Tema 20

Representação de Imagem

Objetivos:

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

20.1 Introdução

As imagens como fotografias, desenhos ou gráficos são deveras importantes para as aplicações computacionais actuais. Neste guião analisamos as formas de representação e processamento digital de imagens.

Para a realização deste guião será necessário instalar a biblioteca `Pillow`. Num computador com Debian/Ubuntu pode instalá-la através do comando:

```
sudo apt-get install python3-pil
```

Em alternativa, pode instalar o pacote através do comando `pip3`, mas antes terá de instalar várias dependências. Assim, em *Ubuntu* deverá correr:

```
sudo apt-get install libtiff4-dev libjpeg8-dev zlib1g-dev \
     libfreetype6-dev liblcms2-dev libwebp-dev tcl8.5-dev tk8.5-dev python-tk
pip3 install Pillow
```

Em *OS X* deverá correr:

```
brew install libtiff libjpeg webp little-cms2
pip3 install Pillow
```

Por razões pedagógicas, neste guião serão sugeridos processos que podem ser sub-ótimos. As bibliotecas `Pillow` e `OpenCV` possuem transformações otimizadas para muitos dos processos aqui tratados, devendo qualquer código de produção fazer uso desses métodos.

20.2 Imagens

Num computador, as imagens são representadas na forma de uma matriz composta de pequenas células organizadas em linhas e colunas. A cada célula da matriz dá-se o nome de píxel e corresponde ao menor elemento da imagem que pode ter cor distinta dos restantes. A *geometria da imagem* é definida pela largura e altura, medida em número de píxeis (ver Figura 20.1).

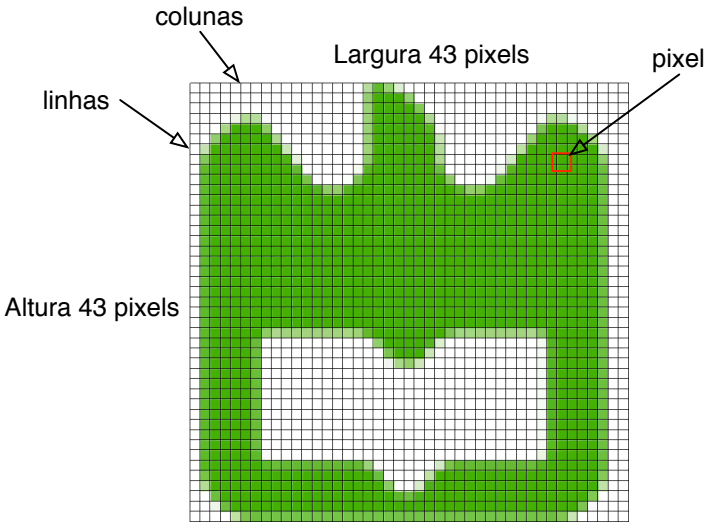

Figura 20.1: Composição de uma imagem digital.

Além da informação para representação da imagem, a maioria dos formatos de representação adiciona dados extra aos ficheiros, que são utilizados para indicar como deve o ficheiro ser processado ou que codificação é utilizada.

Alguns formatos, como é o caso do Joint Photographic Experts Group (JPEG), podem incluir dados num formato denominado Exchangeable image file format (Exif), que indicam entre outros aspetos que programa produziu a imagem ou qual a máquina fotográfica utilizada e os parâmetros da lente (no caso de uma fotografia). Também pode ser incluída uma versão miniatura da imagem para pré-visualização no navegador de ficheiros do sistema.

