CHAPTER 3

이미지 에디팅과 이미지 프로세싱

편하게 들고 다닐 수 있는 각종 기기들이 점점 더 강력한 성능으로 무장하면서 데스크톱 컴퓨터에만 존재했던 기능들이 모바일 기기로 옮겨가고 있다. 포토샵처럼 데스크톱 애플리케이션의 범주에 속했던 이미지 에디팅 및 프로세싱 도구가 이제는 휴대전화에서도 빛을 발한다.

이 장에서는 캡처한 이미지로 무엇을 할 수 있는지 파헤친다. 이미지의 회전이나 크기 변경을 어떻게 구현하는지 살펴보고, 밝기와 대비(콘트라스트)를 조정하는 방법이나 여러 이미지를 하나로 구성하는 방법 등에 관해서도 알아본다.

내장 갤러리 애플리케이션으로 이미지 선택하기

앞에서도 언급했지만 인텐트를 사용하는 것이 안드로이드의 내장 애플리케이션에 존재하는 기능을 이렇게 저렇게 써먹을 수 있는 제일 빠른 방법이다. 이 장에서 소개하는 예제들에는 작업할 이미지를 선택하려고 내장 Gallery 애플리케이션을 써먹는 과정이 펼쳐져 있다.

이제부터 사용할 제네릭 Intent.ACTION_PICK 인텐트는 어떤 데이터를 선택하겠다고 안드로이드한테 신호를 보내게 된다. 또한 해당 데이터의 URI도 필요한데, 여기서는 android.provier.MediaStore.Images.Media.EXTERNAL_CONTENT_URI를 사용한다. 이 URI는 SD 카드에 MediaStore를 사용하여 저장한 이미지를 선택하겠다는 의미다.

```
Intent choosePictureIntent = new Intent(Intent.ACTION_PICK, ↵
 android.provider.MediaStore.Images.Media.EXTERNAL_CONTENT_URI);
```

이 인텐트가 트리거되면 Gallery 애플리케이션이 사용자가 이미지를 선택할 수 있는 모드에서 시작한다.

사용자가 이미지를 선택하면 인텐트의 리턴 결과가 다 그렇듯 우리의 onActivityResult 메소드가 트리거된다. 리턴된 인텐트의 데이터 형태로 선택된 이미지의 URI가 리턴된다.

```
onActivityResult(int requestCode, int resultCode, Intent intent) {
    super.onActivityResult(requestCode, resultCode, intent);

    if (resultCode == RESULT_OK) {
        Uri imageFileUri = intent.getData();
    }
}
```

다음은 전체 예제다.

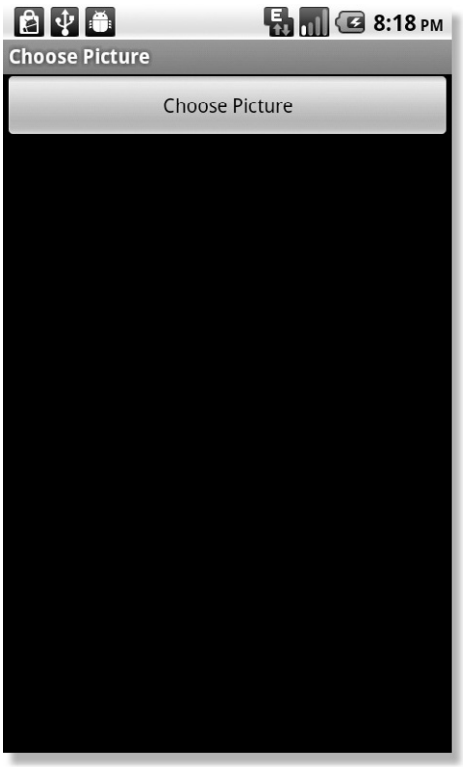
```
package com.apress.proandroidmedia.ch3.choosepicture;

import java.io.FileNotFoundException;
import android.app.Activity;
import android.content.Intent;
import android.graphics.Bitmap;
import android.graphics.BitmapFactory;
import android.net.Uri;
import android.os.Bundle;
import android.util.Log;
import android.view.Display;
```

```
import android.view.View;
import android.view.View.OnClickListener;
import android.widget.Button;
import android.widget.ImageView;
```

우리의 액티비티는 버튼이 만들어낸 클릭 이벤트에 응답하게 된다. 따라서 지금 OnClickListener를 구현한다. onCreate 메소드에서는 필요한 UI 요소에 접근하기 위해 findViewById 메소드를 사용한다. UI 요소들은 레이아웃 XML에서 정의된다.

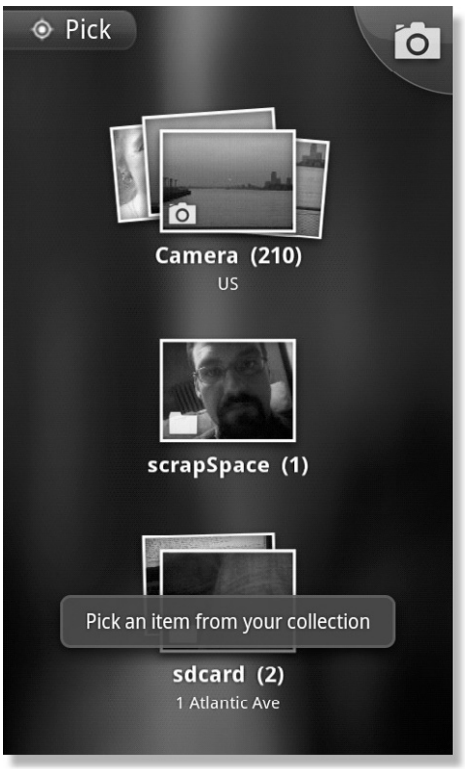
```
public class ChoosePicture extends Activity implements OnClickListener {

    ImageView chosenImageView;
    Button choosePicture;

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

        chosenImageView = (ImageView) this.findViewById(R.id.ChosenImageView);
        choosePicture = (Button) this.findViewById(R.id.ChoosePictureButton);

        choosePicture.setOnClickListener(this);
    }
```

뒤에 이어지는 onClick 메소드는 choosePicture 버튼을 누르면 응답한다. 이 버튼은 앱이 처음 시작하면 그림 3-1과 같이 나타난다. 이 메소드로 Gallery 애플리케이션을 트리거할 인텐트를 생성한다. Gallery 애플리케이션은 그림 3-2와 같이 사용자가 그림을 선택할 수 있는 모드에서 시작한다.



