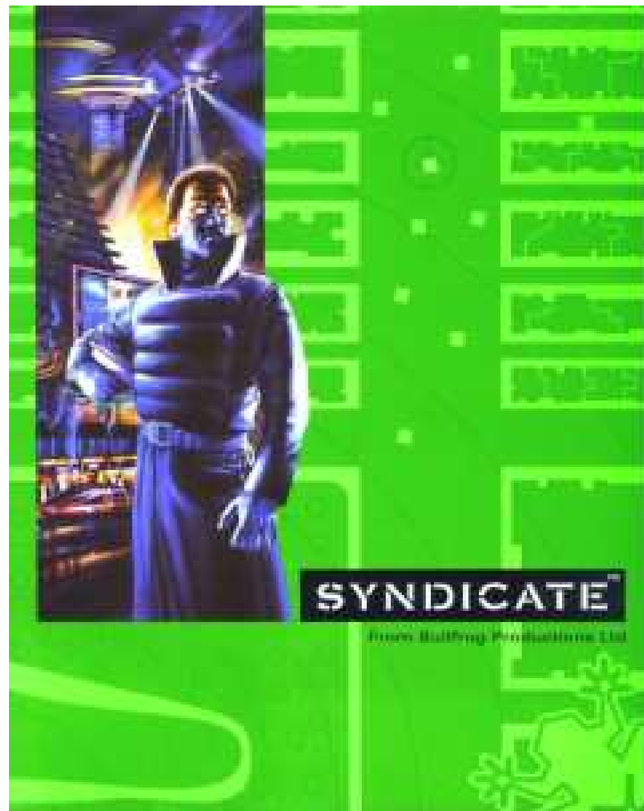

Syndicate file format

(This is the reference for coding libsyndicate)

September 30, 2007

The libsyndicate project is in no way affiliated with Electronic Arts and/or Bullfrog Entertainment.
Syndicate and Bullfrog are trademarks of Electronic Arts.
Syndicate is © 1993 Electronic Arts.

Contents

1 Introduction 6

2 List of files and types 6
- 2.1 List of data files . . . 6
- 2.2 List of data types . . . 8

3 File format 9
- 3.1 RNC . . . 9
  - 3.1.1 Header . . . 9
  - 3.1.2 Compressed data . . . 9
- 3.2 Palette . . . 10
- 3.3 Font . . . 11
- 3.4 Req . . . 12
- 3.5 MapData . . . 13
- 3.6 MapColumn . . . 14
- 3.7 MapTile . . . 14
- 3.8 SpriteAnim . . . 15
- 3.9 SpriteFrame . . . 16
- 3.10 SpriteElement . . . 18
- 3.11 SpriteTab . . . 18
- 3.12 SpriteData . . . 18
- 3.13 Mission . . . 19
- 3.14 Game . . . 19
  - 3.14.1 Common structure . . . 20
  - 3.14.2 Pedestrians . . . 22
  - 3.14.3 Vehicles . . . 25
  - 3.14.4 Objects . . . 25
  - 3.14.5 Weapons . . . 26
  - 3.14.6 Sfx . . . 27
  - 3.14.7 Scenarios . . . 28
  - 3.14.8 Mapinfos . . . 28
  - 3.14.9 Objectives . . . 29
- 3.15 Fli . . . 29
- 3.16 Raw . . . 30

4 Matrix of missions, maps, and games 30

5 Menus sequence 32

6 Equipement guide 32

7 Agents names **34**

8 Methods and tools **34**

8.1 Cheat codes 34
8.2 Hexadecimal editor 34
8.3 Opened files 34
8.4 Strings 35
8.5 Memdumps 35

9 References **35**

10 TODO **35**

1 Introduction

This is a documentation that resume informations that was needed to code the libsyndicate.

Thanks to :

- Jon skeet for dernc.
- Andrew Sampson for the first reverse ingenering of graphic files.
- Marcin Olak and the Desyndicate wiki.
- Stuart Binge, Joost Peters, Trent Waddington and the freesynd project staff.
- Tomasz Lis for his fan site that resume the whole above.
- Mike Melanson for details on fli files.

The author of this document is Paul Chavent (valefor at icculus dot org).

The Syndicate version covered by this document is the one for PC.

2 List of files and types

2.1 List of data files

This is a list of files with the type of data. Types of data are defined in section 2.2.

The prefix 'h' could mean Hight resolution. For example there are `HPOITNER.DAT` and `LPOINTER.DAT`.

The prefix 'm' could mean menu.

The prefix 'i' could mean intro.

File	Type	Comment
COL01.DAT	MapColumn	Gives the type for each of the 256 tiles
GAME[xx].DAT	Game	The description of the games
HBLK01.DAT	MapTile	The 256 base tiles that compund maps
HELE-0.ANI	SpriteElement	The descriptions of sprite elements
HFNT01.DAT	Font	FIXME : don't know for what is it used ?
HFRA-0.ANI	SpriteFrame	The descriptions of sprite frames
HPAL01.DAT	Palette	palettes for maps
HPAL02.DAT	Palette	
HPAL03.DAT	Palette	
HPAL04.DAT	Palette	
HPAL05.DAT	Palette	
HPALETTE.DAT	Palette	FIXME
HPOINTER.DAT	SpriteData	arrow, pick, target, pointers
HPOINTER.TAB	SpriteTab	
HREQ.DAT	GameFont	Fonts used for the game screen

File	Type	Comment
HSPR-0.DAT	SpriteData	sprites for maps
HSPR-0.TAB	SpriteTab	
HSTA-0.ANI	SpriteAnim	The descriptions of sprite anims
INTRO.DAT	Fli	Animation for the introduction
INTRO.XMI	Music	Music for the introduction
ISNDS-0.DAT	SoundData	FIXME
ISNDS-0.TAB	SoundTab	FIXME
ISNDS-1.DAT	SoundData	FIXME
ISNDS-1.TAB	SoundTab	FIXME
MAP[xx].DAT	MapData	Map data (tiles reconstitution)
MBRIEF.DAT	Fli	
MBRIEOUT.DAT	Fli	
MCONFOUT.DAT	Fli	
MCONFUP.DAT	Fli	
MCONSCR.DAT	Raw	
MDEBRIEF.DAT	Fli	
MDEOUT.DAT	Fli	
MENDLOSE.DAT	Fli	
MENDWIN.DAT	Fli	
MFNT-0.DAT	SpriteData	The menu fonts (rle encoded)
MFNT-0.TAB	SpriteTab	
MGAMEWIN.DAT	Fli	
MISS[xx].DAT	Mission	Defines the mission objectives etc.
MLOGOS.DAT	Raw	
MLOSA.DAT	Fli	
MLOSAOUT.DAT	Fli	
MLOSEGAM.DAT	Fli	
MMAP.DAT	Fli	
MMAPBLK.DAT	Raw	
MMAPOUT.DAT	Fli	
MMINLOGO.DAT	Raw	
MMULTI.DAT	Fli	
MMULTOUT.DAT	Fli	
MOPTION.DAT	Fli	
MOPTOUT.DAT	Fli	
MRESOUT.DAT	Fli	
MRESRCH.DAT	Fli	
MSCRENUP.DAT	Fli	
MSELECT.DAT	Fli	
MSELECT.PAL	Palette	Palette for the menus
MSELOUT.DAT	Fli	
MSPR-0.DAT	SpriteData	The menu sprites (rle encoded)
MSPR-0.TAB	SpriteTab	
MTITLE.DAT	Fli	
SAMPLE.AD	Audio/sound	
SAMPLE.OPL	Audio/sound	

File	Type	Comment
SOUND-0.DAT	SoundData	
SOUND-0.TAB	SoundTab	
SOUND-1.DAT	SoundData	
SOUND-1.TAB	SoundTab	
SYNGAME.XMI	Music	

Table 1: Table of files.

