

Le format MD2

par [David Henry](#)

Date de publication : 04/05/2004

Dernière mise à jour : 04/05/2004

Le format MD2 est un format de fichier contenant les données des modèles 3D de Quake II. Cet article a pour objectif de vous expliquer comment lire ces données puis les dessiner à l'écran avec OpenGL.

- 1 - Introduction
- 2 - Le format MD2
- 3 - La structure du fichier
- 4 - Implémentation d'un objet « modèle MD2 »
- 5 - Lecture du fichier MD2
- 6 - Rendu d'une frame du modèle
- 7 - Animation du modèle
- 8 - Rendu du modèle MD2 avec les commandes OpenGL
- 9 - Conclusion
 - 9.1 - Code Source
 - 9.2 - Ressources
 - 9.3 - Licence

1 - Introduction

Le format MD2 est un format de fichier contenant les données des modèles 3D de Quake II. Cet article a pour objectif de vous expliquer comment lire ces données puis les dessiner à l'écran avec OpenGL.

Pourquoi utiliser ce format de modèles, datant de 1997 ? Parce que c'est un format assez simple et très intéressant pour apprendre à charger des modèles en 3D depuis un fichier. Donc si vous êtes novice dans ce domaine, vous êtes tombé sur le bon article. Certaines choses que l'on verra sont également valable pour d'autres formats de modèles, et même d'autres types de fichiers. Bien qu'écrit en C++, il est facilement portable en C. Cet article s'adresse à des programmeurs initiés connaissant déjà le C++ et l'API Application Programming Interface OpenGL.

Pour commencer, nous allons faire un tour d'horizon de ce format de modèles 3D.

2 - Le format MD2

Explorons un peu ce format et voyons ce que l'on risque de rencontrer dedans. Les modèles MD2 sont les modèles utilisés dans le jeu Quake II et sont donc composés :

- de données géométriques.
- d'animations par frame.
- de « commandes OpenGL ».

Les données géométriques sont les triangles et les sommets du modèle, ainsi que leurs coordonnées de texture. Ce sont des données que vous avez déjà dû manipuler de nombreuses fois.

Les animations du modèle sont ce que l'on appelle des animations par frame. Ce ne sont pas des animations squelettiques. C'est pour ça aussi que ce format est simple ; les animations squelettiques demandant un peu plus de gymnastique mathématique. Les animations sont donc décomposées en frames. Une animation est faite de plusieurs frames. Une frame est une séquence d' une animation. C'est comme un film composé d'une multitude d'images défilant rapidement, ce qui donne l'effet d'être animé. Les frames sont ces « images ». Chaque frame contient toutes les données géométrique du modèle (triangles, etc.) dans une certaine position. Donc en réalité, un modèle MD2 est composé d'une multitude de modèles qui, affichés successivement et rapidement, lui donnent un effet d'animation.

Les commandes OpenGL sont des données structurées de sorte que l'on puisse dessiner le modèle uniquement à l'aide des primitives *GL_TRIANGLE_STRIP* et *GL_TRIANGLE_FAN*. Cet algorithme de rendu a été implémenté dans le but de gagner en temps d'exécution, ce qui est très important dans un jeu en 3D car permet alors l'affichage d'un plus grand nombre d'objets. Ces données ont donc été triées puis rangées de sorte à ce que le rendu du modèle se fasse rapidement et simplement.

Dans cet article, nous verrons d'abord une version plus classique pour le rendu du modèle en utilisant des primitive de type *GL_TRIANGLES*. La méthode de rendu par ces commandes OpenGL, moins intuitive, est décrite en fin d'article.

Les textures des modèles sont stockées dans d'autres fichiers. Généralement, il s'agit d'images TGA ou PCX, mais étant indépendantes du modèle, rien ne vous empêche de choisir un autre format. Cet article ne traitera donc pas de la lecture des textures des modèles MD2.

Voyons maintenant comment ces données sont structurées à l'intérieur de ces fichiers.

3 - La structure du fichier

Le fichier se décompose en deux parties : l'en-tête (*header*) de taille constante et les données (*data*) de taille variable. L'en-tête contient les informations nécessaires à la lecture des données du fichier comme le nombre de triangles, le nombre de frames, la position des données à l'intérieur de fichier, etc. Ce découpage se retrouve d'ailleurs dans la plupart des formats de fichiers.

L'en-tête de fichier peut-être stockée dans une structure, ce qui évitera d'éparpiller des infos un peu partout et permettra une lecture propre du fichier. Cette structure se présente ainsi :

```
// header md2
typedef struct
{
    int ident;           // numéro magique : "IDP2"
    int version;        // version du format : 8

    int skinwidth;      // largeur texture
    int skinheight;     // hauteur texture

    int framesize;      // taille d'une frame en octets

    int num_skins;      // nombre de skins
    int num_vertices;   // nombre de vertices par frame
    int num_st;         // nombre de coordonnées de texture
    int num_tris;       // nombre de triangles
    int num_glcmds;     // nombre de commandes opengl
    int num_frames;     // nombre de frames

    int offset_skins;   // offset données skins
    int offset_st;      // offset données coordonnées de texture
    int offset_tris;    // offset données triangles
    int offset_frames;  // offset données frames
    int offset_glcmds;  // offset données commandes OpenGL
    int offset_end;     // offset fin de fichier
} md2_header_t;
```

Étudions cette structure plus en détails, ce qui nous permettra de voir ce que l'on va trouver dans la seconde partie du fichier, à savoir, les données.

L'en-tête commence avec deux variables permettant l'identification du type de fichier. On rencontre souvent ce genre de données pour les différents formats de fichier, généralement en début de fichier mais pas forcément de type *int*. Dans le cas du MD2, les quatre premiers octets du fichier, stockés dans *ident* doivent former le mot « IDP2 ». On peut également le regrouper sous forme d'entier et faire une comparaison plus rapide. Cette variable est parfois appelée « numéro magique ». La seconde variable est le numéro de version du format. Dans notre cas, *version* devra toujours être égal à 8.

Les variables *skinwidth* et *skinheight* contiennent les dimensions de la texture du modèle. Bien que l'on charge ces textures indépendamment du modèle, on aura besoin de ces deux données pour calculer les coordonnées de texture exactes (nous verrons ça plus loin). *Skin* en anglais signifie « peau », ici donc la texture.

framesize est la taille en octets de chaque frame du modèle. Dans l'implémentation que je vais vous proposer on ne s'en servira pas.

num_skins est le nombre de textures du modèle. Cette variable est spécifique au code de Quake II et nous en tiendrons pas compte. *num_vertices* est le nombre de sommets (*vertices*) utilisés par une frame. *num_st* lui est le nombre de couples de coordonnées de texture. Ce nombre n'est pas forcément égal au nombre de sommets. Certains sommets peuvent en effet partager le même couple, ou l'inverse, ils peuvent en avoir plusieurs. *num_tris* est le nombre de triangles du modèle. Ces triangles sont les mêmes peu importe la frame à dessiner. Il en est de même pour les coordonnées de texture. Seules les positions des sommets varient d'une frame à l'autre. Tout comme pour les coordonnées de texture, les triangles peuvent se partager plusieurs sommets. *num_glcmds* est le nombre de commandes OpenGL. Enfin, *num_frames* le nombre de frames du modèle.

