INSTITUTO MILITAR DE ENGENHARIA

THIAGO LIMA DE OLIVEIRA

TAMARA SANT'ANA DE CARVALHO

GABRIEL DA CRUZ FONTENELLE

Projeto de Fim de Curso apresentado ao Curso de Graduação em Engenharia Eletrônica do Instituto Militar de Engenharia.

Orientadora: Carla Liberal Pagliari – Ph.D.

Rio de Janeiro

2014

INSTITUTO MILITAR DE ENGENHARIA

THIAGO LIMA DE OLIVEIRA

TAMARA SANT'ANA DE CARVALHO

GABRIEL DA CRUZ FONTENELLE

Projeto de Fim de Curso apresentado ao Curso de Graduação em Engenharia Eletrônica do Instituto Militar de Engenharia.

Orientadora: Carla Liberal Pagliari – Ph.D.

Aprovada em 3 de Junho de 2014 pela seguinte Banca Examinadora:

Profª. Carla Liberal Pagliari – Ph.D. do IME

Mauro Cezar Rebello Cordeiro

Ricardo Choren Noya

Rio de Janeiro

2014

Dedicamos a todos aqueles que de alguma forma contribuíram para a realização do mesmo.

Agradecemos a todas as pessoas que nos incentivaram, apoiaram e possibilitaram esta oportunidade de ampliar nossos horizontes.

Nossos familiares e mestres.

"A melhor maneira de nos prepararmos para o futuro é concentrar toda a imaginação e entusiasmo na execução perfeita do trabalho de hoje."

- Dale Carnegie

SUMÁRIO

LISTA DE ILUSTRAÇÕES

LISTA DE SIGLAS E SÍMBOLOS

SDK	Software Development Kit
NDK	Native Development Kit
JNI	Java Native Interface
RGB	Red, Green, Blue
HSV	Hue, Saturation, Value
YCbCr	Y: Luminância; CbCr: Crominância
LUX	Logarithmic Hue Extension

RESUMO

O Rastreamento (*Tracking*) de Movimentos e Reconhecimento de Gestos são tópicos muito pesquisados na área de visão computacional e são um verdadeiro desafio para os pesquisadores da área.

Ainda que na última década tenha se alcançado resultados surpreendentes, como é o caso do Kinect, o desafio agora é portar tais resultados para a palma das mãos dos usuários, permitindo que seus dispositivos móveis agreguem funcionalidades de rastreamento de movimento e reconhecimento de gestos, que os auxiliem em suas tarefas cotidianas.

Tanto o Rastreamento de Movimentos quanto o Reconhecimento de Gestos podem ser utilizados para facilitar a interação dos usuários com os dispositivos ou mesmo prover novas formas de interação com eles.

Os Gestos podem ser estáticos ou não. No caso de gestos estáticos uma determinada pose correspondente a uma determinada ação (comando) no dispositivo.

Para se reconhecer tais gestos é necessário que se implemente alguma forma de segmentação espacial, processo que é naturalmente executado pelo sistema visual humano e pelo cérebro.

Já em se tratando de gestos dinâmicos, é necessário marcar o início e o final do gesto. É importante ressaltar que o reconhecimento automático de gestos contínuos emprega técnicas de segmentação espacial e temporal.

Como não há contato físico com a tela do dispositivo, estes processos diferem dos métodos baseados no toque (*touchscreen-based methods*). Se o usuário está dirigindo, ou está com as mãos molhadas ou sujas, ainda pode controlar o dispositivo apenas com gestos sem ter que tocar no dispositivo.

Mas também há muitos problemas a serem considerados. Um deles é o aumento do consumo de bateria e um outro o tempo de processamento.

Além destes há vários desafios para que o reconhecimento de gestos e o rastreamento de movimentos sejam processos robustos em dispositivos móveis. As condições variáveis de iluminação e variação de fundo (*background*) são problemas que demandam a integração de diversas técnicas de processamento de imagens.

O objetivo deste trabalho é estudar tanto o rastreamento de movimento quanto o reconhecimento de gestos nas plataformas móveis em especial para as plataformas Android, através de um *framework*.

Embora o enfoque deste trabalho esteja nas plataformas móveis nada impede a migração do *framework* para outras plataformas.

ABSTRACT

Movement Tracking and Gesture Recognition are very researched topics in the area of computer vision and that present a tough challenge for researchers.

Although the in last decade it has achieved amazing results, as is the case of the Kinect, the challenge now is to bring such results to the palm of users' hands, allowing their mobile devices add features such as motion tracking and gesture recognition, which help them in their everyday tasks.

Both the Motion Tracking and Recognition of Gestures can be used to facilitate interaction between users and devices or even provide new ways to interact with them.

Gestures can be static or not . In the case of static gestures a particular pose correspond to a particular action (command) in the device.

To recognize such gestures is necessary to implement some form of spatial segmentation process that is performed naturally by the human visual system and the brain.

Already in the case of dynamic gestures, you must mark the beginning and end of the gesture. Importantly, the automatic recognition of continuous gestures employs techniques of temporal and spatial segmentation.

Since there is no physical contact with the screen of the device, these processes differ from methods based on touch (touchscreen -based methods). If you 're driving, or have wet or dirty hands, you can still control the device only by gestures without touching the device.

But there are many issues to consider. One is the increased battery consumption and another is the processing time.

Besides these there are several challenges for gesture recognition and motion tracking to become robust processes on mobile devices . The variable lighting conditions and variation in background are problems that require the integration of several techniques of image processing.

The objective of this work is to study both the motion tracking and gesture recognition on mobile platforms, in particular for Android platforms, through a framework.

Although the focus of this work is on mobile platforms nothing prevents the migration of the framework to other platforms.

1 INTRODUÇÃO

Gerar imagens sintéticas, nos computadores, a partir de dados foi um dos primeiros desafios vencidos pela computação gráfica e graças a isso fomos presenteados com diversas facilidades como jogos tridimensionais, simuladores, efeitos especiais, etc. Embora esses exemplos sejam surpreendentes e continuem nos impressionando quando vemos filmes como Avatar e similares, a computação gráfica ainda é uma visão de mão única no sentido do homem para máquina. O que os pesquisadores sempre se questionaram é como fazer com que a máquina enxergasse o homem e o mundo ao seu redor com o mesmo nível de cognição da visão humana ou pelo menos similar. Assim surgiu a visão computacional, que de maneira simples pode ser definida como a área que trata do caminho inverso ao da computação gráfica, isto é, busca recuperar dados a partir de uma imagem ou conjuntos delas e a partir desses dados recuperados auxiliar as máquinas na tomada de decisões.

O que diferencia a visão humana da visão da máquina é que nós seres humanos temos a cognição intimamente associada a nossa visão, isto é, enxergar é um complexo sistema que vai muito além de gerar uma imagem em nossa retina, enxergar compreende identificar e diferenciar objetos, inferir profundidade, reconhecer cores, texturas e qualquer outro tipo de conhecimento gerado a partir das imagens capturadas por nossa retina. Não é por menos que os estímulos visuais são responsáveis por cerca de 70% da atividade cerebral.

Uma parte fundamental da visão humana consiste em identificar objetos que se movem e reconhecer sinais, essas duas funções da visão humana embora não contemplem toda complexidade da visão humana são os aspectos fundamentais a serem portados para os computadores a fim de lhes conferir algum nível de visão sobre o mundo externo. Por essa razão estes dois pontos serão nossos objetos de estudo ao longo deste trabalho. Antes, contudo é preciso entender como são capturadas as imagens que serão fontes de obtenção de dados.

Basicamente um computador precisa de câmeras que capturem imagens do ambiente e então submetêndo-as a análise extrai informações dessas imagens. Ao processo de análise de imagens dá-se o nome Processamento de Imagem.

De maneira geral podemos dividir o processamento de imagem em três etapas: Aquisição de Imagens, Análise de Imagens e Apresentação dos Resultados.

Baseado nessas três etapas, foi construído o framework apresentado no capítulo 3, cujo objetivo principal é o desenvolvimento de aplicações que façam uso de visão computacional. O framework será base para construção do sistema de testes que foi desenvolvido para auxiliar e testar as técnicas de processamento de imagem abordadas nessa pesquisa.

Diversas pesquisas em visão computacional com o enfoque em rastreamento de movimento e reconhecimento de gestos podem ser encontradas e muitas delas foram utilizadas

ao longo deste trabalho, contudo, em se tratando de plataformas móveis, temos que levar em conta alguns aspectos limitantes dessas plataformas como, por exemplo, capacidade de processamento e bateria. Diferente de outras plataformas, dispositivos móveis tem limitações de processamento e bateria, sendo assim não é possível utilizar todas as soluções que já existem justamente por serem onerosas computacionalmente e por isso consomem muita energia. Utilizá-los faria com que o nível de utilização dos aparelhos móveis fosse muito reduzido ou forçar-nos-ia a utilizá-los ligados a tomada.

1.1 MOTIVAÇÃO

Interagir com computadores sempre foi uma atividade muito interessante e permitiu ao homem realizar tarefas antes impossíveis em virtude de limitações de tempo e capacidade de cálculo dos seres humanos. Graças aos computadores o homem pode ir a lua, por exemplo, sendo resguardado a todo momento por computadores que realizavam cálculos que garantiam a segurança da missão.

A sétima arte, o cinema, e os jogos são outros exemplos de áreas que fizeram e fazem cada vez mais o uso dos computadores em conjunto com técnicas de computação gráfica. Contudo todos esses exemplos tem em comum do mundo real ser invisível no sentido das máquinas não conseguirem inferir cognição alguma em relação ao mundo como reconhecer pessoas, cores, sons e qualquer outra forma de conhecimento tal como um ser humano. Fazer os computadores enxergarem o mundo tal como os seres humanos ou pelo menos de maneira similar constitui um grande desafio ainda mais se tratando de plataformas móveis como smartphones e tablets que nos acompanham quase que durante o dia todo em todas as atividades que desempenhamos desde o trabalho, estudo até o lazer.

Outro fator motivacional é que aplicações em visão computacional sempre esbarraram em limitações de desempenho uma vez que as técnicas utilizadas têm alto custo computacional e em especial aquelas que requerem resposta em tempo real, novamente em se tratando de plataformas móveis precisamos de cuidados especiais para que as aplicações não degradem o desempenho e inviabilizem sua utilização. Para isso, muitos conceitos consolidados no desenvolvimento de sistemas têm de ser revistos nesses sistemas para que se alcance o desempenho desejável, mas sem, contudo comprometer a manutenibilidade desses sistemas.

1.2 OBJETIVOS

Os principais objetivos desta pesquisa são portar para dispositivos móveis o rastreamento de movimento (Figura 1.1) e a detecção de gestos simples (Figura 1.2) utilizando as câmeras destes dispositivos. Além disso, um framework básico para desenvolvimento de aplicações de visão computacional foi desenvolvido e serviu de base para construção de um sistema de testes para cada etapa da pesquisa.

Figura 1.1: Rastreamento de movimento

Figura 1.2: Exemplos de gestos simples

1.2.1 RASTREAMENTO DE MOVIMENTO

Rastrear um movimento consiste em identificar a movimentação de um ou mais objetos em uma sequência de frames.

Para que possamos rastrear movimento precisamos:

a) Definir critérios que sinalizem a ocorrência de movimento e;

b) Identificar características que possam ser identificadas nas diversas imagens de que dispormos.

Para identificar a ocorrência de movimento podemos utilizar uma técnica simples chamada de diferença de frames (Figura 1.3), que consiste basicamente em subtrair dois frames consecutivos. Essa técnica pressupõe que entre dois frames consecutivos o background da cena permanece imóvel enquanto que o objeto ou os objetos em movimento mudam de posição, para que a técnica funcione é necessário que a câmera permaneça imóvel caso contrário todo o background estará em movimento.

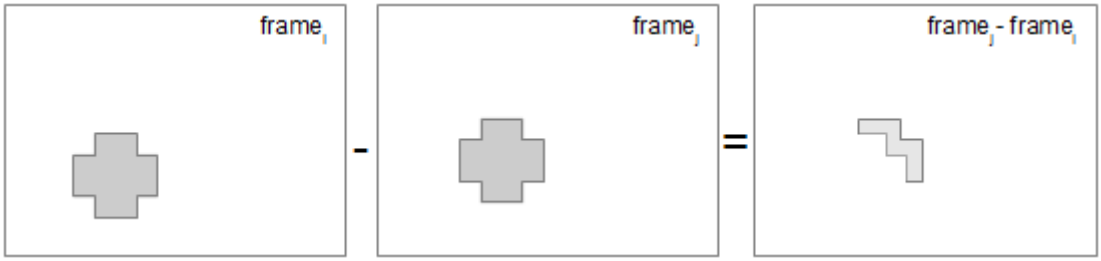


Figura 1.3: Diferença de Frames

1.2.2 DETECÇÃO DE GESTOS

Detectar gestos consiste, basicamente, em analisar uma cena em busca de algum elemento que se encaixe em padrões pré-estabelecidos, como por exemplo, uma mão aberta numa cena pode ser associada a um pedido de interrupção ou uma mão se movendo numa sequência de frames pode ser associada a prosseguimento.

O reconhecimento por máquina algo bem complexo se tomamos como base os seres humanos em que a capacidade cognitiva é tão apurada que uma mão aberta pode ter diferentes significados associados com o ambiente, a expressão facial e muitas outras variáveis. Assim quando falamos em reconhecimento de gestos por máquina devemos ter em mente que inicialmente, até pelas limitações tecnológicas, o reconhecimento de gestos se restringe a reconhecer gestos simples como mostrados na Figura 1.4.

Figura 1.4: Exemplos de gestos reconhecidos por máquina (NEWS BUREAU, 2012)

Assim nossas ambições nesta pesquisa no que diz respeito a reconhecimento de gestos restringe-se a detecção de gestos simples.

1.3 JUSTIFICATIVA

Ainda que a área de visão computacional voltada para rastreamento de movimento e detecção de gestos já esteja avançada e com ótimos resultados, a emergente proliferação de plataformas móveis abre um novo horizonte de aplicações que façam uso de técnicas de visão computacional, contudo é necessário adequar às técnicas já existentes às limitações de tais dispositivos que contam com menor capacidade de processamento e bateria.

Buscar alternativas viáveis que permitam incorporar a visão computacional a esses dispositivos tornando-os capazes enxergar ativamente o ambiente que nos cerca, sem, contudo diminuir o nível de utilização para os usuários é um grande desafio e ainda não existem resultados conclusivos.

1.4 METODOLOGIA

Inicialmente uma pesquisa sobre os principais conceitos a respeito de imagem digital foi realizada com a finalidade de permitir a compreensão das técnicas de processamento de imagens e entender os problemas decorrentes da discretização das imagens reais em imagens digitais.

Em seguida uma vasta pesquisa bibliográfica foi realizada com o objetivo de encontrar os pontos chave para a detecção de gestos e o rastreamento de movimento, preocupando-se em buscar soluções que não fossem muito onerosas computacionalmente.

De posse dos pontos chave para a detecção de gestos e rastreamento de movimento passou-se a etapa de testes das soluções encontradas para isso um framework foi desenvolvido, que por sua vez serviu de base par construção de um sistema de testes.

Entendidos e testados passamos a etapa de concatenação das diversas técnicas pesquisadas a fim de chegarmos aos nossos principais objetivos que são a detecção de gestos e o rastreamento de movimento. Para isso diversos programas testes foram desenvolvidos a fim de mostrar o funcionamento das técnicas pesquisadas trabalhando em conjunto.

Por fim uma aplicação simples de detecção de gestos e rastreamento de movimento foi desenvolvida e que une todos os conceitos abordados nesta pesquisa.

1.5 ESTRUTURA DA MONOGRAFIA

O capítulo 2 explica com detalhes as etapas envolvidas no processamento de imagem e apresenta técnicas de processamento de imagem referentes as detecção de bordas, detecção de pele, minimização de ruídos e regulagem de contraste em imagens.

O capítulo 3 trata da especificação do framework, para plataforma Android, desenvolvido para auxiliar em aplicações que façam uso de técnicas de visão computacional e que foi utilizado como base para construção do sistema de testes.

O capítulo 4 propõe uma heurística para detecção da mão humana para posteriormente realizar o reconhecimento de gestos baseados no número de dedos da mão.

O capítulo 5 apresenta os resultados das diversas técnicas apresentadas nesta pesquisa em dispositivos Android.

2 PROCESSAMENTO DE IMAGEM

Ainda não se encontrou uma única definição para processamento de imagem, mas aqui consideraremos que são processos cujas entradas e as saídas são imagens e, em adição, engloba processos que extraem atributos da imagem, e incluem o reconhecimento de objetos da imagem.

De maneira geral podemos dividir o processamento de imagem em três etapas (Figura 2.1): Aquisição de Imagens, Análise de Imagens e Apresentação dos Resultados.

Figura 2.1: Diagrama em blocos do Processamento de Imagem

A Aquisição ainda não é processamento de imagem propriamente dito, contudo é fundamental para obter imagens por meio de algum sensor geralmente câmeras VGA.

A Análise é onde de fato é feito o processamento da imagem. Nela é são aplicados filtros para extração de características da imagem e a partir disso gerar os dados que serão classificados e reconhecidos, similar a cognição humana.

Existem três níveis de análise sobre uma imagem: os de baixo nível, os de nível médio e os de alto nível.

Os processos de baixo nível envolvem operações de primitivas como imagem de pré-processamento para diminuir ruído e realce de contraste, por exemplo, (neste nível tanto a entrada quanto a saída são imagens). Nos processos de nível médio estão envolvidas tarefas de segmentação (particionamento de uma imagem em regiões e objetos). As suas entradas

geralmente são imagens e as saídas são atributos extraídos dessas imagens (reconhecimento de objetos). Já no nível alto existe a ideia do "fazer sentido", ou seja, funções cognitivas que são mais comumente associadas com a visão humana.

Por fim a Apresentação consiste em gerar ações apartir da informação extraída como, por exemplo, executar um comando ou simplesmente exibir a imagem resultante do processamento.

2.1 IMAGEM DIGITAL

Uma imagem pode ser definida como uma função de duas dimensões, $f(x, y)$, onde x e y são coordenadas do plano espacial e a amplitude f de um par de coordenadas (x, y) é chamada de intensidade ou nível de cinza da imagem naquele ponto. Se x, y e f forem finitos em quantidades discretas, diz-se que a imagem é uma imagem digital.

Quando uma imagem é gerada de um processo físico, seus valores são proporcionais à energia irradiada pela fonte física (ondas eletromagnéticas, por exemplo). Consequentemente: $f(x, y)$ é finito.

A saída dos sensores de imagem é uma forma de onda de tensão contínua cuja amplitude e o comportamento espacial estão relacionados ao fenômeno físico que está sendo detectado.

Uma imagem digital é criada fazendo-se a amostragem nas direções x e y, e a quantização na amplitude.

