Desenvolvimento de um controlador embarcado com visão digital para o Robô-VD

Trabalho de Conclusão de Curso

Engenharia da Computação

Autor: Cristóvão Zuppardo Rufino

Orientador: Prof. Dr. Sergio Campello Oliveira

Cristóvão Zuppardo Rufino

Desenvolvimento de um controlador embarcado com visão digital para o Robô-VD

Monografia apresentada como requisito parcial para a obtenção do diploma de Bacharel em Engenharia da Computação pela Escola Politécnica de Pernambuco – Universidade de Pernambuco.

Recife, junho de 2011.

De acordo,

Recife

06 de junho de 2011

_____

Sérgio Campello Oliveira

Dedico a meus pais, Francisco Arnaldo e Weldina, e a Laiz, minha futura esposa.

Agradecimentos

Agradeço, primeiramente, a Deus pela saúde, a capacidade e os talentos que Ele me deu para me dedicar aos estudos de engenharia e áreas de tecnologia.

Agradeço também a meus pais, Francisco Arnaldo e Weldina, que me incentivaram, desde pequeno, a estudar e desenvolver minhas capacidades e talentos e que criaram um ambiente propício ao estudo. Agradeço também a meu irmão, Estêvão, pelas ideias e sugestões trocadas.

Agradeço a minha noiva pela paciência e compreensão durante minha ausência neste período de estudo, implementação e escrita da monografia.

Agradeço a meu professor orientador que me deu excelentes ideias e ajuda na escrita deste trabalho. Agradeço também a meus outros professores e colegas de graduação que ensinaram as técnicas e deram ideias para a implementação deste projeto.

Resumo

Processamento de imagens tem sido amplamente utilizado em vários dispositivos que nos cercam. Câmeras digitais, computadores e outros dispositivos já contam com *software* e *hardware* dedicados exclusivamente para isso. Apesar do uso de processamento de imagens e visão computacional estar se difundindo nas atividades cotidianas os dispositivos que são mais utilizados para se processar imagens são os computadores de uso geral. Neste trabalho foi utilizado um dispositivo embarcado para processar imagem e, a partir desse processamento, tomar uma decisão sem auxílio de um computador de propósito geral. Foi desenvolvida uma plataforma de simulação para testar a implementação das rotinas de controle do microcontrolador. Para validar a plataforma testes foram feitos com imagens construídas com programas de edição de imagem, tal como o GIMP, e imagens reais utilizando uma câmera comum. O principal objetivo deste trabalho é a plataforma em si assim como o *software* desenvolvido que será distribuído para estudo e modificação.

Abstract

Image processing has been widely used in various devices that surround us. Digital cameras, computers and other devices already come with software and hardware dedicated exclusively for this. Despite the use of image processing and computational vision techniques in our day-to-day activities this processing is done mostly by general-purpose computers. This work used an embedded device for image processing and then take a decision without the aid of a general-purpose computer. It has been developed a simulation platform to test the implementation of the control routines on the microcontroller. To validate the plataform tests were performed with images created with image editing programs, such as GIMP, and real images using a regular camera. The main objective of this work is the platform itself as well as software developed which will be distributed for study and modification.

Sumário

Índice de figuras

Índice de tabelas

Lista de acrônimos

- AD ou ADC / DA ou DAC – *Analogic to Digital Converter* (Conversor analógico para digital) e *Digital to Analogic Converter* (Conversor digital para analógico)
- CAN – *Controller-area Network* (Rede de controladores em área)
- CCD – *Charge-Coupled Device* (Dispositivo de carga acoplada)
- CDF – Cumulative Distribution Function (Função de distribuição cumulativa)
- CISC – *Complex Instruction Set Computer* (Computador com conjunto complexo de instruções)
- CPU – Central Processing Unity (Unidade Central de Processamento)
- DSP – *Digital Signal Processor* (Processador digital de sinal)
- DMA – *Direct Memory Access* (Acesso direto a memória)
- EEPROM ou E²PROM – *Eletrically Eraseable Programable Read-Only Memory* (Memória de somente-leitura programável e apagável eletricamente)
- GPIO – *General Purpose Input/Output* (Entrada e saída de propósito geral)
- I2C ou I²C – *Inter-Integrated Circuit* (Cirtuito entre circuitos integrados)
- I2S ou I²S – *Inter-IC Sound* ou *Integrated Interchip Sound* (Som integrado entre circuitos integrados)
- I/O – *Input / Output* (Entrada e saída)
- MAC¹ – *Multiply and Accumulate* (Multiplicar e Acumular)
- MAC² – *Media Access Control* (Controlador de Acesso ao meio)
- PWM – *Pulse Width Modulation* (Modulação por largura de pulso)
- RAM – *Random Access Memory* (Memória de Acesso Aleatório)
- RISC – *Reduced Instruction Set Computer* (Computador de conjunto de instruções restrito)
- RTC – *Real Time Clock* (Relógio de tempo real)
- SPI – *Serial Peripheral Interface* (Interface periférica serial)
- SSP – *Synchronous Serial Port* (Porta serial síncrona)
- UART – *Universal Asynchronous Receiver/Transmitter* (Transmissor/Receptor

universal assíncrono)

- USART – *Universal Synchronous/Asynchronous Receiver/Transmitter* (Transmissor/Receptor universal síncrono/assíncrono)
- USB – *Universal Serial Bus* (Barramento serial universal)
- WDT – *Watchdog Timer* (Temporizador cão-de-guarda)

Capítulo 1 – Introdução

Processamento de imagem e visão computacional possuem muitas aplicações na automação de processos como a automação industrial, médica, segurança e outras [1] [2].

Uma característica marcante de todas essas aplicações é que normalmente o processamento é realizado por um computador de propósito geral o que pode tornar o projeto de alguma solução inviável tanto em termos de custo como de tamanho.

Os microcontroladores, hoje, possuem vários periféricos embarcados, tais como portas seriais, *I2C*, *Ethernet*, *USB*, conversores AD/DA entre outros. São comuns microcontroladores com *clock* na ordem de dezenas de *megahertz* o que permite a execução de milhares de instruções por segundo. Também já encontramos microcontroladores de 8, 16 e até 32 *bits*. Alguns tipos de microcontroladores possuem instruções próprias para processamento de sinais, tais como instruções *MAC* (multiplicar e acumular – do inglês *Multiply and Accumulate*) presentes em *DSP's* (Processadores de Sinais Digitais – do inglês *Digital Signal Processor*).

Dados tais avanços, vários algoritmos de processamento de imagem e visão computacional tem sido implementados em microcontroladores, como podemos ver nos trabalhos de *Wang* [3], *Aggarwal* [4] e Mascarenhas [5]. Em 2003, Rufino [1] desenvolveu um sistema de visão digital com detecção de objetos implementado em um computador de propósito geral para controlar o Robô-VD, um braço robótico cujo objetivo é interagir com objetos em uma área de trabalho. Estes algoritmos são bastante simples e consomem poucos recursos computacionais tornando possível a implementação deles em sistemas embarcados.

Visando miniaturização e redução de custos, neste trabalho foi desenvolvido um sistema de processamento de imagens totalmente embarcado, o que possibilita que seu uso seja feito sem a necessidade de um computador de propósito geral. Um microcontrolador é muito menor, gasta bem menos energia e é centenas de vezes mais barato que um computador convencional. Neste trabalho foi desenvolvido uma plataforma de simulação. Devido a complexidade da integração da plataforma desenvolvida ao braço robótico será deixado como possível trabalho futuro.

1.1 Estrutura do documento

Este trabalho está dividido em seis capítulos: o primeiro capítulo contém a introdução ao trabalho. No segundo capítulo apresentar-se-á uma explicação sobre imagens digitais, elementos básicos e alguns algoritmos de processamento de imagens. O terceiro capítulo conterá uma breve explicação de sistemas embarcados, aplicações e as características do microcontrolador utilizado neste trabalho. O quarto capítulo falará do processamento embarcado de imagens e das soluções implementadas neste trabalho. No quinto capítulo serão apresentados os resultados obtidos e no sexto capítulo apresentar-se-ão as conclusões e trabalhos futuros.

Capítulo 2 – Processamento digital de imagens

Neste capítulo será apresentado uma introdução às imagens digitais. Serão apresentados os elementos básicos das imagens digitais, métodos de aquisição de imagens, métodos de tratamento de imagens digitais e algoritmos de processamento e detecção de objetos.

2.1. Imagem

Uma imagem pode ser vista como uma função bidimensional $f(x, y)$. O valor, ou amplitude de f em coordenadas espaciais é uma quantidade escalar positiva cujo significado físico está relacionado à fonte da imagem. Quando uma imagem é gerada por um processo físico, seu valor é proporcional a energia irradiada da fonte. Como consequência seu valor é finito, isto é [6]:

$$0 < f(x,y) < \infty$$
$$f(x,y) = i(x,y) \cdot r(x,y) \quad (1)$$

Como podemos ver na equação (1), a imagem $f(x, y)$ é uma relação entre a iluminação (representada pela função $i(x, y)$) e a reflectância (representada pela função $r(x, y)$). A iluminação é uma função que representa a quantidade de energia que a fonte de luz pode fornecer e a reflectância é uma função das características do objeto [6].

Um objeto pode refletir totalmente a luz que incide sobre ele (reflectância máxima, representada por 1) ou absorver a luz completamente (reflectância mínima, representada por 0).

2.2 Métodos de aquisição de imagens

Os tipos de imagens que normalmente são processadas vem de uma combinação da iluminação da fonte e da reflectância do objeto iluminado, como vimos na sessão anterior. Existem outros métodos de aquisição de imagens que não necessitam desses elementos, tais como imagens geradas por radar, infravermelho, raios-X, ultrassom e outras [6].

Todas essas imagens precisam ser capturadas por um dispositivo sensível a grandeza física que essas fontes estão fornecendo. Tais dispositivos são chamados de “sensores” de imagem. Existem vários tipos de sensores, na Figura 1 vemos a representação de um modelo de sensor.

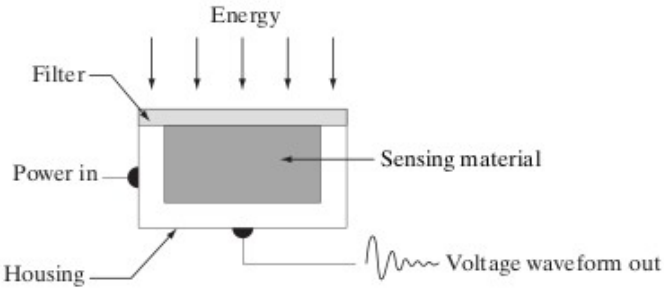


Figura 1: Exemplo de sensor de imagem.

Extraído de [6]

O sensor representado pela Figura 1 recebe energia de uma fonte (um objeto, por exemplo), é filtrada para remoção de ruídos e outros tipos de energia que são desnecessárias e, então, é exposto a um material sensível ao tipo de energia que o sensor foi projetado para detectar. Se o sensor for elétrico, um sinal elétrico proporcional à quantidade de energia detectada é gerado e pode ser capturado. Esse sinal representa a intensidade de energia a que o sensor está exposto. Vários desses sensores podem ser arranjados para se obter um arranjo em linha ou uma matriz deles.

2.2.1 Amostragem, quantização e representação de imagens digitais

Na sessão 2.2 foi mostrado um tipo de sensor. Para se obter imagens digitais é necessário transformar o sinal contínuo detectado pelos sensores em informações digitais. Tal processo é composto de duas etapas chamadas de *amostragem* e *quantização*. A transformação da representação espacial contínua do objeto é chamada de amostragem. Já a transformação da representação de intensidade luminosa do objeto é chamada de quantização.

Como pode-se ver representado na Figura 2, o objeto tem uma região demarcada, como a região *AB* (representado pelas figuras *(a)* e *(b)*). Ela, por sua vez, é lida em intervalos regulares (representado por *(c)*). Essas leituras são chamadas de amostragem. Com esses valores amostrados, as leituras de intensidade são realizadas e passam a ser representadas por um valor numérico correspondente (representado por *(d)*). Este processo é a quantização.

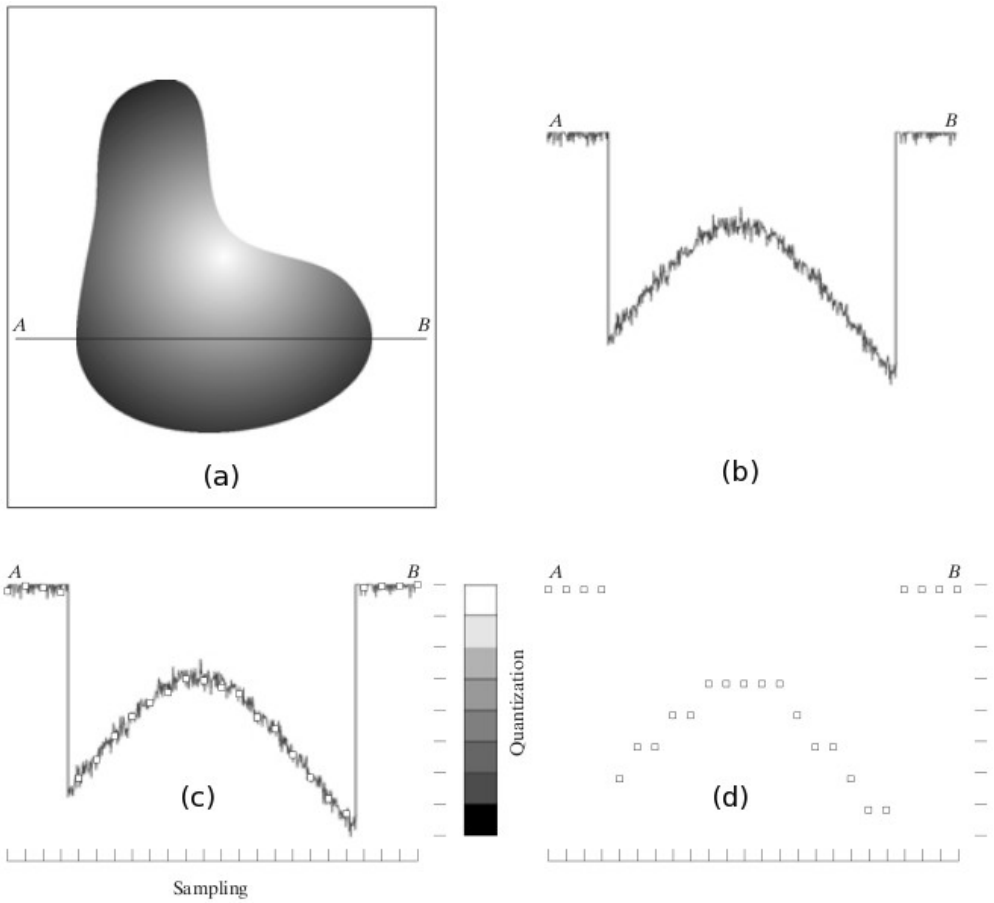


Figura 2: Representação da amostragem e quantização.

