Extracted from:

OpenGL ES 2 for Android

A Quick-Start Guide

This PDF file contains pages extracted from *OpenGL ES 2 for Android*, published by the Pragmatic Bookshelf. For more information or to purchase a paperback or PDF copy, please visit http://www.pragprog.com.

Note: This extract contains some colored text (particularly in code listing). This is available only in online versions of the books. The printed versions are black and white. Pagination might vary between the online and printed versions; the content is otherwise identical.

Copyright © 2013 The Pragmatic Programmers, LLC.

All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

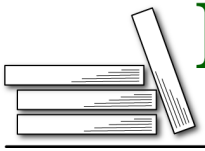
OpenGL ES 2 for Android

A Quick-Start Guide

Kevin Brothaler

Foreword by Mario Zechner, creator of libgdx, author of *Beginning Android Games*

Edited by Susannah Davidson Pfalzer



Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and The Pragmatic Programmers, LLC was aware of a trademark claim, the designations have been printed in initial capital letters or in all capitals. The Pragmatic Starter Kit, The Pragmatic Programmer, Pragmatic Programming, Pragmatic Bookshelf, PragProg and the linking *g* device are trademarks of The Pragmatic Programmers, LLC.

Every precaution was taken in the preparation of this book. However, the publisher assumes no responsibility for errors or omissions, or for damages that may result from the use of information (including program listings) contained herein.

Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at *http://pragprog.com*.

The Android robot is reproduced from work created and shared by Google and is used according to terms described in the Creative Commons 3.0 Attribution License (*http://creativecommons.org/licenses/by/3.0/us/legalcode*).

The unit circle image in Figure 43, from *http://en.wikipedia.org/wiki/File:Unit_circle.svg*, is used according to the terms described in the Creative Commons Attribution-ShareAlike license, located at *http://creativecommons.org/licenses/by-sa/3.0/legalcode*.

Day skybox and night skybox courtesy of Jockum Skoglund, also known as hipshot, hipshot@zfight.com,*http://www.zfight.com*.

The image of the trace capture button is created and shared by the Android Open Source Project and is used according to terms described in the Creative Commons 2.5 Attribution License.

Copyright © 2013 The Pragmatic Programmers, LLC.
All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

Printed in the United States of America.
ISBN-13: 978-1-937785-34-5
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—June 2013

This chapter will continue the work that we started in the last chapter. As our game plan for this chapter, we'll first load and compile the shaders that we've defined, and then we'll link them together into an OpenGL program. We'll then be able to use this shader program to draw our air hockey table to the screen.

Let's open AirHockey1, the project we started in the previous chapter, and pick up from where we left off.

3.1 Loading Shaders

Now that we've written the code for our shaders, the next step is to load them into memory. To do that, we'll first need to write a method to read in the code from our resources folder.

Loading Text from a Resource

Create a new Java source package in your project, com.airhockey.android.util, and in that package, create a new class called TextResourceReader. Add the following code inside the class:

AirHockey1/src/com/airhockey/android/util/TextResourceReader.java

```
public static String readTextFileFromResource(Context context,
    int resourceId) {
    StringBuilder body = new StringBuilder();

    try {
        InputStream inputStream =
            context.getResources().openRawResource(resourceId);
        InputStreamReader inputStreamReader =
            new InputStreamReader(inputStream);
        BufferedReader bufferedReader = new BufferedReader(inputStreamReader);

        String nextLine;

        while ((nextLine = bufferedReader.readLine()) != null) {
            body.append(nextLine);
            body.append('\n');
        }
    } catch (IOException e) {
        throw new RuntimeException(
            "Could not open resource: " + resourceId, e);
    } catch (Resources.NotFoundException nfe) {
        throw new RuntimeException("Resource not found: " + resourceId, nfe);
    }

    return body.toString();
}
```

We've defined a method to read in text from a resource, readTextFileFromResource(). The way this will work is that we'll call readTextFileFromResource() from our code, and we'll pass in the current Android context and the resource ID. The Android context is required in order to access the resources. For example, to read in the vertex shader, we might call the method as follows: readTextFileFromResource(this.context, R.raw.simple_fragment_shader).

We also check for a couple of standard scenarios we might run into. The resource might not exist, or there might be an error trying to read the resource. In those cases, we trap the errors and throw a wrapped exception with an explanation of what happened. If this code fails and an exception does get thrown, we'll have a better idea of what happened when we take a look at the exception's message and the stack trace.

Don't forget to press Ctrl-Shift-O (⌘⇧O on a Mac) to bring in any missing imports.

