Processamento de Imagens

Processamento Gráfico

Sumário

- Introdução e conceitos
- Sistema de cores
- Estrutura de dados para imagens
  - Estrutura de dados, representação e manipulação de pontos da imagem
  - Indexação com paleta de cores.
- Algoritmos para manipulação de cor e pixel
- Algoritmos de PI
  - Quantização de cores
  - Composição de imagens
  - Transformações
  - Filtros de imagem
- Representação e Formatos de arquivo de imagens
  - Compressão de dados (RLE, LZW, JPEG e fractal)
  - Formatos de arquivo: imagens bitmap, vetoriais e de vídeo
- Estudo de caso: animação tipo *Color Cycling*

Introdução

- Processamento de imagens como ferramenta para diversas aplicações:
  - Reconhecimento de padrões
  - Tratamento e melhoramento de imagens
- Tipo de informação básica: imagens.
- Envolve manipulação imagens:
  - Estrutura de dados / representação
  - I/O + compressão de dados
  - Filtros de imagem
  - Transformações: rotação, escala e translação

Introdução

- Imagem como um tipo de dado
  - Representação para uma fotografia, ilustração, diagrama, desenho, textura ou pintura.
  - É o tipo de dado básico para de entrada processamento de imagens (manipulação de imagens, detecção, reconhecimento e visão computacional).
  - É o tipo de dado de saída em computação gráfica (síntese de imagens).
  - Muitas aplicações fazem uso de processamento gráfico e de algum tipo de dado gráfico.
  - Outras ainda exibem dados numéricos ou textuais como gráficos.

Introdução

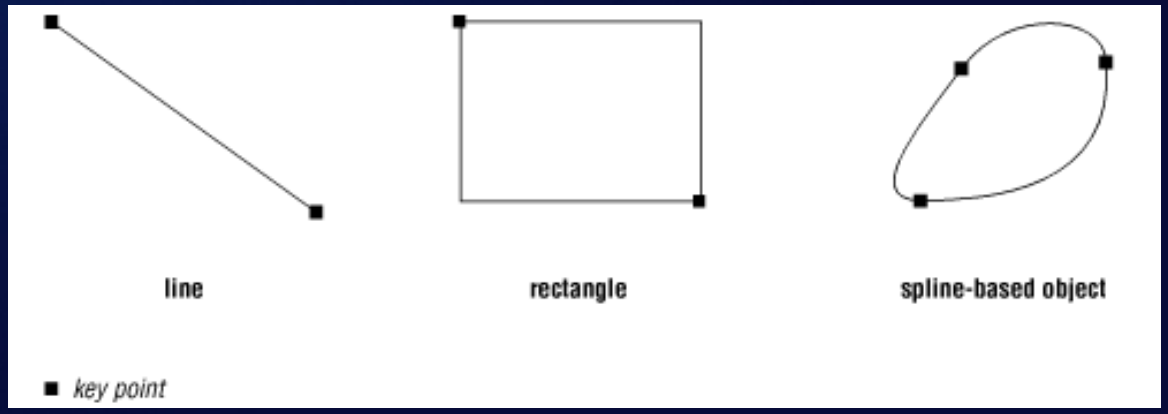
- Em jogos, imagens são muito utilizadas na forma de *sprites*, imagens de cenários e na composição da visualização de objetos, sendo utilizadas na forma de texturas (mapeamento de texturas).

Introdução

- Para a disciplina de processamento gráfico, estamos interessados nas seguintes questões:
  - Representação de imagens como estrutura de dados.
  - Visualização de imagens
  - Manipulação de imagens através de filtros de imagens
    - Modificação de tons, separação de canais, eliminação seletiva de cores e efeitos em geral.
    - Aplicação de transformações afins: translação, escala e rotação.
  - Desenho animado ou animação com *sprites* (através de sobreposição e composição de cena com cenário de fundo).
  - Transformações em imagens.
  - Representação de cenários de jogos com paleta de cores, *tiles*, *tilesets* e *tilemaps*
  - Armazenamento de imagens

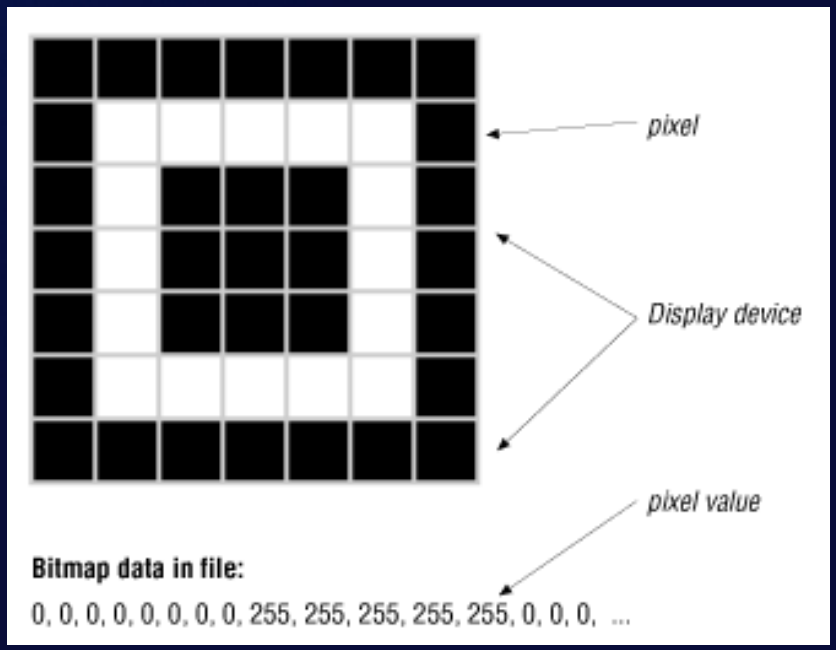
Conceitos

- Tipos de dados gráficos:
  - Dados vetoriais
    - São informações que descrevem primitivas gráficas para formar desenhos
    - Primitivas gráficas: pontos, linhas, curvas ou formas geométricas quaisquer.
    - Um programa que manipula este tipo de dado devem interpretar esta informação primitiva e transformá-la numa imagem.



