UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL

LABORATÓRIO DE PROCESSAMENTO DE SINAIS E IMAGENS

LAPSI IMAGE PROCESSING LIBRARY

lili
versão 0.1.4

MANUAL DO USUÁRIO / PROGRAMADOR

VERSÃO DO MANUAL 0.0.1

ENDEREÇO ELETRÔNICO
WWW.LAPSI.ELETRO.UFRGS.BR/LILI

ESCRITO POR: THIAGO ROSA FIGUEIRÓ

SUMÁRIO

1. INTRODUÇÃO À BIBLIOTECA *lili*

1.1. DESCRIÇÃO E CARACTERÍSTICAS

A Biblioteca de Processamento de Imagens do LaPSI (Laboratório de Processamento de Sinais e Imagens), ou simplesmente *lili* (***L**aPSI **I**mage Processing **Li**brary*), vem sendo desenvolvida com o objetivo de oferecer uma maneira única e eficiente para manipulação de arquivos e processamento das imagens. Mais do que uma simples biblioteca, a *lili* é um ambiente aberto para desenvolvimento de aplicações em processamento de imagens, sendo apta para auxiliar seu usuário/programador a realizar tarefas nas mais diferentes áreas, dependendo apenas do conhecimento específico.

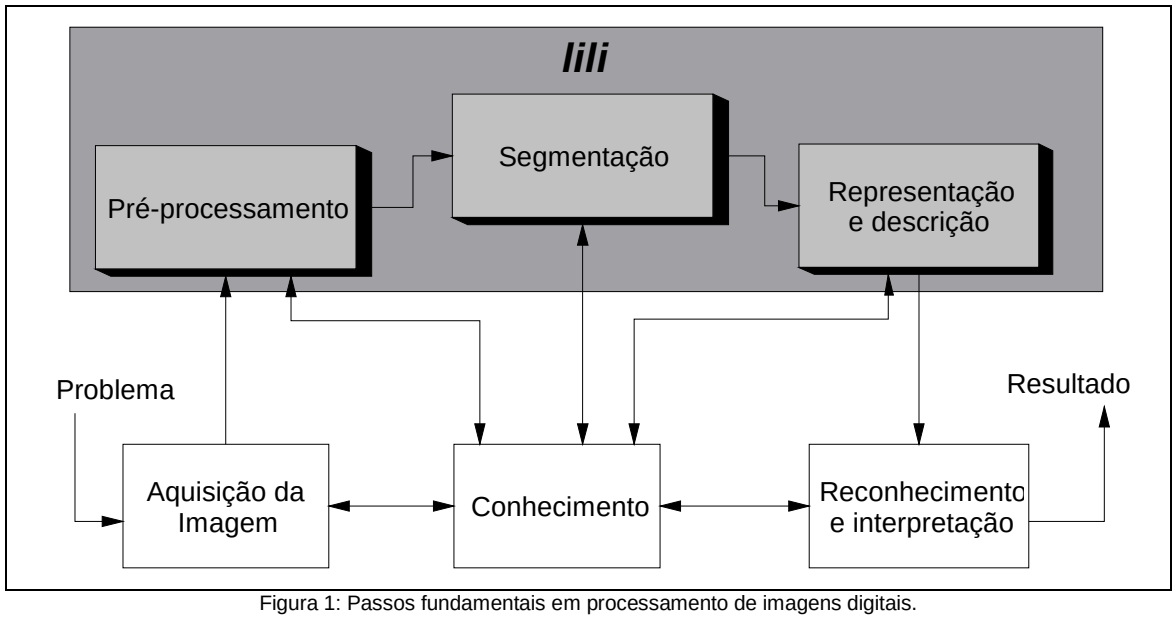

Figura 1: Passos fundamentais em processamento de imagens digitais.

Qualquer aplicação em processamento de imagens pode ser dividida em passos fundamentais (como mostra o esquema na figura 1). A solução de um problema inicia com a aquisição da imagem, segue por um mecanismo de pré-processamento, uma etapa de segmentação, outra de representação e descrição e, por fim, o reconhecimento e interpretação.