:: **그림 3-1** 애플리케이션이 처음 시작하면 나타나는 choosePicture 버튼

```
    public void onClick(View v) {
        Intent choosePictureIntent = new Intent(Intent.ACTION_PICK, ↵
android.provider.MediaStore.Images.Media.EXTERNAL_CONTENT_URI);
        startActivityForResult(choosePictureIntent, 0);
    }
```



:: **그림 3-2** Gallery 애플리케이션의 모습. 이미지를 선택하라고 사용자에게 알리는 ACTION_PICK 인텐트로 트리거된다. Gallery 애플리케이션의 UI는 기기마다 다를 수 있다

사용자가 이미지를 선택하고 나서 Gallery 애플리케이션으로 돌아오면 onActivityResult 메소드가 호출된다. 선택된 이미지의 URI는 넘겨받은 인텐트의 데이터로 가져온다.

```
protected void onActivityResult(int requestCode, int resultCode, Intent intent) {
super.onActivityResult(requestCode, resultCode, intent);

if (resultCode == RESULT_OK) {
    Uri imageFileUri = intent.getData();
```

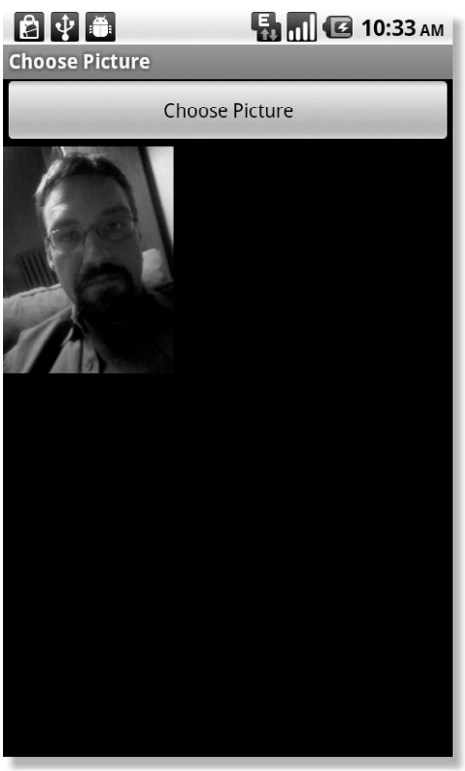
리턴받은 이미지는 메모리에 로드되기에 너무 클 가능성이 높다. 따라서 1장에서 다뤘던 기술인 로드할 때 크기를 다시 조절하는 기능이 필요하다. dw int와 dh int가 각각 최

대 너비와 최대 높이를 나타낸다. 최대 높이는 화면 높이의 반보다 작게 되므로 두 이미지를 세로로 정렬하여 표시하기로 한다.

```
            Display currentDisplay = getWindowManager().getDefaultDisplay();
            int dw = currentDisplay.getWidth();
            int dh = currentDisplay.getHeight() / 2 - 100;

            try {
                // 이미지 자체가 아니라 이미지의 치수를 로드한다.
                BitmapFactory.Options bmpFactoryOptions = new BitmapFactory.Options();
                bmpFactoryOptions.inJustDecodeBounds = true;
                Bitmap bmp = BitmapFactory.decodeStream(getContentResolver().
openInputStream(imageFileUri), null, bmpFactoryOptions);

                int heightRatio = (int) Math.ceil(bmpFactoryOptions.outHeight/(float) dh);
                int widthRatio = (int) Math.ceil(bmpFactoryOptions.outWidth/(float) dw);

                if (heightRatio > 1 && widthRatio > 1) {
                    if (heightRatio > widthRatio) {
                        bmpFactoryOptions.inSampleSize = heightRatio;
                    } else {
                        bmpFactoryOptions.inSampleSize = widthRatio;
                    }
                }

                bmpFactoryOptions.inJustDecodeBounds = false;
                bmp = BitmapFactory.decodeStream(getContentResolver().
openInputStream(imageFileUri), null, bmpFactoryOptions);

                chosenImageView.setImageBitmap(bmp);

            } catch (FileNotFoundException e) {
                Log.v("ERROR", e.toString());
            }
        }
    }
}
```

이제 레이아웃 XML 파일이 필요할 때다. 다음은 layout/main.xml 파일의 내용이다.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    >
    <Button
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="Choose Picture" android:id="@+id/ChoosePictureButton"/>
    <ImageView android:layout_width="wrap_content" android:layout_height=
    "wrap_content" android:id="@+id/ChosenImageView"></ImageView>
</LinearLayout>
```



:: **그림 3-3** 사용자가 그림을 선택하고 난 애플리케이션의 모습

여기까지다. 이제 사용자가 선택한 이미지를 Bitmap 객체로 손에 넣었다. 이 Bitmap 객체는 사용자에게 그림 3–3에서처럼 표시된다. 지금부터 Bitmap을 출발점으로 하여 다른 작업들로 나아가기로 한다.

비트맵 위에 비트맵 그리기

이미지를 이렇게 저렇게 조작하는 데 사용하는 구체적인 메커니즘을 들여다보기에 앞서, 우선 아무 것도 없는 새로운 Bitmap 객체를 만들어 기존에 있던 Bitmap을 그 위에 그리는 과정부터 살펴본다. 우리의 이미지들을 조작한 버전이 만들어지는 데 사용할 과정이 바로 이것이다.

앞의 연습을 통해 우리는 사용자가 선택한 이미지로 인스턴스화한 Bitmap 객체를 손에 넣었다. Bitmap 객체는 1장에서 다뤘던 대로 BitmapFactory의 decodeStream 메소드를 호출하여 인스턴스화했다.

```
Bitmap bmp = BitmapFactory.decodeStream(getContentResolver().➥
openInputStream(imageFileUri), null, bmpFactoryOptions);
```

이 Bitmap을 이미지 에디팅 작업에 소스로 사용하기 위해서는 화면에 여러 효과를 적용하며 이 Bitmap을 그려야 한다. 또한 결과 이미지를 저장하는 데 사용할 객체에 그릴 수 있다면 참 좋을 것이다. 이 Bitmap과 같은 크기의 빈 Bitmap 객체를 만들고, 편집되는 Bitmap의 목적지로 이 빈 객체를 사용하면 된다.

```
Bitmap alteredBitmap = Bitmap.createBitmap(bmp.getWidth(),➥
bmp.getHeight(), bmp.getConfig());
```

alteredBitmap 객체는 소스 Bitmap인 bmp와 동일한 너비, 높이, 색 수(색 농도, color depth)로 생성된다. 너비, 높이, Bitmap.Config 객체를 가지는, Bitmap 클래스의 createBitmap 메소드를 파라미터로 사용했기 때문에 변경 가능한(mutable) 객체인 Bitmap을 리턴받게 된다. 여기서 객체가 변경 가능하다(mutable)는 말은 이 Bitmap이

표현하는 픽셀 값들을 바꿀 수 있다는 뜻이다. 만일 변경이 불가능한(immutable) Bitmap을 얻으면 그 위에 그릴 수 없게 된다. 이런 메소드 호출은 변경 가능한 Bitmap 객체를 인스턴스화할 수 있는 유일한 방법이라고도 할 수 있다.

그 다음으로는 Canvas 객체가 필요하다. 안드로이드에서 Canvas는 예상하는 대로 그릴 도화지에 해당한다. Canvas는 Bitmap 객체를 그 컨스트럭터에 넘겨주면 생성된다. 그러면 그리는 데 사용할 수 있다.

```
Canvas canvas = new Canvas(alteredBitmap);
```

마지막으로 Paint 객체가 필요하다. 실제로 그리는 작업을 하면 Paint 객체가 그 일을 담당한다. 구체적으로 설명하면, 색이나 콘트라스트 같은 것을 바꿀 수 있도록 해준다는 뜻이다. 하지만 여기서는 이렇게만 설명하고 넘어가기로 한다. 당분간 디폴트 Paint 객체를 그대로 사용한다.

```
Paint paint = new Paint();
```

이제 소스 Bitmap을 빈 변경 가능한(mutable) Bitmap 객체로 그리는 데 필요한 구성요소는 다 갖췄다. 다음은 앞에서 설명한 부분들을 하나로 모은 것이다.

