14 센서

이번 장에서 다루는 내용

- ☑ 센서 매니저 이용법
- ☑ 사용할 수 있는 센서들의 종류
- ☑ 센서 모니터링과 센서 값 해석
- ☑ 나침반, 가속도계, 방향 센서 이용법
- ☑ 방향 기준 좌표계의 재배치
- ☑ 진동 제어

요즘 나오는 휴대폰은 단순히 인터넷만 가능한 통신 기기가 아니다. 최신 스마트폰은 마이크, 카메라, 가속도 센서, 나침반, 온도계, 조도 센서 등을 갖추고 있어 여러분의 지각 능력을 확장시켜 주는 특별한 기기가 되었다.

카메라와 마이크에 대해서는 11장에서 살펴봤다. 이번 장에서는 안드로이드 기기에서 잠재적으로 사용 가능한 주변 센서에 대해 살펴본다.

물리적 성질과 환경적 특성을 감지하는 센서는 모바일 애플리케이션의 사용자 경험 향상을 위한 흥미로운 혁신을 제공한다. 전자식 나침반, 중력 센서, 조도 센서, 근접 센서의 결합은 증강 현실이나 신체 움직임에 기반한 입력처럼 새로운 방식으로 기기를 다룰 수 있도록 여러 가지 가능성을 제시한다.

이번 장에서는 안드로이드에서 사용 가능한 센서들을 소개하며, 센서 매니저를 이용해 이들 센서를 모니터하는 방법을 살펴본다. 가속도 센서와 방향 센서에 대해 자세히 살펴보며, 이 센서들을 가지고 기기의 방향과 가속도의 변화를 측정해볼 것이다. 이는 특히 모션 기반 사용자 인터페이스를 만들 때 유용하며, 여러분의 위치 기반 애플리케이션에 새로운 차원을 추가할 수 있게 해준다.

또한 기기의 진동 기능을 제어하는 법에 대해서도 배운다. 이를 이용하면 애플리케이션에 포스 피드백force feedback[1] 같은 기능을 넣을 수 있다.

센서와 센서 매니저 이용하기

센서 매니저는 안드로이드 기기에서 사용할 수 있는 센서 하드웨어를 관리하는 데 사용된다. 센서 매니저를 이용하려면, 아래의 코드에서 보이는 것처럼 getSystemService를 이용해 센서 매니저 서비스의 레퍼런스를 얻어온다.

```
String service_name = Context.SENSOR_SERVICE;
SensorManager sensorManager = (SensorManager)getSystemService(service_name);
```

센서 소개

안드로이드는 위치 기반 서비스와 마찬가지로 각 기기의 센서 구현을 추상화하고 있다. Sensor 클래스는 각각의 하드웨어 센서가 지닌 속성을 기술하는 데 사용된다. 여기에는 센서의 종류, 이름, 제조사, 그리고 정확도와 범위에 관련된 세부 정보 등이 포함된다.

Sensor 클래스에는 센서 객체가 추상화하고 있는 하드웨어 센서의 종류를 나타내기 위한 일련의 상수들이 포함되어 있다. 이 상수들은 Sensor.TYPE_<센서 종류>의 형태로 정의되어 있다. 다음 절에서는 안드로이드가 지원하는 센서들의 종류를 설명하며, 이 센서들을 찾아 이용하는 법을 배운다.

안드로이드가 지원하는 센서들의 종류

다음의 목록은 현재 안드로이드에서 이용 가능한 센서들의 종류를 보여준다. 애플리케이션에서 실제로 이용할 수 있는 센서들의 종류는 호스트 기기의 하드웨어에 따라 다르다는 사실을 기억하자.

- Sensor.TYPE_ACCELEROMETER 3축 가속도 센서. 세 축에 대한 현재 가속도를 m/s^2으로 리턴한다. 가속도 센서에 대해서는 잠시 뒤에 자세히 살펴본다.

[1] **옮긴이** 게임 도중 일어나는 충격이나 진동을 조이스틱이나 레이싱 휠 등의 게임용 장비를 통해 실제로 체감할 수 있게 하는 기능을 말한다.

- Sensor.TYPE_GYROSCOPE 자이로스코프 센서. 세 축에 대한 기기의 현재 방향을 도(°) 단위로 리턴한다.
- Sensor.TYPE_LIGHT 주변 광 센서. 주변 조도를 럭스(lux) 단위로 리턴한다. 광 센서는 화면 밝기를 동적으로 조절하는 데 흔히 쓰인다.
- Sensor.TYPE_MAGNETIC_FIELD 자기장 센서. 세 축에 대한 현재 자기장을 마이크로테슬라(μT) 단위로 측정한다.
- Sensor.TYPE_ORIENTATION 방향 센서. 세 축에 대한 기기의 현재 방향을 도(°) 단위로 리턴한다. 방향 센서에 대해서는 잠시 뒤에 자세히 살펴본다.
- Sensor.TYPE_PRESSURE 압력 센서. 기기에 가해진 현재 압력을 킬로파스칼(kPa) 단위로 리턴한다.
- Sensor.TYPE_PROXIMITY 근접 센서. 기기와 대상 물체 간의 거리를 미터(m) 단위로 표시한다. 대상 물체를 선택하는 방법과 지원되는 거리는 근접 센서의 하드웨어 구현에 따라 다르다. 보통 근접 센서는 사용자가 통화를 위해 기기를 귀에 가져다 댈 때, 자동으로 화면 밝기를 조절하거나 음성 명령을 실행하기 위한 용도로 쓰인다.
- Sensor.TYPE_TEMPERATURE 온도 센서. 온도를 섭씨(℃) 단위로 리턴한다. 하드웨어 구현에 따라 주변 실온을 리턴할 수도 있고, 기기의 배터리 온도를 리턴할 수도 있으며, 원격 센서 온도를 리턴할 수도 있다.

센서 찾기

안드로이드 기기는 같은 종류의 센서를 여러 개 가질 수 있다. 같은 종류의 센서들 중 기본 구현을 찾으려면, 앞 절에서 설명한 센서들의 종류를 나타내는 상수들 중에서 원하는 것을 가지고 센서 매니저의 getDefaultSensor 메서드를 호출한다.

아래의 코드는 기본 자이로스코프를 리턴한다. 주어진 종류에 대한 기본 센서가 존재하지 않는 경우에는 null이 리턴된다.

```
Sensor defaultGyroscope = sensorManager.getDefaultSensor(Sensor.TYPE_GYROSCOPE);
```

getSensorList를 이용하면, 이용 가능한 센서들 중 주어진 종류에 해당하는 모든 센서들의 리스트를 얻어올 수 있다. 다음의 코드는 이용 가능한 모든 압력 센서 객체들을 얻어온다.

```
List<Sensor> pressureSensors = sensorManager.getSensorList(Sensor.TYPE_PRESSURE);
```

아래의 코드에서 보이는 것처럼 getSensorList에 Sensor.TYPE_ALL을 전달해 호출하면, 호스트 플랫폼에서 이용할 수 있는 모든 센서들의 리스트를 얻어올 수 있다.

```
List<Sensor> allSensors = sensorManager.getSensorList(Sensor.TYPE_ALL);
```

이렇게 하면, 호스트 플랫폼에서 이용할 수 있는 센서들의 종류를 파악할 수 있다.

