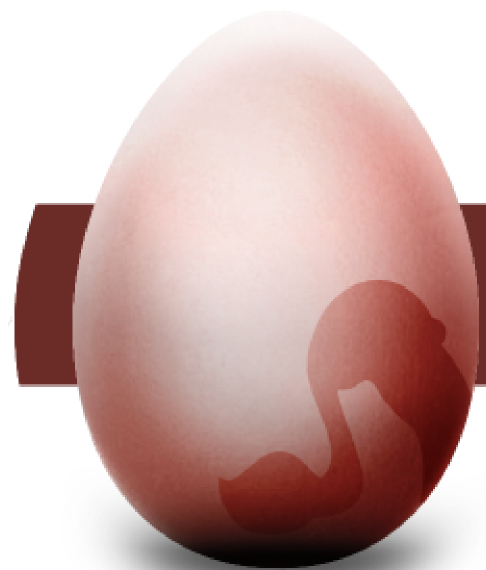
The “Making Of” picoFlamingo

An OpenGL ES 2.0 programming tutorial for OMAP3

Edition 0 Rev 1 DRAFT by DMO

June 6, 2010

Created: April, 25, 2010



picoflamingo@papermint-designs.com
http://www.papermint-designs.com/picoflamingo
http://www.papermint-designs.com/community/

.

Contents

1 Setting up your development environment **5**

1.1 Host Development System 6
1.2 Target Development System 6
1.2.1 Using your own ToolChain 7
1.2.2 Using OpenEmbedded 7
1.3 OMAP 3D Graphics Overview 7
1.3.1 Kernel Drivers 7
1.3.2 User Space Libraries 7

2 Hello Triangle **9**

2.1 Introduction 10
2.2 The Main Application 10
2.3 A Basic Render Engine 11
2.3.1 EGL initialisation 12
2.3.2 Finishing the Render Engine Interface 13
2.4 The Triangle Object 14
2.4.1 Initialising the Triangle Object 14
2.4.2 Rendering the Triangle 16
2.5 Low Level Graphical Interface 17
2.5.1 Using the Console Frame Buffer 17
2.5.2 Using the X-Windows System 17
2.6 Running the application 19
2.6.1 Running on the host PC 19
2.6.2 Running on the BeagleBoard and Zoom2 21
2.7 So far 22

Chapter 1

Setting up your development environment

Before starting to code as crazy your cool graphics stuff, you need to setup a development environment to work. As a minimum you will need to setup two environments... well, you should.

- A host environment on a normal PC for easy development.
- A target environment on your embedded system for final tweaking and testing.

In this chapter you will learn how to setup those environments... It is really easy.

1.1 Host Development System

To setup your host development system you need to install and OpenGL ES 2.0 emulator. Desktop computers supports OpenGL but, in general, they do not support OpenGL ES out of the box, so you need such an emulator to work.

We will use the Imagination Technologies SDK. To get it, you just need to register on they web and download the right package for your architecture. We are going to use a GNU/Linux platform for our development, but there are versions for other Operating Systems. The process should be very similar but we are not going to cover it.

The SDK is just a tar file with a lot of directories, including documentation, examples and other support files. All the SDK uses a helper library called PVRShell. You can use that for your development, and some stuff will be a lot easier, but in this tutorial we will cover OpenGL ES 2.0, not that library. So, at some extent, we are going to rewrite part of PVRShell during this tutorial.

What we need from the SDK is only the contents of the `Builds` directory. In that directory you will find the include files and libraries needed to compile our OpenGL ES applications.

That's it. When we compile our first application (that will happen in a while :), we will tell you how to use those files.

1.2 Target Development System

Our target systems for this tutorial will be the BeagleBoard and the OMAP Zoom II MDP. Why?, because those are the only real devices we have access to. Please, be free to donate any other device you will like to see included in this tutorial.

Setting up 3D acceleration on these devices has been for some time a bit of a pain, but at the moment of writing this text we are very lucky. The Angstrom distribution for them includes by default the libraries we need for building our applications.

