TRABALHO DE GRADUAÇÃO

DETECÇÃO DE FADIGA DE CONDUTORES ATRAVÉS DE PROCESSAMENTO DE IMAGEM EM TEMPO REAL EM SISTEMAS EMBARCADOS

VALERY NOBL ROZENTAL

Brasília, Junho de 2009

UNIVERSIDADE DE BRASÍLIA

FACULDADE DE TECNOLOGIA
CURSO DE GRADUAÇÃO EM ENGENHARIA ELÉTRICA

UNIVERSIDADE DE BRASÍLIA
Faculdade de Tecnologia
Curso de Graduação em Engenharia Elétrica

TRABALHO DE GRADUAÇÃO

Detecção de Fadiga de Condutores Através de Processamento de Imagem em Tempo Real em Sistemas Embarcados

Valery Nobl Rozental

Banca Examinadora

Prof. Ícaro dos Santos (Orientador) _____

Profa. Janaína Gonçalves Guimarães _____

Ivar Alves Ferreira _____

Brasília, julho de 2009

FICHA CATALOGRÁFICA

ROZENTAL, VALERY N

Detecção de Fadiga de Condutores Através de Processamento de Imagem em Tempo Real em Sistemas Embarcados

[Distrito Federal] 2009.

RESUMO

Entre as principais causas de acidentes que envolvem motoristas profissionais, tais como os condutores de frotas de caminhões e ônibus interurbanos são sonolência e adormecimento no volante, causados por fadiga dos condutores, devido à carga horária exagerada e trabalho noturno. Em um país de grande extensão territorial, como o Brasil, com o transporte pesado predominantemente rodoviário, a redução de acidentes devidos aos fatores mencionados resultaria em economia significativa de recursos financeiros e salvamento de vidas. Assim, há uma necessidade de um dispositivo portátil e robusto que possa, de forma não invasiva, monitorar o estado de alerta do condutor. Por outro lado, os microprocessadores de arquitetura ARM são conhecidos como sendo eficientes em sistemas embarcados, com o número de tarefas reduzido. Visando o objetivo geral, de criação de dispositivo competitivo no mercado, o objetivo deste trabalho é configurar um sistema embarcado do tipo SBC (*Single Board Computer*) para adquirir imagens de uma câmera USB e realizar processamento de imagem pertinente ao objetivo geral (detecção de face/olho) em tempo real, o que envolve escolha e configuração do sistema operacional, configuração da câmera, escolha e implementação do algoritmo de detecção.

Palavras-chave: Sistemas embarcados, Linux, Máquina Virtual QEMU, Detector Viola-Jones, Classificador Haar Cascade Remasterização do Sistema de Arquivos

ABSTRACT

Some of the main causes of accidents that involve professional drivers, such as truck and interstate bus conductors, are drowsiness and falling asleep “on the wheel”, caused by fatigue due to excessive work hours and night activity. In a country of a great territorial extension, such as Brazil, with predominantly terrestrial heavy transportation, the reduction of accidents from the mentioned factors would result in great economization of financial resources and life saving. Thus, there is a need for a portable and robust device, which could, in a non-invasive form, monitor the alertness state of the driver. Targeting the general goal of creating a market competitive device, the objective of this work is to configure an embedded system of SBC type (*Single Board Computer*) to obtain images from a USB camera and execute a pertinent to the purpose real time image processing (face/eye detection). That involves the choice and configuration of the operational system, camera configuration, choice and implementation of the detection algorithm.

Keywords: Embedded systems, Linux, QEMU Virtual Machine, Viola-Jones Detector, Haar-Cascade Classifier, File System Customization

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE SÍMBOLOS

ADABOOST	*Adaptive Boost*
ARM	*Advanced RISC Machine*
BSD	*Berkeley Software Distribution*
CDROM	*Compact Disk Read Only Memory*
CF	*Compact Flash*
CISC	*Complete Instruction set Computer*
CMOS	*Complementary metal-oxide semiconductor*
CPU	*Central Processing Unit*
FT	Faculdade de Tecnologia
GB	*Giga Bytes*
GRUB	*Grand Unified Bootloader*
GPDS	Grupo de Processamento Digital de Sinais
HD	*Hard Disk*
IDE	*Integral Development Environment*
IPT	*Image Processing Toolbox*
JTAG	*Joint Test Action Group*
LED	*Light Emitting Diode*
MB	*Mega Bytes*
MHz	*Mega Hertz*
MMU	*Memory Management Unit*
OpenCV	*Open Source Computer Vision Library*
PC	*Personal Computer*
RAM	*Random Access Memory*
RecSum	*Rectangular Sum*
RISC	*Reduced Instruction Set Computer*
SBC	*Single Board computer*
SDRAM	Syncronous Dynamic RAM
SDV	*Standard deviation* – Desvio Padrão
UI	Unidade de Intensidade
UnB	Universidade de Brasília
USB	*Universal Serial Bus*
YAFFS	*Yet Another File System*

1 INTRODUÇÃO

1.1 IMPORTÂNCIA DA DETECÇÃO DE FADIGA DE CONDUTORES

Analisando os dados estatísticos, podemos afirmar que sonolência no volante é uma das principais causas de acidentes de trânsito que envolvem motoristas profissionais que costumam trabalhar no horário noturno, como os condutores de caminhões e ônibus interurbanos. As difíceis condições de trabalho, as grandes extensões rodoviárias favorecem a exaustão dos funcionários. Segundo o Prof. Marco Mello do Centro de Estudos Multidisciplinar em Sonolência e Acidentes (CEMSA), cerca de 30% dos acidentes de trânsito são causados por adormecimento no volante e entre 17 e 19% das mortes no trânsito são causados por sonolência ao dirigir. Pela Operação Hypnos – PRF (dados de 2007) foi constatado que 87% dos motoristas trabalham de 12 a 20 horas por dia.

Devemos considerar, também, a grande escala de emprego do transporte rodoviário no país, devido à extensão territorial. Segundo a RNTRC-ANNT, o Brasil possui uma frota de 1.030.445 de caminhões e ônibus interestaduais, cerca de 1.300.00 caminhoneiros, e frota de 100.000 ônibus urbanos. No total são 208 empresas que prestam serviços interestaduais, 1390 empresas de ônibus urbanos e 140.816 empresas e cooperativas no ramo de transportes. A economista e doutora em Engenharia de Transportes, Ieda Lima, apresentou em maio de 2007 um estudo atualizado dos acidentes de trânsito no país envolvendo os caminhões, durante o primeiro painel do VII Seminário Brasileiro do Transporte Rodoviário de Cargas, realizado na Câmara Federal, em Brasília, que mostrou registro de um acidente com veículo de carga a cada cinco minutos no Brasil e um custo total dos acidentes com caminhões no território nacional calculado em R$ 7,7 bilhões. (Fonte: Associação Brasileira de Concessionárias de Rodovias).

No panorama mundial a situação é semelhante. Segundo o estudo dirigido por Dr. Daniel Perez-Chada do hospital Universitário Austral em Buenos Aires, Argentina, a média de sono dos condutores de caminhão é de menos de 4 horas por dia (2005). Nos Estados Unidos, 58% dos acidentes são causados por fadiga do condutor (fonte: FHWA – Federal Highway Administration).

Preocupantes são acidentes que envolvem ônibus, pela grande quantidade de passageiros envolvidos. Estes acidentes resultam em número grande de mortes.

Portanto, fica evidente a importância de redução de quantidade de acidentes que envolvem sonolência no volante. Esta redução resultaria em salvamento de vidas, pouparia sofrimento de muitas famílias e reduziria os gastos de recursos financeiros da sociedade, como internações do SUS, seguro de saúde etc.

1.2 METODOLOGIA

Visando a redução de acidentes, foi proposto o projeto de criação de dispositivo que consiga monitorar o estado de alerta de condutor e, caso seja necessário, enviar um sinal preventivo (toque de alarme, ou até interrupção de injeção de combustível). O dispositivo deve ter alto poder competitivo no mercado, o que implica custo reduzido, robustez e confiabilidade. O tamanho reduzido e a objetividade no sentido de realizar apenas tarefa específica do sistema em questão sugeriram desenvolvimento em um sistema embarcado. Os processadores x86 e processadores ARM encontram-se entre as mais populares arquiteturas de processadores de sistemas embarcados e apresentam um bom desempenho com quantidade de tarefas reduzida. Assim, o desenvolvimento foi feito em cima das plataformas de desenvolvimento Embest SBC2440-I, que emprega microprocessador Samsung S3C2440A de arquitetura ARM920T [1], e PCM-9375 da empresa Advantech, com microprocessador AMD Geode LX800, de arquitetura x86 [2].

A mais óbvia forma não invasiva de monitoramento de estado de alerta é a detecção de estado de olhos, ou seja, se os olhos estão abertos ou fechados. O dispositivo deve adquirir imagens e fazer processamento em tempo real. Devem ser consideradas as condições de funcionamento. Entre elas, a aquisição de imagens com iluminação reduzida, como nas condições noturnas, sem atrapalhar o motorista. O sistema funcionaria, então, a partir de análise de imagem infravermelha, iluminando a face do motorista por LEDs, filtrando os outros componentes do espectro para obter condições de iluminação uniformes de dia e de noite. Portanto, para simular as condições de iluminação infravermelha, o processamento neste trabalho foi feito essencialmente em imagens escala de cinza.

No desenvolvimento do presente trabalho foram pesquisadas e estudadas possíveis soluções para o desafio: sistemas operacionais, bibliotecas de processamento de imagem, algoritmos de detecção de face, as ferramentas auxiliares.

A placa SBC2440-I vem com suporte para dois sistemas operacionais: Linux Yaffs embarcado e Windows CE. Foi decidido usar o sistema operacional Linux Yaffs embarcado. Já para a placa PCM-9375 foram remasterizados os sistemas operacionais DSL 3.0.1 (*Damn Small Linux*) (http://www.damnsmalllinux.org) e Debian Etch (http://ftp.debian.org).

A opção para a ferramenta de processamento de imagem foi a biblioteca OpenCV (Open Computer Vision Library), desenvolvida por Intel e livre para o use comercial sob a licença BSD. OpenCV possui excelente documentação e é desenvolvida em C++, o que permite implementação mais simples em Linux.

Analisando vários algoritmos de detecção de objetos (face e olhos, em particular), observou-se que o algoritmo *Haar-Cascade*, que faz uso da forma de detecção de objetos proposta por Viola & Jones, implementado na biblioteca OpenCV possui excelente desempenho e robustez necessários para o projeto. Alem disso, diferenciando de outros algoritmos disponíveis, ele é implementado em uma imagem de escala de cinza, é quase insensível ao tamanho do objeto, por fazer varredura de imagem em escalas diferentes, e às eventuais variações, como a cor da pele e condições de iluminação, por fazer tratamento de um conjunto de características do tipo Haar.

Um sistema embarcado em geral possui algumas limitações, como o tamanho reduzido da memória, e conjunto reduzido de funções que o sistema operacional oferece. Portanto, a prática comum é criar o código fonte e compilá-lo em uma máquina PC (*host*) de forma cruzada (para arquitetura diferente da arquitetura do host), e levar para o sistema embarcado

apenas o arquivo executável gerado, livrando o sistema da tarefa pesada de compilação. Porém, existem dois problemas nesta abordagem: Ligação de bibliotecas compartilhadas e *debugging*.

Ligação de bibliotecas compartilhadas – Normalmente, um programa de computador depende de várias bibliotecas que devem ser ligadas ao programa principal para o correto funcionamento. Este processo é chamado de ligação (*linking*). Há dois tipos de ligação: a ligação dinâmica (que usa bibliotecas compartilhadas) e a ligação estática. Na ligação estática, o compilador liga as bibliotecas estáticas necessárias na hora de compilação, gerando um arquivo de grande tamanho, porem contendo todo o código binário necessário para a execução. Já na ligação dinâmica, o arquivo executável fica de tamanho reduzido, porém, na hora de execução as bibliotecas necessárias devem ser carregadas na memória, senão, o programa não pode ser executado. Isto significa que as bibliotecas dinâmicas de dependência devem estar presentes no sistema embarcado, para execução de programa compilado dinamicamente. Para arquitetura ARM, sem uma ferramenta auxiliar, o procedimento seria usar o comando `ldd` de Linux para obter os nomes de bibliotecas necessários na máquina host e, já que não são compatíveis com a arquitetura ARM, procurar as versões corretas uma por uma e instalá-las no sistema embarcado. Evidentemente, esta solução não é vantajosa.

Debugging – O dispositivo com sistema operacional Linux embarcado, por ter as funcionalidades reduzidas, muitas vezes não oferece as ferramentas para depuração (*debugging*), presentes em distribuições de Linux comuns, como a função `gdb`.

A solução para estes dois problemas (entende-se principalmente a arquitetura ARM, já que a arquitetura x86 é a mesma para o *host* e o *target* no caso da placa PCM-9375) foi de instalar no PC uma máquina virtual que emule o funcionamento da arquitetura ARM, porém possua as funcionalidades de uma distribuição Linux comum, e não tenha uma restrição tão rígida de espaço de memória. Assim, as bibliotecas dinâmicas nas versões compatíveis podem ser copiadas da máquina virtual, e o *debugging* pode ser feito dentro dela. Ademais, a máquina virtual pode oferecer uma alternativa para a compilação cruzada, realizando a compilação nativa, que gere arquivo executável compatível com o sistema destino.

