PS2 Icon Format v0.5

©2003 Martin Akesson (ma@placid.tv)

Contents

1 Introduction

This document was created as a result of my project to create a open source interface to the EMS USB cable[1].

During this project I had to literally reverse-engineer the icon format aswell as the .psu[2] format and the USB communications protocol. All of this was very time consuming and I wish there had been more information available on the net. That is also why I now release this information so that others who seek to do similar things can concentrate more on their work than on reverse-engineering file formats.

This document will not describe how to actually render the data in an icon, you will have to buy a good 3D book for this (OpenGL red book is nice). If you use the ps2icon library available at http://ps2dev.placid.tv/ you can use that data with OpenGL without any data conversion.

1.1 License

[1]Used to transfer data from PS2 memorycard to a PC via USB cable

[2]File format used by the original EMS software

1.2 Legend

I will use a few acronyms in this document to try to describe the different datatypes used in icon structures. Unfortunately I am not fully sure of the Float32 type . . .

s8, s16, s32 Signed 8, 16 and 32 bit integers

u8, u16, u32 Unsigned 8, 16 and 32 bit integers

f16 Float 16, read as s16 and divide by 4096

f32 Float 32.

2 Icon layout

A PS2 icon is made up of several sections that all appear in linear order. Icon files always start with a icon header. The header is then followed by the actual icon data starting with vertex coordinates followed by animation and texture data.

Icon Header
Vertex segment
Animation segment
Texture segment

3 Icon header

The icon header (fig. 1) stores all the vital information we need to decode the different data segments. From this we get the number of vertices contained in the "Vertex segment" aswell as the number of animation shapes. We also get information wether the texture data is compressed or not. The header is always found at offset 0 in an icon file.

4 Vertex segment

The vertex segment contains data for all vertices in an icon. For each vertex (fig. 2) there is one set of normal and texture coordinates and one set of RGBA data. Depending on the number of animation shapes in an icon there is an equal number of vertex coordinates for each vertex. This data is then repeated for the number of defined vertices.

"Vertex coordinates" (fig. 3) for animations are stored sequentially thus shape one corresponds to the first vertex coordinate and shape eight to the eighth coordinate.

The normal for each vertex is stored directly following the last shape vertex. Normals are defined just as vertex coordinates were defined so you can use the same struct to read them. These are the normals used by the PS2 itself when rendering the icons however some icons have very strange looking normals, keep that in mind when you render them yourself and think they look strange.

The vertex normal is in turn followed by "Texture data" (fig. 4). This defines the texture coordinates for this vertex. It also defines RGBA data for this vertex.

Offset	Type	Description
0000	u32	PS2 icon file id = 0x010000
0004	u32	Animation shapes
0008	u32	Texture type (0x07 = uncompressed)
0012	u32	UNKNOWN, always "0x3F800000"
0016	u32	Number of vertices, always a multiple of 3

Figure 1: Icon header structure

1 shape	4 shapes
Vertex coordinates (1)	Vertex coordinates (1) Vertex coordinates (2) Vertex coordinates (3) Vertex coordinates (4)
Normal coordinates	Normal coordinates
Texture coordinates Vertex RGBA	Texture coordinates Vertex RGBA

Figure 2: Vertex structure

Offset	Type	Description
0000	f16	Coordinate X
0002	f16	Coordinate Y
0004	f16	Coordinate Z
0006	u16	Lighting on/off ????

Figure 3: Vertex coordinate structure

Offset	Type	Description
0000	f16	Texture X coordinate
0002	f16	Texture Y coordinate
0004	u8[4]	RGBA

Figure 4: Texture data structure

4.1 Polygon layout

Polygons in PS2 icons are always made up out of three vertices thus forming a triangle shaped polygon. Since vertices are listed sequentially it is very easy to build a polygon simply by reading vertex data three at a time. Extract all vertex data grouping them in threes until you have reached the end of the defined vertices. Rendering this data with help of OpenGL or similar should give you a nice wireframe of the icon.

5 Animation segment

This segment was a big mystery until I found a tool that extracted icon data to a text file, this is where I got some of the names for various data items. Unfortunately the tool has no known author so there is noone to give credit.

The segment is built up of a header and frame data. Each frame data holds a number of frame keys. An icon with only one frame will thus only have one set of frame data and a four frame animation will have four fram data sets.

There is a different ammount of frame keys for each frame data so you must read this for each frame.

Animation Header
Frame data
Frame keys
Frame data
Frame keys
...

5.1 Animation header

The animation header (fig. 5) tells us the frame length, animation speed, play offset and the number of frames. I have yet to really figure out what these all mean except for the number of frames which is needed to iterate through each frame data.

Offset	Type	Description
0000	u32	ID tag = 0x01
0004	u32	Frame length
0008	f32	Anim speed
0012	u32	Play offset ?
0016	u32	Number of frames

Figure 5: Animation header structure

5.2 Frame data

The frame data (fig. 6) follows directly after the "Animation header". Each set of frame data defines what shape to use for that particular frame and the number of frame keys (fig. 7). The number of frame keys may be different for each new frame data.

Offset	Type	Description
0000	u32	Shape id
0004	u32	Number of keys
0008	u32	UNKNOWN
0012	u32	UNKNOWN

Figure 6: Frame data structure

5.3 Frame keys

...

Offset	Type	Description
0000	f32	Time
0004	f32	Value

Figure 7: Key data structure

5.4 Composing an animation

To animate an icon you should display the corresponding shape for each animation frame. Timing values should be possible to figure out or simply use a generic delay. Eight frames per second seems to be a good start ...

6 Texture segment

Textures do not have any header since they always are 128x128x16 in size. They are encoded using the Playstation®TIM image format[3].

6.1 Textures

Uncompressed textures can be converted to 32bpp RGBA data by simply multiplying each red, green and blue value by eight and setting alpha to 255. You can use the table below (fig. 8) to create your own function for this, remember to convert input data to big endian[4] before applying conversion.

[3]Klarth has a description of the image format at http://rpgd.emulationworld.com/klarth
[4]See Klarths document for more information

6.2 Compressed textures

Compressed textures are compressed using a very simple RLE algorithm. The first **u32** in texture data is the size of the compressed texture data. The actual data follows directly after and is always **u16** format.

The first **u16** is always an RLE code, this code is then used to decode the actual data that follows.

- If this code is less than 0xFF00 then you should treat it as a replication counter. This means you must replicate the following **u16** data as many times.
- If the RLE code is greater than or equal to 0xFF00 then you should treat it as a data length. To get the length of data to copy you use (0xFFFF - code), this gives you a length between 0 and 255. Now copy the length of **u16** data as they are.

Repeat this until you have exhausted the compressed data and you should if all goes well have a 128x128x16 TIM image reade for use.

7 History

- Version 0.5 - Mar 19, 2003
  Major cleanup and lots of additional info
  Changed to a BSD license
- Version 0.4 - Feb 17, 2003
  Modified animation section, thanks to anonymous
- Version 0.3 - Feb 14, 2003
  Corrections on vertex and normal coordinates
  Figured out the compressed texture format
- Version 0.1 - Feb 12, 2003
  Started working on this document

Input	Apply conversion	Output
16bit data	8 * (input AND 0x1F)	8bit red
16bit data	8 * ((input >>5) AND 0x1F)	8bit green
16bit data	8 * (input >>10)	8bit blue
NULL	0xFF	8bit alpha

Figure 8: TIM conversion table