O primeiro passo envolvido é a aquisição da imagem, o qual requer algum tipo de sensor e a capacidade para digitalizar o seu sinal - essa etapa já envolve o conhecimento específico do problema, pois existem vários métodos de adquiri-las. O seguinte, pré-processamento, envolve o realce das características desejáveis na imagem a fim de que aumentem as chances de sucesso nas demais etapas do processamento. A segmentação ocorre em seguida, particionando as imagens nos objetos que as compõe. A escolha de uma forma correta de representação é fundamental para transformar o resultado da segmentação em uma estrutura computável. A descrição separa as diferentes classes de objetos na imagem. Por fim, a etapa de reconhecimento rotula cada objeto a partir dos dados provenientes

do descritor, enquanto reconhecimento trata de dar significado aos objetos reconhecidos.

Todas as etapas descritas são baseadas no conhecimento do problema que se está tentando solucionar, sendo necessário esse embasamento para escolher os processos mais apropriados para pré-processar, segmentar e representar/descrever. A *lili* está diretamente envolvida com essas etapas, cabendo ao usuário/programador apenas escolher entre as funções disponibilizadas no repositório da *lili* ou, eventualmente, desenvolver uma nova - que ainda não esteja implementada na biblioteca - e inseri-la na *lili*.

A *lili* é um ambiente livre de licença, o que permite que todos possam desenvolver sua aplicação (mesmo comercial) sem pagar pelo uso ou comercialização. Além disso, a *lili* tem seu código aberto, possibilitando que seus usuários conheçam a fundo o seu funcionamento, podendo alterá-lo de acordo com a sua necessidade e aprender seus algoritmos.

Visando a portabilidade, a biblioteca foi escrita na linguagem ANSI/ISO C. Assim, um único código fonte pode ser utilizado em diversas plataformas, requerendo apenas um projeto que vincule seus diversos códigos fonte para cada ambiente. Versões para os ambientes Linux/Unix e MS-Windows estão disponíveis.

1.2. DISTRIBUIÇÃO E ORGANIZAÇÃO DOS ARQUIVOS

A biblioteca *lili*, juntamente com sua documentação, imagens de teste e alguns programas de exemplo, é disponibilizada através do arquivo lili-0.1.4.tar.gz para os ambientes Linux/Unix, ou lili-0.1.4.zip para MS-Windows. Os números 0.1.4 indicam a atual versão da biblioteca. Quando descompactada, a biblioteca possui a seguinte estrutura de diretórios:

bench: *benchmarks* para avaliar a bliblioteca ;

demo: programas de exemplo desenvolvidos para auxiliar os novos usuários ao ambiente *lili* e servir como modelo para as primeiras implementações;

doc: documentação eletrônica da biblioteca para consultas rápidas;

img: imagens disponibilizadas para testar os algoritmos implementados com a biblioteca *lili*. As imagens estão divididas em coloridas (color) e tons de cinza (gray);

include: arquivos de inclusão da biblioteca (*header files*) , que contém o cabeçalho de todas as funções que o usuário poderá utilizar ao incluir a *lili* em seu projeto;

lib: todos os arquivos de códigos fonte da biblioteca *lili* e seus cabeçalhos. Este diretório contém o corpo da biblioteca;

win32: código específico para desenvolver aplicações para o ambiente Windows;

Em um ambiente Windows, o aplicativo WinZip ou similiar (WinRAR, pkunzip, etc.) deve ser utilizado para descompatar o arquivo lili-0.1.4.zip. Já em Linux/Unix o arquivo lili-0.1.4.tar.gz deve ser descompactado com a seguinte linha de comando:
$ tar xzf lili-0.1.4.tar.gz

1.3. COMPILAÇÃO E USO PARA WINDOWS

Para o ambiente Windows, foi utilizado e recomenda-se o uso do software Borland C++ Builder 3. O arquivo de projeto da *lili* para este software (..\win32\lib\lili.bpr) deve ser aberto (menu File -> Open) e compilado (menu Project -> Build), o que deve gerar o arquivo lili.lib no mesmo diretório do projeto (..\win32\lib\lili.lib).

Para se criar um novo programa utilizando a biblioteca *lili* no Borland C++ Builder 3, deve-se primeiro configurar os diretórios onde o compilador procura os arquivos de inclusão (include) e biblioteca (lib). Para tanto, deve-se clicar em menu Project -> Options (figura 2), selecionar a aba Directories/Conditionals (figura 3), e digitar o caminho dos arquivos, sem apagar os diretórios padrões do Borland C++ Builder 3.

