

CHAPTER 1

안드로이드 게임 개발을 시작하며

안드로이드 게임 개발의 세계로 잘 오셨습니다! 이 책의 목표는 안드로이드 플랫폼을 위한 본격적인 게임을 만들려는 개발자들을 돕는 것이다. 이 책에서 다루는 게임은 크게 두 종류이다. 하나는 순수 Java로만 작성된 게임이고, 또 하나는 Java의 우아한 설계와 C의 강력한 성능을 결합한 혼성 게임(hybrid game)이다. 아마도 관심이 더 가는 쪽은 후자일 것이다. Java와 C를 결합해서 게임을 만드는 방식은 아직 생소한 분야이고 Google의 지원도 그리 좋지 않다. 그런데 게임 개발에 필요한 Java API들이 충분히 갖추어져 있는 상황에서 굳이 그런 혼성 방식을 사용해야 할까? 그러나 게임 장르에 따라서는 Java만으로는 충분한 성능을 얻지 못할 수 있다. 또한 C로 작성된 많은 게임들의 C 소스 코드를 안드로이드 플랫폼에 맞게 컴파일하고 JNI(Java Native Interface)를 이용해 Java의 GUI로 감싸서 안드로이드용 고성능 게임을 만들어 내는 것이 가능하다. 실제로 이 책 후반부에서는 3D 슈팅 게임 역사의 두 걸작인 *Wolfenstein 3D*와 *Doom*(둘 다 PC용)을 안드로이드 플랫폼으로 이식하는 방법을 설명한다.

정리하자면, 이 책의 목표는 여러분에게 안드로이드용 게임 개발을 위한 문서화된, 그리고 문서화되지 않은 비법들을 전달하는 것이다. 더 나아가서, PC용 게임을 이식하고자 하는 독자라면 다른 곳에서는 찾기 힘든 귀중한 정보도 이 책에서 얻을 수 있을 것이다. 실질적인 게임 개발로 들어가기 전에, 여러분이 이 책을 최대한 활용하려면 무엇이 필요한지부터 짚고 넘어가자.

필요한 능력

이 책은 Java는 물론 C에도 익숙한 게임 개발자를 대상으로 한다. C 경험이 필요한 이유는 게임에서 성능이 워낙 중요하기 때문이다. 본격적인 게임에 필요한 성능을 얻으려면 C를 사용할 수밖에 없다. C의 강력함을 Java의 우아한 객체지향 기능들과 결합하는 개발 방식이 최상의 조합일 것이다. 이 책은 또한 독자가 안드로이드와 리눅스 및 셸 스크립팅에도 익숙하다고 가정한다.

안드로이드 개발의 기초 지식

이 책은 여러분이 안드로이드 개발의 기초를 이미 알고 있다고 가정한다. 예를 들어 활동(activity), 뷰, 레이아웃 등이 무엇인지 알아야 한다. 다음 코드 조각을 보자. 이 코드의 의미를 즉시 이해하기 힘들다면 사전 준비가 더 필요한 것이다.

```
public class MainActivity extends Activity
{
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
    }
}
```

이 코드는 응용프로그램의 주 활동을 정의한다. 이미 알고 있겠지만, 활동은 안드로이드 응용프로그램의 수명 주기를 구성하는 클래스이다. onCreate() 메서드는 응용프로그램이 시작될 때 한 번 호출된다. 위의 예에 나온 onCreate()는 응용프로그램의 내용 레이아웃(전반적인 GUI)을 설정한다.

또한 GUI를 XML로 정의하는 방법도 어느 정도는 알고 있어야 한다. 다음 XML이 무슨 의미인지 파악이 되는가?

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent">

    <ImageView android:id="@+id/doom_iv"
```

```
        android:layout_width="fill_parent"
        android:layout_height="fill_parent"
        android:background="@drawable/doom"
        android:focusableInTouchMode="true" android:focusable="true"/>

<ImageButton android:id="@+id/btn_upleft"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_alignParentBottom="true"
    android:layout_alignParentLeft="true"
    android:src="@drawable/img1" />
</RelativeLayout>
```

이 코드는 상대적 레이아웃을 위한 `RelativeLayout`을 정의한다. 상대적 레이아웃에서는 위젯의 위치와 크기가 다른 위젯에 상대적으로 결정된다. 위의 예는 화면 전체를 채우는 이미지 뷰 위젯 하나와 화면 하단에 배치되는 버튼 하나로 구성된 레이아웃이다. 이미지 뷰는 프로젝트의 `res/drawable` 디렉터리에 있는 `doom.png`이라는 파일에 담긴 배경 이미지를 표시하며, 키 입력과 터치 사건들도 받아들인다. 버튼은 화면 하단에, 이미지 위에 놓인다. 버튼 자체도 이미지(`res/drawable` 디렉터리에 있는 `img1.png`)를 표시한다.

안드로이드 개발 참고자료

안드로이드 개발에 관련된 개념들과 용어들은 아주 많다. 안드로이드 개발의 전반적인 개요와 주요 개념들, 관례와 조언들을 안드로이드 개발자 가이드에서 얻을 수 있다.

<http://developer.android.com/guide/index.html>

안드로이드의 수많은 패키지들과 클래스들의 세부사항들을 모두 외우고 다니는 것은 불가능한 일이므로, 레퍼런스 문서화 역시 자주 찾아보게 될 것이다. 주소는 다음과 같다. 중요한 자료이다.

<http://developer.android.com/reference/packages.html>

마지막으로, 다음 장소에서는 최신 SDK(그리고 NDK) 릴리스에 대한 정보와 다운로드 등을 제공한다.

<http://developer.android.com/sdk/index.html>

이 책 전반에서(특히 네이티브 코드를 다루는 장들에서) 안드로이드 SDK의 명령줄 도구들을 자주 사용한다(시스템 관리 작업 등을 위해). 따라서 그런 도구들, 특히 adb(Android Debug Bridge)에 대한 확고한 지식이 필요하다. 다음과 같은 일들을 능숙하게 수행할 수 있어야 한다.

- AVD(Android Virtual Device, 안드로이드 가상 기기) 만들기. AVD는 에뮬레이터를 위한 구체적인 기기 설정들(펌웨어 버전이나 SD 카드 경로 등)을 캡슐화한 파일이다. AVD 만드는 것은 아주 간단하다. 명령줄에서 만들어도 되고, GUI 방식의 AVD Manager(Eclipse 툴바의 까만 휴대폰 아이콘을 클릭하면 뜬다)로 만들어도 된다.
- SD 카드 파일 만들기. 이 책 후반부의 일부 게임들은 파일의 덩치가 크다(5MB 이상). 저장 공간을 절약하기 위해 게임은 모든 게임 파일을 기기의 SD 카드에 저장한다. 따라서 에뮬레이터에서 게임을 실행해 보려면 SD 카드 파일을 만들어야 한다. 다음은 홈 디렉터리에 `sdcard.iso`라는 100MB짜리 SD 카드 파일을 만드는 예이다.

```
$ mksdcard 100M $HOME/sdcard.iso
```

- 기기³⁾에 연결하기. 라이브러리 파일 추출 등 여러 시스템 관리 작업을 위해서는 기기에 연결해야 한다. 다음 명령을 수행하면 기기 내부에 접근할 수 있는 셸이 열린다.

```
$ adb shell
```

- 기기와 파일 주고받기. 다음 명령들로 기기 안에 있는 파일을 데스크톱으로 추출하거나 데스크톱의 파일을 기기에 집어넣을 수 있다.

