Laboratórios de Informática 2014/2015
DETI - Universidade de Aveiro

Docentes
João Paulo Barraca <jpbarraca@ua.pt>
Diogo Gomes <dgomes@ua.pt>
João Manuel Rodrigues <jmr@ua.pt>
Mário Antunes <mario.antunes@ua.pt>

TEMA 21

Representação de Informação Visual

Objetivos:
- Representação de cor nos diferentes espaços
- Efeitos básicos sobre imagens
- Marcas de água

21.1 Introdução

As imagens gráficas, tais como fotografias e gráficos são deveras importantes para as aplicações computacionais actuais. Desta forma torna-se importante analisar como a informação visual é representada e como pode ser processada.

Para a realização deste guião será necessário instalar a biblioteca **`Pillow`**, podendo esta ser instalada através do comando **`apt-get install python-imaging`**. Sendo este o método preferencial.

Em alternativa pode-se utilizar o comando **`pip install Pillow`**. No entanto, antes da instalação é necessário instalar várias dependências:

Para *Ubuntu*:

```
sudo apt-get install libtiff4-dev libjpeg8-dev zlib1g-dev\
    libfreetype6-dev liblcms2-dev libwebp-dev tcl8.5-dev tk8.5-dev python-tk
```

Para *OS X*:

```
brew install libtiff libjpeg webp little-cms2
```

Neste guião serão sugeridos processos que são sub-ótimos de forma a aumentar a compreensão do processo. A biblioteca `Pillow` possui transformações otimizadas para muitos dos processos aqui tratados, devendo código de produção fazer uso dos métodos disponíveis. A documentação da biblioteca pode ser encontrada no endereço `http://pillow.readthedocs.org/en/latest/reference`.

21.2 Imagens

As imagens são representadas numa forma matricial de linhas e colunas a que se dá o nome de geometria da imagem. A cada ponto da matriz dá-se o nome de píxel, sendo que uma imagem é sempre definida por uma largura e altura, medida em número de píxeis (ver Figura 21.1).

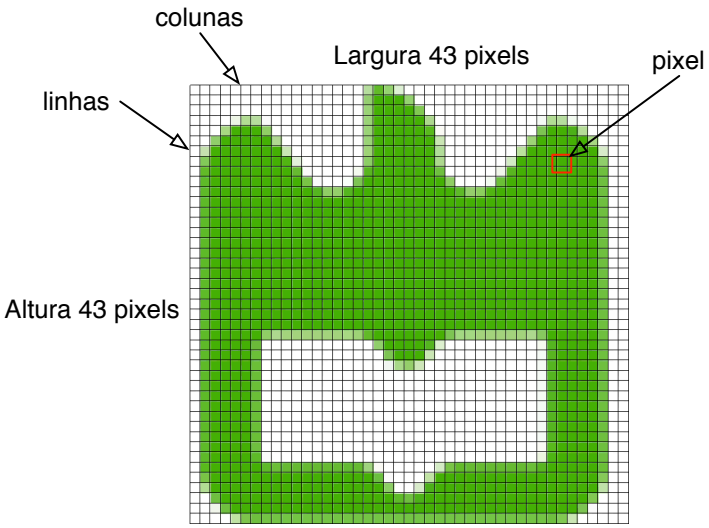

Figura 21.1: Geometria de uma imagem.

Além da informação para representação da imagem, a maioria dos formatos de representação adiciona dados extra aos ficheiros, sendo estes utilizados para indicar como

deve o ficheiro ser processado ou que codificação é utilizada. Alguns outros, como o formato Joint Photographic Experts Group (JPEG), permitem ainda a existência de dados num formato denominado Exchangeable image file format (Exif), indicando entre outros aspetos que programa foi utilizado, ou qual a máquina fotográfica utilizada e os parâmetros da lente (no caso de uma fotografia). Também por vezes pode ser incluída uma pré-visualização da imagem para apresentação no navegador de ficheiros do sistema.

De uma forma programática em *Python* é possível inspeccionar esta informação através da biblioteca **`Pillow`**, tal como demonstrado no exemplo que se segue:

```
from PIL import Image
from PIL import ExifTags
import sys

def main(fname):
    im = Image.open(fname)

    width, height = im.size

    print "Largura: %dpx" % width
    print "Altura: %dpx" % height
    print "Formato: %s" % im.format

    tags = im._getexif()

    for k,v in tags.items():
        print str(ExifTags.TAGS[k])+" : "+str(v)


main(sys.argv[1])
```

Exercício 21.1

Analise os ficheiros fornecidos pelos docentes utilizando o código anterior.

