Chapter 4 Android Graphics and OpenGL ES 1.X

4.1 Graphics Classes

Android provides rich features for developers to create high-quality 2D and 3D graphics applications. Besides the conventional Java graphics interface, it supports OpenGL ES, an open source API meaning *Open Graphics Library for Embedded System*, which is widely used in 3D graphics modeling and creation of 3D scenes. We will discuss Android graphics programming with OpenGL ES in this chapter and Chapter 6. In this section, we discuss briefly basic graphics classes that we use often in our applications. For details of the classes, one can refer to the official Android developer site:

http://developer.android.com/reference/packages.html

From the site, we can find all the Android APIs and all the API classes.

4.1.1 Class *View* and Subclasses

Class *View* is public and is a child of *Object*. It is used as a basis for building user interface components. A *View* object consists of a rectangular area on the screen to handle events and drawings. *View* is the parent of widgets for creating interactive user interface (UI) components such as buttons, and text fields. Its child *ViewGroup* is the parent of layouts, which are invisible containers holding other *Views*:

```
public class View  extends Object
  implements Drawable.Callback KeyEvent.Callback AccessibilityEventSource

java.lang.Object
  |--android.view.View

Known Direct Subclasses
  AnalogClock, ImageView, KeyboardView, MediaRouteButton, ProgressBar,
  Space, SurfaceView, TextView, TextureView, ViewGroup, ViewStub
```

The views in a window are all organized in a single tree. We can add views either from code or by defining a tree of views in XML layout files. Once we have created a tree of views, we typically wish to perform a few types of common operations:

1. **Set properties**: Set the properties of view such as setting the text of a *TextView*. We can also set the properties that are known at build time in the XML layout files.
2. **Set focus**: Set moving focus in response to user input. We can call **requestFocus**() to force focus to a specific view.
3. **Set up listeners**: Set up listeners to be notified when something interesting happens to the view, such as gaining or losing focus, or button clicking.
4. **Set visibility**: Hide or show views using **setVisibility**(int).

We can implement a custom view by implementing some of the following methods:

Category	Methods	Description
Creation	**onFinishInflate**()	Called after a view and all of its children have been inflated from XML.
Layout	**onMeasure**(int, int)	Called to determine the size requirements of this view and its children.
	onLayout(boolean, int,int,int,int)	Called upon this view's assigning a size and position to its children.
	onSizeChanged(int, int,int,int)	Called upon the changing of the view size.
Drawing	**onDraw**(android.graphics.Canvas)	Called upon the view's rendering its content.
Event processing	**onKeyDown**(int,KeyEvent)	Called upon occurence of a new hardware key event.
	onKeyUp(int,KeyEvent)	Called upon occurence of a hardware key-up event.
	onTrackballEvent(MotionEvent)	Called upon a trackball motion event.
	onTouchEvent(MotionEvent)	Called upon a touch screen motion event.
Focus	**onFocusChanged**(boolean, int,android.graphics.Rect)	Called upon the view gaining or losing focus.
	onWindowFocusChanged(boolean)	Called when the window that contains the view gains or loses focus.
Attaching	**onAttachedToWindow**()	Called when the view is being attached to a window.
	onDetachedFromWindow()	Called when the view is begin detached from its window.
	onWindowVisibilityChanged(int)	Called when the visibility of the window that contains the view has changed.

A *View* object may have an integer ID associated with it. The ID is typically assigned in a layout XML file. It is used to find a specific view within the view tree. The following is a common pattern of using the ID:

1. Define a *Button* in the layout file and assign it a unique ID:

```
<Button
    android:id="@+id/my_button"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/my_button_text"/>
```

2. From the **onCreate** method of an *Activity*, find the *Button*:

```
Button myButton = (Button) findViewById(R.id.my_button);
```

4.1.2 Class *SurfaceView*

Class *SurfaceView* is a child of *View*. It provides a drawing surface embedded in a view hierarchy. A user can control the size and format of a *SurfaceView* object, but the class places the surface at the proper location on the screen:

```
java.lang.Object
  |- android.view.View
       |- android.view.SurfaceView

Known Direct Subclasses:
  GLSurfaceView, RSSurfaceView, VideoView
```

A surface object is ordered by its z-coordinate so that it is behind the window holding its *SurfaceView*. The *SurfaceView* object creates a hole in its window to display its surface. The transparent region that makes the surface visible depends on the layout positions in the view hierarchy.

We can access the underlying surface via the *SurfaceHolder* interface, which can be retrieved by calling **getHolder**(). The following is a summary of the class.

Public Methods

void **draw** (Canvas canvas)
It renders this view and its children to the given *Canvas*.
boolean **gatherTransparentRegion** (Region region)
It performs an optimization on the view hierarchy.
void **setSecure** (boolean isSecure)
It sets whether the content of the surface is to be viewed as secure, preventing it from appearing in screenshots or on non-secure displays.
void **setVisibility** (int visibility)
It sets whether the surface is visible or not.
Variable *visibility* is one of VISIBLE, INVISIBLE, or GONE.
void **setZOrderMediaOverlay** (boolean isMediaOverlay)
It sets whether the surface is overlayed on top of another surface.
void **setZOrderOnTop**(boolean onTop)
It sets whether the surface is overlayed on top of its window.
SurfaceHolder **getHolder**()
It returns the *SurfaceHolder* of the surface.

Protected Methods

void **dispatchDraw**(Canvas canvas)
It is called by **draw** to draw the child views.
void **onAttachedToWindow**()
It is called when the view is attached to a window.
void **onDetachedFromWindow**()
It is called when the view is detached from a window.
void **onMeasure**(int width, int height)
It measures the view to determine the width and the height.
void **onWindowVisibilityChanged**(int visibility)
It called when its window has changed its visibility.

Public Constructors

public **SurfaceView** (Context context)
public **SurfaceView** (Context context, AttributeSet attrs)
public **SurfaceView** (Context context, AttributeSet attrs, int defStyle)

4.1.3 Class *GLSurfaceView*

When we write an Android graphics application, we usually start by extending the class *GLSurfaceView*. It is a public class, a child of class *SurfaceView*, using the dedicated surface for rendering using OpenGL. It provides the following features:

1. Manages a surface that can be embedded in the Android view system.
2. Manages an EGL display, enabling OpenGL objects to be rendered on a surface.
3. Accepts a *Renderer* object provided by a user to do the actual rendering.
4. Renders on a dedicated thread to separate rendering from the tasks of an UI thread.
5. Supports both continuous and on-demand rendering.
6. Wraps, traces, and/or error-checks the OpenGL calls of the renderer.

The following are its relations with other related classes:

```
public class GLSurfaceView  extends SurfaceView
       implements SurfaceHolder.Callback

java.lang.Object
 |-- android.view.View
      |-- android.view.SurfaceView
           |-- android.opengl.GLSurfaceView
```

We typically use *GLSurfaceView* by extending it and overriding one or more of the *View* system input event methods. We often use the **set** methods to customize views.

We can call **setRenderer**(*Renderer*) to initialize *GLSurfaceView*. On the other hand, we can modify the default behavior of *GLSurfaceView* by calling one or more of the following methods prior to calling **setRenderer**:

setDebugFlags(int)
setEGLConfigChooser(boolean)
setEGLConfigChooser(EGLConfigChooser)
setEGLConfigChooser(int, int, int, int, int, int)
setGLWrapper(GLWrapper)

By default *GLSurfaceView* creates a *PixelFormat.RGB_888* format surface. We can also call **getHolder().setFormat**(PixelFormat.TRANSLUCENT) to set a translucent surface, which is usually a 32-bit-per-pixel surface with 8 bits per component.

There are a few steps involved in setting up a *GLSurfaceView*.

Choosing an EGL Configuration

An Android device may support multiple *EGLConfig* rendering configurations, which may differ in the number of available data channels, and the number of bits allocated to each channel. Therefore, *GLSurfaceView* has to first choose what *EGLConfig* to use, and by default it chooses an *EGLConfig* that has an RGB_888 pixel format, with at least a 16-bit depth buffer and without any stencil buffer. We can always choose a different *EGLConfig* by calling one of the **setEGLConfigChooser** methods to override the default behavior.

Debug Behavior

We can optionally modify the *GLSurfaceView* behavior by calling one or more of the debugging methods, **setDebugFlags(int)** , and **setGLWrapper**(*GLSurfaceView.GLWrapper*), which may be called before and/or after **setRenderer**. Normally the methods are called before **setRenderer** so that they take effect immediately.

Setting a Renderer

Finally, we need to register a *GLSurfaceView.Renderer* by calling **setRenderer**(*GLSurfaceView.Renderer*). The renderer will do the actual OpenGL rendering.

Rendering Mode

Once we have setup the renderer, we can choose to draw continuously or on-demand by calling **setRenderMode(int)**. The default is continuous rendering.

Activity Life-cycle

A *GLSurfaceView* must be notified when the associated activity is paused or resumed. *GLSurfaceView* clients are required to call **onPause**() when the activity pauses and **onResume**() when the activity resumes. These calls let *GLSurfaceView* to pause and resume the rendering thread. They also allow *GLSurfaceView* to release and recreate the OpenGL display.

