



Code > Android SDK

Android SDK: Creating a Rotating Dialer

Ralf Wondratschek Last updated Dec 19, 2011

🕒 15 min | 💬 English

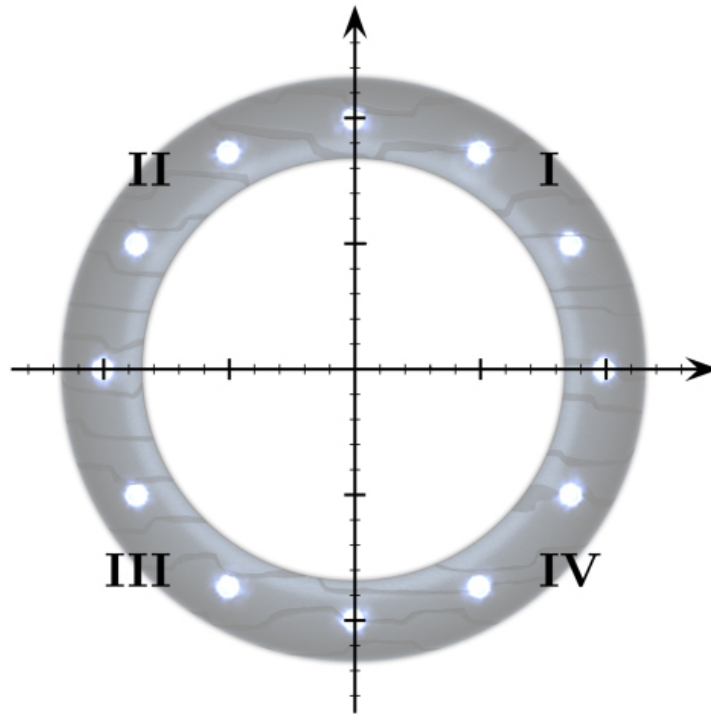


The Android SDK offers a wide range of interface components, including TextViews, Buttons, and EditText boxes. However, adding custom interface elements to your app is a great way to stand out in the App Market. This tutorial will dive into custom interface component creation by teaching you how to build a sleek rotary dialer.

This tutorial assumes some basic knowledge of Java and Android. Furthermore, I won't introduce a complete API. Instead, specific classes are going to be used and explained. I will also explain, step by step, why I have chosen a particular implementation for this interface component.

Step 1: The Concept

We have a circle and want to rotate it around its center. The simplest approach is to take a two dimensional Cartesian coordinate system as a template.

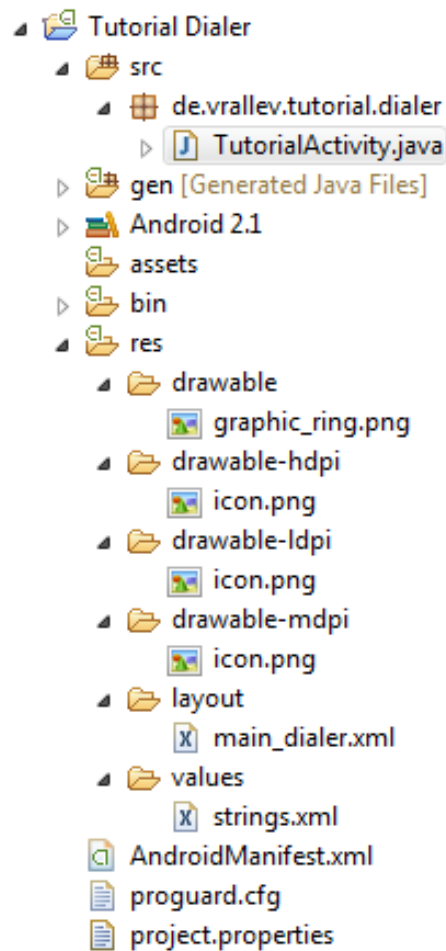


You touch the circle, rotate it, and then let it go. Moreover, the circle should register a "fling" when this occurs.

Android provides a very simple interface to transform images with the help of the `Bitmap` class and the `Matrix` class. The circle gets displayed in an `ImageView`. The mathematical basis of the unit circle can be implemented by using the `Math` class. Android also offers a good API to recognize touch events and gestures. As you can see, most of the work is already available to us with the SDK!

