Freescale Semiconductor
Application Note

Document Number: AN3994
Rev. 0, 01/2010

High-End 3D Graphics with OpenGL ES 2.0

by Multimedia Application Division
Freescale Semiconductor, Inc.
Austin, TX

These days 3D graphics for embedded devices have become more common and complex. The i.MX51 multimedia processor offers a complete solution for a developer interested in 3D graphics.

This application note intends to:

- Be a startup guide on understanding the principles of 3D technology and the new OpenGL ES 2.0 standard.
- Help the reader to understand the foundation of 3D graphics, the OpenGL ES 2.0, and its programmable pipeline.
- Explain the 3D concepts needed to develop applications and culminates with a basic example that the developer can use as a template for further development.

Contents

1. OpenGL Summary 1
2. 3D Graphics Primer 2
3. Embedded-System Graphics Library (EGL) 6
4. OpenGL Summary 10
5. OpenGL ES 2.0 Code Walkthrough 16
6. Conclusion 19
7. References. 19
8. Revision History 20
A. Developing OpenGL ES 2.0 Applications with Microsoft Visual Studio and WinCE 6.0 20

1 OpenGL Summary

OpenGL® is the most widely adopted standard for real-time computer graphics, covering most operating systems. It has several applications. OpenGL enables speed and innovation by incorporating a broad set of rendering, texture mapping,

© Freescale Semiconductor, Inc., 2010. All rights reserved.

freescale™ semiconductor

special effects, and other powerful visualization functions. Developers can leverage the power of OpenGL across all popular desktop and workstation platforms, ensuring wide application deployment.

OpenGL is very stable because it has been available for more than seven years on a wide variety of platforms, with a very high standard for additions, including backward compatibility. Therefore, the existing applications do not become obsolete. Also, OpenGL is totally portable. This allows developers to be sure that their applications have the same visual result on any OpenGL API-compliant hardware, not worrying about the underlying operating system. Additionally, OpenGL allows the new hardware innovations to be implemented by the extensions mechanism, thereby letting the vendors incorporate new features in their new products.

OpenGL is well structured with an intuitive design and logical commands. Efficient OpenGL routines typically result in applications with fewer lines of code than those that make up programs generated using other graphics libraries or packages. In addition, OpenGL drivers encapsulate information about the underlying hardware, freeing the application developer from having to design for specific hardware features. Figure 1 shows the OpenGL visualization programming pipeline.

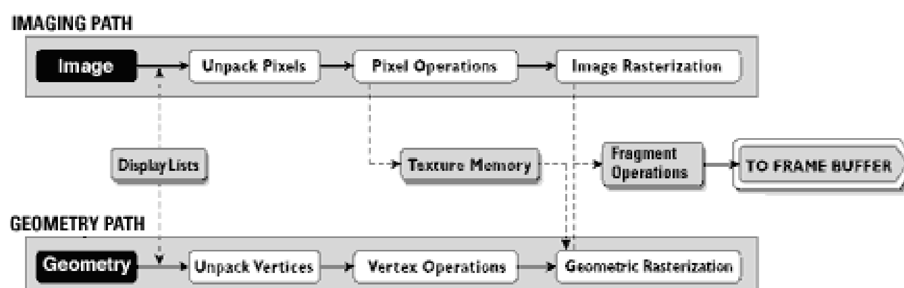

Figure 1. The OpenGL Visualization Programming Pipeline

OpenGL routines simplify the development of graphics software from rendering a simple geometric point, line, or filled polygon to the creation of the most complex lighted and texture-mapped surfaces. OpenGL gives software developers access to geometric and image primitives, display lists, modeling transformations, lighting and texturing, anti-aliasing, blending, and many other features.

All elements of the OpenGL state including the contents of the texture memory and the frame buffer are obtained by an OpenGL application. OpenGL also supports visualization applications with 2D images treated as types of primitives that can be manipulated just like 3D geometric objects.

2 3D Graphics Primer

Before going into further details of the OpenGL ES 2.0 example code, take a look at the general concepts and terminology related to 3D graphics in the subsections that follow. Examples relating to OpenGL ES have been used where ever possible.

2.1 Triangles and Lines

A vertex is the most basic concept in OpenGL, which defines a point in a 3D space. A vertex has three coordinates— x,y,z, usually defined as float values. The union of three vertices (v1,v2,v3) constructs a

triangle. A line is constructed by the union of only two vertices. Figure 2 shows that OpenGL can draw lines, triangles, and quadrilaterals, while OpenGL ES 2.0 can only draw triangles and lines.

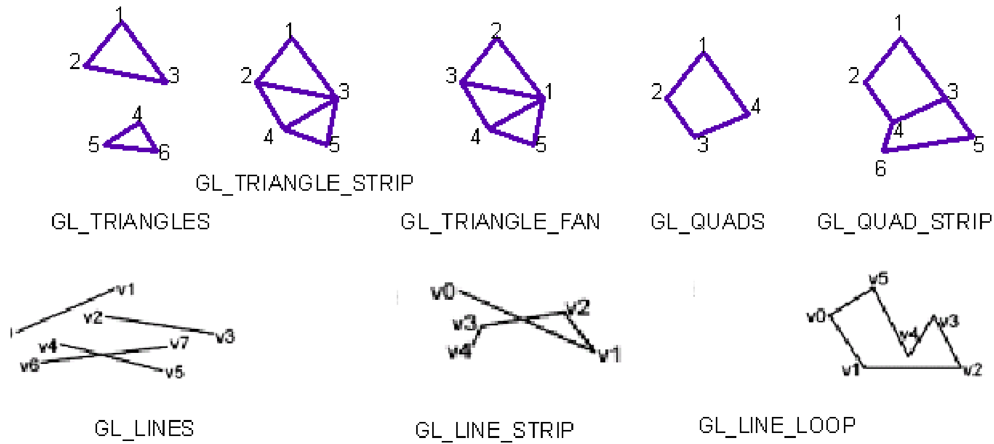

Figure 2. Triangles and Lines

2.2 3D Meshes

A 3D model or mesh is a collection of triangles, which can be represented in an array. This is called the vertex array, usually in the form of a triangle strip. Figure 3 shows an example of a 3D mesh.

Figure 3. 3D Mesh

The above 3D mesh can be created with a 3D modeling tool, such as: Blender, 3D Studio Max, Maya, Lightwave 3D, and so on. A 3D mesh file can be exported in many formats, optimized or non-optimized. Example 1 shows how a .OBJ file (an universal 3D mesh format) looks.

Example 1. Sample .OBJ File

```
####
#
#        OBJ File Generated by LightWave3D
#        LightWave3D OBJ Export v2.2
#
####
#        Object: 1
#
#                        Vertices: 5
#                          Points: 1
#                           Lines: 0
#                           Faces: 6
#                       Materials: 1
#
####

o 1

#                      Vertex list

v -0.5 -0.5 0.5
v -0.5 -0.5 -0.5
v 0.5 -0.5 0.5
v 0.5 -0.5 -0.5
v 0 0.5 0

#            Point/Line/Face list

f 5 2 1
f 2 3 1
f 2 4 3
f 5 4 2
f 3 5 1
f 4 5 3
#                      End of file
```

This file represents a pyramid. A pyramid has five vertices and six faces (triangles):

- The first part is the general information of the 3D mesh. It states the number of vertices and faces (triangles).
- The next part is the vertex list as follows:

  ```
  # Vertex list
  v -0.5 -0.5 0.5
  ```

  The **v** character stands for vertex. This line means that a vextex is constructed by the three x,y,z coordinates (-0.5, -0.5, 0.5).
- The last part is about how to create the actual triangles

  ```
  # Point/Line/Face list
  f 5 2 1
  ```

The **f** character stands for face which actually means a triangle, and the following three numbers are the vertex indices. This means that a face is constructed by 3 vertices (the 5th, the 2nd, and the 1st).