Num programa em *Python* é possível inspeccionar esta informação através da biblioteca **`Pillow`**, como se demonstra no exemplo abaixo.

```
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 20.1

Utilize o programa anterior para analisar os ficheiros de imagem fornecidos.

O tamanho em píxeis de uma imagem apresentada num monitor pode não ser igual ao tamanho da imagem armazenada. Por exemplo, quando se utiliza HyperText Markup Language (HTML)[1], o tamanho de apresentação de uma imagem pode ser especificado através de atributos que se aplicam ao elemento **`<img>`**, independentemente do tamanho original da imagem. Este redimensionamento obriga o computador a gerar uma nova imagem mais pequena ou maior que a original.

Note que este processo é destrutivo e não cria informação. Ou seja, ao reduzir a dimensão de uma imagem, não se está a reduzir a dimensão de cada píxel, mas antes a substituir vários por um só, perdendo informação dos outros. Ao aumentar a dimensão de uma imagem, criam-se novos píxeis, mas os seus valores (cores) são dependentes apenas dos píxeis originais; a definição da imagem não melhora, fica apenas mais "esbatida".

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 20.2

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

Exercício 20.3

O método utilizado para modificar as imagens é um dos mais simples (**`Nearest Neighbour`**), 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.

20.3 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. Já os conteúdos gráficos na *Web* utilizam sobretudo o formato Portable Network Graphics (PNG). 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 objetivas para preferir um formato em detrimento de outro. As diferentes formas de representação e de compressão de dados dos diferentes formatos são mais ou menos adequados consoante o tipo de imagem (se é um desenho ou uma fotografia, por exemplo).

No caso concreto da compressão, os formatos BMP, PNG e TIFF não aplicam compressão ou aplicam uma compressão sem perda de informação. Por outro lado, o formato JPEG aplica algoritmos de compressão com perdas para conseguir reduzir substancialmente o tamanho dos ficheiros.

Como os algoritmos estão otimizados para imagens naturais como fotografias, os artefactos introduzidos tornam-se mais evidentes quando aplicados a imagens artificiais como desenhos com áreas de cores sólidas e alto contraste.

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 20.2 demonstra o resultado de utilizar uma compressão de 30% para uma imagem com cores sólidas. Como se pode notar, a imagem foi bastante adulterada e torna-se evidente que o algoritmo processa as imagens por blocos.

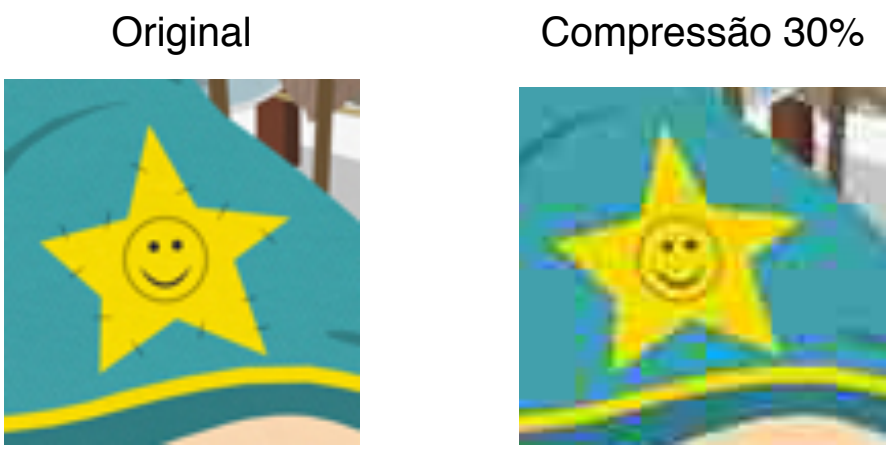

Figura 20.2: Compressão com JPG.

Exercício 20.4

Determine o tamanho dos blocos utilizados para comprimir a imagem anterior. Recomenda-se que simplesmente aumente o zoom do visualizador e conte os píxeis. (É conveniente desativar qualquer opção de suavização nas preferências do visualizador.) A imagem possui uma geometria de 90px por 88px.

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 20.5

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 20.6

Escolha um dos ficheiros fornecidos em formato JPEG e crie uma versão em cada um dos formatos: PNG, TIFF, BMP. Compare o tamanho do ficheiro criado.

20.4 Representação de cor

Existem várias formas para representar cores numa imagem. A escolha do formato mais apropriado depende do tipo de imagem que se possui e do objectivo da informação. O método mais comum e já visto nas aulas anteriores (ex, HTML) é o formato **`RGB`**. Neste formato, cada cor é representada por 3 componentes de cores primárias: vermelho (R), verde (G), e azul (B). A Figura 20.3 apresenta alguns valores **`RGB`** do logótipo 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 (absorvidas). Num ecrã os píxeis emitem luz, portanto as cores somam a sua intensidade e é utilizado o modelo **`RGB`**. Numa impressora as tintas depositadas num papel absorvem luz e por isso é frequentemente utilizado o modelo **`CMYK`**.

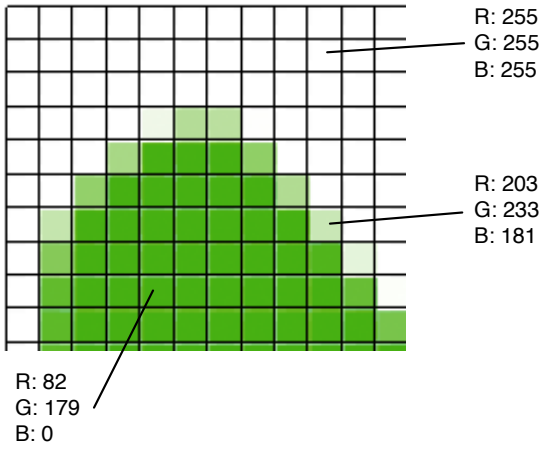

Figura 20.3: Valores RGB do logótipo da UA

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

- `BW` ou `1`: 1 canal com apenas 1 bit por píxel, que pode representar uma de duas cores. Utilizado para representar imagens a preto e branco.
- `RGB`: 3 canais, representando as intensidades de vermelho, verde e azul. Muito utilizado para a representação de cores em monitores.
- `RGBA`: O mesmo que `RGB` com um canal adicional (canal alfa) 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 tem um número que é usado como indíce para uma tabela de cores (uma *paleta*), que contém a cor desejada num dos outros formatos (tipicamente RGB). Utilizada para formatos como o Graphics Interchange Format (GIF).
- `YCbCr`: 3 canais representando a luminância (Y), crominância azul (Cb), crominância vermelha (Cr). A luminância pode ser obtida por uma média ponderada dos valores RGB, enquanto os canais Cb/Cr são determinados pela diferença entre os valores de vermelho/azul e a luminância, $Cb = B - Y$ e $Cr = R - Y$. O sistema é utilizado para fotografias (JPEG) e em vídeo.

Exercício 20.7

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 trazer vantagens do ponto de vista de representação e processamento. Para casos de utilização dos modos de cor, consulte a secção 20.5.

Repare que na Figura 20.3 os valores apresentados se referem a uma escala com $2^8 = 256$ níveis para cada cor, sendo que o total de cores representáveis será igual a $2^{3\times8}$. 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 componente, que depois são convertidas para o formato de 8 bits. No processo perde-se informação, pelo que alguns formatos como o TIFF permitem guardar 8 bits, 16 bits e 32 bits por componente. 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 (coloca a 0) os 4 bits menos significativos de cada componente, reduzindo o ficheiro para 4 bits de resolução efetiva.

