in: *Fallout and Fallout 2 file formats*

MVE File Format

Contents

Introduction

Throughout this document, a "word" is a 16-bit value. All values are little-endian, unless otherwise specified.

Structure

The high-level format of an Interplay MVE file is a small header, followed by variable-sized stream chunks; each stream chunk consists of a word giving the length of the chunk, and another giving the type, followed by a stream of 1 or more stream opcodes, which consist of a two-word count for the length of the stream opcode, a single byte for

the type, a single byte (which I believe to be a "version" field, to allow backwards compatibility), and then variable data depending on the type of opcode.

So, just to make sure that's clear, we've got the header, followed by a 2-level hierarchical structure:

```
        ||            CHUNK1          ||            CHUNK2          ||
```

header || op1 || op2 || op3 || op4 || op1 || op2 || op3 || op4 || ...

Header

The Header of an Interplay MVE file must start with the sequence of bytes:

```
    "Interplay MVE File\x1A\0"
```

where \x1A represents ASCII 0x1a (^Z), the old DOS end-of-file character, and \0 represents ASCII 0x00 (NUL). The reason for this is then, under DOS, if you do:

C:\>TYPE foobar.MVE

you'll see

Interplay MVE File

After these 20 bytes, there are 6 more bytes, which I believe are either a file format version, a "magic" number, or were, once upon a time, parameters. In modern Interplay games, these parameters appear to need to be hard-coded. They take the form of 3 words:

001a 0100 1133

Immediately following this are the chunks.

Chunks

Each chunk consists of a word giving the total length of the data contained in the chunk, and another word which represents the type of the chunk. After these four bytes (which are NOT included in the chunk length), comes the chunk data. The types of chunks I know about (i.e. which are used in BG/BG2 movies that I've examined; the chunk types are not used at all in the movie playback code in BG/BG2) are:

0000: initialize audio 0001: audio only chunk (or maybe only used for audio pre-buffering) 0002: initialize video 0003: video chunk (usually includes audio. possibly always includes audio) 0004: shutdown chunk 0005: end chunk

I don't know why an "end chunk" is needed, since the "shutdown chunk" seems to do that job nicely. The "end chunk" appears to contain no opcodes.

Opcodes

The opcodes I've observed range from 0x00 to 0x15. Of these, the current code used in BG/BG2 uses only from 0x00 through 0x11, so any guesses as to the function of 0x12 through 0x15 would be merely speculation. I have no idea what any of these opcodes are used for. But, again, since they are unused in the BG/BG2 movie player code, they are unnecessary for playback.

Opcode 0x00: End Of Stream

```
    No data associated with this.  When this opcode is seen, the playback of
    the movie stops immediately.
```

Opcode 0x01: End Of Chunk

```
    All this opcode does in theory is to terminate a chunk.  In practice, it
    signals the code to fetch and decode the next chunk.
```

Opcode 0x02: Create Timer

```
    DWORD   timer rate
    WORD    timer subdivision
```

```
    This sets up the timer that drives the animation.  Basically, every time
```

```
the timer expires, it should be starting to pump out the next frame in
order to keep up with the desired frame rate.
```

```
The normal values I've seen here are 0x2095 for the timer rate (8341), and
8 for the timer subdivision.  What this means in practice is that every
8*8341 (=66728) microseconds, it should be ready to send out the next
frame.  So...  10000000/66728 == 14.9 frames per second typically.  The
exact purpose for the timer subdivision is unclear to me, but it may be an
artifact of earlier methods of timer handling, since some of the code here
appears to possibly even date back to the DOS days.
```

