Importing Assets

Data Driven By You

Whenever we start writing a piece of software, we usually use data that's been hard-coded into the program. Nearly every 3D demo features a spinning cube.

BY STEVEN GOODWIN

With the possible exception of the 16 bit demo scene, nothing ever uses a spinning cube. We want rockets, tanks, people – anything, in fact, except spinning cubes!

In this article we look at how to move away from primitive, hard-coded, data, and learn how to resource and handle external assets – to use the geek vernacular, it becomes *data-driven*. As an example, we'll be developing a 3D mesh viewer (using 3DS files – see Box 1) from first principles, starting with a look at the file format, and step-by-step learning how to effectively and efficiently bring it into memory and manipulate it.

I Owe You Nothing

Before we can load a file we need to understand it. Whilst it is usually desirable to use a pre-written library, it would make for a very short article! We are therefore only re-inventing the wheel for the purpose of education.

However, I usually find writing a file loader helps me not only understand the format better, but allows me to make better use of the data I have to process. For all data-driven work, I recommend having good documentation and samples to hand – even if you have to pay money for it!

The documentation we shall be using to describe the format can be found at *http://www.whisqu.se/per/docs/graphics56.htm* This is a simplified version of the (now released) specification from *http://sparks.discreet.com/downloads/downloads.cfm?f=2&wf_id=45*. You do not need to rush for Konqueror just yet however, as I shall be describing the pertinent parts here.

Tall Trees In Georgia

Every 3DS file is split into chunks, and each chunk is part of a hierarchy. This means that each chunk (a block describing an individual piece of data – an object's colour, for instance) can have one or more child blocks 'inside' it, in much the same way as XML or HTML embeds tags.

Unlike XML, it is a binary format. The parent-child relationship of chunks are conceptually no different to that of the Linux file-system. You can sneak a peak at the hierarchy of a sample 3DS file in Figure 1 (p72) to see how this will look for our program.

Chunked formats allow the vendor (along with standards bodies, other developers, and end users) to add custom chunks into the format without breaking other applications; it becomes both forward, and backwards, compatible. Because it is a hierarchy, an application may omit or include entire sub-trees of data depending on how relevant they are to the file, or program in question.

The application reading the file can then ignore chunks (in their entirety) if they don't recognise (or need) one, and continue with those that they can, or want to, read. This also makes it easier to write a parser (which in turn ensures

a greater penetration for the format) since you only need to handle the elements you're interested in.

This predictable structure encompasses unpredictable data, that makes self-describing formats like this and XML loved – and others, like that of Microsoft Word, hated!

Head Music

A 3DS file chunk begins with a small header describing the data: an ID, and its size in bytes. This is then followed by the data itself (see Table 1). It really is that simple! Every step of the process involves the same set of operations:

- Read two bytes to discover the chunk ID
- Read four bytes to discover the size
- If we understand what to do with the chunk, process it!
- If we don't, skip over that chunk
- Repeat from 1, until there's no file left

Processing a chunk is also very simple – it's either data (so we load it into memory accordingly), or it's more chunks (in which case we repeat the steps above). This is known as 'parsing' the file; we use the known file structure to perform an analysis, giving meaning to the data.

High and Dry

Since we now have a basic grasp of the format we should get a 3DS file and process it by hand. This is known as 'dry running'. To do this we need a file to use as our proverbial guinea pig. Naturally, we want to be impressed by our work, so we start looking for a T-Rex from Jurassic Park, or Princess Fiona from Shrek, perhaps.

Alas, this is misplaced optimism – start small. Very small. Ideally, you should start with a model of a cube (Sorry)! At this size, it can be manually examined by hand with a tool named 'hexdump'. Each point, line and parameter can then be compared with your source model as an integrity check. Then, if you are happy with your understanding of that section of the format, you can add features (incrementally) to the model: if you only change one thing at a time, there's only one thing to go wrong!

Since I am not an artist, I shall use a 9K rocket model from the Internet (see Box 2: 3DS Models). Its validity has been independently verified by running it through several different rendering packages and other model viewers to check for errors.

Keep on Running

Having now got a 3DS file and an understanding of the format, the next step is to take a hexdump of the 3DS file and print it out. I'll wait while you do that. No really. Print it out! You may be surprised how much easier it is to work through with a hard copy of the file (see Listing 1).