Uma das propriedades básicas da imagem é a sua dimensão. Este dimensão, representada em píxeis, pode ser alterada, gerando ora novas imagens mais pequenas ou maiores. De notar que o tamanho da imagem em píxeis pode não estar relacionado com o tamanho como ela é apresentada num monitor. Certamente se recorda que, por exemplo, quando se utiliza HyperText Markup Language (HTML)[1] é elemento **`<img>`** permite especificar qual o tamanho da imagem.

Também deve considerar que este processo é destrutivo e não cria informação. Ou seja, ao reduzir a dimensão de uma imagem, está-se a perder informação. Ao aumentar a dimensão de uma imagem não se está a aumentar a informação (definição) da imagem.

Podem-se criar imagens com diversos tamanhos usando o código seguinte:

```
...
def main(fname):
    im = Image.open(fname)
    width, height = im.size

    for s in [0.2, 8]:
        dimension = ( int(width*s), int(height*s) )
        new_im = im.resize( dimension, Image.NEAREST)
        new_im.save(fname+'-%.2f.jpg' % s)

...
```

Exercício 21.2

Implemente o código anterior e experimente modificar a dimensão de vários ficheiros.

Exercício 21.3

O método utilizado para modificar as imagens é um dos mais simples (**`Nearest Neighbour`**), sendo que existem vários outros, tais como **`BILINEAR`**, **`BICUBIC`** ou **`ANTIALIAS`**. Uns são mais adaptados à redução, outros à ampliação. Teste-os e compare visualmente o resultado.

21.3 Formatos de ficheiros

Existem vários formatos para a representação de imagens, sendo que cada um é optimizado para um tipo particular de informação. O formato mais utilizado para representar fotografias é sem dúvida o formato JPEG, sendo os conteúdos *Web* como gráficos utilizam sobretudo o formato Portable Network Graphics (PNG). Um outro formato popular é o Tagged Image File Format (TIFF), especialmente nos meios criativos, ou o BitMaP image file (BMP) em sistemas mais antigos. Existem razões concretas para a preferência de um formato sobre o outro, devendo-se principalmente ao tipo de compressão utilizada

em cada um deles e como representam a informação.

No caso em concreto da compressão, os formatos BMP, PNG e TIFF não aplicam compressão substancial, ou aplicam uma compressão em que não se perde qualquer informação. Por outro lado, o formato JPEG é otimizado para a representação de imagens como fotografias, criando artefactos quando utilizado para representar informação com cores sólidas.

O formato JPEG possui uma escala de compressão que varia em 0 e 100, sendo que quanto mais baixo for o valor, maior compressão e maiores perdas serão observadas. A Figura 21.2 demonstra o resultado de utilizar uma compressão de 30% para uma imagem com cores sólidas. Como se pode notar, foram adicionados muitos erros à imagem, sendo que é claro que o algoritmo processa as imagens em blocos.

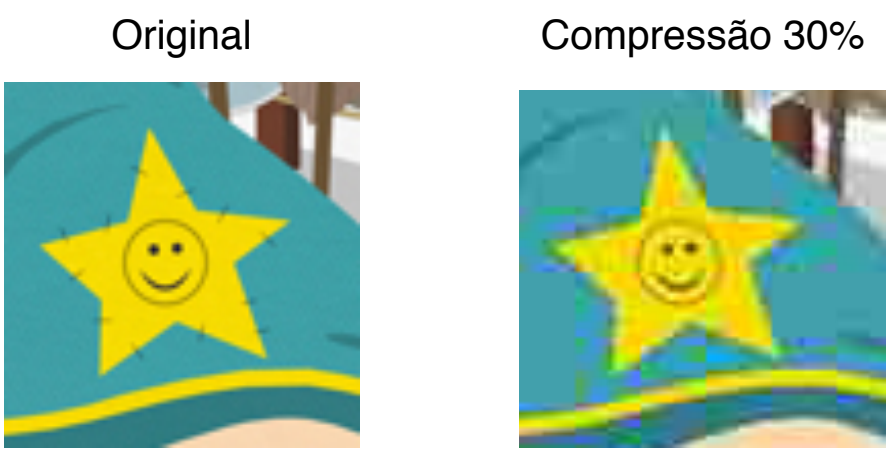

Figura 21.2: Compressão com JPG.