```
...

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

    width, height = im.size

    for x in range(width):
        for y in range(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 20.8

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

20.5 Efeitos sobre imagens

São várias as manipulações que podem ser aplicadas a imagens de forma a alterar o seu aspeto. 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 programas foram escritos como forma de explicar os métodos da forma mais simples. Privilegiou-se a legibilidade e compreensibilidade, não a eficiência. A biblioteca `Pillow` possui métodos otimizados para muitas destas operações, que deverão ser utilizados em situações reais.

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

20.5.1 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 20.4

```
...
    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 20.4: Figura original em RGB e com os canais R e G trocados.

Exercício 20.9

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

Exercício 20.10

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

20.5.2 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` e guarda-a.

```
...
def main(fname):
    im = Image.open(fname)
    new_im = im.convert("L")
    new_im.save(fname+"-L.jpg")
...
```

O resultado é o demonstrado na Figura 20.5.

Figura 20.5: 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 $L = \frac{299}{1000}R + \frac{587}{1000}G + \frac{114}{1000}B$, mas podem ser utilizados outras expressões, 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 range(width):
        for y in range(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 20.11

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.

20.5.3 Controlo de Intensidade

O controlo de intensidade resulta na alteração dos valores de intensidade da imagem. Valores mais altos 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 20.6 demonstra o resultado de manipular a intensidade de uma imagem.

Figura 20.6: 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 resultantes têm de ser inteiros e nunca podem ultrapassar o mínimo ou o máximo da escala.

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

    for x in range(width):
        for y in range(height):
            pixel = new_im.getpixel( (x,y) )
            py = min(255, int(pixel[0] * f))    # [0] is the Y channel
            new_img.putpixel( (x,y), (py, pixel[1], pixel[2]) )
    ...
```

Exercício 20.12

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

20.5.4 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 20.7 apresenta este problema.