```
$ adb push <지역 파일> <기기 경로>
```

```
$ adb pull <기기 파일> <지역 경로>
```

참고 이 명령들을 실행하기 전에 `SDK_HOME/tools` 디렉터리를 시스템의 `PATH` 변수에 추가하기 바란다.

3) [역주] 이 책 전반에서 기기(device)라는 용어는 실제 안드로이드 기기는 물론 에뮬레이터도 의미한다. 실제(물리적인) 안드로이드 기기는 ‘실제 기기’ 또는 ‘단말기’로 지칭한다.

리눅스와 셸 스크립팅 기초 지식

혼성 게임을 다루는 장들에서는 우분투 리눅스에서 작업을 진행하므로 전통적인 리눅스/유닉스 도구들과 작업 환경에 익숙해질 필요가 있다.

특히 파일 목록 보기 등 기본적인 셸 명령들의 사용과 소프트웨어 구성요소 설치, 그리고 기본적인 시스템 관리 작업에 어려움이 없어야 한다.

이 책에는 아주 간단한 셸 스크립트가 몇 가지 나온다. bash 셸은 꼭 이 책 때문이 아니더라도 요긴할 때가 많으니 기본적인나마 알아두면 좋을 것이다.

팁 웹을 찾아보면 리눅스와 셸 스크립팅에 대한 좋은 자료가 많이 있다. 다음은 그 중 하나이다.⁴⁾

<http://supportweb.cs.bham.ac.uk/documentation/tutorials/docsystem/build/tutorials/unixscripting/unixscripting.html>

필요한 소프트웨어 도구들

이번 장의 나머지 부분에서는 혼성(C/Java) 게임의 빌드(컴파일과 링크) 환경을 설정하는 방법에 대해 설명한다. 또한 기본적인 안드로이드 응용프로그램의 개발에 필요한 IDE(Eclipse)와 안드로이드 SDK의 설치 및 설정도 이야기한다. 이 책을 통해서 Java의 우아한 객체지향적 기능들과 최상의 성능을 위한 C의 위력을 결합하는 방법을 배우고자 한다면, 그리고 이후 장들에서 Doom과 Wolfenstein 3D를 직접 빌드해 보고 싶다면 지금부터의 내용이 아주 중요하다.

우선, 독자의 데스크톱 컴퓨터에 다음과 같은 소프트웨어가 설치되어 있어야 한다.

- VMware Player 또는 Workstation: 혼성 게임 개발 작업을 진행할 리눅스 가상 기계(VM)를 돌리기 위해 필요하다.⁵⁾ VMware Player는 <http://www.vmware.com/products/player/>에서 무료로 내려 받을 수 있다.

4) [역주] 한글로 된 문서들도 꽤 있다. 이를테면: <http://wiki.kldp.org/wiki.php/BashProgIntroHowto>

5) [역주] 꼭 VMware일 필요는 없다. 예를 들어 VirtualBox(<http://www.virtualbox.org/>)를 사용할 수도 있다. VirtualBox 역시 무료이다. 물론 리눅스를 실제로 설치해서 사용해도 된다. 한 가지 조언하자면, 이후 과정에서 안드로이드 소스를 비롯해서 덩치 큰 여러 도구들과 SDK들을 설치하게 되므로 (가상)하드디스크 용량을 넉넉히 잡아야 한다는 것이다. 적어도 15GB 이상이 필요할 것이다.

- **우분투 리눅스 VMware 어플라이언스:** VMware로 돌릴 우분투(리눅스 배포판의 하나)의 실제 이미지이다. VMware Virtual Appliance Marketplace(<http://www.vmware.com/appliances/>)에서 내려 받으면 된다. 크기가 상당히 크다는 점을 주의할 것(600MB 이상).
- **Eclipse:** 안드로이드 응용프로그램의 표준적인 통합 개발 환경 IDE이다. 버전 3.4(Ganymede)나 3.5(Galileo)를 설치하면 된다.
- **안드로이드 SDK:** 이 글을 쓰는 현재, SDK 최신 버전은 1.6이다.⁶⁾ SDK에는 안드로이드 응용프로그램 개발에 필요한 라이브러리와 여러 도구들(이클립스 기기에 연결하기 위한 adb 등)이 포함되어 있다. <http://developer.android.com/sdk/>에서 설치 패키지를 다운받아서 적당한 디렉터리에 풀고 SDK Readme.txt를 참고해서 설치를 진행하기 바란다.⁷⁾ 설치를 마친 후에는 안드로이드 SDK의 명령줄 도구들이 있는 디렉터를 시스템의 PATH에 추가해야 한다(이클립스 홈 디렉터리의 .bashrc에서 PATH 환경 변수를 설정하는 줄을 PATH=[SDK_경로]/tools:\$PATH로 고치는 등). 시스템 경로를 제대로 설정했다면 명령줄에서 에뮬레이터를 띄우거나 기타 안드로이드 도구들을 실행할 수 있다. 시험 삼아 터미널을 열고 adb를 입력해보기 바란다. 도움말이 주르륵 나타나면 제대로 된 것이다.
- **Java JDK 5.0 이상:** Eclipse와 안드로이드 SDK 자체를 실행하는 데 필요하다.

이제 우분투로 들어가서 나머지 필수 소프트웨어들을 설치해 보자.

개발용 리눅스 컴퓨터 준비

혼성 안드로이드 게임을 빌드할 수 있으려면 리눅스 데스크톱에 다음과 같은 세 가지 소프트웨어 구성요소들을 갖추어야 한다.

- **안드로이드 소스:** 여기에는 안드로이드 소스 코드 전체와 커스텀 공유 라이브러리의 빌드에 필요한 C/C++/JNI 헤더 파일들이 들어 있다.
- **안드로이드 네이티브 라이브러리들:** C 런타임, 수학, XML, 사운드 등 안드로이드 네이티브 응용프로그램의 작동에 필요한 라이브러리들.

6) [역주] 2010년 4월 현재 최신 버전은 2.1이다.

7) [역주] 1.6 이상이 이미 설치되어 있는 독자라면 Android SDK and AVD Manager(명령줄에서 인자 없이 android를 실행하거나, Eclipse 안에서 안드로이드 SDK 관리자 아이콘을 클릭하면 나타난다)를 통해서 새 버전들을 설치할 수 있다.

- ARM 프로세서용 GNU C/C++ 도구사슬: 네이티브 라이브러리의 빌드에 필요한 C/C++ 컴파일러 및 링커를 제공하며, 디버거나 프로파일러 같은 디버깅에 도움이 되는 여러 도구들도 제공한다.

그 외에, 혼성 게임의 컴파일 및 링크 과정을 좀 더 수월하게 진행하기 위한 커스텀 셸 스크립트들도 만들어 볼 것이다.

안드로이드 소스 얻기

Google은 Gittool이라고 하는 소프트웨어 버전 관리 도구(<http://git-scm.com/>에서 내려 받을 수 있다)를 이용해서 안드로이드 소스를 저장, 관리한다. 우분투 버전에 따라서는 몇 가지 필수 패키지들을 더 설치해야 할 수 있다. 콘솔(터미널)을 열고 다음 명령을 실행하면 필요한 패키지들이 모두 설치된다⁸⁾.

```
$ sudo apt-get install git-core gnupg \
sun-java6-jdk flex bison gperf \
libsdl-dev libesd0-dev libwxgtk2.6-dev \
build-essential zip curl libncurses5-dev zlib1g-dev
```

시스템 관리자(sysadmin) 계정으로 로그인했다면 `sudo`는 생략해도 된다.

팁 안드로이드 소스 빌드 환경의 설정에 대한 좀 더 자세한 내용은 Android Open Source Project(<http://source.android.com/download>)에 나와 있다.

8) [역주] 원서에는 `sun-java6-jdk`가 아니라 `sun-java5-jdk`로 되어 있으나, JDK 5는 안드로이드 소스 자체를 컴파일해서 안드로이드 OS 펌웨어를 만들려고 할 때에만 필요하다(사실 이 부분은 안드로이드 개발자 사이트의 안드로이드 소스 컴파일 관련 문서에서 그대로 가져온 듯하다). 이 책의 예제들을 위해서는 JDK 6을 사용해야 한다(JDK 5로 컴파일하면 오류가 발생한다).

혹시 펌웨어 빌드를 위해 JDK5를 설치하려 한다면, 우분투 버전에 따라서는 패키지 관리자가 `sun-java5-jdk` 패키지를 찾지 못할 수 있음을 주의할 것. 최근 버전의 우분투에서 `sun-java5-jdk`를 설치하려면 패키지 관리자에 추가적인 원본 저장소를 등록해야 하는데, 자세한 내용은 <http://ekwang.tistory.com/26>을 보기 바란다.

다음으로, Git를 좀 더 쉽게 다루기 위한 `repo`라는 도구(Google이 제공함)를 설치, 설정해야 한다. 홈 디렉터리에 `bin`이라는 폴더를 만들고 거기에 `repo`를 내려 받는다.

```
$ cd ~
$ mkdir bin
$ curl http://android.git.kernel.org/repo >~/bin/repo
$ chmod a+x ~/bin/repo
```

다음 명령을 이용해서 홈 디렉터리의 `bin` 폴더를 시스템 경로에 추가한다.

```
$ export PATH=$HOME/bin:$PATH
```

이 명령을 `$HOME/.bashrc`에도 추가해 두면 시스템에 로그인할 때 자동으로 적용되니 편하다.

다음으로는 소스를 저장할 `mydroid`라는 폴더를 홈 디렉터리에 만들고 그 폴더로 들어간다.

```
$ mkdir mydroid
$ cd mydroid
```

이제 안드로이드 소스 코드를 내려 받는다.

```
$ repo init -u git://android.git.kernel.org/platform/manifest.git
$ repo sync
```

팁 위의 `repo init` 명령은 안드로이드 소스의 주 가지(main branch)를 내려 받도록 한 것이다. 다른 가지를 내려 받고 싶다면 다음과 같은 형태로 명령을 내리면 된다.

```
repo init -u git://android.git.kernel.org/platform/manifest.git -b  
[가지_이름]
```

안드로이드 소스 전체를 내려 받아야 하므로 시간이 좀 걸린다. 네트워크 속도에 따라서는 한 시간을 넘길 수도 있다.

내려 받기가 다 끝났다면 `mydroid` 폴더의 모습이 그림 1-1과 같을 것이다.⁹⁾ 여기서 가장 중요한 폴더는 `bionic`이다. Bionic은 ARM과 x86 명령어 집합을 지원하는, 안드로이드 기기에서 실행되도록 되어 있는 C 라이브러리이다. Bionic은 일부는 BSD이고 일부는 리눅스이다. 소스 코드가 BSD C 라이브러리와 리눅스 관련 커스텀 코드들(스레드, 프로세스, 신호 등을 다루기 위한)로 이루어져 있기 때문이다. 혼성 계임을 위한 공유 라이브러리를 빌드하는 데 필요한 C 헤더 파일들 대부분이 이 `bionic` 폴더에 들어 있다.

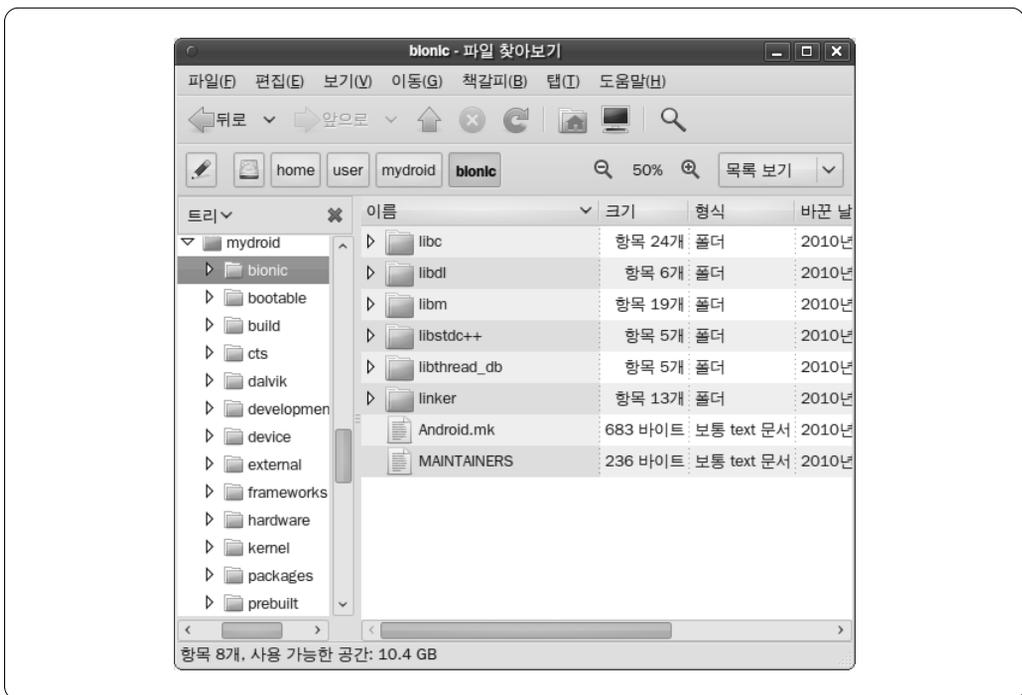


그림 1-1 안드로이드 소스 트리

9) [역주] 만일 `mydroid` 폴더에 `kernel` 폴더가 없다면, `kernel` 트리를 따로 내려 받아야 한다. 현재 디렉터리가 `mydroid`인 상태에서 다음 명령을 실행하면 된다.