센서 이용하기

코드 14-1은 하드웨어 센서의 결과 값을 모니터하기 위한 표준 패턴을 보여준다. 이번 장 후반부에서는 여러 센서들 중에서도 방향 센서와 가속도 센서에 대해 자세히 살펴본다.

SensorEventListener를 정의한 다음, 센서 값을 모니터하려면 onSensorChanged 메서드를, 센서의 정확도 변경을 다루려면 onAccuracyChanged 메서드를 구현한다.


Available for download on Wrox.com

코드 14-1 센서 이벤트 리스너의 골격 코드

```
final SensorEventListener mySensorEventListener = new SensorEventListener() {
    public void onSensorChanged(SensorEvent sensorEvent) {
        // TODO: 센서의 변화를 모니터한다.
    }

    public void onAccuracyChanged(Sensor sensor, int accuracy) {
        // TODO: 센서의 정확도가 변경될 경우 적절히 대응한다.
    }
};
```

onSensorChanged 메서드의 SensorEvent 매개변수는 센서 이벤트를 기술하는 네 가지 속성을 가지고 있다.

- sensor 이벤트를 발생시킨 센서 객체
- accuracy 이벤트가 발생했을 당시 센서의 정확도. 낮음low, 중간medium, 높음high, 신뢰할 수 없음unreliable으로 표현한다. 다음 리스트를 참고하자.
- values 감지된 새로운 값(들)을 담고 있는 float 배열. 각 센서 종류에 따라 리턴되는 값은 다음 절에서 설명한다.

- timestamp 센서 이벤트가 발생한 시간. 나노초nanoseconds 단위

onAccuracyChanged 메서드를 이용하면, 센서의 정확도에 생기는 변화를 별도로 모니터할 수 있다. 이 두 핸들러 모두에서 accuracy 값은 모니터링되고 있는 센서의 정확도에 대한 피드백을 아래의 상수들 중 하나로 나타낸다.

- SensorManager.SENSOR_STATUS_ACCURACY_LOW 센서가 낮은 정확도로 보고하고 있으며, 조정이 필요함을 나타낸다.
- SensorManager.SENSOR_STATUS_ACCURACY_MEDIUM 센서 데이터가 평균적인 정확도로 측정된 것이며, 조정을 통해 측정 능력을 향상시킬 수 있음을 나타낸다.
- SensorManager.SENSOR_STATUS_ACCURACY_HIGH 센서가 가능한 가장 높은 정확도로 보고하고 있음을 나타낸다.
- SensorManager.SENSOR_STATUS_UNRELIABLE 센서 데이터를 신뢰할 수 없음을 나타낸다. 조정이 필요하며, 현재로서는 측정이 불가능하다.

센서 이벤트를 받으려면, 센서 매니저에 센서 이벤트 리스너를 등록해야 한다. 이때, 관찰할 센서 객체와 원하는 업데이트 속도를 지정한다. 아래의 예는 기본 근접 센서에 대한 센서 이벤트 리스너를 보통의 업데이트 속도로 등록한다.

```
Sensor sensor = sensorManager.getDefaultSensor(Sensor.TYPE_PROXIMITY);
sensorManager.registerListener(mySensorEventListener,
                               sensor,
                               SensorManager.SENSOR_DELAY_NORMAL);
```

센서 매니저는 적절한 업데이트 속도를 보다 쉽게 선택할 수 있도록, 이를 상수로 정의해두고 있다. 이 상수들을 반응이 빠른 것에서부터 느린 것 순으로 나열하면 아래와 같다.

- SensorManager.SENSOR_DELAY_FASTEST 가능한 가장 빠른 업데이트 속도를 지정한다.
- SensorManager.SENSOR_DELAY_GAME 게임 제어에 사용하기 적합한 업데이트 속도를 선택한다.
- SensorManager.SENSOR_DELAY_NORMAL 기본 업데이트 속도를 지정한다.
- SensorManager.SENSOR_DELAY_UI UI 기능을 업데이트하는 데 적합한 속도를 지정한다.

여러분이 선택한 속도가 무조건 반영되는 것은 아니다. 센서 매니저는 여러분이 지정한 것보다 더 빠르거나 느린 결과를 리턴할 수도 있다. 그러나 센서 매니저는 빠른 값을 리턴하려는 경향이 있다. 애플리케이션에서 센서를 이용하는 데 드는 리소스 비용을 최소화하려면, 적절한 속도 중 가장 느린 것을 선택해야 한다.

애플리케이션이 더 이상 센서 값을 업데이트받을 필요가 없을 때는 센서 이벤트 리스너를 등록 해제해야 한다.

```
sensorManager.unregisterListener(mySensorEventListener);
```

액티비티가 활성화되어 있을 때만 센서 이벤트 리스너가 사용되도록 액티비티의 onResume 메서드에서 등록하고, onPause 메서드에서는 등록 해제하는 것이 좋다.

센서 값 해석하기

onSensorChanged에 전달되는 SensorEvent 객체의 values 값은 모니터링하고 있는 센서의 종류에 따라 크기와 구성이 서로 다르다.

이에 대한 자세한 내용이 표 14-1에 정리되어 있다. 가속도 센서, 방향 센서, 자기장 센서에 대해서는 이어지는 절에서 자세히 살펴본다.

각 센서 별로 리턴되는 값에 대한 보다 자세한 설명은 안드로이드 개발자 사이트인 http://developer.android.com/reference/android/hardware/SensorEvent.html을 참고하자.

표 14-1 센서 리턴 값

센서 종류	값 개수	값 구성	설명
TYPE_ACCELEROMETER	3	values[0]: X축(Lateral) values[1]: Y축(Longitudinal) values[2]: Z축(Vertical)	세 축에 대한 가속도. 단위는 m/s^2. 센서 매니저는 SensorManager.GRAVITY_* 형태로 정의된 일련의 중력 상수들을 가지고 있다.