Conceitos

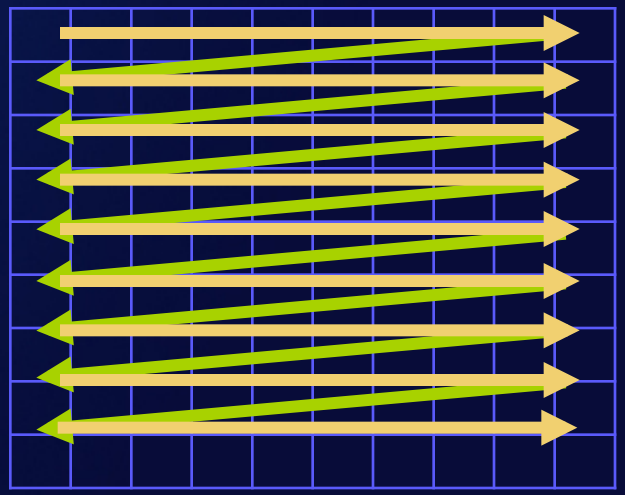
- Tipos de dados gráficos:
  - Dados tipo bitmap
    - Dado gráfico é descrito como uma array de valores, aonde cada valor representa uma cor.
    - Chamamos cada elemento da imagem de pixel.
    - O pixel é uma estrutura de dados que contém múltiplos bits para representação de cores.
    - A quantidade de bits determina a quantidade de cores que possível de se representar numa imagem.
      - 1, 2, 4, 8, 15, 16, 24 e 32 são quantidades comuns de bits para pixels



Conceitos

- Rasterização
  - Processo de desenho de uma imagem num dispositivo gráfico.
  - *Raster scan-lines*.
    - *Raster-images* é definido como um conjunto de em linhas de pixels (*scan-lines*). Linhas são formadas por colunas.

Rastar-scan order



```
for (int y=0; y<height; y++){
   for(int x=0; x<width; x++){
       putPixel(x, y, pixels[x][y]);
   }
}
```

Conceitos

- Imagem:
- Formalmente, uma imagem representa uma imagem abstrata, a qual é uma função variável contínua [1].
  - Função cujo domínio é um retângulo no plano Euclidiano.
  - Para cada f(x,y) é amostrado um valor de cor, em resumo um pixel.

f(x,y) = cor

f(172,101)=

Conceitos

- Imagem digital:
  - Apesar de o domínio da imagem abstrata ser contínuo, portanto, com infinitos pontos, trabalhamos com imagens discretas, ou seja, uma amostra com finitos pontos.
- Uma fotografia digital é uma amostra discreta de uma imagem real analógica.
  - Uma vez amostrados X pontos da imagem real, não há como obter pontos intermediários da imagem discreta, eles devem ser re-amostrados da real ou gerados computacionalmente.

Conceitos

Imagem abstrata ou imagem real ou imagem contínua

Qualidade da imagem não é afetada pelo aumento da resolução da amostragem da imagem contínua.

Conceitos

Imagem discreta ou imagem digital

Uma vez amostrada digitalmente, uma imagem perde qualidade sem tentarmos ampliar a sua resolução. Amostragem sobre amostragem.

Programando com imagens

- Computacionalmente, podemos representar uma imagem como um array de valores, onde cada valor é uma cor (pixel).
  - Pixel: *picture element*.
- Outras propriedades importantes da imagem são suas dimensões, chamadas: largura (*width*) e altura (*height*).
- E *depth* determina a quantidade de cores que o pixel de uma imagem pode representar.

Programando com imagens

- Pixel-depth
  - Pixels são representados por bytes e bytes são combinações de bits.
  - Cada bit pode armazenar uma informação binária, portanto, ou zero ou um.
  - Assim, quanto maior a profundidade de bits do pixel maior faixa de representação.
  - Por exemplo, com 3 bytes é possível armazenar 2^{24} valores diferentes dentro destes 3 bytes.

Conceitos

- Profundidade de cores:
  - Número máximo de cores que ser representadas. Expressa em bits:
    - 24 bits (RGB):
      - 1 byte (8 bits) para cada componente R, G e B
      - Total de 2^{24} cores = 16 Mega-cores = 16.777.216 de cores.
      - *True-color*: todas as cores que o olho humano consegue perceber.
    - 32 bits (RGBA ou ARGB):
      - Idem ao anterior, porém possui um componente a mais, o canal Alpha destinado a representação de opacidade/transparência da cor.
      - Pode ser utilizado também para representar mais valores por canal (RGB).
      - Máscaras: 10-10-10 (sobram 2) ou 8-8-8-8 (usando alpha)
    - 16 bits:
      - Número reduzido de cores, menor necessidade de armazenamento.
      - Total de 2^{16} = 64 K-cores = 65.536 cores diferentes
      - Máscaras: 5-5-5 (sobra 1) ou 5-6-5 (verde a mais por causa do brilho)

Conceitos

- Profundidade de cores:
  - Número máximo de cores que ser representadas. Expressa em bits:
    - 8 bits (*gray-scale* ou *indexed-color*)
      - Um byte para representar as cores: 2^8 = 256 cores possíveis (índices da paleta)
      - Utilizado também para representação em tons-de-cinza.
    - 4 bits (*indexed-color*, 2 pixels por byte)
      - Um byte representa dois pixels 2^4 = 16 cores possíveis, ou melhor, 16 índices possíveis na paleta.
    - 1 bit (*b&w* – preto e branco, imagem binária)
      - Preto e branco, duas possibilidades de cores, para bitmap
      - Um byte pode representar 8 pixels

Conceitos

- Resolução
  - Termo utilizado para representar as dimensões qualquer matriz de pontos (tela ou imagem, por exemplo)
  - Pode ser expressa tanto em número de pontos de largura e altura, quanto de quantidade total de pixels.
  - Exemplo:
    - 640 x 480, 800 x 600, 1280 x 1024, 256 x 256, ...
    - Ou: 1 Mega pixels, 3 Mega pixels, ... Neste caso, o produto da largura pela altura.

Programando com imagens

- Pixel
  - Pixels armazenam a informação de cor.
  - Formatos:
    - *Binary*: onde um bit é destinado a cada pixel e apenas imagens em preto-e-branco
    - *Indexed-color* ou *color-map*: o pixel armazena uma informação de índice para uma tabela de cores.
    - *Direct color*: representação numérica direta. Portanto, o inteiro armazena a cor, de acordo com algum modelo cor (esquema de definição de cor).

