V M L A B S

MGL 3D Library

Application Programmer's Interface

Preliminary Specification

Revision 0.80

26-Sep-00

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

1. OVERVIEW

This document describes how to get started with mGL: OpenGL™ for the Merlin Architecture. Since mGL is a partial implementation of the established OpenGL™ 1.1 standard, we are only concerned here with showing how to initialize mGL and enumerating the implemented OpenGL™ 1.1 API calls. For an excellent tutorial and reference for OpenGL™, check out the following books:

The OpenGL™ Programming Guide, Second Edition by Mason Woo, Jackie Neider, and Tom Davis.

The OpenGL™ Reference Manual, Second Edition by Renate Kempf and Chris Frazier.

2. INITIALIZATION

While our intent is to provide a useful fraction of the OpenGL™ 1.1 API, initialization and functions like buffer swapping are always platform-specific. Accordingly, mGL is built on top of mml2D, the Merlin 2D API. To initialize mGL, one must first allocate 2 or more screen-size pixmaps in SDRAM with the following definitions and calls to the mml2D API.

```
#include <nuon/gl.h>                    // OpenGL API header file
#include <nuon/mml2d.h>                 // mml2D
#define PIXEL_TYPE e655Z                // 16 bit YCB plus 16 bit Z
#define PIXEL_FILTER eNoVideoFilter     // Video filter
#define SCRN_WIDTH 360                  // Display width
#define SCRN_HEIGHT 240                 // Display height
#define NUM_BUFFERS 2                   // 2 for double-buffering, 3 for triple
#define NUM_MPES 0                      // Number of MPEs to dedicate to mGL rendering;
                                        // the special value zero indicates that as many as
                                        // possible should be used
mmlGC               mgc;                // Merlin 2D graphics context
mmlSysResources     sysRes;             // System resource management structure
mmlDisplayPixmap    screen[NUM_BUFFERS]; // Onscreen pixmap buffers

// Initialize 2D API
mmlPowerUpGraphics(&sysRes);
mmlInitGC(&mgc, &sysRes);
mmlInitDisplayPixmaps(screen, &sysRes, SCRN_WIDTH, SCRN_HEIGHT,
                      (mmlPixFormat)PIXEL_TYPE, NUM_BUFFERS, NULL);
```

Once, mml2D is enabled, one then initializes mGL on top of it via the following steps:

```
// mGL Initialization
mglInit(screen, PIXEL_FILTER, NUM_BUFFERS, NUM_MPES);
```

At this point, mGL is ready to process OpenGL™ 1.1 API calls.

Swapping display buffers is a two step process. First, one calls **mglSwapBuffers**()which signals the video ISR to swap display buffers during the next vertical blank. At this point, one can run any application code that does not need to wait for the next vertical blank such as game logic or controller input processing. When such code is complete, one must now call **mglVideoSync**() to wait for the video ISR if it has not yet occurred or return immediately if it has happened.

3. MGL API

Platform-specific functionality for mGL is handled through a series of API calls prefixed with "mgl". These calls are documented below.

GLint **mglInit**(mmlDisplayPixmap **screen*, GLint *pixelFilter*, GLint *numBuffers*, GLint *numMPEs*);

Initializes the mGL API as described above where *screen* is a pointer to a list of *numBuffers* SDRAM-resident pixmaps, *pixelFilter* is the mml2D pixel filter, *numBuffers* is the number of display buffers, and *numMPEs* is the number of MPEs to dedicate to mGL rendering. The special value zero indicates that as many as possible should be used. Since one MPE runs the application and the high-level mGL, this number is one less than the number of MPEs in the system. **MglInit** returns 0 for success or -1 for failure.

GLint **mglEnd**(void)

Shuts down the mGL API. Returns 0 for success and –1 for failure.

