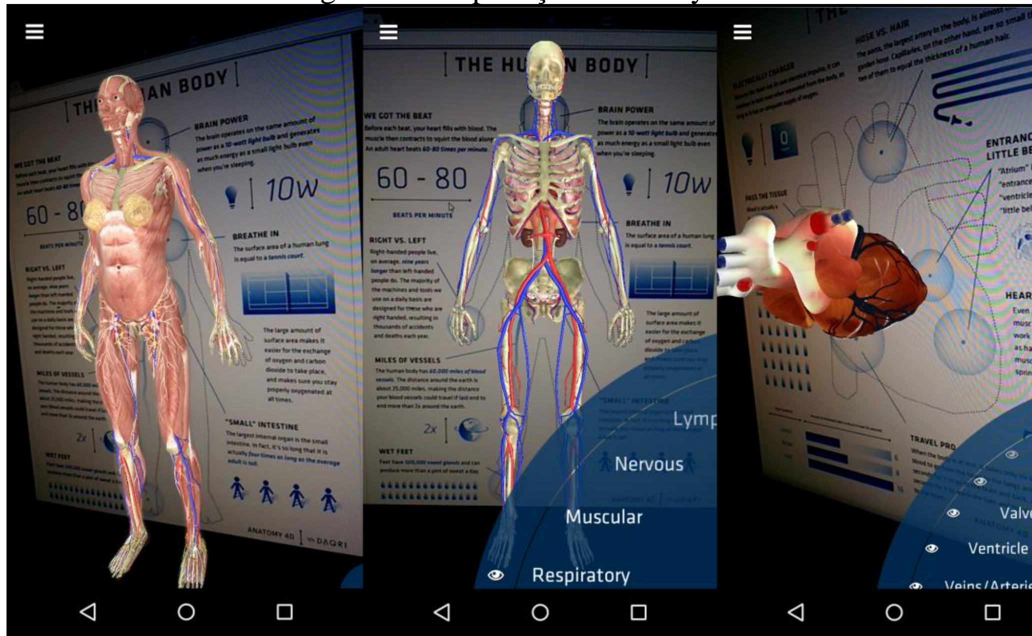
Figura 13 – Imagens da anatomia humana



Fonte: Daqri (2014).

Ao apontar a câmera do dispositivo móvel para uma das imagens da Figura 13, o aplicativo as reconhece e então projeta a animação correspondente. Na Figura 14 são apresentadas três telas capturadas da aplicação. A primeira mostra a anatomia do corpo humano feminino, exibindo todos os sistemas fisiológicos. Na segunda, são mostrados os sistemas esquelético e respiratório. Já na terceira tela é exibido a anatomia do coração humano.

Figura 14 – Aplicação Anatomy 4D



Fonte: Daqri (2014).

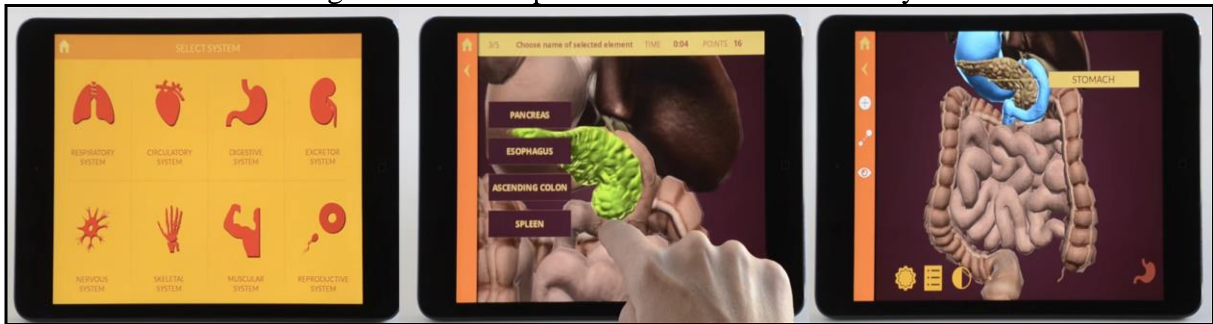
O Anatomy 4D (DAQRI, 2014) facilita o entendimento da anatomia do coração e do corpo humano de forma interativa. Por ser uma aplicação gratuita, fazer uso de dispositivos e materiais de fácil acesso, é ideal para ser utilizada em instituições de ensino como ferramenta de aprendizagem.

2.4.3 ARLOON ANATOMY

O Arloon Anatomy é uma aplicação de RA disponível para as plataformas Android e iOS. A aplicação permite que o usuário visualize os sistemas do corpo humano através da combinação de modelos em 3D e uso de RA. Na aplicação também é possível interagir com os modelos exibidos, selecionando órgãos específicos é possível obter uma breve descrição de seu funcionamento, assim como observá-lo de diferentes ângulos. A aplicação também dispõe de vídeos animados sobre o funcionamento de algumas funções do corpo humano, como processo de digestão, respiração, circulação, excreção e impulsos nervosos. Além disso, é possível exercitar o conteúdo aprendido sobre a anatomia humana, com os exercícios disponíveis na aplicação (ARLOON, 2015).

Na Figura 15 é possível conferir as principais telas da aplicação desenvolvida pela Arloon (2015). A primeira delas mostra a tela inicial da aplicação, onde a partir dela é possível escolher o sistema fisiológico a ser estudado. Na segunda, é apresentado um questionário para o usuário, onde o mesmo deve responder selecionando o nome do órgão que está em destaque (cor verde). Já na terceira tela, a aplicação exibe um modelo 3D animado do sistema digestório, onde o estômago se encontra destacado em azul.

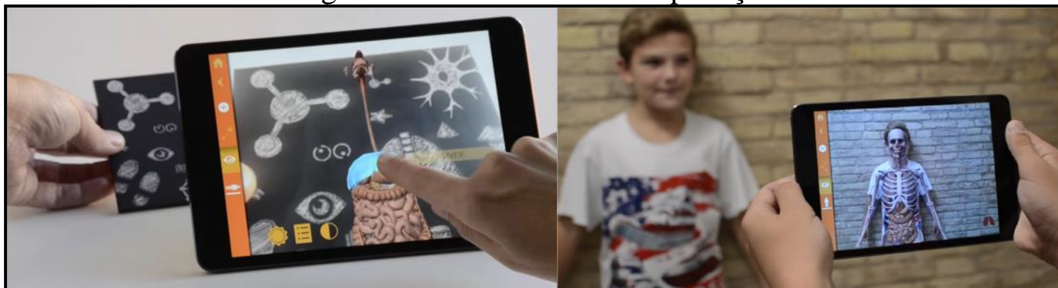
Figura 15 – Principais telas do Arloon Anatomy



Fonte: Arloon (2015).

A Figura 16 exemplifica como a RA foi empregada na aplicação. Na primeira tela a aplicação faz uso de um cartão de RA disponibilizado pela própria aplicação para exibir o sistema digestório. A segunda tela a aplicação faz uso de uma cobaia para detectar o contorno do corpo humano e projetar a animação da sua anatomia.

Figura 16 – Telas de RA da aplicação



Fonte: Arloon (2015).

A aplicação Arloon (2015) possui uma dinâmica propícia para ser utilizado em sala de aula como ferramenta de ensino. Além de ensinar sobre a anatomia do corpo humano de forma interativa, a aplicação traz uma série de questionários com a finalidade de avaliar o conteúdo aprendido. A aplicação é comercializada por R$ 7,99 na loja Google Play.

3 DESENVOLVIMENTO DO PROTÓTIPO

Neste capítulo são demonstradas as etapas do desenvolvimento do protótipo. Na seção 3.1 é apresentado os principais requisitos. A seção 3.2 apresenta a implementação de forma detalhada. Por fim, a seção 3.3 demonstra os resultados dos testes, sugestões e melhorias do protótipo.

3.1 REQUISITOS PRINCIPAIS DO PROTÓTIPO PROPOSTO

O protótipo a ser desenvolvido deve:

a) permitir que o usuário utilize a câmera do dispositivo para obter as imagens (Requisito Funcional - RF);
b) segmentar uma imagem com base na cor da pele (RF);
c) identificar o maior contorno de uma imagem segmentada (RF);
d) identificar os pontos convexos de um contorno (RF);
e) identificar os principais pontos convexos (RF);
f) estimar a pose da mão a partir de uma lista de pontos convexos (RF);
g) permitir ao usuário de visualizar o modelo 3D do sistema esquelético da mão (RF);
h) imitar a pose da mão com o modelo 3D do sistema esquelético (RF);
i) ser desenvolvida para a plataforma Android (Requisito Não Funcional - RNF);
j) utilizar o ambiente de desenvolvimento Android Studio (RNF);
k) utilizar a biblioteca de visão computacional OpenCV para fazer o processamento de imagem (RNF);
l) utilizar o motor gráfico Rajawali para renderizar o sistema esquelético da mão (RNF).

3.2 FERRAMENTAS E TÉCNICAS UTILIZADAS

O protótipo foi desenvolvido utilizando a linguagem de programação JAVA juntamente com o ambiente de desenvolvimento Android Studio 3.1.3 no sistema operacional Windows 10. Para o processamento de imagem foi utilizado a biblioteca de visão computacional OpenCV. Dentre as funcionalidades mais importantes da biblioteca para a implementação estão: conversão de imagem, segmentação, filtro de imagem, descritores de imagem e entre outros. Já para a renderização do modelo 3D foi feito uso de uma versão adaptada do motor gráfico Rajawali, no qual incorpora a biblioteca OpenGL ES 2.0 em sua implementação. As funcionalidades mais importantes do motor gráfico para a implementação

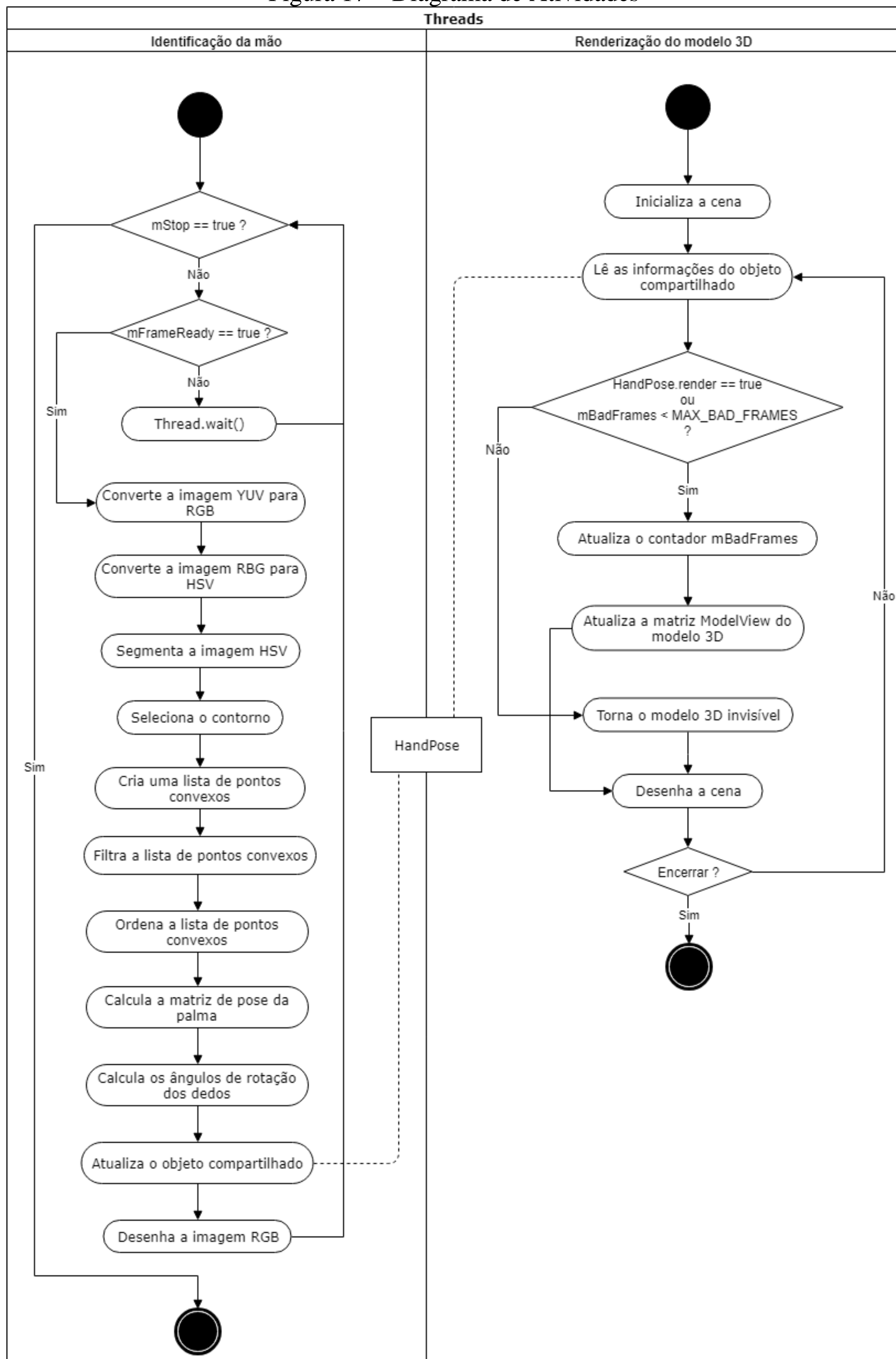
do protótipo foram: importação do arquivo gráfico *wavefront* e objetos, uso de materiais e iluminação.

As principais *threads* do protótipo foram especificadas através do diagrama de atividades da Unified Modeling Language (UML), utilizando a ferramenta web draw.io. Para a elaboração das imagens foram utilizadas as ferramentas Paint 3D, AutoCAD, além de um segundo protótipo desenvolvido na IDE Visual Studio para a plataforma Windows em C++.

3.3 IMPLEMENTAÇÃO

Na Figura 17 é possível observar o diagrama de atividades que ilustra os principais processos executados no protótipo: identificação da mão e renderização do modelo 3D. Estes dois processos são executados concorrentemente, ou seja, cada um em uma *thread*. Além disso, as *threads* compartilham um objeto durante a execução cujo o acesso é controlado através de um semáforo.

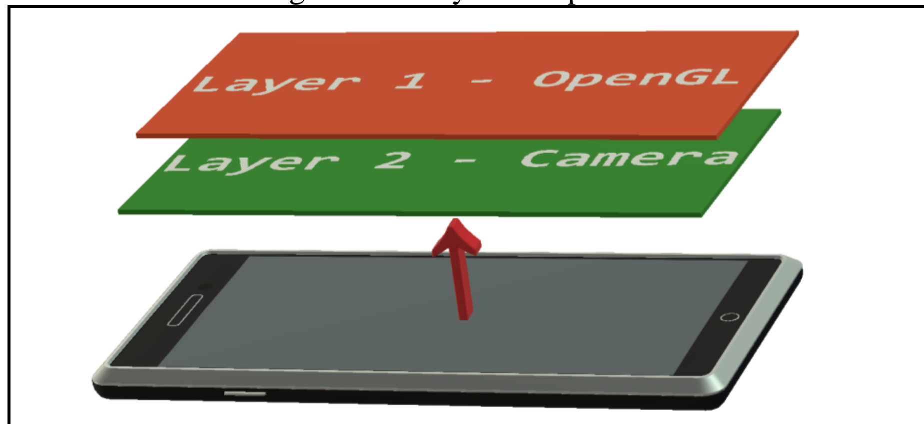
Figura 17 – Diagrama de Atividades



Fonte: elaborado pelo autor.

A *thread* de identificação da mão é responsável por processar cada imagem obtida através da câmera e extrair a pose da mão. Identificada a pose da mão, o objeto compartilhado (`HandPose`) é atualizado com nova posição e orientação. Já a *thread* de renderização do modelo 3D lê constantemente os dados disponíveis no objeto compartilhado e usa os mesmos para renderizar o modelo esquelético 3D da mão. Além disso, ambas as *threads* desenham em diferentes *layers* do aplicativo, conforme ilustra a Figura 18.

Figura 18 – Layers do aplicativo



Fonte: elaborado pelo autor.