Programando com imagens

- Imagens binárias
  - Pixel pode ser representado com apenas um bit. Portanto, 8 pixels são representados com 1 byte.
  - Fizemos uso de operações binárias para acesso aos bits.
  - Para saber a cor de 8 pixels de uma linha:

```
BYTE pixel = pixels[x][y]; // 8 pixels/byte
BYTE px1 = (pixel & 0x80) >> 7; // 1000 0000
BYTE px2 = (pixel & 0x40) >> 6; // 0100 0000
BYTE px3 = (pixel & 0x20) >> 5; // 0010 0000
BYTE px4 = (pixel & 0x10) >> 4; // 0001 0000
BYTE px5 = (pixel & 0x08) >> 3; // 0000 1000
BYTE px6 = (pixel & 0x04) >> 2; // 0000 0100
BYTE px7 = (pixel & 0x02) >> 1; // 0000 0010
BYTE px8 = (pixel & 0x01);      // 0000 0001
```

Programando com imagens

- Imagens binárias
  - Para definir 8 pixels de uma linha:

```
BYTE pixel = (px1 << 7);  // ?000 0000
pixel |= (px2 << 6); // 0?00 0000
pixel |= (px3 << 5); // 00?0 0000
pixel |= (px4 << 4); // 000? 0000
pixel |= (px5 << 3); // 0000 ?000
pixel |= (px6 << 2); // 0000 0?00
pixel |= (px7 << 1); // 0000 00?0
pixel |= px8;        // 0000 000?
pixels[x][y] = pixel;
```

Programando com imagens

- Imagens tipo *indexed-color*
  - Paleta de cores: *Color map*, *index map*, *color table* e *look-up*.
  - Paleta é um conjunto pré-definido de cores.
  - Pixels **não** armazenam diretamente as cores que eles representam, mas sim um índice para a paleta de cores.
  - Número de cores possíveis para representação é reduzido.
  - Normalmente, as paletas possuem 2, 4, 16 ou 256 cores, representadas, respectivamente, por 1, 2, 4 ou 8 bits.
  - De longe o tipo mais comum de paleta é a de 8 bits ou um byte por pixel. Um informação de cor é representada por 3 bytes (RGB). A paleta ocupa :

    2^8 = 256 * 3 bytes = 768 bytes

Programando com imagens

- Imagens tipo *indexed-color*

Números da economia:

- Imagem de 100 x 200
- 24 bits: 200*200*3 ≈ 117 Kb
- Paleta 8 bits:
  - 768 + 200*200 ≈ 40 Kb

Programando com imagens

- Imagens tipo *indexed-color*
  - Para definir um índice para pixel:

```
BYTE index = getColorIndex(image[x][y],
  palette);
pixels[x][y] = index;
```

  - Para desenhar ou saber a cor do pixel:

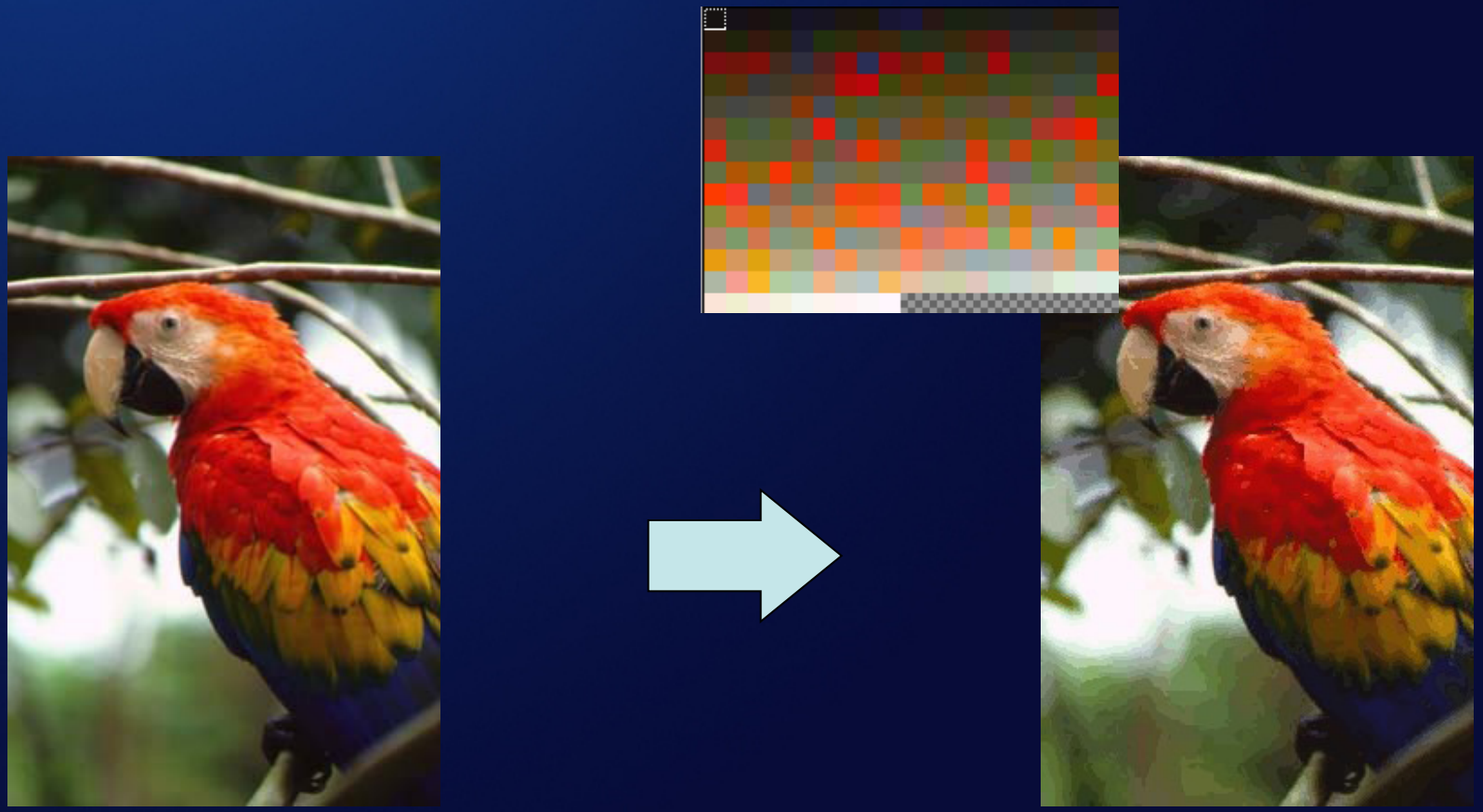
```
COLOR color = palette[pixels[x][y]];
putPixel(x, y, color);
```

Programando com imagens

- Imagens tipo *indexed-color*
  - As paletas podem ser pré-definidas pelos sistemas. O que retira a necessidade de armazenamento da mesma.
    - Como o padrão "*IBM VGA palette*": http://www.fileformat.info/mirror/egff/ch02_02.htm
  - Mas as melhores paletas são aquelas otimizadas a partir imagem que elas representam. Os *n* cores que melhor representam a imagem. Chamada de paleta otimizada.
  - Das 16 milhões de cores a paleta utiliza apenas 256 cores, obviamente, que há uma perda de qualidade de informação, em favorecimento a economia de armazenamento.
  - É necessário fazermos um processo de quantização de cores para definir as 256 melhores cores que compõem a nossa paleta de cores de uma determinada imagem.
  - Muitas vezes a redução não afeta a qualidade da imagem ou o resultado é aceitável.