Step 2: Project Setup

Create a new Android project and add an Activity. Then add the dialer graphic to the "drawable" folder. We won't need different versions for different display densities, because later we will scale the image programmatically. One image with a high resolution, which covers all display sizes, should be enough and will save storage space on the user's phone.



Step 3: Layout

For this tutorial, we will keep the user interface very simple. We only add one `ImageView` to the layout. For the `ImageView`'s content, we select our dialer graphic. It is no problem to add more `Views` later as doing so doesn't influence our rotating dialer.

```
1 |  
2 | <?xml version="1.0" encoding="utf-8"?>  
3 | <LinearLayout  
4 |     xmlns:android="https://schemas.android.com/apk/res/android"  
5 |     android:orientation="vertical"  
6 |     android:layout_width="fill_parent"  
7 |     android:layout_height="fill_parent"  
8 |     android:background="#FFCCCC">  
9 |     <ImageView  
10 |         android:src="@drawable/graphic_ring"  
11 |         android:id="@+id/imageView_ring"  
12 |         android:layout_height="fill_parent"  
13 |         android:layout_width="fill_parent"></ImageView>  
14 | </LinearLayout>
```

Step 4: The Activity

As described in the steps above, we need to load the image as Bitmap while creating the activity. Furthermore, we need a matrix for the transformations, like the rotation. We do all of this in the onCreate method.

We also need a properly scaled copy of the image. Since we only know after measuring the layout how much space our ImageView fills, we add an OnGlobalLayoutListener. In the onGlobalLayout method, we can intercept the event that the layout has been drawn and query the size of our view.

```
1
2     private static Bitmap imageOriginal, imageScaled;
3     private static Matrix matrix;
4
5     private ImageView dialer;
6     private int dialerHeight, dialerWidth;
7
8     @Override
9     public void onCreate(Bundle savedInstanceState) {
10        super.onCreate(savedInstanceState);
11        setContentView(R.layout.main);
12
13        // load the image only once
14        if (imageOriginal == null) {
15            imageOriginal = BitmapFactory.decodeResource(getResources(), R.c
16        }
17
18        // initialize the matrix only once
19        if (matrix == null) {
20            matrix = new Matrix();
21        } else {
22            // not needed, you can also post the matrix immediately to rest
23            matrix.reset();
24        }
25
26
27        dialer = (ImageView) findViewById(R.id.imageView_ring);
28        dialer.setOnTouchListener(new MyOnTouchListener());
29        dialer.getViewTreeObserver().addOnGlobalLayoutListener(new OnGlobalLayo
30
31        @Override
32            public void onGlobalLayout() {
33                // method called more than once, but the values only nee
34                if (dialerHeight == 0 || dialerWidth == 0) {
35                    dialerHeight = dialer.getHeight();
36                    dialerWidth = dialer.getWidth();
37
38                    // resize
39                    Matrix resize = new Matrix();
40                    resize.postScale((float)Math.min(dialerV
41                    imageScaled = Bitmap.createBitmap(imageC
```

```

42 |         }
43 |     }
44 |     });
45 |
46 | }

```

The `OnTouchListener` is kept very simple. In the `ACTION_DOWN` event we initialize the angle (i.e. the unit circle). With each movement, the difference between the old and new angle gets additively incremented to the dialer.

We add the `OnTouchListener` as private inner class. Doing so will save us from the unnecessary passing of required parameters (e.g. the size of the view).

```

1 |
2 |     /**
3 |     * Simple implementation of an {@link OnTouchListener} for registering the dialer
4 |     */
5 |     private class MyOnTouchListener implements OnTouchListener {
6 |
7 |         private double startAngle;
8 |
9 |         @Override
10 |        public boolean onTouch(View v, MotionEvent event) {
11 |
12 |            switch (event.getAction()) {
13 |
14 |                case MotionEvent.ACTION_DOWN:
15 |                    startAngle = getAngle(event.getX(), event.getY());
16 |                    break;
17 |
18 |                case MotionEvent.ACTION_MOVE:
19 |                    double currentAngle = getAngle(event.getX(), event.getY());
20 |                    rotateDialer((float) (startAngle - currentAngle));
21 |                    startAngle = currentAngle;
22 |                    break;
23 |
24 |                case MotionEvent.ACTION_UP:
25 |
26 |                    break;
27 |            }
28 |
29 |            return true;
30 |        }
31 |    }
32 | }

