VM LABS

Merlin 3D Library Functions

Programmers' Manual

Version 0.4

VM Labs, Inc.
520 San Antonio Rd
Mountain View, CA 94040
Tel: (650) 917 8050
Fax: (650) 917 8052

Copyright notice

This is a preliminary specification. VM Labs reserves the right to make changes to any and all of the interfaces described in this document.

Contents

1. Introduction

PLEASE NOTE: The MML3D library is no longer supported, and is provided as reference material only for those wishing to implement their own 3D libraries. Other 3D libraries (such as mGL) are available for NUON and are supported; see the appropriate documentation in the NUON SDK.

1.1 Polygon Rendering Pipeline

The standard 3D rendering pipeline is shown in figure 3. The direct mode C API loads MPEs with appropriate functions to do geometry transformations, clipping, lighting, and rendering. The pipeline as implemented is fully customizable. User components may be used to replace any stage of the pipeline. Input to the pipeline may come from standard API functions that read memory directly (for polygons) or that generate polygons from patch descriptions. Or, the programmer can write a custom front end that creates polygons to be drawn by the rest of the pipeline.

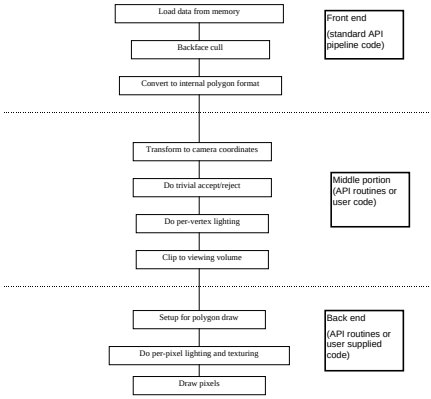

Figure 1: Standard Rendering Pipeline

The functions provided by the APIs are all designed to be used with a Z buffer.

The 3D API does not require that Z buffering be enabled; for example, objects may be pre-sorted and then be rendered in back to front order without Z buffering. However, sorting moving objects in 3D is a difficult problem to solve without Z buffering, and usually results in unpleasant visual artifacts. Moreover, if Z buffering is not used then at most 1 MPE can be used for rendering pixels (unless some screen space subdivision algorithm is used).

1.2 Lighting Model

The 3D API allows a number of distinct types of light sources. Position independent (aka directional) lights are considered to be outside of the scene, and provide parallel rays of light from an infinitely distant light source. Sunlight is the classic example of position independent light. Point light sources are inside of the scene, and the light from them varies according to distance and relative position. Spot lights are like point light sources, but shine only in a specific direction and in a limited cone. Ambient light is the level of constant background illumination. (*NOTE: in the current library release, point sources and spot lights are not yet implemented.*)

The low-level MPE functions that implement the lighting model may be replaced by user supplied lighting functions. This means that any kind of user defined light sources or lighting effects may be provided to enhance the API.

The default lights provided by the API are white lights. Colored lights may be implemented by means of user defined light sources and user defined texture mapping functions.

1.3 Texture Maps

Any Merlin Media Library display pixmap may be used as a texture on a 3D object (see the libmml2d documentation for a discussion of display pixmaps). Most textures will normally be this sort of traditional bitmap textures. The MML supports other textures that are particularly useful for 3D rendering. Mip-map textures, for example, are a series of images (large or small) representing different resolution views of the same texture. This allows for more effective filtering (anti-aliasing) of the texture, since the appropriate level of the mip-map may be chosen for the size that the texture appears on screen. They also reduce bus bandwidth requirements.

These standard texture maps may be supplemented by user designed texture mapping functions. At the lowest level, the MPE polygon draw function calls pixel generating functions to draw strips of pixels based upon texture maps. The pixel generating function may be supplied explicitly by the application, to provide custom texture mapping and shading effects or procedural textures.

1.4 Procedural Textures

Texture maps are one way to add color and detail to polygons. However, they suffer from a number of limitations. For example, unless a very large texture map is

used, the individual pixels that make up the texture image will be visible when the it is viewed up close. Texture maps can also consume a great deal of memory. An alternative way of providing a texture for a polygon is to write a function which specifies the color of the texture at each coordinate. Such a procedural texture can be scaled to any size without pixellation, while consuming very little memory and no bus bandwidth. The Merlin 3D API will provide a number of predefined procedural textures, and also allows user supplied functions to be used as procedural textures. (*NOTE: in the present library release, procedural textures are still somewhat cumbersome to use (requiring a library recompile). This will be corrected in a future release.*)

1.5 Shading

Shading is the task of combining the color information from textures with the light intensity determined by the lighting model. The Merlin 3D API supports 4 kinds of shading: flat shading, Gouraud shading, Gouraud shading with specular highlights, and Phong shading. Flat shading uses only the surface normal of a polygon to determine the shading of pixels. This means that the intensity of light is constant over the whole polygon. In Gouraud shading, the light intensity is calculated separately at each vertex and then interpolated across the polygon. This allows a more realistic appearance for curved surfaces. Gouraud shading with specular highlights adds an extra (specular) term to Gouraud shading to take into account the reflection of the light itself from a surface. This allows for more realistic rendering of shiny surfaces such as plastic or metal. Phong shading is the most realistic (and most computationally expensive) shading model. In Phong shading the vertex normal vectors are interpolated, and the lighting model is evaluated at each pixel as it is drawn. This allows for the very accurate specular highlights, and a smoother appearance to curved surfaces.