표 14-1 (계속)

센서 종류	값 개수	값 구성	설명
TYPE_GYROSCOPE	3	values[0]: Z축(Azimuth) values[1]: X축(Pitch) values[2]: Y축(Roll)	세 축에 대한 기기의 방향. 단위는 도(°)
TYPE_LIGHT	1	values[0]: 조도(Illumination)	단위는 럭스(lux). 센서 매니저는 SensorManager.LIGHT_* 형태로 정의된 일련의 상수들을 가지고 있다. 이 상수들은 다양한 표준 조도를 나타낸다.
TYPE_MAGNETIC_FIELD	3	values[0]: X축(Lateral) values[1]: Y축(Longitudinal) values[2]: Z축(Vertical)	주변 자기장. 단위는 마이크로 테슬라(μT)
TYPE_ORIENTATION	3	values[0]: Z축(Azimuth) values[1]: X축(Pitch) values[2]: Y축(Roll)	세 축에 대한 기기의 방향. 단위는 도(°)
TYPE_PRESSURE	1	values[0]: 압력(Pressure)	단위는 킬로파스칼(kPa)
TYPE_PROXIMITY	1	values[0]: 거리(Distance)	단위는 미터(m)
TYPE_TEMPERATURE	1	values[0]: 온도 (Temperature)	단위는 섭씨(℃)

나침반 센서, 가속도 센서, 방향 센서 이용하기

요즘 나오는 기기들은 대부분 방향 센서와 가속도 센서를 가지고 있다. 덕분에 애플리케이션에서 기기의 움직임과 방향을 이용할 수 있게 됐다.

최근에는 이러한 센서들이 점점 더 보편화되어 닌텐도 위Nintendo Wii 같은 게임 컨트롤러나, 애플 아이폰Apple iPhone, 팜 프리Palm Pre, 안드로이드 폰 같은 스마트폰에서 독특한 방식으로 활용되고 있다.

가속도 센서와 나침반은 기기의 방향, 방위, 움직임에 기반한 기능을 제공하는 데 쓰인다. 최근에는 전통적인 터치스크린, 트랙볼, 키보드와 더불어 이 기능을 이용한 입력 메커니즘을 제공하는 것이 트랜드다.

나침반과 가속도 센서의 이용가능 여부는 애플리케이션이 실행되는 하드웨어에 따라 좌우된다. 이용가능할 경우 해당 센서들은 센서 매니저를 통해 노출되며, 아래와 같은 일들을 할 수 있게 된다.

- 기기의 현재 방향 측정
- 방향 변화 모니터링 및 추적
- 사용자가 마주하고 있는 방향 식별
- X, Y, Z 방향 가속도(이동 속도의 변화) 모니터링

이를 잘 이용하면 재미있는 기능을 만들어 낼 수 있다. 방향, 방위, 움직임을 모니터링하면 아래와 같은 일들이 가능해진다.

- 나침반과 가속도 센서를 이용해 여러분의 속도와 방향을 측정할 수 있다. 이를 지도, 카메라, 위치 기반 서비스와 함께 연동하면, 실시간 카메라 영상 위에 위치 기반 데이터를 표시하는 증강 현실augmented reality 인터페이스를 만들 수 있다.
- 기기의 방향에 알맞게 동적으로 조정되는 사용자 인터페이스를 만들 수 있다. 안드로이드는 세로 모드에서 가로 모드로 혹은 그 반대로 기기가 회전될 때, 네이티브 화면의 방향을 바꾸는 기능을 이미 적용하고 있다.
- 가속도를 빠른 속도로 모니터링해 기기가 떨어진 경우나 던져진 경우를 감지할 수 있다.
- 움직임이나 진동을 측정할 수 있다. 이를 테면 도난 방지 애플리케이션을 생각해볼 수 있는데, 기기가 잠겨 있는 동안 어떠한 움직임이 감지될 경우, 기기의 현재 위치를 SMS에 담아 보낼 수 있다.
- 물리적인 제스처와 움직임을 입력으로 이용하는 사용자 인터페이스 컨트롤을 만들 수 있다.

센서를 이용하는 애플리케이션을 만들 때는 필요한 센서들이 호스트 기기에서 이용 가능한지를 항상 확인해야 한다. 또한 이용할 수 없는 센서가 있을 경우, 적절히 대처하도록 구현해야 한다.

가속도 센서 소개

가속도 센서accelerometer는 그 이름에서도 알 수 있듯이 가속도를 측정하는 데 쓰인다. 중력 센서gravity sensor라 부르기도 한다.

가속도 센서는 중력 센서라고도 하는데, 이는 움직임에 기인한 가속도와 중력에 기인한 가속도 간의 차이를 구분하지 못하기 때문이다. 때문에 Z축(위/아래)에 대한 가속도를 감지하는 가속도 센서는 정지 상태에 있을 때 −9.8 m/s^2을 읽을 것이다(이 값은 상수 `SensorManager.STANDARD_GRAVITY`로 정의되어 있다).

가속도는 속도 변화의 비율로 정의되므로, 가속도 센서는 주어진 방향에서 기기의 속도가 얼마나 빨리 변화하고 있는지를 측정한다. 가속도 센서를 이용하면 움직임과 이 움직임의 속도 변화 비율을 감지할 수 있다. 대개 후자가 더 유용하다.

가속도 센서는 속도를 측정하지 않는다. 따라서 가속도 센서가 읽은 값 하나만 가지고는 속도를 계산해낼 수 없다. 속도를 측정하려면, 시간의 흐름에 따른 가속도의 변화를 측정해야 할 필요가 있다.

여러분은 대개 정지 상태에 상대적인 가속도 변화나 사용자 입력으로 사용되는 제스처같이 (가속도의 빠른 변화를 통해 감지되는) 빠른 움직임에 관심 있을 것이다. 전자의 경우, 상대적인 값의 변화를 측정하기 위해서는 기기를 조정해 기준 값으로 사용할 초기 방향과 가속도를 계산할 필요가 있다.

가속도 변화 감지하기

가속도는 좌−우(X축, Lateral), 앞−뒤(Y축, Longitudinal), 위−아래(Z축, Vertical) 이렇게 세 방향 축에 대해 측정할 수 있다. 센서 매니저는 이 세 방향 축 모두에 대한 가속도 센서의 변화를 보고한다.

이 값들은 센서 이벤트 리스너에 전달되는 `SensorEvent` 매개변수의 `values` 속성을 통해 X축, Y축, Z축 순으로 보고된다.

그림 14−1은 정지 상태의 기기 위에 세 방향 축을 매핑한 모습이다. 센서 매니저는 기기가 평면 위에서 정면이 하늘을 향한 채 세로 모드로 놓여 있을 때를 "정지 상태"로 간주한다.