```

The calculation of the angle can be quickly solved with the help of the unit circle. However, the touch event's coordinate needs to be converted to the coordinates of the two dimensional Cartesian coordinate system first. It would probably help you to set up these formulas by yourself once again to understand them completely.

```

1
2     /**
3  * @return The angle of the unit circle with the image view's center
4  */
5     private double getAngle(double xTouch, double yTouch) {
6         double x = xTouch - (dialerWidth / 2d);
7         double y = dialerHeight - yTouch - (dialerHeight / 2d);
8
9         switch (getQuadrant(x, y)) {
10            case 1:
11                return Math.asin(y / Math.hypot(x, y)) * 180 / Math.PI;
12            case 2:
13                return 180 - Math.asin(y / Math.hypot(x, y)) * 180 / Math.PI;
14            case 3:
15                return 180 + (-1 * Math.asin(y / Math.hypot(x, y)) * 180 / Math.PI);
16            case 4:
17                return 360 + Math.asin(y / Math.hypot(x, y)) * 180 / Math.PI;
18            default:
19                return 0;
20        }
21    }
22
23     /**
24  * @return The selected quadrant.
25  */
26     private static int getQuadrant(double x, double y) {
27         if (x >= 0) {
28             return y >= 0 ? 1 : 4;
29         } else {
30             return y >= 0 ? 2 : 3;
31         }
32     }

```

The method for rotating the dialer is very simple. We replace the old `ImageView`'s content every time with the new rotated `Bitmap`.

```

1
2     /**
3  * Rotate the dialer.
4  *
5  * @param degrees The degrees, the dialer should get rotated.
6  */
7     private void rotateDialer(float degrees) {
8         matrix.postRotate(degrees);
9         dialer.setImageBitmap(Bitmap.createBitmap(imageScaled, 0, 0, imageScaled.getWidth(), imageScaled.getHeight()));
10    }

```

Notice that, apart from some small issues, rotating the dialer already works. Also note that if the rotated `bitmap` does not fit into the view, the `bitmap` gets scaled down automatically (see the `scale` type of the `ImageView`). This causes the varying size.

Step 5: Algorithm Correction

A few bugs were introduced in the step above, for example the varying scale, but they can be easily fixed. Do you see where the problem is with the method implemented above?

If you take a deeper look into the implementation of the calculation of the rotated image, you will quickly discover that a new Bitmap instance is created each time. This wastes a lot of RAM and results in often running garbage collector in the background (you notice this issue in the output of LogCat). This causes the animation to look choppy.

The problem can only be avoided by changing the approach. Instead of rotating a bitmap and applying it to the ImageView, we should directly rotate the ImageView's content. For that purpose, we can change the scale type of the ImageView to "Matrix" and apply the matrix to the ImageView each time. This also adds another small change: now we need to add the scaled bitmap to the ImageView directly after the initialization.

```

1 |
2 | <?xml version="1.0" encoding="utf-8"?>
3 | <LinearLayout
4 |     xmlns:android="http://schemas.android.com/apk/res/android"
5 |     android:orientation="vertical"
6 |     android:layout_width="fill_parent"
7 |     android:layout_height="fill_parent"
8 |     android:background="#FFCCCCC">
9 |     <ImageView
10 |         android:src="@drawable/graphic_ring"
11 |         android:id="@+id/imageView_ring"
12 |         android:scaleType="matrix"
13 |         android:layout_height="fill_parent"
14 |         android:layout_width="fill_parent"></ImageView>
15 | </LinearLayout>