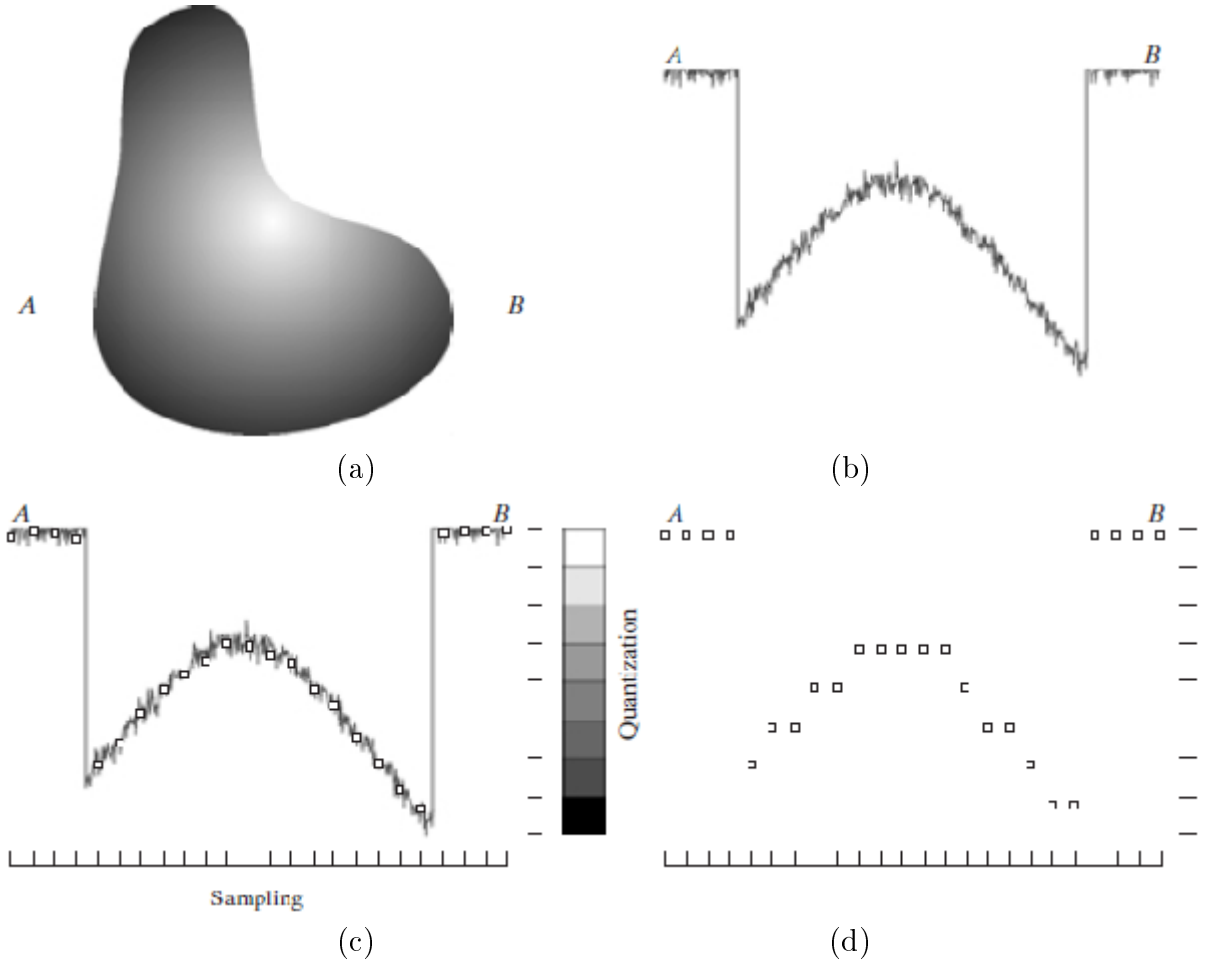
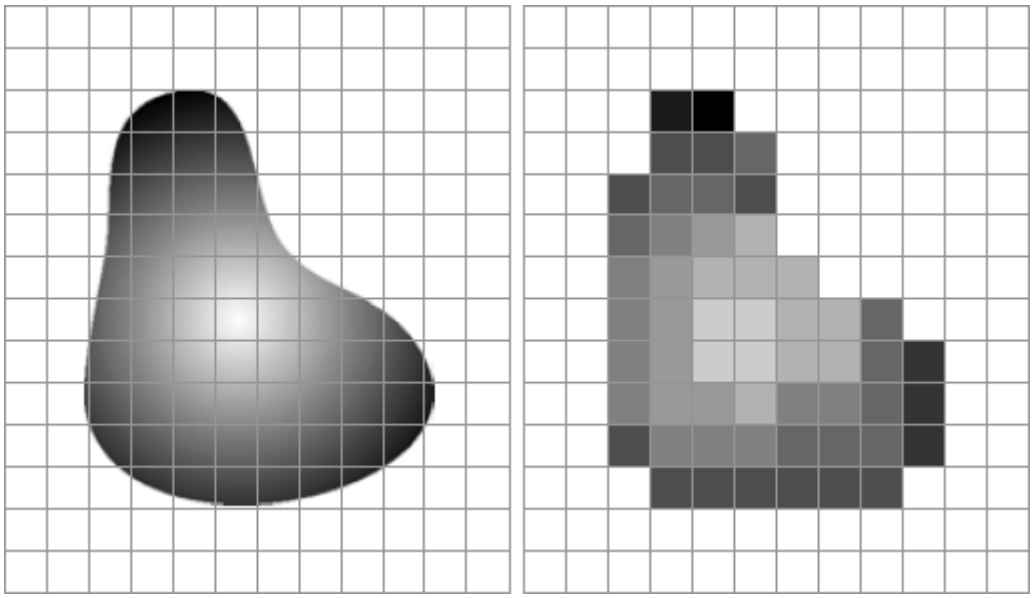


Figura 2.2: Amostragem (GONZALEZ, 1987)

Na Figura 2.2 (a) mostra a imagem contínua a ser transformada em imagem digital. A Figura 2.2 (b) representa a função unidimensional que plota os níveis de cinza da imagem ao longo da linha AB da Figura 2.2 (a). As variações aleatórias são devido ao ruído da imagem. A Figura 2.2 (c) mostra a Figura 2.2 (b) amostrada. Foram pegas amostras igualmente espaçadas ao longo da linha AB. A localização de cada amostra é dada pela parte inferior da figura. As amostras são os quadrados brancos ao longo da figura. O conjunto dessas localizações discretas fornece a função amostrada. Entretanto ainda falta fazaer a discretização.

Ainda na Figura 2.2 (c), na parte vertical, temos a escala de cinza variando em oito níveis discretos do preto ao branco. Os níveis de cinza são quantizados simplesmente por determinar um dos níveis de cinza para cada amostra com a marcação vertical.

A Figura 2.2 (d) mostra o resultado da amostragem e da quantização que é obtido fazendo o processo acima.



(a) (b)

Figura 2.3: Imagens contínua (a) e digital (b) (GONZALEZ, 1987)

A figura Figura 2.3 representa em (a) a imagem contínua e em (b) a imagem digital.

A qualidade da imagem digital é determinada em grande medida pelo número de amostras e os níveis de cinza discretos utilizados na amostragem e na quantização.

Uma imagem digital pode ser definida pela seguinte equação representada pela Figura 2.4.

$$f(x,y) = \begin{bmatrix} f(0,0) & f(0,1) & \cdots & f(0,N-1) \\ f(1,0) & f(1,1) & \cdots & f(1,N-1) \\ \vdots & \vdots & \ddots & \vdots \\ f(M-1,0) & f(M-1,1) & \cdots & f(M-1,N-1) \end{bmatrix}$$

Figura 2.4: Imagem Digital

O lado direito desta equação é, por definição, uma imagem digital e cada elemento dessa matriz é chamado de um elemento de imagem, pixel.

Tratar a imagem como uma matriz possibilita fazer aplicações como redução de ruído (convolução), adição (normalização de brilho), subtração (detecção de diferenças entre duas imagens, eventualmente adquiridas de forma consecutiva, da mesma cena), multiplicação (calibração do brilho) e outros diversos exemplos de operações que possuem o objetivo de melhorar a imagem de acordo com os critérios desejados.

A função $f(x,y)$ representa o produto da interação entre luminância $i(x,y)$, que exprime a quantidade de luz que incide sobre o objeto, e as propriedades de refletância ou de transmitância próprias do objeto, que podem ser representadas pela função $r(x,y)$, cujo

valor exprime a fração de luz incidente que o objeto vai transmitir ou refletir ao ponto (x, y).

Matematicamente:

$$f(x, y) = i(x, y) * r(x, y) \tag{2.1}$$

com $i(x, y)$ e $r(x, y)$ finitos (GONZALEZ, 1987).

2.2 DETECÇÃO DE PELE

Detectar pele numa imagem digital, basicamente consiste em avaliar se um dado pixel corresponde a pele ou não. Para realizar esta tarefa utilizamos os chamados classificadores (AHMED, 2009) que podem conter um critério ou um conjunto de critérios um ou um conjunto de critérios que irão julgar se o pixel é ou não um tom de pele.

Nesta seção apresentaremos alguns dos classificadores utilizados em detecção de pele e que podem ser encontrados em AHMED, 2009. Também são apresentados diferentes espaços de cores, bem como diversas técnicas de processamento de imagens necessárias para o processo de detecção de pele.

2.2.1 CLASSIFICADORES

Como dissemos, os classificadores tem a incumbência de avaliar se um pixel corresponde ou não a um tom de pele, simplificadamente podemos dizer que dado um pixel$P\,(c_0, c_1, ..., c_n)$ um classificador A é uma função que leva do espaço de cores (S) ao qual o pixel P pertence ao espaço binário $0, 1$ onde 0 (zero) indica que o pixel não representa um tom de pele e 1 (um) indica que o pixel representa um tom de pele.

$$A : S \rightarrow \{0, 1\} \tag{2.2}$$

$$A(P) = \begin{cases} 1 & , se\ P\ for\ um\ tom\ de\ pele \\ 0 & , caso\ contrário \end{cases} \tag{2.3}$$

Existem duas abordagens básicas para os classificadores de tons de pele:

a) Avaliação estatística e;

b) Avaliação espacial.

A abordagem a) avalia estatísticamente a probabilidade de um dado pixel ser ou não correspondente a um tom de pele. Estes classificadores geralmente tem uma pré-etapa chamada fase de treinamento em que são geradas estatísticas a respeito de tons de pele a partir de diversas imagens de testes, maiores detalhes podem ser conferidos em AHMED, 2009.

Já a abordagem b) avalia um pixel ou uma vizinhança de pixel afim de avaliar se o pixel é ou não um tom de pele. É a abordagem mais simples e não requer nenhuma pré-etapa de treino, mas ainda sim apresenta resultados satisfatórios e é nessa abordagem que focaremos nosso trabalho.

2.3 ESPAÇOS DE CORES

Para se representar uma imagem ou vídeo é preciso utilizar um padrão de codificação da informação. Os padrões de codificação são baseados em espaços de representação de cores, ou simplesmente espaços de cores.

Neste capítulo serão apresentados 3 dos principais espaços de cores comumente utilizados em codificação de imagem e vídeo: o RGB, o YCbCr e o HSV. Sendo os dois primeiros orientados a hardware e o último orientado ao usuário.

É importante conhecer cada um deles para que seja possível definir o espaço que apresenta as características que facilitarão a detecção e pele.

2.3.1 RGB

O espaço RGB (Red, Green, Blue) é o mais simples e intuitívo espaço de cores que existe Nele cada cor é representada com base na sua intensidade de vermelho, verde e azul. Este modelo é utilizado em monitores de computador, por exemplo.

As cores primárias RGB são aditivas, isto é, as contribuições individuais de cada primária são adicionadas para produzir uma cor. Comumente utiliza-se o cubo de cores RGB mostrado na Figura 2.5 para representar este espaço de cores. A diagonal principal do cubo, com quantidades iguais de cada primária, representa os níveis de cinza, onde (0,0,0) corresponde ao preto e (1,1,1) corresponde ao branco (GONZALEZ, 1987).

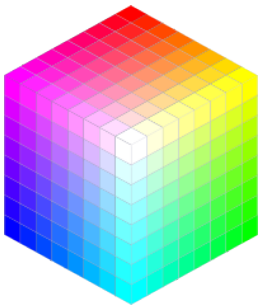


Figura 2.5: Espaço RGB (WIKIPEDIA)

Computacionalmente as cores neste espaço são representadas através de uma combinação linear entre vermelho (R), verde (G) e azul (B) (Eq. 2.4)

$$C_i = (aR, bG, cB) \tag{2.4}$$

onde, $a, b, c \in [0, 1]$ e $R = G = B = 255$.

Apesar da simplicidade desse espaço de cores, duas características limitam sua utilização para a detecção de pele:

a) Não faz distinção entre luminância e crominância;

b) É perceptualmente uniforme, o que significa que distâncias no RGB não tem uma correspondência linear com a percepção humana.

É fácil justificar a característica a) uma vez que a luminância de um pixel $P_{i,rgb} = (r_i, g_i, b_i)$ em RGB é dada pela combinação linear de R, G e B conforme a Eq. 2.5, enquanto a informação de crominância está intrínseca nas componentes R, G e B.

$$Y_i = 0.3086 * r_i + 0.6094 * g_i + 0.082 * b_i \tag{2.5}$$

O coeficientes de r_i, g_i e b_i refletem o nível de sensibilidade da visão humana em relação ao vermelho, verde e o azul.

A característica b) pode ser precebida a partir da Eq. 2.5 onde percebemos que a mudança na luminância em RGB, afeta as componentes r_i, g_i e b_i.

2.3.2 YCBCR

O espaço YCbCr é uma codificação comumente utilizada em vídeo, imagens e televisão (Figura 2.6), WIKIPEDIA.

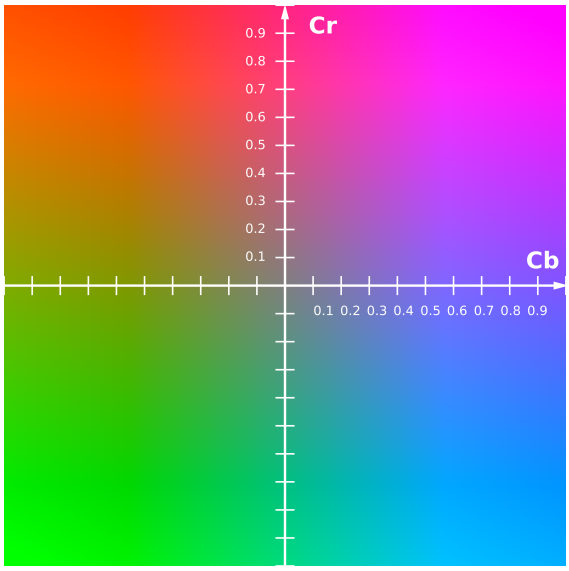


Figura 2.6: Espaço YCbCr (WIKIPEDIA)

O diferencial desse espaço em relação ao RGB é que ele separa a informação de luminância (Y) (Eq. 2.6) da informação de cor (CbCr), sendo Cb a diferença entre a luminância e a quantidade de azul (Eq. 2.7) e Cr a diferença entre a luminância e a quantidade de vermelho (Eq. 2.8).

$$Y = 0.3086 * R + 0.6094 * G + 0.082 * B \tag{2.6}$$

$$Cb = B - Y \tag{2.7}$$

$$Cr = R - Y \tag{2.8}$$

Outra característica importante deste espaço de cores é que a informação de cor é subamostrada (Figura 2.7) permitindo uma redução de informação sem grandes prejuízos perceptíveis de qualidade. Para a codificação YCbCr 4:2:0 para cada quatro pixels distribuídos espacialmente com mostrado na Figura 2.7 apenas uma informação de cor (CbCr) é necessária, calculada como uma média dos valores de cor de cada um dos pixels.

Y_i	Y_{i+1}	C_bC_r - Informação de cor CbCr
C_bC_r		$Cb = \frac{Cb_i+Cb_{i+1}+Cb_j+Cb_{j+1}}{4}$ $Cr = \frac{Cr_i+Cr_{i+1}+Cr_j+Cr_{j+1}}{4}$
Y_j	Y_{j+1}	Y_k-Luminâmcia do pixel k

Figura 2.7: Codificação YCbCr 4:2:0

Diferente do espaço RGB, no YCbCr, a localização de uma cor (CbCr) é fixa e independente da luminosidade (Y).

2.3.3 HSV

Diferente dos espaços RGB e YCbCr qua são orientados a hardware, o espaço HSV (Hue, Saturation, Value) é orientado ao usuário e por isso é o que perceptualmente mais se assemelha a visão humana. Para representar uma cor são utilizadas as componentes de matiz, saturação e brilho, que são representadas segundo um sistema de coordenadas cilíndrico conforme a Figura 2.8, WIKIPEDIA.

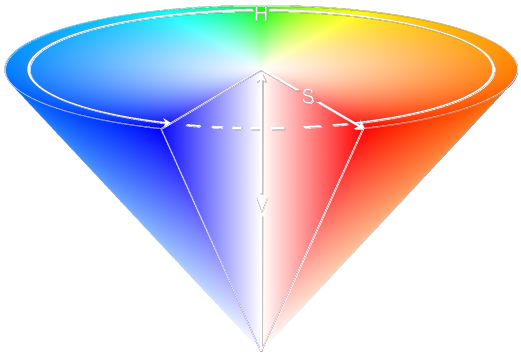


Figura 2.8: Espaço HSV (WIKIPEDIA)

O eixo vertical V corresponde ao brilho das cores, onde topo do cone corresponde a V = 1. O matiz H é mensurado através de do ângulo ao redor do eixo vertical V, com vermelho em 0^{o} e verde em 120^{o}. Cores complementares no cone HSV são opostas 180^{o} uma da outra. O valor de S é um valor entre 0, na linha do centro, e 1 nas extremidades do cone.

Para converter para o espaço HSV a partir do RGB é feito segundo as Eq.2.9 a Eq.2.17.

$$R' = R/255 \tag{2.9}$$

$$G' = G/255 \tag{2.10}$$

$$B' = B/255 \tag{2.11}$$

$$C_{max} = max\{R', G', B'\} \tag{2.12}$$

$$C_{min} = min\{R', G', B'\} \tag{2.13}$$

$$\Delta = C_{max} - C_{min} \tag{2.14}$$

$$H = \begin{cases} 60^{o} \times \left(\frac{G'-B'}{\Delta} mod 6\right) & , C_{max} = R' \\ 60^{o} \times \left(\frac{B'-R'}{\Delta} + 2\right) & , C_{max} = G' \\ 60^{o} \times \left(\frac{R'-G'}{\Delta} + 4\right) & , C_{max} = B' \end{cases} \tag{2.15}$$

$$S = \begin{cases} 0 & , \Delta = 0 \\ \frac{\Delta}{C_{Max}} & , \Delta \neq 0 \end{cases} \tag{2.16}$$

$$V = C_{max} \tag{2.17}$$

2.3.4 ESPAÇO DE CORES IDEAL PARA DETECÇÃO DE PELE

Nas subseções 2.3.1, 2.4.2 e 2.3.3 fizemos uma breve apresentação dos principais espaços de cores existentes, mas qual deles é o ideal para detecção de pele?

Precisamos de um espaço de cores que seja menos suscetível à influência do nível de iluminação uma vez que é muito difícil manter um nível de iluminação satisfatório sem que seja num ambiente com iluminação controlada. Segundo esse critério o espaço de cores RGB não é uma opção uma vez que as características a) e b) da subseção 2.3.1 não atendem a esse critério, assim nossa escolha se restringe aos espaços YCbCr e ao HSV.

Segundo AHMED, 2009, os espaços YCbCr e HSV são os que apresentam os melhores resultados na detecção de pele sendo o espaço HSV é o que apresenta melhores resultados justamente por ser o que perceptualmente mais se assemelha a visão humana, contudo é muito custoso computacionalmente a conversão para este espaço de cores como podemos ver pelas Eq.2.9 a Eq.2.17. Assim o espaço YCbCr é que melhor atende a nossas exigências, além de ser o formato nativo de dispositivos de captura de imagem como câmeras de vídeo.

A Figura 2.9 mostra nível de detecção de pele segundo os principais espaços de cores.

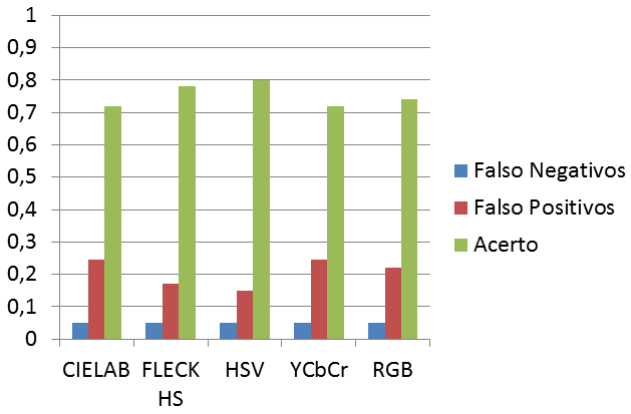


Figura 2.9: Comparativo entre os espaços de cores em detecção de pele (ALBIOL, 2005)

Como podemos perceber os espaços HSV e YCbCr apresentam os maiores níveis de reconhecimento, sendo o HSV ligeiramente superior ao YCbCr.