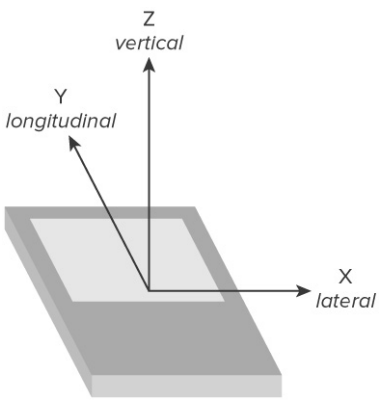

그림 14−1

- **X축(Lateral)** 왼쪽 또는 오른쪽 측면 가속도. 양의 값은 기기의 오른쪽 측면을 향한 움직임을, 음의 값은 왼쪽 측면을 향한 움직임을 나타낸다. 예를 들어, 정지 상태에 있는 기기를 오른쪽으로 이동시키면 양의 X축 가속도가 감지된다.
- **Y축(Longitudinal)** 전방 또는 후방 가속도. 전방 가속도가 양의 값이다. 정지 상태에 있는 기기를 기기의 머리 방향으로 이동시키면 양의 Y축 가속도가 생성된다.
- **Z축(Vertical)** 위쪽 또는 아래쪽 가속도. 정지 상태에 있는 기기를 위로 들어올리면 양의 Z축 가속도가 생성된다. 기기가 정지 상태에 있을 때, Z축 가속도 센서는 중력으로 인해 $-9.8 m/s^2$을 기록한다.

앞서 설명한 것처럼 센서 이벤트 리스너를 이용하면 가속도의 변화를 모니터할 수 있다. 가속도 센서 업데이트를 요청하려면, `Sensor.TYPE_ACCELEROMETER` 센서 타입을 가지고 `SensorEventListener` 구현을 센서 매니저에 등록한다. 코드 14-2는 기본 가속도 센서를 보통의 업데이트 속도로 등록한다.


Available for download on Wrox.com

코드 14-2 기본 가속도 센서에 생기는 변화 감지하기

```
SensorManager sm = (SensorManager)getSystemService(Context.SENSOR_SERVICE);
int sensorType = Sensor.TYPE_ACCELEROMETER;
sm.registerListener(mySensorEventListener,
                    sm.getDefaultSensor(sensorType),
                    SensorManager.SENSOR_DELAY_NORMAL);
```

이때, 센서 리스너는 가속도가 측정됐을 때 호출될 `onSensorChanged` 메서드를 구현하고 있어야 한다.

`onSensorChanged` 메서드는 `SensorEvent` 하나를 매개변수로 받는데, 여기에는 세 가지 축 모두에 대해 측정된 가속도를 담고 있는 float 배열이 들어 있다. 이 배열의 첫 번째 요소는 X축 가속도를, 두 번째 요소는 Y축 가속도를, 세 번째 요소는 Z축 가속도를 나타낸다. 아래의 코드를 참고하자.

```
final SensorEventListener mySensorEventListener = new SensorEventListener() {
    public void onSensorChanged(SensorEvent sensorEvent) {
        if (sensorEvent.sensor.getType() == Sensor.TYPE_ACCELEROMETER) {
            float xAxis_lateralA = sensorEvent.values[0];
            float yAxis_longitudinalA = sensorEvent.values[1];
            float zAxis_verticalA = sensorEvent.values[2];
```

```
            // TODO: 가속도의 변화를 애플리케이션에 적용한다.
        }
    }
};
```

중력 측정기 만들기

세 방향 모두에 대한 가속도를 합산한 뒤, 이를 자유 낙하 가속도와 비교하면 중력을 측정할 수 있다. 이번 예제에서는 현재 기기에 가해지고 있는 힘을 측정하기 위해 가속도계로 중력을 재는 간단한 애플리케이션을 만들어본다.

정지 상태의 기기에 가해지는 힘은 중력으로 인해 지구 중심 쪽으로 9.8m/s²이 된다. 이 예제에서는 이 값에 해당하는 SensorManager.STANDARD_GRAVITY 상수를 이용해 중력을 상쇄시킬 것이다.

1. 먼저 Forceometer라는 이름의 새로운 프로젝트를 만들고, 여기에 Forceometer 액티비티를 만들어 포함시킨다. 이어서, main.xml 레이아웃 리소스를 수정해 텍스트 뷰 두 개를 추가한 다음, 텍스트가 굵은 글꼴로 가운데 정렬하여 표시되도록 설정한다. 이 두 텍스트 뷰는 각각 현재 중력 값과 지금까지 측정된 중력 값 중 가장 큰 값을 표시하는 데 사용된다.

```
<?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">
    <TextView android:id="@+id/acceleration"
        android:gravity="center"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:textStyle="bold"
        android:textSize="32sp"
        android:text="CENTER"
        android:editable="false"
        android:singleLine="true"
        android:layout_margin="10px"/>
    />
    <TextView android:id="@+id/maxAcceleration"
        android:gravity="center"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
```

```
            android:textStyle="bold"
            android:textSize="40sp"
            android:text="CENTER"
            android:editable="false"
            android:singleLine="true"
            android:layout_margin="10px"/>
        />
    </LinearLayout>
```

2. Forceometer 액티비티 안에 이 두 TextView와 SensorManager의 레퍼런스를 저장하기 위한 인스턴스 변수들을 만든다.❷

```
SensorManager sensorManager;
TextView accelerationTextView;
TextView maxAccelerationTextView;
float currentAcceleration = 0;
float maxAcceleration = 0;
```

3. 새로운 SensorEventListener를 구현해 생성한다. 각 축에 대해 감지된 가속도를 모두 더한 뒤 중력 가속도를 빼도록 구현한다. 또한 감지된 가속도가 변할 때마다 현재 가속도와 최대 가속도를 업데이트해야 한다.❸

```
private final SensorEventListener sensorEventListener = new SensorEventListener()
{
    double calibration = SensorManager.STANDARD_GRAVITY;

    public void onAccuracyChanged(Sensor sensor, int accuracy) { }

    public void onSensorChanged(SensorEvent event) {
        double x = event.values[0];
        double y = event.values[1];
        double z = event.values[2];
```

❷ **옮긴이** 아래의 import 문도 추가한다.

```
import android.hardware.SensorManager;
import android.widget.TextView;
```

❸ **옮긴이** 아래의 import 문도 추가한다.

```
import android.hardware.Sensor;
import android.hardware.SensorEvent;
import android.hardware.SensorEventListener;
```

```
            double a = Math.round(Math.sqrt(Math.pow(x, 2) +
                                            Math.pow(y, 2) +
                                            Math.pow(z, 2)));
            currentAcceleration = Math.abs((float)(a-calibration));

            if (currentAcceleration > maxAcceleration)
                maxAcceleration = currentAcceleration;
        }
    };
```

4. onCreate 메서드를 수정해 SensorManager를 이용하여 위 리스너를 등록하도록 업데이트한다. 또한 중력 값을 표시하는 두 텍스트 뷰의 레퍼런스도 얻어온다.❹

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

        accelerationTextView = (TextView)findViewById(R.id.acceleration);
        maxAccelerationTextView = (TextView)findViewById(R.id.maxAcceleration);
        sensorManager = (SensorManager)getSystemService(Context.SENSOR_SERVICE);

        Sensor accelerometer = sensorManager.getDefaultSensor(Sensor.TYPE_ACCELEROMETER);
        sensorManager.registerListener(sensorEventListener,
                                       accelerometer,
                                       SensorManager.SENSOR_DELAY_FASTEST);
    }
```

5. 가속도 센서는 상당히 민감하기 때문에, 매번 가속도의 변화가 감지될 때마다 이를 텍스트 뷰에 업데이트하게 되면 부하가 많을 수 있다. 따라서 타이머에 기반해 텍스트 뷰를 업데이트하는 updateGUI라는 이름의 새로운 메서드를 만든다.

