The MDC File Format

Wolfgang Prinz

May 2, 2004

Abstract

This document describes one of Return To Castle Wolfenstein's model file formats: the MDC. It is used mostly for storing minor models like heads and other, less dynamic game parts. A specialty of this format is that it allows compression of vertex based animation and therefor saving space.

The MDC format can be seen as successor of Quake's MD3, although, in Return To Castle Wolfenstein, the character animation is stored in another format: the MDS, which is not topic of this document.

Note: This is an unofficial document and neither supported, nor approved by official side. All copyrighted items that may appear in this document stay property of their legal owners.

Contents

1 Introduction 3

1.1 Acknowledgements . . . 3

1.2 Open Issues . . . 3

1.2.1 Naming . . . 3

1.2.2 Basic Data Type . . . 3

1.2.3 Base Scaling . . . 3

1.2.4 Delta Scaling . . . 4

1.2.5 Unknown Fields . . . 4

1.3 Contacts . . . 4

2 Primary File Structures 5

2.1 FileHeader . . . 5

2.2 BoundingFrames . . . 6

2.3 Tags . . . 6

3 Surface Structures 8

3.1 Surface Header . . . 8

3.2 Triangles . . . 9

3.3 Shaders . . . 9

3.4 Texture Coordinates . . . 10

3.5 Base Vertices . . . 10

3.6 Compressed Vertices . . . 11

4 Compressed Frames 12

4.1 Building Vertices . . . 12

4.2 Building Normals . . . 13

4.2.1 Theory . . . 13

4.2.2 The MDC Normal Coordinate Space . . . 13

4.2.3 Base Normal Encoding . . . 13

4.2.4 Final Normals . . . 14

5 Compressed Normals 15

5.1 Rho Ranges . . . 15

5.2 Sigma Intervals . . . 16

5.3 NullOffsets . . . 17

5.4 Converting from normal to n . . . 17

5.5 Converting from n to normal . . . 17

6 Conclusion 19

1 Introduction

1.1 Acknowledgements

This document is so to say the combination of information gathered from various sources concerning the Return To Castle Wolfenstein MDC file format.

The main facts come from Chris Cookson's "Return to Castle Wolfenstein MDC Importer".[1] This is also the source of most of the names mentioned here.

The base normal encoding algorithm was taken from the Quake 3 Arena Tool Source.[2] Especially the somehow misleading naming "longitude" and "latitude" for the spherical normal angles come from this code.

The compressed normal vector part of this document was done by myself using some statistical methods. Of course I am a bit proud of this.

1.2 Open Issues

The following section addresses issues that are not cleared up right now. Any suggestions would be gladly appreciated.

1.2.1 Naming

What do the terms MDC MDS and MD3 mean anyway?

I could guess MD stands for Model Definition and the last letter 3, C or S are something like versions, but that's all speculation.

1.2.2 Basic Data Type

The size of the basic data type is for sure 32 bits (least significant byte first). The question is whether it's signed or unsigned. Some MD3 implementations use the int or long (signed), others use unsigned long.

Because I've never seen any number larger than 2^{31} this isn't too important although it might cause severe errors when larger numbers appear somewhere.

1.2.3 Base Scaling

Usually, float base coordinate values are stored integrally as shorts in order to save space. To allow storing of non integral types, these values are multiplied

[1]`http://mojo.gmaxsupport.com/Sections/MaxScripts.html`

[2]`http://www.planetquake.com/quake3/files.shtml`

by a scaling factor, which shifts the binary comma. For regaining the original values, the inverse scaling has to be applied.

This could simply be done by multiplying the integral values from the file with some scaling factor. $\frac{1}{64}$ seems to be a good multiplicator, as the MD3 uses this value too. This is proved by the fact that you usually receive the bounding box values of a frame by dividing the maximum and minimum (unscaled) coordinates on each axis for this particular frame by 64.