(*NOTE: Phong shading and flat shading are not directly supported in the current library release, although it is easy to add either of these with a library recompile.*)

1.6 Antialiasing

There are two kinds of antialiasing: texture filtering, and edge antialiasing. Texture filtering is the process of smoothly scaling texture maps to reduce the artifacts that arise when a texture is scaled up or down. In the former case the artifacts take the form of “pixellation,” *i.e.* the individual pixels that make up the texture map become visible. In the latter case, the artifacts are typically “twinkling” effects; as the object moves, the scaled output points are sampled at different locations in the input texture. Various filtering methods are provided to help overcome these artifacts, including bilinear interpolation and mip-mapping.

Edge antialiasing is the removal of “jaggies” from the edges of polygons. It is a relatively expensive operation, but produces a dramatic improvement in the appearance of rendered scenes.

2. C 3D API

The low level C 3D API layer gives graphics programmers very close access to the underlying hardware and MPE rendering primitives. It is designed for low-overhead, high performance graphics and to provide programmers with a low level API for porting existing applications from other platforms, while at the same time being easier to use than direct programming of MPEs. There are a number of fundamental objects which must be specified in order to render a 3D object. The output routines require the 3D object itself (which is specified by an **m3dBuf**); a transformation matrix (**m3dMatrix**) giving the mapping from object coordinates to world coordinates; a camera (**m3dCamera**) which among other things includes a matrix specifying the mapping from world coordinates to screen coordinates; a lighting model (**m3dLightData**) for illumination; and an output **mmlDisplayPixmap** and rectangle within that pixmap, which together specify where the output will appear.

The separation of camera and output pixmap makes it easy to switch between different cameras when viewing a scene, or to render several different views of the same scene (for example, a rear-view mirror overlaid on the view through the windshield of a car).

2.1 Conventions

The coordinate system used has X increasing from left to right, Y increasing from top to bottom, and Z increases into the screen. The vertices of polygons should be given in clockwise order. Polygon normal vectors point away from the polygon.

2.2 Matrices

A matrix for the 3D library (**m3dMatrix**) is a 4 by 4 grid of 16.16 fixed point values. Points are considered to be 1 by 4 column vectors, again of 16.16 fixed point values, and are transformed by the matrix by multiplying them on the right; that is, if M is a matrix, and P a point, then MP is the transformed point.

2.2.1 m3dIdentityMatrix

void **m3dIdentityMatrix**(**m3dMatrix ***_M_)

Initializes the matrix pointed to by *M* to be the identity matrix.

2.2.2 m3dEulerMatrix

void **m3dEulerMatrix**(**m3dMatrix ***_M_, **m3dreal** *xrot*, **m3dreal** *yrot*, **m3dreal** *zrot*)

Calculates a rotation matrix from Euler angles, that is, angles relative to the X, Y, and Z axes. *M* is the matrix to be initialized. *xrot* is the angle of rotation relative to the X axis, expressed as rotations in 16.16 fixed point format (so 90 degrees is `0x00004000`). Similarly, *yrot* and *zrot* are the angles relative to the Y and Z axes, respectively.

2.2.3 m3dPlaceMatrix

void **m3dPlaceMatrix**(**m3dMatrix ***M*, **m3dreal** *x*, **m3dreal** *y*,**m3dreal** *z*)

Changes the transformation matrix *M* so that the translation component corresponds to a position in space of (x,y,z), where *x*, *y*, and *z* are all 16.16 fixed point numbers.

2.2.4 m3dMatrixMultiply

void **m3dMatrixMultiply**(**m3dMatrix ***dest*, **m3dMatrix ***A*, **m3dMatrix ***B*)

Multiplies the matrix *A* by the matrix *B* and places the result in *dest*. All three pointers must point to different matrices.

2.3 Cameras

A camera (**m3dCamera**) is an abstraction for a viewer. Associated with every camera are a focal length (a small 16.16 fixed point value, typically 1.0), maximum Z value (a 16.16 fixed point number giving the back clipping plane) and a transformation matrix. The matrix is specified as the mapping from camera space to world space, so the same matrix may be used for object rendering and viewing from that object's position in space (this makes attaching a camera to a physical object in the scene trivial). Internally the matrix is actually inverted (so that it gives a mapping from world space to camera space) and then multiplied by each object's matrix to find the object space to camera space mapping.

2.3.1 m3dInitCamera

void **m3dInitCamera**(**m3dCamera ***cam*, **m3dreal** *focalLength*, **m3dreal** *maxZ*)

Initializes a camera. The camera's transformation matrix will initially be the identity matrix (so it will positioned at the origin of world space).

2.3.2 m3dSetCameraMatrix

void **m3dSetCameraMatrix**(**m3dCamera ***cam*, **m3dMatrix ***mat*)

Sets a camera's tramsformation matrix. *mat* is an object space to world space transformation matrix, as set up for example by **m3dEulerMatrix** and **m3dPlaceMatrix**.