Programando com imagens

- Imagens tipo *indexed-color*



Programando com imagens

- Imagens tipo *indexed-color*
  - Dimensão: 240x360 = 86400 pixels
    - Como cada pixel ocupa um byte (índice da paleta): 84 Kb
  - Usa uma paleta com 256 cores
    - Como cada cor ocupa 3 bytes: 768 bytes
  - 86400 + 768 = 85,125 Kb
  - Imagem RGB (24 bits, 3 bytes por pixel) ocupa: 240*360*3 = 253,125 Kb
  - Representação com paleta é ±3x menor

Programando com imagens

- Imagens tipo *direct-color*
  - Cores são armazenadas diretamente nos pixels.
  - Utiliza-se, normalmente, um número inteiro (4 bytes) para armazenar cada cor, portanto, uma imagem é uma matriz de inteiros. Chama-se *packed-bites*
  - Também podemos armazenar em 3 bytes subseqüentes (BYTE[3]). Mas é menos eficiente do que a abordagem anterior.

Programando com imagens

- Estrutura / classe imagem (C++) - com empacotamento de bits num int:

```
class Image {
public:
   Image(int w, int h){
      width = w; height = h;
      pixels = new int[w*h];
   }
   void setPixel(int rgb, int x, int y){
      pixels[x+y*width] = rgb;
   }
   int getPixel(int x, int y){
      return pixels[x+y*width];
   }
   int getWidth(){ return width; }
   int getHeight(){ return height; }
private:
   int *pixels; // alternativamente char *pixels - 1 byte por canal
                // neste caso, pixels = new char[w*h*3];
   int width, height;
}
```

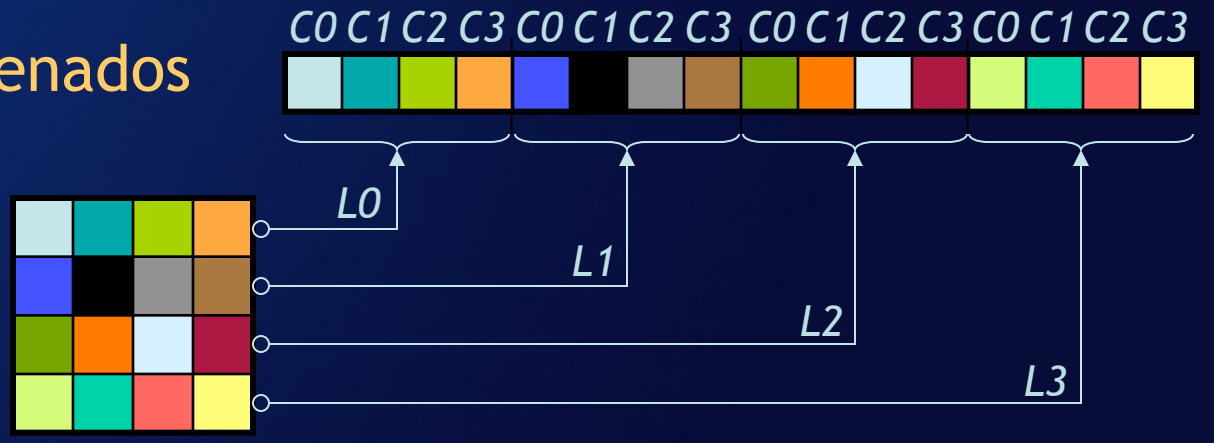
Programando com imagens

- Estrutura / classe imagem (C++) – SEM empacotamento de bits:

```
class Image {
public:
  Image(int w, int h, int channels){
     width = w; height = h; this->channels=channels;
     pixels = new int[w*h*channels];
  }
  void setPixel(char *rgb, int x, int y)
    for(int=0; i<channels; i++){
       pixels[i+x*channels + y*width*channels] = rgb[i];
    }
  }
  char* getPixel(int x, int y){
     char *rgb = new char[channels];
    for(int i=0; i<channels; i++){
      rgb[i]=pixels[i+x*channels+y*width*channels];
    }
    return rgb;
  }
  int getWidth(){ return width; }
  int getHeight(){ return height; }
  int getNumOfChannels(){ return channels; }
private:
  char *pixels;
  int width, height, channels;
}
```

Programando com imagens

- Pixels são armazenados como um array unidimensional:



- Acessando um pixel da imagem num vetor unidimensional:

```
pixels[ x + y * width ]
```

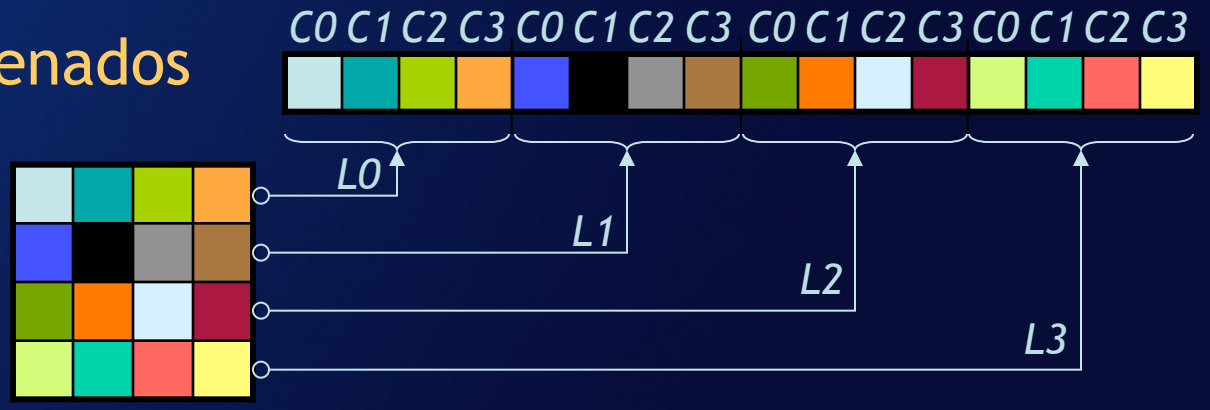
- Para descobrir a coordenada (x,y) do *n*-ésimo elemento do array:

```
x = n % width;
y = n / width;
```

Processo para bytes RGB dentro de um int

Programando com imagens

- Pixels são armazenados como um array unidimensional:



- Acessando um pixel da imagem num vetor unidimensional (para cada canal de cor num byte/num elemento do vetor):

```
pixels[c + x*channels + y*width*channels]
```

