How to utilise OpenGL ES 2.0 on Symbian^3 and Maemo

Introduction

This article analyze and explains in detail a few of the most usable ways to deploy OpenGL ES 2.0 functionality with Nokia's Symbian^3 and Maemo™ devices.

OpenGL ES

OpenGL ES stands for 'Open Graphics Library for Embedded Systems'. It is a subset of the full OpenGL 3D graphics API. As with all full OpenGL, ES also has different versions: 1.0, 1.1, 2.0, and 2.1. Version 2.0 was a major release because the fixed pipeline was replaced by a programmable one in it.

OpenGL ES 2.0 is significant because most of the fixed pipeline functionality was removed from it. As a result, OpenGL ES 2.0 is not backwards compatible with older versions. OpenGL ES 2.0 is simpler and more elegant, with modern graphic chips with a Graphics Processing Unit (GPU). OpenGL ES 2.0 and above should be used whenever possible.

OpenGL (ES) is just a rendering library, not an independent application framework. To create a full working application, such as a game, there are several tasks that the program must handle outside of OpenGL (often provided by the platform), such as input events, image loading, timing, and so on.

OpenGL Shading Language (GLSL), also known as GLslang, is a language used for programming the GPU. It was created by the OpenGL Architecture Review Board (ARB) to give developers more direct control of the graphics pipeline without having to use assembly language or hardware-specific languages. GLSL is based on the C programming language, and it looks a lot like it. The syntax (and at least the accuracy that the implementations follow the syntax) has minor differences between different OpenGL versions, but what works with ES generally works with the full version as well.

Qt

Qt is a cross-platform application and UI framework ported to several different platforms. Qt is used as a base middle-level framework for Nokia devices. More information can be found at http://qt.nokia.com/.

QtOpenGL

Qt is delivered with a built-in module for deploying OpenGL (QtOpenGL). The QtOpenGL component itself doesn't implement OpenGL at any level, but maps the underlying OpenGL library to its structures and provides easy, Qt-like access to it. Because of this, when OpenGL is used through Qt, the version of OpenGL is not specified. Developers can have some control of the version with defines, but they are limited to the versions offered by the underlying library. This is an important issue to remember, since the Qt software aimed to any target is developed in a desktop computer, which has full OpenGL support unless an emulation library has been installed. Accordingly, there will be issues when trying out software developed in this way on an actual embedded device.

QtOpenGL lets you use OpenGL graphics within the rest of the UI. It provides a QGLWidget that you can use almost like any other Qt component. There are some problems with this arrangement: OpenGL that runs via QGLWidget is a little slower than other methods, and some portability issues will arise due to minor syntax differences. Obviously, this slowdown will be addressed in the future.

The example program

You will now build a small example program for Symbian^3 and Maemo devices using a couple of different techniques: QtOpenGL and native OpenGL with a Qt framework (using File:Qtgameenabler v2 1 NokiaDeveloperExample Qt.zip). You will also build a native OpenGL application for Symbian^3 with the Symbian native SDK.

The example uses OpenGL ES 2.0. As a result, the program will output a rotating cube with depth-based gradient lighting implemented with GLSL (programmable pipeline).

Platform-independent parts

These sections of code are exactly the same for all different methods deploying OpenGL ES 2.0. The following define 8 vertices and 12 triangles for the cube; the cube is meant to be rendered with an indexed call such as: glDrawElements. The source code for the vertex and fragment shader are defined as an ASCIIZ (zero terminated, 8-bit wide) string that is transmitted to the GPU at runtime.

example_defines.h

```
#ifndef EXAMPLE_DEFINES_H
#define EXAMPLE_DEFINES_H
extern const float cube_vertices[];
extern const unsigned short cube_indices[];
extern const char *sample_vertex_shader_src;
extern const char *sample_fragment_shader_src;
#endif
```

```
#include "example_defines.h"

const GLfloat cube_vertices[] = {
    -1.0f,-1.0f,1.0f, 1.0f,-1.0f,1.0f, 1.0f,1.0f,1.0f, -1.0f,1.0f,1.0f,
    -1.0f,-1.0f,-1.0f, 1.0f,-1.0f,-1.0f, 1.0f,1.0f,-1.0f, -1.0f,1.0f,-1.0f };

const GLushort cube_indices[]= {
    0,1,2, 0,2,3, 1,5,6, 1,6,2, 4,0,3, 4,3,7, 5,4,7,
    5,7,6, 4,5,1, 4,1,0, 2,6,7, 2,7,3 };

const char *sample_vertex_shader_src =
    "attribute highp vec4 vertex;\n"
    "uniform mediump mat4 matrix;\n"
    "uniform mediump mat4 proj;\n"
    "varying highp vec4 frag_pos;\n"
    "void main(void)\n"
        "{\n"
        "    frag_pos = matrix * vertex;\n"
        "    gl_Position = proj*frag_pos;\n"
        "}\n";

const char *sample_fragment_shader_src =
    "varying highp vec4 frag_pos;\n"
    "void main(void)\n"
    "{\n"
    "    mediump float light = 2.0 / -(frag_pos.z);\n"
    "    gl_FragColor = vec4(light, light, light, 0.5);\n"
    "}\n";
```

Look at the code: There are eight coordinates defined as a table of floats. Floats are ordered in sets of three values. Each valueset (called a 'vertex') defines an X,Y,Z position in a 3D world. Indices point to these coordinates. They form a set of three points defining a list of triangles formed from the vertices. The list of 36 points forms 12 triangles. Two triangles are required for each side of the cube.

Example program with QtOpenGL

You can create an actual program to implement your example with QtOpenGL. When finished, you can test it with the desktop computer. The program should work with Symbian^3 devices and the Nokia N900 device with just a few minor modifications.

