OXFORD BROOKES UNIVERSITY

Other Media Primer

Contents

1. Introduction

1.1 The Browser

In the HTTP Primer, it was pointed out that most Web pages do not consist of a single file but often contain embedded images and other graphics which are not defined in HTML but in their own format. The browser may know how to deal with such formats or it may not. In the case that it does not, it can be enhanced with a **plug-in** that can handle the format or it may even call a **helper** application to deal with it for the browser. Thus there are three main options although the first and second appear quite similar to the user. If the browser supports the format or a plug-in has been added to enhance its ability, the image or whatever it is will appear as part of the browser window. If a helper application is called, it is likely that the image, video or whatever will appear in a separate window possibly with a quite different set of controls. Plug-ins, if they have their own menu options will either enhance the browser's controls or have a separate menu that pops up on a right button depression. Figure 1.1 shows a page that has been downloaded with an embedded **Scalable Vector Graphics (SVG)** drawing of a duck and a small **Graphics Interchange Format (GIF)** European Union Flag. If a **Moving Pictures Experts Group (MPEG)** video with an audio track was also to be played, it might call a **Real Player** helper application to display the video in a separate window.

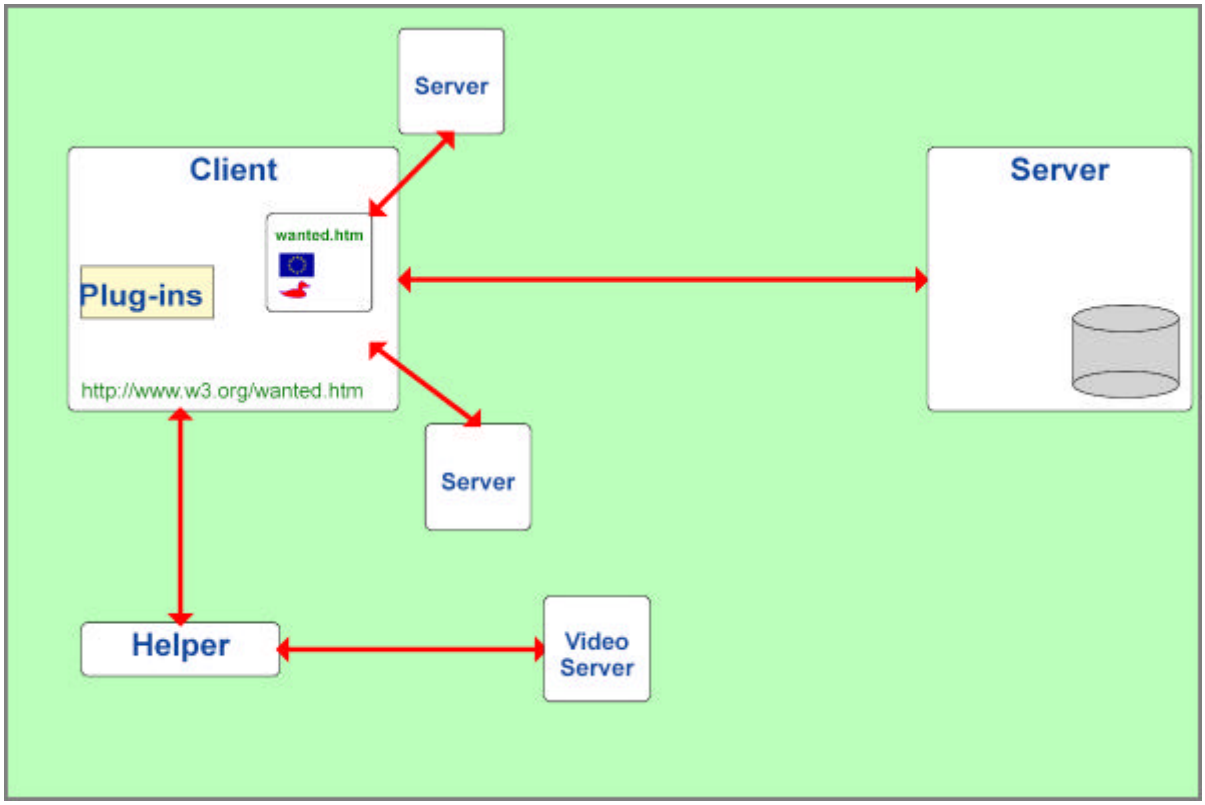


Figure 1.1: Plug Ins and Helper Applications

1.2 Some Formats

In this Primer, we will look at the following formats:

- WBMP, Wireless BitMaP Format: this simple image format is used in the context of WAP and mobile phones
- GIF, Graphics Interchange Format: the original graphics image format used on the Web
- JPEG, Joint Photographic Experts Group: the prefered image format for real world images, photographs etc
- PNG, Portable Network Graphics: the replacement image format for GIF
- CGM, Computer Graphics Metafile: the main vector drawing format before SVG (SVG will be covered in detail in a separate Primer)

We will also touch on some other formats. The aim is not to give a tutorial on image formats but more to indicate what should be used when and some of the important distinctions between the various formats.

1.3 Plug-ins

To give an idea of what a modern browser supports, IE5.5 currently supports GIF, PNG and JPEG itself. SVG, Adobe Acrobat, Macromedia Shockwave, Flash and CGM are supported by plug-ins. WBMP is not supported although it is supported by most of the browsers and simulators for mobile phones.

To add an image to an HTML page requires:

```
<object width="600" height="400" data="fig1p1.svg" type="image/svg+xml">
<img src=-"fig1p1.png" width="600" height="400" />
</object>
```

The object element gives the MIME type of the document requested. MIME (Multipurpose Internet Mail Extensions) media types define data types on the Internet. They consist of a major type (such as application, image, or text) followed by a minor type. By looking at the MIME type, the browser can make a decision what to do. If it supports the MIME type, it can download the file and process it. If it does not support it directly, it can see if it has been enhanced by a plug-in or helper to support that MIME type. The browser loads the appropriate plug-in into memory, creates a new instance of it, initializes it, and hands the data over to the plug-in. If there is no plug-in available, it can look at the second alternative which is to download a PNG equivalent. If the list is long enough, it will eventually find a format that can be handled.

In the case of the img element, no information about the file is known other than its file extension. Normally this can be used to deduce the type of the image.

The browser allows the plug-in to:

- Register one or more MIME types as objects that it can handle
- Draw into the browser's window at a point specified
- Receive keyboard and mouse events>These are passed on by the browser
- Receive information from the network via URLs. Thus the plug-in can nearly work autonomously if it downloads objects of the same type

When downloading a file from a Web server, the browser also receives the MIME type as part of the HTTP file. This can also be used to choose the plug-in but you have had to download the file to do it this way. When an img element refers to a local file, there is no MIME type information available, so the browser has to fall back on the file extensions that plug-ins have been registered for.

1.4 Compression

A good quality image, even a small one, can take up a lot of space. For example, a 400 by 400 pixel image which might cover a quarter of your display screen if you have an old low resolution display can be over 1 Mbyte in size if it has a reasonable colour range and is transparent in places. This creates several problems:

- It takes up a lot of space both on the server, proxies and caches and the browser
- The time to download may be of the order of 10 minutes
- HTTP is not really designed for files that size

In consequence, there is a need to **compress** the file. Unless the image is completely random, there will be areas where the information does not change rapidly or the image can be varied as subtle changes in the image may not be visible to the human eye. In consequence, much of the discussion will be around file size, speed of compression and decompression, intermediate storage required, etc.

2. WBMP

2.1 Introduction

One of the organizations that is bringing the Web to mobile phones is the Wireless Application Protocol (WAP) Consortium. The current mobile phones have limits in performance, memory and bandwidth. In consequence, if images are to be added to Web pages on mobile phones then an efficient protocol is needed. Here efficiency means not just that the number of bits transmitted is low but also that the processing of the information is low and that the memory requirements are low. To some extent, the bandwidth is less of a problem than the other two. None of the protocols in use were aiming at such constraints. Consequently, the WAP Consortium defined a new protocol called Wireless BitMap (WBMP) with a media type image/x-wap and a file extension of .wbmp.

2.2 Layout

The WBMP protocol is extremely simple. The first four bytes of a file are something like:

```
00000000
00000000
01100000
01000001
```

The first 8 bits define the type of WBMP. Only one type is currently available and that is Type 0. This states the following:

- There is no compression technique employed
- The picture is black and white with a **1** bit indicating **white** and a **0** bit indicating **black**.
- The high order bit of each byte is the left-most bit of the byte
- The first row of the image is the top row in the image

The second byte (also all 0's) states that this is the last field of the header and that no extension headers follows.

The third byte has the binary value of 96 in the example, which is the number of pixels in each row. The fourth byte has the binary value of 65 that indicates that there are 65 rows in the image.

The image data follows immediately after the width and height. In the case above that means that the next 12 bytes define the first row of the image and so on. If the row was not an exact number of bytes in length, say it had been 94, then the last few bits in the byte are not used and are set to 0. Thus each row starts on a byte boundary. The 4-byte header above would therefore be followed by 780 bytes (96 times 65 bits) defining the image. Something like:

```
11111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111100000000000000000111111111111111111111111111111111111111111111111
11111111111111111111111111111111111110000000000000000000001111111111111111111111111111111111111111111111
11111111111111111111111111111111111000000000000000000000000011111111111111111111111111111111111111111111
11111111111111111111111111111111100000000000000000000000000001111111111111111111111111111111111111111111
11111111111111111111111111111111000000000000000000000000000000111111111111111111111111111111111111111111
11111111111111111111111111111110000000000000000000000000000000011111111111111111111111111111111111111111
11111111111111111111111111111100000000000000000000000000000000001111111111111111111111111111111111111111
11111111111111111111111111111000000000000000000000000000000000000111111111111111111111111111111111111111
11111111111111111111111111110000000000000000000000000000000000000011111111111111111111111111111111111111
11111111111111111111111111100000000000000000000000000000000000000001111111111111111111111111111111111111
11111111111111111111111111000000000000000000000000000000000000000000111111111111111111111111111111111111
11111111111111111111111110000000000000000000000000000000000000000000011111111111111111111111111111111111
11111111111111111111111100000000000000000000000000000000000000000000001111111111111111111111111111111111
11111111111111111111111000000000000000000000000000000000000000000000000111111111111111111111111111111111
11111111111111111111110000000000000000000000000000000000000000000000000011111111111111111111111111111111
11111111111111111111100000000000000000000000000000000000000000000000000001111111111111111111111111111111
11111111111111111111000000000000000000000000000000000000000000000000000000111111111111111111111111111111
11111111111111111110000000000000000000000000000000000000000000000000000000011111111111111111111111111111
11111111111111111100000000000000000000000000000000000000000000000000000000001111111111111111111111111111
11111111111111111000000000000000000000000000000000000000000000000000000000000111111111111111111111111111
11111111111111110000000000000000000000000000000000000000000000000000000000000011111111111111111111111111
11111111111111100000000000000000000000000000000000000000000000000000000000000001111111111111111111111111
11111111111111000000000000000000000000000000000000000000000000000000000000000000111111111111111111111111
11111111111110000000000000000000000000000000000000000000000000000000000000000000011111111111111111111111
11111111111100000000000000000000000000000000000000000000000000000000000000000000001111111111111111111111
11111111111000000000000000000000000000000000000000000000000000000000000000000000000111111111111111111111
11111111110000000000000000000000000000000000000000000000000000000000000000000000000011111111111111111111
11111111100000000000000000000000000000000000000000000000000000000000000000000000000001111111111111111111
11111111000000000000000000000000000000000000000000000000000000000000000000000000000000111111111111111111
11111110000000000000000000000000000000000000000000000000000000000000000000000000000000011111111111111111
11111100000000000000000000000000000000000000000000000000000000000000000000000000000000001111111111111111
11111000000000000000000000000000000000000000000000000000000000000000000000000000000000000111111111111111
11110000000000000000000000000000000000000000000000000000000000000000000000000000000000000011111111111111
11100000000000000000000000000000000000000000000000000000000000000000000000000000000000000001111111111111
11000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000111111111111
10000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000011111111111
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000001111111111
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000111111111
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000011111111
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000001111111
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000111111
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000011111
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000001111
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000111
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000011
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000001
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
```


It is not too difficult to work out that the image is of a black circle on a white background.

This particular image is also displayed in Figure 2.1.

Figure 2.1: A WBMP Picture

2.3 Comments

The image actually displayed in Figure 2.1 is in the Graphics Interchange Format (GIF) as WBMP is mainly aimed at the mobile phone market and is not supported by either IE or Adobe products. The WBMP file is 784 bytes long and the GIF file is 242 bytes. We will look at GIF shortly and see how it manages to do this. The main comment for now is that this format is so simple that the bits as they are received can be displayed immediately. The mobile phone will have a set of bits that represent the display's screen and the image can be downloaded straight into the image. No offscreen storage is needed at all. The program is very short also. The first few bytes determine the number of bytes defining each row and the only real problem is making sure that the few bits left over at the end of a row do not corrupt the display.

In fact this image is quite a large image for a mobile phone and if we decreased the size we would eventually come to a position where the WBMP image is actually smaller than the equivalent GIF image. This is because the compression achieved by GIF becomes less effective and the GIF Header becomes more dominant in term of storage requirements.

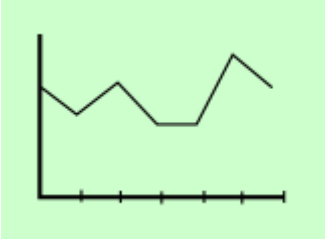


Figure 2.2: Another WBMP Picture

Figure 2.1 shows a slightly larger image that is 161 by 119 pixels and in this case the WBMP file is 2504 bytes compared with (a rather inefficiently defined) GIF of 1377 bytes.

3. Compression

3.1 Introduction

The aim of data compression is to reduce the number of bytes that need to be transmitted. This is even more important with images than with HTML pages just due to their large size. We may be talking Megabytes rather than Kilobytes and if we move to audio or video we can be talking Gigabytes. The main aim is to achieve a high **compression ratio** at low cost. Compression ratio is the number of bytes in the original image compared with the number of bytes transmitted. Compression ratios of at least 10:1 are what we are looking for.

To achieve this, it is clearly a lot easier if we throw information away. If the image we transmit **looks** the same as the original image that may be sufficient. This is called **lossy** compression. If we insist that the image reconstructed after transmission is identical to the one we start with then that is called **lossless** compression which is clearly more difficult in terms of achieving a high compression ratio.

An **encoder** prepares the information for transmission and a **decoder** after transmission recreates the original image. In the case of WBMP, both are very trivial pieces of software. We may choose to make one or other more complex. For example, if you only plan to compress the image once but have it decompressed many times, having the encoder quite complex in order to make the decoder simpler would be an advantage.

There is also the user's **perceived** view of the process. If the user can see part of the image before the remainder has been transmitted, a decision can be made as to whether it is required or not. A major way of doing this is to break the image into **passes** and not transmit the image from top to bottom but instead deliver, say, every 8th row, followed by every 4th row then every second and so on. This is called **interlacing** and is the standard way of transmitting television pictures. Every other row is transmitted first followed by the other rows.

To first order there are three approaches that can be taken:

1. Transform the data to make it smaller. A long series of 1s and 0s in the WBMP image may be transformed to some other representation.
2. Reduce the precision. If colour values are 16 bits each for red, green and blue for each pixel and the image is to be displayed on a system only capable of displaying 8 bits of precision, it is not sensible to transmit all 16 bits.
3. Change the representation of the data. The more common data items, say the primary colours, might be stored using less bits than the less common colours.

Most compression techniques try and use all three to achieve the compression ratio required.

3.2 Run Length Encoding

Let us start by looking at a fairly simple approach, **run length encoding**. The simplest case is to take multiple values that are the same and replace them by the value and a count of how many times it occurs. In the WBMP example in Figure 1.1, we need to make some decisions as to how to encode the information. We could, for example, choose to have individual bytes containing:

BAAAAAAA

The first bit B represents the colour. If B is set to 1 then it is white and if set to 0 it is black. The rest of the byte indicates the number of bits in the row starting from the left that have that value. The result for the WBMP image would be:

```
224
224
168 16 168 165 22 165 163 26 163 161 30 161
159 34 159 158 36 158 156 39 157 155 42 155
154 43 155 153 46 153 152 47 153 152 48 152
151 50 151 150 52 150 150 52 150 149 54 149
149 54 149 148 56 148 148 56 148 147 58 147
147 58 147 147 58 147 146 60 146 146 60 146
146 60 146 146 60 146 146 60 146 146 60 146
146 60 146 146 60 146 146 61 145 146 60 146
146 60 146 146 60 146 146 60 146 146 60 146
146 60 146 146 60 146 147 58 147 147 58 147
147 58 147 148 56 148 148 56 148 149 54 149
149 54 149 150 52 150 150 52 150 151 50 151
152 48 152 152 47 153 153 46 153 154 43 155
155 42 155 156 39 157 158 36 158 159 34 159
161 30 161 163 26 163 165 22 165 168 16 168
224
224
224
```

The first two and last three rows are defined using a single byte and the intervening rows each take three bytes making 185 bytes compared with the 780 bytes that we started with, a compression ratio of just over 4. We could try looking for larger patterns and have a byte in front of the pattern giving in the top half the number of bytes to repeat and the bottom half giving the number of times. This would result in:

```
18 224
193
168 16 168 165 22 165 163 26 163 161 30 161
193
159 34 159 158 36 158 156 39 157 155 42 155
193
154 43 155 153 46 153 152 47 153 152 48 152
49
151 50 151
50
150 52 150
50
149 54 149
50
148 56 148
51
147 58 147
56
146 60 146
49
146 61 145
54
146 60 146
51
147 58 147
50
148 56 148
50
149 54 149
50
150 52 150
49
151 50 151
193
152 48 152 152 47 153 153 46 153 154 43 155
193
155 42 155 156 39 157 158 36 158 159 34 159
193
161 30 161 163 26 163 165 22 165 168 16 168
19 224
```

The first item repeats the 224 byte twice. The 193 byte (16 * 12 +1) repeats the next 12 bytes once. The savings come when we have the 56 byte indicating that the next 3 bytes should be repeated 8 times. In all the size is now 134 bytes, a compression ratio approaching 6. The question will soon come as to whether we are employing compression techniques that only work for circles. Clearly the compression is good for any large black object and if it is regular that will improve the compression. However, the compression for Figure 5.2 would not be as good. On the other hand it would not be bad also.

Run length encoding is used by a number of data encodings including CCiTT Fax, TARGA, TIFF (Tag Image File Format often used for output to printers and input from scanners) and RLE. It is usually difficult to get a compression ratio greater than 5 with just run length encoding and so has limited appeal.

3.3 Code Tables

We can build on the idea of finding patterns introduced in the previous section. As early as the 1950s, compilers and text retrieval systems were building large tables of commonly used expressions or words and then replacing the textual information by the index into the table. For example, paraphrased from The House at Pooh Corner by A. A. Milne:

```
So after breakfast they went round to see Piglet and Pooh explained
as they went that Piglet was a very small animal who did not like bouncing
and asked Tigger not to be too bouncy just at first. And Tigger who had
been hiding behind trees said that a Tigger was only bouncy before
breakfast
and that as soon as they had had breakfast they became quiet and refined.
```

We could make up a table (ignoring capitals for the moment) containing all the words as follows:

01	a	12	behind	23	not	34	soon
02	after	13	bouncing	24	only	35	that
03	and	14	bouncy	25	piglet	36	they
04	animal	15	breakfast	26	pooh	37	tigger
05	as	16	did	27	quiet	38	to
06	asked	17	explained	28	refined	39	too
07	at	18	first	29	round	40	trees
08	be	19	had	30	said	41	very
09	became	20	hiding	31	see	42	was
10	been	21	just	32	small	43	went
11	before	22	like	33	so	44	who

The text could now be written as:

```
33 02 15 36 43 29 38 31 25 03 26 17
05 36 43 35 25 42 01 41 32 04 44 16 23 22 13
03 06 37 23 38 08 39 14 21 07 18 03 37 44 19
10 20 12 40 30 35 01 37 42 24 14 11 15
03 35 05 34 05 36 19 19 15 36 09 27 03 28
```

The original text is about 366 bytes long and the index values could be stored two per byte and only requires 35 bytes.

Unfortunately, the receiver does not know what the codes mean unless we also send the table and that requires at least 240 bytes so the saving is not that great if we have also to send the table giving what the codes mean. There are some compactions that you can do on the table but we are not going to get a high compression ratio as it stands unless the word reuse is high and we send a significantly large body of text that multiple occurrences of words appear very frequently.

