Minification techniques

- We have already discussed scaling images

- Enlarging an image well relies solely on good interpolation. We cannot add information to an image. Nearest neighbour will always give horrible results, the more so the larger the enlargement.

- Image Reduction, for example to make thumbnails of images has more options than enlargement.

- For reduction or "minification" we are throwing information away. The better the choices we make, the better the results.

**Mask based minification**

If we wish to reduce an image evenly, we can use a mask based approach.

3x3 mask will reduce to $1/3^{rd}$ size

5x5 mask will reduce to $1/5^{th}$ size etc.

Masks do not need to be square, or have odd length sides!

If we wish our resulting image to be a fixed size, we can calculate our mask size from those dimensions.

As we are reducing our image in size, edges do not present a problem – simpler but very similar to convolution.

Two common approaches:
**Median substitution**

Here we find the mid value of the pixel values under our mask.

E.g. for a 3x3 mask

- Place mask over image stepping through by both three rows and three columns. (We don't want any overlaps between masked values).
- Get the 9 pixels covered by the mask and sort (we've already met the standard quicksort function)
- Return the middle pixel value.

If we have an even mask we can either simply treat (int) size/2 as the “middle” value, or compute the mean of the two central values

- This is the first pixel of our new image.
- We now move the mask over by the width of the mask & repeat until end of image.

**Average substitution**

Average substitution is simpler but does not give as good quality reduction.

With average substitution exactly the same approach is followed, but rather than sorting and picking the mid value, we simply sum & divide by the number of pixels.

Consider the image values:

0, 0, 0, 1, 255, 1, 1, 1, 1.

Median substitution will give the new pixel value 1.

Average substitution will give the new pixel value 28
**Converting Greyscale to Bitmap**

We have already discussed thresholding, but thresholding does not retain image detail. In fact it is designed not to!

The simplest method for converting greyscale to bitmap images is *patterning*

**Patterning**

In patterning each greyscale pixel is replaced by a block of bitmap pixels

- We want to create random patterns, so we don’t get “blocky” looking images.
- We want to create the impression of the grey scale we are trying to emulate
- We will need \(N-1\) cells in our pattern where \(N\) is the number of greyscale levels we wish to represent
- We need to select a suitable pattern mask for these blocks. One reasonable choice is *Rylander’s matricies* (see next slide)
- Using this technique we generate an image much larger than our original (using Rylanders matricies, would result in an image four times the size of the original).
- If we wish to reduce the converted image, we could then apply one of the previously introduced minification techniques.
Applying Patterning using Rylander’s matricies

Here our approach is very simple.

- Take greyscale pixel values.
- Quantize the pixel to 17 levels (divide by $255/17 = 15$).
• (using a LUT would be ideal here)

<table>
<thead>
<tr>
<th>0-14</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>15-29</td>
<td>1</td>
</tr>
<tr>
<td>30-44</td>
<td>2</td>
</tr>
<tr>
<td>45-59</td>
<td>3</td>
</tr>
<tr>
<td>60-74</td>
<td>4</td>
</tr>
<tr>
<td>75-89</td>
<td>5</td>
</tr>
<tr>
<td>90-104</td>
<td>6</td>
</tr>
<tr>
<td>105-119</td>
<td>7</td>
</tr>
<tr>
<td>120-134</td>
<td>8</td>
</tr>
<tr>
<td>135-149</td>
<td>9</td>
</tr>
<tr>
<td>150-164</td>
<td>10</td>
</tr>
<tr>
<td>165-179</td>
<td>11</td>
</tr>
<tr>
<td>180-194</td>
<td>12</td>
</tr>
<tr>
<td>195-209</td>
<td>13</td>
</tr>
<tr>
<td>210-224</td>
<td>14</td>
</tr>
<tr>
<td>225-239</td>
<td>15</td>
</tr>
<tr>
<td>240-255</td>
<td>16</td>
</tr>
</tbody>
</table>

• Choose the array of pixels from the above table for the output image.

Programming problem:

• We need to handle 4 rows at a time! (4 columns is straightforward we can just increase our pointer by 4 at a time.

• We can allocate the memory required using malloc(), but we are left with complicated addressing.

• One solution is to create an array of pointers to each row.

• Given:

  ```
  unsigned char *rowpointer[height]
  ```

  we can first initialise the above array to point at the start of each row, and then simply use offsets for our width to simplify the programming.
• This however will give us an array of unsigned chars with 0 or 1 values. These would require packing into bytes.

Other File Formats

A common image processing problem is converting from one file format to another:

Why have so many?

Different purposes:

• Gamma translation (e.g. PNG)
  We want our image to appear the same to the human eye on different systems

• File size reduction (e.g. JPG)
  We want to maintain appearance and maximise compression.