```
    private void updateGUI() {
        runOnUiThread(new Runnable() {
            public void run() {
                String currentG = currentAcceleration/SensorManager.STANDARD_GRAVITY
                                  + "Gs'';
                accelerationTextView.setText(currentG);
```

❹ **옮긴이** 아래의 import 문도 추가한다.

```
import android.content.Context;
```

```
            accelerationTextView.invalidate();
            String maxG = maxAcceleration/SensorManager.STANDARD_GRAVITY + "Gs'';
            maxAccelerationTextView.setText(maxG);
            maxAccelerationTextView.invalidate();
        }
    });
};
```

6. 마지막으로, 100ms마다 GUI가 업데이트되도록 onCreate 메서드에 타이머를 시작시키는 코드를 추가한다.[5]

```
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);
    accelerationTextView = (TextView)findViewById(R.id.acceleration);
    maxAccelerationTextView = (TextView)findViewById(R.id.maxAcceleration);
    sensorManager = (SensorManager)getSystemService(Context.SENSOR_SERVICE);
    Sensor accelerometer =
        sensorManager.getDefaultSensor(Sensor.TYPE_ACCELEROMETER);
    sensorManager.registerListener(sensorEventListener,
                                   accelerometer,
                                   SensorManager.SENSOR_DELAY_FASTEST);

    Timer updateTimer = new Timer("gForceUpdate");
    updateTimer.scheduleAtFixedRate(new TimerTask() {
        public void run() {
            updateGUI();
        }
    }, 0, 100);
}
```

이제 이 애플리케이션을 테스트해보고 싶을 것이다. 대서양 상공에서 고중력 기동 훈련을 하는 F16 전투기에 탑승해 테스트해볼 수 있다면 더할 나위 없이 좋겠지만, 그럴 수 없으므로 안전한 인근 지역에서 달리기나 운전을 통해 테스트해보자.

하지만 차를 몰거나, 자전거를 타거나, 비행기를 조종하면서 테스트를 위해 휴대폰을 계속

---

[5] **옮긴이** 아래의 import 문도 추가한다.

```
import java.util.Timer;
import java.util.TimerTask;
```

쳐다보면 위험하다. 위와 같은 상황에서 테스트할 수 있으려면 애플리케이션을 좀더 개량해야 한다.

진동 기능이나 미디어 플레이어 기능을 통합해 현재 중력에 비례하는 강도로 진동을 울리거나 소리를 내는 방법이 있을 수 있고, 아니면 간단히 중력의 변화를 기록으로 남기는 방법도 있을 수 있다.

방향 측정하기

방향 센서는 전자식 나침반 기능을 하는 자기장 센서와 피치pitch 및 롤roll을 측정하는 가속도 센서의 조합이다.

삼각법三角法에 대해 조금 알고 있는 독자들은 가속도 센서와 이 세 가지 축에 대한 자기장 값을 토대로 기기의 방향을 계산하는 데 필요한 기술을 이미 가지고 있는 것이다. 하지만 필자만큼이나 삼각법을 즐겼다면, 안드로이드가 이들을 대신 계산해준다는 사실에 기뻐할 것이다.

사실 안드로이드에서 기기의 방향을 측정하는 방법에는 두 가지가 있다. 첫 번째는 방향 센서에 직접 질의하는 방법이고, 두 번째는 가속도 센서와 자기장 센서를 이용해 방향을 유추해내는 방법이다. 두 번째 방법은 첫 번째 방법에 비해 느리지만 정확도가 높고, 방향 측정 시 기준 좌표계reference frame를 바꿀 수 있다는 이점이 있다. 이 두 기법에 대해서는 다음 절에서 설명한다.

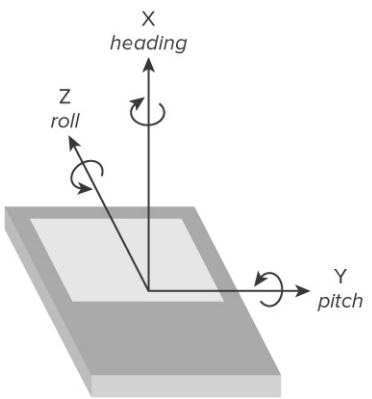

그림 14-2

기기의 방향은 그림 14-2에 설명된 것처럼 표준 기준 좌표계standard reference frame를 이용하여 세 방향 모두에 대해 보고된다.[6] 가속도 센서를 이용할 때와 마찬가지로, 기기는 평면 위에서 정면이 하늘을 향한 채 세로 모드로 놓여 있을 때 "정지 상태"로 간주된다.

⑤ 옮긴이 • 피치(Pitch): Y축을 중심으로 한 회전을 말하며, 고개를 끄덕이는 것과 같다.
• 롤(Roll): Z축을 중심으로 한 회전을 말하며, 고개를 좌우로 갸웃거리는 것과 같다.
• 요(Yaw): X축을 중심으로 한 회전을 말하며, 고개를 좌우로 흔드는 것과 같다.

- **X축(방위각Azimuth)** 방위각(헤딩heading 또는 요yaw라고도 한다)은 기기가 X축을 중심으로 마주하는 방향으로서, 0/360도는 북north, 90/360도는 동east, 180/360도는 남south, 270/360도는 서west다.
- **Y축(피치Pitch)** 피치는 Y축을 중심으로 한 기기의 각도다. 리턴된 경사각은 기기가 평면 위에 놓여 있을 때 0도를, 똑바로 서 있을 때(기기의 머리가 천장을 향하고 있을 때) −90도를, 거꾸로 서 있을 때 90도를, 뒤집어져 있을 때 180/−180도를 나타낸다.
- **Z축(롤Roll)** 롤은 Z축을 중심으로 한 기기의 측면 기울기로서 −90도와 90도 사이의 값을 가진다. 0도는 기기가 평면 위에 놓여 있을 때를, −90도는 화면이 왼쪽을 향하고 있을 때를, 90도는 화면이 오른쪽을 향하고 있을 때를 나타낸다.

방향 센서를 이용한 방향 감지

기기의 방향을 모니터하는 가장 쉬운 방법은 전용 방향 센서를 이용하는 것이다. 코드 14-3에서 보이는 것처럼 센서 이벤트 리스너를 생성한 뒤 이를 기본 방향 센서와 함께 센서 매니저에 등록한다.

코드 14-3 방향 센서를 이용한 방향 감지

```
SensorManager sm = (SensorManager)getSystemService(Context.SENSOR_SERVICE);
int sensorType = Sensor.TYPE_ORIENTATION;
sm.registerListener(myOrientationListener,
                    sm.getDefaultSensor(sensorType),
                    SensorManager.SENSOR_DELAY_NORMAL);
```

이제 기기의 방향에 변화가 생기면, 여러분의 `SensorEventListener` 구현에 있는 `onSensorChanged` 메서드가 호출된다. 이때, `SensorEvent` 매개변수의 `values`에는 세 축에 대한 기기의 방향 값을 담은 float 배열이 전달된다.

이 values 배열의 첫 번째 요소는 방위각(헤딩)을, 두 번째 요소는 피치를, 세 번째 요소는 롤을 나타낸다.