```
Bitmap bmp = BitmapFactory.decodeStream(getContentResolver().
openInputStream(imageFileUri), null, bmpFactoryOptions);
Bitmap alteredBitmap = Bitmap.createBitmap(bmp.getWidth(), bmp.getHeight(),
bmp.getConfig());
Canvas canvas = new Canvas(alteredBitmap);
Paint paint = new Paint();

canvas.drawBitmap(bmp, 0, 0, paint);

ImageView alteredImageView = (ImageView) this.findViewById(R.id.AlteredImageView);
alteredImageView.setImageBitmap(alteredBitmap);
```

Canvas 객체의 drawBitmap 메소드는 Paint 객체와 함께 소스 Bitmap 그리고 x, y 오프셋을 받는다. 이렇게 하면 alteredBitmap 객체가 원래 비트맵과 완전히 동일한 정보를 담을 수 있다.

지금까지 코드 전부를 Choose Picture 예제에 집어넣을 수 있다. onActivityResult 메소드 끝부분에, 즉 bmp = BitmapFactory.decodeStream 행에 넣으면 된다. 이 행을 반복하지 않도록 주의한다. 바로 앞에서 소개한 코드에도 같은 행이 있으니 말이다. 그리고 적절한 import 구문도 빠뜨리지 않도록 한다.

이제 alteredBitmap 객체를 표시한다. 그러기 위해서 표준 ImageView를 사용하고 alteredBitmap과 함께 setImageBitmap을 호출한다. 이때 레이아웃 XML에 선언된 id AlteredImageView가 ImageView에 있다고 가정된다.

다음은 업데이트된 레이아웃 XML로서, 그림 3–4에 나타낸 대로 alteredBitmap에 쓸 새로운 ImageView뿐만 아니라 원래 ImageView도 들었다.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    >
    <Button
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="Choose Picture" android:id="@+id/ChoosePictureButton"/>

    <ImageView android:layout_width="wrap_content" android:layout_height="wrap_content" ↵
     android:id="@+id/ChosenImageView"></ImageView>
    <ImageView android:layout_width="wrap_content" android:layout_height="wrap_content" ↵
     android:id="@+id/AlteredImageView"></ImageView>
</LinearLayout>
```

:: **그림 3-4** 사용자가 선택한 이미지 및 동일한 두 번째 이미지가 함께 표시된 모습

기본적인 이미지 확대/축소와 회전

이미지 확대/축소나 회전 등과 같은 공간 변형을 어떻게 구현하는지를 알아보는 것으로 이미지 에디팅/프로세싱을 시작할까 한다.

매트릭스로 들어가다

안드로이드 API에는 Matrix 클래스가 제공된다. 이 클래스는 이미 있던 Bitmap 위에 그리거나 다른 Bitmap에서 새로운 Bitmap을 생성할 때 사용한다. 그리고 공간 변형

(spatial transformation)을 이미지에 적용할 때도 사용한다. 공간 변형이라고 하면 대개 이미지의 회전, 자르기, 확대/축소 등 이미지의 좌표 공간을 바꾸는 행위를 말한다.

Matrix 클래스에서는 숫자 9개로 이뤄진 배열로 변형을 나타낸다. 대개 어떤 변형을 나타내는 수학 공식으로 각 숫자를 생성하는데, 가령 회전 공식에는 매트릭스의 숫자를 생성하기 위해 사인 함수와 코사인 함수가 들어간다.

물론 공식으로가 아니라 직접 Matrix의 숫자들을 입력할 수도 있다. Matrix의 동작 방식을 이해하기 위해서 직접 숫자를 넣어 계산하기로 한다.

Matrix의 각 숫자들은 이미지의 각 점이 차지하는 좌표(x, y, z)를 나타낸다.

예를 들어 다음은 Matrix의 숫자 9개다.

```
1 0 0
0 1 0
0 0 1
```

첫 번째 행인 (1, 0, 0)은 x = 1x + 0y + 0z라는 공식에 따라 변형될 x 좌표를 나타낸다. 물론 매트릭스 안에서 지정되는 값의 위치에 따라 변형 결과도 달라진다. 첫 번째 행은 항상 x 좌표에 영향을 끼치지만, 그 동작은 x, y, z 좌표로 진행된다.

두 번째 행인 (0, 1, 0)은 y 좌표가 y = 0x + 1y +0z로 결정된다는 의미고, 세 번째 행인 (0, 0, 1)은 z 좌표가 z = 0x +0y + 1z로 결정된다는 의미다.

다시 말해 이 Matrix는 어떤 변형을 진행하지는 않는다. 모든 것이 원래 소스 이미지와 동일하게 생성된다.

지금 이 내용을 코드로 구현하려면 Matrix 객체를 생성한 다음, 일일이 값들을 입력한다. 입력은 setValues 메소드로 한다.

이 Matrix 객체를 사용하여 캔버스 위에 비트맵을 그린다.

```
Matrix matrix = new Matrix();
matrix.setValues(new float[] {
    1, 0, 0,
    0, 1, 0,
    0, 0, 1
});
```

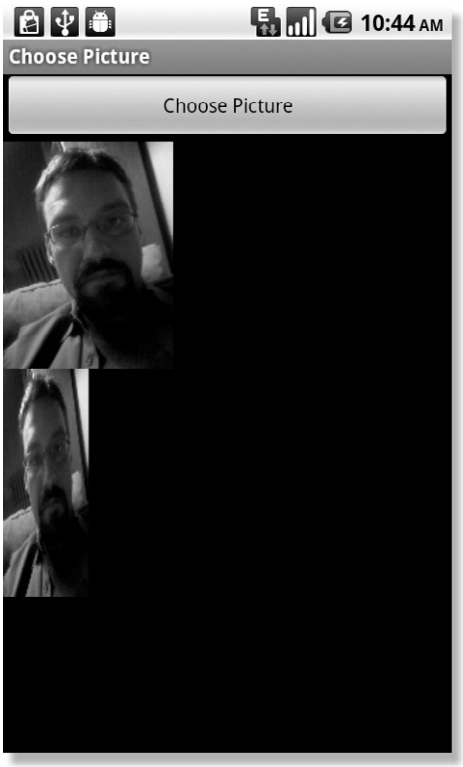
이렇게 하면 앞선 에제에서 사용했던 drawBitmap 메소드를 대신한다.

```
canvas.drawBitmap(bmp, matrix, paint);
```

이 Matrix로 이미지를 어떤 식으로든 변경하기 위해서는 처음 있던 값들을 다른 값들로 교체하면 된다.

첫 번째 행의 1을 .5로 변경하면, 그림 3–5에 나타난 것과 같이 x축으로 눌려 50% 줄어든 이미지가 생성된다. 첫 번째 숫자는 소스 이미지의 x축에서 동작하므로 결과 이미지의 x축에 효과가 적용된다.