GLint **mglSwapBuffers**(void)

Instructs the video interrupt service routine (ISR) to swap display buffers during the next vertical blank. This function does NOT wait for vertical blank! Returns 1 for success, 0 for failure

GLint **mglVideoSync**(void)

Waits for the next video ISR and then returns. This is used in conjunction with **mglSwapBuffers** so that one can start processing data for the next video frame upon calling **mglSwapBuffer**s and the call **mglVideoSync** when such processing cannot proceed until a video ISR has transpired.

mmlDisplayPixmap ***mglGetBuffer**(GLint *buffer*)

Allows the user to get a pointer to an mGL display buffer. This routine takes the same arguments as the OpenGL™ call **glDrawBuffer**, which is currently limited to GL_FRONT and GL_BACK. This routine is of use to the programmer wishing to mix in-house rendering code with mGL such as a voxel engine or a ray tracer.

mmlColor **mglColorFromRGB**(GLuint *r*, GLuint *g*, GLuint *b*)
mmlColor **mglColor16FromRGB**(GLuint *r*, GLuint *g*, GLuint *b*)

Returns a 32 bit YCB color given an input of red green and blue components each ranging from 0 to 255.

The following calls are used to manage textures, as the standard OpenGL™ interfaces have yet to be implemented.

The function

GLTexture ***mglNewTexture**(GLuint *width*, GLuint *height*, GLuint *pixelType*, GLuint *sdramFlag*)

allocates but does not initialize memory for a texture of the stated dimensions and pixel type. The dimensions must be powers of two. Accepted pixel types are *eClut4*, *eClut8*, *e655*, *e888Alpha*, *eClut4GRB888Alpha*, *eClut8GRB888Alpha*, *eGRB655* and *eGRB888Alpha*. If *sdramFlag* is nonzero, then SDRAM is allocated; otherwise, system RAM is allocated. If successful, a pointer to an mGL texture object is returned; otherwise, NULL is returned. The *pbuffer* field of the GLTexture structure points to the allocated texel buffer. When applicable, the *clut* field points to the palette, which always has pixel type *e655Alpha*.

The function

GLTexture ***mglInitJPEGTexture**(JOCTET **jpeg_start*, GLuint *jpeg_size*,
GLuint *pixelType*, GLint *scale,* GLuint *sdramFlag*)

creates a texture from a JPEG image of size *jpeg_size* bytes beginning at *jpeg_start*. The parameter *scale* causes the image to be scaled by a factor of 1 / *scale*. Accepted *scale* values are 1, 2, 4 and 8. Accepted pixel types are *e888Alpha*, *e655* and *eGRB655*. The parameter *sdramFlag* is as described above. If successful, a pointer to an mGL texture object is returned; otherwise, NULL is returned.

The function

GLTexture ***mglInitBMPTexture**(void **bp*, GLuint *convertToYCrCb,* GLuint *sdramFlag*)

creates a texture from a BMP image beginning at *bp*. If *convertToYCrCb* is nonzero, then the palette is converted from RGB to YCrCb. The parameter *sdramFlag* is as described above. If successful, a pointer to an mGL texture object is returned; otherwise, NULL is returned.

To set the current texture, call

void **mglSetTexture**(GLTexture **tp*)

To delete a texture, call

void **mglDeleteTexture**(GLTexture **tp*)

An application can temporarily take control of one or more rendering MPEs using

void **mglInvalidateMPE**(int commBusId)
void **mglInvalidateAllMPEs**(void)

The parameter *commBusId* is the comm. bus ID of an MPE. Attempting to invalidate an MPE not allocated by mGL has no effect. The MPE(s) may be reclaimed by mGL sometime during the next **glBegin**/**glEnd** block, **glFlush**, **glFinish**, or **mglDrawBuffer** call.

4. FIXED POINT EXTENSIONS

MGL is designed around the usage of fixed point rather than floating point math. Platform-specific extensions have been provided to take advantage of this. Their use can greatly accelerate immediate mode rendering. The following OpenGL™ API calls expect fixed point rather than floating point parameters where fixed(*n*) indicates a fixed point number with *n* fractional bits:

glVertex3fp(fixed(10), fixed(10), fixed(10))

glVertex3fpv(Glint **v*) where *v* points to a 3 element array of fixed(10)

glTexCoord1fp(fixed(18), fixed(18))

glTexCoord1fpv(Glint **v*) where *v* points to a 1 element array of fixed(18)

glTexCoord2fp(fixed(18), fixed(18))

glTexCoord2fpv(Glint **v*) where *v* points to a 2 element array of fixed(18)

glLoadMatrixfpExt(Glint **m*) where *m* points to a 16 element array of fixed(14)

glMultMatrixfpExt(Glint **m*) where *m* points to a 16 element array of fixed(14)

5. RENDERING EXTENSIONS

Currently, there is only one platform-specific rendering extension: chroma key texturing. The designation “chroma key” is in fact an historical accident, because when “chroma key” is enabled, the current implementation accepts or rejects any fragment for which alpha is not 255. It is therefore like a fixed-function OpenGL alpha test.

