Document Number: AN3975
Rev. 1, 06/2010

i.MX35 Accelerated 2D Graphics

Optimizing 2D Graphics with OpenVG and i.MX35

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

This application note describes optimizing 2D graphics with OpenVG™ and i.MX35.

Vector graphics and flash-based content are a must in today´s fast-paced embedded world. The i.MX35 includes AMD´s z160 GPU, which specializes in the use and optimization of 2D vector-based graphics.

Contents

1. OpenVG Overview 1
2. Vector Graphics and User Interface 5
3. Running Pre-Built OpenVG Binaries in Linux 6
4. Getting Started with OpenVG and EGL 7
5. Bringing Everything Together 11
6. Developing OpenVG Applications in WinCE with Microsoft Visual Studio 2008 13
7. Conclusion 16
8. Revision History 16

1 OpenVG Overview

OpenVG is a standard created by the Khronos Group (who also created OpenGL® and OpenGL ES), for the need of a powerful low-level 2D vector graphics API. There are many applications that can take advantage of accelerated 2D graphics (for example, portable mapping and GPS applications, portable media players, high end 2D accelerated games, advanced user interfaces, and screen savers).

The OpenVG vector-based accelerated standard are used to display many formats that are in the nature of vector graphics, such as Flash, SVG, PDF, PostScript and vector fonts. This standard makes use of several functions that are low-level, similar to the OpenGL functions, but based on Bezier curves, rather than polygons.

© 2010 Freescale Semiconductor, Inc. All rights reserved.

freescale™ semiconductor

The OpenVG API uses the OpenGL API syntax wherever possible, which helps the experienced OpenGL programmers to learn OpenVG with more ease.

Unlike OpenGL, which uses triangles to simulate lines, OpenVG uses the more efficient native vector-based rendering. OpenVG uses effects such as dithering and blending that enhances its curves with anti-aliasing. OpenVG can also implement complex effects such as Gaussian blur. Font rendering is similar to OpenGL, but instead of rendering a series of triangles for the font shape, OpenVG uses the curve information to draw the fonts.

OpenVG implements a utility library called Video Graphics Unit (VGU), which contains features, such as high level geometric primitives (for example, triangles, circles) and functions for handling images and image filters.

The OpenVG pipeline consists of eight stages as shown in Figure 1. The programmer has absolute control over each stage and a better performance is achieved by using a low-level API (OpenVG).

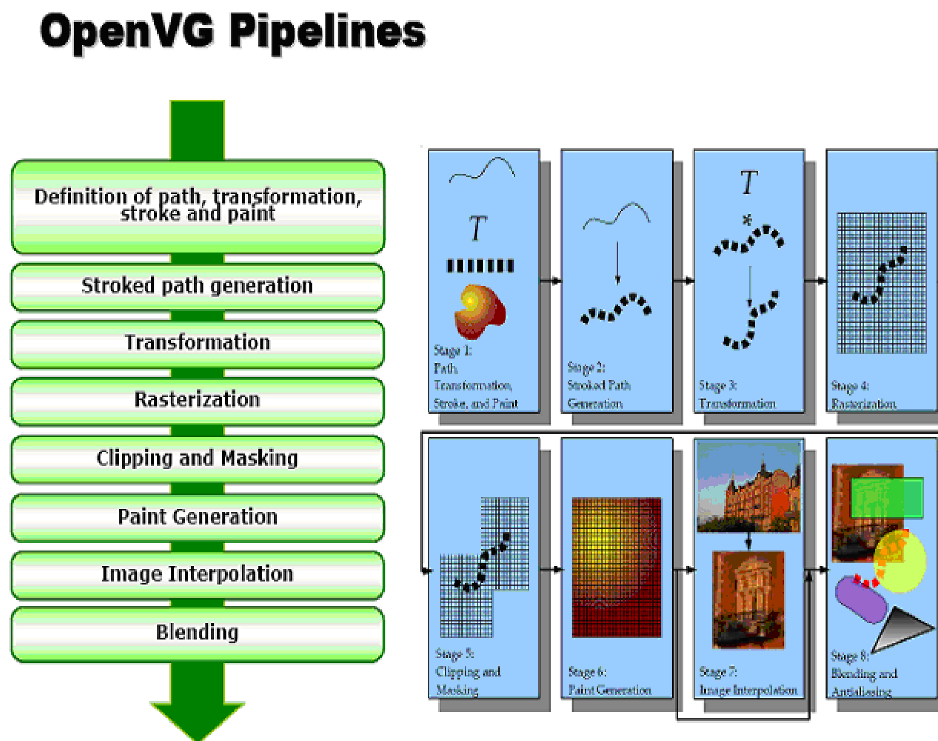

Figure 1. Stages of OpenVG Pipeline

1.1 Drawing and Storing Bitmaps

Bitmaps are drawn by assigning a color value to each pixel in a picture at a given fixed resolution.