3.4 Lempel and Ziv

Using codes for symbols is the basis of the algorithm invented by Lempel and Ziv in 1977. You can start with an empty **Code Table** but it is more usual to put the set of alphabetic symbols and space in the Code Table initially. As symbols (characters) are read, the following algorithm is executed:

```
prefix= emptystring;

repeat
  get nextcharacter;
  if prefix + nextcharacter is in the Code Table then prefix = prefix +
nextcharacter
  else {
      add prefix + nextcharacter to the Code Table;
      output the code of the prefix from the Code Table
      set the prefix to the nextcharacter       }
until complete;
output the code of the last prefix
```

The Code Table produced is:

0	_	35	_b	70	et	105	ery	140	d_t	175	bee	210	re_
1	a	36	br	71	t_a	106	y_s	141	ti	176	en_	211	_br
2	b	37	re	72	an	107	sm	142	igg	177	_h	212	rea
3	c	38	ea	73	nd_	108	ma	143	ge	178	hi	213	akf
4	d	39	ak	74	_po	109	al	144	er_	179	idi	214	fas
5	e	40	kf	75	oo	110	ll	145	_n	180	ing_	215	st_an
6	f	41	fa	76	oh	111	l_	146	not	181	_be	216	nd_th
7	g	42	as	77	h_	112	_an	147	t_t	182	eh	217	hat_
8	h	43	st	78	_e	113	ni	148	to	183	hin	218	_as
9	i	44	t_	79	ex	114	im	149	o_b	184	nd_t	219	s_s
10	j	45	_t	80	xp	115	mal	150	be	185	tr	220	soo
11	k	46	th	81	pl	116	l_w	151	e_t	186	ree	221	on_
12	l	47	he	82	la	117	wh	152	too	187	es	222	_as_
13	m	48	ey	83	ai	118	ho	153	o_bo	188	s_	223	_the
14	n	49	y_	84	in	119	o_d	154	ounc	189	_s	224	ey_
15	o	50	_w	85	ne	120	di	155	cy	190	sa	225	_ha
16	p	51	we	86	ed	121	id	156	y_j	191	aid	226	ad_
17	q	52	en	87	d_a	122	d_n	157	ju	192	d_th	227	_had
18	r	53	nt	88	as_	123	no	158	us	193	hat	228	d_br
19	s	54	t_r	89	_th	124	ot	159	st_	194	t_a_	229	reak
20	t	55	ro	90	hey	125	t_l	160	_at	195	_ti	230	kfa
21	u	56	ou	91	y_w	126	li	161	t_f	196	igger	231	ast
22	v	57	un	92	wen	127	ik	162	fi	197	r_w	232	t_th
23	w	58	nd	93	nt_	128	ke	163	ir	198	wa	233	hey_
24	x	59	d_	94	_tha	129	e_b	164	rs	199	as_o	234	_bec
25	y	60	_to	95	at	130	bo	165	st_a	200	on	235	ca
26	z	61	o_s	96	t_p	131	oun	166	and	201	nl	236	am
27	so	62	se	97	pig	132	nc	167	d_ti	202	ly	237	me
28	o_	63	ee	98	gle	133	ci	168	igge	203	y_b	238	e_q
29	_a	64	e_	99	et_	134	ing	169	er_w	204	bou	239	qu
30	af	65	_p	100	_wa	135	g_	170	who	205	unc	240	ui
31	ft	66	pi	101	as_a	136	_and	171	o_h	206	cy_	241	ie
32	te	67	ig	102	a_	137	d_as	172	ha	207	_bef	242	et_a
33	er	68	gl	103	_v	138	sk	173	ad	208	fo	243	and_
34	r_	69	le	104	ve	139	ked	174	d_b	209	or	244	_r

The output would be:

0	**19**	35	**19**	70	**66**	105	**14**	140	**140**	175	**12**	210	**13**
1	**15**	36	**5**	71	**68**	106	**3**	141	**142**	176	**49**	211	**64**
2	**0**	37	**5**	72	**70**	107	**84**	142	**144**	177	**130**	212	**17**
3	**1**	38	**0**	73	**50**	108	**7**	143	**117**	178	**57**	213	**21**
4	**6**	39	**16**	74	**88**	109	**112**	144	**28**	179	**155**	214	**9**
5	**20**	40	**9**	75	**1**	110	**87**	145	**8**	180	**181**	215	**99**
6	**5**	41	**7**	76	**0**	111	**19**	146	**1**	181	**6**	216	**166**
7	**18**	42	**12**	77	**22**	112	**128**	147	**59**	182	**15**	217	**0**
8	**0**	43	**5**	78	**33**	113	**59**	148	**150**	183	**37**	218	**37**
9	**2**	44	**44**	79	**49**	114	**20**	149	**52**	184	**35**	219	**162**
10	**18**	45	**1**	80	**19**	115	**67**	150	**0**	185	**37**	220	**85**
11	**5**	46	**58**	81	**13**	116	**7**	151	**8**	186	**39**	221	**4**
12	**1**	47	**65**	82	**1**	117	**33**	152	**121**	187	**41**	222	
13	**11**	48	**15**	83	**12**	118	**0**	153	**134**	188	**165**	223	
14	**6**	49	**15**	84	**12**	119	**123**	154	**35**	189	**184**	224	
15	**1**	50	**8**	85	**29**	120	**44**	155	**5**	190	**193**	225	
16	**19**	51	**0**	86	**14**	121	**20**	156	**178**	191	**29**	226	
17	**20**	52	**5**	87	**9**	122	**28**	157	**73**	192	**188**	227	
18	**0**	53	**24**	88	**108**	123	**2**	158	**20**	193	**27**	228	
19	**20**	54	**16**	89	**111**	124	**64**	159	**37**	194	**200**	229	
20	**8**	55	**12**	90	**23**	125	**148**	160	**5**	195	**218**	230	
21	**5**	56	**1**	91	**8**	126	**149**	161	**19**	196	**89**	231	
22	**25**	57	**9**	92	**28**	127	**131**	162	**0**	197	**48**	232	
23	**0**	58	**14**	93	**4**	128	**3**	163	**19**	198	**177**	233	
24	**23**	59	**5**	94	**9**	129	**49**	164	**83**	199	**173**	234	
25	**5**	60	**59**	95	**59**	130	**10**	165	**140**	200	**225**	235	
26	**14**	61	**42**	96	**14**	131	**21**	166	**172**	201	**174**	236	
27	**44**	62	**45**	97	**15**	132	**43**	167	**71**	202	**212**	237	
28	**18**	63	**47**	98	**44**	133	**29**	168	**45**	203	**40**	238	
29	**15**	64	**49**	99	**12**	134	**44**	169	**168**	204	**42**	239	
30	**21**	65	**51**	100	**9**	135	**6**	170	**34**	205	**147**	240	
31	**14**	66	**53**	101	**11**	136	**9**	171	**23**	206	**90**	241	
32	**4**	67	**89**	102	**64**	137	**18**	172	**88**	207	**181**	242	
33	**45**	68	**1**	103	**2**	138	**159**	173	**15**	208	**3**	243	
34	**28**	69	**44**	104	**56**	139	**72**	174	**14**	209	**1**	244	**-1**

One thing that is obvious immediately is that it takes a while to build up entries in the Code Table that are able to cut the number of bytes transmitted. On the face of it we are worse off than before as we have to transmit 220 code values (not much of a compression considering we started with 366 characters and we have a Code Table with 247 entries. But there is some good news:

- The maximum number we have to send is 221 so all the output numbers fit into a byte.
- The first 100 numbers are all less than 63 so could fit into 6 bits each.
- As we build up the transmitted information, we have enough information to construct the Code Table that comes after the basic symbols so the whole table does not need to be transmitted (just the first 27 entries).

The last point is the major advantage of the set of Lempel and Ziv algorithms, They spent a number of years refining the basic algorithm described so far.

The decoder will look something like:

```
initialize the Code Table with entries 0 to 26
set Code to the first input value; look up in Code Table and output

repeat
OldCode=Code;
Code=next input value;
if in Code Table
then {
  look up in Code Table and output
  Prefix=OldCode entry;
  Suffix=First Value from the Code Table entry;
  }
else {
  Prefix=OldCode entry;
  Suffix=First Value from the Prefix;
  Output Prefix and Suffix;   }
add Prefix+Suffix as an entry in the Code Table; until complete;
```

What happens if we try to compress the circle shown in Figure 2.1? We can start with a Code Table just containing two entries 0 and 1 and attempt to compress the circle. The resulting Code Table is 235 entries long which are all pretty boring so we have just shown the first 105:

0	0	35	00000000	70	0001
1	1	36	000000000	71	11111111111111111111111111111111111
2	11	37	011	72	11111111111111111110
3	111	38	111111111111111111111111	73	000000000000000000000
4	1111	39	1111111111111111111111111	74	0000000000000000000001
5	11111	40	11111111111111111110	75	11111111111111111111111111111111111
6	111111	41	0000000000	76	111111111111111110
7	1111111	42	00000000000	77	000000000000000000000
8	11111111	43	00000001	78	0000000000000000000000
9	111111111	44	11111111111111111111111111	79	0111
10	1111111111	45	111111111111111111111111111	80	11111111111111111111111111111111111
11	11111111111	46	11111111111110	81	111111111110
12	111111111111	47	000000000000	82	000000000000000000000000
13	1111111111111	48	0000000000000	83	000000000000000000000001
14	11111111111111	49	000000011	84	11111111111111111111111111111111111
15	111111111111111	50	1111111111111111111111111111	85	1111111111110
16	1111111111111111	51	11111111111111111111111111111	86	000000000000000000000000
17	11111111111111111	52	11110	87	000000000000000000000011
18	111111111111111111	53	00000000000000	88	11111111111111111111111111111111111
19	1111111111111111111	54	000000000000000	89	1111111100
20	11111111111111111111	55	0000000111	90	000000000000000000000000000
21	111111111111111111111	56	1111111111111111111111111111111	91	000000000000000000000000111
22	1111111111111111111111	57	11111111111111111111111111110	92	11111111111111111111111111111111111111
23	10	58	0000000000000000	93	111100
24	00	59	00000000000000000	94	000000000000000000000000000
25	000	60	000001	95	000000000000000000000001
26	0000	61	1111111111111111111111111111111	96	11111111111111111111111111111111111111
27	00000	62	111111111111111111111111110	97	1111000
28	000000	63	000000000000000000	98	0000000000000000000000000000
29	01	64	0000000000000000000	99	000000000000000000000011
30	11111111111111111111	65	00001	100	11111111111111111111111111111111111111
31	111111111111111111111	66	1111111111111111111111111111111	101	100
32	1111111111111111111111111	67	11111111111111111110	102	000000000000000000000000000
33	111111110	68	0000000000000000000	103	000000000000000000000011
34	0000000	69	00000000000000000000	104	11111111111111111111111111111111111111110

The compressed circle is:

0	**1**	35	**29**	70	**19**	105	**88**	140	**54**	175	**174**	210	**126**
1	**2**	36	**32**	71	**69**	106	**105**	141	**134**	176	**79**	211	**210**
2	**3**	37	**38**	72	**73**	107	**82**	142	**141**	177	**158**	212	**9**
3	**4**	38	**20**	73	**71**	108	**92**	143	**142**	178	**162**	213	**120**
4	**5**	39	**36**	74	**17**	109	**108**	144	**137**	179	**110**	214	**213**
5	**6**	40	**41**	75	**73**	110	**106**	145	**144**	180	**177**	215	**10**
6	**7**	41	**34**	76	**77**	111	**75**	146	**58**	181	**104**	216	**114**
7	**8**	42	**39**	77	**37**	112	**111**	147	**146**	182	**171**	217	**216**
8	**9**	43	**44**	78	**75**	113	**109**	148	**147**	183	**100**	218	**85**
9	**10**	44	**13**	79	**11**	114	**80**	149	**48**	184	**174**	219	**98**
10	**11**	45	**42**	80	**78**	115	**114**	150	**143**	185	**100**	220	**219**
11	**12**	46	**47**	81	**77**	116	**112**	151	**150**	186	**101**	221	**15**
12	**13**	47	**43**	82	**80**	117	**66**	152	**42**	187	**180**	222	**86**
13	**14**	48	**45**	83	**12**	118	**117**	153	**152**	188	**187**	223	**222**
14	**15**	49	**50**	84	**82**	119	**91**	154	**153**	189	**168**	224	**18**
15	**16**	50	**4**	85	**83**	120	**119**	155	**36**	190	**187**	225	**130**
16	**17**	51	**48**	86	**84**	121	**120**	156	**155**	191	**2**	226	**225**
17	**18**	52	**53**	87	**33**	122	**121**	157	**156**	192	**189**	227	**72**
18	**19**	53	**49**	88	**86**	123	**56**	158	**55**	193	**192**	228	**145**
19	**20**	54	**51**	89	**87**	124	**123**	159	**61**	194	**188**	229	**228**
20	**21**	55	**44**	90	**88**	125	**124**	160	**159**	195	**147**	230	**231**
21	**1**	56	**54**	91	**52**	126	**51**	161	**154**	196	**195**	231	**232**
22	**0**	57	**58**	92	**90**	127	**126**	162	**149**	197	**3**	232	**233**
23	**24**	58	**27**	93	**78**	128	**99**	163	**162**	198	**153**	233	**234**
24	**25**	59	**56**	94	**92**	129	**122**	164	**160**	199	**198**	234	**71**
25	**26**	60	**38**	95	**93**	130	**129**	165	**125**	200	**97**	235	**-1**
26	**27**	61	**59**	96	**94**	131	**64**	166	**165**	201	**141**	236	**-1**
27	**0**	62	**63**	97	**74**	132	**113**	167	**35**	202	**201**	237	**-1**
28	**22**	63	**26**	98	**96**	133	**132**	168	**116**	203	**202**	238	**-1**
29	**30**	64	**61**	99	**23**	134	**133**	169	**168**	204	**129**	239	**-1**
30	**31**	65	**30**	100	**98**	135	**71**	170	**60**	205	**204**	240	**-1**
31	**8**	66	**64**	101	**95**	136	**135**	171	**170**	206	**6**	241	**-1**
32	**28**	67	**68**	102	**96**	137	**136**	172	**171**	207	**138**	242	**-1**
33	**34**	68	**25**	103	**102**	138	**131**	173	**70**	208	**207**	243	**-1**
34	**35**	69	**66**	104	**103**	139	**138**	174	**173**	209	**208**	244	**-1**

The compressed output values each fit into a byte so it will compress the circle down to under 240 bytes. This is not as good as the run length encoding but it will handle a greater range of images with a similar level of compression.

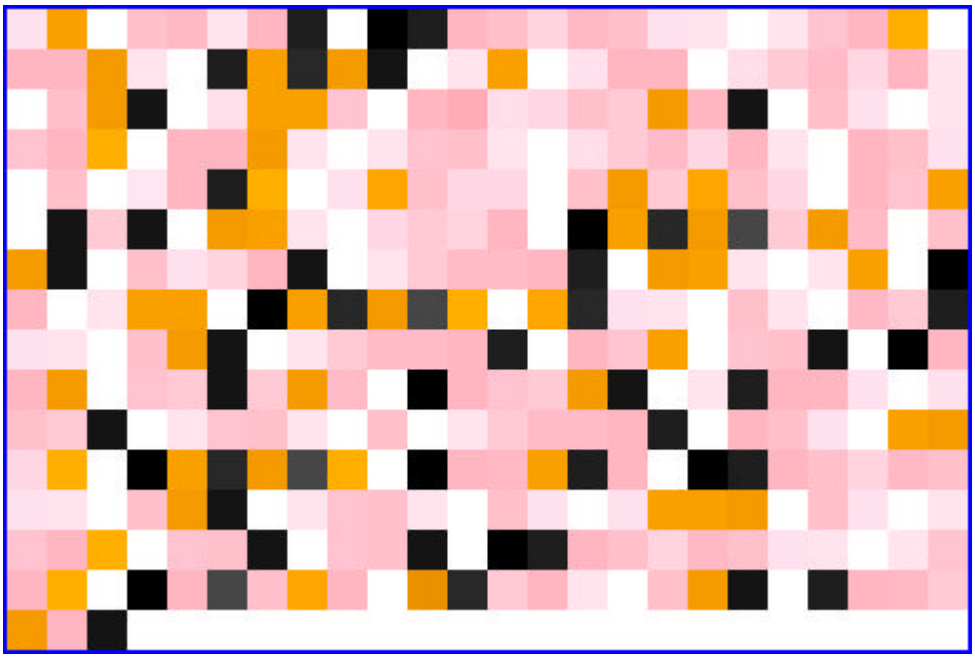


Figure 3.1: Modern Art: Piglet meets Tigger

Figure 3.1 shows a piece of modern art which is an image that might appear in a Web page devoted to A A Milne. The question now comes as to how should we compress images like this. Lempel and Ziv seems to manage reasonably well on black and white images with a great deal of repetition but how would it handle an image like this. The image is 24 pixels across by 16 pixels deep (we have magnified each pixel so it can be seen more easily. The orange and black represent Tigger and the pink shades represent Piglet. There are actually about 30 different colours in the picture and each pixel is defined by giving the Red, Blue and Green (RGB) intensities for each pixel.

The table below gives the colours used:

01	rgb(255,192,203)	16	rgb(255,222,233)
02	rgb(0,0,0)	17	rgb(235,145,0)
03	rgb(70,70,70)	18	rgb(30,30,30)
04	rgb(20,20,20)	19	rgb(255,225,238)
05	rgb(255,182,193)	20	rgb(255,228,238)
06	rgb(255,185,198)	21	rgb(40,40,40)
07	rgb(255,187,200)	22	rgb(255,230,240)
08	rgb(255,195,208)	23	rgb(255,182,193)
09	rgb(255,202,213)	24	rgb(255,172,183)
10	rgb(255,165,0)	25	rgb(255,175,0)
11	rgb(255,212,223)	26	rgb(0,0,255)
12	rgb(255,215,228)	27	rgb(55,55,55)
13	rgb(255,165,0)	28	rgb(155,155,155)
14	rgb(245,155,0)	29	rgb(205,205,205)
15	rgb(250,160,0)	30	rgb(225,225,225)

The image itself consists of:

```
19 15 27 01 06 20 05 18 27 02 18 05 01 11 06 01 19 20 27 20 08 05 25 27
23 05 14 20 27 18 15 21 14 04 27 20 15 27 19 05 05 27 16 09 07 12 05 20
27 01 14 04 27 16 15 15 08 27 05 24 16 12 01 09 14 05 04 27 01 19 27 20
08 05 25 27 23 05 14 20 27 20 08 01 20 27 16 09 07 12 05 20 27 23 01 19
27 01 27 22 05 18 25 27 19 13 01 12 12 27 01 14 09 13 01 12 27 23 08 15
27 04 09 04 27 14 15 20 27 12 09 11 05 27 02 15 21 14 03 09 14 07 27 01
14 04 27 01 19 11 05 04 27 20 09 07 07 05 18 27 14 15 20 27 20 15 27 02
05 27 20 15 15 27 02 15 21 14 03 25 27 10 21 19 20 27 01 20 27 06 09 18
19 20 27 01 14 04 27 20 09 07 07 05 18 27 23 08 15 27 08 01 04 27 02 05
05 14 27 08 09 04 09 14 07 27 02 05 08 09 14 04 27 20 18 05 05 19 27 19
01 09 04 27 20 08 01 20 27 01 27 20 09 07 07 05 18 27 23 01 19 27 15 14
12 25 27 02 15 21 14 03 25 27 02 05 06 15 18 05 27 02 18 05 01 11 06 01
19 20 27 01 14 04 27 20 08 01 20 27 01 19 27 19 15 15 14 27 01 19 27 20
08 05 25 27 08 01 04 27 08 01 04 27 02 18 05 01 11 06 01 19 20 27 20 08
05 25 27 02 05 03 01 13 05 27 17 21 09 05 20 27 01 14 04 27 18 05 06 09
14 05 04 27 27 27 27 27 27 27 27 27 27 27 27 27 27 27 27 27 27 27 27 27
```

It is also pretty easy to get an idea how well this rather random image will compress if we substitute letters in the alphabet for the numeric pixel values:

```
so after breakfast they
went round to see piglet
and pooh explained as t
hey went that piglet was
a very small animal who
did not like bouncing a
nd asked tigger not to b
e too bouncy just at fir
st and tigger who had be
en hiding behind trees s
aid that a tigger was on
ly bouncy before breakfa
st and that as soon as t
hey had had breakfast th
ey became quiet and refi
ned
```

So if Lempel and Ziv can handle quite random images and handles ones with long runs of pixels the same quite well, it seems that it would be a good candidate for compressing **computer graphics** images. A computer graphics image is one that has been generated by some computer system that creates pictures rather than **real world** images where the colour of each pixel is almost always slightly different from its neighbour and so may need a different technique to compress it.

3.5 LZW

Between 1977 and 1984, Lempel and Ziv produced a number of variations on the basic algorithm. There is a choice as to whether you check the whole of the document so far for matches or limit it to some window. By limiting you can reduce the size of the Code table required. If you are transmitting a pointer into a Code Table, it is feasible that at the beginning you can use a smaller number of bits than later on.

In 1984, Terry Welch of Sperry Research Center proposed a variant, later called LZW. Welch was interested in compressing large volumes of data stored on discs in commercial environments. The information is assumed to be a mixture of text and numerical data intermixed. He was particularly conscious that in numeric data you often got stock numbers like 000474 and there was therefore opportunities for compression that never appeared in purely textual information. He was also interested in positional redundancy. The same character might appear in the same position of a record over many records (this is equivalent to a vertical line in an image where the same pixel in a set of scan lines is a different colour). Welch was interested in the work of Lempel and Ziv but was unimpressed by the poor performance early on when the system was learning what the basic set of symbols was. Also, they did not adapt when the information changed its basic characteristics.

LZW compression is quite similar to what has been described so far. The main points are:

- The Code Table was a fixed size, normally 4096, requiring 12 bits to define an entry
- The algorithm is a greedy one as we have described. The input tokens are read until the longest possible string is encountered in the Code Table. A new entry is made consisting of the Prefix plus the next character when a match is not made
- The table is initialized with the single symbol alphabet that is likely to be encountered
- The table is made fixed in width by not storing the characters in the Prefix but the position of the Prefix in the Code Table. So all new entries consist of two values, a pointer to the Prefix's Code Table entry and the character added to make this new entry
- Welch recognised the problem at the decode stage where the new symbol was not yet in the Code Table and described how to handle this
- The Code Table was a hash table to keep the lookup speed down
- The algorithm was aimed at hardware compression as part of a disc subsystem on a large mainframe

The Welch paper also pointed out that the Code Table could start without the basic symbols already installed and that the length of pointers to the Code Table could be smaller early on. His view was that by starting with 8 bits per pointer early on rather than 12 would save about 7% but would increase the complexity significantly. Welch had the paper published while he was working for Digital Equipment Corporation but the work was done in 1983 when he worked at the Sperry Research Center.

In December 1985, Unisys was awarded the patent **US4558302: High speed data compression and decompression apparatus and method** that effectively patents the LZW algorithm. Unisys claim that the patent was put in a drawer and forgotten.

4. GIF: the Graphics Interchange Format

4.1 Introduction

In 1987, CompuServe developed a format for the interchange of computer graphics images called the Graphics Interchange Format (GIF). CompuServe was unaware of the Unisys patent and so used LZW compression as the basis for the GIF image format. It could have used alternative compression technologies such as run-length encoding. GIF became the standard format for interchanging computer graphics images on the Web and, even today, there are probably more GIF images around than any other format. It was not until 1993 that Unisys notified CompuServe that they had a patent on the algorithm and it took CompuServe until December 1994 to come to a licence agreement with Unisys. This allowed software developers associated with CompuServe to use the algorithm and Unisys agreed not to pursue royalty claims against them for past use. CompuServe developers had to pay a royalty of 15 cents to Unisys for every copy of the software sold.

In consequence, building a GIF decoder probably contravenes the Unisys Patent but you are unlikely to be sued. Building a GIF encoder definitely does contravene the Unisys patent and so would require the organisation involved to pay a royalty fee.

4.2 Format

Figure 4.1 gives the overall structure of the GIF file that is transmitted.