Prerequisites

When an application needs to run with a desktop computer as well (as in this case), you will need the Qt SDK for Open Source C++ Development on Windows. The package can be found at forum.nokia.com: http://www.developer.nokia.com/info/sw.nokia.com/id/da8df288-e615-443d-be5c-00c8a72435f8/Qt_SDK.html.

Step by step

- Start the Qt Creator installed with the Qt SDK.
- Select File > New file or project.
- Create a new Qt Gui application. At the Qt Versions phase, select all of the targets except the Qt Simulator. At the details phase, unselect Generate form; you don't need one.
- Finish the wizard with your preferred choices. When the wizard has finished, your project should contain three files: main.cpp, mainwindow.h, and mainwindow.cpp.
- When working with QtOpenGL, you need to add an opengl flag into your .PRO file. Go to your .PRO file and add `opengl` to the end of the line with the Qt flags, so it looks like this: {{{1}}}.
- The application's mainwindow object is derived from QMainWindow by default. You don't want that; instead, you want QGLWidget, so go to mainwindow.h and change `#include <QMainWindow>` to `#include <QGLWidget>`. Then change the class to be derived from QGLWidget: `class MainWindow : public QGLWidget`. Now your application has an OpenGL view as its mainview.
- You need to override a few functions from QGLWidget, so add the following methods into the mainwindow as public members:

```
void resizeGL(int width, int height);
void initializeGL();
void paintGL();
void mousePressEvent(QMouseEvent *e);
```

- Add a protected member QGLShaderProgram m_sampleProgram into the mainwindow object. This requires a new include <QtOpenGL/qglshaderprogram.h>. Add that to mainwindow.h. The m_sampleProgram is a capsule that will contain the compiled vertex and fragment shaders for the rendering.
- Add example_defines.h and example_defines.cpp into your project. Include the header at the beginning of mainwindow.cpp; this is the only place you will need the information provided by it. You can also choose not to use example_defines as a separate file, but copy the contents of example_defines.cpp on top of mainwindow.cpp.
- Start the implementations by tweaking your constructor a little. The following will set a few attributes of the window in order to maximise the speed of OpenGL. It also creates and starts a timer (QTimer) object that will be responsible for requesting new frames from the application by calling updateGL(). The timer requires a new include <QTimer>; add it into the beginning of mainwindow.cpp.

```
MainWindow::MainWindow(QWidget *parent) {
    setAutoFillBackground( false );
    setAttribute (Qt::WA_OpaquePaintEvent);
    setAttribute( Qt::WA_NoSystemBackground );
    setAttribute( Qt::WA_NativeWindow );
    setAttribute( Qt::WA_PaintOnScreen, true );
    setAttribute( Qt::WA_StyledBackground,false);
    setAutoBufferSwap(false);
    // Startup the timer which will call updateGL as fast as it can.
    QTimer *timer = new QTimer(this);
    QObject::connect(timer, SIGNAL(timeout()), this, SLOT(updateGL()));
    timer->start();
}
```

- resizeGL is quite straightforward. When your window's size changes, reset the OpenGL viewport to match it. The mousePressEvent is overridden just to make sure you have a clean way to exit the application. Whenever you acquire any kind of mousePressEvent, shut down the application.

```
void MainWindow::resizeGL(int width, int height) {
        // Reset the GL viewport for current resolution.
    glViewport(0,0, width, height);
};
```

```
void MainWindow::mousePressEvent(QMouseEvent *e) {
    exit(0);
};
```

- The OpenGL initialisation. The Qt framework will call this when everything is ready for OpenGL setup. Before the first rendering call, compile your shaders and add them into m_sampleProgram. The source code for shaders is provided by example_defines, and changes to it should be implemented there. Also, some OpenGL flags are set that are going to remain the same through the entire execution.

```
void MainWindow::initializeGL() {
    // Create and compile the vertex-shader
    QGLShader *vertex_shader = new QGLShader(QGLShader::Vertex, this);
    vertex_shader->compileSourceCode(sample_vertex_shader_src);
    // Create and compile the fragment-shader
    QGLShader *fragment_shader = new QGLShader(QGLShader::Fragment, this);
    fragment_shader->compileSourceCode(sample_fragment_shader_src);
    // Add vertex-shader to our program
    m_sampleProgram.addShader(vertex_shader);
    // Add fragment-shader to our program
    m_sampleProgram.addShader(fragment_shader);
    // Link our program. It's now ready for use.
    m_sampleProgram.link();
    glFrontFace(GL_CCW);
    glEnable(GL_CULL_FACE);
    glCullFace(GL_BACK);
    glClearColor(0.1f, 0.4f, 0.1f, 1.0f);
};
```

