Freescale Semiconductor
Application Note

Document Number: AN4274
Rev. 0, 2/2011

High-End 3D Graphics with OpenGL ES 2.0

Three-dimensional (3D) graphics for embedded devices is common in PDAs to tablets. The i.MX53 multimedia processor offers a complete solution for the developers developing 3D graphics applications.

This application note may be used as a startup guide to understand the principles of 3D graphics and the new OpenGL ES 2.0 Application Programming Interface (API).

Contents

1 Introduction

3D graphics is a broad topic. This application note presents an overview of the 3D graphics and OpenGL ES 2.0 with its programmable pipeline.

It also covers the 3D concepts needed for application development along with a basic example, which a developer can use as a template for further development.

1.1 What is OpenGL

OpenGL is a widely adopted standard for real-time computer graphics, covering the most operating systems, and has thousands of applications.

freescale semiconductor

OpenGL allows speed and innovation by incorporating a broad set of rendering, texture mapping, 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, so the existing applications do not become obsolete.

Also OpenGL is totally portable, allowing the developers to be sure that their applications will have the same visual result on any OpenGL API-compliant hardware, not worrying about the underlying operating system. Additionally, it allows the new hardware innovations to be implemented by the extensions mechanism, enabling the vendors to 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.

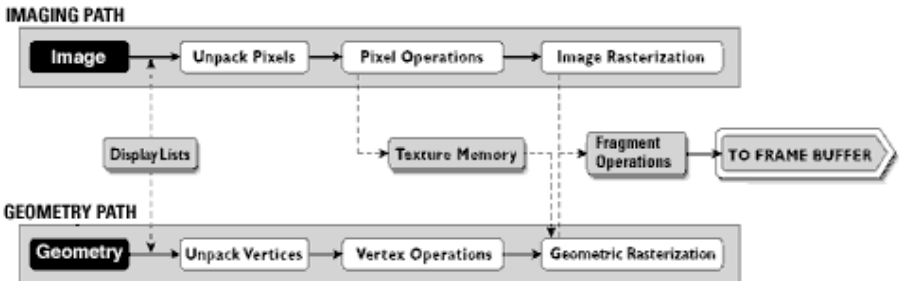

Figure 1. The OpenGL Visualization Programming Pipeline

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, even the contents of the texture memory and the frame buffer, can be 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 detail with OpenGL ES 2.0 example code, the concepts and terminology related to general 3D graphics need to be discussed. Some examples related to OpenGL ES will also be given, wherever possible.

2.1 Triangles and Lines

Vertex is the most basic concept in OpenGL. It defines a “point” in a 3D space and has 3 coordinates, x, y, and z, usually defined as float values. Given this, a triangle is constructed by the union of three vertices (v1, v2, and v3). A line is constructed by the union of only two vertices.

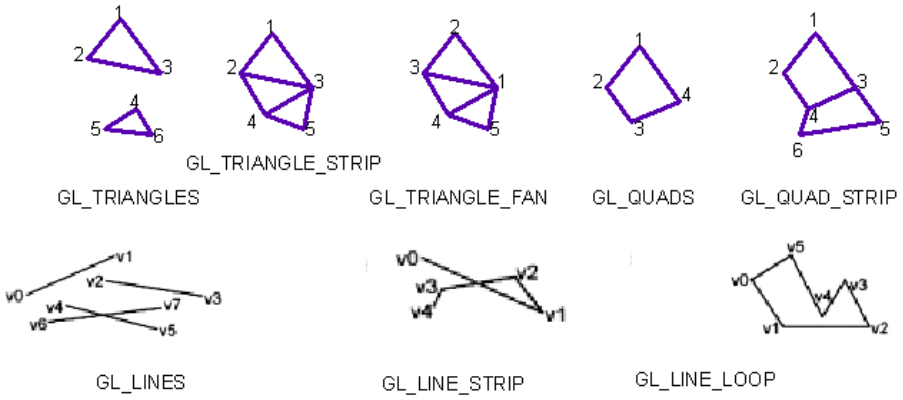

Figure 2. OpenGL and OpenGL ES 2.0

OpenGL can draw lines, triangles and quads, while OpenGL ES 2.0 can only draw and lines.

2.2 3D Meshes

A 3D model or “Mesh” is a collection of triangles, which can be represented with an array. This is called as the vertex array, usually in the form of a triangle strip.

Figure 3. 3D Mesh Model

The 3D mesh in Figure 3 can be created with a 3D modeling tool, such as Blender, 3D Studio Max, Maya, and Lightwave 3D. A 3D mesh file can be exported in many formats, optimized or non-optimized. Below is an example of how an OBJ file looks (a universal 3D mesh format).

```
####
#
#        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 5 vertices and 6 faces (triangles). The first part is the general information of the 3D mesh, which states the number of vertices and faces (triangles).

The next part, vertex list, is as follows:

```
#                    Vertex list

v -0.5 -0.5 0.5
```

The “v” character stands for vertex. This line means that a vertex is constructed by the three x, y, and z coordinates (-0.5, -0.5, 0.5).

The last part is about creating 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 indexes. This means that a “face” is constructed by three vertices (the 5th, the 2nd, and the 1st).

With all this information, any 3D model can be recreated from a “.OBJ” file. Though this file is not optimized at all, 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 (such as bmp, jpeg, png, gif, and 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.

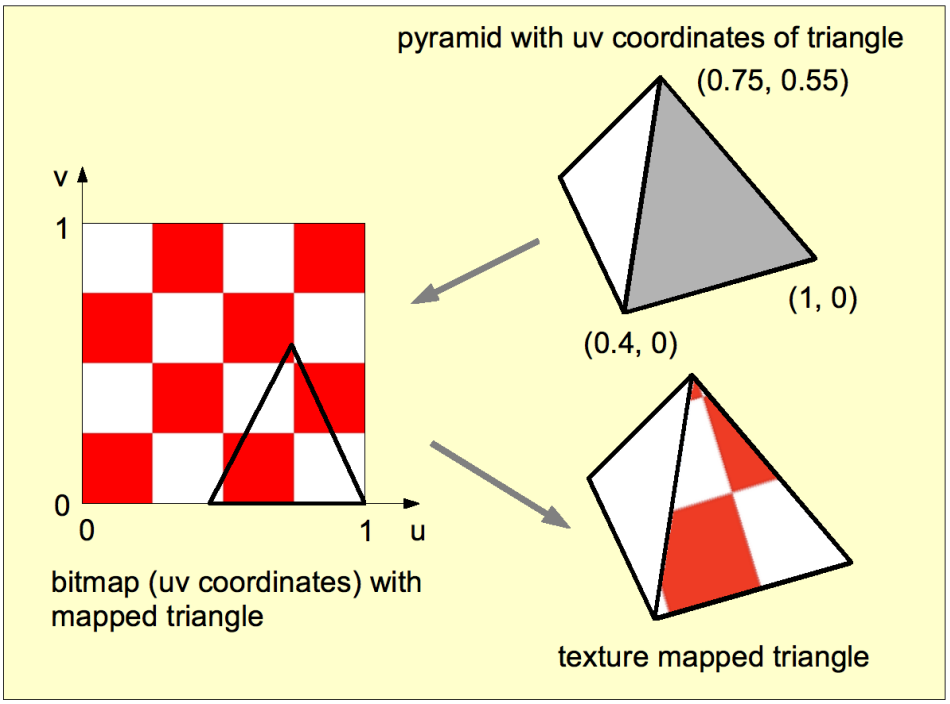

Figure 4. Textures

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

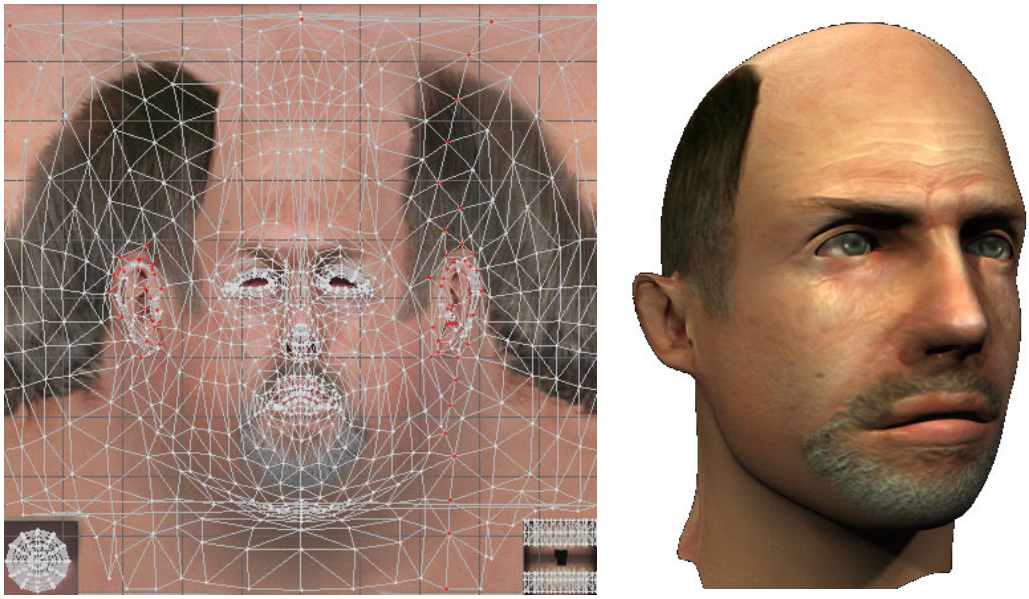

Figure 5. 3D Mesh Example

Texture coordinates are specified at each vertex of a given triangle. These coordinates are interpolated using an extended Bresenham's line algorithm. If these texture coordinates are linearly interpolated across the screen, the result will be 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.

The 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), and 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). In the parts that are farther, 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 EGL

EGL handles graphics context management, surface/buffer binding, and rendering synchronization. It enables high-performance, and 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 via other supported native rendering APIs, such as Xlib or GDI.

EGL provides:

- 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 as well as native platform rendering APIs

To the extent possible, EGL itself is independent of definitions and concepts specific to any native window system or rendering API. 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. EGL

3.1 How to Initialize EGL and Create a Rendering Context

3.1.1 Displays

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

Following are the steps required to initialize and configure EGL, and render contexts.

Step 1

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.

Names of `EGLConfig` attributes are shown below. These names may be passed to `eglChooseConfig` to specify required attribute properties.

```
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
};
```

Attributes 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 2: 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. 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. *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 a valid *dpy*.

For example:

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

The function eglBindAPI() sets a given state to the “Current Rendering API,” which in this case is OpenGL.

Step 3: 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.