2.2 List of data types

This is a list of types.

Type	Reverse	Files associated
Fli		INTRO.DAT INTRO.XMI MBRIEF.DAT MBRIEOUT.DAT MCONFOUT.DAT MCONFUP.DAT MCONSCR.DAT MDEBRIEF.DAT MDEOUT.DAT MENDLOSE.DAT MENDWIN.DAT MGAMEWIN.DAT MLOSA.DAT MLOSAOUT.DAT MLOSEGAM.DAT MMAP.DAT MMAPOUT.DAT MOPTION.DAT MOPTOUT.DAT MRESOUT.DAT MRESRCH.DAT MSCRENUP.DAT MSELECT.DAT MSELOUT.DAT MTITLE.DAT MMULTI.DAT MMULTOUT.DAT
Font	100%	HFNT01.DAT
Game	50%	GAME[xx].DAT
MapColumn	100%	COL01.DAT
MapData	100%	MAP[xx].DAT
MapTile	100%	HBLK01.DAT
Mission	100%	MISS[xx].DAT
Music		SYNGAME.XMI
Palette	100%	HPAL[xx].DAT HPALETTE.DAT MSELECT.PAL
Raw	100%	MLOGOS.DAT MMAPBLK.DAT MMINLOGO.DAT
Req	75%	HREQ.DAT
SoundData		ISNDS-[x].DAT SOUND-[x].DAT
SoundTab		ISNDS-[x].TAB SOUND-[x].TAB
SpriteAnim	100%	HSTA-0.ANI
SpriteFrame	100%	HFRA-0.ANI
SpriteElement	100%	HELE-0.ANI
SpriteTab	100%	HPOINTER.TAB HSPR-0.TAB MFNT-0.TAB MSPR-0.TAB
SpriteData	100%	HPOINTER.DAT HSPR-0.DAT MFNT-0.DAT MSPR-0.DAT

Type	Reverse	Files associated

Table 2: Table of types.

3 File format

The field are integers only, so we will use the standard int types. We prefix them with `le_` if it's coded in little endian, with `be_` else. For example, if a field is a 16 bit unsigned integer, coded in little endian we call it `le_uint16_t`.

We will also use a base type called `Block`. They will be explained later.

3.1 RNC

The files are compressed with a tool called Pro-Pack from Rob Northen Computing.

There are two main parts

- an header
- the compressed data

3.1.1 Header

The header is in big endian.

```
struct Header
{
  be_uint32_t _signature;
  be_uint32_t _unpacked_lentgh;
  be_uint32_t _packed_lentgh;
  be_uint16_t _unpacked_crc;
  be_uint16_t _packed_crc;
  be_uint8_t  _unknown;
  be_uint8_t  _pack_count;
} _header;
```

The signature is always `0x524E4301` (ie the string "RNC ").

3.1.2 Compressed data

The compressed data should be read as a stream by block of 16 bits (big endian) as in the example on figure 1.

At the begining of the stream there are *two unknown bits.* We can skip them (don't know what there are for now).

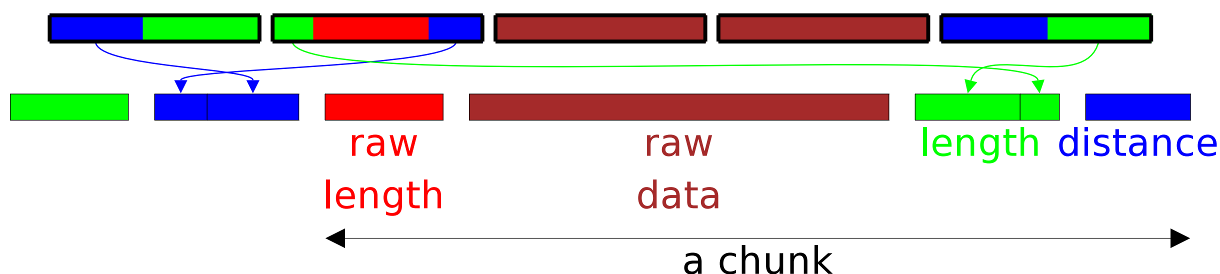

Figure 1: An extract of the bitstream.

Then the compressed data are divided in *pack*. The number of pack is given by the header field `_pack_count`.
Each pack begin with three *huffman tables*[1]. The structure of a table is :

- 5 bits that give the maximum value of the nodes
- 4 bits for each values that give their leaf depth

The first table is a *raw table*, the second is a *distance table* and the third is a *length table*.
Then there is 16 bits that give the *chunk count*.
A chunk is (see figure 2) :

- a block of raw data
- a copy of a part from a previous block

The *raw length* is given by the first huffman table and next bits of the stream. The *raw length* next bytes (aligned on 16 bits) are sent to the output.
The *distance* is given by the second huffman table and next bits of the stream. The *distance* is the offset from the current output of the pattern. We have to add 1 to the value given by the table.
The *length* is given by the third huffman table and next bits of the stream. The *length* is the length of the pattern to copy to the output. We have to add 2 to the value given by the table.

3.2 Palette

The Palette files are : `HPAL[xx].DAT`, `HPALETTE.DAT`, `MSELECT.PAL`.
Their structure is :

```
struct Palette
{
  struct Color
  {
    uint8_t _r;
    uint8_t _g;
    uint8_t _b;
```

[1] I like this tutorial (fr) http://tcharles.developpez.com/Huffman/

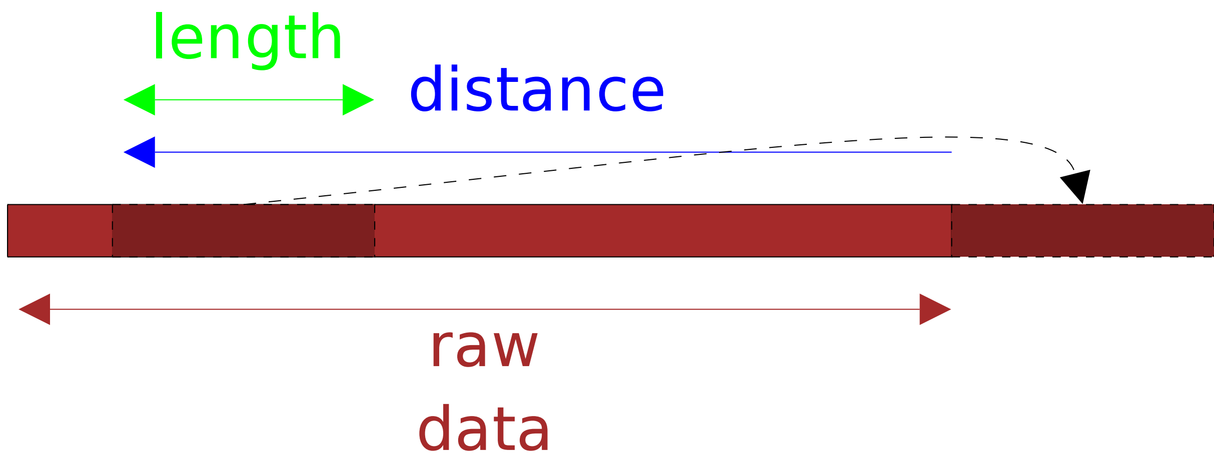

Figure 2: A chunk

```
  } _rgb[256];
};
```

There are 16 colours defined, and the value are between 0 and 63 (ie 7 bits).
For an 8 bits system, we need to scale the values between 0 and 255.