With this information, one can re-create any 3D model from a .OBJ file. Note that the .OBJ file is not optimized. However, there are several options, like using the Power VR (from Imagination Technologies) .POD file plug-in to export and load 3D meshes. This plug-in sorts the geometry (as triangle strips) to optimize them and avoid vertex redundancies.

2.3 Textures

A texture is a 2D still image (.bmp, .jpeg, .png, .gif, .tga files). This image can be applied to a 3D surface (a triangle) by adding a pair of coordinates (U,V) to each vertex of the 3D mesh. After this pixels are sampled from the image and interpolated (with the GPU) to achieve a decal effect on the 3D mesh. Figure 4 shows textures.

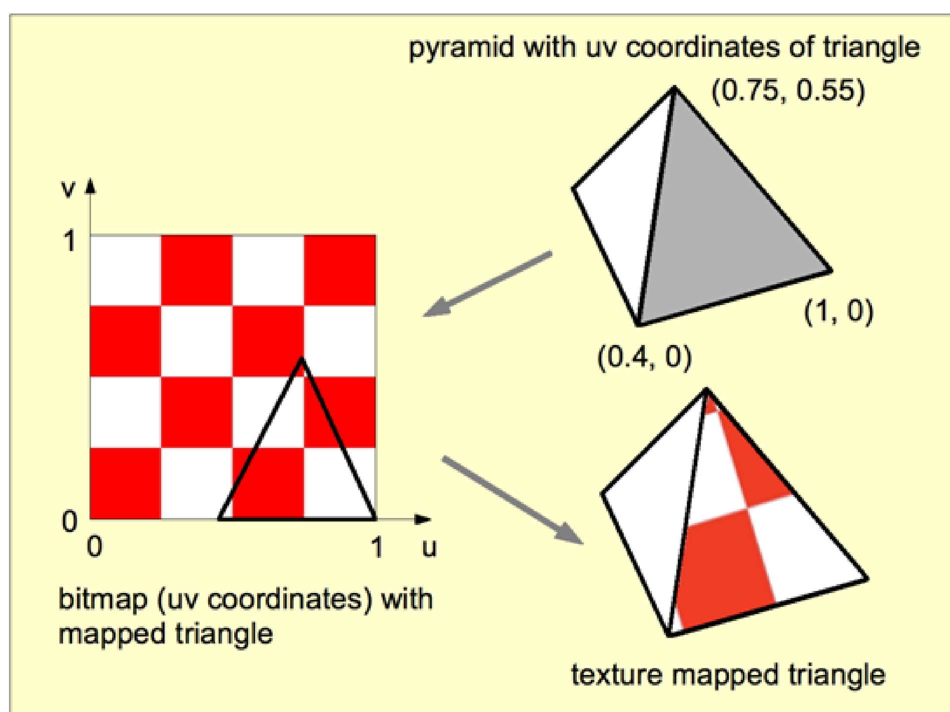

Figure 4. Textures

Figure 5 shows a .jpg image (a human face) being mapped into a 3D mesh.

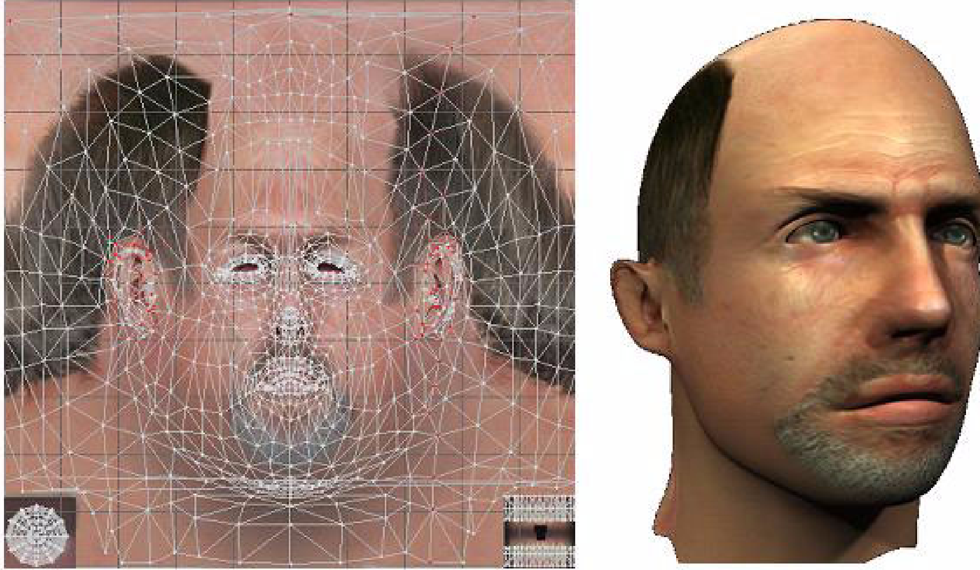

Figure 5. Mapping a .jpg Image Into a 3D Mesh

Texture coordinates are specified at each vertex of a given triangle, and these coordinates are interpolated using an extended Bresenham´s line algorithm. If these texture coordinates are linearly interpolated across the screen, the result is an affine texture mapping. This is a fast calculation but there can be a noticeable discontinuity between adjacent triangles when these triangles are at an angle to the plane of the screen.

Perspective-correct texturing accounts for the vertices' positions in 3D space, rather than simply interpolating a 2D triangle. This achieves the correct visual effect, but it is slower to calculate. Instead of interpolating the texture coordinates directly, the coordinates are divided by their depth (relative to the viewer). The reciprocal of the depth value is also interpolated and used to recover the perspective-correct coordinate. This correction makes it such that in parts of the polygon, that are closer to the viewer, the difference from pixel to pixel between texture coordinates is smaller (stretching the texture wider), and in parts that are farther away, this difference is larger (compressing the texture). The GPU hardware implements perspective-correct texturing so none of this has to be done by the CPU.

3 Embedded-System Graphics Library (EGL)

EGL™ handles graphics context management, surface/buffer binding, and rendering synchronization. EGL enables high-performance, accelerated, mixed-mode 2D and 3D rendering using other Khronos APIs, such as OpenVG™ and OpenGL.

EGL can be implemented on multiple operating systems (embedded Linux®, WinCE, and Windows®) and native window systems (such as X and Microsoft Windows). Implementations may also choose to allow rendering into specific types of EGL surfaces through other supported native rendering APIs such as Xlib or GDI.

EGL provides the following features:

- Mechanisms for creating rendering surfaces (windows, pbuffers, pixmaps) onto which client APIs can draw and share
- Methods to create and manage graphics contexts for client APIs
- Ways to synchronize drawing by client APIs and native platform rendering APIs

EGL itself is independent of definitions and concepts specific to any native Windows system or rendering API to a certain extent. However, there are a few places where native concepts must be mapped into EGL-specific concepts, including the definition of the display on which graphics are drawn, and the definition of native windows. Figure 6 shows the overall EGL structure and how it is used to create graphics contexts and surfaces to higher level APIs such as OpenVG and OpenGL ES.

