The Quake II's MD2 file format

written by David Henry, december 21st of 2002

Introduction

“Yeah a new MD2 tutorial yet...” Yes but mine will show you how to render them by a different way ;-) But what the heck is an MD2 anyway? the MD2 file format is a 3D model file format used in Id Software's Quake II engine! And here I'll show you how to load and display it to the screen using OpenGL!

You probably think “Damn, this guy stucks in the 1997 ol' days” but there are good reasons to use it. First because the MD2 file format is a good 3D model file format for learning because of it simplicity (if you're a beginner, you may not understand this, but look at other model file formats and you'll see ;-)) and then, because this format is absolutely free!!! (but not models, be careful!).

So what? here is a little overview of what we'll see in this article:

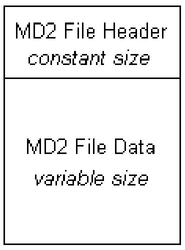
- The MD2 File Format
- Developping a CMD2MODEL class
- Reading and storing a MD2 model
- Displaying it to the screen
- Animation

Also, the source code is totaly free and downloadable at the end of this document.

Before starting I would like to say that I am assuming that you're familiar with the C++ and the OpenGL API. Ok let's start with some theory about the MD2 file format!

The MD2 File Format

Like most file formats, the MD2 file format is composed of two things: a file header and the data. The header contains some very important variables used when loading the file like the size of the file data to load, a magic number or the version of the format, so its size must be always the same. That's why generally the header is a structure. In opposition, the size of the data can vary from one file to another and contains multiple structures for vertices, triangles, texture coordinates, etc... Figure 1 represents the file architecture:



Here is the MD2 header structure definition (called **`md2_t`**):

```
// md2 header
typedef struct
{
    int     ident;              // magic number. must be equal to "IDP2"
    int     version;            // md2 version. must be equal to 8

    int     skinwidth;          // width of the texture
    int     skinheight;         // height of the texture
    int     framesize;          // size of one frame in bytes

    int     num_skins;          // number of textures
    int     num_xyz;            // number of vertices
    int     num_st;             // number of texture coordinates
    int     num_tris;           // number of triangles
    int     num_glcmds;         // number of opengl commands
    int     num_frames;         // total number of frames

    int     ofs_skins;          // offset to skin names (64 bytes each)
    int     ofs_st;             // offset to s-t texture coordinates
    int     ofs_tris;           // offset to triangles
    int     ofs_frames;         // offset to frame data
    int     ofs_glcmds;         // offset to opengl commands
    int     ofs_end;            // offset to end of file

} md2_t;
```

Ok I'll explain briefly all these variables.

First you have what is called a “magic number”. When loading the file in memory, check this value and be sure it's equal to “IPD2”. If it isn't equal to “IPD2” then you can close the file and stop the loading. It is not an MD2 file. The next variable indicates the file version and must be equal to 8.

Then we've got the dimensions of the texture (respectively the width and the height). We won't use these variables because the md2 model's texture is stored in another file, most of the time a PCX or a TGA file, and we obtain the texture's dimensions from these files.

`framesize` specifies the size in bytes of each frame. Yes but what the hell is a frame? A frame is like a picture in a movie. Looping many frames at a certain speed, you get an animation! So one frame stores model's vertices and triangles in a particular position. So a classic md2 file is composed of 199 frames distributed in 21 animations. A frame contains a list of vertices for all triangles of this frame (each frame has the same number of triangles). For the moment, remember that we will need this variable to know how much memory we need to allocate for storing each frame.

The next variables are quite similar.

`num_skins` tells you the number of textures avalaible for this model. For exemple, you can have a texture for the red team and an other for the blue team in a team game. The name of each texture is stored in an array of 64 bytes at the **`ofs_skins`** offset in the file. However we won't use these names because they are specific to the Quake2 directory, for exemple: “player/ogro/igdosh.pcx”.

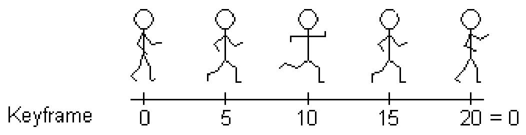
`num_xyz` is the total amount of vertices of the model. It correspond to the sum of the number of vertices in each frame.

`num_st` is the number of texture coordinates which are stored in the file at the offset **`ofs_st`**. Note that this number isn't inevitably equal to the number of vertices. In our code we will use another way to obtain these textures coordinates and in real-time so we won't need to load the texture coordinate array from the file.

`num_tris` gives us the total amount of triangles in the model.

`num_glcmds` is the number of OpenGL command. The GL command list is an array of integers that allows us to render the model using only triangle fans and triangle strip (**`GL_TRIANGLE_STRIP`** and **`GL_TRIANGLE_FAN`**), instead of classic triangles (**`GL_TRIANGLES`**). GL commands are very powerful. It is easy to get a rendering about 10 or 15 fps faster!

Finaly there is **`num_frames`** yet. It specifies the total number of frames that holds the model. In fact, each of them are refered to as keyframes, which are frames taken from discrete time intervals because it would be impossible to hold 200 or 300 frames per animation! Consequently, we only keep some of these for each animation and we'll calculate all intermediate frames we'll need when rendering, using linear interpolation (I'll explain that later). Look at Figure 2 to see an exemple. Here is represented a simplistic model with one animation which need 20 frames to be fully displayed, but only 5 of these are kept. Frames from number 1 to 4, 6 to 9, 11 to 14 and 16 to 19 must be calculated before rendering to get a smooth animation.



The last bloc of header's variables contains offsets to access to different types of model's data. **`ofs_skins`** points on model's texture names, **`ofs_st`** on texture coordinates, **`ofs_tris`** points on vertices, **`ofs_frames`** on the first frame of the model, **`ofs_glcmds`** on OpenGL command list and of course, **`ofs_end`** which tells you the end of the file (we won't need it).

Yeah we've finished with the header! Now let's look at structures needed to store model data! Yes, like the header, we'll use structures to hold frames, vertices and OpenGL commands.

The first data type very useful in most 3D applications is the vector! We don't need a complicated Vector Class so I will keep things simple: a simple array of 3 float will represent a vector!

```
typedef float vec3_t[3];
```

Each model is composed of **`(num_frame * num_xyz)`** vertices. Here is the structure that hold a single vertex:

```
// vertex
typedef struct
{
    unsigned char   v[3];                // compressed vertex (x, y, z) coordinates
    unsigned char   lightnormalindex;    // index to a normal vector for the lighting

} vertex_t;
```

You may have noticed that **`v[3]`** contains vertex' (x,y,z) coordinates and because of the unsigned char type, these coordinates can only range from 0 to 255. In fact these 3D coordinates are compressed (3 bytes instead of 12 if we would use float or vec3_t). To uncompress it, we'll use other data proper to each frame. **`lightnormalindex`** is an index to a precalculated normal table. Normal vectors will be used for the lighting.