• Transmission over unreliable communications links (e.g. GIF, FAX)
  We want a useful image to be communicated even if part of the transmission is corrupted or lost.

• Colour table communication (e.g. BMP)
  If we are using a reduced colour display (i.e. less than 24 bits), we may want to communicate our colour table.
Problems

- File format specification
  (how do we find them?, how do we code them?)

- Endian-ness
  (more later)

- Compression
  (lossy v. non-lossy, complex v. simple, how efficient?)

Solutions!

- For many popular file formats, prewritten libraries exist
- Read the documentation, compile & link!

OPTIONAL EXTRA: (*****)
Obtain libpng (http://ftp.libpng.org/pub/png/) and zlib (http://www.infozip.org/pub/infozip/zlib/) & build them. Use these as the basis to save as a PNG file with an assumed gamma & no text a PNM file. Note a comprehensive free program to do this interactively is available called wpng.

File format specifications

Easy to obtain from the WWW

E.g. http://www.dcs.ed.ac.uk/home/mxr/gfx

Or the comp.graphics USENET groups


**Endian-ness**

The term comes from Jonathon Swift’s “Gulliver’s travels” (The Brobdingnagians argued over which end an egg should be broken on!)

The term refers to how integer values are stored. We have avoided this issue by working with byte values. If we want to use integer values in our files, for them to be portable, we need to define the endian-ness of the file. Some computers will then need to do translation.

80x86 family are “little endian” (e.g. IBM-PC’s)

68xxx family are “big endian” (e.g. older Apple Macintosh/Suns)

Endianess is a problem since it determines the byte order of integer variables.

Consider the short int (16 bit value) 0x0100

On an 80x86 processor this would have the value 1 decimal.

On a 68xxx processor this would have the value 256

The little endian 80x86 stores integers as:

<table>
<thead>
<tr>
<th>LSB</th>
<th>MSB</th>
</tr>
</thead>
</table>

So FF00 stored in memory on a Pentium, would actually have the integer value 00FF.

Big endian architectures are the opposite!
Consider the following code:

```c
#include <stdio.h>

void main()
{
    unsigned int x = 0x31323334;

    char *p;
    p = (char *) &x;
    printf("%c %c %c %c\n", p[0], p[1], p[2], p[3]);
}
```

Under Windows on Intel this will print “4 3 2 1”

What will it do under Solaris on SPARC?

Most file formats specify an endianess.

As an aside most network protocols are big-endian.

**Compression**

Most image files support some form of compression.

At its simplest simply compare a PBM raw bitmap with a PBM ASCII bitmap (8 times bigger) or PNM files similarly.

Unfortunately, compression is beset by conflicting standards and patent problems.

It is rarely necessary to redevelop any complex compression software since most compression algorithms are available in public domain or open source libraries.
Lossless versus Lossy compression

Lossless compression is degradation free – what goes in comes back out. Examples would include GIF & TIF.

Lossy compression discards “redundant” data. The definition of redundant has to be carefully monitored! The most popular is JPEG (although the JPEG standard includes a lossless mode).

JPEG assumes that the images are:

- to be viewed by humans
- contain continuous tones (hence high frequency can be safely removed since fine colour changes cannot be seen by humans).

In general, lossy compression techniques are far more complex to implement than lossless ones!

Points to note:
- Lossless is safe regardless of intended use
- Lossless techniques can often be implemented in hardware
- Lossy techniques cannot be used for certain applications

Exercise
(**) for calling cjpeg/djpeg from your own program! (use the library function system())

If you do, the following is interesting:

- Compress the bullfight image using cjpeg and a low quality factor.
- Decompress with djpeg
- Subtract resulting image from original.
- For all colour channels, threshold result (value >0 becomes 255, 0 remains 0)
• Examine result. This is the information that has been thrown away!

We will not consider lossy methods further.

Lossless compression methods

**Run Length Encoding (RLE)**

Run length encoding is available within BMP for 4 bpp (bits per pixels) and 8 bpp images. It is also used in the GEM IMG standard.

Run length encoding works well where blocks of pixels have the same value.

This is commonly the case where files have a low bit depth, (examine the greyscale bullfighter you generated) …

…but not where there is a high bit depth “true colour” file.

RLE also brings us back to thinking about efficiency.

**Efficiency**

If we X bytes before compression and Y after, we have an X:Y compression ratio.

Efficiency has two sides:

• Effectiveness – how large is this ratio?