Opcode 0x03: Initialize Audio Buffers

```
version 0:
    WORD    (unknown)
    WORD    flags
    WORD    sample rate
    WORD    min buffer length
```

```
version 1:
    WORD    (unknown)
    WORD    flags
    WORD    sample rate
    DWORD   min buffer length
```

```
The flags recognized, as of version 0 are:
    bit 0: 0=mono,  1=stereo
    bit 1: 0=8-bit, 1=16-bit
```

```
The flags recognized, as of version 1 are:
    bit 0: 0=mono,  1=stereo
    bit 1: 0=8-bit, 1=16-bit
    bit 2: 0=uncompressed, 1=compressed
```

```
Only uncompressed audio is supported in the version 0 opcode.  I _think_
the other 13 bits (14 bits for ver. 0) here may be garbage.  Whatever they
are, they are not apparently used for the playback engine inside BG/BG2.
```

```
The sample rate is the standard sampling rate in kHz; typically 22050 in
the BG movies.  Buffer length is the size (in bytes) of the buffer that
needs to be allocated for the audio.  (I don't remember if this is the
_total_ audio buffer size needed, or if this number is a "per-channel"
number that needs to be doubled for "stereo" audio streams.)  They use 1.5
times the original buffer size in order to have a "safety zone".
```

```
I will cover the format of the compressed audio data in the audio data
opcode section.
```

Opcode 0x04: Start/Stop Audio

```
This seems to start and/or stop the audio playback.  This opcode contains
no data.
```

Opcode 0x05: Initialize Video Buffer(s)

```
version 0:
    WORD    width
    WORD    height
```

```
version 1:
    WORD    width
    WORD    height
    WORD    ?count?
```

```
version 2:
    WORD    width
    WORD    height
    WORD    ?count?
    WORD    true-color
```

```
Width is the width of the buffer to allocate, and height is the height.
```

Both are given in terms of pixels. Now, the count appears to be used to over-allocate the video buffer. To compute the size to allocate for the video buffer, they take 2 bytes per pixel, and multiply by the height and the width, and then multiply by the count. If scan-line doubling is enabled (which it is not in BG/BG2), it then divides this value by two, on the assumption that there is only enough data for half the resolution. Anyway, the over-allocation may be used to create a larger movie area and pan smoothly or something. I haven't seen a way to use the overallocation with the format details that I've discerned, but the video coding is particularly hairy, as the decoder relies on self-modifying x86 code to function. Yick. Anyway, I'm still in the process of looking for a file that uses over-allocation so that I can figure out exactly why it is used and what it is used for. (Again, this feature doesn't appear to be widely used in the sampling of BG/BG2 movies that I've examined.) Note that an alternate possibility for the usage of the over-allocated space is as scratch space. This possbility will be addressed in the (voluminous!) documentation for opcode 0x11.

Opcode 0x06: unknown

```
4 bytes apparently unused?
WORD    unknown
WORD    unknown
WORD    unknown
WORD    flip back buffer? (0=no, 1=yes)
bytes   unknown
```

I haven't seen this opcode used in any BG/BG2 movies; however, this may be used for the panning or some clever usage of the over-allocation mentioned in opcode 0x05. If "flip back buffer?" has bit 0 set, it will flip the two allocated buffers before it does whatever it is that it does.

The "whatever it does" appears to be characterized by bulk memory moves, which makes it possible that it _is_ used in conjunction with the over-allocated video buffers.

No "version" check is made for this opcode, which makes me suspect that there is only 1 supported version of this opcode. (version 0, presumably)

Opcode 0x07: Send Buffer to Display

```
version 0:
    WORD    palette start
    WORD    palette count

version 1:
    WORD    palette start
    WORD    palette count
    WORD    ???
```

palette start is the index of the first palette entry to be installed before copying from the current back buffer to the display. palette count is the number of palette entries to be installed. As for the mysterious other flag... I am still unclear on its usage. Again, I've seen no example of its usage yet.