```
$ git clone git://android.kernel.org/kernel/common.git kernel
```

네이티브 안드로이드 라이브러리 추출

이전 절에서 내려 받은 파일들에는 코드를 컴파일하는 데 필요한 헤더들이 포함되어 있다. 링크를 위해서는 시스템 이미지의 공유 라이브러리들, 즉 기기의 `/system/lib`에 있는 `*.so` 파일들이 필요하다. ADB 셸을 이용하면 실제 기기나 에뮬레이터의 파일 시스템에 접근할 수 있다. 다음은 ADB 셸에서 `df` 명령과 `ls` 명령으로 기기의 파일 시스템을 살펴보는 예이다.

```
$ adb shell
# df
```

```
/dev: 47284K total, 0K used, 47284K available (block size 4096)
/sqlite_stmt_journals: 4096K total, 0K used, 4096K available (block size 4096)
/system: 65536K total, 43496K used, 22040K available (block size 4096)
/data: 65536K total, 43004K used, 22532K available (block size 4096)
/cache: 65536K total, 1156K used, 64380K available (block size 4096)
/sdcard: 40309K total, 34114K used, 6195K available (block size 512)
```

```
# ls -l /system/lib
```

```
-rw-r--r-- root    root      9076 2008-11-20 00:10 libdl.so
-rw-r--r-- root    root    227480 2008-11-20 00:10 libc.so
-rw-r--r-- root    root    13368 2008-11-20 00:10 libthread_db.so
-rw-r--r-- root    root     9220 2008-11-20 00:10 libstdc++.so
-rw-r--r-- root    root   140244 2008-11-20 00:10 libm.so
-rw-r--r-- root    root    79192 2008-11-20 00:10 libz.so
-rw-r--r-- root    root    92572 2008-11-20 00:10 libexpat.so
-rw-r--r-- root    root   767020 2008-11-20 00:10 libcrypto.so
-rw-r--r-- root    root   155760 2008-11-20 00:10 libssl.so
[...후략...]
```

`df` 명령은 기기의 파일 시스템에 관한 정보를 표시한다. `df` 명령의 출력에 나온 디렉터리들을 `ls` 명령으로 살펴보면 공유 라이브러리들이 `/system/lib` 폴더에 들어 있음을 알 수 있다. 위의 `ls` 명령 출력 예에 `libc.so`(C 런타임 라이브러리), `libm.so`(수학 런타임), `libz.so`(Gzip), `libexpat.so`(XML) 등의 주요 라이브러리들이 나와 있다.

링크 과정을 위해서는 이 파일들을 기기에서 뽑아서 로컬 파일시스템에 저장해야 한다. 기기의 파일을 로컬에 복사하는 명령은 `adb pull`이다.

우선 이 파일들을 저장할 폴더를 홈 디렉터리에 만든다.

```
$ mkdir -p $HOME/tmp/android/system/lib
$ cd $HOME/tmp/android/system/lib
```

다음으로, 추출할 파일들이 꽤 많으므로 작업을 자동화할 간단한 스크립트를 하나 만들어서 실행한다. 목록 1-1의 bash 스크립트는 일단의 라이브러리 이름들을 훑으면서 각각을 기기의 `/system/lib` 폴더에서 지역 파일 시스템의 현재 디렉터리(`$HOME/tmp/android/system/lib` 폴더)로 복사한다.¹⁰⁾

목록 1-1 기기의 `/system/lib`에 있는 라이브러리 파일들을 지역 파일 시스템으로 가져오는 스크립트

```
#!/bin/bash

# 에뮬레이터 1.5 이상에서, 기기의 /system/lib에 있는
# 시스템 라이브러리들.
libs="browsertestplugin.so libEGL.so libFFTEM.so
libGLESv1_CM.so libase.so libagl.so libandroid_runtime.so
libandroid_servers.so libaudioflinger.so libc.so libc_debug.so
libcameraservice.so libcorecg.so libcrypto.so
libctest.so libcutils.so libdl.so libdrm1.so
libdrm1_jni.so libdvm.so libemoji.so
libexif.so libexpat.so libhardware.so libhardware_legacy.so
libicudata.so libicu18n.so libicuuc.so libjni_latnime.so
libjni_pinyinime.so liblog.so libm.so libmedia.so libmedia_jni.so
libmediaplayerservice.so libnativehelper.so libnetutils.so
libopencoreauthor.so libopencorecommon.so libopencoredownload.so
libopencoredownloadreg.so libopencoremp4.so libopencoremp4reg.so
libopencorenet_support.so libopencoreplayer.so libopencorertsp.so
libopencorertsreg.so libpagemap.so libpixelflinger.so
libpvasf.so libpvasfreg.so libreference-ril.so libril.so libsgl.so
libskiagl.so libsonivox.so libsoundpool.so libsqlite.so
libsrc_jni.so libssl.so libstdc++.so libsurfaceflinger.so"
```

10) [역주] 이 스크립트를 실행했을 때 `/system/lib/libase.so`가 없다는 오류가 날 수 있으나, 이 후의 예제들을 실행하는 데에는 별 지장이 없다.

```

libsystem_server.so libthread_db.so libui.so
libutils.so libvorbisidec.so libwbxml.so libwbxml_jni.so
libwebcore.so libwpa_client.so libxml2wbxml.so libz.so"

# 라이브러리 이름들을 하나씩 훑는다.
for lib in $libs
do
    # 라이브러리를 지역 파일 시스템으로 가져온다.
    adb pull /system/lib/$lib ./
done

```

ARM 프로세서용 GNU 도구사슬 설치

지금까지의 과정을 잘 마쳤다면 안드로이드 소스와 기기의 라이브러리들이 갖추어져 있을 것이다. 이제 마지막으로 준비할 소프트웨어는 안드로이드 기기에 쓰이는 ARM 하드웨어를 위한 소프트웨어를 빌드하는 데 필요한 C 라이브러리, 이진 유틸리티들, 전처리기, 컴파일러, 디버거, 프로파일러, 링커를 갖춘 도구사슬(toolchain)이다. 이 책에서는 Sourcery G++ Lite Edition for ARM이 제공하는 도구사슬을 사용한다. 이 도구사슬을 설치하는 방법은 크게 두 가지이다.

- 간편한 설치 프로그램(그림 1-2)을 내려 받아서 사용한다.
- tar 형식의 파일을 받아서 시스템에 직접 풀어 넣는다(필자는 이 방법을 선호한다. 이 방법이 훨씬 더 빠르기 때문이다).

CodeSourcery G++ 다운로드 사이트에서 tar 파일을 받았다면¹¹⁾ 다음 명령으로 홈 디렉터리에 풀면 된다.¹²⁾

```

$ cd ~
$ tar zxvf arm-2008q3-72-arm-none-linux-gnueabi.tar.gz

```

11) [역주] 2010년 3월 현재, <http://www.codesourcery.com/sgpp/lite/arm/portal/subscription?@template=lite>에서 Target OS가 GNU/Linux인 항목을 선택하면 된다.

12) [역주] 내려 받은 tar 파일의 구체적인 파일 이름과 압축 형식이 다르면(예를 들어 이 글을 번역하는 현재 최신 버전은 arm-2009q3-67-arm-none-linux-gnueabi-i686-pc-linux-gnu.tar.bz2이다) 명령을 적절히 변경해서 적용해야 할 것이다. 이후의 경로 설정 역시 마찬가지이다.

어떤 방법으로 도구사슬을 설치했든, 도구들을 어디에서든 실행할 수 있으려면 해당 bin 폴더를 시스템 경로에 추가해야 한다. \$HOME/.bashrc에 다음을 추가하기 바란다.

```
ARM_HOME=$HOME/arm-2008q3
export PATH=$PATH:$ARM_HOME/bin
```

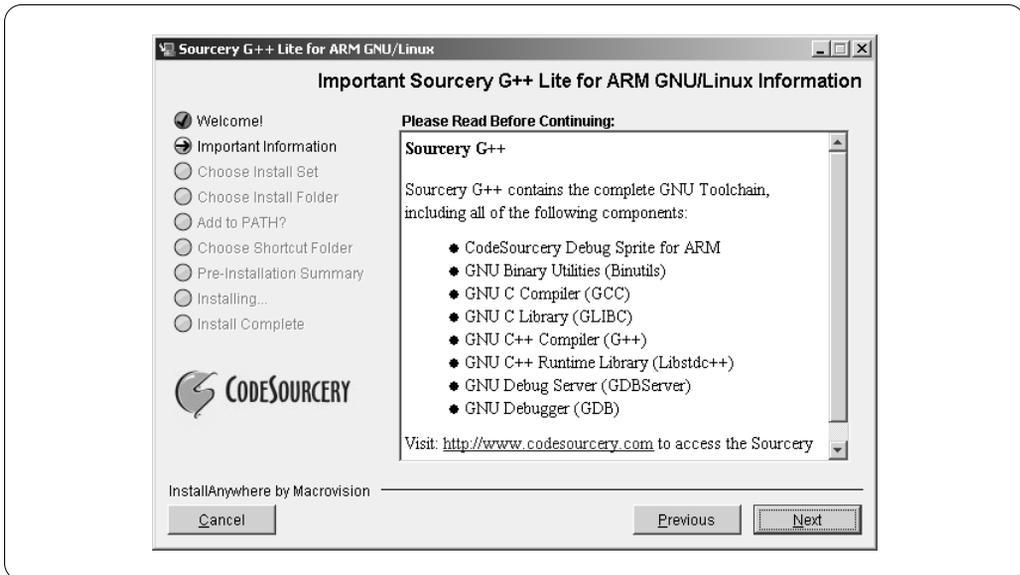


그림 1-2 Sourcery G++ Lite 설치 프로그램

다음과 같이 명령을 실행했을 때 버전 정보가 나타난다면 도구사슬이 제대로 설치, 설정된 것이다.