Le dernier paquet de variables représente les positions des données dans le fichier. Ce sont des tailles en octets qui spécifient le décalage (*offset*) depuis le début du fichier pour atteindre telles ou telles données. Allons-y : *offset_skins* est la position des données de texture du modèle. Elles ne se résument qu'au nom de fichier à charger. Bien que nous n'en aurons pas besoin, nous les stockerons quand même pour que vous compreniez ce que sont ces données. *offset_st* est la position des données des coordonnées de texture, *offset_tris* les données des triangles, *offset_frames* le début des données relatives aux frames et *offset_glcmds* les commandes OpenGL. *offset_end* est la fin du fichier (on peut ainsi connaître rapidement la taille du fichier d'ailleurs).

Passons aux données. Nous allons devoir avant tout créer des structures pour les contenir de façon ordonnées. Pour commencer, définissons le type vecteur qui sera un triplet de *float* contenant une position x,y,z :

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

Les données brutes du modèle relatives aux sommets contiennent pour chacun d'entre eux ses coordonnées géométriques sur trois octets (un octet par composant) et un index de normale. Généralement lorsqu'on dessine un objet avec OpenGL, on utilise des coordonnées de type *GLfloat* sur 4 octets et non pas un qui nous bloque sur un nombre entier compris entre 0 et 255. Il s'agit en fait des coordonnées « compressées » (comprendre : pour réduire la taille du fichier en octets). Pour les décompresser, on utilisera les données qui nous seront fournies avec le type frame. L'index accompagnant ce triplet permet d'accéder à un vecteur normal à ce sommet (préalablement calculé à partir des triangles qu'il compose avant de sauvegarder le fichier) dans une table de normales précalculées. Ici aussi il s'agissait de gagner en temps de calcul en ayant les normales déjà calculées d'avance. Cette table de normale est universelle à tous les modèles MD2 et n'est pas contenue dans les fichiers des modèles. Les normales serviront à OpenGL pour l'éclairage du modèle. Le type sommet est ressemblé donc à ça :

```
// données vertex
typedef struct
{
    unsigned char v[3];           // position dans l'espace (relative au modèle)
    unsigned char normalIndex;   // index normale du vertex
} md2_vertex_t;
```

Voyons les données des triangles. Un triangle est composé de trois sommets et de trois couples de coordonnées de texture. Les sommets et les coordonnées de textures étant séparés des triangles dans l'arrangement des données du fichier, le type triangle contient donc un triplet d'indices pointant vers les sommets et un triplet d'indices pour les coordonnées de texture. Voici sa définition :

```
// données triangle
typedef struct
{
    unsigned short vertex[3]; // indices vertices du triangle
    unsigned short st[3];    // indices coordonnées de texture
} md2_triangle_t;
```

Pour ce qui est des coordonnées de texture, il s'agit d'un simple couple de deux valeurs. Dans les fichiers MD2, chacune de ces coordonnées est aussi compressée et est alors stockée sur deux octets. Pour les décompresser, il suffira de diviser ces valeurs par *skinwidth* et *skinheight* (suivant la composante).

```
// coordonnées de texture
typedef struct
{
    short s;
    short t;
} md2_texCoord_t;
```

On doit maintenant structurer les données relatives aux frames. Chaque frame possède deux triplets de *float* qui vont nous permettre de décompresser les données des sommets vus plus haut. L'un étant un facteur de redimensionnement pour remettre le modèle à l'échelle, l'autre un vecteur de translation. Pour obtenir un sommet décompressé, il suffit juste de multiplier ses coordonnées par le premier triplet puis de lui ajouter le second. Chaque frame est également désignée par un nom stocké dans une chaîne de 16 octets. Généralement ce nom est sous la forme <animation><numéro de frame>. Enfin, ce sont ces frames qui vont chacune stocker leurs sommets, donc il nous faudra un pointeur vers un tableau de sommets alloué dynamiquement.

```
// données frame
typedef struct
{
    vec3_t      scale; // redimensionnement
    vec3_t      translate; // vecteur translation
    char        name[16]; // nom de la frame
    md2_vertex_t *verts; // liste de vertices
} md2_frame_t;
```

Passons rapidement sur le type skin du modèle. Les seules informations sont comme on l'a vu plus haut, des noms de fichier de texture. Ils sont stockés dans une chaîne de caractères de 68 octets chacun. Ce type se résume donc à ceci :

```
// texture
typedef struct
{
    char        name[68]; // nom du fichier texture
} md2_skin_t;
```

Pour finir, il reste les commandes OpenGL à stocker. Leur cas est un peu particulier, on le verra à la fin de l'article. Pour le moment, on se contentera de les stocker dans un tableau d'entiers, soit *int**. On peut résumer la structure des données d'un fichier MD2 par ce schéma simpliste et incomplet :

4 - Implémentation d'un objet « modèle MD2 »

À l'aide d'une classe, nous allons créer un objet « modèle MD2 » qui encapsulera toutes les données d'un modèle MD2. Cette classe possédera des méthodes pour lire un fichier MD2 et le dessiner à l'écran avec OpenGL. Voici sa déclaration :

```
// =====
// CMD2Model - classe objet modèle <.md2>
// =====

class CMD2Model
{
public:
    // constructeur/destructeur
    CMD2Model( void );
    ~CMD2Model( void ) { FreeModel(); }

public:
    // fonctions publiques
    bool    LoadModel( std::string szFilename );
    void    LoadTexture( std::string szFilename );

    void    FreeModel( void );

    void    RenderFrame( int iFrame );

    void    SetScale( GLfloat fScale ) { m_fScale = fScale; }
    GLfloat GetScaleValue( void ) { return m_fScale; }

public:
    // variables membres
    static vec3_t    m_kAnorms[ NUMVERTEXNORMALS ];

    md2_header_t    m_kHeader;        // header md2
    md2_skin_t      *m_pSkins;        // données skins
    md2_texCoord_t  *m_pTexCoords;    // coordonnées de texture
    md2_triangle_t  *m_pTriangles;    // données triangles
    md2_frame_t     *m_pFrames;       // données frames
    int             *m_pGLcmds;       // liste de commandes OpenGL

    GLfloat        m_fScale;          // redimensionnement du modèle

    GLuint         m_uiTexID;         // ID texture du modèle
};
```

Commençons par les fonctions, toutes publiques. *LoadModel()* se charge de lire un le fichier MD2 donné en paramètre et d'initialiser les variables membres de la classe. *LoadTexture()* crée une texture à partir d'une image stockée dans un fichier et stocke l'objet de texture dans *m_uiTexID*. *FreeModel()* est appelée par le destructeur. Cette fonction se charge de nettoyer la mémoire lors de la destruction de l'objet MD2. *RenderFrame()* a pour tâche de dessiner une frame donnée du modèle à l'écran. Enfin, *SetScale()* et *GetScaleValue()* sont deux petites fonctions utilitaires servant respectivement à redimensionner le modèle et à récupérer le facteur de redimensionnement.