3.3 Font

The Font file is : `HFNT01.DAT`.
His structure is :

```
struct Font
{
  struct Table
  {
    le_uint16_t _offset;
    le_uint8_t  _width;
    le_uint8_t  _height;
    le_uint8_t  _line_offset;
  } _tab[128];
  le_uint8_t _data[];
};
```

_offset is the offset in the _data array,

_width is the width of the font,

_height is the height,

_line_offset is the vertical offset where to draw the font from the top (FIXME : check),

_data are the data.

The pixel are coded with one bit.

If the width is strictly less than 8, each line is coded as `le_uint8_t` where each bit represent a pixel.

If the width is greater or equal than 8, each line is coded as `le_uint16_t` where each bit represent a pixel. So the first pixel is the eightith bit, the last pixel is the seventh.

3.4 Req

FIXME : find a better name for this.

The Req file is : `HREQ.DAT`.

His structure is :

```
struct Req
{
  struct Entry
  {
    Block840    _lines[_height];
    le_uint8_t  _spares[16];
  } _entries[];
};
```

These data are fonts. They are 8 width and 16 height. The pixel are packed by line. So we have 16 block of 8 pixels and zero alpha channel.

A block of 8 pixel is 16 bytes. The pixels are coded on 4 bits in little endian.

- There are 32 bits for the lsb of the index of each pixels.
- There are 32 bits for the [].
- There are 32 bits for the [].
- There are 32 bits for the msb of the index of each pixels.

So they are coded as follow :

```
struct Block840
{
  le_uint32_t  _bit_0;
  le_uint32_t  _bit_1;
  le_uint32_t  _bit_2;
  le_uint32_t  _bit_3;
};
```

So bit 0 of pixel 0 is the 7th bit of `_bit_0` and bit 0 of pixel 32 is the 24th bit of `_bit_0`.

FIXME : some fields are unknown. Moreover there isn't alpha channel, so should we consider a value as transparent ?

3.5 MapData

The MapData files are : `MAP[xx].DAT`.
His structure is :

```
struct MapData
{
  le_uint32_t _nb_i;
  le_uint32_t _nb_j;
  le_uint32_t _nb_k;
  le_uint32_t _offset[_nb_i * _nb_j];
  le_uint8_t  _tile[??? * _nb_k];
};
```

_nb_i is the number of tiles on i,

_nb_j is the number of tiles on j,

_nb_k is the number of tiles on k,

_offset is a table with the offset (from byte 12) of the tiles index,

_tile are the tiles index packed by stack.

For example, if we want the tile at (i;j;k), it is `_tile[_offset[j * _nb_i + i] * _nb_k + k]`
The figure 3 illustrate the components of a map.

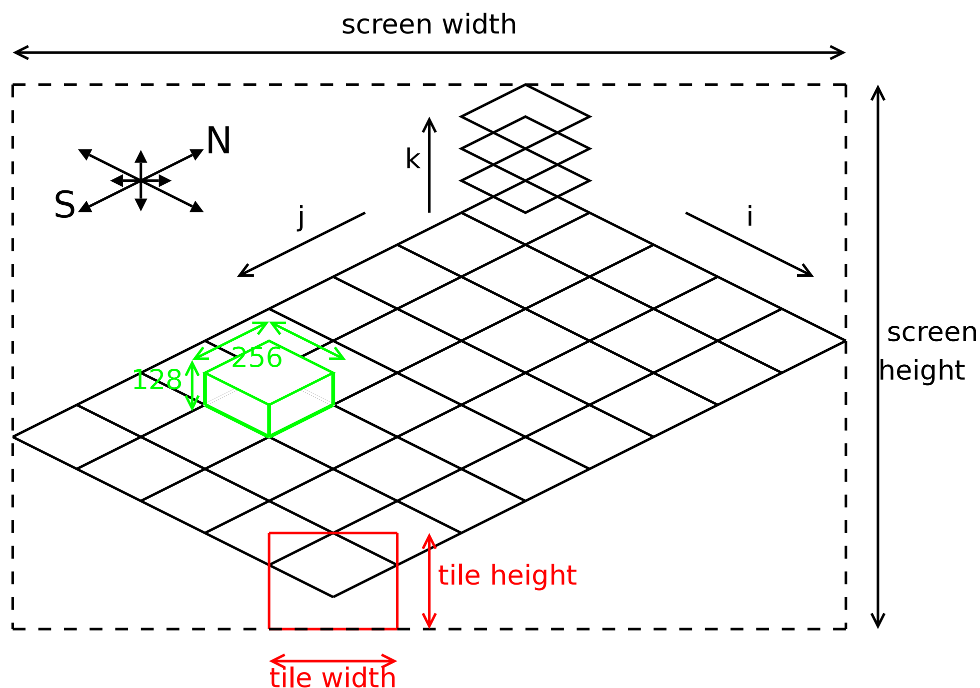

Figure 3: A map.

3.6 MapColumn

The MapColumn file is : `COL01.DAT`.

His structure is :

```
struct MapColumn
{
  le_uint8_t _type[256];
};
```

This file give the type of each tiles from `HBLK01.DAT` :

```
enum ColType
{
  None,
  SlopeSN,
  SlopeNS,
  SlopeEW,
  SlopeWE,
  Ground,
  RoadSideEW,
  RoadSideWE,
  RoadSideSN,
  RoadSideNS,
  Wall,
  RoadCurve,
  HandrailLight,
  Roof,
  RoadPedCross,
  RoadMark,
  NbTypes
};
```

I think that we can use this file to deduce if a tile is walkable and its color for the minimap.

For example, we could say that a tile is walkable :

```
if((None < type_k && type_k != HandrailLight && type_k
<  NbTypes) &&
   (None == type_k+1 || type_k+1 == HandrailLight || type_k+1 == NbTypes)
{
  tile_k is walkable
}
```

For the minimap it is not linear, it seems to be more complicated than associate a type to a colour.

3.7 MapTile

The MapTile file is : `HBLK01.DAT`.

His structure is :

```
struct MapTile
{
  le_uint32_t _offset[256][6];
  struct Subtile
  {
    Block32 _lines[16]
  } _subtile[986];
};
```

A tile is 64 pixels width and 48 pixels height. It is compound of 6 subtiles.

Each subtile is 32 pixels width and 16 pixels height. The pixel are packed by line. So we have 16 block of 32 pixels.

A block of 32 pixel is 20 bytes. The pixels are coded on 5 bits in big-endian : one for the transparency and 4 for the index in the palette :

- There are 32 bits for the transparency bits of each pixels.
- There are 32 bits for the lsb of the index of each pixels.
- There are 32 bits for the [].
- There are 32 bits for the [].
- There are 32 bits for the msb of the index of each pixels.

So they are coded as follow :

```
struct Block32
{
  be_uint32_t _alpha;
  be_uint32_t _bit_0;
  be_uint32_t _bit_1;
  be_uint32_t _bit_2;
  be_uint32_t _bit_3;
};
```

The figure 4 illustrate the components of a map tile.

3.8 SpriteAnim

The SpriteAnim file is : `HSTA-0.ANI`.

His structure is :

```
struct SpriteAnim
{
  le_uin16_t _indexes[];
}
```

This is an array of frame index. So the first frame of the 5th animation is given by `_indexes[5]`. Then the others frames are given in the SpriteFrame array.

The figure 5 illustrate the components of an animation.