A máquina virtual de escolha foi QEMU - um software livre escrito por programador francês Fabrice Bellard, que implementa um emulador de processador e permite a funcionalidade completa de um sistema dentro do outro. Ademais, QEMU permite emular sistemas com arquiteturas diferentes, atendendo, assim, as necessidades do desenvolvimento do projeto.

Assim, a metodologia deste trabalho resume-se em seguintes etapas:

- Configuração de sistema operacional Linux embarcado nos dispositivos
- Configuração para captura de vídeo da câmera USB
- Configuração da máquina virtual QEMU
- Configuração de biblioteca OpenCV para compilação cruzada (ligação estática) no PC e nativa (ligação dinâmica) dentro da máquina virtual (Para Arquitetura ARM)
- Implementação do algoritmo *Haar-cascade* para detecção de face/olho nos sistemas embarcados
- Coleta e análise de dados experimentais

2 FUNDAMENTAÇÃO TEÓRICA

2.1 DESENVOLVIMENTO EM SISTEMAS EMBARCADOS

2.1.1 Sistema Embarcado

Um sistema embarcado é todo sistema baseado em microprocessador ou microcontrolador, desenvolvido para aplicação específica. Comumente, os sistemas embarcados são desenvolvidos em placas específicas, com conjunto de periféricos reduzido, o que implica recursos escassos e capacidade de processamento reduzida. [3]

2.1.2 *Host* e *Target*

No ambiente de desenvolvimento diferenciamos o sistema *host* (hospede) e o sistema *target* (destino). Como sugere o nome, o *target* é o sistema que vai eventualmente executar a aplicação. Já que na maioria das vezes, por causa de recursos escassos, o desenvolvimento no *target* não é possível, utiliza-se outro sistema – um computador, para desenvolver o aplicativo que depois será levado como arquivo executável ao *target*. O computador, no qual o desenvolvimento é feito, é chamado de *host*.

2.1.3 Formas de Desenvolvimento de Aplicação

Existem três formas de desenvolver uma aplicação para o sistema embarcado. Se o *host* possuir recursos suficientes de memória e de processamento, as ferramentas de desenvolvimento podem ser instaladas nele, e assim, o desenvolvimento pode ser feito de forma direta. Entre ferramentas de desenvolvimento podemos listar as bibliotecas de suporte para linguagens de desenvolvimento, compilador, possivelmente IDE e ferramentas de *debugging*, o *linker* (programa que faz a ligação das bibliotecas essenciais para a execução de aplicativo). [3]

Caso o *target* não possua recursos suficientes para suportar as ferramentas de desenvolvimento, mas tenha o kernel sistema de arquivos e forma de comunicação (como a porta serial), desenvolvimento é feito no *host*, levando apenas o arquivo executável para *target*. O sistema pode ser controlado e acessado pelo *host*, por meio de programas de comunicação como hyperterminal ou minicom. [3]

No terceiro caso, o *target* não possui sistema operacional. Portanto, todo o desenvolvimento, inclusive do sistema operacional é feito no *host*, e depois levado por mídia removível para *target*. O *target* neste caso possui apenas *bootloader* (programa de inicialização do sistema primário), cuja função é carregar a o sistema operacional da mídia removível. [3]

2.1.4 Remasterização do Sistema Operacional

O termo remasterização refere-se ao "conjunto de procedimentos para alterar uma distribuição Linux" [3].

2.2 ARQUITETURA ARM

ARM (Máquina RISC Avançada), originalmente desenvolvida pela empresa Arcon Computers Ltd, usa, como sugere o nome, arquitetura RISC no núcleo. Arquitetura RISC (Computador de Conjunto Reduzido de Instruções) foi desenvolvida devido ao crescimento exagerado da área de unidade de controle de processadores, já que a execução de instruções complexas, utilizadas na arquitetura CISC (Computador de Conjunto Complexo de Instruções), exigia verificação de dependência entre dados de grande complexidade. A otimização de execução também era feita por unidade de controle. Na arquitetura RISC foi reduzido o conjunto de instruções, deixando apenas aquelas que podem ser executadas em um ciclo de máquina. Já as tarefas de verificação de dependências e otimização foram passadas ao compilador. As instruções mais complexas são tratadas como conjunto de instruções simples. A arquitetura RISC, ademais, reduziu o tamanho de *pipeline*, porém, devido ao tamanho fixo das instruções, permitiu tirar melhor proveito dele. [4] [5]

Visando sistemas embarcados, processadores ARM tem baixo consumo de energia e baixa tensão de funcionamento. São processadores de 32 bits, e, particularmente o processador ARM920T utilizado neste trabalho possui mais um conjunto de instruções de 16 bits denominado *thumb*, que permite melhorar a densidade de código em cerca de 30%. [5]

2.3 O SISTEMA OPERACIONAL

2.3.1 Boot loader

Boot loader é um programa carregado na memória não volátil do sistema, cuja função é inicializar o sistema e carregar o sistema operacional na memória.

2.3.2 Kernel

Kernel é o núcleo de sistema operacional. Ele representa a camada de software mais próxima do hardware. As principais tarefas do kernel são criação, agendamento e finalização de processos, alocação e liberação de memória, controle do sistema de arquivos, operações de entrada e saída com dispositivos periféricos, gerenciamento de programas executados e divisão de tempo de processador entre eles. A figura abaixo (Fig.2.1) mostra uma visão típica de arquitetura de um computador como uma série de camadas abstratas de nível baixo até o nível alto: hardware, firmware, assembler, kernel e o sistema operacional e aplicações. [6]

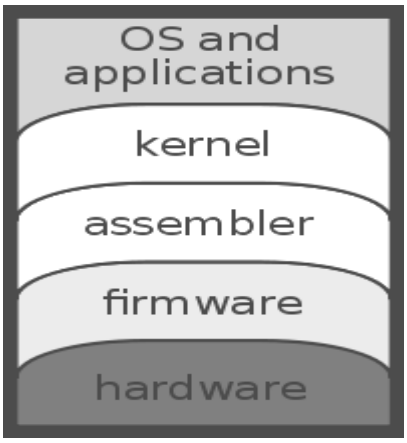


Fig. 2.1 Camadas abstratas de um computador [6]

2.3.3 Sistema de arquivos

Sistema de arquivos é o método de armazenamento e organização de arquivos e dados neles guardados, para facilitar o acesso a eles. A placa SBC2440-I vem com sistema de arquivos YAFFS (*Yet Another File System*), criado pelo neozelandês Charles Manning, e destinado especificamente para memória NAND flash. [7]

2.4 FUNDAMENTOS DE IMAGENS DIGITAIS

Uma imagem monocromática é uma função *f*(*x*, *y*), onde *x* e *y* são coordenadas espaciais e o valor de *f* é proporcional ao brilho (ou níveis de cinza) no ponto (*x*, *y*) considerado. Uma imagem digital é uma imagem *f*(*x*, *y*) discretizada tanto em coordenadas espaciais quanto em brilho (Gonzalez & Woods, 2002 *apud* [8]).

Uma imagem digital pode ser representada por uma matriz, onde o elemento na linha *x* e coluna *y* correspondente ao nível de cinza do ponto (*x*, *y*). Os elementos da matriz são denominados *pixels*, denominação que denota a abreviação do termo *picture elements* (Gonzalez & Woods, 2002 *apud* [8]).

$$f(x,y) = \begin{bmatrix} f(0,0) & f(0,1) & \dots & f(0,M-1) \\ f(1,0) & f(1,1) & \dots & f(1,M-1) \\ \dots & \dots & \dots & \dots \\ f(N-1,0) & f(N-1,1) & \dots & f(N-1,M-1) \end{bmatrix} \quad \text{Equação 1}$$

2.5 ALGORITMO VIOLA – JONES

Em 2004 um artigo escrito por Paul Viola e Michael J. Jones, chamado “Detecção Robusta de Face em Tempo Real” foi publicado na Revista Internacional de Visão

Computacional. O algoritmo apresentado teve tanto sucesso que hoje ele é muito perto de ser o padrão de fato para resolução de tarefas de detecção de face. Este sucesso esta atribuído principalmente a simplicidade relativa, execução rápida e desempenho extraordinário do algoritmo [9].

O algoritmo a princípio restringiu-se a detecção de face frontal não inclinada. Trabalho futuro de Lienhart e Maydt aprimorou-o, adicionando características rotacionais, que permitiram tolerância à inclinação.

O princípio básico do algoritmo Viola – Jones é varrer uma sub-janela capaz de detectar faces através da dada imagem de entrada. A abordagem padrão era de redimensionar a imagem de entrada para tamanhos diferentes e depois varrer as imagens criadas com detector de tamanho fixo. Esta abordagem costuma consumir tempo devido aos cálculos de imagens de diferentes tamanhos. Ao contrário desta abordagem, o algoritmo Viola-Jones redimensiona o próprio detector em vez da imagem, e varre a imagem várias vezes – cada vez com detector de tamanho diferente. Poderíamos à principio pensar que ambas as abordagens consumem tempo igual, porém Viola e Jones desenvolveram um detector invariante de escala que requer o mesmo número de cálculos, independentemente do tamanho. Este detector é construído usando a tal chamada, "imagem integral" e algumas características simples retangulares. [9].

Segundo Viola e Jones, a outra maior contribuição do trabalho deles é o método de combinação sucessiva de classificadores em estrutura cascateada, que aumenta drasticamente a velocidade de detecção focando em regiões promissoras da imagem, já que muitas vezes é possível determinar de forma rápida onde na imagem a ocorrência do objeto de interesse é provável. Assim, o processamento mais complexo é reservado apenas para estas regiões.

2.5.1 Imagem Integral

Imagem integral é uma representação da imagem original, onde o valor em cada ponto (x, y) é dado pela soma dos pixels acima e à esquerda: [10]

$$ii(x, y) = \sum_{x' \leq x, y' \leq y} i(x', y') \qquad \text{Equação 2}$$

A Fig. 2.2 abaixo apresenta este conceito. Supomos que temos uma imagem 3x3, com intensidade de todos os pixels igual a 1 (Fig.2.2(a)). O valor da imagem integral será conforme mostrado na figura 2.2 (b).

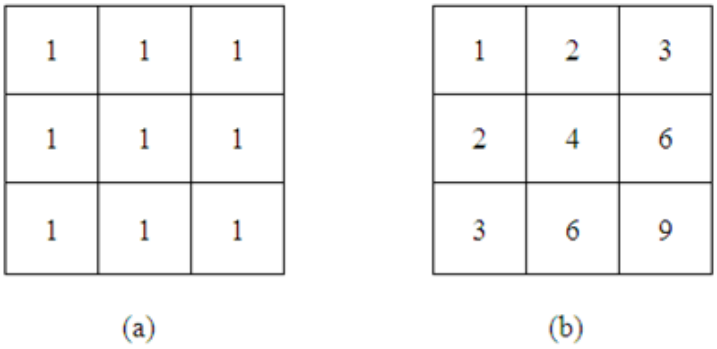


Fig.2.2 Imagem Integral [9]

Assim, o valor para cada pixel pode ser obtido facilmente por uma única varredura da imagem, já que pode ser calculado pelos valores de pixels adjacentes:

$$\boldsymbol{ii(x,y) = ii(x,y-1) + ii(x-1,y) + i(x,y) - ii(x-1,y-1)} \qquad \text{Equação 3}$$

Onde ii representa imagem integral e i o valor da imagem original. Graficamente:

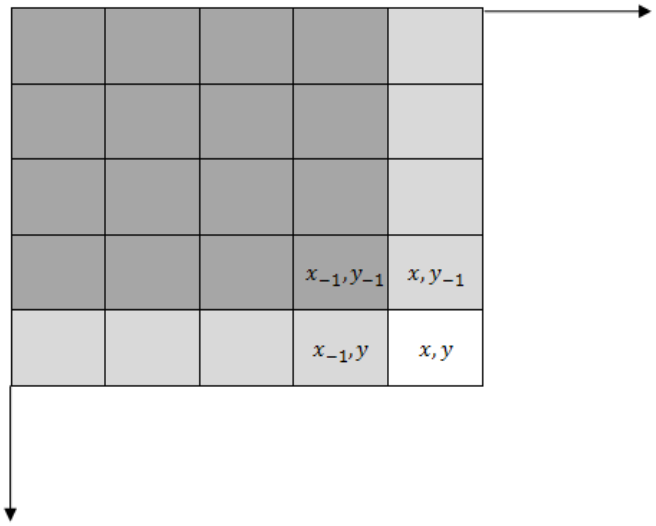


Fig. 2.3 Cálculo de imagem integral

Isto nos permite calcular a soma de pixels em um dado retângulo de tamanho arbitrário usando apenas quatro valores. A figura 2.4 demonstra este conceito: Dados os valores da imagem integral nos pontos A B C e D, a soma do retângulo branco definido pelos pontos A e C é calculada como:

$$\boldsymbol{RectSum(A,C) = ii(C) + ii(A) - ii(B) - ii(D)} \qquad \text{Equação 4}$$

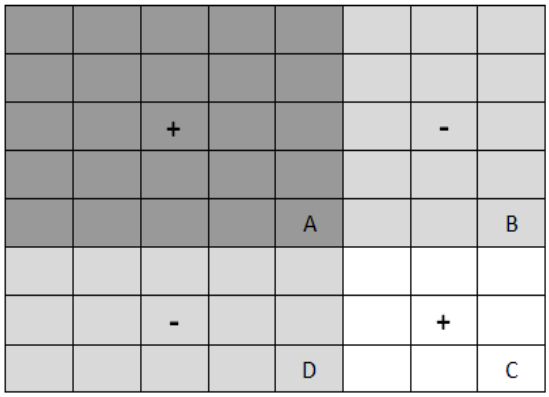


Fig. 2.4 Cálculo de um retângulo

Assim, o calculo da soma de retângulos arbitrários é feita em tempo constante.

