Unit-3: Drawing and Working with Animation

3

Unit Structure

3.0 Learning Objectives

In this unit you will learn about:

- The drawing and animation features built into Android
- Working with Canvas and Paint to draw shapes and text.
- Animation and Types of animation

3.1. Introduction

With Android, we can display images such as PNG and JPG graphics, as well as text and primitive shapes to the screen. We can paint these items with various colors, styles, or gradients and modify them using standard image transforms. We can even animate objects to give the illusion of motion.

3.2 Canvas and Paint

The Canvas class holds the "draw" calls. To draw something, you need 4 basic components:

1. A Bitmap to hold the pixels,
2. A Canvas to host the draw calls (writing into the bitmap),
3. A drawing primitive (e.g. Rect, Path, text, Bitmap), and
4. A paint (to describe the colors and styles for the drawing).

The android.graphics framework divides drawing into two areas:

- What to draw, handled by Canvas
- How to draw, handled by Paint.

For instance, Canvas provides a method to draw a line, while Paint provides methods to define that line's color. Canvas has a method to draw a rectangle, while Paint defines whether to fill that rectangle with a color or leave it empty. Simply put, Canvas defines shapes that you can draw on the screen, while Paint defines the color, style, font, and so forth of each shape you draw.

So, before you draw anything, you need to create one or more Paint objects. The PieChart example does this in a method called init, which is called from the constructor from Java.

```
private void init() {
   textPaint = new Paint(Paint.ANTI_ALIAS_FLAG);
   textPaint.setColor(textColor);
   if (textHeight == 0) {
      textHeight = textPaint.getTextSize();
   } else {
      textPaint.setTextSize(textHeight);
   }

   piePaint = new Paint(Paint.ANTI_ALIAS_FLAG);
   piePaint.setStyle(Paint.Style.FILL);
   piePaint.setTextSize(textHeight);

   shadowPaint = new Paint(0);
   shadowPaint.setColor(0xff101010);
   shadowPaint.setMaskFilter(new BlurMaskFilter(8, BlurMaskFilter.Blur.NORMAL));

      . . .
```

Creating objects ahead of time is an important optimization. Views are redrawn very frequently, and many drawing objects require expensive initialization. Creating drawing objects within your onDraw() method significantly reduces performance and can make your UI appear sluggish.

Once you have your object creation and measuring code defined, you can implement onDraw(). Every view implements onDraw() differently, but there are some common operations that most views share:

- Draw text using drawText(). Specify the typeface by calling setTypeface(), and the text color by calling setColor().
- Draw primitive shapes using drawRect(), drawOval(), and drawArc(). Change whether the shapes are filled, outlined, or both by calling setStyle().
- Draw more complex shapes using the Path class. Define a shape by adding lines and curves to a Path object, then draw the shape using drawPath(). Just as with primitive shapes, paths can be outlined, filled, or both, depending on the setStyle().

- Define gradient fills by creating LinearGradient objects. Call setShader() to use your LinearGradient on filled shapes.
- Draw bitmaps using drawBitmap().

For example, here's the code that draws PieChart. It uses a mix of text, lines, and shapes.

```
protected void onDraw(Canvas canvas) {
  super.onDraw(canvas);

  // Draw the shadow
  canvas.drawOval(shadowBounds, shadowPaint);

  // Draw the label text
  canvas.drawText(data.get(currentItem).mLabel, textX, textY, textPaint);

  // Draw the pie slices
  for (int i = 0; i < data.size(); ++i) {
     Item it = data.get(i);
     piePaint.setShader(it.shader);
     canvas.drawArc(bounds,360 - it.endAngle, it.endAngle - it.startAngle,
           true, piePaint);
  }
  // Draw the pointer
  canvas.drawLine(textX, pointerY, pointerX, pointerY, textPaint);
  canvas.drawCircle(pointerX, pointerY, pointerSize, mTextPaint);
}
```

3.3 Bitmaps

You can find lots of goodies for working with graphics such as bitmaps in the android.graphics package. The core class for bitmaps is android.graphics.Bitmap.

Drawing Bitmap Graphics on a Canvas

You can draw bitmaps onto a valid Canvas, such as within the onDraw() method of a View, using one of the drawBitmap() methods. For example, the following code loads a Bitmap resource and draws it on a canvas:

```
import android.graphics.Bitmap;
import android.graphics.BitmapFactory;
...
Bitmap pic = BitmapFactory.decodeResource(getResources(),
R.drawable.bluejay);
canvas.drawBitmap(pic, 0, 0, null);
```

Scaling Bitmap Graphics

Perhaps you want to scale your graphic to a smaller size. In this case, you can use the createScaledBitmap() method, like this:

```
Bitmap sm = Bitmap.createScaledBitmap(pic, 50, 75, false);
```

You can preserve the aspect ratio of the Bitmap by checking the getWidth() and getHeight() methods and scaling appropriately.

