Back to basics: JPEG compression

Ben Everard uses the undead to illustrate how JPEG compression maximises image quality while minimising image size.

Our expert

Ben Everard left his job as an IT consultant to spend two years in Tanzania installing Ubuntu-based systems in schools. Now he's putting his skills to use in the roiling cauldron of discovery that is **LXF** Towers.

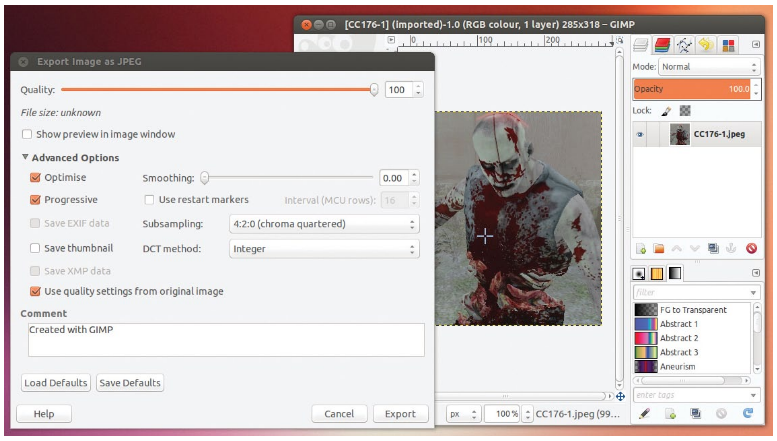
Last month we looked at the DEFLATE algorithm, [**LXF175**, p86] which compresses files so they take up less space when stored, but can still be decompressed to the original file. It works well, but for some types of file we want to squeeze even more space out of them. For example, a 10-megapixel image with a single byte per colour per pixel would take up about 240MB. Using the DEFLATE algorithm, we might get this down to tens of megabytes (this, incidentally, is what the PNG file format does). However, we might want to get it even smaller.

There's no way to create a significantly smaller file and retain the image quality perfectly. Instead, we have to look for the best ways to strip out information so that the impact on the quality is minimised. This is the sort of compression used in MP3s and almost all video formats. Here we're going to look at the process of compressing an image in JPEG format.

When we talk about maximising the quality of the compressed image, it's important to realise that this is quality as it appears to humans, not the actual bit-by-bit difference between the original and compressed files that a computer might see. The trick in creating good lossy compression algorithms is in removing data that humans can't perceive.

We often think about images having a red, green and blue component, but that's actually not quite how our eyes work. We perceive colours in red, green and blue, but we see detail just in intensity, regardless of colour. This, for example, is why we can't see colours in the dark, and why some photographs look better in black and white. JPEG compression takes advantage of this by transforming the image from the RGB colour encoding to the Y'CbCr encoding. This is brightness (Y'), blue (Cb) and red (Cr). Since our eyes only see detail in the Y' channel, we reduce the resolution of the Cb and Cr channels by half. When the image is decompressed, we can revert this to RGB by calculating G=Y'-(Cb+Cr), and also factoring the brightness channel into the red and blue channels. This process (known as 4:2:0 subsampling) reduces the amount of information in the image by half, yet is barely perceptible to the eye.

Now we get to the maths bit. We split each channel up into 8x8 pixel blocks and for each block, apply a Discrete Cosine Transformation (DCT). For those of you who like numbers and Greek letters, take a look at the formula on p85 (**Fig 1**), for those of you who don't, fear not, we're going to focus on what it does, not how it does it.



› Using *Gimp*, you can fine-tune the variables in the JPEG compression.

[CC176-1] (imported)-1.0 (RGB colour, 1 layer) 285x318 – GIMP

Export Image as JPEG

Quality: 100

File size: unknown

☐ Show preview in image window

▼ Advanced Options

☑ Optimise — Smoothing: 0.00

☑ Progressive — ☐ Use restart markers — Interval (MCU rows): 16

☐ Save EXIF data — Subsampling: 4:2:0 (chroma quartered)

☐ Save thumbnail — DCT method: Integer

☐ Save XMP data

☑ Use quality settings from original image

Comment: Created with GIMP

Tricking the eye

In simple terms, a DCT changes an image from a spatial representation to a frequency representation. Imagine for a moment that we take a single strip of pixels along the image. The data for this strip of pixels would be a series of numbers that we could plot on a graph. Assuming the image wasn't plain that graph would have peaks and troughs. The values of the pixels is our spatial representation of the data, while storing the frequency of the peaks and troughs is the frequency representation of the data.

The reason we go to this trouble is, again, to try to find ways to remove as many bits as possible while minimising the effect on how we see the image. It turns out that the eye is quite sensitive to low-frequency changes in an image (that is, gradual variation in colour or brightness), but finds it hard to detect the specifics of high-frequency changes (sudden changes in intensity, such as lines). In other words, we can detect subtle changes in brightness, but as far as the eye is concerned a line is pretty much just a line.

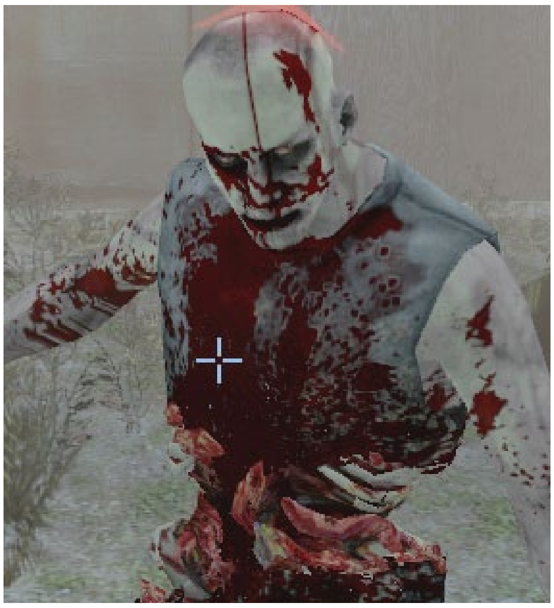
Because of this property of the eye, we can lose a lot of information from the high-frequency portion of the image while minimising what's noticeable. We do this by 'quantisation'. The means that we assign fewer bits to the higher frequency parts of the image than to the low frequency portions. There's no fixed way of doing this, and the amount you assign to each section of the frequency determines the amount of compression (and image quality) you'll finish with.