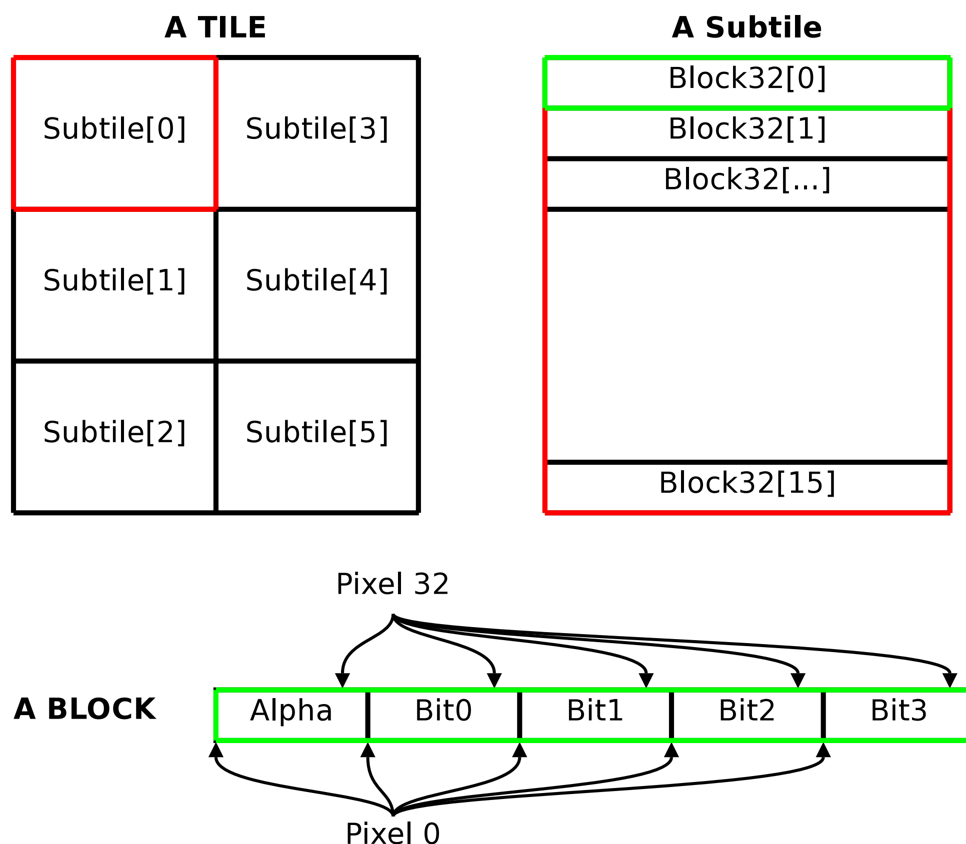

Figure 4: Tiles, Subtiles and blocks.

3.9 SpriteFrame

The SpriteFrame file is : `HFRA-0.ANI`.

His structure is :

```
struct SpriteFrame
{
  struct Frames
  {
    le_uint16_t _first;
    le_uint8_t  _width;
    le_uint8_t  _height;
    le_uint16_t _flags;
    le_uint16_t _next;
  } _frames [];           // FIXME : give the number of frames
};
```

This is an array of frame descritpion. A frame has a width, height, some flags and is compound of sprite elements. The index of the first sprite element is `_first`. This is an index (not an offset) in the Sprite Element tab file (see 3.10). The index of the next frame is given by `_next`. The frame index automatically loop to the first frame.

The `_flags` field seems to be 0x0100 when it is the first frame of an animation.

The frame 1450 is the persuadotron in the inventory. Then every 6 index, there is the

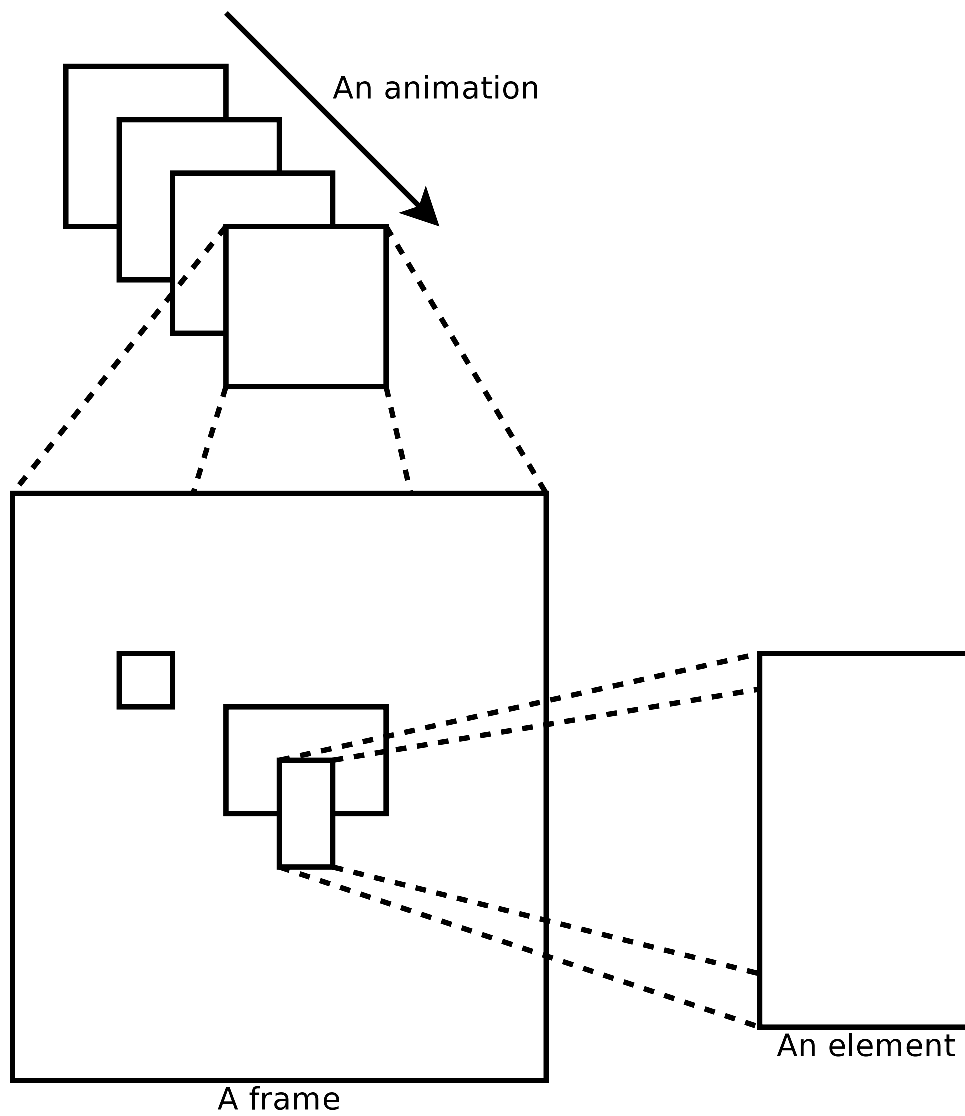

Figure 5: Components of an animation.

next weapon.

3.10 SpriteElement

The SpriteElement file is : `HELE-0.ANI`.
His structure is :

```
struct SpriteElement
{
  struct Element
  {
    le_uint16_t _sprite;
    le_int16_t  _x_offset;
    le_int16_t  _y_offset;
    le_uint16_t _x_flipped;
    le_uint16_t _next;
  } _elements[];          // FIXME : give the number of elements
};
```

The `_sprite` field give the index (not an offset) in the sprite tab file (see 3.11). The `_next` is the index of the next element. If it is zero there isn't any more elements. The `_x_flipped` attribut tells if it is horizontaly flipped.