Faisons le tour des variables membres. Elles sont toutes privées, seules les routines de cette classe pourront donc y accéder. Tout d'abord on trouve *m_kAnorms*. C'est la fameuse table de vecteurs normaux précalculés. Cette variable est déclarée avec le mot clef *static* car cette table est valable pour tous les modèles MD2, il est donc inutile de l'avoir en double, en triple, etc. Il vous faudra définir sa longueur, *NUMVERTEXNORMALS*, avec une instruction *#define*. Sa valeur est 162.

Viennent ensuite les données du modèle. En premier lieu, l'en-tête du fichier soigneusement conservée dans *m_kHeader*. Puis les données mêmes du modèle : *m_pSkins* pour les noms de textures, *m_pTexCoords* un tableau de couples de coordonnées de texture, *m_pTriangles* un tableau de triangles, *m_pFrames* un tableau de frames et enfin la liste des commandes OpenGL dans le tableau *m_pGLcmds*.

Les deux dernières variables, *m_fScale* et *m_uiTexID*, sont le facteur de redimensionnement du modèle (on multiplie les composantes x,y,z de tous les sommets par ce facteur avant de dessiner le modèle) et l'objet de texture OpenGL du modèle.

Nous reviendrons sur cette classe plus tard pour y rajouter quelques méthodes pour le support des animations et du rendu via les commandes OpenGL. À présent, passons aux définitions des fonctions de cette classe.

5 - Lecture du fichier MD2

Nous allons définir dans cette partie, les routines associées à la lecture d'un fichier MD2 et la destruction d'un objet MD2, soit les fonctions *LoadModel()*, *LoadTexture()* et *FreeModel()*. Nous ne verrons pas ici le constructeur qui ne fait rien d'autre que d'initialiser toutes les variables membres à 0 (ou *NULL* pour les tableaux) à l'exception de *m_fScale* initialisé à 1,0 (l'échelle du modèle d'origine).

Commençons par la lecture du fichier et l'initialisation des variables membres de notre classe objet MD2. Cette opération est très simple et se fait en trois temps : d'abord on lit l'en-tête du fichier et on vérifie qu'il s'agit bien d'un fichier MD2 (numéro magique égal à « IDP2 » et version 8). Ensuite on alloue de la mémoire dynamiquement pour les tableaux qui vont contenir les données du modèle, grâce aux informations contenues dans l'en-tête du fichier. Enfin, on remplit ces tableaux en lisant les données du modèle.

Pour ce qui est de la vérification du format du fichier, on va comparer la variable *ident* à l'équivalent de « IDP2 » en entier sur 4 octets. Pour cela, on va définir une macro ayant pour valeur la somme des valeurs ASCII des lettres du mot « IDP2 » avec le décalage binaire qui convient suivant leur position dans la chaîne. Bref, concrètement, voilà à quoi l'on va comparer *ident* :

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

// numéro de version
#define MD2_VERSION    8
```

Nous pouvons maintenant lire le fichier. Voici la définition de la fonction *LoadModel()* qui renvoie *true* si tout se passe bien, sinon *false*. Attention, n'oubliez pas d'inclure des fichiers d'en-tête *string* et *fstream* :

```
// -----
// LoadModel() - charge un modèle à partir d'un
// fichier <.md2>.
// -----

bool CMD2Model::LoadModel( std::string szFilename )
{
    std::fstream    file;    // fichier

    // tentative d'ouverture du fichier
    file.open( szFilename.c_str(), std::ios::in | std::ios::binary );

    if( file.fail() )
        return false;

    // lecture du header
    file.read( (char *)&m_kHeader, sizeof( md2_header_t ) );

    // vérification de l'authenticité du modèle
    if( (m_kHeader.version != MD2_VERSION) || m_kHeader.ident != MD2_IDENT )
        return false;

    // allocation de mémoire pour les données du modèle
    m_pSkins = new md2_skin_t[ m_kHeader.num_skins ];
    m_pTexCoords = new md2_texCoord_t[ m_kHeader.num_st ];
    m_pTriangles = new md2_triangle_t[ m_kHeader.num_tris ];
    m_pFrames = new md2_frame_t[ m_kHeader.num_frames ];
    m_pGLcmds = new int[ m_kHeader.num_glcmds ];
}
```

```

// lecture des noms de skins
file.seekg( m_kHeader.offset_skins, std::ios::beg );
file.read( (char *)m_pSkins, sizeof( char ) * 68 * m_kHeader.num_skins );

// lecture des coordonnées de texture
file.seekg( m_kHeader.offset_st, std::ios::beg );
file.read( (char *)m_pTexCoords, sizeof( md2_texCoord_t ) * m_kHeader.num_st );

// lecture des triangles
file.seekg( m_kHeader.offset_tris, std::ios::beg );
file.read( (char *)m_pTriangles, sizeof( md2_triangle_t ) * m_kHeader.num_tris );

// lecture des frames
file.seekg( m_kHeader.offset_frames, std::ios::beg );

for( int i = 0; i < m_kHeader.num_frames; i++ )
{
    // allocation de mémoire pour les vertices
    m_pFrames[i].verts = new md2_vertex_t[ m_kHeader.num_vertices ];

    // lecture des données de la frame
    file.read( (char *)&m_pFrames[i].scale, sizeof( vec3_t ) );
    file.read( (char *)&m_pFrames[i].translate, sizeof( vec3_t ) );
    file.read( (char *)&m_pFrames[i].name, sizeof( char ) * 16 );
    file.read( (char *)m_pFrames[i].verts, sizeof( md2_vertex_t ) *
m_kHeader.num_vertices );
}

