CHAPTER 5
Graphics

Computer graphics are used for any kind of display for which there isn't a GUI component: charting, displaying pictures, and so on. Graphics tools are used to create GUI components as well as to draw shapes, lines, pictures, etc. Android is well provisioned for graphics, including a full implementation of OpenGL ES, a subset of OpenGL intended for smaller devices.

This chapter starts with a recipe for using a custom font for special text effects, followed by some recipes on OpenGL graphics and one on graphical "touch" input. From there we continue the input theme with various image capture techniques. Then we have some recipes on graphics files, and one to round out the chapter discussing "pinch to zoom," using user touch input to scale graphical output.

5.1 Using a Custom Font

Ian Darwin

Problem

The range of fonts on Android devices is pretty small. You want something better.

Solution

Install a TTF or OTF version of your font in *assets/fonts* (creating this directory if necessary). In your code, create a typeface from the "asset" and call the `View`'s `setTypeface()` method. You're done!

Discussion

You can provide one or more fonts with your application. We have not yet discovered a documented way to install system-wide fonts. Beware of huge font files, as they will be downloaded with your application, increasing its size.

Your custom font's format should be TTF or OTF (TrueType or OpenType, a TTF extension). You need to create a *fonts* subdirectory under *assets* in your project, and install the font there.

While you can refer to the predefined fonts just using XML, you cannot refer to your own fonts using XML. This may change someday, but for now the content model of the `android:typeface` attribute is an XML enumeration containing only `normal`, `sans`, `serif`, and `monospace`—that's it! Therefore, you have to use code.

There are several `Typeface.create()` methods, including:

- `create(String familyName, int style)`
- `create(TypeFace family, inst style)`
- `createFromAsset(AssetManager mgr, String path)`
- `createFromFile(File path)`
- `createFromFile(String path)`

You can see how most of these should work. The parameter `style` is, as in Java, one of several constants defined on the class representing fonts, here `Typeface`. You can create representations of the built-in fonts, and variations on them, using the first two forms in the list. The code in Example 5-2 uses the `createFromAsset()` method, so we don't have to worry about font locations. If you have stored the file on internal or external storage (see Recipe 10.1), you could provide a `File` object or the font file's absolute path using the last two forms in the list. If the font file is on external storage, remember to request permission in *AndroidManifest.xml*.

I used the nice Iceberg font, from SoftMaker Software GmbH. This font is copyrighted and I do not have permission to redistribute it, so when you download the project and want to run it, you will need to install a TrueType font file at *assets/fonts/fontdemo.ttf*. Note that if the font is missing, the `createFromAsset()` method will return `null`; the online version of the code provides error handling. If the font is invalid, Android will *silently ignore it* and use a built-in font.

In this demo we provide two text areas, one using the built-in serif font and one using a custom font. They are defined, and various attributes added, in *main.xml* (see Example 5-1).

Example 5-1. XML layout with font specification

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    >
<TextView
    android:id="@+id/PlainTextView"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="@string/plain"
    android:textSize="36sp"
    android:typeface="serif"
    android:padding="10sp"
    android:gravity="center"
    />
<TextView
    android:id="@+id/FontView"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="@string/nicer"
    android:textSize="36sp"
    android:typeface="normal"
    android:padding="10sp"
    android:gravity="center"
    />
</LinearLayout>
```

Example 5-2 shows the source code.

Example 5-2. Setting a custom font

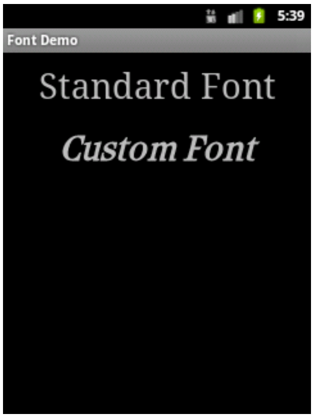
```
public class FontDemo extends Activity {

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);