Handling events

To handle an event we normally extend *GLSurfaceView* and override the appropriate method. We may need to communicate with the *Renderer* object running in the rendering thread. We can do this using any standard Java thread communication techniques, or calling **queueEvent**(*Runnable*) as shown in the following example:

```
class MyGLSurfaceView extends GLSurfaceView {

   private MyRenderer myRenderer;

   public void start() {
     myRenderer = ...;
     setRenderer ( myRenderer );
   }

   public boolean onKeyDown ( int key, KeyEvent event ) {
     if ( key == KeyEvent.KEYCODE_DPAD_CENTER ) {
        queueEvent(new Runnable() {
        // Method  called on rendering thread:
        public void run() {
          myRenderer.handleDpadCenter();
        }});
        return true;
      }
     return super.onKeyDown ( key, event );
   }
}
```

4.2 OpenGL ES

The Android framework supports both the OpenGL ES 1.0/1.1 and OpenGL ES 2.0 APIs. OpenGL ES, where ES is short for embedded system, is a flavor of the OpenGL specification tailored for embedded devices. OpenGL ES is royalty-free and cross-platform. Its APIs have full-function support for 2D and 3D graphics on embedded systems such as mobile phones and appliances. OpenGL ES 1.X uses the traditional fixed-pipeline architecture and emphasizes hardware acceleration of the API. It offers enhanced functionality, good image quality and high performance. OpenGL ES 2.X is for programmable hardware. It emphasizes a programmable 3D graphics pipeline and allows the user to create shader and program objects. With ES 2.X, one can also write vertex and fragment shaders in the OpenGL ES Shading Language. On the other hand, OpenGL ES 2.0 does not support the fixed function transformation and fragment pipeline that OpenGL ES 1.X supports.

Since Android 1.0, the Android framework has supported the OpenGL ES 1.0 and 1.1 API specifications. Starting from Android 2.2 (API Level 8), the framework supports the OpenGL ES 2.0 API specification. One can find the API specifications at the site,

http://developer.android.com/guide/topics/graphics/opengl.html

However, earlier Android emulators do not support OpenGL ES 2.0. In this chapter, our discussions focus on OpenGL ES 1.X, which has certain limitations. In particular, it does not support direct vertex handling. For example, there are no **glBegin/glEnd** and **glVertex*** functions. Some constants such as GL_POLYGONS and GL_QUADS are missing. We will discuss ES 2.0 in Chapter 6.

4.3 OpenGL ES 1.X

We will discuss an example of using OpenGL ES 1.0 in Android, which was adopted from an example presented in the tutorial section of the Android developer site at

http://developer.android.com/resources/tutorials/opengl/opengl-es10.html

4.3.1 Creating an Activity with GLSurfaceView

Before we start using OpenGL to create graphics in Android, we have to implement the class *GLSurfaceView*, which extends *SurfaceView*, and *GLSurfaceView.Renderer*, which is responsible for making OpenGL calls to render a frame. Typically, a *GLSurfaceView* client has a class implementing this interface, and calls **setRenderer**(*GLSurfaceView.Renderer*) to register the renderer with the GLSurfaceView. The renderer is handled by a separate thread, so that the main thread, which normally provides user interface is decoupled from the rendering performance. Clients typically have to communicate with the renderer via the main thread that interacts with the user.

We will use Android 4.3 (Level 18) in our example and use Eclipse IDE to create the activity *GLSurfaceView*:

1. In Eclipse, choose **File** > **New** > **Project** > **Android Application Project**.
2. Enter the following information for the *New Android Project*:

Project Name:	HelloES
Application Name:	HelloES
Package Name:	opengl.es10

   For other entries, use the defaults. Then click **Next** > **Next** > **Next**.
3. Choose *Blank Activity* and click **Next**.
4. Use the default names for *Blank Activity*:

Activity Name:	MainActivity
Layout Name:	activity_main
Navigation Type:	None

   Click **Finish** to create the project *HelloES*
5. Project **HelloES** should appear in *Package Explorer* of Eclipse. Choose **HelloES** > **src** > **opengl.es10** > **MainActivity.java** to open the file "MainActivity.java". Modify this file as follows:

```
package opengl.es10;

import android.app.Activity;
```

```
import android.os.Bundle;
import android.content.Context;
import android.opengl.GLSurfaceView;

public class MainActivity extends Activity {
  private GLSurfaceView mGLView;

  @Override
  public void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      // Create a GLSurfaceView instance and set it
      // as the ContentView for this Activity.
      mGLView = new HelloESSurfaceView(this);
      setContentView(mGLView);
  }

  @Override
  protected void onPause() {
      super.onPause();
      // The following call pauses the rendering thread.
      mGLView.onPause();
  }

  @Override
  protected void onResume() {
      super.onResume();
      // The following call resumes a paused rendering thread.
      mGLView.onResume();
  }

  class HelloESSurfaceView extends GLSurfaceView {

      public HelloESSurfaceView(Context context){
          super(context);

          // Set the Renderer for drawing on the GLSurfaceView
          setRenderer(new HelloESRenderer());
      }
  }
}
```

Note that you should see an error indicator at *setRenderer(new HelloESRenderer());* . This is because up to this point, we have not defined the class *HelloESRenderer*.

In the *MainActivity* class shown above, we use a single *GLSurfaceView* for its view; the class also implements callbacks for pausing and resuming activities. The *HelloESSurfaceView* class is responsible for setting the renderer to draw on the *GLSurfaceView*.

6. In Eclipse, choose **File** > **New** > **File** and enter "HelloESRenderer.java" for *File name* and click **Finish** to create a new file for the following class *HelloESRenderer*, which implements the *GLSurfaceView.Renderer* interface:

```
package opengl.es10;
import javax.microedition.khronos.egl.EGLConfig;
import javax.microedition.khronos.opengles.GL10;
import android.opengl.GLSurfaceView;
```

```
public class HelloESRenderer implements GLSurfaceView.Renderer {

  public void onSurfaceCreated(GL10 gl, EGLConfig config) {
    // Set the background frame color to blue
    gl.glClearColor(0.0f, 0.0f, 0.9f, 1.0f);
    // Enable use of vertex arrays
    gl.glEnableClientState(GL10.GL_VERTEX_ARRAY);
  }
  public void onDrawFrame(GL10 gl) {
    // Redraw background color
    gl.glClear(GL10.GL_COLOR_BUFFER_BIT | GL10.GL_DEPTH_BUFFER_BIT);
  }
  public void onSurfaceChanged(GL10 gl, int width, int height) {
    gl.glViewport(0, 0, width, height);
  }
}
```

7. Now you can run the application by choosing **Run** > **Run** > **Android Application** and click **OK**. The Android emulator will start and will show a blue background screen. The code above is mostly self-explained. The functions **glClear**(), **glClearColor**(), and **glViewport**() are the standard OpneGL commands. (If you are not familiar with OpenGL commands, you may refer to the book *An Introduction to 3D Computer Graphics, Stereoscopic Image, and Animation in OpenGL and C/C++* by Fore June, for a brief quick introduction.) The only non-OpenGL functions are the few used by Android to do the initialization, which include the following:

   - **onSurfaceCreated**() is called once for setting up the *GLSurfaceView* environment.
   - **onDrawFrame**() is called whenever we redraw the *GLSurfaceView*. It is called to draw the current frame.
   - **onSurfaceChanged**() is called when the geometry of the **GLSurfaceView** changes. This is similar to the **glutPostRedisplay**() used in C programs.

4.3.2 Drawing a Triangle on GLSurfaceView

With the template code provided above, we should be able to make 2D or 3D graphics using OpenGL ES 1.X commands. We discuss here how to draw a triangle.

By default OpenGL ES assumes a world coordinate system where the center of the *GLSurfaceView* frame is at $(0, 0, 0)$; the coordinates of the lower left corner and upper right corner are at $(-1, -1, 0)$ and $(1, 1, 0)$ respectively. Therefore, as an example, we specify a triangle with the following vertex coordinates:

$$(-0.6, -0.5, 0), (0.6, -0.5, 0), (0.0, 0.5, 0)$$

We will display this triangle with green color on the blue background. To accomplish this, we modify the *HelloESRenderer* class to the following:

```
package opengl.es10;
import java.nio.*;
import javax.microedition.khronos.egl.EGLConfig;
import javax.microedition.khronos.opengles.GL10;
import android.opengl.GLSurfaceView;

public class HelloESRenderer implements GLSurfaceView.Renderer {
  private FloatBuffer triangle;
```

```
    public void onSurfaceCreated(GL10 gl, EGLConfig config) {
        // Set the background frame color to blue
        gl.glClearColor(0.0f, 0.0f, 0.9f, 1.0f);
        // initialize the triangle vertex array
        initShapes();
        // Enable use of vertex arrays
        gl.glEnableClientState(GL10.GL_VERTEX_ARRAY);
    }
    public void onDrawFrame(GL10 gl) {
        // Redraw background color
        gl.glClear(GL10.GL_COLOR_BUFFER_BIT | GL10.GL_DEPTH_BUFFER_BIT);
        // Draw the triangle using green color
        gl.glColor4f(0.0f, 1.0f, 0.0f, 0.0f);
        gl.glVertexPointer(3, GL10.GL_FLOAT, 0, triangle);
        gl.glDrawArrays(GL10.GL_TRIANGLES, 0, 3);
    }
    public void onSurfaceChanged(GL10 gl, int width, int height) {
        gl.glViewport(0, 0, width, height);
    }
    private void initShapes(){
        float vertices[] = {  // (x, y, z) of triangle
            -0.6f, -0.5f, 0,
             0.6f, -0.5f, 0,
             0.0f,  0.5f, 0
        };
        // initialize vertex Buffer for triangle
        // argument=(# of coordinate values * 4 bytes per float)
        ByteBuffer vbb = ByteBuffer.allocateDirect(vertices.length * 4);
        // use the device hardware's native byte order
        vbb.order(ByteOrder.nativeOrder());
        // create a floating point buffer from the ByteBuffer
        triangle = vbb.asFloatBuffer();
        // add the coordinates to the FloatBuffer
        triangle.put(vertices);
        // set the buffer to read the first vertex coordinates
        triangle.position(0);
    }
}
```