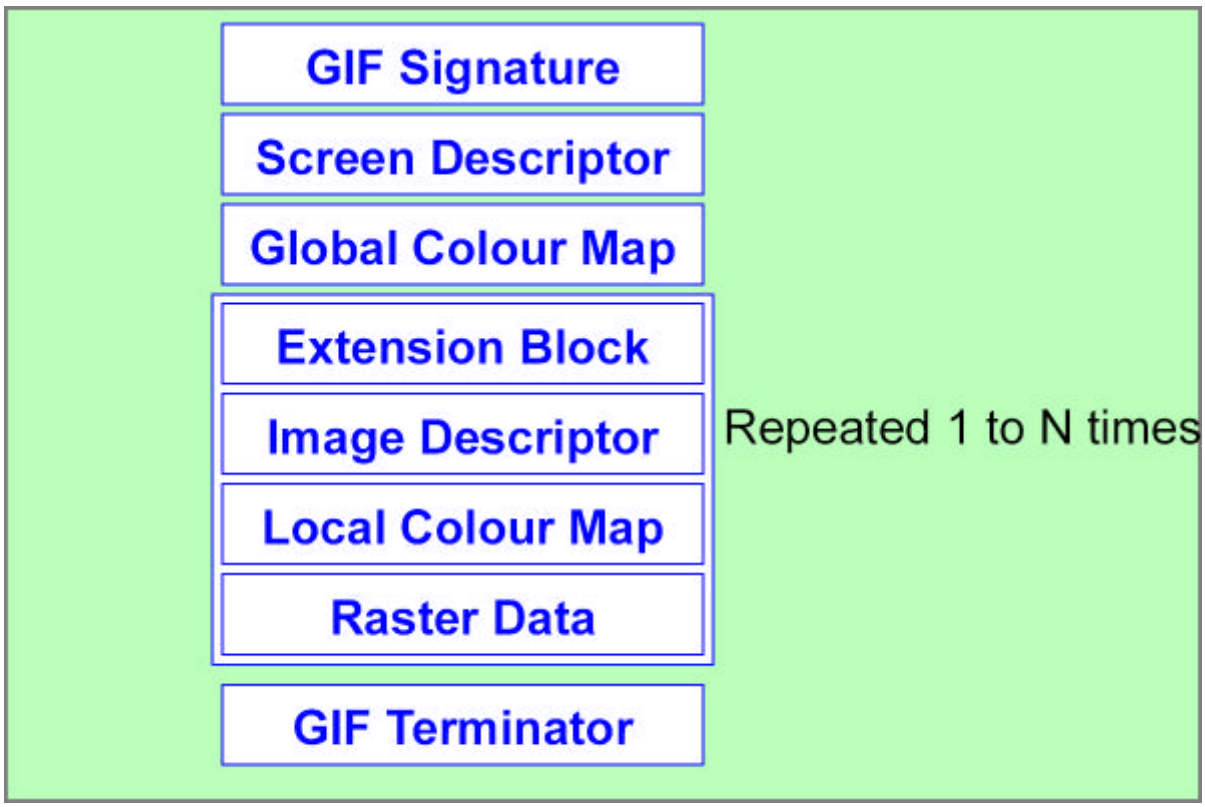


Figure 4.1: Overall GIF File Format

The main parts are:

- **GIF Signature:** this is useful in establishing that the file is in deed a GIF file
- **Screen Descriptor:** this gives some overall parameters to be used in rendering all the GIFs in the file
- **Global Colour Map:** it is usual to define a single Colour Table for all the images in the GIF file
- **Extension Block:** some proprietary extensions are allowed but they can be ignored
- **Image Descriptor:** describes this GIF picture which may be the only one
- **Local Colour Map:** defines the colours to be used for this image if it is different from the global definition of entries
- **Raster Data:** the LZW compressed image
- **GIF Terminator:** allows you to know when you have finished!

Figure 4.2 shows the GIF Signature. It starts with six bytes defining either the ASCII characters GIF87a or GIF89a defined as a set of ASCII characters, one character per byte. This is followed by two bytes (with the first being the least significant) defining the **width** in pixels of the GIF and another two bytes that define the **height**. These will be used to establish the overall positioning on the screen of the output. The main point of interest is that the Lowest Significant Byte arrives first and the highest second.

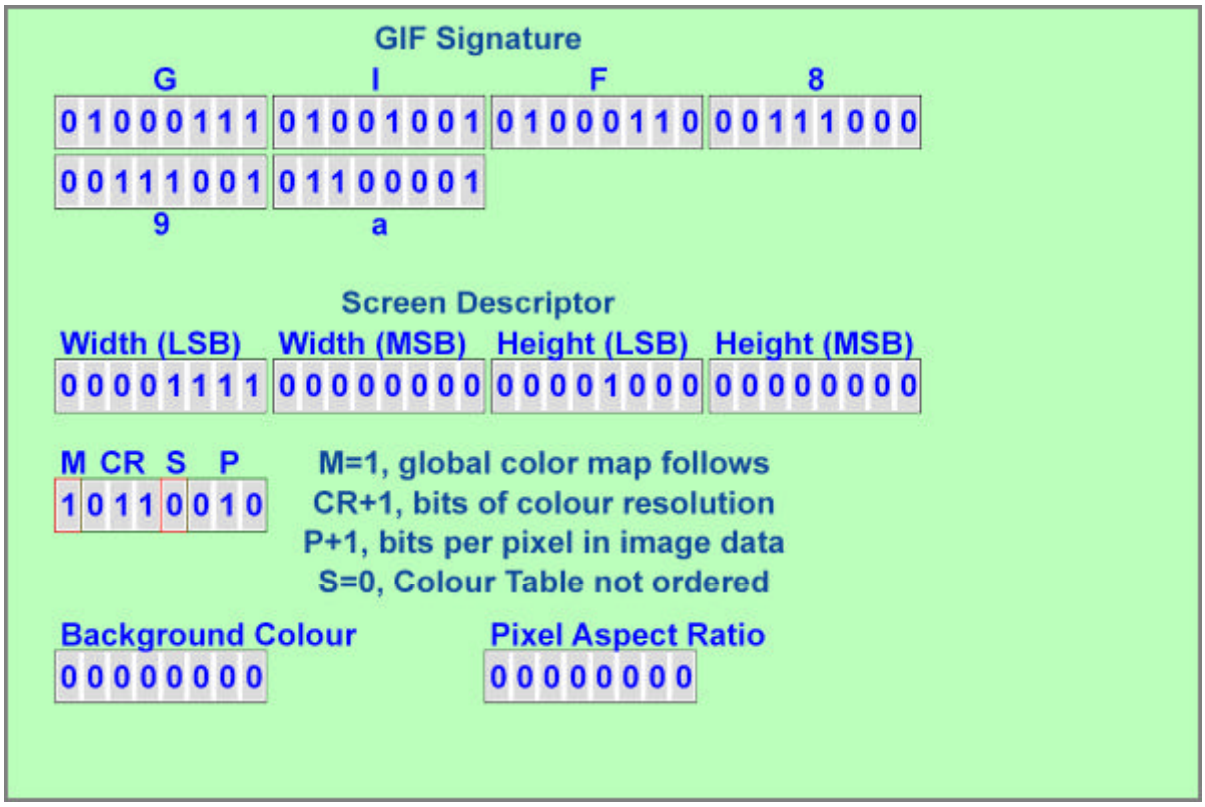


Figure 4.2: GIF Signature and Screen Descriptor

Following the width and height is a set of flag bits in a byte that give (see Figure 4.2):

- **P:** the three bottom bits define the number of bits per pixel (+1) that will be used when we encode the GIF
- **S:** a flag that is rarely used to signify that the colour table is ordered and can be interpolated.
- **CR:** defines the bits per pixel in the image. A value of 2 would define 3 bits per pixel initially
- **M:** if set to 1, it indicates that a global colour table is present.

The last two bytes of the Screen Descriptor define the background colour (relative to the Colour Table) to use and the pixel aspect ratio, which is usually set to 0 to indicate a square pixel.

Following the Screen Descriptor, it is usual to include a Global Colour table that applies to all the GIF images in the file. It may be impossible to change the Colour Table on the fly so having a single Colour Table for all the images is sensible. The table consists of a set of entries (2**(P+1)) of Colour Table entries each consisting of three bytes for the R, G, B values as shown in Figure 4.3. This shows a 4-entry Colour Table defining White, Red, Green and Blue as colours occupying positions 0 to 3. The number of pixels needed to define pixel colours is at least 3 bits.

After the Global Colour Map comes any Extension Blocks that start with a byte containing the hexadecimal value 21, followed by a byte indicating its type. this is followed by a byte indicating the length and a zero byte terminates the Extension Block. This can be ignored.

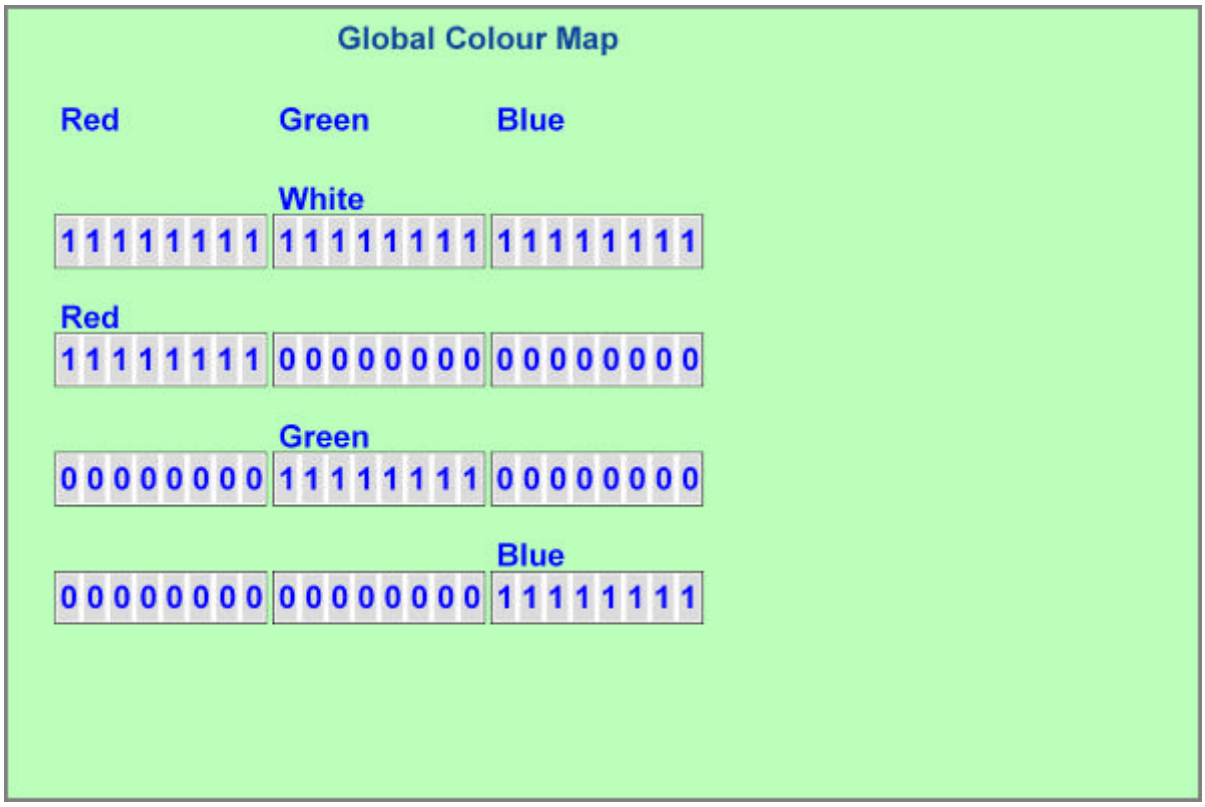


Figure 4.3: GIF Global Colour Map

Each image consists of an **image descriptor** followed by the data that defines the image. The Image Descriptor (see Figure 4.4). The Image Descriptor starts with its unique heading and follows this with the position that this image should take relative to the overall GIF image. This is useful when the GIF is really a set of GIFs making up the complete image. The **left** and **top** entries define the origin where the GIF is displayed in the window and the **width** and **height** define the size of the image. The Image Descriptor concludes with a set of flags that define:

- **M:** if set to zero, the local Colour Map replaces the Global Colour Map. In practice, it rarely happens.
- **I:** if set to 1 it indicates that the image is interlaced and that it would be wrong to allow people unfamiliar with GIF to use it.
- **PX:** defines the bits per pixel when the local Colour Map is used.

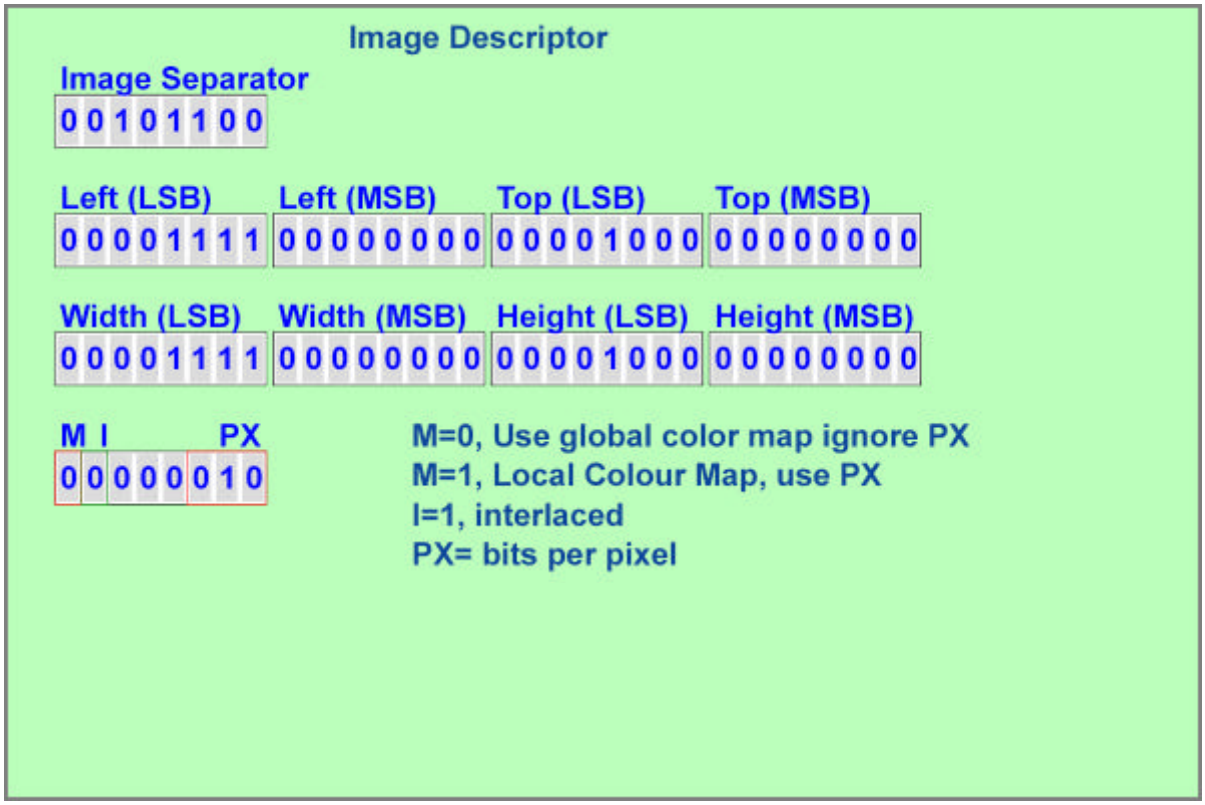


Figure 4.4: GIF Image Descriptor

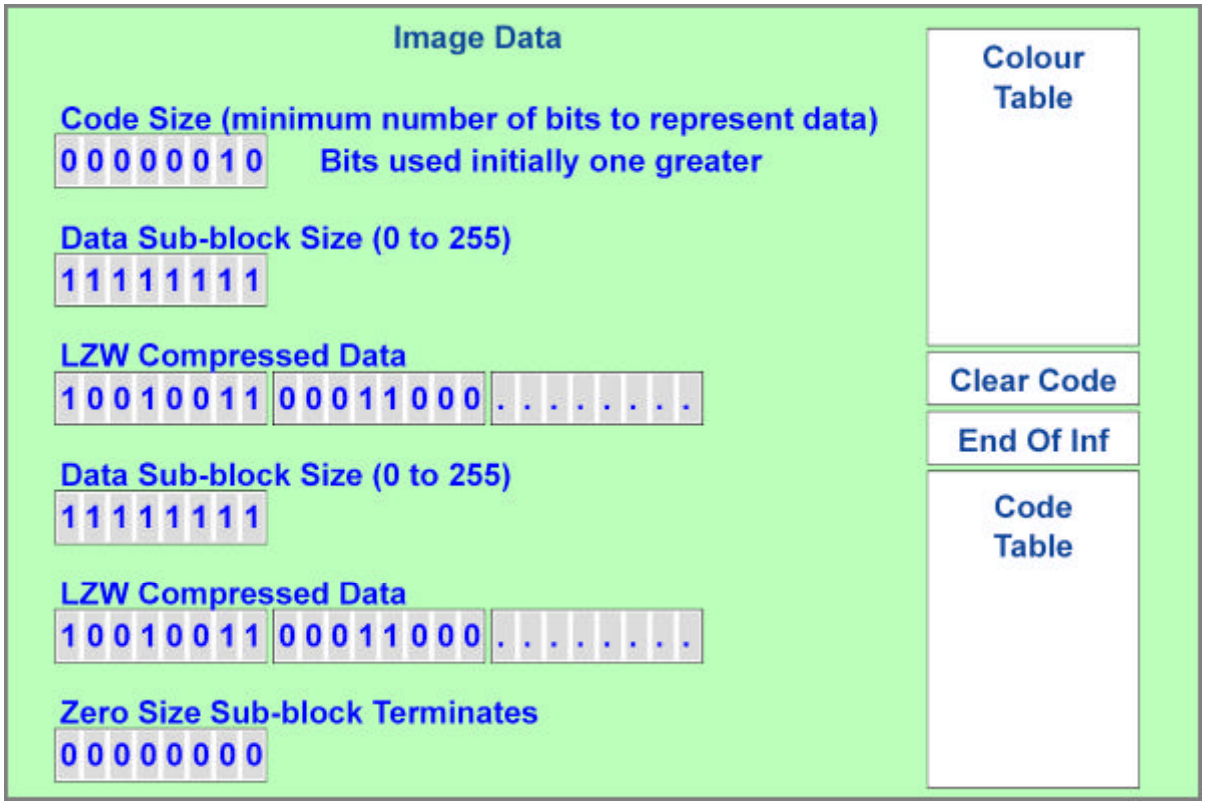


Figure 4.5: GIF Image Data

Figure 4.5 shows the format of the Image Data that defines the GIF image. The first byte gives the number of bits needed to represent the data and this is usually one more than is specified by the **CodeSize**. The value 2^{CodeSize} is called the **ClearCode** entry. Values smaller than the ClearCode specify entries in the Colour Table. Values greater than ClearCode+1 represent entries in the Code Table. The ClearCode+1 entry is called the **End of Information** entry. If it appears in the data stream, it indicates the end of the image data.

In Figure 4.5, the value of 2 for the Code Size indicates that the image data starts with 3 bits per pixel. The ClearCode value is 4 and the End of Information value is 5. If the ClearCode appears in the data stream, it resets all the compression/decompression parameters and tables to the start up state. It is usual to have the ClearCode as the first entry in the data stream. The first entry in the Code Table for a Code Size of 2 is entry 6.

The image consists of a set of LZW compressed blocks each preceded by their length which is not more than 256 bytes. There can be any number of these finishing with a null block.

The LZW Compressed Data Stream is shown in Figure 4.6. The bytes are read with the **Least Significant Bit first**. In Figure 4.6, the bytes are shown from right to left. The alternative is to write them left to right but transposing all the bits in each byte. If the Code Size is 2, the bits will be read 3 at a time initially. If the Code Size was 8, 9 bits would be read at a time initially. With a Code Size of 2, the maximum Code Table entry possible is 7. When entry 8 is required in the Data Stream, the number of bits per output code is increased to 4. when entry 16 is required, the number of bits is increased to 5 and so on. Figure 4.6 shows the arrangement.

Finally, the GIF decoder terminates with a byte containing the hexadecimal value 3b, the GIF Terminator.

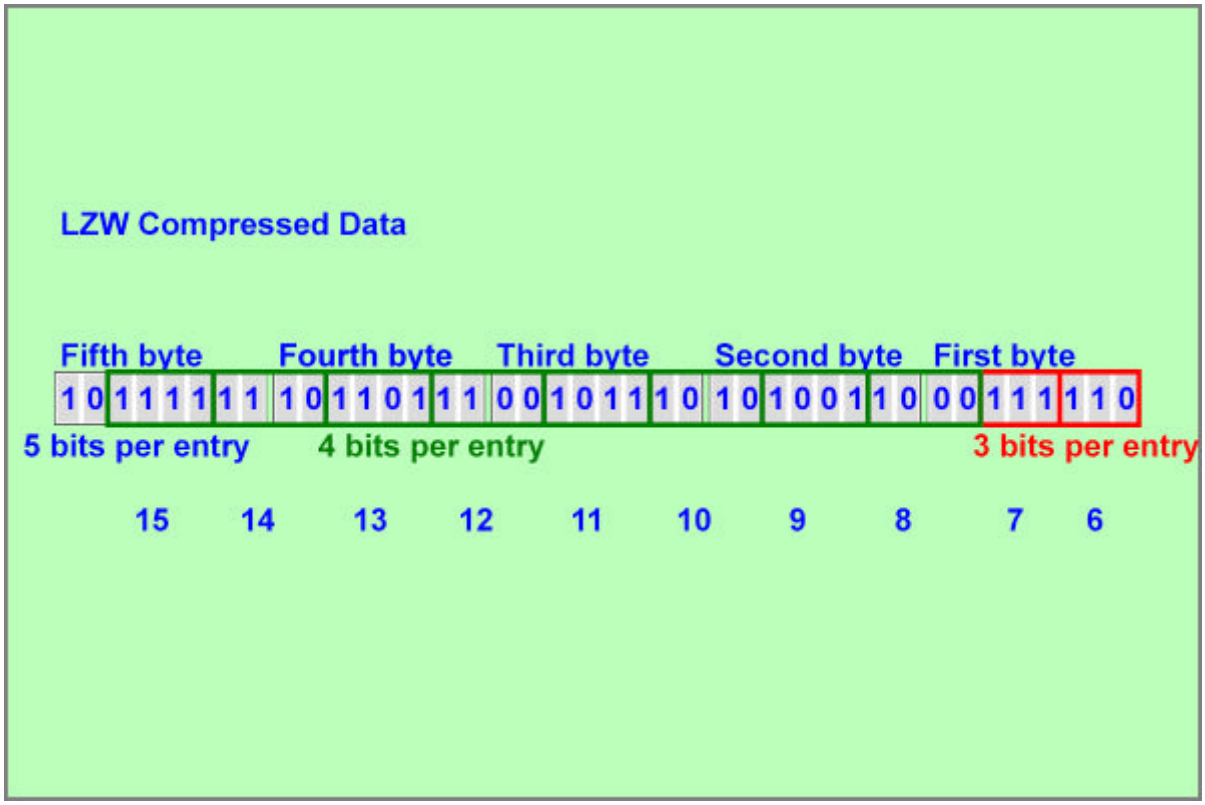


Figure 4.6: LZW Compressed Data

4.3 An Example

Here is a typical small GIF image:

```
47 49 46 38 39 61 27 00 19 00 91 ff 00 ff ff ff 00 00 00
00 ff 00 00 00 21 f9 04 01 00 00 00 00 2c 00 00 00 00 27 00
19 00 00 02 8c 04 62 29 cb 72 01 21 68 34 21 e3 5e c2 ed 45
44 2d d6 c5 8d 5b f7 49 a1 21 9d 1d bb 4d 4e 0a 32 56 1c da
5a 4c cb a6 92 7b ed 58 a9 4d 0b 18 ac 10 89 9f a5 2b 19 bc
f5 48 9c 68 ec 6a 2c 62 b7 1e 9a f7 0b 0e 8b c7 e4 b2 f9 8c
4e ab d7 6c 33 17 db 7b 5f 93 bc ae 84 04 a5 67 99 11 a7 2c
2f b2 37 d2 f4 f3 a7 e7 27 12 27 78 08 53 35 53 14 78 e3 98
f1 b3 d2 23 34 a4 93 69 09 89 b9 13 88 17 75 b9 62 54 09 65
57 93 b3 57 00 00 3b
```

We can identify the component parts more easily if we adjust where the page breaks come:

```
47 49 46 38 39 61  GIF Signature
27 00 19 00  raster width and height, Least Significant Byte first
91  Code value M CR S P (10010001) M=1, CR=1, S=0, P=1
ff 00  Background colour and pixel ratio
ff ff ff  Colour table, first entry White
ff 00 00  Red
00 00 ff  Blue
00 00 00  Black
21 f9  Extension Block
04 01 00 00 00  Four Entries
00  Terminate Extension Block
2c  Image Separator
00 00 00 00  Left and Top
27 00 19 00  Width and Height
00  M=0, I=0, PX=0 (Non-interlaced image using the Global Colour Table)
02  Code Size is 2
8c  140 bytes of image data to follow
04 62 29 cb 72 01 21 68 34 21 e3 5e c2 ed 45 44 2d d6 c5 8d
5b f7 49 a1 21 9d 1d bb 4d 4e 0a 32 56 1c da 5a 4c cb a6 92
7b ed 58 a9 4d 0b 18 ac 10 89 9f a5 2b 19 bc f5 48 9c 68 ec
6a 2c 62 b7 1e 9a f7 0b 0e 8b c7 e4 b2 f9 8c 4e ab d7 6c 33
17 db 7b 5f 93 bc ae 84 04 a5 67 99 11 a7 2c 2f b2 37 d2 f4
f3 a7 e7 27 12 27 78 08 53 35 53 14 78 e3 98 f1 b3 d2 23 34
a4 93 69 09 89 b9 13 88 17 75 b9 62 54 09 65 57 93 b3 57 00
00  Null size next Code Block
3bGIF Terminator
```

This indicates that the image following contains the colours white, red, blue and black. That it is 39 pixels wide (hex 27) and 25 pixels high (hex 19). The image is not interlaced which means that the image rows appear in the correct order. The number of bits making up the data stream initially is 3. We now have the problem of decoding the data stream. The first few bytes are:

```
04 62 29 cb 72 01 21 68 in hexadecimal
00000100 01100010 00101001 11001011 01110010 00000001 00100001 01101000 in
binary
00100000 01000110 10010100 11010011 01001110 10000000 10000100 00010110
with least significant bit first
001 000 000 100
0110 1001 0100 1101 0011 0100 1110 1000
00001 00001 00000 10110 Splitting correctly as we build the Code Table
4 0 0 1 6 9 2 11 12 2 7 1 16 16 0 13 converting to decimal, least significant bit first
```

To make the decision when to change the bit size per entry depends on when the last entry in the Code Table has reached the maximum value. Thus it is necessary to build the Code Table as you decode the GIF.

