What is Computer Science?

An Information Security Perspective

Daniel Page ⟨dan@phoo.org⟩ and Nigel P. Smart ⟨csnps@bristol.ac.uk⟩

git # b4055dd3 @ 2019-05-13

HIDING A NEEDLE IN A HAYSTACK: CONCEALED MESSAGES

Imagine you are employed as a spy or, to add a modicum of glamour to the story, *the* spy, James "007" Bond himself [5]. You have two suspects under surveillance and are tasked with identifying which one is plotting to end the world. The usual laser pen and exploding pants were loaned to 006, but you *do* have a watch that can intercept the emails people send; you use the watch to capture some evidence:

Email from suspect #1:

```
Dear Mum,

I am having a strange holiday.  The
weather here is nice enough, but there
is a weird man in the next room who does
nothing but drink vodka martinis and
smile at women.

Love, David.
```

Email from suspect #2:

```
DE259236 4D503352 8D9ABE72 818B4040
856ECEC7 C9DB0FF2 7E854F98 9B0DF034
D14EC5DC 683D69C5 49C7AA7D AB6BD65A
77DDE93E 815275EC 6F66DD23 22A0333A
B1641A64 FCB533FA E8E210B7 81115EF2
7DB41239 4A942DCF E2C59A6E DC6DB547
3B0F1BC2 23D1E844 FDFD474A 9B0C9D30
FAD65181 3EEAD4FB 5D71AE10 28F8702B

Love, Mr. X.
```

Which suspect should you throw in the shark infested swimming pool? Clearly suspect #2: by the look of it he is sending encrypted emails so if he does not want people to know what they say, he *must* be up to no good. What went wrong for suspect #2? Cryptography was meant to solve all his secrecy problems! In short, the problem here is not really one of secrecy because the message suspect #2 sent is still secure in the sense that it cannot be read. Rather, the problem is that the message he is sending stands out and therefore attracts attention. It would be nice if the message was also secret, but in this case *hiding* the message is perhaps more important to suspect #2. The topic of **steganography** [16], a Greek word meaning "concealed writing", gives us a solution. As we have described previously, cryptography is all about preserving the secrecy of data; steganography on the other hand relates to hiding secret data, typically within non-secret data. For example, imagine we intercept a million emails with the watch. Suspect #2 could hide a secret message inside an innocuous email similar to the one suspect #1 sent: the secret message in this case is one needle in a haystack of a million emails, and we may totally overlook it as a result.

Although steganography is more generally interesting, one reason it makes a good topic is the connection to use within digital media: we get to look at some pictures for a change, rather than long lists of numbers! In this context, steganography can be used as a way to **watermark** [4] digital media. Imagine we have taken a brilliant photograph, and upload it to our web-site. What is to stop a competitor downloading the image, making a copy of it on their web-site, and claiming *they* took the photograph not us? The answer is not much because the beauty (and curse) of digital media is that it is so easy to copy. In Chapter 2 we already saw that CDs and DVDs are just well organised sequences of numbers; digital images and MP3 files are the same thing. So, unlike an oil painting or an analogue cassette tape, it is trivial to make a perfect copy of most digital media. One way to combat this problem is to hide a message within whatever we want to protect: imagine we hide the message "this photograph was taken by X on the date Y" inside the image. Even if someone else copies it, they would have a hard time convincing a court of law that they had taken the image and just happened to put our name in it!

Of course, real digital watermarking techniques usually have additional and complicated requirements that we will not worry about; obviously we require the watermark to be robust, in the sense that it cannot easily be removed, for instance. Our aim here is to use the scenario above as a motivation to look at two types of digital image, and two forms of steganography that follow quite naturally from how such images are represented.

1 Digital images

It pains me to say it, but I am old enough to remember a time before digital photography existed. The question is, how do modern digital cameras do the same thing as older ones that were based mainly on chemical processes? That is, how do they represent a physical image using numbers so that we can store and manipulate them using a computer? The answer is not *too* involved, but it turns out that there are (at least) two quite different varieties of digital image.

1.1 Rasterised images

A **rasterised image** (or "bitmap" image) measures or **samples** the colour of the physical image at regular intervals; the samples are the digital representation we keep. You can think of a rasterised image as being a matrix where each element represents one sample called a **pixel** (or **picture element**) [11].

The number of rows and columns for a particular image is usually fixed; if we have a $(m \times n)$-pixel image then the **resolution** is n by m, meaning it has n columns and m rows. You might have seen images described as "1024 by 768" for example, which would mean $n = 1024$ and $m = 786$. When talking about digital cameras, it is also common to refer to the *total* number of pixels rather than the number of rows and columns. So a camera described as "having 10 megapixels" produces images where $n \cdot m \sim 10 \cdot 10^6$, i.e., we might have a matrix with just over $n = 3000$ columns and just over $m = 3000$ rows.