Reading in the Shader Code

We're now going to add the calls to actually read in the shader code. Switch to AirHockeyRender.java and add the following code after the call to glClearColor() in onSurfaceCreated():

AirHockey1/src/com/airhockey/android/AirHockeyRenderer.java
```
String vertexShaderSource = TextResourceReader
    .readTextFileFromResource(context, R.raw.simple_vertex_shader);
String fragmentShaderSource = TextResourceReader
    .readTextFileFromResource(context, R.raw.simple_fragment_shader);
```

Don't forget to bring in the import for TextResourceReader. The code won't compile because we don't yet have a reference to an Android context. Add the following to the top of the class:

AirHockey1/src/com/airhockey/android/AirHockeyRenderer.java
```
private final Context context;
```

Change the beginning of the constructor as follows:

AirHockey1/src/com/airhockey/android/AirHockeyRenderer.java
```
public AirHockeyRenderer(Context context) {
    this.context = context;
```

We'll also have to change AirHockeyActivity.java to pass in the Android context. Open AirHockeyActivity.java and change the call to glSurfaceView.setRenderer() as follows:

AirHockey1/src/com/airhockey/android/AirHockeyActivity.java
```
glSurfaceView.setRenderer(new AirHockeyRenderer(this));
```

```
rendererSet = true;
```

An Activity is an Android context, so we pass in a reference to this.

Keeping a Log of What's Happening

As we start to write more involved code, it often helps a lot to see a trace of what's happening, just in case we've made a mistake somewhere. With Android, we can use the Log class to log everything to the system log, which we can then view in Eclipse using the LogCat view.

We don't always want to log everything, so let's add a new class called LoggerConfig to com.airhockey.android.util with the following code:

AirHockey1/src/com/airhockey/android/util/LoggerConfig.java

```
package com.airhockey.android.util;

public class LoggerConfig {
    public static final boolean ON = true;
}
```

Whenever we want to log something, we'll check to see if this constant is true or false. To turn logging on or off, all we have to do is update the constant and recompile the application.

3.2 Compiling Shaders

Now that we've read in the shader source from our files, the next step is to compile each shader. We'll create a new helper class that is going to create a new OpenGL shader object, compile our shader code, and return the shader object for that shader code. Once we have this boilerplate code in place, we'll be able to reuse it in our future projects.

To begin, create a new class, ShaderHelper, and add the following code inside the class:

AirHockey1/src/com/airhockey/android/util/ShaderHelper.java

```
private static final String TAG = "ShaderHelper";

public static int compileVertexShader(String shaderCode) {
    return compileShader(GL_VERTEX_SHADER, shaderCode);
}

public static int compileFragmentShader(String shaderCode) {
    return compileShader(GL_FRAGMENT_SHADER, shaderCode);
}

private static int compileShader(int type, String shaderCode) {
```

```
}
```

We'll use this as the base for our shader helper. As before, don't forget to bring in the imports. If you are having issues with the static imports, please see Section 1.5, *Using Static Imports*, on page ?; we'll follow this style for the rest of the book.

In the next section, we'll build up compileShader() step by step:

Creating a New Shader Object

The first thing we should do is create a new shader object and check if the creation was successful. Add the following code to compileShader():

AirHockey1/src/com/airhockey/android/util/ShaderHelper.java

```
final int shaderObjectId = glCreateShader(type);

if (shaderObjectId == 0) {
    if (LoggerConfig.ON) {
        Log.w(TAG, "Could not create new shader.");
    }

    return 0;
}
```

We create a new shader object with a call to glCreateShader() and store the ID of that object in shaderObjectId. The type can be GL_VERTEX_SHADER for a vertex shader, or GL_FRAGMENT_SHADER for a fragment shader. The rest of the code is the same either way.

Take note of how we create the object and check if it's valid; this pattern is used everywhere in OpenGL:

1. We first create an object using a call such as glCreateShader(). This call will return an integer.
2. This integer is the reference to our OpenGL object. Whenever we want to refer to this object in the future, we'll pass the same integer back to OpenGL.
3. A return value of 0 indicates that the object creation failed and is analogous to a return value of null in Java code.

If the object creation failed, we'll return 0 to the calling code. Why do we return 0 instead of throwing an exception? Well, OpenGL doesn't actually throw any exceptions internally. Instead, we'll get a return value of 0 or OpenGL will inform us of the error through glGetError(), a method that lets us

ask OpenGL if any of our API calls have resulted in an error. We'll follow the same convention to stay consistent.

To learn more about glGetError() and other ways of debugging your OpenGL code, see Appendix 2, *Debugging*, on page ?.