A Figura 2.10 retirada de AHMED, 2009, representa o mapeamento de dos tons de pele asiática, africana e caucasiana nos espaços de cores RGB e YCbCr. Como podemos perceber no espaço RGB os tons correspondentes a pele são mapeadas numa ampla área com forma que dificulta a avaliação. Já nos espaço YCbCr a região correspondente a pele é bem definida o que permite uma fácil avaliação.

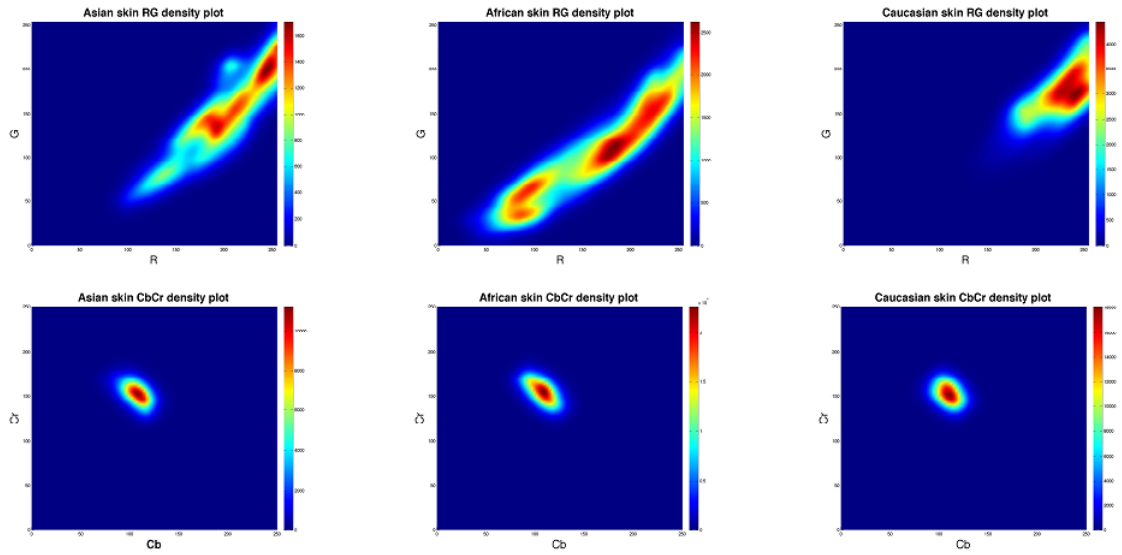


Figura 2.10: Gráficos de densidade da pele asiática, africana e branca em diferentes espaços de cor (AHMED, 2009)

2.3.5 DETECÇÃO DE PELE NO ESPAÇO YCBCR

Embora o espaço de cores YCbCr, como mostrado na subseção 2.3.4, não seja o que produza os melhores resultados há fatores que tornam sua escolha como espaço de cores para detecção de pele favoráveis:

- É um espaço cuja crominância é invariante sobre luminosidade;
- É o espaço nativo de imagem das câmeras digitais;
- A conversão para outros espaços de cores como RGB e HSV é muito onerosa computacionalmente, o que inviabiliza o uso em dispositivos móveis;
- Existem trabalhos relacionados com detecção de pele no espaço YCbCr que obtiveream bons resultados.

Entre os trabalhos que tratam de detecção de pele no espaço YCbCr, citamos MAHMOUD, 2008, que propõe uma técnica de detecção de pele com baixo custo computacional e que obtém resultados melhores que métodos tradicionais de detecção de pele.

A técnica consiste em avaliar pixel a pixel da imagem tal como um classificador espacial (subseção 2.2.1), submetendo a informação de crominância ao seguinte critério:

Seja $P_i$um pixel com valores de crominância Cb e Cr, o pixel P_ié pele se:

$$77 < Cb < 127 \; e \; 133 < Cr < 173 \tag{2.18}$$

Segundo MAHMOUD, 2008, a região delimitada por estes valores de Cb e Cr é a que melhor representa o conjunto de tons de pele humana.

2.3.6 DETECÇÃO DE PELE NO ESPAÇO LUX

Como podemos conferir na subseção 5.1.1 a detecção de pele utilizando o espaço de cores YCbCr apresenta sérios problemas e mostrou-se muito suscetível a variação luminosa, ainda que seja um espaço de cor invariante à iluminação. Parte desse problema é decorrente de ajustes que a câmera dos dispositivos faz como, por exemplo, o ajuste de branco que provoca uma translação da região correspondente a pele humana (Figura 2.10), por esse motivo, buscou-se uma alternativa a esse espaço de cores. A princípio pensou-se em utilizar o HSV que segundo ALBIOL, 2005, apresenta os melhores resultados para detecção de pele. Contudo, uma alternativa mais simples foi encontrada em FUH, 2013, onde foi utilizado o espaço de cores LUX (Logarithmic hUe eXtension).

O espaço LUX é muito sensível ao vermelho. Nos testes realizados em FUH, 2013, este espaço de cores conseguiu segmentar eficientemente pele humana.

A conversão para o espaço LUX é feita segunda as equações:

$$L = (R+1)^{0.3}.(G+1)^{0.6}.(B+1)^{0.1} - 1 \tag{2.19}$$

$$U = \begin{cases} \frac{M}{2}\left(\frac{R+1}{L+1}\right) & , \, se \, R < L \\ M - \frac{M}{2}\left(\frac{L+1}{R+1}\right) & , \, c.c. \end{cases} \tag{2.20}$$

Seja $P_i$um pixel com valores de crominância Cb e Cr, o pixel P_ié pele se:

$$X = \begin{cases} \frac{M}{2}\left(\frac{B+1}{L+1}\right) & , \, se \, R < L \\ M - \frac{M}{2}\left(\frac{L+1}{B+1}\right) & , \, c.c. \end{cases} \tag{2.21}$$

KUMAR, 2008, propõe uma simplificação da equação 2.20, para detecção de pele:

$$U' = \begin{cases} 256.\frac{G}{R} & , se\, \frac{R}{G} < 1.5\, e\, R > G > 0 \\ 255 & , c.c. \end{cases} \tag{2.22}$$

A detecção de pele proposta em FUH, 2013, é feita segundo o critério:
Seja P_i um pixel da imagem, P_i é pele se:

$$77 \leq Cr \leq 127\, e\, 0 \leq U \leq 249 \tag{2.23}$$

Observe que o espaço de cores YCbCr não é completamente negligenciado apenas robustecemos a detecção de pele substituindo o teste feito com a componete Cb por um teste que avalia melhor o vermelho da imagem.

2.4 MINIMIZAÇÃO DO RUÍDO

Um dos principais problemas encontrados quando se quer extrair características de imagens é o ruído, originado por erros nos algoritmos de filtragem ou devido às características do sensor, etc. Por mais criteriosos que sejam os filtros de processamento de imagem eles não conseguem prever interferências do ambiente externo, como variações de luminosidade, por exemplo, que acabam por conduzir a falsos positivos para os critérios utilizados pelos filtros.

O ruído causa inúmeros problemas. Mas, poderíamos classificá-los em duas categorias básicas; os que não influenciam na detecção global da, ou das características buscadas, e os que não influenciam na detecção destas características. O primeiro geralmente conduz a um aumento do nível computacional necessário para identificar algum aspecto a partir da característica filtrada já o segundo é mais complicado, pois os ruídos acentuam o nível global do erro da característica filtrada podendo conduzir a resultados completamente errados.

Em nosso trabalho felizmente estaremos suscetíveis ao primeiro dos erros especialmente em se tratando da detecção de pele que sofre com os falsos positivos e os falsos negativos.

Para reduzir o nível de ruído em nossas imagens utilizaremos técnicas simples da morfologia matemática comumente conhecidos como erosão e dilatação. Contudo eles não são capazes de resolver por completo o problema, mas conferem um nível significante de melhora no processo de detecção de pele.

2.4.1 MORFOLOGIA MATEMÁTICA

O termo morfologia vem do grego, *morphê* (forma) e *logos* (ciência) e é a ciência que estuda as formas que a matéria pode tomar (FACON, 1996).

A morfologia matemática por sua vez leva em conta modelos matemáticos. Isto significa que são consideradas estruturas matemáticas para descrever as formas que a matéria pode tomar.

Quando nos referimos a matéria temos que ter em mente que estamos nos restringindo a imagens. Embora, por definição imagens não sejam matéria e sim representações gráficas, plásticas ou fotográficas de objetos, as imagens podem conter qualquer tipo de matéria.

Justamente por tratar de qualquer tipo de matéria, a morfologia pode ser aplicada em inúmeras áreas. De fato, é possível analisar matéria da biologia, da metalografia, da medicina, da visão robótica, de controle de qualidade, do reconhecimento de padrões e muitas outras.

Em termos de imagens, a morfologia matemática, que representa um ramo do processamento não linear, permite processar imagens com diversos objetivos, como por exemplo, realce, segmentação, detecção de bordas, esqueletização e afinamento.

O princípio básico da morfologia matemática consiste em extrair informações relativas à geometria e à topologia de um conjunto desconhecido de uma imagem. A potencialidade da morfologia matemática reside nos chamados elementos estruturantes.

Definição: Elemento Estruturante

É um conjunto, completamente definido e conhecido (forma e tamanho), que é utilizado como elemento comparador, a partir de uma transformação, ao conjunto desconhecido da imagem.

A Figura 2.11 mostrar um elemento estruturante (Círculo) sendo utilizado como elemento estruturante de uma imagem genérica (Forma cinza).

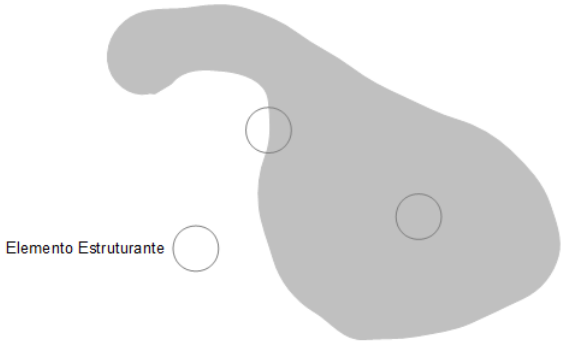


Figura 2.11: Elemento Estruturante (FACON, 1996)

O resultado da transformação permite avaliar o conjunto desconhecido.

O formato e tamanho do elemento estruturante possibilitam testar e quantificar de que maneira o elemento estruturante "está ou não está contido" na imagem.

• CONCEITOS BÁSICOS

Serão apresentados conceitos básicos que são importantes para compreender o funcionamento dos operadores morfológicos que serão vistos nas subseções 2.4.6 e 2.4.7. Mais do que apresentá-los, a propriedades que veremos dão subsídios para não só prever resultados mas também nos possibilitam compor elementos estruturantes afim de obter um dado resultado, que em nosso caso em especial consistirá na redução de ruídos.

Translação e Simetria

Seja um subconjunto B de um conjunto Γ. Esse subconjunto pode sofrer algumas modificações. Por exemplo, o conjunto obitdo de B pela translação do vetor h, denotado $B + h$, é:

$$B + h = \{\, y \in \Gamma \mid y - h \in B \,\} \tag{2.24}$$

Este conjunto transladado, chamado B_h, pode ser também definido como:

$$y \in B \Rightarrow y + h \in B_h \tag{2.25}$$

com as seguintes propriedades:

$$B_0 = B$$
$$(B_h)_k = B_{h+k} = (B_k)_h$$

O conjunto deduzido de B por simetria central pela origem O do sistema de referência, denotado por $\tilde{B}$ e chamado de B transposto é:

$$y \in B \Rightarrow -y \in \tilde{B} \tag{2.26}$$

$$B_h = \left(\tilde{B}\right)_{-h} \tag{2.27}$$

- **OPERAÇÕES DE MINKOWSKI**

Sejam dois subconjuntos de A e B de Γ. A adição de Minkowski de A com B, denotada $A \oplus B$, é o seguinte conjunto:

$$A \oplus B = \{\, x \in \Gamma \mid \exists a \in A \, e \, \exists b \in B,\, x = a + b \,\} \tag{2.28}$$

$$A \oplus B = \bigcup_{b \in B} A_b \tag{2.29}$$

Da mesma forma, a subtração de Minkowski do subconjunto A com o subconjunto B, denotada $A \ominus B$, é:

$$A \oplus B = \{\, x \in \Gamma \mid \forall b \in B,\, \exists a \in A,\, x = a - b \,\} \tag{2.30}$$

$$A \ominus B = \bigcap_{b \in B} A_b \tag{2.31}$$

- **PROPRIEDADES BÁSICAS**

Toda operação compõem-se de uma transformação Ψ de um conjunto em outro e de medida μ. $\Psi(X)$ é um conjunto, $\mu(\Psi(X))$é um número. Esses operadores não são simplesmente matemáticos. Seus efeitos não podem violar a realidade física que eles representam. Ψe μ devem respeitar as condições as condições de contorno impostas pelas leis da percepção visual:

a) Invariância à translação (Figura 2.12);

b) Invariância à homotetia (Figura 2.13);

c) Conhecimento local;

d) Continuidade.

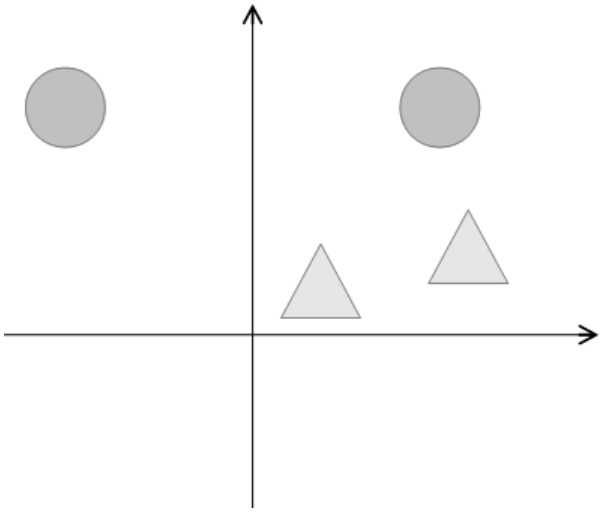


Figura 2.12: Translação e Simetria

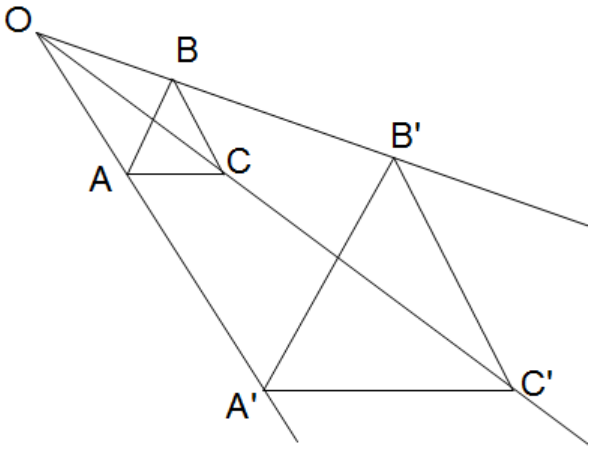


Figura 2.13: Homotetia

Seja X_h o conhecimento transladado do conjunto X pelo vetor h. De forma geral, existem dois tipos de transformações, as que não dependem da posição do referêncial e as que dependem desta posição.

2.4.2 MORFOLOGIA BINÁRIA

A morfologia binária trata especificamente de imagens em dois níveis de cinza, preto e branco. É clara a analogia que pode ser feita com os estados lógicos zero (0) para o preto

e um (1) para o branco, essa analogia nos permite utilizar diversas operações lógicas sobre as imagens.

Devido ao fato da morfologia binária basear-se em imagens em dois níveis de cinza, são necessárias etapas prévias a fim de obtermos uma imagem limiarizada apropriada para que através da morfologia binária obtenhamos os resultados esperados.

Veremos duas técnicas fundamentais e de grande utilidade na morfologia binária: a erosão e a dilatação. Embora simples, estas duas técnicas isoladamente ou combinadas nos permitem obter resultados surpreendentes.

- **EROSÃO**

Definição: A erosão de um conjunto X pelo elemento estruturante B é:

$$ero^B(X) = X\, ero\, B = \{\, X \in \Gamma \mid B_X \in X \,\} \tag{2.32}$$

$$X = \begin{bmatrix} \circ & \circ & \circ & \circ & \circ \\ \circ & \bullet & \bullet & \bullet & \circ \\ \circ & \bullet & \bullet & \bullet & \circ \\ \circ & \bullet & \bullet & \bullet & \circ \\ \circ & \circ & \circ & \circ & \circ \end{bmatrix} e\, B = \begin{Bmatrix} \circ & \bullet & \circ \\ \circ & (\bullet) & \circ \\ \circ & \bullet & \circ \end{Bmatrix}$$

$$\begin{bmatrix} \circ & \circ & \circ & \circ & \circ \\ \circ & [\bullet] & \bullet & \bullet & \circ \\ \circ & \bullet & \bullet & \bullet & \circ \\ \circ & \bullet & \bullet & \bullet & \circ \\ \circ & \circ & \circ & \circ & \circ \end{bmatrix} ero \begin{Bmatrix} \circ & \bullet & \circ \\ \circ & (\bullet) & \circ \\ \circ & \bullet & \circ \end{Bmatrix} = \begin{bmatrix} \circ & \circ & \circ & \circ & \circ \\ \circ & (\circ) & \circ & \circ & \circ \\ \circ & \circ & \circ & \circ & \circ \\ \circ & \circ & \circ & \circ & \circ \\ \circ & \circ & \circ & \circ & \circ \end{bmatrix}$$

$$\begin{bmatrix} \circ & \circ & \circ & \circ & \circ \\ \circ & \bullet & \bullet & \bullet & \circ \\ \circ & \bullet & \bullet & \bullet & \circ \\ \circ & \bullet & \bullet & \bullet & \circ \\ \circ & \circ & \circ & \circ & \circ \end{bmatrix} ero \begin{Bmatrix} \circ & \bullet & \circ \\ \circ & (\bullet) & \circ \\ \circ & \bullet & \circ \end{Bmatrix} = \begin{bmatrix} \circ & \circ & \circ & \circ & \circ \\ \circ & \circ & \circ & \circ & \circ \\ \circ & \bullet & \bullet & \bullet & \circ \\ \circ & \circ & \circ & \circ & \circ \\ \circ & \circ & \circ & \circ & \circ \end{bmatrix}$$

Figura 2.14: Erosão

- **DILATAÇÃO**

Definição: A dilatação de um conjunto X pelo elemento estruturante B é:

$$dil^B(X) = X\ dil\ B = \{\, X \in \Gamma \mid B_X \in X \,\} \tag{2.33}$$

$$X = \begin{bmatrix} \circ & \circ & \circ & \circ & \circ \\ \circ & \bullet & \bullet & \bullet & \circ \\ \circ & \bullet & \bullet & \bullet & \circ \\ \circ & \bullet & \bullet & \bullet & \circ \\ \circ & \circ & \circ & \circ & \circ \end{bmatrix} e\ B = \begin{Bmatrix} \circ & \bullet & \circ \\ \circ & (\bullet) & \circ \\ \circ & \bullet & \circ \end{Bmatrix}$$

$$\begin{bmatrix} \circ & \circ & \circ & \circ & \circ \\ \circ & [\bullet] & \bullet & \bullet & \circ \\ \circ & \bullet & \bullet & \bullet & \circ \\ \circ & \bullet & \bullet & \bullet & \circ \\ \circ & \circ & \circ & \circ & \circ \end{bmatrix} dil \begin{Bmatrix} \circ & \bullet & \circ \\ \circ & (\bullet) & \circ \\ \circ & \bullet & \circ \end{Bmatrix} = \begin{bmatrix} \circ & \bullet & \circ & \circ & \circ \\ \circ & \bullet & \circ & \circ & \circ \\ \circ & \bullet & \circ & \circ & \circ \\ \circ & \circ & \circ & \circ & \circ \\ \circ & \circ & \circ & \circ & \circ \end{bmatrix}$$

$$\begin{bmatrix} \circ & \circ & \circ & \circ & \circ \\ \circ & \bullet & \bullet & \bullet & \circ \\ \circ & \bullet & \bullet & \bullet & \circ \\ \circ & \bullet & \bullet & \bullet & \circ \\ \circ & \circ & \circ & \circ & \circ \end{bmatrix} dil \begin{Bmatrix} \circ & \bullet & \circ \\ \circ & (\bullet) & \circ \\ \circ & \bullet & \circ \end{Bmatrix} = \begin{bmatrix} \circ & \bullet & \bullet & \bullet & \circ \\ \circ & \bullet & \bullet & \bullet & \circ \\ \circ & \bullet & \bullet & \bullet & \circ \\ \circ & \bullet & \bullet & \bullet & \circ \\ \circ & \bullet & \bullet & \bullet & \circ \end{bmatrix}$$

Figura 2.15: Dilatação

2.5 FILTRO HOMOMÓRFICO

O filtro homomórfico funciona atenuando as baixas frequências e realça as altas frequências baseado no modelo de iluminação-reflectância (PARKER, 2012).