2.4 Materials

Materials mapped onto 3D objects may be either solid colors or texture maps. They may also have properties such as shininess or transparency associated with them.

(*NOTE: Material properties are not implemented in the current version of the library.*)

2.4.1 m3dInitMaterialFromColor

void **m3dInitMaterialFromColor**(**m3dMaterial ****mat*, **mmlColor** *color*)

Creates a solid color material from a **mmlColor** object. For example, to create a solid yellow material:

```
m3dMaterial mat;
m3dInitMaterialFromColor(&mat,
        mmlColorFromRGB(200, 200, 0));
```

It is a good idea not to use fully saturated colors, since televisions typically will smear such colors.

2.4.2 m3dInitMaterialFromPixmap

void **m3dInitMaterialFromPixmap**(**m3dMaterial ****mat*, **mmlDisplayPixmap ****pmap*)

Creates a material from an existing display pixmap. For example, the 2D library could be used to create a picture which can then be used in a 3D scene as a texture map.

2.4.3 m3dInitMaterialFromJPEG

void **m3dInitMaterialFromJPEG**(**m3dMaterial ****mat*, **mmlSysResources ****sr*, **void ****jpegStart*, **int** *jpegSize*, **mmlPixFormat** *pix*)

Creates a texture map material from a JPEG image. *sr* is the system resources structure initialized by a call to **mmlPowerUpGraphics** at start up time. *jpegStart* is the starting address of the JPEG image in memory. *jpegSize* is the size of the JPEG image, in bytes. *pix* is the pixel format to use for the data; this should normally be either `e655` for 16 bit pixels or `e888Alpha` for 32 bit pixels. 32 bit pixels look slightly better, but consume twice as much memory.

2.4.4 m3dInitMipMapFromJPEG

void **m3dInitMipMapFromJPEG**(**m3dMaterial ****mat*, **int** *maxLevel*, **mmlSysResources ****sr*, **void ****jpegStart*, **int** *jpegSize*, **mmlPixFormat** *pix*)

Creates a mip-mapped texture map material from a JPEG image. *maxLevel* is the number of mip-map levels to be created. Each level is one-half the size of the previous level, and the first level is the initial size of the image. In the current implementation *maxLevel* may be at most 5, which means that the smallest image will be 1/32 the “normal” size of the JPEG. *sr* is the system resources structure initialized by a call to **mmlPowerUpGraphics** at start up time. *jpegStart* is the starting address of the JPEG image in memory. *jpegSize* is the size of the JPEG image, in bytes. *pix* is the pixel format to use for the data; this should normally be either `e655` for 16 bit pixels or `e888Alpha` for 32 bit pixels. 32 bit pixels look slightly better, but consume twice as much memory.

2.5 Display Buffers

3D objects are built into a display buffer (an object of type **m3dBuf**). This buffer is saved and displayed on screen at a later time. In order to render a display buffer a transformation matrix mapping the buffer into “world space” must be specified. By changing this transformation matrix, the buffer may be viewed at different positions in 3D space. Thus, for example, a display buffer which represents a space ship may be drawn several times, in different positions, in order to represent different space ships in a game.

Display buffers may be statically compiled (for example, converted from a pre-existing polygonal model), or they may be dynamically constructed at run time.

2.5.1 m3dInitBuf

void **m3dInitBuf**(**m3dBuf ***_buf_)

Initializes a display buffer. If the buffer already contains 3D objects such as polygons, all such objects are deleted and the associated memory freed. **m3dInitBuf** must be called before any other display buffer functions are applied to the buffer.

2.5.2 m3dFreeBuf

void **m3dFreeBuf**(**m3dBuf ***_buf_)

Frees memory associated with a display buffer. After this call no more display buffer functions may be called for the buffer *buf* (except that the **m3dInitBuf** function may be called to re-initialize the buffer).

2.5.3 m3dSetMaterial

void **m3dSetMaterial**(**m3dBuf ***_buf_, **m3dMaterial ***_mat_)

Sets the material to use for a buffer. All polygons added to the buffer after this call will use the material *mat*.

2.5.4 m3dStartTriangle

void **m3dStartTriangle**(**m3dBuf ***_buf_)

Starts adding a new triangle to the display buffer *buf*. Between this call and the next **m3dEndTriangle** call there must be exactly three **m3dAddVertex** calls.

2.5.5 m3dEndTriangle

void **m3dEndTriangle**(**m3dBuf ***_buf_)

Finishes adding a triangle to the display buffer *buf*.

2.5.6 *m3dAddNormal*

void **m3dAddNormal**(**m3dBuf ***_buf_, **m3dreal** _x_, **m3dreal** _y_, **m3dreal** _z_)

Adds a normal vector to a polygon. *x*, *y*, and *z* are the components of the vector as 16.16 fixed point numbers, and must be in normal form (with magnitude 1.0). All vertices added to the polygon after this one will use (x, y, z) as their normal vector. Typically each vertex will have its own normal vector, so one would make a sequence of **m3dAddNormal** and **m3dAddVertex** calls. However, for flat shaded polygons it is only necessary to call **m3dAddNormal** once; all of the vertices in the polygon can use the same normal vector.