3.11 SpriteTab

The SpriteTab file are : `HPOINTER.TAB`, `HSPR-0.TAB`, `MFNT-0.TAB`, `MSPR-0.TAB`.
Their structure is :

```
struct SpriteTab
{
  struct Entry
  {
    le_uint32_t _offset;
    le_uint8_t  _width;
    le_uint8_t  _height;
  } _entries[];           // FIXME : give the number of entries
};
```

The offset give the number of bytes to skip from the begining of the sprite data file (see 3.12).
FIXME : give the apropriate palette for each file.

3.12 SpriteData

The SpriteData file are : `HPOINTER.DAT`, `HSPR-0.DAT`, `MFNT-0.DAT`, `MSPR-0.DAT`.
Their structure is :

```
struct SpriteData
```

```
{
  le_uint16_t  _nb_sprites;
  union
  {
    Block8      _blocks[];
    le_uint8_t  _rle[];
  } _data;
};
```

The sprite datas are encoded as lines of pixels (structured as block) or as rle datas.

The `_nb_sprite` field gives the number of sprites. It seems that when it is not zero, the data are encoded as rle (the flag is 0x0053 for `MSPR-0.DAT` and 0x00CD for `MFNT-0.DAT`). Else, the datas are encoded as lines of pixels (structured in blocks).

If the pixel are packed by line, each line is one or more block of eight pixels. A block is 5 bytes : one for the transparency and 4 for the index in the palette.

- There are 8 bits for the transparency bits of each pixels.
- There are 8 bits for the lsb of the index of each pixels.
- There are 8 bits for the ....
- There are 8 bits for the ....
- There are 8 bits for the msb of the index of each pixels.

3.13 Mission

The Mission file are : `MISSXX.DAT`. They contain text and available in 4 language :

english from 01 to 50

french from 101 to 150

italian from 201 to 250

german from 301 to 350

Each string is separated with an EOL (0x0a). Other separator is the pipe '—' (0x7c).

3.14 Game

The Game file are : `GAMEXX.DAT`.

Their structure is :

```
struct GameStruct
{
  le_uint8_t          _header[6];
  le_uint16_t         _offsets[128][128];
  le_uint16_t         _offset_ref;          //          (32774)
```

```
  struct Pedestrian  _pedestrians[256];   // 0x0:8008 (32776)
  struct Vehicle     _vehicles[64];       // 0x0:DC08 (56328)
  struct Object      _objects[400];       // 0x0:E688 (59016)
  struct Weapon      _weapons[512];       // 0x1:1568 (71016)
  struct Sfx         _sfx[256];           // 0x1:5D68 (89448)
  struct Scenario    _scenarios[2048];    // 0x1:7B68 (97128)
  le_uint8_t         _unkn08[448];        // 0x1:BB68 (113512)
  struct Mapinfos    _mapinfos;           // 0x1:BD28 (113960)
  struct Objectives  _objectives[10];     // 0x1:BD36 (113974)
  le_uint8_t         _unkn11[1896];       // 0x1:BD98 (114114)
};
```

The header could be seeds for example (FIXME, not sure).

The `_offsets` field is an array that represent the tiles of the map (every map are 128x128 tiles). The values plus 32774 give an offset in this file that is the entity placed on this tile. The resulting offset can be 98309 max and only peds, vehicle, objects and weapons can be indexed.
It is used for the minimap in the breifing menu. As a clue, for the first level, there are three red points, that are in the `_offsets` array. It is also probably (not sure) used for te minimap in the game.
The values for offset are :

- [2; 23554[pedestrian
- [23554; 26242[vehicle
- [26242; 38242[objects
- [38242; 56674[weapons
- [56674; 64354[sfx

The `_unkn08` field is an array of 2048 structures of 8 bytes. Perhaps it as something to do with A.I.

The `_unkn11` field is an array of 129 structures of 15 bytes.

There are 116010 bytes in all files.

3.14.1 Common structure

The Pedestrian, Vehicle, Object and Weapon have a common header like this :

```
struct
{
  le_uint16_t  _offset_next;
  le_uint16_t  _offset_prev;
  le_uint16_t  _tile_i;
  le_uint16_t  _tile_j;
  le_uint16_t  _tile_k;
  le_uint8_t   _unkn10;
  le_uint8_t   _unkn11;
```

```
  le_uint8_t   _unkn12[2];
  le_uint16_t _index_base_anim;
  le_uint16_t _index_current_frame;
  le_uint16_t _index_current_anim;
  le_int16_t   _health;
  le_uint16_t _offset_unknown;
  le_uint8_t   _type;
  le_uint8_t   _status;
  le_uint16_t _orientation;
};
```

The `_offset_prev` and `_offset_next` plus 32774 gives the offset in this file of the previous and next entity. This is probably used for drawing the scene. I think that for drawing the scene the pseudo-algo could be :

```
for (k = 0; k < max_k; k++)
  for(j = 0; j < 128; j++)
    for(i = 0; i < 128; i++)
      draw(tile[j * 128 + i]);
      if(_offsets[j * 128 + i]._tile_k >> 8 == k)
        entity = _offsets[j * 128 + i]
        while(entity)
        {
          draw(entity)
          entity = entity._offset_next
        }
```

The `_tile_*` give the location of the entity on the map along i, j or k. Each tile is a cube of side 256x256x128 (see fig. 3). We can deduce the tile id dividing by 256 (along i and j) or 128 (along k).

The `_unkn10` is unknown but contain 0x04 for peds, 0x05 for weapons (FIXME check this).

The `_index_base_anim` give an index (not an offset) in the file `HSTA-0.ANI`. It is the base offset for the animation of this ped.

The `_index_current_frame` give an index (not an offset) in the file `HFRA-0.ANI`. It is the current frame of the current animation displayed.

The `_index_current_anim` give an index (not an offset) in the file `HSTA-0.ANI`. It is the current animation.

The `_health` give the ressources of the element. For a pedestrian it will be the health, for a weapon, the amo. I'am not sure it's alwas a signed int.

The `_offset_unknown` added to 32774 give an offset in this file. This seems to be a kind of "dependency" (see section 3.14.2).

The `_type` give the type of objects :

- 0x01 ped,
- 0x02 vehicle,
- 0x03 sfx,

- 0x04 weapon,
- 0x05 object.

For example, it allow to display a target or a pickup on the game screen or for the minimap.
The _status may contain informations about the status of the object, or for weapons the “subtype”.
The _orientation give the initial orientation (illustrated on fig. 6) of the element :

- from 0xF0 to 0x10 : south
- from 0x10 to 0x30 : south-east
- from 0x30 to 0x50 : east
- from 0x50 to 0x70 : east-north
- from 0x70 to 0x90 : north
- from 0x90 to 0xB0 : north-west
- from 0xB0 to 0xD0 : west
- from 0xD0 to 0xF0 : west-south

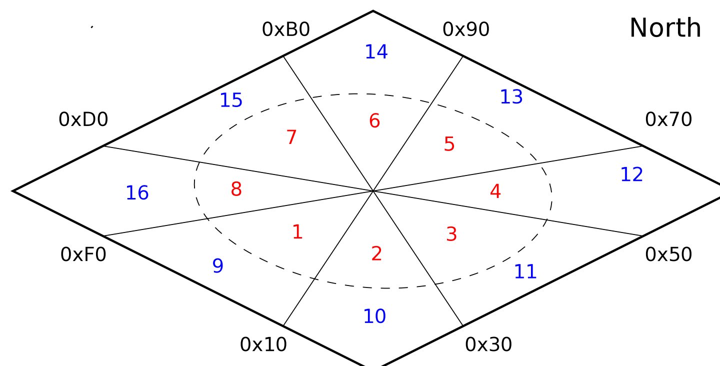

Figure 6: Orientation of the ped is given i black (0x20 represent 45 degree). The offset of the animation if the ped doesn’t move is given in red. And the offset of the ped if he moves is given in blue.