- You are now ready for the actual painting -- the paintGL method, which refers to static data defined at example_defines for vertices and indices.

```
void MainWindow::paintGL() {
    // Clear the background and depth-buffer for this frame
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    // Get the attribute locations from the shaderprogram
    GLint vertexAttr = m_sampleProgram.attributeLocation("vertex");
    GLint matrixAttr = m_sampleProgram.uniformLocation("matrix");
    GLint projAttr = m_sampleProgram.uniformLocation("proj");
    // Set our own shaderprogram as an active one to GPU. There
    // can be only one active program at the time.
    m_sampleProgram.bind();
    // Create the projection-matrix with QMatrix4x4
    QMatrix4x4 projection;
    projection.perspective(45.0f, (float)width()/(float)height(), 0.1f, 20.0f );    // Use perspective projection
    // Set our newly created projection matrix to our shaderprogram
    m_sampleProgram.setUniformValue( projAttr, projection );                        // Set projection to the shader
    // Set vertexarray to the shaderprogram
    m_sampleProgram.enableAttributeArray(vertexAttr);
    m_sampleProgram.setAttributeArray(vertexAttr, cube_vertices, 3 );
    // Create orientation for the cube we are about to render
    QMatrix4x4 orientation;
    orientation.setToIdentity();
    // 4.5 units away from camera
    orientation.translate(0.0f, 0.0f, -4.5f );
    // Spinning rotation
    static float ang = 0.0f;
    ang+=0.1f;
    // Rotations around 3 main axis. Angles changing in varying speed.
    orientation.rotate(ang, 1.0f, 0.0f, 0.0f );
    orientation.rotate(ang*0.7f, 0.0f, 1.0f, 0.0f );
    orientation.rotate(ang*1.4f, 0.0f, 0.0f, 1.0f );
    // Set orientation matrix to the shaderprogram
    m_sampleProgram.setUniformValue( matrixAttr, orientation );
    // draw the cube.
    glDrawElements(GL_TRIANGLES, 6*2*3, GL_UNSIGNED_SHORT, cube_indices );
    // Disable the program
    m_sampleProgram.disableAttributeArray(vertexAttr);
    m_sampleProgram.release();
    // Swap buffers manually (automatic swapping disabled).
    swapBuffers();
};
```

Now you can build your application on the desktop. In Qt Creator, open a target selector by clicking it or pressing CTRL-T. Select 'Desktop' target with debug. If you can't find the Desktop target, the Qt installation has failed to auto scan it (it depends on the install order). In that case, it will need to be added manually. Go to Tools > Options, and select the Qt4 tab. At the subtab 'Qt Versions', click the plus sign on the right. You will need to provide the location of qmake.exe and the main folder for the MinGW directory. It should look something like this:

```
Qmake location: C:\qt\2010.05\qt\bin\qmake.exe
MinGW directory: C:\Qt\2010.05\mingw
```

Build and launch. The results should look like this:

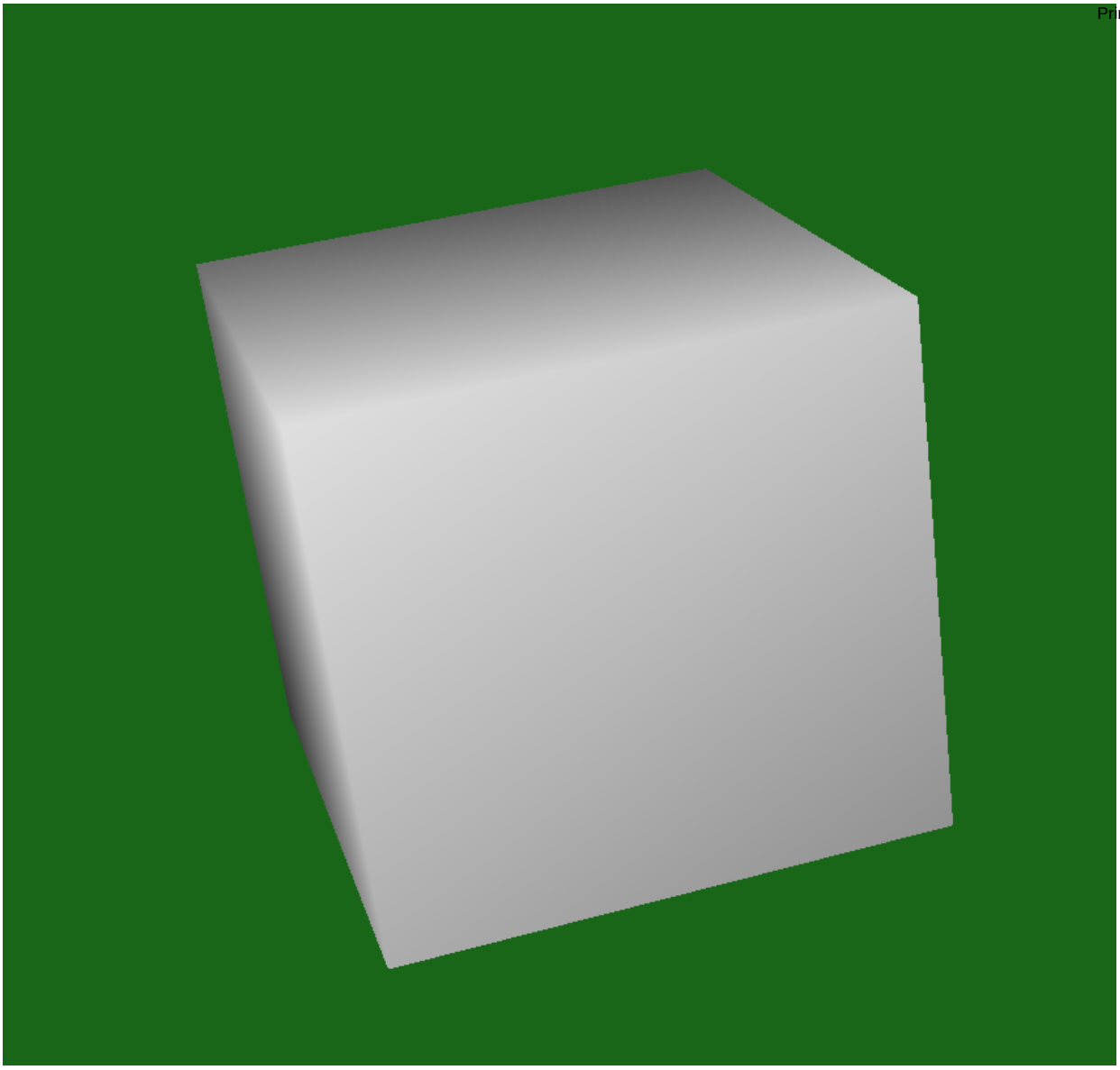

Figure 1: Example running in a desktop target

Differences between native OpenGL and QtOpenGL

The main differences are due to the fact that some OpenGL functionality is encapsulated inside QtOpenGL's own structures. Practically, that means some parts must be written a little differently compared to native OpenGL. For example, in QtOpenGL, shaders are inside the class QGLShaderProgram, and instead of calling a global glCompileShader function, you will use a method QGLShaderProgram.compile().