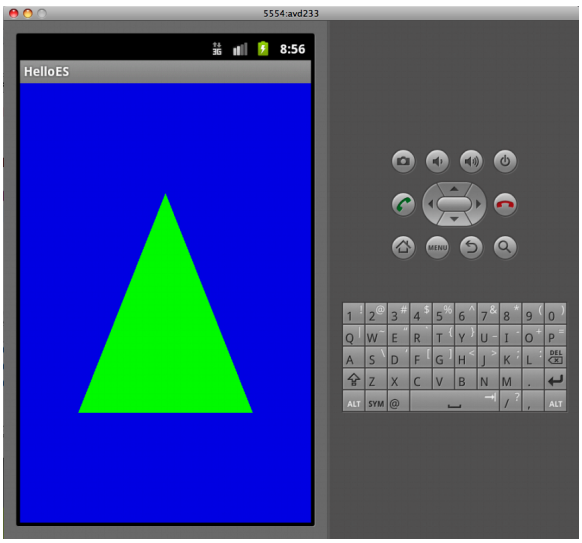


Figure 4-1 HelloES Graphics Output

(Do not forget to save the file by clicking the *save* icon, which is a small disk image.) We can re-

compile our graphics application by first clicking on the file **MainActivity.java** and then choosing **Run** in Eclipse. The Android emulator will run the application and we should see a green triangle displayed on a blue background as shown in Figure 4-1 above.

4.3.3 Setting Camera View and Transformations

Just as we do in a normal C program, we use **gluLookAt**() to set the camera view and position. The modelview transformation and projection transformation functions of basic OpenGL programming also apply here. As an example, we modify the functions **onDrawFrame**() and **onSurfaceChange**() of the class *HelloESRenderer* as follows to include projection and modelview transformations; our camera is at $(0, 0, 5)$ viewing along the negative z direction. We also scale the triangle by a factor of 3 along the y-direction and rotate it about the z-axis by 30^o. (Note again that we first scale, then rotate.)

```
import android.opengl.GLU;
....
public void onDrawFrame(GL10 gl) {
 // Redraw background color
 gl.glClear(GL10.GL_COLOR_BUFFER_BIT|GL10.GL_DEPTH_BUFFER_BIT);
 // Set GL_MODELVIEW transformation mode
 gl.glMatrixMode(GL10.GL_MODELVIEW);
 gl.glLoadIdentity();  //reset the matrix to its default state

 // When using GL_MODELVIEW, you must set the view point.
 //   camera at (0, 0, 5) look at (0,0,0), up = (0, 1, 0)
 GLU.gluLookAt(gl, 0, 0, 5, 0f, 0f, 0f, 0f, 1.0f, 0.0f);
 //rotate about z-axis for 30 degrees
 gl.glRotatef(30, 0, 0, 1);
 //magnify triangle by x3 in y-direction
 gl.glScalef ( 1, 3, 1);
 // Draw the triangle
 gl.glColor4f(0.0f, 1.0f, 0.0f, 0.0f);
 gl.glVertexPointer(3, GL10.GL_FLOAT, 0, triangle);
 gl.glDrawArrays(GL10.GL_TRIANGLES, 0, 3);
}

public void onSurfaceChanged(GL10 gl, int width, int height){
 gl.glViewport(0, 0, width, height);
 float aratio = (float) width / height;  //aspect ratio
  float l, r, b, t, n, f;    //left,right,bottom,top,near,far
  b = -1.5f; t = 1.5f; n = 3.0f; f = 7.0f;
  l =  b * aratio; r = t * aratio;
  gl.glMatrixMode(GL10.GL_PROJECTION);//set projection mode
  gl.glLoadIdentity();                // reset the matrix
  gl.glFrustumf( l, r, b, t, n, f);   //apply projection matrix
}
```

We also need to include the header statement "import android.opengl.GLU;" at the beginning of the file as we need to use the function **GLU.gluLookAt**(). After making the modifications, we can run the program **MainActivity.java** again. The output of the program is shown in Figure 4-2 below.

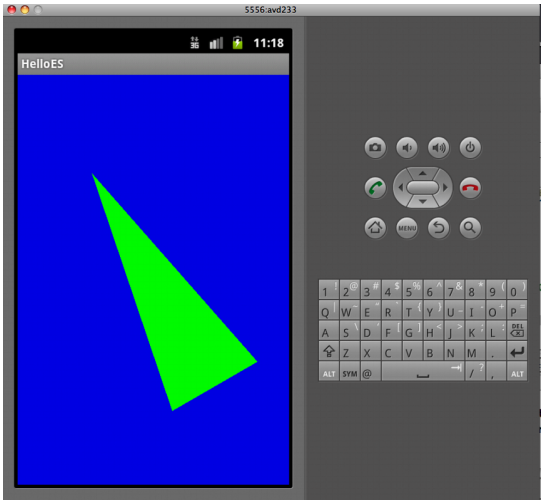


Figure 4-2 HelloES Output with Projection and Modelview Transformations

4.4 Animation and Event Handling

We can make use of the methods (functions) provided by the Android class *SystemClock* to create animated graphics. The class consists of core timekeeping facilities. To use the methods. we have to import the class by adding the header statement,

import android.os.SystemClock;

There are three clocks we can use to keep time. Each method returns a **long** data type:

1. **SystemClock.currentTimeMillis**() gives the current time and date expressed in milliseconds since the epoch. This clock can be set by the user or the phone network.

2. **SystemClock.uptimeMillis**() gives the active time lapse in milliseconds since the system was booted. This clock stops when the process is in a blocked or a sleep state like waiting for an I/O event or executing **Thread.sleep()**.

3. **SystemClock.elapsedRealtime**() gives the counts in milliseconds since the system was booted, including the time when the process is blocked or in a sleep state.

There are several ways to control timing events in an animation process:

1. **Thread.sleep**(*millis*) and **Object.wait**(*millis*) are standard blocking functions that can be used to generate desired time delays. When these functions are executed, the **uptimeMillis**() clock stops. The thread can be woken up by the function **Thread.interrupt**().

2. **SystemClock.sleep**(*millis*) is a utility function very similar to **Thread.sleep**(*millis*), except that it ignores **InterruptedException**.

3. We can use the *Handler* class to schedule asynchronous callbacks at an absolute or relative time. A handler object uses the **uptimeMillis**() clock to keep time. It requires an event loop to wait for an event to happen.

4. We can use the *AlarmManager* class to access the system alarm services such as triggering one-time or recurring events when the thread is in a blocked state.

As an example, let us rotate the triangle of Figure 4-2 discussed above for 6^o every second. To accomplish this, all we need to do is to add to the class *HelloESRenderer* a data member *angle* of type float:

public float angle = 0.0f;

Then we make the following modifications to the code of its member function **onDrawFrame**():

```
public void onDrawFrame(GL10 gl) {
  ....
  gl.glLoadIdentity();   // reset the matrix to its default state
  GLU.gluLookAt(gl, 0, 0, 5, 0f, 0f, 0f, 0f, 1.0f, 0.0f);
  SystemClock.sleep ( 1000 );  //delay for 1 second
  angle += 6;         //increment angle by 6 degrees
  //rotate triangle about z-axis
  gl.glRotatef(angle, 0.0f, 0.0f, 1.0f);
  //magnify triangle by x3 in y-direction
  gl.glScalef ( 1, 3, 1);
  // Draw the triangle
  .....
}
```

When we run the modified program, we will see the triangle of Figure 4-2 rotating anticlockwise 6^o every second.