// lecture des commandes opengl
file.seekg( m_kHeader.offset_glcmds, std::ios::beg );
file.read( (char *)m_pGLcmds, sizeof( int ) * m_kHeader.num_glcmds );

// fermeture du fichier
file.close();

// succès
return true;
}

```

Passons rapidement sur *LoadTexture()*. Comme je l'ai dit plus haut, nous n'allons pas voir le chargement d'un fichier TGA ou PCX ou tout autre type d'image. Cette fonction a pour but de charger en mémoire une texture et d'initialiser la variable *m_uiTexID*, objet de texture OpenGL. On imaginera que la fonction *LoadTextureFromFile()* est définie dans votre code, et qu'elle s'occupe de charger une texture et de retourner son objet. Il y a mille façons de le faire :

```

// -----
// LoadTextures() - charge la texture du modèle
// à partir du chemin d'accès spécifié.
// -----

void CMD2Model::LoadTexture( std::string szFilename )
{
    // chargement de la texture
    m_uiTexID = LoadTextureFromFile( szFilename );
}

```

Enfin, *FreeModel()*, fonction très importante qui se chargera de libérer la mémoire allouée pour les données à la destruction de l'objet MD2. Cette fonction est automatiquement appelée par le destructeur, donc vous n'avez pas besoin de l'appeler manuellement avant la destruction de votre objet :

```

// -----
// FreeModel() - libère la mémoire allouée pour
// le modèle.
// -----

```

```
void CMD2Model::FreeModel( void )
{
    if( m_pSkins )
        delete [] m_pSkins;

    if( m_pTexCoords )
        delete [] m_pTexCoords;

    if( m_pTriangles )
        delete [] m_pTriangles;

    if( m_pFrames )
    {
        for( int i = 0; i < m_kHeader.num_frames; i++ )
        {
            if( m_pFrames[i].verts )
                delete [] m_pFrames[i].verts;
        }

        delete [] m_pFrames;
    }
}
```

6 - Rendu d'une frame du modèle

Sans doute la partie que vous attendiez le plus, non ? ;-). Nous allons donc nous intéresser à la dernière fonction, *RenderFrame()*, qui dessine une frame donnée du modèle à l'écran. Son paramètre est le numéro de la frame à dessiner. Donc avant de commencer à dessiner des triangles, on vérifie que ce numéro est bien valide : compris entre 0 et le nombre maximum moins un de frames (car on commence l'indexation à zéro). Ensuite on sélectionne la texture de l'objet à l'aide de *glBindTexture()* et notre objet de texture *m_uiTexID*.

On peut à présent commencer le rendu à l'aide de primitives *GL_TRIANGLES*. On entame une boucle sur tous les triangles du modèle. À l'intérieur de cette boucle, on parcourt une seconde boucle pour chacun des sommets du triangle. Chaque triangle *i* possède trois indices de sommets et trois indices de coordonnées de texture. Ces indices sont représentés par *k* dans la seconde boucle. On peut ainsi accéder aux sommets du triangles dans le tableau de sommets de la frame à dessiner, et aux coordonnées de texture de ces sommets dans le tableau de coordonnées de textures (qui n'est pas spécifique à la frame, lui).

Les coordonnées de textures sont stockées sous forme d'entiers sur deux octets. Pour obtenir les véritables valeurs de ces coordonnées sur un intervalle de 0,0 à 1,0, il suffit de diviser ces entiers par *skinwidth* ou *skinheight*, suivant la composante.

Nous accédons également au vecteur normal au sommet dans la table des normales précalculées *via* un index (voir le type *md2_vertex_t*). Cette table est une liste de vecteurs (donc de type *vec3_t* ou triplets de *float*) stockée dans le fichier *anorms.h* disponible avec le code source de cet article (voir bas de page). Il est bien trop long et peu intéressant pour être copié ici, mais vous pouvez l'avoir en suivant ce lien : [anorms.h](#) (6,7 Ko). Par ailleurs, il nous faut définir la variable membre statique *m_kAnorms* :

```
// {{{ À définir avant la déclaration de classe
// nombre de vecteurs normaux précalculés
#define NUMVERTEXNORMALS    162
// }}}

// table de 162 vecteurs normaux précalculés
vec3_t CMD2Model::m_kAnorms[ NUMVERTEXNORMALS ] = {
#include    "anorms.h"
};
```

Il ne nous reste plus qu'à calculer les valeurs réelles des composantes x,y,z des sommets des triangles. Il suffit de prendre leur valeur compressée et de la multiplier par la variable *scale* de la frame, puis d'ajouter le vecteur *translate*. Enfin, on oublie pas de multiplier le tout par le scalaire *m_fScale* pour pouvoir changer l'échelle du modèle sans soucis par rapport aux normales (*glScalef()* est parfois dangereux). Voici la définition de *RenderFrame()* :

```
// -----
// RenderFrame() - dessine le modèle à la frame
// spécifiée.
// -----

void CMD2Model::RenderFrame( int iFrame )
{
    // calcul de l'index maximum d'une frame du modèle
    int iMaxFrame = m_kHeader.num_frames - 1;

    // vérification de la validité de iFrame
```

```
if( (iFrame < 0) || (iFrame > iMaxFrame) )
    return;

// activation de la texture du modèle
glBindTexture( GL_TEXTURE_2D, m_uiTexID );

// dessin du modèle
glBegin( GL_TRIANGLES );
// dessine chaque triangle
for( int i = 0; i < m_kHeader.num_tris; i++ )
{
    // dessigne chaque vertex du triangle
    for( int k = 0; k < 3; k++ )
    {
        md2_frame_t *pFrame = &m_pFrames[ iFrame ];
        md2_vertex_t *pVert = &pFrame->verts[ m_pTriangles[i].vertex[k] ];

        // [coordonnées de texture]
        GLfloat s = (GLfloat)m_pTexCoords[ m_pTriangles[i].st[k] ].s /
m_kHeader.skinwidth;
        GLfloat t = (GLfloat)m_pTexCoords[ m_pTriangles[i].st[k] ].t /
m_kHeader.skinheight;

        // application des coordonnées de texture
        glTexCoord2f( s, t );

        // [normale]
        glNormal3fv( m_kAnorms[ pVert->normalIndex ] );

        // [vertex]
        vec3_t v;