```



```

1 |
2 | ...
3 |
4 |         @Override
5 |         public void onGlobalLayout() {
6 |             // method called more than once, but the values only need
7 |             if (dialerHeight == 0 || dialerWidth == 0) {
8 |                 dialerHeight = dialer.getHeight();
9 |                 dialerWidth = dialer.getWidth();
10 |
11 |                 // resize
12 |                 Matrix resize = new Matrix();
13 |                 resize.postScale((float)Math.min(dialerV
14 |                 imageScaled = Bitmap.createBitmap(imageV
15 |
16 |                 dialer.setImageBitmap(imageScaled);

```

```

17         dialer.setImageMatrix(matrix);
18     }
19 }
20
21 ...
22
23     /**
24     * Rotate the dialer.
25     *
26     * @param degrees The degrees, the dialer should get rotated.
27     */
28     private void rotateDialer(float degrees) {
29         matrix.postRotate(degrees);
30
31         dialer.setImageMatrix(matrix);
32     }

```

Now another bug occurs. The calculation of the angle works properly with the `ImageView`'s center. However, the image rotates around the coordinate `[0, 0]`. Therefore, we need to move the image at initialization to the correct position. The same applies to the rotation. First the bitmap has to be shifted, so that the center is located in the origin of the `ImageView`, then the bitmap can be rotated and moved back again.

```

1
2     ...
3         @Override
4         public void onGlobalLayout() {
5             // method called more than once, but the values only need
6             if (dialerHeight == 0 || dialerWidth == 0) {
7                 dialerHeight = dialer.getHeight();
8                 dialerWidth = dialer.getWidth();
9
10                // resize
11                Matrix resize = new Matrix();
12                resize.postScale((float)Math.min(dialerWidth, dialerHeight),
13                imageScaled = Bitmap.createBitmap(image, 0, 0, imageWidth, imageHeight);
14
15                // translate to the image view's center
16                float translateX = dialerWidth / 2 - imageWidth / 2;
17                float translateY = dialerHeight / 2 - imageHeight / 2;
18                matrix.postTranslate(translateX, translateY);
19
20                dialer.setImageBitmap(imageScaled);
21                dialer.setImageMatrix(matrix);
22            }
23        }
24
25     ...
26
27     /**
28     * Rotate the dialer.
29     *
30     * @param degrees The degrees, the dialer should get rotated.
31     */
32     private void rotateDialer(float degrees) {

```



```
33     matrix.postRotate(degrees, dialerWidth / 2, dialerHeight / 2);
34
35     dialer.setImageMatrix(matrix);
36 }
```

Now our animation works much better and is very smooth.

Step 6: Fling

Android provides an easy-to-use API for recognizing gestures. For example a scroll, long press, and a fling can be detected. For our purposes, we extend the `SimpleOnGestureListener` class and override the method `onFling`. This class is recommended by the documentation.

Furthermore, we need to animate the rotation and thereby decrease its speed. Since these are just small calculations, it is not necessary to outsource the functionality in an extra thread. Instead, you can transfer the action to a `Handler` that can manipulate the GUI in the main thread. We can even avoid the handler instance by passing the action to the `ImageView` that uses its own `Handler` for it.

```
1
2   ...
3
4     private int dialerHeight, dialerWidth;
5
6     private GestureDetector detector;
7
8     @Override
9     public void onCreate(Bundle savedInstanceState) {
10
11     ...
12
13     detector = new GestureDetector(this, new MyGestureDetector());
14
15     dialer = (ImageView) findViewById(R.id.imageView_ring);
16     dialer.setOnTouchListener(new MyOnTouchListener());
17
18     ...
19
20     /**
21     * Simple implementation of an {@link OnTouchListener} for registering the dialer
22     */
23     private class MyOnTouchListener implements OnTouchListener {
24
25         private double startAngle;
26
```

```
27         @Override
28         public boolean onTouch(View v, MotionEvent event) {
29
30             switch (event.getAction()) {
31
32                 case MotionEvent.ACTION_DOWN:
33                     startAngle = getAngle(event.getX(), event.getY());
34                     break;
35
36                 case MotionEvent.ACTION_MOVE:
37                     double currentAngle = getAngle(event.getX(), event.getY());
38                     rotateDialer((float) (startAngle - currentAngle));
39                     startAngle = currentAngle;
40                     break;
41
42                 case MotionEvent.ACTION_UP:
43
44                     break;
45             }
46
47             detector.onTouchEvent(event);
48
49             return true;
50         }
51     }
52
53     /**
54     * Simple implementation of a {@link SimpleOnGestureListener} for detecting a fling.
55     */
56     private class MyGestureDetector extends SimpleOnGestureListener {
57         @Override
58         public boolean onFling(MotionEvent e1, MotionEvent e2, float velocityX, float velocityY) {
59             dialer.post(new FlingRunnable(velocityX + velocityY));
60             return true;
61         }
62     }
63
64     /**
65     * A {@link Runnable} for animating the the dialer's fling.
66     */
67     private class FlingRunnable implements Runnable {
68
69         private float velocity;
70
71         public FlingRunnable(float velocity) {
72             this.velocity = velocity;
73         }
74
75         @Override
76         public void run() {
77             if (Math.abs(velocity) > 5) {
78                 rotateDialer(velocity / 75);
79                 velocity /= 1.06666F;
80
81                 // post this instance again
82                 dialer.post(this);
83             }
84         }
85     }
```