1.2.4 Delta Scaling

Simply adding the delta frames to their base frames doesn't seem to perform so right. Usually such an approach moves the animated parts not far enough. For example, the mouth of a head doesn't open wide enough to form some speech.

This suggests that there is another scale factor to be applied. A good value seems to be 4.

1.2.5 Unknown Fields

There are fields in the data structures that couldn't be assigned a purpose. Usually those fields have the same number for all different MDCs.

1.3 Contacts

For suggestions, found errors and something like that see:

`http://www.planetwolfenstein.com/themdcfile`

or email to:

Wolfgang Prinz
wpATplanetwolfenstein.com (substitute AT with @)

2 Primary File Structures

Like most data files, the MDC File starts with a so called file header. The header provides information about where further data structures can be found in the rest of the file.

2.1 FileHeader

The FileHeader starts exactly at the beginning of the file.

```
typedef struct FileHeader
{
    unsigned long ulIdent;
    unsigned long ulVersion;
    unsigned char ucName [64];
    unsigned long ulFlags;
    unsigned long ulNumFrames;
    unsigned long ulNumTags;
    unsigned long ulNumSurfaces;
    unsigned long ulNumSkins;
    unsigned long ulOffsetBorderFrames;
    unsigned long ulOffsetTagNames;
    unsigned long ulOffsetTagFrames;
    unsigned long ulOffsetSurfaces;
    unsigned long ulOffsetEnd;
} FileHeader;
```

ulIdent should be ”IDPC” - this is the File Identification.

ulVersion should be 2, which means version 2.

ucName the file’s name (sometimes with path and extension).

ulFlags ??? - unknown (seems to be 0) Could be some file related flags.

ulNum* gives the number of * objects in the file (there are ulNum* objects at the appropriate offset).

ulOffset* is the offset (in bytes) of object * counting from the beginning of the file.

ulOffsetEnd is the offset to the first byte that does not belong to the file anymore (in other words - the file’s size).

2.2 BoundingFrames

A bounding box describes the outline box of a single animation frame. No Vertex should be out of this border.

```
typedef struct MDCPoint
{
  float x;
  float y;
  float z;
} MDCPoint;

struct MDCFrame
{
  MDCPoint bboxMin;
  MDCPoint bboxMax;
  MDCPoint localOrigin;
  float radius;
  char name[16];
};
```

bboxMin is the bottom left corner of the bounding box.

bboxMax is the top right corner of the bounding box.

localOrigin defines an offset added to all vertices (even the bounding box itself). It simply translates the whole model. This becomes important when this model is tagged to an MDS.

radius ??? - unknown - perhaps the radius of this frame.

name is the name of this frame.

Note: These coordinates here are not scaled (like the integral base frame coordinates) because they are stored directly as floating point values.

2.3 Tags

Tags are connections between the outer world and the MDC. For example, the head tag matches to the corresponding head tag in the corresponding MDS saying that the head should be put on that position.

```
typedef struct TagName
{
  unsigned char ucName [64];
} TagName;
typedef struct TagFrame
{
  typedef struct TagPosition
  {
    unsigned short x;
    unsigned short y;
    unsigned short z;
  } TagPosition;
  typedef TagPosition TagAngle;

  TagPosition  tpPosition;
  TagAngle     taAngle;
} TagFrame;
```

Every tag consists of a Tag Name and a Tag Frame, which are, for some reason, stored separately.

3 Surface Structures

This is where the vertex and texture coordinates are stored. A surface describes a continuous area on the model. For example, in the heads there is one surface for the head itself, the teeth (upper and lower respectively), the inner side of the mouth, and so on.