The last piece of information needed for a vertex is its texture coordinates. They are also packed into a structure:

```
// texture coordinates
typedef struct
{
    short    s;
    short    t;

} texCoord_t;
```

Like for vertices, data is compresed. Here we use short (2 bytes) instead of float (4 bytes) for storing texture coordinates. But to use them, we must convert them to float because texture coordinates range from 0.0 to 1.0, and if we kept short values, we could have only 0 or 1 and any intermediate value! So how to uncompress them? It's quite simple. Divide the short value by the texture size:

```
RealST[i].s = (float)texCoord[i].s / header.skinwidth;
RealST[i].t = (float)texCoord[i].t / header.skinheight;
```

supposing that RealST is an object of a structure similar to **`texCoord_t`** but with float instead of short types and **`texCoord`** is an array of **`texCoord_t`** loaded from a MD2 file.

Each frame (or keyframe) of the model is stored in a structure defined like that:

```
// frame
typedef struct
{
    float       scale[3];       // scale values
    float       translate[3];   // translation vector
    char        name[16];       // frame name
    vertex_t    verts[1];       // first vertex of this frame

} frame_t;
```

Each frame is stored as a frame_t structure, holding all specific data to this frame. So a classical model (i.e. a player model) has 199 **`frame_t`** objects. I said one minute ago that we will uncompress vertices using frame data. Here is the data! To uncompress each vertex, we will scale it multiplying its coordinates by the **`scale[3]`** values and then translate it by the **`translate[3]`** vector (we could also write **`vec3_t translate`** instead of **`float translate[3]`**).

`name[16]` is simply the name of the frame. Finaly, **`verts[1]`** is the first vertex of the frame. Other vertices of this frame are stored just after the first vertex, so we can access to them like that:

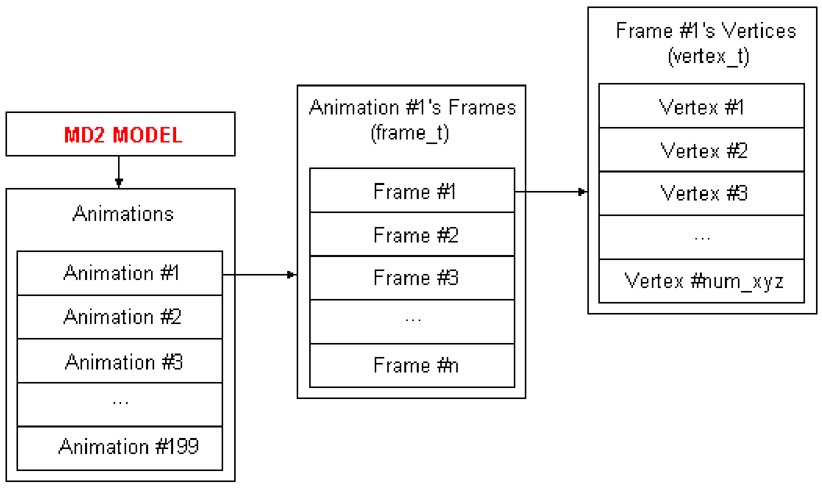
```
frame.verts[ 2 ] // get the second vertex of the frame
frame.verts[ i ] // get the i th vertex of the frame
frame.verts[ num_xyz - 1 ] // get the last vertex of this frame
```

Thus we get the real vertex coordinates:

```
vertex.x = (frame.verts[i].v[0] * frame.scale[0]) + frame.translate[0]
vertex.y = (frame.verts[i].v[1] * frame.scale[1]) + frame.translate[1]
vertex.z = (frame.verts[i].v[2] * frame.scale[2]) + frame.translate[2]
```

where **`i`** ranges from 0 to **`(num_xyz - 1)`**.

Look at Figure 3 to see a representation of relations between animations, frames and vertices:



So each animation contains **`n`** frames which contain each **`num_xyz`** vertices.

We need now to link each vertex with its texture coordinates couple. But instead of linking one **`vertex_t`** with one **`texCoord_t`**, they are linked by triplet to form a triangle, or a mesh:

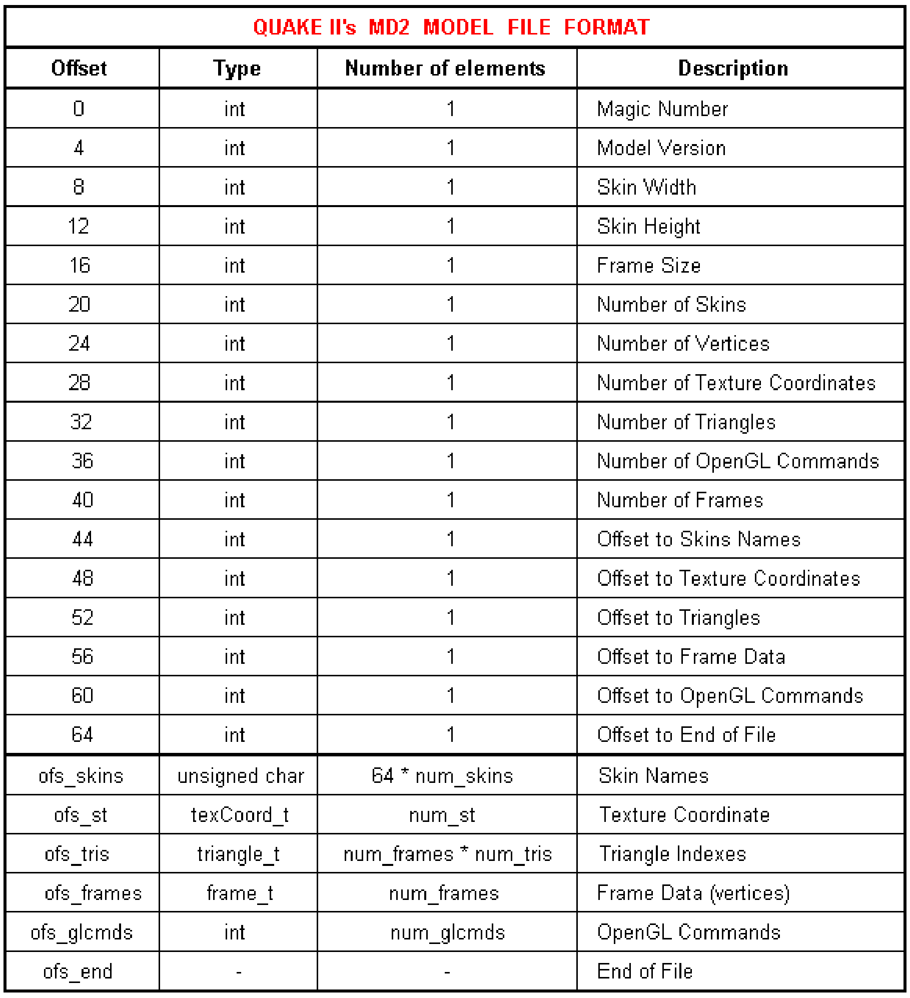
```
// triangle
typedef struct
{
    short   index_xyz[3];    // indexes to triangle's vertices
    short   index_st[3];     // indexes to vertices' texture coorinates

} triangle_t;
```