Step 7: Fixing The Inversed Rotations

If you extensively test the fling, it quickly becomes clear that at various positions the dialer rotates in the wrong direction. For example, this is the case when the start and end points are only in the third quadrant. However there are also more difficult error cases (e. g. when wiping from quadrant two to four over quadrant three).

The simplest solution is to remember the touched quadrants. The error cases are caught in the onFling method and the velocity gets inverted accordingly.

```

1
2   ...
3
4   private GestureDetector detector;
5
6   // needed for detecting the inversed rotations
7   private boolean[] quadrantTouched;
8
9   ...
10
11  detector = new GestureDetector(this, new MyGestureDetector());
12
13      // there is no 0th quadrant, to keep it simple the first value of
14  quadrantTouched = new boolean[] { false, false, false, false, false };
15
16  dialer = (ImageView) findViewById(R.id.imageView_ring);
17
18  ...
19
20  /**
21  * Simple implementation of an {@link OnTouchListener} for registering the dialer
22  */
23  private class MyOnTouchListener implements OnTouchListener {
24
25      private double startAngle;
26
27      @Override
28      public boolean onTouch(View v, MotionEvent event) {
29
30          switch (event.getAction()) {
31
32              case MotionEvent.ACTION_DOWN:
33
34                  // reset the touched quadrants
35                  for (int i = 0; i < quadrantTouched.length; i++)
36                      quadrantTouched[i] = false;
37              }
38
39              startAngle = getAngle(event.getX(), event.getY());
40              break;
41
42              case MotionEvent.ACTION_MOVE:
43                  double currentAngle = getAngle(event.getX(), event.getY());

```

```

44         rotateDialer((float) (startAngle - currentAngle));
45         startAngle = currentAngle;
46         break;
47
48     case MotionEvent.ACTION_UP:
49
50         break;
51     }
52
53     // set the touched quadrant to true
54     quadrantTouched[getQuadrant(event.getX() - (dialerWidth / 2), dialerHeight / 2)] = true;
55
56     detector.onTouchEvent(event);
57
58     return true;
59     }
60 }
61
62 /**
63  * Simple implementation of a {@link SimpleOnGestureListener} for detecting a fling.
64  */
65 private class MyGestureDetector extends SimpleOnGestureListener {
66     @Override
67     public boolean onFling(MotionEvent e1, MotionEvent e2, float velocityX, float velocityY) {
68
69         // get the quadrant of the start and the end of the fling
70         int q1 = getQuadrant(e1.getX() - (dialerWidth / 2), dialerHeight / 2);
71         int q2 = getQuadrant(e2.getX() - (dialerWidth / 2), dialerHeight / 2);
72
73         // the inversed rotations
74         if ((q1 == 2 && q2 == 2 && Math.abs(velocityX) < Math.abs(velocityY)) ||
75             || (q1 == 3 && q2 == 3)
76             || (q1 == 1 && q2 == 3)
77             || (q1 == 4 && q2 == 4 && Math.abs(velocityX) < Math.abs(velocityY)) ||
78             || ((q1 == 2 && q2 == 3) || (q1 == 3 && q2 == 2)) ||
79             || ((q1 == 3 && q2 == 4) || (q1 == 4 && q2 == 3)) ||
80             || (q1 == 2 && q2 == 4 && quadrantTouched[q2]) ||
81             || (q1 == 4 && q2 == 2 && quadrantTouched[q2])
82         ) {
83             dialer.post(new FlingRunnable(-1 * (velocityX + velocityY)));
84         } else {
85             // the normal rotation
86             dialer.post(new FlingRunnable(velocityX + velocityY));
87         }
88
89         return true;
90     }
91 }