        TextView v = (TextView) findViewById(R.id.FontView); ❶
        Typeface t = Typeface.createFromAsset(getAssets(),   ❷
                "fonts/fontdemo.ttf");
        v.setTypeface(t, Typeface.BOLD_ITALIC);              ❸
    }
}
```

❶ Find the `View` you want to use your font in.

❷ Create a `Typeface` object from one of the `Typeface` class's static `create()` methods.

❸ Message the `Typeface` into the `View`'s `setTypeface()` method.

If all is well, running the app should look like Figure 5-1.



Standard Font

Custom Font

Figure 5-1. Custom font

Source Download URL

The source code for this example is in the Android Cookbook repository, in the subdirectory *FontDemo* (see "Getting and Using the Code Examples" on page 18).

5.2 Drawing a Spinning Cube with OpenGL ES

Marco Dinacci

Problem

You want to create a basic OpenGL ES application.

Solution

Create a `GLSurfaceView` and a custom `Renderer` that will draw a spinning cube.

Discussion

Android supports 3D graphics via the OpenGL ES API, a flavor of OpenGL specifically designed for embedded devices. This recipe is not an OpenGL tutorial; it assumes the reader already has basic OpenGL knowledge. The final result will look like Figure 5-2.

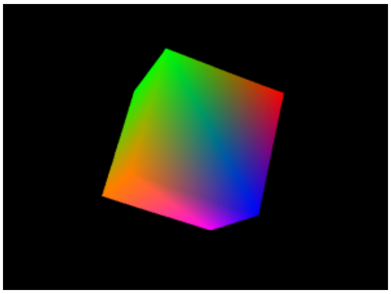


Figure 5-2. OpenGL graphics sample

First we write a new `Activity`, and in the `onCreate()` method we create the two fundamental objects we need to use the OpenGL API: a `GLSurfaceView` and a `Renderer` (see Example 5-3).

Example 5-3. OpenGL demo Activity

```
public class OpenGLDemoActivity extends Activity {

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        // Go fullscreen
        requestWindowFeature(Window.FEATURE_NO_TITLE);
        getWindow().setFlags(WindowManager.LayoutParams.FLAG_FULLSCREEN,
                        WindowManager.LayoutParams.FLAG_FULLSCREEN);

        GLSurfaceView view = new GLSurfaceView(this);
        view.setRenderer(new OpenGLRenderer());
        setContentView(view);
    }
}
```

Example 5-4 is the code for our `Renderer`; it uses a simple `Cube` object we'll describe later to display a spinning cube.

Example 5-4. The renderer implementation

```
class OpenGLRenderer implements Renderer {

        private Cube mCube = new Cube();
        private float mCubeRotation;

        @Override
        public void onSurfaceCreated(GL10 gl, EGLConfig config) {
            gl.glClearColor(0.0f, 0.0f, 0.0f, 0.5f);
            gl.glClearDepthf(1.0f);
            gl.glEnable(GL10.GL_DEPTH_TEST);
            gl.glDepthFunc(GL10.GL_LEQUAL);

            gl.glHint(GL10.GL_PERSPECTIVE_CORRECTION_HINT,
                      GL10.GL_NICEST);

        }

        @Override
        public void onDrawFrame(GL10 gl) {
            gl.glClear(GL10.GL_COLOR_BUFFER_BIT | GL10.GL_DEPTH_BUFFER_BIT);
            gl.glLoadIdentity();

            gl.glTranslatef(0.0f, 0.0f, -10.0f);
            gl.glRotatef(mCubeRotation, 1.0f, 1.0f, 1.0f);

            mCube.draw(gl);

            gl.glLoadIdentity();

            mCubeRotation -= 0.15f;
        }

        @Override
        public void onSurfaceChanged(GL10 gl, int width, int height) {
            gl.glViewport(0, 0, width, height);
            gl.glMatrixMode(GL10.GL_PROJECTION);
            gl.glLoadIdentity();
            GLU.gluPerspective(gl, 45.0f, (float)width / (float)height, 0.1f, 100.0f);
            gl.glViewport(0, 0, width, height);

            gl.glMatrixMode(GL10.GL_MODELVIEW);
            gl.glLoadIdentity();
        }
}
```

Our `onSurfaceChanged()` and `onDrawFrame()` methods are basically the equivalent of the GLUT `glutReshapeFunc()` and `glutDisplayFunc()`. The first is called when the surface is resized—for instance, when the phone switches between landscape and portrait modes. The second is called at every frame, and that's where we put the code to draw our cube (see Example 5-5).

Example 5-5. The Cube class