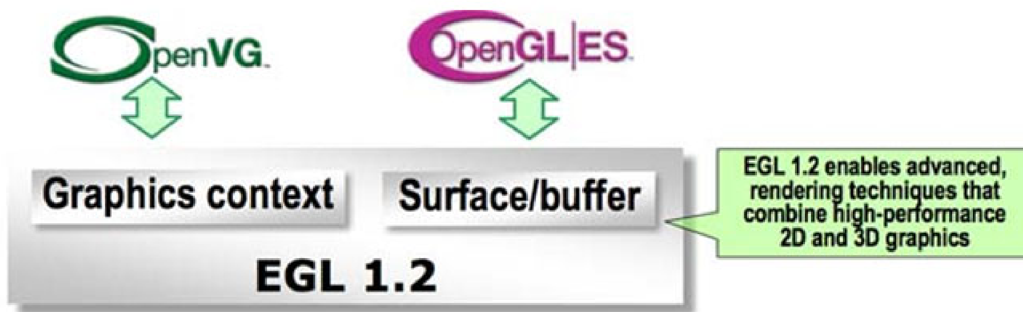

Figure 6. The EGL Structure

3.1 Steps to Initialize EGL and Create a Rendering Context

Most EGL calls include an **EGLDisplay** parameter. This parameter represents the abstract display on which graphics are drawn. In most environments a display corresponds to a single physical screen.

STEP 0:

The **EGLConfig** describes the depth of the color buffer components and the types, quantities and sizes of the *ancillary buffers* for an EGLSurface. If the EGLSurface is a window, then the EGLConfig describing it may have an associated native *visual type*. Example 2 shows the names of EGLConfig attributes. These names may be passed to **eglChooseConfig** to specify required attribute properties.

Example 2. EGLConfig Attribute Names

```
static const EGLint s_configAttribs[] =
         {
                  EGL_RED_SIZE,      5,
                  EGL_GREEN_SIZE, 6,
                  EGL_BLUE_SIZE,     5,
                  EGL_ALPHA_SIZE, 0,
                  EGL_LUMINANCE_SIZE, EGL_DONT_CARE,
                  EGL_SURFACE_TYPE,EGL_VG_COLORSPACE_LINEAR_BIT,
                  EGL_SAMPLES,       0,
                  EGL_NONE
         };
```

- EGL RED SIZE, EGL GREEN SIZE, EGL BLUE SIZE, EGL BLUE SIZE, and EGL ALPHA SIZE give the depth of the color in bits.
- EGL SURFACE TYPE is a mask indicating the surface types that can be created with the corresponding EGLConfig.
- EGL_SAMPLES is the number of samples per pixel.

STEP 1: Initialization

Initialization must be performed once for each display prior to calling most other EGL functions. A display can be obtained by calling:

EGLDisplay **eglGetDisplay**(NativeDisplayTypedisplay_id);

The type and format of *display id* are implementation-specific, and it describes a specific display provided by the system EGL is running on. For example, an EGL implementation under X windows would require *display id* to be an X Display, while an implementation under Microsoft Windows would require *display id* to be a Windows Device Context. If *display id* is EGL DEFAULT_DISPLAY, a *default display* is returned.

EGL may be initialized on a display by calling:

EGLBoolean **eglInitialize**(EGLDisplay *dpy*, EGLint **major*, EGLint **minor*);

- EGL_TRUE is returned on success, and *major* and *minor* are updated with the *major* and *minor* version numbers of the EGL implementation. *major* and *minor* are not updated if they are specified as NULL.
- EGL_FALSE is returned on failure and *major* and *minor* are not updated.
- An EGL_BAD_DISPLAY error is generated if the *dpy* argument does not refer to a valid EGLDisplay.
- An EGL NOT_INITIALIZED error is generated if EGL cannot be initialized for an otherwise valid *dpy*.

Example 3 shows EGL initialization.

Example 3. EGL Initialization

```
egldisplay = eglGetDisplay(EGL_DEFAULT_DISPLAY);
eglInitialize(egldisplay, NULL, NULL);
assert(eglGetError() == EGL_SUCCESS);
eglBindAPI(EGL_OPENVG_API);
```

eglBindAPI() sets a given state to the Current Rendering API in this case is OpenVG.

STEP 2: Configuration

As mentioned before, EGLConfig describes the format, type and size of the color buffers and ancillary buffers for an EGLSurface. If the EGLSurface is a window, then the EGLConfig describing it may have an associated native *visual type*. Names of EGLConfig attributes may be passed to **eglChooseConfig()** to specify required attribute properties. Example 4 shows EGL configuration.

```
EGLBoolean eglChooseConfig(EGLDisplay dpy, const EGLint *attrib list,
EGLConfig *configs, EGLint config_size,
EGLint *num config);
```

Example 4. EGL Configuration

```
eglChooseConfig(egldisplay, s_configAttribs, &eglconfig, 1, &numconfigs);
assert(eglGetError() == EGL_SUCCESS);
assert(numconfigs == 1);
```

STEP 3: Rendering Contexts and Drawing Surfaces

- **EGLSurfaces**

  One of the purposes of EGL is to provide a means to create an OpenVG context and associate it with a surface. EGL defines several types of drawing surfaces collectively referred to as EGLSurfaces. To create an on-screen rendering surface, first create a native platform window with attributes corresponding to the desired EGLConfig. Using a platform-specific type (here called NativeWindowType) referring to a handle to that native window, then call:

  ```
  EGLSurface eglCreateWindowSurface(EGLDisplay dpy,
  EGLConfig config, NativeWindowType win,
  const EGLint *attrib list);
  ```

  eglCreateWindowSurface creates an onscreen EGLSurface and returns a handle to it. Any EGL rendering context created with a compatible EGLConfig can be used to render into this surface.

- **Rendering Contexts**

  To create a **Rendering Context** call:

  ```
  EGLContext eglCreateContext(EGLDisplay dpy, EGLConfig config,
  EGLContext share context,
  const EGLint *attrib list);
  ```

  If **eglCreateContext** succeeds, it initializes the rendering context to the initial OpenVG state and returns a handle to it. The handle can be used to render to any compatible EGLSurface. Currently no attributes are recognized, so *attrib_list* are normally NULL or empty (first attribute is EGL_NONE).

STEP 4: Bind Context and Surfaces

To make a context current, call:

```
EGLBoolean eglMakeCurrent(EGLDisplay dpy,EGLSurface draw,
EGLSurface read, EGLContext ctx);
```

eglMakeCurrent binds *ctx* to the current rendering thread and to the *draw* and *read* surfaces. Note that the same EGLSurface may be specified for both *draw* and *read*. **eglMakeCurrent** returns EGL_FALSE on failure. If *draw* or *read* are not compatible with *ctx*, then an EGL_BAD_MATCH error is generated.

See implementation in Example 5 below:

Example 5. Bind Context and Surfaces

```
eglsurface = eglCreateWindowSurface(egldisplay, eglconfig, open("/dev/fb0",
        O_RDWR), NULL);
assert(eglGetError() == EGL_SUCCESS);
```

```
eglcontext = eglCreateContext(egldisplay, eglconfig, NULL, NULL);
assert(eglGetError() == EGL_SUCCESS);
eglMakeCurrent(egldisplay, eglsurface, eglsurface, eglcontext);
assert(eglGetError() == EGL_SUCCESS);
```

3.2 EGL Deinitialization

Deinitialization of EGL is quite straight-forward. See Example 6 below:

Example 6. Deinitializing EGL

```
eglMakeCurrent(egldisplay, EGL_NO_SURFACE, EGL_NO_SURFACE, EGL_NO_CONTEXT);
assert(eglGetError() == EGL_SUCCESS);
eglTerminate(egldisplay);
assert(eglGetError() == EGL_SUCCESS);
eglReleaseThread();
```

First, eglMakeCurrent() is called with EGL_NO_SURFACE and EGL_NO_CONTEXT. To release resources associated with use of EGL and OpenVG on a display call:

EGLBoolean `eglTerminate`(EGLDisplay *dpy*);

Termination marks all EGL-specific resources associated with the specified display for deletion. `eglTerminate` returns EGL_TRUE on success. If the dpy argument does not refer to a valid EGLDisplay, EGL_FALSE is returned, and an EGL_BAD_DISPLAY error is generated. EGL maintains a small amount of per-thread state, including the error status returned by **eglGetError**, the currently bound rendering API defined by **eglBindAPI**, and the current contexts for each supported client API.