O filtro homomórfico trabalha com a idéia de que a componente de iluminação (I) é componente de baixa frequência e a reflectância (R) de alta frequência.

A idéia do filtro é aumentar o contraste da imagem atuando sobre as componentes de iluminação e reflectância. Aumenta-se o contraste se a iluminação é diminuída ($I < 1$) e a reflectância é aumentada ($R > 1$).

Para enteder o funcionamento do filtro vamos entender como uma imagem é expressa em função da iluminação-reflectância.

Seja a imagem G cujos pixels são expressos por $f(x, y)$, então:

$$f(x, y) = i(x, y).r(x, y)$$

, onde

$i(x, y)$ representa a componente de iluminação e,

$r(x, y)$ representa a componente de reflectância.

Em geral a filtragem homomórfica é feita no domínio espacial e para isso utiliza a Transformada de Fourier (DFT) e sua inversa (DFT^{-1}), além das funções logarítmica (ln) e exponêncial (exp), tal como mostrado na Figura 2.16.

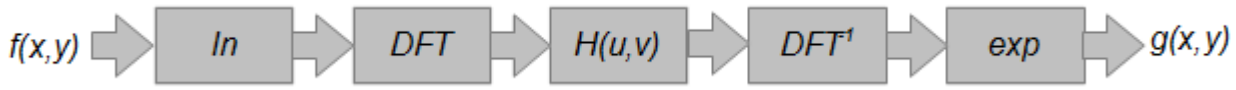
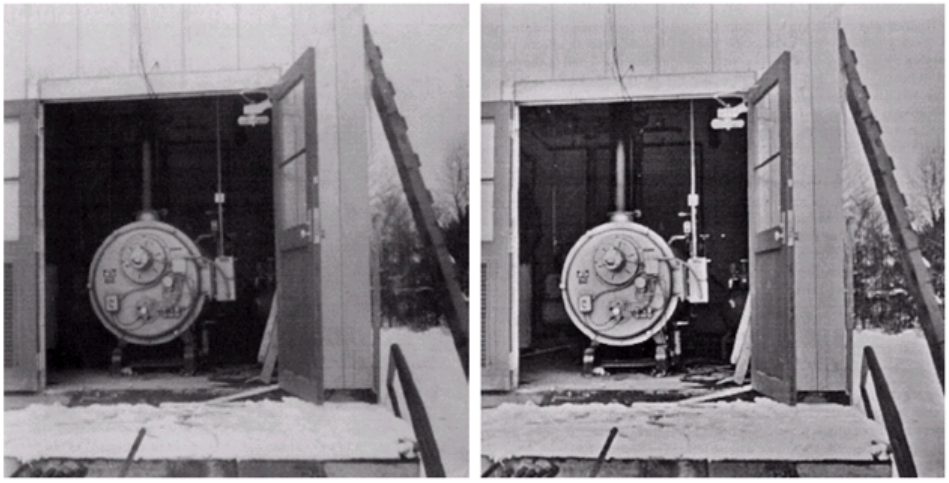


Figura 2.16: Filtragem Homomórfica no domínio da frequência



(a) (b)

Figura 2.17: Imagem Original (a); Após Filtragem Homomórfica (b) (PARKER, 2012)

O fato de usar a Transformada de Fourier inviabiliza a utilização do filtro para aplicações móveis uma vez que essa transfomada é muito honerosa computacionalmente. Contudo é possível manter o conceito da filtragem homomórfica sem contudo recorrer ao domínio da frequência, isto é mantendo o domínio da frequência.

O que faremos é substituir a sequência DFT, $H(u, v)$e DFT^{-1} da Figura 2.16, por uma aplicação de filtros passa-baixa e passa-alta.

Dado um pixel $f(x, y)$ da imagem original, seu novo valor $g(x, y)$será dado por:

1. Primeiramente aplicaremos a função logarítmica (ln) sobre a imagem original;

2. Aplicaremos um filtro passa-baixa ($L(u, v)$) sobre a imagem original, armazenado a imagem resultante.

3. Aplicaremos um filtro passa-alta ($H(u, v)$) sobre a imagem original, armazenado a imagem resultante.

4. Efetuaremos uma soma ponderada das imagens obtidas no passos 2 e 3 obtendo assim uma nova imagem.

5. Por fim aplicaremos a função exponencial (exp) para obeter os valores de $g(x, y)$ e então reconstruir a imagem.

A Figura 2.18 mostra o passo a passo da aplicação do filtro.

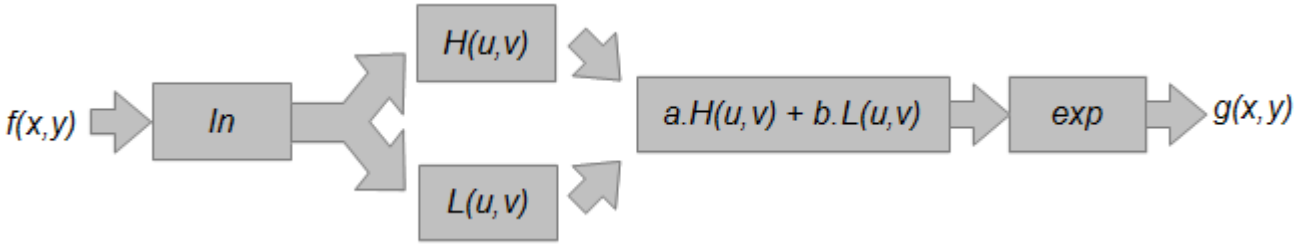
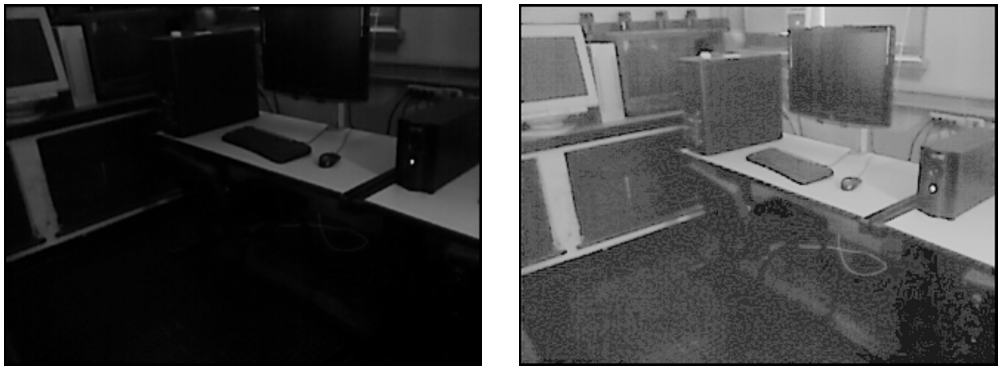


Figura 2.18: Filtragem Homomórfica no domínio espacial

A Figura 2.19 mostra o resultado da filtragem homomórfica proposta.



(a) (b)

Figura 2.19: Filtragem Homomórfica proposta: Imagem Original (a); Após Filtragem (b)

3 AMBIENTE DE DESENVOLVIMENTO

3.1 ANDROID

O Android é um projeto de código aberto iniciado pela Google. Ele foi desenvolvido com enfoque nas plataformas móveis (Telefones e Tablets) ele constitui-se de um sistema operacional e kits de desenvolvimento (SDK e NDK). Ele é desenvolvido pela Open Handset Alliance (OHA), um consórcio de 65 empresas de tecnologia. Entre elas estão a própria Google, a Motorola, a HTC, a Qualcomm e a T-Mobile.

Como colocado em GOOGLE DEVELOPERS, 2013, o Android é mais do que um sistema operacional. Ele é na verdade uma pilha de software composto por cinco camadas, como mostra a Figura 3.1. A base do Android é uma versão modificada do kernel Linux 2.6, que prove vários serviços essenciais, como segurança, rede e gerenciamento de memória e processos, além de uma camada de abstração de hardware para as outras camadas de software.

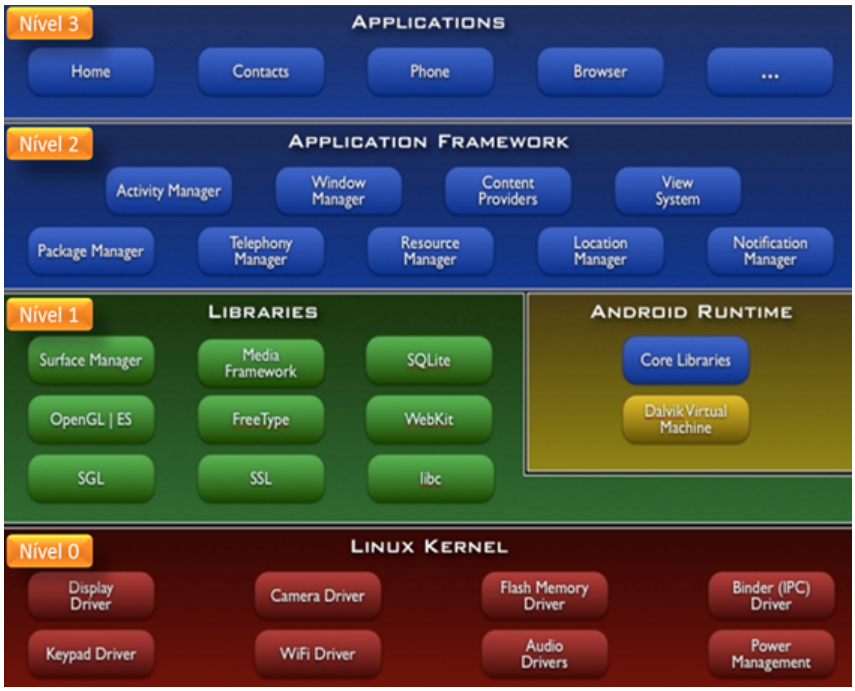


Figura 3.1: Arquitetura Android (GOOGLE DEVELOPERS, 2013)

Acima do kernel ficam as bibliotecas C/C++ utilizadas por diversos dos componentes

do sistema, como: uma implementação da biblioteca padrão do C (*libc*) otimizada para dispositivos embarcados; bibliotecas para suporte a formatos de áudio, vídeo e imagens; um gerenciador que intermedia o acesso ao display e compõe as camadas de imagem 2D e 3D; o engine para navegadores *WebKit*; bibliotecas para gráficos 2D (SGL) e 3D (OpenGL ES); um renderizador de fontes bitmap e vetoriais; e o banco de dados relacional *SQLite*.

No Android, aplicações escritas em Java são executadas em sua própria máquina virtual, que por sua vez é executada em seu próprio processo no Linux, isolando-a de outras aplicações e facilitando o controle de recursos. O Android Runtime é composto pela máquina virtual chamada Dalvik VM, onde as aplicações são executadas, e por um conjunto de bibliotecas que fornecem boa parte das funcionalidades encontradas nas bibliotecas padrão do Java.

Na camada acima, escrita em Java, fica a framework de aplicações, que fornece todas as funcionalidades necessárias para a construção de aplicativos, através das bibliotecas nativas. Aplicações Android podem possuir quatro tipos de componentes: activities, services, content providers e broadcast receivers. Além destas peças fundamentais em uma aplicação, existem os recursos, que são compostos por layouts, strings, estilos e imagens e o arquivo de manifesto, que declara os componentes da aplicação e os recursos do dispositivo que ela irá utilizar (GOOGLE DEVELOPERS, 2014).

3.1.1 ANDROID SDK

Para desenvolver aplicações para a plataforma android a Google disponibiliza um SDK[1] (*Software Development Kit*). Nesse kit são disponibilizadas todas as ferramentas necessárias para construir uma aplicação em Android desde um emulador até um plugin (ADT - *Android Development Tool*) para integrar o Android ao Eclipse permitindo o desenvolvimento através de um editor gráfico. O debug pode ser feito através do emulador disponibilizado ou diretamente num dispositivo Android. Além disso, as aplicações podem ser distribuídas através de aquivos **.apk* que podem ser instaladas em quaisquer dispositivos que atendam os requisitos da aplicação.

[1]Disponível em: http://developer.android.com/sdk/index.html

3.1.2 ANDROID NDK

O NDK (*Native Development Kit*) é um conjunto de ferramentas que permite a implementação de código nativo no Android, como C e C++, disponível em GOOGLE DEVELOPERS, 2013.

A integração do código nativo com o código em Java é feita do mesmo jeito que em qualquer outro lugar: através do JNI - *Java Native Interface*. Através do JNI é possível chamar métodos em C++ do Java e vice versa, assim como a comunicação do Java com outras linguagens como C e Assembly.

O código nativo possui esse nome, pois é o código programado nas linguagens do próprio sistema operacional, por isso o nome nativo, eles falam a mesma língua. Com isso, sua performance é bem maior que em outras linguagens. No caso do Android, o código nativo possui mais um fator que lhe permite ser mais rápido, pois o Java é uma linguagem que precisa ser interpretada em tempo de execução pela JVM, já o código nativo é inteiramente compilado.

Como as outras plataformas também oferece suporte para desenvolvimento de código C e C++ se for bem estruturado o código, o mesmo código pode ser usado para outras plataformas, sendo necessário mudar somente a parte do código que utiliza as APIs nativas. Vários aplicativos e jogos que estão disponíveis para Android e IOS utilizam grande parte do mesmo código através do código nativo. Não só códigos para dispositivos móveis, mas também podendo reutilizar o mesmo código para plataforma desktop e web.

3.1.3 API JAVA

Oficialmente o Android não suporta Java SE nem Java ME. Basicamente temos que reescrever ou modificar aplicações existentes para torná-las compatíveis com o sistema. Entretanto exitem partes destas APIs que são suportadas pelo Android. Entre elas podemos citar as classes *Vector* e *HashMap*. O Android introduziu uma API própria para interface gráfica do usuário (GUI) devido a descontinuidade das GUIs próprias do Java (*Swing* e *AWT*). Outro aspecto relevante é que Java Beans não é suportado embora seja utilizado por diversos projetos *open source.*

3.1.4 API GRÁFICA

Android suporta OpenGL ES, uma versão desenvolvida especialmente para dispositivos embarcados. A OpenGL ES 1.0 e 1.1 são suportadas por todos os dispositivos, a versão 2.0 somente para dispositivos com versão Android 2.2 ou maior e apartir do Android 4.3 a versão OpenGL 3.0 passou a ser suportada motivada em parte pelo crescente mercado de jogos para os dispositivos móveis permitindo a criação de jogos mais realistas e com efeitos visuais melhores.

3.1.5 DISPOSITIVOS DE TESTES

Nesta pesquisa foram utilizados três aparelhos de testes que estão listados em Figura 3.2, Figura 3.3 e Figura 3.4.

Android 2.3 (Gingerbread), Processor Single-Core de 1 GHz, 398 MB de RAM, Câmera de 3.2 Mega-Pixels

Figura 3.2: Galaxy S Wi-Fi 5.0

Figura 3.3: Motorola Moto G - Android 4.4.2 (KitKat), Qualcomm Snapdragon 400 de 1.2 GHz, 1 GB RAM, Câmera de 5 Mega-Pixels

Figura 3.4: Asus Nexus 10 - Android 4.3 (Jelly Bean), Processador Dual-Core de 1.7 GHz Cortex-A15, 2 GB de RAM, Câmera de 5 Mega-Pixels

3.2 FRAMEWORK PARA TESTES

Em decorrência da necessidade da realização de testes a fim de avaliar os resultados dos algoritmos estudados ao longo deste trabalho um framework de processamento de imagens foi construído, além de sistematizar o processo de testes o framework é robusto o suficiente para auxiliar o desenvolvimento de diversas outras aplicações que façam uso de técnicas de processamento de imagem.

Chamado de Dolphin Framework foi projetado segundo um sistema de camadas que representam as etapas básicas de um sistema genérico de processamento de imagem que são: Aquisição, Análise e Apresentação.

A Figura 3.5 mostra como estão organizadas as etapas de um sistema genérico de processamento de imagem tal como um sistema de camadas.

Figura 3.5: Etapas de processamento de imagem organizado em camadas

A camada de aquisição é responsável por adquirir as imagens da câmera. A camada de análise é responsável por processar as imagens adquiridas da câmera. Por fim a camada de apresentação é responsável por apresentar resultados provenientes da análise, como por exemplo, exibir as imagens processadas ou mesmo tomando alguma decisão baseada na imagem processada e exibindo ao usuário alguma informação.

Essa divisão em camadas além de dividir funcionalmente o framework permite que ele seja utilizado em conjunto com diversas APIs, bibliotecas e mesmo outros frameworks, como será o caso do nosso sistema de testes que incorporada à camada de análise a biblioteca OpenCV para realizar o reuso dos diversos algoritmos de processamento de imagens existentes. Já na camada de apresentação será incorporada a biblioteca OpenGL para e exibição das imagens resultantes do processamento de imagem realizado na camada de análise.

O Framework está disponível em `http://github.com/tlimao/dolphin`.

3.3 ARQUITETURA DO FRAMEWORK

O Dolphin framework foi desenvolvido para aplicações que utilizem visão computacional como aplicações de realidade aumentada e processamento de imagem na plataforma Android. Ele oferece orientação a objeto através da linguagem Java interfaceando chamadas nativas à linguagem C, conferindo assim todos os benefícios de uma linguagem

orientada a objeto aliada a velocidade da linguagem C.

O framework tem por objetivo permitir a criação de diversas aplicações que façam uso de processamento de imagem a partir de imagens obtidas através da câmera de dispositivos Android. Simplificadamente, temos a etapas descritas na Figura 3.6 numa aplicação típica de processamento de imagem.

Início	1. Iniciar a captura de vídeo (Streaming de vídeo)
Loop Principal	2. Processar os frames do streaming de vídeo 2.1 Aplicar filtros de processamento de imagem 2.2 Extrair informações da imagem pré-processada 2.3 Concluir a partir das informações extraídas 3. Exibir resultados obtidos a partir do processamento da imagem
Fim	4. Finalizar a captura de vídeo

Figura 3.6: Descrição básica do framework

A etapa 1 é responsável por iniciar a captura do vídeo, cujos frames serão submetidos ao processamento de imagem. Nesta etapa também são definidos os parâmetros da câmera como formato de vídeo e resolução da imagem.

A etapa 2 é onde os frames serão processados segundo, mas não obrigatoriamente, segundo as etapas 2.1, 2.2 e 2.3.

A etapa 3 é responsável por exibir para o usuário, resultados obtidos através do processamento das imagens.

A etapa 4 é uma etapa de finalização, onde os recursos de hardware e software, como liberar a câmera e descarregar bibliotecas auxiliares utilizadas como o OpenCV, respectivamente.