```
EGLBoolean eglChooseConfig(EGLDisplay dpy, const EGLint *attrib list,
                        EGLConfig *configs, EGLint config_size,
                        EGLint *num config);
eglChooseConfig(egldisplay, s_configAttribs, &eglconfig, 1, &numconfigs);
assert(eglGetError() == EGL_SUCCESS);
assert(numconfigs == 1);
```

Step 4: Rendering Contexts and Drawing Surfaces

EGLSurfaces

One of the purposes of EGL is to provide a way to create an OpenGL 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 a native platform window should be created with attributes corresponding to the desired EGLConfig.

Using a platform-specific type (called as NativeWindowType) referring to a handle to the native window, the following function should be called:

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

The function 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, the following need to be called:

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

If eglCreateContext succeeds, it initializes the rendering context to the initial OpenGL 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 will normally be NULL or empty (first attribute is EGL_NONE).

Step 5: Bind Context and Surfaces

To make a context current, the following should be called:

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

The eglMakeCurrent function 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. The eglMakeCurrent function returns EGL_FALSE on failure. If draw or read are not compatible with *ctx*, then an EGL_BAD_MATCH error is generated.

Implementation:

```
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

Deinitializing EGL is simple.

For example:

```
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 OpenGL on a display, the following needs to be called:

```
EGLBoolean eglTerminate(EGLDisplay dpy);
```

Termination marks all EGL-specific resources associated with the specified display for deletion.

The function 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, the following function is called:

```
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 What is OpenGL ES

OpenGL ES is an application programming interface (API) for advanced 3D graphics targeted at handheld and embedded devices such as the i.MX53 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 portably binding to native windowing systems.

The OpenGL API is very large and complex and for OpenGL ES, 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, but maintaining compatibility with OpenGL was also important. As much as possible, OpenGL ES was designed such that applications that were written to the embedded subset of functionality in OpenGL would also run on OpenGL ES. The reason 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 increase the performance of shaders, precision qualifiers were introduced to the shading language.

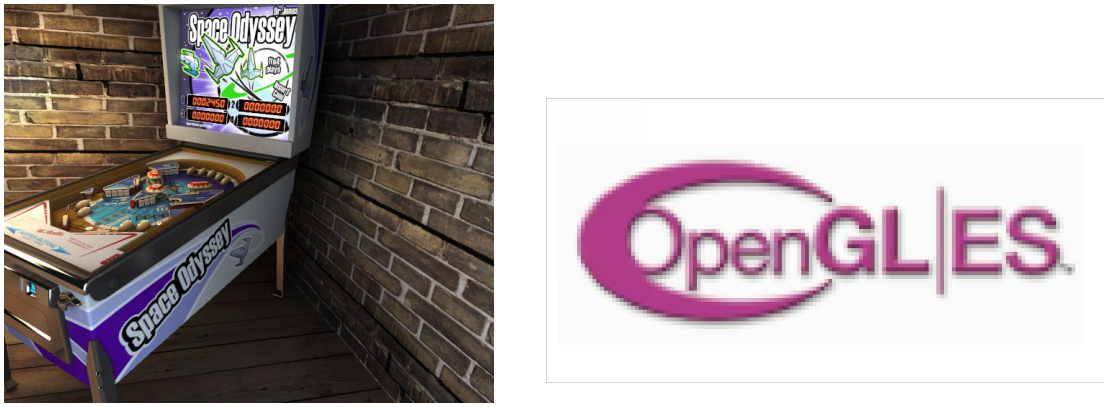

Figure 7. OpenGL ES

The OpenGL ES specifications include the definition of several profiles. Each profile is a subset of a version of the desktop OpenGL specification plus 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 so far, OpenGL ES 1.0, OpenGL ES 1.1, and 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.

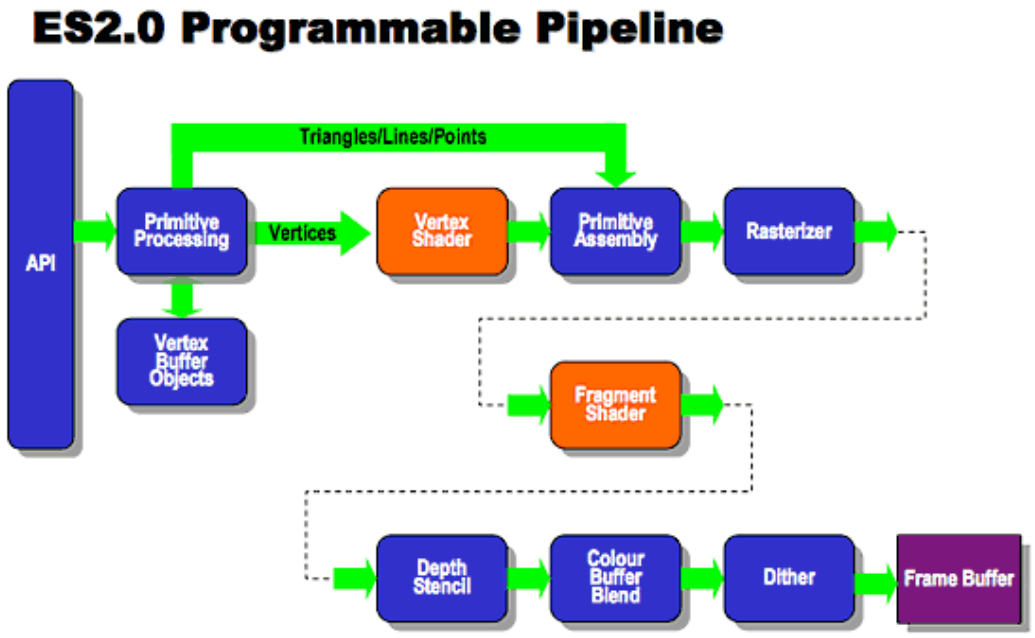

Figure 8. ES 2.0 Programmable Pipeline

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. The streamlined API from OpenGL ES 1.1 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.