So, in order to setup your target development system you just need to follow the OpenEmbedded instructions for building Angstrom for your device or use the on- line tool Narcissus to produce an image for your platform.

This will solve your run-time setup. All the libraries, header files and initialisation scripts will be in place using OpenEmbeded or Narcissus, however, you still need to solve the development stage.

For the development stage there are basically three options.

- Use OpenEmbedded for cross-compiling your applications
- Set up a different cross-compilation environment
- Install the development tools (if available) in your target platform and work directly on it.

The first option is pretty straightforward but it needs a look of free space and a lot of time to build the whole thing.

The second option is lighter (less hard drive space and compilation time) but you will need to fight some integrations issues. For this you need to select a toolchain (you can use the openembedded toolchain if you want).

The third option is quite straightforward. For the Angstrom distribution you just need to install the `native-sdk` package using the `opkg` tool.

1.2.1 Using your own ToolChain

TBC

1.2.2 Using OpenEmbedded

TBC

Here we need support from some OE hacker

1.3 OMAP 3D Graphics Overview

OMAP 3 is the application processor in computers like the BeagleBoard, the OMAP Zoom II MDP or the Gumstik Overo. The 3D graphics are provided by an SGX code inside the OMAP, that is, a part of the chip has some special circuitry for dealing with 3D graphics.

For us, as developers, the gory details of all this stuff is not that much important, but having a general overview of the architecture is always a good thing.

From our software point of view, there are two main elements we need to be aware of:

- The Kernel Drivers
- The User Space Libraries

1.3.1 Kernel Drivers

In order to use 3D/2D acceleration you need a couple of kernel drivers to talk to the SGX core in your OMAP3 processor and to access the framebuffer... that is, the screen.

Those drivers are:

- The `omaplfb`, in charge of controlling the framebuffer... I think :P
- The `pvrsrvkm` in charge of the communication with the SGX core

Both of them are open source. You can get the source code and recompile them yourself.

1.3.2 User Space Libraries

The second component is a set of user space libraries. These libraries are, in fact, the implementation of EGL, OpenGL 1.1/2.0 and OpenVG APIs.

These are the close source pieces of all the setup (not the drivers as some of us thought at the beginning of all this). Making a long history short, the driver that sends data to the SGX core is open, what is not open is the data that you need to send to make the SGX do the graphics.

Those are the libraries you need to get from Texas Instruments website. Just follow the information on the elinux wiki or check the BeagleBoard mailing list in google.

Chapter 2

Hello Triangle

Yep, we should release the picoFlamingo source code sometime ago. Sorry about that. Instead of just releasing a tarball for you to dive in, we are going to release the different steps we follow to develop the application in the hope that such an information will be useful for you.

So, let's start with the Hello Triangle application. This is a pretty silly program just to show you how to initialize the graphical system, and start playing with OpenGL ES 2.0.

2.1 Introduction

This first application is composed of three small modules that will grow in the following releases of this "Making Of..." series as we advance on the terrific world of embedded 3D graphics. These modules are:

- The main application
- The render engine
- The triangle object

There is a fourth module that actually is part of the render engine but is kept in a different file. This module is in charge of initializing the system dependant elements needed for EGL. We will describe it in detail later on.

So, let's start taking a look to the main.c file.

2.2 The Main Application

Here is the code of the whole thing:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#include "render_engine.h"
#include "pf_tri.h"

int
init_render_engine ()
{
  fprintf (stderr, "Starting Render Engine...\n");
  pf_re_init (640, 480);
  if (pf_tri_init()) exit (1);

  return 0;
}

int
render_scene ()
{
  pf_tri_render ();

  return 0;
}

int
main (int argc, char *argv[])
{
  init_render_engine ();

  while (1)
    {
      pf_re_render (render_scene);