This is how they are stored in the file. Notice that **`index_xyz`** and **`index_st`** are indexes on the data and not the data themselves! The data must be stored separately in **`vertex_t`** and **`texCoord_t`** arrays or if you prefer uncompress them during the model loading, in similar structures with float types. Supposing that **`Vertices[]`** is an array of **`vertex_t`**, **`TexCoord[]`** an array of **`texCoord_t`**, **`Meshes[]`** an array of **`triangle_t`** and **`anorms[]`** an array of **`vec3_t`** which stores all precalculed normal vectors. You could draw the model using this method:

```
glBegin( GL_TRIANGLES );
  // draw each triangle
  for( int i = 0; i < header.num_tris; i++ )
  {
      // draw triangle #i
      for( int j = 0; j < 3; j++ )
      {
          // k is the frame to draw
          // i is the current triangle of the frame
          // j is the current vertex of the triangle

          glTexCoord2f( (float)TexCoord[ Meshes[i].index_st[j] ].s / header.skinwidth,
                        (float)TexCoord[ Meshes[i].index_st[j] ].t / header.skinheight );

          glNormal3fv( anorms[ Vertices[ Meshes[i].index_xyz[j] ].lightnormalindex ] );

          glVertex3f( (Vertices[ Meshes[i].index_xyz[j] ].v[0] * frame[k].scale[0]) + frame[k].translate[0],
                      (Vertices[ Meshes[i].index_xyz[j] ].v[1] * frame[k].scale[1]) + frame[k].translate[1],
                      (Vertices[ Meshes[i].index_xyz[j] ].v[2] * frame[k].scale[2]) + frame[k].translate[2] );
      }
  }
glEnd();
```

Ok this is not very easy to visualize and the method uses **`GL_TRIANGLES`**. We can get better performances using **`GL_TRIANGLE_SRTIP`** and **`GL_TRIANGLE_FAN`**. But how? Using the OpenGL commands!

This is all about data structures! I can now show you the entire file architecture:



QUAKE II's MD2 MODEL FILE FORMAT

Offset	Type	Number of elements	Description
0	int	1	Magic Number
4	int	1	Model Version
8	int	1	Skin Width
12	int	1	Skin Height
16	int	1	Frame Size
20	int	1	Number of Skins
24	int	1	Number of Vertices
28	int	1	Number of Texture Coordinates
32	int	1	Number of Triangles
36	int	1	Number of OpenGL Commands
40	int	1	Number of Frames
44	int	1	Offset to Skins Names
48	int	1	Offset to Texture Coordinates
52	int	1	Offset to Triangles
56	int	1	Offset to Frame Data
60	int	1	Offset to OpenGL Commands
64	int	1	Offset to End of File
ofs_skins	unsigned char	64 * num_skins	Skin Names
ofs_st	texCoord_t	num_st	Texture Coordinate
ofs_tris	triangle_t	num_frames * num_tris	Triangle Indexes
ofs_frames	frame_t	num_frames	Frame Data (vertices)
ofs_glcmds	int	num_glcmds	OpenGL Commands
ofs_end	-	-	End of File

Developing a CMD2Model Class

Thanks to OpenGL commands, we won't need to use **triangle_t** and **texCoord_t** structures, because all this is also included in the OpenGL command list, which we'll use. I covered them in case you don't want to use OpenGL commands or if you don't want to render using OpenGL.

We're now ready to develop a class which will represent an MD2 model object. Here is the prototype:

```
// ============================================
// CMD2Model - MD2 model class object.
// ============================================

class CMD2Model
{
public:
    // constructor/destructor
    CMD2Model( void );
    ~CMD2Model( void );


    // functions
    bool    LoadModel( const char *filename );
    bool    LoadSkin( const char *filename );

    void    DrawModel( float time );
    void    DrawFrame( int frame );

    void    SetAnim( int type );
    void    ScaleModel( float s ) { m_scale = s; }


private:
    void    Animate( float time );
    void    ProcessLighting( void );
    void    Interpolate( vec3_t *vertlist );
    void    RenderFrame( void );


public:
    // member variables
    static vec3_t   anorms[ NUMVERTEXNORMALS ];
    static float    anorms_dots[ SHADEDOT_QUANT ][256];

    static anim_t   animlist[21];       // animation list


private:
    int             num_frames;         // number of frames
    int             num_xyz;            // number of vertices
    int             num_glcmds;         // number of opengl commands

    vec3_t          *m_vertices;        // vertex array
    int             *m_glcmds;          // opengl command array
    int             *m_lightnormals;    // normal index array

    unsigned int    m_texid;            // texture id
    animState_t     m_anim;             // animation
    float           m_scale;            // scale value

};
```

Each MD2 model will be a **CMD2Model** object. Hum this class looks quite strange more especially as there is nor **frame_t** object neither **vertex_t** object! And where are texture coordinates stored? some explanations are required...

First we've got classic constructor and destructor that initialize all member variables to 0 (excepted m_scale) and free allocated memory during the loading of data.

What about functions? I think they are self-explanatory. **LoadModel()** will load the model from a file and initialize it and **LoadSkin()** will load the texture and initialize **m_texid**.

DrawModel() is the function we'll use to draw the animated model with all transformation needed. The time parameter is needed to calculate the frame to render from the actual animation.

DrawFrame() is the function we'll use to draw the model at a specific frame.

SetAnim() and **ScaleModel()** are used to set the current animation and the scale value.