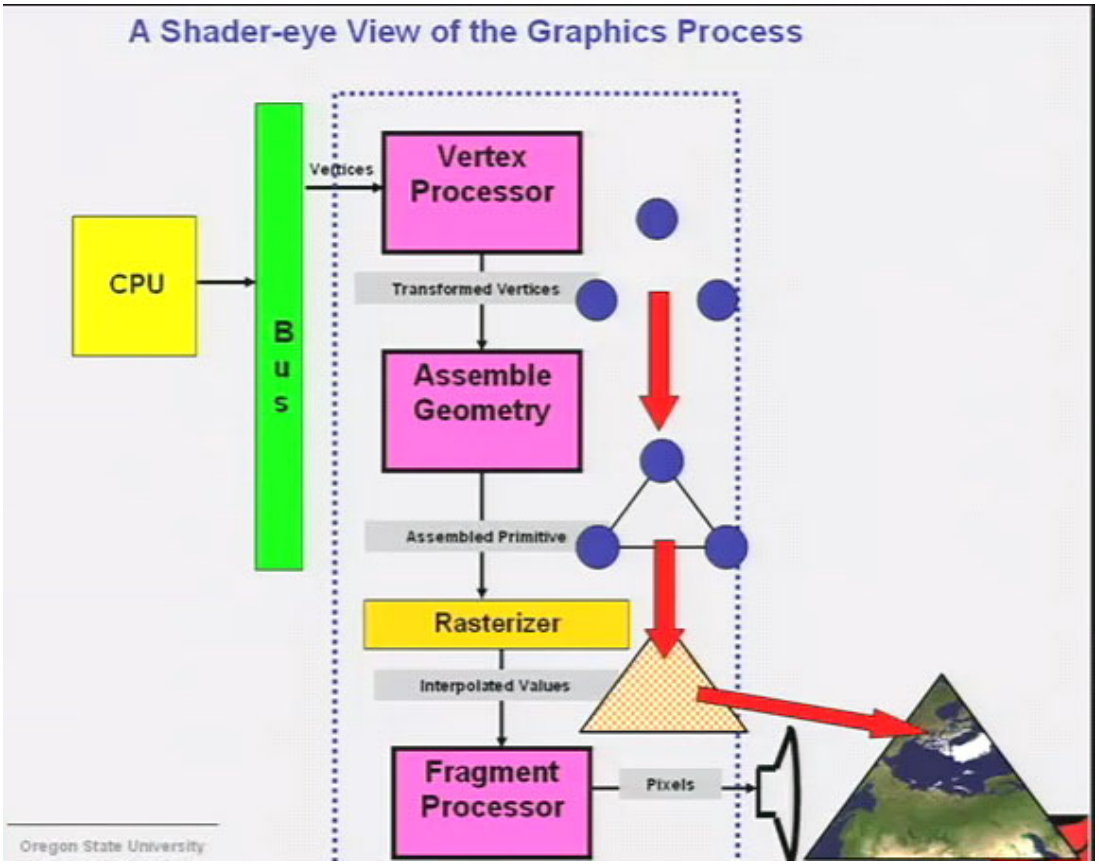

Figure 9. Programmable Graphics Pipeline

4.1 The Vertex Shader

It 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.

A vertex shader has the following inputs:

- Attributes: Consist 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 will be performed on the vertex.

4.1.1 Assemble Geometry

A primitive is a geometric object that can be drawn using appropriate drawing commands in OpenGL ES 2.0, as mentioned in the previous section. These drawing commands specify a set of vertex attributes that describes the primitive's geometry and a primitive type (triangles and 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 that 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 in 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).

4.1.2 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.

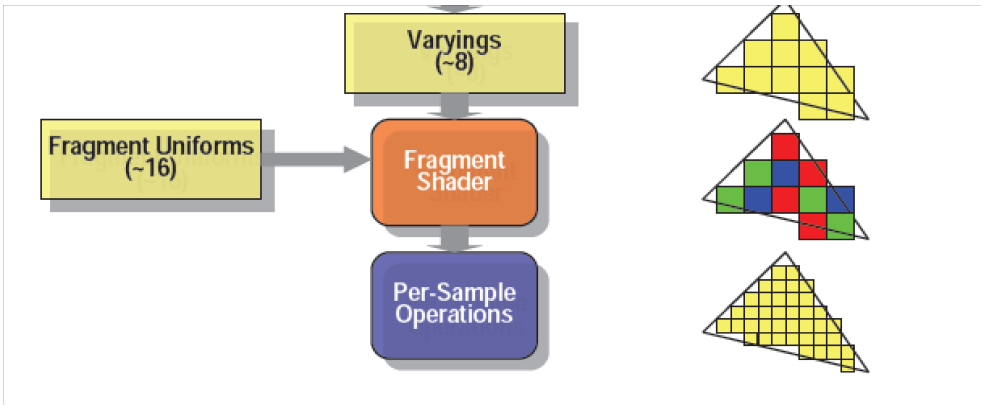

Figure 10. Rasterization

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. It 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 uniforms that represent textures used by the fragment shader.
- Shader program: Fragment shader program source code or executable that describes the operations that will be 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 screen coordinates (x,y) can only modify the pixel at location (x,y) in the framebuffer, as shown in Figure 11.

