VM LABS

M3DL

2D & 3D Graphics Library For

NUON™

Revision 1.13
29-Oct-01

VM Labs, Inc.
520 San Antonio Rd
Mountain View, CA 94040

Tel: (650) 917-8050
Fax: (650) 917-8052

This is a preliminary specification. VM Labs reserves the right to make changes to any information described in this document.

Table of Contents

1. Introduction

1.1 What is M3DL?

M3DL is a library for creating 2D and 3D graphics on NUON. It was originally created by one of the first NUON developers, Miracle Designs, for use in their own NUON game development project. VM Labs has arranged to make the M3DL library available to all NUON developers. It is designed to be familiar and easy for experienced console game developers to learn and use.

1.2 The Structure of M3DL

The M3DL library is broken down into two main parts. The main API runs on the primary processor of the NUON chip (MPE 3) and is accessible through standard C language functions. This code is also responsible for setup functions, 3D coordinate transformation, and generally everything except the actual rendering of the display.

The other main part of the library is responsible for low-level rendering tasks. This code is not accessed directly by the application, and does not run on the primary processor.

The M3DL rendering code is downloaded to the one or more processors as specified by your application. Each processor used for rendering is known as an "MPR", which stands for "Merlin Primitive Renderer".[1] The term "MPR chain" is used to collectively refer to all of the MPEs[2] being used for M3DL rendering.

The MPR chain is discussed in further detail in chapter 2.

[1] "Merlin" is a reference to the early days of NUON, when "Merlin" was the codename used within VM Labs to refer to the chip itself.

[2] An "MPE" is a "Merlin Processing Element". This is what we call one of the individual processors within the NUON chip. If you didn't already know what "MPE" meant, then you may want to refer to the NUON "Programmer's Guide" document to familiarize yourself with the basic concept and terminology of NUON.

This page intentionally left blank

2. M3DL Configuration

2.1 Introduction

This section discusses general functions for configuring the MPR rendering chain, utility functions for color manipulation, etc.

2.1.1 The MPR Chain

The MPR chain is the portion of the M3DL library responsible for low-level rendering tasks. This code runs on MPE 0, 1, and/or 2. It does no 3D calculations, polygon backface culling, hidden surface removal, etc. The MPR code is dedicated to the requirements of rendering individual primitives as quickly as possible.

Likewise, the code that runs on the primary processor (MPE 3), is responsible for 3D calculations, polygon backface culling, hidden surface removal, etc.

Before you can render anything with M3DL, you must first configure the MPR chain to run on the desired MPE processors using the ***mdSetupMPRChain()*** function. This causes the main library to download the MPR program code to the processors which you have specified.

When you call an M3DL function that requires rendering, the library code running on the primary processor sends a message to the next available MPR processor. In response, that processor performs whatever specific rendering task is indicated.

2.1.2 Using M3DL Without An MPR Chain

Many M3DL functions that do not require rendering may be used without an active MPR chain. This is an important consideration if you are sharing processors between M3DL and other tasks.

For the most part, if an M3DL function does not require rendering, or does not attempt to update the current rendering state, then that function can be used without an active MPR chain. This includes functions such as the frame buffer setup functions described in sections 3.5 and 3.6, most of the utility functions described in chapter 4, the 3D graphics functions described in chapter 0, and the materials functions described in chapter 8.

The following M3DL library functions require an active MPR chain to work.

mdClearDraw()	mdClearDisp()	mdDrawPoly()	mdDrawSprite()
mdDrawImage()	mdDrawTile()	mdDrawPrim()	mdRenderObject()
mdRenderObjectAmbient()	mdSetTransparencyMode()	mdActiveBlendColor()	mdActiveDrawContext()
mdDrawSync()	mdRemoveMPRChain()	SwapDrawBufYCC()	SwapDrawBufRGB()

Any functions not in this list should work without an active MPR chain.

2.2 MPE Usage Configuration Functions

2.2.1 mdSetupMPRChain

Set up the specified MPEs as part of the M3DL rendering chain.

```
void mdSetupMPRChain(startmpe, nummpes)
```

Function Arguments		
Type	Name	Description
`mdUINT32`	`startmpe`	First MPE to use
`mdUINT32`	`nummpes`	Number of MPEs to use

2.2.2 mdRemoveMPRChain

Free MPEs reserved by the ***mdSetupMPRChain***() function.

```
void mdRemoveMPRChain(void)
```

2.3 MPR Configuration Functions

2.3.1 mdActiveDrawContext

Send Screen Buffer information to MPRs

```
void mdActiveDrawContext(dcx)
```

Function Arguments		
Type	Name	Description
`mdDRAWCONTEXT *`	`dcx`	Pointer to drawing context

2.4 MPR Synchronization Functions

2.4.1 mdDrawSync

Wait for MPR activity to finish.

```
void mdDrawSync(void)
```

3. Frame Buffer Setup

3.1 Introduction

The M3DL library includes a variety of frame buffer allocation functions that create buffers in SDRAM for rendering and displaying graphics. This section will discuss these functions and related concepts.

3.1.1 Select Your Color Model

The M3DL library supports rendering in either the RGB color model, or the YCrCb color model. Each mode has advantages and disadvantages.

3.1.1.1 What is RGB?

The term “RGB” is an acronym for “Red, Green, Blue”. In computer graphics terms, an RGB mode pixel is constructed of three separate values which represent the Red, Green, and Blue portions of the particular color assigned to that pixel.

The RGB color model is useful in many ways for computer graphics. In particular, RGB allows mathematical operations on color values to be done quickly and easily, with results that match how light works in the real world.

For example, if you use colored lights in your 3D world, and you add a red light source and a green light source, you get yellow. This sort of thing is not possible with any other color model.

The main disadvantage to RGB mode is that the current generation of NUON hardware does not allow your video display buffer to be in RGB format. This ability will probably be added to future generations of the NUON chip, but for now it means that whenever a program creates a display in RGB mode, it must be converted to YCrCb before it is displayed.

Fortunately, the M3DL library does that for you. The only downside to using RGB mode is that there is a small performance hit because each completed frame of graphics must be converted before it is displayed.

3.1.1.2 What is YCrCb?

Using YCrCb mode is normally recommended on NUON because this is the mode natively supported by the hardware. The terms “YCrCb” and “YCC” may be used interchangeably.

YCrCb mode pixels consist of three components. The first component is “Y” and represents the luminance value, or brightness. The “Cr” and “Cb” components define the “chroma” part of the pixel, or the color. Together these values represent a particular position on a 2-dimensional grid that contains all of the available hues.

YCrCb is used as the native mode by the NUON hardware because the YCrCb color space is used by color television as well as the MPEG video compression format.

Unfortunately, the YCrCb color space isn’t exactly ideal for computer graphics. While there are a few unique advantages, they are outweighed by the fact that YCrCb color values cannot be mathematically manipulated as easily as RGB color values. This means that color values must frequently be translated back and forth to RGB mode. This is done by M3DL in most cases, but there may be instances where your program will have to address this issue.

For a good visual example of what YCrCb is all about, look at the YCrCb color square sample program included in the NUON SDK.

3.2 Frame Buffer Initialization

The first M3DL function normally used by an application is one of the M3DL functions for allocating one or more frame buffers. This initializes the drawing context structures that will be used hereafter by the library.

Which specific function is called depends on the desired frame buffer format. M3DL supports a variety of different buffer formats. You may choose 16-bit or 32-bit and either YCrCb color space or RGB color space.

3.2.1 Using YCrCb Mode

When using YCrCb mode, you may select between 1 and 3 buffers. If you select a single buffer, it will be used for both rendering and as the display. This is very economical regarding memory usage, but may show “tearing” when your application draws into the buffer.

If you specify two buffers, they can be used for a traditional double buffering setup where one buffer is used to render while the other is used for the current display. Your application can also use triple buffering if you specify three buffers.

The main drawback to using YCrCb mode is that blending doesn’t really work right, nor do colored lighting effects. These require the RGB color space. It would be possible to perform color conversions on the fly, but this would slow down performance so much that it’s not really practical.

3.2.2 Using RGB Mode

RGB modes are actually represented internally as GRB.[3] The terms “RGB” and “GRB” may be used interchangeably unless referring specifically to the layout of individual pixels.

The main drawback to using RGB mode is that the NUON video display hardware works only with YCrCb data. Therefore, when your program renders data in RGB mode, M3DL must convert it to YCrCb before it can be displayed. The library handles this automatically, but it does consume a certain amount of processing time that may impact your application’s performance.

The screen setup functions that specify RGB mode always set up a total of three buffers:

- Render buffer in GRB format, either 16-bit or 32-bit depending on the setup function used. In 32-bit mode the buffer format is Green-Red-Blue-Alpha.
- Two display buffers in YCrCb format. One is used to hold the data currently being displayed, the other is used as the target of the required RGB to YCrCb color space conversion.

There are separate calls for selecting various combinations of 16-bit and 32-bit buffers. You can, for example, render into a 16-bit RGB mode but use a 32-bit YCrCb mode for the display buffer.

3.2.3 A Note About Color Values

Note that the **mdCOLOR** structure used by the M3DL library always describes a color in RGB color space, even when rendering in YCrCb mode. The M3DL library automatically handles conversion to YCrCb color space when required.

3.2.3.1 Texture Data & Frame Buffer Format

Note that your texture data must always match the current rendering mode. If you are using RGB mode, then your textures must all be in RGB format. If you are using YCrCb mode, then your textures must all be in YCrCb format.

3.3 Drawing Context

The drawing context structure initialized by your M3DL frame buffer setup call is used to track which frame buffer is being used for rendering, which one is used for the current display, and so forth. You normally have either one or two drawing context structures, stored as an array, depending on which color model you are using. The table below shows which drawing context structure is associated with what:

[3] This refers to the way the individual values for Red, Green, and Blue are stored within the pixel. “RGB” means that the “red” value comes first, followed by green and blue, whereas “GRB” means green first, then red, then blue. However, you need to be aware that this convention is not consistently followed from one platform to another, and does not always take into consideration the issue of how data is stored by a given processor (i.e. big-endian versus little-endian).

	dcx[0]	dcx[1]
RGB Mode	Refers to RGB rendering buffer	Refers to YCrCb display buffer
YCrCb Mode	Refers to YCrCb rendering & display buffer(s)	Not used

The **actbuf** field of the drawing context indicates the "active" buffer. That is, the buffer that should be currently on display or used for rendering. This value is used to determine which buffer data is passed to the ***VidSetup()*** call.

3.4 Video Setup

All of the buffer setup functions described in sections 3.5 and 3.6 below expect that an application will call the BIOS ***VidSetup()*** function to configure the display. For example:

```
mdDrawContext dcx[2];
mdBYTE *buf, *sdramadddr;
int dmaflags;

sdramaddr = (mdBYTE*)0x40000000;

sdramlen = mdSetBufGRB16B_WITHZ_YCC32B( dcx, sdramaddr, 360, 240, 20, 8, 320, 224);

buf = (mdBYTE*)dcx[1].buf[dcx[1].actbuf].sdramaddr;

dmaflags = dcx[1].buf[dcx[1].actbuf].dmaflags;

VidSetup( buf, dmaflags, dcx[1].dispw, dcx[1].disph,0);
```

The ***mdSetBuf******()*** function initializes the drawing context contained in **dcx**. After the call, the drawing context will contain the information that must be passed to the ***VidSetup()*** function to configure the display.

Note that we used the values from **dcx[1]** for doing our video setup. This is because **dcx[1]** refers to the YCrCb buffers used for display, while **dcx[0]** refers to our RGB rendering buffer. If the example above had specified a YCrCb mode, then the video setup would have used **dcx[0]** instead.

3.5 YCrCb Screen Setup Functions

3.5.1 mdSetBufYCC16B_NOZ

Set up screen buffers. Pixel format is 16-bit YCrCb, no Z-buffer.

```
mdUINT32 size = mdSetBufYCC16B_NOZ( dcx, sdram, nbuf, dispw, disph,
                                    rendx, rendy, rendw, rendh)
```

Function Arguments & Return Code		
Type	Name	Description
`mdDRAWCONTEXT *`	`dcx`	Pointer to drawing context
`mdBYTE *`	`sdram`	Address of buffer in SDRAM
`mdUINT32`	`nbuf`	Number of buffers
`mdUINT32`	`dispw`	Display Width
`mdUINT32`	`disph`	Display Height
`mdUINT32`	`rendx`	Render buffer x offset
`mdUINT32`	`rendy`	Render buffer y offset
`mdUINT32`	`rendw`	Render buffer width
`mdUINT32`	`rendh`	Render buffer height
`mdUINT32`	`size`	Amount of SDRAM used for buffers

3.5.2 mdSetBufYCC32B_NOZ

Set up screen buffers. Pixel format is 32-bit YCrCb, no Z-buffer.

```
mdUINT32 size = mdSetBufYCC32B_NOZ( dcx, sdram, nbuf, dispw, disph,
                                    rendx, rendy, rendw, rendh)
```

Function Arguments & Return Code		
Type	**Name**	**Description**
mdDRAWCONTEXT *	dcx	Pointer to drawing context
mdBYTE *	sdram	Address of buffer in SDRAM
mdUINT32	nbuf	Number of buffers
mdUINT32	dispw	Display Width
mdUINT32	disph	Display Height
mdUINT32	rendx	Render buffer x offset
mdUINT32	rendy	Render buffer y offset
mdUINT32	rendw	Render buffer width
mdUINT32	rendh	Render buffer height
mdUINT32	size	Amount of SDRAM used for buffers

3.5.3 mdSetBufYCC16B_WITHZ

Set up screen buffers. Pixel format is 16-bit YCrCb, with 16-bit Z-buffer.

```
mdUINT32 size = mdSetBufYCC16B_WITHZ( dcx, sdram, nbuf, dispw, disph,
                                      rendx, rendy, rendw, rendh)
```

Function Arguments & Return Code		
Type	**Name**	**Description**
mdDRAWCONTEXT *	dcx	Pointer to drawing context
mdBYTE *	sdram	Address of buffer in SDRAM
mdUINT32	nbuf	Number of buffers
mdUINT32	dispw	Display Width
mdUINT32	disph	Display Height
mdUINT32	rendx	Render buffer x offset
mdUINT32	rendy	Render buffer y offset
mdUINT32	rendw	Render buffer width
mdUINT32	rendh	Render buffer height
mdUINT32	size	Amount of SDRAM used for buffers

3.5.4 mdSetBufYCC32B_WITHZ

Set up screen buffers. Pixel format is 32-bit YCrCb, with 32-bit Z-buffer.

```
mdUINT32 size = mdSetBufYCC32B_WITHZ( dcx, sdram, nbuf, dispw, disph,
                                      rendx, rendy, rendw, rendh)
```

Function Arguments & Return Code		
Type	**Name**	**Description**
mdDRAWCONTEXT *	dcx	Pointer to drawing context
mdBYTE *	sdram	Address of buffer in SDRAM
mdUINT32	nbuf	Number of buffers
mdUINT32	dispw	Display Width
mdUINT32	disph	Display Height
mdUINT32	rendx	Render buffer x offset
mdUINT32	rendy	Render buffer y offset
mdUINT32	rendw	Render buffer width
mdUINT32	rendh	Render buffer height
mdUINT32	size	Amount of SDRAM used for buffers

3.5.5 mdSetBufYCC16B_WITHZSHARED

Set up screen buffers. Pixel format is 16-bit YcrCb for both the render buffer and display buffer, with a shared 16-bit Z-buffer.

```
mdUINT32 size = mdSetBufYCC16B_WITHZSHARED( dcx, sdram, nbuf, dispw, disph,
                                            rendx, rendy, rendw, rendh)
```

Function Arguments & Return Code		
Type	**Name**	**Description**
mdDRAWCONTEXT *	dcx	Pointer to drawing context

Function Arguments & Return Code		
Type	Name	Description
mdBYTE *	sdram	Address of buffer in SDRAM
mdUINT32	nbuf	Number of buffers
mdUINT32	dispw	Display Width
mdUINT32	disph	Display Height
mdUINT32	rendx	Render buffer x offset
mdUINT32	rendy	Render buffer y offset
mdUINT32	rendw	Render buffer width
mdUINT32	rendh	Render buffer height
mdUINT32	size	Amount of SDRAM used for buffers

3.6 RGB Screen Setup Functions

3.6.1 mdSetBufGRB16B_NOZ_YCC16B

Set up screen buffers. Pixel format for render buffer is 16-bit GRB (Green, Red, Blue). Display buffer is 16-bit YCrCb. No Z-buffer. Always sets up a total of 3 buffers (1 GRB render buffer + 2 YCrCb display buffers.)

```
mdUINT32 size = mdSetBufGRB16B_NOZ_YCC16B( dcx, sdram, dispw, disph, rendx, rendy,
rendw, rendh)
```

Function Arguments & Return Code		
Type	Name	Description
mdDRAWCONTEXT *	dcx	Pointer to drawing context
mdBYTE *	sdram	Address of buffer in SDRAM
mdUINT32	dispw	Display Width
mdUINT32	disph	Display Height
mdUINT32	rendx	Render buffer x offset
mdUINT32	rendy	Render buffer y offset
mdUINT32	rendw	Render buffer width
mdUINT32	rendh	Render buffer height
mdUINT32	size	Amount of SDRAM used for buffers

3.6.2 mdSetBufGRB16B_NOZ_YCC32B

Set up screen buffers. Pixel format for render buffer is 16-bit GRB (Green, Red, Blue). Display buffer is 32-bit YCrCb. No Z-buffer. Always sets up a total of 3 buffers (1 GRB render buffer + 2 YCrCb display buffers.)

```
mdUINT32 size = mdSetBufGRB16B_NOZ_YCC32B( dcx, sdram, dispw, disph,
                                           rendx, rendy, rendw, rendh)
```

Function Arguments & Return Code		
Type	Name	Description
mdDRAWCONTEXT *	dcx	Pointer to drawing context
mdBYTE *	sdram	Address of buffer in SDRAM
mdUINT32	dispw	Display Width
mdUINT32	disph	Display Height
mdUINT32	rendx	Render buffer x offset
mdUINT32	rendy	Render buffer y offset
mdUINT32	rendw	Render buffer width
mdUINT32	rendh	Render buffer height
mdUINT32	size	Amount of SDRAM used for buffers