When QtOpenGL software is being developed, these classes are used together with native OpenGL calls. The following table lists the classes responsible for encapsulation in the QtOpenGL library:

Name	**Description**
QGLContext	Encapsulates an OpenGL rendering context
QGLWidget	Widget for rendering OpenGL graphics
QGLFormat	Specifies the display format of an OpenGL rendering context
QGLFrameBufferObject	Encapsulates an OpenGL framebuffer object
QGLShader	Allows OpenGL shaders to be compiled
QGLShaderProgram	Allows OpenGL shader programs to be linked and used
QGLColormap	Used for installing custom colormaps into a QGLWidget
QGLFramebufferObject	Encapsulates an OpenGL framebuffer object
QGLFramebufferObjectFormat	Specifies the format of an OpenGL framebuffer object
QGLPixelBuffer	Encapsulates an OpenGL pbuffer
QWSGLWindowSurface	Drawing area for top-level windows with Qt for Embedded Linux on EGL/OpenGL ES. Also provides drawing area for QGLWidgets, whether they are top-level windows or children of another QWidget

Qt has some problems with glext, which is the standard for many OpenGL 'extended' features, the most significant being multi-texturing (glActiveTexture). Even though the previous example doesn't use these features, it is important to know that they exist. Qt doesn't provide direct headers for them. If the application wishes to use them, it requires a small hack.

Qt's own examples solve this by adding a class that searches the pointers for these functions from the OpenGL implementation below and maps them into its own structures. It is used in the Boxes demo, which can be found in the SDK (glextensions.h and glextensions.cpp).

Symbian^3

QtOpenGL on Symbian^3

Prerequisites

The Qt SDK has everything you need for compiling QtOpenGL to Symbian^3. You can find the SDK at Nokia Developer: http://www.developer.nokia.com/info/sw.nokia.com/id/da8df288-e615-443d-be5c-00c8a72435f8/Qt_SDK.html. It can be installed on top of previously installed Qt packages.

Step by step

You are about to build the QtOpenGL example that you created earlier for the desktop target.

- Open the example projects to Qt Creator.
- Select the Symbian^3 target from Qt Creator's target selector.
- Connect your Symbian^3 mobile device to your computer with a USB cable.
- Press Run at the bottom left of the Qt Creator screen.
- The application should build, deploy, and start running in your Symbian^3 device.

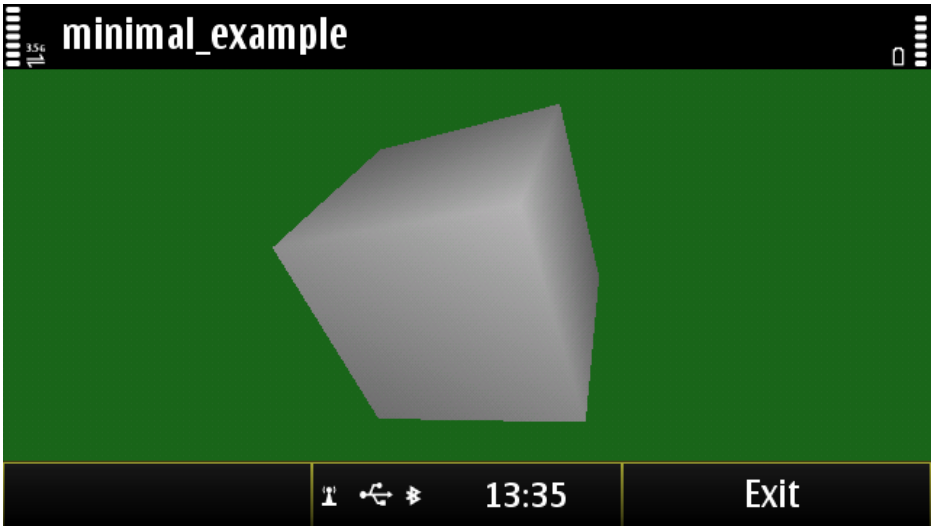

Figure 2: Example program running on Nokia N8 device in landscape mode

Native OpenGL with Qt application framework using Qt GameEnabler

Qt's framework can be mixed with an underlying native OpenGL ES 2.0 implementation as well. When this method is used, Qt provides the complete application framework, helper classes (such as vectors, matrices, etc.), control events, image loaders, and so on; OpenGL is used only for rendering. Qt's components cannot be rendered when native OpenGL is used. Nokia Developer's File:Qtgameenabler v2 1 NokiaDeveloperExample Qt.zip example project does this. It offers a QGLWidget kind-of class (derived from QWidget) that has native OpenGL ES 2.0 instead of QGLWidget's QtOpenGL. This is a very easy way to do game-kind-of (custom UI) applications for Symbian^3 devices.

Native OpenGL ES 2.0 cannot be compiled directly to a desktop, since desktop OpenGL libraries do not normally support it directly. Some new OpenGL libraries have direct support for OpenGL ES 2.0. There are freely available debug OpenGL ES 2.0 emulation libraries that developers can use to compile their software into a desktop. The two most popular are:

- PowerVR: http://www.imgtec.com/powervr/insider/sdkdownloads/index.asp
- ARM: http://www.malideveloper.com/developer-resources/tools/opengl-es-20-emulator.php

NOTE: If OpenGL ES 2.0 emulation libraries are not provided, the following application won't compile to the desktop.