```

Step 8: Stop Fling

Of course the fling animation has to stop if you touch the dialer while the animation is running. For this it is only necessary to add a boolean value for checking whether the

animation is allowed to be played or not.

In the ACTION_DOWN event you want to stop it. Once you let the dialer go (ACTION_UP), the animation should be played.

```

1
2   ...
3
4       // needed for detecting the inversed rotations
5       private boolean[] quadrantTouched;
6
7       private boolean allowRotating;
8
9   ...
10
11           allowRotating = true;
12
13       dialer = (ImageView) findViewById(R.id.imageView_ring);
14
15   ...
16
17           switch (event.getAction()) {
18
19               case MotionEvent.ACTION_DOWN:
20
21                   // reset the touched quadrants
22                   for (int i = 0; i < quadrantTouched.length; i++)
23                       quadrantTouched[i] = false;
24                   }
25
26                   allowRotating = false;
27
28                   startAngle = getAngle(event.getX(), event.getY());
29                   break;
30
31               case MotionEvent.ACTION_MOVE:
32                   double currentAngle = getAngle(event.getX(), event.getY());
33                   rotateDialer((float) (startAngle - currentAngle));
34                   startAngle = currentAngle;
35                   break;
36
37               case MotionEvent.ACTION_UP:
38                   allowRotating = true;
39                   break;
40           }
41
42   ...
43
44       @Override
45       public void run() {
46           if (Math.abs(velocity) > 5 && allowRotating) {
47               rotateDialer(velocity / 75);
48               velocity /= 1.0666F;
49
50               // post this instance again
51               dialer.post(this);
52           }

```

Step 9: What's Next

All the desired features are included in the dialer. However, this is only an example and can be extended indefinitely.

A few additional enhancements that come to mind include: you could track the angle and invert or decrease certain values (see my app Shutdown Remote for an example of this). Also, after releasing the dialer you could animate it back to the original position to replicate the mechanical function of real, antique rotary dials. Further, the dialer could be extracted into a custom View class, which would increase the reusability of the component. The possibilities for enhancement abound!

Conclusion

In this tutorial, I have shown you how to implement a simple rotary dialer. Of course, the functionality can be extended and improved further as discussed above. The Android SDK offers a great and very useful API with many classes. There is no need to reinvent a lot, you normally just need to implement what is already there. Do not hesitate to give comments or leave suggestions!

Sources

http://en.wikipedia.org/wiki/File:Cartesian_coordinates_2D.svg

Android SDK

Mobile Development

Did you find this post useful?



Want a weekly email summary?

Subscribe below and we'll send you a weekly email summary of all new Code tutorials. Never miss out on learning about the next big thing.

Sign up



Ralf Wondratschek

Ralf Wondratschek is a computer science student from Germany. Next to his study, Ralf works for a German company as a software developer. In the last few years, he got in touch with Java, XML, HTML, JSP, JSF, Eclipse, Google App Engine, and the Android SDK. In his free time, he has [written and published two Android apps](#). You can find out more about the author's work on [his homepage](#).

Download Attachment



QUICK LINKS - Explore popular categories

ENVATO TUTORIALS+



HELP



 **tuts+** 
31,147 Tutorials 1,281 Courses 47,342 Translations

[Envato](#) [Envato Elements](#) [Envato Market](#) [Placeit by Envato](#) [All products](#) [Careers](#) [Sitemap](#)

© 2023 Envato Pty Ltd. Trademarks and brands are the property of their respective owners.

