Developing Games with NDK

In this chapter, we will cover the following recipes:

- Setting up the game display on Android
- Adding a game control on Android
- Handling audio with MediaPlayer and SoundPool through JNI

Introduction

Besides multimedia applications, games are another category of apps where Android NDK is widely used. Game development is a complex topic and many books have been written to discuss it. In this chapter, we will illustrate Android game development with NDK by porting a classic PC game, **Wolfenstein 3D** (also known as Wolf3D) to Android.

The Wolf3D game has been ported to many platforms, and there are many different open source versions available. We will use Wolfenstein 3D for the GP2X (a Linux-based hand-held video game console available in South Korea) because the source code is clean and relatively easy to understand.

The original Wolf3D game for GP2X uses **Simple Directmedia Layer** (**SDL**) for rendering. We will first set up the game display on Android with SurfaceView and Bitmap. We will then implement a poor man's game control, so that we can play the game. Finally, we will discuss how to enable the audio for Wolf3D on Android.

Setting up the game display on Android

This recipe discusses how to set up the game display using SurfaceView and bitmap.

Getting ready

This recipe uses the `jnigraphics` library API discussed in *Chapter 7*, *Other Android NDK API*. Readers are recommended to read the *Programming with jnigraphics Library at Android NDK* recipe first.

How to do it...

The following steps describe how to create an Android project, which sets up the game display for Wolf3D on Android:

1. Create an Android application named `Wolf3DPort` with native support. Set the package name as `cookbook.chapter11.wolf3dport`. Please refer to the *Loading native libraries and registering native methods* recipe of *Chapter 2*, *Java Native Interface*, if you want more detailed instructions.
2. Download the source code of `Wolf3D gp2xwolf3d-src.rar` from `http://sourceforge.net/projects/gp2xwolf3d/files/gp2xwolf3d/Wolf3d%20for%20GP2X%20v0.6/`. Extract the archive and add all the source files to the `jni` folder.
3. Download the shareware version 1.4 of `Wolf3D` data files from `http://www.users.globalnet.co.uk/~brlowe/wolf3d14.zip`, and the 1.0 data files from `http://www.belowe.com/wlf3dv10.zip`. Extract the downloaded ZIP files to two folders, namely `wolf3d14` and `wolf3d10`. Change all files with extension `WL1` from `wolf3d14` and `CONFIG.WL1` from `wolf3d10` to lowercase letters. Compress these files to `wolfsw.zip`, and add it under the `assets` folder.

   Note that we need to change names to lowercase letters because the names used in the code (`id_ca.c` and `wl_main.c`) are in lowercase.

4. Apply the following changes to the `Wolf3D` source files. We only list a few listings here and the details can be found in the source code available at the book website:
   - `id_heads.h`: Comment the following line, because Android NDK does not support `glob`:

     ```
     #include <glob.h>
     ```

   - `wl_def.h`: Comment the following line, because the GP2X specific header is not supported on Android:

     ```
     #include "GP2XCONT.h"
     ```

- wl_menu.c: Comment the last few lines that use glob.
- sd_null.c: Change the input parameter of SD_PlaySound from soundnames to soundnamesWL6. Add SD_PlaySoundWL6 and SD_PlaySoundSOD so the build can succeed.
- wl_main.c: Comment out a few lines using glob. Add the following code to read datadir and display dimensions:

```
w0 = true;  //we'll use the *.wl1 game files
//get the directory for the game data files
int idx = MS_CheckParm("datadir");
if (idx) {
  datadir = argv[idx+1];
} else {
  //by default, set it to current directory
  datadir = "./";
}
//get the dimension
idx = MS_CheckParm("width");
if (idx) {
  vwidth = atoi(argv[idx+1]);
} else {
  vwidth = 320;
}
idx = MS_CheckParm("height");
if (idx) {
  vheight = atoi(argv[idx+1]);
} else {
  vheight = 200;
}
```

- misc.c: Update the read/write functions to read files from datadir, including OpenRead, OpenWrite, and OpenWriteAppend.