3.6.3 mdSetBufGRB32B_NOZ_YCC32B

Set up screen buffers. Pixel format for render buffer is 32-bit GRB (Green, Red, Blue). Display buffer is 32-bit YCrCb. No Z-buffer. Always sets up a total of 3 buffers (1 GRB render buffer + 2 YCrCb display buffers.)

```
mdUINT32 size = mdSetBufGRB32B_NOZ_YCC32B( dcx, sdram, dispw, disph,
                                            rendx, rendy, rendw, rendh)
```

Function Arguments & Return Code		
Type	**Name**	**Description**
mdDRAWCONTEXT *	dcx	Pointer to drawing context
mdBYTE *	sdram	Address of buffer in SDRAM
mdUINT32	dispw	Display Width
mdUINT32	disph	Display Height
mdUINT32	rendx	Render buffer x offset
mdUINT32	rendy	Render buffer y offset
mdUINT32	rendw	Render buffer width
mdUINT32	rendh	Render buffer height
mdUINT32	size	Amount of SDRAM used for buffers

3.6.4 mdSetBufGRB16B_WITHZ_YCC16B

Set up screen buffers. Pixel format of render buffer is 16-bit bit GRB (Green, Red, Blue) with 16-Bit Z-buffer. Display buffer is 16-bit YCrCb, no Z-buffer. Always sets up a total of 3 buffers (1 GRB render buffer + 2 YCrCb display buffers.)

```
mdUINT32 size = mdSetBufGRB16B_WITHZ_YCC16B( dcx, sdram, dispw, disph, rendx, rendy,
rendw, rendh)
```

Function Arguments & Return Code		
Type	**Name**	**Description**
mdDRAWCONTEXT *	dcx	Pointer to drawing context
mdBYTE *	sdram	Address of buffer in SDRAM
mdUINT32	dispw	Display Width
mdUINT32	disph	Display Height
mdUINT32	rendx	Render buffer x offset
mdUINT32	rendy	Render buffer y offset
mdUINT32	rendw	Render buffer width
mdUINT32	rendh	Render buffer height
mdUINT32	size	Amount of SDRAM used for buffers

3.6.5 mdSetBufGRB16B_WITHZ_YCC32B

Set up screen buffers. Pixel format of render buffer is 16-bit bit GRB (Green, Red, Blue) with 16-bit Z buffer. Display buffer is 32-bit YCrCb with no Z-buffer. Always sets up a total of 3 buffers (1 GRB render buffer + 2 YCrCb display buffers.)

```
mdUINT32 size = mdSetBufGRB16B_WITHZ_YCC32B( dcx, sdram, dispw, disph,
                                             rendx, rendy, rendw, rendh)
```

Function Arguments & Return Code		
Type	**Name**	**Description**
mdDRAWCONTEXT *	dcx	Pointer to drawing context
mdBYTE *	sdram	Address of buffer in SDRAM
mdUINT32	dispw	Display Width
mdUINT32	disph	Display Height
mdUINT32	rendx	Render buffer x offset
mdUINT32	rendy	Render buffer y offset
mdUINT32	rendw	Render buffer width
mdUINT32	rendh	Render buffer height
mdUINT32	size	Amount of SDRAM used for buffers

3.6.6 mdSetBufGRB32B_WITHZ_YCC32B

Set up screen buffers. Pixel format of render buffer is 32-bit bit GRB (Green, Red, Blue) with 32-bit Z buffer. Display buffer is 32-bit YCrCb with no Z-buffer. Always sets up a total of 3 buffers (1 GRB render buffer + 2 YCrCb display buffers.)

```
mdUINT32 size = mdSetBufGRB32B_WITHZ_YCC32B( dcx, sdram, dispw, disph,
                                             rendx, rendy, rendw, rendh)
```

Function Arguments & Return Code		
Type	Name	Description
`mdDRAWCONTEXT *`	`dcx`	Pointer to drawing context
`mdBYTE *`	`sdram`	Address of buffer in SDRAM
`mdUINT32`	`dispw`	Display Width
`mdUINT32`	`disph`	Display Height
`mdUINT32`	`rendx`	Render buffer x offset
`mdUINT32`	`rendy`	Render buffer y offset
`mdUINT32`	`rendw`	Render buffer width
`mdUINT32`	`rendh`	Render buffer height
`mdUINT32`	`size`	Amount of SDRAM used for buffers

3.7 Buffer Swap Functions

3.7.1 SwapDrawBufGRB

Swap between buffers in GRB (Green, Red, Blue) mode. Converts a GRB render buffer into YCrCb format for display. It then updates the drawing context information and calls the BIOS function ***_VidChangeBase()*** to set the new buffer address.

```
mdUINT32 fields = SwapDrawBufGRB(dcx)
```

Function Arguments & Return Code		
Type	Name	Description
`mdDRAWCONTEXT *`	`dcx`	Pointer to drawing context
`mdUINT32`	`fields`	Number of video fields elapsed since previous buffer swap.

3.7.2 SwapDrawBufYCC

Swap between buffers in YCrCb mode. It updates the drawing context information and calls the BIOS function ***_VidChangeBase()*** to set the new buffer address.

```
mdUINT32 fields = SwapDrawBufYCC(dcx)
```

Function Arguments & Return Code		
Type	Name	Description
`mdDRAWCONTEXT *`	`dcx`	Pointer to drawing context
`mdUINT32`	`fields`	Number of video fields elapsed since previous buffer swap.

3.8 Buffer Clear Functions

3.8.1 mdClearDraw

Clear the current draw buffer with the specified color.

```
void mdClearDraw(dcx,color)
```

Function Arguments		
Type	Name	Description
`mdDRAWCONTEXT *`	`dcx`	Pointer to drawing context
`mdCOLOR *`	`color`	Pointer to mdCOLOR structure containing RGB color to use. (See section 3.2.3.)

3.8.2 mdClearDisp

Clear the display buffer with the specified color.

```
void mdClearDisp(dcx,color)
```

Function Arguments		
Type	Name	Description
mdDRAWCONTEXT *	dcx	Pointer to drawing context
mdCOLOR *	color	Pointer to mdCOLOR structure containing RGB color to use. (See section 3.2.3.)

4. Utility Functions

4.1 Introduction

This chapter describes functions of the library which are primarily designed to make things more convenient for the programmer, or which do not conveniently fall into any other category.

Please note that many of these functions are implemented as macros within the M3DL.H header file.

4.2 Utility Function Reference

4.2.1 mdSetRGB

Set the *r, g,* and *b* fields of the **mdCOLOR** structure.

```
void mdSetRGB(color, r, g, b)
```

Function Arguments		
Type	Name	Description
`mdCOLOR *`	`color`	Pointer to mdCOLOR structure containing RGB color value.
`mdUINT32`	`r`	Red component of color
`mdUINT32`	`g`	Green component of color
`mdUINT32`	`b`	Blue component of color

4.2.2 mdSetRGBA

Set the *r, g, b* and *a* fields of the **mdCOLOR** structure.

```
void mdSetRGBA(color, r, g, b, a)
```

Function Arguments		
Type	Name	Description
`mdCOLOR *`	`color`	Pointer to mdCOLOR structure containing RGB color value.
`mdUINT32`	`r`	Red component of color
`mdUINT32`	`g`	Green component of color
`mdUINT32`	`b`	Blue component of color
`mdUINT32`	`a`	Alpha component

4.2.3 mdSetAlpha

Set the *a* field of the **mdCOLOR** structure.

```
void mdSetAlpha(color, a)
```

Function Arguments		
Type	Name	Description
`mdCOLOR *`	`color`	Pointer to mdCOLOR structure containing RGB color value.
`mdUINT32`	`a`	Alpha component

4.2.4 mdSetScrVector

Set the fields of a screen coordinate vector.

```
void mdSetScrVector(v, tx, ty, tz)
```

Function Arguments		
Type	Name	Description
mdScrV3 *	v	Pointer to vector structure
md28DOT4	tx	Vector X value
md28DOT4	ty	Vector Y value
md28DOT4	tz	Vector Z value

4.2.5 mdSetScrRECT

Set the fields of a screen coordinate rectangle consisting of a vector, width, and height.

```
void mdSetScrVector(sr, x, y, z, w, h)
```

Function Arguments		
Type	Name	Description
mdScrRECT *	sr	Pointer to mdScrRECT structure
md28DOT4	tx	Vector X value
md28DOT4	ty	Vector Y value
md28DOT4	tz	Vector Z value
mdU12DOT4	w	Width value
mdU12DOT4	h	Height value

5. 3D Graphics Transform Functions

5.1 Introduction

This section discusses the various functions available for performing 3D graphics operations such as coordinate transformation, rotation, perspective projection, etc.

5.2 M3DL Coordinate System

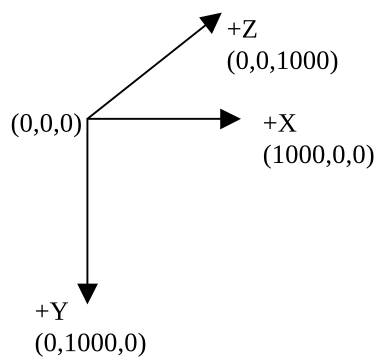
5.2.1 Coordinate Types

3D coordinate values are generally expressed as 16.16 fixed point values. Screen coordinates are expressed as 28.4 or 12.4 fixed point values. Those formats used at the application level are described in the table below. Other formats may be used internally by the library.

Fixed Point Data Type	Description
mdU12DOT4	Used for screen width and height values.
md12DOT20	Used for X, Y, & Z Scale factors
md16DOT16	Used for 3D world coordinates in most cases
md28DOT4	Used for screen coordinates. Truncated by M3DL to mdU12DOT4 before rendering.

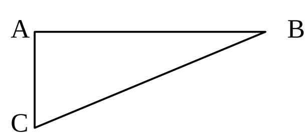
5.2.2 Coordinate System

The coordinate system used by M3DL is oriented as shown below:



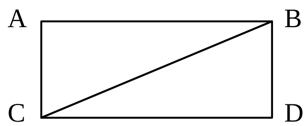
5.2.3 Triangle Vertex Ordering

Vertices for a triangles are expressed in clockwise order:



5.2.4 Quadrangle Vertex Ordering

Quadrangles are expressed with ABC and BCD defining 2 triangles. The triangle 012 is clockwise, triangle 123 is counter-clockwise



5.2.5 Rotation Angles

Rotation angles are specified as 16.16 fixed point values containing the number of rotations. For example, 45 degrees would be calculated as:

```
angle = (45<<16) / 360
```

5.2.6 Clipping

The M3DL library contains a variety of functions for determining if a vertex or group of vertices is within the current view frustum and/or screen display area. They return a bitmapped code indicating the test results.

5.2.6.1 Clipping 3D World Coordinates

The function ***mdRotTransClip***() performs clipping on a single 3D vertex. The functions ***mdRotTransClip3***(), ***mdRotTransClip4***(), and ***mdRotTransClipN***() are similar except that they perform clipping on 3, 4, or more vertices.

These functions work on coordinates in 3D world space. They are rotated and translated into the view frustum, but perspective projection is not performed.

5.2.6.2 Clipping Screen Display Coordinates

The function ***mdClip***() performs clipping on a single screen vertex. The functions ***mdClip3***(), ***mdClip4***(), and ***mdClipN***() are similar except that they perform clipping on 3, 4, or more screen display vertices.

These functions work on screen display coordinates. Any 3D transformations or perspective projection must already be done.

5.2.6.3 Results From Clipping A Single Vertex

The area to be tested is known as the *clipping region*. This may be the 3D view frustum or the actual screen display, depending on which clipping function is being used.

If the specified coordinate is inside the clipping region, the return code is zero.

If the coordinate is outside the clipping region, the return code is a bitmapped group of flags as described in the table below.

Bit #	Description	Values
0	NearZ plane clipping	0: coordinate has Z value >= Current NearZ 1: coordinate has Z value < Current NearZ
1	FarZ plane clipping	0: coordinate has Z value <= Current FarZ 1: coordinate has Z value > Current FarZ
2	Bottom Y plane	0: coordinate has a Y value which is on positive side of Y-Bottom 1: coordinate has a Y value which is on negative side of Y-Bottom
3	Top Y plane	0: coordinate has a Y value which is on positive side of Y-Top 1: coordinate has a Y value which is on negative side of Y-Top

Bit #	Description	Values
4	Right X plane	0: coordinate has an X value which is on positive side of X-Right 1: coordinate has an X value which is on negative side of X-Right
5	Left X plane	0: coordinate has an X value which is positive side of X-Left 1: coordinate has an X value which is negative side of X-Left

5.2.6.4 Results From Clipping Multiple Vertices

The area to be tested is known as the *clipping region*. This may be the 3D view frustum or the actual screen display, depending on which clipping function is being used.

If the tested vertices are all located inside the clipping region, the return code is zero.

If any of the tested vertices are outside the clipping region, then the return code will be non-zero. For those functions that test three or four vertices for a polygon, you can determine which portion is visible and which portion is not by examining the return code. It is a bitmapped group of flags as described in the table below.

Bits 0-5 contain a logical AND of all of the results for each vertex. If any of these bits are set, then the polygon described is completely outside the view frustum. If these bits are all clear, then the polygon is at least partially visible.

Bits 6-11 contain a logical OR of all of the results for each vertex. If these bits are all zero, the polygon is completely visible. Otherwise, the polygon is partially visible.

Bit #	Description	Values
0	NearZ plane clipping Logical AND of all vertices.	0: None of the 3D coordinates has Z value >= current NearZ 1: All of the 3D coordinates has Z value < current NearZ
1	FarZ plane clipping Logical AND of all vertices.	0: None of the 3D coordinates has Z value <= current NearZ 1: All of the 3D coordinates has Z value > current NearZ
2	Bottom Y plane Logical AND of all vertices.	0: None of the 3D coordinates has a Y value which is on positive side of Y-Bottom 1: All of the 3D coordinates has a Y value which is on negative side of Y-Bottom
3	Top Y plane Logical AND of all vertices.	0: None of the 3D coordinates has a Y value which is on positive side of Y-Top 1: All of the 3D coordinates has a Y value which is on negative side of Y-Top
4	Right X plane Logical AND of all vertices.	0: None of the 3D coordinates has a Y value which is on positive side of X-Right 1: All of the 3D coordinates has a Y value which is on negative side of X-Right
5	Left X plane Logical AND of all vertices.	0: None of the 3D coordinates has a Y value which is on positive side of X-Left 1: All of the 3D coordinates has a Y value which is on negative side of X-Left
6	NearZ plane clipping Logical OR of all vertices.	0: None of the 3D coordinates has Z value >= current NearZ 1: At least one of the 3D coordinates has Z value < current NearZ
7	FarZ plane clipping Logical OR of all vertices.	0: None of the 3D coordinates has Z value <= current NearZ 1: At least one of the 3D coordinates has Z value > current NearZ
8	Bottom Y plane Logical OR of all vertices.	0: None of the 3D coordinates has a Y value which is on positive side of Y-Bottom 1: At least one of the 3D coordinates has a Y value which is on negative side of Y-Bottom
9	Top Y plane Logical OR of all vertices.	0: None of the 3D coordinates has a Y value which is on positive side of Y-Top 1: At least one of the 3D coordinates has a Y value which is on negative side of Y-Top
10	Right X plane Logical OR of all vertices.	0: None of the 3D coordinates has a Y value which is on positive side of X-Right 1: At least one of the 3D coordinates has a Y value which is on negative side of X-Right
11	Left X plane Logical OR of all vertices.	0: None of the 3D coordinates has a Y value which is on positive side of X-Left 1: At least one of the 3D coordinates has a Y value which is on negative side of X-Left

5.2.7 Backface Culling

The functions for backface culling operate on 2D screen coordinates after perspective projection has been performed. There are separate calls for culling triangles and quadrangles.

5.2.7.1 Triangle Culling

The functions ***mdCull3***(), ***mdPersCull3***(), and ***mdRotTransPersCull3***() perform backface culling on triangles. The return value indicates the result of the test:

> Bit 0 = 0: Area of triangle, in screen coordinates, is <= 0
> Bit 0 = 1: Area of triangle, in screen coordinates, is >0

The remaining bits are cleared.

5.2.7.2 Quadrangle Culling

The functions ***mdCull4***(), ***mdPersCull4***(), and ***mdRotTransPersCull4***() perform backface culling on quadrangles. The return value indicates the result of the test for each of the two triangles that make up the quadrangle (see section 5.2.4):

> Bit 0 = 0: Area of triangle ABC in screen coordinates is <= 0
> Bit 0 = 1: Area of triangle ABC in screen coordinates is >0
>
> Bit 1 = 0: Area of triangle BCD in screen coordinates is <= 0
> Bit 1 = 1: Area of triangle BCD in screen coordinates is >0

The remaining bits are cleared.

So as far as culling is concerned, quads need not be coplanar. It is possible to verify the 2 triangles separately.

Also note that the MPRs only render triangles. They do not perform backface culling on their own. But they will skip a triangle with ZERO area)

5.2.8 Perspective Transformation

The Perspective transformation is done with the following formulae:

```
SX   = (X * (ScaleX / Z)) + OffsetX
SY   = (Y * (ScaleY / Z)) + OffsetY

SX, SY           : (28.4)
X , Y            : (16.16)
ScaleX, ScaleY   : (12.20)
OffsetX, OffsetY : (28.4)
```

Instead of Setting up ScaleX, ScaleY, OffsetX and OffsetY by hand, it is much easier to use ***mdSetFrustum***() instead.

5.2.9 Depth Cueing

Depth cueing, also known as fog blending, may be performed using the M3DL library functions provided. In order to perform depth-cueing, an application must do the following:

- Set the active blend color, or fog color, using the ***mdActiveBlendColor()*** function.
- Set the near and far Z-depth values for cueing using the ***mdSetFogNearFar()*** function.
- Call one of the ***mdDepthCue()*** functions for one or more vertices. This will specify if any vertex requires blending, and if so it will return a color value or values that is the vertex color blended against the active blend color. The alpha channel information will return the sum of all the vertex colors calculated.
- Set the **mpcDPQ** property of the polygon that will be rendered. See section 6.3.1.1 for more information.