Let us now perform a 'dry-run'! The first chunk we can see is 4d4d, and is ae220000 bytes long. This translates to a 'main block' chunk (see Table 2: Chunk IDs for a partial list) that is 8878 bytes in size (since it's in hex, and we must read all numbers backwards – we'll see why later). We are only 6 bytes into this chunk, so we can conclude there's another 8872 bytes of this chunk to go, and so we process it. Which, by the file specification and definition means we must read the next header!

This one has an id of 0002 (reading backwards, remember!) and is 10 bytes long. We don't know what the 0002 chunk is, so we skip over the remaining 4 bytes and read another chunk! 3D3D is

Table 1: 3DS Chunk

Start location (*)	End location (*)	Size (bytes)	Name
0	1	2	Chunk ID
2	5	4	Next chunk
6	??	??	Data in chunk

(*) An offset in bytes from the beginning of the chunk

Table 2: Chunk IDs

Name	Parent Of	ID (hex)
Main Block	-	4d4d
Mesh Data	Main Block	3d3d
Keyframes	Main Block	b000
Object Description	Mesh Data	4000
Polygon Data	Object Description	4100
Light	Object Description	4600
Camera	Object Description	4700
Vertex List	Polygon Data	4110
Face List	Polygon Data	4120

A more comprehensive list can be downloaded from: http://sparks.discreet.com/downloads

Box 1: 3DS?

3D Studio was a DOS-only package from Autodesk, and was, for many years, the industry standard for 3D computer game modelling. Although Discreet (or its parent company, Autodesk) no longer supports the 3DS format, many packages still do. This is for a good reason.

It can hold data for a number of different models, as well as storing keyframe (animation) data and dummy nodes (indicating a position on the model for attaching other objects). There are also several open source viewers and tools available, such as view3ds and lib3ds.

Box 2: 3DS models

As a programmer with little-to-no artistic ability I have to download my own personal artist from the Internet! There are several web sites that will provide you with free 3DS models for personal use.

Google will return you the following:

http://www.fantasticarts.com/3dmodels/
http://www.egypt3d.com/3D_Models/
3d_models.html

Listing 1: 3DS file header in hex

```
$ hexdump -C rocket.3ds | head
00000000 4d 4d ae 22 00 00 02 00 0a 00 00 00 03 00 00 00 |MM."............|
00000010 3d 3d b2 21 00 00 01 00 0a 00 00 00 01 00 00 00 |==.!............|
00000020 ff af 67 00 00 00 00 a0 16 00 00 00 52 6f 63 6b |..g.........Rock|
00000030  65 74 73 68 69 70 20 62  6c 75 65 00 10 a0 0f 00 |etship blue.....|
00000040  00 00 11 00 09 00 00 00  00 00 ef 20 a0 0f 00 00 |........ .....|
00000050  00 11 00 09 00 00 00 00  00 ef 30 a0 0f 00 00 00 |.........0.....|
00000060  11 00 09 00 00 00 00 00  00 40 a0 10 00 00 00 31 |.........@.....1|
00000070  00 0a 00 00 00 00 00 00  00 00 a1 08 00 00 00 01 |................|
00000080  00 81 a0 06 00 00 00 00  40 3b 21 00 00 52 6f 63 |........@;!..Roc|
00000090  6b 65 74 73 68 00 00 41  2c 21 00 00 10 41 80 0c |ketsh..A,!...A..|
```

next and easily recognised as a model chunk... and so on. This is one reason for using small files – it's a much easier process.

Having done this, we can understand what the program should be doing which makes it possible to check that it's working properly. We can now move on to implementing it!

Doin' the Do

Looking at the algorithm referenced above, it seems logical to write the ReadChunkHeader function first. However, the code to do it is not as obvious as perhaps you'd think.

```
fread(&id, sizeof(short), 1, ➥
file_ptr);
fread(&size, sizeof(long), 1, ➥
file_ptr);
```

There are three basic errors shown here, which are common to a lot of asset-handling routines.

- The size of the 'short' variable might not always be two bytes.
- No errors are handled.
- This might not be compiled on an x86 machine.

The first can be fixed simply by creating two custom types in a common header file, say mv_types.h.