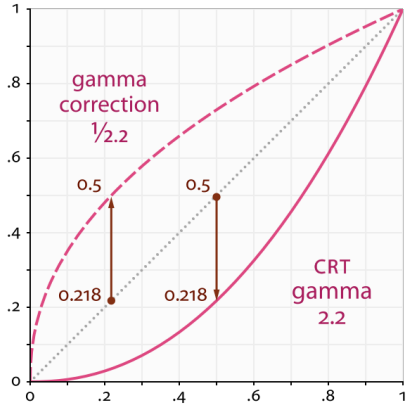

Figura 20.7: Correção de gama para um CRT (Fonte Wikimedia Foundation)

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.

Para além da compensação de não-linearidades dos dispositivos, este tipo de operações também se usa para melhorar imagens obtidas em condições de iluminação deficiente, por exemplo. A Figura 20.8 apresenta duas imagens com compensações de gama distintas.

Figura 20.8: Figura com gama de 2.2 ou de 0.5.

Este ajuste pode ser obtido aplicando-se uma fórmula $py = y^g * f$ no modo `YCbCr`, onde y é o valor do canal `Y`, g é o factor de correção gamma e f é um factor de normalização da intensidade de imagem. O fator de normalização é dado pela fórmula $f = \frac{m}{m^g}$, onde m representa o valor máximo de intensidade (frequentemente 255).

Exercício 20.13

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

20.5.5 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 20.9 demonstra duas imagens com saturações bastante distintas.

Figura 20.9: 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 de controlo de saturação é o apresentado no exemplo que se segue, onde se considera que **`p`** é um dado píxel da imagem.

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

Exercício 20.14

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

20.5.6 Sépia

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

$$\begin{aligned} R' &= 0.189R + 0.769G + 0.393B \\ G' &= 0.168R + 0.686G + 0.349B \\ B' &= 0.131R + 0.534G + 0.272B \end{aligned}$$

onde R, G e B são os valores da imagem original e R', G' e B' são os novos valores.

Este efeito pode servir de base para muitos outros, bastando apenas a manipulação dos coeficientes aplicados em cada multiplicação. O resultado será tal como representado na imagem da esquerda da Figura 20.10. A imagem da direita é semelhante a um efeito chamado de *Lomography* e é obtido pela troca de R' por B' na fórmula anterior.

Figura 20.10: Figura convertida para tons Sépia ou Lomo.

Exercício 20.15

Construa duas funções que implementem os efeitos descritos.

20.5.7 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. Existem diversos algoritmos de deteção de bordas, que funcionam em diferentes modos de representação de cor e aplicam pesquisas mais ou menos exaustivas.

Um algoritmo simples para este problema consiste em detectar variações através do cálculo da diferença entre píxeis adjacentes. Caso algum vizinho apresente uma diferença superior a um limiar pré-definido, estamos perante uma borda.

O resultado será o demonstrado na Figura 20.11.

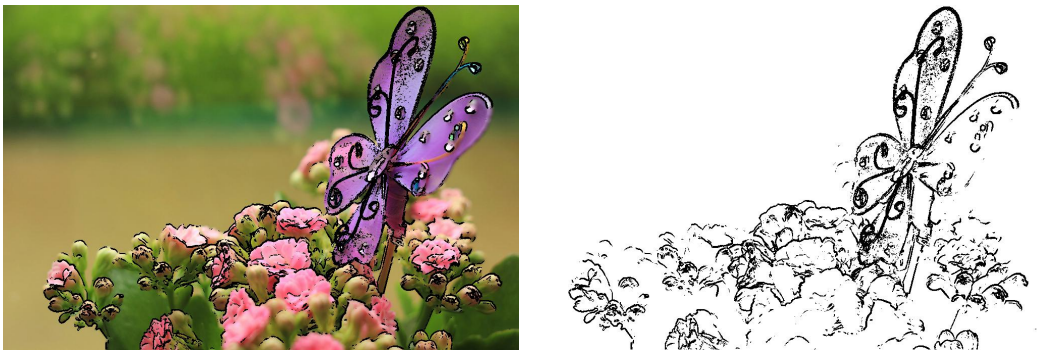

Figura 20.11: Bordas detectadas numa imagem.