5.3 Coordinate System Functions

5.3.1 mdSetXScale

Set perspective projection XScale value.

```
void mdSetXScale(xs)
```

Function Arguments		
Type	Name	Description
md12DOT20	xs	X Scale value

5.3.2 mdGetXScale

Retrieve perspective projection XScale value.

```
md12DOT20 xs = mdGetXScale(void)
```

Function Return Value		
Type	Name	Description
md12DOT20	xs	X Scale value

5.3.3 mdSetYScale

Set perspective projection YScale value.

```
void mdSetYScale(ys)
```

Function Arguments		
Type	Name	Description
md12DOT20	ys	Y Scale value

5.3.4 mdGetYScale

Retrieve perspective projection YScale value.

```
md12DOT20 ys = mdGetYScale(void)
```

Function Return Value		
Type	Name	Description
md12DOT20	ys	Y Scale value

5.3.5 mdSetXYScale

Set perspective projection Xscale & YScale values.

```
void mdSetXYScale(xs,ys)
```

Function Arguments		
Type	Name	Description
md12DOT20	xs	X Scale value
md12DOT20	ys	Y Scale value

5.3.6 mdSetXOffset

Set perspective projection X offset value.

```
void mdSetXOffset(xo)
```

Function Arguments		
Type	Name	Description
md28DOT4	xo	New X offset value

5.3.7 mdGetXOffset

Set perspective projection X offset value.

```
md28DOT4 xo = mdGetXOffset(void)
```

Function Return Value		
Type	Name	Description
md28DOT4	xo	X offset value

5.3.8 mdSetYOffset

Set perspective projection Y offset value.

```
void mdSetYOffset(yo)
```

Function Arguments		
Type	Name	Description
md28DOT4	yo	New Y offset value

5.3.9 mdGetYOffset

Set perspective projection Y offset value.

```
md28DOT4 yo = mdGetYOffset(void)
```

Function Return Value		
Type	Name	Description
md28DOT4	yo	Y offset value

5.3.10 mdSetXYOffset

Set perspective projection X & Y offset values.

```
void mdSetXYOffset(xo,yo)
```

Function Arguments		
Type	Name	Description
md28DOT4	xo	X Scale value
md28DOT4	yo	Y Scale value

5.4 Transformation Matrix Functions

5.4.1 mdSetTransformMatrix

Set the current transform matrix.

```
void mdSetTransformMatrix(tmat)
```

Function Arguments		
Type	Name	Description
mdMATRIX *	tmat	Pointer to matrix to be used for future transforms.

5.4.2 mdGetTransformMatrix

Retrieve the current transform matrix.

```
void mdSetTransformMatrix(tmat)
```

Function Arguments		
Type	Name	Description
`mdMATRIX *`	`tmat`	Pointer to a buffer that will receive the matrix.

5.4.3 mdPlaceTransformMatrix

Set the coordinate translation fields of the current Transform matrix using separate X, Y, and Z values

```
void mdPlaceTransformMatrix(tx,ty,tz)
```

Function Arguments		
Type	Name	Description
`md16DOT16`	`tx`	Transform X value
`md16DOT16`	`ty`	Transform Y value
`md16DOT16`	`tz`	Transform Z value

5.4.4 mdVecPlaceTransformMatrix

Set the coordinate translation fields of the current Transform matrix using a vector.

```
void mdVecPlaceTransformMatrix(v)
```

Function Arguments		
Type	Name	Description
`mdV3 *`	`v`	Pointer to input vector

5.4.5 mdTransTransformMatrix

Translate the current transform matrix by the specified X, Y, & Z coordinates

```
void mdTransTransformMatrix(tx,ty,tz)
```

Function Arguments		
Type	Name	Description
`md16DOT16`	`tx`	Translate X value
`md16DOT16`	`ty`	Translate Y value
`md16DOT16`	`tz`	Translate Z value

5.4.6 mdVecTransTransformMatrix

Translate the current transform matrix by the coordinates in the specified vector.

```
void mdVecTransTransformMatrix(v)
```

Function Arguments		
Type	Name	Description
`mdV3 *`	`v`	Pointer to input vector

5.4.7 mdMulTransformMatrix

Multiply matrix *m0* by the current transform matrix and store the result back into matrix *m0*.

```
void mdMulTransformMatrix(m0)
```

Function Arguments		
Type	Name	Description
`mdMATRIX *`	`m0`	Pointer to matrix, used as source and destination

5.4.8 mdGetTransformMatrixTrans

Return the translation vector from the current transform matrix.

```
void mdGetTransformMatrixTrans(vout)
```

Function Arguments		
Type	Name	Description
mdV3 *	vout	Pointer to a vector that will receive the result.

5.5 General Matrix Functions

5.5.1 mdPlaceMatrix

Set the vector field of a matrix from the specified X, Y, & Z coordinate values.

```
void mdPlaceMatrix(m, tx, ty, tz)
```

Function Arguments		
Type	Name	Description
mdMATRIX *	m	Pointer to matrix
md16DOT16	tx	Vector X value
md16DOT16	ty	Vector Y value
md16DOT16	tz	Vector Z value

5.5.2 mdVecPlaceMatrix

Set the vector field of a matrix.

```
void mdVecPlaceMatrix(m, v)
```

Function Arguments		
Type	Name	Description
mdMATRIX *	m	Pointer to matrix
mdV3 *	v	Pointer to input vector

5.5.3 mdTransMatrix

Translate the coordinates in the matrix using the specified vector.

```
void mdTransMatrix(m, tx, ty, tz)
```

Function Arguments		
Type	Name	Description
mdMATRIX *	m	Pointer to matrix to be translated
md16DOT16	tx	Vector X value
md16DOT16	ty	Vector Y value
md16DOT16	tz	Vector Z value

5.5.4 mdVecTransMatrix

Translate the coordinates in the matrix using the specified vector.

```
void mdVecTransMatrix(m, v)
```

Function Arguments		
Type	Name	Description
mdMATRIX *	m	Pointer to matrix to be translated
mdV3 *	v	Pointer to input vector

5.5.5 mdGetMatrixTrans

Return the translation vector from the specified matrix.

```
void mdGetMatrixTrans(mat, vout)
```

Function Arguments		
Type	Name	Description
`mdMATRIX *`	`mat`	Pointer to matrix
`mdV3 *`	`vout`	Pointer to a vector that will receive the result.

5.5.6 mdIdentityMatrix

Set the current identity matrix. Resets the matrix coordinates to (0,0,0).

```
void mdIdentityMatrix(im)
```

Function Arguments		
Type	Name	Description
`mdMATRIX *`	`im`	Pointer to new identity matrix

5.5.7 mdTransposeMatrix

Transpose matrix m0 into matrix m1.

```
void mdTransposeMatrix(m0,m1)
```

Function Arguments		
Type	Name	Description
`mdMATRIX *`	`m0`	Pointer to matrix 0
`mdMATRIX *`	`m1`	Pointer to matrix 1

5.5.8 mdSetMatrixStack

Set the start address of the matrix stack.

```
void mdSetMatrixStack(msp)
```

Function Arguments		
Type	Name	Description
`mdBYTE *`	`msp`	Matrix stack pointer

5.5.9 mdPushMatrix

Push the current matrix onto the matrix stack. May only be used AFTER a Matrix Stack Ptr has been set by ***mdSetMatrixStack***().

```
void mdPushMatrix(void)
```

5.5.10 mdPopMatrix

Pop a matrix off the matrix stack and make it the current matrix.

```
void mdPopMatrix(void)
```

5.5.11 mdMulMatrix

Multiply matrix *m0* by matrix *m1*, return the result in matrix *m2*. All arguments may point to the same matrix.

```
void mdMulMatrix(m0,m1,m2)
```

Function Arguments		
Type	Name	Description
mdMATRIX *	m0	Pointer to source matrix #1
mdMATRIX *	m1	Pointer to source matrix #2
mdMATRIX *	m2	Pointer to destination matrix

5.5.12 mdRotMatrixX

Rotate specified matrix around X-axis.

```
void mdRotMatrixX(angle,m0)
```

Function Arguments		
Type	Name	Description
md16DOT16	angle	Desired rotation angle
mdMATRIX *	m0	Pointer to matrix, used as source and destination

Rotation part of matrix	Translation part of matrix
$m0 = \begin{vmatrix} 1, & 0, & 0 \\ 0, & c, & -s \\ 0, & s, & c \end{vmatrix} * m0$	$\begin{vmatrix} Tx \\ cTy - sTz \\ sTy + cTz \end{vmatrix}$
c = cos(angle) s = sin (angle)	

5.5.13 mdRotMatrixY

Rotate specified matrix around Y-axis.

```
void mdRotMatrixY(angle,m0)
```

Function Arguments		
Type	Name	Description
md16DOT16	angle	Desired rotation angle
mdMATRIX *	m0	Pointer to matrix, used as source and destination

Rotation part of matrix	Translation part of matrix
$m0 = \begin{vmatrix} c, & 0, & s \\ 0, & 1, & 0 \\ -s, & 0, & c \end{vmatrix} * m0$	$\begin{vmatrix} cTx + sTz \\ Ty \\ cTz - sTx \end{vmatrix}$
c = cos(angle) s = sin (angle)	

5.5.14 mdRotMatrixZ

Rotate specified matrix around Z-axis.

```
void mdRotMatrixZ(angle,m0)
```

Function Arguments		
Type	Name	Description
md16DOT16	angle	Desired rotation angle
mdMATRIX *	m0	Pointer to matrix, used as source and destination

Rotation part of matrix	Translation part of matrix
`| c,-s, 0 |` `| s, c, 0	* m0` `| 0, 0, 1 |`	`| cTx - sTy` `| sTx + cTy` `| cTz`
c = cos(angle) s = sin (angle)	

5.5.15 mdRotMatrix

See description of ***mdRotMatrixXYZ*** in section 5.5.16 below.

5.5.16 mdRotMatrixXYZ

Rotate the specified matrix around X, Y, & Z using the specified rotation angles.

```
void mdRotMatrixXYZ(angle,m0)
```

Function Arguments		
Type	Name	Description
`mdV3 *`	`angles`	Array containing desired rotation angles
`mdMATRIX *`	`m0`	Pointer to matrix, used as source and destination

```
What it does:
| 1,  0,  0 |     | c1, 0, s1 |     | c2,-s2, 0 |
| 0, c0,-s0 |  *  |  0, 1,  0 |  *  | s2, c2, 0 |
| 0, s0, c0 |     |-s1, 0, c1 |     |  0,  0, 1 |
```

c0 = cos(angle.x) c1 = cos(angle.y) c2 = cos(angle.z)	s0 = sin (angle.x) s1 = sin (angle.y) s2 = sin (angle.z)

5.5.17 mdRotMatrixYXZ

Rotate the specified matrix around Y, X, & Z using the specified rotation angles.

```
void mdRotMatrixYXZ(angle,m0)
```

Function Arguments		
Type	Name	Description
`mdV3 *`	`angles`	Array containing desired rotation angles
`mdMATRIX *`	`m0`	Pointer to matrix, used as source and destination

```
What it does:
| c0, 0, s0 |     | 1,  0,  0 |     | c2,-s2, 0 |
|  0, 1,  0 |  *  | 0, c1,-s1 |  *  | s2, c2, 0 |
|-s0, 0, c0 |     | 0, s1, c1 |     |  0,  0, 1 |
```

c0 = cos(angle.x) c1 = cos(angle.y) c2 = cos(angle.z)	s0 = sin (angle.x) s1 = sin (angle.y) s2 = sin (angle.z)

5.5.18 mdRotMatrixZYX

Rotate the specified matrix around Z, Y, & X using the specified rotation angles.

```
void mdRotMatrixZYX(angle,m0)
```

Function Arguments		
Type	Name	Description
mdV3 *	angles	Array containing desired rotation angles
mdMATRIX *	m0	Pointer to matrix, used as source and destination

What it does:

$$\begin{vmatrix} c0, & -s0, & 0 \\ s0, & c0, & 0 \\ 0, & 0, & 1 \end{vmatrix} * \begin{vmatrix} c1, & 0, & s1 \\ 0, & 1, & 0 \\ -s1, & 0, & c1 \end{vmatrix} * \begin{vmatrix} 1, & 0, & 0 \\ 0, & c2, & -s2 \\ 0, & s2, & c2 \end{vmatrix}$$

c0 = cos(angle.x) c1 = cos(angle.y) c2 = cos(angle.z)	s0 = sin (angle.x) s1 = sin (angle.y) s2 = sin (angle.z)

5.6 Frustrum Setting Functions

5.6.1 mdSetNearZ

Set the Near Z value.

```
void mdSetNearZ(nz)
```

Function Arguments		
Type	Name	Description
mdU16DOT16	nz	Near Z value to set

5.6.2 mdGetNearZ

Return the current near Z value.

```
md16DOT16 near_z = mdGetNearZ(void)
```

Function Return Value		
Type	Name	Description
md16DOT16	near_z	Near Z value

5.6.3 mdSetFarZ

Set the Far Z value.

```
void mdSetFarZ(fz)
```

Function Arguments		
Type	Name	Description
mdU16DOT16	fz	Far Z value to set

5.6.4 mdGetFarZ

Return the current far Z value.

```
md16DOT16 far_z = mdGetFarZ(void)
```

Function Return Value		
Type	Name	Description
md16DOT16	far_z	Far Z value

5.6.5 mdSetNearFarZ

Set the Near and Far Z values at the same time.

```
void mdSetNearFarZ(nz,fz)
```

Function Arguments		
Type	Name	Description
`mdU16DOT16`	`nz`	Near Z value to set
`mdU16DOT16`	`fz`	Far Z value to set

5.6.6 mdSetFrustum

Set the 3D view frustum information.

```
void mdSetFrustum(fov, width, height, aspect, nz, fz )
```

Function Arguments		
Type	Name	Description
`md16DOT16`	`fov`	Field of view
`mdUINT32`	`width`	Width of viewport
`mdUINT32`	`height`	Height of viewport
`md16DOT16`	`aspect`	Aspect ratio of viewport
`mdU16DOT16`	`nz`	Near Z value to set
`mdU16DOT16`	`fz`	Far Z value to set

5.7 Vertex Transformation Functions

5.7.1 mdRot

Rotate a single vertex from source to destination.

```
void mdRot(vin, vout)
```

Function Arguments & Return Code		
Type	Name	Description
`mdV3 *`	`vin`	Pointer to source vertex
`mdV3 *`	`vout`	Vector containing rotation

5.7.2 mdRot3

Rotate 3 sets of vertices from source to destination.

```
void mdRot3(vin, vout)
```

Function Arguments & Return Code		
Type	Name	Description
`mdV3 *`	`vin`	Pointer to array of 3 source vertices
`mdV3 *`	`vout`	Destination for translated vertices

5.7.3 mdRot4

Rotate 4 sets of vertices from source to destination.

```
void mdRot4(vin, vout)
```

Function Arguments & Return Code		
Type	Name	Description
`mdV3 *`	`vin`	Pointer to array of 4 source vertices
`mdV3 *`	`vout`	Destination for translated vertices

5.7.4 mdRotN

Rotate 'N' sets of vertices from source to destination.

```
void mdRotN(vin, vout, num)
```

Function Arguments & Return Code		
Type	Name	Description
`mdV3 *`	`vin`	Pointer to array of *num* source vertices
`mdV3 *`	`vout`	Destination for translated vertices
`mdUINT32`	`num`	Number of sets of vertices to translate

5.7.5 mdRotTrans

Rotate and translate a single vertex from source to destination.

```
void mdRotTrans(vin, vout)
```

Function Arguments & Return Code		
Type	Name	Description
`mdV3 *`	`vin`	Pointer to source vertex
`mdV3 *`	`vout`	Destination for translated vertex

5.7.6 mdRotTrans3

Rotate and translate 3 sets of vertices from source to destination.

```
void mdRotTrans3(vin, vout)
```

Function Arguments & Return Code		
Type	Name	Description
`mdV3 *`	`vin`	Pointer to array of 3 source vertices
`mdV3 *`	`vout`	Destination for translated vertices

5.7.7 mdRotTrans4

Rotate and translate 4 sets of vertices from source to destination.

```
void mdRotTrans4(vin, vout)
```

Function Arguments & Return Code		
Type	Name	Description
`mdV3 *`	`vin`	Pointer to array of 4 source vertices
`mdV3 *`	`vout`	Destination for translated vertices

5.7.8 mdRotTransN

Rotate and translate 'N' sets of vertices from source to destination.

```
void mdRotTransN(vin, vout, num)
```

Function Arguments & Return Code		
Type	Name	Description
`mdV3 *`	`vin`	Pointer to array of *num* source vertices
`mdV3 *`	`vout`	Destination for translated vertices
`mdUINT32`	`num`	Number of sets of vertices to translate

5.7.9 mdPers

Perform perspective projection on 1 set of vertices from source to destination.

```
void mdPers(vin, vsout)
```

Function Arguments & Return Code		
Type	Name	Description
`mdV3 *`	`vin`	Pointer to source vertex
`mdScrV3 *`	`vsout`	Pointer to destination vertex

5.7.10 mdPers3

Perform perspective projection on 3 sets of vertices from source to destination.

```
void mdPers3(vin, vsout)
```

Function Arguments & Return Code		
Type	Name	Description
`mdV3 *`	`vin`	Pointer to array of 3 source vertices
`mdScrV3 *`	`vsout`	Pointer to destination for projected vertices

5.7.11 mdPers4

Perform perspective projection on 4 sets of vertices from source to destination.

```
void mdPers4(vin, vsout)
```

Function Arguments & Return Code		
Type	Name	Description
`mdV3 *`	`vin`	Pointer to array of 4 source vertices
`mdScrV3 *`	`vsout`	Pointer to destination for projected vertices

5.7.12 mdPersN

Perform perspective projection on ‘N’ sets of vertices from source to destination.

```
void mdPersN(vin, vsout, num)
```

Function Arguments & Return Code		
Type	Name	Description
`mdV3 *`	`vin`	Pointer to array of *num* source vertices
`mdScrV3 *`	`vsout`	Pointer to destination for projected vertices
`mdUINT32`	`num`	Number of sets of vertices to translate

5.7.13 mdCull3

Perform backface culling on a triangle polygon.

```
mdUINT32 backface = mdCull3(scrn_xyz)
```

Function Arguments & Return Code		
Type	Name	Description
`mdScrV3 *`	`scrn_xyz`	Pointer to array of 3 vertices
`mdUINT32`	`backface`	Backface culling result. A non-zero result indicates that the polygon is not backfaced. See section 5.2.7.1 for more information.

5.7.14 mdCull4

Perform backface culling on a quad polygon.

```
mdUINT32 backface = mdCull4(scrn_xyz)
```

Function Arguments & Return Code		
Type	Name	Description
mdScrV3 *	scrn_xyz	Pointer to array of 4 vertices
mdUINT32	backface	Backface culling result. A non-zero result indicates that the polygon is not backfaced. See section 5.2.7.2 for more information.