2.5.2 Características Retangulares (*Rectangular Features*)

Podemos observar que para subtração de dois retângulos adjacentes, precisamos de seis valores de imagem integral, e de oito valores para três triângulos adjacentes. Quatro características retangulares, denominadas do tipo *Haar* por ter semelhança com *Haar wavelets*, foram apresentadas [11]:

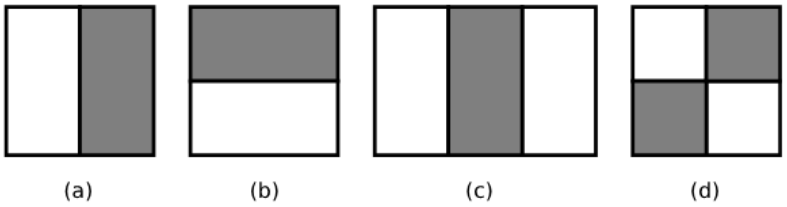


Fig. 2.5 Características tipo Haar [11]

O valor da característica é a diferença entre a soma dos pixels na entre as áreas brancas e escuras.

Viola e Jones tinham encontrado empiricamente que um detector com resolução base de 24x24 pixels retorna resultados satisfatórios. Permitindo todas as possíveis dimensões e posições das características da figura, um total de aproximadamente 160.000 agrupamentos diferentes são construídos. (Para caso de cinco características. Para quatro características o número é 45.396) “Assim, a quantidade de características possíveis ultrapassa por muito os 576 pixels contidos no detector na resolução base. Estas características podem parecer demasiadamente simples para realizar uma tarefa tão avançada como detecção de face, porém, o que lhes falta em complexidade elas ganham na eficiência computacional”. [9]

2.5.3 Algoritmo *AdaBoost* Modificado

Como observamos anteriormente, pelo fato de o número de características associadas com cada detector é muito grande, é pouco vantajoso calcular o conjunto inteiro. A hipótese

proposta por Viola e Jones é que um número pequeno destas características é suficiente para formar um classificador eficiente. Eles usaram uma modificação do algoritmo *AdaBoost* tanto para selecionar as características e treinar o classificador. A idéia do algoritmo é apresentada pelo professor Padhraic Smyth do Departamento de Ciências de Computação da Universidade de Califórnia pelo seguinte procedimento:

- Fazer aprendizagem de uma única característica (em várias escalas e posições).
- Classificar os dados e avaliar onde os erros foram cometidos e escolher a característica que resultou em menor erro. (seria o primeiro classificador fraco)
- Redistribuir pesos para os dados de tal forma que os dados que foram classificados erroneamente recebam pesos maiores.
- Fazer aprendizagem da segunda característica sobre os dados com novos valores de pesos.
- Combinar o primeiro e o segundo classificador e redistribuir pesos segundo os erros resultantes da combinação.
- Repetir o processo T vezes. O classificador "forte" resultante é a combinação dos classificadores 1,2... T "fracos". Estes classificadores são denominados fracos por apresentar taxas de acertos apenas pouco melhores que 50%.

O exemplo abaixo ilustra essa idéia:

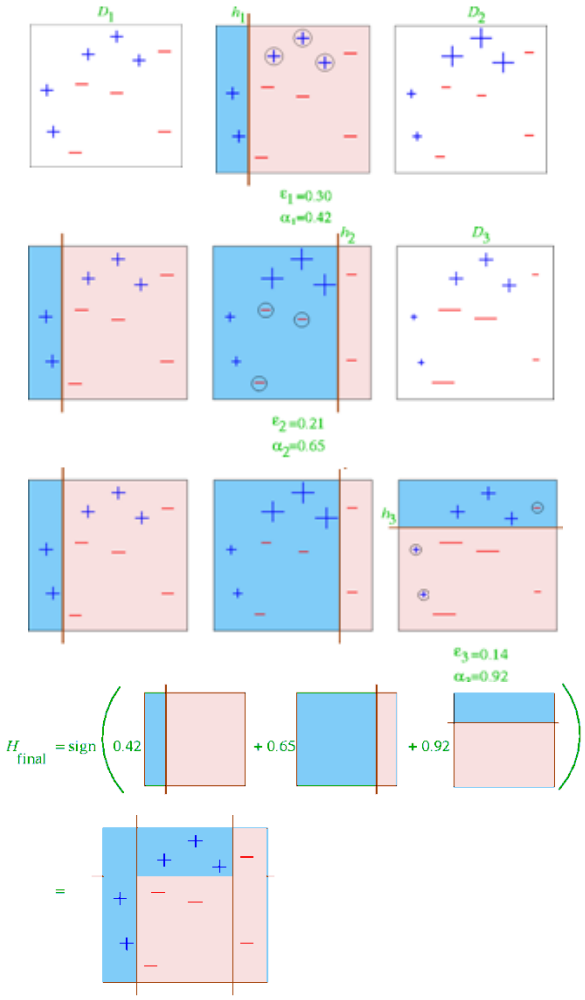


Fig.2.6 Criação do classificador forte pelo algoritmo *AdaBoost* [12]

Na figura 2.6 temos um conjunto de dados representado por dez símbolos, + e -, que podem representar imagens com e sem face, por exemplo. O tamanho de cada símbolo representa o peso atribuído. Inicialmente os pesos são iguais. A primeira série de aprendizagem resultou em característica que conseguiu classificar os dados com 30% de erro. A ela é atribuído símbolo h_1 – o primeiro classificador fraco. Em seguida os pesos são redistribuídos. Na figura observamos isso pelos tamanhos diferentes dos símbolos. Da mesma forma é criado o segundo e o terceiro classificador fraco. O classificador forte H é a combinação de h_1,h_2 e h_3, que consegue classificar os dados sem erro. Em seguida é apresentado o pseudo–código para o Algoritmo *AdaBoost* modificado [10]:

- Dados exemplos de Imagens $(x_1, y_1),..., (x_n, y_n)$ onde $y_i = 0, 1$ para exemplos negativos e positivos respectivamente.
- Inicialize os pesos $w_{1,i.} = 1/2m, 1/2l$ para y_1=0,1 onde m e l são número de exemplos positivos e negativos.
- Para $t = 1 ... T$:
    1. Normalize os pesos,
    $$w_{t,i} \leftarrow \frac{w_{t,i}}{\sum_{j=1}^{n} w_{t,j}}$$
    Onde w_t é distribuição da probabilidade.
    2. Para cada característica j, treine o classificador h_j, restrito a usar apenas uma característica.
    3. O erro é avaliado em relação a w_t, $\epsilon_j = \sum_i w_i \left| h_j(x_i) - y_i \right|$.
    4. Atualize os pesos:
    $$w_{t+1,i} = w_{t,i} \times \beta_t^{1-e_i}$$
    Onde $e_i = 0$ se o exemplo x_i é classificado corretamente, $e_i = 1$ caso contrário e $\beta_t = {\epsilon_t}/{1 - \epsilon_t}$

O classificador final forte é

$$h(x) = \begin{cases} 1 & \sum_{t=1}^{T} \alpha_t h_t(x) \geq 0.5 \sum_{t=1}^{T} \alpha_t \\ 0 & Caso\ contrário \end{cases}$$

e $\alpha_t = \log \frac{1}{\beta_t}$

Na figura abaixo estão mostrados primeira e segunda características escolhidas por *AdaBoost*, sobrepostas em cima da imagem de rosto. A primeira característica mede a diferença em intensidade entre a região de olhos e a região infra-orbital. A segunda característica compara a intensidade da região dos olhos e a região nasal superior.

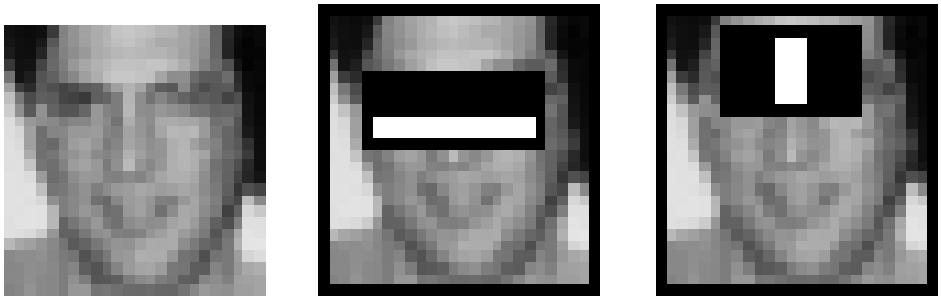


Fig. 2.7 Primeiras duas características encontradas por *AdaBoost* [10]

2.5.4 Classificador em Cascata

O princípio básico deste classificador é construir uma rede que em vez de procurar regiões que contenham face (ou outro objeto de interesse), rejeite em cada etapa as regiões que definitivamente não contém face. Isto resulta com que as regiões sem face sejam descartadas rapidamente. Assim, com o classificador forte do primeiro estágio, que consiste de apenas duas características (como mostrado na Fig. 2.7), através de ajuste do limiar de rejeição é possível obter 100% de detecção de faces, com em torno de 40% de detecções falsas [10]. Estas falsas classificações são descartadas pelos estágios subseqüentes, de forma que o processamento mais complexo seja feito apenas em regiões de interesse. A árvore de decisão pode ser representada pelo seguinte diagrama:

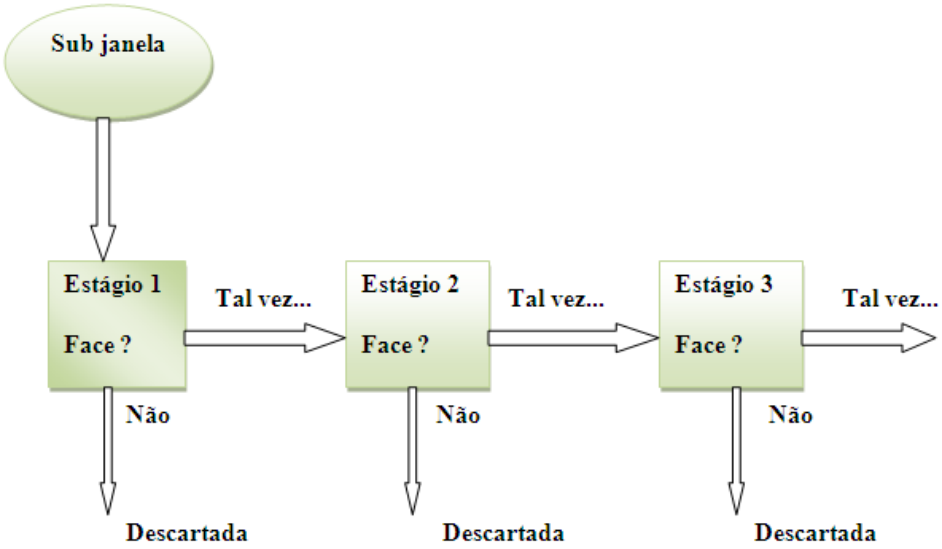


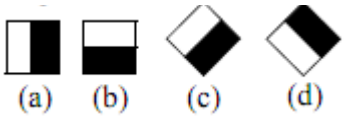
Fig. 2.8 Diagrama da árvore de decisão

Segundo Viola e Jones, aplicação de classificador de duas características resume-se em 60 instruções de microprocessador. "Dificilmente outro filtro simples atinge taxa de rejeição tão alta. Por comparação, escaneamento de um *template* simples ou aplicação de perceptron de camada única precisaria de no mínimo 20 vezes a mais de operações por sub-janela." [10].

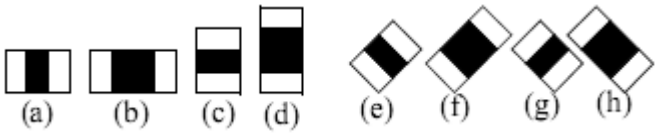
2.5.5 Conjunto Extenso de Características

Rainer Lienhart e Jochen Maydt da Corporação Intel, no artigo *"An Extended Set of Haar-like Features for Rapid Object Detection"*, sugeriram um aprimoramento do algoritmo de Viola e Jones, adicionando características rotacionais de 45°. Foi sugerido conjunto de 15 características, com uma não usada por poder ser aproximada por outras. O conjunto está mostrado a seguir.

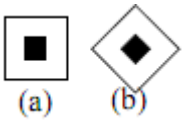
1. Características de borda



2. Características de linha



3. Características do centro rodeado



4. Característica não usada

Fig. 2.9. Características do conjunto extenso [13]

O cálculo da imagem integral é feito da mesma maneira. Por exemplo, Calculando a imagem integral da região vermelha da figura 2.10 obtemos:

$$\boldsymbol{RecSum = rii(x + w, y + y) + rii(x - h, y + h) - rii(x, y) - rii(x + w - h, y + w + h)} \quad \text{Equação 5}$$

onde rii representa imagem integral rotacional. Desta vez foram usadas dimensões w e h da imagem, conforme mostra a figura. 2.10

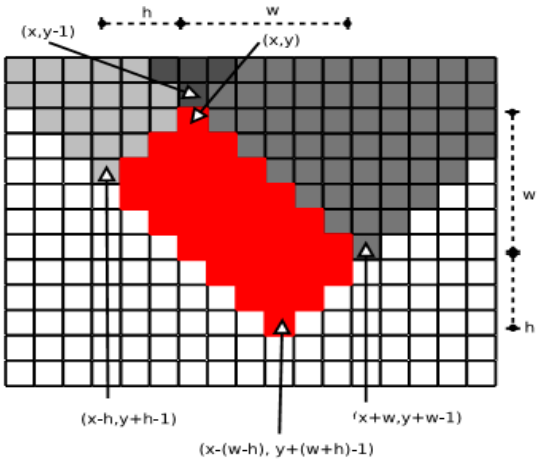


Fig. 2.10 Cálculo da Imagem Integral Rotacional [11]

Autores alegam melhoramento de em torno de 10% no desempenho em relação à classificação falsa, em comparação com algoritmo Viola- Jones, sem grande aumento de esforço computacional.