3.1 Surface Header

The header of the surface (similar to the FileHeader).

```
typedef struct SurfaceHeader
{
  unsigned long ulIdent;
  unsigned char ucName[64];
  unsigned long ulFlags;
  unsigned long ulNumCompFrames;
  unsigned long ulNumBaseFrames;
  unsigned long ulNumShaders;
  unsigned long ulNumVertices;
  unsigned long ulNumTriangles;
  unsigned long ulOffsetTriangles;
  unsigned long ulOffsetShaders;
  unsigned long ulOffsetTexCoords;
  unsigned long ulOffsetBaseVerts;
  unsigned long ulOffsetCompVerts;
  unsigned long ulOffsetFrameBaseFrames;
  unsigned long ulOffsetFrameCompFrames;
  unsigned long ulOffsetEnd;
} SurfaceHeader;
```

ulIdent ??? - unknown (seems to be 7).

ucName is the surface's name (e.g. “teeth_up”).

ulFlags ??? - unknown (seems to be 0).

ulNum* is the number of * objects located at the corresponding offsets.

ulOffset* is the offset of * counted from the start of this surface (not the beginning of the file).

ulOffsetEnd is the first byte of the next surface in the surface list counted from the start of this surface (remind that the space between the surface offset and ulOffsetEnd does not necessarily contain all surface data, but it's sure that ulOffsetEnd points to the next SurfaceHeader).

3.2 Triangles

The smallest 3D unit. Indices three Vertices in the form of a triangle list (which means that each "triangle" has three own vertices - its not a triangle fan or strip).

Note: This triangle list is responsible for all frames. Only vertex positions change during animation scaling or translating the triangles, but the triangle indices are not changed. All frames (of a single surface) are using the same triangle list (but they use different vertex sets for this list per animation frame).

```
typedef struct Triangle
{
  unsigned long ulVertex1;
  unsigned long ulVertex2;
  unsigned long ulVertex3;
} Triangle;
```

ulVertex1 indices the first vertex of this triangle.

ulVertex2 indices the second vertex of this triangle.

ulVertex3 indices the third vertex of this triangles.

The order of these three vertices is considered to be clockwise for defining the front face of the triangle.

3.3 Shaders

A shader holds the name of the texture of a surface. Because of skinning (which allows to "dress" one model with different "skins") the ucName does not always reflect a valid picture file in the file system. If there is a shader as well as a skin defining the texture file for a certain surface, the skin always wins.

```
typedef struct Shader
{
  unsigned char ucName [64];
```

```
    unsigned long ulFlags;
} Shader;
```

ucName is a string giving the name of the texture file.

ulFlags ??? - unknown

3.4 Texture Coordinates

Every vertex has a pair of values between 0.0 and 1.0 that tells the renderer which point of the texture should be mapped onto this pixel. Simply think of a glove (the texture - a 2D bitmap) that is pulled over your hand (the model) which itself consists of atoms (the vertices). Because we are in 3D, you cannot simply pull over the glove, but you have to give every single vertex the coordinates of a point on the glove. The rendering engine then interpolates the area between these vertices by using the color data from the texture.

```
typedef struct TexCoord
{
  float s;
  float t;
} TexCoord;
```

s is the x coordinate of the texture point counted from the left of the texture.

t is the y coordinate of the texture point counted from the top of the texture.

3.5 Base Vertices

The FrameBaseFrames is an array of numFrames unsigned short values. They are indices pointing to a BaseFrame.

Every BaseFrame contains numVertices BaseVertex elements. At offsetBaseFrames, there are numBaseFreames of such BaseFrames.

Note: It is possible that more than one FrameBaseFrame index points to the same BaseFrame.

```
typedef struct BaseVertex
{
  signed short x;
  signed short y;
  signed short z;
```

```
    unsigned short normal;
  } BaseVertex;
```

x is the x coordinate of that vertex.

y is the y coordinate of that vertex.

z is the z coordinate of that vertex.

normal is the normal vector encoded, like the Quake MD3 normal vectors, in spherical coordinates.

3.6 Compressed Vertices

Similar to the BaseFrames, there exists a table of indices, the FrameCompressedFrames, pointing to the CompressedFrames.

At offsetCompFrames, there are numCompFrames CompFrames each of them consisting of numVertices CompVertices.