For 24 bpp color depth:

8 bits for alpha = 32 bpp – (8 bits for R, 8 bits for G, 8 bits for B, 8 bits for A)

Bitmap is useful for photographs and pictures with many details.

Figure 2 shows an enlarged portion of a bitmap image.

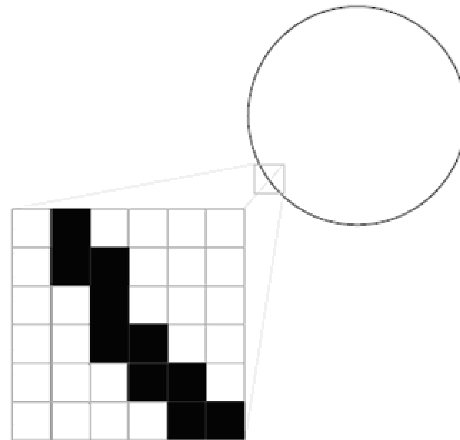

Figure 2. Enlarged Portion of a Bitmap Image

1.2 Drawing and Storing Vectors

The key points of vector graphics is that the vector graphics are drawn and stored as a mathematical vector formula. The instructions for how to get to a point are as follows:

1. Draw a line or a curve.
2. Paint the line with a stroke and apply colors.

Each vector and fill is assigned a color value. Instead of assigning color to each separate pixel, the color is assigned by segments. The color value ranges from 0b0000 (black) to 0b1111 (white).

The scene is redrawn from vector information of each frame, which is called rendering, and the rendering occurs as fast as the graphics processing unit (GPU). See *i.MX35 Graphics Core Specification* for performance numbers. This approach makes vector graphics independent of the screen resolution, because the formula and instructions are performed by each frame for any given screen size and resolution.

Another important point is that if zooming is not needed, the image stays in the screen buffer. Vector graphics data also saves memory because there is no need to store images and animation frames as bitmaps and series of bitmaps for animation. Animation is performed by interpolation between key frames.

Figure 3 shows an enlarged portion of a vector image.

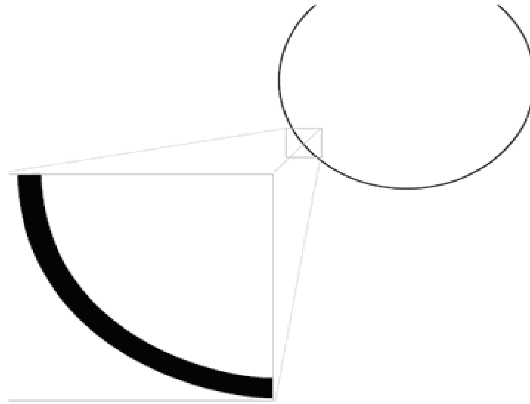

Figure 3. Enlarged Portion of a Vector Image

Figure 4 shows a comparison between a vector image and a bitmap image.

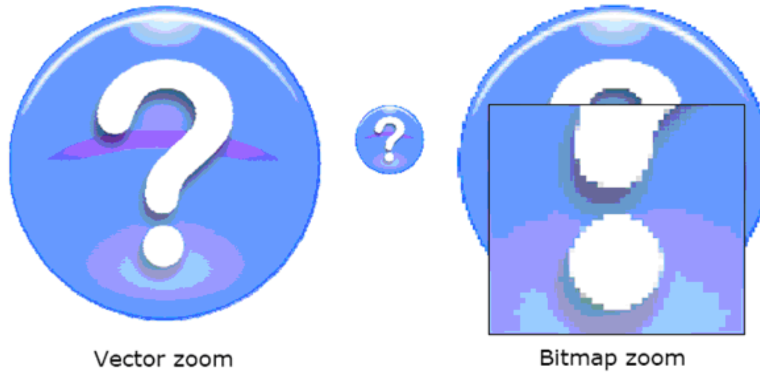

Figure 4. Comparison between Vector Image and Bitmap Image

We can avoid pixelated images and loss of quality while zooming and upscaling the images and fonts using vector zoom as shown in Figure 5.