O *layer* 2 fica situado a baixo do *layer* 1 e tem como conteúdo as imagens obtidas pela câmera. Estas imagens por sua vez são desenhadas no *layer* pela *thread* de identificação da mão após terminar o processamento da imagem. O *layer* 1 adiciona uma camada de informações sobre as imagens da câmera. Estas informações são imagens desenhadas pela *thread* de renderização do modelo 3D. Deste modo, combinando as informações dos 2 *layers*, é obtido e efeito de realidade aumentada almejado. Nas próximas seções serão detalhados os processos de identificação da mão e o de renderização do modelo 3D.

3.3.1 Identificação da mão

Antes do protótipo iniciar a execução da *thread* de identificação da mão, são realizados dois eventos. Primeiramente é criado um *buffer* com duas posições chamado de `mFrameChain`. Este *buffer* é responsável por armazenar as imagens obtidas através da função *call-back* `onPreviewFrame`. Em seguida, durante a configuração da câmera, é definida uma função do tipo *call-back* para receber as imagens da câmera assim que estiverem prontas. A função *call-back* definida é chamada de `onPreviewFrame` e tem como função armazenar o frame obtido no *buffer* e notificar a *thread* de identificação da mão sobre a nova imagem obtida, conforme ilustrado no Quadro 1.

Quadro 1 – Callback da câmera

```
1.  @Override
2.  public void onPreviewFrame(byte[] frame, Camera arg1)
3.  {
4.      synchronized (this)
5.      {
6.          mFrameChain[mChainIdx].put(0, 0, frame);
7.          mCameraFrameReady = true;
8.          this.notify();
9.      }
10.     if (mCamera != null)
11.         mCamera.addCallbackBuffer(mBuffer);
12. }
```

Fonte: OpenCV (2018c).

No início da execução da *thread* de identificação da mão são verificadas as condições de duas *flags*: `mStop` e `mFrameReady`. A *flag* `mStop` quando verdadeira indica que a execução precisa ser encerrada. Já a *flag* `mFrameReady` indica se existe ainda alguma imagem aguardando processamento no *buffer* `mFrameChain`. No caso de ambas as *flags* serem falsas, a *thread* entra em estado de espera. Caso apenas a *flag* `mFrameReady` seja verdadeira, é calculado o índice `mChainIdx` do *buffer* para obter a imagem a ser processada. Como o *buffer* `mFrameChain` possui apenas duas posições, os únicos valores possíveis para `mChainIdx` são: 0 e 1. Calculado o índice do *buffer* é dado início ao processamento da imagem com a chamada da função `deliverAndDrawFrame`, conforme mostra o Quadro 2.

Quadro 2 – Loop principal da thread de identificação da mão

```
1.  do
2.  {
3.      boolean hasFrame = false;
4.      synchronized (JavaCameraView.this)
5.      {
6.          try
7.          {
8.              while (!mFrameReady && !mStop)
9.              {
10.                 JavaCameraView.this.wait();
11.             }
12.         }
13.         catch (InterruptedException e)
14.         {
15.             e.printStackTrace();
16.         }
17.         if (mFrameReady)
18.         {
19.             mChainIdx = 1 - mChainIdx;
20.             mFrameReady = false;
21.             hasFrame = true;
22.         }
23.     }
24.
25.     if (!mStop && hasFrame)
26.     {
27.         if (!mFrameChain[1 - mChainIdx].empty())
28.             deliverAndDrawFrame(mCameraFrame[1 - mChainIdx]);
29.     }
30. } while (!mStop);
```

Fonte: OpenCV (2018c).

A função `deliverAndDrawFrame` tem como objetivo processar o parâmetro `mCameraFrame` recebido e, no final, desenha-lo no *layer* inferior do aplicativo. O parâmetro `mCameraFrame` processado pela função nada mais é do que uma instância do *buffer* `mFrameChain` criado anteriormente, porém encapsulada pela classe `JavaCameraFrame`. A função da classe `JavaCameraFrame` converte a imagem obtida pela câmera do dispositivo do formato YUV para RGB.

Na primeira etapa do processamento, a imagem RGB é convertida para HSV. Em seguida, a imagem HSV é segmentada com base em dois intervalos: `mMinHSV1` e `mMaxHSV1`, cujo os valores destes intervalos tentam incluir as tonalidades mais comuns da cor de pele. No trecho de código no Quadro 3 é possível conferir a inicialização destes intervalos.

Quadro 3 – Intervalo HSV

```
1.  mMinHSV1 = new Scalar(0, 20, 60);
2.  mMaxHSV1 = new Scalar(20, 150, 255);
```

Fonte: elaborado pelo autor.

O resultado da segmentação é uma imagem binária (`mBinMat`) que é submetida ao filtro MedianBlur para corrigir pequenas imperfeições. No Quadro 4 é apresentado o trecho de código executado nestas etapas iniciais do processamento.

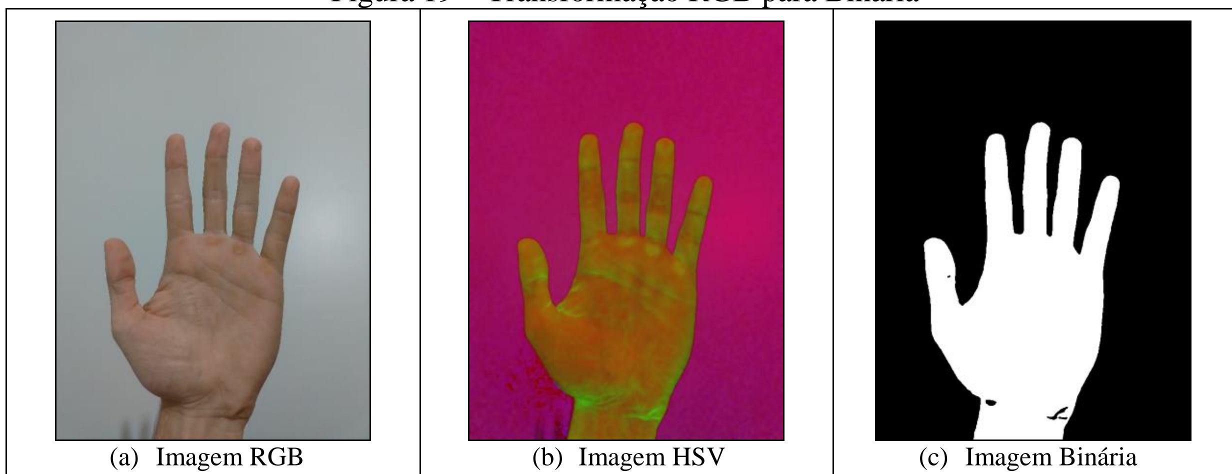
Quadro 4 – Segmentação da imagem

```
1.  Imgproc.cvtColor(mRGBA, mHSV, Imgproc.COLOR_RGB2HSV);
2.  Core.inRange(mHSV, mMinHSV1, mMaxHSV1, mBinMat);
3.  Imgproc.medianBlur(mBinMat, mBinMat, BLUR_SIZE);
```

Fonte: elaborado pelo autor.

A Figura 19 ilustra as etapas realizadas até a obtenção da imagem segmentada. Na primeira imagem encontra-se a imagem obtida pela câmera do dispositivo em formato RGB. Já a segunda imagem é o resultado da conversão da imagem RGB para HSV. Por último, tem-se a imagem resultante do processo de segmentação e `medianBlur`.

Figura 19 – Transformação RGB para Binária



(a) Imagem RGB (b) Imagem HSV (c) Imagem Binária

Fonte: elaborado pelo autor.

Na etapa seguinte é criada uma lista de contornos com base na imagem binária gerada na etapa anterior. Tal lista é percorrida e cada vez que o maior contorno é encontrado, atualiza-se a variável `mLargestContour`. O Quadro 5 mostra o trecho de código responsável por esta etapa.

Quadro 5 – Seleção do contorno

```
1.  // Monta a lista de contornos
2.  Imgproc.findContours(mBinMat, mListOfContours, mHierarchy, Imgproc.RETR_EXTERNAL,
     Imgproc.CHAIN_APPROX_SIMPLE);
3.  if (!mListOfContours.isEmpty())
4.  {
5.      mLargestContour = 0;
6.      for (mIndex = 1; mIndex < mListOfContours.size(); ++mIndex)
7.      {
8.          // Compara a área de dois contornos
9.          if (Imgproc.contourArea(mListOfContours.get(mIndex)) > Imgproc.contourAre
    a(mListOfContours.get(mLargestContour)))
10.         {
11.             mListOfContours.get(mLargestContour).release();
12.             // Atualiza o índice do maior contorno
13.             mLargestContour = mIndex;
14.         }
15.         ...
16.     }
17.     ...
18. }
```

Fonte: elaborado pelo autor.

Em seguida, o maior contorno é submetido aos métodos `convexHull` e `convexityDefects`. O método `convexHull` tem como finalidade retornar uma lista de pontos convexos. Esta lista de pontos convexos é retornada pelo objeto `mHull` e serve como parâmetro de entrada para o método `convexityDefects`. O método `convexityDefects` retorna uma matriz de tamanho Nx4 na variável `mDefects`, como é representado no Quadro 6.

Quadro 6 – Pontos convexos de um contorno

```
1.  // Monta a lista de pontos convexos em mHull
2.  Imgproc.convexHull(mListOfContours.get(mLargestContour), mHull, true);
3.  if (!mHull.empty())
4.  {
5.      // Monta a matriz de defeitos de convexidade em mDefects
6.      Imgproc.convexityDefects(mListOfContours.get(mLargestContour), mHull, mDefect
    s);
7.      if (mDefects.rows() >= HandTracking.MIN_HAND_DEFECTS)
8.      {
9.          ...
10.     }
11.     ...
12. }
```

Fonte: elaborado pelo autor.

A matriz `mDefects` obtida na etapa anterior possui N linhas e 4 colunas, onde o valor de N varia de acordo com a complexidade do contorno a ser processado. Caso o número de linhas da matriz for menor do que a constante `HandTracking.MIN_HAND_DEFECTS`(ou 4) o processamento da imagem atual é abortado.

No Quadro 7 é possível conferir um exemplo de matriz de defeitos de convexidade de tamanho 3x4. Cada coluna da matriz representa as seguintes informações respectivamente: ponto convexo inicial, ponto convexo final, ponto de defeito de convexidade e distância.

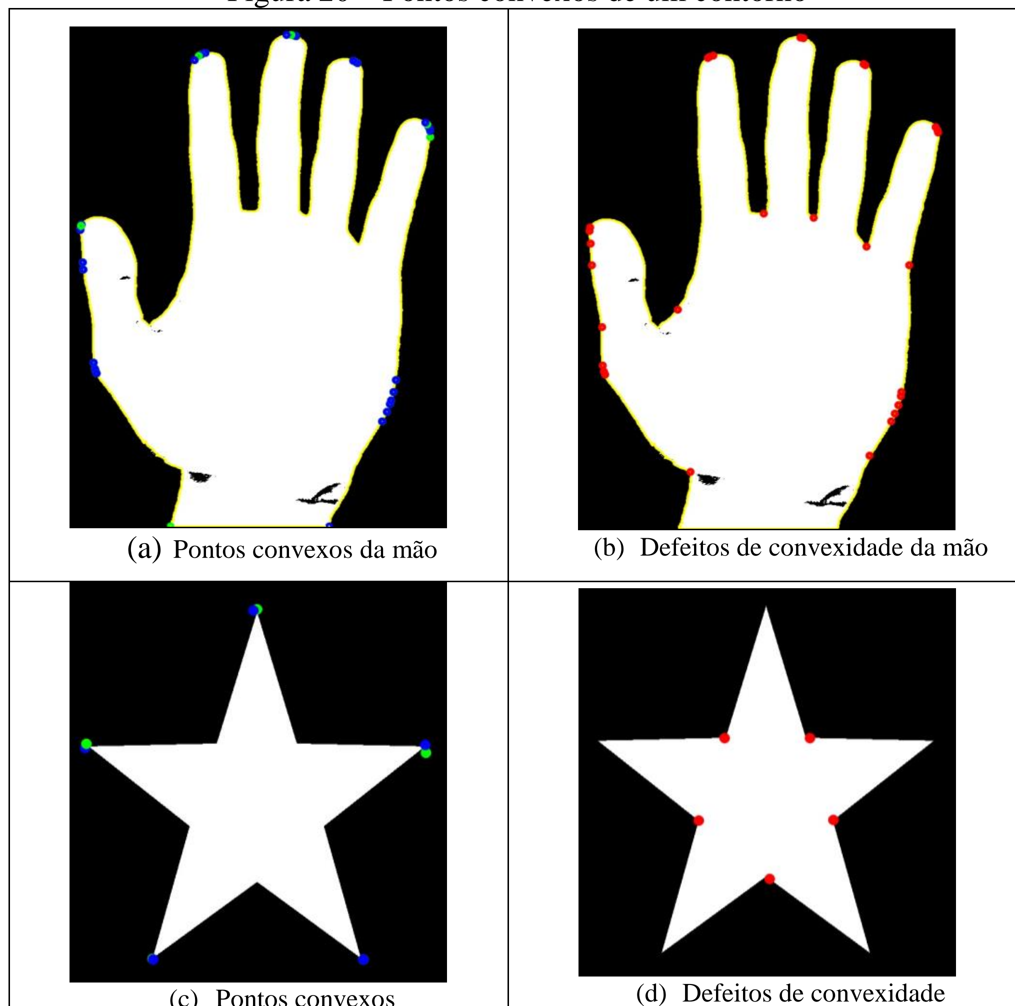
Quadro 7 – Matriz de defeitos de convexidade

Pontos convexos iniciais	Pontos convexos finais	Pontos de defeito de convexidade	Distância
24.00, 367.00	136.00, 718.00	198.00, 497.00	~ 126.24
137.00, 719.00	487.00, 719.00	316.00, 594.00	~125.00
487.00, 719.00	591.00, 375.00	422.00, 496.00	~126.75

Fonte: elaborado pelo autor.

Nos contornos da Figura 20 os pontos convexos iniciais, finais e pontos de defeito de convexidade são representados com as cores (azul, verde e vermelho) das colunas do Quadro 7.

Figura 20 – Pontos convexos de um contorno



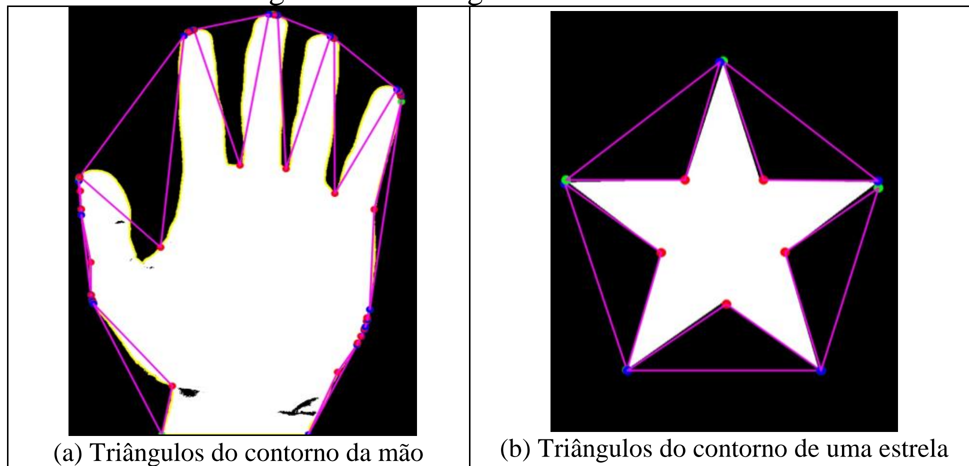
(a) Pontos convexos da mão
(b) Defeitos de convexidade da mão
(c) Pontos convexos
(d) Defeitos de convexidade

Fonte: elaborado pelo autor.

Se traçado um segmento de reta entre os três pontos obtidos em uma linha de informações da matriz `mDefects` é possível formar um triângulo. A altura deste triângulo formado será igual ao valor da informação “distância” presente na última coluna da matriz.

Na Figura 21 é ilustrado os triângulos de convexidade formados ao traçar um segmento de reta entre os pontos obtidos de uma linha da matriz.