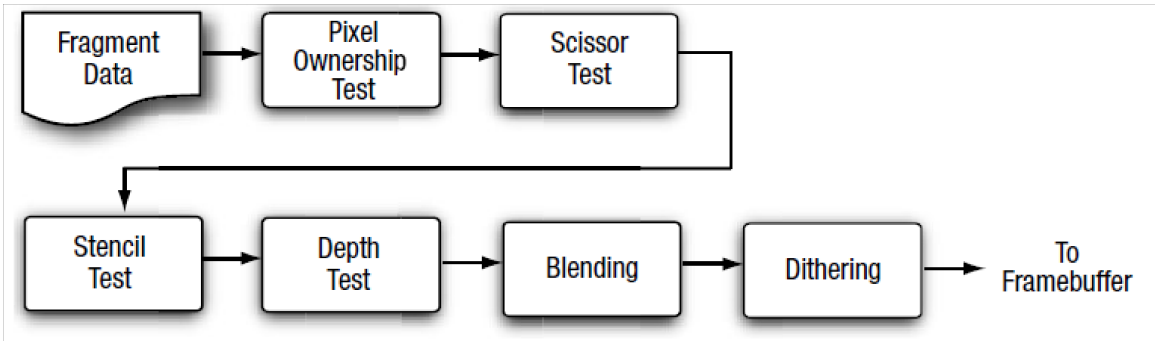

Figure 11. Fragment Shader

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 location (x,y) of the screen. The fragment color, depth, and stencil values are written depending on whether the appropriate write masks were enabled.

4.2.1 Compiling and Using the Shader

Compiling is done 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).

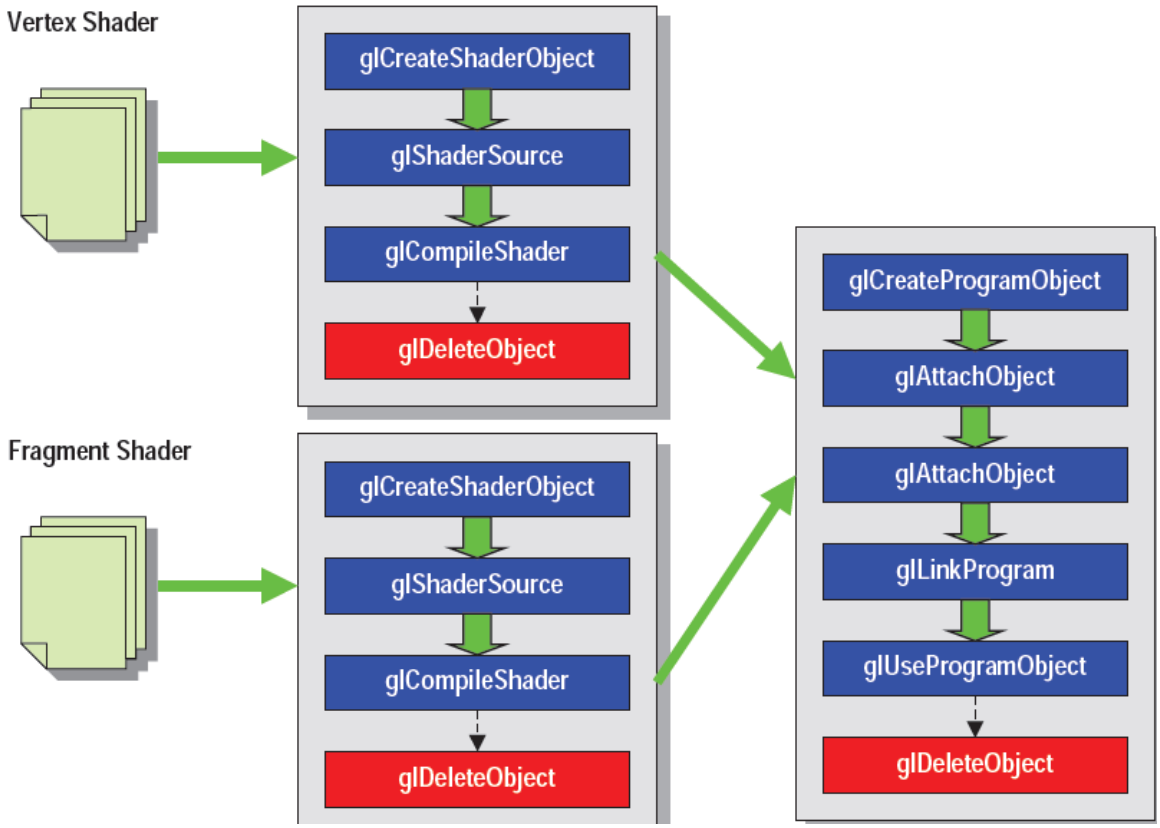

Figure 12. Shader Compiling Process

The shader object is created by using `glCreateShader`, which creates a new shader object of the type specified.