A Figura 3.7 mostra a arquitetura do framework. A *Standard Library* é uma biblioteca que utiliza a API disponibilizada pelo Android para acessar a câmera e receber o streaming de vídeo. As imagens obtidas são manipuladas conjuntamente pela biblioteca de processamento de imagens OpenCV e por algoritmos da biblioteca *ImgProc*. Concorrentemente as imagens da câmera são convertidas para o espaço de cores apropriado para serem carregadas e exibidas como texturas da OpenGL. O Multithreading bem como a sincronização necessária estão descritos na seção 3.6.

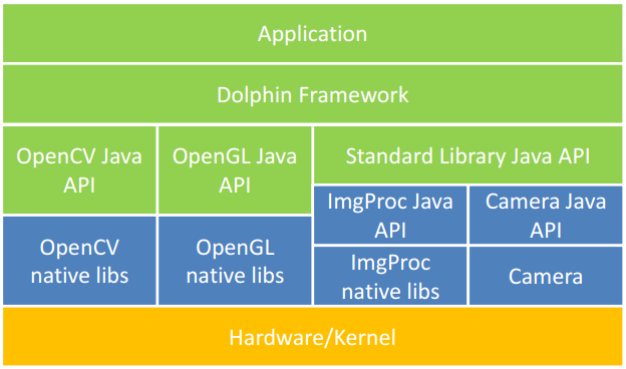


Figura 3.7: Arquitetura do Framework

3.4 EXIBIÇÃO DE IMAGENS E RENDERIZAÇÃO

Existem diversas maneiras de exibir as imagens resultantes do processamento de imagem inclusive utilizando a API disponibilizada pelo Android, contudo optou-se por utilizar o OpenGL para a apresentação das imagens resultantes do processamento de imagem pois desta forma tornamos o framework mais flexível podendo utilizá-lo como um framework de realidade aumentada, por exemplo.

Há contudo um outro motivo que torna a exibição de imagens através de texturas OpenGL mais atrativa: a renderização via OpenGL conta com uma thread dedicada, assim podemos obter desempenho maior utilizando esse aspecto, por exemplo podemos colocar a etapa de conversão de espaço de cor como uma tarefa delegada a thread do OpenGL e em sistemas que contam com GPU dedicada o desempenho é muito superior pois tais GPUs são poderosos coprocessadores que contam com diversas otimizações para a manipulação de texturas, um exemplo é o nvidia tegra (NVIDIA, 2013).

3.4.1 CONVERSÃO DE ESPAÇOS DE CORES

Para obter o streaming de vídeo em dispositivos Android é necessário implementar um método callback, esse método será chamado toda vez que um novo frame chega da câmera, fornecendo imagens como arrays de bytes. Os formatos suportados pelo hardware

da câmera são restritos. Por padrão, o YCbCr 4:2:0 SP é definido nos dispositivos Android (GOOGLE DEVELOPERS, 2013). Todo dispositivo Android deve suportar esse formato por padrão.

O formato YCbCr 4:2:0 SP é ilustrado na Figura 3.8. Os primeiros width x height bytes contém a informação de luminância da imagem, é nada mais que uma imagem em níveis de cinza. Estes bytes são seguidos pela informação de crominância, para cada quatro bytes de luminância há um byte para diferença de azul (Cb) e um byte para a diferença de vermelho (Cr) tal como as Eq. 2.7 e Eq. 2.8 respectivamente.

As imagens da câmera são no formato YCbCr 4:2:0 SP contudo o OpenGL só suporta texturas RGB, logo é necessária a conversão entre os espaços de cores.

YCbCr 4:2:0 sp

Como array e bytes:

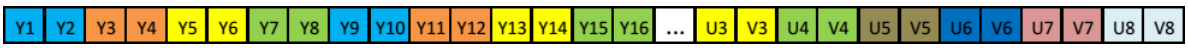


Figura 3.8: Formato YCbCr 4:2:0 SP

A conversão do YCbCr 4:2:0 SP para RGB é feita segundo as Eq. 3.1 , Eq. 3.2e Eq. 3.3.

$$R = Y + 1.403 * (Cr - 128) \tag{3.1}$$

$$G = Y - 0.714 * (Cr - 128) - 0.344 * (Cb - 128) \tag{3.2}$$

$$B = Y + 1.773 * (Cr - 128) \tag{3.3}$$

3.5 COMPONETES DO FRAMEWORK

Como já mencionado anteriormente um framework de processamento de imagem, contém três etapas básicas que podem ser organizadas segundo um sistema de camadas. Cada camada do framework por sua vez tem suas delegações, a camada de aquisição de imagens irá comunicar-se com a câmera do dispositivo obtendo assim os frames gerados pela mesma, a camada de análise irá processar as imagens provenientes da câmera e por fim a camada de apresentação irá mostrar os resultados do processamento.

Para aumentar a manutenibilidade optou-se por inicialmente construir uma biblioteca básica denominada *Standard Library* que reúne as classes fundamentais para a aquisição de imagens da câmera e de interfaces para cada camada do sistema de camadas da Figura 3.5. A partir dessa biblioteca construiu-se o framework propriamente dito. Há também uma biblioteca denominada *ImgProc* contendo alguns dos algoritmos de segmentação como o detector de pele, filtro homomórfico e diferença de frames que não são disponibilizados diretamente pelo OpenCV.

Para maiores detalhes conferir as referências do framework.

- *STANDARD LIBRARY*

Para tornar o framework extensível, todas as classes necessárias ao correto funcionamento de cada camada estão organizadas numa biblioteca denominada *Standard Library*. Essa bibliotecaé composta da classe abstrata *FrameGrabber*, das interfaces *FrameProcessor*, *FrameContainer* e *FrameListener* e da classe abstrata *GlColorSpace*. Além dessas há também a classe abstrata que contém todas as constantes do framework.

A classe abstrata *FrameGrabber* é a responsável por comunicar-se com a câmera, iniciando o canal de comunicação entre o hardware e o software. Nela também são ajustados os parâmetros da câmera como dimensões dos frames que serão gerados, esquema de cores, modos de cena, modos de foco, etc. Nela não é implementado o método que permite a manipulação dos frames uma vez que isso será definido na classe concreta do framework.

A interface *FrameProcessor* encapsula um método denominado *doFrameProcessing* que irá realizar o processamento de imagem sobre o frame que for passado ao método.

A interface *FrameContainer* é uma interface que permite adquirir frames de qualquer classe de objeto que a implemente, bem como as dimensões espaciais da imagem (width e height), a princípio qualquer classe que receba ou manipule um frame da câmera pode implementar essa interface.

A interface *FrameListener* é a interface implementada por toda classe do framework que por ventura necessite de imagens provenientes da câmera, isto é, toda vez que um novo frame for gerado pela câmera e capturado por uma classe que extenda a classe *FrameGrabber* os objetos do tipo *FrameListener* associados ao este *FrameGrabber* serão atualizados com o novo frame.

A classe abstrata *GLColorSpace* é responsável por converter os frames da câmera em arrays de de bytes RGB que são compreeendidos pela OpenGL através do método *toRGB*, que recebe como parâmetros um frame da câmera no espaço YCbCr bem como as dimensões espaciais da imagem (width e height).

A Figura 3.9 mostra o diagrama de classes da *Standard Library*.

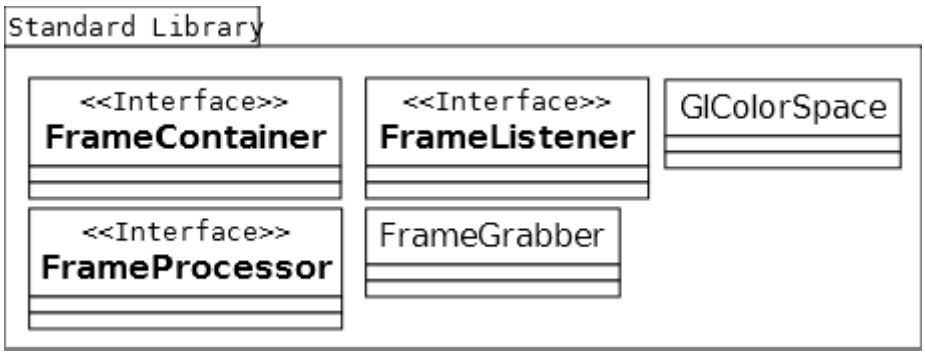


Figura 3.9: Diagrama de classes simplificado da *Standard Library*

- *IMGPROC*

As classe que constam nesta biblioteca são a *FrameDiff*, *SkinDetect* e *HomomorphicFilter*, que são todas abstratas, pois são apenas interfaces para chamadas nativas, uma vez que os algoritmos estão implementados em C. Todas recebem como parâmetro um frame da câmera bem como suas dimensões horizontal e vertical (width e height).

A classe *FrameDiff* tem um particularidade que ela foi projetada para frames em *streaming*, ou seja, é preciso uma sequência de frames (pelo menos 2) para que seja possivel executar a diferença.

A classe *SkinDetect* fornede duas implementações para detecção de pele. A primeira, realizada através do método *skinDetect1*, faz a detecção de pele no frame passado como parâmetro baseado no critério mostrado na subseção 2.3.5. Já o método *skinDetect2* faz a detecção de pele no frame passado como parâmetro baseado no critério mostrado na subseção 2.3.6.

A classe *HomomorphicFilter* faz a filtragem homomórfica tal como descrito na seção 2.5.

A Figura 3.10mostra o diagrama de classe da *ImgProc*.

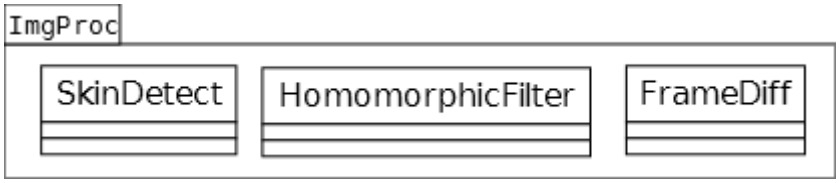


Figura 3.10: Diagrama de classes simplificado da *ImgProc*

- *FRAMEWORK*

As classes que compõem o framework são a *FrameDispatcher*, a *FrameBuffer*, *FrameProcessorWorker* e a *FramePreviewSurface*.

A classe *FrameDispatcher* extende a classe *FrameGrabber* e a cada frame gerado pela câmera é a responsável por entregar a todos objetos do tipo *FrameListener* (Ouvintes de Frames) o frame gerado bem como suas dimensões. Esta classe corresponde a camada de aquisição mencionada capítulo 3.

A classe *FrameBuffer* implementa as interfaces *FrameListener* e *FrameContainer*. Esta classe é uma classe intermediária entre as camadas de Aquisição e Análise e sua função é evitar o problema evidênciado por DOMHAN, 2010, em que há esgotamento da memória do dispositivo devido ao excesso de frames. Além disso, essa classe extende a classe *Observable* (GANG OF FOUR, 1994) uma vez que ela será observável por qualquer que implemente a interface *Observer* (GANG OF FOUR, 1994) que tenha interesse nos frames gerados pela câmera como é o caso da classe *FrameProcessorWorker* que é responsável por realizar o processamento de imagem.

A classe *FrameProcessorWorker* implementa as interfaces *FrameContainer*, *Observer* e *Runnable*. Os objetos desta classe executam em thread dedicada exclusivamente para o processamento de imagem realizados por objetos de classes que implementam a interface *FrameProcessor*. Além disso, ela extende a classe *Observer*, assim toda vez que um frame tiver sido processado, qualquer objeto que observe esta classe é notificado de que um frame foi processado, como é o caso de objetos da classe *FramePreviewSurface*, responsáveis por exibir o resultado do processamento de imagem. Maiores detalhes sobre multithreading serão abordados na seção 3.6.

A classe *FramePreviewSurface* é uma superfície OpenGL (extende *GlSurfaceView*) e nela será exibida na forma de uma textura OpenGL o frame resultante do processamento da imagem. Ela implementa as interfaces *Renderer* e *Observer*, esta última necessária pois ela observa a classe objetos da classe *FrameProcessorWorker* e toda vez que um frame é processado ela atualiza a textura OpenGL com o frame processado. Antes de renderizar o

frame para exibição essa classe chama o método *toRGB* da classe abstrata *GlColorSpace*, para que seja possível exibir o frame como uma textura OpenGL.

A Figura 3.11 mostra o diagrama de classes simplificado do framework.

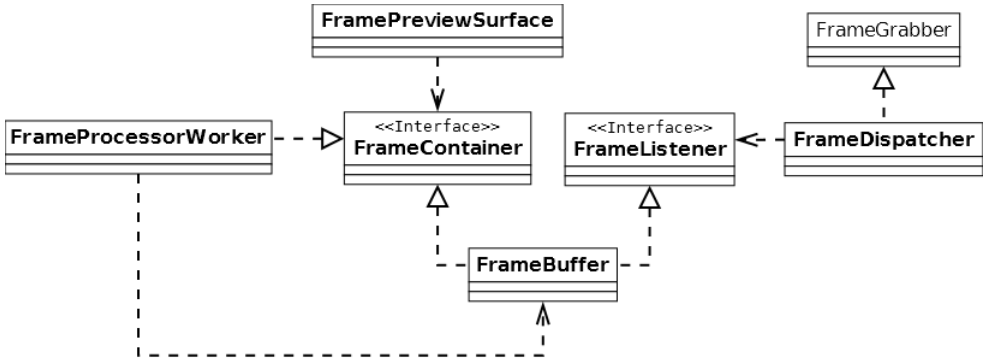


Figura 3.11: Diagrama de classes simplificado do Framework

3.6 MULTITHREADING

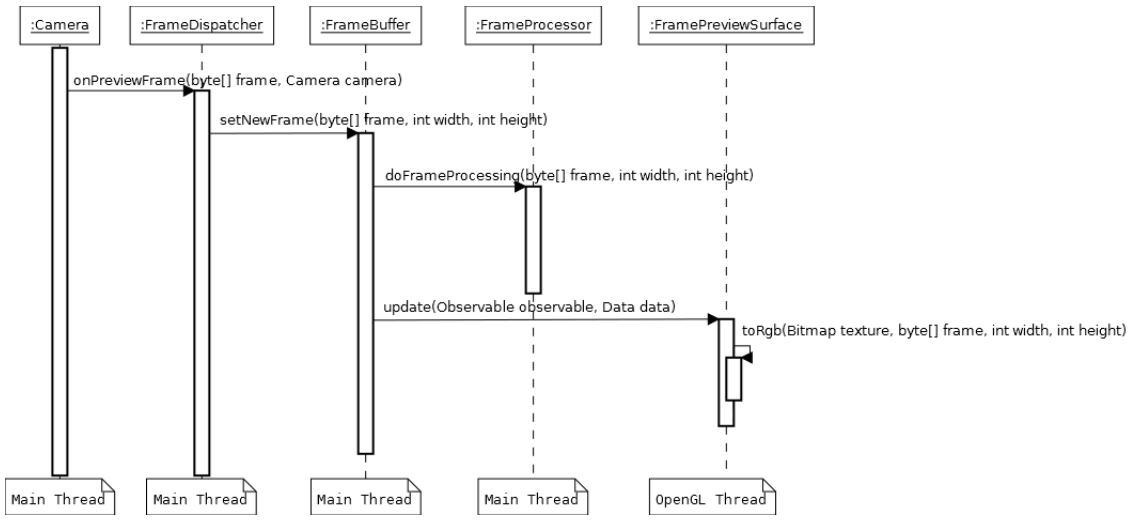
Um método *callback* é chamado toda vez que um novo frame é enviado da câmera, disponibilizando a imagem como um array de bytes. Quando um novo frame chega da câmera temos que realizar dois trabalhos. Um deles é o processamento da imagem com o intuito de extrair a informação de interesse. O segundo é necessário caso queiramos exibir a imagem resultante do processamento da imagem da câmera. Para exibi-la como já mencionado optou-se pela exibição como uma textura OpenGL, para isso é necessário convertê-la para um espaço de cores apropriado ao OpenGL.

Embora a conversão da imagem processada para o espaço de cores do OpenGL seja feita posteriormente ao processamento da imagem, uma vez que a imagem for processada ela pode ser delegada à etapa de conversão de espaço de cores e imediatamente a etapa de processamento fica livre para processar outro frame da câmera.

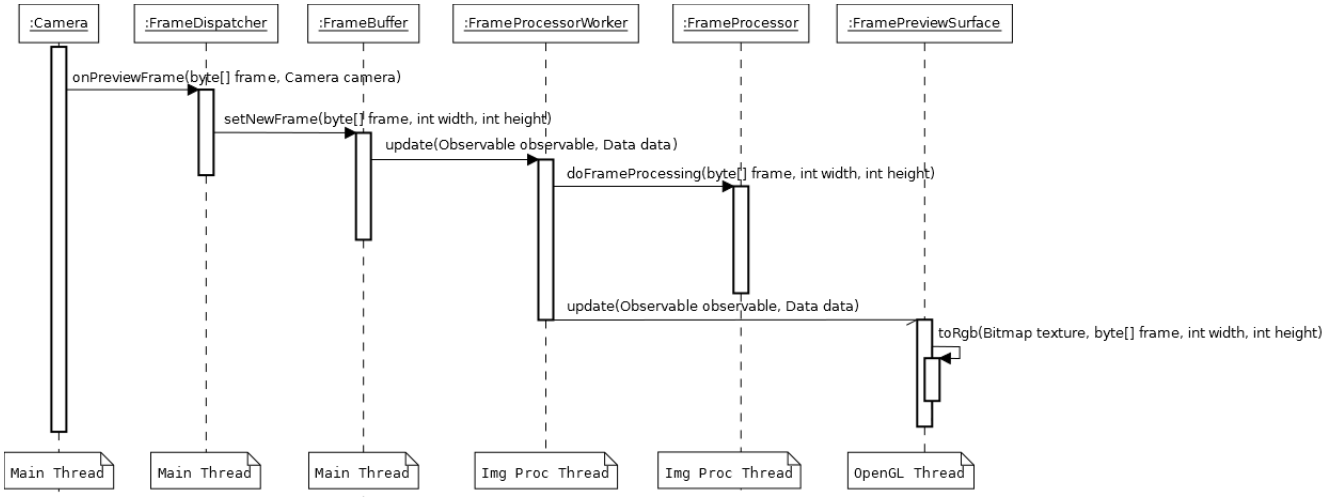
Testes iniciais revelaram que se sequenciarmos a etapa de processamento da imagem e a etapa de conversão de espaço de cores dentro do método callback, os frames provenientes da câmera são enfileirados até que o dispositivo fique sem memória ou na melhor das hipóteses será notado um retardo considerável entre os frames.

Para resolver esse problema optou-se por uma abordagem concorrente através de uma estrutura multithread em que uma thread, de máxima prioridade, destinada exclusivamente ao processamento de imagem, enquanto que a exibição e conversão de imagens

para o espaço de cores compatível com o OpenGL foi delegada a thread da OpenGL. A aquisição de imagem apartir da câmera é mantida na thread principal do sistema Android. A Figura 3.12 (a) e (b) contém o diagrama de sequência seguido por um sistema básico de processamento de imagem segundo as abordagens sequencial e concorrente, respectivamente.



(a)



(b)

Figura 3.12: Diagrama de sequência típico de um sistema de processamento de imagem

As threads trabalham continuamente e concorrentemente da seguinte forma: quando um novo frame chega da câmera a thread principal notifica a thread de processamento de imagem atualizando-a com o novo frame proveniente da câmera. Após a thread de processamento processar a imagem ela notifica a thread do OpenGL, atualizando-a com o frame recém processado que por sua vez é transformado numa textura OpenGL para então ser exibida. Por questões de desempenho foi criado um sistema com travas de acesso

às seções críticas e variáveis de controle em cada thread. Toda vez que um novo frame é entregue a thread de processamento de imagem uma variável é modificada indicando que se trata de um novo frame permitindo que a entrada na seção crítica da thread que é responsável por processar a imagem. Após o processamento da imagem esta variável muda para uma condição que indica que o frame já foi processado e qualquer nova tentativa processar o frame novamente é impedida até que se trate de um novo frame. Após o frame ter sido processado a thread do OpenGL é notificada e o frame processado é passado para ela , neste momento uma variável de controle é modificada indicando que se trata de um novo frame, no momento da exibição o frame só é convertido para o espaço de cores do OpenGL se a variável de controle indicar que se trata de um novo frame e após a conversão a mesma variável é modificada para uma condição que indica que o frame já foi convertido, isso impede que um frame seja convertido mais de uma vez para o espaço de cores do OpenGL o que acabaria minando o desempenho com conversões desnecessárias. Estas variáveis de controle também são responsáveis por permitir ou não a aquisição das travas para acesso às seções críticas das threads.