3 EQUIPAMENTOS UTILIZADOS

3.1 PLACA SBC2440-I

SBC2440-I é um sistema embarcado do tipo Computador de Placa Única (*Single Board Computer*) desenvolvido e manufaturado por Embest Info & Tech Co. LTD, baseado no microprocessador da Samsung S3C2440A. O processador emprega arquitetura ARM920T com unidade de gerenciamento de memória (MMU). Conforme visto na figura 3.1, a placa possui vários módulos e interfaces, sendo que os relevantes a este trabalho são: (1) Porta serial COM0, (2) Porta USB Host, (3) Porta USB para instalação e atualização do sistema, (4) Interface JTAG de 10 pinos.

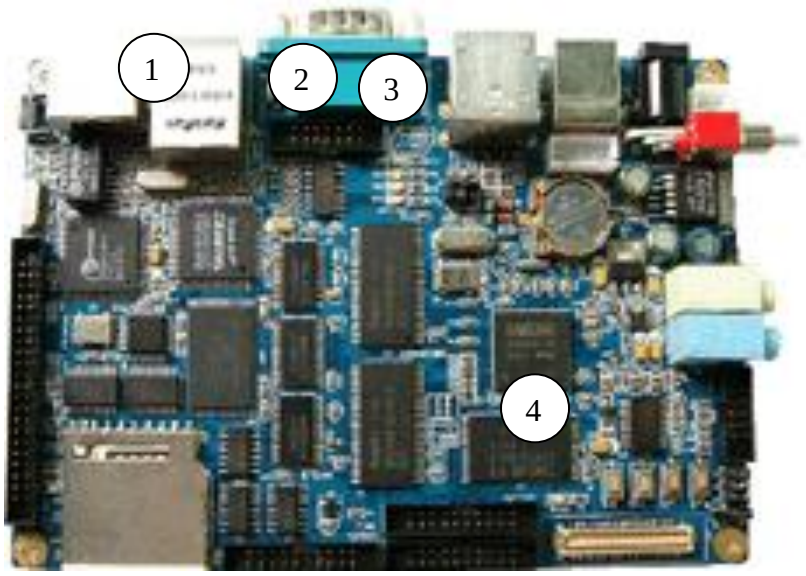


Fig. 3.1 Placa SBC2440-I [1]

A placa possui 1MB de memória Flash do tipo Nor, 64 MB de memória Flash tipo Nand e 64 MB de SDRAM. O processador S3C2440A também possui duas memórias cache de 16 KB cada – cache para instruções e cache para dados. Freqüência máxima do relógio é de 400 MHz. Tensão de funcionamento interna do processador em torno de 1.2 V.

3.2 PLACA PCM-9375

O CPU da placa é o processador AMD Geode LX800 com freqüência máxima de relógio 500 MHz. O sistema usa *chipset* CS5536/CS5535, memória cache de 128 KB no processador, e emprega arquitetura otimizada para memória compartilhada. [2]

Fig. 3.2 Placa PCM 9375 [2]

A placa vem com em um kit montado pelo grupo de processamento de sinais (GPDS) da Universidade de Brasília, e dispõe, alem do dispositivo embarcado, de uma leitora de CD-ROM com conector EIDE-IDE, cartão CompactFlash e fonte de alimentação ATX-300 W.

3.3 CÂMERA DIGITAL MATTEL APB-26065-99A

A câmera digital Mattel Inc. APB-26065-99A usa tecnologia CMOS e gera imagens na resolução de 288x352 pixels, na taxa de 30 frames por segundo.

3.4 COMPUTADOR HOST

O desenvolvimento foi feito no computador PC, CPU- Intel Core Duo, com velocidade de 1.8 GHz e memória RAM de 1.5 GB. O sistema operacional empregado é Linux Ubuntu com kernel 2.6.27-7.

4 PROCEDIMENTOS EXPERIMENTAIS – ELABORAÇÃO DO PROGRAMA

O programa elaborado segue o fluxograma apresentado na Fig 4.1:

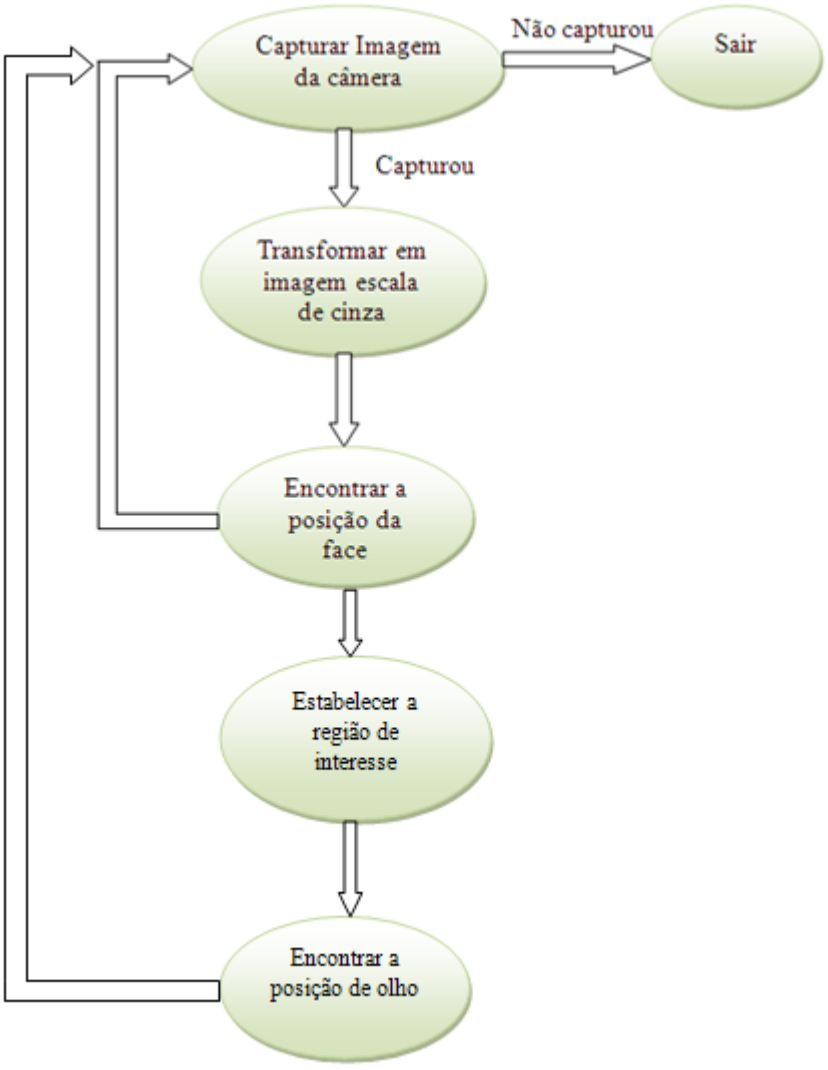


Fig.4.1 Fluxograma do programa

Visando os testes a serem feitos, funcionalidades adicionais foram colocadas:

- limitação do laço de execução para vinte vezes (limitação arbitrária para facilitar coleta e análise de dados);
- medição de tempo de execução;
- mensagens na tela durante execução: o número do frame que está sendo analisado, sucesso ou não na detecção de face e o tempo de processamento;

- medição de tempo médio de execução e a porcentagem de detecção;
- armazenamento dos dados dentro de um arquivo para análise posterior;
- gravação de um frame arbitrário (o vigésimo) num arquivo para confirmação posterior do funcionamento correto do algoritmo.

A distribuição de OpenCV possui arquivos na linguagem XML, com os valores do classificador já treinado para a tarefa de detecção de face e detecção de olhos. Isto simplifica o trabalho tremendamente, tornando desnecessário o processo de treinamento.
O programa final, apresentado no anexo I, é baseado em grande parte no programa exemplo 13.4 da referência (Bradski e Kaehler, 2008, p. 511). A explicação dos comandos internos de OpenCV pode ser encontrada na referência [14].

Ademais, foram feitas tentativas de decisão sobre o estado do olho na imagem final resultante. Uma tentativa envolvia a comparação de desvio padrão (SDV) da imagem de olho aberto e de olho fechado. Neste caso a suposição fundamental é que o desvio padrão da intensidade de *pixels* de um olho aberto é maior do que de um olho fechado. Uma vantagem de uso do desvio padrão é a pouca dependência da luminosidade Ou seja, idealmente, o desvio padrão deve permanecer mesmo para imagens mais claras ou mais escuras (obviamente isso não acontece por causa de saturação). Portanto foram coletadas 100 amostras das magnitudes do desvio padrão para olho aberto e olho fechado. Os resultados estão apresentados na tabela 4.1.

Tabela 4.1 Magnitude do desvio padrão para olho aberto e olho fechado

Olho Aberto		Olho Fechado	
SDV mínimo	8.86 ui	SDV mínimo	7.65 ui
SDV máximo	14.42 ui	SDVmáximo	13.95 ui
SDV médio	11.82 ui	SDV médio	10.98 ui

Embora haja uma leve diferença no desvio padrão entre o olho aberto e o olho fechado, ela não é suficientemente grande para servir de base para desenvolvimento de um algoritmo de decisão.

Outra forma testada foi derivada da propriedade de reflexão dos raios infravermelhos pela córnea do olho. Esta propriedade faz com que o olho iluminado pela luz infravermelha apareça na imagem de uma câmera com um círculo de alta intensidade na região da córnea. Este efeito pode ser observado na Fig.4.2. Uma característica tão marcante na imagem deve diminuir a complexidade computacional. Assim, foi proposta decisão na base de limiar alto. A imagem do olho é transformada em imagem binária com limiar muito alto, que supostamente retornaria imagem totalmente preta, com apenas a região de córnea em branco. Para olho fechado a região inteira estaria preta. Depois é calculada a intensidade média da imagem. Se a intensidade média fosse zero, é plausível supor que o olho está fechado. Caso a intensidade média é diferente de zero, a córnea é detectada – olho aberto. Portanto foi feita uma simulação desta situação a partir de uma imagem. A região da córnea foi pintada em branco, simulando a reflexão. (Fig. 4.3). Evidentemente esta simulação não substitui trabalho com imagem em tempo real que exige escolha cuidadosa do limiar, estudo de vários casos particulares, como

por exemplo, a detecção da reflexão quando o olhar não está direcionado à câmera. Contudo, esta simulação permitiu obter dados sobre a velocidade de processamento.

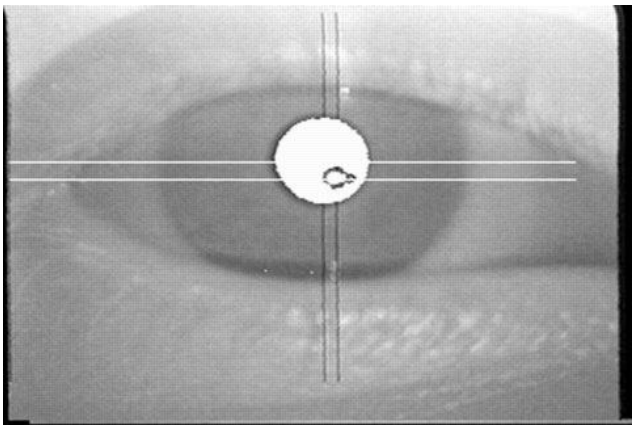


Fig.4.2 Reflexão da córnea na imagem infravermelha [15]

Fig.4.3 Imagem usada na simulação da reflexão da córnea

5 PROCEDIMENTOS EXPERIMENTAIS – PLACA EMBEST 2440-I

5.1 INSTALAÇÃO DE BOOTLOADER

O bootloader que acompanha a placa SBC2440-I é Vivi, desenvolvido pela empresa coreana, MIZI Research, Inc. Para a instalação de *bootloader* foi seguido o procedimento descrito no CD de apoio de faz parte do kit da placa. A instalação foi feita pela interface JTAG na memória NAND flash através da porta paralela do PC.

5.2 INSTALAÇÃO DO KERNEL E SISTEMA DE ARQUIVOS

A placa SBC2440-I vem com suporte para versão 2.6.13 de kernel de Linux. Alem do código fonte, que também pode ser encontrado na internet, existem imagens do sistema operacional prontos para o uso. Como o tamanho da imagem de Linux não é grande, foi decidido usar imagem pronta e não fazer configuração própria do sistema operacional. O carregamento do kernel foi feito pela conexão USB, e controlada pela porta serial, por meio do programa de comunicação DNW que também é fornecida por Embest.

O mesmo procedimento foi usado para carregar o sistema de arquivos. Foi escolhido o sistema de arquivos sem o pacote gráfico QT (de menor tamanho entre as opções), já que o dispositivo deve funcionar de forma *headless*, ou seja, sem conexão a uma tela.