Extraído de [6]

Quando um objeto é amostrado e quantizado por um arranjo matricial, por exemplo, tem-se uma representação amostrada e quantizada do objeto. Um exemplo disso pode ser visto na Figura 3. Neste exemplo, o objeto (à esquerda da imagem) está sendo exposto a um sensor com arranjo matricial e a imagem digital correspondente (à direita) a cada ponto representado é um nível de cinza representando a intensidade de luz detectada. Deste modo tem-se uma representação digital de um objeto.

Esta representação digital é tal que a imagem final possua M linhas por N colunas. Os valores das coordenadas (x, y) passam a ser valores discretos e é padronizado que a origem da imagem, ou seja o ponto (0, 0) da imagem, fica no canto superior esquerdo. A coordenada x cresce da esquerda para a direita e a coordenada y cresce de cima para baixo.

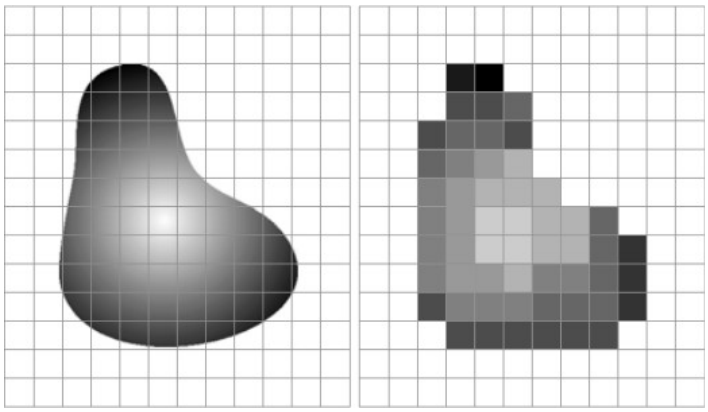


Figura 3: Representação da amostragem e quantização de um objeto.

Extraído de [6]

Cada ponto da imagem digital é denominado *pixel*. Um *pixel* é o menor elemento da imagem e o valor dele representa a quantização da intensidade de luz do objeto. Se a imagem for apenas em níveis de cinza, tal intensidade é comumente representada pelo intervalo discreto [0, 255], sendo 0 a menor intensidade possível (representada pela cor preta) e 255 a maior intensidade possível (representada pela cor branca).

Neste trabalho serão utilizadas imagens em níveis de cinza com intervalos variando entre 0 e 255 (do preto ao branco, respectivamente). Tal representação foi escolhida por ser bastante utilizada e por responder bem aos algoritmos utilizados.

É bastante comum representar uma imagem digital como uma matriz de M por N elementos onde cada elemento da matriz representa um pixel da imagem:

$$f(x,y)=\begin{bmatrix} f(0,0) & f(0,1) & \cdots & f(0,M-1) \\ f(1,0) & f(1,1) & \cdots & f(1,M-1) \\ \vdots & \vdots & \vdots & \vdots \\ f(N-1,0) & \cdots & \cdots & f(N-1,M-1) \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1M} \\ a_{21} & a_{22} & \cdots & a_{2M} \\ \vdots & \vdots & \vdots & \vdots \\ a_{N1} & a_{N2} & \cdots & a_{NM} \end{bmatrix} = \quad (2)$$

Onde em (2) $\boldsymbol{A} = \{a_{ij}\} = f(x = i - 1, y = j - 1)$.

2.2.2 Histograma da imagem

O histograma de frequência, comumente conhecido apenas como *histograma*, é uma maneira de representar graficamente a frequência dos dados de uma amostra [7]. É uma ferramenta grandemente utilizada na estatística e normalmente é apresentado em um gráfico chamado de *histograma* como pode ser visto na Figura 4.

O histograma de uma imagem é um gráfico de barras, onde a abcissa representa a intensidade do nível de cinza e a ordenada representa a frequência, ou seja, a quantidade que tal nível de cinza aparece na amostra. A Figura 4 é um exemplo de

histograma de uma imagem de 256*x*256 *pixels* com 256 níveis de cinza.

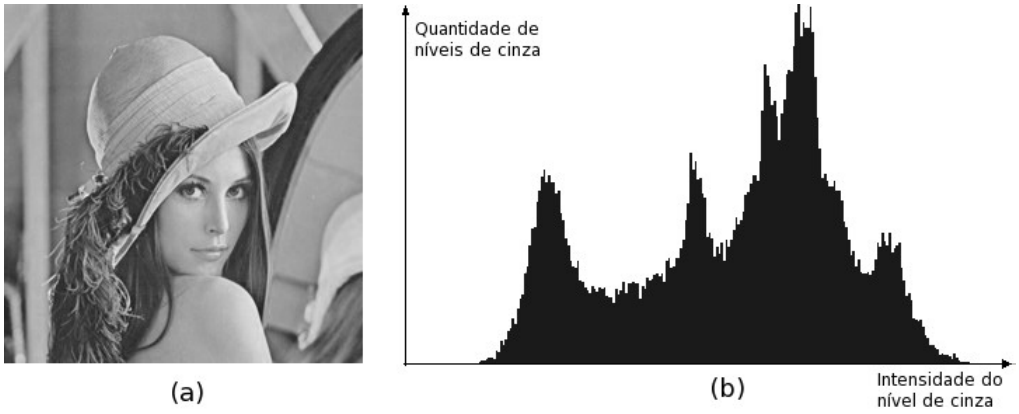


Figura 4: Exemplo de histograma.

(a) a imagem original, (b) o histograma correspondente da imagem

2.3 Realce espacial de imagens

Por realce entende-se um tratamento para melhorar algum aspecto da imagem, tais como bordas, linhas, contornos, contraste e outros [6]. Por espacial entende-se que o domínio de trabalho são as coordenadas cartesianas. Consequentemente o realce espacial de imagens trata de operações que são realizadas diretamente sobre os *pixels* da imagem.

Algumas técnicas de realce utilizam operadores que atuam diretamente sobre um único *pixel*, tal como as técnicas de equalização de histograma, que será descrita no próximo tópico. Outras técnicas de realce utilizam uma ferramenta matemática chamada de convolução de matrizes (que será abordada mais adiante) que consiste em uma operação entre duas matrizes, uma delas é a própria imagem e a outra matriz é chamada de “máscara” ou “filtro” [6].

2.3.1 Equalização de histograma

A equalização de histograma é uma técnica simples que pode melhorar imagens com baixo contraste. Por contraste entende-se uma relação entre a diferença de luminescência com a luminescência média. A equalização de histograma pode auxiliar no processo de detecção de objetos por tornar mais visível a diferença entre os objetos presentes na imagem [6].

A equalização de histograma redistribui os níveis de cinza da imagem de modo que eles fiquem mais distribuídos. Essa redistribuição é feita a partir de um novo mapeamento dos níveis de cinza da imagem original na nova imagem.

O resultado da operação é visto na Figura 5. Podemos ver que houve maior

distribuição nos níveis de cinza depois que o histograma foi equalizado. Tal efeito pode ser benéfico em alguns casos, mas pode trazer alguns efeitos indesejados como surgimento de ruídos e manchas que não existiam na imagem original [6].

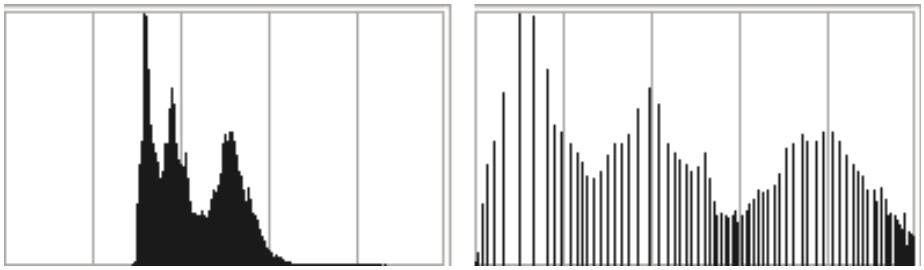


Figura 5: Histograma equalizado.

2.3.2 Convolução de matrizes

Conforme mencionado anteriormente o realce no domínio espacial faz uso da operação de convolução de matrizes. A convolução de matrizes é uma operação realizada entre duas matrizes e o resultado da operação é outra matriz, isto é:

$$\boldsymbol{G} = \boldsymbol{F} * \boldsymbol{W} \tag{3}$$

Onde $\boldsymbol{G}$ é o resultado da convolução entre a matriz $\boldsymbol{F}$ e a matriz $\boldsymbol{W}$. Normalmente em processamento digital de imagens, a matriz $\boldsymbol{W}$ é chamada de máscara. Alguns exemplos de máscaras bastante utilizadas em processamento de imagens digitais serão apresentados nas sessões posteriores.

A operação de convolução é computada da seguinte maneira: dadas uma matriz (ou uma imagem digital) $\boldsymbol{F}$ e uma máscara $\boldsymbol{W}$, a convolução será uma nova matriz $\boldsymbol{G}$ que é calculada da seguinte maneira [6]:

$$\boldsymbol{G} = \{g(x,y)\}$$
$$g(x,y) = \sum_{s=-a}^{a} \sum_{t=-b}^{b} w(s,t) \cdot f(x+s, y+t) \forall (x,y) \in M \times N \tag{4}$$

Em (4), admite-se que s e t variem entre [-a, a] e [-b, b] respectivamente. Como consequência a máscara $\boldsymbol{W}$ é uma matriz de m por n elementos, onde $m = 2a + 1$ e $n = 2b + 1$ respectivamente. A máscara $\boldsymbol{W}$ é passada na imagem toda, como representado na Figura 6.

A operação de convolução é bastante utilizada em processamento de sinais e de imagens. Uma grande aplicação é a filtragem de sinais, para remoção de ruídos ou aguçamento de propriedades. Uma imagem com muitos ruídos pode ser melhorada com um filtro de média aritmética, como será visto mais a frente.

Essa operação pode ser computacionalmente custosa pois é calculado o somatório de vários produtos. Porém será visto, no próximo capítulo, que essa operação pode ser bastante acelerada em microcontroladores modernos por causa de algumas

instruções que foram incorporadas neles recentemente.

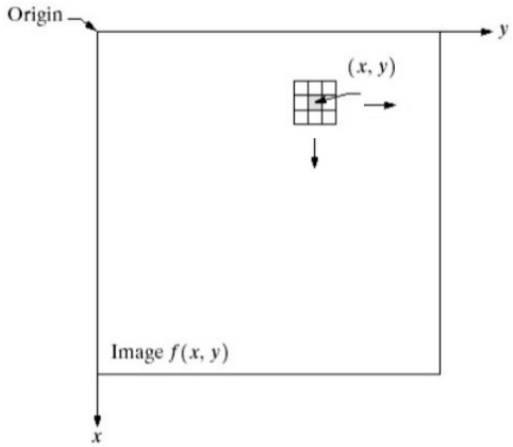


Figura 6: Representação gráfica da operação de convolução.

2.3.3 Suavização

A suavização de imagens digitais serve para borrá-las e remover ruído. Normalmente são utilizadas em etapas de pré-processamento para remover ruídos e prepará-las apara algum processamento posterior. A suavização pode ser alcançada por meio de filtros lineares, como o filtro de média, e filtros não lineares, como o filtro de mediana.

A ideia por trás dos filtros de média é direta: o valor do *pixel* da imagem filtrada é composto pela média aritmética (ou outro tipo de média) da soma dos valores dos *pixels* vizinhos da imagem original. Imagens com ruído aleatório possuem várias transições bruscas de valores de níveis de cinza, portanto uma suavização pode remover essas singularidades e facilitar o processamento da imagem [6].

A seguinte matriz representa um filtro de média aritmética 3x3 simples:

$$W_{média} = \frac{1}{9} \times \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

A próxima matriz representa um filtro de média ponderada 3x3:

$$W_{ponderada} = \frac{1}{16} \times \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

O resultado da filtragem de uma imagem por um filtro de média aritmética pode ser visto na Figura 7. Podemos observar que vários detalhes da imagem foram borrados, que as regiões com ruídos foram reduzidas e o efeito acima mencionado que as bordas sofrem perda de informação já que a máscara fica fora da imagem.

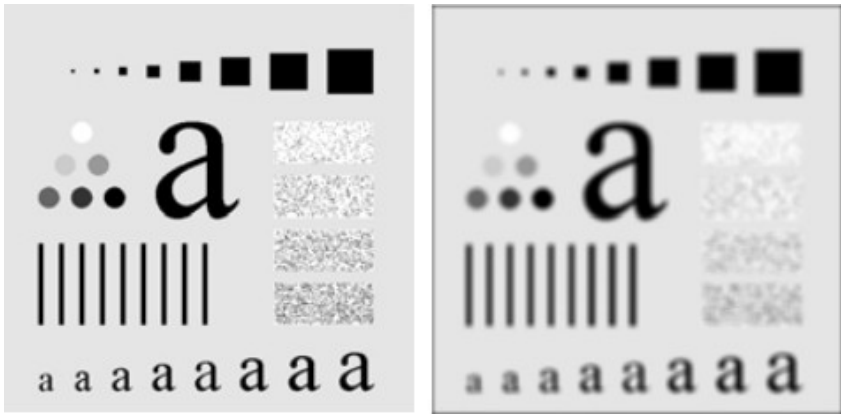


Figura 7: Aplicação de um filtro de média aritmética de 9x9.

Extraído de [6]

Outro filtro de suavização utilizado é o filtro de mediana. Diferentemente do filtro de média que é um filtro linear, esse filtro é não linear. A ideia por trás desse filtro é ler uma janela de *pixels* da imagem original, ordená-los num vetor, por ordem (crescente ou decrescente) e utilizar o valor do meio, conforme pode se ver na Figura 8. Neste exemplo, o valor do novo *pixel* será **4** pois é o valor do meio do vetor ordenado.

$$\begin{bmatrix} 9 & 4 & 4 \\ 5 & 3 & 2 \\ 4 & 4 & 5 \end{bmatrix} \rightarrow (2,3,4,4,4,4,5,5,9)$$

Figura 8: Aplicação do filtro de mediana numa secção da imagem.

O filtro de mediana é melhor utilizado quando a imagem possui *pixels* cujos níveis de cinza são *outliers*, ou seja, não pertencem ao valor médio dos *pixels* da vizinhança. Um exemplo clássico é o ruído sal-e-pimenta (conhecido como ruído *salt and pepper* por causa da semelhança entre o ruído e a mistura de sal com pimenta).

2.3.4 Aguçamento

Os filtros de aguçamento são filtros baseados em derivadas [6]. Eles tem como propriedade principal realçar as transições entre os *pixels* de uma imagem. Dois filtros de aguçamento são mostrados a seguir:

$$W_1 = \begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix} \quad W_2 = \begin{bmatrix} -1 & -1 & -1 \\ -1 & 9 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

A ideia básica por trás destes filtros é melhorar detalhes finos da imagem. As derivadas tem uma característica interessante que é mostrar variações nas funções e, portanto, podem revelar mudanças nos níveis de cinza. Para uma demonstração de como esses filtros podem ser obtidos ver o Apêndice A.

A Figura 9 demonstra o uso do filtro W_1. Podemos observar que *(b)* apresenta linhas e contornos mais bem definidos que a *(a)*. Esses filtros são bastante parecidos com dois filtros que serão mostrados na sessão seguinte. A diferença básica entre eles é que estes filtros servem para realçar enquanto que os filtros da próxima sessão servirão para detectar bordas.