A Figura 3.13 mostra um comparativo entre a abordagem sequencial e a abordagem concorrente realizada no dispositivo de teste mencionado na subseção 3.1.5. Os testes consistem em submeter o dispositivo uma cadeia de processamento de imagem que encadeia 10 filtros canny e 10 filtros morfológicos de dilatação alternadamente sob diferentes resoluções de imagens da câmera (800 x 480 e 320 x 240).

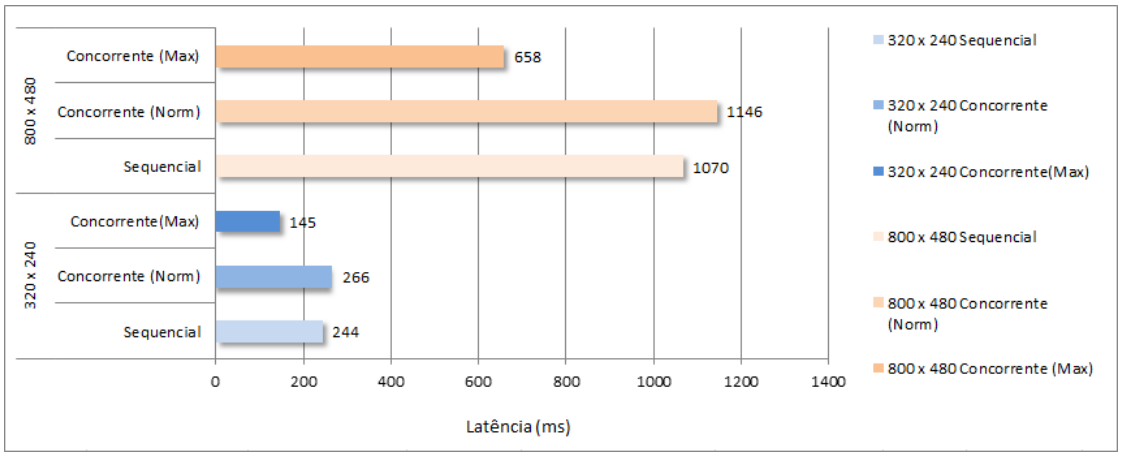


Figura 3.13: Comparativo entre abordagem sequencial e concorrente num dispositivo single-core

Como pode ser observado ao migrar da abordagem sequencial para concorrente com a thread de processamento de imagem com prioridade normal ou média, a latência do processamento de imagem degrada um pouco, cerca de 9%. Isso é devido ao chaveamento entre as threads. Contudo colocando a thread de processamento de imagem com prioridade máxima observamos um incremento de desempenho de cerca de 40%.

É importante ressaltar que estes testes iniciais foram feitos num dispositivo single-core, ou seja, a abordagem multithread aqui faz uso da divisão de tempo de processamento somente.

Na Figura 3.14 foram realizados testes sob as mesmas condições que os testes anteriores contudo foi utilizado um dispositivo quad-core, agora estamos utilizando uma abordagem que faz uso da divisão física de processamento.

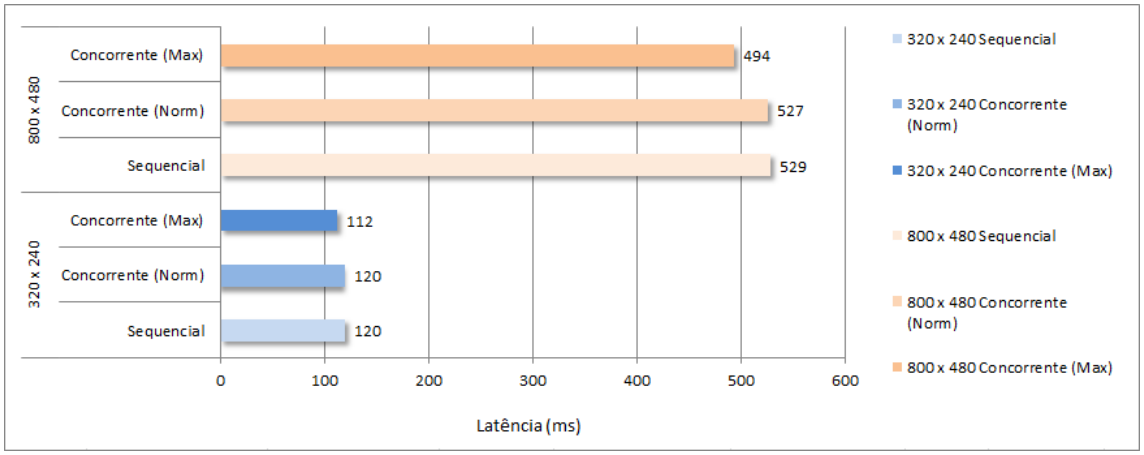


Figura 3.14: Comparativo entre abordagem sequencial e concorrente num dispositivo quad-core

Como podemos perceber no gráfico da Figura 3.14, como não há o custo do chaveamento entre as threads do OpenGL, do Processamento de Imagem e da User Interface, não há degradação de desempenho ao passar da abordagem sequencial para a abordagem concorrente. É importante ressaltar que como há quatro núcleos a thread do OpenGL é automaticamente direcionada para um núcleo físico, assim mesmo na abordagem sequencial já há um multithread com divisão física. No entanto quando tornamos a thread de processamento de imagem com prioridade máxima, observa-se uma ligeira melhora de cerca de 6% na latência do processamento de imagem.

A conclusão que se chega é que independênte de quantos núcleos físicos de processamento tenha o dispositivo, a abordagem multithread é melhor não só pela ganho de performace, mais também sob o ponto de vista de flexibilidade do framework, pois, permite que várias thread trabalhem concorrentemente possivelmente buscando extrair características diferentes ou mesmo utilizando abordagens distintas para detecção de uma determinada característica nas imagens entregues pela câmera.

4 TÉCNICA DE DETECÇÃO DA MÃO

Neste capítulo apresentaremos uma técnica para detecção de uma mão que é uma adaptação da técnica apresentada em DAVID, 2012.

A técnica consiste em inicialmente filtrar a imagem extraindo os pixels que correspondem a pele, em seguida, extraímos a borda dos pixels que foram filtrados admitindo que a mão é o elemento mais significativo, selecionamos a borda de maior perímetro. Após a extração do contorno é feita a aproximação do mesmo por um polígono seguido da detecção das regiões convexas onde é aplicada uma heurística para determinar o centro da mão e as pontas dos dedos.

A seguir são apresentadas as diversas técnicas empregadas no processo de detecção da mão.

4.1 DETECÇÃO DE BORDAS

A partir de uma imagem binária correspondendo aos pixels classificados como pele, precisamos determinar quais destes correspondem à mão. Para isso é necessário identificar o contorno da mão na imagem.

A detecção de bordas é uma ferramenta muito utilizada em processamento de imagem e visão computacional especialmente quando é necessário extrair ou detectar características da imagem.

Uma borda pode ser definida como o limite entre os elementos que estão representados na imagem.

A detecção de bordas consiste basicamente em identificar os pixels da imagem que apresentam saltos de luminância (variação brusca de iluminação). Quando as bordas da imagem são corretamente identificadas, os objetos da cena podem ser localizados, o que permite determinar características desses objetos como área, perímetro e forma, que podem por exemplo serem usadas para classificá-los ou identificar um objeto específico.

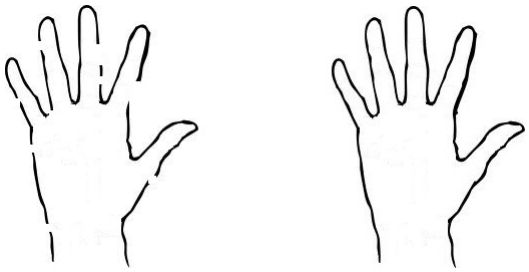
O método mais comum utilizado para detecção de bordas baseia-se no gradiente da imagem. Esse método consiste em identificar os locais onde a magnitude da intensidade do gradiente (que é a mudança que ocorre na intensidade dos pixels em deslocamentos

pequenos entre pixels adjacentes) ultrapassa um limiar, para indicar de forma confiável a borda de um objeto.

Existem diversos algoritmos para detectar bordas baseado em variação de gradiente (Sobel, Prewitt, Roberts, ...), contudo nesta pesquisa fui utilizado o detector ótimo Canny (PARKER, 2011), que além de gerar bordas bem finas garante a localização correta da borda.

4.2 FECHO CONEXO E FECHO CONVEXO

Ainda que tenhamos bordas bem definidas com o detector de bordas de Canny, estas bordas não são conexas, isto é, apresentam uma série de descontinuidades ao longo das bordas. Isto é ruim para inferirmos qualquer coisa sobre os objetos aos quais tais bordas pertencem. Por isso é comum utilizar técnicas que geram componentes conexas a partir dessas bordas.



(a) (b)

Figura 4.1: (a) Borda Detectada; (b) Fecho Conexo

Para gerar esse conjunto conexo, selecionamos um pixel e recursivamente adicionamos a um conjunto de pixels que também são pixels que representam o contorno e que são adjacentes de pelo menos um pixel no conjunto, esse processo é feito até que não existam mais pixels adjacentes no contorno. Se sobrarem pixels no contorno previamente encontrado, então seleciona-se outro pixel e reinicia-se o processo até que todos os pixels do contorno estejam num componente conexo.

Assumimos que a mão é componente conexo de maior perímetro, enquanto que os demais são descartados.

Existem diversas formas de gerar esse conjunto conexo, nesta pesquisa foi utilizada a aproximação por polígono que consiste basicamente, em aproximar os contornos detectados pelo detector de bordas Canny por polígonos fechados.

Após determinarmos o fecho conexo estamos aptos a delimitar a região da imagem que contém a mão, para isso utilizamos um algoritmo denominado *Convex Hull* ou Fecho Convexo, que determina baseado no fecho conexo que acabamos de determinar, o menor polígono convexo que contém inteiramente a mão, esse polígono convexo nos auxiliará na detecção das regiões convexas que veremos a seguir na seção 4.3.

Figura 4.2: Convex Hull (Fecho Convexo)

4.3 REGIÕES CONVEXAS

A partir dos feixos conexo e convexo podemos determinar as regiões convexas da mão segundo o algoritmo *Convexity Defects* disponibilizado pelo OpenCV. A partir do fecho conexo esse algoritmo retorna as regiões convexas.

Uma região conevexa é definida segundo 4 elementos (Figura 4.3): ponto inicial da região convexa (s), ponto final da região convexa (e), ponto mais profundo da região convexa (d) e profundidade da região (l).

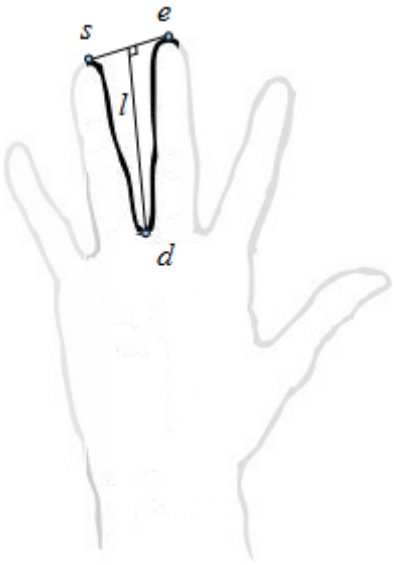


Figura 4.3: Elementos de uma região convexa

Para determinar estas regiões convexas entre os dedos, varremos os pontos que compõem cada segmento do fecho conexo do contorno evidenciando os pontos inicial (s), final (e) e mais profundo (d) tal como mostrado na figura Figura 4.3. Os pontos inicial (s) e final (e) facilmente determinados uma vez que são os limites de cada segmento do fecho convexo, mais profundo (d) é o ponto de inflexão da região convexa é determinado tomando-se o ponto do contorno cuja distância euclidiana até o segmento do fecho convexo correspondente é o máximo. A profundidade é a distância euclidiana do ponto mais profundo e o segmento de reta que une os pontos inicial (s) e final (e).

4.4 HEURÍSTICA DE DETECÇÃO DOS DEDOS

Detectar os dedos da mão é fundamental para detecção do gesto feito pela mão, uma vez que os gestos previstos são baseados na quantidade de dedos e na posição relativa entre eles.

Em geral utilizam-se heurísticas baseadas na topologia da mão para determinar os dedos e então identificar o gesto que está sendo feito, a robustez dessa heurística influi diretamente nos tipos de gestos que serão suportados pelos sistemas de detecção.

Nesta pesquisa focaremos na detecção dos gestos mostrados na Figura 4.4, que correspondem os números de zero a cinco.

Figura 4.4: Gestos suportados pelo sistema

Assim os gestos que iremos detectar correspondem a quantidade de dedos que estão estendidos, não importando, contudo o quanto e quais dedos estão estendidos. Esta restrição representa uma vantagem pois a detecção não será sensível a qual mão está sendo utilizada (direita ou esquerda) e se estamos utilizando a palma ou a parte superior da mão.

Para determinar quantos dedos estão estendidos primeiro devemos analisar a topologia da mão humana mostrada na Figura 4.5, nela podemos notar que os dedos determinam regiões convexas bem pronunciadas.

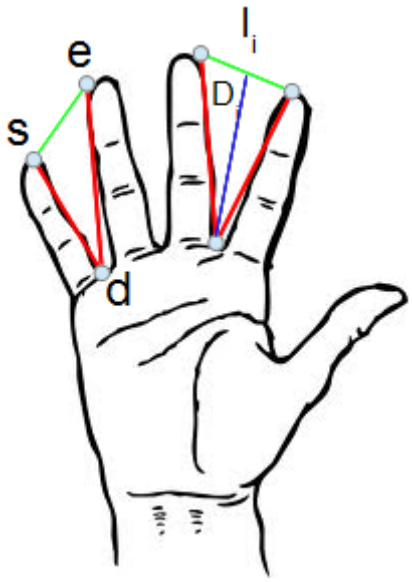


Figura 4.5: Topologia da mão humana

Apartir destes pontos verificamos o que chamamos de coeficiente de profundidade μ_d (4.1) que consiste em dividir o tamanho de cada segmento do fecho convexo pela profundidade a região convexa correspondente, assim, comparando com um limiar pré-estabelecido, avaliamos se os dedos estão estendidos ou não.

$$\mu_d = \frac{l_i}{D_i} m \tag{4.1}$$

onde, l_i é o tamanho do segmento convexo e D_i é a profundidade da região convexa correspondente.

Contudo, utilizar somente esse critério, não gera bons resultados, conduzindo a muitos falsos positivos. Para melhorar o processo de detecção dos dedos estimamos o centro da mão (C_h) apartir do pontos d de cada região convexa e, então, fazemos uma média desse ponto com o centro geométrico G da região de interesse (Eq.4.2).

$$C_h = \left(\frac{\sum_1^n d_x}{n} + G_x, \frac{\sum_1^n d_y}{n} + G_y \right) \tag{4.2}$$

Determinado o centro da mão estimamos o raio médio da mão (R_h) que é dado pela média das distâncias D_i dos pontos d_i ao centro da mão (Eq.)

$$R_h = \left(\frac{\sum_1^n D_i}{n} \right) \tag{4.3}$$

Agora um candidato a dedo segundo a Eq.4.1 só é defato julgado como um dedo se seu ponto s estiver uma distância ao centro da mão superior ao raio médio. Por segurança esse distância é calculada colocando-se uma folga sobre o raio médio, isto é, define-se um raio médio $R_h^{'} = a.R_h$, onde $1 < a < 1.5$.

Apartir da quantidade de dedos detectados segundo esses critérios podemos inferir o gesto correspondente e uma possível ação associada ao gesto.

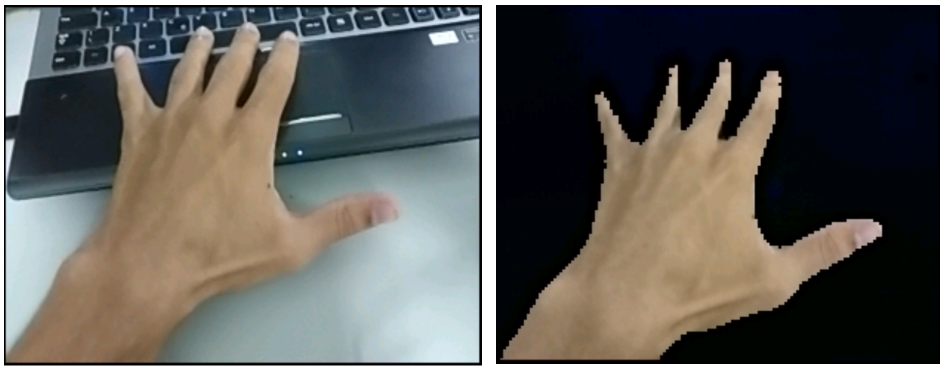
5 TESTES

5.1 DETECÇÃO DE PELE

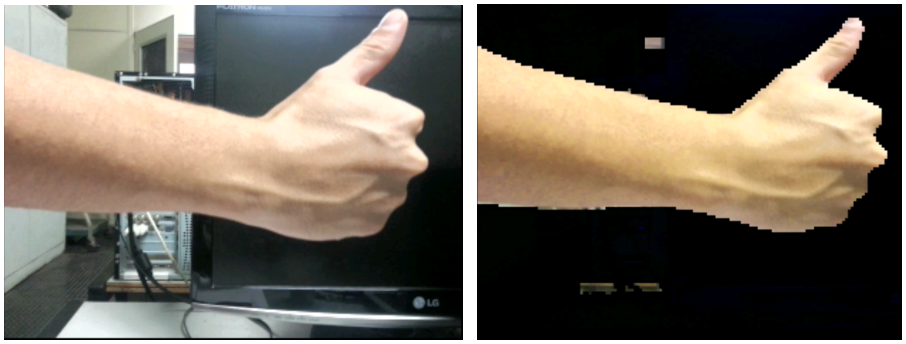
5.1.1 EMPREGO DO ESPAÇO DE CORES YCBCR

Como já constatado em diversos trabalhos, a detecção de pele é uma tarefa bem complicada, pois a iluminação interfere muito nos mecanismos de detecção a ponto de passar da condição de plena detecção à condição de total falha na detecção devido a variações de iluminação do ambiente. Aliado a isso existem também as falsas detecções de elementos da cena que são avaliados como se fossem pele como é caso de tons próximos ao da pele humana como, por exemplo, a madeira.

A Figura 5.1 (a) e (b) mostram testes realizados com a técnica de detecção apresentada na seção 2.3.5.



(a)



(b)

(c)

Figura 5.1: Detecção de Pele

Como podemos perceber na Figura 5.1 (b) a técnica utilizada para detecção de pele é suscetível a ruídos decorrentes tanto de falso positivos quanto de falso negativos para pele, para minimizar esse ruídos utilizou-se filtros morfológicos para eliminar pequenas ilhas na imagem filtrada e promovar o fechamento de buracos nas regiões de pele como veremos na seção 5.2. Contudo não há o que fazer em situações em que ocorra o mostrado na Figura 5.1 (c), onde ocorre uma falha grave na detecção de pele a ponto da mão praticamente não ser detectada.

5.1.2 EMPREGO DO ESPAÇO DE CORES LUX

Como já mencionado na subseção 2.3.6, o espaço YCbCr isoladamente não é eficaz na detecção de pele em decorrência das variações de iluminação, ainda que esse espaço de cores seja invariante sobre iluminação. Isso ocorre por que as câmera dos dispositivos implementam uma série de correções sobre a imagem como, por exemplo, a compensação de branco.