To return EGL to its state at thread initialization, call:

EGLBoolean `eglReleaseThread`(void);

EGL_TRUE is returned on success, and the following actions are taken:

- For each client API supported by EGL, if there is a currently bound context, that context is released. This is equivalent to calling eglMakeCurrent with ctx set to EGL_NO_CONTEXT and both draw and read set to EGL_NO_SURFACE.
- The current rendering API is reset to its value at thread initialization.
- Any additional implementation-dependent per-thread state maintained by EGL is marked for deletion as soon as possible.

4 OpenGL Summary

OpenGL ES™ is an API for advanced 3D graphics targeted at handheld and embedded devices, such as the i.MX51 Freescale Multimedia Processor.

OpenGL ES is a subset of desktop OpenGL, creating a flexible and powerful low-level interface between software and graphics acceleration. OpenGL ES includes profiles for floating-point and fixed-point systems and the EGL specification for portable binding to native windowing systems.

The OpenGL API is very large and complex. However, the goal was to create an API suitable for constrained devices. To achieve this goal, redundancy was removed from the OpenGL API. In any case,

where there was more than one way of performing the same operation, the most useful method was taken and the redundant techniques were removed.

Removing redundancy was an important goal. However, maintaining compatibility with OpenGL was equally important. OpenGL ES was designed such that applications that were written to the embedded subset of functionality in OpenGL would also run on OpenGL ES. This was an important goal as it allows developers to leverage both APIs and develop applications and tools that use the common subset of functionality.

New features were introduced to address specific constraints of handheld and embedded devices. For example, to reduce the power consumption and to increase the performance of shaders, precision qualifiers were introduced to the shading language. See Figure 7.

Figure 7. OpenGL ES

The OpenGL ES specification includes the definition of several profiles. Each profile is a subset of a version of the desktop OpenGL specification and some additional OpenGL ES-specific extensions. The OpenGL ES profiles are part of a wider family of OpenGL-derived application programming interfaces. As such, the profiles share a similar processing pipeline, command structure, and the same OpenGL name space. There are three OpenGL ES specifications that have been released by Khronos which are as follows:

- OpenGL Es 1.0
- OpenGL Es 1.1
- OpenGL Es 2.0

The OpenGL ES 1.0 and 1.1 specifications implement a fixed function pipeline and are derived from the OpenGL 1.3 and 1.5 specifications, respectively.

Figure 8 shows the ES 2.0 programmable pipeline.

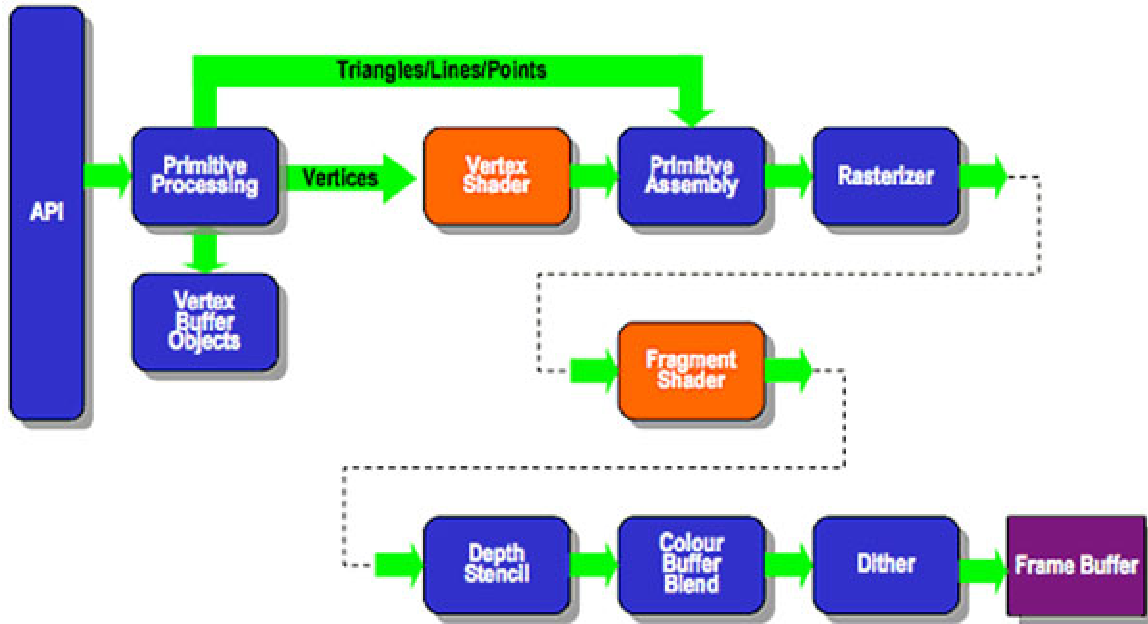

Figure 8. ES2.0 Programmable Pipeline

OpenGL ES 2.0 combines a version of the OpenGL shading language for programming vertex and fragment shaders that have been adapted for embedded platforms, together with a streamlined API from OpenGL ES 1.1 that has removed any fixed functionality that can be easily replaced by shader programs, to minimize the cost and power consumption of advanced programmable graphics subsystems.

Figure 9 shows how the programmable graphics pipeline works.

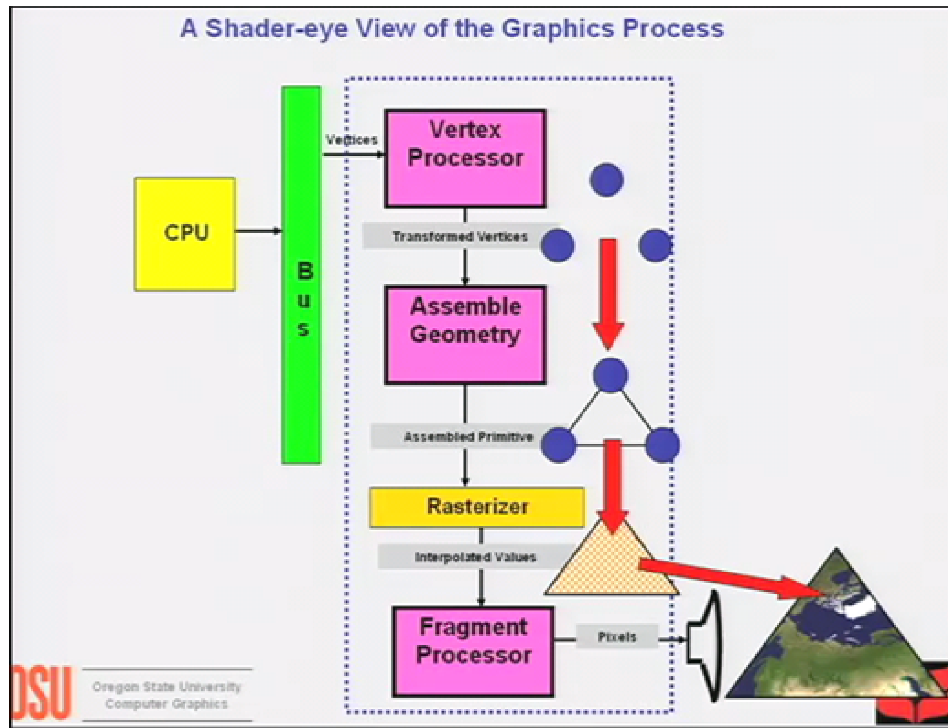

Figure 9. Graphics Process

4.1 The Vertex Shader

The vertex shader is a general-purpose programmable method for operating on vertices. Vertex shaders can be used for traditional vertex-based operations such as transforming the position by a matrix, computing the lighting equation to generate a per-vertex color, and generating or transforming texture coordinates.