5.7.15 mdPersCull3

Perform backface culling and perspective transformation on triangle polygons.

```
mdUINT32 backface = mdPersCull3(vxyz, scrn_xyz)
```

Function Arguments & Return Code		
Type	Name	Description
mdV3 *	vxyz	Pointer to array of 3 source vertices
mdScrV3 *	scrn_xyz	Pointer to destination for projected vertices
mdUINT32	backface	Backface culling result. A non-zero result indicates that the polygon is not backfaced. See section 5.2.7.1 for more information.

5.7.16 mdPersCull4

Perform backface culling and perspective transformation on quad polygons.

```
mdUINT32 backface = mdPersCull4(vxyz, scrn_xyz)
```

Function Arguments & Return Code		
Type	Name	Description
mdV3 *	vxyz	Pointer to array of 4 source vertices
mdScrV3 *	scrn_xyz	Pointer to destination for projected vertices
mdUINT32	backface	Backface culling result. A non-zero result indicates that the polygon is not backfaced. See section 5.2.7.2 for more information.

5.7.17 mdRotTransClip

Perform rotation, transformation, and view frustum clipping for a single vertex.

```
mdUINT32 clipcodes = mdRotTransClip(vin, vout)
```

Function Arguments & Return Code		
Type	Name	Description
mdV3 *	vin	Pointer to source vertex
mdV3 *	vout	Pointer to destination for translated vertex
mdUINT32	clipcodes	Clipping result. A value of 0 indicates the vertex is within the view frustum. Non-zero values indicate that the vertex is outside the view frustum. See section 5.2.6.3 for more information.

5.7.18 mdRotTransClip3

Perform rotation, transformation, and view frustum clipping for a group of 3 vertices.

```
mdUINT32 clipcodes = mdRotTransClip3(vin, vout)
```

Function Arguments & Return Code		
Type	Name	Description
mdV3 *	vin	Pointer to array of 3 source vertices
mdV3 *	vout	Destination for translated vertices
mdUINT32	clipcodes	Clipping result. A value of 0 indicates the vertices are completely within the view frustum. Non-zero values indicate that one or more vertices are outside the view frustum. See section 5.2.6.4 for more information.

5.7.19 mdRotTransClip4

Perform rotation, transformation, and view frustum clipping for a group of 4 vertices.

```
mdUINT32 clipcodes = mdRotTransClip4(vin, vout)
```

Function Arguments & Return Code		
Type	Name	Description
mdV3 *	vin	Pointer to array of 4 source vertices
mdV3 *	vout	Destination for translated vertices
mdUINT32	clipcodes	Clipping result. A value of 0 indicates the vertices are completely within the view frustum. Non-zero values indicate that one or more vertices are outside the view frustum. See section 5.2.6.4 for more information.

5.7.20 mdRotTransClipN

Perform rotation, transformation, and view frustum clipping for a group of 'N' vertices.

```
mdUINT32 clipcodes = mdRotTransClipN(vin, vout, num)
```

Function Arguments & Return Code		
Type	Name	Description
mdV3 *	vin	Pointer to array of *num* source vertices
mdV3 *	vout	Destination for translated vertices
mdUINT32	num	Number of vertices
mdUINT32	clipcodes	Clipping result. A value of 0 indicates the vertices are completely within the view frustum. Non-zero values indicate that one or more vertices are outside the view frustum. See section 5.2.6.4 for more information.

5.7.21 mdRotTransPers

Perform rotation and perspective projection for a single vertex.

```
void mdRotTransPers(vin, vsout)
```

Function Arguments & Return Code		
Type	Name	Description
mdV3 *	vin	Pointer to source vertex

Function Arguments & Return Code		
Type	Name	Description
mdScrV3 *	vsout	Destination for translated vertex

5.7.22 mdRotTransPers3

Perform rotation and perspective projection for a group of 3 vertices.

```
void mdRotTransPers3(vin, vsout)
```

Function Arguments & Return Code		
Type	Name	Description
mdV3 *	vin	Pointer to array of 3 source vertices
mdScrV3 *	vsout	Destination for translated vertices

5.7.23 mdRotTransPers4

Perform rotation and perspective projection for a group of 4 vertices.

```
void mdRotTransPers4(vin, vsout)
```

Function Arguments & Return Code		
Type	Name	Description
mdV3 *	vin	Pointer to array of 4 source vertices
mdScrV3 *	vsout	Destination for translated vertices

5.7.24 mdRotTransPersN

Perform rotation and perspective projection for a group of ‘N’ vertices.

```
void mdRotTransPersN(vin, vsout, num)
```

Function Arguments & Return Code		
Type	Name	Description
mdV3 *	vin	Pointer to array of *num* source vertices
mdScrV3 *	vsout	Destination for translated vertices
mdUINT32	num	Number of vertices

5.7.25 mdRotTransPersCull3

Perform rotation, perspective projection, and backface culling for a group of 3 vertices.

```
mdUINT32 backface = mdRotTransPersCull3(vxyz, scrn_xyz)
```

Function Arguments & Return Code		
Type	Name	Description
mdV3 *	vxyz	Pointer to array of 3 source vertices
mdScrV3 *	scrn_xyz	Destination for translated vertices
mdUINT32	backface	Backface culling result. A non-zero result indicates that the polygon is not backfaced. See section 5.2.7.1 for more information.

5.7.26 mdRotTransPersCull4

Perform rotation, perspective projection, and backface culling for a group of 4 vertices.

```
mdUINT32 backface = mdRotTransPersCull4(vxyz, scrn_xyz)
```

Function Arguments & Return Code		
Type	Name	Description
mdV3 *	vxyz	Pointer to array of 4 source vertices

Function Arguments & Return Code		
Type	Name	Description
mdScrV3 *	scrn_xyz	Destination for translated vertices
mdUINT32	backface	Backface culling result. A non-zero result indicates that the polygon is not backfaced. See section 5.2.7.2 for more information.

5.7.27 mdClip

Perform a screen clipping test on the specified vertex.

```
mdUINT32 clipflag = mdClip(scrn_xyz)
```

Function Arguments & Return Code		
Type	Name	Description
mdScrV3 *	scrn_xyz	Pointer to a single vertex to be tested.
mdUINT32	clipflag	Clipping test result. A non-zero result indicates that the vertex is outside the clipping region. See section 5.2.6.3 for more information.

5.7.28 mdClip3

Perform a screen clipping test on a group of three vertices.

```
mdUINT32 clipflag = mdClip3(scrn_xyz)
```

Function Arguments & Return Code		
Type	Name	Description
mdScrV3 *	scrn_xyz	Pointer to an array of three vertices to be tested.
mdUINT32	clipflag	Clipping test result. A non-zero result indicates that one or more vertices are outside the clipping region. See section 5.2.6.3 for more information.

5.7.29 mdClip4

Perform a screen clipping test on a group of four vertices.

```
mdUINT32 clipflag = mdClip4(scrn_xyz)
```

Function Arguments & Return Code		
Type	Name	Description
mdScrV3 *	scrn_xyz	Destination for translated vertices
mdUINT32	clipflag	Clipping test result. A non-zero result indicates that one or more vertices are outside the clipping region. See section 5.2.6.3 for more information.

5.7.30 mdClipN

Perform a screen clipping test on a group of three vertices.

```
mdUINT32 clipflag = mdClipN(scrn_xyz, count)
```

Function Arguments & Return Code		
Type	Name	Description
mdScrV3 *	scrn_xyz	Destination for translated vertices
mdUINT32	count	Number of vertices in list

Function Arguments & Return Code		
Type	Name	Description
mdUINT32	clipflag	Clipping test result. A non-zero result indicates that one or more vertices are outside the clipping region. See section 5.2.6.3 for more information.

5.8 Vector Functions

5.8.1 mdSetVector

Set the fields of a vector.

```
void mdSetVector(v, tx, ty, tz)
```

Function Arguments		
Type	Name	Description
mdV3 *	v	Pointer to vector structure
md16DOT16	tx	Vector X value
md16DOT16	ty	Vector Y value
md16DOT16	tz	Vector Z value

5.8.2 mdAddVector

Add the fields of two vectors to create a new vector.

```
void mdAddVector(v1, v2, vout)
```

Function Arguments		
Type	Name	Description
mdV3 *	v1	Pointer to input vector 1
mdV3 *	v2	Pointer to input vector 2
mdV3 *	v1	Pointer to result vector

5.8.3 mdSubVector

Subtract vector 1 from vector 2, resulting in a new vector.

```
void mdSubVector(v1, v2, vout)
```

Function Arguments		
Type	Name	Description
mdV3 *	v1	Pointer to input vector 1
mdV3 *	v2	Pointer to input vector 2
mdV3 *	v1	Pointer to result vector

5.8.4 mdDotProduct

Return the dot product of the specified vectors.

```
mdUINT32 overflow = mdDotProduct(v1, v2, dotprod)
```

Function Arguments & Return Code		
Type	Name	Description
mdV3 *	v1	Pointer to input vector 1
mdV3 *	v2	Pointer to input vector 2
mdU16DOT16	dotprod	Dot product result of two vectors: dotprod = (v1->x * v2->x) >> 16 + (v1->y * v2->y) >> 16 + (v1->z * v2->z) >> 16;

Function Arguments & Return Code		
Type	Name	Description
mdUINT32	overflow	Returns 0 if no overflow occurred, or 1 if overflow occurred. When overflow occurs, *dotprod* is undefined.

5.8.5 mdDotProductSFT

Return the dot product of the specified vectors, after performing the specified shift operation.

```
mdUINT32 overflow = mdDotProductSFT(v1, v2, shift, dotprod)
```

Function Arguments & Return Code		
Type	Name	Description
mdV3 *	v1	Pointer to input vector 1
mdV3 *	v2	Pointer to input vector 2
mdINT32	shift	Shift value to apply.
mdU16DOT16	dotprod	Dot product result of two vectors: dotprod = (v1->x * v1->x) >> shift + (v1->y * v2->y) >> shift + (v1->z * v2->z) >> shift;
mdUINT32	overflow	Returns 0 if no overflow occurred, or 1 if overflow occurred. When overflow occurs, *dotprod* is undefined.

5.8.6 mdCrossProduct

Return the cross product of the specified vectors.

```
void mdCrossProduct(vin1, vin2, cross)
```

Function Arguments & Return Code		
Type	Name	Description
mdV3 *	vin1	Pointer to input vector 1
mdV3 *	vin2	Pointer to input vector 2
mdV3 *	cross	Output vector that will receive the cross product. cross->x = ((vin1->y * vin2->z) - (vin2->y * vin1->z)) >> 16 cross->y = ((vin1->z * vin2->x) - (vin2->z * vin1->x)) >> 16 cross->z = ((vin1->x * vin2->y) - (vin2->x * vin1->y)) >> 16

5.8.7 mdCrossProductSFT

Return the cross product of the specified vectors, after performing the specified shift operation.

```
void mdCrossProductSFT(vin1, vin2, shift, cross)
```

Function Arguments & Return Code		
Type	Name	Description
mdV3 *	vin1	Pointer to input vector 1
mdV3 *	vin2	Pointer to input vector 2

Function Arguments & Return Code		
Type	Name	Description
`mdINT32`	`shift`	Shift value to apply. Normally when the coordinate values in *vin1* and *vin2* are 16.16 fixed point, the proper shift value would be 16.
`mdV3 *`	`cross`	Output vector that will receive the cross product. cross->x = ((vin1->y * vin2->z) - (vin2->y * vin1->z)) >> shift cross->y = ((vin1->z * vin2->x) - (vin2->z * vin1->x)) >> shift cross->z = ((vin1->x * vin2->y) - (vin2->x * vin1->y)) >> shift

5.8.8 mdVectorNormal

Return the normal of the specified vectors.

```
void mdVectorNormal(vin, vout)
```

Function Arguments & Return Code		
Type	Name	Description
`mdV3 *`	`vin`	Pointer to an array of 3 vectors describing a polygon
`mdV3 *`	`vout`	Pointer to destination vector that will receive the normal.

5.8.9 mdVectorNormalSFT

Return the normal of the specified vectors, after performing the specified shift operation.

```
void mdVectorNormalSFT(vin, vout, shift)
```

Function Arguments & Return Code		
Type	Name	Description
`mdV3 *`	`vin`	Pointer to an array of 3 vectors describing a polygon
`mdV3 *`	`vout`	Pointer to destination vector that will receive the normal.
`mdINT32`	`shift`	Shift value to apply. Normally when the coordinate values in *vin1* are 16.16 fixed point, the proper shift value would be 16.

5.8.10 mdVectorMagnitude

Return the magnitude of the specified vector.

```
mdU16DOT16 mag = mdVectorMagnitude(vin)
```

Function Arguments & Return Code		
Type	Name	Description
`mdV3 *`	`vin`	Pointer to a vector

Function Arguments & Return Code		
Type	Name	Description
mdU16DOT16	mag	Magnitude of the vector. This is defined as SQRT(x*x+y*y+z*z). There is some round-off error, but it should not matter as long as the result fits in an mdU16DOT16.

5.8.11 mdApplyMatrix

Apply the specified matrix to the specified vector.

```
void mdApplyMatrix(m, vin, vout)
```

Function Arguments & Return Code		
Type	Name	Description
mdMATRIX *	m	Pointer to matrix
mdV3 *	vin	Pointer to source vector
mdV3 *	vout	Pointer to result vector

5.9 Depth Cue Functions

5.9.1 mdActiveBlendColor

Specifies the color to be used for blending by primitives that use the mpcDPQ attribute. (See section 6.3.1.1).

```
void mdActiveBlendColor(color)
```

Function Arguments		
Type	Name	Description
mdCOLOR *	color	Pointer to mdCOLOR structure containing RGB color value.

5.9.2 mdSetFogColor

Specifies the color to be used for fog blending.

```
void mdSetFogColor(color)
```

Function Arguments		
Type	Name	Description
mdCOLOR *	color	Pointer to mdCOLOR structure containing RGB color value.

5.9.3 mdSetFogNearFar

Set the near and far Z-depth values used for fog blending or depth-cueing.

```
void mdSetFogNearFar(fognear,fogfar)
```

Function Arguments & Return Code		
Type	Name	Description
mdU16DOT16	fognear	Near Z-depth value used for fog blending
mdU16DOT16	fogfar	Far Z-depth value used for fog blending

5.9.4 mdDepthCue

Test a vertex against the current depth cue distances and return a modified color value as indicated..

```
void mdDepthCue(vin,cout)
```

Function Arguments & Return Code		
Type	Name	Description
`mdScrV3 *`	`vin`	Pointer to vertex information
`mdCOLOR *`	`cout`	Pointer to mdCOLOR structure that will receive modified color value.

5.9.5 mdDepthCue3

Test a group of three vertices against the current depth cue distances and return modified color values as indicated..

```
void mdDepthCue3(vin,cout)
```

Function Arguments & Return Code		
Type	Name	Description
`mdScrV3 *`	`vin`	Pointer to array of three vertices
`mdCOLOR *`	`cout`	Pointer to array of mdCOLOR structures that will receive modified color values.

5.9.6 mdDepthCue4

Test a group of four vertices against the current depth cue distances and return modified color values as indicated..

```
void mdDepthCue4(vin,cout)
```

Function Arguments & Return Code		
Type	Name	Description
`mdScrV3 *`	`vin`	Pointer to array of four vertices
`mdCOLOR *`	`cout`	Pointer to array of mdCOLOR structures that will receive modified color values.

5.9.7 mdDepthCueN

Test a group of ‘N’ vertices against the current depth cue distances and return modified color values as indicated..

```
void mdDepthCueN(vin,cout)
```

Function Arguments & Return Code		
Type	Name	Description
`mdScrV3 *`	`vin`	Pointer to array of vertices
`mdCOLOR *`	`cout`	Pointer to array of mdCOLOR structures that will receive modified color values.

5.10 Clipping Functions

5.10.1 mdNearClip3

This function should be called when ***mdRotTransClip***() indicates a triangle that intersects the near clipping plane. It creates a new polygon, either a triangle or quad as needed, that is clipped to the near clipping plane.

```
mdUINT32 numverts = mdNearClip3( ptype, vsrc, csrc, uvsrc, vdst, cdst, uvdst )
```

Function Arguments & Return Code		
Type	Name	Description
`mdUINT32`	`ptype`	Primitive type code
`mdV3 *`	`vsrc`	Pointer to triangle vertices
`mdCOLOR *`	`csrc`	Pointer to array of vertex color information
`mdUINT32 *`	`uvsrc`	Pointer to triangle UV coordinates
`mdV3 *`	`vdst`	Pointer to buffer that will contain new coordinates

Function Arguments & Return Code		
Type	Name	Description
mdCOLOR *	cdst	Pointer to buffer that will contain new vertex color information
mdUINT32 *	uvdst	Pointer to buffer that will contain new UV coordinates
mdUINT32	numverts	Number of vertices in new polygon after clipping. If 3, it's a triangle. If 4, then it's a quad. Other values indicate a degenerate case that should not be drawn.

This page intentionally left blank.

6. Rendering Functions

6.1 Introduction

This function discusses the functions that actually render polygons and other primitives, as well as the various primitive types and rendering attributes each type supports.

Note that the functions in this chapter require an active MPR chain to operate. See chapter 2 for more information.

6.2 Screen Coordinates

Screen X & Y coordinates are expressed in 28.4 format, but are truncated to 12.4 bit before sending them to the MPRs. This means that large, textured polygons may seem to behave weird if viewed really close. Try to avoid large polygons. Always subdivide where appropriate!

6.2.1 2D Clipping

The 2D rendering routines always perform 2D view window clipping with no additional overhead. The MPR always renders only that portion of a polygon that is visible. However, one should not rely on the MPR to perform all of your clipping tasks. If you are processing large numbers of polygons that are completely off screen, you are not operating efficiently. You will achieve greater rendering throughput if you reject those polygons in your own code and never pass them to the renderer. Please see the information on clipping in chapter 5.

6.2.2 UV Coordinates

Texture UV coordinates are expressed in 6.10 format, with ONE (actually 1<<10) meaning 1 times the texture width/height. (UV coordinates do NOT need to be pre-multiplied with the texture width/height).

6.2.3 Z-Depth Values

Z-depth is expressed in 16.16 format, the Z value is normalized to 16-bit and de-normalized at the MPR level to avoid "jumping" textures.

The MPR will actually use 1/Z for Z-Buffer and perspective correct texturing. Perspective correct modes operate with a true divide per pixel.