Exercício 21.4

Determine o tamanho dos blocos utilizados para comprimir a imagem anterior. Recomenda-se que simplesmente aumente o zoom do visualizador e conte os píxeis. A imagem possui uma geometria de 90px por 88 px.

O exemplo que se segue comprime o mesmo ficheiro com 11 valores de qualidade distintos e pode ser utilizado para verificar este aspeto.

```
from PIL import Image
import sys

def main(fname):
    im = Image.open(fname)

    for i in [1, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100]:
        im.save(fname+"-test-%i.jpg" % i, quality=i)

main(sys.argv[1])
```

Exercício 21.5

Utilize o exemplo anterior para criar versões comprimidas do ficheiro **`southpark.png`** e do ficheiro **`vasos.jpg`**. Determine para ambos os casos a partir de que valor de qualidade os erros adicionados perturbam a imagem. Tenha em especial atenção as áreas de alto contraste, tais como linhas.

Exercício 21.6

Tendo em consideração um qualquer ficheiro fornecido com a extensão JPEG e crie uma versão em cada um dos formatos: PNG, TIFF, BMP. Compare o tamanho do ficheiro criado.

21.4 Representação de cor

Existem várias formas como uma cor pode ser representada. A escolha do formato mais apropriado depende do tipo de imagem que se possui e do objectivo da informação. O método mais comum, e tratado até agora nas aulas anteriores (ex, HTML) é o formato **`RGB`**. Segundo este formato cada cor é representada por 3 componentes, respectivamente: Vermelho (R), Verde (G), e Azul (B). A Figura 21.3 apresenta alguns valores **`RGB`** do logotipo da Universidade de Aveiro.

As cores **`RGB`** são denominadas por cores primárias aditivas. Ao se somarem estas cores é possível reconstruir todas as outras. Em contrapartida, o modelo **`CMYK`** (Cyan, Magenta, Yellow, Black), constituído pelas cores primárias subtrativas também é capaz de formar todas as cores mas quando elas são subtraídas. Num ecrã os píxeis são iluminados,

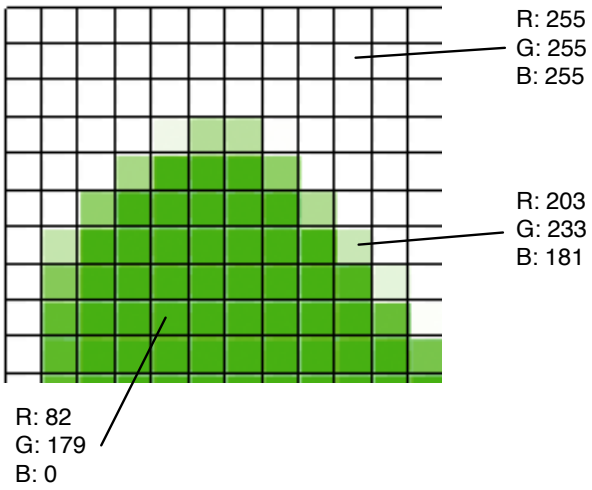

Figura 21.3: Valores RGB do logo da UA

portanto as cores somam a sua intensidade e é utilizado o modelo `RGB`. Numa impressora em que as cores depositadas num papel subtraem-se é utilizado frequentemente o modelo `CMYK`.

Existem vários modos de representação de cor que podem ser comumente utilizados:

- `BW` ou `1`: 1 canal com 1 bit apenas representando a existência de informação ou não. Utilizado para representar imagens a preto e branco.
- `RGB`: 3 canais, utilizando Vermelho, Verde e Azul. Muito utilizado para a representação de cores em monitores.
- `RGBA`: O mesmo que `RGB` com um canal adicional para representar a transparência. Suportado por alguns formatos como o PNG.
- `CMYK`: 4 canais, utilizando Ciano/Cião, Magenta, Amarelo e Preto. Muito utilizado para a representação de cores para impressão.
- `L`: Apenas um canal representando a Luminância (intensidade de luz). Utilizado para representar imagens em tons de cinza.
- `P`: Paleta de cores específica. Cada píxel da imagem aponta para um número de cor e existe uma paleta que informa como essa cor é representada. Utilizada para formatos como o Graphics Interchange Format (GIF).

· **YCbCr**: 3 canais representando a Luminância (Y), Crominância Azul (Cb), Crominância Vermelha (Cb). Os canais Cr e Cb são construídos da forma $Cr = R - Y$, sendo portanto a indicação de diferença entre o valor Vermelho e a Luminosidade da imagem. O sistema é utilizado para fotografias (JPEG) e em video.