5.3 INSTALAÇÃO DA MÁQUINA VIRTUAL QEMU

O objetivo de instalação da máquina virtual neste caso era principalmente obter uma ferramenta para *debugging*, e também uma maneira alternativa de compilação, que contenha em forma concentrada todas as bibliotecas de dependência para arquivos executáveis. O objetivo não era a simulação do sistema embarcado. Assim, optou se por usar uma distribuição Linux completa, com todas as ferramentas que as distribuições comuns oferecem. Depois de uma extensa pesquisa, optou se por instalar distribuição de Debian, na plataforma do kernel 2.6.18. O procedimento e os arquivos necessários foram oferecidos por Dr. Aurélien Jarno, Engenheiro de Pesquisa de Centro de Pesquisa em Astrofísica de Lyon.

Primeiramente, foi baixada e instalada a versão mais recente do software QEMU na máquina *host*, utilizando os comandos abaixo:

```
sudo wget http://download.savannah.gnu.org/releases/qemu/qemu-0.10.2.tar.gz
sudo apt-get build-dep qemu
tar zxf qemu-0.10.2.tar.gz
```

```
cd qemu -0.10.2
./configure
make
sudo make install
```

Em seguida foi criada uma imagem de HD no formato qcow, que é um dos formatos suportados por QEMU:

```
qemu-img create -f qcow hda.img 10G
```

Foi baixado um kernel pré-configurado por Dr. Jarrno:

```
wget http://people.debian.org/~aurel32/arm-versatile/vmlinuz-2.6.18-6-versatile
wget http://people.debian.org/~aurel32/arm-versatile/initrd.img-2.6.18-6-versatile
wget http://ftp.de.debian.org/debian/dists/etch/main/installer-arm/current/images/rpc/netboot/initrd.gz
```

Vmlinuz é o arquivo executável Linux kernel e *initrd.img* (Initial RAMDISK) é a imagem de sistema de arquivos temporário, usado em processo de boot. A *initrd.gz* é o RAMDISK inicial da destribuição Etch de Debian.

Foi feita a instalação de Debian no arquivo imagem, com parâmetro de memória atribuída de 256MB

```
qemu-system-arm -m 256M -M versatilepb -kernel vmlinuz-2.6.18-6-versatile -initrd initrd.gz -hda hda.img -append "root=/dev/ram"
```

Em seguida é executado o primeiro boot:

```
qemu-system-arm -m 256M -M versatilepb -kernel vmlinuz-2.6.18-6-versatile -initrd initrd.img-2.6.18-6-versatile -hda hda.img -append "root=/dev/sda1"
```

A instalação da imagem do kernel foi feita conforme mostrado a seguir:

```
apt-get install initramfs-tools
wget http://people.debian.org/~aurel32/arm-versatile/linux-image-2.6.18-6-versatile_2.6.18.dfsg.1-23+versatile_arm.deb
su -c "dpkg -i linux-image-2.6.18-6-versatile_2.6.18.dfsg.1-23+versatile_arm.deb"
```

Com isto a instalação de QEMU Debian para ARM é concluída.

5.4 TRANSFERÊNCIA DE ARQUIVOS

Após a instalação, houve ainda a necessidade de transferir arquivos entre a máquina virtual e o sistema de arquivos do sistema principal do *host*. Existem várias maneiras para resolver este problema. Optou-se, neste caso pelo uso da ferramenta *kpartx*. O nome *kpartx* é derivado de *Create Device Maps from Partition Tables*. Ou seja, esta ferramenta lê tabelas de partições de um dispositivo específico e faz mapeamento dos blocos.

Foi criada uma imagem QEMU adicional no formato *raw* de 650MB que serviria de pasta de compartilhamento:

```
qemu-img create hdb.raw 650M
```

A máquina virtual foi iniciada, agregando esta imagem:

```
qemu-system-arm -m 256M -M versatilepb -kernel vmlinuz-2.6.18-6-versatile -initrd initrd.img-2.6.18-6-versatile -hda hda.img -append "root=/dev/sda1" -hdb hdb.raw
```

Dentro de QEMU é aberta a tabela de partição de disco da imagem hdb:

```
cfdisk /dev/sdb
```

Na tabela foi escolhida a partição primaria com opção de escrita. (`new-primary-type(83)-write-quit`)

Nesta partição é criada sistema de arquivos ext3, e em seguida é criado um diretório para compartilhamento de arquivos:

```
mkfs.ext3 /dev/sdb1
mkdir /root/copyFiles
```

Após, foi instalada no *host* a ferramenta *kpartx* é criado diretório de compartilhamento:

```
sudo apt-get install kpartx
mkdir /root/copyFiles
```

Os dois comandos a seguir fizeram com que o `hdb.raw` fosse acessível como bloco, e o mapa dele fosse criado. (Estes comandos devem ser executados toda vez que Linux for reiniciado).

```
losetup /dev/loop0 ~/qemu-0.10.2/hdb.raw
kpartx -av /dev/loop0
```

Cada vez que há necessidade de transferir arquivos, este bloco pode ser montado na pasta de compartilhamento pelo comando:

```
mount /dev/mapper/loop0p1 /root/copyFiles
```

No QEMU a montagem é feita de forma direta:

```
mount /dev/sdb1 /root/copyFiles
```

Há de observar que a biblioteca de compartilhamento não pode ser montada no host é no QEMU simultaneamente, assim, ela é desmontada cada vez que acesso for concluído, para poder ser montada no outro sistema, seja *host* ou QEMU.

5.5 INSTALAÇÃO DAS FERRAMENTAS DE DESENVOLVIMENTO E OPENCV

Nesta fase, as ferramentas se desenvolvimento e OpenCV foram instaladas na máquina virtual QEMU, e paralelamente na máquina *host* para compilação cruzada. Na máquina virtual, sendo ela executando uma distribuição Debian, foi usada a ferramenta APT (*Advanced Packaging Tool*) para instalar o compilador e os pacotes necessários automaticamente de forma nativa com o comando

```
apt-get install build-essential
```

OpenCV também foi instalado de forma nativa:

```
wget    http://ufpr.dl.sourceforge.net/sourceforge/opencvlibrary/opencv-
1.1pre1.tar.gz
tar zxf opencv-1.1pre1.tar.gz
cd opencv-1.1
./configure CXXFLAGS=-fsigned-char
make
make install
```

Sem uma especificação, esta configuração instala as bibliotecas dinâmicas de OpenCV. Uma opção que também foi usada é habilitar *debugging* passando argumento `--enable-debugging` para o comando de configuração. As variáveis do tipo *char* devem estar com sinal habilitado, segundo a sugestão do site de OpenCV para funcionar com arquitetura ARM. [15]

A compatibilidade de arquivos compilados dentro da máquina virtual com a placa foi testada por meio de um arquivo "*Hello World*" escrito em C++ e compilado de forma nativa na máquina virtual. O arquivo executável gerado foi então transferido primeiramente para o *host* pelo procedimento descrito em 3.4.4, e depois para a placa, com uso de um pendrive. O programa foi executado com sucesso na placa, confirmando, assim, a compatibilidade do compilador.

As ferramentas para a compilação cruzada são fornecidas no CD do suporte da placa SBC2440-I. Este CD contém a versão 3.4.1 do compilador *gcc* (*GNU C Compiler*) para ARM e as bibliotecas de dependência em arquivo comprimido. (*.tar.gz*) O comando abaixo extrai as bibliotecas e instala segundo o caminho especificado automaticamente a partir do diretório *root*:

```
tar xvzf arm-linux-gcc-3.4.1.tgz –C /
```

O caminho especificado dos arquivos das ferramentas de compilação cruzada é `/usr/local/arm/3.4.1`. Para usar o compilador, este caminho deve ser adicionado à variável PATH do sistema. De forma alternativa, o conteúdo do diretório

`/usr/local/arm/3.4.1/bin` que contém os arquivos binários do compilador foi copiado para um dos diretórios já especificado na variável PATH - `/usr/local/sbin/`.

A biblioteca OpenCV foi compilada de forma cruzada com os comandos abaixo:

```
./configure --host=i686-pc-linux-gnu --target=arm-linux CC=arm-linux-gcc CXX=arm-linux-g++ LD=arm-linux-ld CPP=arm-linux-cpp CXXCPP=arm-linux-cpp AR=arm-linux-ar RANLIB=arm-linux-ranlib NM=arm-linux-nm STRIP=arm-linux-strip AS=arm-linux-as --prefix=/usr/local/arm/ --exec-prefix=/usr/local/arm/ --enable-static --without-imageio --without-carbon --without-quicktime --without-python --without-gtk --without-swig --disable-apps CXXFLAGS="-fsigned-char -O0 -pipe"
make
make install
```

O comando especifica quais compiladores devem ser usados para criar as bibliotecas de OpenCV, especifica os diretórios para estas bibliotecas por meio da variável `prefix` (no nosso caso `/usr/local/arm/`), desabilita opções desnecessárias para o aplicativo, habilita sinal de variáveis *char* e não permite otimização do código por meio de especificação `-O0`. Outras opções de otimização neste caso resultam em falhas de segmentação do programa. [16] Depois, o comando `make` cria os arquivos e as bibliotecas segundo especificado na configuração. O último comando coloca estes arquivos nos diretórios especificados pela variável `prefix`. Neste caso a opção de criação de arquivos estáticos foi criada, adicionando `--enable-static`. Com isso, são criadas bibliotecas do tipo .a (*libcv.a* e *libhighgui.a*), utilizadas na compilação estática de programas que utilizam OpenCV.

O compilador foi testado com um arquivo "*Hello World*" compilado de forma cruzada. O arquivo executável foi copiado para a placa e executado com sucesso.

5.6 INSTALAÇÃO DA CÂMERA

O kernel da placa possui um driver compatível com as câmeras baseadas em ov511, um chip da câmera/USB produzido por *Omnivision Technologies*. A câmera é reconhecida, então, ao ser conectada na porta USB. Porém, para poder fazer leitura correta da imagem houve necessidade de mudar as permissões do arquivo `video0`, no qual a câmera é montada:

```
chmod a=rw /dev/v4l/video0
```

Este comando atribui permissão de leitura para todo usuário. Provável explicação para esta restrição é que o sistema operacional vê em câmera um arquivo que tenta modificar o sistema e não possui permissão para isso. Infelizmente, o modo retorna-se ao original quando a placa é reiniciada. Assim, para o funcionamento correto há necessidade de atribuição do modo de leitura por meio da remasterização do kernel. Ademais, foi necessário criar duas ligações dinâmicas para o arquivo `/dev/v4l/video0`:

```
ln -s /dev/v4l/video0 /dev/video
ln -s /dev/v4l/video0 /dev/video0
```

Com estas mudanças foi possível gravar imagens da câmera. Isto foi confirmado por meio do programa-teste que grava um frame da câmera. Confirmando o mencionado em [16],

os arquivos no formato *jpeg* foram corrompidos, fazendo com que a imagem teste fosse gravada em arquivo *bmp* que não usa compressão.

5.7 EXECUÇÃO DO PROGRAMA E *DEBUGGING*

5.7.1 Compilação do programa

Primeiramente o programa foi compilado de forma dinâmica dentro da máquina virtual QEMU, levando para a placa apenas o arquivo executável:

```
gcc -I/usr/local/include/opencv/ -L/usr/local/lib/ -lhighgui -lcv -o <nome>
<nome.cpp> -O0
```

Houve necessidade de instalar manualmente um conjunto de bibliotecas de dependência dinâmicas na placa, que foram identificadas com o comando `ldd` executado dentro da máquina virtual, e depois copiados para o diretório `/usr/lib` da placa. A lista dessas bibliotecas está apresentada no anexo II. O sistema de arquivos YAFFS, utilizado na placa não possui este diretório e por padrão instala todas as bibliotecas dinâmicas no diretório `/lib`. Contudo, as bibliotecas de dependência foram instaladas no diretório diferente por causa de incompatibilidade de versões de algumas delas, entre aquelas já presentes no sistema, e outras copiadas da máquina virtual. Simples substituição por bibliotecas da versão mais nova resultava em travamento do sistema operacional. Assim, as manipulações resultaram em dois arquivos com mesmo nome - `ld-linux.so`, chamados de *loaders*, cuja responsabilidade é carregar o conjunto das bibliotecas dinâmicas: `/lib/ld-linux.so.2` (que carrega as bibliotecas do sistema) e `/usr/lib/ld-linux.so.2` (que carrega as bibliotecas para o executável do programa). O programa é invocado, então como argumento do *loader*:

```
/usr/lib/ld-linux.so.2 /<PATH>/<nome_do_arquivo>
```

Esta forma, embora pouco confusa, permitiu o carregamento correto do programa sem influenciar no funcionamento correto do sistema operacional.

Outra opção testada foi a de compilação cruzada estática no *host*:

```
arm-linux-gcc –static -O0 –I /usr/local/arm/include/opencv/ -L
/usr/local/arm/lib/ -lhighgui -lcv -o <nome> <nome.cpp>
```

A compilação resultou em arquivo executável sem dependências não resolvidas.

5.7.2 Execução e Debugging

A execução do programa em ambos os casos de compilação resultou em falha de segmentação (*Segmentation Fault*). Após análise verificou-se que o programa ao princípio está sendo executado, e a falha resulta ao tentar executar uma função interna de OpenCV – `cvHaarClassifierCascade`. Hipótese de falta de memória RAM foi descartada executando o programa (modificando-o para ler imagem de um arquivo e não do frame da câmera) dentro

da máquina virtual que foi definida a emular sistema com 256MB de RAM. A execução retornou o mesmo erro. OpenCV e o programa foram então recompilados, habilitando *debugging* com `--enable-debug` na configuração de OpenCV e `-g` na compilação do programa.