```
final SensorEventListener myOrientationListener = new SensorEventListener() {
    public void onSensorChanged(SensorEvent sensorEvent) {
        if (sensorEvent.sensor.getType() == Sensor.TYPE_ORIENTATION) {
            float headingAngle = sensorEvent.values[0];
            float pitchAngle = sensorEvent.values[1];
```

```
            float rollAngle = sensorEvent.values[2];

            // TODO: 방향 변화를 애플리케이션에 적용한다.
        }
    }

    public void onAccuracyChanged(Sensor sensor, int accuracy) {}
};
```

가속도 센서와 자기장 센서를 이용한 방향 계산

기기의 방향을 찾는 가장 좋은 방법은 가속도 센서와 자기장 센서의 결과를 이용해 방향을 직접 계산해내는 것이다.

이 기법은 기기를 사용하는 중에 방향 기준 좌표계orientation reference frame를 바꿔 여러분이 생각하는 기기의 방향에 맞게 X, Y, Z축을 다시 매핑할 수 있게 해준다.

이 기법은 가속도 센서와 자기장 센서를 같이 이용하기 때문에 센서 이벤트 리스너도 두 개를 만들어 등록해야 한다. 각 센서 이벤트 리스너의 onSensorChanged 메서드에서는 전달받은 values 배열 속성을 별도의 필드 변수에 저장한다. 코드 14-4를 참고하자.

코드 14-4 가속도 센서와 자기장 센서를 이용한 방향 계산

```
float[] accelerometerValues;
float[] magneticFieldValues;

final SensorEventListener myAccelerometerListener = new SensorEventListener() {
    public void onSensorChanged(SensorEvent sensorEvent) {
        if (sensorEvent.sensor.getType() == Sensor.TYPE_ACCELEROMETER)
            accelerometerValues = sensorEvent.values;
    }

    public void onAccuracyChanged(Sensor sensor, int accuracy) {}
};

final SensorEventListener myMagneticFieldListener = new SensorEventListener() {
    public void onSensorChanged(SensorEvent sensorEvent) {
        if (sensorEvent.sensor.getType() == Sensor.TYPE_MAGNETIC_FIELD)
            magneticFieldValues = sensorEvent.values;
    }

    public void onAccuracyChanged(Sensor sensor, int accuracy) {}
};
```

이제 아래의 코드에서 보이는 것처럼 앞의 센서 이벤트 리스너들을 센서 매니저에 등록한다. 아래의 코드는 두 센서 모두에 대해 기본 하드웨어와 UI 업데이트 속도를 사용한다.

```
SensorManager sm = (SensorManager)getSystemService(Context.SENSOR_SERVICE);
Sensor aSensor = sm.getDefaultSensor(Sensor.TYPE_ACCELEROMETER);
Sensor mfSensor = sm.getDefaultSensor(Sensor.TYPE_MAGNETIC_FIELD);

sm.registerListener(myAccelerometerListener,
                    aSensor,
                    SensorManager.SENSOR_DELAY_UI);

sm.registerListener(myMagneticFieldListener,
                    mfSensor,
                    SensorManager.SENSOR_DELAY_UI);
```

이 센서 값들을 가지고 현재 방향을 계산하려면, 아래와 같이 센서 매니저의 getRotation Matrix 메서드와 getOrientation 메서드를 이용한다. getOrientation이 리턴하는 값은 도degree 단위가 아니라 라디안radian 단위이므로 주의하자.

```
float[] values = new float[3];
float[] R = new float[9];
SensorManager.getRotationMatrix(R, null,
                                accelerometerValues,
                                magneticFieldValues);
SensorManager.getOrientation(R, values);

// 라디안 단위로 된 값을 도 단위로 변환한다.
values[0] = (float) Math.toDegrees(values[0]);
values[1] = (float) Math.toDegrees(values[1]);
values[2] = (float) Math.toDegrees(values[2]);
```

방향 기준 좌표계 재설정하기

센서 매니저의 remapCoordinateSystem 메서드를 이용하면, 앞서 설명한 표준 기준 좌표계가 아닌 다른 기준 좌표계를 이용해 기기의 방향을 측정할 수 있다.

이번 장 앞에서는 "기기가 평면 위에서 정면이 하늘을 향해 놓여 있는 상태"를 표준 기준 좌표계라고 설명했다. remapCoordinateSystem 메서드는 여러분의 방향을 계산하는 데 사용되는 좌표계를 재설정할 수 있게 해준다. 이를 테면, 기기가 수직으로 마운트된 상태를 정지 상태로 설정할 수 있다.

remapCoordinateSystem 메서드는 네 개의 매개변수를 받는다.

- getRotationMatrix로 얻은 초기 회전 행렬rotation matrix
- 재설정 된 X축
- 재설정 된 Y축
- 변환된 결과 회전 행렬을 저장하기 위한 변수

중간의 두 매개변수는 새로운 기준 좌표계를 지정하는 데 쓰인다. 새로운 X축과 Y축은 기본 좌표계에 상대적으로 지정한다. 센서 매니저는 이렇게 축을 지정할 때 사용할 수 있는 일련의 상수들을 제공한다(AXIS_X, AXIS_Y, AXIS_Z, AXIS_MINUS_X, AXIS_MINUS_Y, AXIS_MINUS_Z).

코드 14-5는 기기가 그림 14-3처럼 수직으로 마운트됐을 때를 정지 상태로 만들도록 기준 좌표계를 재설정하는 법을 보여준다.

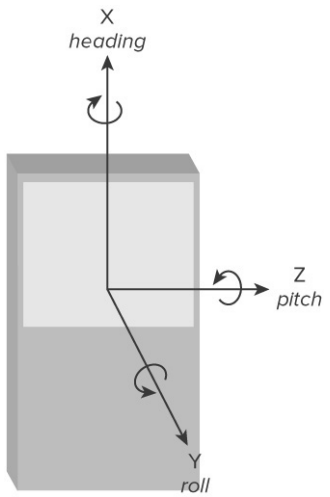

그림 14-3

코드 14-5 방향 기준 좌표계 재설정하기