If we want the triangle to interact with us rather than rotating automatically, we need to expand our implementation of *GLSurfaceView* to override the **onTouchEvent**() function to listen for touch events. Since we have defined above the data member *angle* of the *HelloESRenderer* class to be public, the member is exposed to other classes. We just need to modify the *HelloESSurfaceView* class to process touch events and pass the data to the renderer. To accomplish this, we have to include the import statement,

import android.view.MotionEvent;

In the **onDrawFrame**() function, we just comment out the time delay and increment statements as the *angle* value is determined by touch events:

```
public void onDrawFrame(GL10 gl) {
  ....
  // SystemClock.sleep ( 1000 );  //delay for 1 second
  // angle += 6;

  gl.glRotatef(angle, 0.0f, 0.0f, 1.0f);
}
```

Then we modify the *HelloESSurfaceView* class (in the file "MainActivity.java") as follows. We set the *renderer* member so that we have a handle to pass in rotation input and set the render mode to RENDERMODE_WHEN_DIRTY, which means that the method **onDrawFrame**() is not called unless something calls **requestRender**() explicitly for rendering. Consequently, setting rendering to this mode forbids the renderer to refresh automatically. We also override the **onTouchEvent**() method to listen for touch events and pass the parameters to our renderer:

```
class HelloESSurfaceView extends GLSurfaceView {
  private final float TOUCH_SCALE_FACTOR = 180.0f / 320;
  private HelloESRenderer renderer;
  private float previousX;
  private float previousY;

  public HelloESSurfaceView(Context context){
    super(context);
    // set the renderer member
    renderer = new HelloESRenderer();
    setRenderer(renderer);
```

```
    // Render the view only when there is a change; onDrawFrame
    // is not called unless requestRender() is called explicitly
    setRenderMode(GLSurfaceView.RENDERMODE_WHEN_DIRTY);
  }
  @Override
  public boolean onTouchEvent(MotionEvent e) {
    // MotionEvent reports input details from the touch screen
    // and other input controls. Here, we are only interested
    // in events where the touch position has changed.

    float x = e.getX();
    float y = e.getY();

    switch (e.getAction()) {
      case MotionEvent.ACTION_MOVE:

      float dx = x - previousX;
      float dy = y - previousY;

      // reverse direction of rotation above the mid-line
      if (y > getHeight() / 2)
        dx = dx * -1 ;

      // reverse direction of rotation to left of the mid-line
      if (x < getWidth() / 2)
        dy = dy * -1 ;

        renderer.angle += (dx + dy) * TOUCH_SCALE_FACTOR;
        requestRender();
    }

    previousX = x;
    previousY = y;
    return true;
  }
}
```

As we have set the render mode to *GLSurfaceView.RENDERMODE_WHEN_DIRTY*, the scene will be rendered only when there is a change in the scene. When we run the application, we should see the green triangle again. If we drag our mouse and move its cursor around the center in an anticlockwise direction, the triangle will rotate in a clockwise direction. Conversely, if we drag the mouse clockwise, the triangle rotates anticlockwise.

In summary, these sections give us an introduction of creating 2D and 3D graphics in the Android platform using OpenGL. We can find a lot more examples and resources of creating graphics in Android at its official site and associated links at

`http://developer.android.com/guide/topics/graphics/opengl.html`

4.5 Rendering a 3D Color Cube

In these few sections, we extend the example of the previous section where we rotate a 2D triangle when we drag the mouse. First, we discuss rendering a color cube, which can be rotated by dragging the mouse. Second, we discuss putting textures on the cube. In the process, we will discuss other related but stand-alone graphics topics.

4.5.1 Color Cube

A cube consists of 8 vertices and 6 faces, each of which is a square. OpenGL ES 1.X does not support primitives of GL_QUAD or GL_POLYGON. It only renders triangles. So to render any polygon other than a triangle, we have to first decompose it into triangles.

Any polygon has two faces: a front face and a back face. Whether a face is front or back depends on our winding convention, the way we order the vertices of the polygon. In general we want to specify the winding in a way that the back faces of an object are those facing the interior of the object and front faces are those facing the exterior. The winding of a front face could be clockwise or counterclockwise and can be specified by the command **gl.glFrontFace()**. For example,

```
gl.glFrontFace ( GL10.GL_CCW );
```

specifies that a face is a front face if the vertices of the triangle is ordered in the counter-clockwise (CCW) direction, and it is a back face if the vertices are ordered in the clockwise (CW) direction. In our discussion, we always consider a face with counter-clockwise winding to be a front face.

In practice, if an object is opaque, we do not want to display any of its back faces as they are facing the interior of the object. We can suppress rendering back faces using the commands,

```
gl.glEnable( GL10.GL_CULL_FACE );
gl.glCullFace ( GL10.GL_BACK );
```

Now consider a cube whose center is at the origin with length 2. The coordinates of the vertices of the face (square) at $z = 1$ are given by

$$v_4 = (-1, -1, 1),\ v_5 = (1, -1, 1),\ v_6 = (1, 1, 1),\ v_7 = (-1, 1, 1)$$

The front face of the square is specified by $v_4v_5v_6v_7$ which are CCW and its back face is specified by $v_4v_7v_6v_5$ which are CW when we observe the face from a point at $z > 1$. The square can be decomposed into two triangles as shown in Figure 4-3. When we make the decomposition, we must be careful that the windings of the triangles are consistent with that of the face considered.

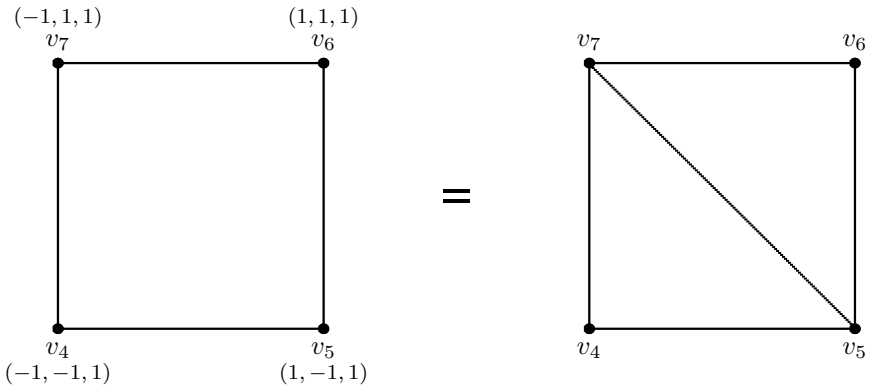


Figure 4-3. Decomposing a square into 2 triangles

In Figure 4-3, the vertices of a polygon are always specified in counterclockwise order.

Suppose we call this application and project *cube*. We follow the steps discussed in Section 4.3.1 to create an activity with *GLSurfaceView*. The file *MainActivity.java* of the previous section is slightly modified to the code shown in Listing 4-1 below.

Program Listing 4-1 *MainActivity.java* for Rendering Cube

```
package opengl.cube;

import android.app.Activity;
import android.os.Bundle;
import android.content.Context;
import android.opengl.GLSurfaceView;
import android.view.MotionEvent;

public class MainActivity extends Activity {
  private GLSurfaceView mGLView;

  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    mGLView = new CubeSurfaceView(this);
    setContentView(mGLView);
  }
  @Override
  protected void onPause() {
    super.onPause();
    // The following call pauses the rendering thread.
    mGLView.onPause();
  }
  @Override
  protected void onResume() {
    super.onResume();
    // The following call resumes a paused rendering thread.
    mGLView.onResume();
  }

  class CubeSurfaceView extends GLSurfaceView {
    private final float TOUCH_SCALE_FACTOR = 180.0f / 320;
    private CubeRenderer renderer;
    private float previousX;
    private float previousY;

    public CubeSurfaceView(Context context){
      super(context);
      // set the renderer member
      renderer = new CubeRenderer();
      setRenderer(renderer);
      // Render the view only when there is a change
      setRenderMode(GLSurfaceView.RENDERMODE_WHEN_DIRTY);
    }
    @Override
    public boolean onTouchEvent(MotionEvent e) {
      // MotionEvent reports input details from the touch screen
      // and other input controls. Here, we are only interested
      // in events where the touch position has changed.
      float x = e.getX();
      float y = e.getY();
```

```
        switch (e.getAction()) {
          case MotionEvent.ACTION_MOVE:
            float dx = x - previousX;
            float dy = y - previousY;

            // reverse direction of rotation above the mid-line
            if (y > getHeight() / 2)
               dx = dx * -1 ;
            // reverse direction of rotation to left of the mid-line
            if (x < getWidth() / 2)
               dy = dy * -1 ;
            renderer.angle += (dx + dy) * TOUCH_SCALE_FACTOR;
            requestRender();
        }
        previousX = x;
        previousY = y;
        return true;
      }
    }
}
```

As we can see from Listing 4-1, the class *CubeSurfaceView* extends *GLSurfaceView* and creates an object of the *CubeRenderer* class, which is discussed below, to render a color cube.

Listing 4-2 shows the complete code of the file *CubeRenderer.java*, which contains the code of the class *CubeRenderer* that implements *GLSurfaceView.Renderer*, and the code of the class *Cube* that has the attributes of a cube with length 2 and center at the origin of the coordinate system.

The methods **onSurfaceChanged** and **onDrawFrame** are basically the equivalent of the GLUT functions **glutReshapeFunc** and **glutDisplayFunc**. The former is called when there is a change in surface size like the case when the phone switches between landscape and portrait modes. The latter is called each time the cube is rendered.

The class *Cube* uses two *FloatBuffer* objects to store vertex and color data and a *ByteBuffer* to store the face indices. To understand the code, we may also refer to the comments of it, which give more detailed explanations of the processing of the data.