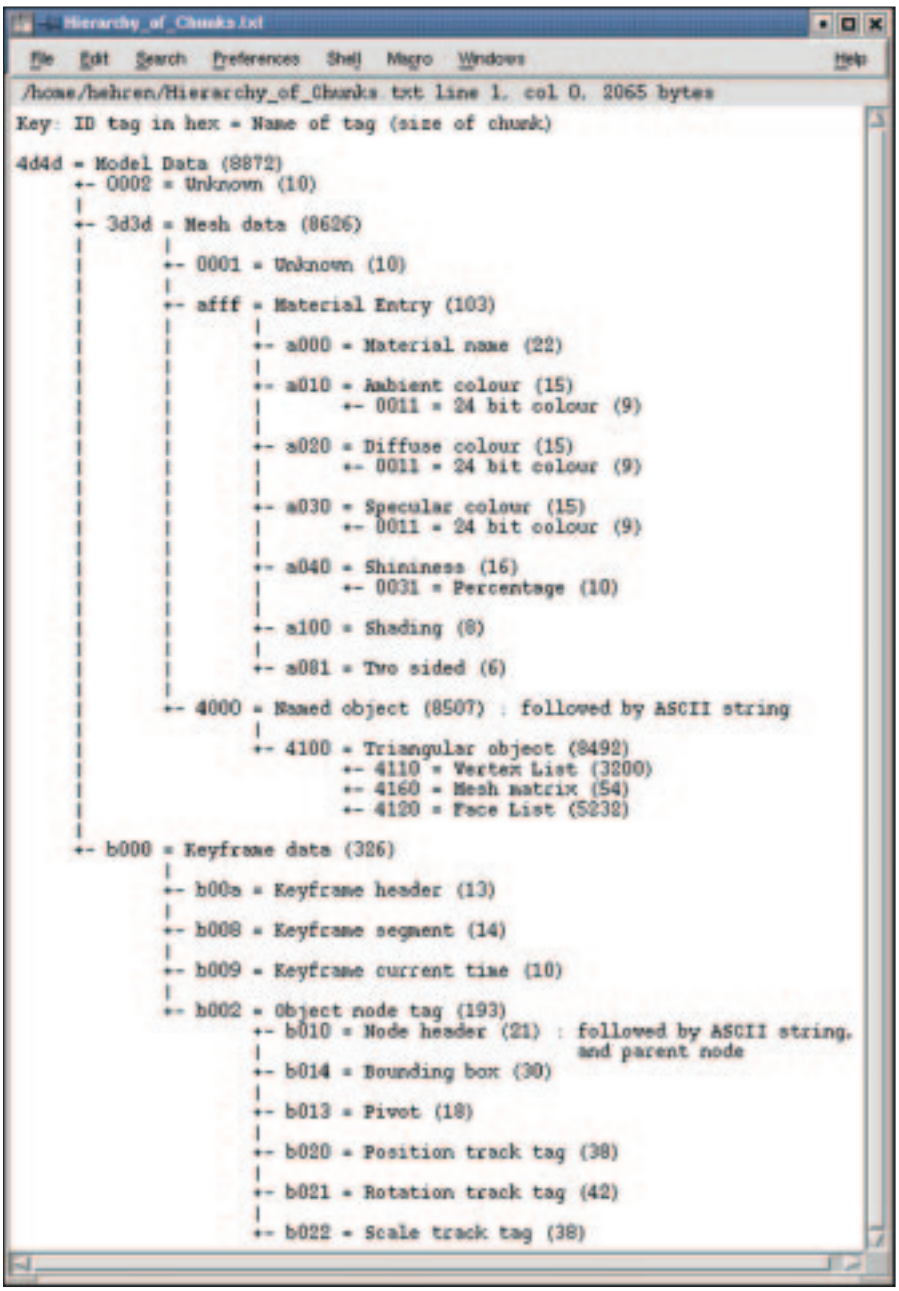

Figure 1: Sample Rocket Mesh – Hierarchy of Chunks

```
typedef unsigned short  tWORD;
typedef long    tLONG;
```

The second problem is a question of discipline: all possible circumstances must be catered for, so the error codes from 'fread' must be checked. See Box 3: Stability, for more information.

Most of the examples presented here are without full error checking, allowing us to focus on the more pertinent parts of the code. However the full source is available on the Subscriber CD.

The third situation is subtler. It is the issue of endian-ness (see LM, issue 23, p69). Basically, this is where the order of the individual bytes within a word vary between processors. This is why we had to read the chunk size backwards earlier. The x86 family are considered little endian and would work fine with the above code. The 68000 Motorola architecture would not, however.