For best results, try to set NearZ as far away as possible and FarZ as close as possible.

6.3 Drawing Primitives

6.3.1 Basic Primitive Types

The tables below describe the attributes of the various drawing primitive types.

Basic Primitive Type	Description
mptTILE	A tile is a very simple primitive that does not support texture mapping and related attributes, shading, blending, etc. This is essentially a 2D object that is normally used to set the entire screen or some rectangular portion to a specific color and/or Z-depth value.
mptSPRT	A sprite primitive is a 2D primitive specified using a position and size. This is essentially a 2D rectangular object that is always drawn parallel to the view plane. It supports Z-depth values so that it may be used together with 3D objects.

Basic Primitive Type	Description
mptIMG	Essentially the same thing as the **mptSPRT** type, except that it is can be used for large images that do not fit in the usual texture buffer.
mptTRI	A three-sided, three-vertex polygon. It is a 3D object normally used to render 3D graphics display.
mptQUAD	A four-sided, four-vertex polygon. It is a 3D object normally used to render 3D graphics display.

6.3.1.1 Primitive Attributes

Each primitive type supports certain attributes that may be specified by adding the attribute flag to the base type. The attribute flags are defined in the M3DL.H header file. Please note that not all attributes are supported with all primitive types, and some attributes are required.

Primitive Attributes	Description
mpcPC	Perspective Corrected Texture Mapping. Perspective correct modes operate with a true divide per pixel. This attribute should not be used with **mptTILE** primitives.
mpcBIL	Bilinear Filtering. This attribute should not be used with **mptTILE** primitives.
mpcTEX	Texture Mapping. For **mptTRI** and **mptQUAD** primitives, this flag indicates that the object uses a texture. For **mptSPRT** and **mptIMG** primitives, this flag is required. For **mptTILE** primitives, this attribute should not set.
mpcZBUF	Uses Z-buffer. When the Z buffer is part of your frame buffer, this attribute must be set for all variations of the **mptSPRT**, **mptIMG**, **mptTRI**, and **mptQUAD** types.
mpcRGB	Shading flag. Shade the primitive using the built-in color information. The **mptTILE**, **mptSPRT** and **mptIMG** primitives specify a single color for the entire primitive. Therefore, this flag indicates that the object is flat shaded, not a bitmap. For **mpTRI** and **mpQUAD** primitives, there is a color value for each vertex, and gouraud shading is use to blend between them.
mpcALP	Uses alpha-channel to specify transparency. This attribute should be set only if the **mpcDPQ** attribute is not set. See the section titled "**Transparency**" below for more information.
mpcDPQ	Uses alpha-channel information to blend against the active blending color set via the **mdActiveBlendColor**() function. This attribute should be set only if the **mpcALP** attribute is not set.
mpcCLU	Reserved for future use
mpcCLV	Reserved for future use

6.3.1.2 mptTILE Primitives

The attributes supported by the **mptTILE** primitive type are shown in the table below, along with the corresponding extended primitive type definition.

Note that none of the **mptTILE** primitive types may have the **mpcTEX** attribute set.

Primitive Type	Flat Shaded?	Description
mpTILE_F	Yes	Sets the pixel color.

Primitive Type	Flat Shaded?	Description
mpTILE_FZ	Yes	Sets the pixel color and Z-Buffer value.
mpTILE_Z	No	Sets only the Z-buffer value

6.3.1.3 mptSPRT Primitive Types

The attributes supported by the **mptSPRT** primitive type are shown in the table below, along with the corresponding extended primitive type definition.

Note that all **mptSPRT** primitive types must have the **mpcTEX** and **mpcZBUF** attributes set.

Primitive Type	mpcRGB	mpcALP	mpcBIL	mpcDPQ
mpSPRT	No	No	No	No
mpSPRT_F	Yes	No	No	No
mpSPRT_A	No	Yes	No	No
mpSPRT_FA	Yes	Yes	No	No
mpSPRT_D	No	No	No	Yes
mpSPRT_FD	Yes	No	No	Yes
mpSPRT_B	No	No	Yes	No
mpSPRT_BF	Yes	No	Yes	No
mpSPRT_BA	No	Yes	Yes	No
mpSPRT_BFA	Yes	Yes	Yes	No
mpSPRT_BD	No	No	Yes	Yes
mpSPRT_BFD	Yes	No	Yes	Yes

6.3.1.4 mptIMG Primitive Types

The attributes supported by the **mptIMG** primitive type are shown in the table below, along with the corresponding extended primitive type definition.

Note that all **mptIMG** primitive types must have the **mpcTEX** and **mpcZBUF** attributes set.

Primitive Type	mpcRGB	mpcALP	mpcBIL	mpcDPQ
mpIMG	No	No	No	No
mpIMG_F	Yes	No	No	No
mpIMG_A	No	Yes	No	No
mpIMG_FA	Yes	Yes	No	No
mpIMG_D	No	No	No	Yes
mpIMG_FD	Yes	No	No	Yes
mpIMG_B	No	No	Yes	No
mpIMG_BF	Yes	No	Yes	No
mpIMG_BA	No	Yes	Yes	No
mpIMG_BFA	Yes	Yes	Yes	No
mpIMG_BD	No	No	Yes	Yes
mpIMG_BFD	Yes	No	Yes	Yes

6.3.1.5 mpTRI Primitive Types

The table below shows the extended primitive type definition that corresponds to each valid combination of attributes. Note that all **mptTRI** types must have the **mpcZBUF** flag, so that attribute is not included in the table.

Primitive Type	mpcRGB	mpcALP	mpcBIL	mpcDPQ	mpcTEX	mpcPC
mpTRI_G	Yes	No	No	No	No	No
mpTRI_GA	Yes	Yes	No	No	No	No
mpTRI_GD	Yes	No	No	Yes	No	No
mpTRI_T	No	No	No	No	Yes	No
mpTRI_TG	Yes	No	No	No	Yes	No
mpTRI_TA	No	Yes	No	No	Yes	No
mpTRI_TGA	Yes	Yes	No	No	Yes	No

Primitive Type	mpcRGB	mpcALP	mpcBIL	mpcDPQ	mpcTEX	mpcPC
mpTRI_TD	No	No	No	Yes	Yes	No
mpTRI_TGD	Yes	No	No	Yes	Yes	No
mpTRI_BT	No	No	Yes	No	Yes	No
mpTRI_BTG	Yes	No	Yes	No	Yes	No
mpTRI_BTA	No	Yes	Yes	No	Yes	No
mpTRI_BTGA	Yes	Yes	Yes	No	Yes	No
mpTRI_BTD	No	No	Yes	Yes	Yes	No
mpTRI_BTGD	Yes	No	Yes	Yes	Yes	No
mpTRI_PCT	No	No	No	No	Yes	Yes
mpTRI_PCTG	Yes	No	No	No	Yes	Yes
mpTRI_PCTA	No	Yes	No	No	Yes	Yes
mpTRI_PCTGA	Yes	Yes	No	No	Yes	Yes
mpTRI_PCTD	No	No	No	Yes	Yes	Yes
mpTRI_PCTGD	Yes	No	No	Yes	Yes	Yes
mpTRI_PCBT	No	No	Yes	No	Yes	Yes
mpTRI_PCBTG	Yes	No	Yes	No	Yes	Yes
mpTRI_PCBTA	No	Yes	Yes	No	Yes	Yes
mpTRI_PCBTGA	Yes	Yes	Yes	No	Yes	Yes
mpTRI_PCBTD	No	No	Yes	Yes	Yes	Yes
mpTRI_PCBTGD	Yes	No	Yes	Yes	Yes	Yes

6.3.1.6 mptQUAD Primitive Types

The table below shows the extended primitive type definition that corresponds to each valid combination of attributes. Note that all **mptQUAD** types must have the **mpcZBUF** flag, so that attribute is not included in the table.

Primitive Type	mpcRGB	mpcALP	mpcBIL	mpcDPQ	mpcTEX	mpcPC
mpQUAD_G	Yes	No	No	No	No	No
mpQUAD_GA	Yes	Yes	No	No	No	No
mpQUAD_GD	Yes	No	No	Yes	No	No
mpQUAD_T	No	No	No	No	Yes	No
mpQUAD_TG	Yes	No	No	No	Yes	No
mpQUAD_TA	No	Yes	No	No	Yes	No
mpQUAD_TGA	Yes	Yes	No	No	Yes	No
mpQUAD_TD	No	No	No	Yes	Yes	No
mpQUAD_TGD	Yes	No	No	Yes	Yes	No
mpQUAD_BT	No	No	Yes	No	Yes	No
mpQUAD_BTG	Yes	No	Yes	No	Yes	No
mpQUAD_BTA	No	Yes	Yes	No	Yes	No
mpQUAD_BTGA	Yes	Yes	Yes	No	Yes	No
mpQUAD_BTD	No	No	Yes	Yes	Yes	No
mpQUAD_BTGD	Yes	No	Yes	Yes	Yes	No
mpQUAD_PCT	No	No	No	No	Yes	Yes
mpQUAD_PCTG	Yes	No	No	No	Yes	Yes
mpQUAD_PCTA	No	Yes	No	No	Yes	Yes
mpQUAD_PCTGA	Yes	Yes	No	No	Yes	Yes
mpQUAD_PCTD	No	No	No	Yes	Yes	Yes
mpQUAD_PCTGD	Yes	No	No	Yes	Yes	Yes
mpQUAD_PCBT	No	No	Yes	No	Yes	Yes
mpQUAD_PCBTG	Yes	No	Yes	No	Yes	Yes
mpQUAD_PCBTA	No	Yes	Yes	No	Yes	Yes
mpQUAD_PCBTGA	Yes	Yes	Yes	No	Yes	Yes
mpQUAD_PCBTD	No	No	Yes	Yes	Yes	Yes

Primitive Type	mpcRGB	mpcALP	mpcBIL	mpcDPQ	mpcTEX	mpcPC
mpQUAD_PCBTGD	Yes	No	Yes	Yes	Yes	Yes

6.4 Transparency

When the **mpcALP** attribute is set for an appropriate primitive type, it indicates to M3DL that it should blend that primitive against the existing contents of the frame buffer. The alpha channel value of each pixel belonging to the primitive is used to determine the degree of transparency.

When a primitive is rendered, the alpha channel value for each pixel is derived from the texture information (if any) and the vertex color value(s) associated with the primitive.

There are three transparency modes available:

- TRANSMODE_NORMAL
- TRANSMODE_ADDITIVE
- TRANSMOTE_SUBTRACTIVE

The desired mode is set using the ***mdSetTransparencyMode()*** function. Each mode causes blending to work in a different way. These are described in the subsections below.

Additionally, a special background multiplication factor may be selected for use with the additive and subtractive modes. This 2.30 fixed-point value is used as a multiplier for the background pixel values before blending is performed. However, no overflow is checked, so the results can be somewhat unpredictable.

6.4.1 Normal Mode

In normal mode, the alpha channel is interpreted as follows:

If the alpha channel value is…	0	128	255
Then the primitive's pixel is…	Opaque	Half transparent	Fully transparent
Then the resulting pixel is…	Taken from primitive	Result of blending	Unchanged

6.4.2 Additive Mode

In additive mode, the alpha channel is interpreted as follows:

If the alpha channel value is…	0	128	255
And the primitive's pixel is…	Fully transparent	Half transparent	Opaque
Then the resulting pixel is…	Taken from primitive		Unchanged

6.4.3 Subtractive Mode

In subtractive mode, the alpha channel is interpreted as follows:

If the alpha channel value is…	0	128	255
And the primitive's pixel is…	Opaque	Half transparent	Fully transparent
Then the resulting pixel is…	Color of primitive pixel		Unchanged

6.5 Rendering Efficiency

Many programs are designed to create a buffer containing multiple primitives in a list structure and then render them in a batch. This is usually because these programs were originally designed for other platforms where this method provides the best use of the hardware. A pointer to the list is passed to the hardware and the program can perform other processing while the list is being rendered.

However, with NUON and the M3DL library, this is not the most efficient method. While you are creating this buffer, the MPR code running on the other processors is sitting idle.

The method of creating a large buffer of primitives also has substantial memory requirements, since there must typically be enough space to describe your entire display.

When using M3DL, the greatest possible rendering throughput will be achieved by calling the appropriate drawing function as soon as each primitive is created. This will keep the MPR code running on the other processors busy and negate the need for large primitive buffers.

The most likely circumstance where a batch rendering method might be desired would be when you want to use the other processors in the system for other tasks in addition to rendering. In this case, you would build a small batch of primitives, shut down the other tasks, load the MPR code into those processors, perform your rendering, and then return to the other tasks. The main overhead involved is paging your code overlays through the other processors.

6.6 Primitive Drawing Functions

6.6.1 mdDrawPoly

Send an mdTRI or mdQUAD polygon primitive to the rendering chain.

```
void mdDrawPoly(ptype, vertices, color, texture, uv)
```

Function Arguments		
Type	Name	Description
`mdUINT32`	`ptype`	Polygon primitive type code based on either **mptTRI** or **mptQUAD**.
`mdScrV3 *`	`vertices`	Pointer to array of either 3 or 4 vertices for corners of polygon
`mdCOLOR *`	`color`	Pointer to an array of either 3 or 4 **mdCOLOR** structures containing the RGB color information for each vertex.
`mdTEXTURE *`	`texture`	Pointer to structure containing texture information
`mdUINT32 *`	`uv`	Pointer to UV coordinates

6.6.2 mdDrawSprite

Send an **mdSPRT** primitive to the rendering chain.

```
void mdDrawSprite(ptype, sr, color, texture, uv)
```

Function Arguments		
Type	**Name**	**Description**
`mdUINT32`	`ptype`	Sprite primitive type code based on **mptSPRT**
`mdScrRECT *`	`sr`	Pointer to **mdScrRECT** structure containing screen coordinates and size information.
`mdCOLOR *`	`color`	Pointer to RGB color
`mdTEXTURE *`	`texture`	Pointer to texture information
`mdUINT32 *`	`uv`	Pointer to UV coordinates

6.6.3 mdDrawImage

Send an **mdIMAGE** primitive to the rendering chain.

```
void mdDrawImage(ptype, sr, color, img, uv)
```

Function Arguments		
Type	**Name**	**Description**
`mdUINT32`	`ptype`	Sprite primitive type code based on **mptSPRT**
`mdScrRECT *`	`sr`	Pointer to **mdScrRECT** structure containing screen coordinates and size information.
`mdCOLOR *`	`color`	Pointer to RGB color
`mdIMAGEDATA *`	`img`	Pointer to image information
`mdUINT32 *`	`uv`	Pointer to UV coordinates

6.6.4 mdDrawTile

Send an mdTILE primitive to the rendering chain.

```
void mdDrawTile(ptype, sr, color)
```

Function Arguments		
Type	**Name**	**Description**
`mdUINT32`	`ptype`	Tile primitive type code
`mdScrRECT *`	`sr`	Pointer to **mdScrRECT** structure containing screen coordinates and size information.
`mdCOLOR *`	`color`	Pointer to RGB color

6.6.5 mdDrawPrim

Send a primitive to the rendering chain.

```
void mdDrawPrim(prim)
```

Function Arguments		
Type	**Name**	**Description**
`mdPRIM *`	`prim`	Pointer to **mdPRIM** structure describing either an **mdQUAD** or **mdTRI** primitive

6.7 Screen Clear Functions

Please see section 3.8, ***Buffer Clear Functions***.

6.8 3D Model Functions

6.8.1 mdRenderObject

Render object data exported from 3D Studio MAX using M3DL export plug-in (or created by other 3D software using a compatible export module).

```
void mdRenderObject(obj, tex)
```

Function Arguments & Return Code		
Type	Name	Description
`mdBYTE *`	`obj`	Pointer to object data (such as that created by the 3D Studio MAX M3DL export plug-in)
`mdTEXTURE *`	`tex`	Pointer to exported texture data as returned by the function ***mdGetTextureFromMBM***().

6.8.2 mdRenderObjectAmbient

Render object data exported from 3D Studio MAX using M3DL export plug-in (or created by other 3D software using a compatible export module).

This differs from ***mdRenderObject()*** in that it uses the color set by ***mdSetAmbientColor()*** to provide a simple mode of ambient lighting.

```
void mdRenderObjectAmbient(obj, tex)
```

Function Arguments & Return Code		
Type	Name	Description
`mdBYTE *`	`obj`	Pointer to object data (such as that created by the 3D Studio MAX M3DL export plug-in)
`mdTEXTURE *`	`tex`	Pointer to exported texture data as returned by the function ***mdGetTextureFromMBM***().

6.9 Rendering Control Functions

6.9.1 mdSetTransparencyMode

Set the method used for blending objects with alpha-channel transparency against whatever is already in the frame buffer.

```
void mdSetTransparencyMode(mode, bgmultiplier )
```

Function Arguments & Return Code		
Type	Name	Description
`mdTRANSMODE`	`mode`	An enumerated type containing values of: TRANSMODE_NORMAL TRANSMODE_ADDITIVE TRANSMODE_SUBTRACTIVE See the table in section xxx for more information.

Function Arguments & Return Code		
Type	Name	Description
md2DOT30	bgmultiplier	A 2.30 fixed point value indicating a multiplication factor that will be applied to the blending calculation.

This page intentionally left blank.

7. Billboards

Billboards are special variations on some of the basic primitives. They are pseudo 3D entities that always face the camera after transformation. The plane of the object is parallel to the view plane, and the Z value is constant over the entire polygon.

Using a billboard is simple. You fill out the fields of the structure corresponding to the desired variety of billboard, and then call the appropriate billboard function. That function will rotate, transform, and perspective project the billboard from the 3D world space into display screen coordinates. You may optionally perform depth cueing calculations and/or clipping operations.

The function will return the necessary information to construct an **mdSPRITE**, **mdTRI**, or **mdQUAD** primitive which may then be passed along to the appropriate rendering function.

7.1 Billboard Types

There are several different types of billboard: **SBOARD, TBOARD,** and **QBOARD**.   Each type is described below.

7.1.1 SBOARD

The SBOARD billboard is defined by the **mdSBOARD** structure, which specifies parameters such as the position in 3D space and the object size. Figure 7-1 demonstrates the parameters of an SBOARD billboard type.