Opcode 0x08: Audio Frame (data)

Opcode 0x09: Audio Frame (silence)

```
WORD    seq-index
WORD    stream-mask
WORD    stream-len
data    audio data (only for Opcode 0x08)
```

seq-index is the sequential index of this audio chunk, numbered from 0000 (0000 being the first chunk in the audio file). stream-mask works as follows:

a given mve file can contain up to 16 parallel audio streams. Presumably this is for alternate languages. The stream-mask determines which stream(s) a given audio chunk belongs to. So, if bit 0 is set in the stream-mask, it belongs to stream 0. Typically, in the English language version of BG, I've seen the sole audio frame (opcode 8) having bit 0 set,

and the next silent frame having all 15 of the other bits set.

So, just to make this clear, what we see is:
opcode 8: idx=0 mask=0x0001 len=0x16d8 data=...
opcode 9: idx=0 mask=0xfffe len=0x16d8

These audio chunks appear to always come in pairs.

stream-len is the total number of samples in the chunk.

Now, the audio data, if it is compressed, is compressed by first applying a delta coding. These deltas are then quantized to a particular set of codes and are stored as 8-bit offsets into this quantized table. It is important to track the last "delta" for each channel from audio chunk to audio chunk (with the initial value being the first word in the first chunk). For stereo data, the two channels are compressed separately; the first TWO words in the first chunk, then are the left and right channel's initial bias, respectively, and with the samples being interleaved L R L R. (i.e. every other byte being Left channel). The particular table used for the delta coding is:

```
       0,      1,      2,      3,      4,      5,      6,      7,      8,      9,     10,     11,     12,
13,     14,     15,
      16,     17,     18,     19,     20,     21,     22,     23,     24,     25,     26,     27,     28,
29,     30,     31,
      32,     33,     34,     35,     36,     37,     38,     39,     40,     41,     42,     43,     47,
51,     56,     61,
      66,     72,     79,     86,     94,    102,    112,    122,    133,    145,    158,    173,    189,
206,    225,    245,
     267,    292,    318,    348,    379,    414,    452,    493,    538,    587,    640,    699,    763,
832,    908,    991,
    1081,   1180,   1288,   1405,   1534,   1673,   1826,   1993,   2175,   2373,   2590,   2826,   3084,
3365,   3672,   4008,
    4373,   4772,   5208,   5683,   6202,   6767,   7385,   8059,   8794,   9597,  10472,  11428,  12471,
13609,  14851,  16206,
   17685,  19298,  21060,  22981,  25078,  27367,  29864,  32589, -29973, -26728, -23186, -19322, -15105,
-10503,  -5481,     -1,
       1,      1,   5481,  10503,  15105,  19322,  23186,  26728,  29973, -32589, -29864, -27367, -25078,
-22981, -21060, -19298,
  -17685, -16206, -14851, -13609, -12471, -11428, -10472,  -9597,  -8794,  -8059,  -7385,  -6767,  -6202,
-5683,  -5208,  -4772,
   -4373,  -4008,  -3672,  -3365,  -3084,  -2826,  -2590,  -2373,  -2175,  -1993,  -1826,  -1673,  -1534,
-1405,  -1288,  -1180,
   -1081,   -991,   -908,   -832,   -763,   -699,   -640,   -587,   -538,   -493,   -452,   -414,   -379,
-348,   -318,   -292,
    -267,   -245,   -225,   -206,   -189,   -173,   -158,   -145,   -133,   -122,   -112,   -102,    -94,
-86,    -79,    -72,
     -66,    -61,    -56,    -51,    -47,    -43,    -42,    -41,    -40,    -39,    -38,    -37,    -36,
-35,    -34,    -33,
     -32,    -31,    -30,    -29,    -28,    -27,    -26,    -25,    -24,    -23,    -22,    -21,    -20,
-19,    -18,    -17,
     -16,    -15,    -14,    -13,    -12,    -11,    -10,     -9,     -8,     -7,     -6,     -5,     -4,
-3,     -2,     -1
```

The included code for decompressing audio in this format should make this a little clearer. (It handles only the stereo case at present.)