        // calcul de la position absolue du vertex et redimensionnement
        v[0] = (pFrame->scale[0] * pVert->v[0] + pFrame->translate[0]) *
m_fScale;
        v[1] = (pFrame->scale[1] * pVert->v[1] + pFrame->translate[1]) *
m_fScale;
        v[2] = (pFrame->scale[2] * pVert->v[2] + pFrame->translate[2]) *
m_fScale;

        glVertex3fv( v );
    }
}
glEnd();
}
```

7 - Animation du modèle

L'animation d'un modèle MD2 est un peu plus complexe que ce que l'on pourrait penser à la première approche du problème, mais reste relativement simple. On a vu juste au dessus comment dessiner une frame du modèle. On pourrait alors penser que pour jouer une animation du modèle, il suffirait de faire tourner en boucle les frames constituant l'animation. Par exemple, si l'animation « courir » était définie par les frames de 20 à 30 on appellerait *RenderFrame()* en incrémentant le numéro de frame à chaque boucle de rendu. Le problème si l'on suit cette méthode, c'est qu'on aura une animation très saccadée, du fait du petit nombre de frames qui composent les animations. La mémoire étant précieuse, il a fallu trancher des les frames pour ne garder que les plus importantes, surtout à l'époque de Quake II. Ce qui fait que les animations se retrouvent composées d'une quinzaine de frames généralement.

Pour éviter les saccades il va donc falloir « inventer » des frames intermédiaires en faisant une interpolation linéaire entre deux frames consécutives. Imaginez un objet qui doit se déplacer d'un point A à un point B en une seconde (on ne connaît que ces deux points de la trajectoire). Si on attend une seconde entière pour déplacer l'objet, ça donnera un effet de téléportation, ce qui n'est pas fluide du tout. On peut alors calculer la position de l'objet à 100 ms, 200 ms, etc, en supposant sa vitesse constante. À 100 ms sa position sera d'un dixième de la distance AB, à 200 ms se sera un cinquième...

Les frames du modèle sont suffisantes pour pouvoir faire une interpolation linéaire sans pour autant obtenir des effets bizarres (si un point doit parcourir un arc de cercle il faudra plusieurs frames pour éviter que ça finisse en ligne droite). La formule à appliquer aux trois composantes des coordonnées cartésiennes d'un point est la suivantes :

$X_{calculé} = X_{initial} + \text{pourcentageInterpolation} * (X_{final} - X_{initial})$

Dans l'exemple précédent (avec les points A et B), le pourcentage d'interpolation pour obtenir la position de l'objet à 200 ms est 20 % (un cinquième), soit 0,2 sur une échelle de 0 à 1.

Nous allons devoir rajouter une fonction membre à notre classe objet MD2 qui aura pour tâche de dessiner le modèle avec une interpolation entre deux frames données. La fonction ressemble beaucoup à *RenderFrame()*, la différence est que l'on applique la formule ci-dessus aux sommets des triangles et à leur vecteur normal. La fonction prend donc en paramètres les numéros des deux frames à interpoler et le pourcentage d'interpolation (n'oubliez pas de rajouter son prototype dans la définition de classe) :

```
// -----
// DrawModelItp() - dessine le modèle avec une
// interpolation de deux frames données.
// -----

void CMD2Model::DrawModelItp( int iFrameA, int iFrameB, float fInterp )
{
    // calcul de l'index maximum d'une frame du modèle
    int iMaxFrame = m_kHeader.num_frames - 1;

    // vérification de la validité des indices de frame
    if( (iFrameA < 0) || (iFrameB < 0) )
        return;

    if( (iFrameA > iMaxFrame) || (iFrameB > iMaxFrame) )
        return;
}
```

```

// activation de la texture du modèle
glBindTexture( GL_TEXTURE_2D, m_uiTexID );

// dessin du modèle
glBegin( GL_TRIANGLES );
// dessine chaque triangle
for( int i = 0; i < m_kHeader.num_tris; i++ )
{
    // dessigne chaque vertex du triangle
    for( int k = 0; k < 3; k++ )
    {
        md2_frame_t *pFrameA = &m_pFrames[ iFrameA ];
        md2_frame_t *pFrameB = &m_pFrames[ iFrameB ];

        md2_vertex_t *pVertA = &pFrameA->verts[ m_pTriangles[i].vertex[k] ];
        md2_vertex_t *pVertB = &pFrameB->verts[ m_pTriangles[i].vertex[k] ];

        // [coordonnées de texture]
        GLfloat s = (GLfloat)m_pTexCoords[ m_pTriangles[i].st[k] ].s /
m_kHeader.skinwidth;
        GLfloat t = (GLfloat)m_pTexCoords[ m_pTriangles[i].st[k] ].t /
m_kHeader.skinheight;

        // application des coordonnées de texture
        glTexCoord2f( s, t );

        // [normale]
        vec3_t normA, normB, n;

        memcpy( normA, m_kAnorms[ pVertA->normalIndex ], 3 * sizeof( float ) );
        memcpy( normB, m_kAnorms[ pVertB->normalIndex ], 3 * sizeof( float ) );

        // interpolation linéaire
        n[0] = normA[0] + fInterp * (normB[0] - normA[0]);
        n[1] = normA[1] + fInterp * (normB[1] - normA[1]);
        n[2] = normA[2] + fInterp * (normB[2] - normA[2]);

        // spécification de la normale
        glNormal3fv( n );

        // [vertex]
        vec3_t vecA, vecB, v;

        // calcul de la position absolue des vertices
        vecA[0] = pFrameA->scale[0] * pVertA->v[0] + pFrameA->translate[0];
        vecA[1] = pFrameA->scale[1] * pVertA->v[1] + pFrameA->translate[1];
        vecA[2] = pFrameA->scale[2] * pVertA->v[2] + pFrameA->translate[2];

        vecB[0] = pFrameB->scale[0] * pVertB->v[0] + pFrameB->translate[0];
        vecB[1] = pFrameB->scale[1] * pVertB->v[1] + pFrameB->translate[1];
        vecB[2] = pFrameB->scale[2] * pVertB->v[2] + pFrameB->translate[2];

        // interpolation linéaire et redimensionnement
        v[0] = (vecA[0] + fInterp * (vecB[0] - vecA[0])) * m_fScale;
        v[1] = (vecA[1] + fInterp * (vecB[1] - vecA[1])) * m_fScale;
        v[2] = (vecA[2] + fInterp * (vecB[2] - vecA[2])) * m_fScale;