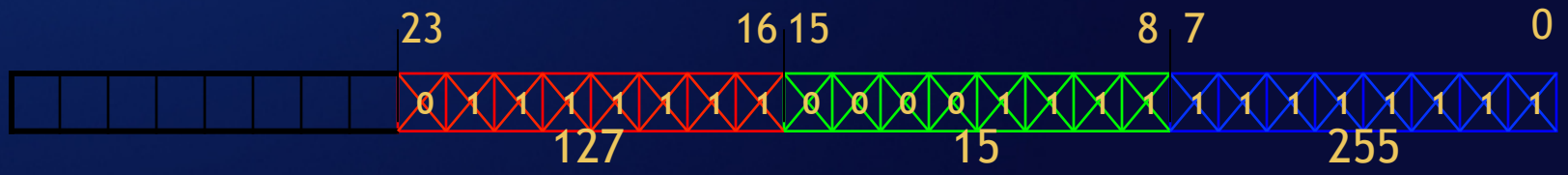
- Para descobrir a coordenada (x,y) do *n*-ésimo elemento do array:

```
n = n / channels;
x = n % width;
y = n / width;
```

Processo para bytes RGB armazenados separadamente: um byte por canal dentro do array de pixels

Programando com imagens

- Pixel como um valor RGB (empacotado num `int`):
  - Cada pixel representa um ponto ou valor de cor da imagem.
  - Um pixel é formado pelos canais de cor ou componentes de cor R (*red* - vermelho), G (*green* - verde) e B (*blue* - azul)
  - Como estamos representando um pixel como número inteiro, cada byte do inteiro (um int tem 32 bits ou 4 bytes) é utilizado para um canal.



O objetivo é colocar o valor de cada componente no seu conjunto de bits específico (direita para esquerda):
B - ocupa os primeiros 8 bits (0 a 7)
G - ocupa os próximos 8 bits (8 a 15)
R - ocupa os próximos 8 bits (16 a 23)

Este é o caso típico da linguagem Java. Para usar com C++/OpenGL, passe formato GL_BGRA_EXT:

```
glDrawPixels(image->getWidth(), image->getHeight(), GL_BGRA_EXT, GL_UNSIGNED_BYTE,
    image->getPixels());
```

Programando com imagens

- Definindo os bits correspondentes a cada canal num inteiro:

```
// por exemplo, para a cor 100, 95, 255
int rgb = (100 << 16) | (95 << 8) | 255;
```

- Repare que devemos saltar/deslocar os bits anteriores ao canal com operação *shift-left* (<<). Para o canal azul não é necessário porque ele é o primeiro. Por fim, é necessário juntar os três valores com o operador binário *or* (|).

Programando com imagens

- Canal Alfa (*alpha channel*):
  - Representa a opacidade de um ponto. O quanto ele está visível.
  - É forma como podemos aplicar "transparência" os pontos da imagem.
  - Valores de `0` a `255`
    - `0` - totalmente transparente
    - `255` - totalmente opaco
    - Valores intermediários para níveis de transparência.
  - Normalmente, o canal alpha é utilizado para combinação de cores de imagens sobrepostas. Exemplo:
    - Considerando que o ponto (X,Y) da imagem 1 tem cor (90,90,250) e que o ponto correspondente na imagem 2 tem cor (240, 200, 100). Considerando ainda uma transparência de 50%:

Fórmula para mistura:

$C = c1*(1-\alpha) + c2*\alpha$

No exemplo, para o canal B:

$C = 250*(1-0.5) + 100*0.5$

$C = 250*0.5 + 100*0.5$

$c = 125 + 50 = 175$

Programando com imagens

- Definindo os bits correspondentes a cada canal num inteiro, considerando também o canal alpha:

```
// por exemplo, para a cor 100, 95, 255
// considerando que deve ser opaco a=255
int argb=(255<<24)|(100<<16)|(95<<8)|255;
```

- A única diferença é que o canal alpha ocupará os 8 bits mais a esquerda e, portanto, deve saltar/deslocar os 24 bits anteriores.

Este é o caso típico da linguagem Java. Para usar com C++/OpenGL, passe formato GL_BGRA_EXT:

```
glDrawPixels(image->getWidth(), image->getHeight(), GL_BGRA_EXT, GL_UNSIGNED_BYTE,
    image->getPixels());
```

Programando com imagens

- Obtendo os canais de cor e canal alpha a partir de um pixel:

```
// considerando que deve ser opaco a=255
int a = (argb >> 24) & 0xff;
int r = (argb >> 16) & 0xff;
int g = (argb >> 8)  & 0xff;
int b =  argb        & 0xff;
```

- Deve-se fazer o processo inverso: voltar ou reduzir os bits até os primeiros 8 bits com *shift-right* (>>) e deve-se "limpar" os demais, fazendo um *and* binário (`&`) com 255 (`0xff` em hexadecimal).

Composição de imagens

- A composição é utilizada para compor uma cena de fundo com objetos de primeiro plano.
- Similar ao “homem-do-tempo” (ou “mulher-do-tempo”). O ator é filmado sobre um background (fundo) de cor constante. Este fundo é removido (chroma-key). Por fim, a imagem ator é apresentado sobre um fundo que é o mapa do tempo. Esta técnica chama-se *image composite*.
- Faz uso do canal alpha nos pontos da imagem para indicar aonde há transparência (contribuição maior do fundo) e aonde há opacidade (contribuição maior primeiro-plano)

Composição de imagens

Imagem "foreground"

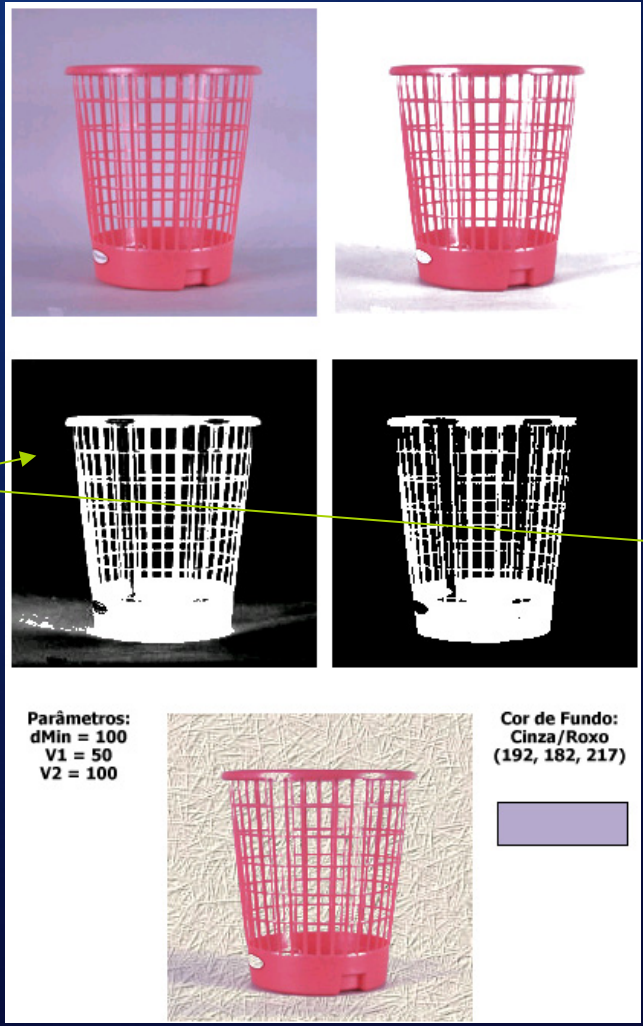
Imagem "background"