Figura 21 – Triângulos de convexidade



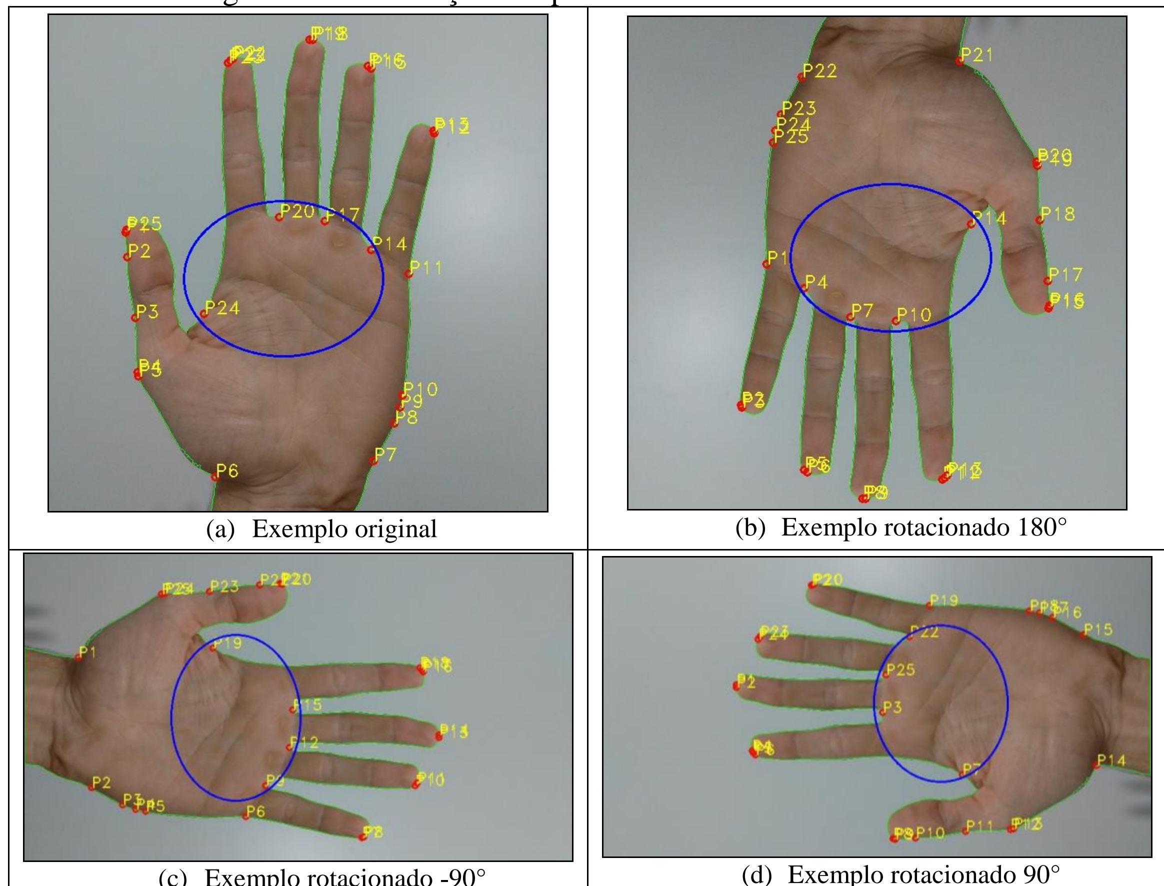
(a) Triângulos do contorno da mão

(b) Triângulos do contorno de uma estrela

Fonte: elaborado pelo autor.

As linhas da matriz `mDefects` já vem ordenadas, permitindo assim que as informações possam ser acessadas como uma lista circular. Por exemplo, se todos os pontos de defeitos de convexidade de um contorno fossem enumerados de P1 até PN o resultado seria como ilustrado na Figura 22.

Figura 22 – Ordenação dos pontos de defeitos de convexidade



(a) Exemplo original

(b) Exemplo rotacionado 180°

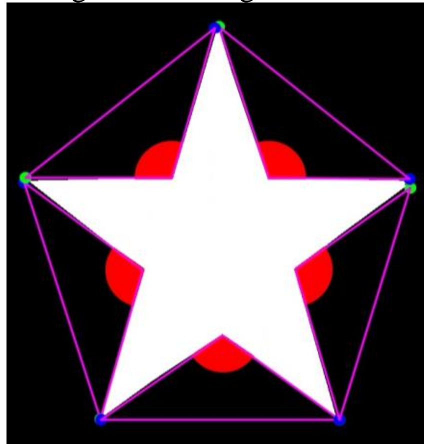
(c) Exemplo rotacionado -90°

(d) Exemplo rotacionado 90°

Fonte: elaborado pelo autor.

Muitos dos triângulos de convexidade formados pelo contorno da mão não contribuem significativamente para o processo de identificação da mão. Por isso, todos os triângulos passam por um processo de seleção. Este processo leva em consideração duas informações de um triângulo: altura e ângulo interno. O ângulo interno calculado neste caso é o ângulo formado próximo ao ponto de defeito de convexidade. Na Figura 23 os ângulos internos dos triângulos de convexidade são representados pelos semicírculos em vermelho.

Figura 23 – Ângulo interno



Fonte: elaborado pelo autor.

No Quadro 8 é apresentado o trecho de código que filtra os triângulos de convexidade. A matriz de defeitos de convexidade mDefects é convertida para um vetor antes de ser processada. Deste modo apenas um índice é necessário para acessar as informações do vetor e a cada interação do loop, o índice é incrementado quatro vezes. Os triângulos de convexidade que forem passando nos testes são adicionados a uma nova lista dentro da classe HandTracking.

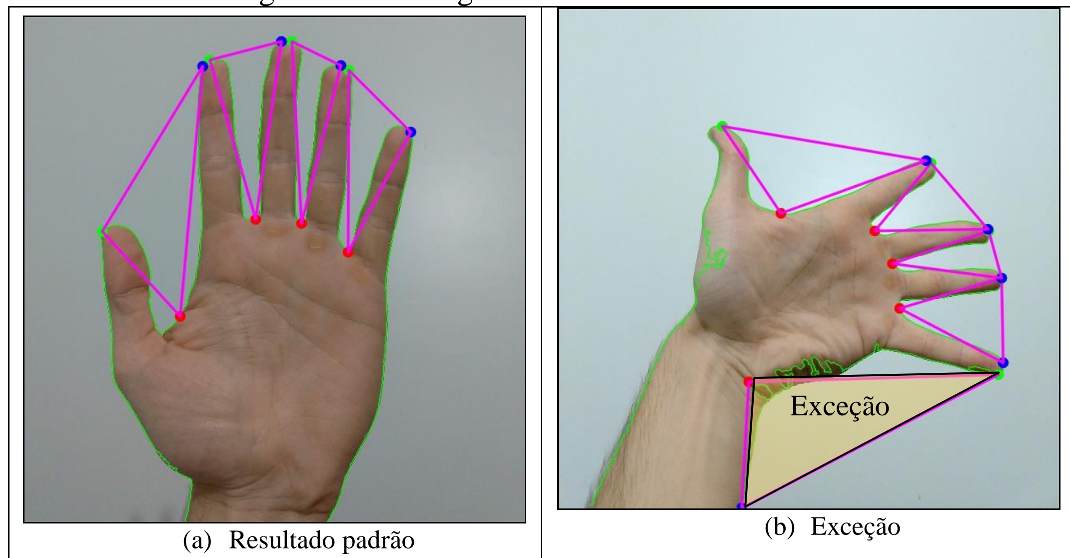
Quadro 8 – Filtrando pontos de convexidade

```
1.  double angle;
2.  int[] defectsList = mDefects.toArray();
3.  for (mIndex = 0; mIndex < defectsList.length; ++mIndex)
4.  {
5.      // Adiciona as informações a uma nova estrutura
6.      mHandDefect.startPoint.set(mListOfContours.get(mLargestContour).get(defectsList
    [mIndex++], 0));
7.      mHandDefect.endPoint.set(mListOfContours.get(mLargestContour).get(defectsList[m
    Index++], 0));
8.      mHandDefect.farthestPoint.set(mListOfContours.get(mLargestContour).get(defectsL
    ist[mIndex++], 0));
9.
10.     // Altura do triângulo de convexidade
11.     mHandDefect.length = defectsList[mIndex] / 256.0f;
12.
13.     // Ângulo interno do triângulo de convexidade
14.     angle = innerAngle(mHandDefect.startPoint, mHandDefect.endPoint, mHandDefect.fa
    rthestPoint);
15.
16.     // Filtra os triângulos pela altura e ângulo interno
17.     if (mHandDefect.length >= MIN_FINGER_LENGTH &&
18.         mHandDefect.length <= MAX_FINGER_LENGTH &&
19.         angle >= MIN_INNER_ANGLE &&
20.         angle <= MAX_INNER_ANGLE)
21.     {
22.         // Adiciona a estrutura a uma nova lista
23.         if (!mHandTracking.addHandDefect(mHandDefect))
24.             // Interrompe a execução caso a lista esteja cheia
25.             break;
26.     }
27. }
```

Fonte: elaborado pelo autor.

Durante a execução desta etapa, se o número de triângulos de convexidade adicionados a nova lista passar de 5, a execução é interrompida e o processamento desta imagem é abortado. Está ação se faz necessária visto que o contorno da mão normalmente possui apenas 4 triângulos de convexidade. Porém, o limite da lista são 5 triângulos porque existe uma exceção. A exceção é ilustrada na Figura 24 ao lado do resultado padrão.

Figura 24 – Triângulos de convexidade filtrados

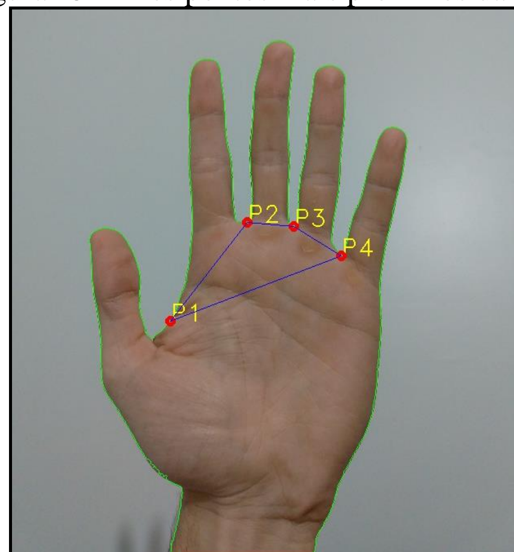


(a) Resultado padrão (b) Exceção

Fonte: elaborado pelo autor.

Na etapa seguinte, a nova lista de triângulos de convexidade é utilizada para identificar a que par de dedos cada triangulo pertence. Estes pares podem ser: polegar com indicador, indicador com dedo médio, dedo médio com dedo anelar e dedo anelar com dedo mínimo. Para este propósito foi criada uma função chamada `findClosest3Points` no qual leva em consideração apenas os pontos de defeito de convexidade da lista de triângulos convexos. A função encontra os três pontos mais próximos da lista e retorna o índice do ponto do meio. Ao executar esta função com triângulos de convexidade da mão, ela retorna o índice do ponto de defeito de convexidade pertencente ao dedo médio e o anelar. No exemplo ilustrado na Figura 25 o ponto retornado pela função será o P3.

Figura 25 – Três pontos mais próximos da mão



Fonte: elaborado pelo autor.

Ao executar a função findClosest3Points primeiramente é feita a inicialização de algumas variáveis usadas durante a execução. A variável defectsSize guarda o tamanho da lista de triângulos de convexidade da mão a ser percorrido. O vetor d possui três posições e é responsável por guardar as distâncias obtidas entre os pontos nas duas primeiras posições. Já a última posição do vetor guarda o somatório dos valores das distâncias presentes nas duas primeiras posições. A variável i representa o índice da lista a ser percorrida. As variáveis shortesDist e oldDist guardam valores de distância obtidos durante a execução.

A lista a ser percorrida é interpretada como circular e adota o sentido horário para examinar os pontos. A cada ponto examinado é levado em consideração as distâncias formadas com os dois possíveis pontos vizinhos. Caso alguma destas distâncias esteja fora do intervalo MIN_FINGER_WIDTH e MAX_FINGER_WIDTH a função avança para a próxima comparação, caso houver. Se a distância d[0] estiver fora deste intervalo o índice é decrementado duas vezes, pulando assim uma comparação. No final do loop, caso o valor armazenado em shortesDist seja maior do d[2], atualiza-se as informações do novo índice encontrado. Quando o valor do índice i chegar a ser menor do que zero, a função termina retornando o valor armazenado em result, como é possível conferir no Quadro 9.

Quadro 9 – Procura os 3 pontos mais próximos

```
1.  private int findClosest3Points()
2.  {    // Tamanho da lista de triângulos de convexidade
3.      int defectsSize = mHandDefectsList.size();
4.      if (defectsSize < MIN_HAND_DEFECTS)
5.          return -1;
6.      // Inicializa valor das 3 distâncias com 0
7.      d[0] = d[1] = d[2] = 0;
8.      int i = defectsSize, result = -1;
9.      float shortestDist = MAX_DISTANCE, oldDist ;
10.
11.     for (; i > 0; --i)
12.     {
13.         oldDist = d[0];
14.         d[0] = distanceP2P(mHandDefectsList.get(i % defectsSize).farthestPoint,
15.                      mHandDefectsList.get((i - 1) % defectsSize).farthestPoint);
16.         if (d[0] < MIN_FINGER_WIDTH || d[0] > MAX_FINGER_WIDTH)
17.         {
18.             // Pula a próxima comparação
19.             d[0] = 0;
20.             --i;
21.             continue;
22.         }
23.         d[1] = oldDist > 0 ? oldDist :
24.                 distanceP2P(mHandDefectsList.get(i % defectsSize).farthestPoint,
25.                 mHandDefectsList.get((i + 1) % defectsSize).farthestPoint);
26.         if (d[1] < MIN_FINGER_WIDTH || d[1] > MAX_FINGER_WIDTH)
27.             continue;
28.
29.         d[2] = d[0] + d[1];
30.
31.         if ((d[2]) < shortestDist)
32.         {
33.             // Atualiza as informações do novo índice econtrado
34.             shortestDist = d[2];
35.             result = i % defectsSize;
36.         }
37.     }
38.     return result;
39. }
```

Fonte: elaborado pelo autor.

A partir da identificação do índice do triângulo de convexidade do dedo médio com o dedo anelar é calculado os índices dos dedos vizinhos. Os cálculos dos índices consistem em simplesmente incrementar ou decrementar em 1 o valor de um índice já calculado, mas sempre mantendo o aspecto da lista circular, isto é, o valor deve sempre estar entre 0 e a quantidade de triângulos de convexidade. No Quadro 10 é apresentado o trecho de código destes cálculos.

Quadro 10 – Identificando os índices dos triângulos de convexidade

```
1.  int index = findClosest3Points();
2.  if (index != -1)
3.  {
4.      int prev_index = index == 0 ? size - 1 : index - 1;
5.      int next_index = (index + 1) % size;
6.      int prev_index_2 = prev_index == 0 ? size - 1 : prev_index - 1;
7.      int next_index_2 = (next_index + 1) % size;
8.      ...
9.  }
```

Fonte: elaborado pelo autor.

Para identificar a que par de dedos pertence o restante dos índices calculados, é necessário descobrir em que lado se encontra o polegar. Para obter esta informação são calculadas duas distâncias entre pontos de defeitos de convexidade. O cálculo das distâncias é realizado conforme demonstrado no Quadro 11.

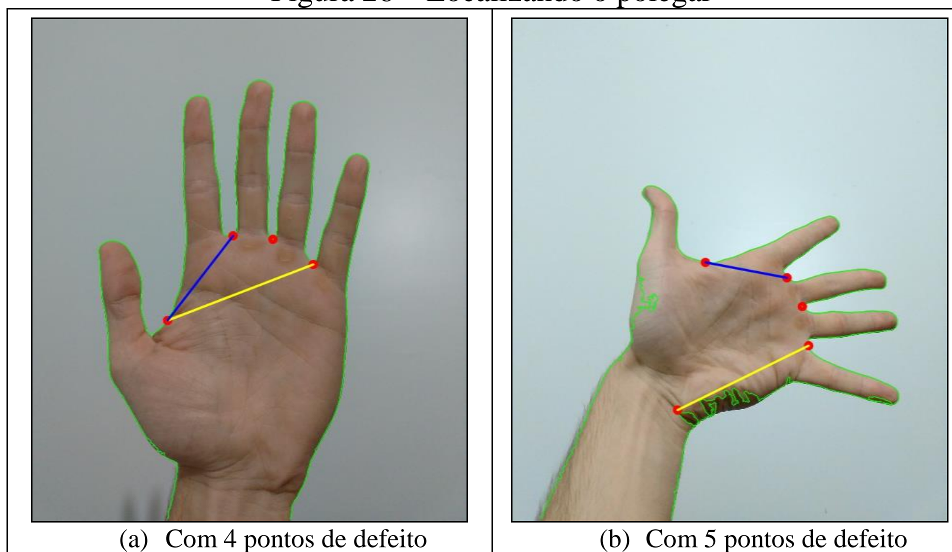
Quadro 11 – Calculando distâncias

```
1.  float dist1 = distanceP2P(mHandDefectsList.get(prev_index).farthestPoint, mHandDefe
    ctsList.get(prev_index_2).farthestPoint);
2.  float dist2 = distanceP2P(mHandDefectsList.get(next_index).farthestPoint, mHandDefe
    ctsList.get(next_index_2).farthestPoint);
```

Fonte: elaborado pelo autor.