        // Update objects states
      usleep (1000);
    }

  return 0;
}
```

At the beginning of the file we can found a couple of standard includes we need in order to use functions like fprintf, exit or usleep. Then we found the include files for the other two modules of our application: the render engine and our triangle object.

Next we found three functions. Let's move to the main function at the very end to get the big picture and then we will come back to the details in the other two.

The main function basically does the following:

- Initializes the render engine (calling `init_render_engine()`)
- Then it enters an infinite loop doing the following:
  - Call the function that render a scene frame
  - Update objects state. No state so far, we will go into this in the next chapter.
  - Finally we just wait for sometime so the triangle doesn't spin to fast. We will come back to this later to properly calculate waiting times.

Now let's take a look to the other two functions. The `init_render_engine` function just call the `pf_re_init` function to initialize the OpenGL ES stuff, and then calls to `pf_tri_init`, to initialize our triangle object.

Finally, the `render_scene` function is the actual drawing stuff. We are using an indirection level in order to keep isolated the OpenGL ES function calls from the rest of the components. As we will see soon, the `pf_re_engine` function does a couple of specific EGL/OpenGL ES things before calling our `render_scene` function.

2.3 A Basic Render Engine

The next file we have to check is the `render_engine.c`. This module is the actual EGL interface for our application. The EGL library provides the glue between the OpenGL ES libraries and the underlying graphical system. For our application and the BeagleBoard platform we will deal with two main graphical systems: the console framebuffer and the X-Windows server.

The details to set up each one of those graphical system are in the `render_engine_sys.c` file, so our render engine "object" only depends on EGL and not on the underlying graphical system.

For this very simple application, our render engine only provides four functions:

- `TestEGLError`. A helper function to show EGL errors in text format
- `pf_re_init`. The render engine initialization function
- `pf_re_end`. The render engine function call when finishing
- `pf_re_render`. The render engine function called to render each frame.

Let's take a look to each one of them:

```
int
TestEGLError(const char* err_str)
{
  EGLint err = eglGetError();

  if (err != EGL_SUCCESS)
    {
      printf ("%s failed (%d).\n", err_str, err);
      return 0;
    }
```

```
  return 1;
}
```

TestEGLError, does just that. It calls the eglGetError function that returns the latest EGL error and checks if something happens. This function should be a bit more complex providing more information depending on the actual error. For now, this is enough for us.

2.3.1 EGL initialisation

The next function to check is pf_re_init. This is the most complex function so far, however it is pretty straightforward if you read the EGL specification.

Here is the code:

```
int
pf_re_init (int w, int h)
{
  EGLint v1, v2;
  EGLint config_attr[5];
  int    n_confs;

  /* Initialise Native Graphics System */
  pf_re_native_init (&native_display, &native_window, w, h);

  /* Initialising EGL */
  eglDisplay = eglGetDisplay (native_display);

  /* Initialise EGL*/
  if (!eglInitialize (eglDisplay, &v1, &v2))
    {
      printf("Error: eglInitialize() failed.\n");
      goto cleanup;
    }
  printf ("EGL Version %d.%d\n", v1, v2);

  /* Make it current API */
  eglBindAPI (EGL_OPENGL_ES_API);
  if (!TestEGLError ("eglBindAPI"))  goto cleanup;

  /* Configuration Attributes */
  config_attr[0] = EGL_SURFACE_TYPE;
  config_attr[1] = EGL_WINDOW_BIT;
  config_attr[2] = EGL_RENDERABLE_TYPE;
  config_attr[3] = EGL_OPENGL_ES2_BIT;
  config_attr[4] = EGL_DEPTH_SIZE;
  config_attr[5] = 16;

  config_attr[6] = EGL_NONE;

  /* Find Config */
  if (!eglChooseConfig (eglDisplay, config_attr, &eglConfig, 1, &n_confs) ||
      (n_confs != 1))
    {
      printf("Error: eglChooseConfig() failed.\n");
      goto cleanup;
    }

  /* Create surface to draw */
  eglSurface = eglCreateWindowSurface(eglDisplay, eglConfig,
                                      native_window, NULL);
  if (!TestEGLError("eglCreateWindowSurface")) goto cleanup;