        // dessin du vertex
        glVertex3fv( v );
    }
}
glEnd();
}

```

Nous allons maintenant avoir besoin de stocker de nouvelles données : les informations à propos de l'animation en cours. C'est-à-dire, la frame courante et suivante, le pourcentage d'interpolation et la distance (en pour-cent) que l'on ajoute au pourcentage d'interpolation à chaque frame (grossièrement, la vitesse de l'animation).

Ces nouvelles données ne sont pas propre au modèle lui-même, mais à l'objet représenté par ce modèle. Prenez par exemple un jeu (Quake II tiens !), vous avez plusieurs ennemis représentés par le même modèle. Plutôt que de recharger vingt fois le même modèle pour chaque ennemi et de faire un gaspillage énorme de mémoire, il est plus judicieux de ne charger le modèle (la géométrie) qu'une fois et de le partager avec ces trois entités. Chaque objet (ou entité, c'est la même chose) de type « ennemi » possède alors un pointeur vers les données de ce même modèle. Ces objets possèdent par ailleurs des informations spécifique à ce type d'entité (vie, armure, position, etc.).

Les informations relatives à l'animation du modèle (en fait, l'animation de l'objet !) font partie elles aussi de l'objet et non pas du modèle. Plusieurs objets peuvent se partager le même modèle, mais leur animation est propre à eux. Par exemple, un ennemi peut être en train de courir vers vous, il jouera alors l'animation « courir », tandis qu'un second ennemi présent à l'écran et de même nature (utilisant le même modèle donc) peut, lui, jouer l'animation « attaquer ». Si les informations d'animations appartenaient au modèle, tous les ennemis représentés par ce modèle joueraient la même animation en même temps ! On se retrouverait bien embêté.

Nous allons donc créer une nouvelle classe de type entité, pour contenir les informations d'animation. Cette classe ressemblera plutôt à une ébauche puisqu'elle ne contiendra rien d'autre qu'un pointeur vers un objet modèle MD2, un facteur d'échelle (le même que vu plus haut) et des informations relatives à l'animation du modèle :

```
// =====
// CEntity - classe objet entitée.
// =====

class CEntity
{
public:
    // constructeur/destructeur
    CEntity( void );
    ~CEntity( void ) { }

public:
    // fonctions publiques
    void SetModel( CMD2Model *pModel ) { m_pModel = pModel; }

    void DrawEntity( int iFrame, bool bAnimated, bool bUseGLCmds );
    void Animate( int iStartFrame, int iEndFrame, float fPercent );

    void SetScale( float fScale ) { m_fScale = fScale; }
    float GetScaleValue( void ) { return m_fScale; }

private:
    // variables membres
    CMD2Model *m_pModel; // pointeur modèle <.md2>

    int m_iCurrFrame; // index frame courante
    int m_iNextFrame; // index frame suivante
    float m_fInterp; // pourcentage interpolation

    float m_fPercent; // incrémentation de m_fInterp
    float m_fScale; // redimensionnement du modèle
};
```

La classe *CEntity* possède une routine pour lui « attacher » un modèle MD2 (*SetModel()*) et les mêmes routines que la classe *CMD2Model* pour le facteur de redimensionnement (*m_fScale*). Elle

possède également une fonction pour dessiner l'entité (le modèle MD2, en fait) : *DrawEntity()* et une fonction *Animate()* qui s'occupera de calculer les frames courante et suivante à partir des numéros de frames de début et de fin de l'animation souhaitée. Pour ce qui est des variables membres, ce sont donc les informations relatives à l'animation et le pointeur vers le modèle MD2.

Commençons par *Animate()*. Cette fonction se charge de calculer la frame courante et la suivante # que l'on passera en paramètre à *DrawModelltp()* de la classe *CMD2Model* # à partir des frames de début et de fin de l'animation et du pourcentage d'interpolation. Cette fonction prend en troisième paramètre une valeur en pourcent qui sera ajoutée à *m_fInterp* à chaque cycle de rendu afin de faire progresser l'animation. Afin que l'animation dure le même temps sur toutes les machines, il est préférable que ce paramètre soit en fonction du nombre d'images par seconde sorties par le programme.

La fonction vérifie d'abord la validité de *m_iCurrFrame* qui doit se situer entre les frames de début et de fin. Dans le cas où elle dépasserait cet intervalle, la variable prendrait alors la valeur de la première frame de l'animation. Ensuite, elle teste si *m_fInterp* a atteint la valeur de 1,0 (100%). Si c'est le cas, ça signifie qu'on a atteint la frame suivante. On incrémente alors *m_iCurrFrame* et *m_iNextFrame* et *m_fInterp* retourne à 0. Voici le code de *Animate()* :

```
// -----
// Animate() - calcul les frames courante et
// suivante à partir des frames de début et de
// fin d'animation et du pourcentage d'interpolation.
// -----

void CEntity::Animate( int iStartFrame, int iEndFrame, float fPercent )
{
    // m_iCurrFrame doit être compris entre iStartFrame et iEndFrame
    if( m_iCurrFrame < iStartFrame )
        m_iCurrFrame = iStartFrame;

    if( m_iCurrFrame > iEndFrame )
        m_iCurrFrame = iStartFrame;

    m_fPercent = fPercent;

    // animation : calcul des frames courante et suivante
    if( m_fInterp >= 1.0 )
    {
        m_fInterp = 0.0f;
        m_iCurrFrame++;

        if( m_iCurrFrame >= iEndFrame )
            m_iCurrFrame = iStartFrame;

        m_iNextFrame = m_iCurrFrame + 1;

        if( m_iNextFrame >= iEndFrame )
            m_iNextFrame = iStartFrame;
    }
}
```

Le rendu de l'entité se fait en appelant la fonction *DrawEntity()*. Cette fonction peut dessiner le modèle animé (avec interpolation) ou pas (rendu d'une frame simple). Il prend en premier paramètre un numéro de frame à dessiner. Si cette valeur est positive, on dessine la frame demandée. Si elle vaut -1, alors on dessine la frame *m_iCurrFrame* (utile dans le cas d'une pause du programme par exemple). Le second paramètre est un booléen indiquant si le modèle doit être rendu avec interpolation linéaire (animation) ou pas. Enfin le dernier paramètre nous servira dans la partie suivante. Il nous permettra de choisir si l'on dessine le modèle comme jusqu'à présent ou à l'aide des commandes OpenGL. Pour le moment, ignorez-le.

Avant d'appeler *RenderFrame()* ou *DrawModelItp()* du modèle, il faut appliquer une rotation des axes pour « remettre à l'endroit » le modèle MD2. Celui-ci prend l'axe des Z comme hauteur (l'axe standard en mathématiques ou en physique), tandis qu'OpenGL le considère comme étant celui de la profondeur. Ensuite on passe le facteur de redimensionnement au modèle, puis on le dessine à l'écran. Dans le cas où le modèle doit être animé, on incrémente le pourcentage d'interpolation avec la valeur donnée à *Animate()*.