The vertex shader has the following inputs:

- Attributes: constant in per-vertex data using vertex arrays
- Uniforms: constant data used by the vertex shader
- Samplers: represents textures used by the vertex shader, these are optional
- Shader program: the actual source code of the shader, contains the instructions and operations that are performed on the vertex.
- Assemble Geometry:

  A primitive is a geometric object that can be drawn using appropriate drawing commands in OpenGL ES 2.0. These drawing commands specify a set of vertex attributes that describe a primitive's geometry and type (triangles, lines). Each vertex is described with a set of vertex attributes. These vertex attributes contain information that the vertex shader uses to calculate a position and other information that can be passed to the fragment shader, such as its color and texture coordinates.

The outputs of the vertex shader are called varying variables. In the primitive rasterization stage, the varying values are calculated for each generated fragment and are passed as inputs to the fragment shader. The mechanism used to generate a varying value for each fragment from the varying values assigned to each vertex of the primitive is called interpolation (see Figure 9).

- Rasterizer:

  Rasterization is the process that converts primitives, which can be points, lines, or triangles into a set of two-dimensional fragments, which are processed by the fragment shader. These two-dimensional fragments represent pixels that can be drawn on the screen as shown in Figure 10.

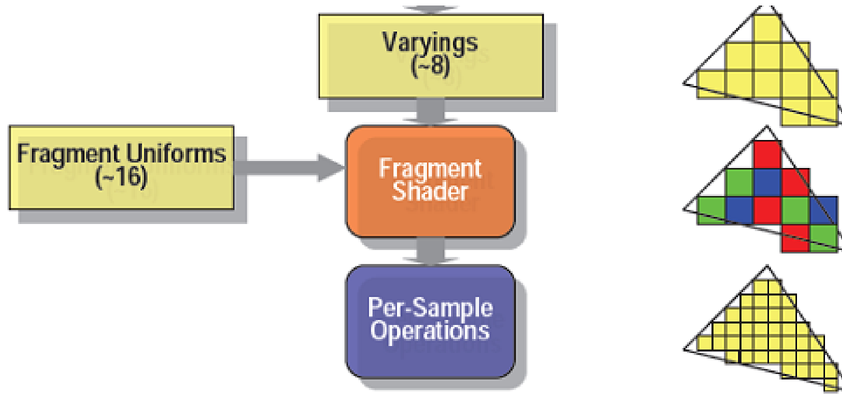

Figure 10. Rasterization

4.2 The Fragment Shader

The fragment shader is a general-purpose method for interacting with fragments. The fragment shader program is executed for each fragment in the rasterization stage.

The fragment shader has the following inputs:

- Varying variables: Outputs of the vertex shader that are generated by the rasterization unit for each fragment using interpolation.
- Uniforms: Constant data used by the fragment shader.
- Samplers: A specific type of uniform that represent textures used by the fragment shader.
- Shader program: Fragment shader program source code or executable that describes the operations that are performed on the fragment.

  The fragment shader can either discard the fragment or generate a color value referred to as gl_FragColor. The color, depth, stencil, and screen coordinate location (x, y) of screen coordinates generated by the rasterization stage become inputs to the per-fragment operations stage of the pipeline.

After the fragment shader, the next stage is per-fragment operations. A fragment produced by rasterization with (x,y) screen coordinates can only modify the pixel at location (x,y) in the framebuffer (see Figure 11).

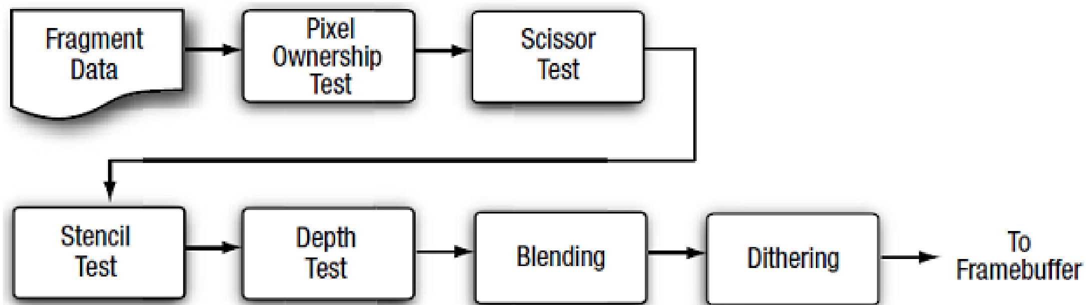

Figure 11. Fragment Shader

At the end of the per-fragment stage, either the fragment is rejected or a fragment color, depth, or stencil value is written to the framebuffer at the location (x,y) of the screen. The fragment color, depth, and stencil values are written depending on whether the appropriate write masks were enabled.

- Compiling and using the shader

  After understanding the basic elements of a shader and defining a source code for each shader (vertex and fragment), a series of steps are required in order to first create the objects, compile, and use the shaders (see Figure 12).

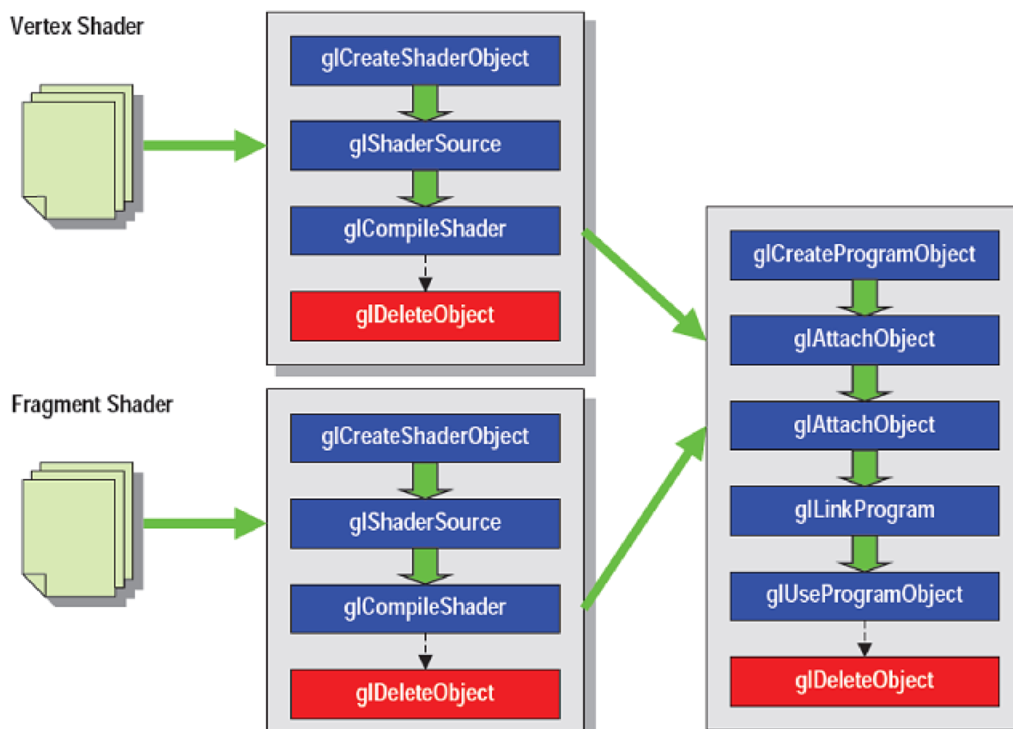

Figure 12. Compiling and Using the Shader

Example 7 shows how the shader object is created using glCreateShader, which creates a new shader object of the type specified.