Com base na menor distância obtida entre as duas distâncias calculadas anteriormente, é possível determinar em que lado fica o polegar. Como é demonstrado na Figura 26, onde os segmentos de reta em azul e amarelo representam as distâncias `dist2` e `dist1` respectivamente.

Figura 26 – Localizando o polegar



(a) Com 4 pontos de defeito (b) Com 5 pontos de defeito

Fonte: elaborado pelo autor.

Com base nas distâncias obtidas na etapa anterior, agora é possível determinar se a pose analisada é da mão direita ou esquerda. A identificação do polegar também permite a criação de uma lista de pontos de defeito de convexidade ordenada. A ordenação da lista

sempre segue a ordem do polegar até o dedo mínimo, independentemente se é a mão esquerda ou direita. Além disso, a lista de índices dos dedos é atualizada com o índice dos pontos de defeitos de convexidade. Esta lista por sua vez será usada nas etapas seguintes para identificar o ângulo de rotação de cada dedo. No Quadro 12 é demonstrado o trecho de código desta etapa.

Quadro 12 – Ordenando lista de pontos de defeito

```
1.  // Indica se a mão da direita ou esquerda
2.  boolean right_model;
3.  // Escolhe a menor distância
4.  if (dist1 > dist2)
5.  {
6.      // Mão esquerda
7.      right_model = false;
8.
9.      // Cria uma lista de pontos de defeito de convexidade ordenada
10.     mPalmPointsMat.fromArray(mHandDefectsList.get(next_index_2).farthestPoint,
11.                              mHandDefectsList.get(next_index).farthestPoint,
12.                              mHandDefectsList.get(index).farthestPoint,
13.                              mHandDefectsList.get(prev_index).farthestPoint);

14.     // Atualiza lista de indices
15.     mFingerIndices[0] = next_index_2;
16.     mFingerIndices[1] = next_index_2;
17.     mFingerIndices[2] = next_index;
18.     mFingerIndices[3] = index;
19.     mFingerIndices[4] = prev_index;
20. }
21. else
22. {
23.     // Mão direita
24.     ...
25. }
```

Fonte: elaborado pelo autor.

Na etapa seguinte é identificada a pose da palma da mão com base nas informações obtidas até o momento, juntamente com os parâmetros intrínsecos de uma câmera calibrada. Os parâmetros intrínsecos são calculados com base nas informações obtidas pela própria câmera do dispositivo junto com a resolução da imagem. Os parâmetros obtidos pela câmera são `mFOVX` e `mFOVY` que representam *field of view* horizontal e *field of view* vertical, respectivamente. Com estes parâmetros é possível calcular os valores necessários para montar a matriz 3x3 de parâmetros intrínsecos, como é apresentado no Quadro 13.

Quadro 13 – Matriz de parâmetros intrínsecos

```
1.  ...
2.  final float fovAspectRatio = mFOVX / mFOVY;
3.  double diagonalPx = Math.sqrt((Math.pow(mScreenWidth, 2.0) +
4.                                Math.pow(mScreenWidth / fovAspectRatio, 2.0)));
5.  double diagonalFOV = Math.sqrt((Math.pow(mFOVX, 2.0) + Math.pow(mFOVY, 2.0)));
6.  double focalLengthPx = diagonalPx / (2.0 * Math.tan(0.5 *
7.                         diagonalFOV * Math.PI / 180f));
8.  // Matriz 3x3 de parâmetros intrínsecos da câmera
9.  mProjectionCV.put(0, 0, focalLengthPx);
10. mProjectionCV.put(0, 1, 0.0);
11. mProjectionCV.put(0, 2, 0.5 * mScreenWidth);
12.
13. mProjectionCV.put(1, 0, 0.0);
14. mProjectionCV.put(1, 1, focalLengthPx);
15. mProjectionCV.put(1, 2, 0.5 * mScreenHeight);
16.
17. mProjectionCV.put(2, 0, 0.0); // Optical center x
18. mProjectionCV.put(2, 1, 0.0); // Optical center y
19. mProjectionCV.put(2, 2, 1.0);
20. ...
```

Fonte: elaborado pelo autor.

Os parâmetros intrínsecos calculados anteriormente são usados como parâmetro de entrada para função `solvePNP`. Além dos parâmetros intrínsecos, são necessários mais três parâmetros, como é apresentado no Quadro 14.

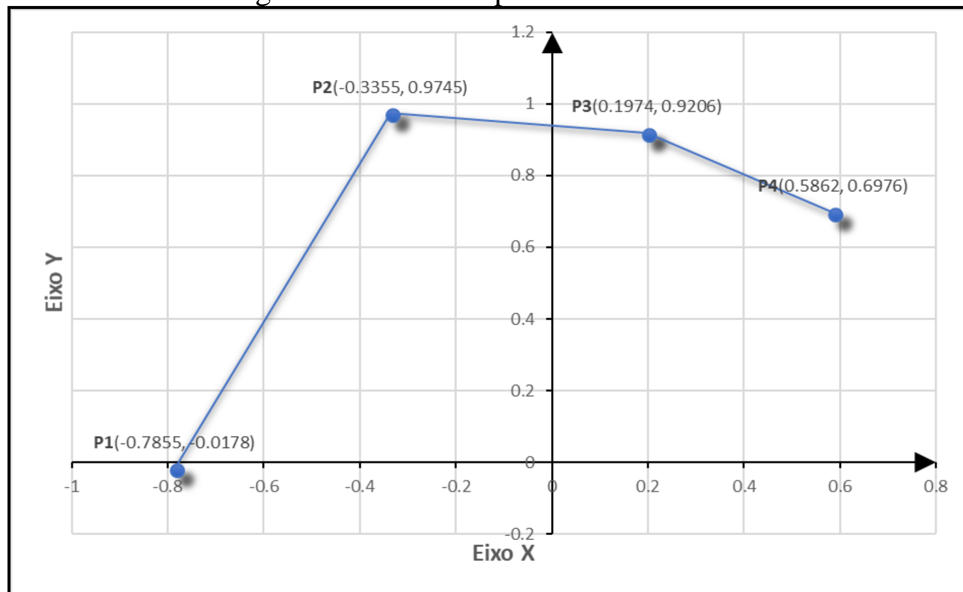
Quadro 14 – Função solvePNP

```
1.  Calib3d.solvePnP(mRefPoints, mPalmPointsMat, intrinsicParam, mDistCoeffs, mRVec,
2.                                      mTVec, false, Calib3d.SOLVEPNP_ITERATIVE);
```

Fonte: elaborado pelo autor.

Os parâmetros `mRefPoints` e `mPalmPointsMat` são listas de pontos de defeito de convexidade de mesmo tamanho e ordenação. A lista `mRefPoints` contém quatro pontos que fazem referência a pose da palma da mão na origem de um plano em três dimensões. Esta lista tem os pontos definidos no início da execução do aplicativo e tem estes valores mantidos até o final da execução. Na Figura 27 estes pontos de referência são ilustrados em um plano cartesiano.

Figura 27 – Lista de pontos de referência



Fonte: elaborado pelo autor.

Por último, o parâmetro `mDistCoeffs` representa uma matriz com informações sobre o coeficiente de distorção. Como a distorção não é levada em consideração pelo protótipo, este parâmetro é inicializado com zero.

Como resultado, a função `solvePNP` retorna dois vetores: `mRVec` e `mTVec`. O vetor `mRVec` contém informações de rotação nos três eixos. Já o vetor `mTVec` contém informações de translação, também nos três eixos. No trecho de código do Quadro 15, o vetor `mRVec` tem o valor do eixo X invertido para ser utilizado como parâmetro para a função `Rodrigues`. A função `Rodrigues` por sua vez, transforma `mRVec` em uma matriz de rotação 3x3 e retorna em `mRotation`.

Quadro 15 – Convertendo para matriz 3x3

```
1.  ...
2.  double[] rVecArray = mRVec.toArray();
3.  rVecArray[0] *= -1.0; // Inverte valor do eixo X
4.  mRVec.fromArray(rVecArray);
5.
6.  // Transforma mRVec em uma matriz de rotação 3x3
7.  Calib3d.Rodrigues(mRVec, mRotation);
8.
9.  double[] tVecArray = mTVec.toArray();
10. ...
```

Fonte: elaborado pelo autor.

Em seguida, os valores obtidos até aqui são atribuídos a um objeto temporário chamado `mHandPoseTemp`. O objeto `mHandPoseTemp` é do mesmo tipo que o objeto compartilhado entre as duas principais *threads* do protótipo. Posteriormente, as informações atribuídas a este objeto serão passadas ao objeto compartilhado, como é demonstrado no Quadro 16.

Quadro 16 – Manipulando objeto temporário

```
1.  ...
2.
3.  mHandPoseTemp.render = true
4.  mHandPoseTemp.right_model = right_model;
5.
6.  mHandPoseTemp.pose[0] = mRotation.get(0,0)[0];
7.  mHandPoseTemp.pose[1] = mRotation.get(0,1)[0];
8.  mHandPoseTemp.pose[2] = mRotation.get(0,2)[0];
9.  mHandPoseTemp.pose[3] = 0;
10. mHandPoseTemp.pose[4] = mRotation.get(1,0)[0];
11. mHandPoseTemp.pose[5] = mRotation.get(1,1)[0];
12. mHandPoseTemp.pose[6] = mRotation.get(1,2)[0];
13. mHandPoseTemp.pose[7] = 0;
14. mHandPoseTemp.pose[8] = mRotation.get(2,0)[0];
15. mHandPoseTemp.pose[9] = mRotation.get(2,1)[0];
16. mHandPoseTemp.pose[10] = mRotation.get(2,2)[0];
17. mHandPoseTemp.pose[11] = 0;
18. mHandPoseTemp.pose[12] = tVecArray[0];
19. mHandPoseTemp.pose[13] = - tVecArray[1];
20. mHandPoseTemp.pose[14] = - tVecArray[2];
21. mHandPoseTemp.pose[15] = 1;
22.
23. ...
```

Fonte: elaborado pelo autor.

Após o cálculo da pose da mão, é necessário calcular o ângulo de rotação de cada dedo. Para calcular o ângulo de rotação primeiramente é calculado a tangente da reta formada por dois pontos. O primeiro ponto da reta sempre será um ponto de defeito de convexidade. Já o segundo ponto pode variar entre ponto inicial e ponto final, conforme é exposto no Quadro 17.

Quadro 17 – Ângulo de rotação no eixo Z dos dedos

```
1.  ...
2.  // Calcula o ângulo de rotação no eixo Z da palma da mão
3.  double palmRotationZ = Math.atan2(mHandPoseTemp.pose[4], mHandPoseTemp.pose[0]);
4.  // Calcula o ângulo de rotação no eixo Z do polegar
5.  mHandPoseTemp.fingerAngles[0] =
6.                  arcTang(mHandDefectsList.get(mFingerIndices[0]).farthestPoint,
7.                  right_model ?
8.                  mHandDefectsList.get(mFingerIndices[0]).startPoint :
9.                  mHandDefectsList.get(mFingerIndices[0]).endPoint);
10. mHandPoseTemp.fingerAngles[0] -= palmRotationZ;
11. mHandPoseTemp.fingerAngles[0] -= DEF_FINGER_ANGLES[0];
12. mHandPoseTemp.fingerAngles[0] *= -MathUtil.PRE_180_DIV_PI;
13.
14. // Calcula o ângulo de rotação no eixo Z dos demais dedos
15. for (int i = 1, j = mHandPoseTemp.fingerAngles.length; i < j; ++i)
16. {
17.     mHandPoseTemp.fingerAngles[i] =
18.
    arcTang(mHandDefectsList.get(mFingerIndices[i]).farthestPoint,
19.                         right_model ?
20.                         mHandDefectsList.get(mFingerIndices[i]).endPoint :
21.                         mHandDefectsList.get(mFingerIndices[i]).startPoint);
22.     mHandPoseTemp.fingerAngles[i] -= palmRotationZ;
23.     mHandPoseTemp.fingerAngles[i] -= DEF_FINGER_ANGLES[i];
24.     mHandPoseTemp.fingerAngles[i] *= -MathUtil.PRE_180_DIV_PI;
25. }
26. ...
```

Fonte: elaborado pelo autor.

Após a obtenção da tangente da reta, o valor é subtraído pelo ângulo de rotação da palma da mão (`palmRotationZ`). O resultado da subtração anterior é subtraído pelo valor correspondente ao dedo em questão no vetor `DEF_FINGER_ANGLES`. Os valores do vetor `DEF_FINGER_ANGLES` foram definidos previamente com base na tangente dos ângulos formados pelos segmentos de retas ilustrados na Figura 28. Onde os segmentos de reta de cor: branca, amarela, vermelha, azul e verde representam os dedos: polegar, indicador, dedo médio, dedo anelar e dedo mínimo, respectivamente.

Figura 28 – Ângulos de rotação padrão

Fonte: elaborado pelo autor.

Por fim, o valor obtido através da última subtração é multiplicado pela constante `PRE_180_DIV_PI` para que resultado final possa ser convertido de radianos para graus. Esta etapa é repetida até que o valor de rotação no eixo Z de todos os dedos sejam obtidos e atualizados no objeto temporário `mHandPoseTemp`.

Nas etapas finais de execução, o objeto temporário `mHandPoseTemp` tem o conteúdo copiado para o objeto compartilhado `mHandPose`, cujo o acesso é controlado através de um semáforo. No Quadro 18 é apresentado a implementação da função `updateHandPose`, responsável por atualizar as informações do objeto `mHandPose`. Depois de atualizadas as informações do objeto `mHandPose`, a *thread* então desenha a imagem original no *layer* inferior do aplicativo.

Quadro 18 – Atualizando o objeto compartilhado

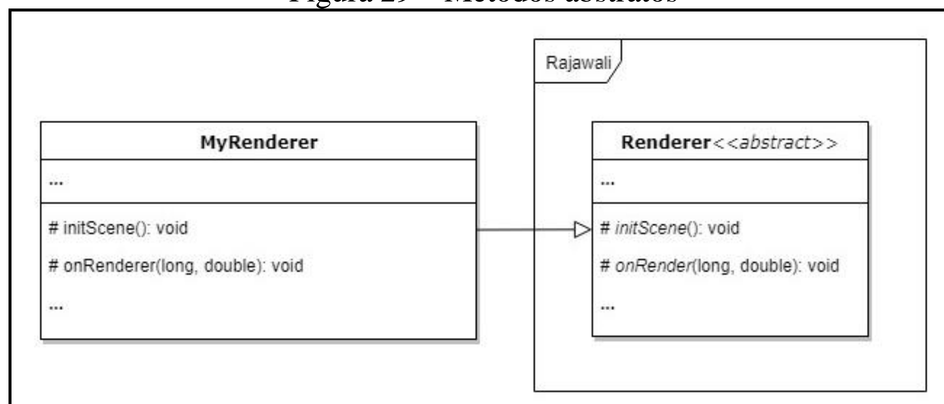
```
1.  public void updateHandPose()
2.  {
3.      // Atualiza o objeto compartilhado real
4.      try
5.      {
6.          mSemaphore.acquire();
7.          mHandPose.copyFrom(mHandPoseTemp);
8.          mSemaphore.release();
9.      }
10.     catch (InterruptedException e)
11.     {
12.         e.printStackTrace();
13.     }
14.     mHandPoseTemp.render = false;
15. }
```

Fonte: elaborado pelo autor.