Nos testes realizados a simples utilização da filtragem baseada no espaço LUX, mostrou-se mais robusto as variações de iluminação e a objetos com cor similar a pele humana. Contudo os melhores resultados foram obtidos quando aplicou-se a filtragem homomórfica antes da filtragem da pele propriamente dita.

A Figura 5.2(a) mostra filtragem de pele no espaço YCbCr em comparação com as filtragens no espaço LUX sem (Figura 5.2 (b)) e com (Figura 5.2 (c)) aplicação da filtragem homomórfica, respectivamente.

(a) (b) (c)

Figura 5.2: (a) Detecção no Espaço YCbCr; (b)Detecção sem filtragem Homomórfica; (c) Detecção com filtragem Homomórfica

5.2 OPERADORES MORFOLÓGICOS

Como podemos perceber, a etapa de detecção de pele gera resultados com muitos ruídos, para minimizar a presença dos mesmos, utilizaram-se os operadores morfológicos mostrados na subseção 2.4.2.

Utilizando a erosão é possível eliminar pequenos ruídos que aparecem nas imagens. Contudo a erosão corrói elementos de interesse da imagem processada, para diminuir as perdas utilizamos após as erosões, dilatações.

A Figura 5.4 e Figura 5.5 mostram os resultados obtidos com as erosões e dilatações sobre a imagem previamente filtrada como filtro de detecção de pele (Figura 5.3).

Figura 5.3: Imagem Inicial (Após a detecção de pele)

Figura 5.4: Exemplo de Erosão (3x)

Figura 5.5: Exemplo de Dilatação (3x)

Inicialmente realizou-se 3 erosões sobre a imagem da Figura 5.3, como podemos ver na Figura 5.4 os pequenos ruídos que antes existiam foram eliminados, contudo parte da mão foi corroído no processo, para minimizar esse efeito de corrosão foram realizadas 3 dilatações comnsecutivas, com isso recuperamos partes da mão como podemos ver naFigura 5.5. A não correspondência perfeita entre as três imagens deve-se ao fato do teste ter sido realizado em tempo real sendo difícil manter perfeitamente imóvel mão e dispositivo.

5.3 DETECÇÃO DE BORDAS

A técnica para detecção de bordas foi o filtro ótimo de detecção de bordas Canny (PARKER, 2011) esse filtro além de extrair bordas com espessura de um pixel somente, também dá a localização exata da borda.

A Figura 5.6 é um exemplo do filtro detectando o contorno da mão.

Figura 5.6: Detecção de Bordas (Canny)

5.4 FECHO CONEXO E CONVEXO

Após determinarmos a borda da mão realizou-se a aproximação da mesmo por um polígono, linha branca mostrada na Figura 5.7.

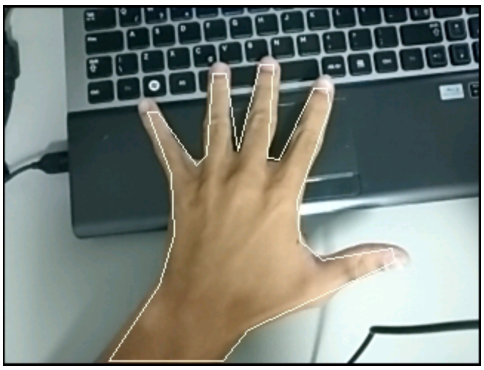


Figura 5.7: Aproximação por Polígono

A Figura 5.8 mostra o resultado da determinação do fecho convexo (Linha branca) descrito na seção 4.2.

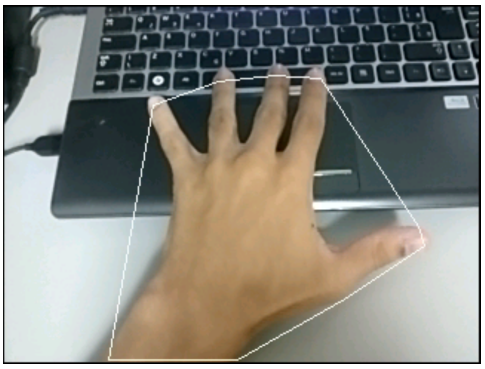


Figura 5.8: Fecho Convexo

5.5 REGIÕES CONVEXAS

A Figura 5.9 mostra o resultado da busca das regiões convexas da mão descrito na seção 4.3.

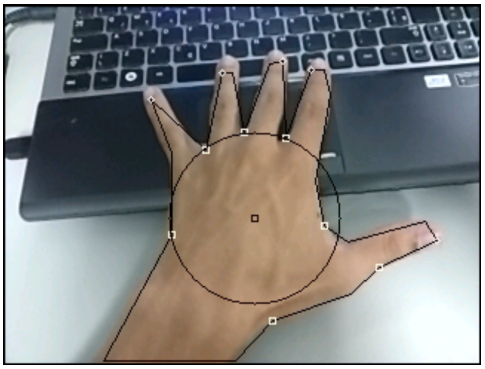


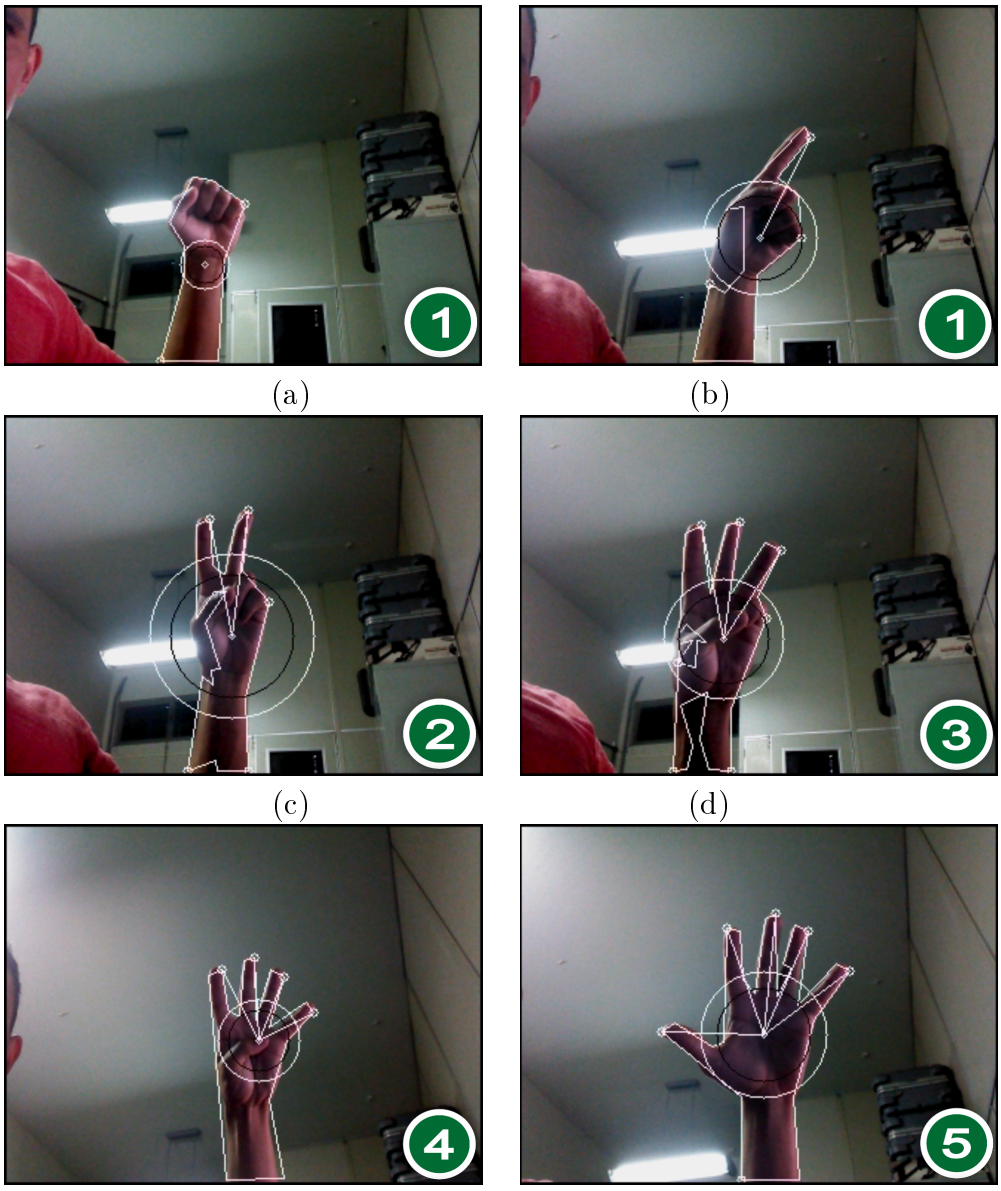
Figura 5.9: Regiões convexas da mão

5.6 DETECÇÃO DE GESTOS DA MÃO

Foram realizadas baterias de testes em tempo real para a deteção dos dedos das mãos em seis gestos distintos tal como mostrado na Figura 4.4 da seção 4.3.

Respeitadas boas condiçoes de iluminação os gestos correspondentes de 1 a 5 foram detectados em 90% dos casos enquanto que o gesto correspondente ao zero (mão fechada) não foi detectado com precisão devido a baixa robustez da heurística de detcção dos dedos que confunde protuberancias da mão fechada com dedos e indica falsos positivos para os gestos correspondentes de 1 a 5.

A Figura 5.10 mostra a detecção dos gestos de 0 a 5. Como podemos perceber na Figura 5.10 (a) a detecção do 0 (zero) é confundida com o 1 (um).



(a) (b) (c) (d) (e) (f)

Figura 5.10: Detecção de Gestos da Mão

6 CONCLUSÃO

A detecção de gestos e o rastreamento de movimento são factíveis nas plataformas móveis, contudo sofrem muito com as variações de iluminação e pelo fato dos dispositivos móveis não terem a restrição de estarem imóveis e em ambientes controlados. Para resolver a falta de robustez dos sistemas de reconhecimento de gestos e rastreamento de movimento é necessário que os algoritmos tornem-se mais complexos a fim de minimizar os efeitos externos como iluminação e movimento dos dispositivos.

O processo de detecção de pele no espaço YCbCr mostrou-se pouco robusto, pois sofre muito com as variações de iluminação, ainda que seja um espaço de cor invariante sobre a iluminação. Uma alternativa encontrada que melhora os resultados consiste em utilizar esse espaço em conjunto com o espaço LUX e utilizar uma pré-filtragem homomórfica dos frames. Além destas técnicas, também foi necessário o uso de operadores morfológicos para reduzir o ruído.

A utilização de um framework para auxílio na pesquisa mostrou-se fundamental uma vez que permite a realização de testes rápidos e a integração com bibliotecas já existentes como o OpenCV o que facilita a experimentação de diversas técnicas utilizando as mais diversas abordagens encontradas na literatura.

A respeito dos requisitos não funcionais do framework, obtivemos resultados muito bons uma vez que obtivemos foi superior ao framework disponibilizado pelo próprio OpenCV, obtendo taxas de latência de processamento de imagem mais baixas e, perceptualmente, taxas de frames por segundo mais altas.

REFERÊNCIAS

[1] GOOGLE DEVELOPERS. Web Site: https://developers.google.com/?hl=pt-BR.

[2] GONZALEZ, R.C. , WINTZ, P. Digital Image Processing, 2º Edition. Ed. Addison-Wesley Publishing Company, 1987.

[3] FACON, Jacques. Morfologia Matemática (Teoria e Exemplos). Ed. Champagnat da PUC Paraná – Curitiba, 1996.

[4] PARKER, J.R. Algorithms for Image Processing and Computer Vision, 2º Edition. Ed. Wiley, 2011.

[5] MOHAMED Alsheakhali, AHMED Skaik, MOHAMMED Aldahdouh, MOHAMOUD Alhelou. Hand Gesture Recognition System. Computer Engineering Dep., The Islamic University of Gaza, Gaza Strip, Palestina, 2011.

[6] AHMED, Elgammal, MUANG, Crystal, DUNXU Hu. Skin Detection – A Short Tutorial. 2009.

[7] MAHMOUD, Tarek M. A New Fast Skin Color Detection Technique. 2008.

[8] LECHETA, Ricardo R. Google Android: aprenda a criar aplicações para dispositivos móveis com o Android SDK. Ed. Novatec. 2ª ed. São Paulo, 2010.

[9] RATABOUIL, Sylvain. Android NDK, Beginner's Guide. 2º Edition. Ed. Packt, 2012.

[10] ECLIPSE. Disponível em: http://www.eclipse.org/downloads/.

[11] SUBVERSION. Disponível em: http://subversion.apache.org.

[12] OPENCV. Disponível em: http://sourceforge.net/projects/opencvlibrary/.

[12.1] OPENGL. Website. Disponível em: http://www.opengl.org/. Acesso em: 14 Jul. 2013.

[13] WIKIPEDIA. Disponível em: http://pt.wikipedia.org/wiki/Homotetia.

[14] NEWS BUREAU. Disponível em: http://news.illinois.edu/news/12/0206 brain Kyle Mathewson_MingHsu.mhtml.

[15] NVIDIA. Disponível em: http://www.nvidia.com.br/object/tegra-br.html

[16] DOMHAN, Tobias. Augmented Reality on Android Smartphones. Dualen Hochschule Baden-Württemberg Stuttgart, Stuttgart, Alemanha 2010.

[17] MAHMOUD,Tarek M. A New Fast Skin Color Detection Technique. World Academy of Science, Engineering and Technology, 2008.

[18] GANG OF FOUR. The "Gang of Four": Erich Gamma, Richard Helm, Ralph Johnson, John VlissidesDesign Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, USA, 1994.

[19] ALBIOL, Alberto, TORRES, Luis, EDWARD J. Delp. Optimum Color Spaces for Skin Detection. Polithecnic University of Valência, Spain, 2005.

[20] DAVID J. Rios-Soria, SCHAEFFER, Satu E., SARA E. Garza-Villarreal.Hand-gesture recognition using computer-vision techniques. Universidad Autónoma de Nuevo León (UANL) San Nicolás de los Garza, NL, Mexico, 2012.

[21] HUANG, Da-Yuan, FUH, Chiou-Shan. Automatic Face Color Enhancement. Dept. of Computer Science and Information Engineering, National Taiwan University.

[22] KUMAR, C. N. R. e BINDU, A. . An Efficient Skin Illumination Compensation Model for Efficient Face Detection, 2008. Disponível em: https://www.facedetect.741.com/.

[23] MELO, Rafael Heitor Correia de. VIEIRA, Evelyn de Almeida. TOUMA, Victor Lima. CONCI, Aura. Sistema de realce de detalhes ocultos em imagens com grande diferença de iluminação fazendo uso de filtragem não-linear. Universidade Federal Fluminense - UFF. Rio de Janeiro, 2005.

A INSTALAÇÃO

Os pré-requsitos para instalação no Windows, Mac e Linux são ter o Eclipse (ECLIPSE) e o SDK do Android (SDK) instalados.

Faça o download do Dolphin framework, disponível em `http://github.com/tlimao/dolphin`. Importe o Dolphin framework para o Eclipse (Figura A.1 a Figura A.3).

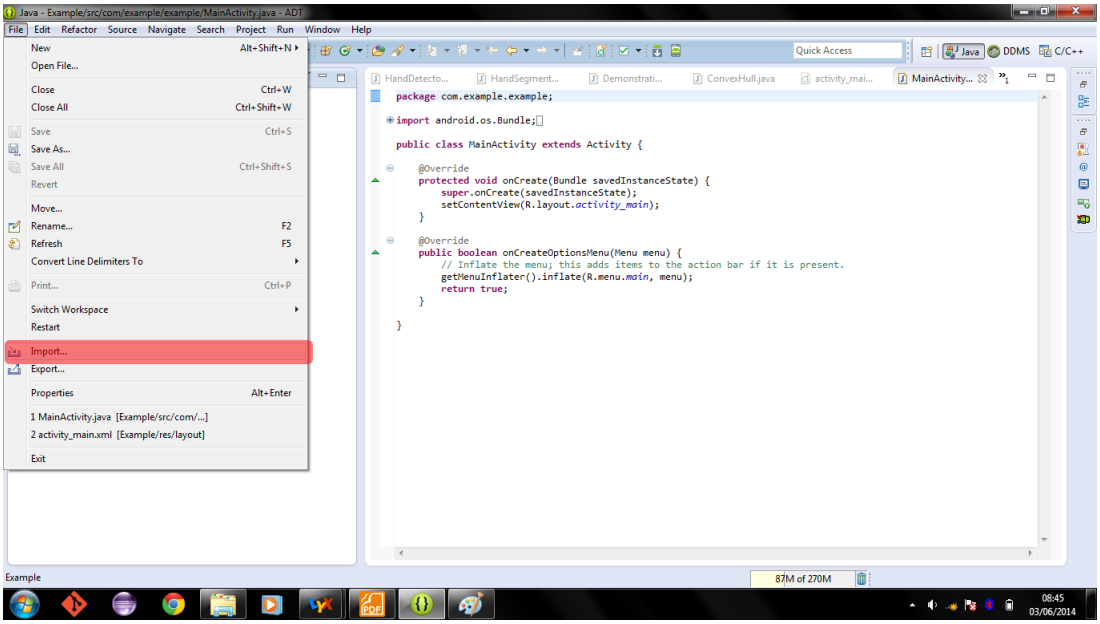


Figura A.1: Importação do framework

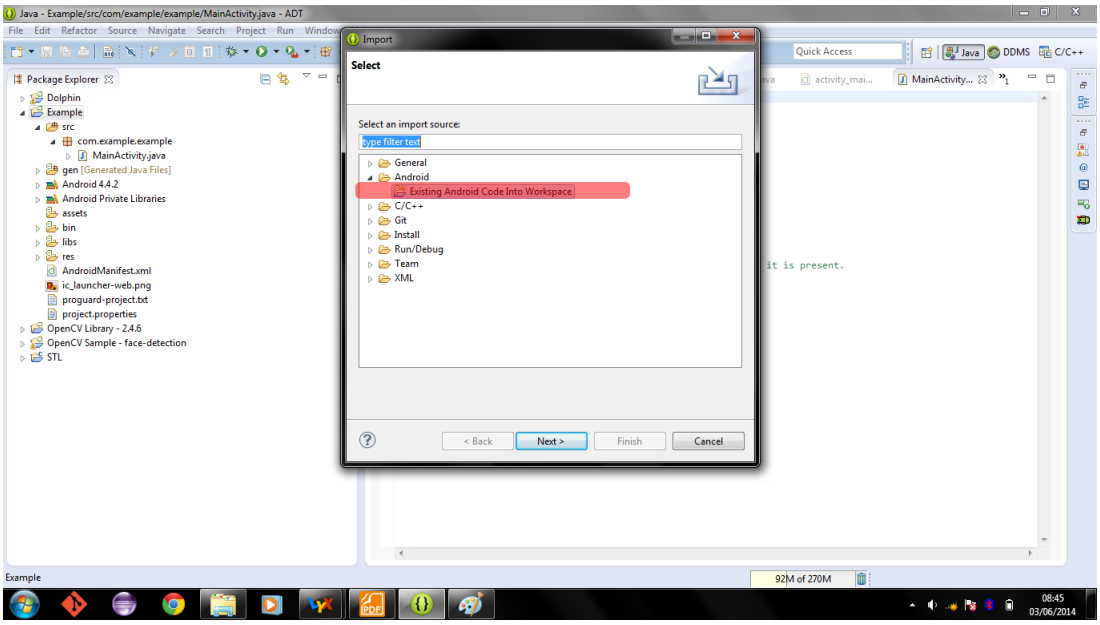


Figura A.2: Importação do framework

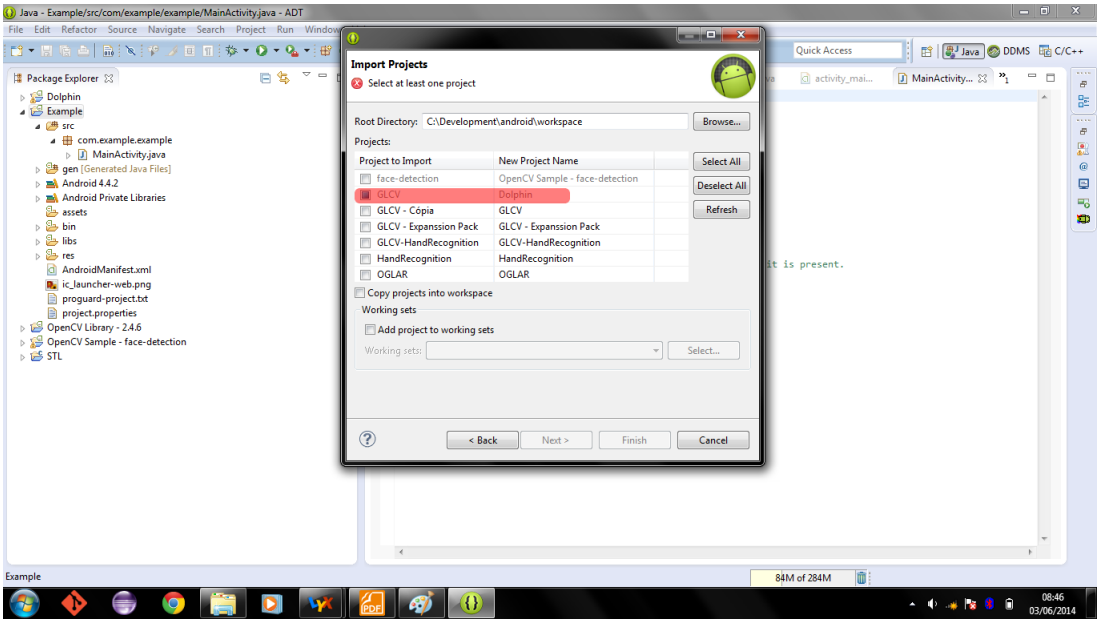


Figura A.3: Importação do framework

Crie um novo projeto Android e importe o Dolphin framework como biblioteca para o projeto recém criado (Figura A.4).