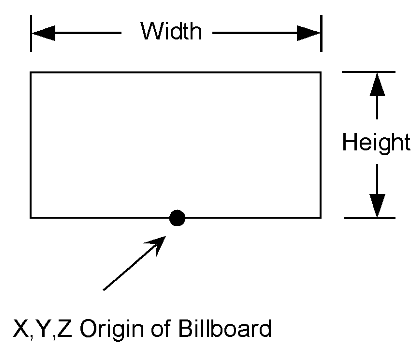


Figure 7-1, SBOARD Billboard Type

There are four functions for converting an SBOARD into an **mdSPRITE** primitive, with different options for depth cue and clipping calculations.

7.1.2 TBOARD

This type of billboard devolves into multiple **mdTRIANGLE** primitives. It is similar to the **SBOARD** billboard, except that the transformation will rotate the polygon around the camera’s Z-axis.

The TBOARD billboard uses the **mdTBOARD** structure to define parameters such as the position in 3D space, rotation matrix, and the size of the object.

7.1.3 QBOARD

This type of billboard devolves into the **mdQUAD** primitive. It is similar to the **SBOARD** billboard, except that the transformation will rotate the polygon around the camera’s Z-axis.

The QBOARD billboard uses the **mdQBOARD** structure to define parameters such as the position in 3D space, rotation matrix, and the size of the object.

7.2 Using Billboards

7.2.1 Perspective Correction & Billboards

Billboards always resolve to 2D polygons with a constant Z value. Therefore, it is not necessary to enable perspective correction, as this is only needed when the Z value changes across the polygon. In fact, perspective correction should always be disabled for billboards to avoid any possible slow down in rendering time.

7.2.2 Billboard Parameters

For all billboard types, the X,Y origin used for all transformations and projections is taken from the middle pixel of the bottom row, as shown in Figure 7-2.

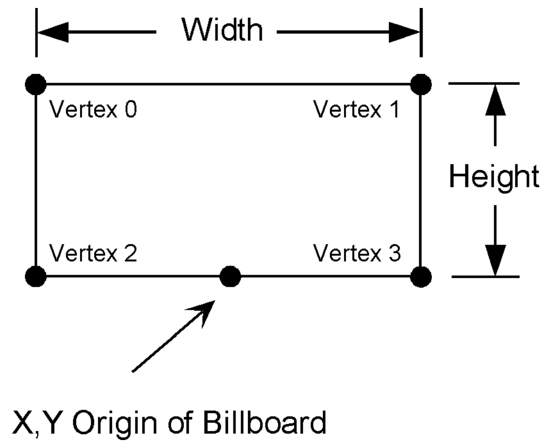


Figure 7-2, Billboard Parameters (SBOARD and QBOARD)

7.3 Billboard Functions

7.3.1 mdRTPSBoard

Calculate an SBOARD billboard. Rotates, translates, and perspective projects the SBOARD and fills out an **mdScrRECT** structure with the finalized screen coordinates of the **mdSPRITE** primitive that should be drawn.

```
mdUINT32 status = mdRTPSBoard( mdSBOARD *board, mdScrRECT *r )
```

Function Arguments & Return Code		
Type	**Name**	**Description**
`mdSBOARD *`	`board`	Pointer to structure containing billboard definition
`mdScrRECT`	`r`	Pointer to **mdScrRECT** structure that will receive information about 2D sprite to draw. This can point to the *sr* field of an **mdSPRITE** structure.
`mdUINT32`	`status`	Returns 0xFFFFFFFF if the resulting Z value is negative, meaning the sprite is not visible and the ***mdDrawSprite()*** function should not be called. Returns 0x00000000 when the Z value is positive, the sprite is visible, and the ***mdDrawSprite()*** function may be called to render it.

7.3.2 mdRTPDpqSBoard

Calculate an SBOARD billboard. Basically the same as the ***mdRTPSBoard()*** function, except that this function also calculates an alpha value to be used for depth cueing.

```
mdUINT32 status = mdRTPDpqSBoard( mdSBOARD *board, mdScrRECT *r, mdCOLOR *rgba )
```

Function Arguments & Return Code		
Type	Name	Description
`mdSBOARD *`	`board`	Pointer to structure containing billboard definition
`mdScrRECT`	`r`	Pointer to **mdScrRECT** structure that will receive information about 2D sprite to draw. This can point to the *sr* field of an **mdSPRITE** structure.
`mdCOLOR *`	`rgba`	Pointer to an mdCOLOR structure that will receive the calculated alpha value in the *alpha* field. Other fields in the structure are not changed.
`mdUINT32`	`status`	Returns 0xFFFFFFFF if the resulting Z value is negative, meaning the sprite is not visible and the ***mdDrawSprite()*** function should not be called. Returns 0x00000000 when the Z value is positive, the sprite is visible, and the ***mdDrawSprite()*** function may be called to render it. The alpha value is non-zero. Returns 0x00000040 when the Z value is positive and the alpha value is zero (blending against depth cue color not required).

7.3.3 mdRTPClipSBoard

Calculate an SBOARD billboard. Basically the same as the ***mdRTPSBoard()*** function, except that this function also calculates clipping results.

```
mdUINT32 status = mdRTPClipSBoard( mdSBOARD *board, mdScrRECT *r )
```

Function Arguments & Return Code		
Type	Name	Description
`mdSBOARD *`	`board`	Pointer to structure containing billboard definition
`mdScrRECT`	`r`	Pointer to **mdScrRECT** structure that will receive information about 2D sprite to draw. This can point to the *sr* field of an **mdSPRITE** structure.
`mdUINT32`	`status`	Returns 0xFFFFFFFF if the resulting Z value is negative, meaning the sprite is not visible and the mdDrawSprite() function should not be called. Otherwise, bits 0-5 return vertex clipping values. See section 5.2.6.3 for details.

7.3.4 mdRTPDpqClipSBoard

Calculate an SBOARD billboard. Basically the same as the ***mdRTPSBoard()*** function, except that this function also calculates clipping results and alpha value for depth cueing.

```
mdUINT32 status = mdRTPClipSBoard( mdSBOARD *board, mdScrRECT *r, mdCOLOR *rgba )
```

Function Arguments & Return Code		
Type	Name	Description
`mdSBOARD *`	`board`	Pointer to structure containing billboard definition
`mdScrRECT`	`r`	Pointer to **mdScrRECT** structure that will receive information about 2D sprite to draw. This can point to the *sr* field of an **mdSPRITE** structure.
`mdCOLOR *`	`rgba`	Pointer to an **mdCOLOR** structure that will receive the calculated alpha value in the *alpha* field. Other fields in the structure are not changed.
`mdUINT32`	`status`	Returns 0xFFFFFFFF if the resulting Z value is negative, meaning the sprite is not visible and the ***mdDrawSprite()*** function should not be called. Otherwise, bits 0-5 return vertex clipping flag values. See section 5.2.6.3 for details.

7.3.5 mdRTPTBoard

Calculate a TBOARD billboard. Rotates, translates, and perspective projects the TBOARD and fills out an **mdScrV3** structure with the finalized screen coordinates of the **mdTRI** primitive that should be drawn.

```
mdUINT32 status = mdRTPTBoard( mdSBOARD *board, mdScrV3 *vsxyz )
```

Function Arguments & Return Code		
Type	Name	Description
`mdSBOARD *`	`board`	Pointer to structure containing billboard definition.
`mdScrV3`	`vsxyz`	Pointer to **mdScrV3** structure that will receive the vector for the TBOARD position.
`mdUINT32`	`status`	Returns 0xFFFFFFFF if the resulting Z value is negative, meaning the sprite is not visible and the ***mdDrawPoly()*** function should not be called. Returns 0x00000000 when the Z value is positive, the sprite is visible, and the ***mdDrawPoly()*** function may be called to render it.

7.3.6 mdRTPDpqTBoard

Calculate a TBOARD billboard. Basically the same as the ***mdRTPTBoard()*** function, except that this function also calculates an alpha value to be used for depth cueing.

```
mdUINT32 status = mdRTPDpqTBoard( mdSBOARD *board, mdScrV3 *vsxyz, mdCOLOR *rgba )
```

Function Arguments & Return Code		
Type	Name	Description
`mdSBOARD *`	`board`	Pointer to structure containing billboard definition
`mdScrV3 *`	`vsxyz`	Pointer to **mdScrV3** structure that will receive information about 2D sprite to draw.
`mdCOLOR *`	`rgba`	Pointer to an mdCOLOR structure that will receive the calculated alpha value in the *alpha* field. Other fields in the structure are not changed.
`mdUINT32`	`status`	Returns 0xFFFFFFFF if the resulting Z value is negative, meaning the sprite is not visible and the ***mdDrawPoly()*** function should not be called. Returns 0x00000000 when the Z value is positive, the sprite is visible, and the ***mdDrawPoly()*** function may be called to render it. The alpha value is non-zero. Returns 0x00000040 when the Z value is positive and the alpha value is zero (blending against depth cue color not required).

7.3.7 mdRTPClipTBoard

Calculate a TBOARD billboard. Basically the same as the ***mdRTPTBoard()*** function, except that this function also calculates clipping results.

```
mdUINT32 status = mdRTPClipTBoard( mdSBOARD *board, mdScrV3 *vsxyz)
```

Function Arguments & Return Code		
Type	Name	Description
`mdSBOARD *`	`board`	Pointer to structure containing billboard definition
`mdScrV3 *`	`vsxyz`	Pointer to **mdScrV3** structure that will receive information about polygon to draw.
`mdUINT32`	`status`	Returns 0xFFFFFFFF if the resulting Z value is negative, meaning the sprite is not visible and the mdDrawSprite() function should not be called. Otherwise, bits 0-5 return vertex clipping values. See section 5.2.6.3 for details.

7.3.8 mdRTPDpqClipTBoard

Calculate a TBOARD billboard. Basically the same as the ***mdRTPTBoard()*** function, except that this function also calculates clipping results and alpha value for depth cueing.

```
mdUINT32 status = mdRTPDpqClipTBoard( mdSBOARD *board, mdScrV3 *vsxyz, mdCOLOR *rgba)
```

Function Arguments & Return Code		
Type	Name	Description
`mdSBOARD *`	`board`	Pointer to structure containing billboard definition

Function Arguments & Return Code		
Type	Name	Description
`mdScrV3 *`	`vsxyz`	Pointer to **mdScrV3** structure that will receive information about polygon to draw.
`mdCOLOR *`	`rgba`	Pointer to an **mdCOLOR** structure that will receive the calculated alpha value in the *alpha* field. Other fields in the structure are not changed.
`mdUINT32`	`status`	Returns 0xFFFFFFFF if the resulting Z value is negative, meaning the sprite is not visible and the ***mdDrawPoly()*** function should not be called. Otherwise, bits 0-5 return vertex clipping flag values. See section 5.2.6.3 for details.

7.3.9 mdRTPQBoard

Calculate a QBOARD billboard. Rotates, translates, and perspective projects the QBOARD and fills out a pair of **mdScrV3** structures with the finalized screen coordinates of the **mdQUAD** primitive that should be drawn.

```
mdUINT32 status = mdRTPQBoard( mdSBOARD *board, mdScrV3 *vsxyz )
```

Function Arguments & Return Code		
Type	Name	Description
`mdSBOARD *`	`board`	Pointer to structure containing billboard definition.
`mdScrV3`	`vsxyz`	Pointer to two **mdScrV3** structures that will receive the vectors for the QBOARD position.
`mdUINT32`	`status`	Returns 0xFFFFFFFF if the resulting Z value is negative, meaning the sprite is not visible and the ***mdDrawPoly()*** function should not be called. Returns 0x00000000 when the Z value is positive, the sprite is visible, and the ***mdDrawPoly()*** function may be called to render it.

7.3.10 mdRTPDpqQBoard

Calculate a QBOARD billboard. Basically the same as the ***mdRTPQBoard()*** function, except that this function also calculates an alpha value to be used for depth cueing.

```
mdUINT32 status = mdRTPDpqQBoard( mdQBOARD *board, mdScrV3 *vsxyz, mdCOLOR *rgba )
```

Function Arguments & Return Code		
Type	Name	Description
`mdSBOARD *`	`board`	Pointer to structure containing billboard definition
`mdScrV3`	`vsxyz`	Pointer to a pair of **mdScrV3** structures that will receive information about 2D sprite to draw.

Function Arguments & Return Code		
Type	Name	Description
mdUINT32	status	Returns 0xFFFFFFFF if the resulting Z value is negative, meaning the sprite is not visible and the ***mdDrawPoly()*** function should not be called. Returns 0x00000000 when the Z value is positive, the sprite is visible, and the ***mdDrawPoly()*** function may be called to render it. The alpha value is non-zero. Returns 0x00000040 when the Z value is positive and the alpha value is zero (blending against depth cue color not required).
mdCOLOR *	rgba	Pointer to an **mdCOLOR** structure that will receive the calculated alpha value in the *alpha* field. Other fields in the structure are not changed.

7.3.11 mdRTPClipQBoard

Calculate an QBOARD billboard. Basically the same as the ***mdRTPQBoard()*** function, except that this function also calculates clipping results.

```
mdUINT32 status = mdRTPClipSBoard( mdSBOARD *board, mdScrV3 *vsxyz )
```

Function Arguments & Return Code		
Type	Name	Description
mdSBOARD *	board	Pointer to structure containing billboard definition
mdScrV3 *	vsxyz	Pointer to **mdScrRECT** structure that will receive information about 2D sprite to draw. This can point to the *sr* field of an **mdSPRITE** structure.
mdUINT32	status	Returns 0xFFFFFFFF if the resulting Z value is negative, meaning the sprite is not visible and the ***mdDrawPoly()*** function should not be called. Otherwise, bits 0-5 return vertex clipping flag values. See section 5.2.6.3 for details.

7.3.12 mdRTPDpqClipQBoard

Calculate an QBOARD billboard. Basically the same as the ***mdRTPQBoard()*** function, except that this function also calculates clipping results and alpha value for depth cueing.

```
mdUINT32 status = mdRTPDpqClipQBoard( mdSBOARD *board, mdScrV3 *vsxyz, mdCOLOR *rgba )
```

Function Arguments & Return Code		
Type	Name	Description
mdSBOARD *	board	Pointer to structure containing billboard definition

Function Arguments & Return Code		
Type	Name	Description
mdScrV3	vsxyz	Pointer to **mdScrRECT** structure that will receive information about 2D sprite to draw. This can point to the *sr* field of an **mdSPRITE** structure.
mdCOLOR *	rgba	Pointer to an **mdCOLOR** structure that will receive the calculated alpha value in the *alpha* field. Other fields in the structure are not changed.
mdUINT32	status	Returns 0xFFFFFFFF if the resulting Z value is negative, meaning the sprite is not visible and the ***mdDrawPoly()*** function should not be called. Otherwise, bits 0-5 return vertex clipping flag values. See section 5.2.6.3 for details.

8. Material Functions

8.1 Introduction

This section discusses the functions that are used to retrieve and manipulate bitmapped images that may be used as textures or with other drawing operations.

Note that your image data must always match the current rendering mode. If you are using RGB mode, then your textures must all be in RGB format. If you are using YCrCb mode, then your textures must all be in YCrCb format.

8.2 M3DL Image File Formats

There are two image file formats associated with the M3DL library.

MBM files are used primarily to contain bitmaps that are constrained to the texture format restrictions described later in this chapter. These bitmaps will normally be used with the **mdQUAD, mdTRI,** and **mdSPRT** primitive types.

MBI files are used for images that do not necessarily correspond to those restrictions. These bitmaps will normally be used with the **mdIMAGE** primitive type.

Utilities exist to create MBM and MBI files from other bitmapped graphics files. These programs are discussed in a later chapter.

8.3 Material Format Capabilties & Restrictions

The restrictions listed below apply to versions of M3DL dated from April 12, 2000 and earlier. Later versions of M3DL may have different restrictions.

8.3.1 mdTEXTURE Formats

Materials that are accessed through the mdTEXTURE structure are subject to the following capabilities and restrictions:

- 4-bit CLUT-based with 16-entry palette
- 8-bit CLUT-based with 256-entry palette
- 16-bit direct RGB or YCrCb
- Textures with 32-bit data are not allowed.
- Textures can be no more than 2048 bytes total. This works out to the following sizes for different bit-depths:

Depth	Total Pixels
4-bit	4096 pixels
8-bit	2048 pixels
16-bit	1024 pixels

- Texture widths must be a power of two.
- The maximum height of any texture is 256 pixels. The table below shows the maximum texture height for each allowed texture width, for each possible bit-depth.

Texture Width in Pixels	Maximum Height		
	4-bit	8-bit	16-bit
2	256	256	256
4	256	256	256
8	256	256	128
16	256	128	64

32	128	64	32
64	64	32	16
128	32	16	8

- The texture data should begin on an 8-byte boundary in memory.
- Palette data for the texture should begin on an 8-byte boundary in memory.

8.3.2 mdIMAGEDATA Formats

Materials that are accessed through the **mdIMAGEDATA** structure are subject to the following capabilities and restrictions:

- 4-bit CLUT-based with 16-entry palette
- 8-bit CLUT-based with 256-entry palette
- 16-bit direct RGB or YCrCb
- Images with 32-bit pixel types are not allowed.
- Images can be any size up to 1024x1024 pixels.
- Image widths must be a multiple of 4.
- The image data should begin on an 8-byte boundary in memory.
- Palette data for the image should begin on an 8-byte boundary in memory.

8.4 Material Functions

8.4.1 mdGetMBMInfo

Get information about specified MBM.

```
mdUINT32 ret = mdGetMBMInfo(mbm, numtexs, num_mbms)
```

Function Arguments & Return Code		
Type	Name	Description
`mdBYTE *`	`mbm`	Pointer to MBM Data
`mdUINT32 *`	`numtexs`	Pointer to a value that will receive the number of the textures in the MBM
`mdUINT32 *`	`num_mbms`	Pointer to a value that will receive the number of the bitmaps in the MBM
`mdUINT32`	`ret`	Returns 0 if an error occurred, or 1 if successful.

8.4.2 mdGetMBIInfo

Get information about specified MBI.

```
mdUINT32 ret = mdGetMBIInfo(mbi, numtexs,num_mbi)
```

Function Arguments & Return Code		
Type	Name	Description
`mdBYTE *`	`mbi`	Pointer to MBI Data
`mdUINT32 *`	`numtexs`	Pointer to a value that will receive the number of the textures in the MBI

Function Arguments & Return Code		
Type	Name	Description
`mdUINT32 *`	`num_mbi`	Pointer to a value that will receive the number of the bitmaps in the MBI
`mdUINT32`	`ret`	Returns 0 if an error occurred, or 1 if successful.