Animate(), **ProcessLighting()**, **Interpolate()** and **RenderFrame()** are private functions because they must be only used inside the public **DrawModel()** function. They process all calculations to render the proper frame interpolated and lightened.

Now member variables. anorms is an array of precalculated normal vectors. Each vertex will have an index stored in the **m_lightnormals** array to access to its own normal vector. **anorms_dots** looks like **anorms** but this time it stores precalculated dot products. We will need it when processing lighting. **animlist** is an array of animations. Here is the **anim_t** structure prototype:

```
// animation
typedef struct
{
    int     first_frame;            // first frame of the animation
    int     last_frame;             // number of frames
    int     fps;                    // number of frames per second

} anim_t;
```

You may have noticed that these three last member variables are static. This is because they are the same for every MD2 model so we need only one copy of them.

Then we have **num_frames** which stores the total number of frames, **num_xyz** the number of vertices per frame and **num_glcmds** the number of OpenGL commands.

m_vertices holds 3D coordinates in floating point number for each vertex. The **m_glcmds** array stores OpenGL command list. For the moment, don't be afraid of these “OpenGL commands”, just think that it is magic. I'll explain when we'll need them to draw model's meshes. For these three last array, we will allocate memory dynamically.

m_texid will store the OpenGL texture ID. **m_anim** store information about the current animation to play. It is an **animState_t** object (look at comments for a brief description):

```
// animation state
typedef struct
{
    int     startframe;              // first frame
    int     endframe;                // last frame
    int     fps;                     // frame per second for this animation

    float   curr_time;               // current time
    float   old_time;                // old time
    float   interpol;                // percent of interpolation

    int     type;                    // animation type

    int     curr_frame;              // current frame
    int     next_frame;              // next frame

} animState_t;
```

Finaly, **m_scale** stores the scale value for all axes. This is better to scale vertices by multiplying them with the **m_scale** value than using **glScalef()** because this function would scale normal vectors also and would bring to strange lighting effects.

I have said that we won't use neither **triangle_t** nor **texCoord_t** structures, but what about **vertex_t** and **frame_t** structures ? We'll only use these when loading the model in the **LoadModel()** function and transform frame data to be stored in **m_vertices** and **m_lightnormals** arrays.

Before ending this section, I want to give you constructor and destructor definitions:

```
// ----------------------------------------------
// constructor - reset all data.
// ----------------------------------------------

CMD2Model::CMD2Model( void )
{
    m_vertices      = 0;
    m_glcmds        = 0;
    m_lightnormals  = 0;

    num_frames      = 0;
    num_xyz         = 0;
    num_glcmds      = 0;

    m_texid         = 0;
    m_scale         = 1.0;

    SetAnim( 0 );
}


// ----------------------------------------------
// destructor - free allocated memory.
// ----------------------------------------------

CMD2Model::~CMD2Model( void )
{
    delete [] m_vertices;
    delete [] m_glcmds;
    delete [] m_lightnormals;
}
```

For the constructor, we set all member variables (excepts static variables and **`m_scale`**) to 0. We initialize **`m_scale`** to 1.0 because if we would set it to 0, there would be nothing rendered! For the destructor, we just desallocate memory...

Ok, we're ready to start really! Let's move to the next section: loading a MD2 model file!

Reading and storing a MD2 model

We load an MD2 model passing its filename in parameter to the **`LoadModel()`** function. It returns true if success and false if something fails during the loading. Look at the first part of the function:

```
// ----------------------------------------------
// LoadModel() - load model from file.
// ----------------------------------------------

bool CMD2Model::LoadModel( const char *filename )
{
    std::ifstream   file;           // file stream
    md2_t           header;         // md2 header
    char            *buffer;        // buffer storing frame data
    frame_t         *frame;         // temporary variable
    vec3_t          *ptrverts;      // pointer on m_vertices
    int             *ptrnormals;    // pointer on m_lightnormals


    // try to open filename
    file.open( filename, std::ios::in | std::ios::binary );

    if( file.fail() )
        return false;

    // read header file
    file.read( (char *)&header, sizeof( md2_t ) );


    /////////////////////////////////////////////
    //      verify that this is a MD2 file

    // check for the ident and the version number

    if( (header.ident != MD2_IDENT) && (header.version != MD2_VERSION) )
    {
        // this is not a MD2 model
        file.close();
        return false;
    }

    /////////////////////////////////////////////
```

First we define some local variables that we'll need during the loading of the model. **`file`** is a file stream to extract model data from a file. header is a **`md2_t`** object which will store the header of the model file. Then we have **`buffer`**. It's a large buffer for storing all frame data. The three last variables are different pointers to access data from **`buffer`**.

We start by trying to open the specified file in read only mode and return false if it fails. The file opened, we then load the model header. Thus we can check for the magic number (the ident) and the version of the model to be sure that it is a MD2 file. The ident must allways equal to “IDP2” and the version of the model to 8. So we can define **`MD2_IDENT`** and **`MD2_VERSION`** like this:

```
// magic number "IDP2" or 844121161
#define MD2_IDENT               (('2'<<24) + ('P'<<16) + ('D'<<8) + 'I')

// model version
#define MD2_VERSION             8
```

Notice that we could also check for the magic number comparing the ident to 844121161 or using the **`strcmp()`** function (ident must then be defined as a char [4]).

Now that we are sure that it's a valid MD2 file, we can continue te loading:

```
    // initialize member variables
    num_frames  = header.num_frames;
    num_xyz     = header.num_xyz;
    num_glcmds  = header.num_glcmds;


    // allocate memory
    m_vertices      = new vec3_t[ num_xyz * num_frames ];
    m_glcmds        = new int[ num_glcmds ];
    m_lightnormals  = new int[ num_xyz * num_frames ];
    buffer          = new char[ num_frames * header.framesize ];


    /////////////////////////////////////////////
    //          reading file data

    // read frame data...
    file.seekg( header.ofs_frames, std::ios::beg );
    file.read( (char *)buffer, num_frames * header.framesize );

    // read opengl commands...
    file.seekg( header.ofs_glcmds, std::ios::beg );
    file.read( (char *)m_glcmds, num_glcmds * sizeof( int ) );

    /////////////////////////////////////////////
```

Here we first initialize our numerical variables from the model header. Then we can allocate necessary memory for our **`m_vertices`**, **`m_glcmds`**, **`m_lightnormals`** and buffer arrays. Notice that there is the same number of elements for **`m_vertices`** and **`m_lightnormals`**. Thus we can have one index for a vertex which would points both on its 3D coordinates and in its normal index. We'll get this pointer from the **`m_glcmds`** array.