3.3.2 Renderização do modelo 3D

A execução da *thread* de renderização do modelo 3D começa a partir do momento que uma instância do motor gráfico Rajawali é associada a um *layer* do aplicativo, neste caso, o *layer* superior. O gerenciamento da *thread* é então feito inteiramente pelo motor gráfico na classe `Renderer`. Após realizar suas inicializações internas, o motor gráfico invoca o método `initScene` para que possam ser instanciados os objetos a serem utilizados durante a renderização da cena. A implementação do método abstrato `initScene` é feita na classe `MyRenderer` juntamente com a implementação do método `onRenderer`, como é ilustrado na Figura 29.

Figura 29 – Métodos abstratos



Fonte: elaborado pelo autor.

Durante a etapa de inicialização da cena é definido uma matriz de projeção para a câmera da cena se houver, como mostra o Quadro 19.

Quadro 19 – Inicialização da matriz de projeção

```
1.  ...
2.  // Inicializa a matriz de projeção da câmera
3.  if (mProjMat != null)
4.      getCurrentCamera().setProjectionMatrix(mProjMat);
5.  ...
```

Fonte: elaborado pelo autor.

A construção desta matriz de projeção é feita através da função `getProjectionMat` que leva em consideração alguns parâmetros da câmera real do dispositivo. Estes parâmetros sendo: `mScreenWidth` (largura da imagem), `mScreenHeight` (altura da imagem) e `mFOVX` (*field of view* horizontal), como mostra o Quadro 20.

Quadro 20 – Construção da matriz de projeção da câmera

```
1. public float[] getProjectionMat()
2. {
3.     if (mProjectionGL == null)
4.     {
5.         mProjectionGL = new float[16];
6.
7.         float aspectR = (float) mScreenWidth / (float) mScreenHeight;
8.         float right = (float) Math.tan(0.5f * mFOVX * Math.PI / 180.0f) * NEAR;
9.         float top = right / aspectR;
10.        Matrix.frustumM(mProjectionGL, 0, -right, right, -top, top, NEAR, FAR);
11.    }
12.    return mProjectionGL;
13. }
```

Fonte: elaborado pelo autor.

O resultado da função `getProjectionMat` é então devolvido como uma matriz 4x4 pela variável `mProjectionGL`, representado a perspectiva da câmera. Esta matriz é recalculada toda vez que a câmera é inicializada ou reconfigurada, por isso, foi criado uma solução baseada no padrão de projeto *observer*. Portanto, toda vez que a matriz de projeção é recalculada pela função `getProjectionMat`, a classe `MyRenderer` é notificada através do método `onProjectionChanged` e atualiza a variável `mProjMat` com a nova matriz. Caso a cena já tenha sido inicializada, então a matriz é atualizada na câmera da cena também. No Quadro 21 é possível observar a implementação do método `onProjectionChanged`.

Quadro 21 – Método onProjectionChanged

```
1. @Override
2. public void onProjectionChanged(int width, int height, float[] projectionMat)
3. {
4.     // Converte de float[] para Matrix4
5.     if (mProjMat == null)
6.         mProjMat = new Matrix4(projectionMat);
7.     else
8.         mProjMat.setAll(projectionMat);
9.
10.    // Caso a cena já tenha sido inicializada, atualiza a matriz da cena
11.    if (mSceneInitialized)
12.        getCurrentCamera().setProjectionMatrix(mProjMat);
13. }
```

Fonte: elaborado pelo autor.

Depois de definida a matriz de projeção da câmera, são instanciadas três matrizes temporárias. Estas matrizes são utilizadas durante a renderização da cena para auxiliar nos cálculos das transformações do modelo 3D. Em seguida, são instanciadas as matrizes *ModelView* de cada dedo e da palma da mão, além de um objeto de luz básico para

visualização da cena. No Quadro 22 é possível conferir o trecho de código desta etapa da inicialização.

Quadro 22 – Inicialização dos objetos da cena

```
1.  ...
2.  
3.  // Inicializa as matrizes temporárias
4.  mTempTransf = new Matrix4();
5.  mTempViewMat = new Matrix4();
6.  mTempModelMat = new Matrix4();
7.  
8.  // Inicializa as matrizes ModelView de cada dedo
9.  mFingerModelViewMat = new Matrix4[5];
10. mFingerModelViewMat[0] = new Matrix4();
11. mFingerModelViewMat[1] = new Matrix4();
12. mFingerModelViewMat[2] = new Matrix4();
13. mFingerModelViewMat[3] = new Matrix4();
14. mFingerModelViewMat[4] = new Matrix4();
15. 
16. // Inicializa a matriz ModelView da palma
17. mPalmModeViewMat = new Matrix4();
18. 
19. // Inicializa a luz da cena
20. mDirectionalLight = new DirectionalLight(4, 4, 4);
21. mDirectionalLight.setColor(1.0f, 1.0f, 1.0f);
22. mDirectionalLight.setPower(1.25f);
23. getCurrentScene().addLight(mDirectionalLight);
24. 
25. ...
```

Fonte: elaborado pelo autor.

Na sequência, o modelo esquelético da mão esquerda em três dimensões é carregado juntamente com o arquivo MTL contendo o material utilizado pelo modelo. O modelo carregado é então atribuído a variável `mLeftHandModel` e adicionado a cena, conforme demonstrado no Quadro 23.

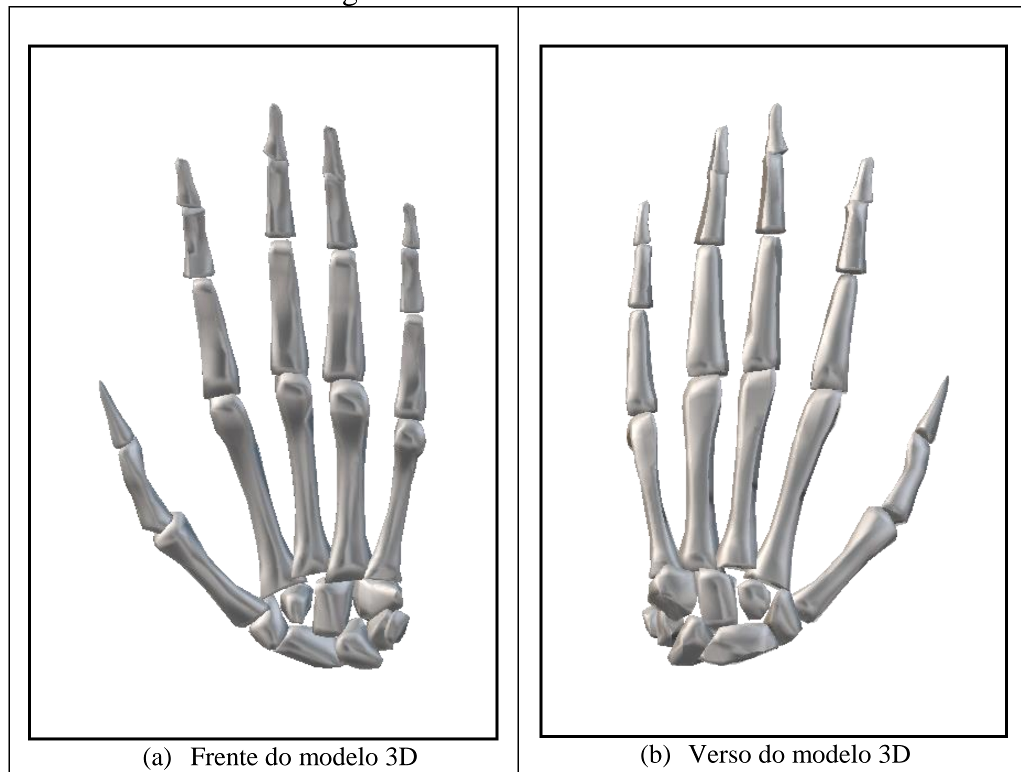
Quadro 23 – Carregando o modelo 3D

```
1.  ...
2.  
3.  // Carrega o arquivo .OBJ e .MTL
4.  LoaderOBJ loaderOBJ = new LoaderOBJ(this, R.raw.lefthand_obj);
5.  try
6.  {
7.      loaderOBJ.parse();
8.  }
9.  catch (ParsingException e)
10. {
11.     e.printStackTrace();
12. }
13. 
14. // Obtém a instância do objeto carregado
15. mLeftHandModel = loaderOBJ.getParsedObject();
16. 
17. // Adiciona o objeto na cena
18. getCurrentScene().addChild(mLeftHandModel);
19. 
20. ...
```

Fonte: elaborado pelo autor.

O modelo esquelético carregado é formado por 27 diferentes objetos, cada um representado um osso especifico do sistema esquelético da mão. Na Figura 30 é possível visualizar o modelo de frente e verso.

Figura 30 – Modelo 3D da mão



(a) Frente do modelo 3D (b) Verso do modelo 3D

Fonte: elaborado pelo autor.

Depois de carregado o modelo 3D da mão, o vetor `mBoneFingerIndex` é inicializado com o número de objetos presentes no modelo 3D, portanto 27. Em seguida cada objeto do modelo 3D é percorrido e classificado. Esta classificação é obtida através da função `getFingerIndex` como é apresentado no Quadro 24.

Quadro 24 – Identificação dos objetos carregados

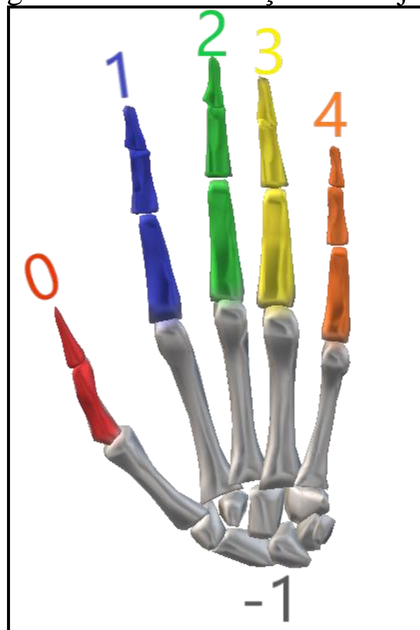
```
1.  ...
2.
3.  int i, j = mLeftHandModel.getNumChildren();
4.  mBoneFingerIndex = new int[j];
5.  for (i = 0; i < j; ++i)
6.  {
7.      // Permite que seja utilizada uma matriz customizada
8.      mLeftHandModel.getChildAt(i).setUseCustomModelView(true);
9.
10.     // Classifica o objeto
11.  mBoneFingerIndex[i] = getFingerIndex(mLeftHandModel.getChildAt(i).getName());
12. }
13.
14. ...
```

Fonte: elaborado pelo autor.

A classificação retornada pela função `getFingerIndex` é um número inteiro entre -1 e 4. Onde, -1 indica que o modelo informado para a função faz parte da região da palma ou

pulso e números entre 0 e 4 indicam que o modelo faz parte de um dedo, assim como é ilustrado na Figura 31.

Figura 31 – Classificação dos objetos



Fonte: elaborado pelo autor.

No Quadro 25 é apresentado o trecho de código da etapa final da inicialização, onde o objeto que contém o modelo 3D é definido como invisível na cena e o contador `mBadFrames` é iniciado com o valor da constante `MAX_BAD_FRAMES` (8).

Quadro 25 – Etapa final da inicialização

```
1.      ...
2.
3.      // Define o modelo como invisível na cena
4.      mLeftHandModel.setVisible(false);
5.
6.      // Inicializa o contador
7.      mBadFrames = MAX_BAD_FRAMES;
8.  } // Fim da inicialização
```

Fonte: elaborado pelo autor.

Na etapa seguinte, o método `onRender` da classe `MyRenderer` é invocado pela *thread* do motor gráfico toda vez que um frame precisa ser desenhado no *layer*. Esta invocação permite que a imagem possa ser renderizada na tela do dispositivo de forma sucessiva. No início da função `onRender` o objeto compartilhado com a *thread* de identificação da mão é lido e os valores atualizados na variável `mHandPose`. Em seguida são feitas duas comparações em relação as informações obtidas em `mHandPose`. A primeira comparação verifica se a *flag* `mHandPose.render` é verdadeira e a segunda, se o contador `mBadFrames` é menor que a constante `MAX_BAD_FRAMES` (ou 8). Apresentando uma das duas condições como verdadeira a próxima etapa atualiza o contador `mBadFrames`, caso contrário o modelo 3D é ocultado da cena e a função `onRender` retorna. Na atualização do contador `mBadFrames`, é levado em

consideração novamente o conteúdo da *flag* `mHandPose.render`. Caso a *flag* seja verdadeira o contador tem o seu valor definido como zero, caso contrário seu valor é incrementado em um, como é demonstrado no trecho de código do Quadro 26.

Quadro 26 – Etapa inicial da renderização

```
1.  ...
2.
3.  // Lê as informações do objeto compartilhado
4.  mHandPose = mHandTracking.getObjStatus();
5.
6.  if (mHandPose.render || mBadFrames < MAX_BAD_FRAMES)
7.  {
8.      // Atualiza o valor do contador
9.      if (mHandPose.render)
10.         mBadFrames = 0;
11.     else
12.         ++mBadFrames;
13.
14.     ...
```

Fonte: elaborado pelo autor.

O contador `mBadFrames` tem a finalidade de reduzir o efeito de *flickering* durante a visualização. Em outras palavras, impedir que o usuário do aplicativo veja o modelo 3D piscando na cena. Este efeito acontece quando a pose é detectada em um frame, no frame seguinte não e no próximo frame volta a ser detectada. Desta forma, o contador permite que o modelo continue sendo visualizado continuamente por pelo menos mais 8 frames (valor da constante `MAX_BAD_FRAMES`).

Na sequência, o modelo 3D da mão tem a visibilidade alterada para aparecer na cena e as matrizes *ModelView* da palma e dos dedos são atualizadas. A *ModelView* da palma é atualizada com os valores da pose obtidos através do objeto compartilhado. Já as matrizes *ModelView* dos dedos são calculadas pela função `calculateFingerTransf` como é exibido no Quadro 27.

Quadro 27 – Atualização das matrizes ModelView

```
1.  ...
2.
3.  // Muda a visibilidade do objeto
4.  if (!mLeftHandModel.isVisible())
5.      mLeftHandModel.setVisible(true);
6.
7.  // Atualiza a ModelView da palma com a nova pose
8.  mPalmModeViewMat.setAll(mHandPose.pose);
9.
10. // Calcula as ModelView's dos dedos
11. calculateFingerTransf(mPalmModeViewMat);
12.
13. ...
```

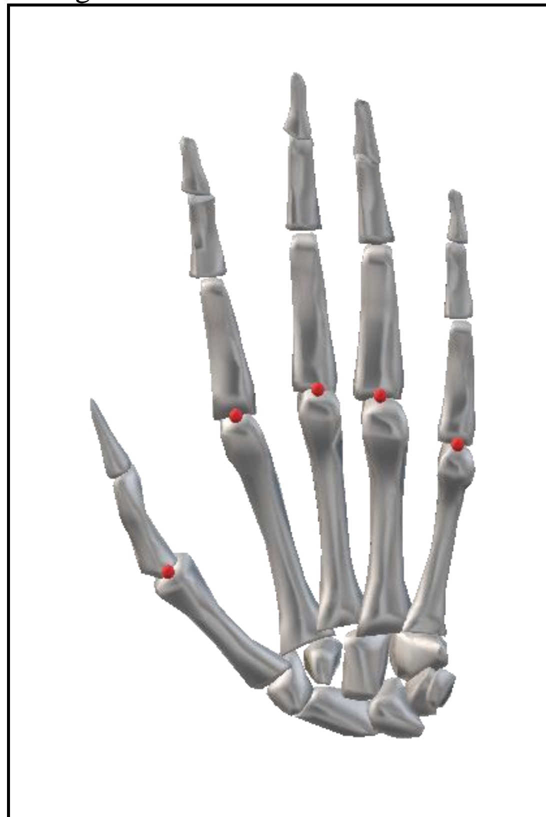
Fonte: elaborado pelo autor.

A função `calculateFingerTransf` leva como parâmetro a pose da palma e tem como proposito calcular a pose de cada dedo individualmente. Durante cada interação do loop na

função, a matriz temporária `mTempModelMat` tem como papel auxiliar no cálculo da matriz *Model* de cada dedo. Inicialmente, a matriz é definida como identidade, e ao longo da execução são aplicadas diversas transformações. A primeira delas hipoteticamente translada os ossos de um dedo para origem, para em seguida rotacionar estes ossos no eixo Z pelo ângulo de rotação obtido do objeto compartilhado (`fingerAngles[i]`). Na sequência, os ossos do dedo são transladados de volta para a posição original e então são aplicadas as mesmas transformações da palma para a matriz *View* temporária (`mTempViewMat`).