Program Listing 4-2 *CubeRenderer.java*

```
package opengl.cube;
import java.nio.ByteBuffer;
import java.nio.ByteOrder;
import java.nio.FloatBuffer;
import javax.microedition.khronos.egl.EGLConfig;
import javax.microedition.khronos.opengles.GL10;
import android.opengl.GLSurfaceView;
import android.opengl.GLU;
import android.os.SystemClock;
import android.view.MotionEvent;

public class CubeRenderer implements GLSurfaceView.Renderer {
  public float angle = 0.0f;       //rotation angle
  private Cube cube = new Cube();
```

```
    public void onSurfaceCreated(GL10 gl, EGLConfig config) {
      // Set the background frame color to grey, opaque
      gl.glClearColor(0.5f, 0.5f, 0.5f, 1.0f);
      gl.glEnable( GL10.GL_CULL_FACE ); //Enable culling faces
      gl.glCullFace ( GL10.GL_BACK );   //don't render back faces
    }

    public void onDrawFrame(GL10 gl) {
      // Redraw background color
      gl.glClear(GL10.GL_COLOR_BUFFER_BIT | GL10.GL_DEPTH_BUFFER_BIT);
      // Set GL_MODELVIEW transformation mode
      gl.glMatrixMode(GL10.GL_MODELVIEW);
      gl.glLoadIdentity();   // Reset the matrix to identity matrix
      // Move objects away from view point to observe
      gl.glTranslatef(0.0f, 0.0f, -10.0f);
      // Rotate about a diagonal of cube
      gl.glRotatef(angle, 1.0f, 1.0f, 1.0f);
      cube.draw(gl);         // Draw the cube
      gl.glLoadIdentity();   // Reset transformation matrix
    }
    @Override
    public void onSurfaceChanged(GL10 gl, int width, int height) {
      gl.glViewport(0, 0, width, height);
      gl.glMatrixMode(GL10.GL_PROJECTION);
      gl.glLoadIdentity();  // Reset projection matrix
      // Setup viewing volume
      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();  // Reset transformation matrix
    }
  }

  class Cube {
    private FloatBuffer vertexBuffer;
    private FloatBuffer colorBuffer;
    private ByteBuffer  indexBuffer;

    // Coordinates of 8 vertices of 6 cube faces
    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 };
    // Colors of vertices
    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 };

    // indices of 12 triangles (6 squares) in GL_CCW
    //  referencing vertices[] array coordinates
    private byte indices[] = {
```

```
      5, 4, 0,   1, 5, 0,   6, 5, 1,   2, 6, 1,
      7, 6, 2,   3, 7, 2,   4, 7, 3,   0, 4, 3,
      6, 7, 4,   5, 6, 4,   1, 0, 3,   2, 1, 3 };

  public Cube() {
    // initialize vertex Buffer for cube
    // argument=(# of coordinate values * 4 bytes per float)
    ByteBuffer byteBuf = ByteBuffer.allocateDirect(vertices.length * 4);
    byteBuf.order(ByteOrder.nativeOrder());
    // create a floating point buffer from the ByteBuffer
    vertexBuffer = byteBuf.asFloatBuffer();
    // add the vertices coordinates to the FloatBuffer
    vertexBuffer.put(vertices);
    // set the buffer to read the first vertex coordinates
    vertexBuffer.position(0);

    // Do the same to colors array
    byteBuf=ByteBuffer.allocateDirect(colors.length*4);
    byteBuf.order(ByteOrder.nativeOrder());
    colorBuffer = byteBuf.asFloatBuffer();
    colorBuffer.put(colors);
    colorBuffer.position(0);
    // indices are integers
    indexBuffer = ByteBuffer.allocateDirect(indices.length);
    indexBuffer.put(indices);
    indexBuffer.position(0);
  }

  // Typical drawing routine using vertex array
  public void draw(GL10 gl) {
    //Counterclockwise order for front face vertices
    gl.glFrontFace(GL10.GL_CCW);

    //Points to the vertex buffers
    gl.glVertexPointer(3, GL10.GL_FLOAT, 0, vertexBuffer);
    gl.glColorPointer(4, GL10.GL_FLOAT, 0, colorBuffer);

    //Enable client states
    gl.glEnableClientState(GL10.GL_VERTEX_ARRAY);
    gl.glEnableClientState(GL10.GL_COLOR_ARRAY);
    //Draw vertices as triangles
    gl.glDrawElements(GL10.GL_TRIANGLES, 36, GL10.GL_UNSIGNED_BYTE,
                                      indexBuffer);
    //Disable client state
    gl.glDisableClientState(GL10.GL_VERTEX_ARRAY);
    gl.glDisableClientState(GL10.GL_COLOR_ARRAY);
  }
}
```

In Listing 4-2 above, the **draw** method of class *Cube* is a typical OpenGL drawing routine using vertex array:

1. **gl.glVertexPointer** tells the renderer where to read the vertices coordinates and of what data type they are. The first parameter is the number of coordinates of a vertex and it is 3 here for

the (x, y, z) 3D coordinates. The second parameter tells that data type of each coordinate value is float. The third parameter, referred to as *stride*, is the offset between neighboring vertices in the array. A value of 0 indicates that array is tightly packed, not containing other data such as color values other than the vertex coordinates. The last parameter points to a buffer, *vertexBuffer* in our example, where the vertex coordinates are held.

2. **gl.glColorPointer** tells the renderer where to read the color data of the vertices. It works similarly to **gl.glVertexPointer**. The first parameter is 4 because a color tuple (r, g, b, a) consists of four values representing red, green, blue and alpha (transparency).
3. **gl.glEnableClientState** enables OpenGL to use a vertex array for rendering.
4. **gl.glDrawArrays** tells OpenGL to draw the primitive. In our code, the first parameter, GL10.GL_TRIANGLES tells the renderer to draw the vertices held in the vertex buffer as triangles. The second parameter specifies the number (*count*) of vertices. In our example, we have 6 faces. Each face has 2 triangles and each triangle has 3 vertices. Therefore, the total number of vertices is

$$count = 6 \times 2 \times 3 = 36$$

   The third parameter specifies the type of values in the array specified by the fourth parameter which is a pointer pointing to the location where the indices of vertex data are stored.
5. We may think of *glEnableClientState* and *glDisableClientState* as *begin ... end* statements in a program.

When we compile and run the app, we will see a color cube which may appear 2D as the viewing point is right in front of it. We will see its 3D shape when we drag the mouse, which rotates it. Figure 4-4 below shows an output of the app.

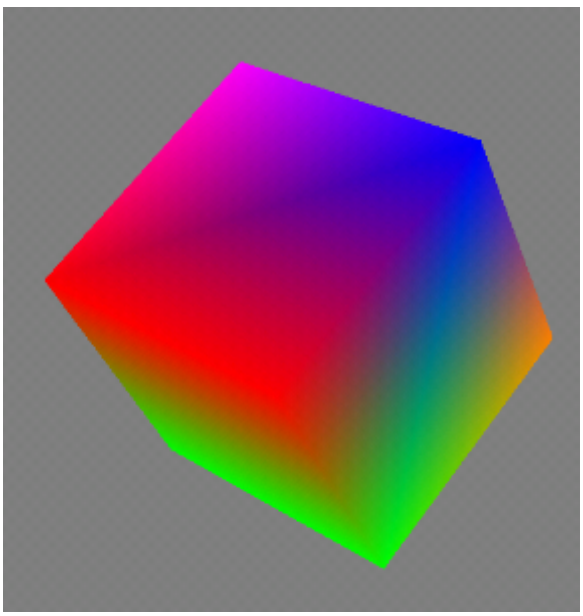


Figure 4-4 Color Cube

4.5.2 Rendering a Square Only

If we want to render a square only, we can define a class *Square* similar to the class *Cube* presented in Listing 4-2, except that we use the primitive GL_TRIANGLE_STRIP and method **gl.glDrawArrays** rather than **gl.glDrawElements** to render the two triangles. A triangle strip is a

series of connected triangles, two in our case. In this method, we only have to define four vertices for a square. Using the square shown in Figure 4-3 as an example, OpenGL first draws the triangle using vertices in the order of $v_4v_5v_7$, then it takes the last vertex v_7 from the previous triangle and uses the last side v_7v_5 of it as the basis for the new triangle which will be drawn in the order $v_7v_5v_6$. Listing 4-3 shows the code of class *Square*.

Program Listing 4-3 Class *Square*

```
class Square {
  private FloatBuffer vertexBuffer;   //buffer holding the vertices
  private float vertices[] = {        //Figure 4-3
    -1.0f, -1.0f,  1.0f,        // v4 - bottom left
     1.0f, -1.0f,  1.0f,        // v5 - bottom right
    -1.0f,  1.0f,  1.0f,        // v6 - top left
     1.0f,  1.0f,  1.0f         // v7 - top right
  };

  public Square() {
    ByteBuffer byteBuffer=ByteBuffer.allocateDirect(vertices.length*4);
    byteBuffer.order(ByteOrder.nativeOrder());
    vertexBuffer = byteBuffer.asFloatBuffer();
    vertexBuffer.put(vertices);
    vertexBuffer.position(0);
  }

  public void draw(GL10 gl) {
     gl.glFrontFace(GL10.GL_CCW);
     gl.glColor4f(1.0f, 1.0f, 1.0f, 1.0f);
     gl.glVertexPointer(3, GL10.GL_FLOAT, 0, vertexBuffer);
     gl.glEnableClientState(GL10.GL_VERTEX_ARRAY);

     gl.glDrawArrays(GL10.GL_TRIANGLE_STRIP, 0, vertices.length / 3);

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

The only other changes to render the square using the class *CubeRenderer* is to add the data member declaration statement

private Square square = new Square();

to the class and to replace *cube.draw()* by *square.draw()*. Upon running the code, we should see a white square over a grey background. Again we can drag the mouse to rotate the square.

4.5.3 Rendering a Rotating Cube

If we want to render a cube that rotates by itself, we simply do **not** set the render mode to *RENDERMODE_WHEN_DIRTY*. Then the method **onDrawFrame**(), which renders a frame, will be called automatically at a certain frame rate. We just need to increment the rotation angles in this method, assuming that each time **onDrawFrame** is called, the transformation matrix is first reset to the identity matrix. The following code, which can be part of the file *MainActivity.java* shows such a renderer.