This function is also available in the variant **m3dAddNormal3f**, which takes three floating point numbers as parameters; these floating point numbers are automatically converted to fixed point.

2.5.7 *m3dAddTextureCoords*

void **m3dAddTextureCoords**(**m3dBuf ***_buf_, **m3dreal** _u_, **m3dreal** _v_)

Adds texture coordinates to a polygon. The next vertex added to the polygon with **m3dAddVertex** will have *u* and *v* as its texture map coordinates. *u* and *v* are both 16.16 fixed point numbers, and should normally be between 0.0 and 1.0, with (0.0,0.0) being the upper left corner of the texture, and (1.0,1.0) being the lower right corner. The texture coordinates are automatically converted to the proper pixel coordinates for the texture map size or mip-map level.

2.5.8 *m3dAddVertex*

void **m3dAddVertex**(**m3dBuf ***_buf_, **m3dreal** _x_, **m3dreal** _y_, **m3dreal** _z_)

Adds a vertex to a polygon. *x*, *y*, and *z* are the coordinates of the vertex as 16.16 fixed point numbers.

This function is also available in the variant **m3dAddVertex3f**, which takes three floating point numbers as parameters; these floating point numbers are automatically converted to fixed point.

2.6 Rendering and General Initialization

2.6.1 *m3dInit*

void **m3dInit**(**mmlSysResources ***_sr_, **int** _nummpes_)

Initializes the 3D library. This call must be made before any other 3D library calls. *sr* is a system resources structure which must have previously been initialized with a call to the **mmlPowerUpGraphics** function in the 2D library. *nummpes* is the number of MPEs to be used for 3D graphics. If this value is 0, then only one MPE is used. If this value is more than the number of free MPEs, then all available MPEs will be used. The `M3D_MPE_USAGE` hint may be used to control which MPEs are chosen; setting this hint to `M3D_MPE_USE_SELF` (the default) will cause the current MPE to

be chosen first; setting it to M3D_MPE_USE_OTHERS will cause the current MPE to be used last.

Example of library initialization:

```
mmlSysResources sysRes;
mmlGC gc;

mmlPowerUpGraphics( &sysRes );
mmlInitGC( &gc, &sysRes );
m3dInit( &sysRes, 4 );
```

2.6.2 m3dExecuteBuffer

void **m3dExecuteBuffer**(**mmlGC ****gc*, **mmlDisplayPixmap ****pixmap*, **m2dRect ****rect*, **m3dBuf ****buf*, **m3dMatrix ****objmat*, **m3dCamera ****cam*, **m3dLightData ****lights*)

Draws a previously constructed display buffer into a subrectangle of a display pixmap. *gc* is a 2D graphics context previously initialized by the **mmlInitGC** function. *pixmap* is a display pixmap (that is, one located in SDRAM). *rect* specifies the subrectangle of the display pixmap into which rendering will be performed. *buf* is a display buffer (section 2.5) containing the 3D object to be rendered. *objmat* is a transformation matrix giving the orientation and position of the object. *cam* is the camera (section 2.3) to use for rendering. *lights* is the lighting model to use for illumination.

Multiple objects may be rendered into the same pixmap and rectangle by invoking the **m3dExecuteBuffer** function repeatedly. After all rendering into a rectangle is finished, **m3dEndScene** should be used to terminate rendering for this frame.

2.6.3 m3dHint

void **m3dHint**(**int** *kind*, **int** *how*)

Provides hints to the renderer about how the scene should be drawn. At present these hints are used to select the filtering and antialiasing options to be used. *kind* specifies what kind of hint is being provided.

If *kind* is M3D_MPE_USAGE, then the hint specifies how MPEs should be allocated/ *how* may then either be M3D_MPE_USE_SELF, which says that the current MPE should be used before any others, or M3D_MPE_USE_OTHERS, which says that the current MPE should be used only if all other MPEs are in use. This interacts with the argument passed to the **m3dInit** function which specifies how many MPEs to use. If four or more MPEs are requested, then clearly the current MPE will be used in all cases. If only 1 MPE is requested, then only the current MPE will be used if M3D_MPE_USE_SELF was given, and otherwise the current MPE will be used only if no other MPEs are available.

If *kind* is M3D_TEXTURE_FILTER, then the hint specifies the filtering to be applied to textures. *how* may then either be M3D_NONE to specify no filtering, or M3D_BILERP to specify bilinear filtering.

If *kind* is M3D_EDGE_AA, then the hint specfies how edge antialiasing is to be performed. In this case, *kind* may be either M3D_NONE, for no edge antialiasing, or M3D_EDGE_VMLABS to use the VM Labs edge antialiasing code. *NOTE: this edge*

antialiasing code is still under development, and performance has not been properly optimized yet.

2.6.4 m3dEndScene

void **m3dEndScene**(**mmlGC ****gc*, **mmlDisplayPixmap ****pixmap*, **m2dRect ****rect*)

Indicates that all rendering is completed, and performs any post rendering effects that are necessary. This function should be called once per frame for each output rectangle, after all calls to **m3dExecuteBuffer** for that rectangle have been finished.