Memory is allocated so we can read data from the file. Before reading data, we move to the position specified by header's offsets. We only read frame data and OpenGL commands. We'll initialize **`m_vertices`** and **`m_lightnormals`** with buffer like that:

```
    // vertex array initialization
    for( int j = 0; j < num_frames; j++ )
    {
        // adjust pointers
        frame       = (frame_t *)&buffer[ header.framesize * j ];
        ptrverts    = &m_vertices[ num_xyz * j ];
        ptrnormals  = &m_lightnormals[ num_xyz * j ];

        for( int i = 0; i < num_xyz; i++ )
        {
            ptrverts[i][0] = (frame->verts[i].v[0] * frame->scale[0]) + frame->translate[0];
            ptrverts[i][1] = (frame->verts[i].v[1] * frame->scale[1]) + frame->translate[1];
            ptrverts[i][2] = (frame->verts[i].v[2] * frame->scale[2]) + frame->translate[2];

            ptrnormals[i] = frame->verts[i].lightnormalindex;
        }
    }
```

This is the more difficult to understand. First we loop through each frame. For each frame, we extract frame data from buffer using our **`frame_t*`** pointer defined at the beginning of the function. We also adjust our pointers on **`*m_vertices`** and **`*m_lightnormals`** so that they point at the beginning of where must be stored the current frame data.

Then we loop through each vertex of the current frame that we are processing. We initialize vertex's 3D coordinates with the formula I explained before, in the section about the MD2 file format. We also initialize the normal index stored in the vertex's **`vertex_t`** structure.

We have initialized our three numerical variables and our three data arrays, so we've finished with the model file! Was it so difficult? We have just to close the file, free **`buffer`** and return true:

```
    // free buffer's memory
    delete [] buffer;

    // close the file and return
    file.close();
    return true;
}
```

Now what about the texture? For the texture, we only have its texture ID to store in **`m_texid`**. MD2's textures are stored in classical TGA or PCX files. Loading a texture from a file is beyond the scope of this article, so I won't cover how it works. I assume that you have a function which loads a texture from a file and returns a valid ID. In the source code that you can download, I have written a simple Texture Manager which can loads and initializes a texture from a bitmap, targa of pcx file. Here is how we load the texture with the **`LoadSkin()`** function:

```
// ----------------------------------------------
// LoadSkin() - load model texture.
// ----------------------------------------------

bool CMD2Model::LoadSkin( const char *filename )
{
    m_texid = LoadTexture( filename );

    return (m_texid != LoadTexture( "default" ));
}
```

Just a few words about my texture manager: first I have written an inline **`LoadTexture()`** function for easier code reading. This function access to the Texture Manager's **`LoadTexture()`** function. The Texture Manager is a singleton. When initializing, it creates a default texture (which is a black and white checker). When loading a texture from a file, it first checks if the texture has already been loaded. If yes it returns the texture ID, else it tries to open the file and load it. If the loading fails, or the file doesn't exist, it returns the default texture ID. So when calling **`texmgr.LoadTexture( "default" )`**, this doesn't load a texture but returns the default texture ID. When returning, we check the texture ID this function gived us when loading our texture and return false if it equals to the default texture ID.

This is all for this section. We have loaded all data we need.

Drawing the model

It's time to render the model we've loaded!

The main drawing model function is **`DrawModel()`**. However this function won't render directly the model, but will process some transformations and calculus before calling the **`RenderFrame()`** function. Let's look at the function definition:

```
// ----------------------------------------------
// DrawModel() - draw the model.
// ----------------------------------------------

void CMD2Model::DrawModel( float time )
{
    glPushMatrix();
        // rotate the model
        glRotatef( -90.0, 1.0, 0.0, 0.0 );
        glRotatef( -90.0, 0.0, 0.0, 1.0 );

        // render it on the screen
        RenderFrame();
    glPopMatrix();
}
```

Ok, there only are two simple rotations before rendering and for the moment, the **`time`** parameter is not used... But we'll update this function later, when animating! We need to rotate the model on the X and Z axis because it isn't stored using OpenGL axis. You can comment the two calls to **`glRotatef()`** to see why we do that :-)

Remember the **`m_scale`** value and **`ScaleModel()`** function I discussed earlier. To avoid having a huge model at the screen once the rendering finished, we scale each vertices of the current frame we're rendering. The scaling operation is processed by the **`Interpolate()`** function called by **`RenderFrame()`**. Normaly vertex interpolation have nothing to do with scaling, but because for the moment we are not animating, the **`Interpolate()`** function will only scale vertices. Later we'll rewrite it to really interpolate vertices from two frames. Here is the code:

```
// ----------------------------------------------
// Interpolate() - interpolate and scale vertices
// from the current and the next frame.
// ----------------------------------------------

void CMD2Model::Interpolate( vec3_t *vertlist )
{
    for( int i = 0; i < num_xyz ; i++ )
    {
        vertlist[i][0] = m_vertices[ i + (num_xyz * m_anim.curr_frame) ][0] * m_scale;
        vertlist[i][1] = m_vertices[ i + (num_xyz * m_anim.curr_frame) ][1] * m_scale;
        vertlist[i][2] = m_vertices[ i + (num_xyz * m_anim.curr_frame) ][2] * m_scale;
    }
}
```

This function initializes an array of vertices with the current frame scaled vertices. So the **`RenderFrame()`** function will use the array passed in parameter for rendering and won't use the original **`m_vertices`** array directly. It will also be easier manipulating vertlist than **`m_vertices`**.

Now I would like to talk about lighting a little. There is two way to light the model. The first way is using OpenGL lighting functions. For that, we just need to set the normal of each vertex we're rendering. There is no difficulty, the index stored in **`m_lightnormals`** give us a precalculated normal from the anorms table.

The second way to light the model is using **`glColor()`** for each vertex to fake lighting and shading. Also this is the way used in Quake II's engine. For this method, there is some work to do. So we'll put all it in the **`ProcessLighting()`** function, called by **`RenderFrame()`** like the **`Interpolate()`** function. But before, we need to create some global variables and initialize others...

```
// number of precalculated normals
#define NUMVERTEXNORMALS        162

// number of precalculated dot product results (for lighting)
#define SHADEDOT_QUANT          16
```

```
// precalculated normal vectors
vec3_t   CMD2Model::anorms[ NUMVERTEXNORMALS ] = {
#include    "anorms.h"
};

// precalculated dot product results
float    CMD2Model::anorms_dots[ SHADEDOT_QUANT ][256] = {
#include    "anormtab.h"
};

static float    *shadedots = CMD2Model::anorms_dots[0];
static vec3_t   lcolor;

/////////////////////////////////////////////////

vec3_t          g_lightcolor    = { 1.0, 1.0, 1.0 };
int             g_ambientlight  = 32;
float           g_shadelight    = 128;
float           g_angle         = 0.0;

/////////////////////////////////////////////////
```

The precalculated normal and dot result lists are two big and not very interesting to show, so they are stored in header files that we simply include to initialize static arrays.

shadedots is a pointer which will ajusted in the **ProcessLighting()** function. It will pointer in an element of the **anorms_dots** array.

lcolor will store RGB values for the final light color.