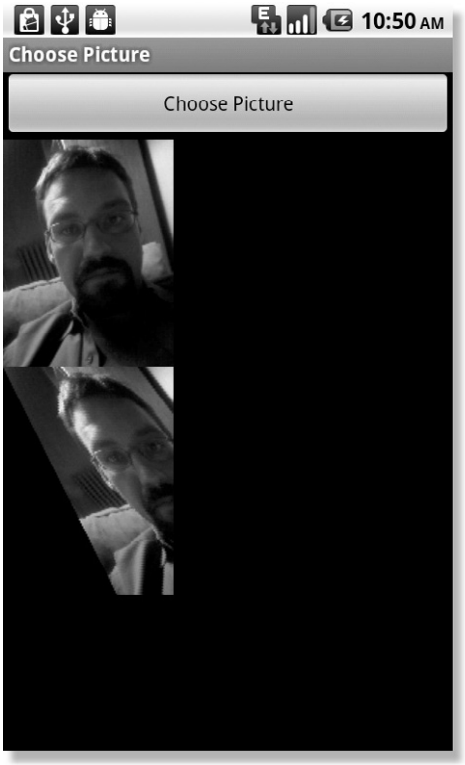
```
.5 0 0
0 1 0
0 0 1

Matrix matrix = new Matrix();
matrix.setValues(new float[] {
    .5f, 0, 0,
    0, 1, 0,
    0, 0, 1
});
canvas.drawBitmap(bmp, matrix, paint);
```



:: **그림 3-5** 매트릭스를 변형하여 생성된 두 번째 이미지. x축으로 50% 축소됨

x축이 y축에도 영향을 받게 하려면 두 번째 숫자를 변경한다.

```
Matrix matrix = new Matrix();
matrix.setValues(new float[] {
    1, .5f, 0,
    0, 1, 0,
    0, 0, 1
});
canvas.drawBitmap(bmp, matrix, paint);
```



:: **그림 3-6** 매트릭스를 변형하여 생성된 두 번째 이미지. 왼쪽으로 기울어짐

그림 3-6에서는 두 번째 이미지가 기울어져 표시됐다. 첫 번째 행, 다시 말해 각 픽셀의 x축에서 동작하는 첫 번째 행이 각 픽셀의 y축 값에 영향을 받았기 때문이다. y값이 증가하면서, 즉 이미지가 아래로 이동하면서 x값이 증가했으므로 결과 이미지는 기울어졌다. 음수 값을 사용하면 반대 방향으로 기울어진다. 그리고 그림을 잘 들여다보면 그림의 일부가 잘려나갔음을 알 수 있다. 좌표가 변했기 때문이다.

결과 비트맵의 크기를 늘려야 그림 3-7처럼 나타난다.

```
alteredBitmap = Bitmap.createBitmap(bmp.getWidth()*2,bmp.getHeight(),bmp.getConfig());
```

:: **그림 3-7** 매트릭스를 변형하여 생성된 두 번째 이미지. 너비가 넓어져 이미지가 잘리지 않음

이미 눈치 챘겠지만 이와 같은 Matrix 변형은 강력한 힘을 발휘한다. 물론 일일이 손으로 작업한다는 것은 고된 일이다. 아쉽게도 Matrix로 할 수 있는 여러 작업에 필요한 공식들은 이 책의 범위를 넘어선다. 하지만 온라인에는 괜찮은 리소스가 널렸다. 조금만 관심을 기울인다면 유용하게 써먹을 수 있을 것이다. 한 곳을 추천하자면 위키피디어 변형 매트릭스(Wikipedia Transformation Matrix)인데, http://en.wikipedia.org/wiki/Transformation_matrix에서 찾아볼 수 있다.

매트릭스 메소드들

지금부터 살펴볼 내용은 Matrix의 다른 메소드들인데, 고등학교 때 배운 수학을 굳이 다시 꺼내보지 않아도 이미지 작업에 도움이 될 만한 내용들이다.

직접 Matrix에 숫자를 입력하지 않고 사용하려는 변형에 필요한 메소드들을 호출할 것이다.

앞으로 소개하는 코드 조각들은 "비트맵 위에 비트맵 그리기" 예제의 canvas. drawBitmap 행에 대신 들어간다.

회전

내장돼 제공되는 메소드에 setRotation이 있다. setRotation 메소드는 회전 각도를 나타내는 float 타입 실수를 받는다. 양수는 시계 방향으로 회전하는 것을 의미하고, 음수는 그 반대인데, 디폴트 포인트는 (0,0)이다. 그림 3-8에 나타낸 것처럼 이미지의 상단 왼쪽 끝점을 의미한다.

```
Matrix matrix = new Matrix();
matrix.setRotate(15);
canvas.drawBitmap(bmp, matrix, paint);
```

:: **그림 3-8** 디폴트 포인트인 (0,0)을 기준으로 회전한다

setRotation 메소드는 회전 각도와 회전 기준점으로 호출할 수도 있다. 이미지의 중심점을 선택하면 그림 3-9에 나타낸 것처럼 좀 더 보기 편한 이미지를 생성할 수 있다.

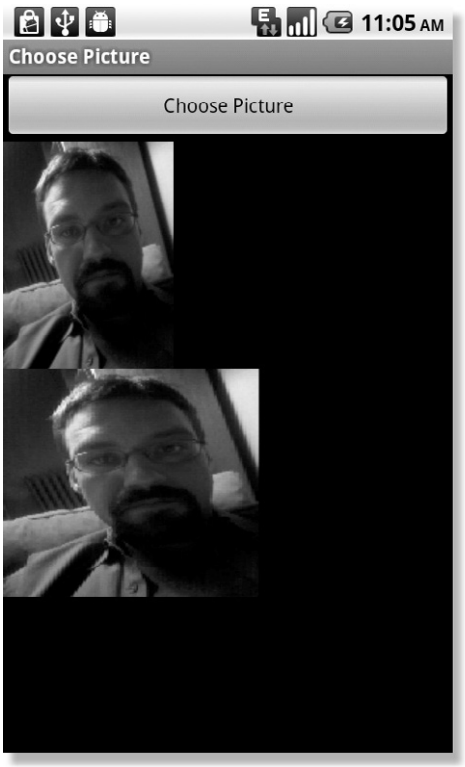
```
matrix.setRotate(15,bmp.getWidth()/2,bmp.getHeight()/2);
```

:: **그림 3-9** 이미지의 중심을 기준으로 한 회전

확대/축소

Matrix에서 유용하게 써먹을 수 있는 메소드에는 또한 setScale도 있다. setScale 메소드는 확대/축소에 필요한 각 축의 값을 나타내는 float 타입 실수 두 개를 받는다. 첫 번째 인수는 x축 비율이고, 두 번째 인수는 y축 비율이다. 그림 3-10은 다음 setScale 메소드를 호출한 결과다.

```
matrix.setScale(1.5f,1);
```



:: **그림 3-10** x축에 1.5배가 적용된 결과

축 이동

Matrix에서 가장 쓸모가 많은 메소드로는 setTranslate를 들 수 있다. 축 이동(translate)은 말 그대로 이미지를 x축과 y축에 따라 옮기는 동작을 의미한다. setTranslate 메소드는 각 축을 따라 옮길 양을 나타내는 float 타입 실수 두 개를 받는다. 첫 번째 인수는 x축 이동 값이고, 두 번째 인수는 y축 이동 값이다. 양수라면 x축에서는 오른쪽을 가리키고, y축에서는 아래쪽을 가리킨다. 반대로 음수로 지정하면 x축에서는 왼쪽을 가리키고, y축에서는 위쪽을 가리킨다.

```
setTranslate(1.5f,-10);
```

프리와 포스트

물론 지금까지 설명한 것은 빙산의 일각에 불과하다. 유용하게 써먹을 수 있는 메소드로 몇 가지가 더 있는데, 각 메소드마다 프리 버전과 포스트 버전을 함께 갖췄다. 두 버전을 사용하면 한 번에 여러 변형을 순서대로 적용할 수 있다. 예를 들어, preScale을 적용한 다음, setRotate나 setScale을 적용하고, 그 뒤에 postRotate를 적용할 수 있다. 수행되는 작업의 종류에 따라 그 작업 순서를 바꾸면 상당히 다양한 결과를 만들 수 있다. 그림 3-11은 다음 두 메소드 호출로 얻은 결과다.

```
matrix.setScale(1.5f, 1);
matrix.postRotate(15,bmp.getWidth()/2,bmp.getHeight()/2);
```