2.7 Using the C API

To render an object into a display pixmap, one typically will perform the following steps:

1. Create the object (if necessary), including any materials that the object uses.
2. Initialize the libraries (this only needs to be done once).
3. Set up the lighting model.
4. Position the camera at the desired location, with the proper orientation.
5. Provide any anti-aliasing hints that are desired.
6. Set up the transformation matrix for the object.
7. Call **m3dExecuteBuffer** to draw the object.
8. When all objects have been drawn for this frame, call **m3dEndScene** to finish the frame.

The Merlin Software Developer's Kit contains a number of sample programs which illustrate this process.

3. Low level MPE routines

NOTE: Everything in this chapter should be taken with a very large grain of salt. It describes a snapshot of the low level function library at one instant in time, and is intended only as a rough guide to those who want to hack on the library source code. In cases where the source code differs from the documentation, the source code itself should be considered authoritative.

3.1 Introduction

The library provides a number of low level assembly language functions for 3D graphics, including geometry and rendering. This MPE 3D API is not a complete API. Rather, the intention is that a high level rendering pipeline can easily be built from the components provided. Users can mix and match their own components with the standard API components, in order to enhance the functionality or performance of the rendering pipeline.

The higher level C APIs are built using the MPE 3D API. When a scene is rendered, the C functions pass control to MPE code that forms the top level rendering pipeline. A standard rendering pipeline is provided, but users can replace this top level pipeline with a custom one if they wish to.

The standard pipeline uses a table to look up which functions to use for point transformation, clipping, lighting, perspective transformation, and rendering. It is therefore a simple matter to replace one or more of these modules while leaving the others intact. The C APIs use this mechanism to modify the rendering pipeline as different rendering attributes are selected (e.g. as perspective correction for texture maps is turned off or on, or as the shading level is changed).

3.2 Transformations

In the discussions below, we will use the following terms:

object coordinate system the local coordinate system for the model being rendered. Typically this is arranged with the origin at the center (or center of gravity) of the object.

world coordinate system the global coordinate system, used for positioning objects relative to one another.

camera coordinate system the local coordinate system for the camera. In this coordinate system, the positive Z axis points in the direction the camera is pointing, the positive Y axis points below the camera, the positive X axis points to the right of the camera, and the camera is positioned at the point (0,0,0).

screen coordinate system the coordinate system for the screen, where (0,0) is the upper left corner.

All coordinate systems are left handed.

Transformations between the object, world, and camera coordinate systems are specified by a 4x4 transformation matrix. The left 3x3 submatrix specifies a rotation; each element of this submatrix is a 2.30 fixed point number. The final column of the 4x4 matrix is a translation vector; each element of this is a 16.16 fixed point number. The final row of the matrix must always have a certain fixed form. The entire matrix is specified as follows:

```
xrite   xdown   xhead   xposn
yrite   ydown   yhead   yposn
zrite   zdown   zhead   zposn
  0       0       0       1
```

Note that although the matrix entries are given as 16.16 fixed point numbers to the C API, they are translated by that API into 2.30 fixed point numbers where appropriate (in the rotation elements of the matrix).

The world and camera coordinate systems are isometric, i.e. the distance between points is the same in each system. The transformation from object coordinates to world or camera coordinates may involve a scaling, but if so, the scaling must be handled automatically by the “front end” code which converts points to internal format. From the point of view of the ROM library, all transformation matrices must be isometric.

The transformation from camera coordinates to screen coordinates is a perspective transformation, and is handled in a special manner; see the description of MPE viewports (section 3.4) for details.

3.3 Lighting

All lighting calculations are performed in the camera coordinate system. This has three advantages: (1) fog and similar distance cueing effects are easy to implement, since the distance from a point to the camera can be approximated by its Z value, (2) since camera and world coordinates are isometric, distance based lighting effects work correctly in both systems, and (3) it is easy to transform from screen coordinates to camera coordinates, so per-pixel lighting calculations (e.g. for Phong shading) can share code with per-vertex lighting calculations (e.g. for Gouraud shading).

The MPE routines support 4 kinds of light: ambient light, position independent light (e.g. sunlight), position dependent (in-scene) point light sources, and position dependent spot light sources. The ambient light in a scene illuminates all objects with the same intensity, regardless of their positions or orientations. Position independent light sources are like “infinitely distant” light sources; they shine on all objects in the scene, and the direction of the light is constant. Point light sources are located at a specific point in space, and shine in all directions. The angle between a point light source and an object depends on the relative positions of the object and the light. Spot lights are a special case of point light sources; they shine only in a particular direction, and emit a cone of light at most 90 degrees wide.

The entire lighting model for a scene is specified by the following structure:

ambient light intensity	2 bytes	4.12 fixed point number; 1.0 is maximum intensity
number of directional lights	2 bytes	16 bit integer
number of in-scene lights	2 bytes	16 bit integer
number of spot lights	2 bytes	16 bit integer
reserved	8 bytes	reserved for future expansion; set to 0

After this structure the actual lights appear; first the directional lights, then the in-scene lights, then the spot lights.