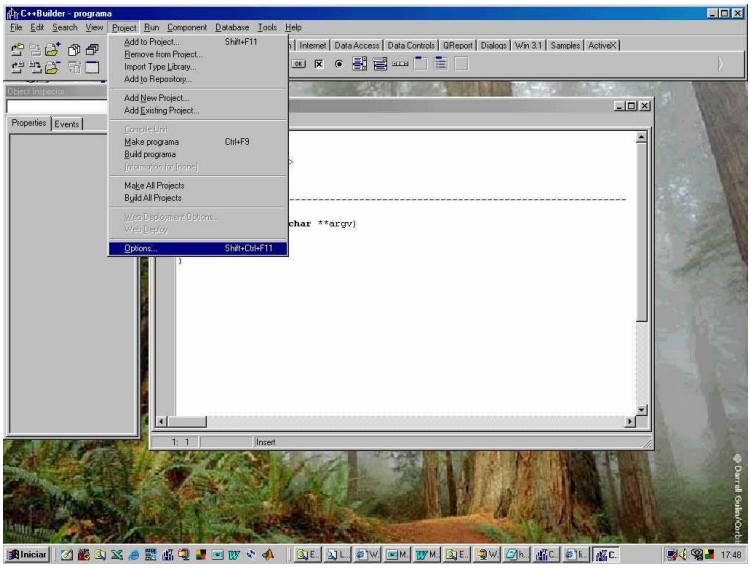

Figura 2: Clicando em project -> options.

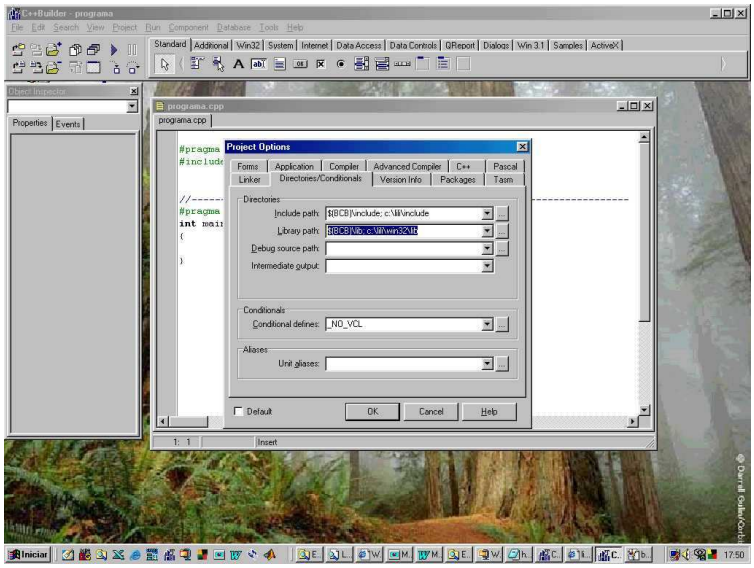

Figura 3: Clicando na aba Directories / Conditionals.

Depois, deve-se adicionar o arquivo de biblioteca lili.lib ao projeto, clicando em Project -> Add to Project (figura 4). O resultado é mostrado na figura 5.

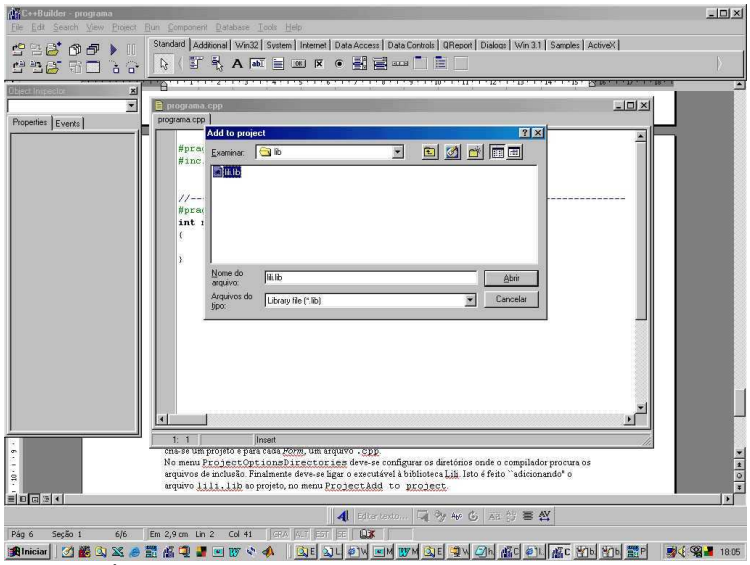

Figura 4: Após clicar em project -> add to project. Basta selecionar o arquivo lili.lib.