Para transladar os ossos dos dedos de uma posição para o outra, foi elaborada uma matriz de tamanho 5x2 denominada `REF_FINGER_PTS`. Esta matriz contém coordenadas em duas dimensões que representam o ponto inicial de cada dedo no modelo 3D. Na Figura 32 estes pontos são ilustrados como pequenas esferas em vermelho.

Figura 32 – Pontos inicias dos dedos



Fonte: elaborado pelo autor.

No final da interação de um loop da função `calculateFingerTransf`, são combinadas as transformações da matriz temporária `mTempModelMat` com as da matriz `mTempViewMat` gerando assim a matriz *ModelView* de um dedo. No Quadro 28 é possível conferir a implementação integral da função `calculateFingerTransf`.

Quadro 28 – Calcula a pose dos dedos

```
1.  private void calculateFingerTransf(Matrix4 palmPose)
2.  {
3.      for (int i = 0; i < 5; ++i)
4.      {
5.          // Carrega identidade
6.          mTempModelMat.identity();
7.  
8.          // Translada para a origem
9.          mTempTransf.identity().setTranslation(- REF_FINGER_PTS[i][0], -
     REF_FINGER_PTS[i][1], 0);
10.         mTempModelMat.leftMultiply(mTempTransf);
11. 
12.         // Rotaciona no eixo Z
13.         mTempTransf.identity().setRotate(0,0,1, mHandPose.fingerAngles[i]);
14.         mTempModelMat.leftMultiply(mTempTransf);
15. 
16.         // Translada de volta
17.         mTempTransf.identity().setTranslation(REF_FINGER_PTS[i][0],
18.                                               REF_FINGER_PTS[i][1], 0);
19.         mTempModelMat.leftMultiply(mTempTransf);
20. 
21.         // Aplica as transformações da palma
22.         mTempViewMat.identity().inverse();
23.         mTempViewMat.leftMultiply(palmPose);
24. 
25.         // Gera a ModelView do dedo(i)
26.         mFingerModelViewMat[i].setAll(mTempViewMat).multiply(mTempModelMat);
27.     }
28. }
```

Fonte: elaborado pelo autor.

Para que o modelo 3D possa reproduzir a pose da mão na cena, é necessário atualizar a matriz *ModelView* de todos ossos que compõem o modelo esquelético da mão. É por isso que na última etapa da função `onRender` a pose de todos os objetos do modelo são atualizados com suas respectivas matrizes *ModelView* calculadas previamente. No Quadro 29 é apresentado o trecho de código responsável pela atualização das poses. Ao final desta etapa a função `onRender` retorna e então o motor gráfico renderiza a cena no devido *layer*.

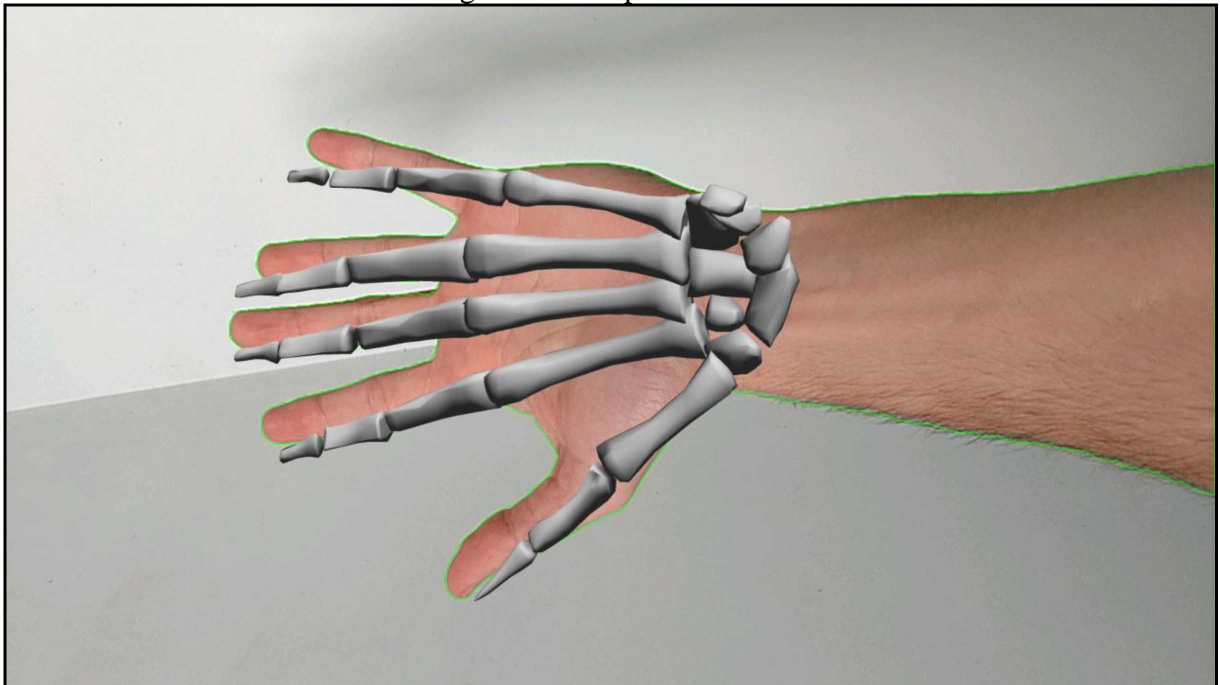
Quadro 29 – Atualização da pose do modelo 3D

```
1.  ...
2.  
3.  // Atualiza todos os objetos do modelo 3D com a nova ModelView
4.  for (int i = 0, j = mLeftHandModel.getNumChildren(), fingerIndex; i < j; ++i)
5.  {
6.      fingerIndex = mBoneFingerIndex[i];
7.      if (fingerIndex != -1) // Objeto de algum dedo
8.      {
9.          mLeftHandModel.getChildAt(i).getModelViewMatrix().setAll(
10.                                     mFingerModelViewMat[fingerIndex]);
11.     }
12.     else // Objeto da palma
13.     {
14.         mLeftHandModel.getChildAt(i).getModelViewMatrix().setAll(
15.                                                   mPalmModeViewMat);
16.     }
17. }
18. 
19. ...
```

Fonte: elaborado pelo autor.

Na Figura 33 é possível visualizar uma captura de tela do protótipo renderizando o modelo 3D com a pose identificada da mão.

Figura 33 – Captura de tela



Fonte: elaborado pelo autor.

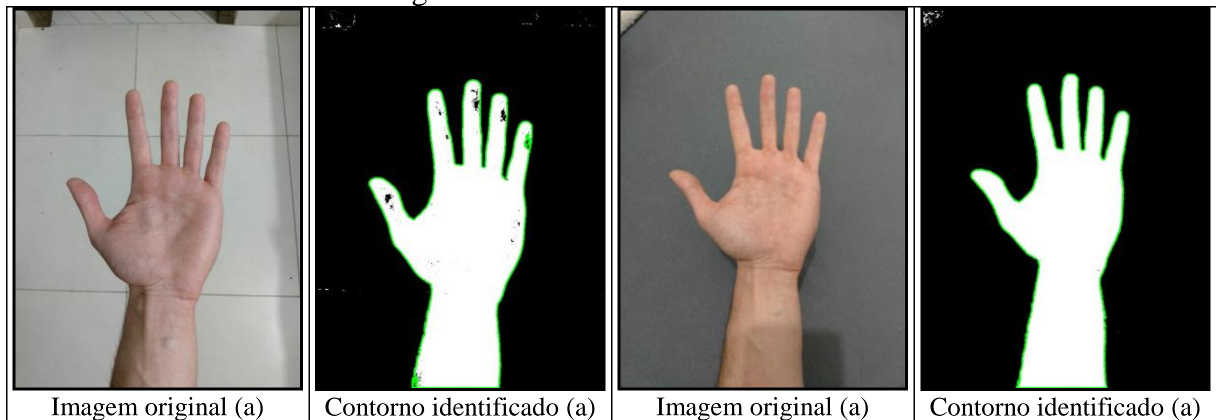
3.4 ANÁLISE DOS RESULTADOS

Nesta seção serão apresentados os resultados obtidos na execução do protótipo. A seção 3.4.1 demonstra os resultados obtidos através da segmentação da imagem com base na cor da pele. Na seção 3.4.2 são apresentados os resultados obtidos após identificar os principais pontos convexos do contorno. Na seção 3.4.3 são apresentados os resultados da renderização do modelo 3D do sistema esquelético com a pose estimada da mão. Por fim, a seção 3.4.4 apresenta as modificações e os resultados obtidos através da otimização do módulo de câmera da biblioteca OpenCV.

3.4.1 Identificando o contorno de interesse

As imagens utilizadas foram capturadas a partir de diversos tipos de ambientes e condições de iluminação. Na primeira etapa dos testes as imagens foram capturadas em 2 ambientes diferentes, porém controlados e com boa iluminação. Nas imagens da Figura 34 é possível analisar os resultados destes ambientes, onde o contorno em verde é o contorno identificado pelo protótipo na imagem original.

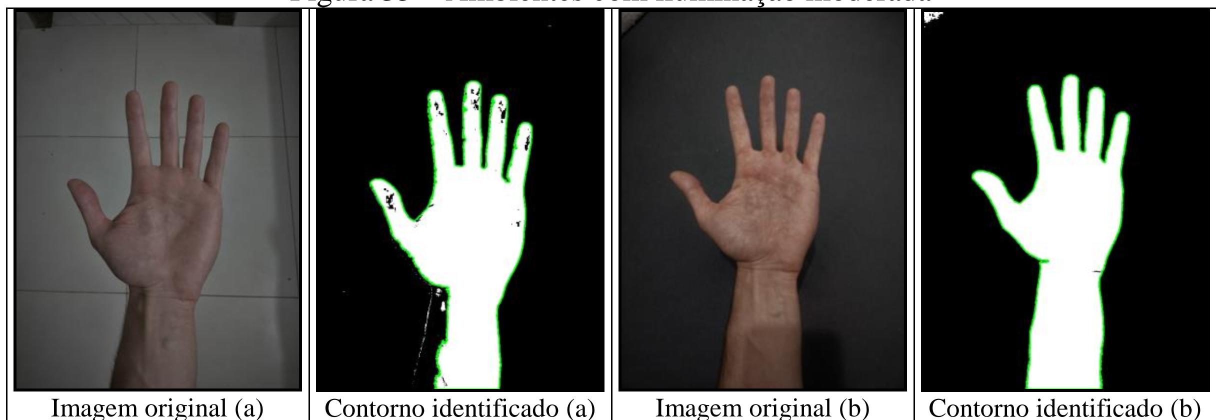
Figura 34 – Ambientes controlados



Fonte: elaborado pelo autor.

Na Figura 35 os testes foram realizados com as mesmas imagens do teste anterior, porém com a iluminação reduzida através de um editor de imagens. Neste caso, o contorno identificado na Figura 35a teve a região do antebraço um pouco comprometida, mas nada que afetasse o processamento de imagem. Já no resultado da Figura 35b o resultado foi bem similar ao obtido na etapa anterior onde à iluminação era melhor, pois o fundo é bem contrastante.

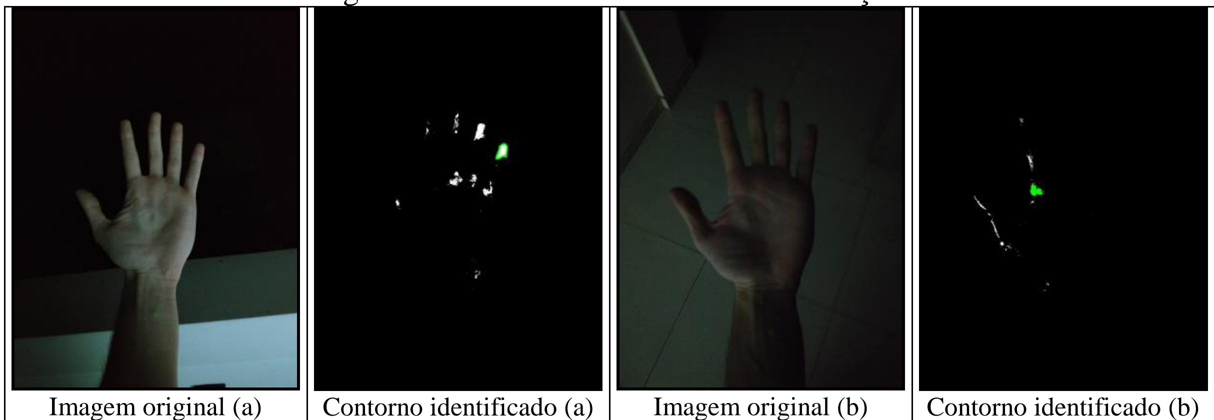
Figura 35 – Ambientes com iluminação moderada



Fonte: elaborado pelo autor.

Na Figura 36, as imagens foram obtidas em 2 ambientes controlados, mas com baixa iluminação. Neste caso, os contornos produzidos por estas imagens, nesses ambientes, são insuficientes para que o processamento da imagem possa identificar a pose da mão.

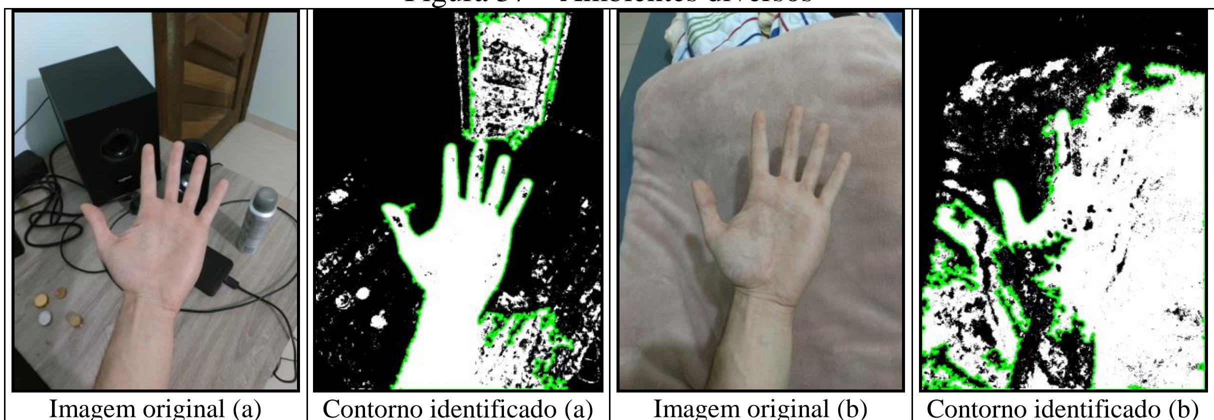
Figura 36 – Ambientes com baixa iluminação



Fonte: elaborado pelo autor.

Por último, na Figura 37 as imagens foram obtidas em ambientes diversos e não controlados, mas com boa iluminação. Em ambos os casos os contornos obtidos apresentaram muito ruído para serem utilizados.

Figura 37 – Ambientes diversos



Fonte: elaborado pelo autor.

Conclui-se que o protótipo é capaz de reconhecer com facilidade o contorno da mão em ambientes controlados onde a iluminação é boa ou moderada. Em ambientes diversos, onde aparecem muitos elementos com cores próximas a tonalidade da cor da pele, o protótipo foi incapaz de identificar o contorno de interesse.

3.4.2 Identificando os principais pontos convexos

As imagens obtidas para a realização destes testes foram tiradas de um mesmo ambiente. O ambiente apresentava boa iluminação e era controlado. Na Figura 38a o dedo indicador, médio, anelar e mínimo ficaram bem próximos uns dos outros, mas sem se encostarem. Neste caso, o protótipo foi capaz de identificar todos os pontos convexos principais necessários para a identificação da pose, da mesma forma no caso da Figura 38b.

Figura 38 – Pose como os dedos separados

Imagem original (a)	Pontos convexos identificados (a)	Imagem original (b)	Pontos convexos identificados (b)

Fonte: elaborado pelo autor.

Na Figura 39 as imagens foram tiradas de uma mão fazendo 2 gestos diferentes. Na Figura 39a os dedos ficaram muito agrupados, resultando na extração de apenas três pontos convexos. Já na Figura 39b o dedo anelar ficou agrupado com o dedo mínimo, impedindo o protótipo de identificar todos os pontos convexos principais necessários.

