Lempel-Ziv-Welch (LZW) Compression

One of the most common algorithms used in computer graphics is the Lempel-Ziv-Welch, or LZW, compression scheme. This lossless method of data compression is found in several image file formats, such as GIF and TIFF, and is also part of the V.42bis modem compression standard and PostScript Level 2.

In 1977, Abraham Lempel and Jakob Ziv created the first of what we now call the LZ family of substitutional compressors. The LZ77 compression algorithms are commonly found in text compression and archiving programs, such as compress, zoo, lha, pkzip, and arj. The LZ78 compression algorithms are more commonly used to compress binary data, such as bitmaps.

In 1984, while working for Unisys, Terry Welch modified the LZ78 compressor for implementation in high-performance disk controllers. The result was the LZW algorithm that is commonly found today.

LZW is a general compression algorithm capable of working on almost any type of data. It is generally fast in both compressing and decompressing data and does not require the use of floating-point operations. Also, because LZW writes compressed data as bytes and not as words, LZW-encoded output can be identical on both big-endian and little-endian systems, although you may still encounter bit order and fill order problems. LZW is referred to as a substitutional or dictionary-based encoding algorithm. The algorithm builds a data dictionary (also called a translation table or string table) of data occurring in an uncompressed data stream. Patterns of data (substrings) are identified in the data stream and are matched to entries in the dictionary. If the substring is not present in the dictionary, a code phrase is created based on the data content of the substring, and it is stored in the dictionary. The phrase is then written to the compressed output stream.

When a reoccurrence of a substring is identified in the data, the phrase of the substring already stored in the dictionary is written to the output. Because the phrase value has a physical size that is smaller than the substring it represents, data compression is achieved.
Decoding LZW data is the reverse of encoding. The decompressor reads a code from the encoded data stream and adds the code to the data dictionary if it is not already there. The code is then translated into the string it represents and is written to the uncompressed output stream.

LZW goes beyond most dictionary-based compressors in that it is not necessary to preserve the dictionary to decode the LZW data stream. This can save quite a bit of space when storing the LZW-encoded data. When compressing text files, LZW initializes the first 256 entries of the dictionary with the 8-bit ASCII character set (values 00h through FFh) as phrases. These phrases represent all possible single-byte values that may occur in the data stream, and all substrings are in turn built from these phrases. Because both LZW encoders and decoders begin with dictionaries initialized to these values, a decoder need not have the original dictionary and instead will build a duplicate dictionary as it decodes.

TIFF, among other file formats, applies the same method for graphic files. In TIFF, the pixel data is packed into bytes before being presented to LZW, so an LZW source byte might be a pixel value, part of a pixel value, or several pixel values, depending on the image's bit depth and number of color channels.

GIF requires each LZW input symbol to be a pixel value. Because GIF allows 1- to 8-bit deep images, there are between 2 and 256 LZW input symbols in GIF, and the LZW dictionary is initialized accordingly. It is irrelevant how the pixels might have been packed into storage originally; LZW will deal with them as a sequence of symbols.

The TIFF approach does not work very well for odd-size pixels, because packing the pixels into bytes creates byte sequences that do not match the original pixel sequences, and any patterns in the pixels are obscured. If pixel boundaries and byte boundaries agree (e.g., two 4-bit pixels per byte, or one 16-bit pixel every two bytes), then TIFF's method works well.

The GIF approach works better for odd-size bit depths, but it is difficult to extend it to more than eight bits per pixel because the LZW dictionary must become very large to achieve useful compression on large input alphabets.

**Noise Removal and Differencing**

LZW does a very good job of compressing image data with a wide variety of pixel depths. 1-, 8-, and 24-bit images all compress at least as well as they do using RLE encoding schemes. Noisy images, however, can significantly degrade the compression effectiveness of LZW. Removing noise from an image, usually by zeroing out one or two of the least significant bit planes of the image, is recommended to increase compression efficiency. In other words, if your data does not compress well in its present form, transform it to a different form that does compress well.

One method that is used to make data more "compressible" by reducing the amount of extraneous information in an image is called differencing. The idea is that unrelated data may be easily converted by an invertible transform into a form that can be more efficiently compressed by an encoding algorithm. Differencing accomplishes this using
the fact that adjacent pixels in many continuous-tone images vary only slightly in value. If we replace the value of a pixel with the difference between the pixel and the adjacent pixel, we will reduce the amount of information stored, without losing any data.