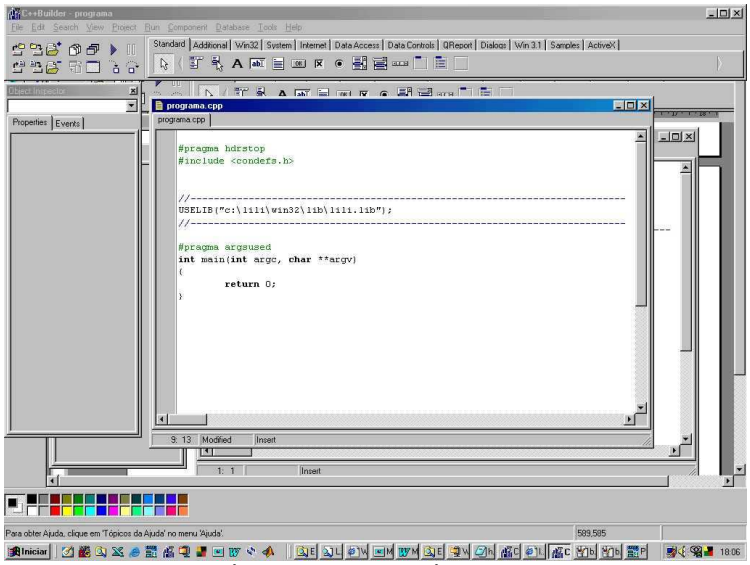

Figura 5: Visão do código da aplicação após adicionar a biblioteca lili.lib.

Por fim, deve-se incluir o arquivo cabeçalho da biblioteca lili (lili.h) na seção dos "includes" do programa recém iniciado. Para tanto, basta digitar:

#include "lili.h"

O nome do arquivo (lili.h) deve ser digitado entre aspas e não entre os sinais de menor que e maior que para indicar aos possíveis leitores do código que o arquivo incluído não pertence a biblioteca padrão do compilador que está sendo utilizado.

Após efetuados todos os passos indicados, o programa em desenvolvimento deve ser bastante semelhante ao apresentado na figura 6.

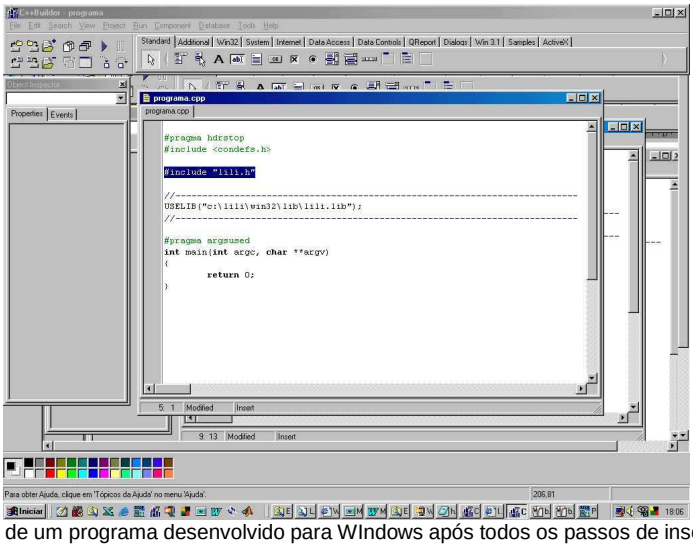

Figura 6: Aparência final de um programa desenvolvido para WIndows após todos os passos de inserção da biblioteca lili no processo.