Opcode 0xa: Initialize Video Mode

```
WORD    X-resolution
WORD    Y-resolution
WORD    flags
```

The usage of the flags field appears to be largely historical. Perhaps with the introduction of DirectX as the underlying medium, rather than the direct graphics hardware manipulation that was, apparently, used in an earlier version, this field is unnecessary. (In fact, in BG, this entire opcode turns into a no-op.) (Note, for the curious: BG actually contains assembly code to do register level manipulation of VGA hardware. Not enough to actually really do much, but it's there, anyway.

Opcode 0xb: Create Gradient

```
BYTE    baseRB
BYTE    numR_RB
```

```
BYTE    numB_RB
BYTE    baseRG
BYTE    numR_RG
BYTE    numG_RG
```

I haven't seen this particular opcode used, but it is clear that it generates a gradient palette. It appears that it will generate two gradient palettes, if both count0 and count1 are non-zero. The first gradient is a pure red-blue gradient, and the second a pure red-green gradient. It appears to be designed for EGA/VGA hardware, since it uses 0-63 as the maximum range for a component within a color. The red component of each gradient moves linearly from 0 to 63 within numR_RB (resp. numR_RG) rows, and the blue or green component moves linearly from 0 to 39 within numB_RB (resp numG_RG) columns for the blue or green gradient respectively.

The colors are ordered in row-major ordering, starting at the 'base'th entry. So, if you had:

```
baseRB=12
numR_RB=5
numB_RB=4
```

You'd get 20 colors starting at index #12, with a row-major gradient. Specifically you'd see:

```
( 0,0,0) ( 0,0,13) ( 0,0,26) ( 0,0,39)  ; 12...15
(15,0,0) (15,0,13) (15,0,26) (15,0,39)  ; 16...19
(31,0,0) (31,0,13) (31,0,26) (31,0,39)  ; 20...23
(47,0,0) (47,0,13) (47,0,26) (47,0,39)  ; 24...27
(63,0,0) (63,0,13) (63,0,26) (63,0,39)  ; 28...31
```

Opcode 0xc: Set Palette

```
WORD    pal-start
WORD    pal-count
data    pal-data
```

pal-start indicates the first palette entry to fill
pal-count indicates the number of palette entries to fill
pal-data is the palette data, 3 bytes per palette entry, packed as:

```
RGBRGBRGB
```

Opcode 0xd: Set Palette Entries Compressed

```
data    compressed palette data
```

This doesn't appear to have been used in the BG movies. This is a series of 32 entries of the following form:

```
<byte> <RGB> <RGB> ... <RGB>
```

Where there are between 0 and 8 <RGB> values, taking 3 bytes apiece.

Each bit in the preceding byte determines which of the 8 palette entries have an RGB value stored for them, with the least significant bit corresponding to the first entry in the group of 8. So, in order to set only the 240th entry in the palette, the data would be:

```
00              ;; 00-07
00              ;; 08-0f
00              ;; 10-17
00              ;; 18-1f
00              ;; 20-27
00              ;; 28-2f
00              ;; 30-37
00              ;; 38-3f
00              ;; 40-47
00              ;; 48-4f
```

```
00              ;; 50-57
00              ;; 58-5f
00              ;; 60-67
00              ;; 68-6f
00              ;; 70-77
00              ;; 78-7f
00              ;; 80-87
00              ;; 88-8f
00              ;; 90-97
00              ;; 98-9f
00              ;; a0-a7
00              ;; a8-af
00              ;; b0-b7
00              ;; b8-bf
00              ;; c0-c7
00              ;; c8-cf
00              ;; d0-d7
00              ;; d8-df
01 rr gg bb     ;; e0-e7
00              ;; e8-ef
00              ;; f0-f7
00              ;; f8-ff
```

```
Giving:
    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
    00 00 00 00 00 00 00 00 00 00 00 00 01 rr gg bb
    00 00 00
```

```
35 bytes of data instead of 768 to store the whole palette.
```