```
$ arm-none-linux-gnueabi-gcc --version
```

```
arm-none-linux-gnueabi-gcc (Sourcery G++ Lite 2008q3-72) 4.3.2
Copyright (C) 2008 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.
There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A
PARTICULAR PURPOSE.
```

이 도구사슬이 제공하는 주요 명령들을 간단히 소개하고 넘어가겠다.

- `arm-none-linux-gnueabi-gcc`: 이것은 C 컴파일러를 실행하는 명령으로, 리눅스의 `gcc`에 해당한다. 대부분의 옵션들이 리눅스의 `gcc`의 것들과 아주 비슷하다.
- `arm-none-linux-gnueabi-g++`: 이것은 C++ 컴파일러를 실행하는 명령으로, 앞의 `gcc` 명령을 C++용 추가 옵션들로 감싼 것이다.
- `arm-none-linux-gnueabi-ld`: 이것은 최종적인 이진 파일을 빌드하는 링커이다. 독립적인 실행 파일은 물론 정적 라이브러리 파일, 동적 라이브러리(공유 라이브러리) 파일도 만들 수 있다.
- `arm-none-linux-gnueabi-objdump`: 이 도구는 이진 파일에 대한 정보를 표시한다.
- `arm-none-linux-gnueabi-strip`: 이 도구는 이진 파일에서 기호들과 구역들(기호 및 재배치 정보, 디버그 기호 및 구역, 비전역 기호 등등)을 제거한다. 최종적인 이진 파일의 크기를 줄이는 데 도움이 된다.

예를 들어 `arm-none-linux-gnueabi-objdump`는 이진 파일의 내부를 들여다보는 데 아주 유용하다. 이 도구를 이용해서 다음과 같은 정보를 얻을 수 있다.

- 라이브러리 헤더 정보
- 전반적인 파일 헤더 내용
- 실행 가능 구역의 어셈블러 내용
- 디스어셈블된 어셈블러 코드와 혼합된 소스 코드
- 동적 기호 테이블의 내용
- 파일의 재배치(relocation) 항목들

다음은 기기에서 `$HOME/tmp/android/system/lib` 폴더로 추출한 C 런타임(`libc.so`)의 기호 테이블 정보를 표시하는 예이다.¹³⁾

```
$ arm-none-linux-gnueabi-objdump -T ~/tmp/android/system/lib/libc.so
```

```
/home/user/tmp/android/system/lib/libc.so: file format elf32-littlearm
```

```
DYNAMIC SYMBOL TABLE:
```

```
000092b0 l d .text 00000000 .text
```

13) [역주] 이 예를 비롯한 이 책의 모든 실행 예와 스크립트들에서 `/home/user`는 어떤 특별한 시스템 디렉터리가 아니라 필자의 홈 디렉터리이다. 따라서, 예제를 따라 스크립트를 만들거나 명령을 실행할 때 `/home/user`를 독자의 홈 디렉터리로 치환해야 한다. 아니면 그냥 `~`를 사용해도 대부분의 경우 문제가 없을 것이다.

```

0002ff70 l d .rodata 00000000 .rodata
00035e34 l d .ARM.extab 00000000 .ARM.extab
00039008 l d .data.rel.ro 00000000 .data.rel.ro
00039eac l d .data 00000000 .data
0003b158 l d .bss 00000000 .bss
0001b340 g DF .text 00000034 getwchar
0000de14 g DF .text 00000000 longjmp
...

```

이런 도구는 여러분의 프로그램에 필요한, 그러나 기기의 표준 라이브러리들에서 빠져 있는 기호들을 검출하는 데 유용하다.

짐작했겠지만, 이 도구사슬은 GNU GCC 도구사슬을 반영한 것이다. 사실, 눈에 띄는 유일한 차이는 명령 이름 앞에 항상 `arm-none-linux-gnueabi`가 붙는다는 점뿐이다.

이렇게 해서 도구사슬까지 갖추었다. 마지막으로, 도구사슬의 명령들을 수행하는 커스텀 스크립트 몇 가지를 작성해보자.

커스텀 컴파일 스크립트 작성

아래에서 소개하는 커스텀 컴파일 스크립트들은 x86용 이진 파일을 위한 전통적인 리눅스 컴파일 공정과 최대한 비슷한 방식으로 ARM 플랫폼용 이진 파일들을 만들어 내기 위한 것이다. 원래의 Makefile을 전혀 고치지 않고도 소스 코드를 ARM용으로 컴파일할 수 있다면 이상적이겠지만, 현실은 그렇지 못하다. 숙련된 C/C++ 리눅스 개발자에게도 ARM용 컴파일 공정이 아주 괴로운 일이 될 수 있는데, 이번 절의 스크립트들이 ARM용 컴파일 공정을 좀 더 원활하게 만들어줄 수 있을 것이다.

이번 절에서 소개할 스크립트는 다음 두 가지이다.

- `agcc`: GNU GCC 컴파일러를 실행하는 통상적인 `gcc` 명령에 해당하는 bash 스크립트. 안드로이드 기기를 위한 모든 의존성 설정을 포함한다.
- `ald`: 전통적인 `ld` 링커에 해당하는 bash 스크립트. 최종 실행 파일이나 라이브러리 파일을 생성하는 데 쓰인다.

이 스크립트들은 필수적이며, 반드시 완전하게 이해해 둘 필요가 있다. 이해를 돕기 위해, 우선 전통적인 x86용 컴파일 공정이 어떤 것이고 그것을 ARM 기기용 컴파일에 그대로 사용할 수 없는 이유는 무엇인지 간단히 살펴보자.

전통적인 리눅스 컴파일 공정

전통적인 x86 리눅스 컴파일 공정에서 개발자는 일단의 C/C++ 소스 파일들과 하나의 Makefile을 이용해 최종 실행 파일을 만들어 낸다. Makefile의 기본적인 뼈대는 다음과 같은 모습이다.

```
# 기본적인 Makefile의 뼈대

# 목적 파일들
OBJS = file1.o file2.o....

# 헤더 파일들
INC = -Ifolder1 -Ifolder2 ...

# 라이브러리들과 그 경로들(링킹에 쓰임)
LIB = -lc -lm -Lpath1 -Lpath2 ...

# 주 빌드 대상
all: $(OBJS)
    @echo Linking..
    gcc -o myapp $(OBJ) $(LIB)

# C 파일들의 컴파일
%o:%.c
    @echo Compiling $<...
    gcc -c $< $(INC)
```

OBJS, INC, LIB는 변수들이고(각각 목적 파일들, 헤더 파일들, 라이브러리들을 담는다), 변수와 = 기호 오른쪽 부분은 해당 변수의 값이다. all, %o 항목은 빌드 대상(target)이라고 부르는 것이다.

대상의 콜론(:) 오른쪽에 있는 것들은 그 대상이 의존하는(즉, 먼저 만들어져 있어야 하는) 대상들이다. 대상 줄 아래에 탭 문자 하나로 들여 쓴 줄들은 그 대상을 만드는 명령들이다.

명령줄에서 make를 실행하면 그 디렉터리에 있는 Makefile의 주 대상인 all(이것은 내장 대상 이름이다)의 빌드가 시작된다. make는 우선 주어진 대상의 의존성이 만족되었는지 점검하고, 만족되지 않았다면 의존 대상들부터 먼저 만든다. 위의 예의 경우, 주 대상 정의 줄은 all: \$(OBJ)이다. 이는 all을 만들려면 먼저 \$(OBJ) 대상부터 만들어야 한다는 뜻이다. 그런데 \$(OBJ)는 OBJ라는 변수의 값이며, 그 값들은 .o로 끝나는

목적 파일(object file)들이다. 이에 의해 %o:%c 대상의 빌드가 진행된다. %는 DOS의 *에 해당한다. 예를 들어 %.c는 현재 디렉터리의 모든 .c 파일을 뜻한다. 따라서 이 대상은 간단히 말하면 “주어진 .o 파일들이 의존하는 모든 .c 파일들을 처리하라”는 뜻이다. 결과적으로, 주어진 .o 파일에 해당하는 현재 디렉터리의 .c 파일(C 소스 파일) 각각마다 터미널에 다음과 같은 출력이 나오게 된다.

```
Compiling file1.c ...
gcc -c file1.c -Ifolder1 -Ifolder2 ...

Compiling file2c ...
gcc -c file2c -Ifolder1 -Ifolder2 ...
```

Makefile의 다음 두 줄에 주목하자.

```
@echo Compiling $<...
gcc -c $< $(INC)
```

GNU make는 @echo를 만나면 그 다음의 메시지를 콘솔에 출력한다. \$<는 %o:%c 대상의 둘째 인수(이 경우 현재 디렉터리의 한 .c 파일의 이름)로 확장된다. 그 다음 줄 gcc -c \$< \$(INC)는 gcc -c file1.c -Ifolder1 -Ifolder2 ...로 확장된다.

이러한 과정이 OBJ 변수에 정의된 모든 목적 파일에 대해 진행된다. 컴파일 과정에서 오류가 없었다면 all 대상으로 돌아가서 해당 명령들이 실행된다. all 대상은 링크 공정을 실행한다. 이에 의해 콘솔에 다음과 같은 메시지가 나타난다.