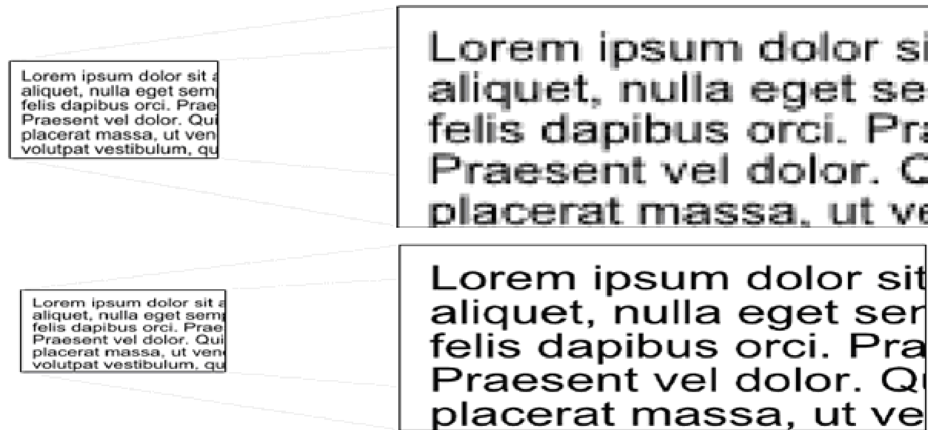

Figure 5. Comparison between Fonts with Bitmaps and Fonts with Vectors

Vectors can be created using artistic tools such as CorelDRAW® and Adobe Illustrator. This content can be converted to a set of OpenVG calls or to a convenient intermediate formats like .QVG and .SVG. AMD tools include a QVG converter and a viewer that has features such as fast vector drawing, fast matrix transformations (scale, rotate, shear, perspective), and fast alpha blend.

2 Vector Graphics and User Interface

Figure 6 shows an example of an end-user experience with OpenVG, using a WVGA display which is supported by the i.MX35. This application combines every OpenVG feature to achieve a high-impact user interface (UI).

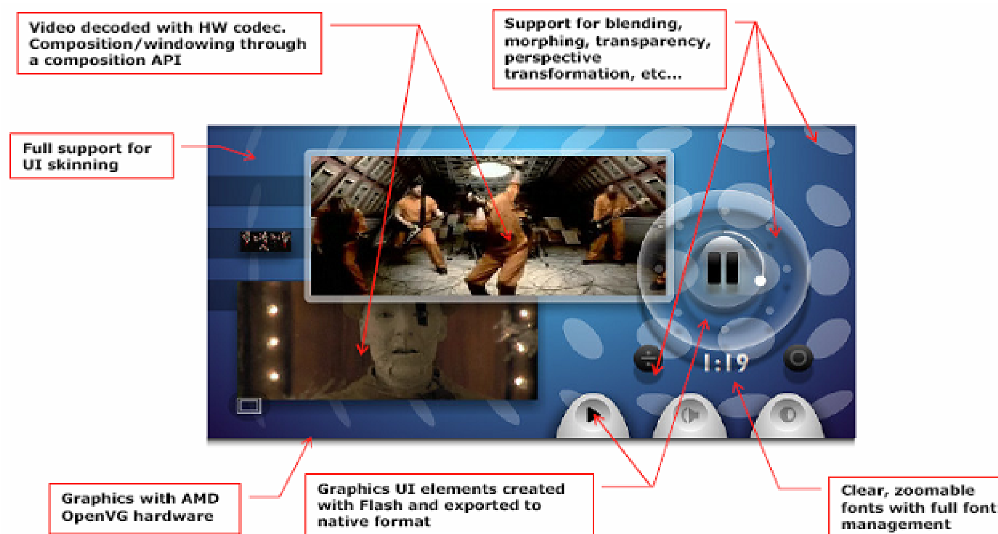

Figure 6. An End-User Experience with OpenVG

Figure 7 shows a combined good-quality UI with some multimedia features such as audio and video player, that has endless possibilities for the mass-market.