Directional lights have the following structure:

nx	2 bytes	X component of normalized direction vector (2.14 format)
ny	2 bytes	Y component of normalized direction vector (2.14 format)
nz	2 bytes	Z component of normalized direction vector (2.14 format)
intensity	2 bytes	intensity of light in 4.12 format; 1.0 is maximum intensity
reserved	8 bytes	reserved for future expansion; set to 0

Positional lights have the following structure:

x position	4 bytes	x position of light in 16.16 fixed point format
y position	4 bytes	y position of light in 16.16 fixed point format
z position	4 bytes	z position of light in 16.16 fixed point format
intensity	2 bytes	intensity of light in 4.12 fixed point format (1.0 = maximum intensity)
reserved	2 bytes	set to 0; reserved for future expansion

Spot lights have the following structure:

x position	4 bytes	x position of spot light (16.16 fixed point)
y position	4 bytes	y position of spot light (16.16 fixed point)
z position	4 bytes	z position of spot light (16.16 fixed point)
intensity	4 bytes	intensity of spot light (4.28 fixed point)
x direction	4 bytes	direction of light (2.30 fixed point)
y direction	4 bytes	direction of light (2.30 fixed point)
z direction	4 bytes	direction of light (2.30 fixed point)
cone angle	4 bytes	cosine of the cone angle, as a 2.30 fixed point number; this gives the angle beyond which the spot light will have no effect

3.4 MPE Viewports and Cameras

The destination for a render is described by a viewport structure. This low level viewport structure corresponds to the higher level **mmlDisplayPixmap** type used in the Merlin Media Library C functions, together with a rectangle specifying exactly where in the display pixmap the output should go.

This structure thus describes output bitmap (its location in memory, width, height, and DMA flags) and also the camera's focal length. These parameters are used to control the perspective transformation and clipping to the viewing frustum.

The MPE viewport structure contains the following fields:

DMA flags	4 bytes	flags used to DMA into output bitmap, including pixel mode and Z buffer compare flags. Note that the Z buffer is actually a "proximity buffer" (it records 1/Z) and so the Z compare should normally be set to inhibit write if target pixel Z $>$ transfer pixel Z.
base address	4 bytes	pointer to start of output bitmap in external RAM
minimum X	2 bytes	smallest value of X to use for clipping
maximum X	2 bytes	largest value of X to use for clipping
minimum Y	2 bytes	smallest value of Y to use for clipping
maximum Y	2 bytes	largest value of Y to use for clipping
Y center of viewport	4 bytes	the center of projection in the output bitmap, as a 16.16 fixed point number; normally this will be height/2

Along with the viewport structure, a camera must be specified for rendering. The MPE camera structure contains the following fields:

view matrix	64 bytes	a 4 by 4 transformation matrix specifying the conversion from world space to camera space
focal length of camera	4 bytes	the focal length of the camera, as a 16.16 fixed point number; this should have been premultiplied by the width of the output buffer
back clipping distance	4 bytes	the distance to the back clipping plane of the viewing frustum, as a 16.16 fixed point number; this value is used only if all 6 clipping planes are enabled
X center of viewport	4 bytes	the center of projection in the output bitmap, as a 16.16 fixed point number
Y center of viewport	4 bytes	the center of projection in the output bitmap, as a 16.16 fixed point number

The 2D screen coordinates of a point are calculated from the 3D camera coordinates of the point using the following perspective transformation formulae:

```
screenX = (focald*cameraX)/cameraZ + xcenter
screenY = (focald*cameraY)/cameraZ + ycenter
```

Note that `focald` has already been multiplied by the screen width.

3.5 MPE Polygon and Point Formats

All of the MPE 3D API functions use a standard format for polygons. The first vector of the polygon has the following layout:

number of points	4 bytes	number of points in the polygon
texture map	4 bytes	pointer to a texture map or other data to be used by pixel generating function
reserved word	4 bytes	reserved, set to 0
material type	4 bytes	set to 1

After this header come the points. Each point in the polygon has the following layout:

X	16.16 fixed point	X value for point
Y	16.16 fixed point	Y value for point
Z	16.16 (later may be 2.30 fixed point)	Initially: Z value for point After perspective transformation this may be 1/Z value of point (if we are doing perspective correct Z buffering)
Iu	8.24 fixed point	shading parameter; typically texture U coordinate (with 0.0 being the left edge, 1.0 the top edge)
I0	2.30 fixed point	Initially: X component of normal vector After per-vertex lighting: Y (i.e. luma) component of color (Gouraud shading) or diffuse intensity (shaded textures) or X component of normal vector (Phong shading)
I1	2.30 fixed point	Initially: Y component of normal vector After per-vertex lighting: Cr component of color (Gouraud shading) or specular intensity (shaded textures) or Y component of normal vector (Phong shading)
I2	2.30 fixed point	Initially: Z component of normal vector After per-vertex lighting: Cb component of color (Gouraud shading) or Z component of normal vector (Phong shading)
Iv	8.24 fixed point	shading parameter, typically texture V coordinate (with 0.0 being the top edge, 1.0 the bottom)

3.6 MPE Geometry Functions

3.6.1 xformlo

Transform a point (low precision version).
Inputs:

r0 = pointer to output vertex (in standard format)
r1 = pointer to input vertex (in standard format)
r2 = pointer to 4x3 transformation matrix

Outputs:

Storage pointed to by r0 is modified.

Transforms a point from one coordinate system to another. Only the most significant 16 bits of the transformation matrix and point coordinates are used in the transformation.