Step by step

The Qt GameEnabler program is just like any other Qt program, and therefore most easily developed with Qt Creator as well. You can build your simple test program with Qt GameEnabler's GameWindow.

- Get the Qt GameEnabler from here: File:Qtgameenabler v2 1 NokiaDeveloperExample Qt.zip
- Open the QtGameEnablerTest.pro template project.
- Run it to make sure everything is working correctly. You should see a slowly flashing green screen.
- Include example_defines.h in mygamewindow.cpp.
- Add members that you need into mygamewindow.h.

```
GLuint vertexShader;
GLuint fragmentShader;
GLuint shaderProgram;
float cubeAngle;
```

Good programming practices force you to zero everything at the constructor. This is always a good idea. In MyGameWindow's constructor, add:

```
vertexShader = 0;
fragmentShader = 0;
shaderProgram = 0;
cubeAngle = 0.0f;
```

- Override methods for creating, destroying, and updating:

```
void onCreate();
void onDestroy();
void onUpdate( const float frameDelta );
```

- Create an implementation of MyGameWindow::onCreate() in mygamewindow.cpp to initialise your shader program and other things related to

rendering.

```
void MyGameWindow::onCreate() {
    fragmentShader = glCreateShader(GL_FRAGMENT_SHADER);
    glShaderSource(fragmentShader, 1, (const char**)&sample_fragment_shader_src, NULL);
    glCompileShader(fragmentShader);
    vertexShader = glCreateShader(GL_VERTEX_SHADER);
    glShaderSource(vertexShader, 1, (const char**)&sample_vertex_shader_src, NULL);
    glCompileShader(vertexShader);
    shaderProgram = glCreateProgram();
    // Attach the fragment and vertex shaders to it
    glAttachShader(shaderProgram, fragmentShader);
    glAttachShader(shaderProgram, vertexShader);
    glBindAttribLocation(shaderProgram, 0, "vertex");
    glLinkProgram(shaderProgram);
    glUseProgram(shaderProgram);
    glEnableVertexAttribArray(0);
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, cube_vertices );
        // Use perspective projection
    QMatrix4x4 projection;
    projection.perspective(45.0f, (float)width()/(float)height(), 0.1f, 20.0f );
        // we need to flip rows and columns since QMatrix4x4
        // is defined in other-way-around than normal opengl
    float tempMat[4][4];
    for (int r=0; r<4; r++)
    for (int c=0; c<4; c++)
        tempMat[r][c] = projection.constData()[ r*4 + c ];
    int projLocation = glGetUniformLocation(shaderProgram, "proj");
    glUniformMatrix4fv(projLocation, 1, GL_FALSE, (float*)&tempMat[0][0]);
    glFrontFace(GL_CCW);
    glEnable(GL_CULL_FACE);
    glCullFace(GL_BACK);
    glClearColor(0.1f, 0.4f, 0.1f, 1.0f);
}
```

- Don't forget to add MyGameWindow::onDestroy() for releasing the program when the program is shutting down:

```
void MyGameWindow::onDestroy() {
    glDeleteShader( vertexShader );
    glDeleteShader( fragmentShader );
    glDeleteProgram( shaderProgram );
};
```

- Now you can start implementing the actual logic. A frame in Qt GameEnabler is divided in two parts: update and render. Update is meant for changing values, running logic, and so on, and render is purely for rendering the application. You only have one attribute to update, an angle describing your cube's orientation: cubeAngle. You can update it in onUpdate().

```
void MyGameWindow::onUpdate( const float frameDelta ) {
    cubeAngle += frameDelta*6.0f;
};
```

The attribute frameDelta describes the number of seconds that have passed since the last frame. When you add it to cubeAngle, it will increase by 6 in a single second. Changing the multiplier increases or slows down the rotation speed.

- Now everything is set up for rendering. Replace MyGameWindow's onRender() with the following:

```
void MyGameWindow::onRender()
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
    QMatrix4x4 orientation;
    orientation.setToIdentity();
    orientation.translate(0.0f, 0.0f, -4.5f );
    orientation.rotate(cubeAngle, 1.0f, 0.0f, 0.0f );
    orientation.rotate(cubeAngle*0.7f, 0.0f, 1.0f, 0.0f );
    orientation.rotate(cubeAngle*1.4f, 0.0f, 0.0f, 1.0f );
        // we need to flip rows and columns since QMatrix4x4
        // is defined in other-way-around than normal opengl
    float tempMat[4][4];
    for (int r=0; r<4; r++)
        for (int c=0; c<4; c++)
            tempMat[r][c] = orientation.constData()[ r*4 + c ];
    int location = glGetUniformLocation(shaderProgram, "matrix");
    glUniformMatrix4fv(location, 1, GL_FALSE, (float*)&tempMat[0][0]);
        // Render!
    glDrawElements( GL_TRIANGLES, 6*2*3, GL_UNSIGNED_SHORT, cube_indices );
}
```

- Select your target (Symbian^3 or Maemo) and hit Run!