“Chroma key” texturing can be activated by the call glEnable(GL_CHROMAKEY_EXT). Currently, mGL supports chroma key in the following cases:

paletted texture, chroma key

paletted texture, chroma key, bilerp

paletted texture, chroma key, intensity lighting

paletted texture, chroma key, bilerp, intensity lighting

Non-paletted “chroma key” textures are not supported at present. Applications will typically create “chroma key” textures using **mglNewTexture**.

6. IMPLEMENTED OPENGL™ 1.1 API CALLS

At this time, 41 out of 74 intended OpenGL™ 1.1 API calls are implemented. Here is a list.

glBegin(GL_TRIANGLES)
glClear
glClearColor
glClearDepth
glColor|34|ifdus| v|
glDepthFunc
glDepthRange
glDisable
glDrawBuffer
glFinish
glFlush
glFrustum
glGet|BooleanDoubleFloatIntegerFixed|v
glGetError
glGetLight|fi|v
glGetMaterial|fi|v
glGetString
glGetTexParameter|fi|v
glEnable
glEnd
glLight|fi| v|
glLightModel|fi| v|
glLoadIdentity
glLoadMatrix|fd|
glMaterial|fi| v|
glMatrixMode
glMultMatrix|fd|
glNormal3|bdfis| v|
glOrtho
glPushMatrix
glPopMatrix
glRotate|fd|
glScale|fd|
glTexCoord|12|fdis| v|
glTexEnv|fi| v|
glTexParameter|fi| v|
glTranslate|fd|
glVertex|23f|fdi|v |
glViewport
gluPerspective
gluProject

7. PLANNED OPENGL™ 1.1 API CALLS

The following OpenGL™ 1.1 API calls will ultimately be available under mGL. API calls denoted with an asterisk (*) should be active in the next revision. If we have missed something you need for your killer app, let us know, we're flexible, or at the very least, we can help you implement it on your own.

glBindTexture*
glBlendFunc*
glClipPlane
glColorMaterial*
glCopyTexImage|12|D
glCopyTexSubImage|12|D
glCullFace
glDeleteTextures*
glDrawArrays
glDrawElements
glFog|fi| v|*
glGenTextures*
glGetClipPlane
glGetPointerv
glGetTexEnv|fi|v*
glGetTexGen|fdi|v
glGetTexLevelParameter|fi|v
glHint
glInterleavedArrays
glIsEnabled
glPointSize
glPolygonMode
glPolygonOffset
glPolygonStipple
glPopAttrib
glPushAttrib
glPopClientAttrib
glPushClientAttrib
glReadBuffer
glShadeModel*
glTexGen|fdi| v|
glTexImage|12|D*
glTexSubImage|12|D*

8. UNIMPLEMENTABLE OPENGL™ 1.1 API CALLS

There are several features of OpenGL™ 1.1 which are difficult to implement on the Merlin architecture. Accordingly, there are no plans to implement the following API calls at this time.

glClearStencil
glColorMask
glDepthMask
glLogicOp
glStencilFunc
glStencilMask
glStencilOp
glTexCoord|34|fdis| v|

9. IDIOSYNCRASIES

There are some limitations which one encounters unavoidably when dealing exclusively with fixed point math for polygon rendering. Here is an enumeration of strange things you might observe and how to work with or around them.