The complete GIF is the set of entries:

```
  4   0   0   1

  6   9   2  11  12   2   7   1

 16  16   0  13  20   9   8   6  14  15   9  24  13  15  17   8

 20  11  22  23  28  35  27  29  31  18  33   6  18  39  29  44

 27  19  14  41  32  12  22  49  33  54  26  49  52  50  38  10

 57  47  59  44  41  27  45  64  65  21  68  68  31  75  46  73

 65  55  61  36  28  81  49  87  70  69  88  91  30  52  94  95

 96  97  98  99 100 101 102 103 104 105 106 107 108 102  92  88

 61 111  87  73  60  93  18  36  80 116  89  76  17  78  50 121

 34 123  35  77  63 127 122 126  34 113 130 135  48  85  51  69

129  55 142  25  63  43  61  66  67  58 153 150 144 152  59 129

120  81 151  43  70 149  80 118  53  57 123   5
```

The first entry is the ClearCode value 4 that resets all the table entries to their initial values. The four Colour Table entries are:

- 0: White (W)
- 1: Red (R)
- 2: Blue (B)
- 3: Black (K)

The GIF Encoder and Decoder are very similar to the LZW Encoder and Decoder defined in Section 3.4. The major difference is where the Prefix and Suffix are stored. Looking at the Code Table generated in Section 3.4, it can be seen that if you output the Code Table entries less the last character each time starting at entry 27, you create the original string. Also the last character of each entry is the first character of the next entry. Using this information, the GIF Decoder is as follows:

```
C=ClearCode+2;
repeat

Get NextCode;

if NextCode = ClearCode initialize;

if NextCode < ClearCode add NextCode as new entry to Code Table

if NextCode > ClearCode+1

   {Create new entry consisting of Code Table [NextCode]

    plus first entry of Code Table [NextCode+1]

   }

Output Entry C and Increment C;
end repeat
```

The Code Table produced is:

6	W	48	WWR	90	RRRRRW	131	WWWRR
7	W	49	WWWW	91	WWWWWWW	132	RRRRW
8	R	50	WB	92	WWWWWWWW	133	WWWRRWW
9	WW	51	BW	93	WRRR	134	WBBB
10	WWW	52	RRRRR	94	RRRRRR	135	BBBBBBBBBWW
11	B	53	RWB	95	RRRRRRR	136	WWRRWWW
12	BB	54	BBB	96	RRRRRRRR	137	BBW
13	BBB	55	WWWR	97	RRRRRRRRR	138	RRRRRRWW
14	B	56	WWWWW	98	RRRRRRRRRR	139	WWWWRRWW
15	WR	57	BBBBBB	99	RRRRRRRRRRR	140	BBBBBBBBBWWW
16	R	58	BBBW	100	RRRRRRRRRRRR	141	WWRW
17	RR	59	WRRW	101	RRRRRRRRRRRRR	142	WWBBB
18	RR	60	WWWWW	102	RRRRRRRRRRRRRR	143	BWR
19	W	61	RRRRRR	103	RRRRRRRRRRRRRRR	144	RRRRRW
20	BBBB	62	WBB	104	RRRRRRRRRRRRRRRR	145	BBWW
21	BBBBB	63	WWWRRW	105	RRRRRRRRRRRRRRRRR	146	WWWRW
22	WWW	64	WWWB	106	RRRRRRRRRRRRRRRRRR	147	WWBBBB
23	RW	65	BBBBBBB	107	RRRRRRRRRRRRRRRRRRR	148	BWW
24	WW	66	BBBBWW	108	RRRRRRRRRRRRRRRRRRRR	149	WWWRRWW
25	BW	67	WRRWW	109	RRRRRRRRRRRRRRRRRRRRR	150	BBBBBBW
26	WRR	68	WWR	110	RRRRRRRRRRRRRR	151	RRRRRRW
27	WWW	69	RRRRR	111	WWWWWWWWW	152	BBBBWWW
28	WWB	70	WWWW	112	WWWWWWW	153	WRRWWW
29	BBBB	71	RRWW	113	RRRRRRW	154	BBBWW
30	WRR	72	WWWBB	114	WWWWWWWWW	155	WRRWWWB
31	RRR	73	BBBBBBBB	115	WWWWWW	156	BBBBBBWR
32	RW	74	BBBBBW	116	BBBBBBBBB	157	RRRRRWB
33	BBBBB	75	WWRR	117	WWWWWR	158	BBBBWWWW
34	BB	76	WWRR	118	WRRRR	159	WRRWW
35	WWWR	77	RRRR	119	RRW	160	BBWW
36	RWW	78	WWRRW	120	RWWW	161	RWWWB
37	WWBB	79	WWWWBB	121	BBBBBBBBBB	162	BBBBBBBBW
38	WWWRR	80	BBBBBBBBB	122	BBBBBBBBBW	163	RRRRRRWB
39	WWWW	81	BBBBBBBB	123	WWWWRR	164	BBBBBBW
40	BBBBW	82	WWWRW	124	WWRRR	165	WWWWR
41	RRRR	83	RRRRRRW	125	RRR	166	WWWRRWWB
42	RRW	84	RWWW	126	WWRRWW	167	BBBBBBBBBB
43	BBBBBB	85	WWBB	127	WBB	168	WRRRRR
44	WW	86	BBBBBBBBW	128	BBBBBBBBBBB	169	RWBB
45	RRW	87	WWWWW	129	BBW	170	BBBBBBB
46	WWWWB	88	WWWWWW	130	WWWWRRW		
47	BBBBW	89	WWWWR	131	WWWRR		

If we go through the Code Table, outputting the entries in turn and placing them in rows of 39 entries:

```
WWRWWWWWBBBBBBBWRRRRRRWBBBBBBBBBWWWRWWW
BWWRRWWWWWBBBBBWRRRRRRWBBBBBBBWWWRRWWWW
BBWWWRRWWWWBBBBWRRRRRRWBBBBBBWWRRWWWWWB
BBBBWWWRWWWWWBBWRRRRRRWBBBBWWWRWWWWWBBB
BBBBBBWWRRWWWWWWRRRRRRWBBWWWRRWWWWBBBBB
BBBBBBBWWWRRWWWWRRRRRRWWWWRRWWWWWBBBBBB
BBBBBBBBBWWWRRWWRRRRRRWWRRWWWWWBBBBBBBB
BBBBBBBBBBWWWRWWRRRRRRWRWWWWWBBBBBBBBBB
WWWWWWWWWWWWWWWWRRRRRRWWWWWWWWWWWWWWWW
RRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRR
RRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRR
RRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRR
RRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRR
RRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRR
RRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRR
WWWWWWWWWWWWWWWWRRRRRRWWWWWWWWWWWWWWW
BBBBBBBBBWWWWWRWRRRRRRWRWWWBBBBBBBBBBB
BBBBBBBWWWWWRRWWRRRRRRWWRRWWWBBBBBBBBB
BBBBBWWWWWRRWWWWRRRRRRWWWWRRWWWBBBBBBB
BBBBWWWWRRWWWBBWRRRRRRWWWWWWRRWWBBBBBB
BBWWWWWRWWWBBBBWRRRRRRWBBWWWWWRWWWBBBB
WWWWWRRWWBBBBBBWRRRRRRWBBBBWWWWRRWWBBB
WWWRRWWWBBBBBBBWRRRRRRWBBBBBWWWWWRRWWB
WWRWWWBBBBBBBBBWRRRRRRWBBBBBBBWWWWWRWW
RRWWBBBBBBBBBBBWRRRRRRWBBBBBBBBBWWWWRR
```

Figure 4.7 shows a blown-up view and also a real size view of the image.

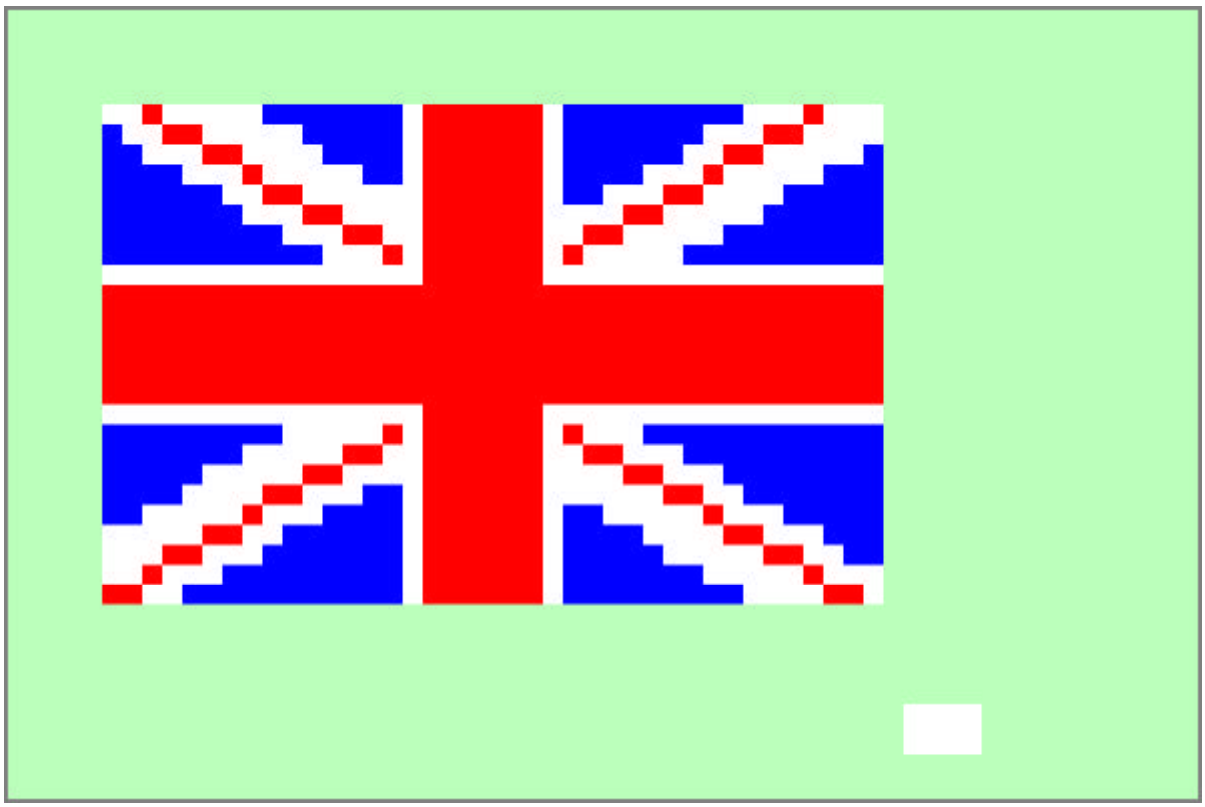


Figure 4.7: GIF Image

4.4 Interlacing

If the Interlace Bit was set to 1 in the GIF Image Descriptor, the GIF image is not transmitted row by row in order from top to bottom. Instead, rows of the image are output in a different order. Four passes are made with each pass filling in the rows not already transmitted in the set:

- Every eighth row is transmitted as pass 1
- Every fourth row is transmitted as pass 2
- Every second row is transmitted as pass 3
- The remaining rows are transmitted

Figure 4.8 shows the blown-up image as it would appear after each pass has been transmitted. The image is clearly identifiable after pass 3 and a guess to its form could be made after pass 2.

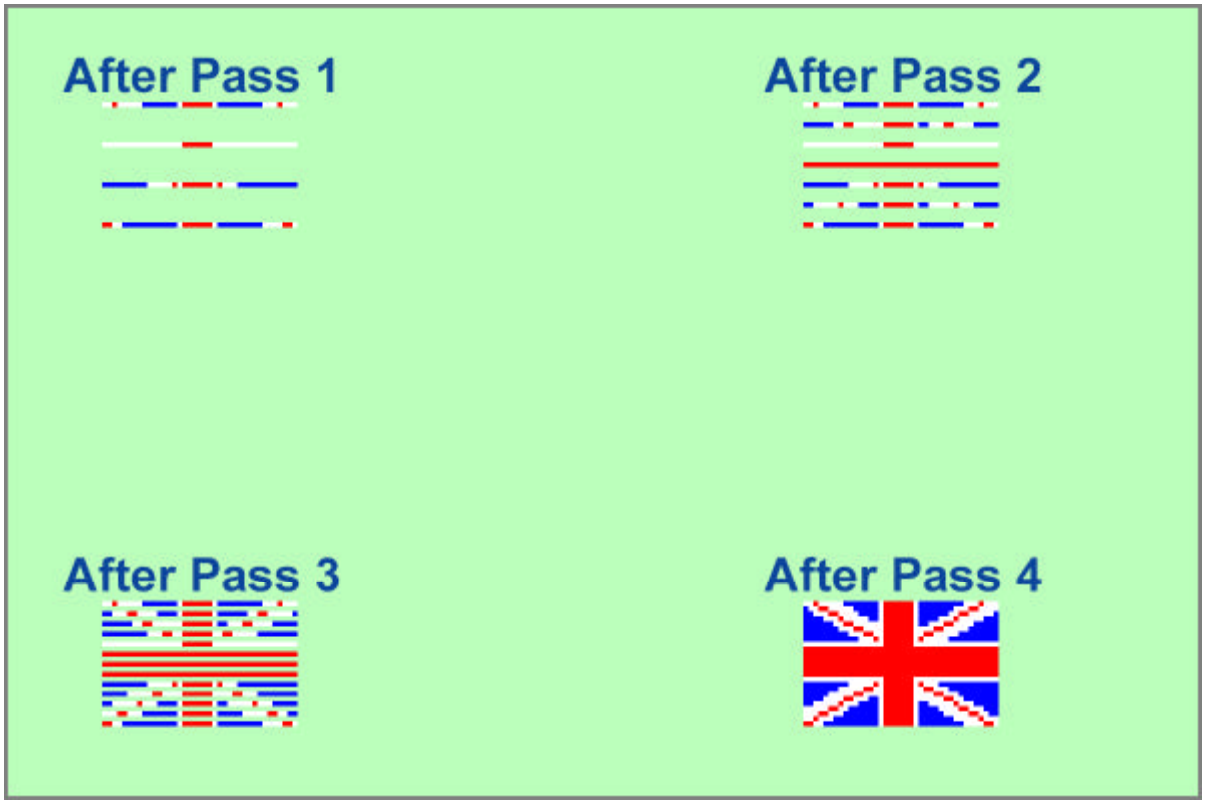


Figure 4.8: GIF Interlacing

4.5 Optimisation

Many GIF images on the Web are not as efficient as they might be. Some of the reasons for this inefficiency are:

- If images have been scanned in, the number of colours in the Colour Table may be larger than needed. Removing duplicate entries or merging two close colours into one. Make sure all the entries in the table are actually used.
- Frequently, images use a larger number of bits per Code Table Entry than is necessary
- Replace local colour tables and have a single Global Colour Table
- For images less than 10 kbytes, using interlacing probably is not sensible

5. PNG: Portable Network Graphics

5.1 Introduction

When the Unisys Patent hit the Web, there was a concerted effort to produce an alternative format that was free of patents. A Working Group was formed and the Portable Network Graphics (PNG) format was the result. PNG 1.0 has been available for over 5 years now and is widely accepted by browsers as an alternative to GIF. It is aimed at the transmission of computer graphics images, as was GIF, but it has a number of extensions including better transparency support, better colour support, better interlacing and is more efficient. Most of today's graphics design tools allow images to be stored in either GIF or PNG formats.

5.2 Concepts

PNG prepares an image for transmission through a number of stages:

1. The starting point is a **source image** to be encoded as a PNG datastream. Conceptually, the source image always has a colour or greyscale value for each pixel
2. A **PNG image** is created by transforming this image by some or all of **alpha separation**, **scaling**, **indexing** and **alpha indexing**. This results in five different types of image with different properties.

The generic starting point for PNG is the PNG pixel shown in Figure 5.1. Unlike GIF images, each PNG pixel can have its own RGB and transparency specified. Some images may have a limited set of colour values in which case they can be indexed as in GIF. Also, if it is a monochrome image, The RGB values can be replaced by a single greyscale value. Note also that each pixel can have its transparency specified as well (alpha value between 0 and a maximum value). Again, there are image types where transparency can be specified as either visible or invisible and no intermediate values.

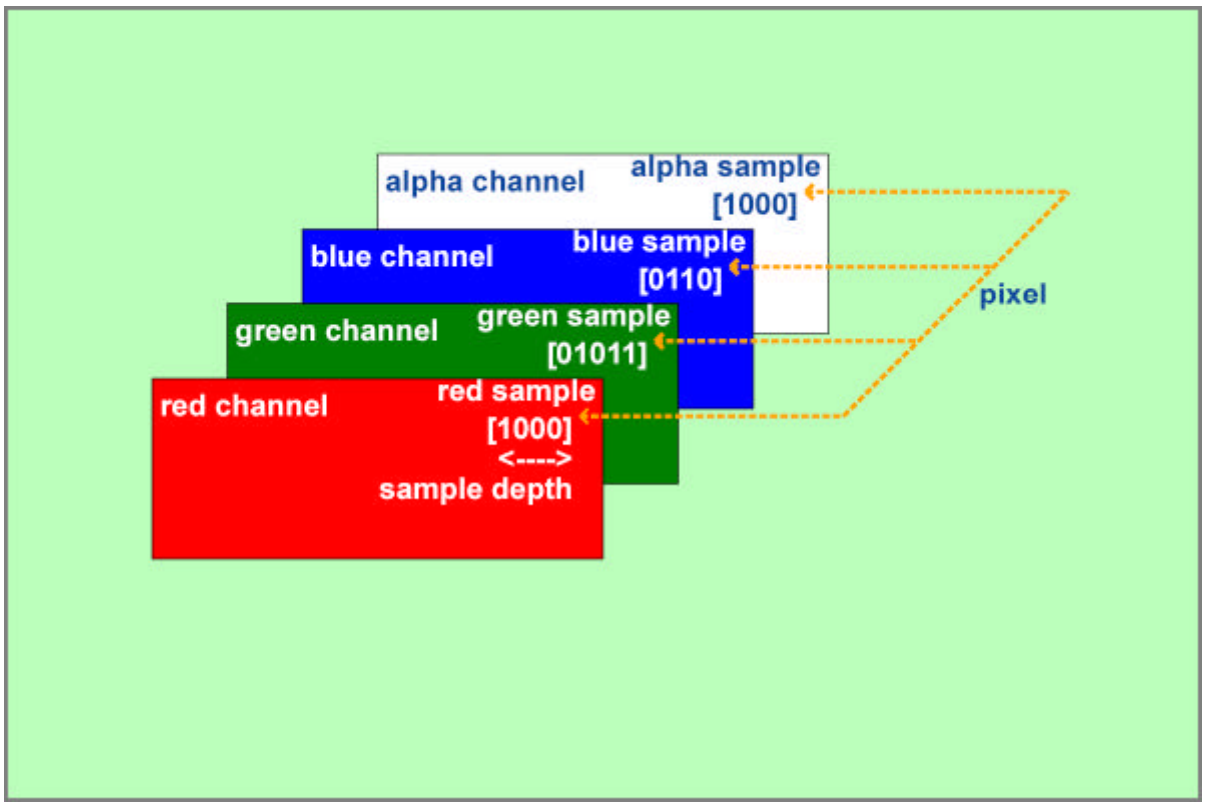


Figure 5.1: PNG Pixel

The aim is to convert all the images that need to be transformed into a PNG datastream in one of the following forms:

- Greyscale image, no alpha channel
- Truecolour image, no alpha channel
- Indexed-colour (a la GIF)
- Greyscale image with alpha channel
- Truecolour image with alpha channel

The different types are illustrated in Figure 5.2. On the left is the value that each pixel can have. Indexed colour is similar to the GIF format where each pixel has an index value that points into a Colour Table. In the figure, the pixel illustrated has a value of 3 which points to the entry 3 in the Colour table so that the RGB values of the pixel are 176, 208, 176 respectively, a light green. The alpha value is set to 255, the maximum value being used and so the pixel is fully opaque. In the other four cases, each pixel has the information about its colour and transparency stored with it and there is no Colour Table. The four cases come from whether the pixel is colour or a greyscale and whether it is opaque or has a transparency value. Clearly, this allows much finer control of the content of each pixel but may result in a much larger image in terms of storage. In consequence, PNG images spend a great deal more time trying to massage the image before compressing it to ensure that the image is compressed as much as possible.