1.4. COMPILAÇÃO E USO PARA LINUX / UNIX

Para o ambiente Linux/Unix, são necessários o comando Make (usado para automatizar o processo de compilação e de distribuição) e qualquer compilador ANSI/ISO C. A compilação da biblioteca é feita de forma automática, através do comando: $ make. Como resultado da compilação é gerado o arquivo ..\lib\lili.a.

Após criar um programa utilizando os comandos da *lili*, além de se incluir o cabeçalho (#include "lili.h") deve-se indicar para o compilador onde estão os arquivos de inclusão (.h) e onde estão as bibliotecas (.a) e quais devem ser ligadas. A linha de comando a seguir é um exemplo de como o programa exemplo.c pode ser compilado pelo gcc: $ gcc -I../include -L../lib -llili test.c -o test

2. CONCEITOS BÁSICOS ENVOLVIDOS

2.1. TIPOS DE DADOS

A biblioteca define alguns tipos de dados, como a estrutura limage*, que representa uma indicação para uma imagem na memória, lpixel, que pode ser vista como do tipo char (pois um pixel deve variar de 0 a 255); e as enumerações limage_type e lfile_format, que enumeram os tipos de imagem (RGB, tons de cinza) e os formatos de arquivos (atualmente apenas o formato BMP é suportado). Todos esses dados devem ser vistos como estruturas opacas, isto é, devem ser manipuladas somente pelas funções fornecidas pela biblioteca.

2.2. OPERAÇÕES BÁSICAS

Uma imagem é representada pela estrutura limage*. Existem três maneiras de se criar uma imagem na memória: através da função limage_new, que cria uma imagem nova "em branco", dadas as dimensões da imagem e o tipo desejado; através da função limage_new_from_file, que cria uma imagem a partir das informações obtidas em um arquivo recebido como parâmetro; através da limage_clone, que duplica uma imagem já existente na memória.

Outra operação fundamental é a destruição de imagens, pois todas as imagens criadas devem ser destruídas para liberar a memória que havia sido alocada.

Exemplo:

```
/* Deve-se declarar ponteiros para limage...*/
limage *im1, *im2, *im3 ;

 /* ... já que as estruturas em si são alocadas automaticamente! */

im1 = limage_new ( 640, 480, LIMAGE_TYPE_RGB ) ;
im2 = limage_new_from_file( "demo/img/lapsi.img" ) ;
im3 = limage_clone( im1 ) ;

/* Assim mesmo, elas devem ser destruídas explicitamente. */
limage_delete( im1 ) ;
limage_delete( im2 ) ;
limage_delete( im3 );
```

É possível também carregar uma imagem a partir de um arquivo em disco para uma imagem já existente na memória. Isto é feito através da função limage_load. Similarmente, a função limage_save salva uma imagem para um arquivo em disco em um formato especificado como argumento. É interessante notar que a função limage_load detecta automaticamente o formato do arquivo especificado, e assim não é necessário passá-lo como argumento.

Exemplo:

```
limage* im = limage_new( 320, 240, LIMAGE_TYPE_GRAYSCALE ) ;

limage_save( im, "minha_imagem.bmp", LIMAGE_TYPE_WINBMP ) ;

limage_load( im, "nova_imagem.png" ) ;
```

2.3. PROCESSAMENTO DE IMAGENS

O processamento das imagens é feito através de iteradores. Iteradores são estruturas responsáveis pelo acesso dos dados das imagens, assim como os ponteiros da linguagem C podem manipular os dados de uma matriz. Esta separação entre o armazenamento da imagem (feito pela estrutura limage) e a maneira pela qual estes dados são acessados (através dos iteradores) torna o processamento mais flexível: pode-se percorrer uma mesma imagem de diversas maneiras, utilizando os diferentes iteradores oferecidos pela biblioteca.

Em imagens coloridas (e, de uma maneira geral, imagens multi-canais), os canais são tratados separadamente, como imagens distintas. Assim, para cada canal a ser processado é criado um iterador, especificando-se o canal através da enumeração lchannel. Imagens do tipo RGB, por exemplo, possuem os canais das cores vermelho, verde e azul, representados por LCHANNEL_RED, LCHANNEL_GREEN e LCHANNEL_BLUE. Já imagens como as em tons de cinza possuem apenas um canal, referenciado por LCHANNEL_GRAY.