Near z-clipping plane

If one sets the near z-clipping plane to <=1.0, the quantity 1/w, which is used for perspective interpolation, will encounter an arithmetic overflow. Setting such a close z-clipping plane is usually a really bad idea on any architecture, so this shouldn't be much trouble. However, if the near z-clipping planes needs to be closer than 1.01 or so, or it is much greater than 1.0, then the minimum near z-clipping plane distance can be adjusted. To do so, locate the constant *GLMINZSHIFT* in glmpe.h and gl.i. By default, it is set to 37. Increasing this quantity by 1 will divide the minimum z-clipping plane distance by 2 and decreasing it by 1 will increase inverse multiply it by 2.

So why would one want to change this quantity? Well, the dynamic range of 1/w where the quantity is guaranteed to possess 16 bits or more of precision ranges from the minimum z-clipping plane distance to 65536.0 times this distance. One bit of precision is lost for every factor of 2 beyond this point. This effect should probably never show up under normal circumstances. However, you are now prepared to deal with the situation if it does. The default fixed point math settings of mGL provide a dynamic range of + or – 1,048,576 for all coordinates with 10 bits of fractional precision. Hopefully, this is sufficient for most applications. If the units here corresponded to feet, this would translate to an operational range of + or - 200 miles in each coordinate with a maximum resolution of 1/100 of an inch.

Big Polygons

If one attempts to render a polygon which violates both the near and far z-clipping plane simultaneously, one might encounter occasional z-buffer errors that result in blank or incorrectly rendered pixels. This is once again an artifact of fixed point math. In this case, a slight error in fixed point interpolation has led to an arithmetic overflow either beyond *zMax* ($7fffffff) or below *zMin* (0). The reason for this is that fixed point interpolants are only guaranteed 16 bits of fractional precision within a 16 bit dynamic range while the polygon in question traverses 32 such orders of magnitude!

Fortunately, there are several workarounds. The obvious solution is not to get in this situation in the first place. Failing that, one can sacrifice 1% or so of the z-buffer on each end with the following API call:

glDepthRange(0.01, 0.99)

This will insure that the fixed point roundoff error no longer results in arithmetic overflow.

Viewport Considerations

Here's a quick reminder that Merlin pixels are NOT square! Rather, they are a 9:8 rectangle. This means that the x width of a viewing frustum should be scaled by a factor of 1.125 in order to achieve squares that are truly square.

A second viewport-related issue is that fixed point clipping math is not perfect. While no polygon cracking should occur, the clipping boundary itself can waver + or - 1 pixel due to truncation and roundoff errors. This effect can occur on hardware accelerators as well, but a hardware clipping rectangle is used to conceal it in that case. Since Merlin has no hardware clipping, this effect is unavoidable. The solution is to a) insure that the viewport boundary is entirely in the overscan region or b) render something on top of this region and the completion of polygon rendering (another solution used in several hardware accelerators which lack hardware clipping).

Lighting Considerations

MGL's lighting model is an approximation of that of OpenGL 1.1:

- Up to 4 positional or directional light sources can be active at any given time.
- Only directional lights are supported.
- Backface material properties are ignored.
- GL_NORMALIZE is not implemented. Normals passed from the application must have unit length.

Texture Tiling

MGL is currently limited to + or - 32 repetitions of a texture rather than + or - 127 as specified by the OpenGL™ 1.1 standard. Once again, this is a user-specifiable quantity. To increase the maximum repetitions by a factor of 2, decrease the quantity *GLTEXCOORDSHIFT*, in both glmpe.h and gl.i, by1. Inversely, to decrease the maximum repetitions by 1, increase *GLTEXCOORDSHIFT* by 1. Why would one choose to do the latter? Increasing *GLTEXCOORDSHIFT* by one increases the fractional precision of texture coordinate interpolation.

Texture Dimensions

This release of mGL only supports 16 bit video and 16 bit YCB, GRB and 4 bit paletted textures. The maximum size of a 16 bit texture is 1024 pixels whose dimensions are arranged as the products of any two powers of two (such as 32x32, 16x64, 8x128, and so on). The maximum size of a 4 bit paletted texture is 4096 pixels whose dimensions are subject to the same constraints as 16 bit textures i.e. 64x64, 128x32, 256x16 and so on.

Texture Environment

Currently, only the *GL_REPLACE* and *GL_MODULATE* texture environment modes are supported.