Those CompVertices are compressed in order to save space. See section 4 and section 5 for more details.

```
  typedef struct CompVertex
  {
    unsigned char x;
    unsigned char y;
    unsigned char z;
    unsigned char n;
  } CompVertex;
```

x is the x coordinate difference.

y is the y coordinate difference.

z is the z coordinate difference.

n is the normal difference vector.

The term “difference” means that only the difference between the resulting frame (as can be seen by the user) and the base frame is stored. The intention behind this is to save space. It is assumed that successive frames differ only slightly, so only small data types are required to store those differences.

4 Compressed Frames

As you can see above, neither base nor compressed vertices are float values, which are required by rendering engines. In order to receive float vertices and normals, the values taken from the Base- and CompFrames have to be converted.

4.1 Building Vertices

The desired vertex vector v of an arbitrary frame vertex can be obtained by adding the difference vector v_{delta}, if any, and the local origin vector $v_{local\ origin}$ to the base vector v_{base}.

$$v = v_{local\ origin} + v_{base} + v_{delta}$$

If there is no compressed frame, which is indicated by the value `0xFFFF` in the FrameCompressedFrames table, Δv is set to zero (is not added).

The local origin vector $v_{local\ origin}$ is already given in world coordinates (floats) by the Bounding Frame data structure's *localOrigin*.

$$v_{local\ origin} = localOrigin$$

The base frame vector v_{base} can be obtained by dividing the integral base frame vector $BaseVertex$ by 64.

$$v_{base} = BaseVertex \cdot \frac{1}{64}$$

The delta vector v_{delta} can be received by subtracting 127 from every $CompVertex$ coordinate and applying some scaling. This scaling consists of the already used base vertex scale factor $\frac{1}{64}$ and an enlargement factor of 4 (which seems to be appropriate).

$$v_{delta} = \left[CompVertex - \begin{pmatrix} 127 \\ 127 \\ 127 \end{pmatrix} \right] \cdot 4 \cdot \frac{1}{64}$$

Summarized, the desired vertex vector v can be obtained by using:

$$v = localOrigin + \frac{1}{64} \cdot \left\{ BaseVertex + \left[CompVertex - \begin{pmatrix} 127 \\ 127 \\ 127 \end{pmatrix} \right] \cdot 4 \right\}$$

4.2 Building Normals

Normals are usually used by the rendering engines to create realistic light effects and shading. For this, every single vertex is assigned a normal vector (also called surface normal). This vector is normal to the surface that this vertex describes with its neighbors. A normal vector is supposed to point away from the outside of the surface.

4.2.1 Theory

As a normal vector only defines the normal direction of a surface, its length is unimportant. Usually, normal vectors have a length of one. Therefor, normal vectors are stored in spherical coordinates, which give the opportunity to omit the length of the vector and therefor save space.

Commonly, spherical coordinates are measured as longitude, latitude and radius. Longitude and latitude are angles, whereas radius is a metric value. This coordinate system is used to determine points on the earth's surface. Longitude is the angle turning from the prime meridian towards east and latitude measures the angle from the equator due north or south.

In an MDC (and MD3), the latitude is measured differently. It describes an angle turning from the north pole towards the south pole. In order to avoid confusion, this latitude will be called ρ ("rho") and the longitude will be renamed to σ ("sigma").

4.2.2 The MDC Normal Coordinate Space

Applied to the cartesian coordinate space of the MDC, ρ turns from the positive z axis towards the x axis (and further). Therefor ρ operates in the xz plane. It usually has a range of $\rho \in [0^\circ \cdots 180^\circ]$ (although the used encoding would allow bigger values).

The σ angle turns from the positive x axis towards the y axis (and further). Therefor σ operates in the xy plane. It usually has a range of $\rho \in [0^\circ \cdots 360^\circ)$.

Now, with given ρ and σ, the desired normal vector can be computed by rotating the z unit vector by ρ around the y axis towards the x axis. Then the resulting vector is rotated by σ around the z axis from the xz plane towards the y axis.

