The mystery of the blend
The blender file-format explained

Author: Jeroen Bakker
Email: j.bakker@atmind.nl
Website: http://www.atmind.nl/blender
Version: 27-03-2009

Jeroen (Amsterdam, the Netherlands, 33 years old) worked as coder in the demo scene. He is interested in open source and 3d animations. At the moment he is working on products supporting impact analysis and change management around a fully automated render pipeline.

The mystery of the blend

The blend-file-format explained

Introduction

I'm working on a product what integrates Blender in a render pipeline by using the Blender command line and blend-files (.blend). The command line is not a problem as it is commonly used, but using blend-files outside Blender is difficult, because it is not that well documented. On the Internet, I've only found some clues about it on the Blender architecture pages [ref: http://www.blender.org/development/architecture/]. These were not sufficient. To really understand the file format I had to go through Blender's source code. In this article I will describe the blend-file-format with a request to tool-makers to support blend-file.

First I'll describe how Blender works with blend-files. You'll notice why the blend-file-format is not that well documented, as from Blender's perspective this is not needed. We look at the global file-structure of a blend-file (the file-header and file-blocks). After this is explained, we go deeper to the core of the blend-file, the DNA-structures. They hold the blue-prints of the blend-file and the key asset of understanding blend-files. When that's done we can use these DNA-structures to read information from elsewhere in the blend-file.

In this article we'll be using the default blend-file from Blender 2.48, with the goal to read the output resolution from the Scene. The article is written to be programming language independent and I've setup a web-site for support.

Loading and saving in Blender

Loading and saving in Blender is very fast and Blender is known to have excellent downward and upward compatibility. Ton Roosendaal demonstrated that in December 2008 by loading a 1.0 blend-file using Blender 2.48a [ref: http://www.blendernation.com/2008/12/01/blender-dna-rna-and-backward-compatibility/].

Saving complex scenes in Blender is done within seconds. Blender achieves this by saving data in memory to disk without any transformations or translations. Blender only adds file-block-headers to this data. A file-block-header contains clues on how to interpret the data. After the data, all internally Blender structures are stored. These structures will act as blue-prints when Blender loads the file. Blend-files can be different when stored on different hardware platforms or Blender releases. There is no effort taken to make blend-files binary the same. Blender creates the blend-files in this manner since release 1.0. Backward and upwards compatibility is not implemented when saving the file, this is done during loading.

When Blender loads a blend-file, the DNA-structures are read first. Blender creates a catalog of these DNA-structures. Blender uses this catalog together with the data in the file, the internal Blender structures of the Blender release you're using and a lot of transformation and translation logic to implement the backward and upward compatibility. In the source code of blender there is actually logic which can transform and translate every structure used by a Blender release to the one of the release you're using [ref: http://download.blender.org/source/blender-2.48a.tar.gz /blender-2.48a/source/blender/blenloader/intern/readfile.c lines 4946-7960]. The more difference between releases the more logic is executed.

The blend-file-format is not well documented, as it does not differ from internally used structures and the file can really explain itself.

Global file-structure

Let us look at the global file-structure. A blend-file always start with the file-header followed by file-blocks. The default blend file of Blender 2.48 contains more than 400 of these file-blocks. Each file-block has a file-block-header and data. This section explains how the global file-structure can be read.

File-Header

The first 12 bytes of every blend-file is the file-header. The file-header has information on Blender (version-number) and the PC the blend-file was saved on (pointer-size and endianness). This is required as all data inside the blend-file is ordered in that way, because no translation or transformation is done during saving. The next table describes the information in the file-header.

File-header

reference	structure	type	offset	size
identifier	char[7]	File identifier (always 'BLENDER')	0	7
pointer-size	char	Size of a pointer; all pointers in the file are stored in this format. '_' means 4 bytes or 32 bit and '-' means 8 bytes or 64 bits.	7	1
endianness	char	Type of byte ordering used; 'v' means little endian and 'V' means big endian.	8	1
version-number	char[3]	Version of Blender the file was created in; '248' means version 2.48	9	3

Endianness addresses the way values are ordered in a sequence of bytes [ref: http://en.wikipedia.org/wiki/Endianness]. Blender supports little-endian and big-endian. In a big endian ordering, the largest part of the value is placed on the first byte and the lowest part of the value is placed on the last byte. In a little endian ordering, largest part of the value is placed on the last byte and the smallest part of the value is placed on the first byte. Example: writing the integer `0x4A3B2C1Dh`, will be ordered in Big endian as `0x4Ah`, `0x3Bh`, `0x2Ch`, `0x1Dh` and be ordered in little endian as `0x1Dh`, `0x2Ch`, `0x3Bh`, `0x4Ah`.

The endianness can be different between the blend-file and the PC your using. When these are different, Blender changes it to the byte ordering of your PC. Nowadays, little-endian is the most commonly used.