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

After this, the shader code is loaded into the object by calling `glShaderSource`. Then the shader is compiled by using `glCompileShader`.

```
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 have to be attached to a program object and linked together before drawing.

```
GLuint programObject;
glAttachObject(programObject, vertexShaderObject);
```

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

After successful linking of the program object, the program object can be finally used for the actual rendering. To use the program object for rendering, it is bound using `glUseProgram`.

5 OpenGL ES 2.0 Code Walkthrough

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

Step 1

Include the necessary header files:

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

Step 2

Define the vertex shader 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";
```

Step 3

After the shader source code, the program has to be created, the shader source has to be set, and the shader has to be compiled.

```
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 );
```

Step 4

The errors during the compiling process should be checked.

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

Step 5

The individual shaders must be attached to the shader program.

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

Step 6

The attributes must be initialized before the linking.

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

Step 7

Then it must be checked that if the linking was a success. After that individual shaders should be deleted.

```
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 );
    }
```

Step 8

Rendering loop, this should be an endless loop or a fixed number of frames depending on the application. The first part consists of the vertex data of the triangles. Then the regular openGL functions are used for clearing the screen (`glClear`). After the screen has been cleared, the shader program must be used with `glUseProgram`.

```
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 );
}
```

After everything is done, a triangle should appear (Figure 13), having three colored vertices (red, green, blue).

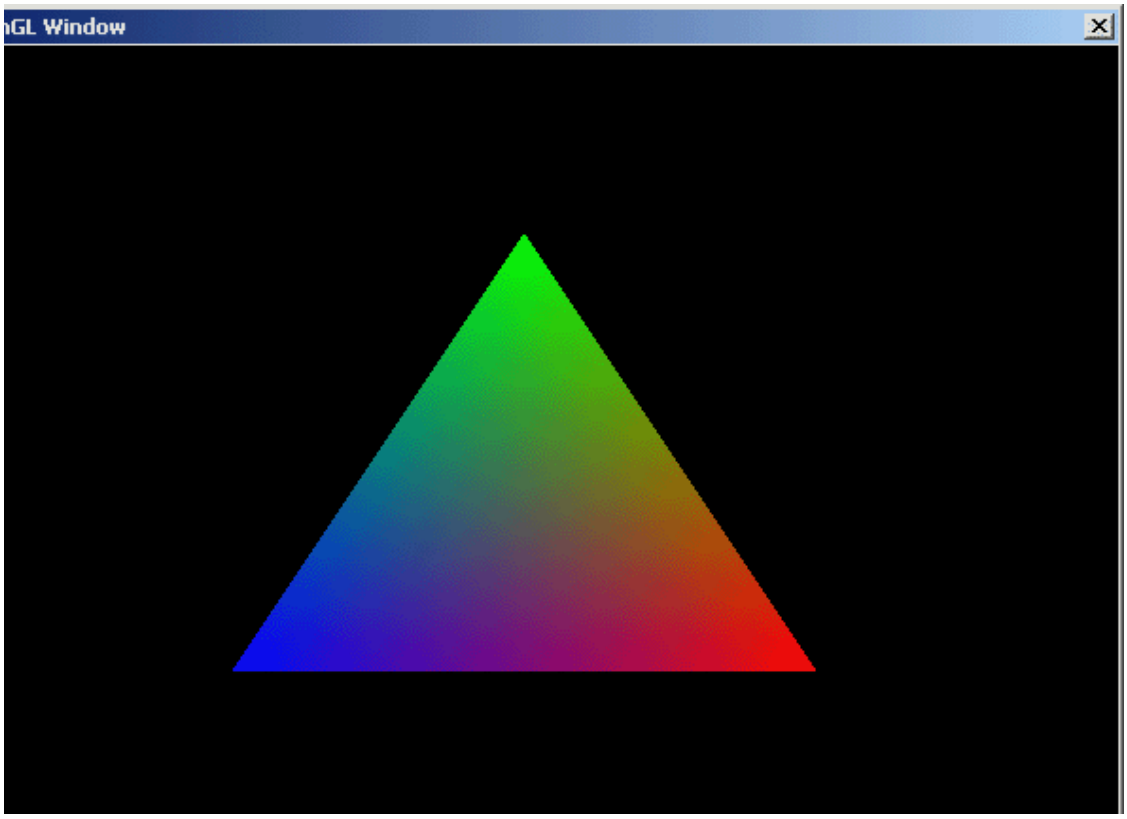

Figure 13. Triangle with Three Colored Vertices

6 Developing OpenGL ES 2.0 Applications With Microsoft Visual Studio and WinCE 6.0

Following are the steps for using Microsoft Visual Studio and WinCE 6.0. to develop OpenGL ES 2.0 Applications:

1. After installing the correct BSP, open the BSP Solution.
2. In the Solution Explorer tab, left-click on “Subprojects.”
3. Select the option “Add New Subproject.”

