- All in one Fallout 1-2 code documentation

- Simplified and Edited by Roby Lla

############################################

FRM file

- Purpose

Unpaletted 256-colour image files containing either one or several frames. The palette for these files is external.

All values within a FRM file are stored in Big Endian (Motorola) format, storing the most-significant (largest digit place)

digit closest to the number's starting address.

- General info

FRM files are used for storing images in both Fallout 1 and 2. Including all animations, except those for splash screens and movies.

FRM files use an index colour model to store the image data, each pixel is represented by an offset into a palette of colours.

A colour index of 0 equals to a transparent pixel.

The palette used generally is color.pal, FRM files can also use their own palette.

These custom palettes must have the same name as the FRM file.

A FRM file may contain image data for one or all 6 rotations used in Fallout.

If a FRM file has the extension .fr[0-5] instead of .FRM or .frm

then that file contains image data for the orientation given by the last digit in the extension,

otherwise the file contains only orientation 0 or all 6 rotations.

The FRM file also contains info to correctly align or center the image data.

Each frame has an offset from the previous frame which must be applied before rendering.

This offset ensures that in an animation sequence each frame is correctly aligned.

The center of the image data is the center of the bottom edge of the frame.

To find the position of the top left corner of the frame (needed for rendering):

left = center\_x - (frame\_width / 2)

top = center\_y - frame\_height

- FRM Header & data

- Palette info

Frame data comes in the form of byte colour indexes to a 256-color palette.

DAT file

DAT1 (Fallout 1’s DAT format) has no documentation, although there are programs to open it.

DAT2 files work like ZIP files, they can contain any type of file compressed or not.

MSG file

MSG files are **indexed text** files. They contain in-game messages, character and item dialogues, and are located in the text\english\game and text\english\dialog folders, respectively.

- Structure

Each indexed text line has the format: "{}{}{}"

* The first set of curly braces "{}" indicate the line index (number) in decimal format
* The second set of braces (optionally) indicates the name of an ACM sound file
* The third set of braces contain the text to be displayed (dialogue box or observation window)

- Example

Dialogue Line Number

|

|

+----+ ACM sound file name

| |

| |

| +----+ Displayed message

| | (dialogue box or observation window)

| | |

| | |

| | +----+

| | |

v v v

{100} {deadbrahmin001} {You see a horribly mutilated, two-headed cow.

Hamburgers, anyone?}

- Notes

* The length of the lines between the braces cannot exceed 1024 characters
* The length of a single word cannot exceed 53 characters (max width of the in-game dialog window), the rest is not displayed
* Linebreaks inside the braces are allowed however, a trailing portion of the string *must* contain at least one non-space character before the last brace. Otherwise, the game will crash.
* Linebreaks are **ignored** by game engine. That is, if you do
* At the end of the file there is a blank line (with no characters in it)
* The game only looks for text within braces, which allows leaving comments in between the indexed lines
* Comments are usually preceded by a "#", or a pound symbol, unlike Fallout's scripting language, *SSL,* which uses two slashes "//", although some modding tools require comments.
* Sound files are *generally* only used in dialog with 'talking heads', characters with specific voices and special sprites.
* Index is supposed to be unique. If it's not unique, only the latest string with matching index is used by the game.

Note: Fallout Tactics uses .txt files for every text in the game that requires one.

ACM file

This description uses the following integer types:

DWORD 32-bit signed integer

WORD 16-bit signed integer

Also C-like declarations and expressions are used.

ACM file consists of two main parts:

1. short header

2. ACM-Stream, containing packed information

1. Header.

------------

This table describes the meaning of header's fields:

# of bits | Value | Description

-----------+---------------------+-----------------

24 | 0x97 0x28 0x03 | ACM-signature1

8 | 0x01 | ACM-signature2

32 | count of samples | a sample is 16 bit

16 | # of channels | 1-mono, 2-stereo (actually not used)

16 | frequency (bitrate) | 22050 in all known ACMs (actually not used)

4 | packAttrs | (see below)

12 | packAttrs2 |

- Explanation

First 4 bytes of every ACM-file contain 0x01032897, this is kind of

signature and its exact meaning is opaque.

Taking in account that Sig1 and Sig2 are read separately, one can conclude

that Sig1 is an actual ACM-signature, while Sig2 is, for example, a version of ACM.

Number of channels and Bitrate are specified in ACM-file, but they are

not used in the games I know (Fallout1, Fallout2 and Planescape). All the

known at the moment files are 22050 Hz. The # of channels are 1 or 2, but

actual count of channels depends on actual usage of a resulting sound samples.

The next two values specify the behaviour of unpacking algorithm.

The packAttrs denotes the level of recursion in packed-data processing. If this value is zero, no actual data-unpacking is performed.

The packAttrs2 is number of sub-blocks in a Packed-Block (see section 2).

- Actions

Based on header values the next variables are defined (the names correspond to the source code):

a) someSize = 1 << packAttrs (size of one sub-block)

b) someSize2 = someSize \* packAttrs2 (size of one PackedBlock)

c) decBuff\_size = 3\*someSize/2 - 2 (size of "memory"-buffer), this value

is the sum of following geometric progression:

2 + 4 + 8 +...+ ( 1 << (packAttrs-1) )

plus ( 1 << (packAttrs-1) ) once again.