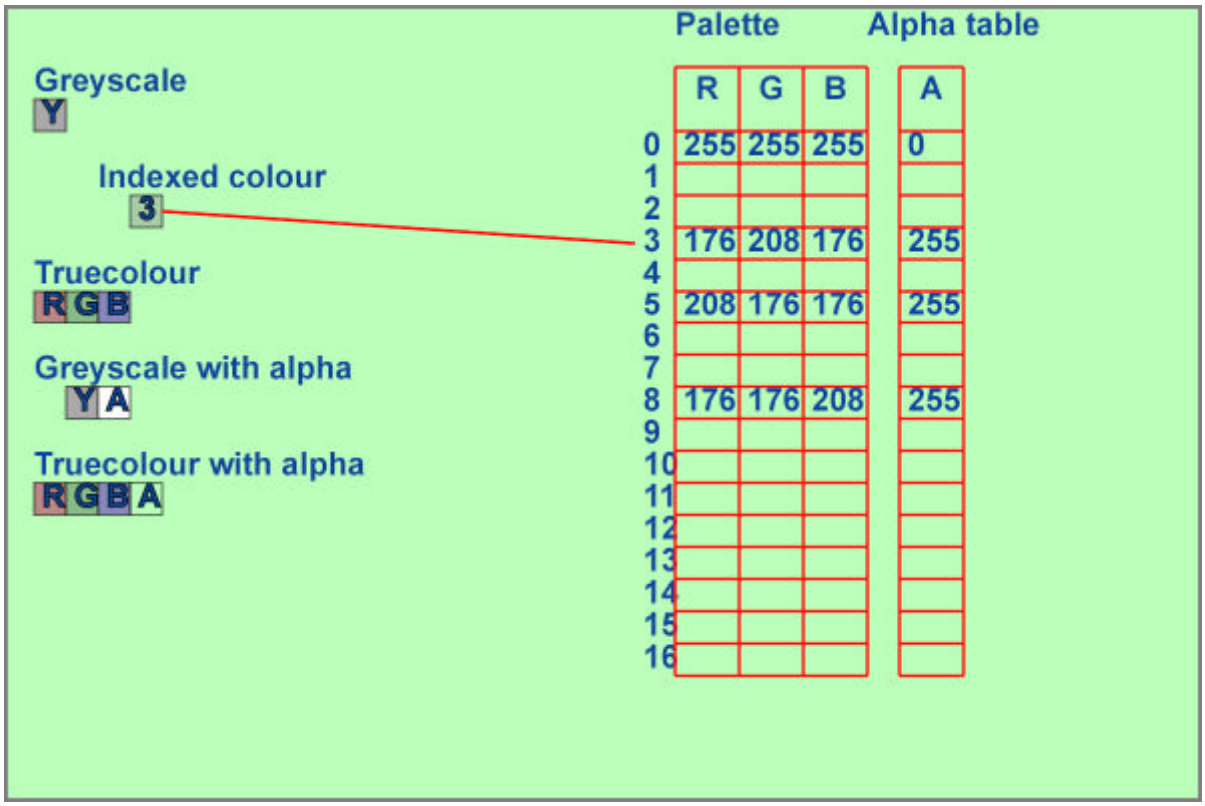


Figure 5.2: PNG Image Types

Rather than allow all possible values for the pixel's RGB and Alpha values, some limits are placed on the types of image that PNG is able to compress:

PNG image type	Colour type	Allowed image bit depth	Allowed sample depths
Greyscale, no alpha channel	0	1, 2, 4, 8, 16	1, 2, 4, 8, 16
Truecolour, no alpha channel	2	8, 16	8, 16
Indexed-colour	3	1, 2, 4, 8	8
Greyscale with alpha channel	4	8, 16	8, 16
Truecolour, with alpha channel	6	8, 16	8, 16

For example, you might start with a 3-bit per sample image which has its own R, G and B values for each pixel. It may have an alpha (transparency value) which is defined in 2 bits. What the table says is that the truecolour image (RGB values for each pixel) must be scaled to 8 bits per pixel unless all the pixels have the same R, G, and B values in which case it could be transformed into a greyscale image. In this case it could be scaled to 4 bits per pixel. Looking at the table above, it can be seen, for example, that you could have a 2-bit per pixel greyscale image or a 4-bit per pixel indexed image (where the Colour Table is still 8 bits for each of R, G and B). Thus there is quite a lot of work to be done in choosing the right format out of the many possibilities in PNG. The aim is to have much more control over the colour of pixels than is possible with GIF images and still be efficient in terms of storage requirements. If this can be done, it gives PNG the opportunity to handle real world images in a lossless way as well as handling the computer graphics images that GIF was aimed at. Thus PNG is also a possible competitor to JPEG for small real world images or ones that have a limited set of colour possibilities.

PNG calls the array of **samples** of a particular type (for example, green samples) a **channel**. Each horizontal row of pixels is called a **scanline**. Pixels are ordered from left-to-right within each scanline and scanlines are ordered top-to-bottom as with GIF images. The **sample depth** is the number of bits used to specify a sample in an image. The samples for a pixel may have different sample depths. An alpha value of zero defines a fully transparent pixel and a fully opaque pixel is defined by the maximum value.

5.3 Image Transformations

Figure 5.2 shows the transformations that can be applied to an image to create the PNG image to be encoded.

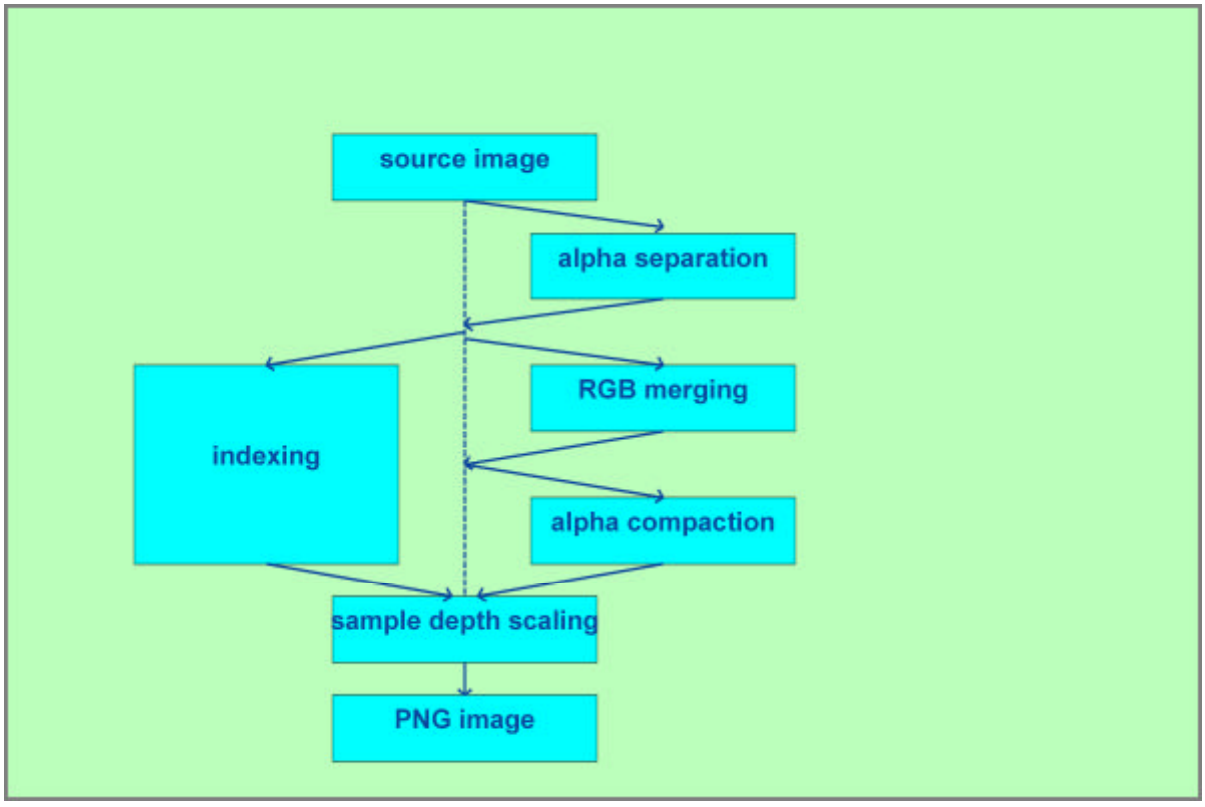


Figure 5.3: PNG Transformations

There are a number of situations where it does not make sense in having an alpha channel. The first is where there is no alpha channel initially and the assumption is that all the samples are opaque. If all the pixels have the same maximum value, the same is true. So the first operation determines whether or not there is a need for an alpha channel.

If the sample depth in the source image does not correspond to an allowed sample depth for the PNG image being encoded, the possible sample values in the reference image are linearly mapped into the next allowable range for the PNG image. Figure 5.4 shows, if the sample depth in the source image was 3 and the next allowable depth in the PNG image was 4, how the possible values are mapped into a PNG image with depth 4.

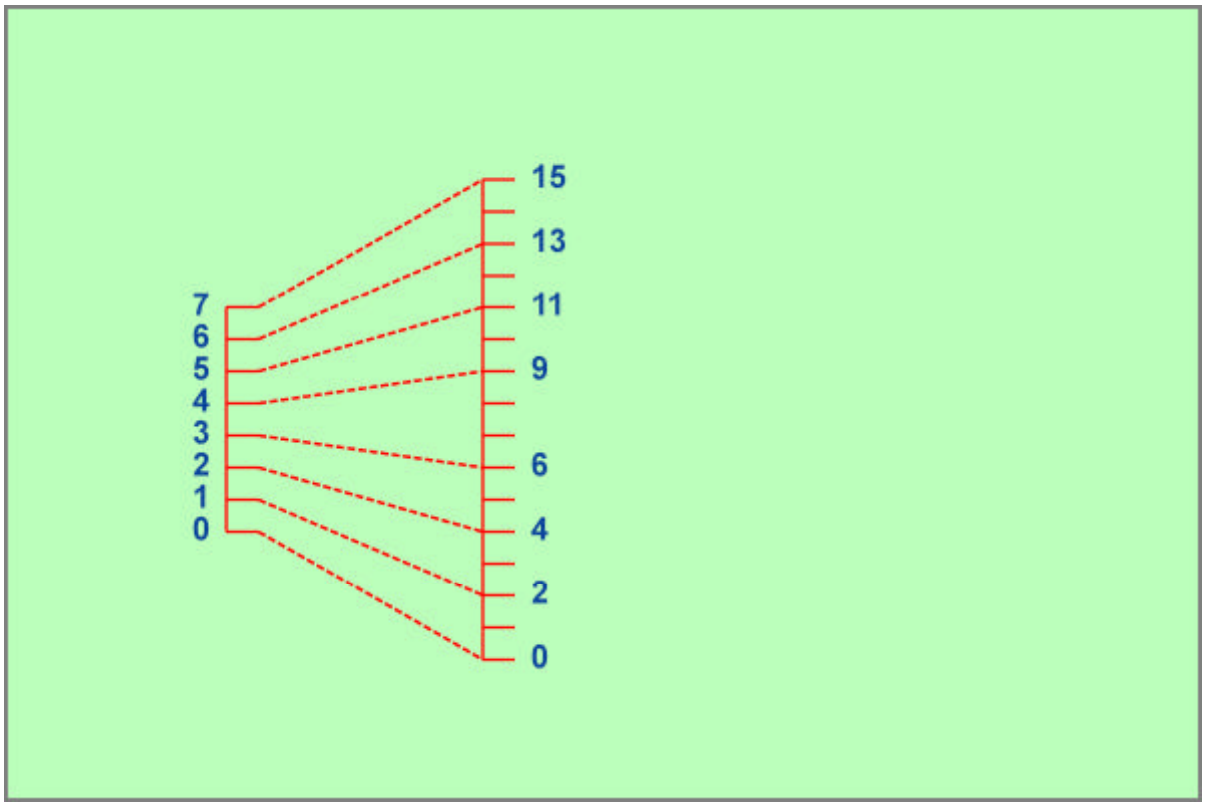


Figure 5.4: PNG Scaling

If the number of distinct pixels in the reference image is small, an indexed-colour representation might be more efficient. In this case, the array of pixels in the image is replaced by an array of indices of the same dimension. Each position in the indexed colour array contains a pointer into a table (see Figure 5.5), called the **palette**, and this has an associated alpha value. Depending on how many different alpha values there are, there are several alternative ways to store the alpha channel.

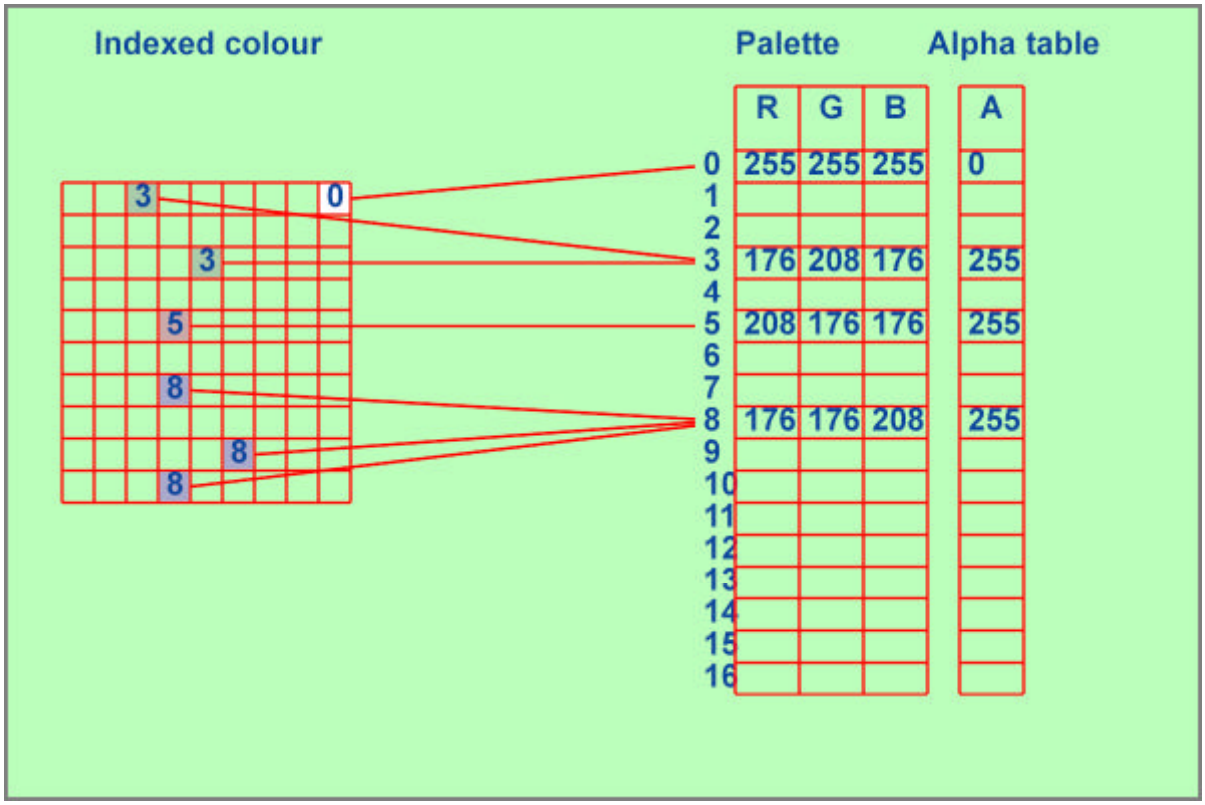


Figure 5.5: Colour Table

The final transformation stage is provided for images where the number of distinct alpha values is limited or where alpha values are either completely opaque or completely transparent.

For indexed-colour images, users are encouraged to rearrange the palette so that the table entries with the opaque alpha value are grouped at the end. In this case, the palette can be shortened to include only the non-opaque entries.

5.4 Encoding the PNG Image

Encoding a PNG image is shown in Figure 5.6.

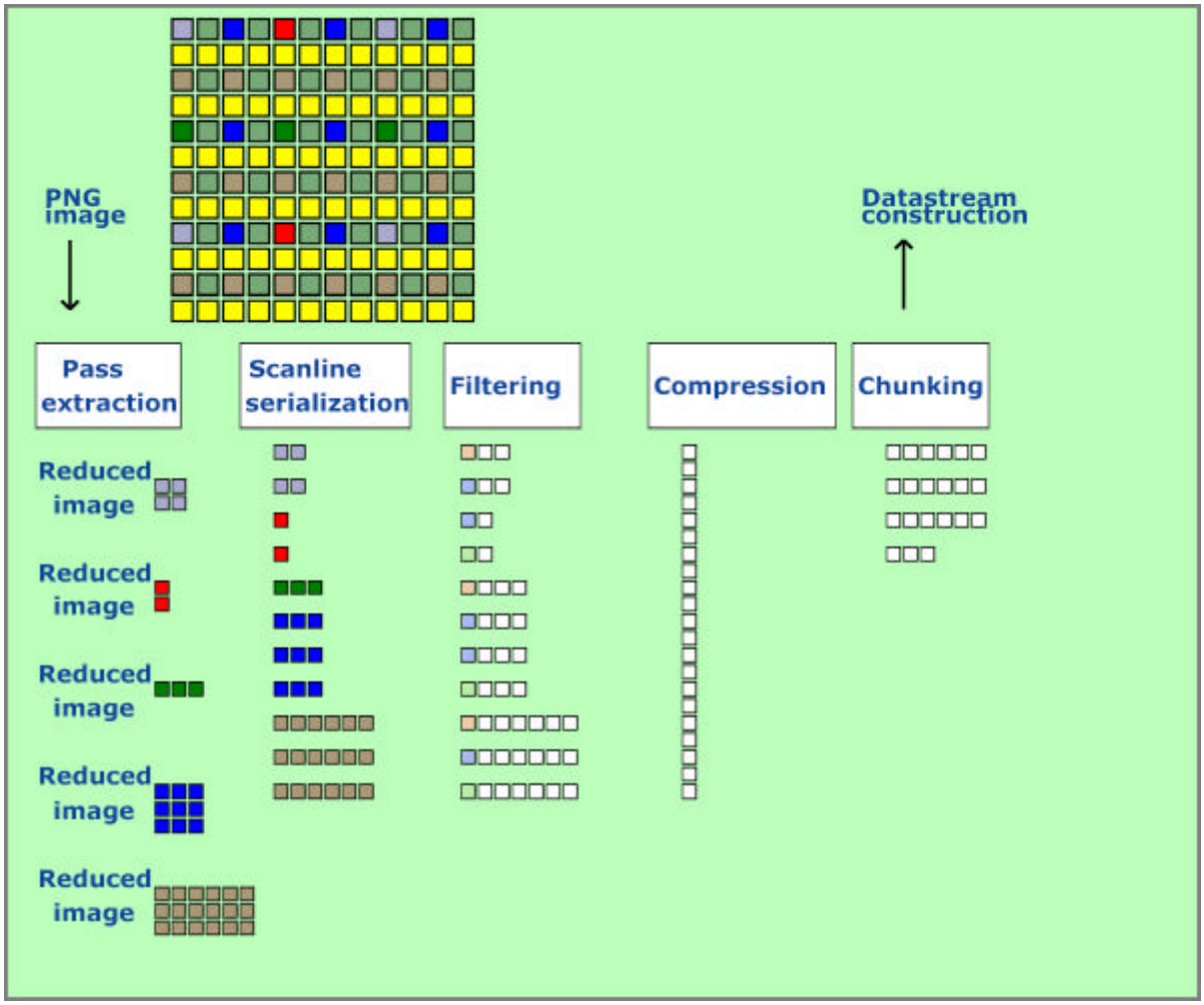


Figure 5.6: Encoding the PNG Image

The stages are:

1. **Pass abstraction:** to allow for progressive display; the PNG image is converted into one or more images. This is often called interlacing although the way PNG does it makes interlacing a less obvious word.
2. **Scanline abstraction:** the image is processed a scanline at a time.
3. **Filtering:** each scanline is transformed into a filtered scanline using one of the defined filter types to prepare the scanline for image compression.
4. **Compression:** occurs on all the filtered scanlines in the image.
5. **Chunking:** the compressed image is divided into conveniently sized chunks and an error detection code is added to each chunk.
6. **Datastream construction:** the chunks are inserted into the datastream.

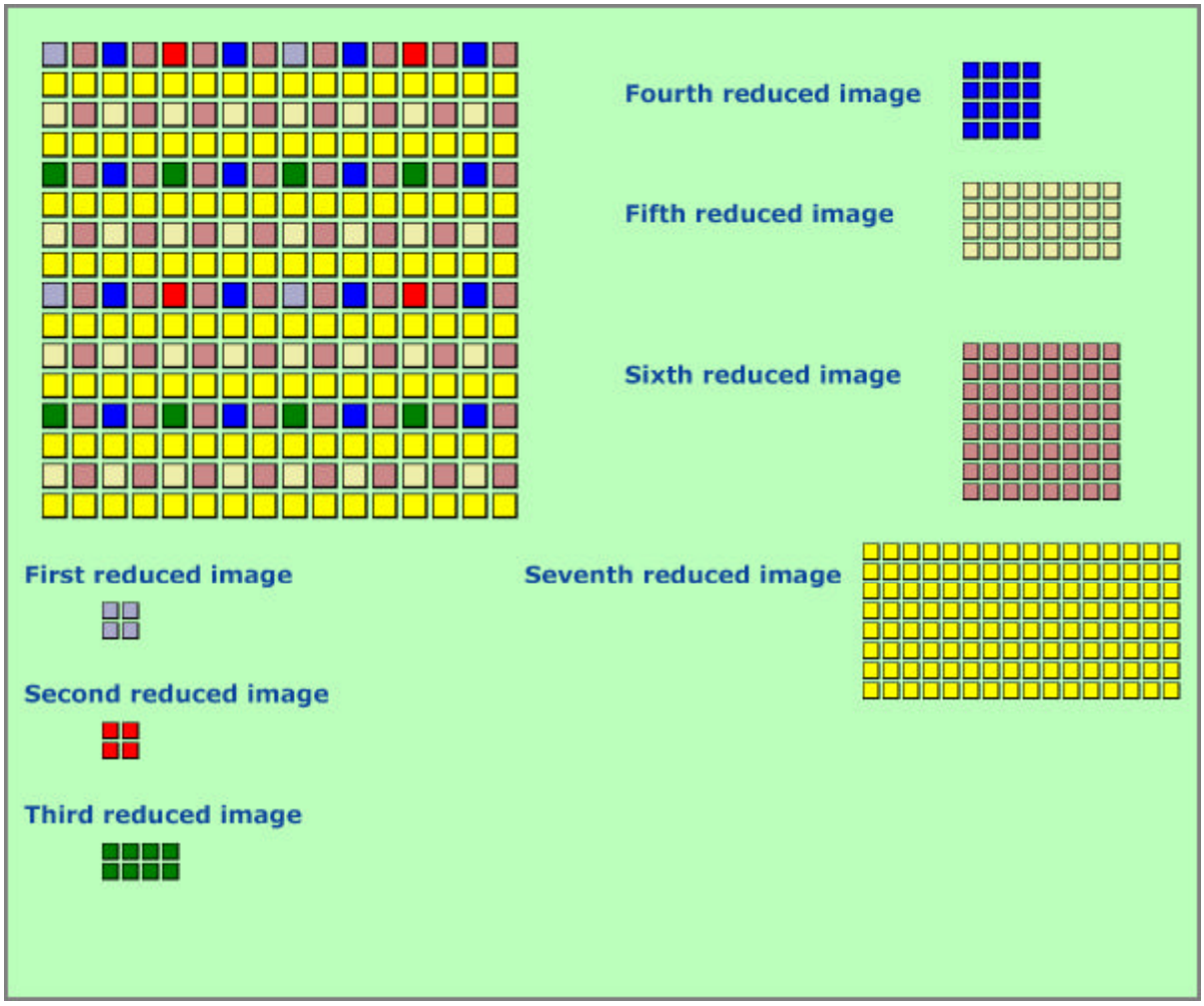


Figure 5.7: PNG Interlacing

Pass abstraction (see Figure 5.7) splits a PNG image into a sequence of smaller images where the first image defines a very coarse view of the original source image and subsequent images enhance this coarse view until the last image completes the original source image. The **set** of reduced images is also called an **interlaced PNG image**. The only interlace method other than scanline at a time is one that makes multiple passes over the image to produce a sequence of seven reduced images. The method is called Adam7.