3.6.2 xformhi

Transform a point (high precision version).
Inputs:

r0 = pointer to output vertex (in standard format)
r1 = pointer to input vertex (in standard format)
r2 = pointer to 4x3 transformation matrix

Outputs:

Storage pointed to by r0 is modified.

Transforms a point from one coordinate system to another. All 32 bits of the transformation matrix and point coordinates are used in the transformation.

3.7 MPE Clipping Functions

3.7.1 calcclip

Calculate outcodes for clipping.
Inputs:

r0 = points to point (in standard format) to check

Outputs:

r0 = bit mask giving the results of testing; bit N is 0 if the point is on the positive (inside) side of plane N, 1 if it is on the negative (outside) side

calcclip is intended to be used for trivial accept/reject processing. All of the points in a polygon will be tested with calcclip against the clipping planes of the viewing frustum. If calcclip returns 0 for all points, the point is within the viewing frustum and can be trivially accepted. If for some plane, calcclip returns a 1 for every point, then the polygon lies completely outside the viewing frustum and may be trivially rejected. The latter test is easily implemented by testing the bitwise AND of the values returned from calcclip.

The clipping planes are stored in a known, fixed location, and are provided in the order:

$z \geq 1$

$x \geq 0$

$x <$ screen width

$y >= 0$

$y <$ screen width

$z <$ max depth

3.7.2 doclip

Clip a polygon against a plane.
Inputs:

r0 = pointer to input polygon (in standard format)
r1 = pointer to space for output polygon
r2 = pointer to clipping plane (a small vector)

Outputs:

r0 = number of points in the clipped polygon

Clips an input polygon against a plane, producing as output a polygon all of whose points lie on the non-negative side (“inside”) of the plane. The generated polygon may have 0 to N+1 points, where N is the number of points in the input polygon. If the input polygon straddles the plane, then the output polygon will contain new points along the polygon edges which cross the plane. These new points will be produced by linearly interpolating the polygon structure elements to find their values at the point of intersection of the edge with the plane.

3.8 MPE Perspective Transformation Functions

3.8.1 persp

Do a perspective transformation on a point.
Inputs:

r0 = pointer to output point (in standard format)
r1 = pointer to input point (in standard format)
r2 = pointer to viewport data

Outputs:

Data is written to the area pointed to by r0.

Transforms a point from camera coordinates to screen coordinates. The X, Y, and Z coordinates are transformed. X and Y are projected to screen coordinates using the algorithm described in the Viewport section above. The Z coordinate is converted to 1/Z (for perspective correct Z buffering). However, no other fields of the point are modified. In particular, texture coordinates are not perspective corrected. See the perspcrct function. Note that the final pixel generation functions used for rendering (see 3D Rendering Functions below) must be matched with the perspective function (persp or perspcrct) chosen by the programmer.

3.8.2 perspcrct

Do a perspective transformation on a point and on its texture coordinates.
Inputs:

r0 = pointer to output point (in standard format)
r1 = pointer to input point (in standard format)
r2 = pointer to viewport data

Outputs:

Data is written to the area pointed to by r0.

Transforms a point from camera coordinates to screen coordinates. The X, Y, and Z coordinates are transformed as in the persp function. The U and V coordinates of the point structure are divided by Z. This makes it possible to linearly interpolate them in screen space, so that perspective correct texture coordinates can be generated. This requires that e.g. the interpolated U/Z be divided by the interpolated 1/Z at each pixel, and hence requires that the appropriate pixel generating functions be used in the rendering process. *NOTE: This function is not yet implemented in the MML3D library.*

3.9 MPE Lighting Functions

An important part of the rendering pipeline is lighting of vertices. There are actually two places where lighting is performed. Some lighting calculations are performed on a per-vertex basis; others are performed at each pixel. For example, Gouraud shading is performed by lighting each vertex of a polygon, and then linearly interpolating the resulting lighting information across the surface of the polygon. In this case the initial lighting calculations are performed per-vertex, whereas the linear interpolation

and final pixel shading are performed on a per-pixel basis. The per-pixel calculations are performed in the final rendering step, in the pixel generation functions (section 3.11). The per-vertex and per-pixel lighting must be synchronized, since the output of the per-vertex step (e.g. the Gouraud lighting coefficients) is used by per-vertex calculations. For example, mixing a Gouraud per-vertex calculation with a Phong shading pixel generation function is unlikely to produce the desired results.

The lighting functions are designed so that they may be used for either per-vertex lighting, or as part of a pixel generating function. They all use the same calling sequence and generate the same return values. Those return values consist of fixed point numbers between 0 and 1, giving the total diffuse and specular intensities at the vertex.

