CHAPTER – 6

GRAPHICS AND MULTIMEDIA

Graphics and Animations:

Android SDK provides a set of API for drawing custom 2D and 3D graphics. When we write an app that requires graphics, we should consider how intensive the graphic usage is. In other words, there could be an app that uses quite static graphics without complex effects and there could be other app that uses intensive graphical effects like games. According to this usage, there are different techniques we can adopt:

- **Canvas and Drawable:**

In this case, we can extend the existing UI widgets so that we can customize their behavior or we can create custom 2D graphics using the standard method provided by the Canvas class.

- **Hardware Acceleration:**

We can use hardware acceleration when drawing with the Canvas API. This is possible from Android 3.0.

- **OpenGL:**

Android supports OpenGL natively using NDK. This technique is very useful when we have an app that uses intensively graphic contents (i.e. games).

The easiest way to use 2D graphics is extending the View class and overriding the onDraw() method. We can use this technique when we do not need a graphics intensive app.

In this case, we can use the Canvas class to create 2D graphics. This class provides a set of method starting with draw that can be used to draw different shapes like:

- Lines
- Circle
- Rectangle
- Oval
- Picture
- Arc

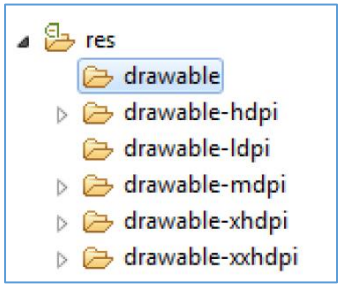
For example, let's suppose we want do draw a rectangle. We create a custom view and then we override onDraw() method. Here we draw the rectangle:

```
public class TestView extends View {
    public TestView(Context context) {
        super(context);
    }
    public TestView(Context context, AttributeSet attrs, int defStyle) {
```

```
        super(context, attrs, defStyle);
    }
    public TestView(Context context, AttributeSet attrs) {
        super(context, attrs);
    }
    @Override
    protected void onDraw(Canvas canvas) {
        super.onDraw(canvas);
        Paint p = new Paint();
        p.setColor(Color.GREEN);
        p.setStrokeWidth(1);
    }
```

Drawable:

In Android, a Drawable is a graphical object that can be shown on the screen. From API point of view all the Drawable objects derive from *Drawable* class. They have an important role in Android programming and we can use XML to create them. They differ from standard widgets because they are not interactive, meaning that they do not react to user touch. Images, colors, shapes, objects that change their aspect according to their state, object that can be animated are all drawable objects. In Android under res directory, there is a sub-dir reserved for Drawable, it is called *res/drawable.*



Under the drawable dir we can add binary files like images or XML files. We can create several directories according to the screen density we want to support. These directories have a name like drawable <>. This is very useful when we use images; in this case, we have to create several image versions: for example, we can create an image for the high dpi screen or another one for medium dpi screen. Once we have our file under drawable directory, we can reference it, in our class, using

R.drawable.file_name

While it is very easy add a binary file to one of these directory, it is a matter of copy and paste, if we want to use a XML file we have to create it. There are several types of drawable:

- Bitmap
- Nine-patch
- State list
- Level list
- Transition drawable

- Inset drawable
- Clip drawable
- Scale drawable
- Shape drawable

Shape Drawable:

This is a generic shape. Using XML we have to create a file with shape element as root. This element as an attribute called android:shape, where we define the type of shape like rectangle, oval, line and ring. We can customize the shape using child elements like:

For example, let us suppose we want to create an oval with solid background color. We create a XML file called formexample *oval.xml:*

```
<shape xmlns:android="http://schemas.android.com/apk/res/android"
android:shape="oval" >
<solid android:color="#FF0000" />
<size
android:height="100dp"
android:width="120dp" />
</shape>
```

Multi-touch and Gestures:

Multi-Touch:

Multi-touch gesture happens when more than one finger touches the screen at the same time. Android allows us to detect these gestures. Android system generates the following touch events whenever multiple fingers touches the screen at the same time.

- **ACTION_DOWN:** For the first pointer that touches the screen. This starts the gesture.
- **ACTION_POINTER_DOWN:** For extra pointers that enter the screen beyond the first.
- **ACTION_MOVE:** A change has happened during a press gesture.
- **ACTION_POINTER_UP:** Sent when a non-primary pointer goes up.
- **ACTION_UP:** Sent when the last pointer leaves the screen.

So in order to detect any of the above mention event, we need to override **onTouchEvent()** method and check these events manually.

```
public boolean onTouchEvent(MotionEvent ev){
    final int actionPeformed = ev.getAction();

    switch(actionPeformed){
        case MotionEvent.ACTION_DOWN:{
            break;
        }

        case MotionEvent.ACTION_MOVE:{
            break;
        }
        return true;
    }
}
```

In these cases, we can perform any calculation we like. For example, zooming, shrinking etc. In order to get the co-ordinates of the X and Y axis, we can call **getX()** and **getY()** method. Its syntax is given below:

- final float x = ev.getX();
- final float y = ev.getY();

Apart from these methods, there are other methods provided by this MotionEvent class for better dealing with multitouch. These methods are listed below:

- **getAction():** This method returns the kind of action being performed
- **getPressure():** This method returns the current pressure of this event for the first index
- **getRawX():** This method returns the original raw X coordinate of this event
- **getRawY():** This method returns the original raw Y coordinate of this event
- **getSize():** This method returns the size for the first pointer index
- **getSource():** This method gets the source of the event
- **getXPrecision():** This method return the precision of the X coordinates being reported
- **getYPrecision():** This method return the precision of the Y coordinates being reported

Gestures:

Gestures allow users to interact with our app by manipulating the screen objects we provide. The following table shows the core gesture set that is supported in Android.