In a sense, the total number of pixels tells us something about the image detail. Basically, if there are more pixels, more samples have been taken from the physical image and therefore there is more detail:

- A 1 megapixel digital camera produces a (1000×1000)-pixel image, i.e., one million pixels in total. Imagine we use the camera to take a picture of an object that is 5m long: we sample about one pixel every 5mm or so. What if there is an ant on the object which is only 2mm long? Chances are we might miss him out if he happens to fall between two samples.
- A 10 megapixel digital camera produces an image with ten times as many pixels, i.e., ten million pixels in total. Now, taking a picture of the same 5m object means we sample about one pixel every 1.6mm. There is more detail in this image: we can capture smaller features within the physical image.

The next question is how we describe the pixels themselves; remember that they are meant to represent samples, or colours, from the physical image. To do this, we use a **colour model** [3] which tells us which numbers represent which colours.

1.1.1 The RGB colour model

The **RGB** colour model [14] represents colours by mixing together red, green and blue (i.e., "R", "G" and "B"). The idea is to write a sequence of three numbers, i.e., a triple

$$\langle r, g, b \rangle$$

where each of r, g and b is called a **channel** and represents the proportion of red, green or blue in the pixel. There are several ways we could specify r, g and b; for example we could use a percentage, e.g., 100% red, which is both easy to read and understand. However, it is more common to see them specified as an n-bit integer between zero and some maximum $2^n - 1$. Imagine we set $n = 8$: this means each channel is some number x in the range $0 \ldots 255$ and the proportion specified is $x/(2^n - 1)$, in this case $x/255$. Here are some examples to make things clearer:

1. The triple $\langle 255, 255, 255 \rangle$ is white: it specifies 255/255 (or 100% red), 255/255 (or 100%) green and 255/255 (or 100%) blue.
2. The triple $\langle 255, 0, 0 \rangle$ is a pure red colour: it specifies 255/255 (or 100% red), 0/255 (or 0%) green and 0/255 (or 0%) blue.
3. The triple $\langle 255, 255, 0 \rangle$ is a pure yellow colour: it specifies 255/255 (or 100% red), 255/255 (or 100%) green and 0/255 (or 0%) blue.

$$\left(\begin{array}{c|c} \langle 255,255,255\rangle & \langle 0,0,0\rangle \\ \hline \langle 255,0,0\rangle & \langle 0,255,0\rangle \\ \hline \langle 0,0,255\rangle & \langle 255,255,0\rangle \\ \hline \langle 255,0,255\rangle & \langle 0,255,255\rangle \end{array}\right)$$

(a) *Representation as a* (4×2)*-element matrix of RGB triples.*

(b) *Representation as a* (4×2)*-pixel image.*

Figure 1: *An example rasterised image in two representations.*

4. The triple $\langle 51, 255, 102\rangle$ is a pea green colour: it specifies 51/255 (or 20% red), 255/255 (or 100%) green and 102/255 (or 40%) blue.

Since we need an n-bit integer for each channel, and we need three channels to specify the colour of each pixel, we need $3n$ bits for each pixel. This is sometimes called the **colour depth** because it tells us how many colours we can represent, i.e., the number of bits-per-pixel (sometimes abbreviated as "bpp"). With $n = 8$, each pixel needs 24 bits and we can represent nearly 17 million colours. This is enough colours to approximate physical images, and is used by many image formats that are produced by digital cameras.

The obvious next step is to put the two concepts together and arrange the RGB triples in a matrix to represent the pixels of an image. Figure 1 demonstrates what this actually looks like, both as a matrix of numbers and the pixels they represent. We can write the numbers that represent each colour channel in *any* base, so the matrix

$$\left(\begin{array}{c|c} \langle 11111111_{(2)}, 11111111_{(2)}, 11111111_{(2)}\rangle & \langle 00000000_{(2)}, 00000000_{(2)}, 00000000_{(2)}\rangle \\ \hline \langle 11111111_{(2)}, 00000000_{(2)}, 00000000_{(2)}\rangle & \langle 00000000_{(2)}, 11111111_{(2)}, 00000000_{(2)}\rangle \\ \hline \langle 00000000_{(2)}, 00000000_{(2)}, 11111111_{(2)}\rangle & \langle 11111111_{(2)}, 11111111_{(2)}, 00000000_{(2)}\rangle \\ \hline \langle 11111111_{(2)}, 00000000_{(2)}, 11111111_{(2)}\rangle & \langle 00000000_{(2)}, 11111111_{(2)}, 11111111_{(2)}\rangle \end{array}\right)$$

is equivalent: it is just written down in a different way.