The next hex-dump describes a file-header created with blender 2.48 on little-endian hardware with a 32 bits pointer length.

```
                         pointer-size  version-number
0000 0000: [42 4C 45 4E  44 45 52] [5F]  [76] [32 34 38]       BLEN DER_ v248
                identifier           endianness
```

File-block

File-blocks contain a file-block-header and data. The start of a file-block is always aligned at 4 bytes. The file-block-header describes the total length of the data, the type of information stored in the file-block, the number of items of this information and the old memory pointer at the moment the data was written to disk. Depending on the pointer-size stored in the file-header, a file-block-header can be 20 or 24 bytes long. The next table describes how a file-block-header is structured.

File-block-header

reference	structure	type	offset	size
code	char[4]	Identifier of the file-block	0	4
size	integer	Total length of the data after the file-block-header	4	4
old memory address	void*	Memory address the structure was located when written to disk	8	pointer-size (4/8)
SDNA index	integer	Index of the SDNA structure	8+pointer-size	4
count	integer	Number of structure located in this file-block	12+pointer-size	4

Code describes different types of file-blocks. The code determines with what logic the data must be read. These codes also allows fast finding of data like Library, Scenes, Object or Materials as they all have a specific code.

The size contains the total length of data after the file-block-header. After the data a new file-block starts. The last file-block in the file has code 'ENDB'.

The old memory address contains the memory address when the structure was last stored. When loading the file the structures can be placed on different memory addresses. Blender updates pointers to these structures to the new memory addresses.

SDNA index contains the index in the DNA structures to be used when reading this file-block-data. More information about this subject will be explained in the Reading scene information section.

Count tells how many elements of the specific SDNA structure can be found in the data.

The next section is an example of a file-block-header. The code `'SC'+0x00h` identifies that it is a Scene. Size of the data is 1376 bytes (0x05h X 256 + 0x60h = 1280 + 96); the old pointer is 0x0A042FA0h and the SDNA index is 139 (8 X 16 + 11). The section contains a single scene. Before we can interpreted the data of this file-block we first have to read the DNA structures in the file. The section structure DNA will show how to do that.

```
0000 4420: [53 43 00 00]  [60 05 00 00]  [A0 2F 04 0A]  [8B 00 00 00]    SC.. `... ./.. ....
0000 4430: [01 00 00 00]  [xx xx xx xx    xx xx xx xx    xx xx xx xx     .... xxxx xxxx xxxx
```

Structure DNA

Structure DNA is stored in a file-block with code 'DNA1'. It can be just before the 'ENDB' file-block. It contains all internal structures of the Blender release the file was created in. The data in this file-block must be interpreted as described in the this section. In a blend-file created with Blender 2.48a this section is 43468 bytes long and contains 309 structures. These structure can be described as C-structures. They can hold fields, arrays and pointers to other structures, just like a normal C-structure.

```
structure Scene {
        ID id; // 52 bytes long (ID is different a structure)
        Object *camera; // 4 bytes long (pointer to an Object structure)
        World *world; // 4 bytes long (pointer to a World structure)
        ...
        float cursor[3]; // 12 bytes long (array of 3 floats)
        ...
}
```

The next section describes how this information is ordered in the data of the 'DNA1' file-block.

Structure of the DNA file-block-data

repeat condition		name	type	length	description
		identifier	char[4]	4	'SDNA'
		name identifier	char[4]	4	'NAME'
		#names	integer	4	Number of names follows
for(#names)		name	char[]	?	Zero terminating string of name, also contains pointer and simple array definitions (e.g. '*vertex[3]\0')
		type identifier	char[4]	4	'TYPE' this field is aligned at 4 bytes
		#types	integer	4	Number of types follows
for(#types)		type	char[]	?	Zero terminating string of type (e.g. 'int\0')
		length identifier	char[4]	4	'TLEN' this field is aligned at 4 bytes
for(#types)		length	short	2	Length in bytes of type (e.g. 4)
		structure identifier	char[4]	4	'STRC' this field is aligned at 4 bytes
		#structures	integer	4	Number of structures follows
for(#structures)		structure type	short	2	Index in types containing the name of the structure
..		#fields	short	2	Number of fields in this structure
..	for(#field)	field type	short	2	Index in type
for end	for end	field name	short	2	Index in name

As you can see, the structures are stored in 4 arrays: names, types, lengths and structures. Every structure also contains an array of fields. A field is the combination of a type and a name. From this information a catalog of all structures can be constructed. The names are stored as how a C-developer defines them. This means that the name also defines pointers and arrays. (When a name starts with '*' it is used as a pointer. when the name contains for example '[3]' it is used as a array of 3 long.) In the types you'll find simple-types (like: 'integer', 'char', 'float'), but also complex-types like 'Scene' and 'MetaBall'. 'TLEN' part describes the length of the types. A 'char' is 1 byte, an 'integer' is 4 bytes and a 'Scene' is 1376 bytes long.