• Consistency – how much does it vary from file to file?
Back to RLE

The idea with run length encoding is rather than store:

```
AAABBBBBCCC
```

We store

```
3A6B9C
```

This gives good compression (3:1), but unfortunately where we have rapid changes

```
ABCDEF
```

we get:

```
1A1B1C1D1E1F
```

A negative compression! (1:2)

We therefore need to adapt so that unique blocks can be stored “as is”, with runs encoded. This requires a prefix symbol for the run count, e.g. in ASCII we could use +.

But then how do we encode the + character – As a run of 1 e.g. +1+.

Let’s consider a real example (from BMP, text taken from the BMP standard):

Bitmap Compression (BMP – Windows DIB)

*(DIB = Device Independent Bitmap)*

Windows versions 3.0 and later support run-length encoded (RLE) formats for compressing bitmaps that use 4 bits per pixel and 8 bits per pixel. Compression reduces the disk and memory storage required for a bitmap.
Compression of 8-Bits-per-Pixel Bitmaps

When the biCompression member of the BITMAPINFOHEADER structure is set to BI_RLE8, the DIB is compressed using a run-length encoded format for a 256-color bitmap. This format uses two modes: encoded mode and absolute mode. Both modes can occur anywhere throughout a single bitmap.

Encoded Mode

A unit of information in encoded mode consists of two bytes. The first byte specifies the number of consecutive pixels to be drawn using the color index contained in the second byte. The first byte of the pair can be set to zero to indicate an escape that denotes the end of a line, the end of the bitmap, or a delta. The interpretation of the escape depends on the value of the second byte of the pair, which must be in the range 0x00 through 0x02. Following are the meanings of the escape values that can be used in the second byte:

<table>
<thead>
<tr>
<th>Second byte</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>End of line.</td>
</tr>
<tr>
<td>1</td>
<td>End of bitmap.</td>
</tr>
<tr>
<td>2</td>
<td>Delta. The two bytes following the escape contain unsigned values indicating the horizontal and vertical offsets of the next pixel from the current position.</td>
</tr>
</tbody>
</table>

Absolute Mode

Absolute mode is signalled by the first byte in the pair being set to zero and the second byte to a value between 0x03 and 0xFF. The second byte represents the number of bytes that follow, each of which contains the colour index of a single pixel. Each run must be aligned on a word boundary. Following is an example of an 8-bit RLE bitmap (the two-digit hexadecimal values in the second column represent a colour index for a single pixel):

<table>
<thead>
<tr>
<th>Compressed data</th>
<th>Expanded data</th>
</tr>
</thead>
<tbody>
<tr>
<td>03 04</td>
<td>04 04 04</td>
</tr>
<tr>
<td>05 06</td>
<td>06 06 06 06 06</td>
</tr>
<tr>
<td>00 03 45 56 67 00</td>
<td>45 56 67</td>
</tr>
<tr>
<td>02 78</td>
<td>78 78</td>
</tr>
<tr>
<td>00 02 05 01</td>
<td>Move 5 right and 1 down</td>
</tr>
<tr>
<td>02 78</td>
<td>78 78</td>
</tr>
<tr>
<td>00 00</td>
<td>End of line</td>
</tr>
<tr>
<td>09 1E</td>
<td>1E 1E 1E 1E 1E 1E 1E</td>
</tr>
<tr>
<td>00 01</td>
<td>End of RLE bitmap</td>
</tr>
</tbody>
</table>

Points to note:
- Run length encoding has limited memory requirements
- Very variable compression
- Files may increase in size
- Simple

**Huffman Coding**

Huffman coding was state of the art between 1952 and 1977.

Whilst it has been superseded by other techniques for many applications, it is not just an historical curiosity.

Modified Huffman coding is the standard for FAX transmission and an optional method of compression for TIF images.

The basic idea borrows from Morse code and Shannon & Weavers information theory. Things we want to say more frequently have shorter codes.

For example, in Morse code:

‘e’  dot
‘q’  dash dash dot dash

Since the letter “e” is common, whilst the letter “q” is rare.
The basic method is:

Pass over the data generating a frequency table (histogram).

Rank these in a rank list.

Generate a binary tree containing a node for each used value.

- take the two lowest entries in the rank list, assign these to the tree with a parent (P), with a weight of their total frequency, remove from the rank list, but add P with its frequency.

- Repeat until only one node remains, this becomes the root of the tree. In the event of a tie between frequencies, any strategy may be used.

- Assigned bit values of 0 to each left branch, 1 to each right branch. These are the codes.

Points to note:
- Space efficient but computationally expensive.

- Requires 2 passes through source data.

- Tree must be communicated to decoder, either
  a) the tree or
  b) the frequency table. Both reduce compression ratio, b) also increases the computational cost of decoding.