Once the DCT, and quantisation have been performed, a Huffman code [see **LXF175**, Tutorials, p87] is created to squeeze out any redundant information without affecting the image quality any further. The result is the JPEG version.

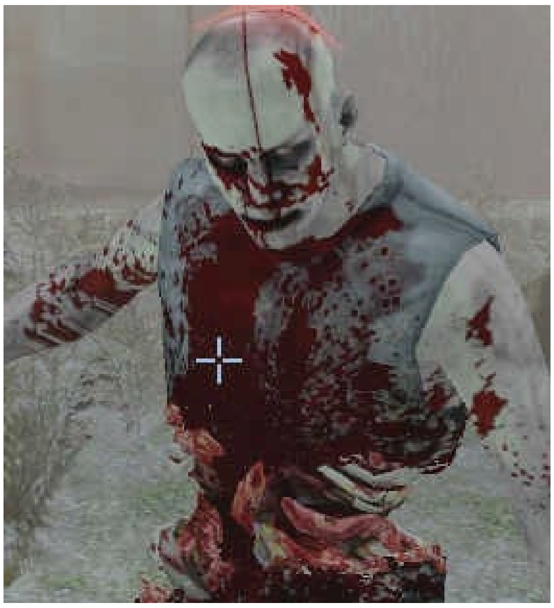
All lossy compression methods use similar techniques to remove as much information as possible while minimising the effect on the user. The science of compressing data in this way is all about understanding the limitations of humans, and exploiting these to hide the errors you inevitably introduce as you cast data aside. LXF

JPEG artifacts on the undead

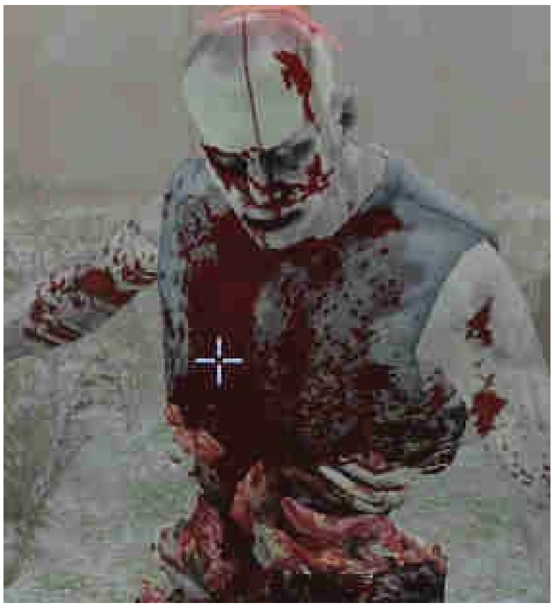
Errors introduced during the compression, known as artifacts, increase as you increase the level of compression. However, they don't increase uniformly across the image. Here are four JPEG images, with increasing levels of compression. Note how the high-frequency parts of the image are the first to degrade, while the low-frequency parts hold their quality longer.



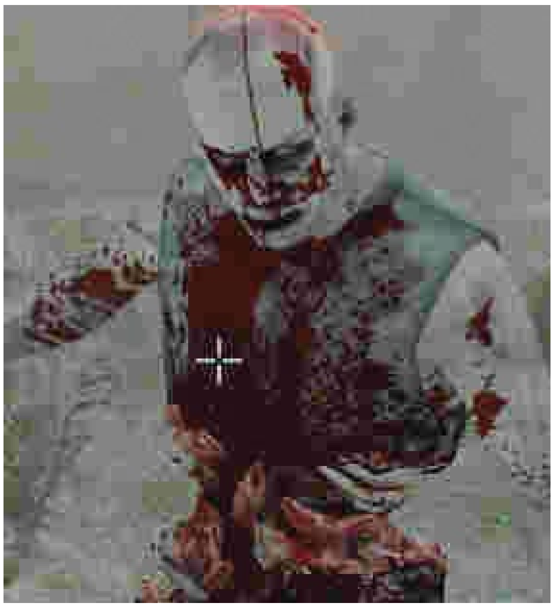
1 First we start will minimum compression. This is still 4:2:0 subsampled, but due to the nature of how our eyes work, we just can't see this.



2 Looking carefully, you'll notice how the high frequency parts of the images are beginning to lose definition as fewer bits are assigned to them.



3 The quantisation has now become so harsh that it's starting to affect the low-frequency portions, such as the edges of the zombie's arms.



4 At this level of compression, the high-frequency portions are almost entirely lost, yet at less than a single bit per pixel, the image is still identifiable.

$$f_{x,y} = \sum_{u=0}^{7} \sum_{v=0}^{7} \alpha(u)\,\alpha(v)\,F_{u,v} \cos\left[\frac{\pi}{8}\left(x+\frac{1}{2}\right)u\right] \cos\left[\frac{\pi}{8}\left(y+\frac{1}{2}\right)v\right]$$

› (Fig 1) This overly complex formula just splits an image up into its frequency components.