A função seguinte implementa este algoritmo. Para cada píxel `p` que esteja na imagem e não seja um limite da imagem, consultam-se todos vizinhos (acima, abaixo, à esquerda e à direita). Caso a diferença para o píxel atual seja superior a `diff`, é devolvido um píxel preto. Caso contrário é devolvido um píxel original ou branco, dependendo da opção especificada em `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 acima e abaixo
        for vx in range(-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 20.16

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

20.5.8 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 baixa qualidade, 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 artísticas.

A Figura 20.12 apresenta uma imagem sofrendo de *Vignette* e outra com um *Vignette* deslocado de forma a realçar o elemento decorativo.

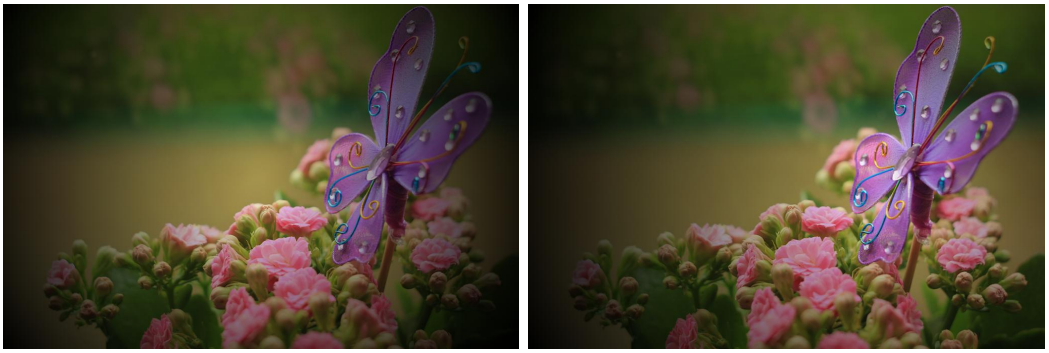

Figura 20.12: Vignette comum (esquerda) e deslocado para a direita (direita)

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,

$$distance = \sqrt{(x - x_0)^2 + (y - y_0)^2}. \tag{1}$$

O factor de atenuação irá assim variar entre 0 (nenhum escurecimento) para o ponto de referência (x_0, y_0) 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 20.17

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

20.6 Marcação de imagens

20.6.1 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)
...
```

A Figura 20.13 mostra o resultado de uma operação destas, neste caso com **`f=0.8`**.

Figura 20.13: Fotografia da UA com o símbolo em marca de água.

Exercício 20.18

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

Uma alternativa para adicionar uma marca de água quase impercetível é manipular os bits individuais da imagem de forma a 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. Isto é uma forma de esteganografia: a marca de água é escondida na imagem original de forma impercetível. Em *Python* o processo é conseguido alterando o exercício anterior para que a transformação aplicada a cada canal seja a seguinte.

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

A recuperação da marca de água 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 marca de água, será possível identificá-la.

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

A Figura 20.14 mostra o resultado da imagem com a marca de água e a versão recuperada.

Figura 20.14: Imagem com marca de água usando técnicas de estanografia e marca de água recuperada.

Exercício 20.19

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?

20.6.2 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.

O exemplo seguinte acrescenta uma mensagem de texto a uma imagem **`im`**, neste caso a palavra **`LabI`** com tamanho 40, escrita 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 especificar onde se encontram os tipos de letra. Esta localização irá depender de cada sistema. Nos sistemas *Linux*, podem ser encontrados no directório **`/usr/share/fonts/truetype`**.

Exercício 20.20

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», URL: `http://www.w3.org/TR/1999/REC-html401-19991224/`.