The diffuse intensity is the “normal” brightness of the surface; the specular intensity measures how much the light itself might be reflected from the surface. The diffuse intensity is calculated based on the assumption that the surface is a perfect reflector (i.e. that it is as bright as possible) and should be scaled according to the actual brightness of the surface. The specular intensity is calculated using the specular coefficient; the higher the coefficient, the smaller the highlight. (For each light, the specular intensity contributed by that light is proportional to the cosine of the angle between the camera’s viewing vector and the vector at which total reflectance would occur, raised to the power of the specular coefficient.) Basically, the diffuse intensity controls the blending between the surface’s color and black, and the specular intensity between the surface’s color and the light’s color (normally white). If d is the diffuse intensity, s the specular intensity, and P is a vector representing the underlying surface color at a point, then the color displayed at that point should be:

```
d*(1-s)*P + s*White
```

3.9.1 glight

Do Gouraud lighting (without specular component) for a point.
Inputs:

r0 = pointer to lighting model
r1 = pointer to vertex to be lit (in standard format)
r2 = specular coefficient of material (NOT USED)

Outputs:

r0 = diffuse intensity (in 2.30 format)
r1 = 0.0 (in 2.30 format)

Calculates the diffuse intensity for a vertex, based upon the vertex normal, the vertex position, and the lighting model. This function does not do specular lighting, and hence is significantly faster than gslight. It is suitable for chalky or dull diffuse surfaces which are not expected to have reflected highlights.

3.9.2 gslight

Do Gouraud lighting (including specular component) for a point.
Inputs:

r0 = pointer to lighting model
r1 = pointer to vertex to be lit (in standard format)
r2 = specular coefficient of material (an integer)

Outputs:

r0 = diffuse intensity (in 2.30 format)
r1 = specular intensity (in 2.30 format)

Calculates the diffuse and specular intensities for a vertex, based upon the vertex normal, the vertex position, and the lighting model. Useful for plastic or metallic surfaces.

3.10 MPE Rendering Functions

3.10.1 drawpoly

Draw a convex polygon, using a specified pixel generating function. No antialiasing is performed.
Inputs:

r0 = pointer to polygon to be drawn (in standard format)
r1 = pointer to output viewport
r2 = pointer to pixel generating function (see below)

Outputs:

None.

Draws a (non-antialiased) polygon on the bitmap associated with the given viewport. The polygon may have any number of sides, but must be completely clipped to the output bitmap (no clipping is performed by this function). The vertices must be specified in clockwise order; if they are in counter-clockwise order then no pixels are drawn. The specified pixel generating function is used for drawing the strips of (horizontal or vertical) pixels that make up the polygon. If the “texture map” field of the input polygon is non-null, and the corresponding texture will fit completely in local cache, then it is loaded into local RAM and the uv bilinear addressing registers are set up to point to it.

3.11 Pixel Generating Functions

The most basic 3D rendering functions are the pixel generating functions . These are called from the higher level polygon draw functions. A pixel generating function must generate a strip of pixels. The strip is generated as if it were a horizontal strip, although it may in fact be drawn vertically (in this case the calling function, e.g. drawpoly, will swap the x and y information input to the pixel generating function). The following predefined functions are provided for application use:

3.11.1 aabilerp

Draws texture mapped, gouraud shaded pixels with specular highlights, bilinear filtering, and edge antialiasing.

3.11.2 bilerppix

Draws texture mapped, gouraud shaded pixels. The pixels are bilinearly filtered to produce some antialiasing. No edge antialiasing is performed.

3.11.3 specpix

Draws texture mapped, gouraud shaded pixels with specular effects (no antialiasing or perspective correction).

3.12 User Supplied Pixel Generating Functions

It is expected that applications will often wish to supply their own pixel generating functions to perform various special effects such as procedural textures, or to take advantage of other special algorithms. Because of efficiency considerations, pixel generating functions do not obey the normal calling conventions. Instead, they follow the conventions described below:
Inputs:
`v4-v7`: Various inputs, as defined in the include file “drawregs.i”. The significant ones for pixel generation are:

```
_D_u:   initial texture u coordinate (16.16 fixed point
        number)
_D_v:   initial texture v coordinate (16.16 fixed point
        number)
_D_i0:  diffuse shading index (2.30 fixed point number)
_D_i1:  specular shading index (2.30 fixed point number)
_D_z:   initial depth (i.e. 1/z)
_D_du:  change in u across scan line (i.e. du/dx)
_D_dv:  change in v across scan line (i.e. dv/dx)
_D_di0: change in i0 across scan line
_D_di1: change in i1 across scan line
```

`_D_dz:  change in depth across scan line`

`rc1`: Number of pixels to generate (always > 0)

`(xy)` bilinear addressing registers: Set up to address the output area, i.e. the strip of pixels to be drawn. `ry` should never be modified; `rx` should be incremented as pixels are stored into the area.

`(uv)` bilinear addressing registers: Set up to address the input texture map, if any. Note, however, that `ru` and `rv` are NOT set up correctly, and must be initialized with a `st_io` from `_D_u` and `_D_v` respectively.

Outputs: Registers `_D_u`, `_D_v`, `_D_i0`, `_D_i1`, and `_D_z` must be updated to their new values. Generated pixels must be stored into the buffer pointed to by `(xy)`.

Other registers which may be modified: `v0`, `v1`, `rc1` All other registers should be left unmodified. In particular, `rc0` must not be changed.

See the sample pixel generating functions distributed with the MMA Software Developer's Kit for some examples of how to write pixel generating functions.

Note that the values held in registers `v2` and `v3` are not needed during pixel generation, and so it would be logical to use these as scratch registers (after saving them on the stack, of course).

Index