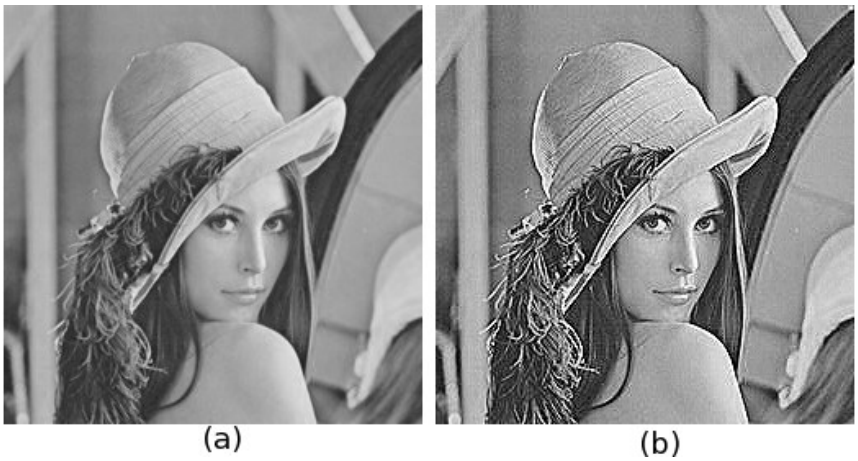


Figura 9: Uso do filtro de aguçamento.

(a) imagem original, (b) imagem após aplicação do filtro W_1

2.3.5 Filtros de detecção de borda e linha

Como mencionado na sessão anterior, os filtros de detecção de borda e linha são bem parecidos com os de aguçamento. São também baseados em derivadas [6] e tem como principal função realçar as bordas da imagem. Os filtros a seguir representam, respectivamente, filtros *Sobel* para linhas horizontais e verticais e o filtro Laplaciano, mencionado anteriormente:

$$W_{S1} = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \quad W_{S2} = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} \quad W_{Lap} = \begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

A Figura 10 ilustra a aplicação dos filtros acima mencionados a mesma imagem. Em *(a)* vê-se a imagem original. *(b)* é a aplicação do filtro W_{S1} à imagem original. Em

(c) tem-se a aplicação do filtro W_{S2} e em *(d)* o filtro W_{Lap}.

Figura 10: Filtros de detecção de borda. (a) representa a imagem original. Em (b) filtro de Sobel horizontal, em (c) filtro de Sobel vertical e em (d) filtro laplaciano.

O filtro horizontal enfatiza linhas horizontais enquanto que o filtro vertical enfatiza linhas verticais. Observando as figuras *(b)* e *(c)* pode-se perceber que algumas linhas estão mais bem marcadas em uma delas do que na outra e vice-versa (observar, por exemplo, o rosto da mulher). Já na figura *(d)* as linhas são mais finas e marcam o contorno tanto horizontal como vertical da imagem.

2.4 Algoritmos de processamento de imagens

Nas sessões anteriores foi visto um conjunto de ferramentas que auxiliam o processamento de imagens. Tais ferramentas visam preprocessar a imagem para que os algoritmos de processamento possam ter uma taxa maior de acerto. Os algoritmos apresentados aqui já são a etapa posterior ao preprocessamento.

2.4.2 Limiarização baseada na variância

A limiarização baseada na variância utiliza técnicas estatísticas para selecionar limiares de acordo com os níveis de cinza. A ideia básica por trás dessa técnica é escolher um ou mais limiares que melhor separem os níveis de cinza entre as classes, ou

seja, que torne a variância máxima. Um caso especial da limiarização estatística é conhecido como método de *Otsu* [8].

O método de *Otsu* seleciona um limiar baseado no histograma da imagem. A partir desse histograma é gerada uma nova imagem com apenas duas cores: branco e preto. Podemos ver a ideia do processo na Figura 11.

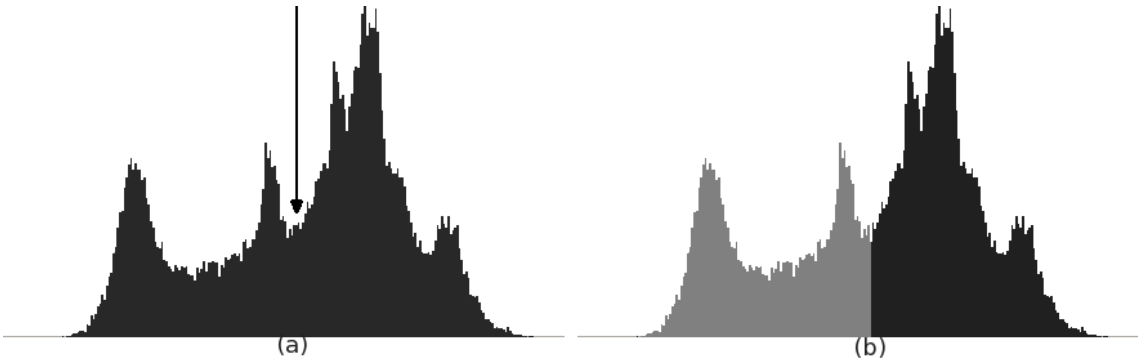


Figura 11: Exemplo de limiarização estatística.

A seta em *(a)* representa a região que melhor separa as classes. A partir desse limiar é feita a separação das cores, como representado em *(b)*. Para se calcular o limiar utiliza-se o seguinte algoritmo:

1. Calcular o histograma da imagem;
2. Para cada nível de cinza da imagem, calcular a variância entre eles;
3. Selecionar o nível de cinza que possui maior variância como limiar.

Para mais detalhes este algoritmo se encontra implementado em C++ no anexo Apêndice B.

2.4.3 Limiarização adaptativa

Diferente da limiarização baseada na variância que foi vista na sessão anterior, a limiarização adaptativa escolhe um limiar de separação dos níveis de cinza baseado na variância de uma região da imagem, ou seja, numa janela. Esse algoritmo é conhecido como Algoritmo de *Niblack* [9].

2.4.4 Algoritmo de detecção de objetos

Algoritmos de detecção de objetos normalmente fazem comparações entre um padrão desejado de se encontrar e uma região da imagem onde se procura. Um algoritmo clássico para detecção de objetos é utilizando correlação [6].

A correlação $c(x, y)$, na sua forma mais simples, da máscara $w(x, y)$ com a imagem $f(x, y)$ é dado pela fórmula:

$$c(x,y)=\sum_{s}\sum_{t} f(s,t)\cdot w(x+s,y+t) \tag{5}$$

para toda a região da imagem. A equação (5) tem alguns problemas de mudança de amplitude que podem ser superadas com a utilização do coeficiente de correlação mostrado em (6):

$$\gamma(x,y)=\frac{\sum_{s}\sum_{t}\left[f(s,t)-\bar{f}(s,t)\right]\cdot\left[w(x+s,y+t)-\bar{w}\right]}{\left[\sum_{s}\sum_{t}\left[f(s,t)-\bar{f}(s,t)\right]^2\cdot\sum_{s}\sum_{t}\left[w(x+s,y+t)-\bar{w}\right]^2\right]^2} \tag{6}$$

Este procedimento gera uma matriz com a correlação de uma máscara ***W*** com a imagem *f*. A região onde a correlação for máxima é uma região que possui um objeto bastante parecido com *w*. Apesar da obtenção do coeficiente de correlação normalizado, obter um coeficiente que seja normalizado para mudanças em tamanho e rotação não é fácil [6].

Em 2003, Rufino [1] desenvolveu um algoritmo de detecção de objetos. Este algoritmo é simples de implementar e conseguiu excelentes resultados. A ideia do algoritmo é encontrar um ponto que aproxime o centro de massa do objeto presente na imagem. Para que o algoritmo funcione de modo próprio a imagem já deve estar preprocessada (sem ruídos) e binarizada. O algoritmo em si será discutido no Capítulo 4.

O algoritmo desenvolvido por Rufino apresenta as seguintes vantagens sobre os algoritmos que realizam a detecção por correlação:

- Fácil implementação;
- Rápido para detectar um objeto na imagem;
- Ocupa pouca memória.

O algoritmo, no entanto, possui a desvantagem de apenas detectar um objeto dentro da imagem [1]. Se mais de um objeto estiver presente o algoritmo não será capaz de detectar de modo correto os objetos.

Capítulo 3 – Sistemas embarcados

Neste capítulo serão apresentados aspectos básicos de microcontroladores: organização e arquitetura básica, periféricos e aplicações comuns dos microcontroladores. Será apresentado, também, o microcontrolador utilizado neste trabalho: o LPC1768 manufaturado pela *NXP Semiconductors N.V.* [10] [11].

3.1 Microcontroladores

Microcontrolador é um componente que é, basicamente, um computador em um *chip* [12]. Dentro do encapsulamento tem-se a *CPU* (*Central Processing Unity* – do inglês, Unidade Central de Processamento), memória de programa, memória *RAM* (*Random Access Memory* – do inglês Memória de Acesso Aleatório), sistemas de entrada e saída e outros periféricos que serão mencionados mais a frente.

3.1.1 Características básicas

Existem microcontroladores de tamanhos variados, com tamanhos de memória de programa e memória *RAM* diferentes e vários tipos de periféricos. Tais características podem tornar um microcontrolador mais apropriado do que outro para uma dada aplicação. Na Tabela 1 mostra-se uma breve comparação entre três tipos de microcontroladores: um ATMEL®, um PIC® e um ARM®.

Tabela 1: Comparação entre microcontroladores.

	Atmel® AT90S1200 [12], [13]	Microchip® PIC18F4550 [14]	Phillips® LPC1768 ARM Cortex-M3 [10]
Memória de Programa	1KB *Flash* + 64B *EEPROM*	32KB *Flash* + 256B *EEPROM*	512KB *Flash*
Memória *RAM*	128 *bits*	2KB + 1BK (*USB*)	32KB + 32KB (*USB*, *Ethernet*, *DMA*)
Número de pinos	20	40/44	100
Clock	Até 16MHz	Até 48MHz	Até 100MHz
Portas de I/O	15	35	70
Periféricos	*SPI*, *Timer* 8 *bits*, Comp. Analógico, *WDT*	*USB*, *USART*, *SPI*, *PWM*, *Timer*, *A/D*, *WDT*	*Ethernet*, *UART*, *SPI*, *CAN*, *I²C*, *A/D*, *D/A*, *PWM*, *RTC*, outros
Preço	Não encontrado	*US$6,80*	*US$11,40*

A Tabela 1 mostra um comparativo simples entre três tipos de microcontroladores. Para uma determinada aplicação um deles pode ser mais adequado do que os outros. A adequação pode ser em termos de preço, complexidade de programação, tamanho físico, gasto de energia e outros. Deve-se fazer um projeto das funções pretendidas no *hardware*, quantos pinos de *I/O* serão utilizados, tamanho do programa que será escrito, entre outras características, antes de se escolher um microcontrolador para utilizar [12].

Além de preço, tamanho e *hardware* interno diferentes, existe outra característica básica que deve ser observada: a arquitetura interna. Existem dois tipos de arquitetura de microcontroladores no mercado: *CISC* (do inglês, Conjunto de Instruções Complexas) e *RISC* (do inglês, Conjunto Reduzido de Instruções). As características básicas de microcontroladores (e de computadores) *CISC* e *RISC* é mostrada na Tabela 2 [12].

Tabela 2:Comparação entre CISC e RISC.

	CISC	*RISC*
Quantidade de instruções	Possui um número maior de instruções	Possui um número menor de instruções
Ciclos de máquina para executar uma instrução	Instruções levam quantidades diferentes de ciclos de máquina (de um ciclo a vários)	Instruções normalmente levam a mesma quantidade de ciclos (que normalmente é um ciclo de máquina)
Tipo de barramento	Barramento único ligando memória *RAM* e memória de programa (Arquitetura de *Von Neumann*)	Barramentos separados para memória *RAM* e memória de programa (normalmente utiliza-se a Arquitetura *Harvard*)

3.2 Microcontrolador LPC1768

O LPC1768 é um microcontrolador da família LPC17xx manufaturado pela *NXP Semiconductors N. V.*, uma antiga divisão da *Phillips*®. A versão utilizada possui um processador *RISC* baseado no *ARM Cortex-M3* operando em 100MHz de *clock*. Como é de arquitetura *RISC* possui barramentos distintos para instruções, dados e um barramento especial para periféricos. A *CPU* também possui uma unidade de *prefetching* (lit. pré busca, ou seja, lê a instrução e decodifica previamente) para instruções que são utilizadas para fazer desvio especulativo [10]. A Figura 12 ilustra o diagrama de blocos do LPC1768 e seus periféricos.

Apesar do grande número de periféricos existentes no microcontrolador, neste projeto foi utilizado apenas alguns deles. Para programação e *debug* foi utilizado a interface *USB* que vem nativa na plataforma *MBED*®. Ao ligar a *MBED*® no

computador, a interface *USB* é reconhecida pelo sistema operacional como uma unidade de disco removível e uma porta serial que é utilizada para *debug*. Pode-se usar um programa leitor de portas seriais como o *Minicom* do *Linux* ou o *Putty* do *Windows*.

A *MBED*® pode ser facilmente programada "arrastando" o arquivo binário compilado para dentro do *drive* virtual criado. A porta serial utilizada para *debug* também foi utilizada como entrada de dados para o microcontrolador.

3.2.1 Instruções MAC

Instruções *MAC* são um conjunto de instruções que foram incorporadas em algumas famílias de microcontroladores, em especial os *DSP*'s. *MAC* é uma sigla para *Multiply and Accumulate* que em português é "multiplicar e acumular". A Tabela 3 ilustra as instruções que a família LPC17xx possui [11].

Tabela 3:Instruções MAC[1].

Mnemônico	*Instrução*	*Sintaxe*	Execução
MLA	*Multiply and Accumulate*, 32 *bits*	MLA *Rd, Rn, Rm, Ra*	$Rd \leftarrow Ra + (Rn \times Rm)$
MLS	*Multiply and Subtract*, 32 *bits*	MLA *Rd, Rn, Rm, Ra*	$Rd \leftarrow Ra - (Rn \times Rm)$
SMLAL	*Signed Multiply with Accumulate*, 64 *bits*	SMLAL *RdLo, RdHi, Rn, Rm*	$(RdHi, RdLo) \leftarrow (RdHi, RdLo) + (Rn \times Rm)$
UMLAL	*Unsigned Multiply with Accumulate*, 64 *bits*	UMLAL *RdLo, RdHi, Rn, Rm*	$(RdHi, RdLo) \leftarrow (RdHi, RdLo) + (Rn \times Rm)$

Estas instruções são bastante úteis para processamento de sinais pois a amostragem de sinais, cálculo de séries de Fourier ou a transformada Z são calculadas multiplicando e acumulando um certo valor. Para este trabalho as instruções *MAC* serão bastante úteis pois a operação de convolução de matrizes é essencialmente uma série de somas com produtos.

1 *Rd, Ra, Rn, Rm, RdLo, RdHi* são as representações para, respectivamente, registrador de destino, registrador de acumulação, registrador "n", registrador "m" (ou seja, dois registradores quaisquer), registrador de destino *lower (*ou seja, os 32 *bits* de mais baixa ordem) e registrador de destino *higher*.