```
// -----
// DrawEntity() - dessine le modèle de l'entité.
// si le modèle n'est pas animé, on dessine la
// frame iFrame. Si cette dernière est négative,
// on dessine la frame courante.
// -----

void CEntity::DrawEntity( int iFrame, bool bAnimated, bool bUseGLCmds )
{
    glPushMatrix();
    // rotation des axes
    glRotatef( -90.0, 1.0, 0.0, 0.0 );
    glRotatef( -90.0, 0.0, 0.0, 1.0 );

    // redimensionnement du modèle
    m_pModel->SetScale( m_fScale );

    // rendu du modèle
    if( bAnimated )
    {
        // dessine chaque triangle du modèle
        m_pModel->DrawModelItp( m_iCurrFrame, m_iNextFrame, m_fInterp );

        // incrémentation du pourcentage d'interpolation entre les deux frames
        m_fInterp += m_fPercent;
    }
    else
    {
        // dessine chaque triangle du modèle
        if( iFrame == -1 )
            m_pModel->RenderFrame( m_iCurrFrame );
        else
            m_pModel->RenderFrame( iFrame );
    }

    glPopMatrix();
}
```

Brièvement, voyons comment utiliser ces classes et ces fonctions dans un programme OpenGL. On commence par créer un objet de type « modèle MD2 » qu'on initialise avec un fichier MD2, et un objet de type « entité » à qui on spécifie le modèle à utiliser pour le rendu en donnant l'adresse de l'objet MD2. Ensuite, dans la boucle de rendu, on appelle *Animate()* avec les première et dernières frames de l'animation et le pas d'interpolation (*m_fPercent*). Juste après on peut appeler *DrawEntity()*. Voici un exemple d'utilisation :

```
CMD2Model CyberPunkMd2;
CEntity CyberPunk;

// ...

// chargement des modèles
CyberPunkMd2.LoadModel( "models/cybpunk.md2" );
CyberPunkMd2.LoadTexture( "models/cybtexture.tga" );
CyberPunk.SetModel( &CyberPunkMd2 );
CyberPunk.SetScale( 0.1f );

// ...
```

```
// dans la boucle de rendu  
CyberPunk.Animate( 0, 39, 10/fps );  
CyberPunk.DrawEntity( -1, true, false );
```

8 - Rendu du modèle MD2 avec les commandes OpenGL

Cette partie aborde une autre méthode pour le rendu d'un modèle MD2, en utilisant des « commandes OpenGL » permettant de dessiner le modèle uniquement à l'aide des primitives `GL_TRIANGLE_FAN` et `GL_TRIANGLE_STRIP`. L'intérêt réside dans le temps de rendu plus rapide qu'en dessinant chaque triangle séparément comme on l'a fait jusqu'à maintenant. En effet, on passe moins de sommets à OpenGL. Et puisque ces commandes sont déjà calculées pour nous, autant ne pas s'en priver !

Comme nous l'avons vu lors de la description et de la lecture des données d'un fichier MD2, les commandes OpenGL sont stockées dans une liste d'entiers sur 4 octets (*int*). On dira que chaque entier équivaut à une commande. Le principe est de dérouler cette liste et d'exécuter une opération particulière selon leur valeur. On peut découper la liste en paquets. La première commande du paquet étant le nombre de sommets à dessiner soit en `GL_TRIANGLE_FAN` (si le nombre est négatif), soit en `GL_TRIANGLE_STRIP` (si le nombre est positif). Ensuite, on prend les commandes qui viennent par trois. Les deux premières valeurs étant les coordonnées de texture, la troisième l'index du sommet à dessiner (parmi la liste des sommets de la frame). La dernière commande de la liste est 0. Pour y voir un peu plus clair, un exemple sur un petit schéma :

Chaque case de bordure noire représente une commande de la liste, chaque case est codée sur quatre octets. Les coordonnées de texture sont des *float*. Elles sont pour le moment stockées dans un tableau de *int* mais ce n'est pas bien grave puisque le type *float* est de 32 bits comme le type *int* (enfin, pas partout... méfiance). Il suffira de faire un *cast* en *float*. Pour mieux s'y retrouver dans les petits blocs de $3 \times int$, on peut les regrouper dans une structure :

```
// commande OpenGL
typedef struct
{
    float    s;           // coordonnée de texture s
    float    t;           // coordonnée de texture t
    int      index;      // index vertex
} md2_glcmd_t;
```

Nous allons rajouter deux fonctions, `RenderFrameWithGLcmds()` et `DrawModelItpWithGLcmds()`, à notre classe `CMD2Model` afin de dessiner le modèle avec cet algorithme. À l'intérieur de ces deux fonctions on va utiliser un pointeur, `pGLcmds`, sur le tableau de commandes OpenGL. Ce pointeur va servir en quelque sorte de « tête de lecture ». On parcourt ainsi le tableau jusqu'à arriver à la dernière valeur qui vaut 0. Voici le code de la première fonction :

```
// -----
// RenderFrameWithGLcmds() - dessine le modèle à
// la frame spécifiée en utilisant les commandes
// OpenGL.
// -----

void CMD2Model::RenderFrameWithGLcmds( int iFrame )
{
    // calcul de l'index maximum d'une frame du modèle
    int iMaxFrame = m_kHeader.num_frames - 1;

    // vérification de la validité des indices de frame
    if( (iFrame < 0) || (iFrame > iMaxFrame) )
        return;
```

```

// activation de la texture du modèle
glBindTexture( GL_TEXTURE_2D, m_uiTexID );

// pointeur sur les commandes opengl
int *pGLCmds = &m_pGLCmds[0];

// on dessine chaque triangle!
while( int i = *(pGLCmds++) )
{
    if( i < 0 )
    {
        glBegin( GL_TRIANGLE_FAN );
        i = -i;
    }
    else
    {
        glBegin( GL_TRIANGLE_STRIP );
    }

    for( /* rien */; i > 0; i--, pGLCmds += 3 )
    {
        // pGLCmds[0] : coordonnée de texture s
        // pGLCmds[1] : coordonnée de texture t
        // pGLCmds[2] : index vertex à dessiner

        md2_glcmd_t *pGLcmd = (md2_glcmd_t *)pGLCmds;
        md2_frame_t *pFrame = &m_pFrames[ iFrame ];
        md2_vertex_t *pVert = &pFrame->verts[ pGLcmd->index ];

        // [coordonnées de texture]
        glTexCoord2f( pGLcmd->s, pGLcmd->t );

        // [normale]
        glNormal3fv( m_kAnorms[ pVert->normalIndex ] );

        // [vertex]
        vec3_t v;