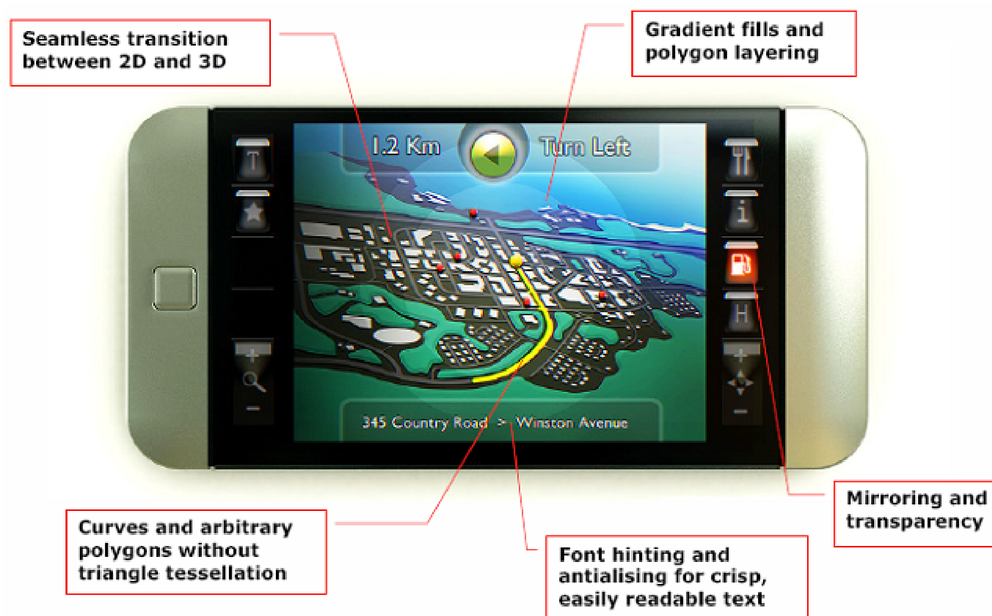

Figure 7. UI with Multimedia Features

3 Running Pre-Built OpenVG Binaries in Linux

The tiger application created by the Khronos group provides an OpenVG demo along with the OpenVG driver.

To run the tiger application complete the following steps:

1. Boot Linux® on the i.MX35 PDK and load the GPU module.

```
root@freescale$ insmod gpu_z160
```

2. Run the tigersample binary.

```
root@freescale $ ./tiger
```

A tiger image generated using OpenVG is shown on the PDK screen.

Figure 8 shows a tiger image on a PDK screen.

Figure 8. Tiger Image on the PDK Screen

4 Getting Started with OpenVG and EGL

The following sections gives an overview of Embedded-System Graphics Library (EGL) and how to initialize and deinitialize EGL™.

4.1 EGL Overview

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

EGL can be implemented on multiple operating systems (such as, embedded Linux, WinCE, and Windows®) and native window systems (X and Microsoft Windows). Implementations would also 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 to create rendering surfaces (Windows, pbuffers, pixmaps) onto which client APIs can be drawn and shared.
- Methods to create and manage graphics contexts for client APIs.
- Ways to synchronize drawing by client APIs as well as 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 9 shows the overall EGL structure and how the EGL structure is used to create graphics contexts and surfaces to higher level API´s, such as OpenVG and OpenGL ES.

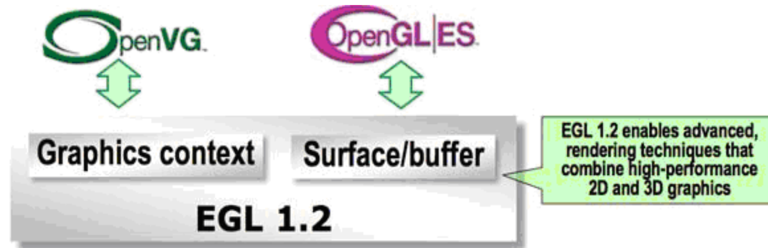

Figure 9. The EGL Structure

4.2 Initializing EGL and Creating 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.

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.