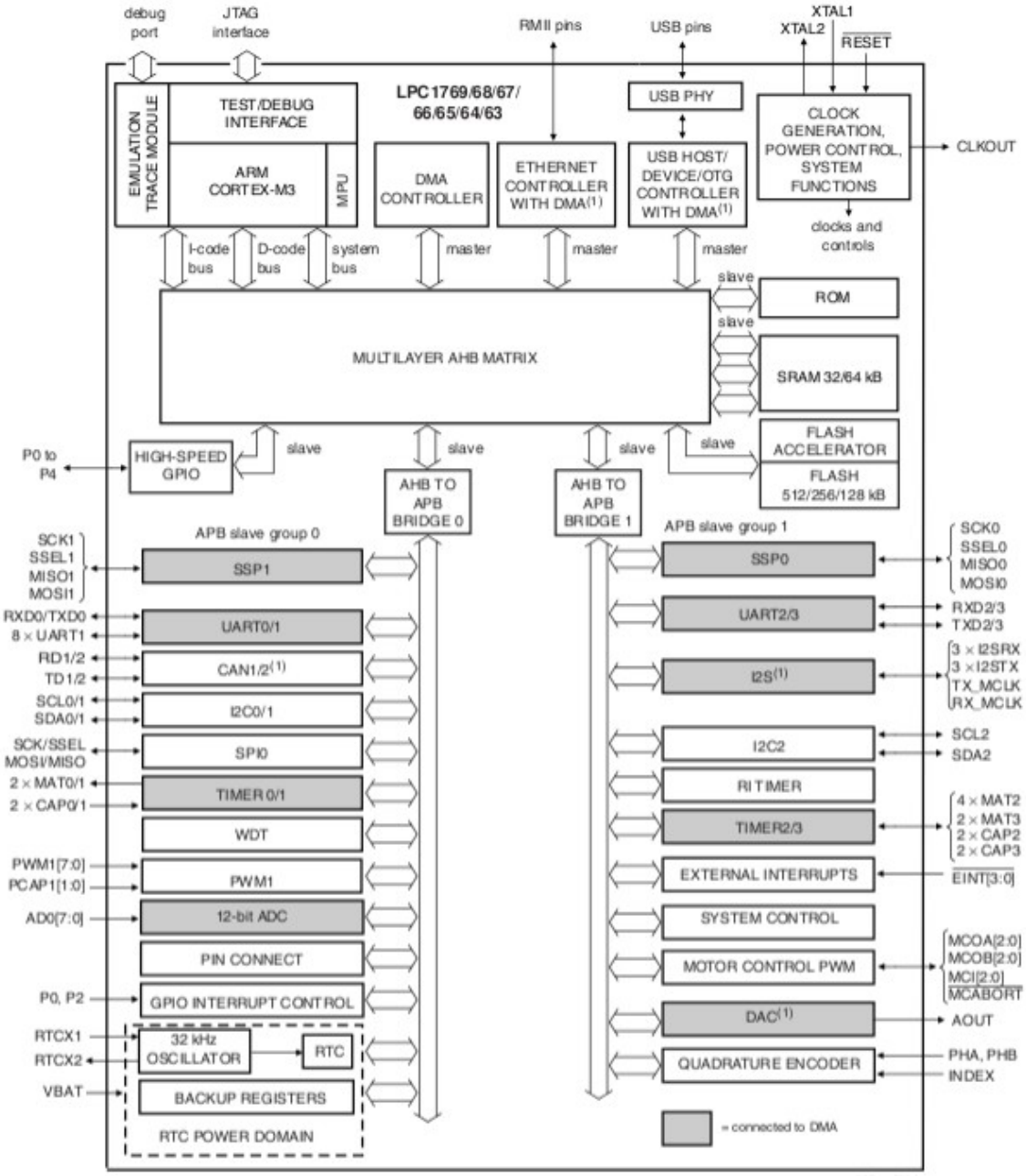


Figura 12: Diagrama de blocos do LPC1768.

Extraído de [10]

Capítulo 4 – Processamento embarcado de imagens

Neste capítulo será apresentado o desenvolvimento do trabalho. Será mostrado como foi realizado a aquisição, pré-processamento, detecção de objetos. Será apresentada as dificuldades encontradas durante o desenvolvimento do projeto assim como as soluções implementadas. O código-fonte das soluções pode ser consultado no Apêndice B.

4.1 Método de aquisição de imagens

Como foi discutido no Capítulo 2, existem várias maneiras de se adquirir imagens. Um dos métodos mais comuns é por meio de câmeras. Uma câmera é um dispositivo sensível a intensidade e cor da luz que a captura. Normalmente uma câmera é construída com um dispositivo chamado *CCD* (*charge-coupled device*, do inglês dispositivo de carga acoplada). Na Figura 13 podemos ver um exemplo de *CCD*.

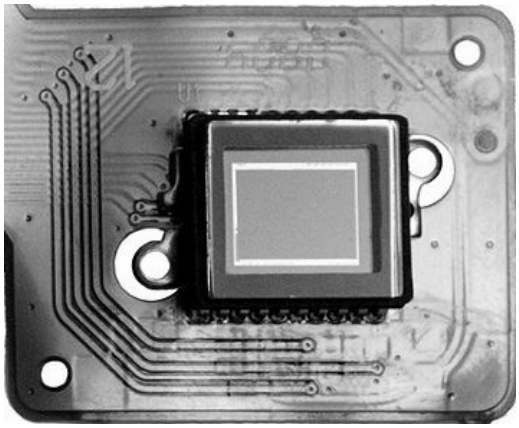


Figura 13: Exemplo de CCD.

Uma câmera normalmente é composta de uma lente e um *CCD* que é formado por uma matriz de capacitores. A lente da câmera projeta a imagem sobre o *CCD* fazendo assim com que cada capacitor da matriz armazene uma quantidade de carga elétrica proporcional a intensidade da luz que está presente naquele segmento discreto da imagem [1], [15].

No mercado existe uma grande quantidade de dispositivos de captura de imagem, entre eles as câmeras digitais e *webcam*'s. Existem também algumas câmeras especiais para dispositivos embarcados, que possuem conexão serial ou *SPI* para transmissão de dados. Tais câmeras possuem a habilidade de transmitir imagens de vários formatos e resoluções diferentes. Um exemplo dessas câmeras é a *LinkSprite® LS-Y201*. A seguir mostram-se algumas características da câmera [16]:

- Captura de imagem em *VGA* (640x480), *QVGA* (320x240) ou 160x120

- Transmissão de imagem em formato *JPEG* ou *RAW* (imagem pura)
- Taxa de transmissão padrão serial: 38400
- Alimentação DC 3.3V ou 5.0V consumindo de 80mA a 100mA de corrente
- Largura: 32mm x Altura: 32mm

A configuração e captura das imagens é realizada trocando-se mensagens com a câmera. Infelizmente não foi possível adquirir uma câmera serial em tempo hábil para o projeto. Foi feita a opção de se utilizar uma *webcam* ligada a um computador com um *software* que simulará a câmera serial e enviará a imagem para o controlador como demonstrado na Figura 14.

Já que o microcontrolador escolhido pode funcionar como *host* de dispositivos *USB*, a *webcam* poderia ser ligada diretamente nele dispensando totalmente o uso do computador. Porém esta funcionalidade não foi implementada por aumentar consideravelmente a complexidade do projeto pois seria necessária a implementação do *driver* da câmera *USB* e, provavelmente, a adição de componentes extras no sistema tais como memórias *RAM* externas. Tal implementação foge aos propósitos deste trabalho.

O código-fonte do *software* construído para simular a câmera serial encontra-se disponível no Apêndice B. Como será visto mais adiante o *software* do controlador foi escrito já prevendo a aquisição de imagens de vários dispositivos diferentes, além de câmeras seriais ou *SPI*. Será mostrado que a imagem poderá vir de praticamente qualquer fonte.

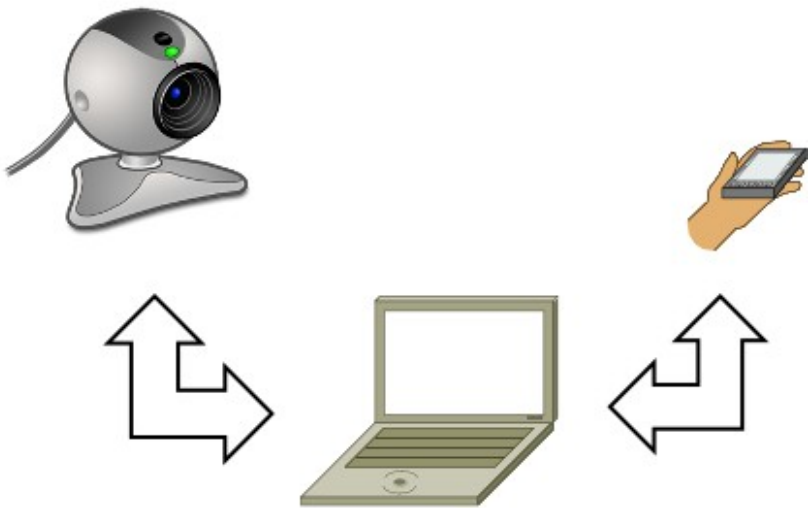


Figura 14: Diagrama do sistema desenvolvido.

Como mencionado anteriormente, a captura da imagem está sendo realizada por uma *webcam* e transmitida de modo serial para o microcontrolador. Para simular de modo mais real o *software* recebe mensagens de configuração oriundas do

microcontrolador e, quando necessário, uma mensagem de captura de quadro é enviada fazendo com que o *software* de simulação da câmera envie uma imagem para o microcontrolador.

Como visto em [16], a menor imagem possível é 160x120 *pixels*. Normalmente esta é a menor imagem possível de se capturar com câmeras comerciais. O microcontrolador utilizado possui memória suficiente para ler uma imagem com 256 níveis de cinza o que ocuparia 18,75*KB* de memória *RAM*. Apesar de ainda restar 13,25*KB* de memória para processamento e o restante das operações não há espaço suficiente para mais de uma imagem na memória ao mesmo tempo. Isto pode se tornar um problema pois quando uma imagem é filtrada é necessário a criação de uma imagem destino, como visto no Capítulo 2.

Para contornar este problema foi realizado um redimensionamento durante o processo de transmissão da imagem de 160x120 *pixels* para 113x84 *pixels* tornando possível ter duas imagens de 256 níveis de cinza na memória e ainda assim ter memória disponível para outros processamentos. Deste modo temos no máximo 18,54*KB* de memória ocupada com imagens e 13,46*KB* de memória livre para outros processamentos. A imagem mesmo redimensionada para 113x84 ainda é suficiente para o que se deseja: detectar um objeto na área de trabalho.

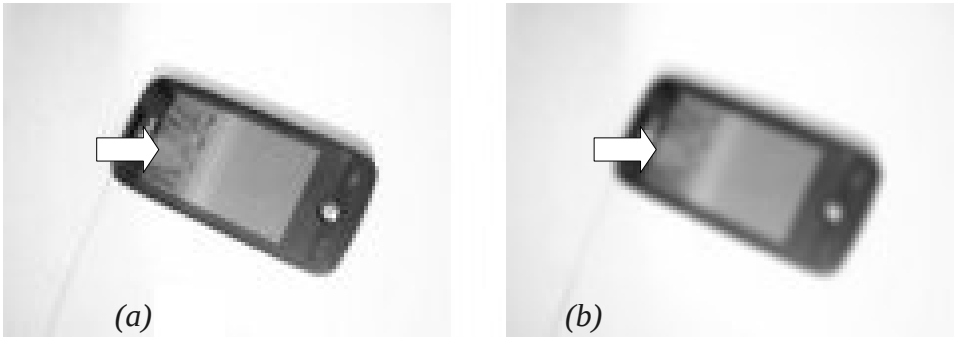
O tempo de processamento é rápido. O tempo de pré-processamento, e processamento é da ordem de algumas dezenas de milissegundos. A demora maior é devida à transmissão da imagem pela porta serial. O problema da demora da transmissão pode ser resolvido utilizando-se uma câmera que transmita mais rapidamente do que a taxa máxima de 115200 *baud/s* da porta serial.

4.2 Pré-processamento

Algumas aplicações realizam pré-processamento da imagem para remover ruídos, por exemplo, processamento da imagem para extrair alguma característica e depois é realizado um pós-processamento para classificação [6]. Neste trabalho só foi utilizada a etapa de pré-processamento e, depois, o processamento propriamente dito, que consiste em detectar um objeto na imagem.

O pré-processamento realizado pelo microcontrolador consiste em uma sequência de algoritmos para reduzir possíveis ruídos, binarização da imagem e, por fim, detecção de bordas para facilitar a detecção.

Para remover os ruídos é utilizado um filtro de média aritmética. Uma característica deste filtro, que foi discutido no Capítulo 2, é que ele reduz os ruídos borrando a imagem. Na Figura 15 mostra-se a imagem original, já redimensionada para 113x84 *pixels* e o efeito do filtro de média aritmética numa imagem capturada pela câmera utilizada no desenvolvimento do projeto.

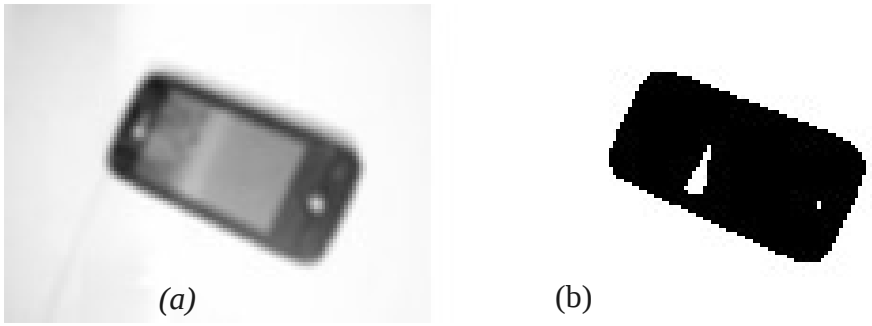


(a) *(b)*

Figura 15: Borramento causado no objeto em (a) por filtro de média em (b).

Ao se passar o filtro de média aritmética a imagem é borrada, como foi discutido na sessão 2.3.3. Como pode-se observar em *(a)* existem pequenos pontos ruidosos que deixaram de existir em *(b)*. Estes pontos de ruído e pequenos detalhes podem ser prejudiciais na hora de detectar bordas utilizando o Laplaciano, também mencionado anteriormente.

Depois da etapa de eliminação de ruídos a imagem será binarizada. O algoritmo de binarização escolhido é o método de *Otsu* por não precisar de parâmetros para escolha do limiar. A Figura 16 demonstra o próximo passo da fase de pré-processamento.



(a) (b)

Figura 16: Imagem binarizada a partir de (a) imagem borrada em (b).

A binarização é utilizada para auxiliar o algoritmo de busca que será utilizado na etapa de processamento. O método de *Otsu* possui algumas desvantagens pois o limiar selecionado é global [8] e leva em conta os níveis de cinza das sombras e podem aparecer manchas como demonstrado na Figura 17.