  /* Create Context */
  eglContext = eglCreateContext(eglDisplay, eglConfig, NULL, contex_attr);
```

```
  if (!TestEGLError("eglCreateContext")) goto cleanup;

  /* Bind Context */
  eglMakeCurrent(eglDisplay, eglSurface, eglSurface, eglContext);
  if (!TestEGLError("eglMakeCurrent")) goto cleanup;

  return 0;

cleanup:
  eglMakeCurrent(eglDisplay, EGL_NO_SURFACE, EGL_NO_SURFACE, EGL_NO_CONTEXT) ;
  eglTerminate(eglDisplay);

  return -1;
}
```

The first thing the function does is to call our system dependant render engine initialization. We will check that soon, but for now, this function is the one that obtains a display and a window for our application.

Then we need to call eglGetDisplay to tell EGL which display we want to use... the one provided by our low level functions. This function will return the EGL identifier for the display that we need to use during the initialization process.

After that, we can initialize the EGL API. The eglInitialize will return us the EGL version provided by the system and, indeed, initialise EGL.

Next we need to tell EGL which API we want to use. We do this using the eglBindAPI function. In our application we want to use OPENGL ES. Other option may be OPENVG for vector graphics.

Now we can configure EGL to serve our needs. Here we can choose if we want to use OpenGL ES 1.1 or 2.0, the resolution of our DEPTH buffer and other parameters. Please refer to the EGL specification for a complete list.

Finally, using this configuration we create a window surface and a context. That is all what we need. After getting those last parameters we've got our EGL context and we can start using it after calling the eglMakeCurrentContext.

At this point, our interface between OpenGL ES and the underlying graphical system is set up and we can forget about it... almost for the simplest applications.

2.3.2 Finishing the Render Engine Interface

The `pf_re_end` just frees resources. Release the current context and terminates EGL. We also call our low level graphical interface to do any further required operation as, for instance, destroying our window.

```
int pf_re_end ()
{
  eglMakeCurrent (eglDisplay, EGL_NO_SURFACE, EGL_NO_SURFACE, EGL_NO_CONTEXT) ;
  eglTerminate (eglDisplay);

  pf_re_native_end ();

  return 0;
}
```

Finally the `pf_re_render` function is in charge of render every single frame for our application. It takes as a parameter a function. If you remember our brief discussion about `main.c` we said that we tried to separate all the EGL related functions from the main application.

```
int
pf_re_render (RENDER_SCENE_FUNC func)
{
```

```
  // Process Render mode
  glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

  func ();

  eglSwapBuffers(eglDisplay, eglSurface);

  return 0;
}
```

Looking at `pf_re_render` you see that we call glClear at the beginning, to clean the colour and depth buffer (basically to reset our render buffers), then we call the function to draw whatever we want, and finally we call eglSwapBuffer that is the function that puts the stuff in the screen.

2.4 The Triangle Object

Our triangle object is defined in the file `pf_tri.c`. This module provides two functions:

`pf_tri_init` and `pf_tri_render`.

The first function (`pf_tri`) sets up the shaders we need to actually draw something with OpenGL ES 2.0. Remember that OpenGL ES 2.0 does not provide a fixed-pipeline and therefore we have to provide it ourselves.

At the very beginning of the file those shaders are defined as character strings:

```
// Fragment and vertex shaders code
const char* fshader_src = "\
  void main (void)          \
  {                                          \
      gl_FragColor = vec4(1.0, 1.0, 1.0 ,1.0);\
  }";

const char* vshader_src = "\
  attribute highp vec4    myVertex;\
  void main(void)        \
  {\
    gl_Position = myVertex;\
  }";
```

This is the very minimum set of shaders you can write to render something in the screen.

The vertex shader just says that the position for every vertex is the vertex itself.

The fragment shader is as simple as the vertex shader. It just assigns the white color (RGB 1,1,1) to every single fragment, that is, every single pixel that needs to be drawn. We will come back to the shader later in this series, for now; just believe that this is the very minimum you need to see something on to the screen.

2.4.1 Initialising the Triangle Object

Even for this very simple example we need to initialize several things. This is all done in the `pf_tri_init` function. A lot of code here will be moved to a more appropriated place, but for now, this makes things easier and more compact.