```
Linking...
```

all 대상의 gcc -o myapp \$(OBJ) \$(LIB) 줄은 다음과 같이 확장된다.

```
gcc -o myapp file1.o file2.o ... -lc -lm -Lpath1 -Lpath2
```

이에 의해 최종적인 실행파일 myapp이 만들어진다.

이상이 x86 PC용 시스템을 위한 이진 실행파일을 만들어 내는 과정이다. 그러나 이 과정을 안드로이드 기기용 이진 파일에 그대로 적용할 수는 없다. 이유는 여러 가지인

데, 하나는 통상적인 GCC가 x86 아키텍처를 위한 이진 파일을 만들어 낼 뿐이며, 그런 이진 파일을 ARM 프로세서가 장착된 시스템에서 실행할 수는 없다는 것이다.

gcc를 arm-none-linux-gnueabi-gcc로 대체하는 것으로는 문제가 완전히 해결되지 않는다. arm-none-linux-gnueabi-gcc -c file1.c -Ifolder1 -Ifolder2 ... 같은 명령은 도구사슬에 포함된 표준 헤더 파일들(C 런타임이나 기타 라이브러리들을 위한)을 이용해서 소스 코드를 컴파일한다. 그러나 사용하는 도구사슬 버전에 따라서는 이 때문에 바람직하지 않은 부작용들(링크 시점에서 기호들을 찾지 못하는 등)이 생길 수 있다. 예를 들어 다음은 3D 게임 Doom의 한 소스 파일을 ARM용으로 컴파일하려 했을 때 보고된 오류 메시지이다.

```
$ arm-none-linux-gnueabi-gcc -Werror -Dstricmp=strcasecmp \
-msoft-float -mcpu=arm9 \
-g -Wall -DX11 -fpic -o sys_linux.o -c \
/home/user/workspace/Android.Quake/native/Doom/sys_linux.c
```

```
cc1: warnings being treated as errors
/home/user/workspace/Android.Quake.Preview/native/Quake/sys_linux.c:
In function 'floating_point_exception_handler':
/home/user/workspace/Android.Quake.Preview/native/Quake/sys_linux.c:350:
error: implicit declaration of function 'sysv_signal'
```

이 오류는 C 시스템 호출 `signal`(사용자 정의 함수를 OS 신호에 묶기 위해 쓰였다)의 명명규약이 도구사슬의 구현과 안드로이드의 구현에서 서로 다르기 때문에 생긴 것이다(전자는 통상적인 `signal`을 사용하는 반면 안드로이드는 `sysv_signal`을 사용한다).

기호 누락 문제와 관련해서 한 마디 하자면, 공유 라이브러리를 빌드하는 경우에는 여러분의 코드에서 일부 함수나 변수가 빠져 있어도 컴파일러가 오류를 보고하지 않는다. 그런 상태에서 프로그램을 실행하면 해당 라이브러리가 제대로 적재되지 못한다. 네이티브 코드를 다루는 도구들이 번거롭고 사용하기 어렵다는(특히 초보에게는) 점에서, 필자는 이런 종류의 문제가 혼성 게임 개발에 가장 걸림돌이 된다고 생각한다. 그래서 필자는 간단한 해결책을 고안했다. 공유 라이브러리의 함수를 호출하는 작은 주 프로그램을 만들어 보는 것이다. 제2장에 그러한 예가 나온다.

안드로이드용 컴파일 스크립트

목록 1-2는 안드로이드용 코드의 컴파일을 편하게 수행하기 위한 agcc라는 bash 스크립트이다.

목록 1-2 안드로이드용 컴파일 보조 스크립트 agcc

```
#!/bin/bash
#####
# 안드로이드 컴파일 보조 스크립트
# CodeSourcery G++ Toolchain for ARM 기준임.
#####

#####
# 파일들이 설치된 디렉터리 트리의 루트
# 여러분의 시스템에 맞게 적절히 수정할 것.
#####
HOME=/home/<사용자이름>

# JVM 위치. 적절히 수정할 것.
JAVA_HOME=/usr/lib/jvm/java-1.5.0-sun

# 기기의 시스템 이미지
SYS_ROOT=$HOME/tmp/android/system

# 안드로이드 소스 코드
SYS_DEV=$HOME/mydroid

# Code Sourcery 도구사슬 위치
TOOLCHAIN_ROOT=$HOME/arm-2008q3

#####
# 헤더 파일들과 라이브러리 파일 위치들
# 변경할 필요 없음.
#####
BASE=$SYS_DEV/frameworks/base

# C 런타임
LIBC=$SYS_DEV/bionic/libc

# 수학 라이브러리
LIBM=${SYS_DEV}/bionic/libm
```

```

# 일부 필수 GCC 컴파일러 라이브러리들
TC=${SYS_DEV}/prebuilt/linux-x86/toolchain/arm-eabi-4.3.1/lib/gcc/
arm-eabi/4.3.1

# 커널 헤더들
KERNEL=${SYS_DEV}/kernel

# GNU GZIP
LIBZ=${SYS_DEV}/external/zlib

# Expat(XML 파서)

EXPAT=${SYS_DEV}/external/expat/lib

# 헤더 파일들
AND_INC="-I$JAVA_HOME/include"
AND_INC+="-I${JAVA_HOME}/include/linux"
AND_INC+="-I${LIBC}/include "
AND_INC+="-I${LIBC}/arch-arm/include"
AND_INC+="-I${LIBC}/kernel/arch-arm/include "
AND_INC+="-I${LIBM}/include"
AND_INC+="-I${BASE}/include"
AND_INC+="-I${TC}/include"
AND_INC+="-I${KERNEL}/include"
AND_INC+="-I${KERNEL}/arch/arm/include -I${KERNEL}/arch/arm/mach-
ebsa110/include"
AND_INC+="-I${SYS_DEV}/system/core/include"
AND_INC+="-I${LIBZ}"
AND_INC+="-I${EXPAT}"

#####
# 도구사슬 컴파일러 명령
#####
CROSS=arm-none-linux-gnueabi-
GCC=${CROSS}gcc

# 디버깅 시 다음 줄의 주석 표시를 제거
# echo ${GCC} -nostdinc ${AND_INC} $@

# 실행!
${GCC} -nostdinc ${AND_INC} "$@"

```

agcc 스크립트의 변수들 중 다음 변수들은 여러분의 시스템에 맞게 조정할 필요가 있다(예를 들어 도구사슬을 앞에 나온 것과 다른 곳에 설치했다면).

- HOME: 이것은 모든 소프트웨어 구성요소가 들어 있는 디렉터리 트리의 루트이다. 기본값은 사용자 홈 디렉터리이다(이 책의 예제들에서는 항상 /home/user). 이 변수는 다른 여러 변수들에 쓰인다.
- JAVA_HOME: 이것은 JDK(Java SDK)의 위치이다. Java 혼성 게임의 C 코드는 JNI를 통해서 Java 코드와 연동하므로, C 코드를 컴파일하려면 JDK의 JNI 헤더 파일들과 라이브러리들이 필요하다.
- SYS_ROOT: 이것은 이전 절에서 기기에서 추출한 시스템 라이브러리들이 있는 경로이다. 기본값은 \$HOME/tmp/android/system.
- SYS_DEV: 앞에서 내려 받은 안드로이드 소스 코드가 있는 곳이다. 그 소스 코드의 일부 헤더 파일들과 공유 라이브러리들이 컴파일에 쓰인다. 기본값은 \$HOME/mydroid.
- TOOLCHAIN_ROOT: CodeSourcery 도구사슬이 설치된 경로이다. 기본값은 \$HOME/arm-2008q3. 버전에 따라 다를 수 있다.

커스텀 컴파일 스크립트는 이러한 변수들에 기초해서 다음과 같은 여러 기본 라이브러리 위치를 위한 변수들을 정의한다. 주요한 변수들을 들자면 다음과 같다.

- LIBC: 이 변수는 안드로이드용 C 런타임 헤더 파일들이 있는 경로를 정의한다. \$SYS_DEV/ bionic/libc이다.
- LIBM: 이 변수는 pow나 삼각함수 등 흔히 쓰이는 수학 함수들을 모은 라이브러리의 경로를 정의한다. 값은 \${SYS_DEV}/bionic/libm이다.
- KERNEL: 커널 헤더들이 있는 경로를 정의한다. 커널 헤더들은 혼성 게임, 특히 고성능 3D 게임에 요긴하다. 값은 \${SYS_DEV}/kernel이다.
- LIBZ: GNU Gzip 라이브러리 헤더들의 경로이다. 게임이 ZIP 파일을 다루는 경우 이 라이브러리가 필요하다. 값은 \${SYS_DEV}/external/zlib이다.
- EXPAT: XML 파서 중 하나인 Expat 라이브러리의 헤더들이 있는 경로를 정의한다. 값은 \${SYS_DEV}/external/expat/lib이다. 요즘 게임들은 대부분 XML 파일에 게임과 구성 정보를 저장하므로 이 라이브러리가 거의 필수이다.

그 외에, 추가적인 컴파일 시점 의존성들을 포함시키는 데 쓰이는 변수 두 개가 있다:

```
BASE=${SYS_DEV}/frameworks/base
TC=${SYS_DEV}/prebuilt/linux-x86/toolchain/arm-eabi-4.3.1/lib/gcc/
arm-eabi/4.3.1
```

스크립트는 이 변수들을 이용해서 필수 헤더 경로들을 -I경로1, -I경로2, ... 형태로 모아 `AND_INC` 변수에 정의한다. 마지막으로, 다음과 같이 도구사슬의 컴파일 명령을 실행한다.