It looks complicated but effectively the last pass has all the pixels in an 8 by 8 block. The sixth path misses out every other row; the fifth path misses out every other column; the fourth pass misses out every other remaining row; the third pass misses out every other remaining column and so on. In Figure 5.8, the pass where pixels appear first out of an 8 by 8 array are shown. The effect of the method are also shown. this should be compared with Figure 4.8. Because the PNG interlacing is based on areas, the understanding of the image versus pixels transmitted is much better for PNG than GIF.

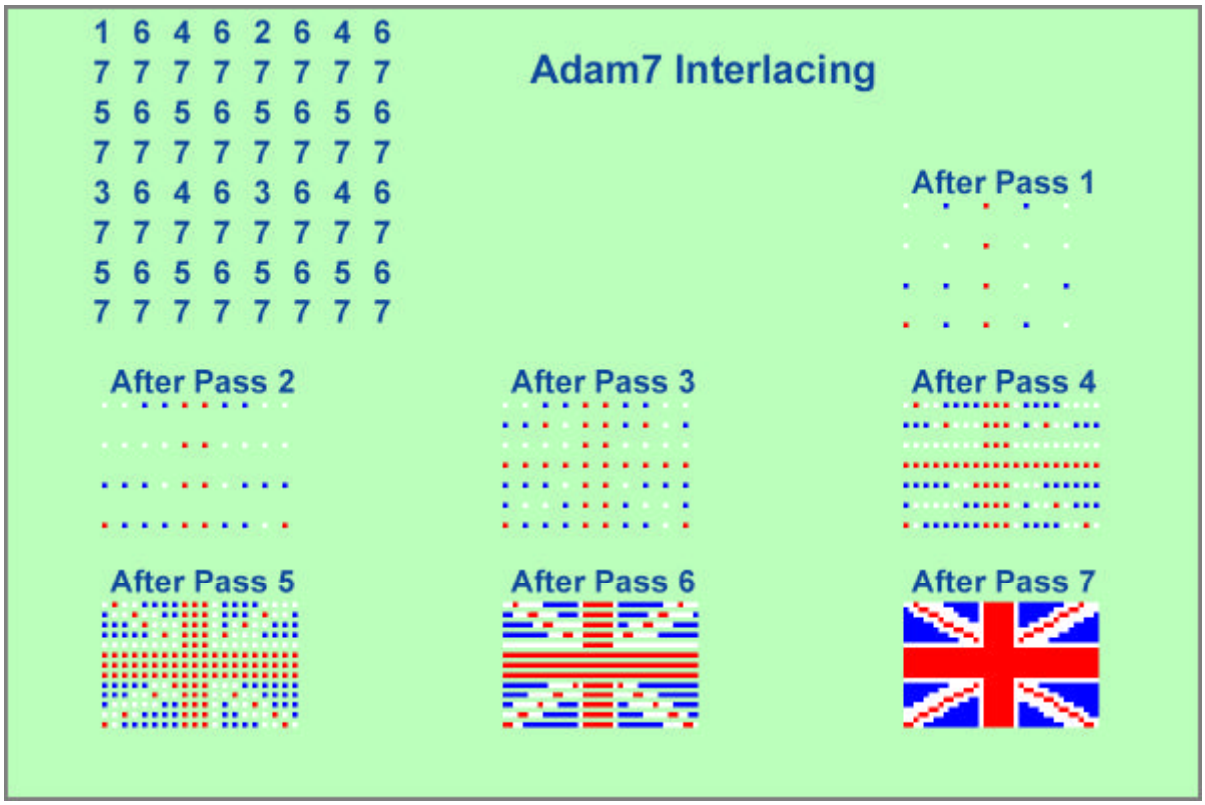


Figure 5.8: Example of Interlacing

PNG defines a number of filters that may be used to prepare the image data for compression. The filters define a set of strategies for changing the representation of each pixel depending on what the neighbouring pixel values are. Different filters can be used for each scanline so there is a certain amount of preprocessing before the optimal one is chosen. Filtering transforms the byte sequence in a scanline to an equal length sequence of bytes preceded by the filter type (see Figure 5.9).

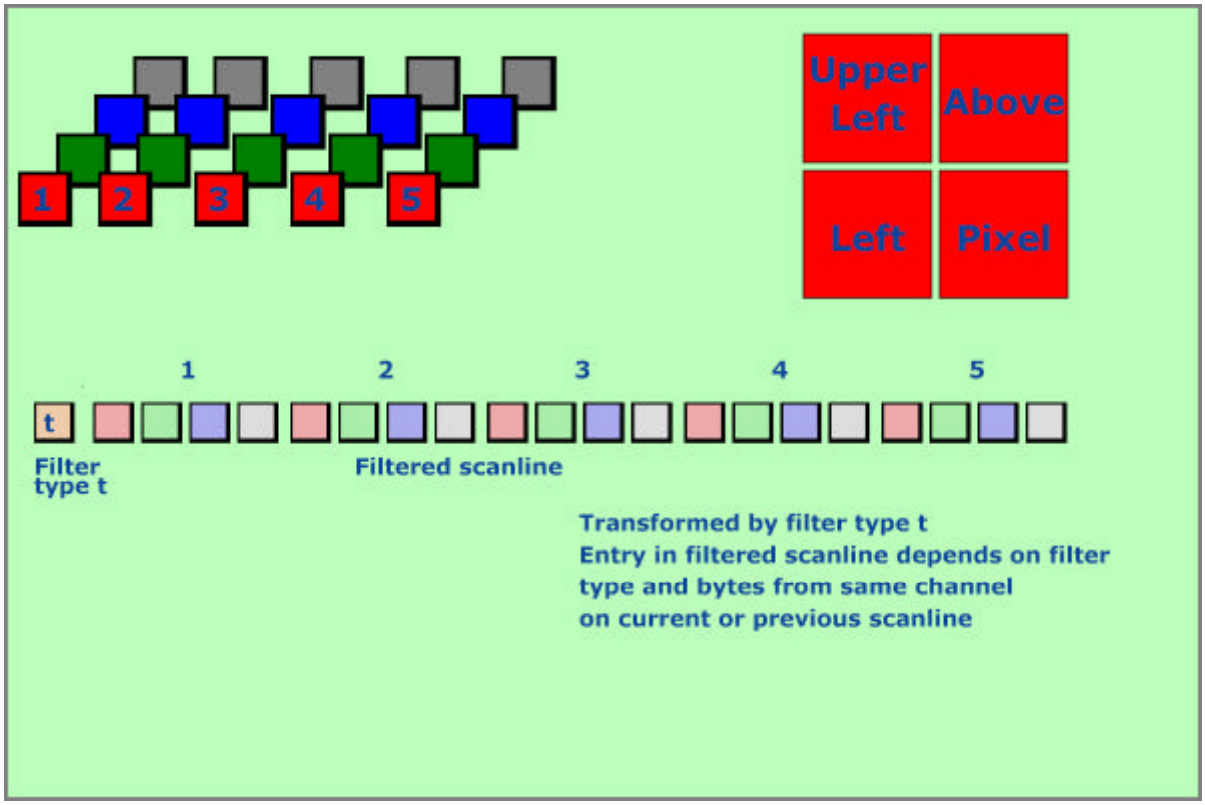


Figure 5.9: PNG Filtering

The set of methods each modify the R, G, B or Alpha value of a pixel by the pixel values of the same type to the **left**, **above**, and **left and above**. The process is easily undone so that at the far end the original values can be recovered. The aim is to produce values that are easy to compress. For example, if each Red value was an increment from the previous value of 10 say (R values could be 10, 20, 30, 40, 50, etc). this would be an image where the red content was rising linearly from one side to the other. If the filter applied was to take the current pixel value and subtract the value of its left neighbour, the values would now be 10, 10, 10, 10, 10 and thus relatively easy to compress. The set of filter types currently are:

- **None:** out=Pixel
- **Sub:** out=Pixel-Left
- **Up:** out=Pixel-Up
- **Average:** out=Pixel-floor(0.5*(Left-Above))
- **Paeth:** see Figure 5.10

The Paeth algorithm is named after Alan W Paeth and there are no patents on the algorithm.

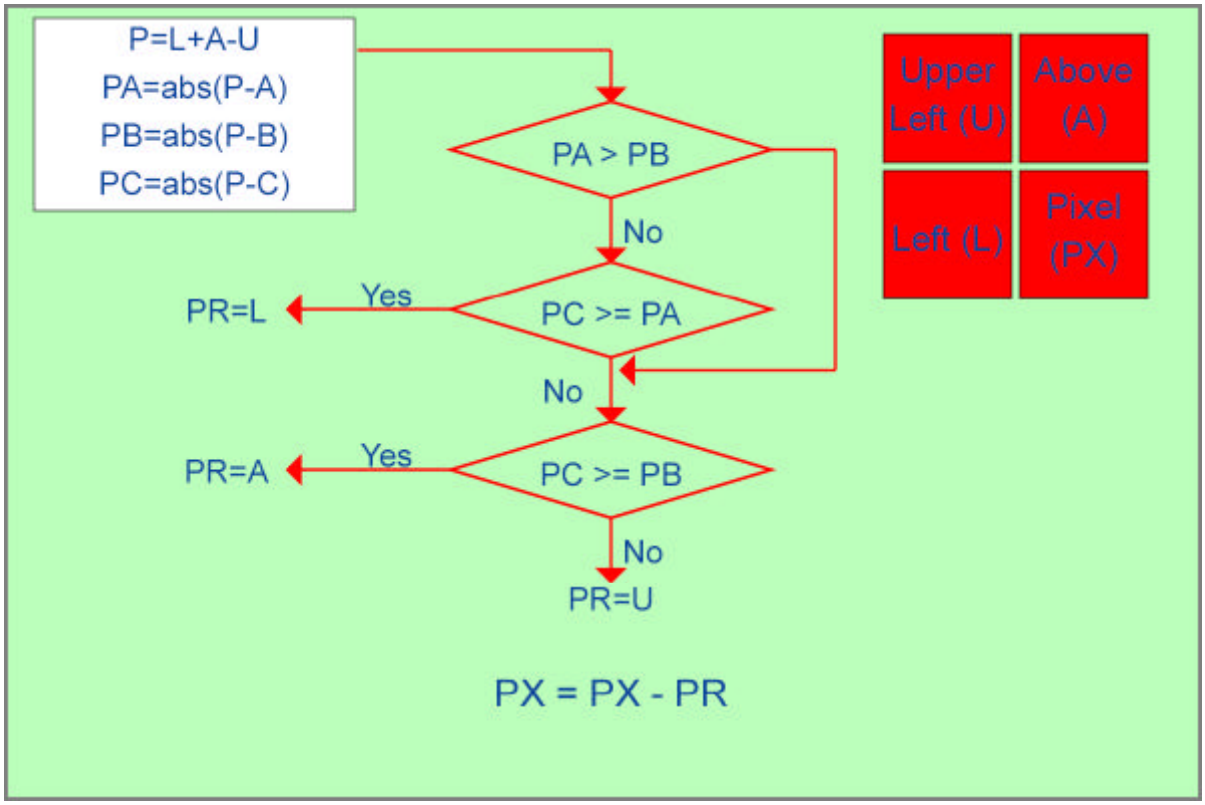


Figure 5.10: Paeth Filtering

The sequence of filtered scanlines in the interlaced PNG image derived are compressed. The concatenated filtered scanlines for the interlaced PNG image are the input to the compression stage. The output from the compression stage is a single compressed datastream.

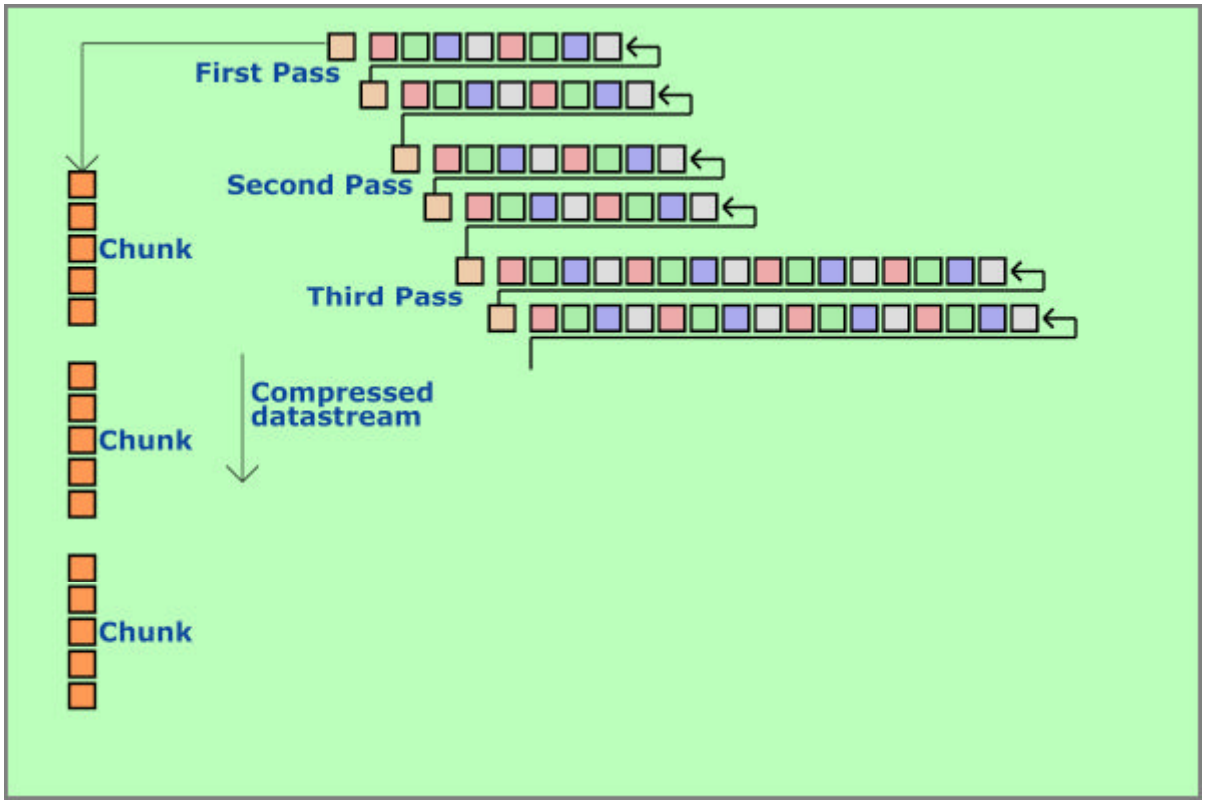


Figure 5.11: PNG Compression

Chunking provides a convenient breakdown of the compressed datastream into manageable chunks (see Figure 5.12). Each chunk has its own redundancy check.

5.5 PNG Datastream Format

The PNG Datastream starts with an 8-byte Signature similar to GIF and is followed by a sequence of **Chunks** as shown in Figure 5.12. Each chunk starts with a **4-byte length** of its chunk data in bytes followed by the **4-byte Chunk Type**. Each Chunk has its own **4-byte CRC** check of the Chunk Data. It is possible to have Chunks with no Chunk Data. The Signature has the characters PNG in it plus some non-printing characters aimed at ensuring that additional carriage returns and line feeds have not been added.

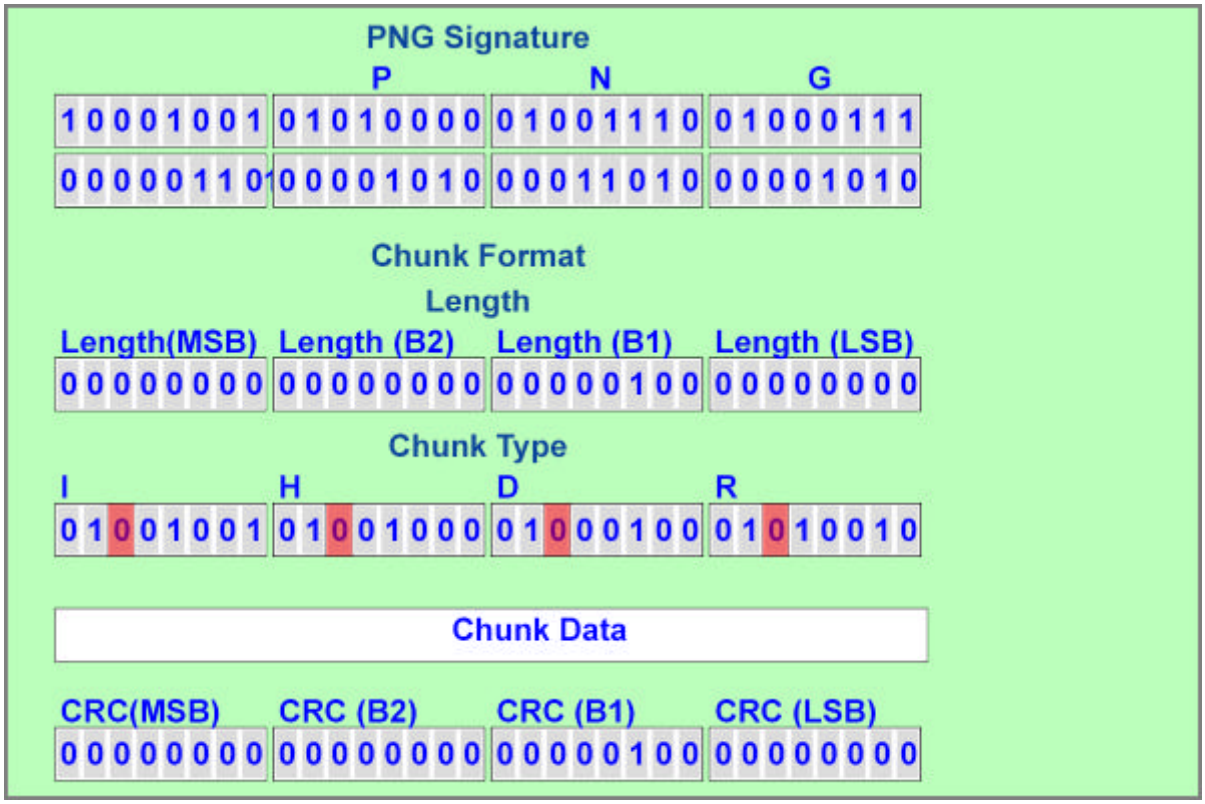


Figure 5.12: PNG Coding

The Chunk Header appears to be a set of ASCII characters but the bit indicated is used as a flag. Whether the four flag bits are set or not shows up in the ASCII value being upper or lower case. If the Flag Bit is not set it is upper case and if it is set it is lower case. From first to last, these four flags are:

- 0=Critical, 1=Ancillary
- 0=public, 1=private
- 0 always
- 0=unsafe to copy, 1=safe to copy

There are 18 chunk types of which the first four are called **critical chunks** that must be understood by a decoder:

- **IHDR**: an image header that is the first chunk
- **PLTE**: palette table associated with indexed PNG images
- **IDAT**: image data chunks
- **IEND**: image trailer that is the last chunk in a PNG datastream

The other 14 chunk types are called **ancillary chunk types** which a decoder may ignore.

- **cHRM**, **gAMA**, **iCCP**, **sBIT**, **sRGB** which provide additional information concerning the colours in the image
- **bKGD**, **hIST**, **tRNS**, **pHYS**, **sPLT** which provide additional information about the image and its presentation
- **iTXt**, **tEXt**, **zTXt**: which provide textual information about the image
- **tIME**: which provides a time stamp indicating when the PNG image was last modified

Briefly, the ancillary information, which may be ignored is:

Type	Description
Background colour (bKGD)	Colour to be used when presenting the image if no better option is available.
Gamma and chromaticity (cHRM, gAMA)	The gamma characteristic of the image with respect to the original scene together with chromaticity characteristics of the RGB values in the source image. Using information about the display device, room lighting, etc allows a more realistic appearance of the image.
ICC profile (iCCP)	Describes the colour space (in the form of an International Color Consortium ICC profile) to which the image samples conform.
Image histogram (hIST)	Frequency estimates of the usage of each entry in the palette by the PNG image.
Physical pixel dimensions (pHYS)	Intended pixel size and aspect ratio to be used in presenting the image.
Significant bits (sBIT)	The number of bits that are significant in the channel values or the palette.
sRGB colour space (sRGB)	The sRGB colour space and the required rendering intent may be specified.
Suggested palette (sPLT)	A reduced palette may be provided for use when the display device is not capable of displaying the full range of colours in the image.
Textual data (iTXt, tEXt, zTXt)	Textual information associated with the source image that may be compressed.
Time (tIME)	Gives the time when the PNG image was last modified.
Transparency (tRNS)	Provides alpha information that allows the source image to be reconstructed when the alpha channel is not retained in the PNG image.

Figure 5.13 shows the format of the IHDR Chunk which contains similar information to the GIF Descriptor. The width and the height of the image are defined in 4 bytes so very large images could be specified. In the diagram, the Colour type is set to 3 indicating that is using a palette. The only compression defined so far is that used for zipping files etc. The only Filter Method defined so far is Filter Method 0 which has the five filters defined earlier. The interlace possibilities are Adam7 and no interlacing (value set to 0).