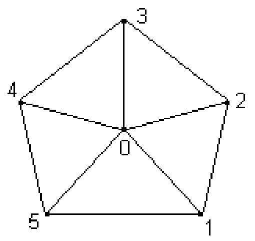
Finaly, the three last global variables are for the ambient light value (which range from 0 to 255), shading value (from 0 to 255) and the angle from where come te light (0.0 to 360.0).

Here is the **ProcessLighting()** function definition:

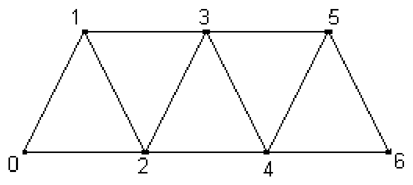
```
// ----------------------------------------------
// ProcessLighting() - process all lighting calculus.
// ----------------------------------------------

void CMD2Model::ProcessLighting( void )
{
    float lightvar = (float)((g_shadelight + g_ambientlight)/256.0);

    lcolor[0] = g_lightcolor[0] * lightvar;
    lcolor[1] = g_lightcolor[1] * lightvar;
    lcolor[2] = g_lightcolor[2] * lightvar;

    shadedots = anorms_dots[ ((int)(g_angle * (SHADEDOT_QUANT / 360.0))) & (SHADEDOT_QUANT - 1) ];
}
```

First we create a local variable which we'll use to initialize the final light color (**lcolor**) and then we adjust the **shadedots** pointer. The formula is quite obscure, don't worry about it, it works fine it's all we want ;-) It comes from the Quake II's source code.

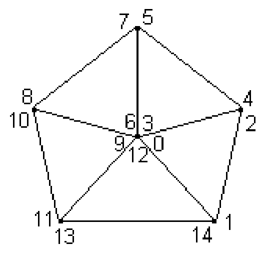
Now drawing each triangle! Remember at the beginning of this document when I gave a piece of code rendering each triangle of the current frame. The bad thing is that we were drawing using **GL_TRIANGLES**, and for that we need to specify three vertices per triangle. Moreover, it is slower than rendering using **GL_TRIANGLE_STRIP** or **GL_TRIANGLE_FAN** which need less vertices to draw more triangles. Figure 5 shows this idea:



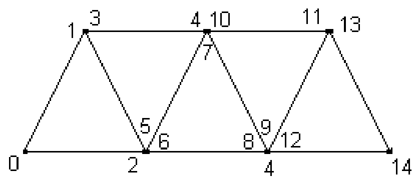
GL_TRIANGLE_FAN



GL_TRIANGLE_STRIP



Same figure with GL_TRIANGLES

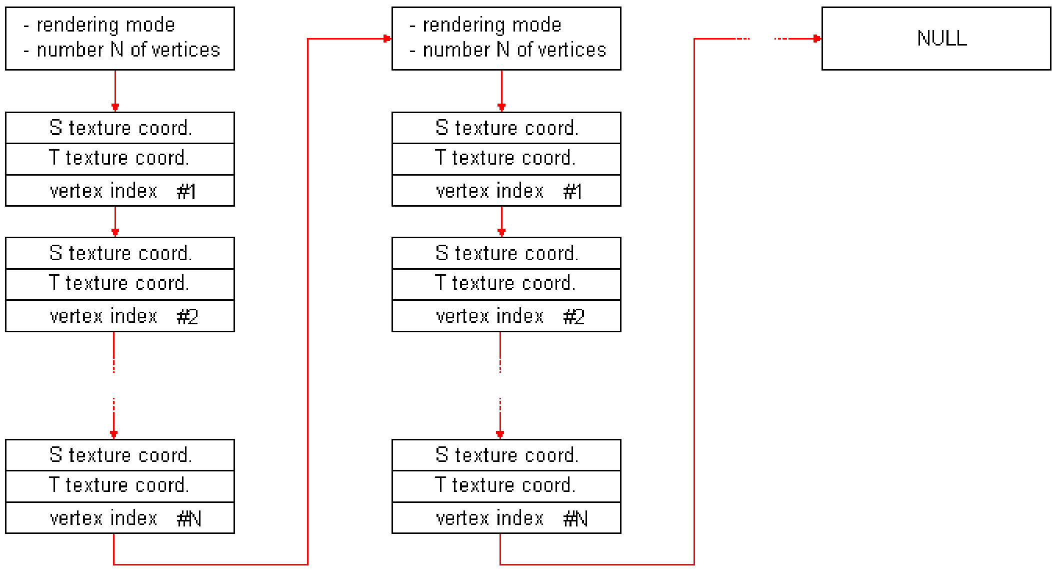


Same figure with GL_TRIANGLES

The best would be that we could draw the entire model using **`GL_TRIANGLE_STRIP`** and **`GL_TRIANGLE_FAN`**. This is what are made gl commands for! The OpenGL command list is a particular array of integers. We'll initialize a pointer pointing at the beginning of the list and read each command until the pointer return 0. 0 is the last value of the OpenGL command list. Now how does it work?

- We read the first value. This value indicates two things: the type of triangle to draw (**`GL_TRIANGLE_STRIP`** if the number is positive and **`GL_TRIANGLE_FAN`** if negative) and the number n of vertices to draw for this rendering mode.
- The n * 3 next values store information about vertices to draw.
- The two first are (s, t) texture coordinates and the third is the vertex index to draw.
- Once all vertices of this group are processed, we read a new value to get a new group... If the read value is 0, it is done!