```
`${GCC} -nostdinc ${AND_INC} "$@"
```

`\${GCC}`는 `arm-none-linux-gnueabi-gcc`로 확장되며 `\${AND_INC}`는 앞에서 정의한 헤더 경로들로 확장된다. 그리고 "\$@"는 사용자가 이 스크립트를 실행하면서 지정한 나머지 모든 인수로 확장된다. 컴파일 명령에 `-nostdinc`라는 옵션을 주었음을 주목할 것. 이 옵션은 표준 헤더 파일들(도구사슬 자체에 포함되어 있던 것들)을 사용하지 말라는 뜻이다. 안드로이드용 이진 파일을 만들려면 표준 헤더들이 아니라 안드로이드 소스에 있는 헤더들을 사용해야 하므로 이 옵션이 꼭 필요하다.

이제 컴파일을 위한 스크립트가 갖추어졌다. 링크 역시 표준 라이브러리들 대신 안드로이드 시스템 라이브러리들을 사용해야 한다. 그럼 링크용 스크립트를 보자.

안드로이드 링커 스크립트

목록 1-3은 안드로이드용 이진 파일의 링크를 편하게 수행하기 위한 `ald`라는 `bash` 스크립트이다.

목록 1-3 안드로이드용 링커 보조 스크립트 `ald`

```
#!/bin/bash

#####
# 안드로이드 링커 보조 스크립트
# 아래 값들을 여러분의 시스템에 맞게 적절히 수정할 것.
#####
HOME=/home/<사용자이름>
JAVA_HOME=/usr/lib/jvm/java-1.5.0-sun

# 기기의 시스템 이미지
```

```
SYS_ROOT=$HOME/tmp/android/system

# 안드로이드 소스 코드
SYS_DEV=$HOME/mydroid

# Code Sourcery 도구사슬 위치
TOOLCHAIN_ROOT=$HOME/arm-2008q3

# 안드로이드용 libgcc.a
LIBGCC=${SYS_DEV}/prebuilt/darwin-x86/toolchain/arm-eabi-4.2.1/lib/gcc/
arm-eabi/4.2.1/libgcc.a

# CodeSourcery의 libgcc.a
#LIBGCC=${TOOLCHAIN_ROOT}/lib/gcc/arm-none-linux-gnueabi/4.3.2/libgcc.a

# 링크할 라이브러리들: C 런타임, 수학, 기타 등등
LIBRARIES="-lc -lm ${LIBGCC}"

# 라이브러리 경로들
LIB_PATHS="-rpath /system/lib \
-rpath ${SYS_ROOT}/lib \

-L${SYS_ROOT}/lib \
-L${JAVA_HOME}/jre/lib/i386 -L."

# 링커 플래그
LD_FLAGS="--dynamic-linker=/system/bin/linker -nostdlib"

#####
# 링커 실행 명령
#####
CROSS=arm-none-linux-gnueabi-
GCC=${CROSS}ld

# 디버깅 시 다음 줄의 주석 표시를 제거
#echo "${GCC} $LD_FLAGS $LIB_PATHS @$ $LIBRARIES"

# 실행!
${GCC} $LD_FLAGS $LIB_PATHS @$ $LIBRARIES
```

agcc처럼 이 `ld`도 필수적인 소프트웨어 구성요소들의 경로나 기타 링커 옵션들을 위한 변수들을 정의한다. 각각을 살펴보면 다음과 같다.

- **LIBRARIES:** 이 변수는 최종 이진 파일에 링크할 필수 라이브러리들을 정의한다. `-lc`는 C 런타임, `-lm`은 수학 라이브러리, `libgcc.a`는 추가적인 기호들을 담은 라이브러리이다. 파일 시스템에서 라이브러리를 읽을 때 링커는 `-lc`를 `libc.so`로 확장한다.
- **LIB_PATHS:** 링크를 위해서는 앞에서 정의한 라이브러리들 외에도 여러 가지 라이브러리들이 필요하다. 이 변수는 링커가 파일 시스템에서 라이브러리들을 찾을 수 있도록 여러 경로들을 `-L경로1, -L경로2` 형태로 정의한다. 그리고 `-rpath`로 실행시점 공유 라이브러리 검색 경로를 설정한다. 실행 시점 검색 경로로는 다음 두 가지가 필요하다.
 - `-rpath /system/lib`는 기기의 실행시점 라이브러리들의 경로이다. 기기 자체에서 프로그램이 제대로 실행되려면 이 경로를 반드시 설정해야 한다.
 - `-rpath ${SYS_ROOT}/lib`는 같은 라이브러리들의 데스크톱 시스템상의 경로이다. 이진 파일을 제대로 링크하려면 이 설정이 필요하다.
- **LD_FLAGS:** 이 변수는 링커 플래그(옵션)들을 정의한다. 구체적으로는 다음 두 플래그를 정의한다.
 - `--dynamic-linker=/system/bin/linker`는 프로그램이 동적 링커를 사용하도록 한다. 이 경우 `/system/bin/linker`는 기기이다.
 - `-nostdlib`은 링커가 도구사슬에 포함된 표준 라이브러리들(C 런타임, 수학 라이브러리 등) 대신 링크 시점에서 사용자가 지정한 라이브러리들을 사용하게 만든다.

마지막으로, `${GCC} $LD_FLAGS $LIB_PATHS @$@ $LIBRARIES` 명령에 의해 실제로 링크가 수행된다. 이 명령은 다음과 같이 확장된다(실제로는 모두 한 줄).

```
arm-none-linux-gnueabi-ld --dynamic-linker=/system/bin/linker
-nostdlib -rpath /system/lib
-rpath /home/user/tmp/android/system/lib
-L/home/user/tmp/android/system /lib
-L/usr/lib/jvm/java-1.5.0-sun/jre/lib/i386
-L. [사용자 인수들] -lc -lm
/home/user/arm-2008q3/lib/gcc/
arm-none-linux-gnueabi/4.3.2${LIBDIR}/libgcc.a
```

스크립트의 \$@가 명령줄에서 사용자가 지정한 모든 인수로 확장되었음을 주목하기 바란다.

스크립트 시험해 보기

목록 1-2와 1-3을 \$HOME/bin 디렉터리에 각각 agcc와 ald라는 이름으로 저장하고 실행 가능한 파일로 설정한 후 다음을 시험해 보기 바란다.

```
$ agcc --version
```

```
arm-none-linux-gnueabi-gcc (Sourcery G++ Lite 2008q3-72) 4.3.2
Copyright (C) 2008 Free Software Foundation, Inc.
This is free software; see the source for copying conditions. There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

```
$ ald --version
```

```
arm-none-linux-gnueabi-ld: warning: library search path
"/usr/lib/jvm/java-6-sun/jre/lib/i386" is unsafe
for cross-compilation
GNU ld (Sourcery G++ Lite 2008q3-72) 2.18.50.20080215
Copyright 2007 Free Software Foundation, Inc.
This program is free software; you may redistribute it under the terms of
the GNU General Public License version 3 or (at your option) a later version.
This program has absolutely no warranty.
```

주의 이 스크립트들을 Windows에서 메모장이나 워드패드로 작성하지는 말 것. 그러면 파일에 유효하지 않은 문자들이 삽입되기 때문에 스크립트 실행 시 잘못된 결과가 나올 수 있다. 사실 필자가 바로 그런 실수를 저질러서 컴파일러의 이상한 결과에 시간을 한참 낭비했다.

안드로이드 개발 환경 설정

다음으로 할 일은 IDE에 안드로이드 SDK를 설정하는 것이다. 여기에서는 Eclipse 3.5(Galileo, <http://www.eclipse.org>)와 안드로이드 SDK 1.6을 기준으로 설명하나, 더 최신 버전들이라도 크게 다르지는 않다. Eclipse가 이미 깔려 있다고 가정하겠다.