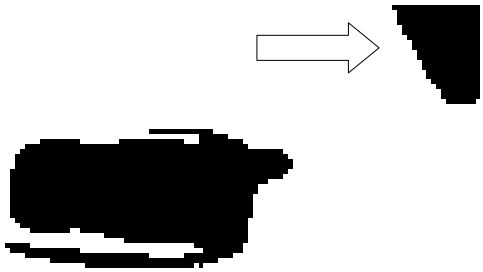


Figura 17: Imagem binarizada com método de Otsu.

A seta demonstra o efeito mencionado. A mancha apareceu devido ao limiar escolhido estar num nível de cinza menor que o nível de cinza mínimo da mancha portanto ele foi escolhido para permanecer na imagem binarizada.

O próximo passo do algoritmo de pré-processamento é a detecção das bordas da imagem utilizando o filtro Laplaciano. Como é característico do filtro Laplaciano a imagem resultante deverá ter as bordas marcadas com a cor branca enquanto que o restante da imagem estará preta. Para se manter o fundo branco e o objeto com a cor preta todos os *pixels* tiveram seus valores invertidos, ou seja, quem é branco torna-se preto e vice-versa. A Figura 18 demostra a etapa final do processo de pré-processamento. Após este último passo a imagem está pronta para ser utilizada na etapa de processamento.

(a) *(b)*

Figura 18: Detecção das bordas da imagem a partir de (a) imagem binarizada resultado em (b).

4.3 Algoritmo de processamento e detecção de objetos

O algoritmo procura por regiões na imagem onde há a mudança do nível de cinza da cor que representa o fundo da imagem (normalmente 255 ou o branco) do nível de cinza que representa os objetos (normalmente 0 ou preto).

A primeira parte do algoritmo consiste em encontrar as coordenadas espaciais que definem as fronteiras superior esquerda e inferior direita do objeto, conforme podemos ver na Figura 19.

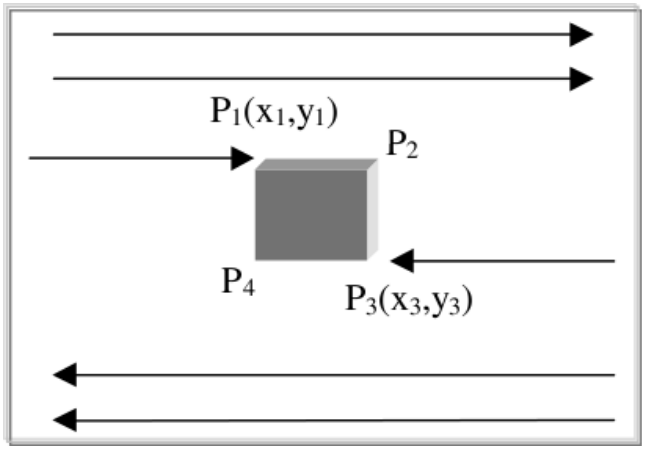


Figura 19: Primeira parte do algoritmo.

Extraído de [1]

A partir de *P1(x1, y1)* e *P3(x3, y3)* é calculado a primeira estimativa da coordenada central da imagem a partir dos valores obtidos utilizando as seguintes fórmulas:

$$x' = \frac{x_1 + x_3}{2} \quad y' = \frac{y_1 + y_3}{2} \tag{7}$$

O resultado da primeira estimativa para o quadrado pode ser visto na Figura 20.

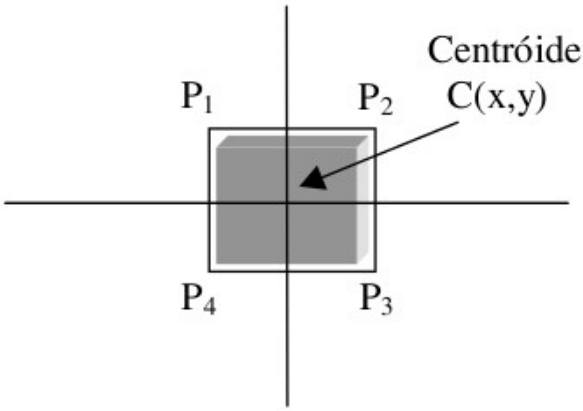


Figura 20: Detecção de um quadrado utilizando a primeira parte do algoritmo

Extraído de [1]

Durante os testes realizados por Rufino essas coordenadas *x'* e *y'* não foram suficientes para se detectar alguns tipos de objetos, entre eles objetos com formas triangulares [1]. Nestes objetos a estimativa de coordenada central ficara sobre um dos lados e não no interior do objeto como mostra a Figura 21.

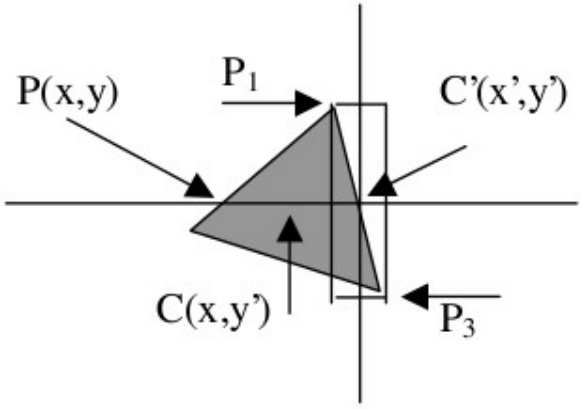


Figura 21: Detecção errada de um objeto triangular com a primeira etapa do algoritmo

Extraído de [1]

Com as coordenadas *x'* e *y'* corrige-se a coordenada *x'* procurando um novo valor, denotado de *x*, que pertença ao interior do objeto que foi detectado. Isto é feito procurando a borda direita e esquerda do objeto utilizando como linha guia da coordenada *x* a coordenada *y'*, como pode-se ver na Figura 22. O algoritmo segue:

1. Calcular *P1* varrendo a imagem da esquerda para a direita, de cima para baixo;

2. Calcular *P3* varrendo a imagem da direita para a esquerda, de baixo para cima;
3. De posse de *P1* e *P3* calcular *x'* e *y';*
4. Calcular *x* utilizando como base *x'* e *y'* de modo que *x* pertença ao interior do objeto em questão e adotar *y = y'*.

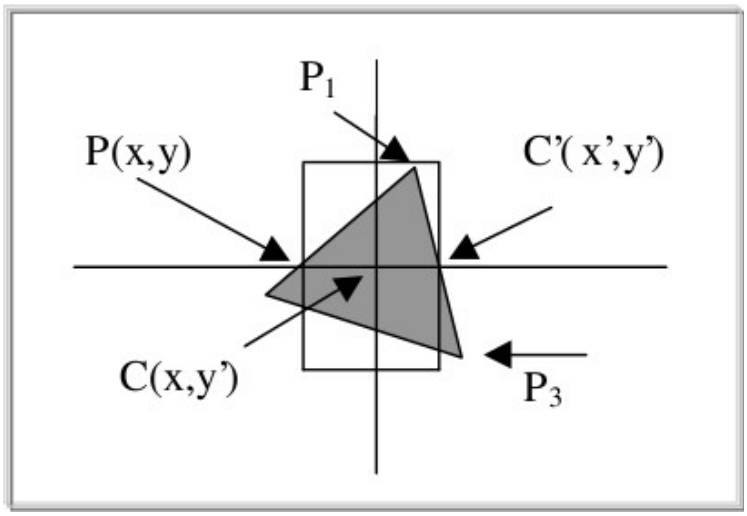


Figura 22: Detecção de um objeto triangular utilizando a segunda parte do algoritmo

Extraído de [1]

A implementação do algoritmo encontra-se no anexo Apêndice B. Como pode-se observar o algoritmo consome apenas alguns *bytes* de memória, pois só necessita armazenar os valores de *x* e de *y* e algumas variáveis temporárias. Outra vantagem deste algoritmo é que ele não é sensível ao tamanho do objeto sendo assim a busca é efetuada apenas uma vez, diferente de alguns algoritmos de detecção de objetos que utilizam máscaras de tamanho fixo, logo se o objeto muda de tamanho ou sofre um pequeno giro a máscara torna-se inadequada, sendo necessária a utilização de uma quantidade grande de máscaras para se detectar apenas um único objeto na imagem.

A desvantagem do algoritmo é que ele pode ser utilizado para encontrar apenas um objeto na área de trabalho.

4.4 Soluções implementadas

4.4.1 Software simulador da câmera

A primeira etapa construída da solução foi o *software* simulador da câmera serial. Como mencionado anteriormente, a *MBED*® uma vez ligada no computador é mapeada como uma porta serial. O *software* simulador da câmera faz uso dessa porta serial para transmitir os dados.

A imagem é, inicialmente, capturada de uma *webcam* qualquer ligada ao computador. A biblioteca utilizada para captura de imagens pela *webcam* foi a biblioteca *opensource OpenCV (Open Source Computer Vision*) na versão 2.2 [17]. Ela é um conjunto de ferramentas para visão computacional em tempo real que pode ser utilizada para fins acadêmicos e comerciais. A biblioteca possui versões para *Windows* e *Unix* (*GNU/Linux* e outros) e tem suporte as linguagens C, C++ e *Python* [17]. A versão utilizada executa em *GNU/Linux*. Apesar de ser uma biblioteca com grandes recursos foi utilizada apenas as funções de captura de imagem de câmeras e de leitura de imagens.

O *software* de simulação da câmera serial, chamado de *Serial Camera Emulator* (*SCE*), foi construído na linguagem C++ e simula a troca de mensagens entre o microcontrolador e uma câmera serial. O *software* implementa uma máquina de estados que lê periodicamente a saída serial da *MBED*® a procura de mensagens de configuração. Uma vez que chega uma mensagem conhecida, o *SCE* interpreta a mensagem e responde a requisição enviando uma mensagem de *ACK* (A*cknowledged*, do inglês reconhecido) ou *NACK* (sigla para *Not Acknowledged*, do inglês não reconhecido). A Tabela 4 mostra as mensagens que são trocadas com o *software* de simulação da câmera.

Tabela 4: Mensagens de troca com o software SCE.

Nome da mensagem	Código da mensagem	Sintaxe	Valores possíveis	Efeito
Set width	*0x0101WWWW*	*WWWW* – Largura da imagem em pixels	160, 320 e 640	Seleciona o comprimento da imagem
Set height	*0x0102HHHH*	*HHHH* – Altura da imagem em pixels	120, 240 e 480	Seleciona a altura da imagem
Capture Frame	*0x0103XXXX*	*XXXX – Don't care*	Não utilizado	Captura um quadro e envia pela porta serial
ACK	*0x7FFFFFFF*	-	-	Indica que o comando foi compreendido
NACK	*0xFFFFFFFF*	-	-	Indica que o comando não foi compreendido

Apesar da possibilidade de captura em vários tamanhos, o *software* de simulação apenas implementa a captura na resolução de 113x84 *pixels* para simplificar a implementação do *software* do microcontrolador e agilizar o processo de envio da

imagem pois a porta serial a uma taxa de 115200 *baud/s* leva aproximadamente 660 milissegundos para enviar um quadro de 113x84 *pixels* enquanto que leva 1,33 segundo para enviar um quadro de 160x120 *pixels* e, no pior caso, aproximadamente 21,33 segundos para o quadro de 640x480. A Figura 23 demonstra o uso do *software* de simulação. O *software* não apresenta interface gráfica, apenas os comandos sendo exibidos num terminal *Linux*.

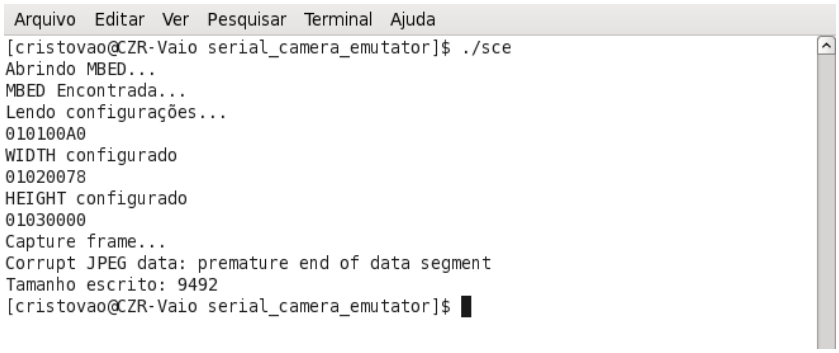


Figura 23: Exemplo da execução do software de simulação da câmera digital.

4.4.2 Software da MBED®

O *software* da *MBED®* foi construído utilizando um compilador *on-line* disponibilizado pela *NXP Semiconductors*. Foi utilizada a linguagem C++ e a biblioteca padrão disponibilizada pela *NXP Semiconductors* [18]. Esta biblioteca é composta de uma série de classes que encapsula o funcionamento básico dos periféricos do *LPC1768*. A biblioteca padrão (chamada de *mbed*) possui documentação *on-line* para auxiliar o desenvolvimento dos *softwares* assim como vários exemplos, aplicações e bibliotecas publicadas por usuários [18].

O compilador *on-line* utiliza uma interface *AJAX*, e compila através de um compilador compatível com o padrão *ARM RVDS 4.1* [18][19] e possui as seguintes funcionalidades entre outras [18]:

- Edição de código-fonte com destaque de sintaxe
- Múltiplos programas abertos
- Suporte a várias arquiteturas de microcontroladores LPC1768
- Informações sobre a compilação, inclusive utilização de memória *RAM* e tamanho do programa binário

A biblioteca *mbed* possui classes para acesso a pinos de *I/O*, para escrita em pinos com capacidade para *PWM*, leitura e escrita de conversores A/D e D/A, portas seriais, barramentos *SPI*, *I²C* e *CAN*, *Ethernet*, controle de *timers* (os *timers* comuns que contam tempo e *tickers* que executam uma determinada função repetidamente num intervalo de tempo pré-determinado) e controle de interrupções [18].

De todas as funcionalidades da biblioteca *mbed* foi utilizada as classes *Serial* e *DigitalOut* que são, respectivamente, a classe utilizada para controle de portas seriais e a classe utilizada para controle de pinos com saída digital. A Figura 25 representa o diagrama de classes do *software* implementado.