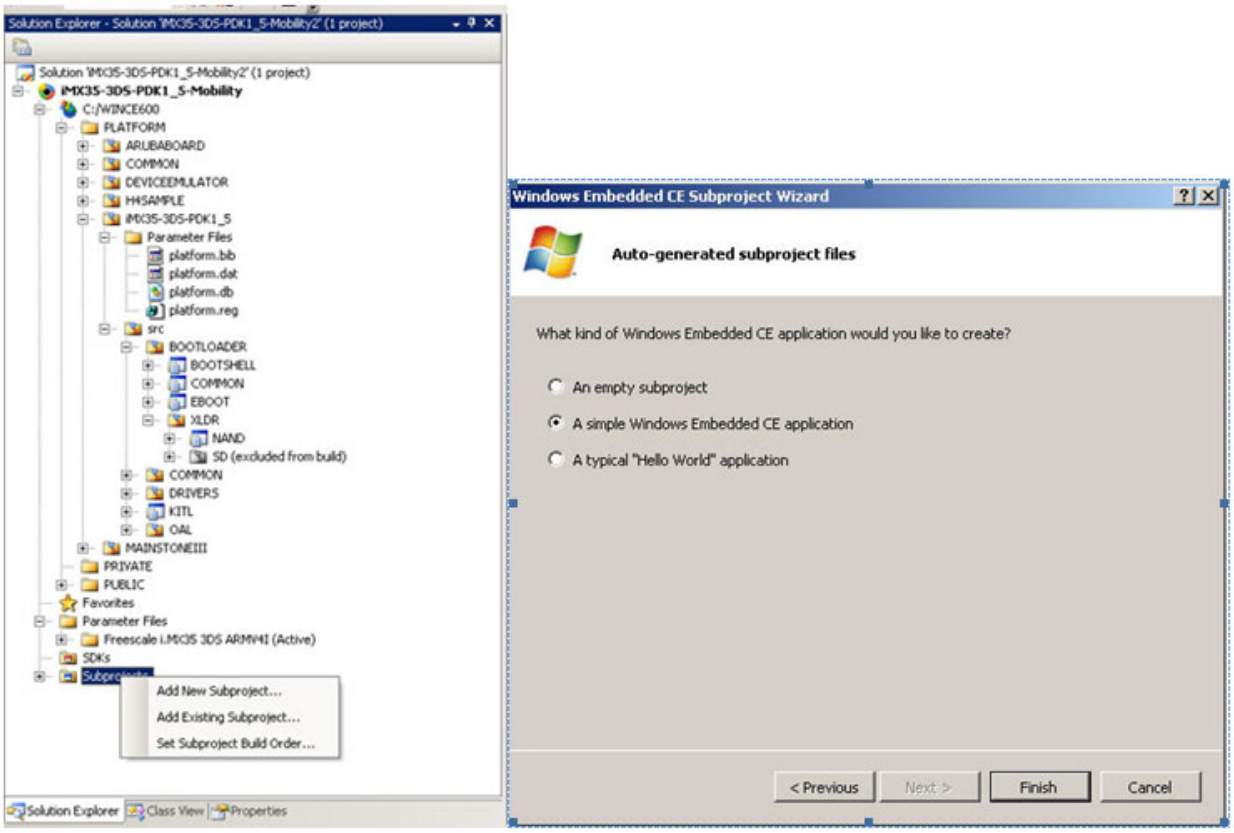

Figure 14. Windows Embedded CE Subproject Wizard

4. In the Wizard, select “A simple Windows Embedded CE application” option.

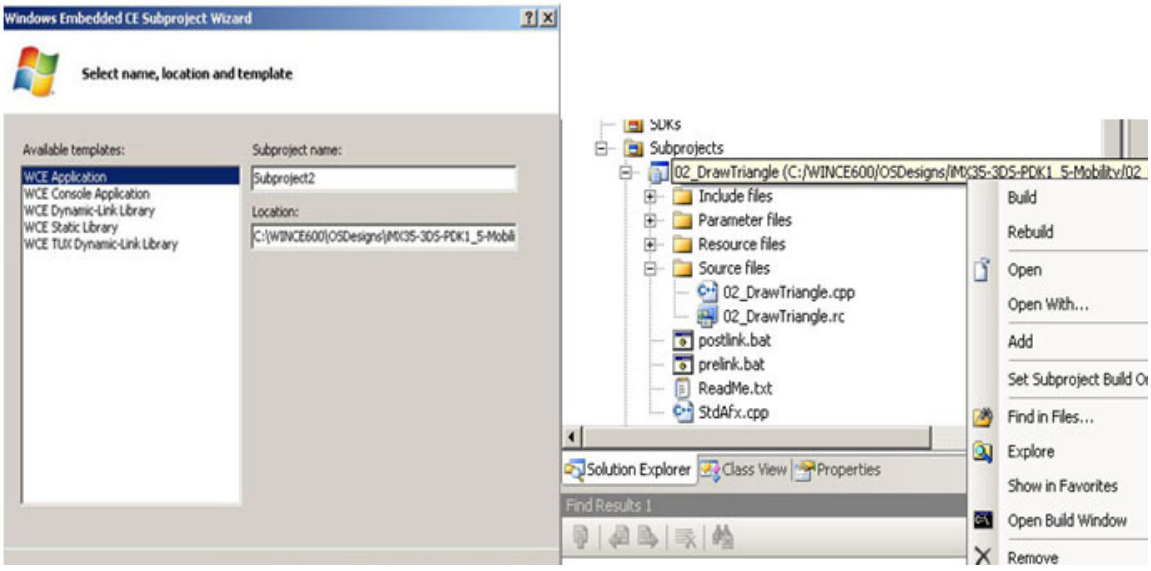

Figure 15. Solution Explorer

5. After typing a name for the project, in the Solution Explorer tab, select the newly created project
6. Right-click on it and select “Properties” menu item.