1. Eclipse를 실행하고 주 메뉴에서 Help ▶ Install New Software를 선택한다(그림 1-3).



그림 1-3 Eclipse 3.5(Galileo)의 Help 메뉴에서 Install New Software를 선택한다

2. Available Software 창(그림 1-4)에서 Add 버튼을 클릭한다.

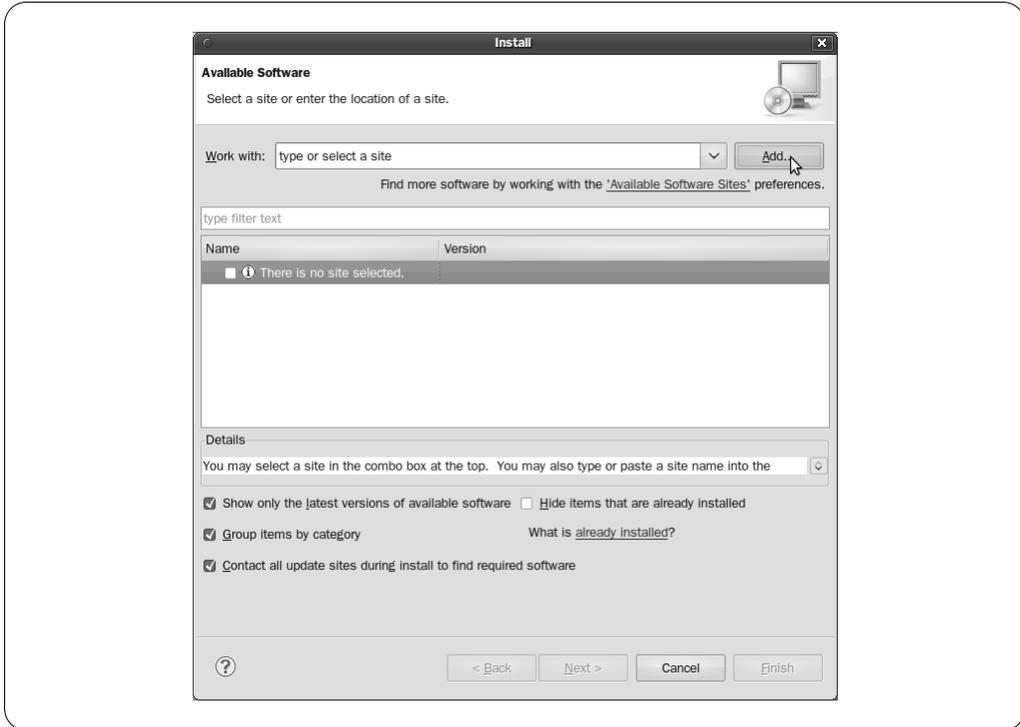


그림 1-4 소프트웨어 추가를 위해 Add 버튼을 클릭한다

3. Add Site 대화상자에서 Name:에 **Android**를, 그리고 Location:에는 **https://dl-ssl.google.com/android/eclipse**를 입력한다(그림 1-5).

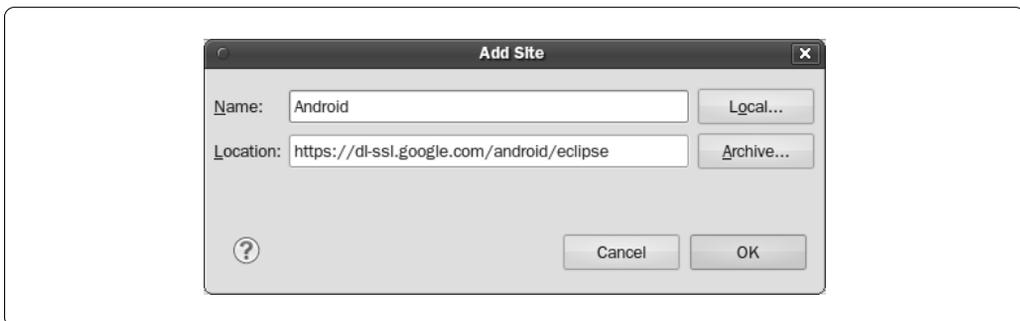


그림 1-5 안드로이드 사이트를 추가한다

4. Available Software 창(그림 1-4)의 목록에서 방금 추가한 안드로이드 사이트를 선택한다. 목록에 그 사이트가 나와 있지 않다면 Available Software Sites 링크를 클릭한 후 Add 버튼을 클릭해서 사이트를 추가하기 바란다(그림 1-6).

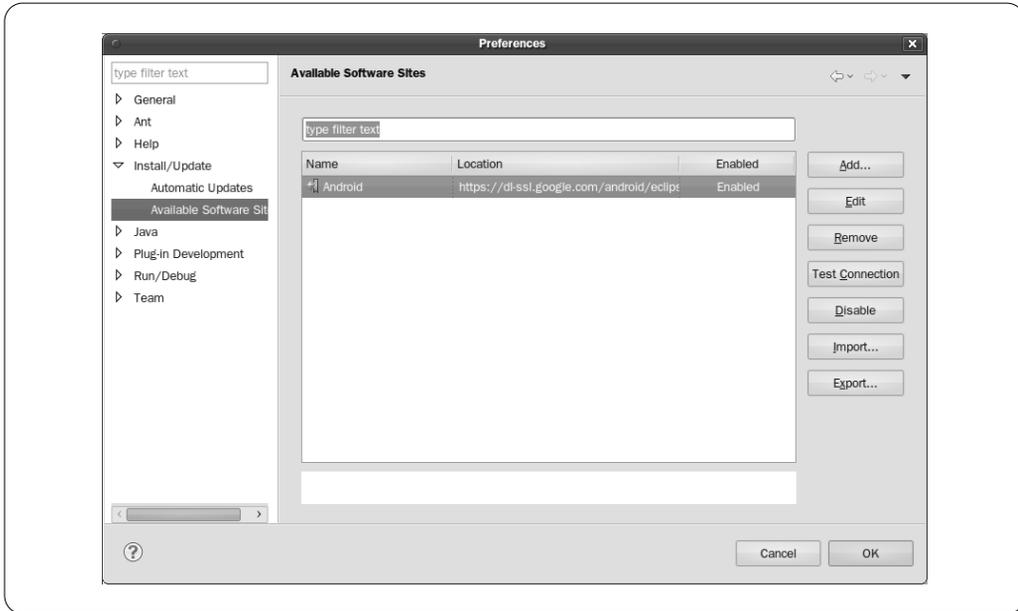


그림 1-6 방금 추가한 안드로이드 사이트가 Preferences의 Available Software Sites 목록에 나와 있다

5. Available Software 창의 목록에서 Developer Tools 체크상자를 체크한다. 그 아래 두 항목이 모두 체크되어야 한다(그림 1-7).

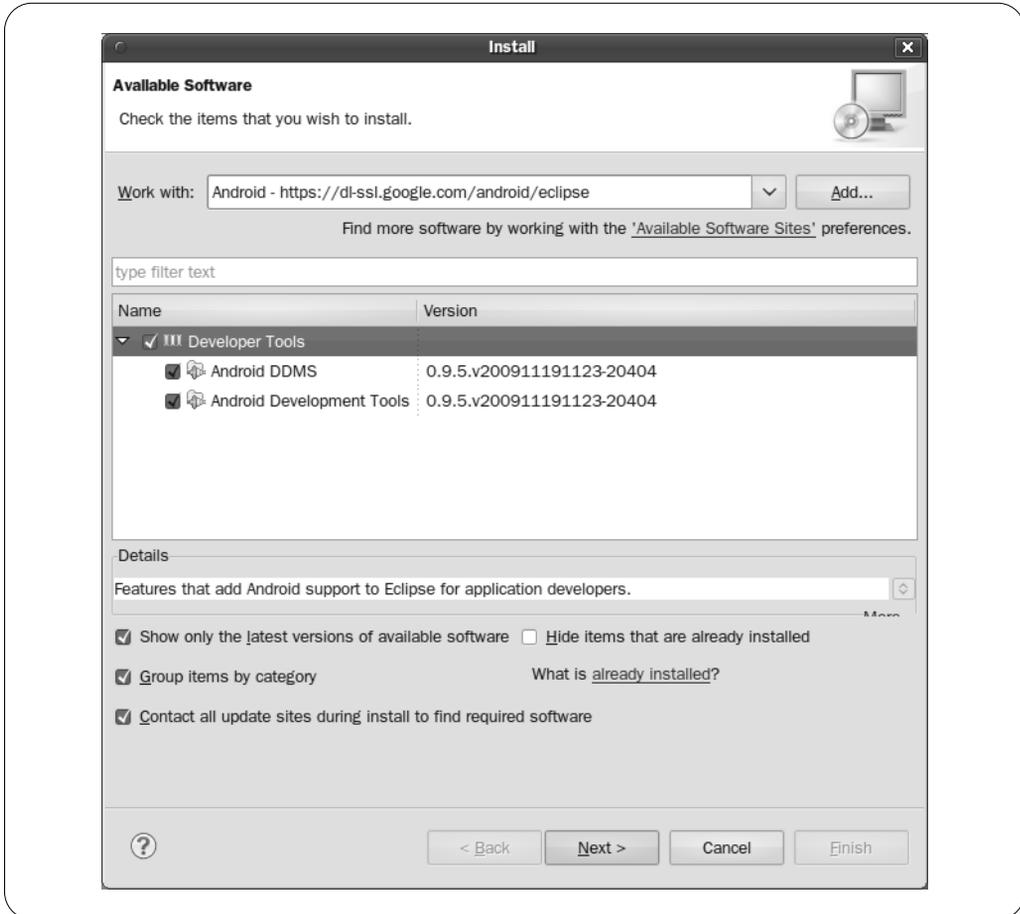


그림 1-7 Available Software 창에서 안드로이드 플러그인이 선택된 모습

6. 이제 Next 버튼을 클릭하고, 설치 마법사(wizard)의 지시를 따라 설치를 진행한다. 그림 1-8은 설치 도중 사용권(license)에 동의하는 단계의 모습이다. 설치가 다 끝나면 Eclipse가 재시작을 요청한다.

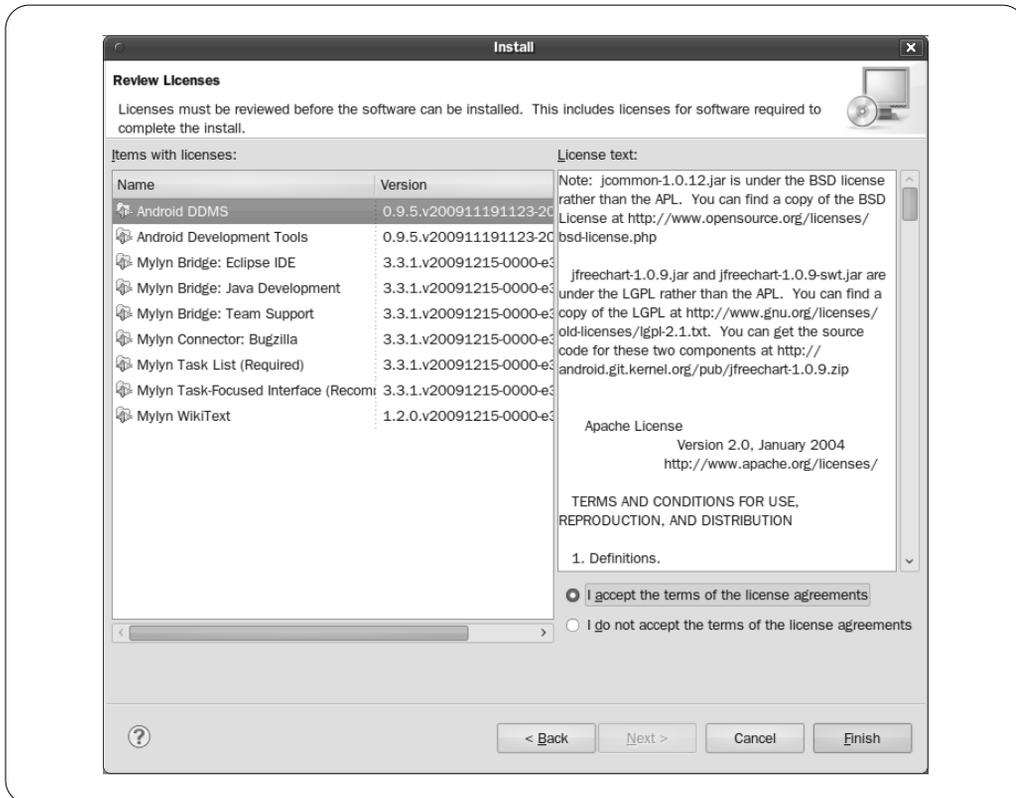


그림 1-8 설치 도중 소프트웨어 사용권 동의를 묻는다

7. Eclipse가 재시작된 후 주 메뉴에서 Window ▶ Preferences를 선택한다. Preferences 창의 왼쪽 트리에서 Android를 선택하고, 오른쪽의 SDK Location 칸에 안드로이드 SDK의 경로를 설정한다(그림 1-9). 그 아래의 Apply를 클릭했을 때 설치된 안드로이드 SDK 버전들이 그 아래 모두 나타나야 제대로 설정한 것이다. 제대로 되었다면 OK를 클릭한다.