4.2.3 Base Normal Encoding

The ρ and σ values of the base frames are not stored as floating point values, but as single bytes. The possible angle range of 360° is linearly mapped to the possible byte range of $2^8 = 256$ values.

The least significant byte of the normal field of a BaseVertex is the encoded ρ, whereas the most significant byte symbolizes σ.

Given the unsigned char *Byte*, the angle *Angle* (in degrees) can be calculated using

$$Angle = Byte \cdot \frac{360^\circ}{256}$$

Note: Here, the angles are measured in degrees, but in reality it would be better to measure them in radiants because most mathematical libraries work only with radiant angles.

4.2.4 Final Normals

Again, to get the final normal of a certain animation frame, the base and the compressed normals have to be added together by means of vector addition.

This is most easily done in cartesian coordinate space. So the base normal (ρ, σ) has to be converted to (x, y, z) using

$$\begin{aligned} x &= \cos(\sigma) \cdot \sin(\rho) \\ y &= \cos(\sigma) \cdot \sin(\rho) \\ z &= \cos(\rho) \end{aligned}$$

Vector addition is performed by simply adding the x, y and z components of both vectors respectively.

How the compressed normal vector can be formed is discussed in section 5.

5 Compressed Normals

The encoding of the compressed normals is a bit more complicated. That's why this is described in an own chapter here.

Like the uncompressed normals, the compressed ones encode a vector in spherical form too. The final normal seems to be an addition of the base and the compressed normal vectors.

Remember how sperical coordinates are mapped to cartesians: First, the z unit vector is rotated along the y axis towards the x axis by the angle of ρ ("rho"). Then the resulting vector is rotated around the z axis towards the y axis by σ ("sigma").

Unfortunately the encoding of these two values ρ and σ into one single byte is a bit more sophisticated than the delta compression of the x, y and z coordinates. In the following text, let n be the normal part of the compressed vertex. As n has a size of one byte it ranges from 0 to 255. Also, let ρ be an angle from 0° to 180° and σ be an angle from 0° to 360° degrees.

As the compressed parts in an MDC file mostly describe head or, more accurate, mouth animations, rotation around the y axis will happen more often than rotation around the other two axis. Remember how a human head is positioned in the cartesian coordinate space. Face and nose point along the positive x axis, the hair points upwards, along the positive z axis, and the left ear points along the positive y axis. So "talking" will result in rotating pixels near to the x axis, the "mouth", up and down the z axis and therefor rotating their normals with them.

This seems to be the reason why the σ values are not evenly distributed over n. Around a ρ of 90° the corresponding σ is finely grained whereas on the poles at 0° and 180° σ is only coarse grained.

In the following, the term ρ-range identifies a certain sub interval of the possible ρ values. The term σ-interval refers to a certain interval of the σ values.

5.1 Rho Ranges

The range of possible ρ values is divided (sub sampled) into 15 disjoint intervals, the ρ-ranges, of nearly equal size.

The boundaries ν_i (in degrees) of these ranges can be calculated by

$$\nu_i = \frac{180^\circ}{16} \cdot i + \frac{1}{2} \cdot \frac{180^\circ}{16}$$

where i goes from 1 to 14.

ρ-ranges [°]	ρ median [°]	σ width	NullOffset
$(0 \cdots 16.875]$	11.25	4	252
$(16.875 \cdots 28.125]$	22.5	8	244
$(28.125 \cdots 39.375]$	33.75	12	232
$(39.375 \cdots 50.625]$	45	16	216
$(50.625 \cdots 61.875)$	56.25	20	196
$[61.875 \cdots 73.125)$	67.5	24	172
$[73.125 \cdots 84.375)$	78.75	28	144
$[84.375 \cdots 95.625)$	90	32	0
$[95.625 \cdots 106.875)$	101.25	28	32
$[106.875 \cdots 118.125)$	112.5	24	60
$[118.125 \cdots 129.375)$	123.75	20	84
$[129.375 \cdots 140.625)$	135	16	104
$[140.625 \cdots 151.875)$	146.25	12	120
$[151.875 \cdots 163.125)$	157.5	8	132
$[163.125 \cdots 180)$	168.75	4	140

Table 1: The ρ-ranges and their medians.