You might be used to creating images using software that allows you to "paint" interactively (using a mouse). When we need careful control over pixel values, however, it can be easier to describe images using a text format such as PPM [9]. For example,

```
P3           # this is an ASCII PPM file
2            # there are 2 columns
4            # there are 4    rows
255          # maximum R, G or B is 255

255 255 255 # P_{0,0}
  0   0   0 # P_{0,1}
255   0   0 # P_{1,0}
  0 255   0 # P_{1,1}
  0   0 255 # P_{2,0}
255 255   0 # P_{2,1}
255   0 255 # P_{3,0}
  0 255 255 # P_{3,1}
```

is a PPM description of Figure 1. The first four lines specify information about the image, namely

- the PPM file identifier (which ensures any software reading the file can interpret the associated content correctly),
- the image dimensions, first the number of columns then the number of rows, and
- the maximum value any colour channel can have.

The subsequent lines specify image content: each line specifies a pixel, from top-left to bottom-right in the image. So the first pixel in the top-left corner is $(255, 255, 255)$ (meaning white).

Try to reproduce this example using image manipulation software of your choice to view the result.

As well as the red, green and blue channels, it is common to include an **alpha** channel. This forms the **RGBA colour space** [15], and typically means that each pixel is represented by 32 rather than 24 bits. Find out about the purpose of this addition, then try to show the effect it has using some experimental images.

1.1.2 The CMYK colour model

RGB is not the *only* colour model; you might reasonably argue that it is not even the most sensible since it does not correspond to what happens when paints are mixed together. Consider some examples:

1. To get green coloured paint you need to mix yellow and blue paint together. With RGB on the other hand, you can get a green colour "for free"; adding red gives a yellow colour. Try mixing red and green paint together: you do not get yellow paint!
2. If you mix together all colours of paint together you get black (or maybe just a mess), but in RGB you end up with a white colour.

So why the difference? Think about what a computer monitor is doing in a physical sense: basically it just emits light. If it emits no light, then we as the viewer see a black colour. If it emits red, green and blue then these "add up" to white. So, with the RGB model, colours are additive. Now think about how we see paint: white light hits the paint, the paint absorbs selected parts of the spectrum and reflects what is left so that we as the viewer can see it. If white light, $\langle 255, 255, 255 \rangle$ as an RGB triple, hits yellow paint, it absorbs all the blue component and we get reflected $\langle 255, 255, 0 \rangle$ back; if white light hits cyan paint, it absorbs all the red component and we get $\langle 0, 255, 255 \rangle$ reflected back. So if we mix yellow and cyan paint the result will absorb all the blue *and* red components from the spectrum and hence reflect $\langle 0, 255, 0 \rangle$, i.e. green, back. Hence, paint absorbs (i.e., subtracts) colours of light, whereas a computer monitor emits (i.e., adds) colours of light.

The idea of using cyan, magenta and yellow as primary colours is standard in the printing world since it deals with ink (which is like paint). So whereas we use red, green and blue as primary colours and hence the RGB colour model *within* computers, when we print rather than display images it is common to use the **CMYK** colour model [2] instead.

If you are *really* serious about printing and typesetting, the name **Pantone** [10] will crop up a lot: the **Pantone Matching System (PMS)** is a standard colour model used in most commercial settings. Do some research about this colour model, and the legal issues which mean it often cannot be used (e.g., within the open source GIMP software mentioned in Chapter 2).

1.2 Vector images

A **vector image** describes a physical image using a combination of geometric primitives such as points and lines. It is also possible to include more complicated curved surfaces (e.g., circles and ellipses), and even to extend from 2-dimensions into 3-dimensions. An example makes this easy to explain. Imagine you have a large sheet of graph paper: a 2-dimensional point on our graph paper is just a pair of coordinates that we write as (x, y). We can draw such a point on the graph paper; scale is not important, but you can imagine the paper has unit sized or 1cm squares on it if that helps. Using two such points, we can write

$$(x, y) \rightsquigarrow (p, q)$$

to describe a 2-dimensional line segment between points (x, y) and (p, q). Again, we can draw such a line on the graph paper: just get a ruler and join up the points. Using lots of lines, we can start to draw basic images. Imagine we start with this one

$$\begin{array}{rcl} (1.0, 1.0) & \rightsquigarrow & (2.0, 1.0) \\ (2.0, 1.0) & \rightsquigarrow & (2.0, 2.0) \\ (2.0, 2.0) & \rightsquigarrow & (1.0, 2.0) \\ (1.0, 2.0) & \rightsquigarrow & (1.0, 1.0) \end{array}$$

$$\begin{array}{rcl} (3.0, 1.0) & \rightsquigarrow & (4.0, 1.0) \\ (4.0, 1.0) & \rightsquigarrow & (3.5, 2.0) \\ (3.5, 2.0) & \rightsquigarrow & (3.0, 1.0) \end{array}$$

which describes seven lines in total. The first four lines form a square with sides of length one unit, the last three form an isosceles triangle of base and altitude one unit. It does not look very impressive when written like this, but if we were to draw it on our graph paper, it would look like Figure 2a. On one hand, this is still a fairly unimpressive image; we could make it more complicated by drawing more lines, but it still might be difficult to represent a physical image such as a face or a flower. On the other hand, what we might have sacrificed in terms of realism we recoup in terms of precision: since the image is effectively described in terms of Mathematics, we can manipulate it via Mathematics as well. There are applications where this is a real benefit; if you are designing a new car or building, accuracy is vital!