Imagem Resultado (combinação)

Imagem Resultado Processo

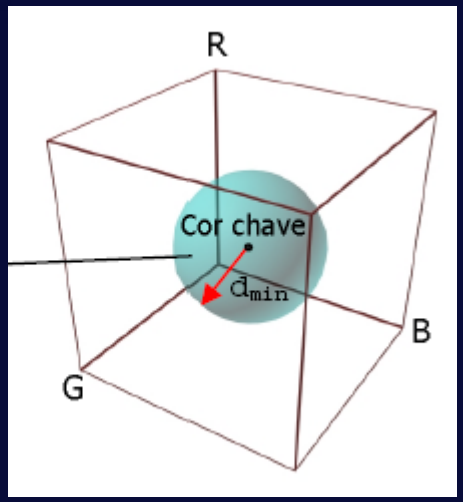
Composição de imagens

Imagem do canal alfa. Opaco= 255, transparente=0



Composição de imagens

- Algoritmos para chroma-key:
  - Trivial (ineficiente) – para fundo azul:
    - `(b>r) && (b>g)`
    - Uma variação possível para melhorar a eficiência deste filtro é colocar um parâmetro de contribuição mínima do azul para ser maior que o R e o G. Exemplo:
      - `((b-10)>r) && ((b-10)>g)`
  - Bergh & Lalioti (1999) – para fundo azul:
    - `d = 2b - r - g > dmax`
  - Tonietto & Walter (2000) – Distância Euclidiana (qualquer cor de fundo):
    - $d = ((r-br)^2 + (g-bg)^2 + (b-bb)^2 < d_{min})$

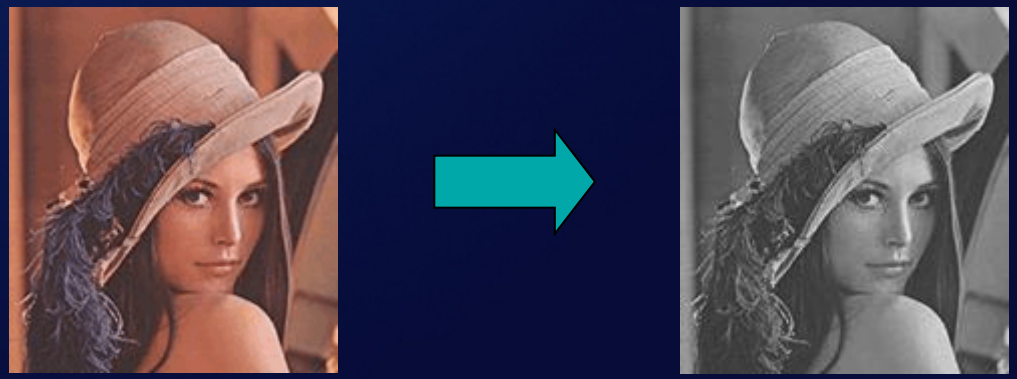


Filtros de imagem

- Filtros de imagem são funções que manipulam a informação de cor dos pontos da imagem.
- São utilizados para fins artísticos, melhoria da imagem ou como pré-processamento para outros algoritmos de PI.
- Exemplos: brilho/contraste, correção gamma, sharpen, blur, sépia, ...

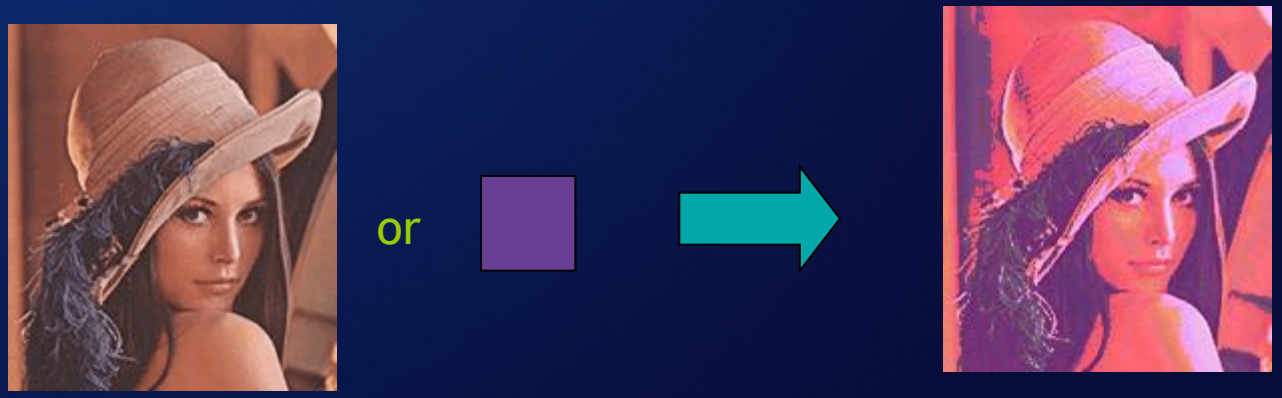
Filtros de imagem

- Conversão de imagem RGB para gray scale:
  - Trivial (média):
    - `gray = r*0.333 + g*0.333 + b*0.333`
  - E acordo com a sensibilidade do olho-humano:
    - `gray = r*0.2125 + g*0.7154 + b*0.0721`