:: **그림 3-11** 확대된 다음 회전된 결과

거울 보기

지금 설명하는 한 쌍의 메소드는 특히 더 유용하다. setScale과 postTranslate, 이 두 메소드는 한 축을 기준으로 이미지를 뒤집어준다(두 축도 가능하다). 음수로 확대/축소하면 이미지는 좌표에서 음수 쪽 공간에 그려진다. 0, 0이 상단 왼쪽 끝점이므로, x축 값이 음수라면 이미지는 그 보다 더 왼쪽 공간에 그려진다. 따라서 postTranslate 메소드를 사용하여 오른쪽으로 옮겨갈 수 있도록 해줘야 한다. 결과는 그림 3-12와 같다.

```
matrix.setScale(-1, 1);
matrix.postTranslate(bmp.getWidth(),0);
```

:: **그림 3-12** 거울에 반사된 이미지

뒤집기

이번에는 y축 위에서 뒤집기가 일어난다. 그러니까 그림의 위아래가 뒤바뀌게 된다. 이미지를 180도 회전한 결과와 같다. 그림 3-13에 결과 이미지가 나타나 있다.

```
matrix.setScale(1, -1);
matrix.postTranslate(0, bmp.getHeight());
```

:: **그림 3-13** 뒤집기

그리기 이외의 방법

이들 메소드의 결점이라면 결과 이미지가 잘려 표시된다는 점을 들 수 있다. 온전하게 표시하려면 변형하고 나서 생긴 결과 이미지의 크기를 계산하여 Bitmap에 그려야 한다.

이런 귀찮은 점을 해결하는 방법이 있다. 빈 Bitmap으로 그려 넣지 말고 Bitmap부터 생성하면서 Matrix를 적용하면 된다.

이렇게 하면 Canvas 객체와 Paint 객체가 필요할 이유가 없어진다. 하지만 여기에도 단점은 있어서, 여러 변형 과정을 거쳐야 한다면 매번 Bitmap 객체를 변경할 수 없으므로 그때마다 다시 생성해야 한다는 것이다.

이럴 때 Bitmap 클래스의 정적 메소드인 createBitmap이 유용하다. 첫 번째 인수는 소스 Bitmap이고, 그 다음 인수는 x축, y축, 너비, 높이 값이다. 그 뒤에는 적용할 Matrix고, 마지막 인수는 불린으로서 이미지에 필터링을 적용할지 말지를 나타낸다. 지금은 필터가 포함된 매트릭스를 적용하는 상황이 아니므로 거짓으로 설정한다. 필터링에 관해서는 이 장 뒷부분에서 다룰 것이다.

```
Matrix matrix = new Matrix();
matrix.setRotate(15,bmp.getWidth()/2,bmp.getHeight()/2);
alteredBitmap = Bitmap.createBitmap(bmp, 0, 0, bmp.getWidth(), bmp.getHeight(), matrix, false);
alteredImageView.setImageBitmap(alteredBitmap);
```

두말하면 잔소리겠지만, 매트릭스를 동일한 방식으로 다뤘다. 그러나 두 번째 Bitmap(alteredBitmap)을 인스턴스화하면서 원래 이미지(bmp)를 소스로 사용하고 Matrix 객체를 넘겨줬다. 이렇게 하면 소스에서 Bitmap 객체의 크기로 확대/축소가 되고 이동된 Bitmap이 생성된다.

:: **그림 3-14** Bitmap이 생성될 때 적용된 매트릭스. Bitmap의 치수가 실제 이미지 데이터와 일치하도록 조정됨

이미지 프로세싱

이미지 에디팅/프로세싱을 다루는 또 다른 형태로는 픽셀 자체의 색상 값을 변경하는 과정도 있다. 색상 값을 변경하면 콘트라스트나 밝기, 전체 색조 등을 조절할 수 있다.

ColorMatrix

Canvas에 그릴 때 Matrix를 사용했던 것처럼 ColorMatrix 객체를 사용하여 Canvas에 그리는 데 사용되는 Paint를 변경할 수 있다.

ColorMatrix의 동작 방식도 Matrix와 비슷하다. 그러나 이미지의 픽셀에 동작하는 숫자 배열인 ColorMatrix는 x, y, z 좌표가 아니라 색상 값으로 동작한다. 각 픽셀의 Red, Green, Blue, Alpha가 기준이라는 말이다.

인수 없이 그 컨스트럭터를 호출하면 디폴트 ColorMatrix 객체를 구성할 수 있다.

```
ColorMatrix cm = new ColorMatrix();
```

이 ColorMatrix는 Canvas에 그리는 방식을 변경하는 데 사용되는데, 그리는 방식을 변경하려면 ColorMatrix 객체를 사용하여 구성된 ColorMatrixColorFilter 객체를 통해 Paint 객체에 적용한다.

```
paint.setColorFilter(new ColorMatrixColorFilter(cm));
```

지금까지 계속 사용해 온 Choose Picture 예제에서 그리는 부분에 이 코드를 집어넣으면 ColorMatrix를 다룰 수 있게 된다.

```
Bitmap bmp = BitmapFactory.decodeStream(getContentResolver().
openInputStream(imageFileUri), null, bmpFactoryOptions);
Bitmap alteredBitmap = Bitmap.createBitmap(bmp.getWidth(),
bmp.getHeight(),bmp.getConfig());
Canvas canvas = new Canvas(alteredBitmap);
Paint paint = new Paint();

ColorMatrix cm = new ColorMatrix();

paint.setColorFilter(new ColorMatrixColorFilter(cm));

Matrix matrix = new Matrix();
canvas.drawBitmap(bmp, matrix, paint);
alteredImageView.setImageBitmap(alteredBitmap);
chosenImageView.setImageBitmap(bmp);
```

디폴트 ColorMatrix를 이른바 아이덴티티(identity)라고 하는데, 적용될 때 이미지를 변경하지 않는다는 점에서 디폴트 Matrix 객체와 같다. 배열에 든 값들을 보면 어떻게 동작하는지 이해하기 쉽다.

```
1 0 0 0 0
0 1 0 0 0
0 0 1 0 0
0 0 0 1 0
```

보이는 것처럼 float 타입 실수 20개로 이뤄진 배열이다. 첫 번째 행은 각 픽셀의 빨강 부분에 적용될 작업을 구성한다. 두 번째 행은 초록 부분에 영향을 끼치고, 세 번째는 파랑 부분, 마지막 네 번째 행은 픽셀의 알파 값에 적용할 작업을 의미한다.

각 행 안에서 첫 번째 숫자는 픽셀의 빨강 값에 사용되는 배수를 나타낸다. 두 번째는 초록에 사용되는 배수를, 세 번째는 파랑, 네 번째는 알파에 사용되는 배수를 나타내며, 마지막 다섯 번째 숫자는 어떤 것에도 배수로 곱해지지 않는다. 이 값들은 전부 더해져서 작업하는 픽셀을 변경한다.