Execução do programa no QEMU com ferramenta de *debugging* – `gdb` retornou o seguinte aviso:

```
cvhar.cpp: 507 void cvSetImagesForHaarClassifierCascade (
CvHaarClassifierCascade* , const CvArr*, const CvArr*, const CvArr*,
double):
Assertion '(unsigned)((equ_rect.y) < (unsigned)((*sum)).rows &&
(unsigned)((equ_rect.x) < (unsigned)((*sum)).rols' failed
```

Aparentemente, o tamanho da janela do classificador é maior que a própria imagem.

Como a função não possui *bugs* no código fonte já que é executada com sucesso no *host*, foi concluído que o erro deve ser causado por peculiaridades da arquitetura ARM na alocação de memória. Apesar de alguns relatos na web de execuções bem sucedidas de programas que utilizam a mesma função em processadores de arquitetura ARM, o problema não foi resolvido. Seguindo sugestões de [16], foram testados compiladores de versões alternativas, 3.3.4 e 3.4.4, tampouco resultantes em execução correta.

Os resultados da execução indicaram duas formas de prosseguir com o projeto na plataforma SBC2440-I:
- Fazer análise de nível baixo de código de OpenCV (possivelmente transformando-o em assembler para ARM e analisando a alocação de memória) na tentativa de corrigir o erro.
- Desistir de uso de classificador *Haar Cascade* e/ou OpenCV, criando alternativa para o algoritmo de detecção de face.

Ambas as maneiras de prosseguir deveriam resultar em um grande esforço e investimento de tempo. Portanto, foi decidido suspender o desenvolvimento nesta placa e continuar o trabalho na placa PCM-9375

6 PROCEDIMENTOS EXPERIMENTAIS – PLACA PCM-9375

A placa PCM-9375 não possui memória flash, sendo necessária a instalação do sistema operacional em um dispositivo externo suportado pela placa: pendrive, cartão CF (*Compact Flash*) ou CDROM. Uso de CDROM no projeto é, obviamente, inviável por motivos de tamanho. Portanto, foi testado o desenvolvimento no cartão CF. Esta configuração, embora exija uso de dispositivo adicional, traz a vantagem de possibilidade de adicionar funcionalidades ao kernel do sistema operacional, incluindo as ferramentas de desenvolvimento, devido à disponibilidade de memória. É claro que a placa deve ter capacidade de processamento suficiente para utilizar estas ferramentas, como no caso de PCM-9375.

Nesta fase foram testados dois sistemas operacionais – DSL e Debian. Cada sistema possui vantagens e desvantagens discutidas posteriormente. Os procedimentos de instalação e remasterização de DSL seguem a orientação da apostila "*Linux e Sistemas Embarcados*" escrita pelo Prof. Geovany A. Borges [3], já a instalação da distribuição Debian é baseada no tutorial "*Building a Real-Time Debian Destribution for Embedded Systems*". [17]

6.1 INSTALAÇÃO DA DISTRIBUIÇÃO DSL

A DSL é uma distribuição não comercial, baseada no Debian e sistema de arquivos Knoppix, com suporte gráfico já instalado na imagem do kernel. O kernel usado em DSL é 2.4. De forma nativa a distribuição não permite desenvolvimento no *target*, sendo que uma remasterização possa adicionar as ferramentas necessárias. Nesta parte é descrita a forma de remasterização e instalação de uma distribuição comum [3], e em seguida são detalhadas as modificações feitas no procedimento particular neste trabalho.

6.1.1 Remasterização de uma Distribuição DSL

A distribuição DSL primeiramente é baixada do site http://damnsmalllinux.org
Depois são criados diretórios auxiliares:

```
mkdir remaster
cd remaster
mkdir –p image
mkdir –p master
```

Isto resulta em criação de três diretórios sob o diretório `remaster`:

`image` – diretório que conterá a distribuição original
`master` – diretório para a distribuição nova

`source/KNOPPIX` – diretório para o código fonte que possa ser alterado no futuro

O conteúdo da distribuição original é copiado, então, para o diretório `master`, e sistema de arquivos raiz é descomprimido para um arquivo .iso:

```
extract_compressed_fs master/KNOPPIX/KNOPPIX >tmp.iso
```

As instruções abaixo montam o arquivo `tmp.iso` no dir. `image`, o copiam para dir. source/KNOPPIX, e em seguida desmontam o dir. `image`:

```
mount –o loop tmp.iso image
rsync –Hav image/. source/KNOPPIX
umount image
```

Depois é preciso entrar no novo sistema de arquivos e montar o arquivo de processos:

```
echo 0 > /proc/sys/vm/vdso_enabled
chroot sourceValery2/KNOPPIX
mount –t proc /proc proc
```

A partir deste momento é possível adicionar ou remover pacotes usando APT. De forma alternativa, os pacotes podem ser baixados manualmente com comando `wget`, e depois instalados. Após terminar a remasterização, é preciso desmontar o arquivo de processos e sair do novo sistema de arquivos:

```
umount /proc
exit
```

O procedimento descrito acima não foi executado neste trabalho a partir da distribuição comercial por causa de dificuldade de instalação dos pacotes necessários com APT. Isto se deve à renovação dos pacotes para versões mais novas do kernel (2.6). Assim, não foi possível encontrar os pacotes da árvore de dependência criada por APT. Portanto, foi remasterizada uma distribuição já alterada previamente, que continha as ferramentas de desenvolvimento (cortesia de Tiago Alves da Fonseca / GPDS FT UnB), baseada na versão DSL-3.0.1. Foi adicionada, então, a biblioteca OpenCV e os arquivos fonte do programa.

Em seguida é criado o novo sistema de arquivos com comando `mkisofs`:

```
mkisofs -iso-level 4  -R -U -V "KNOPPIX.net filesystem" -publisher "KNOPPIX
www.knoppix.net" -hide-rr-moved -cache-inodes -no-bak -pad
sourceValery2/KNOPPIX | nice -5 create_compressed_fs - 65536 >
masterValery2/KNOPPIX/KNOPPIX
```

6.1.2 Gravação do Sistema Operacional no CF

Para gravar o sistema operacional, o CF devia ser preparado pelo procedimento apresentado em [3]. Para isto foi necessário saber qual arquivo do sistema de dispositivos corresponde ao CF. Isto foi feito com comando `mount` sem argumento (outra alternativa é usar o comando `dmesg`.) Neste trabalho o arquivo correspondente é sdd1. Primeiramente o CF foi formatado com o sistema FAT16 com o comando `cfdisk` e opção 06 na tela de escolha do tipo. Depois o dispositivo foi definido como dispositivo de *boot*:

```
mkdosfs /dev/sdd1
```

Logo, a nova imagem foi gravada no CF.

```
mount -t vfat /dev/sdd1 /media/memory
cp -R master/* /media/memory
syslinux -s /dev/sdd1
umount /media/memory
```

6.1.3 Instalação do GRUB

Syslinux é arquivo bootloader que deveria inicializar o sistema. Contudo, o sistema não foi inicializado corretamente pelo *syslinux*, tornando necessária a instalação de GRUB (*Grand Unified Bootloader*). O procedimento é descrito a seguir:

Primeiramente o cartão CF foi montado como tipo FAT16:

```
 mount -t vfat /dev/sdd1 /media/memory
```

Em seguida o GRUB foi instalado:

```
grub-install  --root-directory=/media/memory /dev/sdd
```

Caso o comando retornar

```
"/dev/sdd does not have any corresponding BIOS drive",
```

o seguinte comando deve ser executado para fazer checagem do mapa do dispositivo mesmo se ele existir [18] :

```
grub-install --recheck /dev/sdd –root-directory/media/memory
```

No arquivo `/media/memory/boot/grub/device.map` apenas o `hda` foi habilitado. Logo, foi adicionado o arquivo `menu.lst`, o arquivo de configuração do GRUB, ao diretório `/media/memory/boot/grub/`, deixando as mesmas opções especificadas no *syslinux*:

```
title Valery
root (hd0,0)
kernel /linux24 root=/dev/hda1 ramdisk_size=100000 init=/etc/init
lang=us apm=power-off vga=791 nomce noapic qemu quiet
BOOT_IMAGE=knoppix frugal
initrd /minirt24.gz
makeactive
boot
```

Com isto a instalação do GRUB foi concluída.

6.2 INSTALAÇÃO DA DISTRIBUIÇÃO DEBIAN

6.2.1 Instalação do Sistema Operacional

Foi criado diretório "`embedded`" para a instalação de Debian. Em seguida a distribuição Debian Etch para arquitetura x86 foi baixada e instalada no novo diretório, usando o pacote *debootstrap*:

```
Debootstrap --arch i386 etch embedded http://ftp.debian.org
```

Em seguida foi montado o sistema de arquivos de processos no diretório `embedded/proc`, o sistema de arquivos de dispositivos em `embedded/dev` e o diretório raiz foi mudado para o diretório de Debian:

```
mount -t proc proc embedded/proc/
mount -o bind /dev/ embedded/dev/
LC_ALL=C chroot embedded /bin/bash
```

Logo, foram instalados os pacotes auxiliares e de desenvolvimento, usando a ferramenta APT:

```
apt-get -f install
apt-get install ssh linux-source-2.6.24 gcc make libncurses-dev grub
udev rsync
apt-get install pciutils less build-essential kernel-package ncurses-
dev
apt-get install ifupdown ifrename initramfs-tools bzip2
```

Nesta fase o procedimento descrito em [17] foi simplificado com instalação do kernel já existente em pacotes de APT, sem configuração adicional:

```
apt-get install linux-image-2.6.18-6-486 linux-headers-2.6.18-6-4
```

6.2.2 Modificação do Sistema Operacional

Santana *et al* destacam que sistemas operacionais executados a partir de dispositivo *flash* devem ter quantidade de acessos de escrita minimizada, no intuito de aumentar a vida útil do dispositivo. Com este objetivo, eles apresentam uma série de procedimentos, seguidos neste trabalho, e apresentados a seguir:

Remoção de `/etc/mtab` e ligação simbólica dele para `/proc/mounts`, que também descreve o sistema de arquivos montado, porém não precisa de acesso de escrita:

```
rm /etc/mtab
ln -s /proc/mounts /etc/mtab
```

Remoção do arquivo de escrita `/etc/resolv.conf`, com criação de ligação dinâmica:

```
mv /etc/resolv.conf /var/log
ln -s /var/log/resolv.conf /etc/resolv.conf
```

Mudança em alguns arquivos de configuração:

- /etc/fstab:

```
/dev/hda1 ext2 defaults,noatime 0 0
proc /proc proc defaults 0 0
tmpfs /var/run tmpfs defaults 0 0
tmpfs /var/lock tmpfs defaults 0 0
tmpfs /var/log tmpfs defaults 0 0
tmpfs /tmp tmpfs defaults 0 0
tmpfs /var/lib/dhcp3 tmpfs defaults 0 0
```

- /sbin/dhclient-script:

Ajuste de `new_resolv_conf` para `"tmp/resolv.conf.dhclient-new"`
Mudança de `"mv -f $new_resolv_conf /etc/resolv.conf"` para `"cat $new_resolv_conf > /etc/resolv.conf"`

- /etc/syslog.conf:

Exclusão das linhas que contém `/dev/xconsole`

- etc/init.d/bootclean:

Adição de duas linhas após `[ -f /tmp/.clean]&&...`

```
touch /var/log/resolv.conf
touch /var/log/dmesg
```

Alem disso adicionado usuário com comando `adduser` e criada senha:

```
adduser <user_name>
passwd
```

Ademais, foi instalada a biblioteca OpenCV.

6.2.3 Preparação do Cartão Flash

Sendo Debian uma distribuição de sistema de arquivos mais complexa que o Knoppix, foi necessário usar o formato Ext2 e não FAT16, para conseguir preservar os links simbólicos e as permissões. Portanto, foi criada no CF uma partição com *boot* habilitado (usando o comando `cfdisk`), que depois foi formatada como Ext2, desabilitando a checagem do disco:

```
mkfs.ext2 /dev/sdd1
tune2fs -c 0 -i 0 /dev/sdd1
```

O CF depois foi montado no diretório `/mnt/cf`:

```
mkdir /mnt/cf
mount /dev/sdd1 /mnt/cf
```

Logo, o conteúdo da pasta `embedded` que contem o sistema de arquivos de Debian foi copiado para o diretório de montagem do CF, excluindo o diretório `/proc`, criado manualmente depois. Isto é feito para não copiar os processos da máquina *host*:

```
mv embedded/proc /tmp/.
cp -a embedded/* /mnt/cf/
mv proc embedded/proc
mkdir /mnt/cf/proc
```

Sendo `proc` uma partição, quando criada em vazio, ela está pronta para ser montada na inicialização do Debian, e receber as entradas dos processos executados.

6.2.4 Instalação do GRUB

A instalação do GRUB no caso de Debian difere ligeiramente da instalação anterior, descrita na seção 6.1.3 e, portanto, é apresentada aqui. Erro na inicialização do sistema pelo GRUB tornou necessário o uso de identificador UUID (*Universally Unique Identifier*), para que o sistema possa encontrar sem erro a partição habilitada para *boot* do CF.