Opcode 0xe: ???

```
data    unknown length
```

```
I haven't encountered this value before.  What it does is set a pointer to
an array of words used during decoding of data using the 0x10 opcode,
which I have also not encountered.
```

```
I'm still working on figuring out the use of this opcode and the 0x10
opcode, but they don't appear to be used in the BG movies, again.  See my
comments at opcode 0x10 for more details.
```

Opcode 0xf: Set Decoding Map

```
data    decoding map
```

```
The decoding map is a particular data block used in the decoding of video
frames, as encoded via opcode 0x11.  I'll cover it in detail when I get to
opcode 0x11.
```

Opcode 0x10: ???

```
This is another means of storing video data.  I haven't seen it used yet,
and am still sorting through the details.  This seems to be tied in with
the issue of multiple pages of video memory, as with opcode 6 and the
"count" field of opcode 5.
```

```
Note that this opcode makes use of 3 (!) data streams, as opposed to 2 for
0x11.  Even so, it appears to be a much simpler encoding.  The data
streams used for this are the most recent 0xe opcode data stream, the most
recent 0xf opcode data stream, and this opcode's data stream.
```

```
There appears to be verbatim pixel data encoded in the 0x10 stream, but
the which pixels have been stored, among other things, is determined by
the other streams.  It also appears that in this stream all pixel
manipulation is done in 8-pixel wide and 8-pixel tall units.  This is
set-up to loop first over each column, then over each row, then finally
over each page:
```

```
foreach page
    foreach row
        foreach col
            decode opcode data
```

If I can find an example of one of these files to mess around with, I will complete my analysis of this opcode.

Opcode 0x11: Video Data

Ok, this is the big killer opcode. The way this works is as follows:

First, the data is processed in 8x8 pixel blocks. There are 4 bits associated with each block giving the particular encoding to use for that block, giving a total of 16 possible encodings for a given block. These 4-bit pieces come from the most recent 0xf opcode data stream. They all appear to be used (or at least supported by the player). So, I'll go over the encodings for each of the 16 encoding types. The rendering process keeps track of the most recent frame in a separate buffer, and uses this double-buffering technique in the common way for animation. The current frame's data is used in the construction of the next frame. In the following description, "current frame" will refer to the most recently displayed frame, and "new frame" will refer to the frame currently being constructed for display. "map stream" will refer to the data grabbed from the 0xf Opcode data, and "data stream" will refer to the data grabbed from the 0x11 opcode data.

Encoding 0x0:
    Block is copied from corresponding block from current frame.
    (i.e. this block is unchanged).

Encoding 0x1:
    Block is unmodified. This appears to mean that it has the same
    value it had 2 frames ago, but the net effect is that nothing is
    done to this block of 8x8 pixels.

Encoding 0x2:
    Block is copied from nearby (below and to the right) within the
    new frame. The offset within the buffer from which to grab the
    patch of 8 pixels is given by grabbing a byte B from the data
    stream, which is broken into a positive x and y offset according
    to the following mapping:

```
        if B < 56:
            x = 8 + (B % 7)
            y = B / 7
        else
            x = -14 + ((B - 56) % 29)
            y =   8 + ((B - 56) / 29)
```

    (where % is the 'modulo' operator)

    If you draw the region this represents, you'll see it looks like:

```
                oooooooo#######
                oooooooo#######
                oooooooo#######
                oooooooo#######
                oooooooo#######
                oooooooo#######
                oooooooo#######
                oooooooo#######
 ############################
 ############################
 ############################
 ############################
 ############################
 ############################
 ############################
 ########################
```

Where 'o' are the pixels in the destination frame, and # are the locations where the source frame could start.