Interestingly, those values ν_i are neither all upper nor all lower bounds. In fact, the first 4 are upper, and the last 10 are lower boundaries.

Table 1 depicts which ρ values belong to which ρ-range. The ρ medians are the ρ values in the middle of each ρ-range an can be calculated using

$$\rho \text{ median} = \frac{180°}{16} \cdot i$$

where i goes from 1 to 15.

Note: "(" and ")" mean that a certain value doesn't belong to the ρ-range anymore, whereas "[" and "]" signal that this value is the first or last one belonging to that range.

5.2 Sigma Intervals

The complexity rises with the σ-intervals. A σ width is assigned to every ρ-range telling how many numbers of n are spent to encode the σ angle.

For example, a σ width of 16 means that there are 16 different σ-intervals allocated for that particular ρ-range.

The boundaries μ_i of the σ-intervals can be calculated using

$$\mu_i = \frac{360°}{\sigma \text{ width}} \cdot i - \frac{1}{2} \cdot \frac{360°}{\sigma \text{ width}}$$

where i goes from 0 to σ width.

Note: A negative value of σ can be interpreted as 360° minus the absolute value of that σ.

The σ-intervals then have the ranges:

$$[\mu_i \cdots \mu_{i+1})$$

with i going from 0 to σ width $-$ 1.

The σ medians of the σ-intervals can be calculated using

$$\sigma \text{ median} = \frac{360°}{\sigma \text{ width}} \cdot n'$$

where n' goes from 0 to σ width $-$ 1.

5.3 NullOffsets

In order to compile the ρ-ranges and corresponding σ-intervals into n, the σ-interval's index value n' is added to the ρ-ranges NullOffset.

The NullOffset is just a running sum over the σ-widths starting at the interval with the largest width (at $\rho = 90°$) counting upwards and then wrapping around. Table 1 demonstrates this.

5.4 Converting from normal to n

Given a certain normal (ρ, σ), the first thing to do is match ρ to a ρ-range. This gives the NullOffset and the σ-width. The σ-interval's index value n' can then be calculated matching σ against the boundaries μ_i.

Then n can be obtained with

$$n = \text{NullOffset} + n'$$

5.5 Converting from n to normal

First, the ρ-range has to be determined by matching n against the NullOffsets. The destination ρ value is then the median of the ρ-range.

The σ-interval's index value n' can be calculated using

$$n' = n - \text{NullOffset}$$

Again, the destination σ value is the median of the σ-interval and can be calculated with

$$\sigma = \frac{360°}{\sigma \text{ width}} \cdot n'$$

Note: This conversion can dramatically be improved by using a lookup table. This table simply contains the (ρ, σ) medians or, better, the (x, y, z) values for every possible n, which is ranging from 0 to 255. A good start for such a lookup table is table 1.

6 Conclusion

This document showed how to read and interpret the model data found in an MDC file. It was intended to help people developing plugins for 3D editors or simialr things.

Unfortunately there has never been much official respondence to questions concerning the game engine. To my knowledge, all source code releases so far were only about the user interface engine. Therefor all information here is only as accurate as it could be without knowing engine implementation details.

The first parts of this document are heavily bases upon my knowledge of the topic gathered from Chris Cookson's gmax script. The research about compressed normal vectors was done by myself in painstaking programming, graph plotting and calculator typing sessions.

Although this information here is not official, it seems to be more or less accurate anyway. I have tried out viewing a lot of MDCs using the techniques described in this document and it seems to fit quite well.

Hopefully, this document can be of help to someone.