3.4 Shapes

You can define and draw primitive shapes such as rectangles and ovals using the ShapeDrawable class in conjunction with a variety of specialized Shape classes. You can define Paintable drawables as XML resource files, but more often, especially with more complex shapes, this is done programmatically.

Defining Shape Drawables as XML Resources

In Unit-5, “Application Resources” of block-3, we show you how to define primitive shapes such as rectangles using specially formatted XML files within the /res/drawable/resource directory.

The following resource file called /res/drawable/green_rect.xml describes a simple, green rectangle shape drawable:

```
<?xml version=”1.0” encoding=”utf-8”?>
<shape xmlns:android=“http://schemas.android.com/apk/res/android”
        android:shape=”rectangle”>
        <solid android:color=”#0f0”/>
</shape>
```

You can then load the shape resource and set it as the Drawable as follows:

```
ImageView iView = (ImageView)findViewById(R.id.ImageView1);
iView.setImageResource(R.drawable.green_rect);
```

You should note that many Paint properties can be set via XML as part of the Shape definition. For example, the following Oval shape is defined with a linear gradient (red to white) and stroke style information:

```
<?xml version=”1.0” encoding=”utf-8”?>
<shape xmlns:android=”http://schemas.android.com/apk/res/android”
android:shape=”oval”>
<solid android:color=”#f00”/>
<gradient android:startColor=”#f00” android:endColor=”#fff” android:angle=”180”/>
<stroke android:width=”3dp”       android:color=”#00f”
      android:dashWidth=”5dp”   android:dashGap=”3dp”
/>
</shape>
```

Defining Shape Drawables Programmatically

You can also define this ShapeDrawable instances programmatically. The different shapes are available as classes within the android.graphics.drawable.shapes package. For example, you can programmatically define the aforementioned green rectangle as follows:

```
import android.graphics.drawable.ShapeDrawable;
import android.graphics.drawable.shapes.RectShape;
...

ShapeDrawable rect = new ShapeDrawable(new RectShape());
rect.getPaint().setColor(Color.GREEN);
```

You can then set the Drawable for the ImageView directly:

```
ImageView iView = (ImageView)findViewById(R.id.ImageView1);
iView.setImageDrawable(rect);
```

Drawing Different Shapes

Some of the different shapes available within the android.graphics.drawable.shapes package include

- Rectangles (and squares)
- Rectangles with rounded corners
- Ovals (and circles)
- Arcs and lines
- Other shapes defined as paths

You can create and use these shapes as Drawable resources directly within ImageView views, or you can find corresponding methods for creating these primitive shapes within a Canvas.

Drawing Rectangles and Squares

Drawing rectangles and squares (rectangles with equal height/width values) is simply a matter of creating a ShapeDrawable from a RectShape object. The RectShape object has no dimensions but is bound by the container object—in this case, the ShapeDrawable.

You can set some basic properties of the ShapeDrawable, such as the Paint color and the default size.

For example, here we create a magenta-colored rectangle that is 100-pixels long and 2-pixels wide, which looks like a straight, horizontal line. We then set the shape as the drawable for an ImageView so the shape can be displayed:

```
import android.graphics.drawable.ShapeDrawable;
import android.graphics.drawable.shapes.RectShape;
…

ShapeDrawable rect = new ShapeDrawable(new RectShape());
rect.setIntrinsicHeight(2);
rect.setIntrinsicWidth(100);
rect.getPaint().setColor(Color.MAGENTA);
ImageView iView = (ImageView)findViewById(R.id.ImageView1);
iView.setImageDrawable(rect);
```

Similarly we can draw other shapes.

3.5 Frame by Frame animation

You can think of frame-by-frame animation as a digital flipbook in which a series of similar images display on the screen in a sequence, each subtly different from the last. When you display these images quickly, they give the illusion of movement. This technique is called frame-by-frame animation and is often used on the Web in the form of animated GIF images.

Frame-by-frame animation is best used for complicated graphics transformations that are not easily implemented programmatically.

An object used to create frame-by-frame animations, defined by a series of Drawable objects, which can be used as a View object's background.

The simplest way to create a frame-by-frame animation is to define the animation in an XML file, placed in the res/drawable/ folder, and set it as the background to a View object. Then, call start() to run the animation.