It is not as common now, but a system called **Logo** [7] has been used extensively as a way to teach programming. The idea is that a program controls an on-screen **turtle** that is guided around, drawing as it moves. In a sense, the Mathematical description of images above is very close to a Logo program: see if you can use an online resource such as

http://turtleacademy.com/

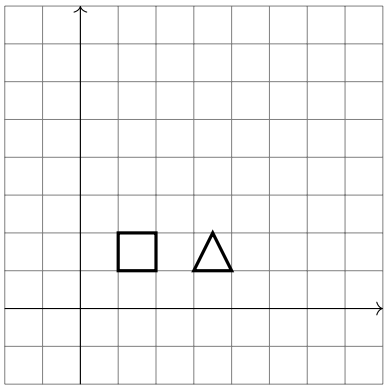
to reproduce our simple example. If you have no background in programming, this is a good chance to explore what else is possible.

To demonstrate the significance of this, we can start to think about taking a point (x, y) and translating it into a new point (x', y') using some form of **transformation**. Three such transformations, which describe how to compute x' and y', are

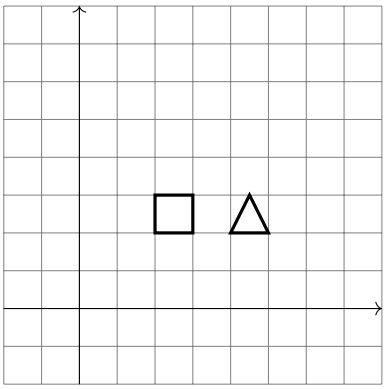
$$\begin{array}{lcl} \textsc{Translate} : & x' & = x + i \\ & y' & = y + j \end{array}$$

$$\begin{array}{lcl} \textsc{Scale} : & x' & = x \cdot i \\ & y' & = y \cdot j \end{array}$$

$$\begin{array}{lcl} \textsc{Rotate} : & x' & = x \cdot \cos\theta - y \cdot \sin\theta \\ & y' & = x \cdot \sin\theta + y \cdot \cos\theta \end{array}$$

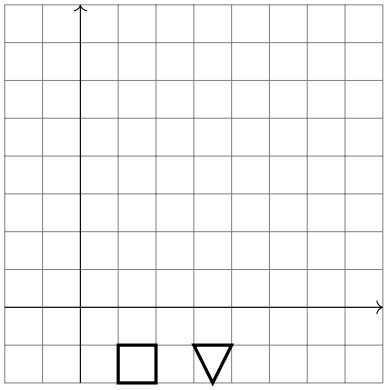


(a) *Original.*



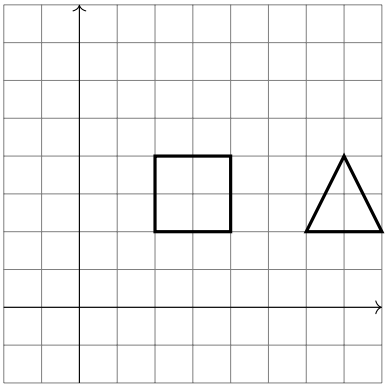
$x' = x + 1$
$y' = y + 1$

(b) *Translate.*



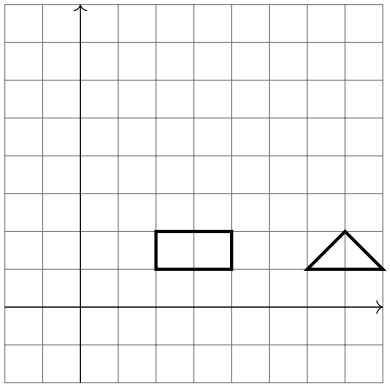
$x' = x$
$y' = -y$

(c) *Mirror vertically.*



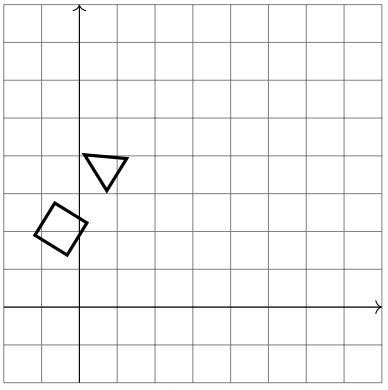
$x' = x \cdot 2$
$y' = y \cdot 2$

(d) *Scale horizontally and vertically.*



$x' = x \cdot 2$
$y' = y \cdot 1$

(e) *Scale horizontally only.*



$x' = x \cdot \cos(45) - y \cdot \sin(45)$
$y' = x \cdot \sin(45) + y \cdot \cos(45)$