중간 정도의 회색 픽셀이 있다면 빨강 값은 128, 파랑 값도 128, 초록 값도 128이 되고 알파 값은 0이다(불투명하다는 의미). 이 픽셀을 색상 매트릭스로 실행하면 계산식은 다음과 같이 된다.

```
New Red Value = 1*128 + 0*128 + 0*128 + 0*0 + 0
New Blue Value = 0*128 + 1*128 + 0*128 + 0*0 + 0
New Green Value = 0*128 + 0*128 + 1*128 + 0*0 + 0
New Alpha Value = 0*128 + 0*128 + 0*128 + 1*0 + 0
```

물론 계산하면 값들은 전부 128이 된다. 픽셀에 사용한 다른 색상에서도 각 행이 해당 색상 위치가 1이고 나머지는 영(0)이므로 같은 결과가 나온다. 다음은 전반적으로 빨강 값을 두 배로 한다.

```
2 0 0 0 0
0 1 0 0 0
0 0 1 0 0
0 0 0 1 0
```

지금 이 내용을 코드로 구현하면 다음과 같다.

```
ColorMatrix cm = new ColorMatrix();
cm.set(new float[] {
    2, 0, 0, 0, 0,
    0, 1, 0, 0, 0,
    0, 0, 1, 0, 0,
    0, 0, 0, 1, 0
});
paint.setColorFilter(new ColorMatrixColorFilter(cm));
```

마찬가지 방법으로 어떤 색이든 일일이 변경할 수 있다.

콘트라스트와 밝기 변경하기

색상 값을 늘리거나 줄이면 이미지의 콘트라스트와 밝기를 변경할 수 있다.

다음은 각 색상 채널의 명도(intensity)를 두 배로 늘리는 코드다. 이렇게 하면 이미지의 밝기와 콘트라스트 둘 다에 영향을 끼친다. 결과는 그림 3-15다.

```
ColorMatrix cm = new ColorMatrix();
float contrast = 2;
cm.set(new float[] {
    contrast, 0, 0, 0, 0,
    0, contrast, 0, 0, 0,
    0, 0, contrast, 0, 0,
    0, 0, 0, 1, 0 });
paint.setColorFilter(new ColorMatrixColorFilter(cm));
```

:: **그림 3-15** 각 색의 명도가 두 배로 돼 밝기와 콘트라스트 둘 다 늘어난 ColorMatrix

여기서는 두 효과가 서로 연결돼 있다. 밝기를 늘리지 않고 콘트라스트만 늘리려면 밝기를 줄여 명도(color intensity)의 증가분을 상쇄해야 한다.

일반적으로 밝기를 조정할 때는 각 색에 해당하는 매트릭스에서 마지막 열을 사용하는 것이 더 쉽다. 마지막 열은 기존 색상 값이 늘어나지 않고 해당 색상 값에 더해지는 양이다.

따라서 밝기를 줄이려면 다음과 같이 매트릭스 코드를 사용해야 한다.

```
ColorMatrix cm = new ColorMatrix();
float brightness = -25;
cm.set(new float[] {
    1, 0, 0, 0, brightness,
    0, 1, 0, 0, brightness,
    0, 0, 1, 0, brightness,
    0, 0, 0, 1, 0 });
paint.setColorFilter(new ColorMatrixColorFilter(cm));
```

지금 이 두 가지 변형을 하나로 합치면 다음 결과가 나온다.

```
ColorMatrix cm = new ColorMatrix();

float contrast = 2;
float brightness = -25;
cm.set(new float[] {
    contrast, 0, 0, 0, brightness,
    0, contrast, 0, 0, brightness,
    0, 0, contrast, 0, brightness,
    0, 0, 0, contrast, 0 });
paint.setColorFilter(new ColorMatrixColorFilter(cm));
```

이 작업의 결과는 그림 3–16과 같다.

:: **그림 3–16** 밝기를 변경하지 않고 콘트라스트에 영향을 끼치기 위해 각 색의 명도가 두 배로 되었지만 밝기가 줄어든 ColorMatrix

채도 변경하기

우리가 어떤 작업을 할 때 그에 해당하는 공식을 일일이 알 필요가 없는 것은 다행이라고 할 수 있다. 예를 들어, ColorMatrix에는 채도(saturation)를 변경하는 메소드가 들었다.

```
ColorMatrix cm = new ColorMatrix();
cm.setSaturation(.5f);
paint.setColorFilter(new ColorMatrixColorFilter(cm));
```

1보다 큰 값을 넘겨주면 채도가 늘어난다. 영(0)과 1 사이의 값이라면 채도는 줄어든다. 영(0)은 회색조(grayscale)의 이미지를 생성한다.

이미지 합성하기

합성이란 두 이미지를 한 데 합치는 과정을 말한다. 합성하면 두 이미지의 특징이 전부 눈에 보이게 된다.

안드로이드 SDK로 합성을 구현하려면 우선 Canvas에 어떤 Bitmap을 그리고, 그 다음 두 번째 Bitmap을 동일한 Canvas에 그려야 한다. 두 번째 이미지를 그릴 때 Paint에 transfermode(Xfermode)를 지정해야 하는 점이 처음 그릴 때와 다르다.

transfermode로 사용할 수 있는 클래스들은 전부 기본 클래스인 Xfermode에서 파생된 것인데, 이 중에는 PorterDuffXfermode라는 클래스가 있다. 이 클래스는 토머스 포터(Thomas Porter)와 톰 더프(Tom Duff)라는 이름을 본뜬 것인데, 이 두 사람은 1984년 ACM의 컴퓨터 그래픽스 분과인 SIGGRAPH의 학술지에 "디지털 이미지의 합성"이라는 논문을 발표했다. 이 논문에는 이미지 서로서로 위에 다른 이미지를 그리기 위한 일련의 규칙들이 상세히 기술돼 있다. 이 규칙들에는 이미지의 어느 부분이 결과 이미지에 나타나는지 정의돼 있다.

포터와 더프가 고안한 규칙들은 확장된 규칙을 포함하여 전부 안드로이드의 PorterDuff.Mode 클래스에 담겨 있으며, 그 중에는 다음과 같은 내용이 있다.

- android.graphics.PorterDuff.Mode.SRC: 지금 이 예제에서 적용할 Paint에 해당하는 source만이 그려진다.
- android.graphics.PorterDuff.Mode.DST: 캔버스에 있던 원래 이미지인 destination만이 나타난다.

SRC, DST 규칙을 따라 각 이미지의 어느 부분이 그려지게 되는지 결정하는 규칙들이 존재하는데, 이 규칙들은 일반적으로 이미지들이 크기가 다르거나 이미지에 투명한 부분이 있을 때 적용된다.

- android.graphics.PorterDuff.Mode.DST_OVER: 소스 이미지 위로 대상 이미지가 그려진다.
- android.graphics.PorterDuff.Mode.DST_IN: 소스 이미지와 대상 이미지가 서로 교차하는 곳에만 대상 이미지가 그려진다.
- android.graphics.PorterDuff.Mode.DST_OUT: 소스 이미지와 대상 이미지가 서로 교차하지 않는 곳에만 대상 이미지가 그려진다.
- android.graphics.PorterDuff.Mode.DST_ATOP: 대상 이미지가 소스 이미지와 교차하는 곳에 그려지고 나머지 곳에는 소스 이미지가 그려진다.
- android.graphics.PorterDuff.Mode.SRC_OVER: 대상 이미지 위로 소스 이미지가 그려진다.
- android.graphics.PorterDuff.Mode.SRC_IN: 대상 이미지와 소스 이미지가 서로 교차하는 곳에만 소스 이미지가 그려진다.
- android.graphics.PorterDuff.Mode.SRC_OUT: 대상 이미지와 소스 이미지가 서로 교차하지 않는 곳에만 소스 이미지가 그려진다.
- android.graphics.PorterDuff.Mode.SRC_ATOP: 소스 이미지가 대상 이미지와 교차하는 곳에 그려지고 나머지 곳에는 대상 이미지가 그려진다.
- android.graphics.PorterDuff.Mode.XOR: 소스 이미지와 대상 이미지가 겹치는 곳을 제외한 모든 곳에 그려진다. 겹치는 곳에는 어느 이미지도 그려지지 않는다.