Example 7. Creating a New Shader Object

```
GLuint vertexShaderObject= glCreateShaderObject(GL_VERTEX_SHADER);
GLuint fragmentShaderObject = glCreateShaderObject(GL_FRAGMENT_SHADER);
```

After a new shader obeject is created, the *glShaderSource* is called to load the shader code into the object. Then the shader is compiled using *glCompileShader*. See Example 8.

Example 8. Loading the Shader Code Into the Object

```
glShaderSource(vertexShaderObject, 1, &shaderSrc, NULL);
glShaderSource(fragmentShaderObject, 1, &shaderSrc, NULL);
glCompileShader(vertexShaderobject);
glCompileShader(vertexShaderobject);
```

After the shader is compiled successfully, a new shader object is returned. This object has to be attached to the program object. The program object is the final linked program. When each shader is compiled into a shader object they are attached to a program object and linked together before drawing as shown in Example 9.

Example 9. Linking the Program Object

```
GLuint programObject;
glAttachObject(programObject, vertexShaderObject);
glAttachObject(programObject, fragmentShaderObject);
glLinkProgram(progamObject);
glUseProgramObject(programObject);
```

After successful linking of the program object, it can now be used for the actual rendering. To use the program object for rendering, bind it using *glUseProgram*.

5 OpenGL ES 2.0 Code Walkthrough

This following section covers a simple, step by step openGL ES 2.0 example, excluding the previously covered EGL section.

1. Include the necessary header files as shown in Example 10.

Example 10. Header Files

```
#include <EGL/egl.h>
#include <GLES2/GL2.h>
#include <GLES2/gl2ext.h>
```

2. Define the vertex shader and fragment shader as shown in Example 11.

Example 11. Define the Vertex and Fragment Shader

```
const CHAR* g_strVSProgram =
         "attribute vec4 g_vVertex; \n"
         "attribute vec4 g_vColor;  \n"
         "varying   vec4 g_vVSColor;  \n"
                                    "                 \n"
```

```
        "void main()                                    \n"
        "{                                              \n"
        " gl_Position  = vec4( g_vVertex.x, g_vVertex.y,    \n"
"                        g_vVertex.z, g_vVertex.w );  \n"
        "    g_vVSColor = g_vColor;                     \n"
"}                                                      \n";
const CHAR* g_strFSProgram =
        "#ifdef GL_FRAGMENT_PRECISION_HIGH \n"
        "   precision highp float;         \n"
        "#else                             \n"
        "   precision mediump float;       \n"
        "#endif                            \n"
        "                           "       \n"
        "varying   vec4 g_vVSColor;         \n"
        "                           "      \n"
        "void main()                       \n"
        "{                                 \n"
        "    gl_FragColor = g_vVSColor;   \n"
        "}                                 \n";
```

3. After the shader source code, create the program, set the shader source and compile the shader as shown in Example 12.

Example 12. Compiling the Shader

```
GLuint       g_hShaderProgram = 0;
GLuint       g_VertexLoc = 0;
GLuint       g_ColorLoc  = 1;
GLuint hVertexShader = glCreateShader( GL_VERTEX_SHADER );
glShaderSource( hVertexShader, 1, &g_strVSProgram, NULL );
glCompileShader( hVertexShader );
GLuint hFragmentShader = glCreateShader( GL_FRAGMENT_SHADER );
glShaderSource( hFragmentShader, 1, &g_strFSProgram, NULL );
glCompileShader( hFragmentShader );
```

4. Check for errors during the compiling process as shown in Example 13.

Example 13. Check for Errors

```
GLint nCompileResult = 0;
glGetShaderiv(hFragmentShader, GL_COMPILE_STATUS,
                 &nCompileResult);
if (!nCompileResult)
         {
           CHAR Log[1024];
           GLint nLength;
           glGetShaderInfoLog(hFragmentShader, 1024, &nLength,
                  Log);
           return FALSE;
          }
```

5. Attach the individual shaders to the shader program as shown in Example 14.

Example 14. Attach Individual Shaders

```
g_hShaderProgram = glCreateProgram();
glAttachShader(g_hShaderProgram, hVertexShader);
glAttachShader(g_hShaderProgram, hFragmentShader);
```

6. The attributes must be initialized before the linking as shown in Example 15.

Example 15. Initialize Attributes

```
glBindAttribLocation(g_hShaderProgram, g_VertexLoc,"g_vVertex");
glBindAttribLocation(g_hShaderProgram, g_ColorLoc,"g_vColor");
glLinkProgram( g_hShaderProgram );
```

7. Check if the linking was a success and after that delete the individual shaders as shown in Example 16.

Example 16. Delete Individual Shaders

```
GLint nLinkResult = 0;
glGetProgramiv(g_hShaderProgram, GL_LINK_STATUS, &nLinkResult);
if (!nLinkResult)
{
CHAR Log[1024];
GLint nLength;
glGetProgramInfoLog(g_hShaderProgram, 1024, &nLength, Log);
return FALSE;
}
glDeleteShader( hVertexShader );
glDeleteShader( hFragmentShader );
}
```

8. A rendering loop is an endless loop or a fixed number of frames depending on the application. The first part consists of the vertex data of the triangles, followed by the regular OpenGL functions for clearing the screen (glClear). After the screen has been cleared, use the shader program with *glUseProgram* as shown in Example 17.

Example 17. Rendering Loop

```
VOID Render()
{
         FLOAT fSize = 0.5f;
         FLOAT VertexPositions[] =
         {
         0.0f,  +fSize*g_fAspectRatio, 0.0f, 1.0f,
         -fSize, -fSize*g_fAspectRatio, 0.0f, 1.0f,
         +fSize, -fSize*g_fAspectRatio, 0.0f, 1.0f,
         };

         FLOAT VertexColors[] = {1.0f, 0.0f, 0.0f, 1.0f,
                                    0.0f, 1.0f, 0.0f, 1.0f,
                                    0.0f, 0.0f, 1.0f, 1.0f
                                    };
// Clear the backbuffer and depth-buffer
glClearColor( 0.0f, 0.0f, 0.5f, 1.0f );
glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );

// Set the shader program and the texture

    glUseProgram( g_hShaderProgram );

// Draw the colored triangle
         glVertexAttribPointer( g_VertexLoc, 4, GL_FLOAT, 0, 0, VertexPositions );
         glEnableVertexAttribArray( g_VertexLoc );
```

```
    glVertexAttribPointer( g_ColorLoc, 4, GL_FLOAT, 0, 0, VertexColors);
    glEnableVertexAttribArray( g_ColorLoc );

    glDrawArrays( GL_TRIANGLE_STRIP, 0, 3 );
    glDisableVertexAttribArray( g_VertexLoc );
    glDisableVertexAttribArray( g_ColorLoc );
}
```