- Note: While reading the DNA you'll will come across some strange names like '(*doit)()'. These are method pointers and Blender updates them to the correct methods.
- Note: The fields 'type identifier', 'length identifier' and 'structure identifier' are aligned at 4 bytes.

The DNA structures inside a Blender 2.48 blend-file can be found at http://www.atmind.nl/blender/blender-sdna.html. If we understand the DNA part of the file it is now possible to read information from other parts file-blocks. The next section will tell us how.

Reading scene information

Let us look at the file-block we have seen earlier. The code is `'SC'+0x00h` and the SDNA index is 139. The 139th structure in the DNA is a structure of type 'Scene'. The associated type ('Scene') has the length of 1376 bytes. This is exact the same length as the data in the file-block. We can map the Scene-structure on the data of the file-blocks. But before we can do that, we have to flatten the Scene-structure.

The first field in the Scene-structure is of type 'ID' with the name 'id'. Inside the list of DNA structures there is a structure defined for type 'ID' (structure index 17). The first field in this structure has type 'void' and name '*next'. Looking in the structure list there is no structure defined for type 'void'. It is a simple type and therefore the data should be read. '*next' describes a pointer. the first 4 bytes of the data can be mapped to 'id.next'. Using this method we'll map a structure to its data. If we want to read a specific field we know at what offset in the data it is located and how much space it takes.

The next table shows the output of this flattening process for some parts of the Scene-structure. Not all rows are described in the table as there is a lot of information in a Scene-structure.

flattened SDNA structure 139: Scene

reference	structure	type	name	offset	size	description
id.next	ID	void	*next	0	4	Refers to the next scene
id.prev	ID	void	*prev	4	4	Refers to the previous scene
id.newid	ID	ID	*newid	8	4	
id.lib	ID	Library	*lib	12	4	
id.name	ID	char	name[24]	16	24	'SC'+the name of the scene as displayed in Blender
id.us	ID	short	us	40	2	
id.flag	ID	short	flag	42	2	
id.icon_id	ID	int	icon_id	44	4	
id.properties	ID	IDProperty	*properties	48	4	
camera	Scene	Object	*camera	52	4	Pointer to the current camera
world	Scene	World	*world	56	4	Pointer to the current world
set	Scene	Scene	*set	60	4	Pointer to the current set
Skipped rows						
r.sfra	RenderData	int	sfra	248	4	Start frame of the scene
r.efra	RenderData	int	efra	252	4	End frame of the scene
Skipped rows						
r.xsch	RenderData	short	xsch	326	2	X-resolution of the output when rendered at 100%
r.ysch	RenderData	short	ysch	328	2	Y-resolution of the output when rendered at 100%
r.xparts	RenderData	short	xparts	330	2	Number of x-part the renderer uses
r.yparts	RenderData	short	yparts	332	2	Number of y-part the renderer uses
Skipped rows						
sculptdata.axislock	SculptData	char	axislock	1365	1	
sculptdata.pad	SculptData	char	pad[2]	1366	2	
frame_step	Scene	int	frame_step	1368	4	
pad	Scene	int	pad	1372	4	

We can now read the X and Y resolution of the Scene. The X-resolution is located on offset 326 of the file-block-data and must be read as a short. The Y-resolution is located on offset 328 and is also a short.

- Note: An array of chars can mean 2 things. The field contains readable text or it contains an array of flags (not humanly readable).
- Note: A file-block containing a list refers to the DNA structure and has a count larger than 1. For example Vertexes and Faces are stored in this way.

Next steps

The implementation of saving in Blender is easy, but loading is difficult. When implementing loading and saving blend-files in a custom tool the difficulty is the opposite. In a custom tool loading a blend-file is easy, and saving a blend-file is difficult. If you want to read blend-files I suggest to start with understanding the the global file structure and parsing the DNA section of the file. After this is done it should be easy to read information from existing blend files like scene data, materials and meshes. When you feel familiar with this you can start creating blend-libraries using the internal Blender structures of a specific release. If you don't want to dive into the Blender source code you can find them all at http://www.atmind.nl/blender/blender-sdna.html.

There is a feature request on supporting an XML based import/export system in Blender. I dont support the request, but it is interesting to look at how this can be implemented. An XML export can be implemented with low effort as an XSD can be used as DNA structures and the data can be written into XML [see http://www.atmind.nl/blender/blender-file.zip to download JAVA example including source code]. Implementing an XML import system uses a lot of memory and CPU. If you really want to implement it, I expect that the easiest way is to convert the XML-file back to a normal blend-file and then load it using the current implementation. One real drawback is that parsing a XML based blend-file uses a lot of memory and CPU and the files can become very large.

At this moment I'm using this information in an automated render pipeline. The render pipeline is build around a web-server and SVN. When an artist commits a new blend-file in SVN, it is picked up by the web-server and it will extract resolutions, frames scenes and libraries from the blend-file. This information is matched with the other files in SVN and the blend-file will be placed in the render pipeline.