II. The following arrays are created (their names differs from source):

a) PackedBlock = DWORD [someSize2] (corresponds to someBuff). It is more

convenient to think of this array as of variable-length array

DWORD [packAttrs2][someSize] or DWORD [cnt\_of\_blocks][block\_len].

This array is used to process packed data.

b) if packAttrs is not zero, then the following array is created and

initialized with zero: MemoryBuffer = DWORD [decBuff\_size] (corresponds to decompBuff). This array can be thought of as the following structure:

{

WORD [someSize/2][2],

DWORD [someSize/4][2],

DWORD [someSize/16][2],

...

DWORD [1][2]

}

Each item is used at a corresponding recursion level. This buffer is used

as a seam, as a glue between two consecutive PackedBlocks during

data-unpacking.

c) AmplitudeBuffer = WORD [0x10000], and

Buff\_Middle = &amplitudeBuffer[0x8000], the latter is used as an array of

WORDs with indices from -0x8000 to 0x7FFF.

The buffer is filled with amplitude values used in subsequent unpacking.

- ACM-Stream.

This is a bit-stream. It is organized as a collection of bit-blocks,

each of which can be unpacked into someSize2 samples, i.e. into one PackedBlock.

Let's call such a bit-block just a BitBlock. Then an ACM-Stream is just a

sequence of BitBlocks (with various length).

Each BitBlock contains information about amplitudes used in corresponding

PackedBlock and the description of how to manage these amplitudes in order to produce sound samples.

The first 20 bits of BitBlock can be considered as its header:

bits | name

------+------

4 | pwr

16 | val

Using these values the AmplitudeBuffer is filled as following:

count = 1 << pwr

Buff\_Middle [+0] = 0

Buff\_Middle [+1] = val

Buff\_Middle [+2] = 2\*val

...

Buff\_Middle [+(count-1)] = (count-1)\*val

Buff\_Middle [-1] = -val

Buff\_Middle [-2] = -2\*val

...

Buff\_Middle [-count] = -count\*val

PackedBlock is filled with values from AmplitudeBuffer. This is done

with help of special filling subroutines (or Fillers).

There are 14 different Fillers (their description will be given later):

Zero, Ret0, Linear, k13, k12, t15, k24,

k23, t27, k35, k34, k45, k44 and t37.

Each (well, almost each) Filler is intended to fill only one column of

PackedBlock, which is considered as DWORD[packAttrs2][someSize] (n-th column in this case is an array

{ PB[0][n], PB[1][n], ... , PB[pa2-1][n] }).

For each column of PackedBlock one of Fillers is invoked. To select the

Filler, 5 bits are read from BitBlock, and this 5-bit value is used as an index in the following array:

Filler[32] = {Zero, Ret0, Ret0, Linear, Linear, Linear, Linear, Linear,

Linear, Linear, Linear, Linear, Linear, Linear, Linear, Linear,

Linear, k13, k12, t15, k24, k23, t27, k35,

k34, Ret0, k45, k44, Ret0, t37, Ret0, Ret0};

Two parameters are passed to a Filler: its index in the array and the number of a column it is applied to.

Thus the filling of PackedBlock with amplitudes can be outlined with the

following C-pseudocode:

for (int i=0; i<someSize; i++) {

int Ind = get\_bits\_from\_BitBlock (5);

(Fillers [i]) (Ind, i);

}

- Description of Fillers.

Zero. Fills the column with zero. Does not use any bits from BitBlock.

Ret0. Breaks the filling of current PackedBlock, discards its contents and starts a new BitBlock. In rather large ACM-files I've tested I have not found this kind of Filler.

Linear. 'Ind' parameter is the number of bits from BitBlocks which are used as an index of a value in AmplitudeBuffer. In pseudocode:

Linear (int Ind, int column\_n) {

for (int i=0; i<packAttrs2; i++) {

int val = get\_bits\_from\_BitBlock (Ind);

PackedBlock [i][column\_n] = Buff\_Middle [val];

}

}

k13. Uses variable count of bits (up to 3) from BitBlock to fill the column:

bit-sequence | action

(in order of |

appearance) |

--------------+-------------

0 | PB[i][n] = 0; PB[++i][n] = 0

1, 0 | PB[i][n] = 0

1, 1, 0 | PB[i][n] = Buff\_Middle [-1]

1, 1, 1 | PB[i][n] = Buff\_Middle [+1]

k12. Up to 2 bits:

bit-seq. | value

----------+-----------

0 | 0

1, 0 | Buff\_Middle [-1]

1, 1 | Buff\_Middle [+1]

k24. Up to 4 bits:

bit-seq. | value(s)

----------+-----------

0 | 0, 0

1,0 | 0

1,1,0,0 | Buff\_Middle [-2]

1,1,1,0 | Buff\_Middle [-1]

1,1,0,1 | Buff\_Middle [+1]