(f) *Rotate about origin.*

Figure 2: *Some example vector image transformations.*

Imagine we select an example point $(1.0, 2.0)$, i.e., we have $x = 1.0$ and $y = 2.0$, which is the bottom left-hand corner of our square. The first thing we might try is to translate (or "move") the point to somewhere else in the image. To do this we *add* a vector (i, j). To move the point one unit horizontally and one unit vertically for example, we compute the new point as

$$\begin{array}{rcccl} x' & = & 1.0 + 1.0 & = & 2.0 \\ y' & = & 2.0 + 1.0 & = & 3.0 \end{array}$$

In isolation this probably does not seem very exciting. But if we repeat the transformation for all the points in the image, we end up with the result in Figure 2b. And that is just for starters! We can reflect an image in either axis. For example by negating the y coordinate we can mirror the image in the y-axis, as in Figure 2c.

If we want to scale a point (x, y) by a factor of i horizontally and j vertically we multiply x and y by i and j. Imagine we want to double the size of the image; setting $i = 2$ and $j = 2$ we compute the new point as

$$\begin{array}{rcccl} x' & = & 1.0 \cdot 2.0 & = & 2.0 \\ y' & = & 2.0 \cdot 2.0 & = & 4.0 \end{array}$$

Again, doing the same thing with all the points in our image gives the result in Figure 2d. This time the result is a bit more impressive: we have doubled the image size, but we have not degraded the quality *at all*. The Mathematics to describe the lines, and hence our image, is "perfect". We can play about even more, and stretch the image horizontally; setting $i = 2$ and $j = 1$ gives Figure 2e.

As a final trick we can think about rotating the image, the result of which is shown in Figure 2f. If we want to rotate the point by θ, say $\theta = 45$ for instance, we compute the new point (to three decimal places) as follows:

$$\begin{array}{rcccl} x' & = & 1.0 \cdot \cos(45) - 2.0 \cdot \sin(45) & = & -0.325 \\ y' & = & 1.0 \cdot \sin(45) + 2.0 \cdot \cos(45) & = & 1.376 \end{array}$$

What *other* types of useful transformation can you think of? As a hint, think about a shearing transform which produces a slanting result. Find out how such transformations can be specified and applied by using matrices [17].

Rotation is the first point where we hit a problem: the "perfectness" of the Mathematics fails us when we want to write down actual values for x' and y', because of the limit on **precision** (put simply, we only have a fixed number of decimal places available). Until then, we can manipulate everything perfectly without ever having to see a number! In a sense, the same problem crops up when we want to print or view a vector image.

Most laser printers [6] or plotters [12] can draw vector images. Many accept **PostScript** [13], a language for describing vector images, as input: programs written in PostScript are essentially lists of geometry, such as the line segments we started off with. But, eventually, physical constraints will cause problems. Such a printer cannot usually draw infinitely small images, for example, since there is a limit to how accurate mechanical parts can ever be. Some other types of display device [18] can also render vector images; early video games like Asteroids [1] used this sort of technology. These are rare however, and most modern computer monitors work in a different way: they first **rasterise** the vector image, turning it into a format that can be displayed. And there lies the problem: the process of rasterisation throws away the "perfect" nature of the Mathematics and forces us to do things like round-up the coordinates of a point, which might be represented using many decimal places of precision, so they match the nearest integer pixel location.

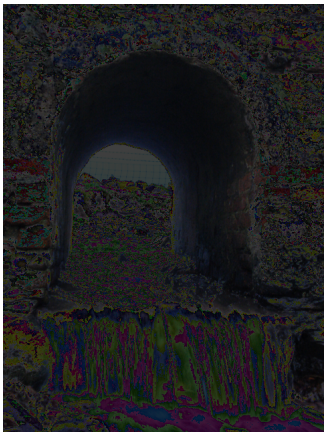
2 Steganography

Back to James Bond, or rather the problem of stopping his pesky snooping. Remember that the idea is to hide some secret data in non-secret data and, by doing so, throw him off the scent. The secret message will be a string of characters (an email if you like) and the non-secret data will be a digital image. Each type of digital image we have looked at gives quite a neat way to do this.