Exercício 21.7

Verifique o atributo **mode** de uma imagem para cada um dos ficheiros fornecidos.

É possível converter as imagens entre modos de representação. O que pode apresentar algumas vantagens do ponto de vista de representação e processamento. Para casos de utilização dos modos de cor, consulte e secção 21.5.

De notar que na Figura 21.3 os valores apresentados consideram que existem 256 valores para cada cor, sendo que o total de cores será igual a 2^{3*8}. O número pode parecer grande mas na realidade não é adequado para todos os fins. As máquinas fotográficas atuais produzem imagens *RAW* com resoluções de 14 a 16 bits por cor, sendo que depois estas imagens são convertidas para o formato 8bits. No processo perde-se informação, pelo que alguns formatos como o TIFF permite utilizar ficheiros de 8bits, 16bits e 32bits. Outros como o Flexible Image Transport System (FITS) permite um número indeterminado de bits por píxel.

Não sendo possível analisar em concreto e de forma fácil o resultado de se terem imagem de 16bits, pode-se fazer o exemplo contrário: restringir o número de bits de uma imagem e observar como isso altera as cores percebidas pelo olho humano. O exemplo seguinte anula 4 bits a cada cor (coloca a 0), efectivamente reduzindo o ficheiro para 4bits por cor.

```
...

def main(fname):
    im = Image.open(fname)

    width, height = im.size

    for x in xrange(width):
        for y in xrange(height):
            p = im.getpixel( (x,y) )

            r = p[0] & 0b11110000
            g = p[1] & 0b11110000
            b = p[2] & 0b11110000

            im.putpixel( (x,y), (r,g,b) )
```

```
    im.save(fname+'-4bits.jpg')
...
```

Exercício 21.8

Replique o exemplo anterior e experimente anular quantidades diferentes de bits.

21.5 Efeitos sobre imagens

São vários as manipulações que podem ser aplicadas a imagens de forma a alterar a sua apresentação. Podem focar-se na manipulação das cores ou mesmo na alteração da geometria da imagem. As sub-secções seguintes demonstram como algumas manipulações podem ser conseguidas. Os métodos apresentados são sub-ótimos pois a biblioteca **Pillow** possui métodos para muitos deles. No entanto a apresentação utilizada justifica-se por motivos didáticos e de compreensão do efeito aplicado.

Recomenda-se que se implemente cada efeito como sendo uma função que aceite como parâmetro uma imagem e devolva novamente uma imagem. Desta forma torna-se possível encadear efeitos.

21.5.1 Troca de Cores

A troca de cores das imagens é um efeito simples que resulta da troca dos canais de uma imagem. Tipicamente aplicado no modo **RGB** pois permite um controlo mais direto sobre a imagem. O exemplo seguinte troca o canal Verde pelo Vermelho, sendo o resultado o demonstrado na Figura 21.4

```
...
    new_im = Image.new(im.mode, im.size)

    for x in range(width):
        for y in range(height):
            p = im.getpixel( (x,y) )
            r = p[1]
            g = p[0]
            b = p[2]
            new_im.putpixel((x,y), (r, g, b) )
...
```

Figura 21.4: Figura original em RGB e com os canais R e G trocados

Exercício 21.9

Construa uma função que troque os canais de cores de uma imagem.

Exercício 21.10

Construa uma função que crie uma imagem negativa. Esta imagem consiste na substituição de cada valor v por $255 - v$.

21.5.2 Tons de Cinza

Converter para tons de cinza implica remover toda a informação cromática, deixando apenas a intensidade. No modo `YCbCr` isto pode ser feito de forma imediata mantendo apenas o primeiro canal, mas não é o método mais poderoso. A biblioteca `Pillow` converte igualmente de forma automática qualquer imagem para tons de cinza, especificando o modo `L`.

O exemplo seguinte converte uma imagem para o modo `L`, guardando-a de seguida.

```
...
def main(fname):
    im = Image.open(fname)

    new_im = im.convert('L')
    new_im.save(fname+'-L.jpg')
...
```

Sendo que o resultado é o demonstrado na Figura 21.5.