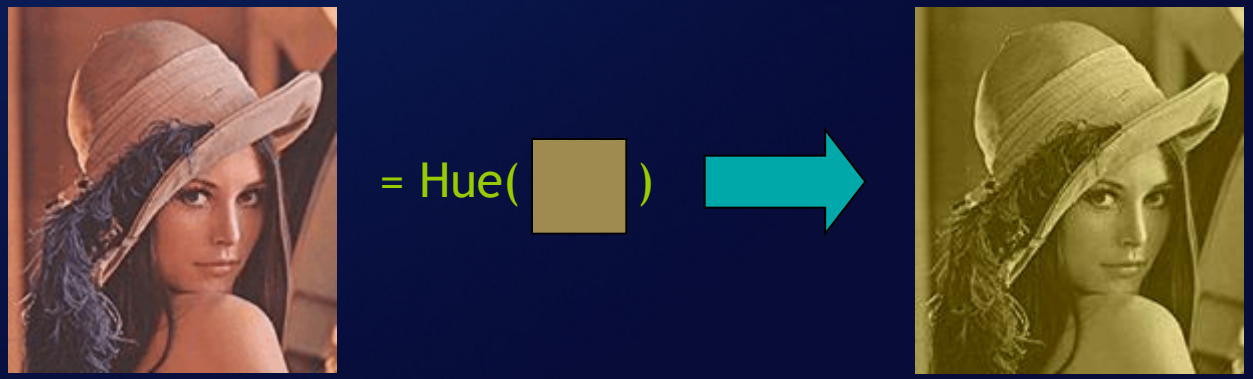
Filtros de imagem

- Colorização de imagem:
  - `newrgb = rbg | rgbModifier`



Filtros de imagem

- Ajuste da componente Hue (HSV):
  - Converter cada ponto de RGB para HSV
  - Dada uma cor modificadora, converter para HSV e obter o seu valor Hue.
  - Aplicar substituir o Hue do ponto atual pelo Hue da cor modificadora.
  - Converter de HSV para RGB



Filtros de imagem

- Imagem “negativo”:
  - Deve-se aplicar para cada canal:
    - canal ^ 255; // operação binária XOR
    - Uma variante é aplicar 1 ou 2 canais apenas

^ 255

Formatos de imagem

- Formato de imagem é forma como a imagem é gravada em disco.
- O objetivo principal é, dado um objeto imagem na memória, como gravar um arquivo que contenha toda a informação necessária para recriar o mesmo objeto em memória.
  - As informações mínimas para gravação são: largura, altura e os pixels da imagem.
- A imagem é um tipo de dados que ocupa grande espaço de armazenamento. Portanto, é necessária uma estratégia de gravação dos dados, a fim de obter um arquivo menor possível considerando-se o resultado desejado para recuperar em outro momento.

Formatos de imagem

- São divididos em duas categorias:
  - Sem compressão: PNM e variação principal do BMP.
  - Com compressão: PNG, JPG, ...
- Os métodos com compressão de dados tem por objetivo eliminar informação "desnecessária" para o armazenamento.
  - Não gravar dados que podem ser reproduzidos/calculados pelo algoritmo no momento da leitura.

Formatos de imagem

- A idéia básica é agrupar pixels de cores semelhantes numa informação menor.
- Dentre os métodos com compressão podemos classificá-los como:
  - Compressão sem perda de informações:
    - Os dados são comprimidos, mas nenhuma informação é perdida. Porém a taxa de compressão é muito menor.
    - Exemplo: BMP, PCX, TIFF, ...
  - Compressão com perda:
    - Cores semelhantes são agrupadas de acordo com um nível de tolerância (mesmo cálculo de distância de cores).
    - Desta forma, mais pixels são agrupados e, conseqüentemente, maior a taxa de compressão.
    - Exemplo: JPG

Formatos de imagem

- Alguns formatos que estudaremos:
  - PNM – formato muito simples, sem compressão, sem canal alfa, permite imagens binárias, tons-de-cinza e coloridas (RGB).
  - PCX – formato de complexidade média, permite canal alpha e compressão RLE. Imagens binárias, indexadas e RGB.
  - BMP – formato mais complexo, permite dois tipos de compressão RLE4 e RLE8, imagens indexadas a 4 e 8 bits de profundidade, permite máscara de bits e canal alfa.

Formatos de imagem

- Formato PNM:
  - O formato PNM é um tipo de arquivo que reúne a especificação de 3 tipos formatos de imagem específicos:
    - PBM – imagens binárias
    - PGM – imagens em tons-de-cinza
    - PPM – imagens coloridas (RGB)
  - Todos os formatos podem ser gravados em modo texto ou em modo binário.
  - Maiores detalhes sobre o formato em [6].

Formatos de imagem

- Resumo do formato PNM:
  - Partes do arquivo:
    - Cabeçalho: que contém tipo de arquivo dimensões da imagem, valor máximo das componentes e comentários (opcional).
      - Cabeçalhos para cada tipo:
        - » PBM: P1 (modo texto) e P4 (modo binário)
        - » PGM: P2 (modo texto) e P5 (modo binário)
        - » PBM: P3 (modo texto) e P6 (modo binário)
      - Comentários são linhas que começam com caractere #
    - Pixels da imagem: seqüência de valores R, G e B.
      - A cada três valores lidos um pixel.

Formatos de imagem

- Exemplo de arquivo PPM em modo texto:

Cabeçalho → `P3`

Comentário → `# feep.ppm`

Dimensões → `4 4`

Max value → `15`

Pixels →

```
0  0  0    0  0  0    0  0  0    15 0 15
0  0  0    0 15  7    0  0  0    0  0  0
0  0  0    0  0  0    0 15  7    0  0  0
15 0 15    0  0  0    0  0  0    0  0  0
```

Formatos de imagem

- Formato PCX:
  - Formato antigo, utilizado no MS-DOS. Ver [7].
  - Possui diversas versões oficiais, a última e a suportada atualmente pela maioria das aplicações é 3.0
  - Separação do arquivo:
    - File header: contém byte de identificação do formato, versão, tipo de compressão, profundidade de cores, dimensões da imagem e etc.
    - Paleta de cores: quando o arquivo é do tipo indexado (8 bits) pode vir junto a paleta, que uma seqüência de RGB.
    - Dados da imagem: os dados são gravados por linha e um canal inteiro por vez. Por exemplo, primeiro são gravados todos os vermelhos da linha, depois todos os verdes e depois todos os azuis.
    - Exemplo:

R R R R R R R R R R R R
G G G G G G G G G G G G
B B B B B B B B B B B B — *Primeira linha*

R R R R R R R R R R R R
G G G G G G G G G G G G
B B B B B B B B B B B B — *Segunda linha*

...