An AnimationDrawable defined in XML consists of a single <animation-list> element and a series of nested <item> tags. Each item defines a frame of the animation. See the example below.

spin_animation.xml file in res/drawable/ folder:

```
<animation-list android:id="@+id/selected" android:oneshot="false">
    <item android:drawable="@drawable/wheel0" android:duration="50" />
    <item android:drawable="@drawable/wheel1" android:duration="50" />
    <item android:drawable="@drawable/wheel2" android:duration="50" />
    <item android:drawable="@drawable/wheel3" android:duration="50" />
    <item android:drawable="@drawable/wheel4" android:duration="50" />
    <item android:drawable="@drawable/wheel5" android:duration="50" />
 </animation-list>
```

Here is the code to load and play this animation.

```
// Load the ImageView that will host the animation and
// set its background to our AnimationDrawable XML resource.
   ImageView img = (ImageView)findViewById(R.id.spinning_wheel_image);
   img.setBackgroundResource(R.drawable.spin_animation);

 // Get the background, which has been compiled to an AnimationDrawable object.
   AnimationDrawable frameAnimation = (AnimationDrawable) img.getBackground();

 // Start the animation (looped playback by default).
    frameAnimation.start();
```

3.6 Tweened Animation

With tweened animation, you can provide a single Drawable resource - it is a Bitmap graphic, a ShapeDrawable, a TextView, or any other type of View object—and the intermediate frames of the animation are rendered by the system. Android provides tweening support for several common image transformations, including alpha, rotate, scale, and translate animations. You can apply tweened animation transformations to any View, whether it is an ImageView with a Bitmap or shape Drawable, or a layout such as a TableLayout.

Defining Tweening Transformations

You can define tweening transformations as XML resource files or programmatically. All tweened animations share some common properties, including when to start, how long to animate, and whether to return to the starting state upon completion.

Defining Tweened Animations as XML Resources

In Unit-5 of Block-3, we showed you how to store animation sequences as specially formatted XML files within the /res/anim/ resource directory. For example, the following resource file called /res/anim/spin.xml describes a simple five-second rotation:

```
<?xml version=”1.0” encoding=”utf-8” ?>
<set xmlns:android= “http://schemas.android.com/apk/res/android”
      android:shareInterpolator=”false”>
        <rotate android:fromDegrees=”0”
                  android:toDegrees=”360”
                  android:pivotX=”50%”
                  android:pivotY=”50%”
                  android:duration=”5000” />
</set>
```

Defining Tweened Animations Programmatically

You can programmatically define these animations.The different types of transformations are available as classes within the android.view.animation package. For example, you can define the aforementioned rotation animation as follows:

```
import android.view.animation.RotateAnimation;
...
RotateAnimation rotate = new RotateAnimation(0, 360,
                                RotateAnimation.RELATIVE_TO_SELF, 0.5f,
                                RotateAnimation.RELATIVE_TO_SELF, 0.5f);
rotate.setDuration(5000);
```

Defining Simultaneous and Sequential Tweened Animations

Animation transformations can happen simultaneously or sequentially when you set the startOffset and duration properties, which control when and for how long an animation takes to complete. You can combine animations into the <set> tag (programmatically, using AnimationSet) to share properties.

For example, the following animation resource file /res/anim/grow.xml includes a set of two scale animations: First, we take 2.5 seconds to double in size, and then at 2.5 seconds, we start a second animation to shrink back to our starting size:

```
<?xml version=”1.0” encoding=”utf-8” ?>
<set xmlns:android=http://schemas.android.com/apk/res/android
                android:shareInterpolator=”false”>
        <scale android:pivotX=”50%”
                 android:pivotY=”50%”
                android:fromXScale=”1.0”
                android:fromYScale=”1.0”
                android:toXScale=”2.0”
                android:toYScale=”2.0”
                android:duration=”2500” />
        <scale
```

```
            android:startOffset="2500"
            android:duration="2500"
            android:pivotX="50%"
            android:pivotY="50%"
            android:fromXScale="1.0"
            android:fromYScale="1.0"
            android:toXScale="0.5"
            android:toYScale="0.5" />
</set>
```

Loading Animations

Loading animations is made simple by using the AnimationUtils helper class.The following code loads an animation XML resource file called /res/anim/grow.xml and applies it to an ImageView whose source resource is a green rectangle shape drawable:

```
import android.view.animation.Animation;
import android.view.animation.AnimationUtils;
...
ImageView iView = (ImageView)findViewById(R.id.ImageView1);
iView.setImageResource(R.drawable.green_rect);
Animation an = AnimationUtils.loadAnimation(this, R.anim.grow);
iView.startAnimation(an);
```

We can listen for Animation events, including the animation start, end, and repeat events, by implementing an AnimationListener class, such as the MyListener class shown here:

```
class MyListener implements Animation.AnimationListener {
        public void onAnimationEnd(Animation animation) {
                // Do at end of animation
        }
        public void onAnimationRepeat(Animation animation) {
                // Do each time the animation loops
```

```
        }
        public void onAnimationStart(Animation animation) {
                // Do at start of animation
        }
}
```

You can then register your AnimationListener as follows:
an.setAnimationListener(new MyListener());

Now let's look at each of the four types of tweening transformations individually. These types are:

- Transparency changes (Alpha)
- Rotations (Rotate)
- Scaling (Scale)
- Movement (Translate)

Working with Alpha Transparency Transformations

Transparency is controlled using Alpha transformations. Alpha transformations can be used to fade objects in and out of view or to layer them on the screen.