Figura 21.5: Figura original em RGB e em tons de cinza (L)

Converter uma imagem para escala de cinzas implica processar as suas cores e aplicar uma fórmula que converta **RGB** (ou outro modo) em **L**. No caso anterior a fórmula utilizada é $L = R * \frac{299}{1000} + G * \frac{587}{1000} + B * \frac{114}{1000}$, mas podem ser utilizadas outros cálculos, tal como utilizar apenas uma cor, ou combinar as cores de forma diferente. Frequentemente existem vantagens em aplicar um processamento diferente. O exemplo seguinte aplica a fórmula descrita, mas de uma forma manual.

```
...
def effect_gray(im):
    width, height = im.size

    new_im = Image.new('L', im.size)

    for x in xrange(width):
        for y in xrange(height):
            p = im.getpixel( (x,y) )
            l = int(p[0] * 0.299 + p[1]*0.587 + p[2]*0.144)
            new_im.putpixel( (x,y), (l) )

    return new_im
...
```

Exercício 21.11

Replique o exemplo anterior mas combinando as cores de forma diferente. Pode, por exemplo, considerar apenas uma ou duas cores, combinadas ou utilizadas sem qualquer processamento. Verifique o resultado final.

21.5.3 Controlo de Intensidade

O controlo de intensidade resulta na alteração dos valores de intensidade da imagem. Valores superiores resultam numa imagem mais clara, valores mais baixos resultam numa imagem mais escura. Esta operação é bastante simplificada quando aplicada no modo `YCbCr`, pois o primeiro canal (Y) é o único que possui informação de intensidade.

A Figura 21.6 demonstra o resultado de manipular a intensidade de uma imagem.

Figura 21.6: Figura com intensidade aumentada (f=1.5) ou diminuída (f=0.5).

A aplicação deste efeito requer assim uma conversão de modo de representação e a multiplicação do canal `Y` por um factor. Se o factor for superior a 1 a imagem ficará com uma intensidade superior, se o valor for inferior a 1 ela aparecerá mais escura. É necessário ter em atenção que os valores nunca podem ultrapassar o mínimo ou máximo e têm de ser inteiros.

```
def effect_intensity(im, f):
    new_im = im.convert("YCbCr")
    width, height = im.size

    for x in xrange(width):
        for y in xrange(height):
            pixel = new_im.getpixel( (x,y) )
            py = min(255, int(pixel[0] * f))

            new_img.putpixel( (x,y), (py, pixel[1], pixel[2]) )
    ...
```

Exercício 21.12

Construa uma função que manipule a intensidade de um ficheiro por um factor.

21.5.4 Controlo de Gama

A gama de uma imagem diz respeito a quanto linear é a representação da intensidade dos seus valores. Tipicamente as imagens necessitam de ser corrigidas de forma a que a intensidade seja adaptada ao meio de reprodução. Nos Cathode Ray Tubes (CRTs), já raramente utilizados, é necessário aplicar curvas de compensação, pois a sua intensidade de reprodução não é linear. A Figura 21.7 apresenta este problema. Os CRTs possuem uma gamma típica de 2.2, o que significa que reproduzem as cores segundo uma linha exponencial. Uma correção típica irá distorcer as cores com o valor $\frac{1}{2.2}$ de forma que a imagem realmente percebida pelos utilizadores seja apresentada de forma linear.

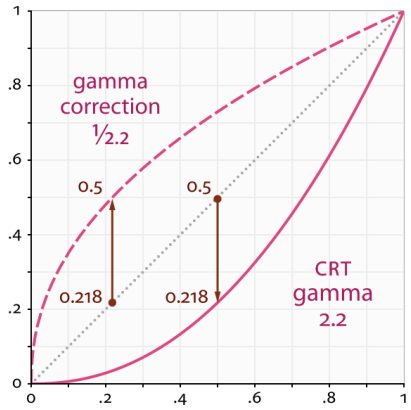

Figura 21.7: Correção de gama para um CRT (Fonte Wikimedia Foundation)

Figura 21.8: Figura com gama de 2.2 ou de 0.5.

De qualquer forma este tipo de operações é util para ajustar imagens de forma a

que a sua apresentação seja melhorada. A Figura 21.8 apresenta duas imagens com compensações de gama distintas.

Este ajuste pode ser obtido aplicando-se uma fórmula $py = y^g * f$ no modo `YCbCr`, em que y é o valor do canal `Y`, g é o factor de correção da gama. Onde f é um factor utilizado para repor a intensidade de imagem e é obtido pela fórmula $f = \frac{m}{m^g}$, e m representa o valor máximo de intensidade que se encontra na imagem (frequentemente 255).