3.14.2 Pedestrians

The _pedestrians is an array of 256 structures that describe pedestrians. This array is at adress 32776 (0x8008), and each structure is 92 bytes.

```
struct Pedestrian
{
  // - 00
  le_uint16_t _offset_next;
  le_uint16_t _offset_prev;
  le_uint16_t _tile_i;
```

```
    le_uint16_t _tile_j;
    le_uint16_t _tile_k;
    // - 10
    le_uint8_t  _unkn10;
    le_uint8_t  _unkn11;
    le_uint8_t  _unkn12;
    le_uint8_t  _unkn13;
    le_uint16_t _index_base_anim;
    le_uint16_t _index_current_frame;
    le_uint16_t _index_current_anim;
    // - 20
    le_int16_t  _health;
    le_uint16_t _offset_last_enemy;
    le_uint8_t  _type;
    le_uint8_t  _status;
    le_uint16_t _orientation;
    le_uint8_t  _unkn28;        // when 01 pedestrian, 02 agent, 04 police, 0
    le_uint8_t  _unkn29;
    // - 30
    le_uint16_t _unkn30;
    le_uint16_t _offset_of_persuader;
    le_uint16_t _unkn34;
    le_uint16_t _offset_of_vehicle;
    le_uint16_t _offset_scenario;
    // - 40
    le_uint16_t _offset_scenario;
    le_uint16_t _unkn42;
    le_uint16_t _offset_of_vehicle;
    le_uint16_t _goto_tile_i;
    le_uint16_t _goto_tile_j;
    // - 50
    le_uint16_t _goto_tile_k;
    le_uint8_t  _unkn52[6];
    le_uint16_t _offset_equipment;
    // - 60
    le_uint16_t _mods_info;
    le_uint8_t  _unkn62[6];
    le_uint16_t _offset_cur_weapon;
    // - 70
    le_uint8_t  _unkn70;
    le_uint8_t  _adrena_amount;
    le_uint8_t  _adrena_dependency;
    le_uint8_t  _adrena_effect;
    le_uint8_t  _unkn74;
    le_uint8_t  _inteli_amount;
    le_uint8_t  _inteli_dependency;
    le_uint8_t  _inteli_effect;
    le_uint8_t  _unkn78;
```

```
  le_uint8_t   _percep_amount;
  le_uint8_t   _percep_dependency;
  le_uint8_t   _percep_effect;
  le_uint8_t   _unkn82;
  le_uint8_t   _unkn83[9];
};
```

The `_unkn10` seems to be 4.

The `_health` of our agent can be `0x10` maximum. When it is less than zero the ped should die.

The `_sub_type` is used for minimap for example (colored dots) and can be (FIXME):

- agent
- enemy agent
- criminals
- civilian
- police
- guard

The `_offset_last_enemy` is the offset of the last peds that hurt, or persuade this ped (Not realy sure).

The `_offset_equipment` + 32774 gives the offset in this file of the first equipment of this ped.

The `_mods_info` gives the level of the mods. It its a bitfield with two bits for each mods :

msb							lsb
spare	brain	eye	heart	chest	arm	leg	gender

. The gender is 1 for femele and 0 for male.

The `_offset_cur_weapon` + 32774 gives the offset in this file of the current weapon in use.

The IPA levels seems to have 4 bytes. At least 3 bytes are sure. They are discribed on the picture 7.

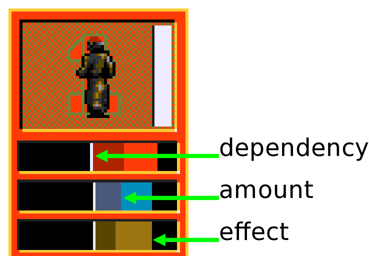

Figure 7: Detail of IPA field.

The `_offset_scenario` + 97128 gives the offset in this file of the first and/or current scenario (there are two fields).

3.14.3 Vehicles

The _vehicles is an array of 64 structures that describe vehicles. This array is at adress 56328 (0xDC08), and each structure is 42 bytes.

```
struct Vehicle
{
  le_uint16_t _offset_next;
  le_uint16_t _offset_prev;
  le_uint16_t _tile_i;
  le_uint16_t _tile_j;
  le_uint16_t _tile_k;
  le_uint8_t  _unkn10;
  le_uint8_t  _unkn11;
  le_uint8_t  _unkn12;
  le_uint8_t  _unkn13;
  le_uint16_t _index_base_anim;
  le_uint16_t _index_current_frame;
  le_uint16_t _index_current_anim;
  le_int16_t  _health;
  le_uint16_t _offset_last_enemy;
  le_uint8_t  _type;
  le_uint8_t  _sub_type;
  le_uint16_t _orientation;
  le_uint8_t  _offset_of_ped;
  le_uint8_t  _unkn30[13];
};
```

The _offset_of_ped + 32774 gives the offset in this file of the first ped in this vehicle.

3.14.4 Objects

The _objects is an array of 400 structures that describe objects (trees, doors, windows, etc.). This array is at adress 59016 (0xE688), and each structure is 30 bytes.

```
struct Object
{
  le_uint16_t _offset_next;
  le_uint16_t _offset_prev;
  le_uint16_t _tile_i;
  le_uint16_t _tile_j;
  le_uint16_t _tile_k;
  le_uint8_t  _unkn10;
  le_uint8_t  _unkn11;
  le_uint8_t  _unkn12;
  le_uint8_t  _unkn13;
  le_uint16_t _index_base_anim;
  le_uint16_t _index_current_frame;
  le_uint16_t _index_current_anim;
  le_uint8_t  _unkn20[4];
```

```
  le_uint8_t   _type;
  le_uint8_t   _sub_type;
  le_uint16_t  _orientation;
};
```

The _sub_type can be :

- 0x0C door,
- 0x12 open window,
- 0x13 close window,
- 0x16 tree,
- ...

3.14.5 Weapons

The _weapons is an array of 512 structures that describe weapons. This array is at adress 71016 (0x11568), and each structure is 36 bytes.

```
struct Weapon
{
  le_uint16_t  _offset_next;
  le_uint16_t  _offset_prev;
  le_uint16_t  _tile_i;
  le_uint16_t  _tile_j;
  le_uint16_t  _tile_k;
  le_uint8_t   _unkn10;
  le_uint8_t   _unkn11;
  le_uint8_t   _unkn12;
  le_uint8_t   _unkn13;
  le_uint16_t  _index_base_anim;
  le_uint16_t  _index_current_frame;
  le_uint16_t  _index_current_anim;
  le_uint16_t  _nb_amos;
  le_uint16_t  _unkn22;
  le_uint8_t   _type;
  le_uint8_t   _sub_type;
  le_uint16_t  _orientation;
  le_uint16_t  _offset_next_inventory;
  le_uint16_t  _offset_prev_inventory;
  le_uint16_t  _offset_owner;
  le_uint16_t  _unkn34;
};
```

The _sub_type can be :

- 0x01 persuadertron,

- 0x02 pistol / air raid com,
- 0x03 gauss gun,
- 0x04 shotgun,
- 0x05 uzi,
- 0x06 minigun,
- 0x07 laser,
- 0x08 flamer,
- 0x09 long range,
- 0x0A scanner,
- 0x0B medikit,
- 0x0C time bomb,
- 0x0D access card / clone shield,
- 0x0E invalid,
- 0x0F invalid,
- 0x10 invalid,
- 0x11 energy shield.