The following example shows the names of `EGLConfig` attributes. These 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
};
```

`EGL_RED_SIZE`, `EGL_GREEN_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` and `EGL_SAMPLES` is the number of samples per pixel.

The procedure to initialize EGL and create a rendering context is described below.

4.2.1 Initialization

Before calling other EGL functions, initialization must be performed once for each display. A display is obtained by calling:

```
EGLDisplay eglGetDisplay(NativeDisplayTypedisplay_id)
```

The type and format of display ID are implementation-specific, and the display ID describes a specific display provided by the system EGL. 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.

To initialize EGL on a display, call:

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

The function returns EGL_TRUE on success, and updates *major* and *minor* with the major and minor version numbers of the EGL implementation. The function does not update *major* and *minor* when they are NULL.

The function returns EGL_FALSE on failure and does not update *major* and *minor*. The function generates EGL_BAD_DISPLAY error if the *dpy* argument does not refer to a valid EGLDisplay. The function also generates an EGL NOT_INITIALIZED error if EGL is not initialized for an otherwise valid *dpy*.

A sample code is shown below:

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

In the sample code shown, `eglBindAPI()` sets a given state to the Current Rendering API which in this case is OpenVG.

4.2.2 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 as shown below:

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

A sample code is shown below:

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

4.2.3 Rendering Contexts and Drawing Surfaces

The following sections describes the process of creating EGLSurfaces and rendering contexts.

4.2.3.1 EGL Surfaces

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 (called as NativeWindowType), refer to a handle to the native window, and then call:

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

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

4.2.3.2 Rendering Contexts

To create a Rendering Context, call the following function:

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

If `eglCreateContext` function is a success, this function initializes the rendering context to the initial OpenVG state and returns its handle. The handle can be used to render to any compatible EGLSurface. Currently no attributes are recognized, therefore, `attrib_list` is NULL or empty (first attribute being EGL_NONE).

4.2.4 Binding Context and Surfaces

To make a context current, call the following function:

```
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` is 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.

The implementation is shown in the sample code below:

```
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);
```

4.3 EGL Deinitialization

Deinitializing EGL is a simple process and is shown in the example below:

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

```
assert(eglGetError() == EGL_SUCCESS);
eglReleaseThread();
```

To deinitialize, call `eglMakeCurrent()` with `EGL_NO_SURFACE` and `EGL_NO_CONTEXT`.

To release resources associated with the use of EGL and OpenVG on a display, call the following function:

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

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

The `eglTerminate` function 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 the following function:

```
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.

5 Bringing Everything Together

For a rendering loop, which has either finite or infinite number of frames, use the sample code below:

```
int main (void)
         {
         init(); //all EGL initialization code is here
         while (currentFrame < 100)
         {

             EGLint width = 0;
             EGLint height = 0;
             eglQuerySurface(egldisplay, eglsurface, EGL_WIDTH, &width);
             eglQuerySurface(egldisplay, eglsurface, EGL_HEIGHT, &height);
             render(width, height);
             currentFrame++;
         }
         deinit(); //all deInit code is here
         return 0;
     }
```

To query an attribute associated with an EGLSurface call the following function:

```
EGLBoolean eglQuerySurface(EGLDisplay dpy, EGLSurface surface, EGLint attribute,EGLint
*value);
```

In the sample code above, the arguments `&width and &height` store the values of width and height values. The width and height values are then used in the render call, where the actual drawing occurs.

5.1 Building OpenVG Applications in Linux

The procedure to build the OpenVG programs on the Linux host is as follows:

1. Specify the application name and destiny directory.

```
$export APPNAME= <user application name>
$export DESTDIR= <user destiny directory>
```

2. Specify the OpenVG program source code name

```
$export VGSOURCES= <user source code names>
```

3. Specify the Libraries location and headers location.

   The Freescale sample code must contain the OpenVG libraries and headers.

```
$export VGINCLUDE= <OpenVG Headers Directory>
$export VGLIB = <OpenVG Libraries Directory>
```

Table 1 shows the OpenVG related headers and libraries.

Table 1. OpenVG Related Headers and Libraries

OpenVG Related Headers	OpenVG Related Libraries
EGL/eglext.h	libbb2d.so
EGL/egl.h	libcsi.so
EGL/eglplatform.h	libgsl.so
VG/ext.h	libpannel2.so
VG/ftvg_utils.h	libc2d.so
VG/openvg.h	libegl13.so
VG/vgu.h	libOpenVG.so
—	libres.so

4. Build the application using the makefile provided with the FSL OpenVG sample code.