8.4.3 mdTextureFromMBM

Creates a texture from an MBM texture file located in memory. Can either use the bitmap data in place or copy it to SDRAM.

Use ***mdGetMBMInfo***() to get information about memory allocation requirements.

The beginning of an MBM file must be aligned to an 8-byte boundary in memory.

```
mdUINT32 ret = mdTextureFromMBM( mbm, dst, texture, bitmap)
```

Function Arguments & Return Code		
Type	Name	Description
`mdBYTE *`	`mbm`	Pointer to MBM Data
`mdBYTE *`	`dst`	Pointer to destination address. If this is NULL, then the bitmap data will not be copied. Otherwise, this should be a valid address in SDRAM.
`mdTEXTURE *`	`texture`	Pointer to texture information
`mdBITMAP *`	`bitmap`	Pointer to bitmap
`mdUINT32`	`ret`	Returns the amount of memory required to copy the texture and CLUTs from the MBM. If the MBM file specifies load addresses for the texture bitmaps and CLUTs, they will be loaded to those addresses instead of to the address specified by the *dst* argument. Function returns 0 in that case, or if an error occurs.

8.4.4 mdImageDataFromMBI

Copy images from an MBI to memory.

Use ***mdGetMBIInfo***() to get information about memory allocation requirements.

The beginning of an MBM file must be aligned to an 8-byte boundary in memory.

```
mdUINT32 ret = mdImageDataFromMBI( mbi, dst, image, bitmap)
```

Function Arguments & Return Code		
Type	Name	Description
`mdBYTE *`	`mbi`	Pointer to MBI Data
`mdBYTE *`	`dst`	Pointer to destination address. If this is NULL, then the bitmap data will not be copied. Otherwise, this should be a valid address in SDRAM.
`mdIMAGEDATA *`	`image`	Pointer to image information
`mdBITMAP *`	`bitmap`	Pointer to bitmap

Function Arguments & Return Code		
Type	Name	Description
`mdUINT32`	`ret`	Returns the amount of memory required to copy the texture and CLUTs from the MBI. If the MBI file specifies load addresses for the texture bitmaps and CLUTs, they will be loaded to those addresses instead of to the address specified by the *dst* argument. Function returns 0 in that case, or if an error occurs.

9. M3DL Data Types

9.1 Introduction

This section describes some of M3DL's common used basic data types and structures, as defined within the M3DL.H and MDTYPES.H header files.

This section only describes the more common types that may be used as function arguments, return values, or which may be part of another structure which meets that criteria. Many additional types are less commonly used. They may be used internally by the library, but aren't generally necessary at the application level. These additional types are defined within the M3DL.H and MDTYPES.H header files.

Note that changes to the library may require changes to the type definitions. Always refer to the header files for the precise definition of a data type.

9.2 Basic Data Types

9.2.1 Integer Types

Integer Type	Description	Minimum Value	Maximum Value
mdUINT8	Unsigned 8-bit	0	256
mdINT8	Signed 8-bit	- 128	127
mdUINT16	Unsigned 16-bit	0	65536
mdINT16	Signed 16-bit	- 32768	32767
mdUINT32	Unsigned 32-bit	0	4294967296
mdINT32	Signed 32-bit	- 2147483648	2147483647

9.2.2 Fixed Point Types

M3DL uses a variety of fixed point types. The ones more commonly used are defined in the table below.

Fixed Point Data Type	Description
mdU12DOT4	Used for screen width and height values.
md12DOT20	Used for X, Y, & Z Scale factors
md16DOT16	Used for 3D world coordinates in most cases
md28DOT4	Used for screen coordinates. Truncated by M3DL to mdU12DOT4 before rendering.

9.3 Structures

This section is broken into smaller subsections that loosely follow the chapter by chapter breakdown of the library function descriptions.

For each structure there is a brief description, followed by the C language definition of the structure.

9.3.1 Frame Buffer & Drawing Context

9.3.1.1 mdDRAWBUF

The **mdDRAWBUF** structure is used by the **mdDRAWCONTEXT** structure described below. It contains the frame buffer address and DMA flags associated with a rendering or display buffer. An application can use this information to combine M3DL rendering with rendering from another library.

The *sdramaddr* field contains the address of the buffer in SDRAM. The *dmaflags* field contains the bit flags required to perform DMA operations using the buffer.

```
typedef struct _mdDRAWBUF
{
	mdUINT32	sdramaddr;	// Address of buffer in SDRAM
	mdUINT32	dmaflags;	// Flags used for DMA operations
} mdDRAWBUF;
```

9.3.1.2 mdDRAWCONTEXT

The **mdDRAWCONTEXT** structure contains all the information M3DL must track regarding the frame buffer and the current library status.

```
typedef struct _mdDRAWCONTEXT
{
	mdUINT16	actbuf;
	mdUINT16	numbuf;

	mdUINT16	dispw;
	mdUINT16	disph;

	mdUINT16	rendx;
	mdUINT16	rendy;

	mdUINT16	rendw;
	mdUINT16	rendh;

	mdUINT16	flags;
	mdUINT16	select;

	mdUINT32	zcmpflags[2];

	mdUINT32	lastfield;

	mdDRAWBUF	buf[3];		// info for up to 3 buffers

} mdDRAWCONTEXT;
```

9.3.2 3D Graphics

9.3.2.1 mdV3

The **mdV3** structure defines a 3D vertex, also known as a vector. The fields define a particular point in the M3DL 3D world.

```
typedef struct _mdV3
{
	md16DOT16	x;
	md16DOT16	y;
	md16DOT16	z;
} mdV3;
```

9.3.2.2 mdScrV3

The **mdScrV3** structure describes a 3D vertex that has been through the process of perspective projection and which now represents a point of the display, rather than a particular point in the M3DL 3D world.

```
typedef struct _mdScrV3
{
	md28DOT4	x;		//28.4 Screen X coordinate
	md28DOT4	y;		//28.4 Screen Y coordinate
	md16DOT16	z;		//Z Value
} mdScrV3;
```

9.3.2.3 mdScrRect

The **mdScrRect** structure is essentially the same as the **mdScrV3** structure, with the addition of a width and height. This structure is used to define the position and size of a two dimensional object that is being drawn into a 3D display, such as the **mdSPRITE** or **mdTILE** drawing primitives.

```
typedef struct _mdScrRECT
{
       md28DOT4       x;              //28.4 Screen X coordinate
       md28DOT4       y;              //28.4 Screen Y coordinate
       md16DOT16      z;              //Z Value
       mdU12DOT4      w;              //Unsigned 12.4 Screen W value
       mdU12DOT4      h;              //Unsigned 12.4 Screen H value
} mdScrRECT;
```

9.3.2.4 mdMATRIX

The **mdMATRIX** structure is used by a variety of the 3D calculations described in chapter 0.

```
typedef struct _mdMATRIX
{
       md4DOT28       m[3][4];        // tx = [0][3]
                                      // ty = [1][3]
                                      // tz = [2][3]
} mdMATRIX;
```

Note that although the matrix is defined as array of **md4DOT28**, the coordinate fields *tx*, *ty*, and *tz* are actually **md16DOT16**:

```
xx xy xz tx
yx yy yz ty
zx zy zz tz
```

9.3.3 Drawing Primitives

9.3.3.1 mdTILE

The **mdTILE** structure defines the position, size, and color of the **mpTILE** family of drawing primitives.

```
typedef struct _mdTILE
{
       mdScrRECT      sr;             //Screen Rectangle
       mdCOLOR        color;
} mdTILE;
```

Structure Member	Description
sr	Screen coordinates of the corners of the screen rectangle where the primitive should be drawn.
color	Color to use for drawing the tile

9.3.3.2 mdSPRITE

The **mdSPRITE** structure defines the position, size, and color of the **mpSPRT** family of drawing primitives.

```
typedef struct _mdSPRITE
{
       mdScrRECT      sr;             //Screen Rectangle
       mdCOLOR        color;
       mdTEXTURE      *tex;

       mdINT16        u0;
       mdINT16        v0;

       mdINT16        uofs;
       mdINT16        vofs;
} mdSPRITE;
```

Structure Member	Description
sr	Screen coordinates of the corners of the screen rectangle where the primitive should be drawn.
color	Color to use for drawing the sprite
tex	Pointer to mdTEXTURE structure defining the texture information for the sprite
u0, v0	UV coordinates defining the bottom right corner of the rectangular portion of the texture that should be used.
uofs, vofs	UV coordinate offsets defining the top left corner of the rectangular portion of the texture that should be useddd.

9.3.3.3 mdTRI

The **mdTRI** structure defines the position, size, and color of the **mpTRI** family of drawing primitives.

```
typedef struct _mdTRI
{
        mdScrV3         v[3];
        mdCOLOR         c[3];

        mdTEXTURE       *tex;

        mdINT16         u0;
        mdINT16         v0;

        mdINT16         u1;
        mdINT16         v1;

        mdINT16         u2;
        mdINT16         v2;
} mdTRI;
```

9.3.3.4 mdQUAD

The **mdQUAD** structure defines the position, size, and color of the **mpQUAD** family of drawing primitives.

```
typedef struct _mdQUAD
{
        mdScrV3         v[4];
        mdCOLOR         c[4];

        mdTEXTURE       *tex;

        mdINT16         u0;
        mdINT16         v0;

        mdINT16         u1;
        mdINT16         v1;

        mdINT16         u2;
        mdINT16         v2;

        mdINT16         u3;
        mdINT16         v3;
} mdQUAD;
```

9.3.3.5 mdCLIPTRI

The **mdCLIPTRI** structure defines the vertex coordinates, colors, and texture UV mapping information common to the **mdQUAD** and **mpTRI** structures. This represents the data created by the ***mdNearClip3()*** function when a polygon intersects the Near-Z plane.

```
typedef struct _mdCLIPTRI
{
        mdV3            v[4];
        mdCOLOR         c[4];

        mdUINT32        uv[4];
} mdCLIPTRI;
```

9.3.3.6 mdPRIM

The **mdPRIM** structure is a superset of the basic drawing structures that is used when building a list of pre-constructed primitives.

```
typedef struct _mdPRIM
{
        mdUINT32        primcode;

        union
        {
                        mdSPRITE        sprt;
                        mdQUAD          poly;
        } prim;
} mdPRIM;
```

9.3.3.7 mdIMAGE

The **mdIMAGE** structure defines the position, size, and color of the **mpIMG** family of drawing primitives.

```
typedef struct _mdIMAGE
{
       mdScrRect      sr;
       mdCOLOR        color;

       mdIMAGEDATA    *img;

       mdINT16        u0;
       mdINT16        v0;

       mdINT16        uofs;
       mdINT16        vofs;

} mdIMAGE;
```

9.3.4 Billboards

9.3.4.1 mdSBOARD

The **mdSBOARD** structure is used to describe a billboard object type, an abstraction of a rectangular 2D image projected into 3D space.

```
typedef struct _mdSBOARD
{
       mdV3           base;
       md16DOT16      w;
       md16DOT16      h;
} mdSBOARD;
```

9.3.4.2 mdTBOARD

The **mdTBOARD** structure is used for a billboard object type, an abstraction of a triangular 2D image projected into 3D space. It can be rotated around the Z axis.

```
typedef struct _mdTBOARD
{
       mdV3           base;
       mdV2           ofs[3];
} mdTBOARD;
```

9.3.4.3 QBOARD

The **mdQBOARD** structure is used for a billboard object type, an abstraction of a rectangular 2D image projected into 3D space. It can be rotated around the Z axis.

```
typedef struct _mdQBOARD
{
       mdV3           base;
       mdV2           ofs[4];
} mdQBOARD;
```

9.3.5 Texture & Bitmap Data

9.3.5.1 mdBITMAP

The **mdBITMAP** structure is used to define the individual bitmaps used for texture information.

The *bitmap* field should contain the address of the bitmap data somewhere in SDRAM. The *clut* field should contain the address of the color look-up table (palette) information associated with the bitmap, provided the bitmap is CLUT-based.

```
typedef struct _mdBITMAP
{
       mdUINT32       bitmap;        // Addr. of bitmap data
```

```
        mdUINT32       clut;          // Addr. of color look-up table,
                                      // if any
} mdBITMAP;
```

- The **mdBITMAP** structure must be aligned to an 8-byte boundary in memory.
- The data pointed to by the *bitmap* and *clut* fields must be aligned to an 8-byte boundary in memory.
- The *bitmap* and *clut* fields should always have bit 29 set to indicate to M3DL that a linear DMA transfer must be used to transfer the data.
- The *clut* field stores the number of colors along with the address of the palette data.

9.3.5.2 mdTEXTURE

The **mdTEXTURE** structure defines the texture information used by the **mdSPRITE**, **mdTRI**, and **mdQUAD** drawing primitives.

The *pixtype* field indicates the pixel type. This is actually a bitfield containing several pieces of information as described in Table 9-1.

Bits	Name	Description
7 – 6	Reserved	Should be set to zero
5	Black Transparency	0: No 1: Yes In 16-bit mode, black (0,0,0) is transparent. In 4-bit & 8-bit modes, pixel value 0 is transparent, regardless of color value in CLUT
4 – 3	Color Mode	0: GRB mode Bits 0-4 = Blue component Bits 5-9 = Red component Bits 10-15 = Green component 1: YCrCb mode Bits 0-4 = Cb component (Chroma) Bits 5-9 = Cr component (Chroma) Bits 10-15 = Y component (Luminance)
2 – 0	Pixel Mode	Pixel format, corresponding to DMA pixel types. 1 : 4-bit pixels 2 : 16-bit pixels 3 : 8-bit pixels

Table 9-1 — mdTEXTURE.pixtype Bitfield Definition

The texture may have multiple bitmaps associated with it for the purposes of mip-mapping. The *miplevels* field indicates the number of bitmaps.

The *width* and *height* fields indicate the size of the first and largest bitmap. Note that these fields contain the actual size divided by 4. If *width* is 180, that indicates an image width of 720 pixels. Each successively smaller bitmap is expected to be half the width and half the height of the previous one.

The *bmnfo* field is a pointer to an array of **mdBITMAP** structures defining the individual bitmaps.

The **mdTEXTURE** structure must be aligned to an 8-byte boundary.

```
typedef struct _mdTEXTURE
{
        mdUINT8        pixtype;       // Pixel type

        mdUINT8        miplevels;     // Number of mip-map levels
                                      // Also number of bitmaps

        mdUINT8        width;         // Width of first bitmap
        mdUINT8        height;        // Height of first bitmap
        mdBITMAP       *bmnfo;        // Pointer to array containing
```

```
                                        // ‘miplevels’ entries
} mdTEXTURE;
```

9.3.5.3 mdIMAGEDATA

The **mdIMAGEDATA** structure is used to define bitmapped image data that can be used with the **mdIMG** drawing primitive.

The *pixtype* field indicates the pixel type and may be any valid NUON pixel DMA type.

The texture may have multiple bitmaps associated with it for the purposes of mip-mapping. The *miplevels* field indicates the number of bitmap used.

The *width* and *height* fields indicate the size of the first and largest bitmap. Note that these fields contain the actual size divided by 4. If *width* is 180, that indicates an image width of 720 pixels. Each successively smaller bitmap is expected to be half the width and half the height of the previous one.

The *bmnfo* field is a pointer to an array of **mdBITMAP** structures defining the individual bitmaps.

The **mdTEXTURE** structure must be aligned to an 8-byte boundary.

```
typedef struct _mdIMAGEDATA
{
       mdUINT8        pixtype;       // Pixel type

       mdUINT8        miplevels;     // Number of mip-map levels
                                     // Also number of bitmaps

       mdUINT8        width;         // Width of first bitmap
       mdUINT8        height;        // Height of first bitmap
       mdBITMAP       *bmnfo;        // Pointer to array containing
                                     // ‘miplevels’ entries
} mdIMAGEDATA;
```

In the current version of M3DL, the **mdIMAGEDATA** and **mdTEXTURE** structures are identical. However, one should not rely on this because it will change in future versions of M3DL.

9.3.5.4 mdCOLOR

The **mdCOLOR** structure defines a color in RGB (red, green, blue), plus an 8-bit alpha channel value. Note that color definitions always use 8 bits per component, even when rendering into 16-bit mode.

```
typedef struct _mdCOLOR
{
       mdUINT8        g;
       mdUINT8        r;
       mdUINT8        b;
       mdUINT8        a;
} mdCOLOR;
```

10. Command Line Tools

10.1 Introduction

This section describes the tools that allow you to create data files for use with the M3DL library.

10.2 Command Line Tools

10.2.1 BMP2MBM

The BMP2MBM program converts bitmapped graphics files from the MS Windows BMP format into the MBM and MBI formats used by the M3DL library. The command line format is:

```
bmp2mbm [options] [source files]
```

The table below shows the command line options:

Option	Description
-YCRCB	Convert bitmap to YCrCb color space
-GRB	Convert bitmap to GRB. This is the default.
-4	Force output to 4-bit color depth (16-colors)
-IMG	Create an MBI image file rather than an MBM texture file. (default is off)
-8	Force output to 8-bit color depth (256-colors)
-16	Force output to 16-bit color depth (65536 colors)
-ADAPT	Adaptive mode. This counts colors used in source image and selects the most appropriate color depth for the output file.
-NQ	No quantization. Leave color depth of output file the same as the source file.
-T[r,g,b]	Set the specified RGB color value to be treated as transparent. NOTE: in 16-bit mode, only 0,0,0 can be made transparent.
-NM	Create no mip-maps (default is –M64)
-M[x]	Create mip maps until the image size reaches *x* pixels total. (Default is –M64) Maximum number of pixels in 16-bit mode = 1024 Maximum number of pixels in 8-bit mode = 2048 Maximum number of pixels in 4-bit mode = 4096
-F	Flip texture image around Y-axis
-CS[r,g,b]	Set background smooth border color to specified RGB color
-CA	Set background smooth border color to average color value
-CM	Set background smooth border color to most commonly used color

10.2.1.1 Command Line Option Combinations

Please note that it only makes sense to specify one option that affects quantization (i.e. –4, -8, -16, -NQ, -ADAPT). If more than one of these options is specified, then only the last one will be used.

With some source image files, certain quantization options make no sense. For example, if you have a 24-bit source file and specify "-NQ" the option will be ignored because you cannot output 24-bit data to an MBM file. In such cases, the output of the program is undefined.

Wild cards may be used to specify the source files. A single output file will be created for each input file.