Organização do arquivo:

Formatos de imagem

- Formato PCX cabeçalho padrão:

```
typedef struct _PcxHeader {
    BYTE   Identifier;          /* PCX Id Number (Always 0x0A) */
    BYTE   Version;             /* Version Number */
    BYTE   Encoding;            /* Encoding Format */
    BYTE   BitsPerPixel;        /* Bits per Pixel */
    WORD   XStart;              /* Left of image */
    WORD   YStart;              /* Top of Image */
    WORD   XEnd;                /* Right of Image
    WORD   YEnd;                /* Bottom of image */
    WORD   HorzRes;             /* Horizontal Resolution */
    WORD   VertRes;             /* Vertical Resolution */
    BYTE   Palette[48];         /* 16-Color EGA Palette */
    BYTE   Reserved1;           /* Reserved (Always 0) */
    BYTE   NumBitPlanes;        /* Number of Bit Planes */
    WORD   BytesPerLine;        /* Bytes per Scan-line */
    WORD   PaletteType;         /* Palette Type */
    WORD   HorzScreenSize;      /* Horizontal Screen Size */
    WORD   VertScreenSize;      /* Vertical Screen Size */
    BYTE   Reserved2[54];       /* Reserved (Always 0) */
    } PCXHEAD;
```

Formatos de imagem

- Formato PCX cabeçalho padrão:

Planes	Bit Depth	Display Type
1	1	Monochrome
1	2	CGA (4 colour palleted)
3	1	EGA (8 color palletted)
4	1	EGA or high-res. (S)VGA (16 color palletted)
1	8	XGA or low-res. VGA (256 color palletted/greyscale)
3 or 4	8	SVGA/XGA and above (24/32-bit "true color", 3x greyscale planes with optional 8-bit alpha channel)

Formatos de imagem

- Estudo de caso formato BMP:
  - Formato de imagem que permite gravação de imagens com indexação, compressão (ou não) e máscara de bits.
  - Formato padrão do MS-Windows e é utilizado internamente como estrutura para representação das imagens.
  - Versões mais atuais: 2, 3, 3-NT e 4
  - Visão geral do formato (maiores detalhes em [5]):
    - *File Header*: descreve informações a respeito do arquivo, como: cabeçalho padrão, tamanho do arquivo e início da área de dados da imagem.
    - *BMP Header*: dados sobre a seção de bytes da imagem: profundidade de cores, dimensões, método de compressão, máscaras e etc.
    - *Paleta*: entradas da paleta de cores conforme o número de bits estipulado.
    - *Image data*: área de dados aonde estão os bytes da imagem (pixels), com ou sem compressão.

Formatos de imagem

- Estudo de caso formato BMP:
  - Tipos de imagem que podem ser gravadas no BMP:
    - 1 bit – binária, um byte representa 8 pixels
    - 4 bits – indexada, um byte representa 2 pixels. Cada valor lido da área da imagem representa um índice da paleta de cores (16 cores possíveis). Permite compressão RLE4.
    - 8 bits – indexada, um byte por pixel. Cada valor é um índice da paleta (256 cores). Permite compressão RLE8.
    - 24 bits – RGB de cada ponto gravado diretamente, sem indexação. Não permite compressão nem alfa. 3 bytes por pixel.
    - 16 bits – modo máscara. Junto com o BMP header são fornecidas máscaras para cada canal RGB. Um pixel ocupa 2 bytes. Máscaras comuns são: 5-5-5 ou 5-6-5.
    - 32 bits – também modo máscara. Além das máscaras RGB também é definida a máscara *alpha*. Um pixel ocupa 4 bytes. Máscaras comuns são: 8-8-8-8 e 10-10-10.

Formatos de imagem

- Estudo de caso formato BMP:
  - A compressão genérica RLE é simples:
    - O primeiro byte é o *runcount* e o segundo o *runvalue*. Então, quando da leitura, deve-se ler um *runcount* que indica quantas vezes deve-se repetir o *runvalue* (lido a seguir). Desta forma é possível repetir valores (*runvalues*) até 255 vezes (*runcounts*).
    - Exemplo:
      - **`04 FF`** → significa que o valor `255` (`FF`) deve ser repetido 4 vezes: `FF FF FF FF` ou `255 255 255 255`
    - A vantagem está para imagens com poucas cores, com grandes regiões da mesma cor. Note que uma linha inteira da imagem pode ser gravada com dois bytes, no melhor dos casos.
    - Problema maior: quando a informação não se repete. Neste caso, corre-se o risco de representar um byte com dois bytes, portanto, ao invés de comprimir estaria se inflando a imagem desnecessariamente. Exemplo, bytes originais seriam: `FF AD 11 23`, porém com "compressão" RLE: **`01 FF 01 AD 01 11 01 23`**
    - Normalmente, algoritmos fazem contornos para este problema.

Formatos de imagem

- Estudo de caso formato BMP – compressão RLE8 e RLE4:
  - A RLE8 em linhas gerais (para maiores detalhes veja [5]):
    - Ainda é mantida a idéia do *runcount* e do *runvalue*, entretanto, quando não é possível comprimir, ou seja, bytes não repetidos, o algoritmo coloca *runcount* `00`, *runvalue* com número de bytes e finaliza linha de bytes não comprimidos com `00`:
      - `04 A1` → `A1 A1 A1 A1` (trecho com compressão)
      - `00 05 B1 10 AD EF B1 00` → `B1 10 AD EF B1` (sem compressão)
    - Desta maneira, diminui muito o problema de inflar os dados.
  - A RLE4 segue a mesma linha, porém os dados são definidos como 2 pixels por byte. Exemplo:
    - `04 A1` → `A 1 A 1` (repete-se os valores A e 1 "quatro" vezes de forma alternada)
    - `00 05 B1 AD E0 00` → `B 1 A D E` (são cinco valores não comprimidos e, por ser seqüência ímpar, é gravado um zero na segunda parte do último byte)

Próximos passos

- Filtros de melhoria da imagem:
  - Brilho/contraste e gamma
- Filtros de convolução:
  - Sharpen, blur e edge detection
- Transformações afins em imagens

Exercício

- Implementar os filtros vistos em aula para executarem no editor de imagens de exemplo do professor.
- Implementar mais algum filtro de sua escolha.
- Criar uma classe imagem e carregar dados para um objeto imagem. A carga pode ser feita a partir de arquivo. Um PPM, por exemplo.

Trabalho para o GA

- O grupo deve estudar e apresentar algum formato de imagem ou vídeo para a turma.
- O formato deve ser aceito pelo professor.
- O trabalho compreende em gerar uma apresentação que contenha a especificação do formato estudado e explicação do algoritmo.
- Não é necessário fazer a implementação do algoritmo.
- Exemplos: JPEG, GIF, PNG, TGA, TIFF, PSD, EPS, RGB, MPEG, AVI e etc.

Referências bibliográficas

1. FOLEY, J.D. et al. **Computer graphics: principles and practice**. Reading: Addison-Wesley, 1990.
2. ACHARYA, Tinku; RAY, Ajoy K. **Image Processing: Principles and Applications**. Wiley Interscience. 2005.
3. WRIGHT Jr., Richard S.; LIPCHAK, Benjamin; HAEMEL, Nicholas. **OpenGL Superbible: Comprehensive Tutorial and Reference**. 4ed.: Addison-Wesley. 2007.
4. TONIETTO, Leandro; WALTER, Marcelo. **Análise de Algoritmos para Chroma-key**. Unisinos, 2000.
5. http://www.fileformat.info/format/bmp/egff.htm
6. http://www.fileformat.info/format/pbm/egff.htm
7. http://www.fileformat.info/format/pcx/egff.htm