Exercício 21.13

Construa uma função que aplique uma correção de gama a uma imagem.

21.5.5 Controlo de Saturação

A saturação de uma imagem refere-se à intensidade das cores apresentadas. Uma imagem muito saturada terá cores vivas enquanto uma imagem pouco saturada terá tons muito suaves. Uma imagem em tons de cinza não possui qualquer saturação. A Figura 21.9 demonstra duas imagens com saturações bastante distintas.

Figura 21.9: Figura com saturação aumentada por 1.5 ou reduzida para 0.5.

O controlo de saturação também faz uso do modo `YCbCr`, alterando neste caso os canais `Cb` e `Cr`. Estes canais codificam a saturação como a diferença em relação ao seu valor central, 128. Portanto, quando mais distante de 128 for o valor maior será a saturação, enquanto que quanto mais próximo de 128 for o valor, menor será a saturação. Um método comum é o apresentado no exemplo que se segue, onde se considera que `p` é um píxel específico:

```
...
py = p[0]
pb = min(255,int((p[1] - 128) * f) + 128)
pr = min(255,int((p[2] - 128) * f) + 128)
...
```

Exercício 21.14

Construa uma função que aplique uma transformação de saturação na imagem.

21.5.6 Sépia

Um efeito artístico que também se baseia na manipulação de cores é o efeito Sépia, frequentemente utilizado para representar fotografias antigas. Este efeito é simplesmente uma manipulação dos valores `RGB` de cada píxel de acordo com a seguinte fórmula:

$$\begin{aligned} nr &= r * 0.189 + g * 0.769 + b * 0.393 \\ ng &= r * 0.168 + g * 0.686 + b * 0.349 \\ nb &= r * 0.131 + g * 0.534 + b * 0.272 \end{aligned}$$

Em que nr, ng e nb são os novos valores para os canais Vermelho, Verde e Azul, enquanto r, g e b são os valores da imagem original.

Este efeito pode servir de base para muitos outros, bastando apenas a manipulação dos valores aplicados em cada multiplicação.

Figura 21.10: Figura convertida para tons Sépia ou Lomo

O resultado será tal como representado na imagem da esquerda da Figura 21.10. A imagem da direita é semelhante a um efeito chamado de *Lomography* e é obtido pela troca de nr por nb na fórmula anterior.

Exercício 21.15

Construa duas funções que implementem os efeitos descritos.

21.5.7 Detecção de Bordas

A detecção de bordas é bastante importante para tarefas como o reconhecimento de texto, mas pode ter outras utilizações no domínio do reconhecimento de padrões. O funcionamento básico deste processo é o de detectar alterações significativas entre dois píxeis e adicionar um traço. O resultado pode ser uma imagem a duas cores (preto e branco), onde o preto indica as zonas de alto contraste, ou uma imagem semelhante à original mas onde é sobreposta informação. A forma como a detecção é realizada pode variar bastante, podendo ser possível a análise da imagem em vários modos de representação de cor e aplicar pesquisas mais ou menos exaustivas.

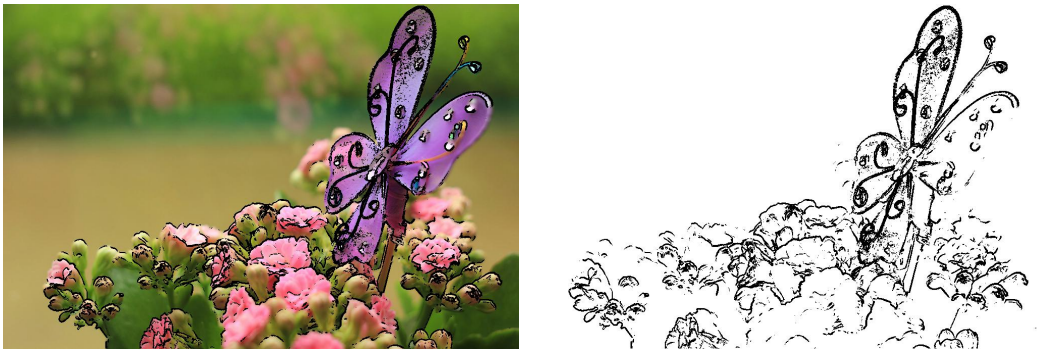

Figura 21.11: Bordas detectadas numa imagem.