Primeiramente, o CF foi identificado pelo UUID:

```
ls -l /dev/disk/by-uuid
```

Depois, como no caso anterior, foi feita a instalação do GRUB no CF:

```
grub-install --root-directory=/mnt/cf /dev/sdd
```

O arquivo `menu.lst` foi criado, usando o número identificador de UUID:

```
title Valery
        uuid     3b5de392-18e9-4643-8f3a-6a5ec44ca57a
        kernel   /boot/vmlinuz-2.6.18-6-486
        root=UUID=3b5de392-18e9-4643-8f3a-6a5ec44ca57a
        initrd   /boot/initrd.img-2.6.18-6-486
        makeactive
        boot
```

7 RESULTADOS EXPERIMENTAIS

Feitos os procedimentos da seção 6, o programa funcionou sob ambas as distribuições de Linux na placa PCM-9375. Em seguida foram coletados, usando a distribuição Debian, os seguintes dados:

- tempo de execução (apenas captura do frame e o processamento);
- porcentagem de acertos.

7.1 DETECÇÃO DE FACE

Houve necessidade de conhecer o desempenho do detector de face, já que este detector serve como base para o algoritmo todo. Portanto, o algoritmo foi testado em duas etapas. Primeiramente foi testado o desempenho apenas na detecção de rosto. Posteriormente foi testado o programa completo, i.e. detecção de rosto e olho. A parte de detecção de rosto foi testada em cinco pessoas, com diferentes características físicas, observadas nas imagens resultantes na Fig. 7.1. Foi feita captura de 20 frames para cada teste, na distância de aproximadamente 1 metro da câmera, permitindo movimentação leve, com condição de permanecer de frente à câmera.

Fig. 7.1 Imagens Resultantes da Detecção de Face

A tabela 7.1 faz resumo de dados estatísticos de processamento:

Tabela 7.1 Dados estatísticos do processamento (Detecção de face) – Debian

Tempo máximo	1260 ms
Tempo mínimo	1060 ms
Tempo médio	1123 ms
Desvio Padrão	31.6 ms

A figura 7.2 mostra o histograma do tempo de processamento, gerado no Matlab. Podemos observar a maior taxa de ocorrência entre 1120 e 1140 ms.

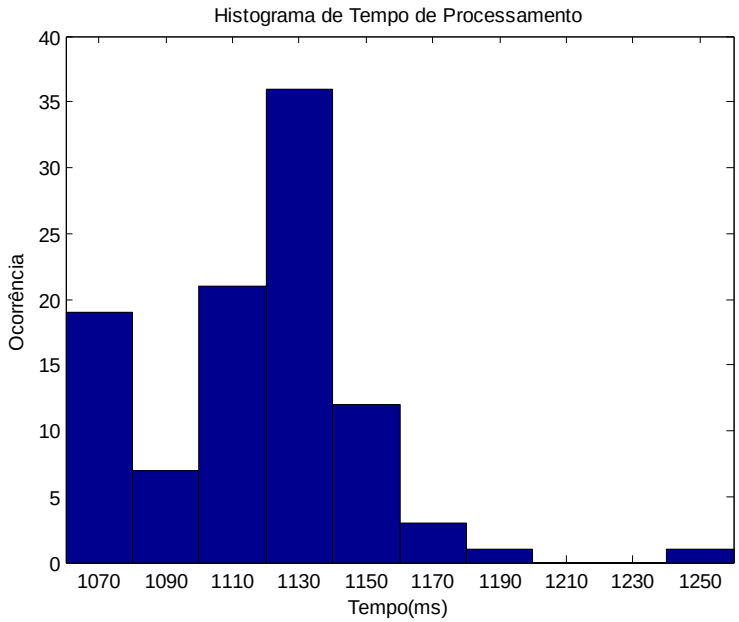


Fig.7.2 Histograma do Tempo de Processamento

A taxa de detecção nas condições mencionadas foi de 96%, ou seja, em quatro quadros de cem o rosto não foi detectado.

O desempenho em termos de tempo de processamento na plataforma DSL, resumido na próxima tabela, foi ligeiramente inferior:

Tabela 7.2 Dados estatísticos do processamento (Detecção de Face) – DSL

Tempo máximo	1510 ms
Tempo mínimo	1240 ms
Tempo médio	1263 ms
Desvio Padrão	51.9 ms

Foi interessante obter a relação entre o tempo de processamento e outras funções. Portanto foi testado o tempo de carregamento do frame. Este tempo resultou em 20 ms. Sendo que o programa converte imagem colorida para escala de cinza – procedimento que não será feito na aplicação eventual (já que a última deve processar imagem infravermelha), foi testado o tempo da conversão que resultou em 10 ms. Assim foi deduzido que a maior parte do tempo gasto é exigida pelo classificador *Haar Cascade*.

Para fins de comparação, o mesmo sistema operacional (DSL) foi testado em duas máquinas adicionais, uma Notebook Asus Eee 1000HA, processador Intel Atom N270, 1.6GHz, 1GB de RAM e outra, HP Compaq , processador Intel Pentium 4 HT, 3GHz, *single core*, 512 MB de RAM.

Resultados de processamento estão mostrados abaixo (Tabela 7.3):

Tabela 7.3 Dados estatísticos do processamento – DSL – Intel Atom e Intel Pentium 4

Intel Atom		Intel Pentium 4	
Tempo mínimo	330 ms	Tempo mínimo	150 ms
Tempo máximo	380 ms	Tempo máximo	200 ms
Tempo médio	339.2 ms	Tempo médio	161.5 ms

7.2 DETECÇÃO DE OLHO

Nesta parte o algoritmo completo foi testado em seis pessoas com sistema operacional DSL. Neste caso também foi usada seqüência de vinte frames para cada testado. Porém, a distância da câmera tinha que ser diminuída para aproximadamente 50 centímetros. As imagens resultantes podem ser vistos na Fig.7.3. Observamos uma imperfeição na localização na Fig. 7.2(a), com olho fechado, e ausência de detecção de olho na Fig. 7.2(f).

Em relação à porcentagem, houve 97% de detecção em cinco primeiros indivíduos, e apenas 5% no sexto indivíduo. Deve ser observado que o ultimo possui características faciais orientais fortemente acentuadas.

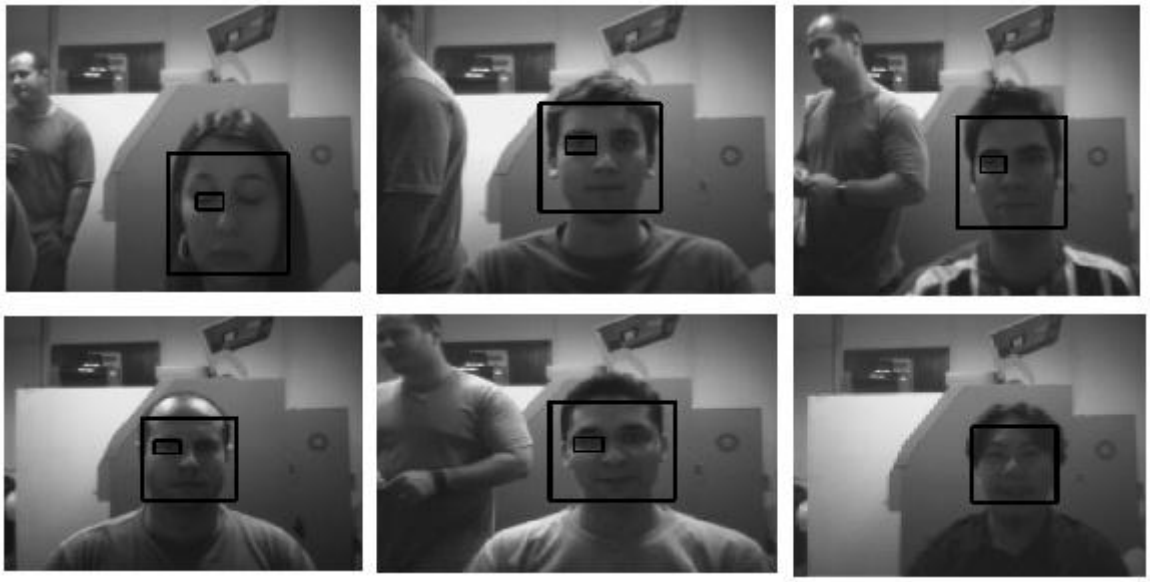


Fig. 7.3 Imagens resultantes de Detecção de Face e Olho (a)(b)(c) (d)(e)(f)

O tempo de execução está resumido na tabela 7.4. Para fins de comparação do desempenho em relação ao tempo de execução, o algoritmo foi testado também em sistema operacional Debian, apresentando resultados melhores.

Tabela 7.4 Dados estatísticos do processamento Detecção de Olho – DSL e Debian

DSL		Debian	
Tempo mínimo	1270 ms	Tempo mínimo	1130 ms
Tempo máximo	1480 ms	Tempo máximo	1150 ms
Tempo médio	1349.3ms	Tempo médio	1140.8 ms

É interessante observar, comparando os dados das tabelas 7.1, 7.2 e 7.4, que a parte de detecção de olho adiciona pouco tempo no processamento total. Este resultado deve ser atribuído à diminuição da região de interesse. Podemos concluir, então, que existe uma forte relação de *trade off* entre a resolução da imagem e o tempo de processamento. Ou seja, imagem com melhor resolução provavelmente apresente resultados mais confiáveis em termos de detecção e detecções falsas, porém aumentaria significativamente o tempo de processamento.

7.3 IMPLEMENTAÇÃO DE DECISÃO SOBRE O ESTADO DO OLHO

Nesta parte foi usada a foto da Fig. 4.3. O programa desenvolvido foi essencialmente o mesmo do anexo I, com captura da imagem a partir de um arquivo e não a partir da câmera. Ademais, foi adicionada a parte de conversão para imagem binária com limiar alto de 240 ui,

cálculo da média de intensidade da imagem binária e a decisão. O *snippet* desta parte do programa, não presente no anexo I, é apresentado a seguir:

```
... /*coloca o olho como a região de interesse*/

cvSetImageROI(
        img,
        cvRect(
            face->x + eye->x,
            face->y + (face->height/5.5) + eye->y,
            eye->width,
            eye->height));

/* aloca meméria para imagem do tamanho de olho */

        IplImage* img2 = 0;
        img2=cvCreateImage(cvSize(eye->width,eye->height),8,1);

/* aplica limiar alto e grava a imagem binária em img2 */

        cvThreshold(img,img2,240,255,CV_THRESH_BINARY);
        cvShowImage("janela2",img2);

/* calcula média e desvio padrão */
        cvAvgSdv( img2, &mean, &sdv );

////////////////////////////////////////////////////////////////

/* classifica */

if ( mean.val[0] == 0)
       {
          cout << " olho fechado \n";
          cvReleaseImage( &temp );
        cvReleaseImage( &img2 );
          cvReleaseHaarClassifierCascade( &cascade_face );
          cvReleaseHaarClassifierCascade( &cascade_eye );
          cvReleaseMemStorage(&storage );
          cvResetImageROI(img);
          return 0;
       }
////////////////////////////////////////////////////////////////
      cout << " olho aberto \n";
            cvReleaseImage( &temp );
      cvReleaseImage( &img2 );
            cvReleaseHaarClassifierCascade( &cascade_face );
            cvReleaseHaarClassifierCascade( &cascade_eye );
            cvReleaseMemStorage(&storage );
        cvResetImageROI(img);
}
```

O tempo de execução para Debian era de 1120 ms e para DSL – 1270 ms.

8 ANÁLISE E CONCLUSÕES

Analisando o desempenho da placa PCM-9375 percebemos que o tempo de processamento não é satisfatório para aplicação alvo. Contudo, este trabalho apresenta apenas uma tentativa inicial, que pode ser melhorada radicalmente, por meio de procedimentos detalhados na frente.

O objetivo não foi alcançado para a placa SBC2440-I, porém, existe relato na web sobre o funcionamento desta placa, utilizando a biblioteca OpenCV e classificador *Haar Cascade* para detecção de face. Assim, Stepura chega ao tempo de processamento de 448151ms na detecção a partir de uma imagem, com placa SBC2440-III, que emprega o mesmo processador. [16] No nosso caso, apesar de não chegar a processar imagem, apenas o carregamento de frames durava em torno de um segundo. Portanto podemos concluir que OpenCV não deve ser usado com arquitetura ARM. Provável causa é que OpenCV esteja otimizada para arquitetura x86, resultando em desempenho drasticamente inferior em outras plataformas. (Um reforço para esta teoria é o fato dela ser desenvolvida por Intel).

Outro detalhe importante a considerar é o melhoramento radical no desempenho quando são utilizados processadores mais potentes. Por outro lado, a nossa aplicação alvo não impõe uma restrição muito severa sobre o tamanho do produto e a dissipação de energia, já que estaria sendo usado em movimento, quando o alternador do veículo consegue suprir a demanda elétrica do sistema. Para bateria de um caminhão esta carga tampouco é considerável. O mesmo raciocínio aplica-se ao tamanho do processador. Como não estamos desenvolvendo um aparelho portátil, seria interessante investigar a possibilidade de compromisso entre o tamanho reduzido e pouca dissipação de energia, e o desempenho em termos de velocidade de processamento, dando a preferência ao último (idealmente pelo mesmo preço).

O processamento da imagem pode ser melhorado, após fazer testes de posicionamento relativo entre a câmera e o motorista. O algoritmo *Haar Cascade* da OpenCV de fato é especializado em encontrar várias faces, de tamanhos diferentes dentro de uma imagem. Isto também leva a algumas poucas detecções falsas, que não foram tratadas neste trabalho. Para fazer este tipo de detecção, *Haar Cascade* muda o tamanho da janela do detector. Sabendo o posicionamento do motorista, podemos restringir o tamanho da janela apenas para o compatível com o rosto, o que diminuiria em muito o tempo de classificação, e ao mesmo tempo eliminaria quase por completo detecções falsas.