```
$make
```

5. Deploy the application.

   Copy the resultant binary contained in the destiny directory to the i.MX35 and run the destiny directory on the i.MX35 PDK.

```
root@freescale$ insmod gpu_z160
root@freescale $ ./<users application name>
```

5.2 Building Tiger Sample Application

In the Linux host, in the directory of the extracted package call the following function:

```
$export APPNAME=tigersample
$export DESTDIR=./bin
$export VGSOURCES="tiger.c main.c"
```

```
$export VGINCLUDE=./include
$export VGLIB =./lib
$make
```

Copy the resultant binary contained in the destiny directory to the i.MX35.

In the i.MX51 PDK call the following function:

```
root@freescale$ insmod gpu_z160
root@freescale $ ./<users application name
```

6 Developing OpenVG Applications in WinCE with Microsoft Visual Studio 2008

The procedure to develop OpenVG applications in WinCE with Microsoft Visual Studio 2008 is as follows:

1. Open the i.MX35 BSP Solution, in the Solution Explorer tab, left-click on **Subprojects** and click the option **Add New Subproject** as shown in Figure 10.

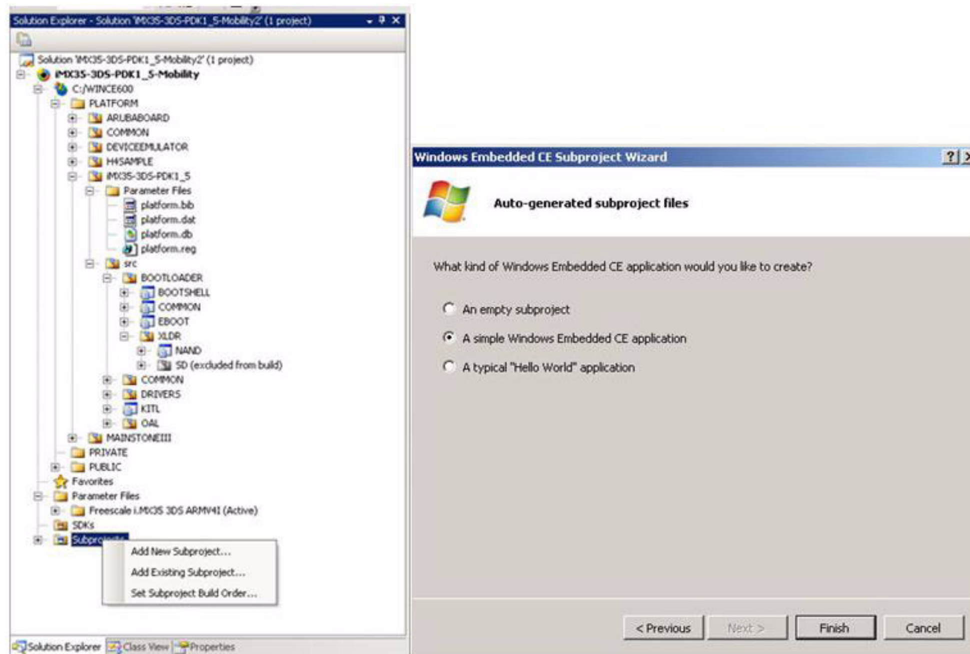

Figure 10. i.MX35 BSP Solution and Windows Embedded CE Subproject Wizard Screenshot