It is not very simple the first time but with some practice you'll see that in reality it is quite simple ;-) Look at figure 6 for a representation of OpenGL command list (each rectangle represent one command which is one integer value):



Ok I've finished with theory. Now the code:

```
// ----------------------------------------------
// RenderFrame() - draw the current model frame
// using OpenGL commands.
// ----------------------------------------------

void CMD2Model::RenderFrame( void )
{
    static vec3_t    vertlist[ MAX_MD2_VERTS ];  // interpolated vertices
    int              *ptricmds = m_glcmds;       // pointer on gl commands


    // reverse the orientation of front-facing
    // polygons because gl command list's triangles
    // have clockwise winding
    glPushAttrib( GL_POLYGON_BIT );
    glFrontFace( GL_CW );

    // enable backface culling
    glEnable( GL_CULL_FACE );
    glCullFace( GL_BACK );


    // process lighting
    ProcessLighting();

    // interpolate
    Interpolate( vertlist );

    // bind model's texture
    glBindTexture( GL_TEXTURE_2D, m_texid );


    // draw each triangle!
    while( int i = *(ptricmds++) )
    {
        if( i < 0 )
        {
            glBegin( GL_TRIANGLE_FAN );
            i = -i;
        }
        else
        {
            glBegin( GL_TRIANGLE_STRIP );
        }


        for( /* nothing */; i > 0; i--, ptricmds += 3 )
        {
            // ptricmds[0] : texture coordinate s
            // ptricmds[1] : texture coordinate t
            // ptricmds[2] : vertex index to render

            float l = shadedots[ m_lightnormals[ ptricmds[2] ] ];

            // set the lighting color
            glColor3f( l * lcolor[0], l * lcolor[1], l * lcolor[2] );

            // parse texture coordinates
            glTexCoord2f( ((float *)ptricmds)[0], ((float *)ptricmds)[1] );

            // parse triangle's normal (for the lighting)
            // >>> only needed if using OpenGL lighting
            glNormal3fv( anorms[ m_lightnormals[ ptricmds[2] ] ] );

            // draw the vertex
            glVertex3fv( vertlist[ ptricmds[2] ] );
        }

        glEnd();
    }

    glDisable( GL_CULL_FACE );
    glPopAttrib();
}
```

We start creating two local variables. **`vertlist[]`** is an array of 3D floating point coordinates which will contains the interpolated and scaled vertices of the frame to render. The array is static so it's declared only once. It's better for performance improvement than creating a new array at each call of this function. The size of the array is constant and is the maximum number of vertices that a model can hold.

The second variable is **`ptricmds`**. It is the pointer which will read OpenGL commands.

Then we save polygon attributes, reverse orientation of front-facing polygons because of the GL commands and enable backface culling. We process all calculus needed for the lighting, interpolate vertices and scale them, and bind the model texture.

All the rendering is done in the while statement. First we get the triangle type and the number of vertices to draw. In the for statement we parse each vertex. Because each vertex has 3 values stored in the gl command list, we increment the pointer by 3 when all vertices of the group are processed.

For each vertex, we set the lighting color using the pointer on the dot product result table for the light angle and the final lighting color calculated by the **`ProcessLighting()`** function. Textures coordinates are casted from int to float. We obtain the normal vector from the anorms table and render the vertex from the array initialized just before.

Notice that if you don't use OpenGL lighting, the call to **`glNormal3fv()`** don't do anything and if you use it, the call to **`glColor3f()`** doesn't affect anything.

Animating

3D models look nicer when they are animated! So let's animate all that.

Remember the static **animlist** array. It has been designed to store all minimal animation data, that is to say the index of the first and last frame, and the fps count for running the animation. All this is regrouped into a structure **anim_t** we've already seen before. Here is the initialisation:

```
// ----------------------------------------------
// initialize the 21 MD2 model animations.
// ----------------------------------------------

anim_t CMD2Model::animlist[ 21 ] =
{
    // first, last, fps

    {   0,  39,  9 },   // STAND
    {  40,  45, 10 },   // RUN
    {  46,  53, 10 },   // ATTACK
    {  54,  57,  7 },   // PAIN_A
    {  58,  61,  7 },   // PAIN_B
    {  62,  65,  7 },   // PAIN_C
    {  66,  71,  7 },   // JUMP
    {  72,  83,  7 },   // FLIP
    {  84,  94,  7 },   // SALUTE
    {  95, 111, 10 },   // FALLBACK
    { 112, 122,  7 },   // WAVE
    { 123, 134,  6 },   // POINT
    { 135, 153, 10 },   // CROUCH_STAND
    { 154, 159,  7 },   // CROUCH_WALK
    { 160, 168, 10 },   // CROUCH_ATTACK
    { 196, 172,  7 },   // CROUCH_PAIN
    { 173, 177,  5 },   // CROUCH_DEATH
    { 178, 183,  7 },   // DEATH_FALLBACK
    { 184, 189,  7 },   // DEATH_FALLFORWARD
    { 190, 197,  7 },   // DEATH_FALLBACKSLOW
    { 198, 198,  5 },   // BOOM
};
```

We'll use an index to access to animation data, but it is better to define a macro for each index for readability of the source code:

```
// animation list
typedef enum {
    STAND,
    RUN,
    ATTACK,
    PAIN_A,
    PAIN_B,
    PAIN_C,
    JUMP,
    FLIP,
    SALUTE,
    FALLBACK,
    WAVE,
    POINT,
    CROUCH_STAND,
    CROUCH_WALK,
    CROUCH_ATTACK,
    CROUCH_PAIN,
    CROUCH_DEATH,
    DEATH_FALLBACK,
    DEATH_FALLFORWARD,
    DEATH_FALLBACKSLOW,
    BOOM,

    MAX_ANIMATIONS

} animType_t;
```

The current animation data is stored in the **m_anim** variable but is a little different from the **anim_t** structure. So to set an animation we must retrieve animation data and initialize current animation data with it. It's the **SetAnim()** function's job:

```
// ----------------------------------------------
// SetAnim() - initialize m_anim from the specified
// animation.
// ----------------------------------------------

void CMD2Model::SetAnim( int type )
{
    if( (type < 0) || (type > MAX_ANIMATIONS) )
        type = 0;

    m_anim.startframe   = animlist[ type ].first_frame;
    m_anim.endframe     = animlist[ type ].last_frame;
    m_anim.next_frame   = animlist[ type ].first_frame + 1;
    m_anim.fps          = animlist[ type ].fps;
    m_anim.type         = type;
}
```

First we check the type is valide and then we initialize **`m_anim`**'s members variables. You can pass to type any macro defined just before.