An example from Crane, R; “A Simplified Approach to Image Processing” is on the next slide.
We first build our histogram and sort to produce a frequency table:

<table>
<thead>
<tr>
<th>Color</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>red</td>
<td>19</td>
</tr>
<tr>
<td>black</td>
<td>17</td>
</tr>
<tr>
<td>green</td>
<td>16</td>
</tr>
<tr>
<td>blue</td>
<td>5</td>
</tr>
<tr>
<td>cyan</td>
<td>4</td>
</tr>
<tr>
<td>magenta</td>
<td>2</td>
</tr>
<tr>
<td>yellow</td>
<td>1</td>
</tr>
</tbody>
</table>
We now build our tree:

1. 19 17 16 5 4 2 1
    R K G B C M Y

2. 19 17 16 5 4
    R K G B

3. 19 17 16 5
    R K G B

4. 19 17 16 12
    R K G

5. 19 17 28
    R K

6. 36 28
    R K G

7. R K G
    C M Y

B
**Modified Huffman Encoding**

Used for Bitmaps. Same idea, but the codes are chosen in advance (based on “typical” samples.) and specify runlength, however only runs of up to 64 bits are coded, longer runs are specified by adding a prefix code, with an EOL character also being specified.

**Coding Huffman Encoding**

More information is available from the link to Huffman encoding from this modules web page.

**Optional extra (*******)!**

- We need to build frequency table & sort
- We need to build tree
- We need to parse tree to build translation table.
- We need to communicate tree (or translation table)
- We need to pack bytes

**Note: These are ideas and suggestions NOT an implementation!**

1. Build Histogram
2. Build tree

**structure for nodes**

```c
struct tnode {
    unsigned char pattern;
    int value;
    struct tnode *left;
    struct tnode *right;
};
```
**root of tree structure**

struct tnode root;

**storage allocator for structures**

```c
struct tnode *tree_alloc(void)
{
    return (struct tnode *) malloc(sizeof(struct tnode));
}
```

**Build tree (e.g)**

```c
root.left = tree_alloc();
root.right = tree_alloc();

root.left->pattern = ‘M’;
root.left->value = 2;
root.left->left = NULL;
root.left->right = NULL;

root.right->pattern = ‘Y’;
root.right->value = 1;
root.right->left = NULL;
root.right->right = NULL

root.pattern = 0;
root.value = 3;
```

**Parse Tree**

See for example, Kernighan & Ritchie "The C Programming Language", or chapter 4 of Sedgewick's "Algorithms in C":

printhuff(struct tnode *p)
{
    if (p != NULL)
    {
        printhuff(p->left);
        printf("%c", p->pattern);
        printhuff(p->right);
    }
}

Communicate tree

How are we going to write the tree to a file?

Pack Bytes

Straightforward, but involves some coding. (Return to the C slides on printing a char in binary)

Lempel Ziv Welch (LZW) Compression

LZW compression was first suggested by Lempel & Ziv in 1977.

Welch published the definitive version, including how to implement in hardware in 1984 (hence LZW).

Welch’s contribution was to devise a code table which could be created the same way by both the compressor and decompressor.

LZW is a patented technology, and must be licensed for commercial use. This is the background to objections to the CompuServe GIF (Graphics Interchange Format) being used, one of the prime motivators for the development of the PNG (Portable Network Graphics) standard.

LZW compression is used by CompuServe GIF file formats and the unix compress (.Z) command

The basic idea is to encode strings or sequences as a single code word.
Most implementations use a 12 bit codeword to encode 8 bit data.

The 12 bit codeword gives $2^{12}$ codes (4096 table entries).

The first 256 of these are used to encode 8 bit values 0 through 255.

As new sequences are seen these are placed in the code table starting at location 256.

When the table is full, we can either continue using it, or reset the entries greater than 255.

GIF uses a more complex version of this process with variable length codewords and table resetting.

The idea is simple, the implementation often more complex!

C like Pseudo-code example

```c
FILE *fp;
char c;
char str[2048];

init_table()
str[0] = fgetc(fp);
while (!feof(fp))
{
    c = fgetc(fp);
    if ( in_table(strcat(str,c)))
        str = strcat(str,c);
    else
    {
        fprintf("%d", table_lookup(str));
        table_insert(strcat(str,c));
        str[0] = c;
    }
}
An Example from Crane:

We wish to compress BABAABAAA

Str = B
C = A
output 66 (for B), add BA to table

Str = A
C = B
output 65 (for A) add AB to table

Str = B
C = A
string = “BA”

C = A
output 256 for “BA”, add BAA to table

Str = A
C = B
string = “AB”

Str = AB
C = A
output 257 for “AB”, add ABA to table

Str = A
C = A
output 65 for “A” add, “AA” to table

Str = A
C=A
output 260 for “AA” end of file

For a real implementation, see the Dr Dobbs paper referenced on the website for this module.