Embora a detecção de face rendesse resultados semelhantes para todos os testados, na detecção de olho observamos uma forte evidência de que o classificador foi treinado com amostras de características européias. Isto reflete no pobre poder de generalização do algoritmo. Um treinamento com amostras adicionais de características variáveis poderia melhorar o poder de generalização, porém, resultaria em pioramento no desempenho geral, alem de dificuldade na implementação de tal treinamento. Por outro lado, a possibilidade de aproveitar a propriedade da reflexão da córnea faz questionar a necessidade de detecção de olho. Uma alternativa seria estabelecer a região de interesse na parte superior da face

encontrada e aplicar o algoritmo descrito na seção 7.3 diretamente. Este procedimento deve resultar em menor complexidade computacional e menor taxa de erros.

Ainda visando melhoramento, trabalho de configuração de kernel deve ser feito, para deixar apenas as funcionalidades necessárias para o funcionamento do sistema, para que processos alheios não consumam os recursos de processamento. Neste trabalho não foi dada suficiente atenção para este problema.

O uso do DSL como sistema operacional não é vantajoso por usar kernel 2.4, que dificulta a instalação de pacotes necessários. Neste sentido o ideal é trabalhar com mesma versão do kernel no *host* e no *target* para manter a plena compatibilidade dos recursos.

REFERÊNCIAS BIBLIOGRÁFICAS

1. EMBEST. SBC2440I Single Board User Manual.
2. ADVANTECH. PCM-9375 User's Manual. Disponivel em: <http://download.advantech.com//DownloadFiles/1-2K8L6G/PCM-9375_User_Manual_ed2.pdf>. Acesso em: 3 jul. 2009.
3. ARAÚJO BORGES, G. Curso de extensão em Programação Linux e Desenvolvimento de Sistemas Embarcados. Brasília: Faculdade de Tecnologia, Universidade de Brasília, v. IV.
4. ZELENOVSKI, R.; MENDOÇA, A. PC: Um Guia Prático de Hardware e Interfaceamento. 4. ed. [S.l.]: MZ, 2006.
5. SLOSS, A.; SYNES, D.; WRIGHT, C. ARM SYSTEM DEVELOPER'S GUIDE Designing and Optimizing System Sofware. [S.l.]: Morgan Kaufmann Publishers , 2004.
6. KERNEL(COMPUTING). Disponivel em: <http://en.wikipedia.org/wiki/Kernel_(computing)>.
7. YAFFS - A Flash File System. Disponivel em: <http://www.yaffs.net/>. Acesso em: 3 jul. 2009.
8. RODRIGUES DE OLIVEIRA, F. A.; GUEDES DE ARAÚJO DIAS, A. Verificação de Falhas em Linhas de Transmissão por Redes Neurais Artificiais a partir dos Espectros de Freqüência de Imagens, Brasília, jul. 2007.
9. JENSEN, O. H. Implementing the Viola-Jones Face Detection Algorithm, Kongens Lyngby, 2008.
10. VIOLA, P.; JONES, M. Robust Real-time Object Detection, Vancouver, 2001.
11. LEK HONG MA, E. Avaliação de Características Haar em Um Modelo de Detecção de Face, Brasília, 2007.
12. SMYTH, P. Face Detection using the Viola-Jones Method, 2007. Disponivel em: <www.ics.uci.edu/~smyth/./slides12_viola_jones_face_detection.ppt>. Acesso em: 3 jul. 2009.
13. LIENHART, R.; MAYDT, J. An Extended Set of Haar-like Features for Rapid Object Detection, 2002.
14. HIGHGUI Reference Manual. Disponivel em: <http://opencv.jp/opencv-1.0.0_org/docs/ref/opencvref_highgui.htm>. Acesso em: 03 julho 2009.
15. OPENCV Installation Guide, 2009. Disponivel em: <http://opencv.willowgarage.com/wiki/InstallGuide?highlight=%28arm%29>. Acesso em: 3 jul. 2009.
16. STEPURA, I. ARM-wrestling with OpenCV, 18 jun. 2009. Disponivel em: <http://www.computer-vision-software.com/blog/2009/03/arm-wrestling-with-opencv/>. Acesso em: 3 jul. 2009.
17. SANTANA, P. H. et al. Building a Real-Time Debian Distribution for Embedded Systemes. Brasília: Robotics and automation Laboratory FT/UnB, 2009.
18. GRUB Manual. Disponivel em: <http://gnu.org/software/grub/manual/grub.html>. Acesso em: 3 jul. 2009.
19. BRADSKI, G.; KAEHLER, A. Learning OpenCV Computer Vision with the OpenCV

Library. 1ª Edição. ed. [S.l.]: O'Rielly, 2008.

20. JARNO, A. Debian on an emulated ARM machine. Disponivel em: <http://www.aurel32.net/info/debian_arm_qemu.php.en>. Acesso em: 03 jul. 2009.
21. ALVES DA FONSECA, T. Introdução ao desenvolvimento com linux. Brasília: GPDS/FT/UnB, v. v1.
22. HAYHOE, M. M.; SHRIVASTAVA, A.; PELZ, J. B. Visual Memory and Motor Planning in a Natural Task. Journal of Vision. Disponivel em: <http://www.journalofvision.org/3/1/6/article.html>. Acesso em: 3 jul. 2009.

ANEXOS

I *Script* do programa de detecção de face e olho
II Lista de bibliotecas dinâmicas que foram instaladas na placa SBC2440-I

ANEXO I

```
/* O programa executa o seguinte laço 20 vezes:
lê entrada de câmera, detecta rosto e olho,
mede o tempo de execução sem interface gráfica
e grava num arquivo. Grava o vigésimo frame
Precisa de argumento nome do arquivo .txt para gravar dados */

#include "highgui.h"
#include "cv.h"

#include <stdlib.h>
#include <stdio.h>
#include <iostream>
#include <cstdio>
#include <string.h>
#include <time.h>

using namespace std;



int detect_and_draw( IplImage* image );


int main( int argc, char** argv ){



 IplImage* img=0;
 IplImage* frame=0;
 int detect = 0;
 int aux = 0;
 double exec_time_total=0;
 clock_t start, end;



CvCapture* capture = cvCreateCameraCapture(0);



FILE * pFile;
pFile = fopen (argv[1],"a");                        // o nome do arquivo deve
ser passado,
                                                    /* flag append-nao apaga
conteudo anterior*/
fprintf(pFile, " Execution Time(ms)(Detection Only):\n \n");

for (int i=0; i<20 ; i++)
{
cout << "Frame " << i+1 << ". \n";

//Tempo comeca aqui
start = clock();
//////////////////////////
        frame=cvQueryFrame(capture);
        if(!frame){ cout << "No input\n";
        return 0;}

img = cvCreateImage(cvSize(frame->width,frame->height),8,1);
```

```
cvCvtColor(frame,img,CV_BGR2GRAY);

aux = detect_and_draw(img);

//Tempo verificado aqui
end = clock();

detect = detect+aux;

////////////////////////////////////////////////////////////////////
/* Calcula tempo de execuÃ§Ã£o e grava na tela e no arquivo */

double exec_time = ((end + 0.0)-(start + 0.0))*1000 / CLOCKS_PER_SEC;

exec_time_total = exec_time_total+exec_time;

printf ("Execution time: %.1f ms.\n",exec_time );
fprintf (pFile,"%.1f\n",exec_time );

////////////////////////////////////////////////////////////////////

// Grava o vigésimo frame com janelas de rosto e olhos no arquivo test1.jpg

if (i==19){
cvSaveImage("test1.jpg",img);
          }
//_____________________________________________________________________

cvReleaseImage( &img );
} // fecha o laco de 20 vezes

////////////////////////////////////////////////////////////////////

/* Calcula e grava dados estatísticos do processamento */

double taxa_acertos = ((double)detect/20)*100;
double exec_time_med = exec_time_total / 20;
printf ("Detection Percentage: %.1f \n",taxa_acertos );
printf ("Time med.: %.1f ms. \n",exec_time_med );
fprintf (pFile,"Detection Percentage: %.1f %\n",taxa_acertos );
fprintf (pFile,"Time med.: %.1f ms. \n",exec_time_med );

////////////////////////////////////////////////////////////////////
      cvReleaseCapture(&capture);
      return 0;
} //termina main


////////////////////////////////////////////////////////////////////////////
//////
/* Função detecta face e olho, desenha retângulos ao redor */

int detect_and_draw( IplImage* img ){

/* declara variÃ¡veis*/
      int i;
      CvSeq* faces = 0;
      CvSeq* eyes = 0;
      CvRect* face = 0;
      CvRect* eye = 0;
      static CvMemStorage* storage = 0;
```

```
      static CvHaarClassifierCascade* cascade_face = 0;
      static CvHaarClassifierCascade* cascade_eye = 0;
      IplImage *temp = cvCreateImage( cvSize( img->width, img->height), 8,
3 );


    cascade_face = (CvHaarClassifierCascade*)cvLoad( "hface.xml", 0, 0, 0
);
    cascade_eye = (CvHaarClassifierCascade*) cvLoad( "heye.xml", 0, 0, 0
);

//////////////////////////////////////////////////////////////////////////
//////

    if( !cascade_face )
    {
        fprintf( stderr, "ERRO!: Nao carregou o classificador de face\n" );
        cvReleaseImage( &temp );
        cvReleaseHaarClassifierCascade( &cascade_face );
        cvReleaseHaarClassifierCascade( &cascade_eye );
        cvReleaseMemStorage(&storage );
        return 0;
    }
//////////////////////////////////////////////////////////////////////////
//////

    storage = cvCreateMemStorage(0);

/*Aplica o classificador para encontrar faces */

    faces = cvHaarDetectObjects(
            img,
            cascade_face,
            storage, 1.1,
            2, 0, cvSize(40, 40));

//////////////////////////////////////////////////////////////////////////
//////

if ( faces->total == 0)
       {
          cout << " Nao achei rosto \n";
          cvReleaseImage( &temp );
          cvReleaseHaarClassifierCascade( &cascade_face );
          cvReleaseHaarClassifierCascade( &cascade_eye );
          cvReleaseMemStorage(&storage );
          return 0;
       }
//////////////////////////////////////////////////////////////////////////
///////

        face = (CvRect*)cvGetSeqElem( faces, 0 ); // apenas a primeira face

/*Desenha o retângulo da face*/

cvRectangle(
            img,
             cvPoint(face->x, face->y),
             cvPoint(face->x + face->width, face->y + face->height),
             CV_RGB(255,0,0), 3, 8, 0 );
```

```
cout << " rosto ok!\n";

////////////////////////////////////////////////////////////////////////
////////

 if( !cascade_eye )
    {
        fprintf( stderr, "ERRO!: Nao carregou o classificador de olho\n" );
        cvReleaseImage( &temp );
        cvReleaseHaarClassifierCascade( &cascade_face );
        cvReleaseHaarClassifierCascade( &cascade_eye );
        cvReleaseMemStorage(&storage );
        return 0;
    }

////////////////////////////////////////////////////////////////////////
////////

/* Define a posição de olhos estimada na face (apenas metade)*/

cvSetImageROI(
        img,
        cvRect(
            face->x,
            face->y + (face->height/5.5), // alguns pixels abaixo da
fronteira superior
            face->width/2.0,        // width = mesmo da face
            face->height/2.5 ));    // procura nos 2/5 superiores da face

/*aplica o classificador*/

eyes = cvHaarDetectObjects(
        img,
        cascade_eye,
        storage,
        1.15, 3, 0,
        cvSize(25, 15));

/*faz teste. se olho ok, continua, se não, volta*/

////////////////////////////////////////////////////////////

if ( eyes->total == 0)
       {
          cout << " Nao achei olhos \n";
          cvReleaseImage( &temp );
          cvReleaseHaarClassifierCascade( &cascade_face );
          cvReleaseHaarClassifierCascade( &cascade_eye );
          cvReleaseMemStorage(&storage );
          cvResetImageROI(img);
          return 0;
       }

/* Desenha o retângulo do olho */

eye = (CvRect*)cvGetSeqElem(eyes, 0);

cvRectangle(
            img,
            cvPoint(eye->x, eye->y),
            cvPoint(eye->x + eye->width, eye->y + eye->height),
```

```
            CV_RGB(255, 0, 0),
            2, 8, 0);

cout << " olho ok!\n";

        cvReleaseImage( &temp );
          cvReleaseHaarClassifierCascade( &cascade_face );
          cvReleaseHaarClassifierCascade( &cascade_eye );
          cvReleaseMemStorage(&storage );
          cvResetImageROI(img);
          return 1;
}
```

ANEXO II

```
ld-linux.so.2
libatk-1.0.so.0
libc.so.6
libcairo.so.2
libcv.so.2
libcxcore.so.2
libexpat.so.1.0.0
libfontconfig.so.1
libfreetype.so.6
libpango-1.0.so.0
libpangocairo-1.0.so.0
libpangoft2-1.0.so.0
libpng12.so.0
libpthread.so.0
librt.so.1
libstdc++.so.6
libtiff.so.3
libX11.so.6
libXau.so.6
libXcursor.so.1
libXdmcp.so
libXext.so.6
libXfixes.so.3
libXi.so.6
libXinerama.so.1
libxml2.so.2.6.32
libxml++-2.6.so.2.0.7
libXrandr.so.2
libXrender.so.1
libz.so.1
```