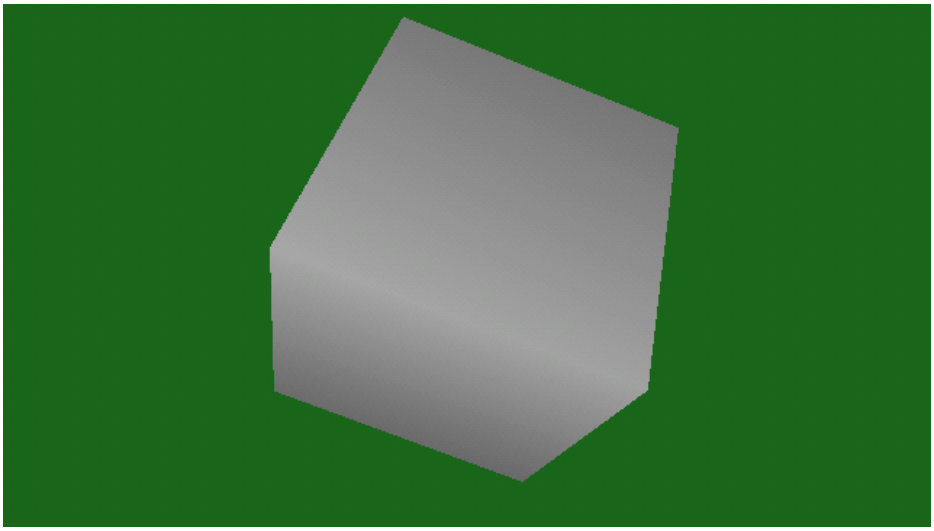

Figure 3: Example running on a Nokia N8 device with Qt GameEnabler

OpenGL with Symbian^3 framework

Prerequisites

The Symbian^3 SDK contains everything needed. It can be downloaded from Nokia Developer:
http://www.developer.nokia.com/info/sw.nokia.com/id/571baaf2-042a-4662-9d43-a5a68eb42d9d/Symbian_3_SDK_v1_0_en.zip.html.

Testing an example

After installing the SDK, the fastest way to make sure that everything is working correctly is to build one of the examples directly from the console. For this article, we chose one that deploys OpenGL ES 2.0: SimpleShader. However, the compilation/installation instructions should work for any other example as well.

Open a console:

```
// go to the SimpleShader example's group folder
cd SYMBIAN^3_SDK_PATH\examples\symbian\gui\openglex\SimpleShader\group
// create the scripts required for build
bldmake bldfiles
// build a release version of the example with GCCE-tool chain
abld build gcce urel
```

The Symbian tools must be found from the current path in order for the compilation to work properly. They should be correct after installing all the packages, but sometimes having multiple SDKs on a single computer can cause problems.

After a successful compilation, the executable is built. Now the SIS package should be created. The SimpleShader example doesn't have a .PKG file, which uses binary compiled with GCCE. You'll need to create one to be able to do this:

- Go to the SimpleShader's sis folder.
- Open SimpleShader_armv5.pkg (you will modify this to use GCCE binary).
- Change SimpleShader.exe's source location from epoc32\release\armv5\urel\SimpleShader.exe to epoc32\release\gcce\urel\SimpleShader.exe.
- Save the file as SimpleShader_gcce.pkg.
- Check that the paths of the files actually exist.

The edited file should look something like this:

```
;Languages
&EN

;Header
#{"SimpleShader"},(0xE0000400), 1, 0, 0,TYPE=SA

;Localised Vendor name
%{"Nokia Test EN"}

;Unique Vendor name
:"Nokia"

;Supports S60 v 3.0
[0x20022e6d], 0, 0, 0, {"Series60ProductID"}

;Files to install, IN THE FOLLOWING LINE IS OUR MODIFICATION
"..\..\..\..\..\..\..\..\epoc32\release\gcce\urel\SimpleShader.exe"-"!:\sys\bin\SimpleShader.exe"
"..\..\..\..\..\..\..\..\epoc32\data\z\resource\apps\SimpleShader.RSC"-"!:\resource\apps\SimpleShader.RSC"
"..\..\..\..\..\..\..\..\epoc32\data\Z\private\10003a3f\import\apps\simpleshader_reg.RSC"-"!:\private\10003a3f\import\apps\simplesha
"..\..\..\..\..\..\..\..\epoc32\data\z\resource\apps\simpleshader_icon.mif"-"!:\resource\apps\simpleshader_icon.mif"

;required for application to be covered by backup/restore facility
"backup_registration.xml"
```

Now you can build the package. Go back to the console, and at the examples sis folder do the following:

```
makesis SimpleExample_gcce.pkg
```

If everything is working correctly, the sis folder now contains a simpleshader.sis. A package with the executable and required assets defined in the MMP file are packed inside. If there was a problem with the packaging, make sure that the files defined in .PKG are actually located at the correct locations. You can replace relative paths with absolute ones if required.

The SIS package needs to be signed before it can be tested on an actual device. The signing requires .CER and .KEY files, which can be obtained in several ways. You can just use self-signing for test purposes. In a console, create the key and the certificate by doing the following:

```
makekeys -cert -password World123 -len 1024 -dname "CN=World User OU=Development OR=WorldCompany CO=FI EM=World@test.com" WorldKey.k
```

This will create `WorldKey.key` and `WorldCert.cer`, which can be used for self-signing. Obviously, the attributes of the key-generation can be changed as well. Now the actual signing can be applied:

```
signsis simpleshader.sis simpleshader_signed.sis WorldCert.cer WorldKey.key World123
```

The signed sis (simpleshader_signed.sis) can now be installed into a Symbian^3 device and run. The SimpleShader example doesn't produce much -- just a static, single triangle with a position-based gradient. However, the result proves that the SDK and API are working correctly with shaders.

Modifying the SimpleShader example

The easiest way to implement your own OpenGL application for Symbian^3 is to take one of the examples as a starting point and modify it. Now you will have a chance to modify the SimpleShader example to implement your earlier OpenGL demo application built with QtOpenGL for the Nokia N900 device.