The `_nb_amos` is the number of amos remaining. If the weapon is empty, it is equal to `0xffff` (and the weapon is not selectable). The equipement table 5 give more information about each equipment (nb amo max, range, etc.).

FIXME : is there any info for the picture of the weapon in the inventory ?

3.14.6 Sfx

The `_sfx` is an array of 256 structures that describe sfx (for the flamer, the gauss gun, etc.). This array is at adress 89448 (0x15D68), and each structure is 30 bytes.

```
struct Sfx
{
  le_uint16_t _offset_next;
  le_uint16_t _offset_prev;
  le_uint16_t _tile_i;
  le_uint16_t _tile_j;
  le_uint16_t _tile_k;
  le_uint16_t _unkn10;
  le_uint16_t _unkn12;
  le_uint16_t _index_base_anim;
  le_uint16_t _index_current_frame;
  le_uint16_t _index_current_anim;
```

```
  le_uint16_t _unkn20;
  le_uint16_t _unkn22;
  le_uint16_t _unkn24;
  le_uint16_t _unkn26;
  le_uint16_t _offset_owner;
};
```

3.14.7 Scenarios

The `_scenarios` is an array of 2048 structures that describe scenarios. This array is at adress 97128 (0x17B68), and each structure is 8 bytes.

```
struct Scen
{
  le_uint16_t _next;
  le_uint16_t _offset;
  le_uint8_t  _i_factor;
  le_uint8_t  _j_factor;
  le_uint8_t  _k_factor;
  le_uint8_t  _type;
};
```

The `_next` field give the offset of the next point from the begining of the structure.

The `_offset` field plus 32774 give an offset in this file.

The `_i_factor`, `_j_factor` and `_k_factor` fields gives (i,j,k) coordinates.

```
i = (_i_factor << 7) | 0x0040
j = (_j_factor << 7) | 0x0040
k = (_k_factor << 7) | 0x0000
```

3.14.8 Mapinfos

The `_mapinfos` is a structure that describe the map. This is at adress 113960 (0x1BD28), and the structure is 14 bytes.

```
struct Mapinfos
{
  le_uint16_t _map;
  le_uint16_t _min_x;
  le_uint16_t _min_y;
  le_uint16_t _max_x;
  le_uint16_t _max_y;
  le_uint8_t  _status;
  le_uint8_t  _unkn11[3];
};
```

The `_map` gives the number of the map. For example, if `_map` is 9 the map we should open is MAP09.DAT.

The `_status` flag is set to 1 if the mission has been successfully completed.

3.14.9 Objectives

The _objectives is an array of 10 structures that describe the objectives of the mission. This array is at adress 113974 (0x1BD36), and the structure is 14 bytes.

```
struct Objectives
{
  le_uint16_t _type;
  le_uint16_t _offset;
  le_uint16_t _tile_i;
  le_uint16_t _tile_j;
  le_uint16_t _tile_k;
  le_uint8_t  _status;
  le_uint8_t  _unkn11[3];
};
```

The _type field can be : 0x00 ??? ,0x01 persuade, 0x02 assassinate, 0x03 protect, 0x05 equipment aquisition, 0x0a combat sweep (police), 0x0b combat sweep, 0x0d raid and rescue, 0x0e use/destroy vehicle, 0x10 evacuate.

The _offset + 32774 gives the offset in this file of the objective.

If "protect", the next objective are the goals and their type is zero. The list finish with zero and the offset of the protected item ? FIXME.

The _status flag is set to 1 if the objective has to be completed.

3.15 Fli

The Fli files are : INTRO.DAT, ...

There are some specific informations about Bullfrog fli files at ftp://ftp.mplayerhq.hu/MPlayer/samples/formats/magiccarpet-fli/.

A description of fli can be found at :

- http://www.martinreddy.net/gfx/anim/FLI.txt,
- http://steve.hollasch.net/cgindex/formats/fli.html,
- http://www.textfiles.com/programming/FORMATS/fli_flc.txt.

Other descriptions can be found at :

- http://www.fileformat.info/format/fli/
- http://www.compuphase.com/flic.htm

There are some difference with the original description. The header is :

```
struct {
  le_uint32_t _size;
  le_uint16_t _magic;
  le_uint16_t _frames;
  le_uint16_t _width;
```

```
  le_uint16_t _height;
};
```

The `_size` give the size of the header.

3.16 Raw

The Raw files are : `MCONSCR.DAT`, `MLOGOS.DAT`, `MMAPBLK.DAT`, `MMINLOGO.DAT`.
These files gives raw pixels.

File	Nb of pictures	Dimension of each picture
MCONSCR	1	320x200
MLOGOS	40	32x32
MMAPBLK	50	64x44
MMINLOGOS	40	16x16

Table 3: Table of raw files.

4 Matrix of missions, maps, and games

The games from 90 to 99 are used for multiplayer games. The others are the 50th.

Games	Mission	Map	Palette	Country
01	01	01	2	Western Europe
02	02	02	3	Far East
03	03	03	4	Mongolia
04	04	04	5	Iran
05	05	05	1	California
06	06	06	2	Iraq
07	07	07	1	India
08	08	08	4	Northeast Territories
09	09	63	5	Kazakhstan
10	10	10	1	Eastern Europe
11	11	11	2	Western Australia
12	12	12	3	Kanchatka
13	13	13	4	Mozambique
14	14	20	5	Peru
15	15	18	1	Central Europe
16	16	19	2	Greenland
17	17	35	3	Alaska
18	18	38	4	Urals
19	19	34	5	Northern Territories
20	20	32	1	Scandinavia

Games	Mission	Map	Palette	Country
21	21	39	2	Yukon
22	22	41	3	Siberia
23	23	90	4	Rockies
24	24	70	5	Southern States
25	25	91	1	Indonesia
26	26	92	2	Mexico
27	27	71	3	Zaire
28	28	72	4	Algeria
29	29	40	5	New England
30	30	67	1	Argentine
31	31	93	2	South Africa
32	32	60	3	Colorado
33	33	61	4	Sudan
34	34	62	5	Mid West
35	35	43	1	Lybia
36	36	31	2	Venezuela
37	37	56	3	Atlantic Accelerator
38	38	57	4	Arabia
39	39	58	5	Northwest Territories
40	40	63	1	Kenya
41	41	64	2	Mauritania
42	42	66	3	Newfoundland
43	43	65	4	New South Wales
44	44	80	5	Colombia
45	45	81	1	Nigeria
46	46	82	2	Brazil
47	47	50	3	Uruguay
48	48	51	4	Pacific Rim
49	49	53	5	Paraguay
50	50	54	0	China (Palette 0 !!! intresting !)
90		16	1	
91		21	1	
92		02	1	
93		05	1	
94		43	1	
95		67	1	
96		72	1	
97		90	1	
98		94	1	
99		17	1	

Table 4: Matrix of games, missions and maps.

5 Menus sequence

The menu sequence is depicted on figure 8.