Encoding 0x3:
This is the same as encoding 0x2, with the exception that the x and y offsets are negated giving:

```
if B < 56:
    x = -(8 + (B % 7))
    y = -(B / 7)
else
    x = -(-14 + ((B - 56) % 29))
    y = -(  8 + ((B - 56) / 29))
```

(where % is the 'modulo' operator)

If you draw the region this represents, you'll see it looks like:

```
   ############################
 ##############################
 ##############################
 ##############################
 ##############################
 ##############################
 ##############################
 #######oooooooo
 #######oooooooo
 #######oooooooo
 #######oooooooo
 #######oooooooo
 #######oooooooo
 #######oooooooo
 #######oooooooo
```

Encoding 0x4:
Similar to 0x2 and 0x3, except this method copies from the "current" frame, rather than the "new" frame, and instead of the lopsided mapping they use, this one uses one which is symmetric and centered around the top-left corner of the block. This uses only 1 byte still, though, so the range is decreased, since we have to encode all directions in a single byte. The byte we pull from the data stream, I'll call B. Call the highest 4 bits of B BH and the lowest 4 bytes BL. Then the offset from which to copy the data is:

```
x = -8 + BL
y = -8 + BH
```

Encoding 0x5:
Similar to 0x4, but instead of one byte for the offset, this uses two bytes to encode a larger range, the first being the x offset as a signed 8-bit value, and the second being the y offset as a signed 8-bit value.

Encoding 0x6:
I can't figure out how any file containing a block of this type could still be playable, since it appears that it would leave the internal bookkeeping in an inconsistent state in the BG player code. Ahh, well. Perhaps it was a bug in the BG player code that just didn't happen to be exposed by any of the included movies. Anyway, this skips the next two blocks, doing nothing to them. Note that if you've reached the end of a row, this means going on to the next row.

Encoding 0x7:
Ok, here's where it starts to get really...interesting. This is, incidentally, the part where they started using self-modifying code. So, most of the following encodings are "patterned" blocks, where we are given a number of pixel values and then bitmapped values to specify which pixel values belong to which squares. For this encoding, we are given the following in the data stream:

```
P0 P1
```

These are pixel values (i.e. 8-bit indices into the palette). If P0 <= P1, we then get 8 more bytes from the data stream, one for

each row in the block:

```
    B0 B1 B2 B3 B4 B5 B6 B7
```

For each row, the rightmost pixel is represented by the low-order bit, and the leftmost by the high-order bit. Use your imagination in between. If a bit is set, the pixel value is P1 and if it is unset, the pixel value is P0.

If, on the other hand, P0 > P1, we get two more bytes from the data stream:

```
    B0 B1
```

Each of these bytes contains a 4-bit pattern. This pattern works exactly like the pattern above with 8 bytes, except each bit represents a 2x2 pixel region.

So, for example, if we had:

```
    11 22 ff 81 81 81 81 81 81 ff
```

This would represent the following layout:

```
    22 22 22 22 22 22 22 22     ; ff == 11111111
    22 11 11 11 11 11 11 22     ; 81 == 10000001
    22 11 11 11 11 11 11 22     ; ..
    22 11 11 11 11 11 11 22
    22 11 11 11 11 11 11 22
    22 11 11 11 11 11 11 22
    22 11 11 11 11 11 11 22     ; 81 == 10000001
    22 22 22 22 22 22 22 22     ; ff == 11111111
```

If, on the other hand, we had:

```
    22 11 ff 81
```

The output would be:

```
    22 22 22 22 22 22 22 22     ; f == 1 1 1 1
    22 22 22 22 22 22 22 22     ;
    22 22 22 22 22 22 22 22     ; f == 1 1 1 1
    22 22 22 22 22 22 22 22     ;
    22 11 11 11 11 11 11 11     ; 8 == 1 0 0 0
    22 11 11 11 11 11 11 11     ;
    11 11 11 11 11 11 11 22     ; 1 == 0 0 0 1
    11 11 11 11 11 11 11 22     ;
```

Encoding 0x8:

Ok, this one is basically like encoding 0x7, only more complicated. Again, we start out by getting two bytes on the data stream:

```
    P0 P1
```

if P0 <= P1 then we get the following from the data stream:

```
          B0 B1
    P2 P3 B2 B3
    P4 P5 B4 B5
    P6 P7 B6 B7
```

P0 P1 and B0 B1 are used for the top-left corner, P2 P3 B2 B3 for the bottom-left corner, P4 P5 B4 B5 for the top-right, P6 P7 B6 B7 for the bottom-right. (So, each codes for a 4x4 pixel array.) Since we have 16 bits in B0 B1, there is one bit for each pixel in the array. The convention for the bit-mapping is, again, left to right and top to bottom.