```
-----------------------------------------------------------------------
class CubeRenderer1 implements Renderer
{
  GL10 gl;
  Cube cube = new Cube();
  private float anglex;
  private float anglez;
  private final int nfaces = 12;
  //@Override
  //Refresh automatically as RENDERMODE_WHEN_DIRTY is not used
  public void onDrawFrame(GL10 gl)
  {
        gl.glClear(GL10.GL_COLOR_BUFFER_BIT | GL10.GL_DEPTH_BUFFER_BIT);
        gl.glEnableClientState(GL10.GL_VERTEX_ARRAY);
        gl.glEnableClientState(GL10.GL_COLOR_ARRAY);
        gl.glMatrixMode(GL10.GL_MODELVIEW);
        gl.glLoadIdentity();
        gl.glTranslatef(0.0f, 0.0f, -3.0f);
        gl.glRotatef( anglex, 1.0f, 0.0f, 0.0f );  // Rotate about  x-axis
        gl.glRotatef( anglez, 0.0f, 0.0f, 1.0f);   // Rotate about z-axis
        cube.draw(gl);
        anglex += 1.0f;
        anglez += 2.0f;
        gl.glDisableClientState(GL10.GL_VERTEX_ARRAY);
        gl.glDisableClientState(GL10.GL_COLOR_ARRAY);
  }


  public void onSurfaceChanged(GL10 gl, int width, int height)
  {
        gl.glViewport(0, 0, width, height);
        float ratio = (float) width / height;
        gl.glMatrixMode(GL10.GL_PROJECTION);
        gl.glLoadIdentity();
        gl.glFrustumf(-ratio, ratio, -1, 1, 1, 10);
  }

  public void onSurfaceCreated(GL10 gl, EGLConfig config)
  {
      gl.glDisable(GL10.GL_DITHER);
      gl.glHint(GL10.GL_PERSPECTIVE_CORRECTION_HINT, GL10.GL_FASTEST);
      gl.glClearColor(1, 1, 1, 0);
      gl.glEnable(GL10.GL_CULL_FACE);
      gl.glShadeModel(GL10.GL_SMOOTH);
      gl.glEnable(GL10.GL_DEPTH_TEST);
  }
}
-----------------------------------------------------------------------
```

The renderer can be called in the **onCreate**() method. Since we do not use DIRTY mode, the thread will automatically invoke **onDrawFrame**() at a certain frame rate.

```
-----------------------------------------------------------------------
public class MainActivity extends Activity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
```

```
        super.onCreate(savedInstanceState);

        // Create our Preview view and set it as the content of our
        // Activity

        mGLSurfaceView = new GLSurfaceView(this);
        CubeRenderer1 renderer = new CubeRenderer1();
        mGLSurfaceView.setRenderer(renderer);
        setContentView(mGLSurfaceView);
    }
-------------------------------------------------------------------------
```

4.6 Rendering a 3D Texture Cube

4.6.1 Displaying Images

Before discussing putting textures on the surfaces of an object, we discuss very **briefly** how to display images.

Android runs on a variety of devices with different screen sizes and resolutions, and supports the three common image formats PNG, JPG, and GIF. Images are saved in the directory *res/layout/drawable*.

As an example, let's create a project called *displayimage* and package *graphics.displayimage* using Eclipse IDE with its *Blank Activity* as we did above. Under the directory *res*, we can find the drawable subdirectories:

drawable-hdpi
drawable-ldpi
drawable-mdpi
drawable-xhdpi
drawable-xxhdpi

which correspond to different screen sizes and resolutions. In the naming convention of the subdirectories, letter 'h' refers to high resolution, 'm' to medium, 'l' to low, and 'x' to extra. So *xhdpi* means extra high resolution. Image file names should be lowercase and contain only letters, numbers, and underscores. Figure 4-5, obtained from the official Android web site, illustrates how Android roughly maps actual sizes and pixel densities to generalized sizes and resolutions.

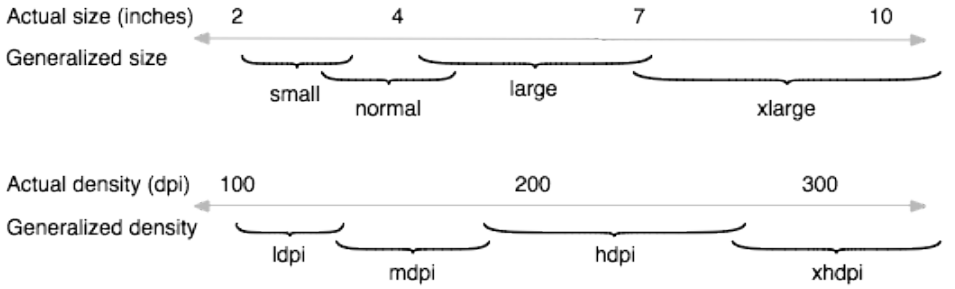


Figure 4-5 Android Screen Sizes and Resolutions

Supporting various screen resolutions allows us to create images at different dpi (dots per inch) to enhance the appearance of our application.

In our example, we want to display an image saved in the file *galaxy.jpg*, which is downloaded from the NASA web site and is an image of the galaxy. To achieve this, we first copy this file to

the directory *res/drawable-hdpi*, which also contains the icon file *ic_launcher.png*. (Alternatively, we can create a directory named *drawable* inside *res* and put the image file there.) Then we add the below *ImageView* layout component, the base element used for displaying images, to the *RelativeLayout* component of the file *activity_main.xml*, which is in the directory *res/layout*, after the *TextView*:

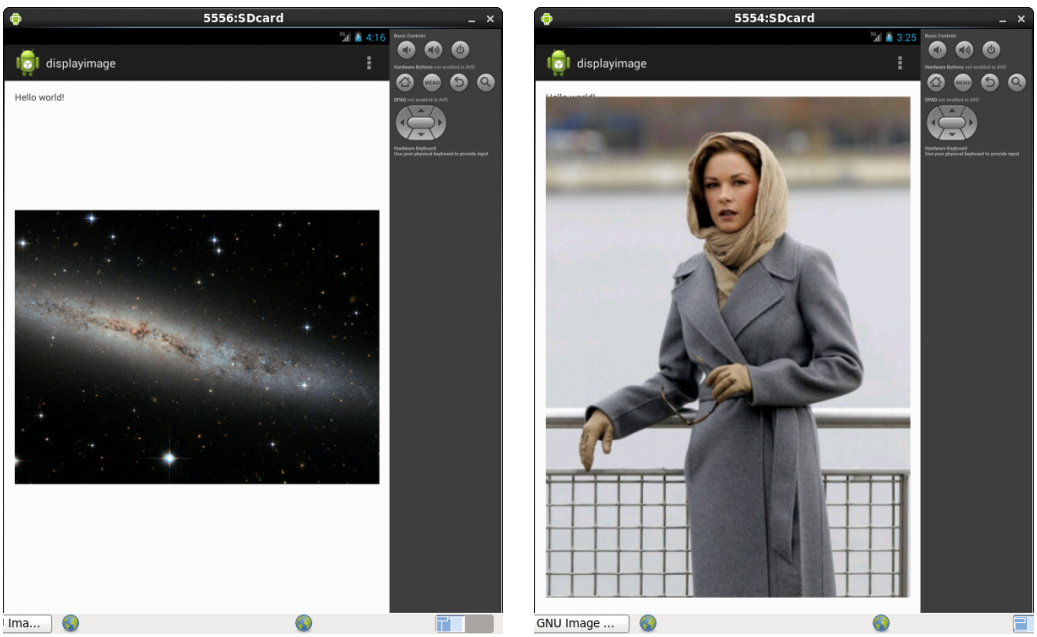
```
<ImageView
    android:id="@+id/galaxy_image"
    android:src="@drawable/galaxy"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent" />
```

In the file *MainActivity.java*, which is generated by Eclipse and contains the class *MainActivity*, we add an image import statement and in the method **onCreate** the *ImageView* statement:

```
import android.widget.ImageView;

public class MainActivity extends Activity {
  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    //Added statement for displaying image
    ImageView image = (ImageView) findViewById(R.id.galaxy_image);
  }
  ....
}
```

When we compile and run the program, we will see the *galaxy* image displayed on the Android screen as shown in Figure 4-6 (a) below.



(a) (b)

Figure 4-6 Output of *displayimage*

If we want to change to another image, say, *catherine.jpg*, we can use the **setImageResource** method of the class *ImageView* to load this new image. So we add a statement in the file *MainActivity.java*:

```
public class MainActivity extends Activity {
  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    //Added statement for displaying image
    ImageView image = (ImageView) findViewById(R.id.galaxy_image);
    //Change to another image
    image.setImageResource(R.drawable.catherine);
  }
}
```

The output is shown in Figure 4-6 (b).
Alternatively, we can use bitmap to change the display of an image:

```
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    ImageView image = (ImageView) findViewById(R.id.galaxy_image);
    Bitmap bitmap = BitmapFactory.decodeResource(this.getResources(),
        R.drawable.catherine);
    image.setImageBitmap( bitmap );
  }
```

In the code, the *BitmapFactory* class creates a bitmap object with the image *catherine.jpg*, which is saved in the directory *res/drawable-hdpi* (or in *res/drawable*). We use the method **ImageView.setImageBitmap()** to update the *ImageView* component.

4.6.2 Rendering Texture Square

Texture mapping is the mapping of a separately defined graphics image, a picture, or a pattern to a surface. The technique helps us to combine pixels with geometric objects to generate complex images without building large geometric models. For example, we can apply texture mapping to 'glue' an image of a brick wall to a polygon and to draw the entire wall as a single polygon.

To apply texture to a surface, we need to load up an image and tell the OpenGL renderer that the image will be used as a texture. We also need to tell the renderer where exactly onto our square we want to "glue" it.

We use texture coordinates to specify the image, which is normalized to size 1×1. That is, any point on the image lies within the range $(0, 1) \times (0, 1)$. However, an Android system considers the upper left corner to be $(0, 0)$ and the lower right corner to be $(1, 1)$. On the other hand, we have set the lower left corner of our square to $(-1, -1, 1)$ and upper right corner to $(1, 1, 1)$. This situation is illustrated in Figure 4-7 below.

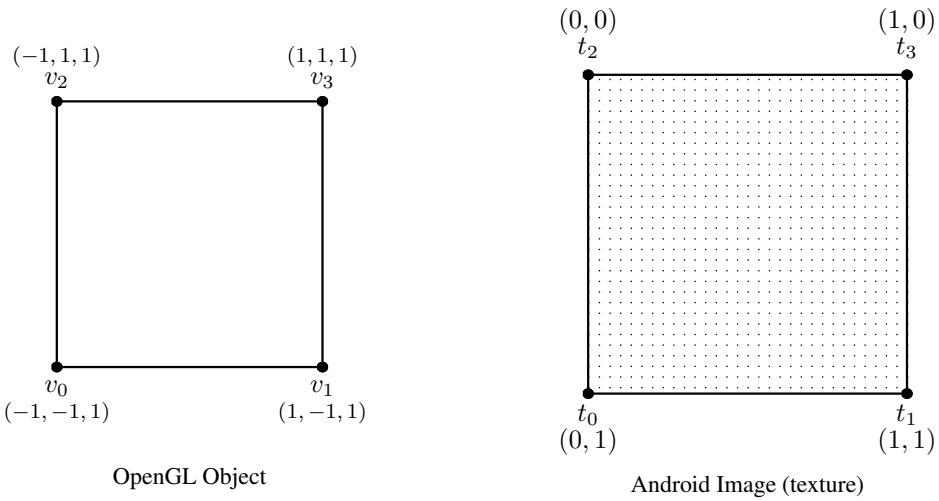


Figure 4-7. Vertex Coordinates and Texture Coordinates

We want to make the mapping of the texture image to the square object vertices in the following way:

$$t_0(0,1) \longrightarrow v_0(-1,-1,1)$$
$$t_1(1,1) \longrightarrow v_1(1,-1,1)$$
$$t_2(0,0) \longrightarrow v_2(-1,1,1)$$
$$t_3(1,0) \longrightarrow v_3(1,1,1)$$

Therefore, we define and initilialize two arrays, one for vertex coordinates and one for texture coordinates in the class *Square*:

```
class Square {
  private FloatBuffer vertexBuffer;   // buffer holding vertices coord
  private FloatBuffer textureBuffer;  // buffer holding texture coord
  private float vertices[] = {        // vertex coordinates
     -1.0f, -1.0f,  1.0f,             // v0 - bottom left
      1.0f, -1.0f,  1.0f,             // v1 - bottom right
     -1.0f,  1.0f,  1.0f,             // v2 - top left
      1.0f,  1.0f,  1.0f              // v3 - top right
  };
  private float texture[] = {         // texture coordinates
      0.0f, 1.0f,                     // bottom left     (t0)
      1.0f, 1.0f,                     // bottom right    (t1)
      0.0f, 0.0f,                     // top left        (t2)
      1.0f, 0.0f,                     // top right       (t3)
  };
  ....
}
```

The above code also shows that we have added the variable *textureBuffer*, which works in a way similar to the *vertexBuffer*.

We modify the *Square* constructor to initialize the buffering of texture data; we will just reuse the variable *byteBuffer*. Also, we define another integer array, *texHandles* to hold the handle to the texture that we will create, and add another method, **initTexture**, which will be called from the renderer in the **onSurfaceCreated** method. The **initTexture** method initializes the OpenGL texture commands.

```
class Square {
  ....
```

```
  private int[] texHandles = new int[1];  //holds handle to textures

  public Square() {
    ByteBuffer byteBuffer = ByteBuffer.allocateDirect(vertices.length*4);
    byteBuffer.order(ByteOrder.nativeOrder());
    vertexBuffer = byteBuffer.asFloatBuffer();
    vertexBuffer.put(vertices);
    vertexBuffer.position(0);

    byteBuffer = ByteBuffer.allocateDirect(texture.length * 4);
    byteBuffer.order(ByteOrder.nativeOrder());
    textureBuffer = byteBuffer.asFloatBuffer();
    textureBuffer.put(texture);
    textureBuffer.position(0);
  }

  public void initTexture(GL10 gl, Context context) {
    // loading texture
    Bitmap bitmap=BitmapFactory.decodeResource(context.getResources(),
          R.drawable.catherine);
    // generate one texture handle
    gl.glGenTextures(1, texHandles, 0);
    // ...and bind it to our array
    gl.glBindTexture(GL10.GL_TEXTURE_2D, texHandles[0]);
    // create nearest filtered texture
    gl.glTexParameterf(GL10.GL_TEXTURE_2D,
                       GL10.GL_TEXTURE_MIN_FILTER,GL10.GL_NEAREST);
    gl.glTexParameterf(GL10.GL_TEXTURE_2D,
                       GL10.GL_TEXTURE_MAG_FILTER, GL10.GL_LINEAR);
    //Use Android GLUtils to specify a 2D texture image from bitmap
    GLUtils.texImage2D(GL10.GL_TEXTURE_2D, 0, bitmap, 0);
    // Clean up
    bitmap.recycle();
  }
 ....
}
```

The **initTexture** method generates a texture handle, binds it to a 2D texture array, loads the image *catherine.jgp* from the the directory *res/drawable-hdpi*. The loaded image will be used as the texture for subsequent operations.

The **draw** method of *Square* is slightly modified to incorporate operations of the texture:

```
class Square {
  ....
  public void draw(GL10 gl) {
    // Vertices of a front face are in counterclockwise order
    gl.glFrontFace(GL10.GL_CCW);
    gl.glColor4f(1.0f, 1.0f, 1.0f, 1.0f);
    // bind the previously generated texture
     gl.glBindTexture(GL10.GL_TEXTURE_2D, texHandles[0]);

    gl.glVertexPointer(3, GL10.GL_FLOAT, 0, vertexBuffer);
    gl.glTexCoordPointer(2, GL10.GL_FLOAT, 0, textureBuffer);
    gl.glEnableClientState(GL10.GL_VERTEX_ARRAY);
    gl.glEnableClientState(GL10.GL_TEXTURE_COORD_ARRAY);
```

```
    gl.glDrawArrays(GL10.GL_TRIANGLE_STRIP, 0, vertices.length / 3);

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

The renderer class now looks like the following:

```
public class SquareRenderer implements GLSurfaceView.Renderer {

  public float angle = 0.0f;
  private Square square = new Square();
  private Context context;

  public SquareRenderer ( Context context0 )
  {
    context = context0;
  }

  public void onSurfaceCreated(GL10 gl, EGLConfig config) {
    // Set the background frame color to grey
    gl.glClearColor(0.5f, 0.5f, 0.5f, 1.0f);
    // Do not render back faces
    gl.glEnable( GL10.GL_CULL_FACE );
    gl.glCullFace ( GL10.GL_BACK );
    square.initTexture(gl, context);
    gl.glEnable(GL10.GL_TEXTURE_2D);
  }
  .....
}
```

When we run the app, we'll see the output of Figure 4-8; the image *catherine.jpg* shown in Figure 4-6 (b) has been glued on a square.

Figure 4-8 Square with Texture

4.6.3 Rendering Texture Cube

To render a cube with texture, we simply put a texture image on each of the 6 faces (squares) of the cube. So it is a straightforward extension of rendering a texture square discussed above.

Since we need to handle 6 images, it is more convenient to reference the images using resource ids. The method **getResources().getIdentifier** of the class *Context* returns the integer ID of an image file in a resource directory. To use this method in our program, we need to add the following import statement:

```
import android.content.res.Resources;
```

Suppose we have put 6 different image files (in .png or .jpg format) in the directory *res/drawable-hdpi*. We can use the following code to obtain their ids:

```
int nFaces = 6;
int rids[] = new int[nFaces];  //resource ids
//Image filenames, omitting extension (.jpg or .png)
String img[] = {"catherine", "lu", "galaxy", "lu1", "lu2", "zhouxun"};
for ( int i = 0; i < nFaces; i++ ){
  rids[i] = context.getResources().getIdentifier( img[i] , "drawable",
            context.getPackageName());
  .....
}
```

The initialization of texture operations, the loading of an image and the rendering of a face with texture are the same as what we did in rendering a square discussed above except that now we need 6 texture handles, 6 vertex buffers and 6 texture buffers. Listing 4-3 shows the code for the modified *cube* class.