1,1,1,1 | Buff\_Middle [+2]

k23. Up to 3 bits:

bit-seq. | value

----------+-----------

0 | 0

1,0,0 | Buff\_Middle [-2]

1,1,0 | Buff\_Middle [-1]

1,0,1 | Buff\_Middle [+1]

1,1,1 | Buff\_Middle [+2]

k35. Up to 5 bits:

bit-seq. | value(s) /----> 2bits | B\_M index

--------------+----------- | -------+-----------

0 | 0, 0 | 0,0 | -3

1,0 | 0 | 1,0 | -2

1,1,0,0 | Buff\_Middle [-1] | 0,1 | +2

1,1,0,1 | Buff\_Middle [+1] | 1,1 | +3

1,1,1, 2bits | (\*) -------------/

k34. Up to 4 bits:

bit-seq. | value /----> 2bits | B\_M index

------------+----------- | -------+-----------

0 | 0 | 0,0 | -3

1,0,0 | Buff\_Middle [-1] | 1,0 | -2

1,0,1 | Buff\_Middle [-1] | 0,1 | +2

1,1, 2bits | (\*) -------------/ 1,1 | +3

k45. Up to 5 bits:

bit-seq. | value

-------------+-----------

0 | 0, 0

1,0 | 0

1,1, 3bits | 3bits->B\_M index: 000-> -4, 100-> -3, 010-> -2, 110-> -1

001-> +1, 101-> +2, 011-> +3, 111-> +4

k45. Up to 4 bits:

bit-seq. | value

-----------+-----------

0 | 0

1, 3bits | 3bits->index: 000-> -4, 100-> -3, 010-> -2, 110-> -1

001-> +1, 101-> +2, 011-> +3, 111-> +4

t15. Takes 5 bits from BitBlock. This value is considered as a base-3

number with 3 digits. Each digit is used as an index in Buff\_Middle. So 3

consecutive items of a column are filled.

t27. Reads 7 bits and treats obtained value as base-5 number with 3 digits.

Each digit is used as an index in Buff\_Middle, in this way 3 consecutive

items of a column are filled.

t37. Takes 7 bits and treats their value as base-11 number with 2 digits.

Each digit is used as an index in Buff\_Middle, in this way two consecutive

items of a column are filled.

IV. Processing of data in PackedBlock.

If packAttrs is zero, we do not have do to anything else, we've got the sound samples in PackedBlock. Otherwise we need to apply unpacking algorithm to values in PackedBlock to gain sound samples. This is done in functions unpackValues, sub\_4d3fcc and sub\_4d420c (see source).

All I can tell about the unpacking is that it is recursive. During the

process the PackedBlock is treated as DWORD [cnt\_of\_blocks][block\_len]. At the beginning of the processing

cnt\_of\_blocks = packAttrs2\*2

block\_len = someSize/2.

From some recursion level to the next level cnt\_of\_blocks is multiplied by 2 and block\_len is divided by 2, so the value cnt\_of\_blocks\*block\_len is a constant.

Also I can describe the properties of results from this data processing.

NOTE: to study these properties, an algorithm was slightly altered: in

unpackValues function the for-loop with increment of items of PackedBlock was

commented out. This increment just adds a constant value to all the items of

resulting array, so it is not a core part of algorithm.

Let's call F(PB) the result of applying of functions mentioned above to a

PackedBlock PB. And denote by PB{[i]=v} a PackedBlock where all the items

are equal zero, except for the i-th one, which is assigned a value of v.

That is PB{[1]=5} symbolizes the array {0,5,0,...,0}.

The following properties of F can be observed:

1) "Compact support".

F (PB{[i]=v}) is zero everywhere, except for items from i to

i+2\*someSize-1.

2) Periodicity with period someSize.

F (PB{[i]=v}) and F (PB{i+someSize}=v) differ only in the way, that the

latter one is shifted by someSize.

3) Linearity.

F (PB{[i]=a, [j]=b}) = a\*F (PB{[i]=1}) + b\*F (PB{[j]=1}).

4) Almost orthogonality (a strange one).

Let's denote f\_i = F (PB{[i]=1}), where i=0..someSize-1. w\_i =

F (PB{ [i]=1, [i+someSize]=1, [i+2\*someSize]=1, ... } ), shifted to the left

by (2\*someSize) items (that is we simply discard first items). And group

their indices in the following way:

g0 = {0}

g1 = {1}

g2 = {2, 3}

g3 = {4, 5, 6, 7}

...

g\_packAttrs = { ..., (someSize-1) }

Then any two vectors w\_i and f\_j are orthogonal (in sense of inner product)

if their indices belong to different groups.

Moreover, w\_i and f\_j belonging to the same group are orthogonal if i and j

are of different parity (I am not sure of this term; what I mean to say is

that one of them is even while the another is odd).

May be w\_i can be treated as test-functions for presence of f\_j in

a signal.

Credits:

TeamX – for providing the original documents

Falloutmods Fandom/Vault Tec labs – for distributing and translating said documents.

TelamonLivesOn – for providing newer conversion/modding tools