So, basically, the top-left quarter of the block is an arbitrary pattern with 2 pixels, the bottom-left a different arbitrary pattern with 2 different pixels, and so on. I'll go through a few examples of this after I discuss the other forms for the data in this encoding.

if P0 > P1 then we get 10 more bytes from the data stream:

```
B0 B1 B2 B3 P2 P3 B4 B5 B6 B7
```

Now, if P2 <= P3, then [P0 P1 B0 B1 B2 B3] represent the left half of the block and [P2 P3 B4 B5 B6 B7] represent the right half.

If P2 > P3, [P0 P1 B0 B1 B2 B3] represent the top half of the block and [P2 P3 B4 B5 B6 B7] represent the bottom half.

In these last two cases, each bit represents a 1x1 pixel. Just to work through an example of each case:

```
00 22 f9 9f 11 33 cc 33 44 55 aa 55 66 77 01 ef
```

```
22 22 22 22 | 33 33 11 11     ; f = 1111, c = 1100
22 00 00 22 | 33 33 11 11     ; 9 = 1001, c = 1100
22 00 00 22 | 11 11 33 33     ; 9 = 1001, 3 = 0011
22 22 22 22 | 11 11 33 33     ; f = 1111, 3 = 0011
------------+------------
55 44 55 44 | 66 66 66 66     ; a = 1010, 0 = 0000
55 44 55 44 | 66 66 66 77     ; a = 1010, 1 = 0001
44 55 44 55 | 77 77 77 66     ; 5 = 0101, e = 1110
44 55 44 55 | 77 77 77 77     ; 5 = 0101, f = 1111
```

I've added a dividing line in the above to clearly delineate the quadrants.

Now, for a horizontally split block:

```
22 00 01 37 f7 31 11 66 8c e6 73 31
```

```
22 22 22 22 66 11 11 11
22 22 22 00 66 66 11 11
22 22 00 00 66 66 66 11
22 00 00 00 11 66 66 11
00 00 00 00 11 66 66 66
22 00 00 00 11 11 66 66
22 22 00 00 11 11 66 66
22 22 22 00 11 11 11 66
```

Finally, for a vertically split block:

```
22 00 cc 66 33 19 66 11 18 24 42 81
```

```
00 00 22 22 00 00 22 22
22 00 00 22 22 00 00 22
22 22 00 00 22 22 00 00
22 22 22 00 00 22 22 00
66 66 66 11 11 66 66 66
66 66 11 66 66 11 66 66
66 11 66 66 66 66 11 66
11 66 66 66 66 66 66 11
```

Encoding 0x9:

Similar to the previous 2 encodings, only more complicated. And it will get worse before it gets better. No longer are we dealing with patterns over two pixel values. Now we are dealing with patterns over 4 pixel values with 2 bits assigned to each pixel (or block of pixels).

So, first on the data stream are our 4 pixel values:

```
P0 P1 P2 P3
```

Now, if P0 <= P1 AND P2 <= P3, we get 16 bytes of pattern, each 2 bits representing a 1x1 pixel (00=P0, 01=P1, 10=P2, 11=P3). The ordering is again left to right and top to bottom. The most significant bits represent the left side at the top, and so on.

If P0 <= P1 AND P2 > P3, we get 4 bytes of pattern, each 2 bits representing a 2x2 pixel. Ordering is left to right and top to bottom.

if P0 > P1 AND P2 <= P3, we get 8 bytes of pattern, each 2 bits representing a 2x1 pixel (i.e. 2 pixels wide, and 1 high).

if P0 > P1 AND P2 > P3, we get 8 bytes of pattern, each 2 bits representing a 1x2 pixel (i.e. 1 pixel wide, and 2 high).