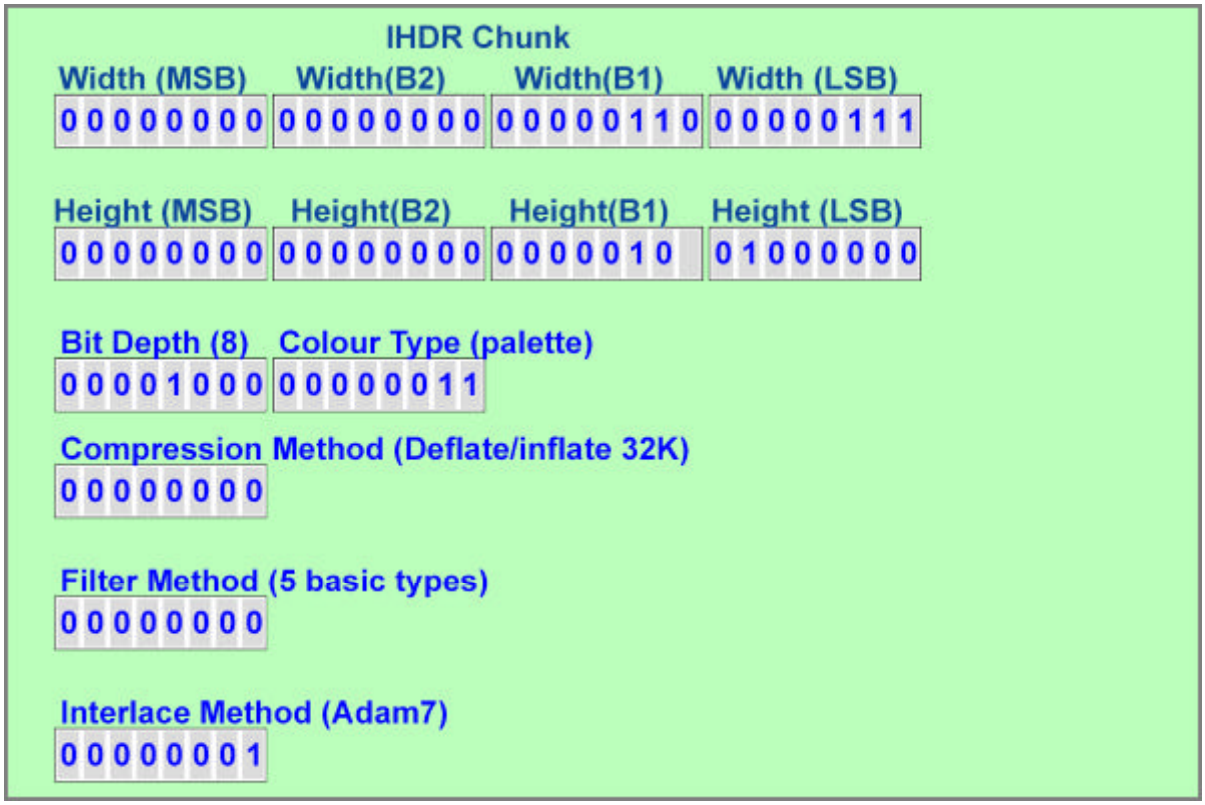


Figure 5.13: IHDR Chunk

The format of the PLTE Chunk is the same as GIF; each palette entry has a byte for Red followed by Green and then Blue.

The IDAT Chunks are zlib (see Appendix D) compressed. This compression method uses Huffman encoding to decrease the number of bits needed. Thus an analysis of the frequency of various values is checked before compression. The method for sending the data is similar to LZW but does not have any patent problem. Normally, a user would call the zlib library to deflate (compress) or inflate (decompress) an IDAT Chunk.

5.6 An Example

If we look at the PNG file equivalent to the GIF one described earlier, it has the form:

```
89 50 4e 47 0d 0a 1a 0a 00 00 00 0d 49 48 44 52 00 00 00 27 00 00 00 19 02
03 00 00 00 cf bd df 7b 00 00 00 0c 50 4c 54 45 ff ff ff ff 00 00 00 00 ff
00 00 00 6f c6 08 f7 00 00 00 01 74 52 4e 53 00 40 e6 d8 66 00 00 00 01 62
4b 47 44 03 11 0c 4c f2 00 00 00 08 68 49 53 54 01 38 01 94 01 03 00 00 87
15 b6 20 00 00 00 1b 74 45 58 74 53 6f 66 74 77 61 72 65 00 67 69 66 32 70
6e 67 20 30 2e 36 20 28 62 65 74 61 29 aa dd 69 92 00 00 00 8c 49 44 41 54
78 9c 9d 90 c1 09 04 21 0c 45 bf e2 c0 e2 d9 22 c4 2a 2c 21 23 9b 7e 9c 4e
3c 86 54 39 89 c3 c0 9e 37 17 1f 44 7c ff 8b 04 15 1e aa 01 b8 7a 76 5c 05
58 25 3a 5e fd 03 db 54 1e 92 20 d0 45 e0 31 29 1b ca 01 a6 03 cd 50 27 31
51 54 47 49 7c a2 19 fa 30 3d c7 3b df 7f f1 e7 31 b3 99 22 aa db 04 2e 76
9b ce 1d 67 67 00 55 bb 4b 86 2b 78 8b 99 2c 2f 4a 33 94 d0 05 a1 ef c6 28
15 29 6f 5c 56 9e f4 f9 07 d0 0d 7d 09 51 da 2a 32 6c f9 00 00 00 00 49 45
4e 44 ae 42 60 82
```

This can be shown better as follows:

```
89 50 4e 47 0d 0a 1a 0a  PNG Signature
00 00 00 0d  IHDR Chunk Length = 13
49 48 44 52  IDAT Chunk
00 00 00 27  Width=39
00 00 00 19  Height=25
02  Bit Depth=2
03  Colour Type=3 (indexed colour)
00  Compression Method=0
00  Filter Method=0
00  Interlace Method=0, No Interlace
cf bd df 7b  Chunk CRC Check
00 00 00 0c  Chunk Length=12
50 4c 54 45  PLTE Chunk
ff ff ff  White
ff 00 00  Red
00 00 ff  Blue
00 00 00  Black
6f c6 08 f7  Chunk CRC Check
00 00 00 01  Chunk Length=1
74 52 4e 53  tRNS Chunk
00  Alpha value for Palette Index 0 is 0 (opaque)
40 e6 d8 66  CRC Check for Chunk
00 00 00 01  Chunk Length=1
62 4b 47 44  bKGD Chunk
03  Palette Index 3
11 0c 4c f2  CRC Check for Chunk
00 00 00 08  Chunk length=8
68 49 53 54  hIST Chunk
01 38  312:White
01 94  404:red
01 03  259:Blue
00 00  0:Black
87 15 b6 20  CRC Check for Chunk
00 00 00 1b  Chunk Length=27
74 45 58 74  tEXT
53 6f 66 74 77 61 72 65 00  Software
67 69 66 32 70 6e 67 20 30  gif2png 0
2e 36 20 28 62 65 74 61 29  .6 (beta)
aa dd 69 92  CRC Check for Chunk
00 00 00 8c  Length=140
49 44 41 54  IDAT Chunk
78 9c 9d 90 c1 09 04 21 0c 45 bf e2
c0 e2 d9 22 c4 2a 2c 21 23 9b 7e 9c 4e 3c 86 54 39 89 c3 c0 9e 37 17 1f
44 7c ff 8b 04 15 1e aa 01 b8 7a 76 5c 05 58 25 3a 5e fd 03 db 54 1e 92
20 d0 45 e0 31 29 1b ca 01 a6 03 cd 50 27 31 51 54 47 49 7c a2 19 fa 30
3d c7 3b df 7f f1 e7 31 b3 99 22 aa db 04 2e 76 9b ce 1d 67 67 00 55 bb
4b 86 2b 78 8b 99 2c 2f 4a 33 94 d0 05 a1 ef c6 28 15 29 6f 5c 56 9e f4
f9 07 d0 0d 7d 09 51 da
2a 32 6c f9  CRC Check
00 00 00 00  Chunk Length=0
49 45 4e 44  IEND Chunk
ae 42 60 82  CRC Check for IEND Chunk
```

The file is 306 bytes long compared with 187 bytes for the equivalent GIF image. Part of this is due to including a number of ancillary chunks to illustrate the format. If we removed these and accepted the defaults (which in this case would be correct), the file size would be decreased by 85 bytes to 221 bytes. So for very small images that do not need the facilities of PNG, it is likely that the PNG file size will be greater than the equivalent GIF. In both cases the size of the image data is 140 bytes. As you would expect with the similarity between the algorithms, there is little to choose between the compression methods.

Greg Roelofs has done some more detailed tests and had these results:

File	Size as GIF	Size as PNG	Size as Crushed PNG	Improvement (%)
Linux penguin	38280	35224	34546	10
Small penguin	1249	722	710	43
Big Crash	298529	283839	282948	5
L Georges	20224	20476	19898	2
Rasterman	4584	4812	4731	-3
Sun Logo	1226	566	550	55
Scsi	27660	20704	19155	31

As a separate exercise, he transformed the W3C icon library of 448 GIF images with a total size of 1.810 Mbytes and this reduced to 1.555 Mbytes as PNG images, a saving of 14%. These savings are not enormous but are significant.

The one GIF image that was smaller than the equivalent PNG in the table was an interlaced image. PNG's interlacing does allow the image to be seen quicker but the many passes does make it more difficult to compress.

6. CGM: the Computer Graphics Metafile

6.1 Introduction

The Computer Graphics Metafile (CGM) is a well established ISO Standard for transmitting 2D vector graphics data between applications. It is widely used in the engineering industry (aeronautics, automobile, process engineering, architectural design etc).

6.2 Structure

The structure of a CGM file is shown in Figure 6.1. Picture size and scaling and properties such as line width and background colour are defined in the Picture Descriptor. This is equivalent to styling provided for the <body> element in an HTML page.

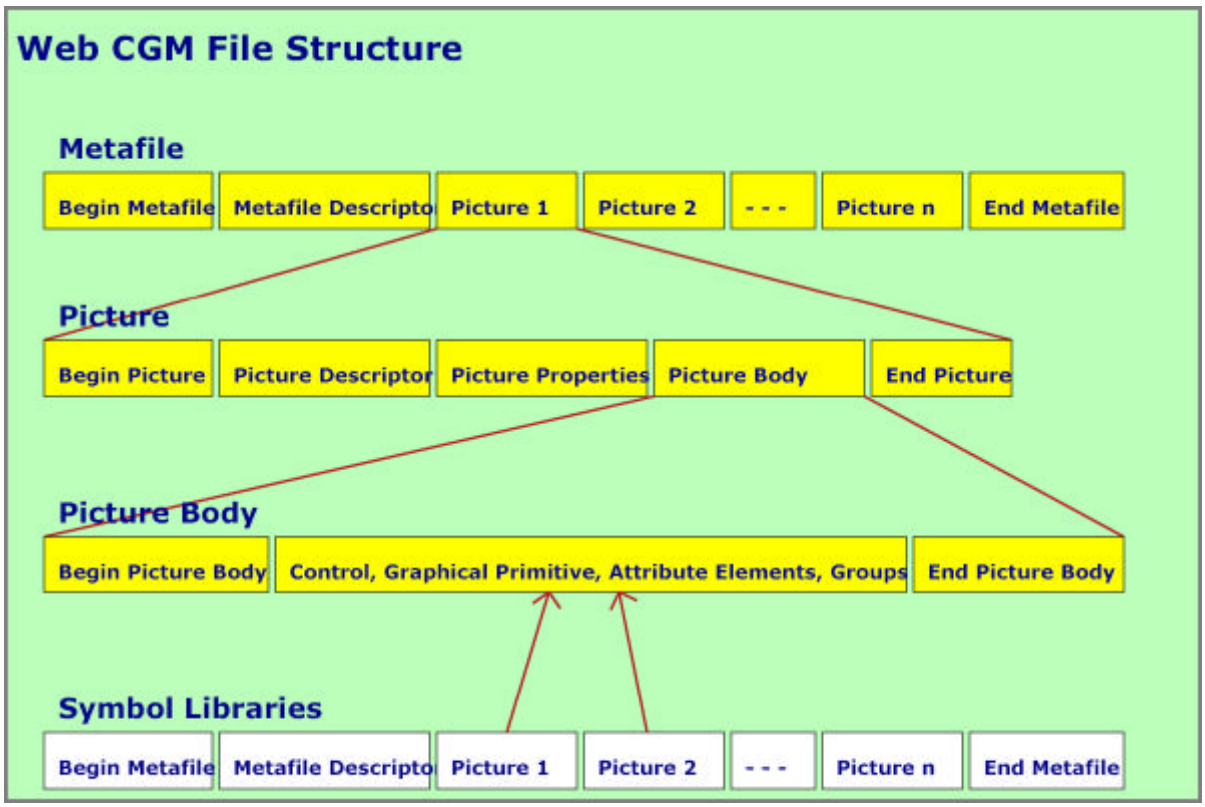


Figure 6.1: CGM Architecture

6.3 CGM Web Profile

The CGM community saw major advantages in using CGM rather than images on the Web:

- Vector graphics can be zoomed in and out while retaining the quality of the picture, unlike images.
- Vector graphics files are smaller and can be downloaded and viewed faster than images.
- Vector graphics can be interacted with in a meaningful way.
- Text in a CGM vector graphics drawing can be searched as easily as text in an HTML page.

In consequence, CGM suppliers provided CGM plug-ins to access CGM vector graphics on the Web using the existing encodings. A CGM MIME type was agreed in 1995. The only problem was that the CGMs produced by one vendor could not be read by viewers produced by another as different Profiles were implemented and the hyperlinking mechanisms introduced differed from one supplier to another.

A joint activity between the World Wide Web Consortium (W3C) and the CGM Open Consortium [12] (launched in May 1998) was initiated to define a common Web Profile for CGM that would be accepted both by ISO and W3C. This resulted in the WebCGM Profile, completed in January 1999 [6].

WebCGM was based on the ATA CGM Profile for graphics interchange (GREXCHANGE). The Graphics Working Group (ATA 2100) of ATA, the Air Transport Association, had defined this CGM Profile for the aerospace industry. It was also working towards an intelligent graphics exchange profile (IGEXCHANGE) that associated semantic information to aid query, searching and navigation.

In WebCGM, each picture contains CGM graphic elements. There are 4 groupings of graphical elements provided:

- **grobject**: a graphical object with a unique id and possibly linkURI and tooltip attributes. It is used to identify sources and destinations of hyperlinks. It is also possible to define the region of the group for picking and the initial view when the group is linked to. For example, more than the group may need to be visible to indicate the context.
- **layer**: this has a name and a list of objects. It allows a picture to be divided into a set of graphical layers that can be used to switch display to parts of an illustration.
- **para**: defines a paragraph as the grouping of several text drawing elements. The elements may be scattered across the drawing but for searching purposes are similar to an HTML paragraph.

- **sub-para**: a sub-paragraph used to identify fragments of text (for example as hotspots within a paragraph).

These provide the basis for searching and linking within and between CGM pictures. An object may be the target of a link. Browsers are expected to move the object into view and scale it to fit into the viewport. If the object has a **ViewContext** attribute the rectangle defining the view context must be within the viewport.

Links from WebCGM objects are defined by **linkURI** elements that are modelled on the XLink facilities. Objects may have multiple links. Links can be bi-directional. Linkage can be from places outside the CGM and links from the CGM can be to any destination defined by a URL. Following a link can display the new picture in a separate window, load the picture into the current frame, load it over the parent of the current frame or replace the current picture.

WebCGM is a reasonably full profile of CGM containing a rich set of graphics elements:

- Polylines, disjoint polylines, polygons, polygon sets.
- Rectangles, circles, ellipses, circular and elliptical arcs, pie slices.
- Text: both the Restricted Text primitive of CGM (which defines its extent box) and the Append Text element (continuation of a text string with a change of attributes).
- Closed Figure and Compound Line: allows complex paths to be defined as a sequence of other primitives.
- Polysymbol: placement of a sequence of symbols defined in the Symbol Library (another valid WebCGM metafile).
- Smooth curves: the smooth piece-wise cubic Bezier defined by CGM's Polybezier element.
- Cell Array and Tile Array allow PNG, and JPEG images to be integrated with the vector drawing.

Most of the line and fill attributes of CGM are included but only as INDIVIDUAL attributes. The bundled attribute functionality of CGM is omitted. Thus, WebCGM diagrams consider properties such as linestyle, color, fill types etc as content rather than styling.

The full set of CGM colour models is provided including sRGB and sRGB-alpha. International text is defined by selecting either Unicode UTF-8 or UTF-16.

Probably the most widely used Viewer is the Micrografx free ActiveCGM plug-in. SDI has also released a CGM Plug-in while Tech Illustrator has a TI/WebCGM Hotspot Plug-in module to author hotspots for exporting to CGMs. There is good industrial support for WebCGM and it is widely used in the CAD and aerospace industries. A major interoperabilty demonstration took place at XML Open in Granada in May 1999.

A good source of further information on CGM is CGM Open [12], an organisation dedicated to open and interoperable standards for the exchange of graphical information. The W3C Web site is also a valuable source of news and reference information.

6.6 An Example

CGM has a number of different encodings. The most verbose is the **clear text encoding** which is primarily designed for debugging. The UK Flag in the Clear Text Encoding would be something like:

```
BEGMF 'UK Flag Clear Text Encoded'; Begin Metafile
MFVERSION 1; Defines the Version of CGM, 1 is the smallest
MFDESC 'RAL GKS 1.11 Cray/COS 88/ 6/17'; Drawn by GKS using a CRAY
VDCTYPE REAL; Real values will be used to define the picture
INTEGERPREC -32768, 32767 Range for integers
REALPREC -8191.0, 8191.0, 8 A default exponent of 8
COLRPREC 255; Colour values in the range 0 to 255
MFELEMLIST 'DRAWINGPLUS'; Defines the set of elements that may appear in the Metafile
BEGPIC 'United Kingdom Flag';Begin Picture
VDCEXT (0.0,0.2) (1.0,0.8);Min and Max of Coordinates
COLRMODE DIRECT; Colour Specified Directly
BEGPICBODY;Start of Drawing
CLIPRECT (0.0,0.2) (1.0,0.8); Anything drawn outside this area will not appear
INTSTYLE SOLID; Area is filled with a solid colour
FILLCOLR 255 255 255; Fill White
POLYGON (0.0,0.2) (0.0,0.8) (1.0,0.8) (1.0,0.2) (0.0,0.2); Fill whole area white
FILLCOLR 255 0 0; Now fill in Red
POLYGON (0.45,0.2) (0.45,0.8) (0.55,0.8) (0.55,0.2) (0.45,0.2);
POLYGON (0.0,0.45) (1.0,0.45) (1.0,0.55) (0.0,0.55) (0.0,0.45);
POLYGON (0.0,0.755) (0.0,0.8) (0.375,0.575) (0.3,0.575) (0.0,0.755);
POLYGON (0.625,0.425) (0.7,0.425) (1.0,0.245) (1.0,0.2) (0.625,0.425);
POLYGON (0.5750,0.59) (0.575,0.575) (0.625,0.575) (1.0,0.8) (0.925,0.8)
(0.575,0.59);
POLYGON (0.0,0.2) (0.075,0.2) (0.425,0.41) (0.425,0.425) (0.375,0.425)
(0.0,0.2);
FILLCOLR 0 0 255; Blue
POLYGON (0.1,0.8) (0.425,0.8) (0.425,0.605) (0.1,0.8);
POLYGON (0.0,0.74) (0.0,0.575) (0.275,0.575) (0.0,0.74);
POLYGON (0.575,0.8) (0.9,0.8) (0.575,0.605) (0.575,0.8);
POLYGON (0.725,0.575) (1.0,0.74) (1.0,0.575) (0.725,0.575);
POLYGON (0.0,0.425) (0.275,0.425) (0.0,0.26) (0.0,0.425);
POLYGON (0.1,0.2) (0.425,0.2) (0.425,0.395) (0.1,0.2);
POLYGON (0.725,0.425) (1.0,0.425) (1.0,0.26) (0.725,0.425);
POLYGON (0.575,0.2) (0.575,0.395) (0.9,0.2) (0.575,0.2);
ENDPIC;
ENDMF;
```

The CGM is more verbose than it might be in that each area is filled independently and no use is made of completely covering one area by another. Figure 6.2 shows the result at various sizes to illustrate that this is not a pixel description but one that be scaled to any size without loss of detail. As written this is around 1330 bytes in size but we have included a number of defaults that could be removed and there are quite a few superfluous zero characters. Removing these would bring the size down to about 1000 bytes. If it had been expressed in integers that would save another 100 bytes. The **Character Encoding** of CGM uses one or two bytes to represent the command and packs up the data values to 1, 2 or 3 bytes. This would bring the size of the flag definition down to about 470 bytes. Thus the CGM file even when it is compressed is likely to be larger than the small pixel image. However if the GIF or PNG image was made the same size as the largest CGM image it would be significantly larger.

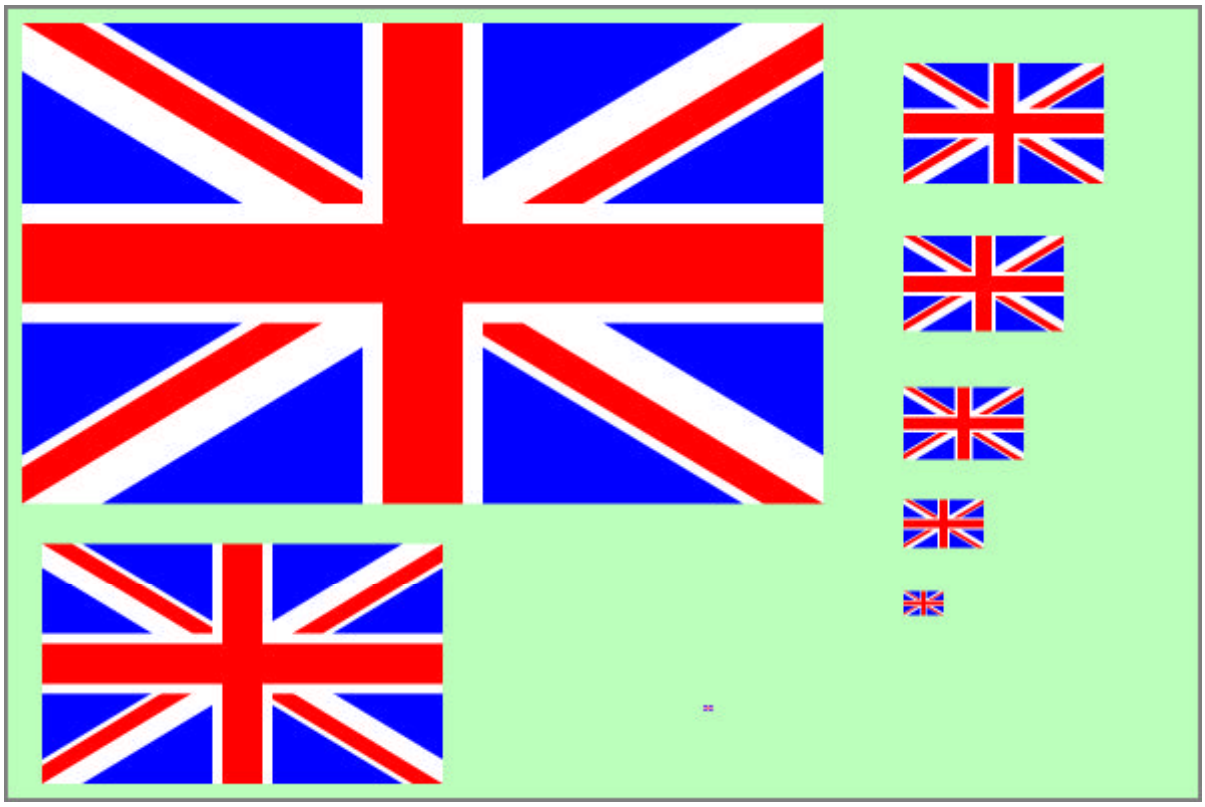


Figure 6.2: CGM Example

7. JPEG: Joint Photographic Experts Group

7.1 Introduction

JPEG is an ISO Standard dating back to 1993. It was designed for the transmission of continuous-tone still images (both greyscale and colour) efficiently. High compression ratios while retaining image fidelity was the aim. A particular feature of JPEG is that users can trade quality against compression ratio. That implies that some loss of information is allowed if very high compression ratios are required. This is the main distinction between JPEG and the 2D image formats discussed so far.