Program Listing 4-3 Class *Cube* with Texture

```
class Cube {
  //6 faces
  private final int nFaces = 6;

  private FloatBuffer vertexBuffer[] = new FloatBuffer[nFaces];
  // buffer holding the texture coordinates
  private FloatBuffer textureBuffer[] = new FloatBuffer[nFaces];
  private int[] texHandles = new int[nFaces];

  // Coordinates of 6 cube faces
  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},     //front
     {  1.0f, -1.0f,-1.0f, -1.0f, -1.0f, -1.0f,
        1.0f,  1.0f,-1.0f, -1.0f,  1.0f, -1.0f},     //back
     { -1.0f, -1.0f,-1.0f, -1.0f, -1.0f,  1.0f,
       -1.0f,  1.0f,-1.0f, -1.0f,  1.0f,  1.0f},     //left
     {  1.0f, -1.0f, 1.0f,  1.0f, -1.0f, -1.0f,
        1.0f,  1.0f, 1.0f,  1.0f,  1.0f, -1.0f},     //right
     { -1.0f, -1.0f,-1.0f,  1.0f, -1.0f, -1.0f,
       -1.0f, -1.0f, 1.0f,  1.0f, -1.0f,  1.0f},     //bottom
     { -1.0f,  1.0f, 1.0f,  1.0f,  1.0f,  1.0f,
       -1.0f,  1.0f, -1.0f, 1.0f,  1.0f, -1.0f} };   //top
```

```
private float texture[][] = {
   // Mapping texture coordinates for the vertices
   { 0.0f, 1.0f, 1.0f, 1.0f, 0.0f, 0.0f, 1.0f, 0.0f},
   { 0.0f, 1.0f, 1.0f, 1.0f, 0.0f, 0.0f, 1.0f, 0.0f},
   { 0.0f, 1.0f, 1.0f, 1.0f, 0.0f, 0.0f, 1.0f, 0.0f},
   { 0.0f, 1.0f, 1.0f, 1.0f, 0.0f, 0.0f, 1.0f, 0.0f},
   { 0.0f, 1.0f, 1.0f, 1.0f, 0.0f, 0.0f, 1.0f, 0.0f},
   { 0.0f, 1.0f, 1.0f, 1.0f, 0.0f, 0.0f, 1.0f, 0.0f} };

public Cube() {
   // initialize vertex Buffer for each  cube face
   // argument=(# of coordinate values * 4 bytes per float)
   for ( int i = 0; i < nFaces; i++ )  // do for 6 faces
   {
     ByteBuffer byteBuf=ByteBuffer.allocateDirect(vertices[i].length*4);
     byteBuf.order(ByteOrder.nativeOrder());
     // create a floating point buffer from the ByteBuffer
     vertexBuffer[i] = byteBuf.asFloatBuffer();
     // add the coordinates to the FloatBuffer
     vertexBuffer[i].put(vertices[i]);
     // set the buffer to read the first vertex coordinates
     vertexBuffer[i].position(0);

     byteBuf = ByteBuffer.allocateDirect(texture[i].length * 4);
     byteBuf.order(ByteOrder.nativeOrder());
     textureBuffer[i] = byteBuf.asFloatBuffer();
     textureBuffer[i].put(texture[i]);
     textureBuffer[i].position(0);
   }
}

public void initTexture(GL10 gl, Context context) {
   // loading texture
   // generate 6 texture pointers
   gl.glGenTextures(nFaces, texHandles, 0);
   // get resource ids of images in res/drawable-hdpi
   int rids[] = new int[nFaces];  //resource ids
   // image file names
   String img[] = {"catherine","lu","galaxy","lu1","lu2","zhouxun"};
   // Initialize texture feature for 6 (nFaces) faces
   for ( int i = 0; i < nFaces; i++ ){
     rids[i] = context.getResources().getIdentifier( img[i],"drawable",
                    context.getPackageName());
     // loading texture
     Bitmap bitmap =
        BitmapFactory.decodeResource(context.getResources(),rids[i]);
     gl.glBindTexture(GL10.GL_TEXTURE_2D, texHandles[i]);
     // create nearest filtered texture
     gl.glTexParameterf(GL10.GL_TEXTURE_2D,GL10.GL_TEXTURE_MIN_FILTER,
        GL10.GL_NEAREST);
     gl.glTexParameterf(GL10.GL_TEXTURE_2D, GL10.GL_TEXTURE_MAG_FILTER,
        GL10.GL_LINEAR);
     //Use Android GLUtils to specify a two-dimensional texture image
     GLUtils.texImage2D(GL10.GL_TEXTURE_2D, 0, bitmap, 0);
```

```
        //cleanup
        bitmap.recycle();
      }
  }

  // Typical drawing routine using vertex array
  public void draw(GL10 gl) {
    // Vertices of a front face are in counterclockwise order
    gl.glFrontFace(GL10.GL_CCW);

    for ( int i = 0; i < nFaces; i++) {
      gl.glBindTexture(GL10.GL_TEXTURE_2D, texHandles[i]);
      gl.glVertexPointer(3, GL10.GL_FLOAT, 0, vertexBuffer[i]);
      gl.glTexCoordPointer(2, GL10.GL_FLOAT, 0, textureBuffer[i]);
      gl.glEnableClientState(GL10.GL_VERTEX_ARRAY);
      gl.glEnableClientState(GL10.GL_TEXTURE_COORD_ARRAY);

      gl.glDrawArrays(GL10.GL_TRIANGLE_STRIP, 0, vertices[i].length / 3 );
      gl.glDisableClientState(GL10.GL_VERTEX_ARRAY);
      gl.glDisableClientState(GL10.GL_TEXTURE_COORD_ARRAY);
    }
  }
}
```

We slightly modify the method **onDrawFrame** of the class *CubeRenderer* to display three cubes at the same time so that we can see the six different faces when we drag the mouse to rotate them:

```
  public class CubeRenderer implements GLSurfaceView.Renderer
  {
    public float angle = 0.0f;
    private Cube cube = new Cube();
    private Context context;

    public CubeRenderer ( Context context0 ) {
       context = context0;
    }

    public void onSurfaceCreated(GL10 gl, EGLConfig config) {
        // Set the background frame color to grey
        gl.glClearColor(0.5f, 0.5f, 0.5f, 1.0f);
        // Do not render back faces
        gl.glEnable( GL10.GL_CULL_FACE );
        gl.glCullFace ( GL10.GL_BACK );
        cube.initTexture(gl, context);
        gl.glEnable(GL10.GL_TEXTURE_2D);
    }

    public void onDrawFrame(GL10 gl) {
      // Redraw background color
      gl.glClear(GL10.GL_COLOR_BUFFER_BIT | GL10.GL_DEPTH_BUFFER_BIT);
      // Set GL_MODELVIEW transformation mode
      gl.glMatrixMode(GL10.GL_MODELVIEW);
      gl.glLoadIdentity();   // Reset the matrix to identity matrix

      // Move objects away from view point to observe
```

```
        gl.glTranslatef(-1.0f, 2.0f, -10.0f);
        // Rotate about a diagonal of cube
        gl.glRotatef(angle, 1.0f, 1.0f, 0.0f);
        // Draw the cube
        cube.draw(gl);

        gl.glLoadIdentity(); //reset matrix, draw another cube
        gl.glTranslatef(0.0f, 0.0f, -10.0f);
        gl.glRotatef(angle, 0.0f, 1.0f, 1.0f);
        cube.draw(gl);

        gl.glLoadIdentity(); //reset matrix, draw another cube
        gl.glTranslatef(1.0f, -2.0f, -10.0f);
        gl.glRotatef(angle, -1.0f, -1.0f, 0.0f);
        cube.draw(gl);
        gl.glLoadIdentity();    // Reset matrix
    }
    .....
}
```

Figure 4-10 Texture Cubes

When we run the app, we should see on the screen 3 cubes with texture images on the faces. If we drag the mouse, we should see the cubes rotate and appear like those showniFigure 4-10 above.

Interested readers may download the complete code of this app from the website, *http://www.forejune.com/android/*

4.7 Rendering a Rotating Cube

Rotating a cube is similar to rotating a triangle that we have discussed in Section 4.4. As an example, let us rotate the 3D color cube discussed in Section 4.5 along a diagonal of the cube for 1^o every 0.1 second. To accomplish this, we first do **not** set the render mode to *GLSurfaceView.RENDERMODE_WHEN_DIRTY* so that the frames will be updated automatically. Second, we let the thread sleep for 0.1 second and then increment a rotation angle by 1^o in the method **onDrawFrame**() of the class *CubeRenderer*:

```
public void onDrawFrame(GL10 gl)
{
   // Redraw background color
   gl.glClear(GL10.GL_COLOR_BUFFER_BIT | GL10.GL_DEPTH_BUFFER_BIT);
   // Set GL_MODELVIEW transformation mode
   gl.glMatrixMode(GL10.GL_MODELVIEW);
   gl.glLoadIdentity();   // Reset the matrix to identity matrix

   SystemClock.sleep ( 100 );  //delay for 0.1 second
   angle += 1;         //increment angle by 6 degrees

   // Move objects away from view point to observe
   gl.glTranslatef(0.0f, 0.0f, -10.0f);
   // Rotate about a diagonal of cube
   gl.glRotatef(angle, 1.0f, 1.0f, 1.0f);
   // Draw the cube
   cube.draw(gl);
}
```

When we run the app, we will see a color cube rotating on the screen. Figure 4-11 below shows two frames of it.

(a) (b)

Figure 4-11 Two Frames of a Rotating Cube