```
class Cube {

    private FloatBuffer mVertexBuffer;
    private FloatBuffer mColorBuffer;
    private ByteBuffer  mIndexBuffer;

    private float 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
                                    };
private float colors[] = {
                                   0.0f,  1.0f,  0.0f,  1.0f,
                                   0.0f,  1.0f,  0.0f,  1.0f,
                                   1.0f,  0.5f,  0.0f,  1.0f,
                                   1.0f,  0.5f,  0.0f,  1.0f,
                                   1.0f,  0.0f,  0.0f,  1.0f,
                                   1.0f,  0.0f,  0.0f,  1.0f,
                                   0.0f,  0.0f,  1.0f,  1.0f,
                                   1.0f,  0.0f,  1.0f,  1.0f
                               };

private byte indices[] = {
                                  0, 4, 5, 0, 5, 1,
                                  1, 5, 6, 1, 6, 2,
                                  2, 6, 7, 2, 7, 3,
                                  3, 7, 4, 3, 4, 0,
                                  4, 7, 6, 4, 6, 5,
                                  3, 0, 1, 3, 1, 2
                                  };

public Cube() {
        ByteBuffer byteBuf = ByteBuffer.allocateDirect(vertices.length * 4);
        byteBuf.order(ByteOrder.nativeOrder());
        mVertexBuffer = byteBuf.asFloatBuffer();
        mVertexBuffer.put(vertices);
        mVertexBuffer.position(0);

        byteBuf = ByteBuffer.allocateDirect(colors.length * 4);
        byteBuf.order(ByteOrder.nativeOrder());
        mColorBuffer = byteBuf.asFloatBuffer();
        mColorBuffer.put(colors);
        mColorBuffer.position(0);

        mIndexBuffer = ByteBuffer.allocateDirect(indices.length);
        mIndexBuffer.put(indices);
        mIndexBuffer.position(0);
}

public void draw(GL10 gl) {
    gl.glFrontFace(GL10.GL_CW);

    gl.glVertexPointer(3, GL10.GL_FLOAT, 0, mVertexBuffer);
    gl.glColorPointer(4, GL10.GL_FLOAT, 0, mColorBuffer);

    gl.glEnableClientState(GL10.GL_VERTEX_ARRAY);
    gl.glEnableClientState(GL10.GL_COLOR_ARRAY);

    gl.glDrawElements(GL10.GL_TRIANGLES, 36, GL10.GL_UNSIGNED_BYTE,
                              mIndexBuffer);

            gl.glDisableClientState(GL10.GL_VERTEX_ARRAY);
            gl.glDisableClientState(GL10.GL_COLOR_ARRAY);
        }
    }
```

The `Cube` uses two `FloatBuffer` objects to store vertex and color information and a `ByteBuffer` to store the face indexes. In order for the buffers to work it is important to set their order according to the endianness of the platform, using the `order()` method. Once the buffers have been filled with the values from the arrays, the internal cursor must be restored to the beginning of the data using `buffer.position(0)`.

See Also

OpenGL ES.

5.3 Adding Controls to the OpenGL Spinning Cube

Marco Dinacci

Problem

You want to interact with an OpenGL polygon using your device's keyboard.

Solution

Create a custom `GLSurfaceView` and override the `onKeyUp()` method to listen to the `KeyEvent` created from a directional pad (D-pad).

Discussion

This recipe builds on Recipe 5.2 to show how to control the cube using a D-pad. We're going to increment the speed rotation along the x-axis and y-axis using the D-pad's directional keys. The biggest change from that recipe is that we now have our custom class that extends the `SurfaceView`. We do this so that we can override the `onKeyUp()` method and be notified when the user uses the D-pad.

The `onCreate()` method of our Activity looks like Example 5-6.

Example 5-6. The spinning cube Activity

```
public class SpinningCubeActivity2 extends Activity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        // go fullscreen
```