Encoding 0xa:
Similar to the previous, only a little more complicated.
We are still dealing with patterns over 4 pixel values with 2 bits assigned to each pixel (or block of pixels).

So, first on the data stream are our 4 pixel values:

```
P0 P1 P2 P3
```

Now, if P0 <= P1, the block is divided into 4 quadrants, ordered (as with opcode 0x8) TL, BL, TR, BR. In this case the next data in the data stream should be:

```
                B0  B1  B2  B3
P4  P5  P6  P7  B4  B5  B6  B7
P8  P9  P10 P11 B8  B9  B10 B11
P12 P13 P14 P15 B12 B13 B14 B15
```

Each 2 bits represent a 1x1 pixel (00=P0, 01=P1, 10=P2, 11=P3).
The ordering is again left to right and top to bottom. The most significant bits represent the left side at the top, and so on.

If P0 > P1 then the next data on the data stream is:

```
             B0 B1 B2  B3  B4  B5  B6  B7
P4 P5 P6 P7 B8 B9 B10 B11 B12 B13 B14 B15
```

Now, in this case, if P4 <= P5,
[P0 P1 P2 P3 B0 B1 B2 B3 B4 B5 B6 B7] represent the left half of the block and the other bytes represent the right half. If P4 > P5, then [P0 P1 P2 P3 B0 B1 B2 B3 B4 B5 B6 B7] represent the top half of the block and the other bytes represent the bottom half.

Encoding 0xb:
In this encoding we get raw pixel data in the data stream -- 64 bytes of pixel data. 1 byte for each pixel, and in the standard order (l->r, t->b).

Encoding 0xc:
In this encoding we get raw pixel data in the data stream -- 16 bytes of pixel data. 1 byte for each block of 2x2 pixels, and in the standard order (l->r, t->b).

```
Encoding 0xd:
    In this encoding we get raw pixel data in the data stream -- 4
    bytes of pixel data.  1 byte for each block of 4x4 pixels, and in
    the standard order (l->r, t->b).
```

```
Encoding 0xe:
    This encoding represents a solid frame.  We get 1 byte of pixel
    data from the data stream.
```

```
Encoding 0xf:
    This encoding represents a "dithered" frame, which is
    checkerboarded with alternate pixels of two colors.  We get 2
    bytes of pixel data from the data stream, and these bytes are
    alternated:

        P0 P1 P0 P1 P0 P1 P0 P1
        P1 P0 P1 P0 P1 P0 P1 P0
        ...
        P0 P1 P0 P1 P0 P1 P0 P1
        P1 P0 P1 P0 P1 P0 P1 P0
```

Opcode 0x12: Not used

```
        Not observed in BG movies, and not used by the player
```

Opcode 0x13: Unknown

```
        Used in the BG movies, but not used by the player.
        Appears to always(?) have 0x84 bytes of data.  This is a
        recurrent opcode, appearing in most, if not all video chunks.
```

Opcode 0x14: Not used

```
        Not observed in BG movies, and not used by the player
```

Opcode 0x15: Unknown

```
        Used in the BG movies, but not used by the player.
        Appears to always(?) have 4 bytes of data.  This one appears
        in the "video initialization chunk"
```

Typical chunk formations

Audio chunks

```
opcode 0x8
opcode 0x9
```

Audio Init Chunk (type 0):

```
opcode 0x3
```

Video chunks

```
opcode 0x2
opcode 0xf
opcode 0x8
opcode 0x9
opcode 0x11
opcode 0x13
opcode 0x4
opcode 0x7
```

Video Init Chunk (type 2):

```
opcode 0xa
opcode 0x5
opcode 0xc
opcode 0x15
```

Author

The author of this documentation is unknown at this point. However, the documentation (and additional source code for an MVE player from presumably the same author) is available for download here .