Adobe Photoshop® TIFF Technical Note 3

April 8, 2005

This document describes additions to the TIFF specification to improve support for floating point values.

Readers are advised to cross reference terms and discussions found in this document with the TIFF 6.0 specification (TIFF6.pdf), the TIFF Technical Note for Adobe PageMaker® 6.0 (TIFF-PM6.pdf), and the File Formats Specification for Adobe Photoshop® (Photoshop File Formats.pdf).

16 and 24 bit Floating Point Values

Introduction

This section describes the format of floating point data with BitsPerSample of 16 and 24.

Field:	**SampleFormat**
Tag:	339 (153.H)
Type:	SHORT
Count:	N = SamplesPerPixel
Value:	3 = IEEE Floating point data

Field:	**BitsPerSample**
Tag:	258 (102.H)
Type:	SHORT
Count:	N = SamplesPerPixel
Value:	16 or 24

16 and 24 bit floating point values may be used to minimize file size versus traditional 32 bit or 64 bit floating point values. The loss of range and precision may be acceptable for many imaging applications.

The 16 bit floating point format is designed to match the HALF data type used by OpenEXR and video graphics card vendors.

Implementation

16 bit floating point values

16 bit floating point numbers have 1 sign bit, 5 exponent bits (biased by 16), and 10 mantissa bits.

The interpretation of the sign, exponent and mantissa is analogous to IEEE-754 floating-point numbers. The 16 bit floating point format supports normalized and denormalized numbers, infinities and NANs (Not A Number).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
S	E	E	E	E	E	M	M	M	M	M	M	M	M	M	M

24 bit floating point values

24 bit floating point numbers have 1 sign bit, 7 exponent bits (biased by 64), and 16 mantissa bits.

The interpretation of the sign, exponent and mantissa is analogous to IEEE-754 floating-point numbers. The 24 bit floating point format supports normalized and denormalized numbers, infinities and NANs (Not A Number).

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
S	E	E	E	E	E	E	E	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M

References

http://www.openexr.com/documentation.html

Floating Point Predictor

Introduction

This section describes TIFF predictor type 3, a byte reordering of the image values followed by horizontal byte differencing used to improve compression of floating point image data.

Field:	**Predictor**
Tag:	317 (13D.H)
Type:	SHORT
Count:	1
Value:	3 = Floating Point predictor

Note that this predictor works well for both Deflate and LZW compression methods.
This predictor should only be applied to floating point data that has BitsPerSample equal to a multiple of 8.

Algorithm

This predicator makes use of the fact that many continuous-tone images don't vary much in pixel value from one pixel to the next. Additionally, in floating point data, the sign and exponent values will not change much from pixel to pixel, and the most significant bits of the mantissa will not change much from pixel to pixel, while the least significant bits of the mantissa may change a great deal (statistically equal to noise).

By rearranging the floating point data to group the sign and exponent data, the upper bytes of the mantissa and the lower bytes of the mantissa separately – we can then use a simple byte differencing predictor to reduce the apparent information content and allow for better compression by LZW or Deflate compressors. By putting the sign and exponent first (slowest changing), followed by most significant bytes of the mantissa (next slowest changing) then the least significant bytes (fastest changing, most noise-like), we achieve the best compression. This means that the predictor reorders the bytes into a semi-BigEndian order, and that the TIFF reader and writer should not change the byte order of the image data outside of the predictor. This predictor also preserves the ordering of interleaved color channels.

A simple C implementation might look like this:

```
/* ------------------------------------------------------------------------ */

void DecodeDeltaBytes( void *ptr, int32 cols, int32 channels )
       {
       int32 COL, CHAN;
       unsigned char *bytePtr = (unsigned char *)ptr;

       for (COL = 1; COL < cols; ++COL)
              {
              for ( CHAN = 0; CHAN < channels; ++CHAN )
                     bytePtr[ COL * channels + CHAN ] =
                                   bytePtr[ COL * channels + CHAN ]
                                   + bytePtr[ (COL - 1) * channels + CHAN ];
              }
       }

/* ------------------------------------------------------------------------ */

void EncodeDeltaBytes( void *ptr, int32 cols, int32 channels )
       {
       int32 COL, CHAN;
       unsigned char *bytePtr = (unsigned char *)ptr;
       for (COL = cols-1; COL > 0; --COL)
              {
              for (CHAN = 0; CHAN < channels; ++CHAN)
                     bytePtr[ COL * channels + CHAN ] =
                                   bytePtr[ COL * channels + CHAN ] -
                                   bytePtr[ (COL - 1) * channels + CHAN ];
              }
       }

/* ---------------------------------------------------------------------------- */

void EncodeFPDelta(  unsigned char *input,
                     unsigned char *output,
                     int32 cols,
                     int32 channels,
                     int32 bytesPerSample)
       {
       int32 COL, BYTE;

       // reorder the bytes into the output buffer
       // result is always in the same byte order (big endian, sort of)

       int32 rowIncrement = cols * channels;

       for (COL = 0; COL < rowIncrement; ++COL)
              {
              for (BYTE = 0; BYTE < bytesPerSample; ++BYTE)
                     {
#if BigEndian
                     output[ BYTE * rowIncrement + COL] =
                            input[ bytesPerSample * COL + BYTE];
#else
                     output[ (bytesPerSample-BYTE-1) * rowIncrement + COL] =
                            input[ bytesPerSample * COL + BYTE];
#endif
                     }
              }

       // do byte difference on output
       EncodeDeltaBytes ( output, cols*bytesPerSample, channels );

       // output data is now in semi-BigEndian byte order

       }

/* ---------------------------------------------------------------------------- */

void DecodeFPDelta(  unsigned char *input,
                     unsigned char *output,
                     int32 cols,
                     int32 channels,
                     int32 bytesPerSample)
       {
       int32 COL, BYTE;

       // undo byte difference on input
       DecodeDeltaBytes ( input, cols*bytesPerSample, channels );

       // reorder the semi-BigEndian bytes into the output buffer

       int32 rowIncrement = cols * channels;

       for (COL = 0; COL < rowIncrement; ++COL)
              {
              for (BYTE = 0; BYTE < bytesPerSample; ++BYTE)
                     {
#if BigEndian
                     output[ bytes * COL + BYTE ] =
                            input[ BYTE * rowIncrement + COL ];
#else
                    output[ bytes * COL + BYTE] =
                           input[ (bytesPerSample-BYTE-1) * rowIncrement + COL ];
#endif
                    }
             }

       // output data is now in native byte order

       }


/* -------------------------------------------------------------------------- */
```