When handling external file formats you should note carefully the endian-

Box 3: Stability

Your program will be working with odd files, of odd dimensions and odd sizes. You can no longer make assumptions about its contents, or limits. If the file supports 65536 options, then make sure your program can cope with 65536 – even if the only software generating those files is limited to 100!

Every return value (especially from file and memory functions, like fread and malloc) must be checked. It is possible the file will be corrupt, broken, or maliciously hacked and so can not be trusted!

Not only that, but once each section of a file has been found (e.g. model data), we should initialise its contents to sensible values:

```
pMesh->iNumFaces = 0;
pMesh->iNumVertices = 0;
```

This way, if a file lacks any particular component, the program will not come across uninitialised data and try to use it. C++ classes support constructors that are automatically called when an object is created, making it an ideal language for these tasks.

If you're not currently in the habit of being this paranoid – start now! Robust, stable, code like this is no bad thing. You wouldn't trust a stranger typing in your root shell, so why should you allow their data through your program without checks?

ness of the format itself – this is another good reason for printing out the hex dump of a sample file, as this makes it easy to see. You should not, however, be concerned with the endian format of the target machine since it is always possible to create endian-independent code:

```
BOOL mv_ReadWord(FILE ➥
*file_ptr, tWORD *pWord)
{
int c;
   if ((c = fgetc(file_ptr)) ➥
== -1)
      return FALSE;
   *pWord = (tWORD)c;
   if ((c = fgetc(file_ptr)) ➥
== -1)
      return FALSE;
   *pWord |= (tWORD)c<<8;
      return TRUE;
}
```

This can be extended to an equivalent mv_ReadLong function, or combined *with it* to make an all-encompassing mv_ReadChunk routine. My reasoning for this particular implementation is that by passing the address of a variable into the function, we can effectively pass two values out – the bytes read in from the disc, and an error condition (see Listing 2). If you think it's paranoia – you're right – now go and read Box 3 again!

Building Steam with a Grain of Salt

From these little acorns, great oaks of code shall grow. Referring back to the file format, we can write a parsing function quite simply – as shown in Listing 3.

This function is fairly typical of the type we'll need for write for this parser. It consists of a prototype that includes the file pointer (telling us where to get the data from), a size of the data to read, and a object pointer telling us where to put the data, once it's been read.

The main loop (lines 18 to 31), consists of each step 1 to 5, outlined above. The functionality of each step should be self-explanatory.

Lines 8 & 9 make a note of when we have to stop reading. The method I've adopted here is to pass the total block size into each function, and let it self-terminate (line 31) at the appropriate time. This isn't the nicest code in the world, but it accurately does the job! If you're designing a chunked format of your own I'd recommend adding a chunk with an ID (say 0xffff) that means 'all done, return to your parent', to make termination easier to handle.

Because we're entering a new branch of the tree, and this branch has some interesting data associated to it, we create a structure (line 11) for this data to fit into. Lines 15 and 16 prepare some default mesh data in case nothing else does. This way the render code can check the data before blindly using pointers (or data) that may be invalid. Another case of writing robust code.

This code can be used as a template for parsing other chunks, say for the

Listing 2: Passing values out

```
BOOL mv_ReadChunk(FILE *file_ptr, tWORD *pID, tLONG *pSize)
{
  if (mv_ReadWord(file_ptr, pID) == FALSE)    return FALSE;
  if (mv_ReadLong(file_ptr, pSize) == FALSE)  return FALSE;
  return TRUE;
}
```

Listing 3: Parsing function

```
01 MV_MODEL *mv_ParseMeshData
(FILE *file_ptr, tLONG mesh_size, MV_OBJECT *pObj)
02 {
03 tWORD id;
04 tLONG size;
05 tLONG end_of_block;
06 MV_MODEL     *pMesh;
07
08     end_of_block = ftell(file_ptr) + mesh_size;
       /* where the block should end... */
09     end_of_block -= sizeof(tWORD)+sizeof(tLONG);
       /*...ignoring the header */
10
11     pMesh = (MV_MODEL *)malloc(sizeof(MV_MODEL));
12     if (!pMesh)
13         return (MV_MODEL *)0;
14
15     pMesh->iNumFaces = 0;
16     pMesh->iNumVertices = 0;
17
18     do
19         {
20         mv_ReadChunk(file_ptr, &id, &size);
21
22         switch(id)
23             {
24             case    SMV_OBJECTDESCRIPTION:
25                    mv_ParseObjectBlock(file_ptr, size, pObj, pMesh);
26                             break;
27             default:
28                             mv_SkipChunk(file_ptr, size);
29             }
30          }
31     while(ftell(file_ptr) < end_of_block);
32
33     return pMesh;
34 }
```