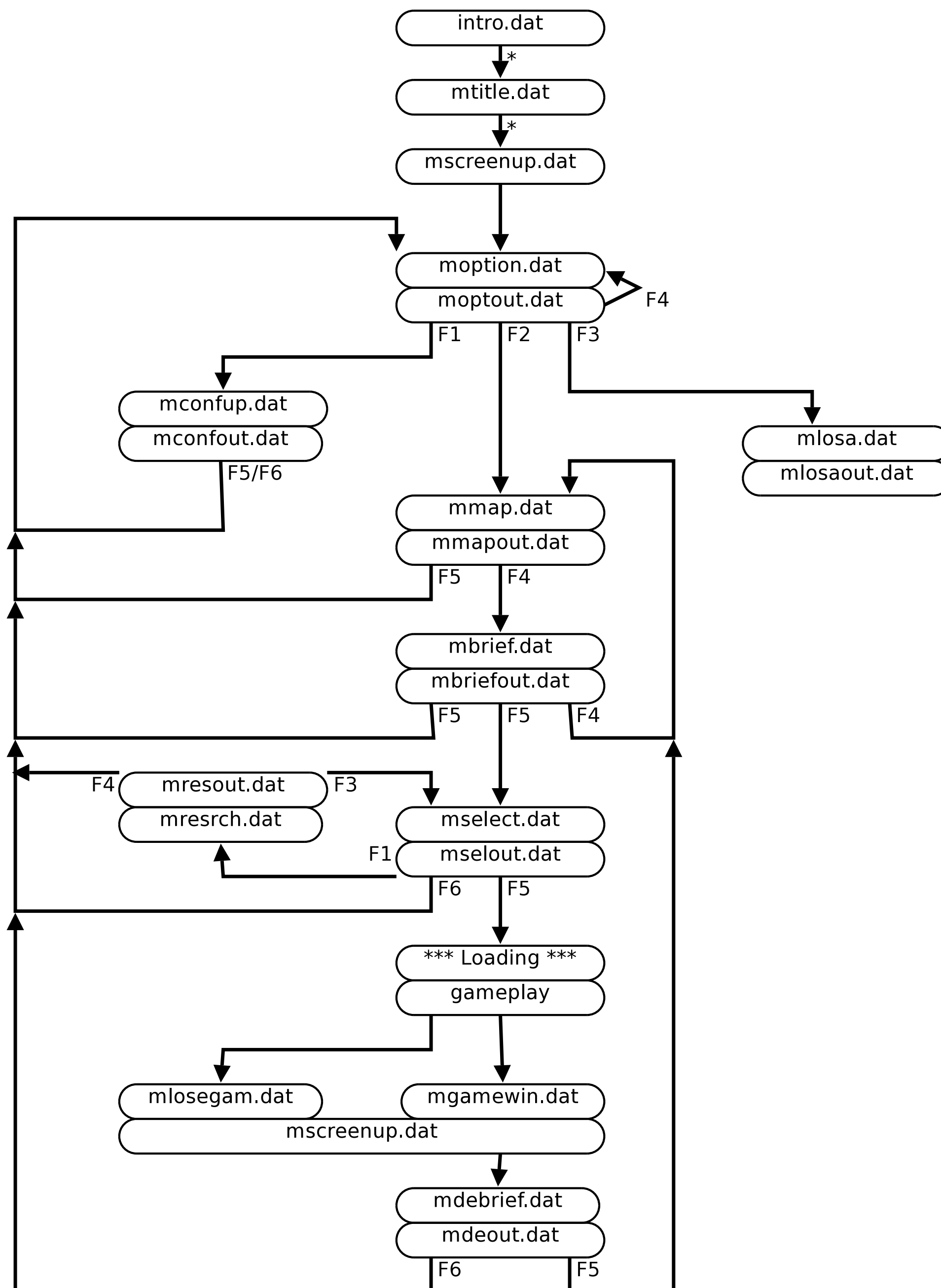

Figure 8: The succession of the differents menu.

6 Equipement guide

Item	Cost	Range	Ammo	Shot
Persuadetron	5000	256	- (0x0032)	-
Pistol	0	1280	13 (0x000c)	0

Item	**Cost**	**Range**	**Ammo**	**Shot**
Guass Gun	50000	5120	3 (0x0002)	15000
Shotgun	250	1024	12 (0x000b)	2
Uzi	750	1792	50 (0x0031)	2
Mini-Gun	10000	2304	500 (0x01f3)	10
Laser	35000	4096	5 (0x0004)	2000
Flamer	1500	512	1000 (0x03e7)	1
Long Ranger	1000	6144	30 (0x001d)	2
Scanner	500	4096	- (0x0013)	-
Medikit	500	256	1 (0x0001)	1
Time Bomb	25000	1000	- (0x00c7)	-
Access Card	1000	256	- (0x0000)	-
Energy Shield	8000	768	200 (0x00c7)	15

Table 5: Table of equipements.

The `_nb_amo` max for each weapon is :

- 0x01 persuadertron 0x32,
- 0x02 pistol 0x0c / air raid com,
- 0x03 gauss gun 0x02,
- 0x04 shotgun 0x0b,
- 0x05 uzi 0x31,
- 0x06 minigun 0x01f3,
- 0x07 laser 0x04,
- 0x08 flamer 0x03e7,
- 0x09 long range 0x1d,
- 0x0A scanner 0x13,
- 0x0B medikit 0x01,
- 0x0C time bomb 0xc7,
- 0x0D access card 0x00 / clone shield,
- 0x11 energy shield 0xc7.

7 Agents names

There are 68 agents :

AFSHAR AARNOLD EBAIRD LTBALDWIN BLACK BOYD PLABOYESEN BRAZIER BROWN R BUSH CARR PLACHRISMAS CLINTON COOPER ECORPES TCOX DAWSON EDONKIN TDISKETT DUNNE EDGAR LAEVANS FAIRLEY FAWCETT FLINT LTFLOYD GRIFFITHS YDHARRIS EHASTINGS HERBERT HICKMAN HICKS LAHILL MASJAMES INJEFFERY JOESEPH JOHNSON JOHNSTON ONKJONES SKLEWIS NNLINDSELL LALOCKLEY MARTIN MCENTEE MCLAUGHIN OYMOLYNEUX ITHMUNRO RRMORRIS TMUMFORD NIXON PARKER PRATT LAREID MASRENNIE NRICE RIPLEY ROBERTSON HNROMANO KSEAT SKSEN SHAW INDSIMMONS SNELLING TAYLOR TROWERS WEBLEY IWELLESLEY UXWILD UNRWILLIS

8 Methods and tools

8.1 Cheat codes

NUK THEM Select any country

ROB A BANK 100 million

TO THE TOP 100 million and select any country

COOPER TEAM Money and items

WATCH THE CLOCK Fast research completion

DO IT AGAIN Press [Ctrl] + C to finish mission

MARKS TEAM All country are yours

OWN THEM Select any country

8.2 Hexadecimal editor

An hexa editor is mandatory. Graphical, ones are handfull when you have to explore only one file a time. For an overview of all files, command line tools are better i think. For example, to see the objectives structures :

```
for i in `ls GAME*.DAT`;
do
  echo $i; od -A x -j 113974 -N 98 -t x1 --width=14 $i;
done | more
```

8.3 Opened files

To see what files are open, we can use :

```
# strace -e trace=open dosbox -conf dosbox.conf MAIN.EXE &
```

8.4 Strings

Talk abouts strings.

8.5 Memdumps

I use the Memshot tool because it seems that GAMEXX.DAT files are mapped in memory. So if we use dosbox for running Syndicate, we can inspect the dynamic of data in memory. For example :

1. - launch dosbox and enter the level 1

```
# dosbox -conf dosbox.conf MAIN.EXE &
```

2. - get the pid

```
# ps -e
```

3. - use Memshot

```
# ./MemshotFe GAME01.DAT [pid [59020 [4]]]
?> s
```

4. - move in the game
5. - use Memshot

```
?> l
```

The game reinit if you took the shot at the begining of the level !

9 References

- Infos for rnc algorithm.
- First reverse ingenering of graphic files.
- Desyndicate (reverse ingenering) wiki.
- The freesynd project.
- A fan site.

10 TODO

Probably a lot of things here !