Figura 39 – Pose com os dedos juntos

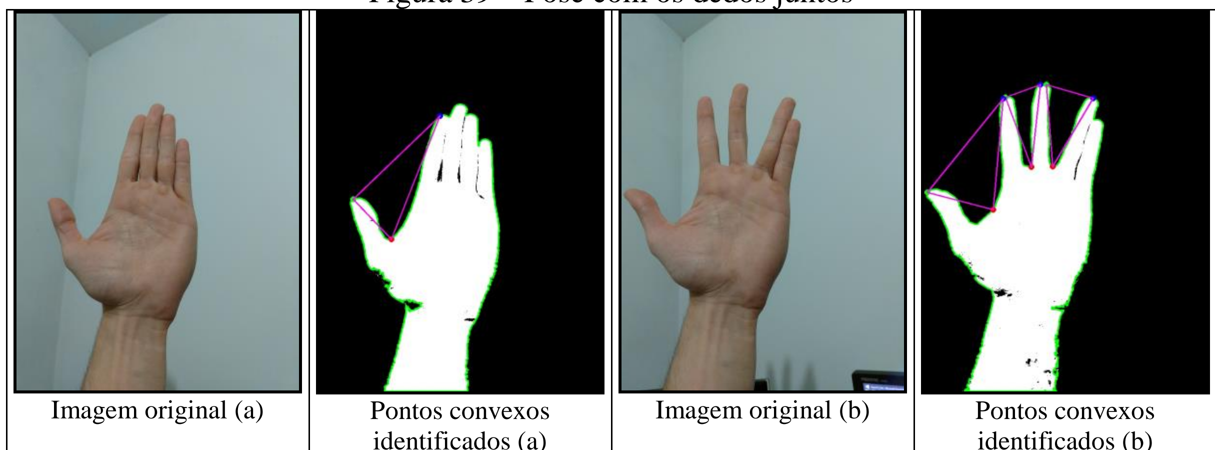


Imagem original (a)	Pontos convexos identificados (a)	Imagem original (b)	Pontos convexos identificados (b)

Fonte: elaborado pelo autor.

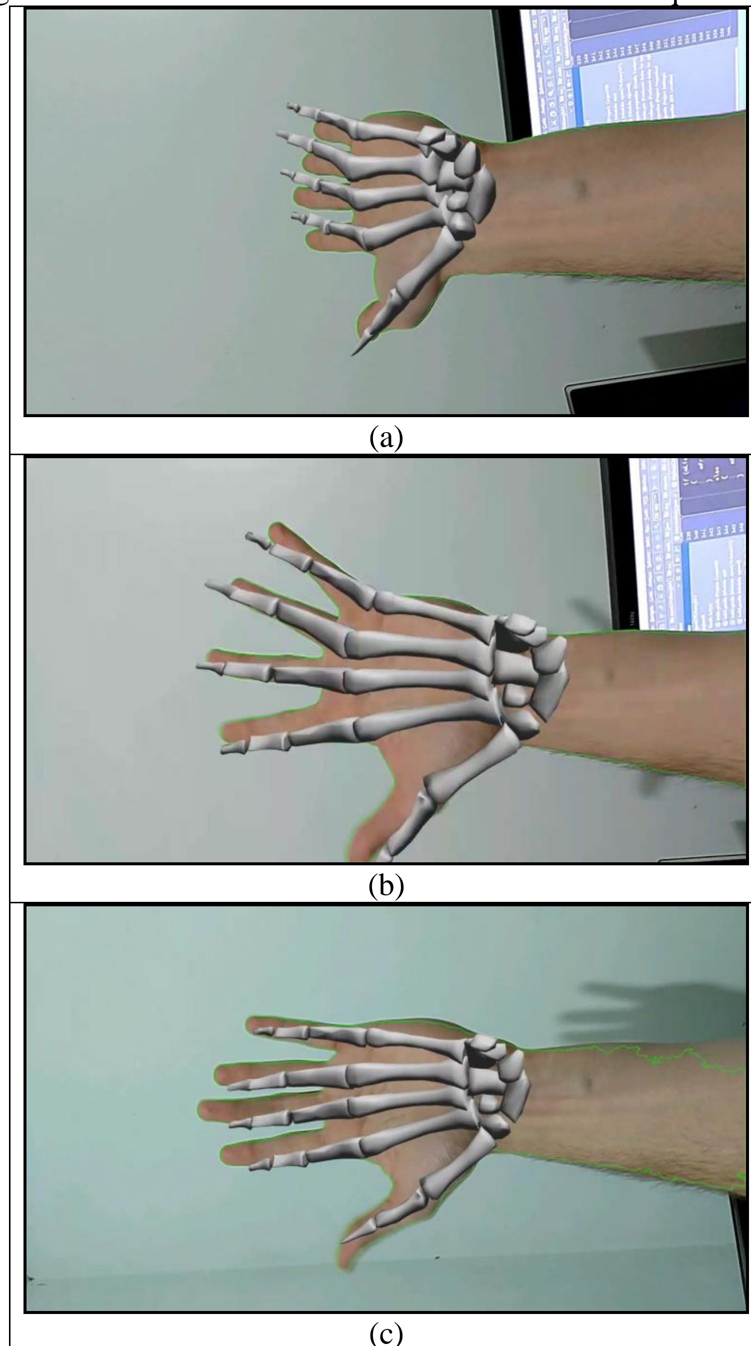
Conclui-se que o protótipo é capaz de identificar os principais pontos convexos da mão em situações onde os dedos não se apresentam agrupados de nenhuma forma durante a pose. Caso contrário, o protótipo não é capaz de identificar os principais pontos convexos necessário para seguir com o processamento.

3.4.3 Renderização do modelo 3D do sistema esquelético com a pose da mão

As imagens apresentadas na Figura 40 foram obtidas diretamente do protótipo em execução. Durante a execução foram realizados diferentes gestos com a mão a fim de visualizar o modelo 3D de diferentes perspectivas. De modo geral, a pose exibida pelo modelo 3D encontrou-se bem alinhada com a imagem da mão, mostrando apenas

desalinhamentos maiores na região dos dedos. Enquanto nas regiões da palma da mão e pulso os desalinhamentos do modelo 3D foram bem menores.

Figura 40 – Alinhamento do Modelo 3D do sistema esquelético



(a)

(b)

(c)

Fonte: elaborado pelo autor.

Conclui-se que o protótipo foi capaz de renderizar o modelo 3D do sistema esquelético sobre a imagem da câmera. Porém, foram identificados desalinhamentos entre o modelo 3D e a imagem, especialmente na região dos dedos.

3.4.4 Otimização do módulo de câmera da biblioteca OpenCV

Para aperfeiçoar a taxa de quadros por segundo que o protótipo consegue processar pela *thread* de identificação da mão, foram feitas algumas alterações nos fontes disponibilizados pela biblioteca de visão computacional OpenCV. Portanto, nesta seção serão apresentadas as modificações feitas, começando pelas alterações na etapa de inicialização da *thread*. Em seguida, serão apresentas as modificações feitas durante a etapa de execução da *thread*, e ao final, será feito um comparativo de desempenho entre a versão original e a modificada.

Durante a etapa de inicialização da *thread* foi adicionado um novo parâmetro para a câmera do dispositivo. Este parâmetro é chamado de *preview fps range* e representa um intervalo mínimo e máximo de números inteiros. O intervalo serve para informar a câmera a meta de quadros por segundo que o aplicativo deseja trabalhar. No Quadro 30 é apresentada a chamada da função que modifica este intervalo com o valor de duas constantes: `MIN_FPS` (ou 30000) e `MAX_FPS` (ou 30000).

Quadro 30 – Parâmetro de câmera

```
1.  params.setPreviewFpsRange(MIN_FPS, MAX_FPS);
```

Fonte: elaborado pelo autor.

Na sequência, foi modificado a função de alocação da imagem *cache* utilizada pela *thread*. Anteriormente, a alocação da *cache* era feita passando a resolução da imagem obtida pela câmera. Agora, a alocação informa a resolução da tela no lugar. Com essa alteração a imagem do *cache* fica com o tamanho significativamente maior, porém na hora de desenhar a imagem no *layer*, a imagem não precisa ser redimensionada toda vez, pois já possui o tamanho exato da tela. No Quadro 31 é possível observar a função de alocação da imagem *cache* depois da alteração.

Quadro 31 – Alocação da imagem cache

```
1.  protected void AllocateCache()
2.  {
3.      // Anteriormente: mCacheBitmap = Bitmap.createBitmap(mFrameWidth,
4.      // mFrameHeight, Bitmap.Config.ARGB_8888);
5.      mCacheBitmap = Bitmap.createBitmap(mScreenWidth, mScreenHeight,
6.                                                     Bitmap.Config.ARGB_8888);
7.  }
```

Fonte: adaptado de OpenCV(2018c).

Durante a execução da *thread*, cada *frame* novo obtido pela câmera era passado como parâmetro para a função `deliverAndDrawFrame`. Dentro do escopo desta função haviam diversos objetos sendo inicializados toda vez que a função era chamada. Para otimizar o desempenho, estas inicializações foram removidas do escopo da função. Objetos usados pela função foram adicionados como membro privado da classe. Além disso, a chamada do

método que desenha no *layer* foi alterada para simplesmente desenhar a imagem *cache*. Visto que anteriormente a chamada do método incluía parâmetros informando o tamanho para qual a imagem deveria ser redimensionada. No Quadro 32 é apresentado o trecho de código da nova chamada da função de desenho.

Quadro 32 – Desenho da imagem cache

```
1.  ...
2.
3.  //Anteriormente: canvas.drawBitmap(mCacheBitmap, new Rect(0, 0,
4.  mCacheBitmap.getWidth(), mCacheBitmap.getHeight()), new Rect((canvas.getWidth() -
     mCacheBitmap.getWidth()) / 2,(canvas.getHeight() - mCacheBitmap.getHeight())/2,
5.  (canvas.getWidth() - mCacheBitmap.getWidth()) / 2 + mCacheBitmap.getWidth(),
6.  (canvas.getHeight() - mCacheBitmap.getHeight()) / 2 +
7.  mCacheBitmap.getHeight()), null);
8.  mCanvas.drawBitmap(mCacheBitmap, 0, 0, null);
9.
10. ...
```

Fonte: adaptado de OpenCV(2018c).

Para gerar os números fornecidos na tabela comparativa foram feitas adaptações em uma função chamada `measure`. A função era responsável por gerar apenas a taxa de quadros por segundo do aplicativo e foi adaptada para gerar a média de FPS, mínima de FPS e máxima de FPS. No trecho de código do Quadro 33 é apresentado o trecho da função `measure` adaptado.

Quadro 33 – Cálculos estatísticos

```
1.  ...
2.
3.  // Contador de frames
4.  ++mFramesCounter;
5.
6.  // Cálculo da Média de FPS
7.  mAvgFPS += (mFPS - mAvgFPS) / mFramesCounter;
8.
9.  // Delay para autualizar o FPS
10. if (mFramesCounter % STEP == 0)
11. {
12.     // Tempo atual
13.     mTime = Core.getTickCount();
14.
15.     // Cálculo do FPS
16.     mFPS = STEP * mFrequency / (mTime - mPrevFrameTime);
17.
18.     // Atualiza o valor mínimo e máximo de FPS
19.     if (mFPS < mMinFPS)
20.         mMinFPS = mFPS;
21.     if (mFPS > mMaxFPS)
22.         mMaxFPS = mFPS;
23.
24.     mPrevFrameTime = mTime;
25.
26.     // Atualiza o texto exibido na tela
27.     mStrFPS = FPS_FORMAT.format(mFPS) + " FPS / " + FPS_FORMAT.format(mAvgFPS) +
28.               " AVG. FPS / " + FPS_FORMAT.format(mMinFPS) + " MIN. FPS / " +
29.                                   FPS_FORMAT.format(mMaxFPS) + " MAX. FPS";
30. }
31. ...
```

Fonte: adaptado de OpenCV(2018c).

Para medir a diferença de performance entre a versão do aplicativo com os arquivos fontes originais do OpenCV *versus* os fontes otimizados do OpenCV, foi realizado um teste com cada versão. Cada teste consistiu em rodar o aplicativo por três minutos seguidos focando a mesma imagem com as mesmas condições de iluminação. A imagem em que o aplicativo processava era a de uma parede e para manter o foco, o dispositivo foi fixado em um tripé. O dispositivo usado em ambos os testes foi smartphone Moto Z XT1650-03 no sistema operacional Android 8.0. Na Figura 41 é ilustrado a captura de tela do dispositivo no momento final do teste, onde as estatísticas de FPS aparecem em verde no topo da imagem.

Figura 41 – Tela do aplicativo no final do teste

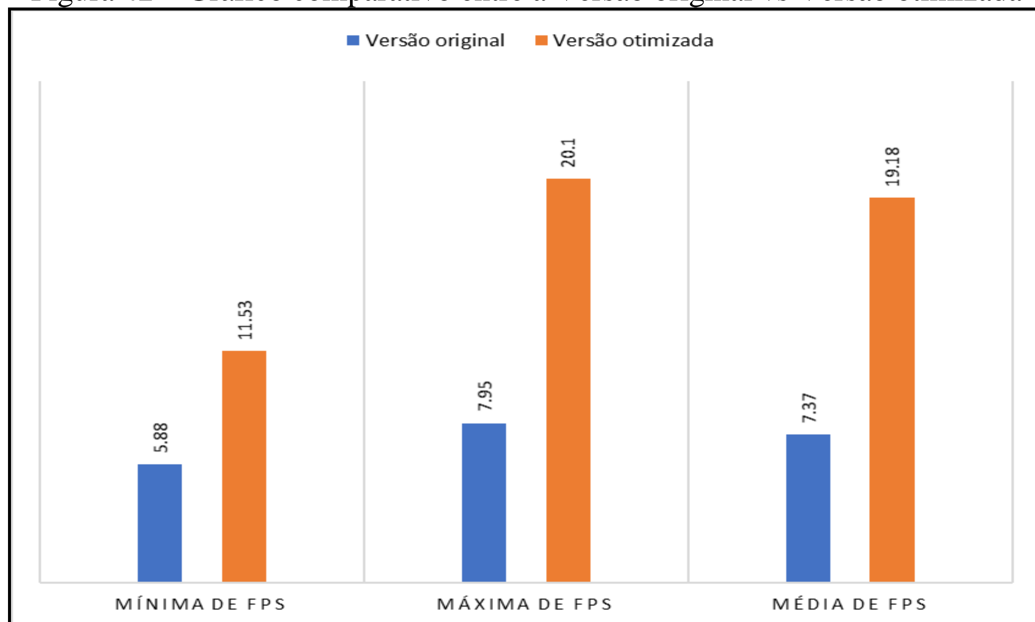
(a) Teste com o fonte original

(b) Teste com o fonte otimizado

Fonte: elaborado pelo autor.

No gráfico da Figura 42 foram compilados os resultados observados na captura de tela do final do teste. Como é possível observar, o ganho de FPS foi notavelmente maior na versão com o fonte otimizado.

Figura 42 – Gráfico comparativo entre a Versão original vs Versão otimizada



Fonte: elaborado pelo autor.

Conclui-se que o protótipo obteve um desempenho consideravelmente maior na versão otimizada do modulo de câmera do OpenCV. O protótipo foi capaz de manter uma taxa média de 19.68 FPS durante os três minutos de teste, se comparado com a versão original, apresenta um ganho de desempenho de aproximadamente 160%.

4 CONCLUSÕES

A proposta deste trabalho era desenvolver um protótipo para visualização da anatomia em três dimensões em tempo real, fazendo uso da realidade aumentada. No desenvolvimento do protótipo foi feito uso da linguagem de programação JAVA juntamente com a biblioteca de visão computacional OpenCV e o motor gráfico Rajawali. A biblioteca OpenCV teve como relevância auxiliar no processo de identificação da mão, enquanto a biblioteca Rajawali ajudou com a renderização do modelo esquelético da mão.