mesh data, polygon data, or vertex list (see Table 2: Chunk IDs). The Main Block will read data, and only respond to Mesh Data, at which point it calls a similar function (called mv_ParseMesh Data) which in turn looks for Object Descriptions. This then looks for Polygon Data, Lights or Cameras.

It is best to separate these into functions because it improves readability, re-emphasises the hierarchal nature of the file, and allows you to take special cases into account.

For example, the Object Description starts with a NUL terminated ASCII string before reading the chunks. We can implement that easily and cleanly with a separate function – an example is shown in Listing 4. Having now got some code to read our data, we need to handle it in an efficient way.

Pictures Of Matchstick Men

Every 3D mesh is composed of faces. Lots of them. Each face is a triangle with three points; each point being called a vertex. So storing a mesh is simply a case of storing every vertex – of every triangle. This is normally done with two lists: a vertex list, and a face list (see Box 4 and 5).

A list of triangle vertices is rarely used because in most meshes, each face normally joins at least one other face along an edge, meaning they will share two vertices. By referencing the points in a list (as opposed to labelling them explicitly) we can save a lot of memory. For example, the rocket has 266 vertices, and 250 faces. At 12 bytes per vertex, and 6 bytes per face, the mesh requires 6,312 bytes. Whereas, if each face was stored with its vertices explicitly listed, it would take 18,720 bytes (as each face is now 36 bytes). The savings become more pronounced as meshes become larger and more complex.

So how does this help us? It tells us that the format is optimised for size, not usage. We must take this format and store it internally in a manner that helps our program. Music formats, such as MP3 and MIDI are intended to be played in a linear fashion, so their formats lend themselves instead to streaming (you may notice the slight pause when jumping into the middle of an MP3).

To start with we should test our parser by creating a simple OpenGL framework, using the data in whatever format we happen to have. As a bonus to those committed Linux Magazine readers; issue 8 (p72) includes a piece of Glut framework code that opens a window, accepts input from the keyboard and mouse and draws a teapot on screen! A quick copy and paste and it's in our project, with the glutSolidTeapot call replaced with our own draw code which looks as shown in Listing 5.

Best That You Can Do

There are two issues when it comes to choosing the best internal format. The first is for handling the object's properties (say, position and orientation) and the second is for the rendering. So is this a trade-off?

No. They should be held in different structures! The properties could be held in an MV_OBJECT structure (for instance) that details where the objects position is and what it is called. And a separate structure (MV_MODEL, for example) should describe how to draw it.

They are, after all, different entities, especially since the position will change more often than the mesh data will. By separating them in this way, the internal format can change several times, so only the rendering function needs to be updated. What's more, the MV_MODEL can describe which format of data it's using, allowing us to use different formats within the same program… for the same type of object!

Listing 4: Reading in chunks

```
BOOL mv_ParseObjectBlock(FILE *file_ptr, ➥
tLONG block_size, MV_OBJECT *pObj, MV_MODEL *pMesh)
{
tWORD id;
tLONG size;
tLONG end_of_block;

   end_of_block = ftell(file_ptr) + block_size;
   /* where the block should end... */
   end_of_block -= sizeof(tWORD)+sizeof(tLONG);
   /*...ignoring the header */

   mv_ReadString(file_ptr, pObj->szName, sizeof(pObj->szName));

   do
      {
      if (mv_ReadChunk(file_ptr, &id, &size) == FALSE)
          return FALSE;
          switch(id)
          {
          case   SMV_POLYGONDATA:
               mv_ParsePolygonData(file_ptr, size, pMesh);
               break;
          case   SMV_MESHLIGHT:
               mv_SkipChunk(file_ptr, size);
               break;
          case   SMV_MESHCAMERA:
               mv_SkipChunk(file_ptr, size);
               break;
          default:
               mv_SkipChunk(file_ptr, size);
          }
      }
   while(ftell(file_ptr) < end_of_block);
              return TRUE;
}
```