```
int
pf_tri_init ()
{
  GLuint frag_shader, vert_shader;
  GLint  compile_ok, link_ok;
  int    error = 0;
```

```
  // Create the fragment shader object
  frag_shader = glCreateShader(GL_FRAGMENT_SHADER);

  // Load the source code into it
  glShaderSource (frag_shader, 1, (const char**)&fshader_src, NULL);

  // Compile the source code and check results
  glCompileShader (frag_shader);
  glGetShaderiv (frag_shader, GL_COMPILE_STATUS, &compile_ok);
  if (!compile_ok)
    {
      show_log (frag_shader, "Failed to compile fragment shader");
      error++;
    }

  // Loads the vertex shader in the same way
  vert_shader = glCreateShader(GL_VERTEX_SHADER);

  glShaderSource (vert_shader, 1, (const char**)&vshader_src, NULL);
  glCompileShader (vert_shader);

  glGetShaderiv (vert_shader, GL_COMPILE_STATUS, &compile_ok);
  if (!compile_ok)
    {
      show_log (vert_shader, "Failed to compile vertex shader");
      error++;
    }

  /* Create the shader program with just created fragment and vertex
     shaders. Also bind uniforms */
   prg = glCreateProgram ();

   glAttachShader(prg, frag_shader);
   glAttachShader(prg, vert_shader);

  // Bind the custom vertex attribute "myVertex" to location VERTEX_ARRAY
  glBindAttribLocation (prg, VERTEX_ARRAY, "myVertex");

  // Link the program and check results
  glLinkProgram(prg);

  glGetProgramiv(prg, GL_LINK_STATUS, &link_ok);
  if (!link_ok)
    {
      show_log (prg, "Failed to link program");
      error++;
    }

  return error;
}
```

The initialisation code just compiles the two shaders and attaches them to our GPU program, the code the SGX will use to render our triangle. The process is very straightforward, and for OpenGL ES 2.0 you always need to provide almost one program (more or less the code we showed above).

The last element needed to complete our initialization code is the `show_log` function. This function is the one that will report any problem with our shaders... in other words, the compilation errors. Here it is:

```
void
show_log (GLuint handler, char *str)
{
  int log_len, n_log;
  char* log_txt;
```

```
  glGetProgramiv (handler, GL_INFO_LOG_LENGTH, &log_len);

  log_txt = malloc (log_len);

  glGetShaderInfoLog(handler, log_len,
                     &n_log, log_txt);
  printf("%s:\n%s\n", str, log_txt);
  printf ("--------------------------------------------------\n");
  free (log_txt);
}
```

2.4.2 Rendering the Triangle

Finally we have everything we need to render our triangle. Let's take a look to the code and then go through the different parts details.

```
int
pf_tri_render ()
{
  // Actually use the created program
  glUseProgram (prg);

  glClear (GL_COLOR_BUFFER_BIT);
  if (!TestEGLError ("glClear")) return -1;

  // Triangle Vertexs position (x,y,z)
  GLfloat pfVertices[] = {-0.4f, -0.4f, 0.0f,
                          +0.4f, -0.4f, 0.0f,
                           0.0f,  0.4f, 0.0f
  };

  glEnableVertexAttribArray (VERTEX_ARRAY);
  glVertexAttribPointer (VERTEX_ARRAY, 3, GL_FLOAT, GL_FALSE, 0, pfVertices);

  glDrawArrays (GL_TRIANGLES, 0, 3);
  if (!TestEGLError("glDrawArrays")) return -1;