O primeiro objetivo foi atendido parcialmente, em muitos casos o protótipo teve dificuldade em identificar o contorno de uma imagem. Durante os testes de identificação do contorno foram examinados diferentes ambientes com diferentes condições de iluminação. Quando o ambiente apresentava condições boas ou até moderadas de iluminação, o protótipo foi capaz de identificar com facilidade o contorno da mão. Já em lugares com baixa iluminação, o protótipo não foi capaz de identificar a pose da mão. Nos testes realizados em ambientes diversos, o protótipo encontrou muito ruído na hora de identificar o contorno na maioria dos casos.

O segundo objetivo também foi atendido parcialmente, muitos gestos não são reconhecidos com o método empregado. Nos testes realizados para identificar os principais pontos convexos, todas as imagens obtidas foram tiradas de um ambiente controlado com boa iluminação. Quando os dedos se apresentavam muito agrupados de alguma forma, o protótipo não foi capaz de identificar os principais pontos convexos. Porém, quando havia certa distância entre os dedos, mesmo que pequena, o protótipo foi capaz de identificar os principais pontos convexos necessários para a estimação da pose da mão. Da mesma forma, o protótipo foi capaz de identificar os principais pontos convexos quando a mão apresentava bastante distância entre os dedos.

O terceiro objetivo foi atendido, porém a visualização do modelo do sistema esquelético 3D se tornou possível durante os testes quando a mão aparecia por completo na imagem da câmera. O modelo 3D também apresentou certo desalinhamento em relação a imagem da câmera, especialmente na região dos dedos. Na região da palma da mão e do pulso os resultados foram melhores, apresentando pouco desalinhamento.

O quarto objetivo foi atendido totalmente. Nos testes de desempenho realizados para comparar a versão otimizada com a versão original do módulo de câmera do OpenCV, foram geradas duas versões do protótipo. Uma versão apresentava os fontes originais do módulo e a outra versão, os fontes otimizados. Nos testes, cada versão foi testada durante três minutos

num mesmo dispositivo, onde a versão otimizada obteve uma taxa de quadros por segundo superior durante todo o teste. O ganho de desempenho foi de aproximadamente 160%.

Por fim, conclui-se que o protótipo desenvolvido foi capaz de identificar e estimar a pose da mão em diversas situações, permitindo assim que as metodologias empregadas neste trabalho possam ser aplicadas no desenvolvimento de trabalhos futuros ou até mesmo aperfeiçoadas para extensão deste. Estas metodologias se tornam ainda mais relevantes para o desenvolvimento de trabalhos que envolvem uso de realidade aumentada e/ou identificação de objetos através da cor. Além disso, o protótipo pode vir a ser utilizado como uma ferramenta de estudo da anatomia humana, visto que o mesmo faz uso de uma técnica bem dinâmica e intuitiva de visualização em três dimensões.

4.1 EXTENSÕES

Abaixo estão listadas algumas sugestões de extensões para o protótipo:

a) melhorar ou utilizar outras técnicas de segmentação para encontrar a mão;
b) estimar a pose da mão através de técnicas de inteligência artificial como Redes Neurais. Elas podem ser empregadas neste contexto para estimar a pose de mais gestos em conjunto com a mão;
c) implementar a visualização de todos os sistemas fisiológicos, permitindo a escolha em tempo real de quais sistemas fisiológicos devem aparecer na animação do aplicativo;
d) transformar em uma aplicação de realidade virtual para proporcionar maior imersão.

REFERÊNCIAS

ARLOON. **Arloon anatomy.** 2015. Disponível em: <http://www.arloon.com/apps/arloon-anatomy/>. Acesso em: 24 jun. 2018.

CURISCOPE. **Virtuali-Tee.** 2016. Disponível em: <https://www.curiscope.com/product/virtuali-tee/>. Acesso em: 25 jun. 2018.

DAQRI. **Anatomy 4D.** [2014?]. Disponível em: <http://anatomy4d.daqri.com/>. Acesso em: 25 jun. 2018.

DIGITALRUNE. **Definitions and Conventions**. 2016. Disponível em: <https://digitalrune.github.io/DigitalRune-Documentation/html/7945ac67-dffc-4197-9fbf-58a9b68c0ea1.htm>. Acesso em: 3 mar. 2018.

GRABNER, Alexander; ROTH, Peter M.; LEPETIT, Vincent. 3D Pose Estimation and 3D Model Retrieval for Objects in the Wild. **Conference On Computer Vision And Pattern Recognition 2018**. [S. l.], p. 1-13. mar. 2018.

FORTE, Cleberson Eugenio. **Software educacional potencializado com realidade aumentada para uso em física e matemática**. 2009. 215 f. Dissertação (Mestrado) - Curso de Mestrado em Ciência da Computação, Universidade Metodista de Piracicaba, Piracicaba, 2009.

LACERDA, Manoel B. **Realidade aumentada como motivação do aluno para a aprendizagem**. 2013. 116 f. TCC (Graduação) - Curso de Licenciatura em Informática, Universidade Estadual do Ceará, Mauriti, 2013.

MATHSISFUN. **Triangles**. 2017. Disponível em: <https://www.mathsisfun.com/triangle.html>. Acesso em: 2 jun. 2018.

MATHWORKS. **What Is Camera Calibration?**. 2018. Disponível em: <https://www.mathworks.com/help/vision/ug/camera-calibration.html>. Acesso em: 2 jun. 2018.

OPENCV. **Real Time pose estimation of a textured object**. 2018a. Disponível em: <https://docs.opencv.org/3.4/dc/d2c/tutorial_real_time_pose.html>. Acesso em: 2 jun. 2018.

______. **Camera Calibration and 3D Reconstruction**. 2018b. Disponível em: <https://docs.opencv.org/3.4/d9/d0c/group__calib3d.html>. Acesso em: 2 jun. 2018.

______. **Android**. 2018c. Disponível em: < https://opencv.org/platforms/android/>. Acesso em: 2 jun. 2018.

OPENGL-TUTORIAL. **Transformation matrices**. 2017. Disponível em: <http://www.opengl-tutorial.org/beginners-tutorials/tutorial-3-matrices/#transformation-matrices>. Acesso em: 3 mar. 2018.

OVERVOORDE, Alexander. **Modern OpenGL Guide**. 2017. Disponível em: < https://open.gl/introduction >. Acesso em: 3 mar. 2018.

RAMEY, Karehka. **The use of technology – in education and teaching process**. 2013. Disponível em: <http://www.useoftechnology.com/the-use-of-technology-in-education/>. Acesso em: 27 jun. 2018.

REARDON, Marguerite. **Augmented reality comes to mobile phones**. 2010. Disponível em: <https://www.cnet.com/news/augmented-reality-comes-to-mobile-phones/>. Acesso em: 2 jun. 2018.

REBELLO, Heverson et al. **Atlas virtual do esqueleto cefálico humano**: uma ferramenta complementar no processo ensinoaprendizagem da anatomia humana. In: CONGRESSO NACIONAL DE AMBIENTES HIPERMÍDIA PARA APRENDIZAGEM, 5., 2011, Pelotas. **Anais...** . Pelotas: Cce/ufsc, 2011. p. 1 - 10.

TODAMATERIA. **Sistemas do Corpo Humano**. 2018a. Disponível em: <https://www.todamateria.com.br/sistemas-do-corpo-humano/>. Acesso em: 2 jul. 2018.

______. **Sistema Cardiovascular**. 2018b. Disponível em: <https://www.todamateria.com.br/sistema-cardiovascular/>. Acesso em: 2 jul. 2018.

______. **Sistema Respiratório**. 2018c. Disponível em: <https://www.todamateria.com.br/sistema-respiratorio/>. Acesso em: 2 jul. 2018.

______. **Sistema Digestivo, Sistema Digestório**. 2018d. Disponível em: <https://www.todamateria.com.br/sistema-digestivo-sistema-digestorio/>. Acesso em: 2 jul. 2018.

______. **Sistema Nervoso**. 2018e. Disponível em: <https://www.todamateria.com.br/sistema-nervoso/>. Acesso em: 2 jul. 2018.

______. **Sistema Endócrino**. 2018f. Disponível em: <https://www.todamateria.com.br/sistema-endocrino/>. Acesso em: 2 jul. 2018.

______. **Sistema Excretor**. 2018g. Disponível em: <https://www.todamateria.com.br/sistema-excretor/>. Acesso em: 2 jul. 2018.

______. **Sistema Urinário**. 2018h. Disponível em: <https://www.todamateria.com.br/sistema-urinario/>. Acesso em: 2 jul. 2018.

______. **Sistema Reprodutor Masculino**. 2018i. Disponível em: <https://www.todamateria.com.br/sistema-reprodutor-masculino/>. Acesso em: 2 jul. 2018.

______. **Sistema Reprodutor Feminino**. 2018j. Disponível em: <https://www.todamateria.com.br/sistema-reprodutor-feminino/>. Acesso em: 2 jul. 2018.

______. **Sistema Muscular**. 2018k. Disponível em: <https://www.todamateria.com.br/sistema-muscular/>. Acesso em: 2 jul. 2018.

______. **Sistema Linfático**. 2018l. Disponível em: <https://www.todamateria.com.br/sistema-linfatico/>. Acesso em: 2 jul. 2018.

______. **Sistema Tegumentar**. 2018m. Disponível em: <https://www.todamateria.com.br/sistema-tegumentar/>. Acesso em: 2 jul. 2018.

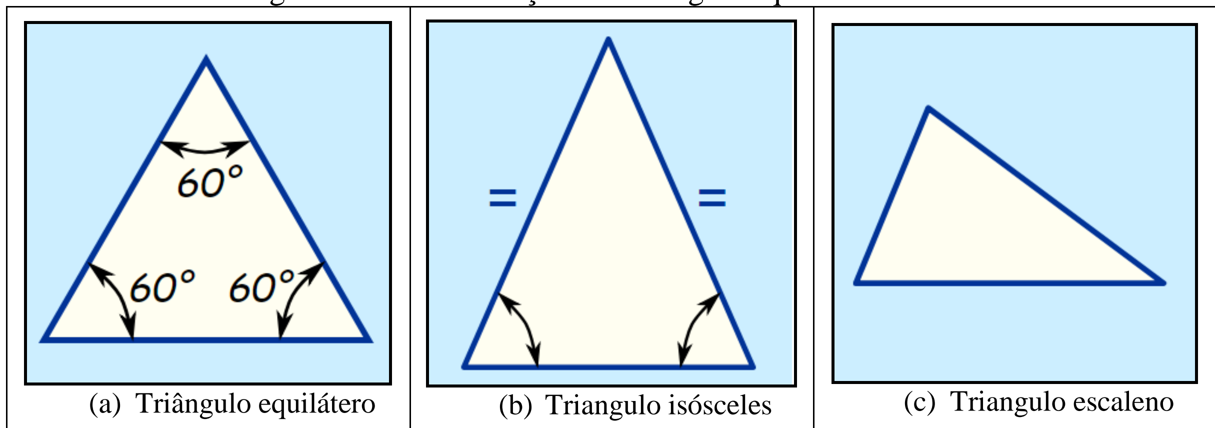
______. **Sistema Esquelético**. 2018n. Disponível em: <https://www.todamateria.com.br/sistema-esqueletico/>. Acesso em: 2 jul. 2018.

______. **Ossos da Mão**. 2018o. Disponível em: <https://www.todamateria.com.br/ossos-da-mao/>. Acesso em: 2 jul. 2018.

APÊNDICE A – Geometria do triângulo

Triângulos são figuras geométricas constituídas de três vértices que quando conectados formam três ângulos. Os três ângulos de um triângulo quando somados sempre totalizam 180°. Os segmentos de reta formados pelos vértices caracterizam os lados de um triângulo que permitem os mesmos de serem classificados. Os triângulos que tiverem os três lados com a mesma medida serão classificados como triângulo equilátero e terão os três ângulos como 60°. No caso de o triângulo possuir os dois lados iguais, será classificado como triângulo isósceles e terá dois ângulos como o mesmo valor. Por fim, se todos os lados de um triângulo tiverem medidas diferentes o mesmo será classificado como triângulo escaleno (MATHSISFUN, 2017). Na Figura 43 é ilustrada a classificação dos triângulos quanto aos lados.

Figura 43 – Classificação dos triângulos quanto aos lados

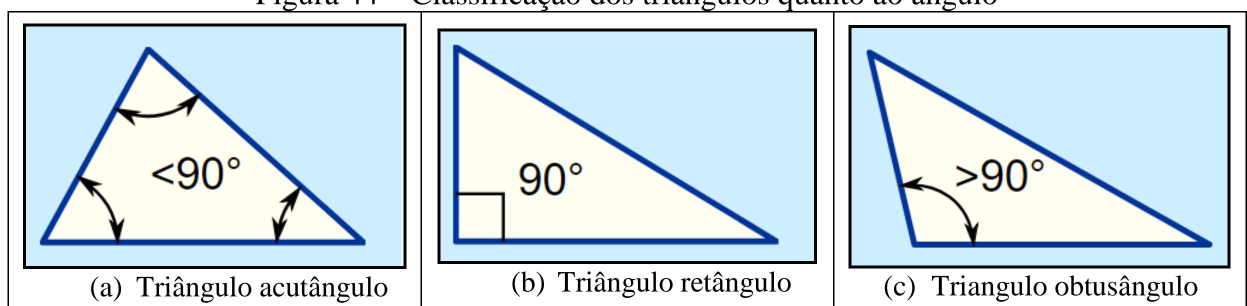


(a) Triângulo equilátero (b) Triangulo isósceles (c) Triangulo escaleno

Fonte: MathsIsFun(2017).

Segundo MathsIsFun (2017), os triângulos também podem ser classificados quanto ao tipo de ângulo. Quando um triângulo possuir todos os ângulos menores que 90° será classificado como triângulo acutângulo. Caso o triângulo possuir um ângulo igual a 90° receberá a classificação de triângulo retângulo. Por último, um triângulo é classificado como triângulo obtusângulo quando tiver um ângulo maior do que 90°, como é ilustrado na Figura 44.

Figura 44 – Classificação dos triângulos quanto ao ângulo

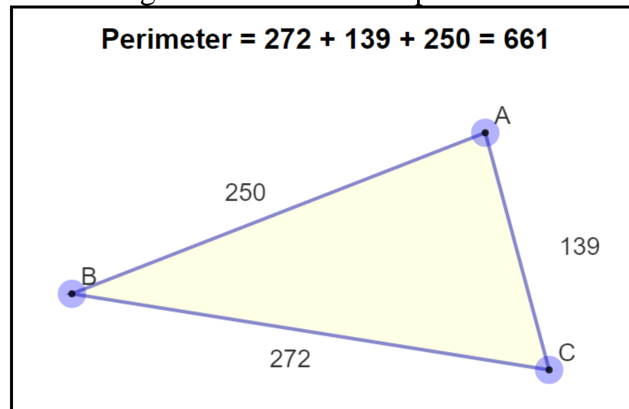


(a) Triângulo acutângulo (b) Triângulo retângulo (c) Triangulo obtusângulo

Fonte: MathsIsFun(2017).

Para calcular o perímetro de um triângulo, é necessário somar a distância dos três lados (MATHSIFFUN, 2017). Na Figura 45 é demonstrado o cálculo do perímetro de um triângulo.

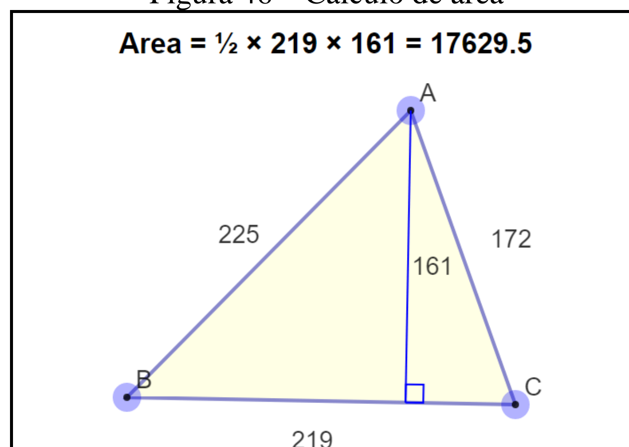
Figura 45 – Cálculo de perímetro



Fonte: MathsIsFun(2017).

Segundo MathsIsFun (2017), para calcular a área de um triangulo, é necessário multiplicar a distância da base do triângulo pela altura do triângulo e o resultado obtido da multiplicação dividir por 2. No exemplo da Figura 46, a base do triângulo tem o valor 219 e a altura 161, portanto o resultado é 17629.5.

Figura 46 – Cálculo de área



Fonte: MathsIsFun(2017).