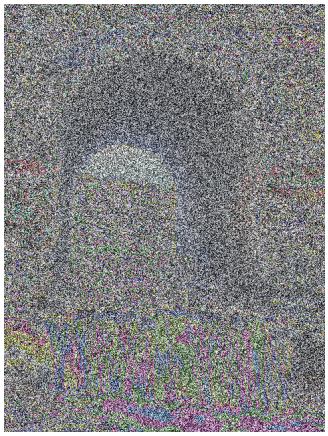
(a) *Original.*

(b) *Two LSBs zeroed.*

(c) *Two LSBs randomised.*



(d) *Two MSBs zeroed.*



(e) *Two MSBs randomised.*

Figure 3: *Four images, each created by altering the pixels from an original image.*

⟨255, 255, 255⟩	⟨0, 0, 0⟩
⟨255, 0, 0⟩	⟨0, 255, 0⟩
⟨0, 0, 255⟩	⟨255, 255, 0⟩
⟨255, 0, 255⟩	⟨0, 255, 255⟩

(a) *Original* (4×2)*-element matrix.*

(b) *Original* (4×2)*-pixel image.*

⟨253, 254, 254⟩	⟨0, 1, 2⟩
⟨253, 1, 1⟩	⟨2, 255, 0⟩
⟨1, 2, 255⟩	⟨252, 253, 2⟩
⟨255, 3, 252⟩	⟨2, 255, 254⟩

(c) *Altered* (4×2)*-element matrix.*

(d) *Altered* (4×2)*-pixel image.*

Figure 4: *The result of injecting a 6-character ASCII message into the LSBs of an eight pixel image.*

2.1 Rasterised images: "stolen LSBs"

We have already learned that a rasterised image can be represented as a matrix of numbers, and ideally binary numbers. For example, our original (4×2)-pixel example image was

$$\left(\begin{array}{c|c} \langle 11111111_{(2)}, 11111111_{(2)}, 11111111_{(2)} \rangle & \langle 00000000_{(2)}, 00000000_{(2)}, 00000000_{(2)} \rangle \\ \hline \langle 11111111_{(2)}, 00000000_{(2)}, 00000000_{(2)} \rangle & \langle 00000000_{(2)}, 11111111_{(2)}, 00000000_{(2)} \rangle \\ \hline \langle 00000000_{(2)}, 00000000_{(2)}, 11111111_{(2)} \rangle & \langle 11111111_{(2)}, 11111111_{(2)}, 00000000_{(2)} \rangle \\ \hline \langle 11111111_{(2)}, 00000000_{(2)}, 11111111_{(2)} \rangle & \langle 00000000_{(2)}, 11111111_{(2)}, 11111111_{(2)} \rangle \end{array}\right).$$

Some of the bits are given special names:

- The bits at the right-hand end of each number (those coloured red), are termed the **least-significant** bits or LSBs: they contribute weights of 2^1 and 2^0 to the overall value, the smallest weights of all.
- The bits at the left-hand end of each number (those coloured blue), are termed the **most-significant** bits or MSBs: they contribute weights of 2^7 and 2^6 to the overall value, the largest weights of all.

What happens if we alter either the LSBs or MSBs of the pixels in an image? Imagine we look at each pixel and set the two LSBs of each colour channel to zero. Or, maybe we look at each pixel and randomise the two MSBs of each colour channel. Figure 3 demonstrates four images that were created by taking an original and slightly altering each pixel along these lines. Figure 3b and Figure 3c had the two LSBs in each pixel altered, either set to zero or a random 2-bit value respectively. Although the images are marginally different (perhaps a little darker), without the original you would be hard pressed to pick them out as having been altered at all. This should make sense: we are altering the LSBs and these have the least impact on the value of each colour, so changing them does not have a lot of impact.

In Figure 3d and Figure 3e, the two MSBs rather than the LSBs were altered and the results are quite striking: if the MSBs are altered, the images are clearly corrupted. In Figure 3d, each pixel has become *much* darker: the image is more or less intact and understandable, but in Figure 3e is much less understandable; there is still some structure if you look closely, but it is really quite random. Again, this should make sense, we are altering the MSBs and these have the most impact on the value of each colour. In particular, if we zero the two MSBs we are basically saying that each colour channel can only take a value $0 \ldots 63$ rather than $0 \ldots 255$ so at *best*, it can only be about a quarter as bright.

The question is, how can we use our findings as a steganographic mechanism? The answer is reasonably simple. We are going to "steal" the LSBs from their original purpose of contributing to an image, and use them to conceal a message. The theory is that altering the LSBs will not corrupt the image too much: we have already seen that *even* if we randomise them, the end result is very close to the original. Our original (4×2)-pixel example image had eight pixels in total; we are stealing six bits from each pixel (two from each of the colour channels), so 48 bits in total. Chapter 5 already showed us than an ASCII character can be stored using 8 bits. Therefore, we can store a 6-character message in the 48 bits we have at our disposal.