  return 0;
}
```

The rendering code is also very simple. The first thing we do is to tell OpenGL ES 2.0 that we want to use the prg program. This is the program we compiled and linked in the `pf_init` function.

Then, we clear the screen and we tell OpenGL ES where to find the vertex of our graphic.

- First of all, we enable the array 0 (that's the value of `VERTEX_ARRAY`) as an array attribute. That means that, for each vertex we send to our vertex shader, there will be almost an attribute. In our simple code, that attribute is the 3D coordinates of the vertex. We will see very soon other common attributes.

- Second we just tell the system where to find the data associated to the vertex attribute 0. This is just a pointer to an array containing the coordinates of the three vertexes we need to provide to draw a triangle.

  Here we also define the data type and how many components we need for each vertex attribute. In this case we need three floats: one for X, one for Y and one for Z.

Now that all the information is configured, we just need to tell OpenGL ES to draw our data. This is done with the `glDrawArrays` function. This function receives three parameters:

- The graphical primitive we want to use. In this example we are using `GL_TRIANGLES`. This primitive will render the triangle defined by every set of three vertexes it found in the vertex attribute.
- Then we indicate the starting item in our vertex array, the index from which we want to start drawing. In this example is 0, as we are going to render the whole set of vertexes. Later we will see some examples where this parameter will make more sense.
- The last parameter is the number of vertex to be render. For our triangle we need three of them.

At this point we have provided all the information to the system and it will do the job. We just need to be sure to call `eglSwapBuffers` to see our data in the screen.

2.5 Low Level Graphical Interface

The last piece of code we need to run our application is the low level interface to the graphical system. Here we have several options that will depend on the specific device we are using.

For the BeagleBoard and the Zoom2 devices we basically have two options.

2.5.1 Using the Console Frame Buffer

Using the console frame buffer is really straightforward. This is how our system dependant render engine code will look like.

```
#include <EGL/egl.h>
#include <GLES2/gl2.h>

#include <render_engine.h>

int
pf_re_native_init (EGLNativeDisplayType *display_out,
                   EGLNativeWindowType *window_out,
                   int w, int h)
{
  display_out = 0;
  window_out = 0;
}

int
pf_re_native_end ()
{

  return 0;
}
```

In console mode there is no display and there is no window, so just return 0 for these two parameters and we are done. We will extend this a little bit in next chapters, but this is basically what you need to do.

2.5.2 Using the X-Windows System

Using the X-Windows system will allow us to run our 3D applications within a window when running X-Windows. This code basically creates a window as usual. In this process, the display and window handlers are acquired, and that is what EGL needs in order to setup the connections between our just create window and the OpenGL ES API.

Note that, when using the emulator on a PC, you need to use this function (almost on a GNU/Linux system) to run your application in a window. Maybe you can setup a framebuffer, but we have not tried that.

The first part of the code adds the needed included files and some variables. Here it is

```
#include <stdio.h>
#include <stdlib.h>

#include "X11/Xlib.h"
#include "X11/Xutil.h"

#include <EGL/egl.h>
#include <GLES2/gl2.h>

#include <render_engine.h>
// Constante
#define WINDOW_WIDTH     640
#define WINDOW_HEIGHT    480

static   Window    x11Window    = 0;
static   Display*  x11Display   = 0;
static   Colormap  x11Colormap  = 0;
```

Then we can open the display and create our windows. This is how to do it using the XLib API.

```
int
pf_re_native_init (EGLNativeDisplayType *X11Display_out,
                   EGLNativeWindowType *X11Window_out, int w, int h)
{
  // X11 variables
  long           x11Screen = 0;
  XVisualInfo*   x11Visual = 0;


  Window                  root_window;
  XSetWindowAttributes    w_attrb;
  unsigned int            ev_mask;
  int                     color_depth;

  // Initializes the display and screen
  x11Display = XOpenDisplay (0);
  if (!x11Display)
    {
      printf("Error: Unable to open X display\n");
      goto cleanup;
    }
  x11Screen = XDefaultScreen (x11Display );