We'll now see a new function: **`Animate()`**. This function will be called in the **`DrawModel()`** function, so we must rewrite it:

```
// ----------------------------------------------
// DrawModel() - draw the model.
// ----------------------------------------------

void CMD2Model::DrawModel( float time )
{
    // animate. calculate current frame and next frame
    if( time > 0.0 )
        Animate( time );

    glPushMatrix();
        // rotate the model
        glRotatef( -90.0, 1.0, 0.0, 0.0 );
        glRotatef( -90.0, 0.0, 0.0, 1.0 );

        // render it on the screen
        RenderFrame();
    glPopMatrix();
}
```

Here we animate only if time is greater than 0.0. Otherwise there is no animation, the model is static. Look at the **`Animate()`** function source code:

```
// ----------------------------------------------
// Animate() - calculate the current frame, next
// frame and interpolation percent.
// ----------------------------------------------

void CMD2Model::Animate( float time )
{
    m_anim.curr_time = time;

    // calculate current and next frames
    if( m_anim.curr_time - m_anim.old_time > (1.0 / m_anim.fps) )
    {
        m_anim.curr_frame = m_anim.next_frame;
        m_anim.next_frame++;

        if( m_anim.next_frame > m_anim.endframe )
            m_anim.next_frame = m_anim.startframe;

        m_anim.old_time = m_anim.curr_time;
    }

    // prevent having a current/next frame greater
    // than the total number of frames...
    if( m_anim.curr_frame > (num_frames - 1) )
        m_anim.curr_frame = 0;

    if( m_anim.next_frame > (num_frames - 1) )
        m_anim.next_frame = 0;

    m_anim.interpol = m_anim.fps * (m_anim.curr_time - m_anim.old_time);
}
```

In a first time, the function calculate the first and next frames using the fps count specified to the current animation. In a second time, it check these values and verify that they are correct (they must not be greater than the total number of frames that holds the model. Finaly, the interpolation percent is calculated from the animation fps count and the time.

We must now review our **`Interpolate()`** function, this time to really interpolate vertices. Otherwise, we would have a very poor animation because of the number of frames the model can holds. With the interpolation, we can create an “infinity” of frames (we create just that we need when rendering). The formula is quite simple:

$$X_{interpolated} = X_{inital} + InterpolationPercent * (X_{final} - X_{inital})$$

So let's interpolate all vertices of the current and the next frames. The new **`Interpolate()`** function looks like this:

```
// ----------------------------------------------
// Interpolate() - interpolate and scale vertices
// from the current and the next frame.
// ----------------------------------------------
```

```
void CMD2Model::Interpolate( vec3_t *vertlist )
{
    vec3_t  *curr_v;    // pointeur to current frame vertices
    vec3_t  *next_v;    // pointeur to next frame vertices

    // create current frame and next frame's vertex list
    // from the whole vertex list
    curr_v = &m_vertices[ num_xyz * m_anim.curr_frame ];
    next_v = &m_vertices[ num_xyz * m_anim.next_frame ];

    // interpolate and scale vertices to avoid ugly animation
    for( int i = 0; i < num_xyz ; i++ )
    {
        vertlist[i][0] = (curr_v[i][0] + m_anim.interpol * (next_v[i][0] - curr_v[i][0])) * m_scale;
        vertlist[i][1] = (curr_v[i][1] + m_anim.interpol * (next_v[i][1] - curr_v[i][1])) * m_scale;
        vertlist[i][2] = (curr_v[i][2] + m_anim.interpol * (next_v[i][2] - curr_v[i][2])) * m_scale;
    }
}
```

By the way, we scale interpolated vertices... And that's all! You just need to call once **`SetAnim()`** and **`ScaleModel()`** functions with the parameter of your choice, and **`DrawModel()`** with the current time in seconds in parameter during the rendering loop. That's not so bad!

Just before ending, I would show you how to render a simple frame in case you'll need (for example: drawing a statue):

```
// ----------------------------------------------
// RenderFrame() - draw one frame of the model
// using gl commands.
// ----------------------------------------------

void CMD2Model::DrawFrame( int frame )
{
    // set new animation parameters...
    m_anim.startframe   = frame;
    m_anim.endframe     = frame;
    m_anim.next_frame   = frame;
    m_anim.fps          = 1;
    m_anim.type         = -1;

    // draw the model
    DrawModel( 1.0 );
}
```

This function adjust animation variables before calling **`DrawModel()`** which will render the specified frame of the model.

Conclusion

Here we are, it is finally finished! :-)

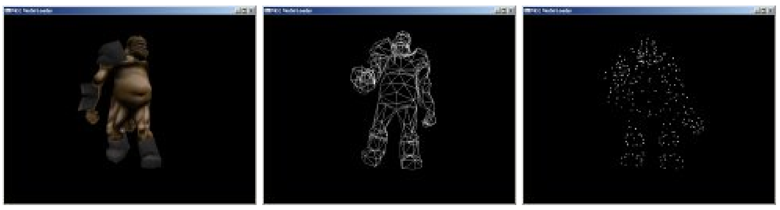
This article is far from being perfect and can be widely improved like including multiple skin support or separating model file data (vertex list, normal list, ...) from model parameters (current frame, current animation, ...) to avoid storing same model data multiple times when more than one entity is representated by the same model... It is difficult to create a perfect **`CMD2Model`** class which would work in any program with a simple cut and paste...

I hope this article helped you to learn about the MD2 model file format and more generally about 3D Model files! Also I hope it was not too confusing. Please don't spam my mailbox about my English, it is not my native language. Otherwise, you can contact me at tfc.duke (AT) gmail (DOT) com for anything you want to say about this article (suggestions, mistakes, ...).

You can download source code (Visual C++ 6.0 version) and binaries with a model and its weapon. Source code of this article is free and is provided without warranty expressed or implied. Use at your own risk! Download: q2md2_us.zip.

NOTE: the code of my MD2 Loader has been completly rewritten since I published this article (better C++ code). You can download the latest version: md2loader.zip.

Thanks to Squintik (squintik*NOSPAM*wanadoo.fr) from Game-Lab who helped me for the english version of this document.



Ressources

- MD2 file format (Quake 2's models), Quick and short, *David Henry* (me).
- OpenGL Game Programing, Ch. 18, *K. Hawkins, D. Astle*.
- Focus on 3D Models, Ch. 3, *Evan Pipho*.
- Game Tutorials, MD2 Loader, *Ben “DigiBen” Humphrey*.
- Game Tutorials, MD2 Animation, *Ben “DigiBen” Humphrey*.
- .md2 File Format Specification, *Daniel E. Schoenblum*.
- Quake II source code (GPL), *ID Software*.
- MD2 Viewer source code, *Mete Ciragan*.
- qview source code, *Mustata “LoneRunner” Bogdan*.
- Qbism Game Engine source code, *Jeff Ford*.
- jawMD2 source code, *Jawed Karim*.