추가로 네 가지 규칙이 정의돼 있다. 이들 규칙에서는 한 이미지가 다른 이미지 위에 놓일 때 두 이미지가 어떤 식으로 합쳐지는지 결정한다.

- android.graphics.PorterDuff.Mode.LIGHTEN: 각 위치의 두 이미지에서 가장 밝은 픽셀을 가져다 보여준다.
- android.graphics.PorterDuff.Mode.DARKEN: 각 위치의 두 이미지에서 가장 어두운 픽셀을 가져다 보여준다.
- android.graphics.PorterDuff.Mode.MULTIPLY: 각 위치의 두 픽셀을 곱한 다음, 255로 나눈다. 표시할 새 픽셀을 생성하는 데 이 값을 사용한다. 결과 색 = 위쪽 색 × 아래쪽 색 / 255
- android.graphics.PorterDuff.Mode.SCREEN: 각 색의 반대 색을 만들어 동일한 작업(두 픽셀을 곱하고 다시 255로 나누는 작업)을 수행한 다음, 다시 반대로 돌린다. 결과 색 = 255 − (((255 − 위쪽 색)×(255 − 아래쪽 색)) / 255)

지금까지 설명한 내용을 예제 애플리케이션으로 구체화하기로 한다.

```
package com.apress.proandroidmedia.ch3.choosepicturecomposite;

import java.io.FileNotFoundException;

import android.app.Activity;
import android.content.Intent;
import android.graphics.Bitmap;
import android.graphics.BitmapFactory;
import android.graphics.Canvas;
import android.graphics.Paint;
import android.graphics.PorterDuffXfermode;
import android.net.Uri;
import android.os.Bundle;
import android.util.Log;
import android.view.Display;
import android.view.View;
import android.view.View.OnClickListener;
import android.widget.Button;
import android.widget.ImageView;

public class ChoosePictureComposite extends Activity implements OnClickListener {
```

액티비티 기반 표준 애플리케이션을 생성할 텐데, 이름은 Choose Picture Composite라고 붙였다. 액티비티에서는 Button 클릭에 반응하도록 OnClickListener를 구현한다.

두 이미지를 합성하는 상황이므로 사용자가 이미지를 두 개 선택했는지 확인하고 나서 합성 버전을 그리려고 해야 할 것이다. 상수 두 개, Button의 누르기 동작에 쓸 변수 하나와 Button이 눌렸는지 추적할 불린 두 개를 만들어 이미지 두 개가 선택됐는지 확인하는 데 사용한다. 물론 Button 객체가 두 개 필요하다.

```
static final int PICKED_ONE = 0;
static final int PICKED_TWO = 1;

boolean onePicked = false;
boolean twoPicked = false;

Button choosePicture1, choosePicture2;
```

최종 합성된 이미지를 표시하기 위해 ImageView가 하나 필요하다. 또한 Bitmap 객체도 두 개가 필요하다. 선택된 이미지 하나에 하나씩 사용할 것이다.

```
ImageView compositeImageView;

Bitmap bmp1, bmp2;
```

앞의 예제에서처럼 그릴 Canvas와 Paint 또한 필요하다.

```
Canvas canvas;
Paint paint;

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

    compositeImageView = (ImageView) this.findViewById(R.id.CompositeImageView);
```

```
        choosePicture1 = (Button) this.findViewById(R.id.ChoosePictureButton1);
        choosePicture2 = (Button) this.findViewById(R.id.ChoosePictureButton2);

        choosePicture1.setOnClickListener(this);
        choosePicture2.setOnClickListener(this);
    }
```

각 Button의 OnClickListener를 이 클래스가 되도록 설정했으므로 이에 응답할 onClick 메소드를 구현해야 한다. 어느 것이 클릭됐는지 구별하려면 넘겨받은 View 객체를 각 Button 객체와 비교해야 한다. 같은 것이 클릭된 버튼이다.

앞에서 정의한 상수들 중 하나의 값으로 which라는 변수를 설정한다. 그래야 어느 Button이 눌렸는지 추적이 가능하다. 이 변수가 우리의 Gallery 애플리케이션을 통해 전달된다. Gallery 애플리케이션은 ACTION_PICK 인텐트로 인스턴스화가 진행된다. 앞의 예제에 나타낸 것처럼 이 인텐트가 이미지 선택 모드에서 애플리케이션이 시작할 수 있도록 해준다.

```
    public void onClick(View v) {

        int which = -1;

        if (v == choosePicture1) {
            which = PICKED_ONE;
        } else if (v == choosePicture2) {
            which = PICKED_TWO;
        }
        Intent choosePictureIntent = new Intent(Intent.ACTION_PICK, ↵
android.provider.MediaStore.Images.Media.EXTERNAL_CONTENT_URI);
        startActivityForResult(choosePictureIntent, which);
    }
```

사용자가 이미지를 선택하고 나면 우리의 onActivityResult 메소드가 호출된다. startActivityForResult 메소드를 통해 넘겨준 변수가 첫 번째 파라미터로 다시 넘어온다. 이 변수를 requestCode라고 부를 것이다. 이렇게 하면 첫 번째 이미지와 두 번째 이미지 중 어느 이미지를 사용자가 방금 전 선택했는지 알 수 있다. 이 값을 사용하여 선택

된 이미지를 어느 Bitmap 객체에 로드할지 결정한다.

```
    protected void onActivityResult(int requestCode, int resultCode,
            Intent intent) {
        super.onActivityResult(requestCode, resultCode, intent);

        if (resultCode == RESULT_OK) {
            Uri imageFileUri = intent.getData();

            if (requestCode == PICKED_ONE) {
                bmp1 = loadBitmap(imageFileUri);
                onePicked = true;
            } else if (requestCode == PICKED_TWO) {
                bmp2 = loadBitmap(imageFileUri);
                twoPicked = true;
            }
```

두 이미지가 선택되고 두 Bitmap 객체의 인스턴스화가 끝나면, 본격적인 합성 작업으로 나아갈 수 있다. 지금 이 과정은 앞의 예제들에서 보인 과정과 매우 흡사하다. 우선 첫 번째 Bitmap인 bmp1과 동일한 크기, 동일한 구성의 빈 변경 가능한(mutable) Bitmap을 만든다. 그런 다음, Canvas와 Paint를 구성한다. 첫 번째 Bitmap(bmp1)을 이 캔버스에 그릴 것이다. 이렇게 하면 합성 작업의 대상이 된다.