Alpha values range from 0.0 (fully transparent or invisible) to 1.0 (fully opaque or visible). Alpha animations involve a starting transparency (fromAlpha) and an ending transparency (toAlpha).

The following XML resource file excerpt defines a transparency-change animation, taking five seconds to fade in from fully transparent to fully opaque:

```
<alpha android:fromAlpha="0.0"
        android:toAlpha="1.0"
        android:duration="5000">
</alpha>
```

Programmatically, you can create this same animation using the AlphaAnimation class within the android.view.animation package.

Working with Rotating Transformations

You can use rotation operations to spin objects clockwise or counterclockwise around a pivot point within the object's boundaries.

Rotations are defined in terms of degrees. For example, you might want an object to make one complete clockwise rotation. To do this, you set the fromDegrees property to 0 and the toDegrees property to 360. To rotate the object counterclockwise instead, you set the toDegrees property to -360.

By default, the object pivots around the (0,0) coordinate, or the top-left corner of the object. This is great for rotations such as those of a clock's hands, but much of the time, you want to pivot from the center of the object; you can do this easily by setting the pivot point, which can be a fixed coordinate or a percentage.

The following XML resource file excerpt defines a rotation animation, taking five seconds to make one full clockwise rotation, pivoting from the center of the object:

```
<rotate android:fromDegrees="0"
        android:toDegrees="360"
        android:pivotX="50%"
        android:pivotY="50%"
        android:duration="5000" />
```

Programmatically, you can create this same animation using the RotateAnimation class within the android.view.animation package.

Working with Scaling Transformations

You can use scaling operations to stretch objects vertically and horizontally. Scaling operations are defined as relative scales.Think of the scale value of 1.0 as 100 percent, or fullsize. To scale to half-size, or 50 percent, set the target scale value of 0.5. You can scale horizontally and vertically on different scales or on the same scale (to preserve aspect ratio).You need to set four values for proper scaling:

starting scale (fromXScale, fromYScale) and target scale (toXScale, toYScale). Again, you can use a pivot point to stretch your object from a specific (x,y) coordinate such as the center or another coordinate.

The following XML resource file excerpt defines a scaling animation, taking five seconds to double an object's size, pivoting from the center of the object:

```
<scale android:pivotX="50%"
        android:pivotY="50%"
        android:fromXScale="1.0"
        android:fromYScale="1.0"
        android:toXScale="2.0"
        android:toYScale="2.0"
        android:duration="5000" />
```

Programmatically, you can create this same animation using the ScaleAnimation class within the android.view.animation package.

Working with Moving Transformations

You can move objects around using translate operations.Translate operations move an object from one position on the (x,y) coordinate to another coordinate.

To perform a translate operation, you must specify the change, or delta, in the object's coordinates. You can set four values for translations: starting position (fromXDelta, fromYDelta) and relative target location (toXDelta, toYDelta).

The following XML resource file excerpt defines a translate animation, taking 5 seconds to move an object up (negative) by 100 on the y-axis.We also set the fillAfter property to be true, so the object doesn't "jump" back to its starting position when the animation finishes:

```
<translate    android:toYDelta="-100"
              android:fillAfter="true"
              android:duration="2500" />
```

Programmatically, you can create this same animation using the TranslateAnimation class within the android.view.animation package.

Check your progress-1

a) You can define and draw primitive shapes such as rectangles and ovals using the _____ class.

b) What to draw, handled by _____.

c) How to draw, handled by _____.

d) We can draw text on canvas using_____ method.

e) _____animation is best used for complicated graphics transformations that are not easily implemented programmatically

(A) Frame-by-Frame (B) Tweened (C) Either (A) or (B) (D) None of these

f) Which of the following is a tweening transformation?

(A) Rotate (B) Scale (C) Translate (C) All of these

3.7 Let's sum up

The Android SDK comes with the android.graphics package, which includes powerful classes for drawing graphics and text to the screen in a variety of different ways. Some features of the graphics library include Bitmap graphics utilities, Typeface and font style support, Paint colors and styles, different types of gradients, and a variety of primitive and not-so-primitive shapes that can be drawn to the screen and even animated using tweening and frame-by-frame animation mechanisms.

3.8. Check your Progress: Possible Answers

1-a) ShapeDrawable	1-b) Canvas	1-c) Paint
1-d) drawText()	1-e) (A) Frame-by-Frame	1-f) (D) All of these