datadir should be set in the WolfMain function at wl_main.c. As an example, we show the code for the OpenRead function.

```
int OpenRead(const char *fn)
{
  int fp;
  char fpath[200];
  sprintf(fpath, "%s%s", datadir, fn);
  fp = open(fpath, O_RDONLY | O_BINARY);
  return fp;
}
```

5. Add the following new source files under the `jni` folder:

    - `vi_android.c`: This file is modified based on `vi_sdl.c`. The key function is `VW_UpdateScreen`. This function converts the display data to pixels, adds them to the bitmap, and calls the `jni_updateGameBitmap` function to update the game display:

    ```
    extern void* gBitmap;
    void VW_UpdateScreen()
    {
      int i;
      int numOfPixels = vwidth*vheight;
      for(i = 0; i < numOfPixels; ++i) {
        byte colIdx = gfxbuf[i];
        ((uint32_t*)gBitmap)[i] = (0xFF << 24)
          | (pal[colIdx].r << 16)
          | (pal[colIdx].g << 8)
          | (pal[colIdx].b);
      }
      //update the screen through JNI
      jni_updateGameBitmap();
    }
    ```

    - `jni_wolf3dport.c`: This file contains the code to glue the C and Java worlds. The two key functions are `naMain`, which initializes a few things and starts the game loop, and `jni_updateGameBitmap`, which calls the `updateGameBitmap` method defined in the Java class `GameView`:

    ```
    jint JNICALL naMain(JNIEnv *pEnv, jobject pObj, jobjectArray 
    pArgv, jobject pGameViewObj) {
      (*pEnv)->GetJavaVM(pEnv, &cachedJvm);
      jclass jGameViewCls = (*pEnv)->GetObjectClass(pEnv, 
    pGameViewObj);
      cachedGameViewObj = (*pEnv)->NewGlobalRef(pEnv, 
    pGameViewObj);

    //get the bitmap object
      jfieldID bitmapFID;
      bitmapFID = (*pEnv)->GetFieldID(pEnv, jGameViewCls, 
    "mBitmap", "Landroid/graphics/Bitmap;");
      jobject pBitmap = (*pEnv)->GetObjectField(pEnv, 
    cachedGameViewObj, bitmapFID);
      AndroidBitmapInfo linfo;
      int lret;
      AndroidBitmap_getInfo(pEnv, pBitmap, &linfo);
      AndroidBitmap_lockPixels(pEnv, pBitmap, &gBitmap);

      //get the arguments
      jsize argc =  (*pEnv)->GetArrayLength(pEnv, pArgv);
    ```

```
  char** argv = (char**) malloc(sizeof(char*)*argc);
  int i;
  for (i = 0; i < argc; i++) {
     jstring anArgv = (jstring)(*pEnv)-
>GetObjectArrayElement(pEnv, pArgv, i);
     const char *anArgvChars  = (*pEnv)-
>GetStringUTFChars(pEnv, anArgv, 0);
     argv[i] = malloc(strlen(anArgvChars) + 1);
     strcpy(argv[i], anArgvChars);
     (*pEnv)->ReleaseStringUTFChars(pEnv, anArgv,
anArgvChars);
  }
  updateGameBitmapMID = (*pEnv)->GetMethodID(pEnv,
jGameViewCls, "updateGameBitmap", "()V");

  //start the game
  WolfMain(argc, argv);
  AndroidBitmap_unlockPixels(pEnv, pBitmap);
  (*pEnv)->DeleteGlobalRef(pEnv, cachedGameViewObj);
  cachedGameViewObj = NULL;
  return 0;
}
//call Java method updateGameBitmap through JNI
void jni_updateGameBitmap() {
  JNIEnv *pEnv;
  (*cachedJvm)->AttachCurrentThread(cachedJvm, &pEnv, NULL);
  (*pEnv)->CallVoidMethod(pEnv, cachedGameViewObj,
updateGameBitmapMID);
}
```

- `jni_wolf3dport.h`: It's the header file for `jni_wolf3dport.c`.
- `mylog.h`: It defines some logging utilities.