Uploading and Compiling the Shader Source Code

Let's add the following code to upload our shader source code into the shader object:

AirHockey1/src/com/airhockey/android/util/ShaderHelper.java

```
glShaderSource(shaderObjectId, shaderCode);
```

Once we have a valid shader object, we call glShaderSource(shaderObjectId, shaderCode) to upload the source code. This call tells OpenGL to read in the source code defined in the String shaderCode and associate it with the shader object referred to by shaderObjectId. We can then call glCompileShader(shaderObjectId) to compile the shader:

```
glCompileShader(shaderObjectId);
```

This tells OpenGL to compile the source code that was previously uploaded to shaderObjectId.

Retrieving the Compilation Status

Let's add the following code to check if OpenGL was able to successfully compile the shader:

```
final int[] compileStatus = new int[1];
glGetShaderiv(shaderObjectId, GL_COMPILE_STATUS, compileStatus, 0);
```

To check whether the compile failed or succeeded, we first create a new int array with a length of 1 and call it compileStatus. We then call glGetShaderiv(shaderObjectId, GLES20.GL_COMPILE_STATUS, compileStatus, 0). This tells OpenGL to read the compile status associated with shaderObjectId and write it to the 0^{th} element of compileStatus.

This is another common pattern with OpenGL on Android. To retrieve a value, we often use arrays with a length of 1 and pass the array into an OpenGL call. In the same call, we tell OpenGL to store the result in the array's first element.

Retrieving the Shader Info Log

When we get the compile status, OpenGL will give us a simple yes or no answer. Wouldn't it also be interesting to know what went wrong and where we screwed up? It turns out that we can get a human-readable message by

calling glGetShaderInfoLog(shaderObjectId). If OpenGL has anything interesting to say about our shader, it will store the message in the shader's info log.

Let's add the following code to get the shader info log:

AirHockey1/src/com/airhockey/android/util/ShaderHelper.java
```
if (LoggerConfig.ON) {
    // Print the shader info log to the Android log output.
    Log.v(TAG, "Results of compiling source:" + "\n" + shaderCode + "\n:"
        + glGetShaderInfoLog(shaderObjectId));
}
```

We print this log to Android's log output, wrapping everything in an if statement that checks the value of LoggerConfig.ON. We can easily turn off these logs by flipping the constant to *false*.

Verifying the Compilation Status and Returning the Shader Object ID

Now that we've logged the shader info log, we can check to see if the compilation was successful:

AirHockey1/src/com/airhockey/android/util/ShaderHelper.java
```
if (compileStatus[0] == 0) {
    // If it failed, delete the shader object.
    glDeleteShader(shaderObjectId);

    if (LoggerConfig.ON) {
        Log.w(TAG, "Compilation of shader failed.");
    }

    return 0;
}
```

All we need to do is check if the value returned in the step *Retrieving the Compilation Status, on page 11*, is 0 or not. If it's 0, then compilation failed. We no longer need the shader object in that case, so we tell OpenGL to delete it and return 0 to the calling code. If the compilation succeeded, then our shader object is valid and we can use it in our code.

That's it for compiling a shader, so let's return the new shader object ID:

AirHockey1/src/com/airhockey/android/util/ShaderHelper.java
```
return shaderObjectId;
```

Compiling the Shaders from Our Renderer Class

Now it's time to make good use of the code that we've just created. Switch to AirHockeyRenderer.java and add the following code to the end of onSurfaceCreated():

AirHockey1/src/com/airhockey/android/AirHockeyRenderer.java

```
int vertexShader = ShaderHelper.compileVertexShader(vertexShaderSource);
int fragmentShader = ShaderHelper.compileFragmentShader(fragmentShaderSource);
```

Let's review the work we've done in this section. First we created a new class, ShaderHelper, and added a method to create and compile a new shader object. We also created LoggerConfig, a class to help us turn logging on and off at one single point.

If you take a look again at ShaderHelper, you'll see that we actually defined three methods:

compileShader()
: The compileShader(shaderCode) method takes in source code for a shader and the shader's type. The type can be GL_VERTEX_SHADER for a vertex shader, or GL_FRAGMENT_SHADER for a fragment shader. If OpenGL was able to successfully compile the shader, then this method will return the shader object ID to the calling code. Otherwise it will return zero.

compileVertexShader()
: The compileVertexShader(shaderCode) method is a helper method that calls compileShader() with shader type GL_VERTEX_SHADER.

compileFragmentShader()
: The compileVertexShader(shaderCode) method is a helper method that calls compileShader() with shader type GL_FRAGMENT_SHADER.

As you can see, the meat of the code is within compileShader(); all the other two methods do is call it with either GL_VERTEX_SHADER or GL_FRAGMENT_SHADER.

• Click HERE to purchase this book now. discuss