9. After everything is done, a triangle having three colored vertices (red, green, blue) is seen, as shown in Figure 13.

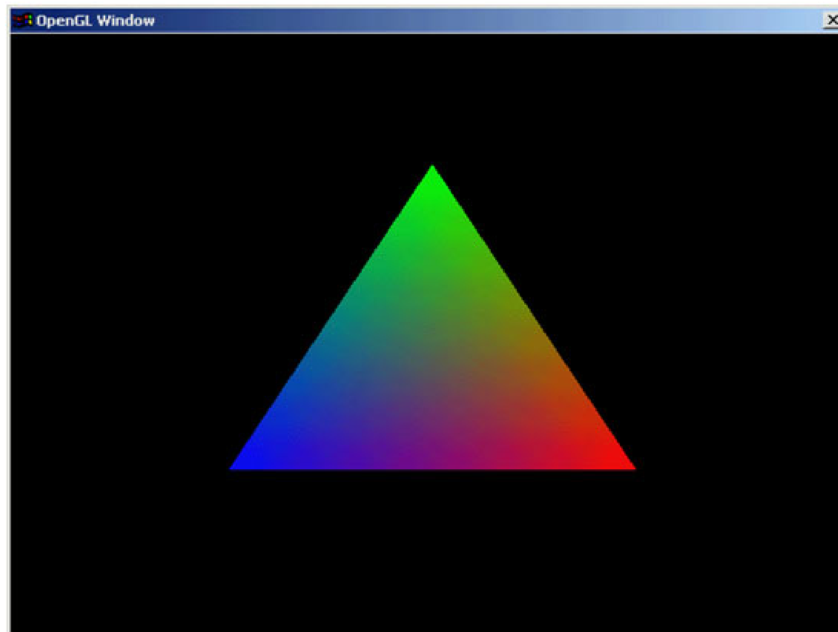

Figure 13. Three Coloured Vertices Triangle

6 Conclusion

This application note has covered some of the basics needed for 3D embedded graphics development. OpenGL ES 2.0 is a state-of-the-art program for powerful 3D graphics. With the i.MX51 multimedia processor, a user can achieve intense 3D user interfaces, games, media players, and many more.

This application note is by no means a know-it-all knowledge base. It is a starting point for all those who want to dive into the rich world of 3D graphics. Section 7, “References” has a list of references for further reading on this subject.

7 References

For more information on the OpenGL, refer to the links below:

- http://www.khronos.org/opengles/
- http://www.khronos.org/opengles/sdk/docs/man/
- http://opengles-book.com/
- http://www.khronos.org/registry/gles/specs/2.0/GLSL_ES_Specification_1.0.17.pdf

8 Revision History

Table 1 provides a revision history for this application note.

Table 1. Document Revision History

Rev. Number	Date	Substantive Change(s)
0	01/2010	Initial Release

Appendix A Developing OpenGL ES 2.0 Applications with Microsoft Visual Studio and WinCE 6.0

The procedure to develop OpenGL ES 2.0 applications with Microsoft Visual Studio 2008 and WinCE 6.0 is as follows:

1. After installing the i.MX51 BSP, open the i.MX51 BSP Solution. In the Solution Explorer tab, left-click on **Subprojects** and click on **Add New Subproject** as shown in Figure 14.

Figure 14. i.MX51 BSP Solution

2. In the wizard that appears, select **A simple Windows Embedded CE application** as shown in Figure 15.

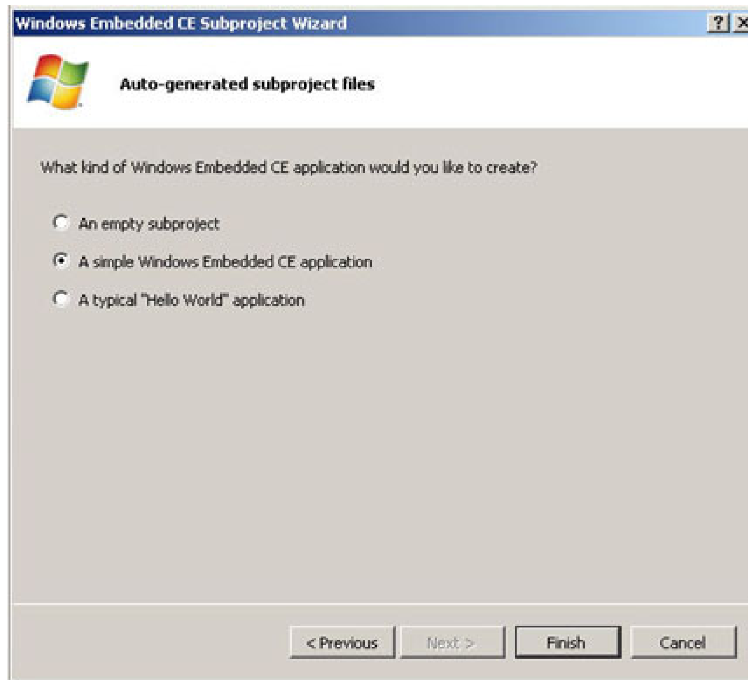

Figure 15. Windows Embedded CE Subproject Wizard Screenshot

3. Enter a name for the project in the Solution Explorer tab. Select the newly created project, right-click on it, and select **Properties** as shown in Figure 16.

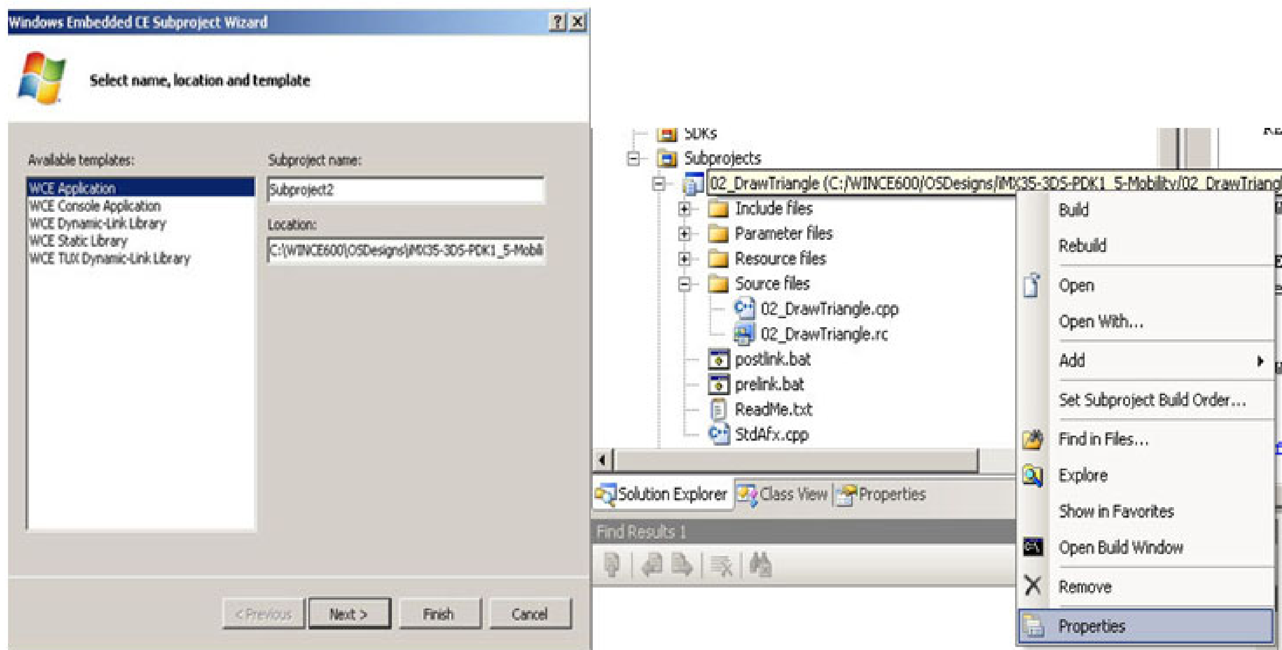

Figure 16. Selecting an Application

4. In the **C/C++** tab, select **Include Directories** and copy/paste the directory path where the OpenGL/EGL header files are located (`GL2.h, gl2ext.h, egl.h`) as shown in Figure 17.