With 1-bit monochrome and 8-bit gray-scale images, the pixel values themselves are differenced. RGB pixels must have each of their color channels differenced separately, rather than the absolute value of the RGB pixels' differences (difference red from red, green from green, and blue from blue).

Differencing is usually applied in a horizontal plane across scan lines. In the following code example, the algorithm starts at the last pixel on the first scan line of the bitmap. The difference between the last two pixels in the line is calculated, and the last pixel is set to this value. The algorithm then moves to the next to last pixel and continues up the scan line and down the bitmap until finished, as shown in the following pseudo-code:

```c
/* Horizontally difference a bitmap */
for (Line = 0; Line < NumberOfLines; Line++)
  for (Pixel = NumberOfPixelsPerLine - 1; Pixel >= 1; Pixel--)
    Bitmap[Line][Pixel] = Bitmap[Line][Pixel-1];
```

Vertical and 2D differencing may also be accomplished in the same way. The type of differencing used will have varied effectiveness depending upon the content of the image. And, regardless of the method used, differenced images compress much more efficiently using LZW.

**Variations on the LZW Algorithm**

Several variations of the LZW algorithm increase its efficiency in some applications. One common variation uses index pointers that vary in length, usually starting at 9 bits and growing upward to 12 or 13 bits. When an index pointer of a particular length has been used up, another bit is tacked on to increase precision.

Another popular variation in LZW compressors involves constantly monitoring the compression process for any drop in efficiency. If a drop is noted, the least recently used (LRU) phrases in the dictionary are discarded to make room for new phrases, or the entire dictionary is discarded and rebuilt.

The LZW variant on the LZW compression method builds phrases by concatenating two phrases together, rather than by concatenating the current phrase and the next character of data. This causes a quicker buildup of longer strings at the cost of a more complex data dictionary.

LZW is a simple algorithm that is difficult to implement efficiently. Deciding when to discard phrases from the dictionary and even how to search the data dictionary during encoding (using a hashing or tree-based scheme) is necessary to improve efficiency and speed.

Variations on the standard LZW algorithm are more common than a programmer may realize. For example, the TIFF and GIF formats use the standard features of the LZW
algorithm, such as a Clear code (the indication to discard the string table), EOF code (end of file), and a 12-bit limit on encoded symbol width. However, GIF treats each pixel value as a separate input symbol. Therefore, the size of the input alphabet, the starting compressed-symbol width, and the values of the Clear and EOF codes will vary depending on the pixel depth of the image being compressed.

GIF also stores compressed codes with the least significant bit first, regardless of the native bit order of the machine on which the algorithm is implemented. When two codes appear in the same byte, the first code is in the lower bits. When a code crosses a byte boundary, its least significant bits appear in the earlier bytes.

TIFF's LZW variation always reads 8-bit input symbols from the uncompressed data regardless of the pixel depth. Each symbol may therefore contain one pixel, more than one pixel, or only part of a pixel, depending upon the depth of the pixels in the image. TIFF always stores compressed codes with the most significant bit first, the opposite of GIF. (Don't confuse the byte-order indicator in the TIFF header, or the value of the FillOrder tag, with the bit order of the LZW compressed data. In a TIFF file, LZW-compressed data is always stored most significant bit first.)

TIFF LZW also contains a bit of a kludge. Compressed code widths are required to be incremented one code sooner than is really necessary. For example, the compressor changes from 9-bit to 10-bit codes after adding code 511 to its table rather than waiting until code 512 is added, thus wasting one bit.

We understand that the explanation for this practice is that the LZW implementation supplied by Aldus in Revision 5.0 of the TIFF Developer's Toolkit contained this bug, although the TIFF 5.0 specification itself specified the LZW algorithm correctly. By the time the problem was identified (by Sam Leffler, the head of the TIFF Advisory Committee), too many applications existed that used the erroneous implementation, and there was no way to identify incorrectly encoded LZW data. The solution was simply to change the TIFF specification to require this "variation" in the TIFF algorithm, rather than to change the code, break all existing TIFF LZW applications, and regard all previously created TIFF 5.0 LZW images as incorrect and useless.