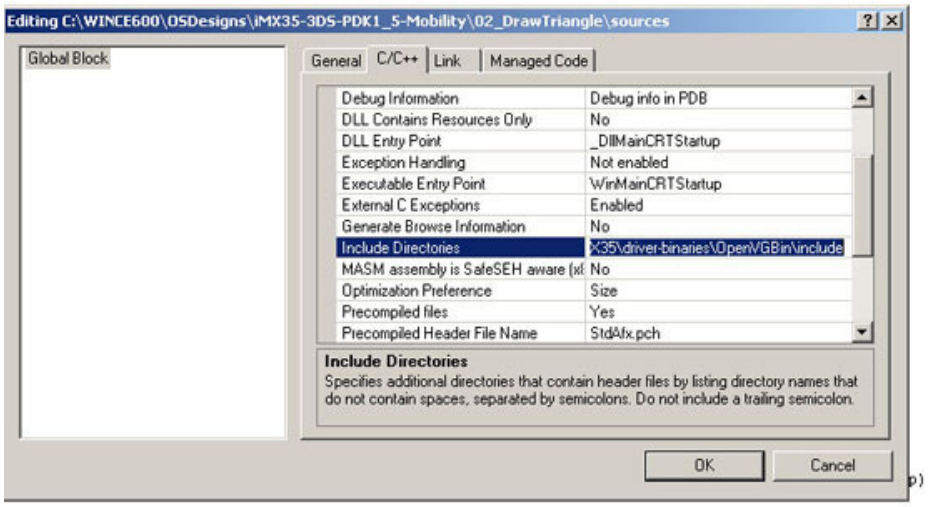

Figure 16. Selecting Sources

7. In the "C/C++" tab, select “Include Directories.”
8. Copy/paste the directory path where the OpenGL/EGL header files are located (GL2.h, gl2ext.h, egl.h).

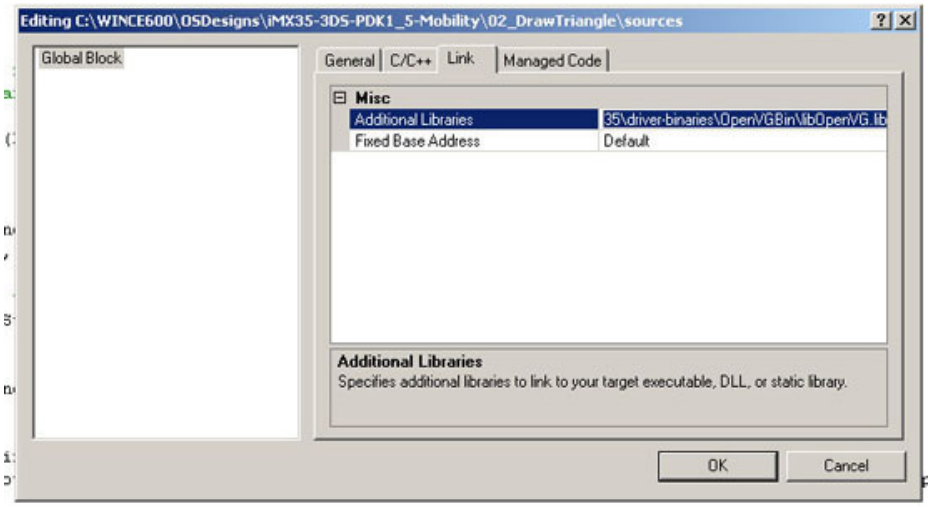

Figure 17. Linking the Library

9. Select the “Link” tab.
10. In the “Additional Libraries” field, copy/paste the directory path where the OpenGL/EGL libraries are located for the linker (amdgslldd.lib, libEGL.lib, libgsl.lib, libgsluser.lib, libGLES2.lib).

7 Conclusion

In this applicaiton note, some of the basics have been covered needed for 3D embedded graphics development. OpenGL ES 2.0 is the state-of-the-art and powerful 3D graphics programming. Using the Freescale i.MX53 multimedia processor, the user can achieve intense 3D user interfaces, games, media players and many more.

This application note by no means is a know-it-all knowledge base, it is a starting point for all of those who want to get into the rich world of 3D graphics. To know more about 3D graphics, please consider further reading. In Section 8, “References,” various links have been provided referring to OpenGL ES 2.0 programming knowledge base.

8 References

- Official OpenGL ES site
  - http://www.khronos.org/opengles/
- OpenGL Reference Pages
  - http://www.khronos.org/opengles/sdk/docs/man/
- OpenGL ES 2.0 Programming Guide
  - http://opengles-book.com/
- OpenGL SL ES Specification
  - http://www.khronos.org/registry/gles/specs/2.0/GLSL_ES_Specification_1.0.17.pdf

9 Revision History

Table 1 provides a revision history for this application note.

Table 1. Document Revision History

Rev. Number	Date	Substantive Change(s)
0	2/2011	Initial Release.

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

Document Number: AN4274
Rev. 0
2/2011