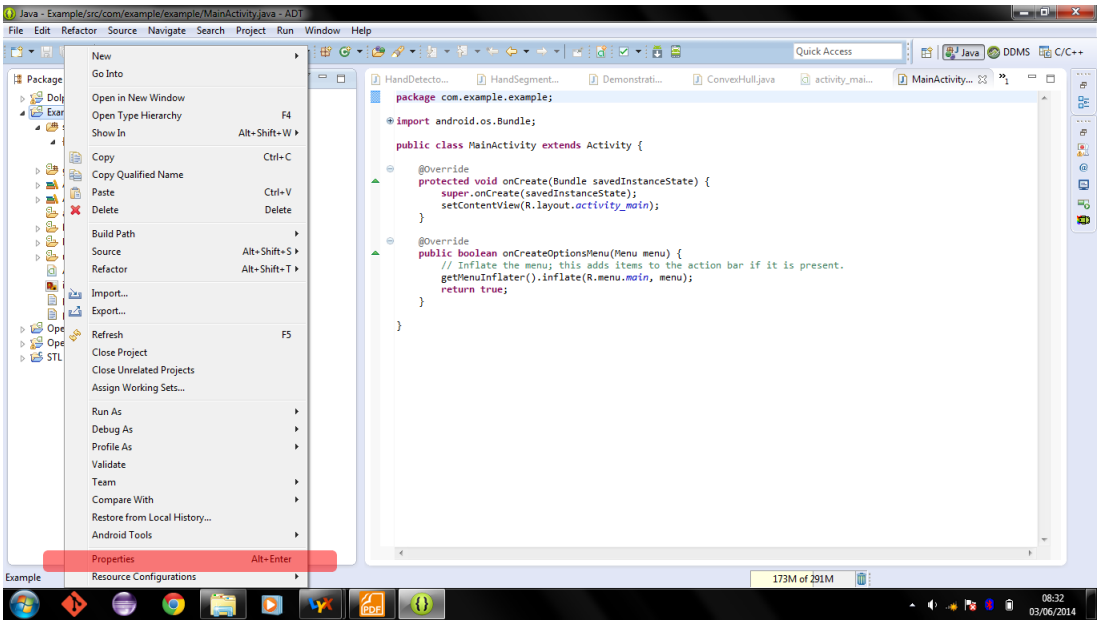


Figura A.4: Importação do framework para o projeto

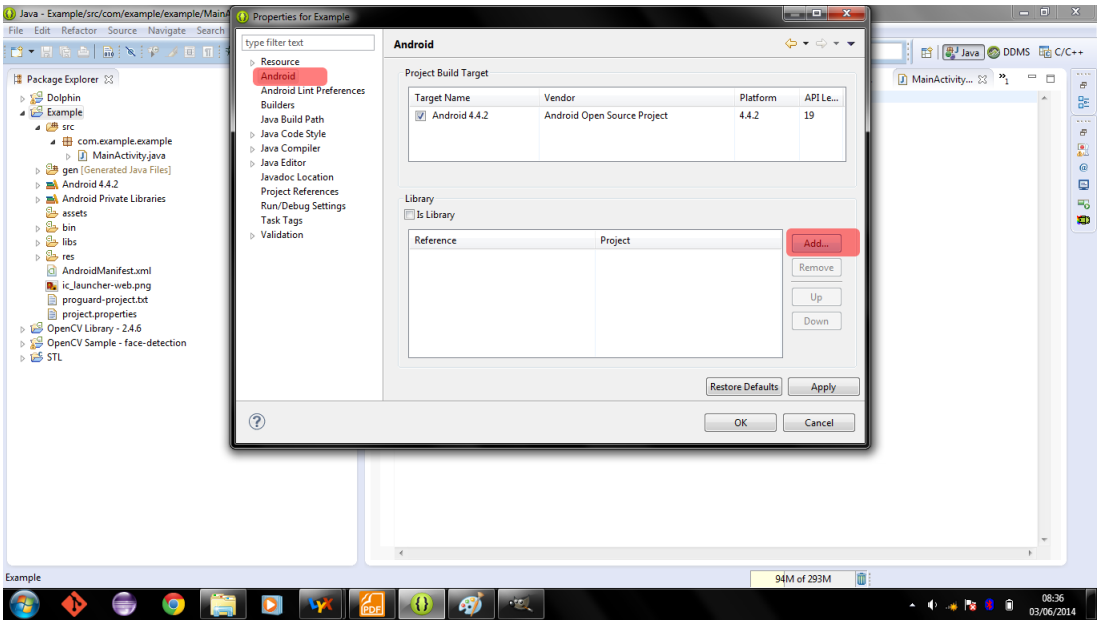


Figura A.5: Importação do framework para o projeto

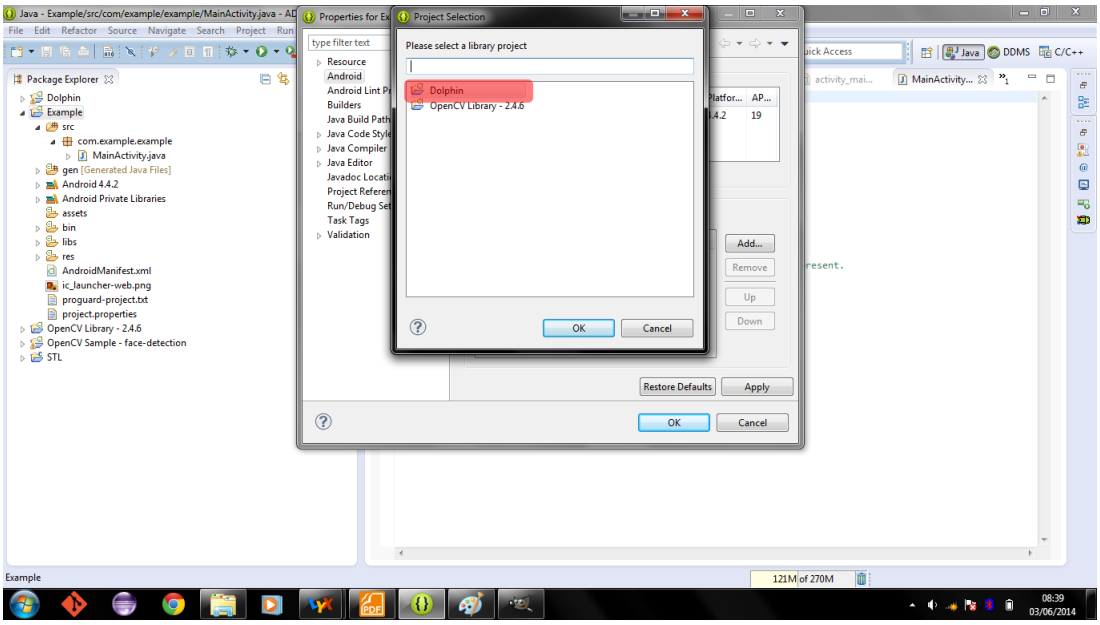


Figura A.6: Importação do framework para o projeto

B CONSTRUINDO APLICAÇÕES COM O FRAMEWORK

Toda aplicação Android basicamente consiste em uma ou mais *Activities*. Uma *Activity* é uma interface visual com o usuário, para um único propósito. Somente uma *Activity* pode ser executada por vez no dispositivo.

Para criarmos uma aplicação de processamento de imagem, precisamos criar uma classe que extenda a classe abstrata *Activity*.

```
public class SimpleActivity extends Activity
{
   @Override
   protected void onCreate(Bundle savedInstanceState)
   {
      super.onCreate(savedInstanceState);
      // elementos da aplicação
   }
}
```

Ao ser iniciada a *Activity* chama o método *onCreate*, neste método serão inicializados todos os elementos básicos da nossa aplicação que são:

- Um *FrameDispatcher* que irá intermediar a comunicação entre a câmera e o processamento de imagem propriamente dito, esta classe representa a classe de aquisição de imagens tal como mencionado no capítulo 3. Ao instanciar um objeto da classe *FrameDispatcher* devemos informar qual o contexto da aplicação, qual a câmera queremos obter os frames bem como as dimensões, vertical e horizontal, do frame.

```
int width = 320;
int height = 240;
int camera_id = 0;
FrameDispatcher frame_dispatcher = new FrameDispatcher(this, camera_id,
width, height);
```

- Um *FrameBuffer* onde irão ser armazenados os frames da câmera. Um *FrameBuffer* implementa a interface *FrameListener* uma vez que ele deve ser preenchido com cada frame produzido pela câmera e também é um *FrameContainer* uma vez que armazena frames para posterior utilização. Uma vez que o classe *FrameBuffer* extende a classe abstrata *Observer* (GANG OF FOUR, 1994), todos os observadores desse objeto serão notificados quando um novo frame chegar ao buffer.

```
FrameBuffer frame_buffer = new FrameBuffer();
```

- Um *FrameProcessorWorker* que representa uma *Thread* que irá trabalhar exclusivamente no processamento das imagens da câmera e por isso implementa a interface *Runnable*. Um *FrameProcessorWorker* trabalha contém uma referência para objetos que implementam a interface *FrameProcessor*, tais objetos atuam como contêineres de cadeias de processamento de imagem. A classe *FrameProcessorWorker* extende a classe *Observable* pois pode ser objeto de observação e implementa as interfaces *FrameContainer*.

```
FrameProcessorWorker frame_processor_worker = new FrameProcessorWorker();
```

- Um *FrameProcessor*, objetos de classes que extendam a interface *FrameProcessor* atuam como cadeias de processamento de imagem, pois neles os filtros de processamento de imagem são organizados com a finalidade de extrair características das

imagens. Neste caso imagine uma classe *SimpleFrameProcessor* que, por exemplo, detecta a pele humana.

```
public class SimpleFrameProcessor implements FrameProcessor
{
  @Override
  protected void doFrameProcessing(byte[] pFrame, int pWidth, int pHeight)
  {
    SkinDetect.skinDetect1(pFrame, pWidth, pHeight);
  }
  @Override
  protected void getDescription()
  {
    return ‘‘Detecção de Pele’’;
  }
}
```

```
FrameProcessor frame_processor = new SimpleFrameProcessor();
```

- Uma *FramePreviewSurface* para apresentar os resultados do processamento de imagem caso haja necessidade de exibí-los. Esta classe extende da classe *GLSurfaceView*, que é uma view disponibilizada pela API do Android para desenho da OpenGL. Para instanciar objetos dessa classe precisamos saber o contexto da aplicação, as dimensões do frame a câmera e o identificador da câmera, esse último necessário para que possamos reajustar o resultado no caso da câmera selecionada ser a frontal, caso contrário terímos uma imagem invertida horizontalmente. Como temos uma superfície de exibição dos resultados do processamento devemos informar que objeto contém o frame que será exibido, nesse caso devemos informar que nossa *FrameSurfaceView* que implementa a interface *Observable*, observa o *FrameProcessorWorker* que extende a classe *FrameContainer*.

```
FramePreviewSurface preview_surface = new FramePreviewSurface(this,
camera_id, width, height);
```

Os passos seguintes consistem em fazer as ligações entre esses objetos.

- Um *FrameDispatcher* tem um *FrameBuffer* associado onde serão armazendos os frames. O método *addFrameListener* faz essa associação.

```
frame_dispatcher.addFrameListener(frame_buffer);
```

- Após isso precisamos informar todos os observadores desse *FrameBuffer,* isso é feito com o método *addObserver*, numa aplicação simples como é o nosso caso o único observador do buffer é o nosso *FrameProcessorWorker*.

```
frame_buffer.addObserver(frame_processor_worker);
```

- Agora precisamos associar ao nosso *FrameProcessorWorker* um *FrameProcessor*, isso é feito através do método *setFrameProcessor*.

```
frame_processor_worker.setFrameProcessor(frame_processor);
```

- Como queremos que os resultados do processamento sejam apresentados precisamos fixar nosso *FramePreviewSurface* a nossa *Activity*, isso é feito através do método *addContentView* que recebe como parâmetros o contexto da *Activity* bem como as dimensões dessa *View*.

```
frame_processor_worker.addObserver(frame_preview);
addContentView(preview_surface, new LayoutParams(LayoutParams.MATCH_PARENT,
LayoutParams.MATCH_PARENT));
// MATCH_PARENT significa que a view ocupará todo o espaço disponível na
// tela do dispositivo.
```

- Precisamos associar nosso *FrameGrabber* a *Activity* em questão caso contrário a câmera não irá iniciar a captura dos frames, isso é feito com o método *reparentTo* que recebe como parâmetro a *Activity* em questão.

```
frame_dispatcher.reparentTo(this);
```

- Por último iniciamos a thread de processamento de imagem.

```
Thread processing_tread = new Thread(frame_processor_worker);
processing_thread.start();
```

A Figura B.1 mostra uma visão geral da nossa aplicação.

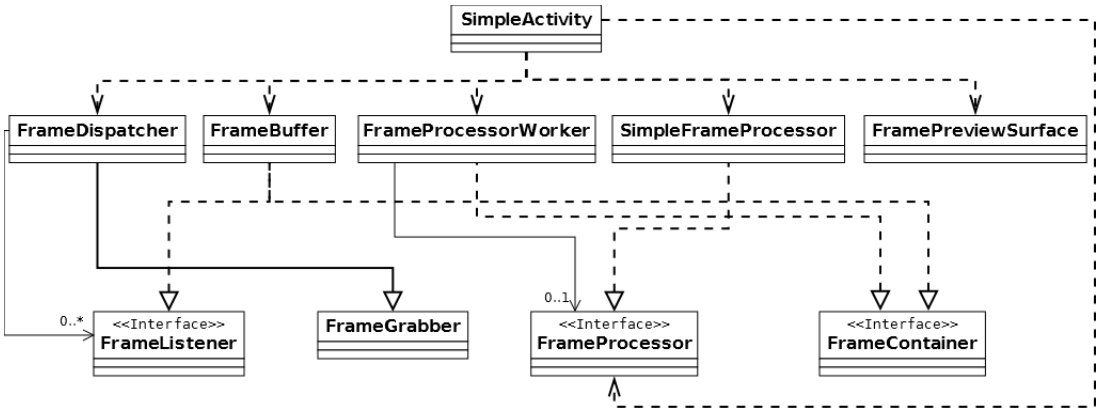


Figura B.1: Diagrama de uma aplicação baseada no Dolphin Framework

A Figura B.2 mostra o código da aplicação descrita acima.

```
public class SimpleActivity extends Activity
{
  @Override
  protected void onCreate(Bundle savedInstanceState)
  {
    super.onCreate(savedInstanceState);
    int width = 320;
    int height = 240;
    int camera_id = 0;
    FrameDispatcher frame_dispatcher = new FrameDispatcher(this, camera_id,
    width, height);
    FrameBuffer frame_buffer = new FrameBuffer();
    FrameProcessorWorker frame_processor_worker = new FrameProcessorWorker();
    FrameProcessor frame_processor = new SimpleFrameProcessor();
    FramePreviewSurface preview_surface = new FramePreviewSurface(this,
    camera_id, width, height);
    frame_dispatcher.addFrameListener(frame_buffer);
    frame_buffer.addObserver(frame_processor_worker);
    frame_processor_worker.setFrameProcessor(frame_processor);
    frame_processor_worker.addObserver(preview_surface);
    addContentView(preview_surface, new LayoutParams(LayoutParams.MATCH_PARENT,
    LayoutParams.MATCH_PARENT));
    frame_dispatcher.reparentTo(this);
    Thread processing_thread = new Thread(frame_processor_worker);
    processing_thread.start();
  }
}
```

Figura B.2: Código da aplicação

C INTEGRAÇÃO COM OPENCV

Uma das características interessantes do framework é a possibilidade de integração com bibliotecas já existentes. Vamos mostrar agora como integrar o framework com a biblioteca OpenCV (OPENCV).

O Dolphin framework trabalha com os frames da câmera em seu formato nativo, um array de bytes (`byte[]`) no formato YCbCr. Para processarmos esse frame (`byte[] frame`) através do OpenCV, devemos transformá-lo numa matriz do tipo `Mat` que é como as imagens são tratadas no OpenCV. Essa transfomração é feita através do seguinte código:

```
Mat opencv_image = new Mat( 1.5 * pHeight, pWidth, CvType.CV_8UC1);
opencv_image.put(0,0, frame);
```

Criamos uma matriz `Mat` com `1.5 * pHeight` linhas e `pWidth` colunas, sendo que cada elemento da matriz tem profundidade de 8 bits e não tem sinal (`CvType.CV_8UC1` equiva a um `unsigned char`) e colocamos o frame da câmera na matriz através do método `put()`, que recebe como parâmetros o frame a ser colocado bem como a posição apartir de onde queremos copiar o frame na matriz, nesse caso, como queremos copiar o frame inteiro colocamos a posição `(0,0)`, que é a primeira posição da matriz.

Podemos converter essa matriz para um outro formato de cor, por exemplo, preto e branco, então vamos guardar a nova imagem na matriz `gray`:

```
Imgproc.cvtColor(opencv_image, gray, Imgproc.COLOR_YUV420sp2GRAY);
```

Agora podemos aplicar qualquer filtro disponibilizado pela biblioteca OpenCV, por exemplo, o filtro de detecção de bordas `Canny`:

```
Imgproc.Canny(gray, gray, 100, 100);
```

Para voltar com a imagem para o padrão do dolphin framework, basta fazermos:

```
Imgproc.cvtColor(gray, opencv_image, Imgproc.COLOR_GRAYsp2YUV420);
opencv_image.get(0, 0, frame);
```

O que fizemos foi converter de volta a imagem em preto e branco para o formato YUV420 (YCbCr) e através do método `get` colocamos a imagemda matriz `opencv_image`

devolta no array de bytes `frame`.

Essa mudança deve ser feita dentro do método `doFrameProcessing` de uma classe que extenda a interface `FrameProcessor`.