Universidade de São Paulo - USP

Instituto de Ciências Matemáticas e de Computação - ICMC

Departamento de Ciências de Computação

SCC-503 - Algoritmos e Estruturas de Dados II

Prof. Dr. Gustavo E. de Almeida Prado Batista

**Projeto – Bitmap to ASCII-Art – Arquivos**

Andrey Gobbo - 7152503

Gabriel Fernandes Gonzales - 8936565

Marcos Adriano de Carvalho - 6791727

São Carlos 2017

**Introdução**

O Projeto constitiu no desenvolvimento de um programa que transforma uma imagem Bitmap (.bmp) para o formato “ASCII-Art”, arte usando caracteres da tabela ASCII.

O programa recebe um arquivo.bmp como parâmetro, faz a leitura do Header e dos dados pixel a pixel e faz um cálculo da intensidade da cor de um bloco de pixels de tamanho fixo MxN. Atribui, então, um caractere pré-definido de acordo com essa intensidade. Blocos com intensidade menor (mais escuros) são substituidos por caracteres mais preenchidos (como ‘#’ ou ‘$’) e blocos com intensidade maior (mais claros) são preenchidos com caracteres mais vazios (como ‘.’ e ‘ ‘).

**Estruturas de dados, Funções e funcionamento**

Temos duas estruturas de dados principais, que são usadas para se guardar os metadados lidos no Header do arquivo Bitmap.

Para ler os dados do Header principal temos a *struct* *Header*, composta por três variáveis *unsigned short int* para guardar os dados do tipo do arquivo e ler e descartar dois valores reservados do Header. Além delas temos duas variáveis *unsigned int* usadas para guardar o tamanho, em Bytes, da imagem Bitmap e o *offset* de dados, que é o ponto do arquivo onde começam os dados dos pixels.

Para ler os demais dados do Header temos a *struct Infoheader,* usada para ler salvar os metadados da imagem, como tamanho do header, número do plano de cores, o número de *bits per pixel* da imagem, entre outros dados. Nessa structo que nos interessa é: número de *bits per pixel* e os valores de altura e largura da imagem, em pixels, salvos nas variáveis *width* e *height*.

typedef struct {

unsigned short int type; /\* Magic identifier \*/

unsigned int size; /\* Tamanho do arquivo, em Bytes \*/

unsigned short int reserved1, reserved2;

unsigned int offset; /\* Offset até o começo da imagem, em Bytes \*/

} HEADER;

typedef struct {

unsigned int size; /\* Header size in bytes \*/

int width,height; /\* Largura e altura da imagem, em pixels \*/

unsigned short int planes; /\* Número de planos de cores \*/

unsigned short int bits; /\* Bits por pixel \*/

unsigned int compression; /\* Tipo de compressão - será descartado \*/

unsigned int imagesize; /\* Tamanho da imagem - será descartado \*/

int xresolution,yresolution; /\* Pixels por metro - será descartado \*/

unsigned int ncolours; /\* Número de cores \*/

unsigned int importantcolours; /\* Cores importantes \*/

} INFOHEADER;

Além das duas estruturas de dados temos duas funções, além da ‘main’, usadas para ajudar na leitura do header.

- Função ReadUShort: lê um pedaço do arquivo de tamanho *unsigned short char*(2 bits) e faz um cast para *unsigned short int*

- Função ReadUInt: lê um pedaço do arquivo de tamanho *unsigned char*(4 bits) e faz um cast para *unsigned int*

Essas funções são usadas para ler a primeira parte do header e salvar na *struct Header.*

Para ler os metadados da imagem e salvar em uma matriz de cor, que guarda a media das cores RGB de cada pixel, usamos um *fseek* passando como parâmetro o tamanho do *offset* definido no header da imagem.

Fazemos, então, a leitura do restante do arquivo bit a bit, utilizando a dica passada na especificação do trabalho para fazer a média das cores RGB de um pixel, usando o código:

fread(cbuf,1,infoheader.bits/8,fptr);

corPixel = ((short)cbuf[2]+(short)cbuf[1]+(short)cbuf[0])/3;

Após a leitura, a cor é salva na matrizCor.

Com essa matriz montada, fazemos agora a leitura da mesma para fazer a substituição do bloco lido por um caractere na saída padrão. Lemos essa matriz começando do canto inferior esquerdo, uma vez que a imagem bitmap de origem apresenta a imagem de ‘ponta cabeça’.

Fazemos um cálculo da media dos valores da matrizCor em um bloco MxN, como descrito na especificação, e substituimos por um caractere do mapa de ‘sombras’ definido:

{'#','$','O','=','+','|','-','^','.',' '} → cores mais escuras à esquerda, mais claras à direita.

**Como executar?**

Para compilar o programa basta rodar o comando a seguir na linha de comando:

gcc converToAscii.c -o convertToAscii

Será gerado um arquivo de saida com o nome *convertToAscii.*

Para executar, usar o comando:

./convertToAscii nome\_do\_arquivo.bmp

Recomendamos usar uma imagem que tenha no máximo 250 pixels de largura, para que a visualização fique ok em um monitor com 1920 pixels de largura. Caso coloque uma imagem maior, as linhas sairão quebradas no terminal de comando e a visualização ficará comprometida.

**Resultados, dificuldades e considerações**

Durante o desenvolvimento do programa, a leitura do Header foi o que tomou mais tempo, devido à diferença de tamanho dos dados contidos. No entanto, conseguimos fazer a leitura de todo o header, descartando os dados que não usuaríamos.

Após a leitura do header, o tratamento bit a bit foi feito sem problemas e paramos apenas na hora de imprimir a saída. Estávamos fazendo a leitura da matriz de cor normal, começando no canto superior esquerdo. Voltamos à especificação e vimos que falava que no bitmap os dados estão ‘de ponta cabeça’. Mudamos a leitura da matriz e a saída foi dada como esperado.

Testamos o programa com a imagem dada na especificação: “socrates.bmp”, além de duas outras imagens encontradas no google: Monalisa.bmp e Einstein.bmp. Nos três casos as saídas foram corretas, para um bloco MxN = 1x2.

No entanto, percebemos que trabalhar com blocos maiores de pixels não é uma boa solução. Blocos menores permitem um ajuste mais fino na substituição dos caracteres. O maior valor que vimos que apresenta um bom resultado é um bloco 3x4.