Um algoritmo simples para este problema é o de detectar variações entre píxeis adjacentes através do cálculo da sua diferença. O resultado será o demonstrado na Figura 21.11. O algoritmo pode ser implementado tal como os anteriores, sendo que para cada píxel é necessário determinar se é uma borda ou não, o que se faz obtendo os seus vizinhos e calculando a diferença. Caso algum vizinho apresente uma diferença superior a um valor pré-definido, estamos perante uma borda.

A função seguinte implementa este aspeto do algoritmo. Para cada píxel **p** que esteja na imagem e não seja um limite da imagem, consultam-se todos vizinhos (superiores, inferiores, esquerda e direita). Caso a diferença para o píxel atual for superior a **diff**, é devolvido um píxel preto. Caso contrário é devolvido um píxel original ou branco, dependendo de **bw**.

```
def is_edge(im, x,y, diff, bw):
    #Obter o pixel
    p = im.getpixel( (x , y) )
    width, height = im.size

    if x < width-1 and y < height-1 and x > 0 and y > 0:

        #Vizinhos superiores e inferiores
        for vx in xrange(-1,1):
            for vy in [-1, 1]:
                px = im.getpixel( (x + vx, y + vy) )

                if abs(p[0]- px[0]) > diff:
                    return (0,128,128)

        #Vizinhos da esquerda e direita
        for vx in [-1, 1]:
            px = im.getpixel( (x + vx, y) )

            if abs(p[0]- px[0]) > diff:
                return (0,128,128)

    if bw :
        return (255,128,128)
    else:
        return p
```

Exercício 21.16

Implemente uma função que calcule as bordas de uma imagem e teste-a para várias imagens fornecidas.

21.5.8 Vignette

O efeito de Vignette é na realidade um defeito das lentes fotográficas em que as bordas das imagens ficam mais escuras que o seu centro. Diferentes lentes apresentarão diferentes níveis de *Vignette*, sendo típico de equipamento de custo mais reduzido, ou mais antigo. Nas lentes modernas este efeito normalmente existe mas é pouco pronunciado.

No entanto o efeito pode ser aplicado a imagens já obtidas, sendo muito comum em algumas comunidades. A Figura 21.12 apresenta uma imagem sofrendo de *Vignette* e outra com um *Vignette* deslocado de forma a realçar o elemento decorativo.

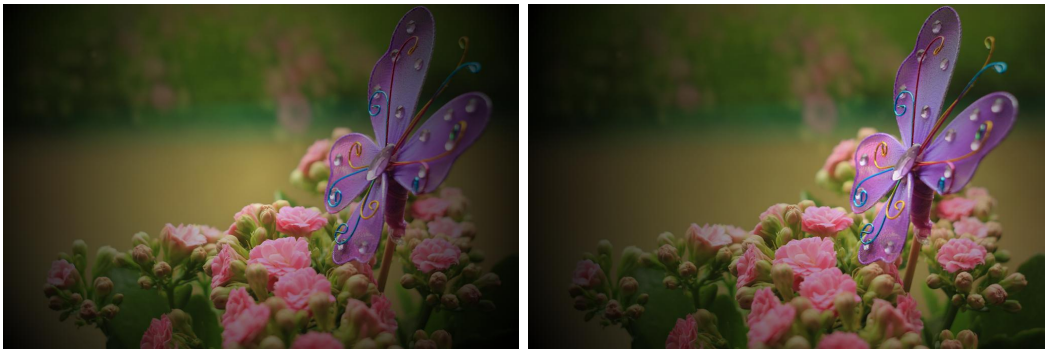

Figura 21.12: Vignette comum (esquerda) e deslocado para a direita (direita)

A implementação deste algoritmo implica que se determine um ponto de referência (normalmente o centro da imagem) e se calcule um factor de atenuação de intensidade (escurecimento) que é tanto maior quanto mais distante um píxel estiver da referência. A intensidade de todos os píxeis é multiplicada por este factor.

A distância entre dois pontos pode ser calculada através da fórmula da distância Euclidiana, que se resume a:

$$distance = \sqrt{(x - xref)^2 + (y - yref)^2} \tag{1}$$

O factor de atenuação irá assim variar entre 0 (nenhum escurecimento) e 1 (100%) para os limites da imagem.

```
def get_factor(x, y, xref, yref):
    distance = math.sqrt( pow(x-xref,2) + pow(y-yref,2))
    distance_to_edge = math.sqrt( pow(xref,2) + pow(yref,2))

    return 1-(distance/distance_to_edge) #Percentagem
```

Exercício 21.17

Implemente um efeito que aplique *Vignette* a uma imagem.