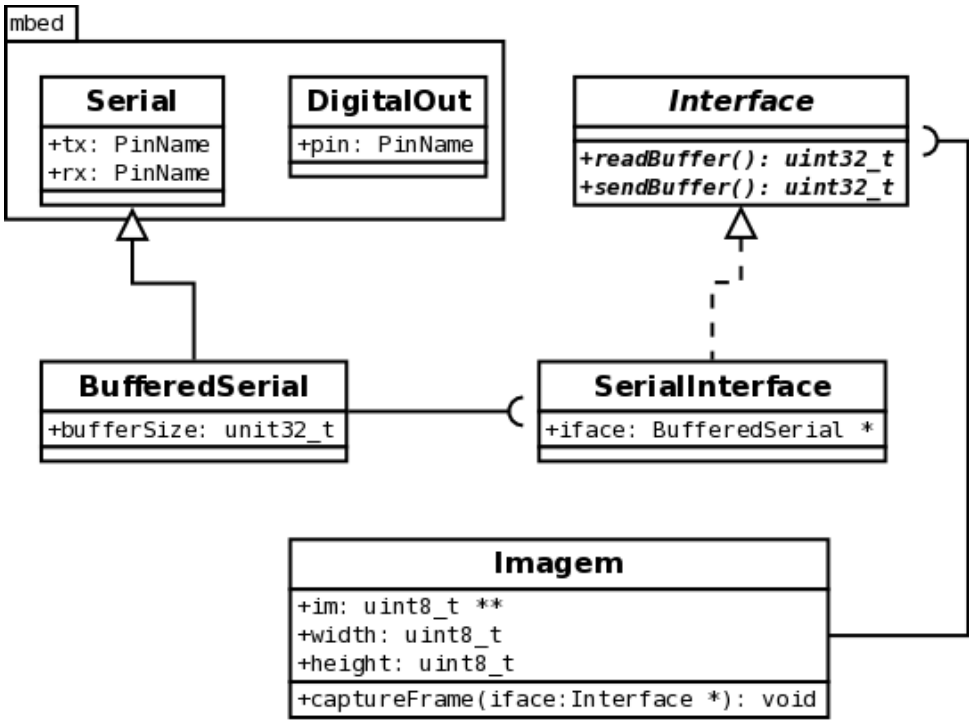


Figura 24: Diagrama de classes do projeto.

Uma das dificuldades encontradas foi a utilização da classe *Serial*. Ela controla uma das muitas saídas seriais da *MBED®*. A Figura 25 demonstra um trecho do código-fonte de uma aplicação que lê dados da porta serial e escreve novamente:

```
#include <mbed.h>

Serial pc (USBTX, USBRX);

int main() {
        pc.printf("Hello World!");
        while (true)
                pc.putc(pc.getc() + 1);
}
```

Figura 25: Exemplo de código-fonte.

No exemplo acima, é criada uma porta serial chamada de "*pc*" que utiliza como pinos de transmissão e recepção os pinos da porta *USB*. A função *putc* e a função *getc*, respectivamente, escreve e lê um caractere (8 *bits*) na porta serial. Existem outras funções na classe tais como funções para selecionar a taxa de transferência da porta

serial, formato de transmissão (como quantidade de *bits* transmitidos por vez, *bit* de paridade e número de *bits* de parada) e funções de captura de interrupção. Para uma lista completa pode-se consultar a documentação *on-line* disponível em [18].

A classe *Serial* teve um bom funcionamento nos testes iniciais. Quando foi iniciada a transmissão de uma grande quantidade de dados, como uma imagem, os códigos simples de leitura, como o da Figura 25, não foram satisfatórios pois houve uma grande perda de dados. Foi necessária a implementação de uma classe serial que possuísse o conceito de *buffer* pois a porta serial possui apenas como *buffer* uma memória de 16 *bits* [10].

A classe implementada para resolver o problema de perda de dados foi chamada de *BufferedSerial*. Ela é uma classe que herda as funcionalidades básicas da classe *Serial* e possui os seguintes protótipos apresentados na Figura 26.

```
class BufferedSerial : public Serial {

    public:
        BufferedSerial ( size_t bufferSize, PinName tx, PinName rx );
        virtual ~BufferedSerial ();

        int getc ();
        int readable ();
        void setTimeout (float seconds);
        size_t readBytes ( uint8_t *buffer, size_t requested );
    private:
        void handleInterrupt ();

        uint8_t     *mem_buff;
        uint16_t    mem_contentStart;
        uint16_t    mem_contentEnd;
        uint16_t    mem_buffSize;
        float       timeout;
        Timer       mem_timer;

};
```

Figura 26: Protótipos da classe BufferedSerial.

A classe *BufferedSerial* sobrescreveu alguns métodos da classe *Serial* original o que permitiu um melhor aproveitamento das funcionalidades da classe original. Foi criado um *buffer* circular de tamanho variável (representado pela variável *uint8_t *mem_buff* e os ponteiros de início e fim de *buffer* *mem_contentStart* e *mem_contentEnd*). A classe também conta com um *timer* para estouro de tempo de leitura dos dados do *buffer* circular.

Foi adicionado a função *readBytes* que lê uma quantidade máxima de *bytes* do *buffer*. Caso a função leve mais do que o tempo máximo para leitura dos dados (que é definido pela função *setTimeOut*) a função retorna o caractere de fim de linha. Para o código-fonte completo da classe *BufferedSerial* ver o Apêndice B.

Foi implementada uma classe chamada *Imagem* que possui todos os métodos para tratamento e captura de imagens pelo microcontrolador. Esta classe possui métodos para realizar todo o procedimento de pré-processamento descrito na sessão 4.2 e para ser utilizada pelo algoritmo descrito na sessão 4.3. A Figura 27 representa os protótipos

das funções da classe *Imagem* e os códigos-fonte estão disponíveis no Apêndice B:

```
class Imagem {

    public:

        UINT8 **im;
        UINT8 width, height;

        Imagem (UINT8 wid, UINT8 hei);
        ~Imagem (void);

        UINT8 captureFrame (Interface *iface);
        UINT8 calculateThreshold (void);
        void calculateHistogram (UINT16 *histogram);
        void filter (Imagem *dst, float filter[][3]);
        void otsu (void);
};
```

Figura 27: Protótipos da classe Imagem.

Para padronizar a comunicação de todos os periféricos do projeto e prevendo a utilização de diferentes dispositivos para comunicação e obtenção de dados foi implementada uma classe abstrata chamada *Interface*. Esta classe possui dois métodos abstratos: *sendBuffer* e *readBuffer*. A partir da classe *Interface* pode-se construir outras classes para utilização de outros periféricos para leitura, como o barramento *SPI* ou a própria interface *USB*.

Para a comunicação serial foi implementada uma classe chamada *SerialInterface* que herda da classe abstrata *Interface* e que utiliza a classe *BufferedSerial*. A Figura 28 representa a classe *Interface* e sua filha a classe *SerialInterface*:

```
class Interface {

    public:
        virtual int sendBuffer (size_t size, void *buffer) = 0 ;
        virtual int readBuffer (size_t size, void *buffer) = 0 ;
};

class SerialInterface : public Interface {

private:
    BufferedSerial *iface;
    float timeout;

public:
    SerialInterface (size_t size, PinName tx, PinName rx, float timeout) ;
    ~SerialInterface (void) ;
    int readBuffer (size_t size, void *buffer) ;
    int sendBuffer (size_t size, void *buffer) ;

};
```

Figura 28: Classe Interface e SerialInterface.

Capítulo 5 – Resultados obtidos

Neste capítulo serão apresentados os resultados obtidos a partir de imagens geradas em computador utilizando uma ferramenta gráfica e com imagens reais capturadas com o sistema desenvolvido.

5.1 Teste do sistema com imagens criadas em editores gráficos de imagens

A primeira imagem apresentada é a imagem de um quadrado preto. O quadrado é a figura geométrica mais simples de se encontrar pelo algoritmo utilizado. A Figura 29 representa a imagem original assim como a imagem tratada pelo sistema.

(a) *(b)*

Figura 29: Detecção de um quadrado em (b) a partir de (a).

Pode-se observar um pequeno ponto preto próximo ao centro do quadrado na Figura 29(b). Esse ponto representa o centroide aproximado percebido pelo algoritmo utilizado. A próxima imagem apresenta um círculo perfeito encontrado pelo algoritmo:

(a) *(b)*

Figura 30: Detecção de um circulo em (b) a partir de (a)

O círculo, apesar de não possuir cantos como o quadrado, também é facilmente detectado pelo algoritmo como pôde-se ver na Figura 30(b). Como foi mostrado as duas imagens puderam ser encontradas pelo algoritmo diretamente. Rufino [1] demonstrou que o algoritmo na sua versão inicial detectava objetos quadrados e circulares sem

muitos problemas. Objetos quadrados funcionam por serem a forma básica com que eles são procurados na imagem. Os objetos circulares quando quantizados possuem um pequeno retângulo o que auxiliava na detecção [1]. Foi observado, no entanto, que triângulos não eram detectados da forma correta por possuírem uma extremidade isolada que confundia o algoritmo localizando o centroide do objeto sobre uma das arestas do triângulo. Rufino realizou uma modificação e o algoritmo passou a detectar o centroide aproximado de objetos triangulares normalmente. A Figura 31 representa um objeto triangular detectado pelo algoritmo implementado no microcontrolador:

(a) *(b)*

Figura 31: Detecção de um objeto triangular em (b) a partir de (a).

Depois de realizado testes com formas geométricas simples, foi realizado um teste com um objeto irregular. Mesmo assim o algoritmo funcionou e detectou o centroide aproximado corretamente. O resultado obtido pode ser visualizado na Figura 32:

(a) (b)

Figura 32: Detecção de um objeto irregular detectado em (b) a partir de (a).

5.2 Teste do sistema com imagens reais capturadas com câmera

Depois de haverem sido realizados testes utilizando imagens geradas com ferramentas gráficas, foram realizados vários testes com imagens capturadas com uma câmera. Uma diferença entre essas imagens é a cor do fundo. Apesar dos testes reais haverem sido realizados com uma área branca, o fundo das imagens não é totalmente branco como nas imagens de teste por causa do ajuste de brilho automático da câmera

utilizada.

Outra diferença que é facilmente perceptível é que as imagens geradas por ferramentas gráficas possuem cor homogênea, ou seja, pouco ruído presente, o que simplifica ou até mesmo pode eliminar a etapa de binarização. Já as imagens capturadas pela câmera não possuem cor homogênea, portanto a etapa de binarização torna-se extremamente necessária, assim como a etapa de remoção de ruídos.

Um aspecto importante da captura de imagens utilizando câmera é a iluminação utilizada. Uma boa iluminação pode tornar o processo de detecção de objetos e outras características mais fácil do que uma má iluminação. Este efeito será demonstrado em breve nas imagens desta sessão.

A primeira imagem exibida é da área de trabalho onde os objetos foram colocados para realizar os testes. Pode-se observar que não há cor homogênea tanto nessa assim como em todas as imagens capturadas pela câmera.

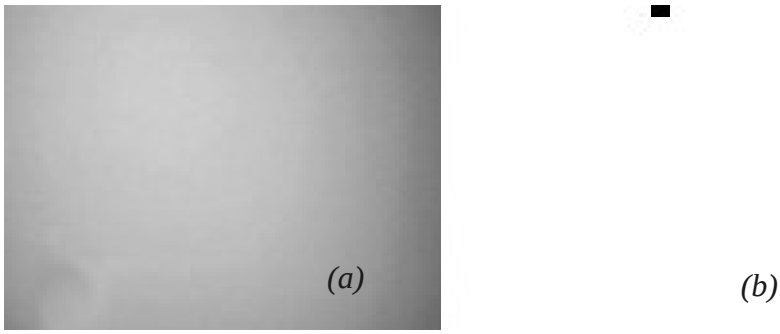


Figura 33: Representação da área de trabalho (a) em (b).

Como pode-se ver na Figura 33 a área de trabalho ficou representada com uma cor acinzentada. O grande ponto em Figura 33(b) representa o centroide aproximado calculado pelo algoritmo da área de trabalho vazia (o ponto foi aumentado para facilitar a visualização). A Figura 34 representa um objeto retangular detectado pelo microcontrolador:

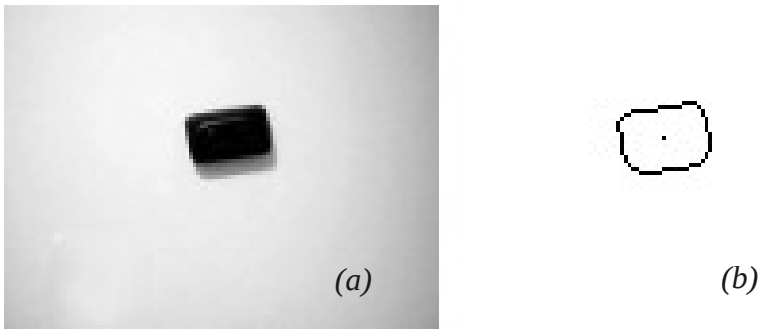


Figura 34:Objeto retangular em (a) detectado em (b).

Apesar das sombras na imagem não houve problemas com o algoritmo. Sombras podem ser bastante prejudiciais caso elas sejam escuras o suficiente, como será visto mais adiante. Como foi mencionado no Capítulo 4, o algoritmo não funciona quando há mais de um objeto na área de trabalho. A Figura 35 ilustra perfeitamente este acontecimento:

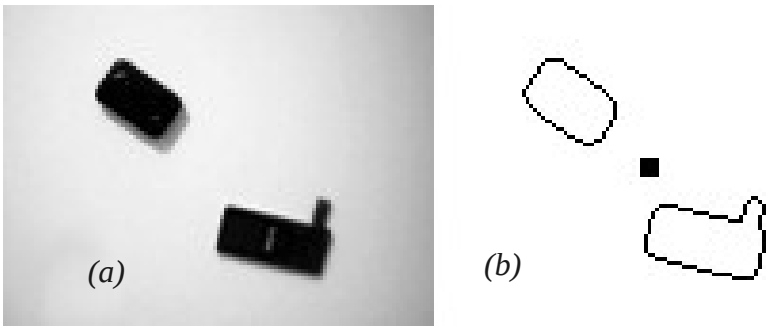


Figura 35: Detecção incorreta em (b) de dois objetos presentes em (a)

O pronto preto na Figura 35(b) foi aumentado e representa a falha na detecção com mais de um objeto presente. Pode-se ver que os contornos foram facilmente percebidos pelo algoritmo de pré-processamento mas o processamento em si falhou. Percebe-se, novamente, que a sombra não atrapalhou no funcionamento do algoritmo. A próxima imagem demonstra um erro que pode ser causado com sombras na área de trabalho:

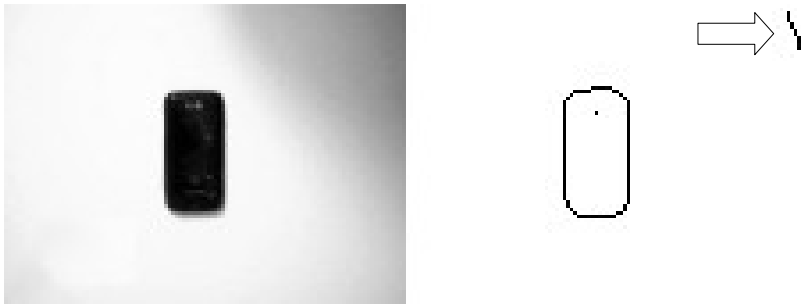


Figura 36: Erro na detecção do objeto (a) em (b). Nota-se a sombra no canto superior direito em (b).

Como pode-se ver na Figura 36 (comparar com a Figura 37) há um erro na detecção do centroide causado pela sombra apontada pela seta na Figura 36(b). Já na Figura 37 o centroide aproximado foi calculado corretamente apesar da sombra estar presente mas não tanto quanto na figura anterior.