Listing 5: Copy and Paste

```
  for(i=0;i<iNumFaces;i++)
        {
        glBegin(GL_LINE_LOOP);
                glVertex3d(pVList[pFList[i].v1].x,
pVList[pFList[i].v1].y, pVList[pFList[i].v1].z);
                glVertex3d(pVList[pFList[i].v2].x,
pVList[pFList[i].v2].y, pVList[pFList[i].v2].z);
                glVertex3d(pVList[pFList[i].v3].x,
pVList[pFList[i].v3].y, pVList[pFList[i].v3].z);
        glEnd();
        }
```

Listing 6: More modules

```
MV_OBJECT *Obj_CreateObject(void)
{
MV_OBJECT *pObj;

    pObj = (MV_OBJECT *)malloc(sizeof(MV_OBJECT));
    if (pObj == 0)
        return (MV_OBJECT *)0;

    pObj->pos.x = pObj->pos.y = pObj->pos.z = 0;
    Obj->xangle = pObj->yangle = pObj->zangle = 0;
    pObj->pMesh = 0;
    return pObj;
}
```

```
typedef struct {
  char    szName[256];
  MVERTEX     position;
  float           xangle,
yangle, zangle;
  MV_MODEL    *pMesh;
  } MV_OBJECT;
```

This object should have its own set of functions to manipulate it, keeping it modular and distinct from the file parsing code. Again, this distance allows features to be added and changed without a major code overhaul (see Listing 6).

And a set of manipulation functions would not go amiss, as in our example:

```
void Obj_SetPosition(MV_OBJECT
pObj, float x, float y, float z)
{
    pObj->pos.x = x;
    pObj->pos.y = y;
    pObj->pos.z = z;
}
```

Improving the format can be done (in OpenGL) using 'array elements' or 'display lists'. These should be computed on load and stored in place of the mesh data we loaded above. The internal methods, or structure, are not important unless you're an OpenGL programmer (it's the same data, but in a different format).

What is important, however, is that such a format exists and may have no relation to the 3DS file we started with! You should arrange program data in a format suitable for the program – not the disc. We are fairly lucky in so much as a good OpenGL format can be created quite easily by expanding the face vertices with fairly minimal work on our part.

Alternative render code using 'array elements'. Made possible because we load the vertices from the 3DS file in the correct manner initially (see Listing 7).

We could also use our MV_MODEL structure to store the colour (or graphic image) for each mesh face within this structure, or add the face normal (the direction it's facing) to perform hidden face removal, or produce better lighting.

This is information that could either be present within the file format, or computed from existing data. We simply put the data at the fingertips of the render code, where it deserves to be. Whatever format results, we could (nay, should!) save the data out as a raw block that can be loaded in (much quicker) next time. These resultant files are platform dependant and target ready: meaning we load them in, set up our pointers and *wham!* away we go! An example is shown in Listing 8.

In a larger project, these files may be packaged with others (in much the same manner as a 'tar' file) to speed up loading, and ease distribution.

As we've seen, there can be a lot of work in parsing a file format and storing it efficiently in memory. When it's done, your programs take on an extra edge of professionalism and the next step towards the big time. ■

Box 4: Vertex List

```
-21.000000   0.000000 100.000000
-34.000000   5.000000  73.000000
-31.000000   8.000000  73.000000
  … etc …
```

Listing 7: Render

```
glEnableClientState
(GL_VERTEX_ARRAY);
glVertexPointer(3, GL_FLOAT, 0,
(void *)pCurrMesh->pVertexList);
 for(i=0;i<pCurrMesh-
>iNumFaces;i++)
 {
 glBegin(GL_LINE_LOOP);
   glArrayElement( pFList[i].v1);
   glArrayElement( pFList[i].v2);
   glArrayElement( pFList[i].v3);
 glEnd();
 }
```

Box 5: Face List

```
   18     1     0
    2     1     0
    3     2     0
… etc …
```

Listing 8: Wham

```
fwrite(&iNumVertices, sizeof(iNumVertices), 1, file_ptr);
fwrite(pVertexList, sizeof(MVERTEX), iNumVertices, file_ptr);

fwrite(&iNumFaces, sizeof(iNumVertices), 1, file_ptr);
fwrite(pFaceList, sizeof(MFACE), iNumFaces, file_ptr);
```