2. In the wizard that appears, select **A simple Windows Embedded CE application** and click **Finish** as shown in Figure 10.

3. In the wizard that appears, type the name of the project and click **Finish** as shown in Figure 11.

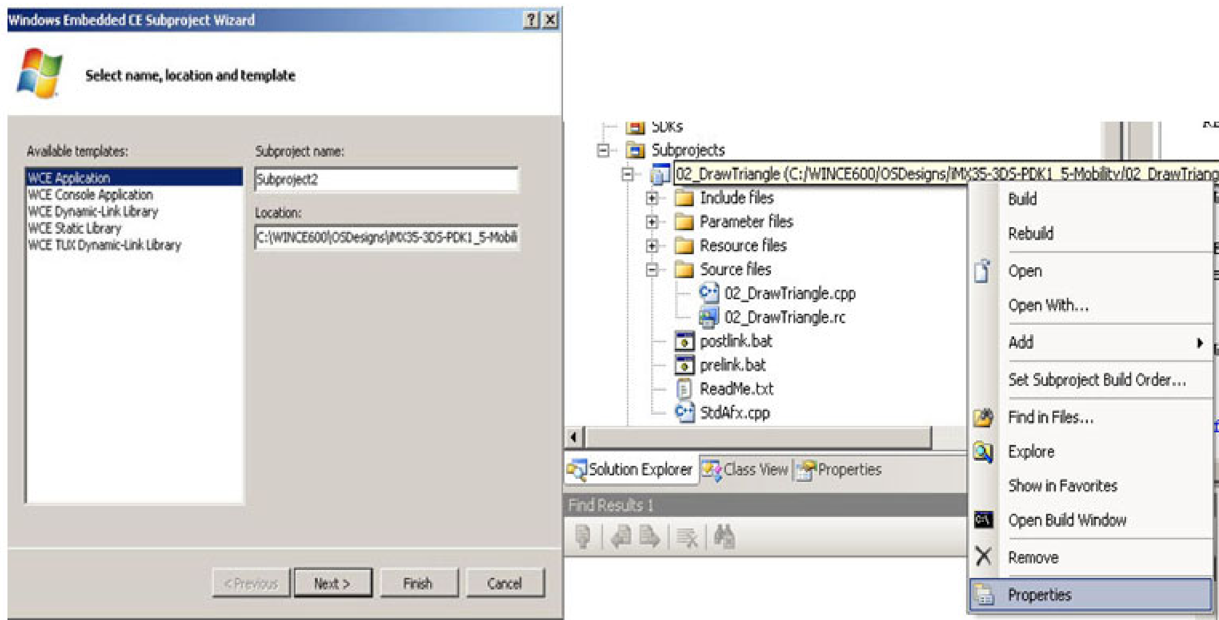

Figure 11. Selecting an Application

4. After typing a name for the project, in the Solution Explorer tab, select and right-click the newly created project, and then select **Properties** (Figure 11).
5. In the **C/C++** tab, select **Include Directories** and then, copy and paste the directory path where the VG/EGL header files are located (`openvg.h`, `vgu.h`, `egl.h`) as shown in Figure 12.

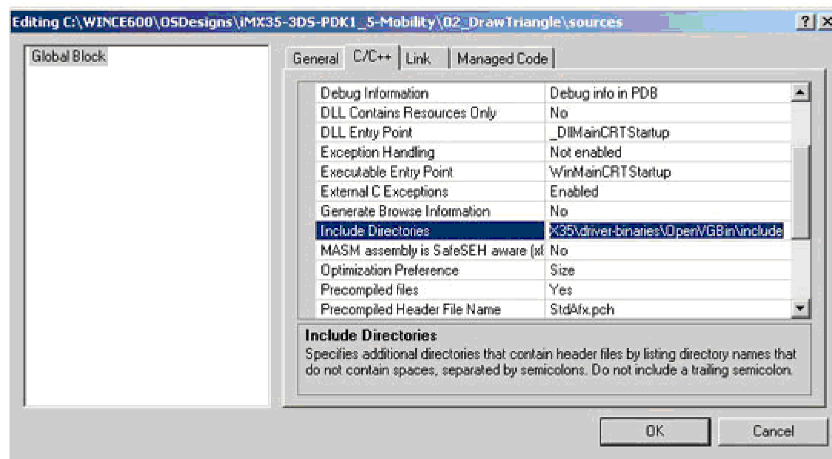

Figure 12. Wizard with C/C++ Tab

6. Select the **Link** tab and in the **Additional Libraries** field copy and paste the directory path where the OpenVG or EGL libraries for the linker is located (`amdgsllddd.lib`, `libEGL.lib`, `libgsl.lib`, `libgsluser.lib`, `libOpenVG.lib`) as shown in Figure 13.

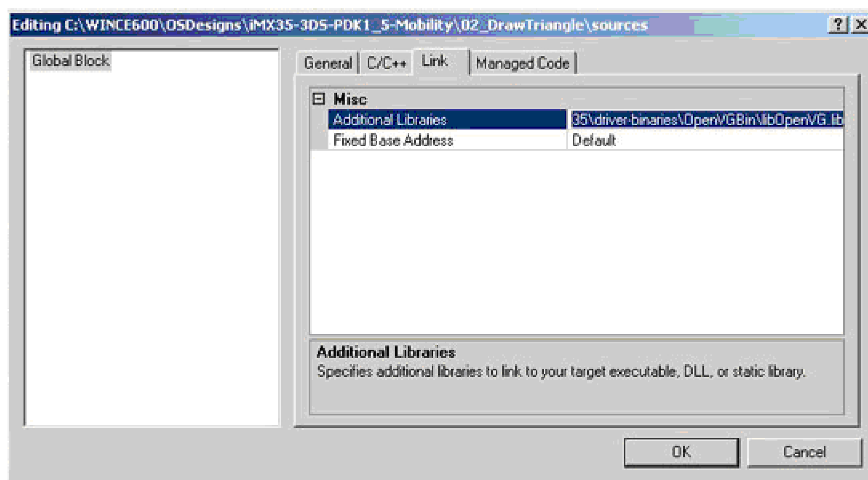

Figure 13. Wizard with the Link Tab

6.1 Initializing EGL in WinCE

To initialize EGL, open the `yourprojectName.cpp` file and follow the instructions above. The difference is that a window handler is given to the function `eglCreateWindowSurface()` as shown in the example below:

```
eglsurface = eglCreateWindowSurface (egldisplay, eglconfig, hWnd, O_RDWR), NULL);
```

6.2 Bringing Everything Together in WinCE

Finally, call the Initialize() function and Render() function in an infinite loop until the user closes the screen, this can be done in the WinMain() function as shown below:

```
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,  LPTSTR    lpCmdLine,
int       nCmdShow)
{

    MyRegisterClass(hInstance);
    if (!InitInstance (hInstance, nCmdShow))
    return FALSE;

Initialize();
     // Run the main loop until the user closes the window
     while( TRUE ) {
     MSG msg;
     if( PeekMessage( &msg, NULL, 0, 0, PM_REMOVE ) )
     {
          if( msg.message == WM_QUIT )
          return FALSE;
          }

         TranslateMessage( &msg );
          DispatchMessage( &msg );
          // Update and render the application
```

```
            Render();
            // Present the scene
            eglSwapBuffers( g_eglDisplay, g_eglSurface );
            }

        return TRUE;
    }
```

NOTE

See the `DrawTrinagle.cpp` file for a complete implementation in WinCE.

To compile the functions, right click on the project and then select **Rebuild**. This must compile and create an .exe file. Deploy the .exe to the i.MX35 PDK board.

7 Conclusion

OpenVG is a high-performance 2D API, created for embedded devices with hardware acceleration with a good set of third-party tools. OpenVG enables developers to create eye-catching and appealing GUI´s, games and Infotainment applications.

OpenVG and i.MX35 is a perfect combination for developers who want the most out of their hardware. The i.MX35 WinCE 6.0 release includes full support for Visual Studio 2005 development, allowing a fast development in a well known environment.

8 Revision History

Table 2 provides a revision history for this application note.

Table 2. Document Revision History

Rev. Number	Date	Substantive Change(s)
1	06/2010	In Section 3, "Running Pre-Built OpenVG Binaries in Linux," changed: • i.MX51 PDK to i.MX35 PDK • `./tigersample` to `./tiger`
0	01/2010	Initial Release

THIS PAGE INTENTIONALLY LEFT BLANK

THIS PAGE INTENTIONALLY LEFT BLANK

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, the Freescale logo, CodeWarrior, ColdFire, PowerQUICC, StarCore, and Symphony are trademarks of Freescale Semiconductor, Inc. Reg. U.S. Pat. & Tm. Off. CoreNet, QorIQ, QUICC Engine, and VortiQa are trademarks of Freescale Semiconductor, Inc. All other product or service names are the property of their respective owners. ARM is the registered trademark of ARM Limited.
© 2010 Freescale Semiconductor, Inc.

Document Number: AN3975
Rev. 1
06/2010