Step by step

- Clone the SimpleShader example's all contents into your preferred location.
- Copy example_defines.h and example_defines.cpp into the project's src folder (or copy the structures from the cpp into the beginning of simpleshader.cpp).
- If you chose to copy the files, add include to the simpleshader.cpp for example_defines.h and add example_defines.cpp into the MMP file in the group folder.
- At simpleshader.cpp's AppInit, replace glVertexAttribPointer to use your own cube_vertices.
- Change the shader sources to point into your own data: sample_vertex_shader_src and sample_fragment_shader_src defined in example_defines.
- Replace the clear colour, add the culling states, and disable the depth buffering from AppInitL:

```
glFrontFace(GL_CCW);
glEnable(GL_CULL_FACE);
glCullFace(GL_BACK);
glClearColor(0.1f, 0.4f, 0.1f, 1.0f);
```

- Add a new function for multiplying matrices together. This is required for the multi-axis rotation of your cube. OpenGL does not have any kind of matrix operations, so you must implement them yourself. In a 'real' application, it is advisable to build a matrix class and do the operations inside: Transformations are widely needed in any 3D application. This is what was done with the Qt version of our example (Qt provided QMatrix4). For the Symbian example, it is done in the following way. In simpleshader.cpp, add a new global function (or define it as a member method; it doesn't matter):

```
void matrix3x3Multiply( float target[][4], float mul[][4] ) {
    float temp[4][4];
    memset( temp, 0, sizeof(float)*16 );

    for (int i=0; i<3; i++) {
        temp[i][0] = target[i][0]*mul[0][0] + target[i][1]*mul[1][0] + target[i][2]*mul[2][0];
        temp[i][1] = target[i][0]*mul[0][1] + target[i][1]*mul[1][1] + target[i][2]*mul[2][1];
        temp[i][2] = target[i][0]*mul[0][2] + target[i][1]*mul[1][2] + target[i][2]*mul[2][2];
    };
    temp[3][3] = 1.0f;
    memcpy( target, temp, sizeof( float ) * 16 );
};
```

- In this simple example you won't be implementing custom projection matrix generation, but the general shader you are using requires one. You can define a static projection matrix for the shader in the beginning of your simpleshader.cpp:

```
const float projectionMatrix[] = {
    1.43092f, 0, 0, 0,
    0, 2.41421f, 0, 0,
    0, 0 , -1.01005f , -1.0f,
    0, 0 , -0.201005f, 0
};
```

You won't have the correct aspect ratio/projection in all cases this way, but for test purposes, it doesn't matter.

- Now you need to modify the AppCycle method. First, uncomment the aFrame attribute so you can use it as a main value for your angles. Then add the code for creating a rotation matrix as a two-dimensional table of floats. Finally, replace the old rendering call glDrawArrays to glDrawElements (which is the indexed version) with cube_indices as its attributes. The new AppCycle looks like this:

```
void CSimpleShader::AppCycle( TInt aFrame,
                      TReal /*aTimeSecs*/,
                      TReal /*aDeltaTimeSecs*/ )
    {
    // Clear the colour and depth buffers
    glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );

    // temporary sin,cos variables
    TReal s,c, angle;

    // Create a rotation matrix.
    float orientation[4][4];
    // Set it to identity
    memset( orientation, 0, sizeof(float) * 16 );
    for (int set=0; set<4; set++) orientation[set][set] = 1.0f;

    // temporary matrix for rotating the orientation
    float rotate[4][4];

    // xz-rotation
    memset( rotate, 0, sizeof( float) * 16 );
    for (int set=0; set<4; set++) rotate[set][set] = 1.0f;
    angle = (TReal)aFrame / 120.0;
    Math::Cos( c, angle );
    Math::Sin( s, angle );
    rotate[0][0] = c;
    rotate[0][2] = s;
    rotate[2][0] = -s;
    rotate[2][2] = c;
    matrix3x3Multiply( orientation, rotate );


    // xy-rotation
    memset( rotate, 0, sizeof( float) * 16 );
    for (int set=0; set<4; set++) rotate[set][set] = 1.0f;
    angle = (TReal)aFrame / 180.0;
    Math::Cos( c, angle );
    Math::Sin( s, angle );
    rotate[0][0] = c;
    rotate[0][1] = s;
    rotate[1][0] = -s;
    rotate[1][1] = c;
    matrix3x3Multiply( orientation, rotate );


    // yz-rotation
    memset( rotate, 0, sizeof( float) * 16 );
    for (int set=0; set<4; set++) rotate[set][set] = 1.0f;
    angle = (TReal)aFrame / 90.0;
    Math::Cos( c, angle );
    Math::Sin( s, angle );
    rotate[1][1] = c;
    rotate[1][2] = s;
    rotate[2][1] = -s;
    rotate[2][2] = c;
    matrix3x3Multiply( orientation, rotate );

    // Set our rotation for the cube.
    int location = glGetUniformLocation(iProgram, "matrix");
    glUniformMatrix4fv(location, 1, GL_FALSE, (float*)&orientation[0][0]);
```

```
    // Set the cube's vertices for the program
    glEnableVertexAttribArray(0);
    glVertexAttribPointer( 0,3, GL_FLOAT, FL_FALSE, 0, cube_vertices );


    // Set our static projection matrix for the shader
    int projLocation = glGetUniformLocation(iProgram, "proj");
    glUniformMatrix4fv(projLocation, 1, GL_FALSE, projectionMatrix);
    // Render !
    glDrawElements( GL_TRIANGLES, 6*2*3, GL_UNSIGNED_SHORT, cube_indices );
    }
```

- Now everything is in place. Compile the code with the instructions from the previous chapter; create the SIS package; and sign, install, and RUN!