```
SensorManager.getRotationMatrix(R, null, aValues, mValues);

float[] outR = new float[9];
SensorManager.remapCoordinateSystem(R,
                                    SensorManager.AXIS_X,
                                    SensorManager.AXIS_Z,
                                    outR);
SensorManager.getOrientation(outR, values);

// 라디안 단위로 된 값을 도 단위로 변환한다.
values[0] = (float) Math.toDegrees(values[0]);
values[1] = (float) Math.toDegrees(values[1]);
values[2] = (float) Math.toDegrees(values[2]);
```

나침반과 인공 수평의水平儀 만들기

4장에서는 오너 드로우 컨트롤을 실험해보기 위해 간단한 CompassView를 만들었다. 이번

예제에서는 이 나침반 뷰의 기능을 확장해 기기의 피치와 롤을 표시하고, 이를 이용해 기기의 방향을 표시한다.

1. 4장에서 만들었던 Compass 프로젝트를 연다. 여러분은 CompassView뿐만 아니라 이 뷰를 표시하는 데 사용되는 Compass 액티비티도 함께 변경할 것이다. 뷰와 컨트롤러를 가능한 한 분리된 채로 유지하기 위해 CompassView는 센서에 직접 연결되지 않으며, 대신 액티비티를 통해 업데이트될 것이다. 먼저 CompassView에 피치와 롤을 위한 필드 변수들과 get/set 메서드들을 추가한다.

```
float pitch = 0;
float roll = 0;

public float getPitch() {
    return pitch;
}

public void setPitch(float pitch) {
    this.pitch = pitch;
}

public float getRoll() {
    return roll;
}

public void setRoll(float roll) {
    this.roll = roll;
}
```

2. onDraw 메서드를 업데이트해 피치와 롤을 표시하기 위한 두 개의 원을 포함시킨다.

```
@Override
protected void onDraw(Canvas canvas) {

    [ ... 기존 onDraw 메서드 ... ]
```

2.1. 내부의 반이 채워진 새로운 원 하나를 만들고, 이를 측면 기울기(롤)만큼 회전시킨다.

```
int mMeasuredWidth = getMeasuredWidth();
int mMeasuredHeight = getMeasuredWidth();

RectF rollOval = new RectF((mMeasuredWidth/3)-mMeasuredWidth/7,
                           (mMeasuredHeight/2)-mMeasuredWidth/7,
```

```
                                (mMeasuredWidth/3)+mMeasuredWidth/7,
                                (mMeasuredHeight/2)+mMeasuredWidth/7
                               );
        markerPaint.setStyle(Paint.Style.STROKE);
        canvas.drawOval(rollOval, markerPaint);
        markerPaint.setStyle(Paint.Style.FILL);
        canvas.save();
        canvas.rotate(roll, mMeasuredWidth/3, mMeasuredHeight/2);
        canvas.drawArc(rollOval, 0, 180, false, markerPaint);

        canvas.restore();
```

2.2. 내부의 반이 채워진 새로운 원 하나를 만들고, 전방 각도forward angle(피치)에 따라 원 내부의 채워지는 정도를 달리한다.

```
        RectF pitchOval = new RectF((2*mMeasuredWidth/3)-mMeasuredWidth/7,
                                (mMeasuredHeight/2)-mMeasuredWidth/7,
                                (2*mMeasuredWidth/3)+mMeasuredWidth/7,
                                (mMeasuredHeight/2)+mMeasuredWidth/7
                               );
        markerPaint.setStyle(Paint.Style.STROKE);
        canvas.drawOval(pitchOval, markerPaint);
        markerPaint.setStyle(Paint.Style.FILL);
        canvas.drawArc(pitchOval, 0-pitch/2, 180+(pitch), false, markerPaint);
        markerPaint.setStyle(Paint.Style.STROKE);
    }
```

3. 이것으로 CompassView 수정을 모두 마쳤다. 지금 이 애플리케이션을 실행하면, 그림 14–4에 보이는 것처럼 나타나야 한다.

4. 이제 Compass 액티비티를 업데이트하자. 센서 매니저를 이용하여 자기장 센서와 가속도 센서를 통한 방향 변화에 귀 기울여야 한다. 먼저 자기장 값과 가속도 값, 그리고 CompassView와 SensorManager의 레퍼런스를 저장하기 위한 지역 필드 변수들을 추가하는 것으로 시작한다.[7]

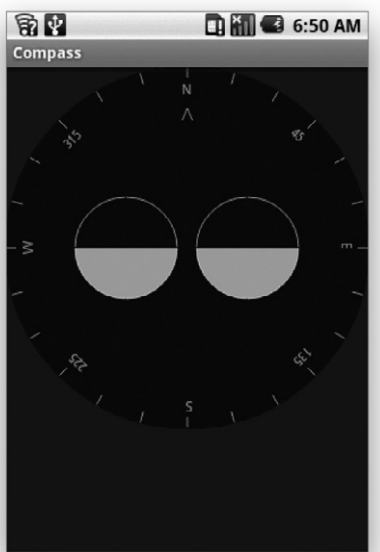

그림 14–4

[7] **옮긴이** 아래의 import 문도 추가한다.

```
import android.hardware.SensorManager;
```

```
float[] aValues = new float[3];
float[] mValues = new float[3];
CompassView compassView;
SensorManager sensorManager;
```

5. 새로운 헤딩, 피치, 롤 값을 이용해 CompassView를 업데이트하는 새로운 updateOrientation 메서드를 만든다.

```
private void updateOrientation(float[] values) {
    if (compassView!= null) {
        compassView.setBearing(values[0]);
        compassView.setPitch(values[1]);
        compassView.setRoll(-values[2]);
        compassView.invalidate();
    }
}
```

6. onCreate 메서드를 수정해 CompassView와 SensorManager의 레퍼런스를 얻어온 뒤, 헤딩, 피치, 롤 값을 초기화하도록 업데이트한다.[8]

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

    compassView = (CompassView)this.findViewById(R.id.compassView);
    sensorManager = (SensorManager)getSystemService(Context.SENSOR_SERVICE);
    updateOrientation(new float[] {0, 0, 0});
}
```

7. 가장 최근에 기록된 가속도 값과 자기장 값을 가지고 기기의 방향을 평가하는 새로운 calculateOrientation 메서드를 만든다.

```
private float[] calculateOrientation() {
    float[] values = new float[3];
    float[] R = new float[9];

    SensorManager.getRotationMatrix(R, null, aValues, mValues);
```

[8] **옮긴이** 아래의 import 문도 추가한다.

```
import android.content.Context;
```

```
        SensorManager.getOrientation(R, values);

        // 라디안 단위로 된 값을 도 단위로 변환한다.
        values[0] = (float) Math.toDegrees(values[0]);
        values[1] = (float) Math.toDegrees(values[1]);
        values[2] = (float) Math.toDegrees(values[2]);

        return values;
    }
```