이제 Paint 객체에 transfermode를 설정한다. 새로운 PorterDuffXfermode 객체를 인스터스화한다. 그러려면 모드를 정의하는 상수들 중 하나를 넘겨준다. 그러고 나서 두 번째 Bitmap을 Canvas에 그리고, ImageView가 새로운 Bitmap이 되도록 설정한다. 여기서는 MULTIPLY 모드를 사용한다.

```
            if (onePicked && twoPicked) {
                Bitmap drawingBitmap = Bitmap.createBitmap(bmp1.getWidth(), ➥
bmp1.getHeight(), bmp1.getConfig());
                canvas = new Canvas(drawingBitmap);
                paint = new Paint();
                canvas.drawBitmap(bmp1, 0, 0, paint);
                paint.setXfermode(new PorterDuffXfermode(android.graphics. ➥
```

```
PorterDuff.Mode.MULTIPLY));
                canvas.drawBitmap(bmp2, 0, 0, paint);

                compositeImageView.setImageBitmap(drawingBitmap);
            }
        }
    }
```

다음은 1장에서 정의했던 것과 같은 헬퍼 클래스다. 이 헬퍼 클래스는 화면 크기보다 크지 않도록 크기가 맞춰지는 Bitmap을 URI에서 로드한다.

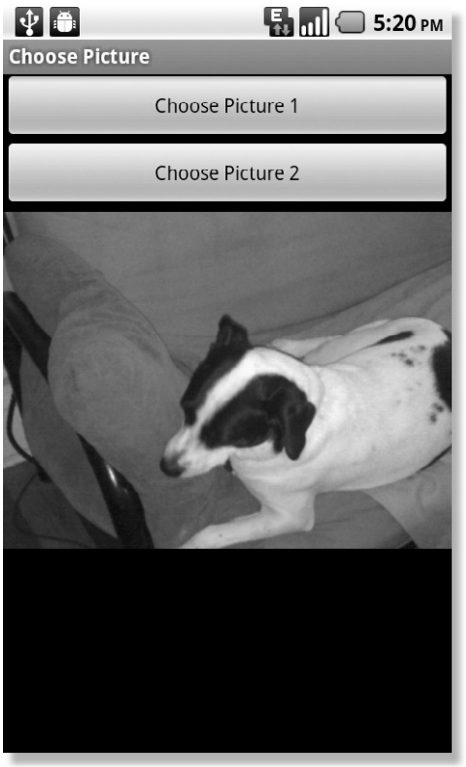
```
    private Bitmap loadBitmap(Uri imageFileUri) {
        Display currentDisplay = getWindowManager().getDefaultDisplay();

        float dw = currentDisplay.getWidth();
        float dh = currentDisplay.getHeight();
        // ARGB_4444로 진행한다.

        Bitmap returnBmp = Bitmap.createBitmap((int) dw, (int) dh,Bitmap.Config.ARGB_4444);

        try {
            // 이미지 자체가 아니라 이미지의 치수를 로드한다.
            BitmapFactory.Options bmpFactoryOptions = new BitmapFactory.Options();
            bmpFactoryOptions.inJustDecodeBounds = true;
            returnBmp = BitmapFactory.decodeStream(getContentResolver().
openInputStream(imageFileUri), null, bmpFactoryOptions);

            int heightRatio = (int) Math.ceil(bmpFactoryOptions.outHeight / dh);
            int widthRatio = (int) Math.ceil(bmpFactoryOptions.outWidth / dw);

            Log.v("HEIGHTRATIO", "" + heightRatio);
            Log.v("WIDTHRATIO", "" + widthRatio);

            // 두 비율 다 1보다 크면 이미지의 어느 한쪽이 화면보다 커진다.
            if (heightRatio > 1 && widthRatio > 1) {
                if (heightRatio > widthRatio) {
                    // 높이 비율이 더 커서 그에 맞춰진다.
                    bmpFactoryOptions.inSampleSize = heightRatio;
```

```
                } else {
                    // 너비 비율이 더 커서 그에 맞춰진다.
                    bmpFactoryOptions.inSampleSize = widthRatio;
                }
            }

            // 실제로 디코딩한다.
            bmpFactoryOptions.inJustDecodeBounds = false;
            returnBmp = BitmapFactory.decodeStream(getContentResolver().
openInputStream(imageFileUri), null, bmpFactoryOptions);
        } catch (FileNotFoundException e) {
            Log.v("ERROR", e.toString());
        }

        return returnBmp;
    }
}
```

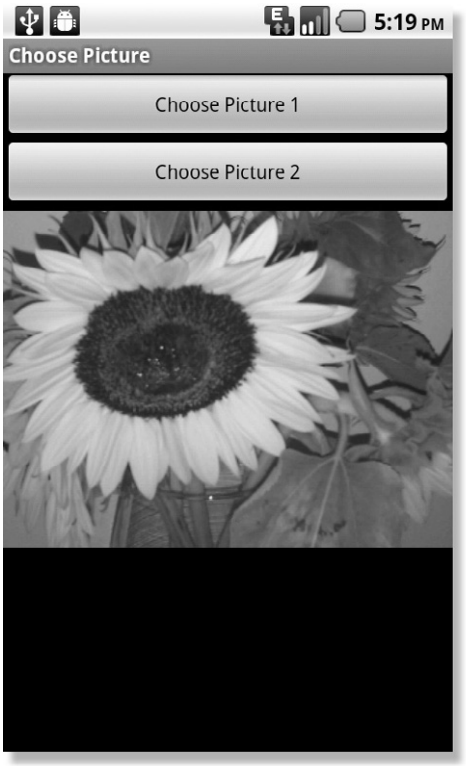
여기까지다. 다음은 이 액티비티에 사용되는 레이아웃 XML이다.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent">
    <Button
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:id="@+id/ChoosePictureButton1" android:text="Choose Picture 1"/>
    <Button
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:id="@+id/ChoosePictureButton2" android:text="Choose Picture 2"/>
    <ImageView android:layout_width="wrap_content" android:layout_height=
"wrap_content" android:id="@+id/CompositeImageView"></ImageView>
</LinearLayout>
```

이번 예제로 transfermode를 달리하여 얻은 결과는 그림 3–17에서 3–22까지 나타나 있다.



:: **그림 3-17** PorterDuffXfermode로 android.graphics.PorterDuff.Mode.DST를 사용한 결과, Picture 1로 선택된 이미지만 표시됐다



:: **그림 3-18** PorterDuffXfermode로 android.graphics.PorterDuff.Mode.SRC를 사용한 결과, Picture 2로 선택된 이미지만 표시됐다

:: **그림 3-19** PorterDuffXfermode로 android.graphics.PorterDuff.Mode.MULTIPLY를 사용한 결과, 두 이미지가 합쳐졌다

:: **그림 3-20** PorterDuffXfermode로 android.graphics.PorterDuff.Mode.LIGHTEN을 사용한 결과

:: **그림 3-21** PorterDuffXfermode로 android.graphics.PorterDuff.Mode.DARKEN을 사용한 결과

:: **그림 3-22** PorterDuffXfermode로 android.graphics.PorterDuff.Mode.SCREEN을 사용한 결과

정리

이 장을 통해 우리는 안드로이드가 비록 크기, 메모리, 프로세서의 성능이 제한된 운영 체제이긴 해도 꽤 정교한 이미지 프로세싱을 지원한다는 사실을 배웠다. 그리고 이미지 프로세싱의 많은 기능들이 이 장에 소개됐다. 하지만 이미지 프로세싱을 파헤치는 과정이 이 장으로만 끝나는 것은 아니다. 다음 장에서는 아예 이미지를 새로 생성할 수 있도록 해주는 API를 살펴볼 것이고, 터치스크린과 같은 센서를 활용하여 이것저것 심도 깊은 조작을 해볼 것이다.