Like GIF, it uses smart algorithms to achieve the compression and almost all of these are subject to patents owned by IBM, AT&T, Mitsibushi and others. JPEG is really a set of 29 different coding methods. Most implementations only implement a subset of the possibilities. The simplest is called the **baseline sequential** mode and this is free of the patents listed above. It has a sample precision of 8 bits (no better than PNG) and uses a Huffman encoding scheme. This is an earlier encoding scheme similar to LZW in that it uses a Code table. Huffman encoding finds the most frequently used sequences and allocates them to early positions in the Code Table (and they can therefore be defined in a small number of bits). The downside of this approach is that the whole image or the part being compressed needs to be analysed before the compression takes place. There is also the need to transmit the Code Table unlike LZW.

7.2 Structure

For baseline sequential JPEGs, the process is as follows:

- Divide the image into 8 by 8 pixel blocks and process each individually
- The block of data is transformed via a **Forward Discrete Cosine Transform (FDCT)**
- The result of the FDCT is put through a **Quantizer** that reduces the precision
- The image is then compressed using run-length, Huffman, or arithmetic encoding

Some mathematical transformations exist that transform a set of values from one system of measurement to another where data in the new system is easier to compress. The PNG filters are transformations of that type. FDCT is mathematically complicated. The aim is to take cosine wave forms with different frequencies and determine how much of each frequency pattern exists in the pixel block. In the simplest example, if all the data values are the same, the FDCT transforms the 64 values into a single data value. For a smoothly changing image, the number of data values in the transformed matrix that are non-zero may be of the order of 6. This is how JPEG achieves its compression. The drawback is that the inverse transformation does not result in the same image. Variations are small and similar to what you would get if you performed an inverse cosine. The reason for choosing 8 by 8 blocks was that the aim was to implement the algorithm in the hardware of devices and that was the limit of the VLSI possible at the time.

The 8 by 8 matrix after the FDCT contains the low-frequency terms in the lower left of the matrix and the high frequency terms in the upper right. The major characteristics of the image come from the low-frequency terms and it is possible that the high frequency terms are just errors or noise in the original image. Quantization **reduces** the accuracy of the high frequency terms and thus reduces the amount of information that needs to be transmitted. Quanization can cause errors in the image after transmission and restoration of the order of 2 or 3% which in a photograph are probably not seen by the average viewer. The quantization table that indicates the reduction in precision either has to be sent with the image or the encoder and decoder agree on always using the same quantization table.

Coding the 8 by 8 result of quantization first linearizes the 64 entries but this is not done by row. Instead the values are taken zigzagging backwards and forwards across the array at 45 degrees. This is done because of the relative positioning of the important terms in the transformed and quantized data. The 64 values are then run-length encoded and Huffman or arithmetic encoded.

JPEG really does not use any great sophistication in the compression. The high compression ratios are achieved by the FDCT picking out the major features from the noise and the quantizer reducing the accuracy of the less important values. The result is an image that is very similar to the original but not the same. If the main information in the image was something that might be mistaken for noise, it could be lost completely. For example, hard edges in an image are almost certainly going to appear fuzzy after the JPEG encoding and decoding.

Attempting to minimize the size of the JPEG file by a high level of compression can result in rather poor images as can be seen in Figure 7.1.

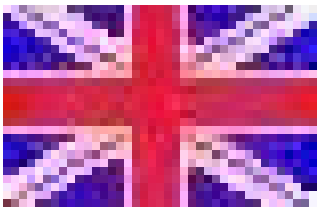


Figure 7.1: Highly Compressed JPEG

JPEG is widely used for photographs of real world scenes on the Web and will continue to be so used in the future. However, the functionality available with PNG does make it a viable competitor to JPEG for some types of photographic images and it does have the merit that it is lossless.

7.3 JPEG 2000

A new version of JPEG has just been completed called JPEG 2000. JPEG 2000 aims to provide a new image coding system using state of the art compression techniques, based on the use of wavelets. It is ambitiously aimed at consumer electronic devices like digital cameras and mobile phones right through to pre-press and medical imaging. It reached Draft International status in 2000 and the hope is to get a full International Standard in 2001.

The new compression scheme, **Wavelet Compression**, is based on wavelets which are widely used in computer graphics and other areas. The technique has the property that it transforms the information into a set of terms where the first contains the main information and each subsequent term enhances the information already described. It has its origins in seismic analysis. It is ideal for the type of transmission where the main characteristics of the whole image are transmitted followed by successive refinements. It thus has a similar effect as interlacing but in this case it is a fundamental part of the image transformation. Thus a low resolution image could be transmitted and the user, if in need of higher resolution, would only need to receive the next wavelet and so on.

The aim will be to define some standard wavelets for transforming all images so that the wavelet definitions themselves do not have to be sent and could even be built into the hardware of consumer electronics.

JPEG 2000 also allows the user to add encrypted copyright information to a JPEG file. There is support for better colour management similar to PNG

PNG still has advantages over JPEG 2000 in some areas. For example, JPEG 2000 has no support for transparency, which is a big liability on the Web. It is also a lossy image format.

Appendix A

References

There are some useful Web sites and books relevant to data compression and image formats:

1. Graphics File Formats, Wayne Brown and Barry Shepherd,
   Prentice-Hall, 1995
2. www.wapforum.org/what/technical/SPEC-WAESpec-19990524.pdf
   WBMP Specification, May 1999
3. http://www.w3.org/Graphics/GIF/spec-gif89a.txt
   GIF89a Specification, 1989.
4. http://www.dcs.ed.ac.uk/home/mxr/gfx/2d/GIF87a.txt
   GIF87a Specification, 1987.
5. http://www.w3.org/TR/REC-png.pdf
   PNG Specification, 1996.
6. http://www.w3.org/Graphics/WebCGM/, WEB CGM Profile, 1999.
7. http://browserwatch.internet.com/plug-in.html, Plug-In List
8. http://www.unisys.com/unisys/lzw/, UniSys LZW Patent
9. A Technique for High-Performance Data Compression, Terry A Welch,
   IEEE Computer, Vol 17 No 6, 1984.
10. The CGM Handbook, Lofton R Henderson and Anne M Mumford,
    Academic Press, 1993
11. PNG The Definitive Guide, Greg Roelofs,
    O Reilly, 1999
12. http://www.cgmopen.org
    CGM Open
13. http://www.ietf.org/rfc/rfc1951.txt,
    DEFLATE Compressed Data Format Specification version 1.3

Appendix B: PC Plug-ins Available

MIME Type	Extensions	Plug-in
application/asf	.asf	Media Player
application/atmosphere	.aer	Atmosphere
application/epic	.pic	RapidVue
application/futuresplash	.spl, .swf	Flash
application/mbedlet	.mbd	mBED
application/netwriter	.nwr	NetWriter
application/octet-stream	.pan	iMove
application/pdf	.pdf, .rmf	Acrobat
application/postscript	.ai, .eps, .ps	GoScript
application/toolbook	.tbk	Neuron
application/truebasic	.tra, .trc	WebBASIC
application/vnd.fdf	.fdf	Acrobat
application/windowsmedia	.wma	Media Player
application/x-calquick	.cqk	Calendar Quick
application/x-mwf	.mwf	MapGuide
application/x-pn-realaudio	.ram, .rm, .rpm	RealPlayer
application/x-shockwave-flash	.spl, .swf	Flash
audio/aiff	.aiff	Apple Quicktime, Beatnik, Media Player, ViewMovie XT
audio/basic	.au	Apple Quicktime, Beatnik, Media Player, ViewMovie XT
audio/midi	.mid, .midi	Apple Quicktime, Beatnik, Crescendo, Koan, Media Player, MidiShare, NET TOOB Stream
audio/mp3	.mp3	Crescendo, Koan, Liquid MusicPlayer, Media Player
audio/mpeg	.mpe, .mpeg	Apple Quicktime, Media Player, NET TOOB Stream
audio/mpeg3	.mp3	Crescendo, Koan, Liquid MusicPlayer, Media Player
audio/mod	.mod	Beatni, MODPlug
audio/rmf	.rmf	Beatnik
audio/wav	.wav	Apple Quicktime, Beatnik, Liquid MusicPlayer, Media Player, NET TOOB Stream
audio/x-aiff	.aif, .aiff	Apple Quicktime, Beatnik, Media Player, ViewMovie XT
audio/x-midi	.mid, .midi	Apple Quicktime, Beatnik, Crescendo, Koan, Media Player, NET TOOB Stream
audio/x-mpeg	.mpg, .mpeg	Apple Quicktime, Media Player, NET TOOB Stream
audio/x-mod	.mod	Beatnik
audio/x-rmf	.rmf	Beatnik
audio/x-wav	.wav	Apple Quicktime, Beatnik, Liquid MusicPlayer, Media Player, NET TOOB Stream

MIME Type	Extensions	Plug-in
image/bmp	.bmp	Innomage, Prizm
image/cals	.cal	Prizm, Corp. Ed
image/cgm	.cgm	MetaWeb
image/gif	.gif	Innomage, Prizm
image/jpeg	.jpg, .jpeg	Innomage, Prizm, RapidVue
image/kqp	.kqp	RapidVue
image/pic	.pic, .pict	Innomage, RapidVue
image/png	.png, .ptng	Apple Quicktime, Innomage
image/psd	.psd	Innomage
image/targa	.tga	Innomage
image/tiff	.tif, .tiff	Apple Quicktime, Innomage, Prizm, RapidVue
image/x-bmp	.bmp	Apple Quicktime, Innomage, Prizm, RapidVue
image/x-cals	.cal	Prizm
image/x-dcx	.dcx	Prizm
image/x-macpaint	.targa	Apple Quicktime, Innomage
image/x-pcx	.pcx	Innomage, Prizm
model/vnd.dwf	.dwf	WHIP!
video/avi	.avi	Apple Quicktime, Media Player, NET TOOB Stream
video/flc	.flc	Apple Quicktime, NET TOOB Stream
video/mpeg	.mpe, .mpeg	Apple Quicktime, Media Player, NET TOOB Stream
video/quicktime	.mov	Apple Quicktime, MovieScreamer, NET TOOB Stream, TEC Player
video/vnd.vivo	.viv	VivoActive PowerPlayer
video/x-mpeg	.mpe, .mpeg	Apple Quicktime, Media Player, NET TOOB Stream
video/x-pn-realvideo	.ram, .rm, .rpm	RealPlayer
x-world/x-svr	.svr, .vrt, .xvr	Superscape e-Visualizer
x-world/x-vrt	.svr, .vrt, .xvr	Superscape e-Visualizer

Appendix C: The Unisys LZW Patent

A data compressor compresses an input stream of data character signals by storing in a string table strings of data character signals encountered in the input stream. The compressor searches the input stream to determine the longest match to a stored string. Each stored string comprises a prefix string and an extension character where the extension character is the last character in the string and the prefix string comprises all but the extension character. Each string has a code signal associated therewith and a string is stored in the string table by, at least implicitly, storing the code signal for the string, the code signal for the string prefix and the extension character. When the longest match between the input data character stream and the stored strings is determined, the code signal for the longest match is transmitted as the compressed code signal for the encountered string of characters and an extension string is stored in the string table. The prefix of the extended string is the longest match and the extension character of the extended string is the next input data character signal following the longest match. Searching through the string table and entering extended strings therein is effected by a limited search hashing procedure. Decompression is effected by a decompressor that receives the compressed code signals and generates a string table similar to that constructed by the compressor to effect lookup of received code signals so as to recover the data character signals comprising a stored string. The decompressor string table is updated by storing a string having a prefix in accordance with a prior received code signal and an extension character in accordance with the first character of the currently recovered string.

United States Patent [19]

Welch

[11] **Patent Number: 4,558,302**

[45] **Date of Patent: Dec. 10, 1985**

[54] **HIGH SPEED DATA COMPRESSION AND DECOMPRESSION APPARATUS AND METHOD**

[75] Inventor: Terry A. Welch, Concord, Mass.

[73] Assignee: Sperry Corporation, New York, N.Y.

[21] Appl. No.: 505,638

[22] Filed: Jun. 20, 1983

[51] Int. Cl.⁴ G06F 5/00

[52] U.S. Cl. ..... 340/347 DD; 235/310

[58] Field of Search ..... 340/347 DD; 235/310, 235/311; 364/200, 900

[56] **References Cited**

U.S. PATENT DOCUMENTS

4,464,650 8/1984 Eastman ..... 340/347 DD

OTHER PUBLICATIONS

Ziv, "IEEE Transactions on Information Theory", IT-24-5, Sep. 1977, pp. 530-537.

Ziv, "IEEE Transactions on Information Theory", IT-23-3, May 1977, pp. 337-343.

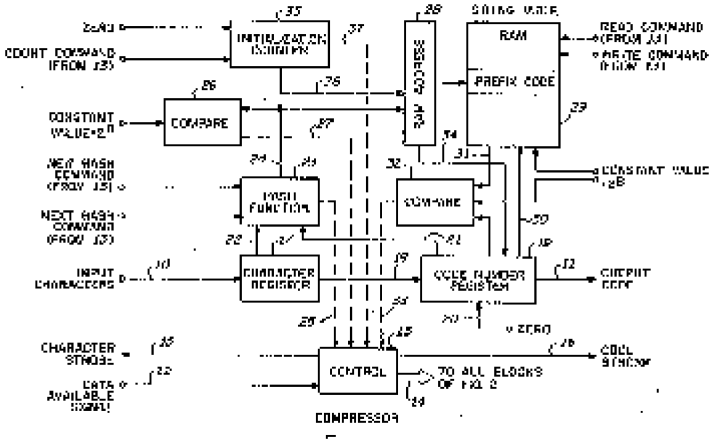
Primary Examiner—Charles D. Miller

Attorney, Agent, or Firm—Howard P. Terry; Albert B. Cooper

[57] **ABSTRACT**

A data compressor compresses an input stream of data character signals by storing in a string table strings of data character signals encountered in the input stream. The compressor searches the input stream to determine the longest match to a stored string. Each stored string comprises a prefix string and an extension character where the extension character is the last character in the string and the prefix string comprises all but the extension character. Each string has a code signal associated therewith and a string is stored in the string table by, at least implicitly, storing the code signal for the string, the code signal for the string prefix and the extension character. When the longest match between the input data character stream and the stored strings is determined, the code signal for the longest match is transmitted as the compressed code signal for the encountered string of characters and an extension string is stored in the string table. The prefix of the extended string is the longest match and the extension character of the extended string is the next input data character signal following the longest match. Searching through the string table and entering extended strings therein is effected by a limited search hashing procedure. Decompression is effected by a decompressor that receives the compressed code signals and generates a string table similar to that constructed by the compressor to effect lookup of received code signals so as to recover the data character signals comprising a stored string. The decompressor string table is updated by storing a string having a prefix in accordance with a prior received code signal and an extension character in accordance with the first character of the currently recovered string.

181 Claims, 9 Drawing Figures



Appendix D: ZLib Compression

D.1 Compression algorithm (deflate)

The deflation algorithm used by gzip (also zip and zlib) is a variation of LZ77 (Lempel-Ziv 1977, see reference below). It finds duplicated strings in the input data. The second occurrence of a string is replaced by a pointer to the previous string, in the form of a pair (distance, length). Distances are limited to 32K bytes, and lengths are limited to 258 bytes. When a string does not occur anywhere in the previous 32K bytes, it is emitted as a sequence of literal bytes. (In this description, `string' must be taken as an arbitrary sequence of bytes, and is not restricted to printable characters.)

Literals or match lengths are compressed with one Huffman tree, and match distances are compressed with another tree. The trees are stored in a compact form at the start of each block. The blocks can have any size (except that the compressed data for one block must fit in available memory). A block is terminated when deflate() determines that it would be useful to start another block with fresh trees. (This is somewhat similar to the behavior of LZW-based _compress_.)

Duplicated strings are found using a hash table. All input strings of length 3 are inserted in the hash table. A hash index is computed for the next 3 bytes. If the hash chain for this index is not empty, all strings in the chain are compared with the current input string, and the longest match is selected.

The hash chains are searched starting with the most recent strings, to favor small distances and thus take advantage of the Huffman encoding. The hash chains are singly linked. There are no deletions from the hash chains, the algorithm simply discards matches that are too old.

To avoid a worst-case situation, very long hash chains are arbitrarily truncated at a certain length, determined by a runtime option (level parameter of deflateInit). So deflate() does not always find the longest possible match but generally finds a match which is long enough.

deflate() also defers the selection of matches with a lazy evaluation mechanism. After a match of length N has been found, deflate() searches for a longer match at the next input byte. If a longer match is found, the previous match is truncated to a length of one (thus producing a single literal byte) and the process of lazy evaluation begins again. Otherwise, the original match is kept, and the next match search is attempted only N steps later.

The lazy match evaluation is also subject to a runtime parameter. If the current match is long enough, deflate () reduces the search for a longer match, thus speeding up the whole process. If compression ratio is more important than speed, deflate() attempts a complete second search even if the first match is already long enough.

The lazy match evaluation is not performed for the fastest compression modes (level parameter 1 to 3). For these fast modes, new strings are inserted in the hash table only when no match was found, or when the match is not too long. This degrades the compression ratio but saves time since there are both fewer insertions and fewer searches.

D.2 Decompression algorithm (inflate)

D.2.1 Introduction

The real question is, given a Huffman tree, how to decode fast. The most important realization is that shorter codes are much more common than longer codes, so pay attention to decoding the short codes fast, and let the long codes take longer to decode.

inflate() sets up a first level table that covers some number of bits of input less than the length of longest code. It gets that many bits from the stream, and looks it up in the table. The table will tell if the next code is that many bits or less and how many, and if it is, it will tell the value, else it will point to the next level table for which inflate() grabs more bits and tries to decode a longer code.

How many bits to make the first lookup is a tradeoff between the time it takes to decode and the time it takes to build the table. If building the table took no time (and if you had infinite memory), then there would only be a first level table to cover all the way to the longest code. However, building the table ends up taking a lot longer for more bits since short codes are replicated many times in such a table. What inflate() does is simply to make the number of bits in the first table a variable, and set it for the maximum speed.

inflate() sends new trees relatively often, so it is possibly set for a smaller first level table than an application that has only one tree for all the data. For inflate, which has 286 possible codes for the literal/length tree, the size of the first table is nine bits. Also the distance trees have 30 possible values, and the size of the first table is six bits. Note that for each of those cases, the table ended up one bit longer than the ``average'' code length, i.e. the code length of an approximately flat code which would be a little more than eight bits for 286 symbols and a little less than five bits for 30 symbols. It would be interesting to see if optimizing the first level table for other applications gave values within a bit or two of the flat code size.

D.2.2 More details on the inflate table lookup

Ok, you want to know what this cleverly obfuscated inflate tree actually looks like. You are correct that it's not a Huffman tree. It is simply a lookup table for the first, let's say, nine bits of a Huffman symbol. The symbol could be as short as one bit or as long as 15 bits. If a particular symbol is shorter than nine bits, then that symbol's translation is duplicated in all those entries that start with that symbol's bits. For example, if the symbol is four bits, then it's duplicated 32 times in a nine-bit table. If a symbol is nine bits long, it appears in the table once.

If the symbol is longer than nine bits, then that entry in the table points to another similar table for the remaining bits. Again, there are duplicated entries as needed. The idea is that most of the time the symbol will be short and there will only be one table look up. (That's whole idea behind data compression in the first place.) For the less frequent long symbols, there will be two lookups. If you had a compression method with really long symbols, you could have as many levels of lookups as is efficient. For inflate, two is enough.

So a table entry either points to another table (in which case nine bits in the above example are gobbled), or it contains the translation for the symbol and the number of bits to gobble. Then you start again with the next ungobbled bit.

You may wonder: why not just have one lookup table for how ever many bits the longest symbol is? The reason is that if you do that, you end up spending more time filling in duplicate symbol entries than you do actually decoding. At least for deflate's output that generates new trees every several 10's of kbytes. You can imagine that filling in a 2^15 entry table for a 15-bit code would take too long if you're only decoding several thousand symbols. At the other extreme, you could make a new table for every bit in the code. In fact, that's essentially a Huffman tree. But then you spend two much time traversing the tree while decoding, even for short symbols.

So the number of bits for the first lookup table is a trade of the time to fill out the table vs. the time spent looking at the second level and above of the table.

Here is an example, scaled down:

The code being decoded, with 10 symbols, from 1 to 6 bits long:

```
A: 0
B: 10
C: 1100
D: 11010
E: 11011
F: 11100
G: 11101
H: 11110
I: 111110
J: 111111
```

Let's make the first table three bits long (eight entries):

```
000: A,1
001: A,1
010: A,1
011: A,1
100: B,2
101: B,2
110: -> table X (gobble 3 bits)
111: -> table Y (gobble 3 bits)
```

Each entry is what the bits decode to and how many bits that is, i.e. how many bits to gobble. Or the entry points to another table, with the number of bits to gobble implicit in the size of the table.

Table X is two bits long since the longest code starting with 110 is five bits long:

```
00: C,1
01: C,1
10: D,2
11: E,2
```

Table Y is three bits long since the longest code starting with 111 is six bits long:

```
000: F,2
001: F,2
010: G,2
011: G,2
100: H,2
101: H,2
110: I,3
111: J,3
```

So what we have here are three tables with a total of 20 entries that had to be constructed. That's compared to 64 entries for a single table. Or compared to 16 entries for a Huffman tree (six two entry tables and one four entry table). Assuming that the code ideally represents the probability of the symbols, it takes on the average 1.25 lookups per symbol. That's compared to one lookup for the single table, or 1.66 lookups per symbol for the Huffman tree.

There, I think that gives you a picture of what's going on. For inflate, the meaning of a particular symbol is often more than just a letter. It can be a byte (a "literal"), or it can be either a length or a distance which indicates a base value and a number of bits to fetch after the code that is added to the base value. Or it might be the special end-of-block code. The data structures created in inftrees.c try to encode all that information compactly in the tables.

Jean-loup Gailly **Mark Adler**
jloup@gzip.org **madler@alumni.caltech.edu**