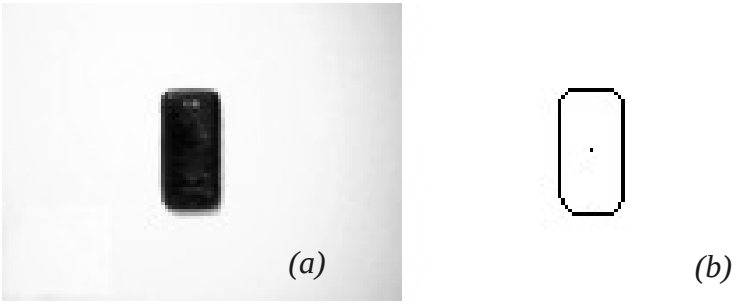


Figura 37: Detecção correta de um objeto (a) em (b).

Alguns objetos não puderam ser detectados. Dependendo da ocasião isto pode ser benéfico ou maléfico. A Figura 38 e Figura 39 demonstram duas ocasiões em que um objeto não foi detectado:

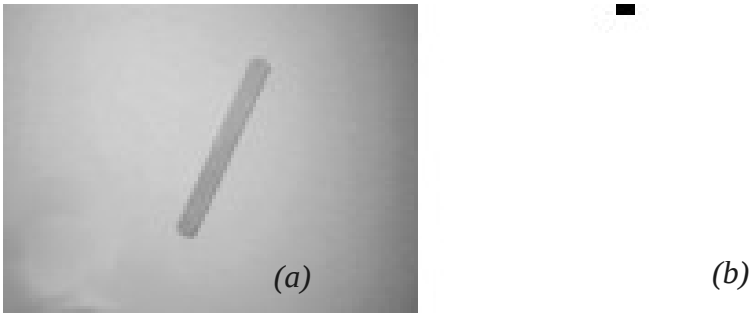


Figura 38: Detecção incorreta do objeto (a) em (b). Nota-se que o nível de cinza de (a) é próximo ao nível fundo da imagem.

A não detecção do objeto na Figura 38 foi devido a sua cor em níveis de cinza ser próximo a cor da área de trabalho. O algoritmo de pré-processamento eliminou o objeto e considerou a imagem como área de trabalho vazia (comparar com Figura 33). O ponto na parte *(b)* da imagem foi aumentado para facilitar a visualização. O mesmo efeito acontece na Figura 39 onde o objeto de nível de cinza menor (cor mais escura) foi detectado perfeitamente.

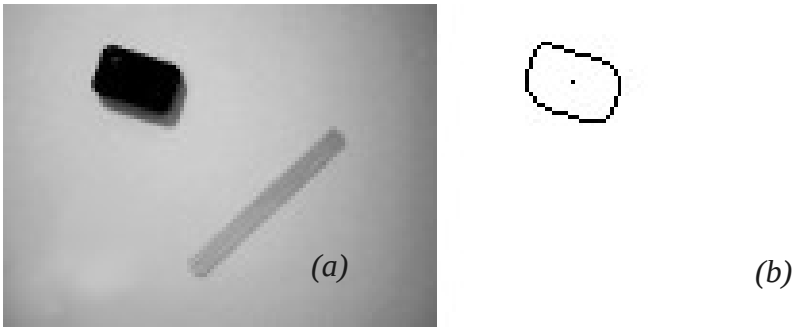


Figura 39: Detecção do objeto mais escuro de (a) em (b). Nota-se que o objeto mais comprido não foi detectado.

Capítulo 6 – Conclusão

Processamento embarcado de imagens é possível. Com os devidos ajustes nos algoritmos e periféricos um microcontrolador pode capturar, processar e controlar algum dispositivo de acordo com as informações adquiridas com a câmera. Alguns ainda conseguem realizar processamento em tempo real.

O microcontrolador é uma escolha importante e deve-se desprender algum tempo para avaliar as características oferecidas para que possa-se escolher o melhor microcontrolador de acordo com a relação custo benefício.

Pode-se concluir que o processo demonstrado neste trabalho é totalmente funcional e atendeu as expectativas detectando objetos na área de trabalho como proposto e pode ser utilizado como controlador do Robô-VD ou até mesmo como controlador de outros dispositivos onde a detecção de objetos em uma área de trabalho sejam úteis.

6.1 Trabalhos futuros

Como mencionado no texto, anteriormente, o sistema é um ponto de partida para aplicações que utilizem visão computacional e processamento de imagens embarcado. Os seguintes trabalhos futuros são sugeridos:

- Ligar o microcontrolador a uma câmera serial real e realizar testes já que não houve tempo hábil para se adquirir uma câmera serial.
- Desenvolver outras interfaces para aquisição de imagens como câmeras SPI, I^2C, etc.
- Testar o microcontrolador e os algoritmos com outras fontes de imagens tais como câmeras SPI, I^2C e outras que possuam taxa de transmissão de dados superior à transmissão serial.
- Aprimorar o algoritmo utilizado de detecção de objetos para que possa ser possível detectar mais de um objeto na área de trabalho.
- Comparar o desempenho do algoritmo utilizado com algoritmos clássicos de detecção de objetos
- Comparar o desempenho de algoritmos de pré-processamento diferentes visando modificação de algum deles para microcontroladores
- Integrar o sistema desenvolvido ao braço robótico

Bibliografia

[1]. RUFINO, F. A. de O., **Desenvolvimento de um robô inteligente com visão digital**, 2003, 118f. Dissertação de mestrado, programa de pós-graduação em engenharia mecânica, Centro de Tecnologia, UFPB

[2]. LIMA, S. M. L. de, **Sistema de Auxílio ao diagnóstico de doenças intracranianas e câncer de mama**, 2009, 71f. Trabalho de conclusão de curso, Escola Politécnica de Pernambuco, Universidade de Pernambuco - UPE

[3]. WANG, Chao, et al, **Real Time Image Collection and Processing Based on OMAP3530**, IEEE 978-1-4244-4994-1/09, 2009

[4]. AGGARWAL, Abhishek, **Embedded Vision System (EVS)**, IEEE 1-4244-2368-2/08, 2008

[5]. MASCARENHAS, I. C., **Plataforma para processamento de imagem com dsPIC com exemplo para reconhecimento de caractere usando rede neural**, 2010, 41f. Trabalho de conclusão de curso, Escola Politécnica de Pernambuco, Universidade de Pernambuco - UPE

[6]. GONZALES, Rafael C., WOODS, Richard E., **Digital Image Processing**. 2ª Edição. Prentice Hall

[7]. ROSS, Sheldon M., **Introduction to probability and statistics for engineers and scientists**. 3ª Edição. Elsevier Academic Press

[8]. OTSU, N., **A Threshold Selection Method from Gray-Level Histograms**, IEEE Transactions on Systems, Man, and Cybernetics, 1979

[9]. NIBLACK, W., **An Introduction to Digital Image Processing**. 1ª Edição. Prentice Hall

[10]. NXP, **LPC1769/68/67/66/65/64/63 Product data sheet**, 25 Agosto 2010, Disponível em <htttp://ics.nxp.com/products/lpc1000/datasheet/lpc1763.lpc1764.lpc1765.lpc1766.lpc1767.lpc1768.lpc1769.pdf> Último acesso: 21/03/2011

[11]. NXP, **UM10360 LPC17xx User Manual**, 2010, Disponível em <http://ics.nxp.com/support/documents/microcontrollers/pdf/user.manual.lpc17xx.pdf>

Último acesso: 16/05/2011

[12]. SCHUNK, Leonardo Marcílio, LUPPI, Aldo, **Microcontroladores AVR**. 1ª Edição. Ed. Érica Ltda.

[13]. ATMEL, **AT90S1200 Datasheet**, 2002, Disponível em <http://www.atmel.com/atmel/acrobat/doc0838.pdf> Último acesso: 07/05/2011

[14]. Microchip, **PIC18F2455/2550/4455/4550 Data Sheet**, 2006, Disponível em <http://ww1.microchip.com/downloads/en/DeviceDoc/39632e.pdf> Último acesso: 14/05/2011

[15]. Wikipedia.org, **Charge-coupled device**, Disponível em <http://en.wikipedia.org/wiki/Charge-coupled_device> Último acesso: 19/05/2011

[16]. LinkSprite, **LinkSprite JPEG Color Camera Serial UART Interface**, , Disponível em <http://www.sparkfun.com/datasheets/Sensors/Imaging/1274419957.pdf> Último acesso: 19/05/2011

[17]. OpenCV, **Open Source Computer Vision**, Disponível em <http://opencv.willowgarage.com/wiki/> Último acesso: 24/05/2011

[18]. NXP, **mbed**, Disponível em <http://mbed.org> Último acesso: 25/05/2011

[19]. ARM, **ARM Compiler**, Disponível em <http://www.arm.com/products/tools/arm-compiler.php> Último acesso: 25/05/2011

Apêndice A

A.1 - Demonstração dos filtros de aguçamento

A derivada de uma imagem digital *f(x, y)* para cada uma de suas variáveis é dada na forma de equações de diferença [6]:

$$\frac{\partial f}{\partial x} = f(x+1, y) - f(x, y) \tag{8}$$

$$\frac{\partial f}{\partial y} = f(x, y+1) - f(x, y) \tag{9}$$

De modo semelhante, pode-se calcular a segunda derivada para cada uma das variáveis. Pode-se facilmente chegar ao seguinte resultado por nova derivação das equações (8) (9):

$$\frac{\partial^2 f}{\partial x^2} = f(x+1, y) + f(x-1, y) - 2 \cdot f(x, y) \tag{10}$$

$$\frac{\partial^2 f}{\partial y^2} = f(x, y+1) + f(x, y-1) - 2 \cdot f(x, y) \tag{11}$$

Utilizando-se das equações (10) e (11), facilmente calcula-se o Laplaciano de uma função digital:

$$\nabla^2 f = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2}$$
$$\nabla^2 f = [f(x+1, y) + f(x-1, y) + f(x, y+1) + f(x, y-1)] - 4 f(x, y) \tag{12}$$

O Laplaciano pode ser implementado utilizando os seguintes filtros [6]:

$$\boldsymbol{W}_{Lap} = \begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix} \quad \text{ou} \quad \boldsymbol{W}_{Lap} = \begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

A diferença de sinal entre ambos é apenas uma questão de convenção. No filtro com elemento central positivo, as diferenças irão ser marcadas com uma mudança positiva no nível de cinza enquanto que no outro filtro serão marcadas com uma mudança negativa. Define-se agora uma nova imagem *g(x, y)* de modo que:

$$g(x, y) = f(x, y) + \nabla^2 f \tag{13}$$

Onde utiliza-se o Laplaciano com parte central positiva. Em notação matricial temos:

$$\boldsymbol{G} = \boldsymbol{F} + \nabla^2 \boldsymbol{F}$$

$$\boldsymbol{G} = \boldsymbol{F} * \delta(x, y) + \boldsymbol{F} * \boldsymbol{W}_{Lap}$$

$$\boldsymbol{G} = \boldsymbol{F} * \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} + \boldsymbol{F} * \begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix} = \boldsymbol{F} * \begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix} = \boldsymbol{F} * \boldsymbol{W}_1 \tag{14}$$

O filtro $\boldsymbol{W_2}$ pode ser calculado utilizando-se outra implementação do Laplaciano que utiliza também as informações dos *pixels* das diagonais. O procedimento para o calculo é semelhante ao do filtro $\boldsymbol{W_1}$ [6].

A.2 - Demonstração dos filtros de *Sobel*

Os filtros de *Sobel* utilizam o gradiente da imagem para detecção de bordas. A ideia por trás deles é detectar linhas horizontais ou verticais, dependendo do sentido do gradiente que foi selecionado.

Seja uma imagem *f(x, y)* o gradiente dele é calculado de acordo com (15)

$$\nabla \cdot \boldsymbol{f} = \begin{bmatrix} G_x \\ G_y \end{bmatrix} = \begin{bmatrix} \dfrac{\partial f}{\partial x} \\ \dfrac{\partial f}{\partial y} \end{bmatrix} . \tag{15}$$

Utiliza-se então a magnitude do valor do gradiente de *f* que pode ser aproximada da seguinte forma [6]

$$\begin{aligned} |\nabla \cdot \boldsymbol{f}| = \nabla f = \sqrt{\left[G_x^2 + G_y^2\right]} = \sqrt{\left[\left(\frac{\partial f}{\partial x}\right)^2 + \left(\frac{\partial f}{\partial y}\right)^2\right]} \quad . \\ \nabla f \approx |G_x| + |G_y| \end{aligned} \tag{16}$$

G_x e G_y podem ser implementados como diferenças, como utilizado nos filtros laplacianos e, portanto, a equação (16) pode ser reescrita da seguinte forma

$$\begin{aligned} \nabla f \approx & \left|(z_7 + z_8 + z_9) - (z_1 + z_2 + z_3)\right| \\ & + \left|(z_3 + z_6 + z_9) - (z_1 + z_4 + z_7)\right| \end{aligned} \tag{17}$$

onde

$$W = \begin{bmatrix} z_1 & z_2 & z_3 \\ z_4 & z_5 & z_6 \\ z_7 & z_8 & z_9 \end{bmatrix} \tag{18}$$

é uma região da imagem. Em notação matricial, a parte *Gx* e *Gy* podem ser expressas como

$$G_x = \begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix} \quad G_y = \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix} \tag{19}$$

e, com uma leve modificação chegamos ao filtro de *Sobel*

$$W_{S1} = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} \quad W_{S2} = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} . \tag{20}$$

Há uma leve diferença entre os filtros de *Sobel* mostrados no Capítulo 2 e os filtros na equação (20). Essas diferenças se seguem por causa da escolha do sentido do gradiente. Pode-se chegar facilmente nas outras representações trocando a escrita da equação (17).