Na versão atual da biblioteca (0.1.4), dois iteradores já estão implementados: larray, que representa a imagem como uma matriz, e lptr, que representa a imagem como um vetor.

A função limage_as_array é responsável pela criação de um iterador do tipo larray. Assim, a imagem é vista como uma matriz e seus dados são acessados da mesma maneira com que é feito na linguagem C. Ou seja, dado um iterator it e uma posição , o pixel correspondente é referenciado como it[y][x].

O exemplo a seguir ilustra como uma imagem RGB (src) pode ser processada dando origem a outra (dst) em tons de cinza.

Exemplo:

```
void process_as_array( limage* src, limage* dst )
{
   larray r, g, b, y ;
   int i, j ;
   int tmp ;

   r = limage_as_array( src, LCHANNEL_RED ) ;
   g = limage_as_array( src, LCHANNEL_GREEN ) ;
   b = limage_as_array( src, LCHANNEL_BLUE ) ;

   y = limage_as_array( dst, LCHANNEL_DEFAULT ) ;

   for( i=0 ; i < limage_get_height(src) ; i++ )
      for( j=0 ; j < limage_get_width(src) ; j++ )
      {
         tmp  = r[i][j] ;
         tmp += g[i][j] ;
         tmp += b[i][j] ;

         y[i][j] = (lpixel)(tmp/3) ;
      } ;
} ;
```

É interessante notar que não é necessário destruir iteradores do tipo larray, uma vez que eles são mantidos em cache pela imagem correspondente. Desta

maneira, as chamadas seguintes a limage_as_array são otimizadas, desde que as características da imagem (tamanho, tipo, etc.) não tenham sido alteradas.

Imagens podem ser tratadas também como vetores (unidimensionais) de pixels e percorridas linearmente. Isto é feito através de iteradores do tipo lptr, que é equivalente a um ponteiro para pixel.

A função limage_as_ptr retorna um iterador para o canal especificado, da mesma maneira que limage_as_array o faz. Os pixels são referenciados através do operador * (indireção ou ``desreferenciação''), extamente como é feito com ponteiros, assim como os operadores ++ e - (incremento e decremento) são usadas em uma iteração. Adicionalmente, a função limage_end_ptr é necessária para verificar o final de uma imagem e, consequentemente, o final da iteração.

Como exemplo, temos a função a seguir que ``binariza'' uma imagem através de iteradores lptr.

Exemplo:

```
void process_as_ptr( limage* im, lpixel th )
{
   lptr it, end ;

   it  = limage_as_ptr ( im, LCHANNEL_DEFAULT ) ;
   end = limage_end_ptr( im, LCHANNEL_DEFAULT ) ;

   while( it++ != end )
      *ptr = ( *ptr >= th ) ? 255 : 0 ;
};
```

2.4. ENUMERAÇÕES E TIPOS

⇒ **limage** Estrutura para armazenar imagens em memória. Seus não devem ser acessados diretamente, mas sim através das funções da biblioteca *lili*.

⇒ **lpixel** Usado para representar um pixel da imagem. Definido como typedef unsigned char lpixel.

⇒ **limage_type** Enumeração, indica o tipo da imagem em memória.
- LIMAGE_TYPE_UNKNOWN
- LIMAGE_TYPE_GRAYSCALE
- LIMAGE_TYPE_RGB

⇒ **lchannel** Enumeração, usado para indicar o canal a ser processado.
- LCHANNEL_ALL
- LCHANNEL_DEFAULT
- LCHANNEL_GRAY
- LCHANNEL_RED
- LCHANNEL_GREEN
- CHANNEL_BLUE

⇒ **lfile_format** Enumeração, formato das imagens em arquivos.
- LFILE_FORMAT_UNKNOWN
- LFILE_FORMAT_WINBMP

2.5. MISCELÂNEA

Duas funções bastante úteis são a lfile_format_name, que retorna um string associado ao nome do formato e limage_type_channels, que retorna o número de canais associados a um dado tipo de imagem. Com essas funções o programador pode generalizar ainda mais o seu programa, desde que suporte as informações obtidas ou faça um adequado tratamento de erros (operações ilegais).

```
const char* lfile_format_name( lfile_format fmt ) ;

int limage_type_channels( limage_type ty ) ;
```