21.6 Marcação de imagens

21.6.1 Marca de Água

Uma marca de água é uma imagem que é sobreposta a outra imagem, normalmente utilizada para questões de direitos de autor. Para sobrepor uma imagem é necessário somar cada um dos píxeis das 2 imagens, depois de multiplicadas por um factor **f**. Este factor irá indicar o grau de transparência da marca de água. No exemplo que se segue quanto maior for **f**, menos transparente será a marca de água. Os valores **start_x** e **start_y** indicam a posição onde se deve iniciar a colocação da imagem.

```
...
#p1 é um pixel da imagem original
#p2 é um pixel da marca de água
    p1 = im1.getpixel( (x+start_x, y+start_y) )
    p2 = im2.getpixel( (x,y) )
    if(p2[3] == 0):
        continue

    r = int(p1[0]*(1-f)+p2[0]*f)
    g = int(p1[1]*(1-f)+p2[1]*f)
    b = int(p1[2]*(1-f)+p2[2]*f)
...
```

O resultado deste exemplo pode ser o apresentado na Figura 21.13, sendo que **f** teve o valor de 0.8.

Figura 21.13: Fotografia da UA com o símbolo em marca de água.

Exercício 21.18

Construa um programa que, aceitando duas imagens como argumento e um factor de transparência, adicione a segunda à primeira.

Uma alternativa de adicionar uma marca de água é manipular os bits individuais da imagem de forma a incluir uma marca que não seja claramente visível. Em particular pode-se considerar codificar o bit mais significativo da marca de água no bit menos significativo da imagem a marcar. Ou seja, manipular o bit que introduz menos erro na imagem a marcar, codificando o bit da marca de água que possui maior valor valor. Em *Python* o processo é conseguido alterando o exercício anterior para que a transformação para cada canal seja efectuada da forma:

```
...
r = (p1[0] & 0b11111110) | (p2[0] >> 7)
b = ...
...
```

A recuperação efectua-se processando toda a imagem e promovendo o bit menos significativo a mais significativo, o que se consegue com uma operação de *shift* à esquerda de 7 bits. Tipicamente este bit irá conter ruído, mas nos sítios onde foi codificada a imagem, será possível identificá-la.

```
...
r = (p1[0] << 7) & 255
...
```

O resultado da imagem com a marca de água e a imagem recuperada será o apresentado na Figura 21.14.

Exercício 21.19

Implemente um programa semelhante ao anterior mas que aplique a marca de água usando técnicas de esteganografia. Experimente processar todos os canais ou apenas alguns. Consegue notar diferenças na imagem original?

Figura 21.14: Imagem com marca de água usando técnicas de estanografia e marca de água recuperada.

21.6.2 Adição de texto

Uma outra forma de marcação de imagens é a sobreposição de textos sobre a imagem. Neste caso, o texto é construído directamente para a imagem através de métodos fornecidos pela biblioteca **`Pillow`**. O processo implica a selecção de um ficheiro de tipo de letra, um texto, um tamanho de letra, uma cor, e a definição de uma posição para colocar o texto.

Considerando a existência de uma imagem **`im`**, o exemplo seguinte permite a adição de uma mensagem de texto, neste caso a palavra **`LabI`** com tamanho 40, escrito a branco, na posição $x = 20$, $y = 20$.

```
from PIL import ImageDraw

...

draw = ImageDraw.Draw(im)
font = ImageFont.truetype('caminho-para-um-ficheiro.ttf', 40)

draw.text( (20, 20) ,'LabI', (255,255,255), font=font)
```

De notar que é necessário localizar onde se encontram os tipos de letra. Esta localiza-

ção irá depender de cada sistema. No caso dos sistemas *Linux*, podem ser encontrados no directório **`/usr/share/fonts/truetype`**.

Exercício 21.20

Implemente um programa que permita adicionar mensagens de texto a imagens.

Glossário

BMP BitMaP image file

CRT Cathode Ray Tube

Exif Exchangeable image file format

FITS Flexible Image Transport System

GIF Graphics Interchange Format

HTML HyperText Markup Language

JPEG Joint Photographic Experts Group

PNG Portable Network Graphics

TIFF Tagged Image File Format

Referências

[1] W3C. (1999). Html 4.01 specification, endereço: `http://www.w3.org/TR/1999/REC-html401-19991224/`.