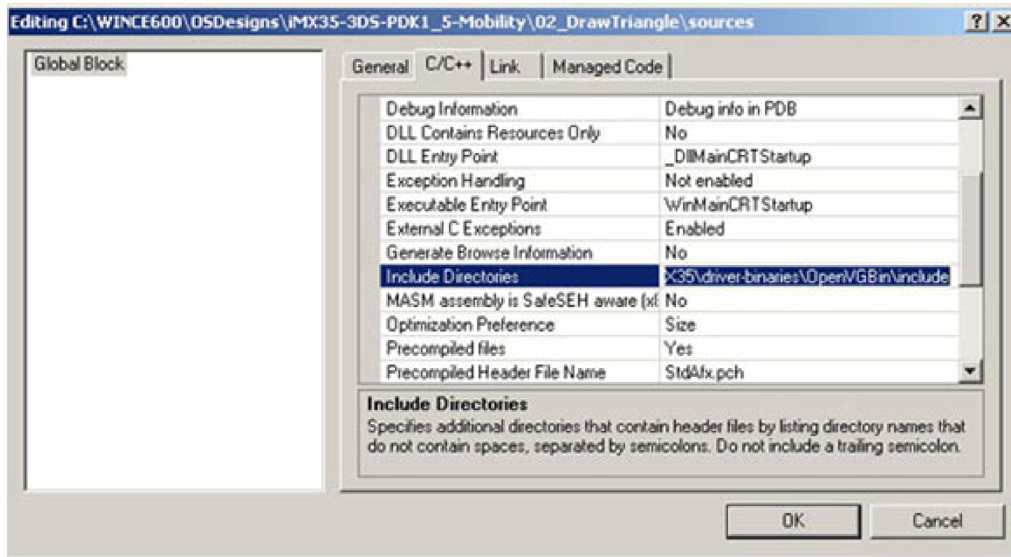

Figure 17. Wizard with C/C++ Tab

5. Click on the **Link** tab and in the **Additional Libraries** field copy/paste the directory path where the OpenGL/EGL libraries for the linker is located (`amdgslldd.lib, libEGL.lib, libgsl.lib, libgsluser.lib, libOpenVG.lib`) as shown in Figure 18.

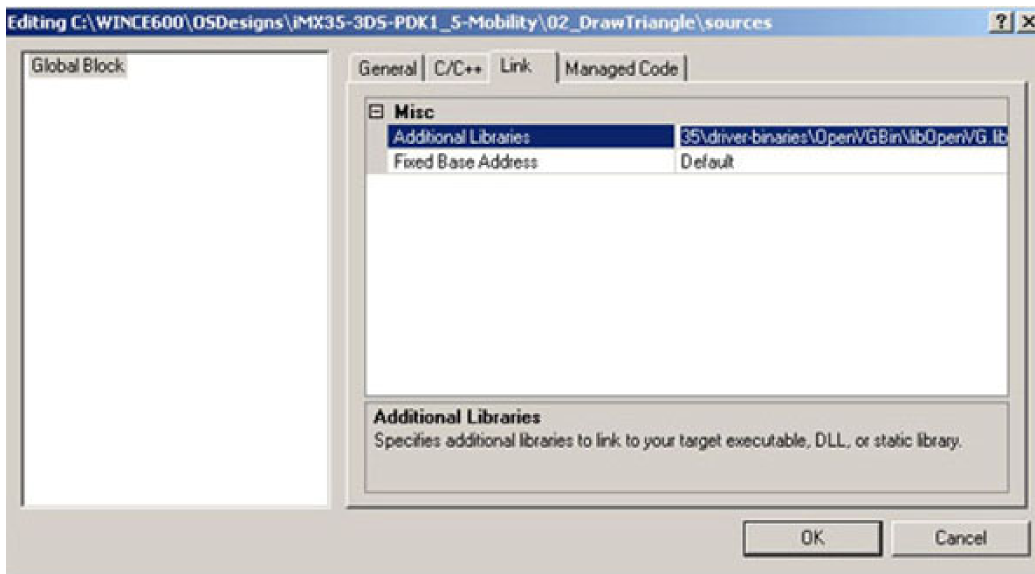

Figure 18. Wizard with the Link Tab

THIS PAGE INTENTIONALLY LEFT BLANK

How to Reach Us:

Home Page:
www.freescale.com

Web Support:
http://www.freescale.com/support

USA/Europe or Locations Not Listed:
Freescale Semiconductor, Inc.
Technical Information Center, EL516
2100 East Elliot Road
Tempe, Arizona 85284
1-800-521-6274 or
+1-480-768-2130
www.freescale.com/support

Europe, Middle East, and Africa:
Freescale Halbleiter Deutschland GmbH
Technical Information Center
Schatzbogen 7
81829 Muenchen, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
www.freescale.com/support

Japan:
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku
Tokyo 153-0064
Japan
0120 191014 or
+81 3 5437 9125
support.japan@freescale.com

Asia/Pacific:
Freescale Semiconductor China Ltd.
Exchange Building 23F
No. 118 Jianguo Road
Chaoyang District
Beijing 100022
China
+86 10 5879 8000
support.asia@freescale.com

For Literature Requests Only:
Freescale Semiconductor
Literature Distribution Center
1-800 441-2447 or
+1-303-675-2140
Fax: +1-303-675-2150
LDCForFreescaleSemiconductor
@hibbertgroup.com

Information in this document is provided solely to enable system and software implementers to use Freescale Semiconductor products. There are no express or implied copyright licenses granted hereunder to design or fabricate any integrated circuits or integrated circuits based on the information in this document.

Freescale Semiconductor reserves the right to make changes without further notice to any products herein. Freescale Semiconductor makes no warranty, representation or guarantee regarding the suitability of its products for any particular purpose, nor does Freescale Semiconductor assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. “Typical” parameters which may be provided in Freescale Semiconductor data sheets and/or specifications can and do vary in different applications and actual performance may vary over time. All operating parameters, including “Typicals” must be validated for each customer application by customer’s technical experts. Freescale Semiconductor does not convey any license under its patent rights nor the rights of others. Freescale Semiconductor products are not designed, intended, or authorized for use as components in systems intended for surgical implant into the body, or other applications intended to support or sustain life, or for any other application in which the failure of the Freescale Semiconductor product could create a situation where personal injury or death may occur. Should Buyer purchase or use Freescale Semiconductor products for any such unintended or unauthorized application, Buyer shall indemnify and hold Freescale Semiconductor and its officers, employees, subsidiaries, affiliates, and distributors harmless against all claims, costs, damages, and expenses, and reasonable attorney fees arising out of, directly or indirectly, any claim of personal injury or death associated with such unintended or unauthorized use, even if such claim alleges that Freescale Semiconductor was negligent regarding the design or manufacture of the part.

Freescale and the Freescale logo are trademarks or registered trademarks of Freescale Semiconductor, Inc. in the U.S. and other countries. All other product or service names are the property of their respective owners. ARM is the registered trademark of ARM Limited. ARM926EJ-S™ is the trademark of ARM Limited.

© Freescale Semiconductor, Inc., 2010. All rights reserved.

Document Number: AN3994
Rev. 0
01/2010