8. SensorEventListener를 구현한다. onSensorChanged는 호출 센서의 타입을 확인해 가속도 값이나 자기장 값을 적절히 업데이트하고, calculateOrientation 메서드를 이용해 updateOrientation 메서드를 호출해야 한다.[9]

```
    private final SensorEventListener sensorEventListener = new SensorEventListener() {

        public void onSensorChanged(SensorEvent event) {
            if (event.sensor.getType() == Sensor.TYPE_ACCELEROMETER)
                aValues = event.values;
            if (event.sensor.getType() == Sensor.TYPE_MAGNETIC_FIELD)
                mValues = event.values;

            updateOrientation(calculateOrientation());
        }

        public void onAccuracyChanged(Sensor sensor, int accuracy) {}
    };
```

9. 이제 onResume과 onStop을 재정의한다. 그리고 액티비티가 화면에 보여질 때와 안 보여질 때, 각각 SensorEventListener를 등록하고 등록 해제하도록 구현한다.

```
    @Override
    protected void onResume() {
        super.onResume();

        Sensor accelerometer = sensorManager.getDefaultSensor(Sensor.TYPE_ACCELEROMETER);
        Sensor magField = sensorManager.getDefaultSensor(Sensor.TYPE_MAGNETIC_FIELD);
```

❾ **옮긴이** 아래의 import 문도 추가한다.

```
import android.hardware.Sensor;
import android.hardware.SensorEvent;
import android.hardware.SensorEventListener;
```

```
        sensorManager.registerListener(sensorEventListener,
                                       accelerometer,
                                       SensorManager.SENSOR_DELAY_FASTEST);
        sensorManager.registerListener(sensorEventListener,
                                       magField,
                                       SensorManager.SENSOR_DELAY_FASTEST);
    }

    @Override
    protected void onStop() {
        sensorManager.unregisterListener(sensorEventListener);
        super.onStop();
    }
```

이제 이 애플리케이션을 실행하면, 기기의 방향이 변할 때 동적으로 업데이트되는 세 개의 다이얼을 볼 수 있을 것이다.

10. 인공 수평의는 수직으로 마운트했을 때 더 유용하다. calculateOrientation을 업데이트하여 이 인공 수평의의 기준 좌표계를 수직으로 마운트했을 때의 방향에 맞게 재설정하자.

```
    private float[] calculateOrientation() {
        float[] values = new float[3];
        float[] R = new float[9];
        float[] outR = new float[9];

        SensorManager.getRotationMatrix(R, null, aValues, mValues);
        SensorManager.remapCoordinateSystem(R,
                                            SensorManager.AXIS_X,
                                            SensorManager.AXIS_Z,
                                            outR);

        SensorManager.getOrientation(outR, values);

        // 라디안 단위로 된 값을 도 단위로 변환한다.
        values[0] = (float) Math.toDegrees(values[0]);
        values[1] = (float) Math.toDegrees(values[1]);
        values[2] = (float) Math.toDegrees(values[2]);

        return values;
    }
```

기기 진동 제어하기

9장에서는 알림Notifications을 만드는 법을 배웠다. 알림은 진동을 이용해 좀더 풍부한 이벤트 피드백을 제공할 수 있다. 경우에 따라서는 알림과 별개로 기기를 진동시키고자 할 때도 있다. 진동은 사용자의 촉각을 통해 피드백을 전달하는 탁월한 방법으로서, 특히 게임을 위한 피드백 메커니즘으로 많이 쓰인다.

기기 진동을 제어하려는 애플리케이션은 `VIBRATE` 권한이 필요하다. 아래의 XML 코드를 이용해 애플리케이션 매니페스트에 `VIBRATE` 권한을 추가하자.

```
<uses-permission android:name="android.permission.VIBRATE"/>
```

기기 진동은 `Vibrator` 서비스를 통해 제어한다. 이 서비스는 코드 14–6에서 보이는 것처럼 `getSystemService` 메서드를 이용해 접근할 수 있다.

코드 14–6 기기 진동 제어하기

```
String vibratorService = Context.VIBRATOR_SERVICE;
Vibrator vibrator = (Vibrator)getSystemService(vibratorService);
```

기기를 진동시키려면 `vibrate`를 호출한다. `vibrate`에는 진동 지속 시간을 전달할 수도 있고, "진동"과 "멈춤"으로 구성된 시퀀스 패턴을 전달할 수도 있다. 후자의 경우에는 인덱스 매개변수를 함께 전달하는데, 이렇게 하면 지정된 인덱스에서부터 패턴을 반복하게 된다. 이 두 기법이 아래의 코드에 설명되어 있다.

```
long[] pattern = {1000, 2000, 4000, 8000, 16000 };
vibrator.vibrate(pattern, 0); // 진동 패턴을 실행한다.
vibrator.vibrate(1000); // 1초간 진동한다.
```

진동을 멈추려면 `cancel`을 호출한다. 애플리케이션을 종료하면 모든 진동이 자동으로 취소된다.

요약

이번 장에서는 애플리케이션이 물리적인 환경에 반응할 수 있도록 센서 매니저를 이용하는 법에 대해 배웠다. 안드로이드 플랫폼에서 사용 가능한 센서들에 대해 살펴봤고, 센서 이벤트 리스너를 이용해 센서 이벤트에 귀 기울이는 법과 결과 값을 해석하는 법을 배웠다.

이어서 가속도 센서, 방향 센서, 자기장 센서에 대해 자세히 살펴봤고, 이들 센서를 이용해 기기의 방향과 가속도를 측정했다. 이 과정에서 중력 측정기와 인공 수평의를 만들었다.

또한 아래와 같은 것들을 배웠다.

- 안드로이드 애플리케이션에서 사용 가능한 센서들의 종류
- 기기의 방향을 측정할 때 기준 좌표계를 재설정하는 법
- 각 센서에 의해 리턴되는 센서 이벤트 값의 조합과 의미
- 기기 진동을 이용해 애플리케이션 이벤트를 물리적인 피드백으로 제공하는 법

마지막 장에서는 안드로이드의 고급 기능 몇 가지를 살펴본다. 보안에 대해 자세히 배우고, AIDL을 이용해 프로세스 간 통신을 용이하게 하는 법과 웨이크 락Wake Locks의 이용법에 대해 자세히 배운다. 안드로이드의 TTS 라이브러리를 소개하며, 애니메이션과 고급 캔버스 드로잉 기법을 살펴봄으로써 안드로이드의 사용자 인터페이스와 그래픽스 기능에 대해 배운다. 마지막으로, 서피스 뷰와 터치스크린 입력 기능에 대해서도 살펴본다.