Touch: Triggers the default functionality for a given item.

- **Action**: Press, lift

Long Press: Enters data selection mode. Allows us to select one or more items in a view and act upon the data using a contextual action bar. Avoid using long press for showing contextual menus.

- **Action:** Press, wait, lift

Swipe or Drag: Scrolls overflowing content, or navigates between views in the same hierarchy. Swipes are quick and affect the screen even after the finger is picked up. Drags are slower and more precise, and the screen stops responding when the finger is picked up.

- **Action**: Press, move, lift

Long Press Drag: Rearranges data within a view, or moves data into a container (e.g. folders on Home Screen).

- **Action**: Long press, move, lift

Double Touch: Scales up the smallest targetable view, if available, or scales a standard amount around the gesture. Also used as a secondary gesture for text selection.

- **Action**: Two touches in quick succession

Double Touch Drag: Scales content by pushing away or pulling closer, centered around gesture.

- **Action**: A single touch followed in quick succession by a drag up or down:
  - ✓ Dragging up decreases content scale
  - ✓ Dragging down increases content scale
  - ✓ Reversing drag direction reverses scaling.

Pinch Open: Zooms into content.

- **Action**: 2-finger press, move outwards, lift

Pinch Close: Zooms out of content.

- **Action**: 2-finger press, move inwards, lift

Detecting Common Gestures:

A "touch gesture" occurs when a user places one or more fingers on the touch screen, and our application interprets that pattern of touches as a particular gesture. There are correspondingly two phases to gesture detection:

1. Gathering data about touch events.
2. Interpreting the data to see if it meets the criteria for any of the gestures our app supports.

Detect Gestures:

Android provides the GestureDetector class for detecting common gestures. Some of the gestures it supports include onDown(), onLongPress(), onFling(), and so on. We can use GestureDetector in conjunction with the onTouchEvent() method described above.

Detecting All Supported Gestures:

When we instantiate a GestureDetectorCompat object, one of the parameters it takes is a class that implements the GestureDetector.OnGestureListener interface. GestureDetector.OnGestureListener notifies users when a particular touch event has occurred. To make it possible for our GestureDetector object to receive events, we override the View or Activity's onTouchEvent() method, and pass along all observed events to the detector instance.

Multimedia:

The Android SDK provides a set of APIs to handle multimedia files, such as audio, video and images. Moreover, the SDK provides other API sets that help developers to implement interesting graphics effects, like animations and so on.

The modern smart phones and tablets have an increasing storage capacity so that we can store music files, video files, images. etc. Not only the storage capacity is important, but also the high definition camera makes it possible to take impressive photos. In this context, the Multimedia API plays an important role.

Multimedia API:

Android supports a wide list of audio, video and image formats. We can give a look here to have an idea; just to name a few formats supported:

Audio:

- MP3
- MIDI
- Vorbis (es: mkv)

Video:

- H.263
- MPEG-4 SP

Images:

- JPEG
- GIF
- PNG

All the classes provided by the Android SDK that we can use to add multimedia capabilities to our apps are under the *android.media package*.

In this package, the heart class is called MediaPlayer. This class has several methods that we can use to play audio and video file stored in our device or streamed from a remote server. This class implements a state machine with well-defined states and we have to know them before playing a file. Simplifying the state diagram, as shown in the official documentation, we can define these macro-states:

1. **Idle State:**

When we create a new instance of the MediaPlayer class.

2. **Initialization State**:

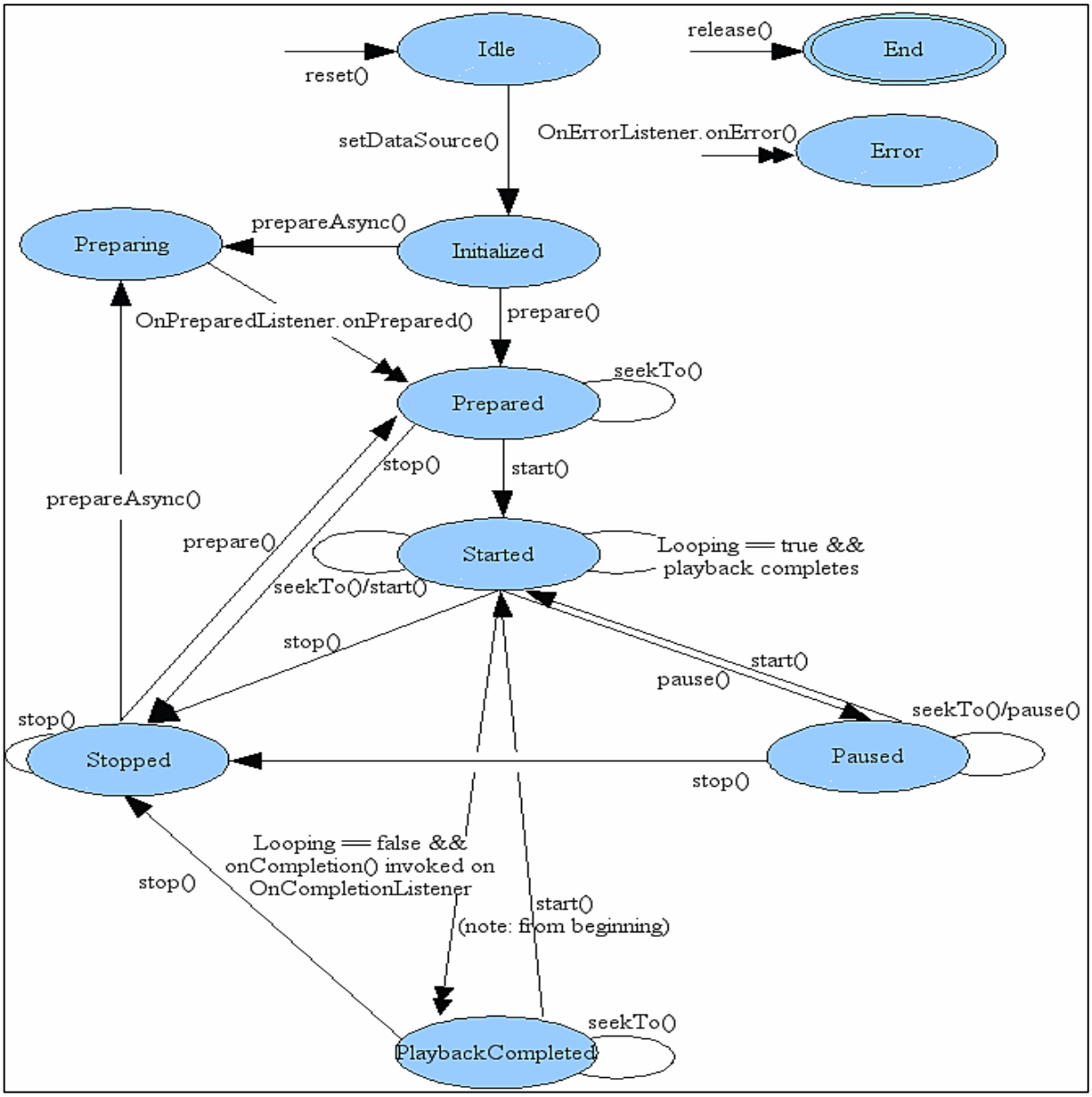
This state is triggered when we use setDataSource to set the information source that MediaPlayer has to use.

3. **Prepared State**:

In this state, the preparation work is completed. We can enter in this state calling prepare method or prepareAsync. In the first case after the method returns the state moves to Prepared. In the async way, we have to implement a listener to be notified when the system is ready and the state moves to Prepared. We have to keep in mind that when calling the prepare method, the entire app could hang before the method returns because the method can take a long time before it completes its work, especially when data is streamed from a remote server. We should avoid calling this method in the main thread because it might cause a ANR (Application Not Responding) problem. Once the MediaPlayer is in prepared state we can play our file, pause it or stop it.

4. **Completed state:**

To end of the stream is reached.



We can play a file in several ways:

// Raw audio file as resource
MediaPlayer mp = MediaPlayer.create(this, R.raw.audio_file);
// Local file
MediaPlayer mp1 = MediaPlayer.create(this, Uri.parse("file:///...."));
// Remote file
MediaPlayer mp2 = MediaPlayer.create(this, Uri.parse("http://website.com"));
or
we can use setDataSource in this way:
MediaPlayer mp3 = new MediaPlayer();
mp3.setDataSource("http://www.website.com");

Once we have created our MediaPlayer we can "prepare" it:

mp3.prepare();
and finally we can play it:
mp3.start();

Using Android Camera:

If we want to add to our apps the capability to take photos using the integrated smart phone camera, then the best way is to use an Intent. For example, let us suppose we want to start the camera as soon as we press a button and show the result in our app.

In the onCreate() method of our Activity, we have to setup a listener of the Button and when clicked to fire the intent:

```
Button b = (Button) findViewById(R.id.btn1);
b.setOnClickListener(new View.OnClickListener() {
@Override
public void onClick(View v) {
// Here we fire the intent to start the camera
Intent i = new Intent(MediaStore.ACTION_IMAGE_CAPTURE);
startActivityForResult(i, 100);
}
});
```

In the onActivityResult method, we retrieve the picture taken and show the result:

```
@Override
protected void onActivityResult(int requestCode, int resultCode, Intent data) {
// This is called when we finish taking the photo
Bitmap bmp = (Bitmap) data.getExtras().get("data");
iv.setImageBitmap(bmp);
}
```