  // Gets the window parameters
  root_window = RootWindow (x11Display, x11Screen);
  color_depth = DefaultDepth (x11Display, x11Screen);

  x11Visual = malloc (sizeof(XVisualInfo));
  XMatchVisualInfo (x11Display, x11Screen,
                    color_depth, TrueColor, x11Visual);

  if (!x11Visual)
    {
      printf("Error: Unable to acquire visual\n");
      goto cleanup;
    }

  x11Colormap = XCreateColormap (x11Display, root_window,
                                 x11Visual->visual, AllocNone );
  w_attrb.colormap = x11Colormap;
```

```
  // Add to these for handling other events
  w_attrb.event_mask = StructureNotifyMask | ExposureMask
    | ButtonPressMask | ButtonReleaseMask | KeyPressMask | KeyReleaseMask;
  ev_mask = CWBackPixel | CWBorderPixel | CWEventMask | CWColormap;

  // Creates the X11 window
  x11Window = XCreateWindow (x11Display, RootWindow (x11Display, x11Screen),
                             0, 0, w, h,
                             0, CopyFromParent, InputOutput,
                             CopyFromParent, ev_mask, &w_attrb);
  XMapWindow (x11Display, x11Window);
  XFlush (x11Display);

  *X11Display_out = (EGLNativeDisplayType) x11Display;
  *X11Window_out = (EGLNativeWindowType) x11Window;

  return 0;

 cleanup:
  if (x11Window) XDestroyWindow (x11Display, x11Window);
  if (x11Colormap) XFreeColormap (x11Display, x11Colormap );
  if (x11Display) XCloseDisplay (x11Display);

  return -1;
}
```

And finally, our end code that destroys the window and closes the connection to the display.

```
int
pf_re_native_end ()
{
  if (x11Window)
     XDestroyWindow (x11Display, x11Window);
  if (x11Colormap)
     XFreeColormap (x11Display, x11Colormap);
  if (x11Display)
     XCloseDisplay (x11Display);

  return 0;
}
```

That's it

2.6 Running the application

Our source code is there and we just need to compile it so we can execute it. Let's see how to do it on our different platforms

2.6.1 Running on the host PC

For compiling the application in a normal PC, you will need and OpenGL ES 2.0 emulator, and some hardware support. Refer to your emulator documentation to find out the features your hardware needs to support.

Then we need the include files and the libraries. If you are working with the source code provided with this text, you need to put this files under the directories `pc/libs` and `pc/includes`. You directory tree should look like this:

```
pc
|-- include
```

```
|   |-- EGL
|   |   |-- egl.h
|   |   `-- eglplatform.h
|   `-- GLES2
|       |-- gl2.h
|       |-- gl2ext.h
|       |-- gl2extimg.h
|       `-- gl2platform.h
|-- libs
|   |-- libEGL.so
|   `-- libGLESv2.so
`-- src
    `-- render_engine_sys.c
```

Now we just need a simple `Makefile` like this:

```
PLATFORM=pc
TARGET=ogles01_${PLATFORM}
INCLUDE_DIR=-I. -I./include -I./${PLATFORM}/include
LIB_DIR=-L./${PLATFORM}/libs
LIBS=-lEGL -lGLESv2 -lm