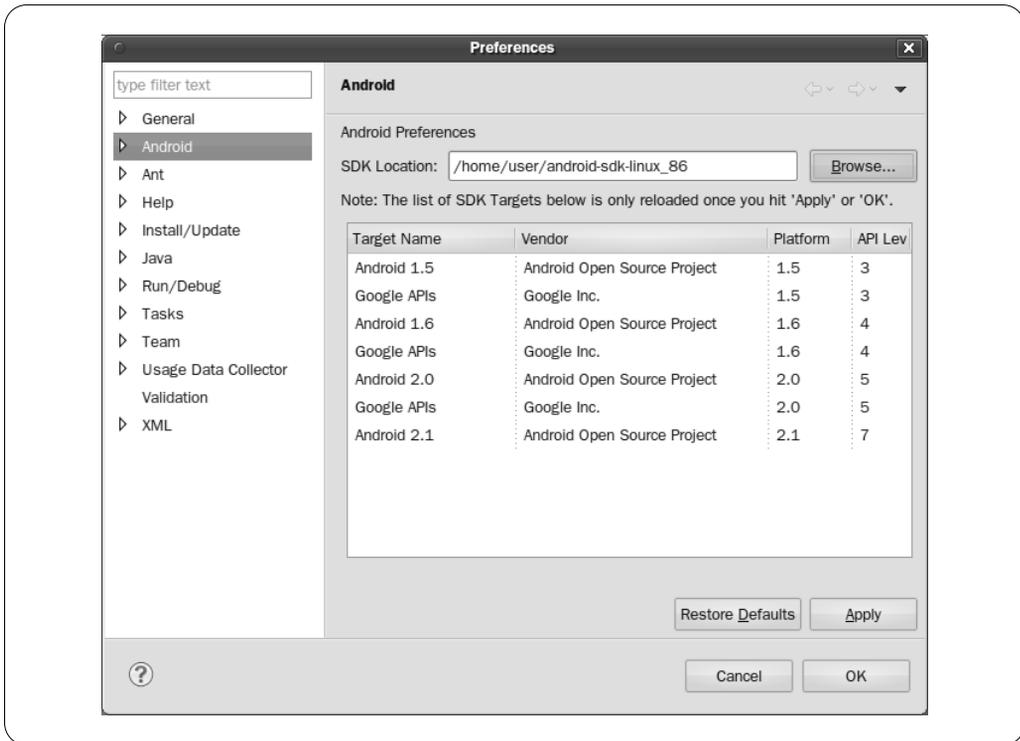


그림 1-9 안드로이드 SDK의 경로를 설정한다

8. 이제 설치, 설정이 제대로 되었는지 확인해 보자. 주 메뉴에서 File ▶ New를 선택했을 때 목록에 Android라는 항목이 있고 그것을 펼쳤을 때 Android Project 등의 항목들이 있으면 제대로 된 것이다(그림 1-10).

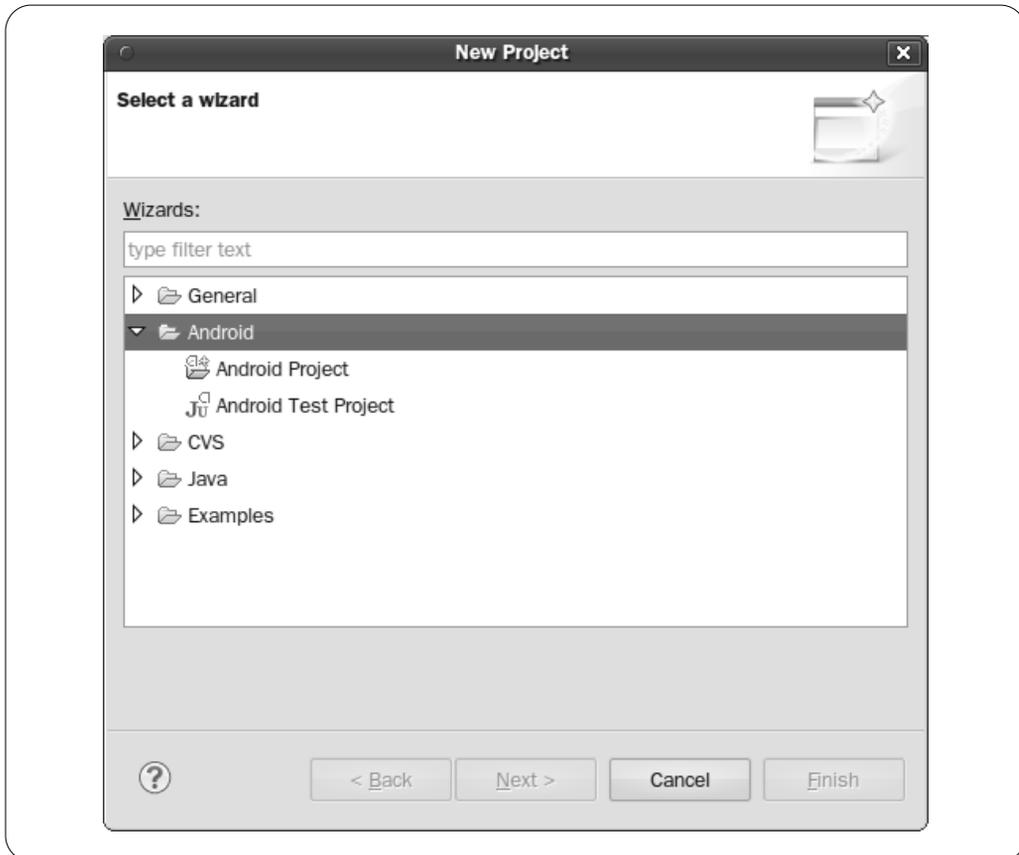


그림 1-10 New Project 창에 Android 항목이 있으면 설정이 제대로 된 것이다

이렇게 해서 안드로이드 개발을 위한 환경이 모두 갖추어졌다.

안드로이드 NDK에 대해

안드로이드 SDK 1.5에서 Google은 안드로이드 NDK(Native Development Kit)라는 새로운 개발 키트를 내놓았다. 현재 NDK의 최신 버전은 1.6이다(http://developer.android.com/sdk/ndk/1.6_r1/index.html). Google에 따르면, NDK는 SDK를 보완하는, 안드로이드 응용프로그램의 네이티브 부분을 빌드하기 위한 수단이다. NDK 1.5는 다음과 같은 일단의 시스템 헤더들을 제공한다.¹⁴⁾

- libc(C 라이브러리) 헤더들
- libm(수학 라이브러리) 헤더들
- JNI 인터페이스 헤더들
- libz(Zlib 압축) 헤더들
- liblog(안드로이드 로깅) 헤더들
- C++ 지원 헤더들

NDK에는 네이티브 ARM 이진 코드를 생성하는 크로스플랫폼 도구사슬들과 문서화, 예제들도 포함되어 있다.

안타깝게도, Google에 따르면 NDK는 안드로이드 1.5 이상의 플랫폼들만 지원한다(도구사슬의 네이티브 라이브러리들이 SDK 버전 1.0, 1.1의 것들과 호환되지 않기 때문이다). 이는 모든 버전을 지원한다는 이 책의 목표와 충돌한다. 그래서 이 책의 본문에서는 NDK 대신 Code Sourcery의 도구사슬을 사용하고, 대신 개별적인 부록에서 제6장과 제7장의 예제들을 NDK를 이용해 컴파일하는 방법을 설명한다.

요약

여기까지 무사하게 왔다면 안드로이드 게임 개발의 초기 진입 장벽을 잘 넘긴 것이다. 이번 장에서 여러분은 혼성 게임을 컴파일하기 위한 리눅스 환경을 설정했다. 구체적으로는,

- 안드로이드 소스를 마련했고,
- 컴파일에 필요한 기기 시스템 라이브러리들을 기기에서 추출했고,
- CodeSourcery G++ 도구사슬을 설정했고,

14) [역주] NDK 1.6은 이들 외에 OpenGL ES 1.1 헤더들도 제공한다. 최근(2010년 3월) 나온 NDK r3(이 버전부터는 SDK 버전 번호 대신 릴리스 번호를 사용한다)에는 OpenGL ES 2.0 관련 헤더들이 추가되었다.

- 커스텀 컴파일 스크립트들을 작성했고,
- 안드로이드 SDK와 Eclipse IDE를 설정했다.

이후의 장들에서 순수 Java 게임들과 Doom, Wolfenstein 같은 본격적인 3D 슈팅 게임들을 만들려면 이러한 과정이 필수적이다. 다음 장에서는 안드로이드 네이티브 코드를 작성, 컴파일하는 기본적인 방법을 살펴보고, 그런 방법으로 만든 네이티브 라이브러리를 Java 응용프로그램에서 호출하는 방법도 이야기한다.