10.2.1.2 Automatic Color Quantization

Please be aware that the current version of BMP2MBM will automatically quantize an image to a lower pixel depth if it detects that the resulting texture exceeds the texture size limitations of M3DL.

Be careful that you don't accidentally convert your graphics to a lower bit depth than you intended.

If the image is too big for the selected pixel bit depth, BMP2MBM will convert a 16-bit image down to 8-bit, and if that is still too big, then it will go down to 4-bit. If the source image is still too big, then an error message is printed and no output image is created.

This conversion is done regardless of the format selected on the command line.

The output message printed to the screen will always indicate the bit depth of the texture that is created.

10.2.2 MBMINFO

The MBMINFO program displays information about a particular MBM file. There are no command line options. The command line format is:

```
mbminfo [source files]
```

There are no command line options.

10.2.3 MBMPOS

The MBMPOS program allows you to set the SDRAM load address that will be used for the MBM file's CLUT and bitmap image(s). The command line format is:

```
mbmpos [options] [source files]
```

The table below shows the command line options:

Option	Description
-B[x]	Specify that *x* should be used as the bitmap load address
-C[x]	Specify that *x* should be used as the CLUT load address

10.2.4 TFN2MBM

The TFN2MBM program allows you create an MBM file from the order specified in the TFN file created by the M3DL plug-in for Kinetix 3D Studio MAX. The command line format is:

```
tfn2mbm [source files]
```

The MBM files from the TFN file are taken from the current directory. There are no command line options.

10.2.5 MERGEMBM

The MERGEMBM program allows you to merge all of the MBM files in the current directory into a single large MBM file. There are no command line options. The command line format is:

```
mergembm outputfile
```

10.2.6 M3DINFO

The M3DINFO program reads an M3D file containing 3D model information (typically exported by the 3D Studio MAX plug-in) and displays a summary of the file contents. The command line format is:

m3dinfo [-v] *m3dfile*

If the **–v** option is used, then the program will dump information about each individual polygon:

- Vertex coordinates (relative to the object origin)
- Vertex colors
- Texture Mapping UV coordinates
- Texture ID number
- Polygon type (i.e. quad, triangle, bilinear filtered or not, perspective correct or not, etc.)

The summary of the file contents is printed last, and includes:

- Bounding box for the 3D model
- A count of how many polygons are represented, broken down by type.

This page intentionally left blank.

11. 3D Studio MAX Plug-in

The M3DL library includes a plug-in module for Kinetix 3D Studio MAX R2.5. The data contained within the M3D files created using this module may be used with the ***mdRenderObject()*** function to quickly and easily add sophisticated 3D models to your NUON application.

11.1 Installation

The M3DL plug-in is located in the following directory of the VM Labs NUON SDK:

\VMLABS\3D Studio MAX Plugin\M3DL

Simply copy the "MERMAX.DLE" file from this directory to the "Plugins" directory of your 3D Studio MAX installation.

If 3D Studio MAX is running, it will be necessary to quit and restart it before the export module is available.

11.2 Using the Plug-In

While running 3D Studio MAX, select "Export" from the "File" menu. This will display the "Export To" file selector.

One of the choices in the "File Types" pop-up menu should be "Merlin 3D". Select this choice. Next, select the desired directory, enter the desired filename, and select the "OK" button.

11.2.1 Export Options

After the file selector is exited, the export options dialog will appear. It should look similar to the one shown in Figure 11-1.

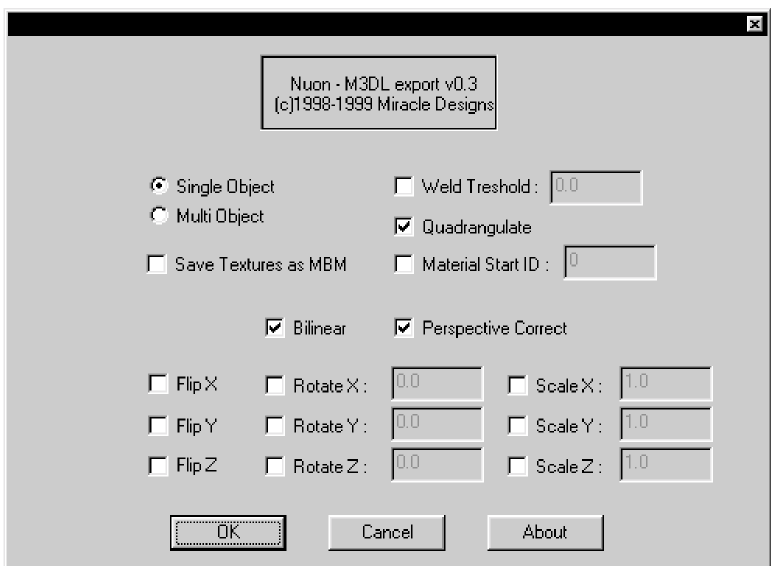


Figure 11-1

11.2.1.1 Single Object –vs– Multi Object

The "Single Object" and "Multi Object" choices define how the data in 3D Studio MAX is exported.

If "Single Object" is selected, then all of the data in 3D Studio MAX is written to a single file that should be treated as a single large 3D object.

If "Multi Object" is selected, then each individual object within 3D Studio MAX is written to a separate M3D file. This allows your application to more easily individually position, scale, and rotate each object within NUON's 3D world.

11.2.1.2 Save Textures As MBM

Selecting this option will cause the export module to create MBM files for each of the bitmap texture files used in materials within the 3D Studio MAX scene.

Note that this will export all texture bitmaps that are used within the scene. Additional textures defined within the material editor that are not assigned to any object within the 3D Studio MAX scene will not be exported.

If this option is selected, then the dialog described in section 11.2.2 below will appear after you select the “OK” button.

11.2.1.3 Weld Threshold

The weld threshold defines the distance used to determine if multiple vertices should be merged together. If multiple vertices are within this distance of each other, they will be merged into a single vertex and the associated polygons adjusted.

This reduces the overall vertex count so that less data is required to describe an object or scene. It also tightens up the boundary between objects that have been placed next to each other visually.

11.2.1.4 Quadrangulate

Selecting this option allows the export module to create quad primitives where possible. Otherwise, the export module will create only triangle primitives.

Quad primitives are more efficient than triangles, since you can describe two adjacent triangles with four vertices instead of six. This results in a small but noticeable advantage in rendering time.

11.2.1.5 Material Start ID

This sets the initial starting value used for material ID codes used within the TFN file. When using multiple TFN files, this allows you to ensure that they do not have overlapping values.

This is not related to the material ID codes used within 3D Studio MAX in any meaningful way.

11.2.1.6 Primitive Attribute Options

The “Bilinear” and “Perspective Correct” choices control the primitive attributes which are assigned to each exported primitive. See section 6.3.1.1 for more information.

11.2.1.7 Coordinate Transformation Options

The options along the bottom of the dialog for “Flip”, “Rotate”, and “Scale” allow you to manipulate the coordinates of the 3D model data.

11.2.2 Export MBM Settings Dialog

If you selected the “Save Textures As MBM” option in the *Export Options* dialog, then selecting the “OK” button will lead you to the *Export MBM – Settings* dialog shown in Figure 11-2.

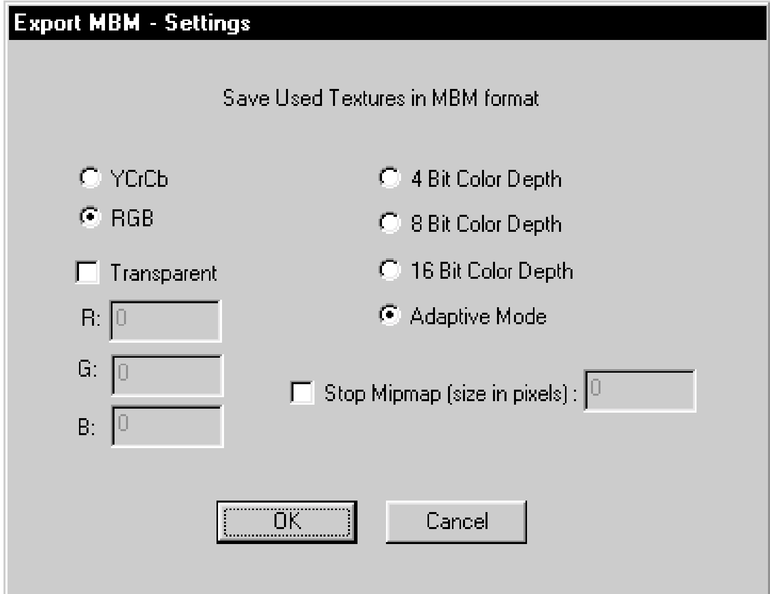


Figure 11-2

11.2.2.1 Color Mode

You may export your texture data in either RGB mode or YCrCb mode. You must select the mode that matches the format of the rendering buffer used by your application.

11.2.2.2 Bit Depth

You may select to have your textures exported as 4-bit, 8-bit, 16-bit, or you may select "Adaptive Mode" and have the export module select the mode which best matches the bitmap specified within 3D Studio MAX.

11.2.2.3 Transparent

If you want to specify a particular RGB color value as being "transparent", you may select the "Transparent" option and then specify the desired RGB color components.

11.2.2.4 Stop MipMap

Selecting this option allows you to specify the size, in pixels, of the smallest mipmap that M3DL should create for the exported textures.

11.2.3 Export Limitations

When you save textures to MBM format, the export module will give a warning each time it encounters a texture that exceeds the M3DL texture size limitations described in section 8.3.

11.2.4 Using The Exported Data

One of the M3DL sample programs demonstrates how to use data exported from 3D Studio MAX using the plug-in. Please look at the sample in the directory:

\VMLABS\SAMPLES\M3DL\3DOBJECT

This page intentionally left blank.

12. MBM File Format

12.1 Introduction

This section discusses the details behind the MBM file format. You may use this information to create your own file conversion utilities, export modules, or other tools.

12.2 MBM Structures

The structures defined in pseudo-code below are used within the MBM file format.

```
struct mbmHEADER
{
    mdUINT8     'M'
    mdUINT8     'B'
    mdUINT8     'M'
    mdUINT8     version ;       //Currently 0x10
}

struct mbmBITMAPNFO
{
    mdUINT32    reserved ;      // Do not use, internally overwritten
    mdUINT32    loadaddress ;   // Zero means automatic
}

struct mbmCLUTNFO
{
    mdUINT32    numcolors;
    mdUINT32    loadaddress ;   //Zero means automatic
}

struct mbmTEXTUREOFFSET
{
    mdUINT32    bitmapoffset;   // Relative to start of file
    mdUINT32    clutoffset;     // ditto, but ZERO for 16-bit texture
}

struct mbmTEXTURE
{
    mdUINT8     pixtype;        //See next section
    mdUINT8     miplevels;      //at least 1
    mdUINT8     width;          //Texture width divided by 4
    mdUINT8     height;         //Texture height divided by 4
}

struct mbmFILEDESC
{
    mdUINT16    numtextures ;
    mdUINT16    numbitmaps;
    mdUINT16    numcluts ;
    mdUINT16    reserved ;      // padding
}
```

12.2.1 mbmTEXTURE.pixtype

The table below defines the bitfields used by the *pixtype* member of the **mbmTEXTURE** structure.

Bits	Name	Description
7 – 6	Reserved	Should be set to zero
5	Black Transparency	0: No 1: Yes In 16-bit mode, black (0,0,0) is transparent. In 4-bit & 8-bit modes, pixel value 0 is transparent, regardless of color value in CLUT

4 – 3	Color Mode	0: GRB mode Bits 0-4 = Blue component Bits 5-9 = Red component Bits 10-15 = Green component 1: YCrCb mode Bits 0-4 = Cb component (Chroma) Bits 5-9 = Cr component (Chroma) Bits 10-15 = Y component (Luminance)
2 – 0	Pixel Mode	Pixel format, corresponding to DMA pixel types. 1 : 4-bit pixels 2 : 16-bit pixels 3 : 8-bit pixels

12.3 MBM File Format

The pseudo-code below defines the structure of an MBM file:

```
mbmHEADER     header
mbmFILEDESC   filedesc

Repeat filedesc.numtextures times:
              mbmTEXTURE         texture
              mbmTEXTUREOFFSET   texture_offsets[texture.miplevels]

Repeat filedesc.numbitmaps times:
              mbmBITMAPNFO       bm_info
              char               bitmapdata[size of bitmap data]

Repeat filedesc.numcluts times:
              mbmCLUTNFO         clut_info
              mdUINT16           palette_data[clut_info.numcolors]
```

All values defined as offsets are relative to the beginning of the overall file.

The table below describes the contents of a particular MBM file that defines two textures.

Sample MBM File Contents		
File offset in bytes	**Contents**	**Details**
0 – 3	mbmHEADER	
4 – 11	mbmFILEDESC	numtextures = 2 numbitmaps = 1 numcluts = 1
12 – 15	mbmTEXTURE #1 (1 of 2)	miplevels = 2 pixtype = 3 (8-bit) width = 16 height = 16
16 – 23	mbmTEXTUREOFFSET #1	bitmapoffset = 44 clutoffset = ????
24 – 31	mbmTEXTUREOFFSET #2	bitmapoffset = 304 clutoffset = ????
32 – 35	mbmTEXTURE #2	miplevels = 1 pixtype = 2 (16-bit) width = 16 height = 16
36 – 39	mbmTEXTUREOFFSET #1	bitmapoffset = 372 clutoffset = 0
40 – 43	mbmBITMAPNFO #1 (1 of 3)	
44 – 299	Bitmap data for bitmap #1 of 3, used for mipmap #1 of texture #1.	16x16 pixels at 8-bits per pixel = 256 bytes
300 – 303	mbmBITMAPNFO #2 (2 of 3)	
304 – 367	Bitmap data for bitmap #2 of 3, used for mipmap #2 of texture #1.	8x8 pixels at 8-bits per pixel = 64 bytes
368 – 371	mbmBITMAPNFO #3 (3 of 3)	

Sample MBM File Contents		
File offset in bytes	**Contents**	**Details**
372 – 883	Bitmap data for bitmap #3 of 3, used for mipmap #1 of texture #2.	16x16 pixels at 16-bits per pixel = 512 bytes
884 – 891	mbmCLUTNFO #1 (1 of 1)	numcolors = 157
892 – 1211	CLUT data for CLUT #1 of 1.	160 entries (157 entries plus padding) = 320 bytes

12.3.1 Restrictions

- The bitmap size in bytes needs to be a multiple of 8 bytes. Always pad the end of the data as needed to reach an 8-byte boundary.
- CLUT entries are always 16-bits.
- The number of entries in a CLUT must be a multiple of 4. (4 x 16-bits = 8 bytes) For example, with an 8 bit texture with 123 colors, 128 (123+5dummy) CLUT values must be stored.

13. M3D File Format

13.1 Introduction

M3D files use a relatively simple format to store 3D model data. It contains vertex coordinates, vertex colors, and texture mapping coordinates. It does not contain vertex or face normals to facilitate lighting calculations.

The M3DL library provides two functions for directly rendering model data in the M3D format. The **mdRenderObject()** and **mdRenderObjectAmbient()** functions take a pointer to M3D data as an argument. These functions provide basic clipping, backface culling, and all of the basic transformations required to render the object on screen. However, while these functions are convenient and perfectly suitable in many cases, there may be situations where an application will need to bypass them.

The information in this section should provide the information necessary to parse an M3D file or to create your own.

13.1.1 M3D File Header

The file header looks like this.

File Offset	Length	Description

0	4	M3D Magic Number. There are two possible values:
		
		#define M3D_VERSION_1 (0x4d334410) // File header is 8 bytes long
		#define M3D_VERSION_2 (0x4d444411) // File header is 32 bytes long

4	4	Number of polygons described in the file

The file header is 8 bytes long for **M3D_VERSION_1**, and 32 bytes long for files that specify **M3D_VERSION_2.**

13.1.1.1 Bounding Box

Files that specify **M3D_VERSION_2** contain extra information in the header to describe a bounding box that specifies the minimum and maximum coordinates used by the 3D model data:

File Offset	Length	Description
8	4	Bounding Box Minimum X axis Coordinate
12	4	Bounding Box Minimum Y axis Coordinate
16	4	Bounding Box Minimum Z axis Coordinate
20	4	Bounding Box Maximum X axis Coordinate
24	4	Bounding Box Maximum Y axis Coordinate
28	4	Bounding Box Maximum Z axis Coordinate

13.1.2 Polygon Data

Immediately following the file header is the information for each polygon, in sequence. Each polygon starts out with a descriptor value that provides the texture ID number and polygon type.

Length	Description
4	Polygon descriptor Bits 0-15 = Texture ID number Bits 16-31 = Polygon Type

```
texture_id = polygoninfo & 0xffff
polygon_type = (polygoninfo >> 16) & 0xffff
```

The texture ID number is an enumerated value into an array of textures that is presumed to exist, typically from an exported MBM file.

The polygon type corresponds to the MPR code required to draw it, as described in section 6.3. The number of vertices is indicated by the polygon type. If the **mptTRI** bit is set, then the polygon is a triangle with three vertices. Otherwise if the **mptQUAD** bit is set, then the polygon is a quad with four vertices.

If neither of these bits is set, then the file may be invalid or corrupted. Note that other primitive types recognized by M3DL (**mptTILE, mptIMG, mptSPRT**) are not stored in M3D files.

Following the polygon descriptor value, we have the following items as indicated by the bits in the ***polygon_type*** value.

* Vertex Coordinates
* Vertex Color Information
* Vertex Texture UV Coordiantes

13.1.2.1 Vertex Coordinates

For each vertex, there are 12 bytes of coordinate data in **md16DOT16** (16.16 fixed point) format:

Offset	Length	Description
0	4	X-Axis Coordinate (relative to object origin)
4	4	Y-Axis Coordinate (relative to object origin)
8	4	Z-Axis Coordinate (relative to object origin)

13.1.2.2 Vertex Color Information

If the **mpcRGB** flag is set in the ***polygon_type*** value, there are 4 bytes of color data for each vertex:

Offset	Length	Description
0	4	mdCOLOR information (Green-Red-Blue-Alpha)

13.1.2.3 Vertex Texture UV Coordinates

If the **mpcTEX** flag is set in the ***polygon_type*** value, there are 4 bytes for the UV coordinates for each vertex in 6.10 fixed point format:

Offset	Length	Description
0	2	Texture U coordinate
2	2	Texture V coordinate

13.1.3 End of File

After the information for the last polygon, we should be at the end of the file.