SOURCES=main.c render_engine.c pf_tri.c

SOURCES_SYS=./${PLATFORM}/src/render_engine_sys.c

all:${TARGET}

${TARGET}: ${SOURCES} ${SOURCES_SYS}
	${CC} -Wall -o $@ ${SOURCES} ${SOURCES_SYS} ${INCLUDE_DIR} ${LIB_DIR}
${LIBS}
```

We can execute `make` to get our executable. But to execute it, we will need to setup the `LD_LIBRARY_PATH` variable to point to the folder containing our libraries. Something like this:

`export LD_LIBRARY_PATH=/path_to_tutorial/pc/libs:$LD_LIBRARY_PATH`

Compile it and run the executable `ogles01_pc`. You will see something like this:

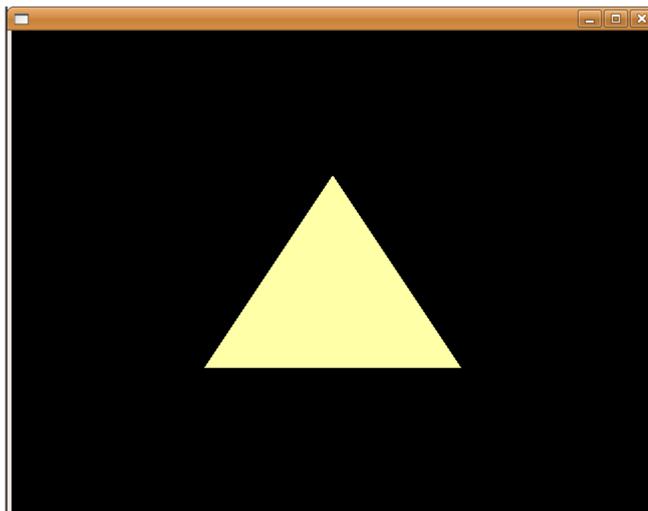


Figure 2.1: Hello triangle using PC emulator

2.6.2 Running on the BeagleBoard and Zoom2

For compiling the application in our OMAP platforms, we have a couple of options. We can compile the application natively in the target platform or we can use a cross compiler.

For the first option we can use the same makefile introduced in the previous sections, but simply removing the `LIBS_DIR` variable, as the libraries should be properly installed and the linker should be able to find them. The `makefile` will look like this:

```
PLATFORM=beagle
TARGET=ogles01_${PLATFORM}
INCLUDE_DIR=-I. -I./include -I./${PLATFORM}/include
LIBS=-lEGL -lGLESv2 -lm

SOURCES=main.c render_engine.c pf_tri.c

SOURCES_SYS=./${PLATFORM}/src/render_engine_sys.c

all:${TARGET}

${TARGET}: ${SOURCES} ${SOURCES_SYS}
        ${CC} -Wall -o $@ ${SOURCES} ${SOURCES_SYS} ${INCLUDE_DIR} ${LIB_DIR}
${LIBS}
```

For cross-compiling we will need to setup a tree similar to the one above, but substituting the libraries from the OpenGL ES 2.0 emulator by the libraries for our target platform. For the Beagleboard and the Zoom2 we will need to include some more extra libraries in our `libs` dir, or make our `LIB_DIR` variable to point to the SDK libraries directory... that is up to you.

Then we need to setup a couple of environmental variables:

```
export PATH=/path_to_toolchain/:$PATH
export CC=your-toolchain-compiler
```

For example, using the `arm-2008q3` toolchain from Code Sourcery, installed in !/opt!, and the graphics SDK from TI, installed also in `/opt`, your environmental variables should look like this:

```
export PATH=/opt/arm-2008q3/bin/:$PATH
export CC=arm-none-linux-gnueabi-gcc
```

And your `Makefile` like this:

```
PLATFORM=beagle
TARGET=ogles01_${PLATFORM}
INCLUDE_DIR=-I. -I./include -I./${PLATFORM}/include
LIB_DIR=-L/home/edma/OMAP35x_Graphics_SDK_3_00_00_06/gfx_rel
LIBS=-lEGL -lGLESv2 -lm -lIMGegl -lsrv_um

SOURCES=main.c render_engine.c pf_tri.c

SOURCES_SYS=./${PLATFORM}/src/render_engine_sys.c

all:${TARGET}

${TARGET}: ${SOURCES} ${SOURCES_SYS}
        ${CC} -o $@ ${SOURCES} ${SOURCES_SYS} ${INCLUDE_DIR} ${LIB_DIR} ${LIBS}
```

The result should look like this:

Figure 2.2: Hello triangle on OMAP Zoom2 MDP

2.7 So far

Not too much in this first chapter, but we had setup a lot of things to work faster from this point on. Most of the things we had set up in this chapter will not change in future and we will concentrate on the graphics in the coming chapters.