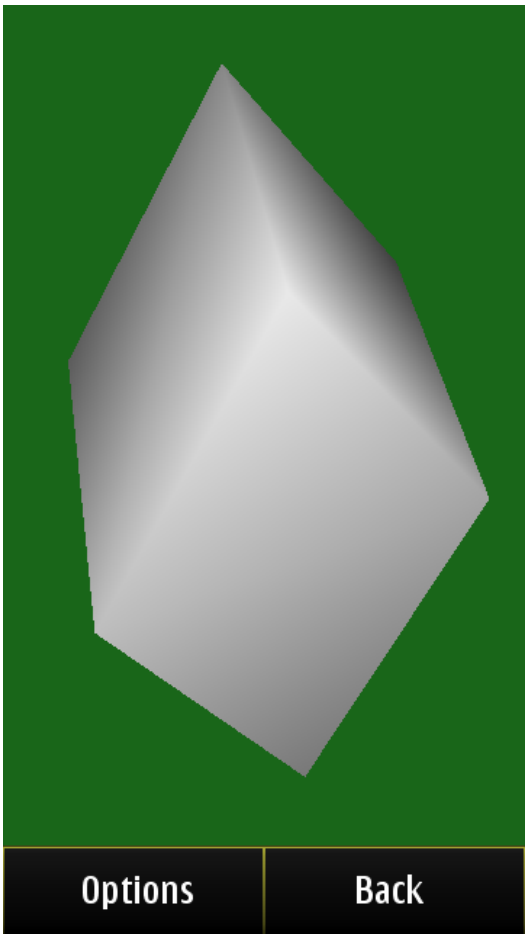

Figure 4: Example running in Nokia N8 device

Open C + OpenGL

Symbian provides an Open C interface that is practically an implementation of C's STDLIBs using Symbian's native components internally. Open C covers most aspects of the STD with quite good stability. Since EGL is just another C library, it can be easily used with Open C as well.

In the Symbian^3 SDK you can look at the example located at: SYMBIAN^3 SDK\examples\symbian\gui\opencex\opencopenglex. An OpenGL program written with Open C itself is very similar to any Glut-based OpenGL application (or Maemo's Xlib + OpenGL application).

Maemo™

The Nokia N900 mobile device provides support for both OpenGL ES 1 and 2. There are several ways to deploy them. Most games use the Xlib + OpenGL solution, but QtOpenGL is a good option as well, even though there's a small performance penalty.

With any OpenGL application using shaders, there is one thing you must remember to do before calling any compile functions: The locale must be set in a way that does not compromise the compilation. So, before compiling the shaders:

```
setLocale( LC_NUMERIC, "C"); // Reset locale for code compilation.
```

And after compiling the shaders, reset the locale:

```
setLocale( LC_ALL, "");     // Restore locale
```

If you forget to do this, the incorrect locale can cause unpredictable results.

QtOpenGL on Maemo

Prerequisites

The Qt SDK has everything you need for compiling QtOpenGL to Maemo. You can find it at: http://www.developer.nokia.com/info/sw.nokia.com/id/da8df288-e615-443d-be5c-00c8a72435f8/Qt_SDK.html. The SDK can be installed on top of previously installed Qt packages.

Step by step

To build the QtOpenGL example created earlier for the Nokia N900 device:

- Open the example projects to Qt Creator.
- Tweak the .PRO file a little bit by adding Maemo target-specific configurations:

```
maemo {
    DEFINES += QT_OPENGL_ES_2
    LIBS += -lGLES2v2 -lQtOpenGL
}
```

- Select the Maemo/release target from Qt Creator's target selector.
- Build the program. It will create a .DEB package for Maemo.
- Copy the built package into your device.
- You can install the .DEB either with 'shell-execute' (click from the file manager) or with the terminal:

```
dpkg -i package.deb
```

Follow these next steps:

- Make sure the installed binary (for example: `/usr/local/bin/software_name`) is executable. (Sometimes the binary installed by .DEB is not marked as executable.) If it is not executable, change its attributes accordingly: 'chmod +x binary_file_full_path'.
- Run the binary with a shell using standalone script (`standalone.sh`), or by clicking the application icon if your installer did one. A typical run with a standalone.sh looks like this:

```
run-standalone.sh /usr/bin/local/software_binary_name
```

Figure 5: Example program running on Nokia N900 device

There are several other examples deploying QtOpenGL on a Maemo device, such as the nice demo combining OpenGL ES 2.0 rendering with a third-party physics engine: OpenGL ES 2.0 Physics demo. It can be located at http://github.com/nokia-developer/bullet-dice

Xlib + OpenGL ES 2

You can use Xlib (X11) and utilise the EGL library for context creation along with OpenGL ES. The Bounce Evolution game uses this technique. Use OpenGL ES 1 or 2 directly. An OpenGL program written with Xlib works quite similarly to any other Glut-type application.

SDL-GLES

The Nokia N900 device is shipped with SDL 1.2 preinstalled. You can use that and an external library, SDL-GLES (available as an extra), to create simple OpenGL ES 1.1 and 2.0 applications. Use any SDL functionality within your software, such as image loaders, input framework, sound, and so on. The problem with SDL programs is the differences between versions. For example, SDL 1.3 for Maemo doesn't have working OpenGL support, so updating to a newer version will break the software. The stability and functionality of SDL for Maemo is evolving, but developers should be cautious when using it in its current state.

Summary

You've explored several different techniques for deploying OpenGL ES 2.0 for Nokia devices. All have advantages and disadvantages. The good thing to remember is that OpenGL is always OpenGL -- it's fairly easy to switch between frameworks without reimplementing too much of the code. If you start from scratch, without any engine or completed code, the use of the Qt framework, either by using QtOpenGL or Qt GameEnabler, is advisable, since with them the framework offers helper libraries for math, images, etc., which you would otherwise have to write yourself. Numerous examples of all the techniques covered here can be found on the internet.

Have fun coding!

Supplementary material

- Working code that can be used to test the functionality described in this article is available for download at Media:qtopengl minimal example.zip.
- A minimal example utilising File:Qtgameenabler v2 1 NokiaDeveloperExample Qt.zip is available for download at Media:gameenabler minimal example.zip.