First, we need to choose a message: the 6-character string "hello." (note the trailing full stop) is not particularly exciting, but will do the job here. Writing the ASCII representation of the string as a sequence we have

$$\langle 104, 101, 108, 108, 111, 46 \rangle.$$

If we write this in binary instead our message is

$$\langle 01101000_{(2)}, 01100101_{(2)}, 01101100_{(2)}, 01101100_{(2)}, 01101111_{(2)}, 00101110_{(2)} \rangle$$

and if we split it up into 2-bit chunks then, reading left-to-right and top-to-bottom, we get

$$\left\langle \begin{array}{cccccc} 01_{(2)}, & 10_{(2)}, & 10_{(2)}, & 00_{(2)}, & 01_{(2)}, & 10_{(2)}, \\ 01_{(2)}, & 01_{(2)}, & 01_{(2)}, & 10_{(2)}, & 11_{(2)}, & 00_{(2)}, \\ 01_{(2)}, & 10_{(2)}, & 11_{(2)}, & 00_{(2)}, & 01_{(2)}, & 10_{(2)}, \\ 11_{(2)}, & 11_{(2)}, & 00_{(2)}, & 10_{(2)}, & 11_{(2)}, & 10_{(2)} \end{array} \right\rangle$$

which is now ready to be injected into the image. Again, this is reasonably simple: we just start with the example image and replace the red LSBs with the sequence of 2-bit chunks derived from our message. The result is as follows:

$$\left(\begin{array}{c|c} \langle 11111101_{(2)}, 11111110_{(2)}, 11111110_{(2)} \rangle & \langle 00000000_{(2)}, 00000001_{(2)}, 00000010_{(2)} \rangle \\ \hline \langle 11111101_{(2)}, 00000001_{(2)}, 00000001_{(2)} \rangle & \langle 00000010_{(2)}, 11111111_{(2)}, 00000000_{(2)} \rangle \\ \hline \langle 00000001_{(2)}, 00000010_{(2)}, 11111111_{(2)} \rangle & \langle 11111100_{(2)}, 11111101_{(2)}, 00000010_{(2)} \rangle \\ \hline \langle 11111111_{(2)}, 00000011_{(2)}, 11111100_{(2)} \rangle & \langle 00000010_{(2)}, 11111111_{(2)}, 11111110_{(2)} \rangle \end{array} \right)$$

If we turn this back into decimal to make it easier to read, and render the matrix as pixels as well, the end result is Figure 4. The left-hand side shows the two matrices: clearly there *is* a difference as a result of the LSBs having been commandeered to store the message.

Reversing the process to extract the message is just as simple: we take the pixels, extract the LSBs and then recombine the 2-bit chunks into 8-bit bytes which represent the characters. But the point is that we would have to know the message was there in the first place before we even attempted to extract it. Looking at the comparison, it is difficult to see *any* difference in the images themselves: marginal changes in the colour channels are not easily detected *even* when we have the original image (which of course we would ordinarily lack).

If you *know* the message (or watermark) has been embedded in an image, it is of course easy to extract it. What about if you just suspect such a watermark is there? Can you think of a way to detect watermarks of this (or some other, similar) type?

2.2 Vector images: "microdots"

You Only Live Twice [19] is the first film to feature cat lover Ernst Blofeld, the head of SPECTRE, as a main character (in previous films you only see a character stroking a white cat, never his face). Part way through the film, Bond recovers a photograph of a cargo ship; on the photograph is something called a **microdot** [8]. A microdot is basically a very tiny image. The idea is to scale one image until it is *so* small it can be placed, in an inconspicuous way, within another image. Because it is so small, the premise is that it will be overlooked by a casual observer. So we take a secret message, scale it until it looks like a barnacle on the side of the cargo ship and then only someone looking for it (or who is far too interested in barnacles) will be able to recover the message.

Far from being limited to science fiction, there is plenty of evidence to suggest microdots being used as a real steganographic mechanism. Various sources have claimed invention, but it was almost certainly used by

Figure 5: *A "real" microdot embedded into a full stop.*

German intelligence agents in WW2 to send covert messages. British counter-intelligence nicknamed the microdots "duff" because they were mixed into letters like raisins were mixed into in the steamed pudding "plum duff". More modern uses include identification of physical resources such as car parts: the car manufacturer prints a unique number on each part in order to trace it in the event it is stolen. Of course a car thief might try to cover their tracks by etching off any obvious markings, but if they cannot even see the microdot then the chances of removing it are slim.

From the terminology we have used so far, it perhaps is not a surprise that we can try to create a real microdot using vector images: remember they these can be scaled without loss of quality, so are an ideal match. Figure 5 shows a somewhat basic example where we have scaled a message so that it is small enough to fit inside a full stop. If you are reading an electronic PDF version of this document, you can actually zoom in and enlarge the dot to see for yourself. If you are reading a version of the document printed on paper, this clearly would not work. Why not? We already talked about the problem: a given printer has limits as to how accurate the print mechanism is. So, in printing the electronic version that includes the perfect Mathematical description of our message, we have lost all the detail. Real microdots are therefore created using a traditional photographic process in which (very roughly) a special camera is used which shrinks an image of the message using magnification.