Apêndice B

B.1: Código-fonte da aplicação simuladora da câmera serial

```
#include <cstdio>
#include <iostream>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <opencv/cv.h>
#include <opencv/highgui.h>
#include "commands.h"

#define DEFAULT_CAMERA              0
#define MBED_DEFAULT_WINDOW   "mbed_window"
#define MBED_DEFAULT_LOCATION"/dev/ttyACM0"

int main (int argc, char ** argv) {
        int mbed_fd, i;
        int size_written;
        unsigned int command_received, command_send;
        bool configured = false;
        cv::Mat frame, tmp_frame;
        cv::namedWindow (MBED_DEFAULT_WINDOW, 1);
        std::cerr << "Abrindo MBED...\n";

        mbed_fd = open (MBED_DEFAULT_LOCATION, O_RDWR);
        if (mbed_fd < 0) {
                std::cerr << MBED_DEFAULT_LOCATION << ": DEVICE NOT FOUND ! ABORT\n";
                return 1;
        }
        std::cerr << "MBED Encontrada...\n" << "Lendo configurações...\n";
        cv::VideoCapture cap(DEFAULT_CAMERA);
```

```
if (!cap.isOpened()) {
		std::cerr << "CAMERA NOT FOUND ! ABORT\n";
		return 1;
}
if (!configured) {
		// Read width configuration
		read (mbed_fd, &command_received, 4);
		printf ("%.8X\n", command_received);
		command_send = ACK;
		switch (command_received) {
		case CAMERA_SET_WID_160:
		case CAMERA_SET_WID_320:
		case CAMERA_SET_WID_640:
			write (mbed_fd, &command_send, 4);
			std::cerr << "WIDTH configurado\n";
			break;
		default:

			break;
		}

		read (mbed_fd, &command_received, 4);
		printf ("%.8X\n", command_received);
		switch (command_received) {
		case CAMERA_SET_HEI_120:
		case CAMERA_SET_HEI_240:
		case CAMERA_SET_HEI_480:
			write (mbed_fd, &command_send, 4);
			std::cerr << "HEIGHT configurado\n";
			configured = true;
			break;
		default:
		break;
		}
}
```

```
        unsigned char buf[9492];

        read (mbed_fd, &command_received, 4);
        printf ("%.8X\n", command_received);
        if (command_received == CAMERA_CAPTURE_FRAME) {
                std::cerr << "Capture frame...\n";
                cap >> frame;
                cv::waitKey(30);
                cv::cvtColor(frame, tmp_frame, CV_RGB2GRAY);
                cv::resize(tmp_frame, frame, cv::Size (113, 84), 0, 0, cv::INTER_LINEAR);
                cv::waitKey(30);      /* Waits for MBED to get ready */
                size_written = write (mbed_fd, (void *)frame.data, 9492);
                std::cout << "Tamanho escrito: " << size_written << "\n";
                imshow (MBED_DEFAULT_WINDOW, frame);
                cv::waitKey(0);
        }

        return 0;
}
```

B.2: Código-fonte da aplicação do microcontrolador

Main.c

```
#include "mbed.h"
#include "imagem.h"
#include "command.h"
#include "algoritmo.h"
#include "default_vars.h"
#include "SerialInterface.h"

DigitalOut readImageError (LED1);
#define DEBUG   0

void capture_frame (void);
void process_frame (void);

/* Global variables */
Imagem im (113, 84);
Imagem tmp (113, 84);
SerialInterface interface (1024, USBTX, USBRX, 3);

int main() {

    UINT32 command_send, command_received;
    UINT32 reader;
    /* Camera configuration routines */
    command_send = CAMERA_SET_WID_160;
    command_received = interface.sendBuffer (4, &command_send);

    command_send = CAMERA_SET_HEI_120;
    command_received = interface.sendBuffer (4, &command_send);
```

```
    capture_frame();
}

void capture_frame (void) {
    UINT32 send_command, received_command;
    UINT8 command;
    send_command = CAMERA_CAPTURE_FRAME;
    received_command = interface.sendBuffer (4, &send_command);

    readImageError = im.captureFrame(&interface);

#if DEBUG
    printf ("PROCESSING FRAME\r\n");
#endif
    // Proccess the frame using algorithm
    process_frame ();

#if DEBUG
    printf ("SEARCH CENTROID\r\n");
#endif
    // Searches for black lines
    Point centroid = searchCentroid (&im, 0x00);
    printf ("%.3d %.3d\r\n", centroid._x, centroid._y);
}

/* Routine for pre-processing image captured */
void process_frame (void) {
    float media [][3] = {
        {0.111111f, 0.111111f, 0.111111f},
        {0.111111f, 0.111111f, 0.111111f},
        {0.111111f, 0.111111f, 0.111111f} };
    float laplacian [][3] = {
        {0.000000f, -1.00000f, 0.000000f},
        {-1.00000f, 4.000000f, -1.00000f},
        {0.000000f, -1.00000f, 0.000000f} };
```

```
    int i, j;
     im.filter (&tmp, media);
    tmp.otsu();
    tmp.filter (&im, laplacian);
    for (j = 0; j < im.height; j++)
        for (i = 0; i < im.width; i++)
            im.im[j][i] = 255 - im.im[j][i];

    for (i = 0; i < 113; i++) {
        im.im[0][i] = 0xFF;
        im.im[1][i] = 0xFF;

        im.im[im.height-1][i] = 0xFF;
        im.im[im.height-2][i] = 0xFF;
    }
    for (j = 0; j < 84; j++) {
        im.im[j][0] = 0xFF;
        im.im[j][1] = 0xFF;

        im.im[j][im.width-1] = 0xFF;
        im.im[j][im.width-2] = 0xFF;
    }
}
```

Default_vars.h

```
#ifndef DEFAULT_VARS_H
#define DEFAULT_VARS_H
            typedef    unsigned char   UINT8;
            typedef    unsigned short  UINT16;
            typedef    unsigned int    UINT32;
#endif
```

BufferedSerial.h e BufferedSerial.cpp

```
#ifndef BUFFEREDSERIAL_H
#define BUFFEREDSERIAL_H

#include "mbed.h"
class BufferedSerial : public Serial {
  public:
      BufferedSerial ( size_t bufferSize, PinName tx, PinName rx );
      virtual ~BufferedSerial ();

      int getc ();
      int readable ();
      void setTimeout (float seconds);
      size_t readBytes ( uint8_t *buffer, size_t requested );
   private:
      void handleInterrupt ();

      uint8_t    *mem_buff;
      uint16_t   mem_contentStart;
      uint16_t   mem_contentEnd;
      uint16_t   mem_buffSize;
      float      timeout;
      Timer      mem_timer;
};
#endif

#include "BufferedSerial.h"
BufferedSerial::BufferedSerial (size_t bufferSize, PinName tx, PinName rx) : Serial (tx, rx) {
   mem_buffSize = 0;
   mem_contentStart = 0;
   mem_contentEnd = 0;
   timeout = 1.0;

   mem_buff = (uint8_t *) malloc (bufferSize);
   if (!mem_buff)
```

```
        printf ("ERRO ALOCANDO MEMORIA!\r\n");
    else {
        mem_buffSize = bufferSize;
        attach (this, &BufferedSerial::handleInterrupt);
    }
}

BufferedSerial::~BufferedSerial () {
    if (mem_buff)   free (mem_buff);
}

void BufferedSerial::setTimeout (float seconds){
    timeout = seconds;
}

size_t BufferedSerial::readBytes (uint8_t *buffer, size_t requested) {
    int i, c;
    for (i = 0; i < requested; i++) {
        c = getc();
        if (c < 0)  break;
        buffer[i] = c;
    }
     return i;
}

int BufferedSerial::getc() {
    uint8_t r;
    mem_timer.reset();
    mem_timer.start();

    while (mem_contentStart == mem_contentEnd) {
        wait_ms (1);
        if (timeout > 0 && mem_timer.read() > timeout)
            return EOF;
    }
```

```
    mem_timer.stop();
     r = mem_buff[mem_contentStart++];
    mem_contentStart = mem_contentStart % mem_buffSize;
     return r;
}

int BufferedSerial::readable() {
    return mem_contentStart != mem_contentEnd;
}

void BufferedSerial::handleInterrupt (){
    while (Serial::readable()) {
        if (mem_contentStart == (mem_contentEnd + 1) % mem_buffSize) {
            Serial::getc();
        } else {
            mem_buff[mem_contentEnd++] = Serial::getc();
            mem_contentEnd = mem_contentEnd % mem_buffSize;
        }
    }
}
```

Imagem.h e Imagem.cpp

```
#ifndef IMAGEM_H
#define IMAGEM_H

#include "mbed.h"
#include "Interface.h"
#include "default_vars.h"

class Imagem {
  public:
      UINT8 **im;
       UINT8 width, height;
      Imagem (UINT8 wid, UINT8 hei);
       ~Imagem (void);
      UINT8 captureFrame (Interface *iface);
```

```
        UINT8 calculateThreshold (void);
        void calculateHistogram (UINT16 *histogram);
        void filter (Imagem *dst, float filter[][3]);
        void otsu (void);
};
#endif

#include "imagem.h"
#include "math.h"

Imagem::Imagem (UINT8 wid, UINT8 hei) {
    UINT8 i;

    width = wid;
    height = hei;
    im = (UINT8 **) malloc (sizeof(UINT8 *)*hei);
    if (im == NULL)
        return ;
    for (i = 0; i < hei; i++) {
        im[i] = (UINT8 *)malloc (sizeof (UINT8)*wid);
        if (im[i] == NULL)
            return ;
    }
}

Imagem::~Imagem (void) {
    UINT8 i;
    for (i = 0; i < height; free (im[i++]));
    free (im);
}

UINT8 Imagem::captureFrame (Interface *iface) {
    UINT8 i, bytesRead;
    if (iface == NULL)
        return 1;
```

```
    for (i = 0; i < height; i++) {
        bytesRead = iface->readBuffer (width, im[i]);
        if (bytesRead < width) {
            return 1;
        }
    }
    return 0;
}

void Imagem::calculateHistogram (UINT16 *histogram) {
    UINT8 i, j;

    for (j = 0; j < height; j++)
        for (i = 0; i < width; i++)
            histogram[im[j][i]]++;
}

void Imagem::filter (Imagem *dst, float filter[][3]) {
    UINT8 i, j;
    short newc;

    for (j = 1; j < height - 1; j++) {
        for (i = 1; i < width - 1 ; i++) {
            newc =  (UINT16)floor(im[j-1][i-1]*filter[0][0] + im[j-1][i]*filter[0][1] + im[j-1][i+1]*filter[0][2] +
                    im[j][i-1]*filter[1][0] + im[j][i]*filter[1][1] + im[j][i+1]*filter[1][2] +
                    im[j+1][i-1]*filter[2][0] + im[j+1][i]*filter[2][1] + im[j+1][i+1]*filter[2][2]);
            if (newc > 0xFF)    newc = 0xFF;
            dst->im[j][i] = (UINT8)newc;
        }
    }
}

UINT8 Imagem::calculateThreshold (void) {
    UINT16 *h = new UINT16[256];
    UINT16 i, j;
    float p, pi0, mu0, mu1, var[256];
```

```
    float tamanho = (float)(width * height);

    for (i = 0; i < 256; h[i++] = 0) ;

    calculateHistogram (h);
    for (i = 1; i < 256; i++) {
        mu0 = 0; mu1 = 0; pi0 = 0;
        for(j = 0; j < i; j++) {
            p = (float)h[j]/tamanho;
            pi0 = pi0 + p;
        }
        for (j = 0; j < i; j++) {
            p = (float)h[j]/tamanho;
            mu0 = mu0 + (float)(j+1)*p/pi0;
        }
        for (j = i; j < 256; j++) {
            p = (float)h[j]/tamanho;
            mu1 = mu1 + (float)(j+1)*p/(1-pi0);
        }
        var[i] = pi0*(1.0f-pi0)*(mu1-mu0)*(mu1-mu0);
    }
    j = 0;
    for (i = 0; i < 256; i++)
        if(var[i] > var[j])    j = i;
    return j;
}

void Imagem::otsu (void) {
    UINT8 tval, i, j;
    tval = calculateThreshold ();
    printf ("%d\r\n", tval);
    for (j = 0; j < height; j++)
        for (i = 0; i < width; i++)
            if (im[j][i] < tval)
                im[j][i] = 0;
            else
```

```
            im[j][i] = 0xFF;
}
```

Command.h e Command.cpp

```
#ifndef COMMAND_H
#define COMMAND_H

#include "mbed.h"
#include "default_vars.h"
#include "Interface.h"

#define CAMERA_SET_WID_160      0x010100A0  // Set Wid=160
#define CAMERA_SET_HEI_120      0x01020078  // Set hei=120
#define CAMERA_SET_WID_320      0x01010140  // Set wid=320
#define CAMERA_SET_HEI_240      0x010200F0  // Set hei=240
#define CAMERA_SET_WID_640      0x01010280  // Set wid=640
#define CAMERA_SET_HEI_480      0x010201E0  // Set hei=480
#define CAMERA_CAPTURE_FRAME    0x01030000  // Reads a frame from camera

#define MBED_CAPTURE_FRAME      0x02010000  // Capture frame and proccess

#define ACK                 0x7FFFFFFF  // Acknowledge
#define NACK                0xFFFFFFFF  // Not-Acknowledge

/* Envia dados do buffer data por iface */
int sendCommand (size_t size, void *data, Interface *iface);

#endif

#include "command.h"
int sendCommand (size_t size, void *data, Interface *iface) {
   if (iface->sendBuffer (size, data) < size)
      return 1;
   else
      return 0;
}
```

Interface.h

```
#ifndef INTERFACE_H
#define INTERFACE_H

/* Classe de Interface generica */
/* Usada para generalizar a ideia de transmissao de dados. */

class Interface {

   public:

      virtual int sendBuffer (size_t size, void *buffer) = 0 ;
      virtual int readBuffer (size_t size, void *buffer) = 0 ;
};

#endif
```

SerialInterface.h e SerialInterface.cpp

```
#ifndef SERIALINTERFACE_H
#define SERIALINTERFACE_H

#include "mbed.h"
#include "BufferedSerial.h"
#include "Interface.h"
class SerialInterface : public Interface {
private:
   BufferedSerial *iface;
   float timeout;
public:
   SerialInterface (size_t size, PinName tx, PinName rx, float timeout) ;
   ~SerialInterface (void) ;
   int readBuffer (size_t size, void *buffer) ;
   int sendBuffer (size_t size, void *buffer) ;
};
#endif
```

```
#include "SerialInterface.h"
SerialInterface::SerialInterface (size_t size, PinName tx, PinName rx, float timeout) {
    iface = new BufferedSerial (size, tx, rx);
    iface->baud (115200);
    iface->setTimeout (timeout);
}

SerialInterface::~SerialInterface (void) {
    delete iface;
}

int SerialInterface::readBuffer (size_t size, void *buffer) {
    return iface->readBytes ((uint8_t *)buffer, size);
}

int SerialInterface::sendBuffer (size_t size, void *buffer) {
    size_t i = 0;
    uint8_t *b = (uint8_t *)buffer;
    while (i < size)
        if (iface->writeable())
            iface->putc (b[i++]);
   return i;
}
```

Algoritmo.h e Algoritmo.cpp

```
#ifndef ALGORITMO_H
#define ALGORITMO_H

#include "default_vars.h"
#include "imagem.h"
class Point {
   public:
       UINT8   _x, _y;
       bool    isSet;
```

```
        Point (UINT8 x, UINT8 y) { _x = x; _y = y; isSet = true; }
        Point (void) { _x = 0; _y = 0; isSet = false; }
        void set (UINT8 x, UINT8 y) { _x = x; _y = y; isSet = true; } };
Point searchCentroid (Imagem *im, UINT8 color) ;
#endif


#include "mbed.h"
#include "default_vars.h"
#include "algoritmo.h"
#include "imagem.h"

Point searchCentroid (Imagem *im, UINT8 color) {
    UINT8 i, j, midy;
    Point upper, lower, centroid;

    for (j = 0; j < im->height; j++) {
        for (i = 0; i < im->width; i++) {
            if (im->im[j][i] == color) {
                upper.set (i, j);
                break;
            }
        }
        if (upper.isSet)    break;
    }

    for (j = im->height - 1; j > 0; j--) {
        for (i = im->width - 1; i > 0; i--) {
            if (im->im[j][i] == color) {
                lower.set (i, j);
                break;
            }
        }
        if (lower.isSet)    break;
    }
```

```
    midy = (upper._y+lower._y) >> 1;
    for (i = 0; i < im->width ; i++)
        if (im->im[midy][i] == color)   break;
   for (j = im->width - 1; j > 0; j--)
        if (im->im[midy][j] == color)   break;
   centroid.set ((i+j)>>1, midy);
    return centroid;
}
```