        // calcul de la position absolue du vertex et redimensionnement
        v[0] = (pFrame->scale[0] * pVert->v[0] + pFrame->translate[0]) * m_fScale;
        v[1] = (pFrame->scale[1] * pVert->v[1] + pFrame->translate[1]) * m_fScale;
        v[2] = (pFrame->scale[2] * pVert->v[2] + pFrame->translate[2]) * m_fScale;

        glVertex3fv( v );
    }
    glEnd();
}
}

```

La variable *i* sert dans ces fonctions de compteur. Elle représente le nombre de sommets à dessiner dans ce bloc de primitives `GL_TRIANGLE_STRIP` ou `GL_TRIANGLE_FAN`. Lorsqu'elle est négative, on prend sa valeur absolue. Ensuite dans une boucle `for` on décrémente ce compteur en dessinant à chaque fois un sommet. Pour chaque sommet, il y a trois commandes OpenGL, donc on incrémente de trois le pointeur sur les commandes OpenGL.

Pour `DrawModelltpWithGLCmds()` c'est la même chose, sauf qu'on applique une interpolation linéaire entre le même sommet de deux frames. Ne pas oublier de vérifier la validité des numéros de frames passés en paramètres. Je ne vais pas copier le code ici, il ressemble beaucoup à `RenderFrameWithGLCmds()`. Vous pourrez vous exercer en la faisant vous même (en vous aidant de la méthode classique pour l'interpolation) ou bien directement aller fouiller dans le code source du programme (voir fin de l'article).

Il ne nous reste plus qu'à revoir notre fonction `DrawEntity()` pour qu'elle supporte le rendu via la

méthode classique et la méthode des commandes OpenGL. Voici la nouvelle fonction dans son intégralité :

```

// -----
// DrawEntity() - dessine le modèle de l'entité.
// si le modèle n'est pas animé, on dessine la
// frame iFrame. Si cette dernière est négative,
// on dessine la frame courante.
// -----

void CEntity::DrawEntity( int iFrame, bool bAnimated, bool bUseGLCmds )
{
    glPushMatrix();
    // rotation des axes
    glRotatef( -90.0, 1.0, 0.0, 0.0 );
    glRotatef( -90.0, 0.0, 0.0, 1.0 );

    // redimensionnement du modèle
    m_pModel->SetScale( m_fScale );

    // rendu du modèle
    if( bAnimated )
    {
        if( bUseGLCmds )
        {
            // dessine le modèle en utilisant les commandes OpenGL
            m_pModel->DrawModelItpWithGLcmds( m_iCurrFrame, m_iNextFrame, m_fInterp
);
        }
        else
        {
            // dessine chaque triangle du modèle
            m_pModel->DrawModelItp( m_iCurrFrame, m_iNextFrame, m_fInterp );
        }

        // incrémentation du pourcentage d'interpolation entre les deux frames
        m_fInterp += m_fPercent;
    }
    else
    {
        if( bUseGLCmds )
        {
            // dessine le modèle en utilisant les commandes OpenGL
            if( iFrame < 0 )
                m_pModel->RenderFrameWithGLcmds( m_iCurrFrame );
            else
                m_pModel->RenderFrameWithGLcmds( iFrame );
        }
        else
        {
            // dessine chaque triangle du modèle
            if( iFrame == -1 )
                m_pModel->RenderFrame( m_iCurrFrame );
            else
                m_pModel->RenderFrame( iFrame );
        }
    }

    glPopMatrix();
}

```

Voilà ce qui en est du rendu avec pour ces « commandes OpenGL ». Cet algorithme est plus rapide que la méthode classique (vous ne le verrez pas juste sur un seul petit modèle comme ça). Mais cela ne signifie pas qu'il faut jeter la première méthode pour autant ! Elle peut vous être utile si vous désirez implémenter d'autres effets sur le rendu du modèle. Par exemple, lorsque j'ai voulu intégrer du bump mapping dans mon programme (basique, avec de vieilles extensions ARB), j'ai du revenir sur le rendu en *GL_TRIANGLES*.

9 - Conclusion

Nous avons vu à travers cet article le format de fichier MD2 et sa structure, comment lire un modèle MD2, le dessiner puis l'animer. Nous avons vu également une méthode de rendu un peu particulière. Le code source complet est disponible avec un programme d'exemple, un modèle d'exemple (Cyber Punk) créé par Cedled (il vient du jeu [Warsow](#) avant qu'ils ne passent au format MD3). Vous pouvez l'utiliser sans restrictions. Aussi je joins le code source de mon implémentation basique (seule la lumière diffuse est appliquée) du bump mapping sur les modèles MD2. J'en ferai peut-être un article, m'enfin cette méthode est un peu vieillissante. Elle garde l'avantage de marcher sur du matériel plus ancien.

9.1 - Code Source

Le code source reste le même pour les trois paquetages, mais les bibliothèques incluses varient. Code source de l'article :

- [Windows MinGW](#) (973 Ko) - Code source, MakeFile et exécutable.

Code source avec Bump Mapping (les paquetages sont les mêmes) :

- [Linux GCC](#) (526,8 Ko) - Code source et MakeFile.
- [Windows MinGW](#) (1,0 Mo) - Code source, MakeFile, bibliothèques SDL et exécutable.
- [Windows Visual C++](#) (744,7 Ko) - Code source, Workspace, bibliothèques SDL et exécutable.

9.2 - Ressources

- [OpenGL Game Programing](#), Ch. 18, *K. Hawkins, D. Astle*.
- [Focus on 3D Models](#), Ch. 3, *Evan Piphlo*.
- [Game Tutorials](#), MD2 Loader, *Ben « DigiBen » Humphrey*.
- [Game Tutorials](#), MD2 Animation, *Ben « DigiBen » Humphrey*.
- [.md2 File Format Specification](#), *Daniel E. Schoenblum*.
- [Quake II](#) code source (GPL), *ID Software*.
- [MD2 Viewer](#) code source, *Mete Ciragan*.
- [qview](#) code source, *Mustata « LoneRunner » Bogdan*.
- [Qbism Game Engine](#) code source, *Jeff Ford*.
- [jawMD2](#) code source, *Jawed Karim*.

9.3 - Licence

Le code source de cet article téléchargeable plus haut est entièrement libre, de même pour la version avec bump-mapping (même pour une utilisation commerciale) ; libre à vous d'en faire ce que vous voulez. Il n'est pas nécessaire de me citer dans le cas d'une réutilisation.

Cet article est sous licence **CC-BY-ND**. Vous pouvez copier et redistribuer cet article à volonté à condition qu'il reste tel quel et dans son intégralité. Les schémas sont libres (domaine public). La correction de fautes de langue et de typographie est néanmoins tolérée (et appréciée !), ainsi que la traduction dans une autre langue (envoyez moi un petit mot, pour informations).

Contact : tfc_duke, chez club-internet point fr (désolé mais ces !%ù*\$ de robots à spam rodent partout :-)).

Version PDF : [télécharger](#) (95.0 Ko)