References

[1] *Wikipedia: Asteroids*. http://en.wikipedia.org/wiki/Asteroids_(game) (see p. 9).
[2] *Wikipedia: CMYK color model*. http://en.wikipedia.org/wiki/CMYK_color_model (see p. 7).
[3] *Wikipedia: Colour model*. http://en.wikipedia.org/wiki/Color_model (see p. 4).
[4] *Wikipedia: Digital watermarking*. http://en.wikipedia.org/wiki/Digital_watermarking (see p. 3).
[5] *Wikipedia: James Bond*. http://en.wikipedia.org/wiki/James_Bond (see p. 3).
[6] *Wikipedia: Laser printer*. http://en.wikipedia.org/wiki/Laser_printer (see p. 9).
[7] *Wikipedia: Logo*. http://en.wikipedia.org/wiki/Logo_(programming_language) (see p. 7).
[8] *Wikipedia: Microdot*. http://en.wikipedia.org/wiki/Microdot (see p. 12).
[9] *Wikipedia: Netpbm format*. http://en.wikipedia.org/wiki/Netpbm_format (see p. 6).
[10] *Wikipedia: Pantone*. http://en.wikipedia.org/wiki/Pantone (see p. 7).
[11] *Wikipedia: Pixel*. http://en.wikipedia.org/wiki/Pixel (see p. 4).
[12] *Wikipedia: Plotter*. http://en.wikipedia.org/wiki/Plotter (see p. 9).
[13] *Wikipedia: PostScript*. http://en.wikipedia.org/wiki/PostScript (see p. 9).
[14] *Wikipedia: RGB color model*. http://en.wikipedia.org/wiki/RGB_color_model (see p. 4).
[15] *Wikipedia: RGBA color space*. http://en.wikipedia.org/wiki/RGBA_color_space (see p. 6).

[16] *Wikipedia: Steganography*. http://en.wikipedia.org/wiki/Steganography (see p. 3).

[17] *Wikipedia: Transformation matrix*. http://en.wikipedia.org/wiki/Transformation_matrix (see p. 9).

[18] *Wikipedia: Vector monitor*. http://en.wikipedia.org/wiki/Vector_monitor (see p. 9).

[19] *Wikipedia: You Only Live Twice*. http://en.wikipedia.org/wiki/You_Only_Live_Twice_(film) (see p. 12).

I have a question/comment/complaint for you. Any (positive *or* negative) feedback, experience or comment is very welcome; this helps us to improve and extend the material in the most useful way.

However, keep in mind we are far from perfect; mistakes are of course possible, although hopefully rare. Some cases are hard for us to check, and make your feedback even more valuable: for instance

1. minor variation in software versions can produce subtle differences in how some commands and hence examples work, and
2. some examples download and use online resources, but web-sites change over time (or even might differ depending on where you access them from) so might cause the example to fail.

Either way, if you spot a problem then let us know: we will try to explain and/or fix things as fast as we can!

Can I use this material for something ? We are distributing these PDFs under the Creative Commons Attribution-ShareAlike License (CC BY-SA)

http://creativecommons.org/licenses/by-sa/4.0/

This is a fancy way to say you can basically do whatever you want with them (i.e., use, copy and distribute it however you see fit) as long as a) you correctly attribute the original source, and b) you distribute the result under the same license.

Is there a printed version of this material I can buy? Yes: Springer have published selected Chapters in

http://www.springer.com/computer/book/978-3-319-04041-7

while *also* allowing us to keep electronic versions online. Any royalties from this published version are donated to the UK-based Computing At School (CAS) group, whose ongoing work can be followed at

http://www.computingatschool.org.uk/

Why are all your references to Wikipedia? Our goal is to give an easily accessible overview, so it made no sense to reference lots of research papers. There are basically two reasons why: research papers are often written in a way that makes them hard to read (even when their intellectual content is not difficult to understand), and although many research papers are available on the Internet, many are not (or have to be paid for). So although some valid criticisms of Wikipedia exist, for introductory material on Computer Science it certainly represents a good place to start.

I like programming; why do the examples include so little programming? We want to focus on interesting topics rather than the mechanics of programming. So even when we include example programs, the idea is to do so in a way where their meaning is fairly clear. For example it makes more sense to use pseudo-code algorithms or reuse existing software tools than complicate a description of something by including pages and pages of program listings.

But you need to be able to program to do Computer Science, right? Yes! But only in the same way as you need to be able to read and write to study English. Put another way, reading and writing, or grammar and vocabulary, are just tools: they simply allow us to study topics such as English literature. Computer Science is the same. Although it *is* possible to study programming as a topic in itself, we are more interested in what can be achieved *using* programs: we treat programming itself as another tool.