6. Add an `Android.mk` file under `jni` to build the native shared library:

```
LOCAL_PATH := $(call my-dir)
include $(CLEAR_VARS)
LOCAL_MODULE:= libwolf3dport
LOCAL_CFLAGS := -DANDROID
LOCAL_SRC_FILES := objs.c misc.c id_ca.c id_vh.c id_us.c \
wl_act1.c wl_act2.c wl_act3.c \
wl_agent.c wl_game.c wl_inter.c wl_menu.c \
wl_play.c wl_state.c wl_text.c wl_main.c \
wl_debug.c vi_comm.c sd_comm.c wl_draw.c \
jni_wolf3dport.c vi_android.c sd_null.c
LOCAL_LDLIBS := -llog -ljnigraphics
include $(BUILD_SHARED_LIBRARY)
```

7. Add three Java files under the `cookbook.chapter11.wolf3dport` package:
   - `GameUtils.java`: It contains the functions to install the game data files in assets to the `/data/data/cookbook.chapter11.wolf3dport/gamedata/` folder.
   - `GameView.java`: It extends a SurfaceView that allows the game content to be drawn on its canvas. The key method is `updateGameBitmap`, which should be called by the native code to draw the updated bitmap to the canvas:

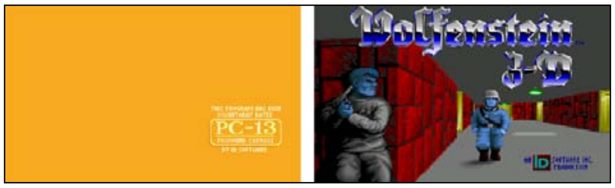
```
public void updateGameBitmap() {
  if(null != surfaceHolder && surfaceHolder.getSurface().
isValid()){
      Canvas canvas = surfaceHolder.lockCanvas();
      if (null != mBitmap) {
        canvas.drawBitmap(mBitmap, 0, 0, prFramePaint);
      }
      surfaceHolder.unlockCanvasAndPost(canvas);
  }
}
```

   - `MainActivity.java`: It installs the game data files, initializes the game display, and starts a thread to run the `naMain` native method.

8. Add a layout file `activity_main.xml` under the `res/layout` folder:

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/
android"
   xmlns:tools="http://schemas.android.com/tools"
   android:layout_width="match_parent"
   android:layout_height="match_parent"
   tools:context=".MainActivity" >
   <cookbook.chapter11.wolf3dport.GameView
       android:id="@+id/gameView"
       android:layout_width="match_parent"
       android:layout_height="match_parent"
       android:layout_alignParentBottom="true"
       android:layout_alignParentTop="true" />
</RelativeLayout>
```

9. Build the project and run it on an Android device or emulator. The Wolf3D game should be displayed as follows:



How it works...

The project shows how to use the Android SurfaceView for the Wolf3D game display. The technique is similar to the one used in the *Decoding and displaying the video frame* recipe in *Bonus chapter 1*, *Developing Multimedia Applications with NDK*.

The `GameView.java` class provides a dedicated drawing surface and a bitmap. In `MainActivity.java`, we call the native method `naMain` with parameters to specify the display dimension, game data files directory, and so on. We also pass the `GameView` object to the method.

In the `naMain` method in `jni_wolf3dport.c`, we lock the pixels of the bitmap so that the pixels won't be moved by the VM and we can manipulate the pixel buffer directly.

Every time the game display is updated, the `VW_UpdateScreen` function in `vi_android.c` will be called. Wolf3D uses a color palette `pal` with an array of indices to the color palette stored in `gfxbuf`. Each element of the `gfxbuf` array points to a color defined at the color palette `pal`. In the `VW_UpdateScreen` function, we obtain the RGB values for every pixel by reading the color values from `pal` and store the values to the bitmap. The `jni_updateGameBitmap` function is finally called, which subsequently invokes the Java method `updateGameBitmap` to draw the bitmap on the canvas provided by `GameView` and update the display.

Adding a game control on Android

The previous recipe allows us to view the game display on the phone screen, but we cannot play without the game control. This recipe illustrates adding a poor man's game control to the Wolf3D on Android.

How to do it...

The following steps describe how to add a simple game control for Wolf3D on Android:

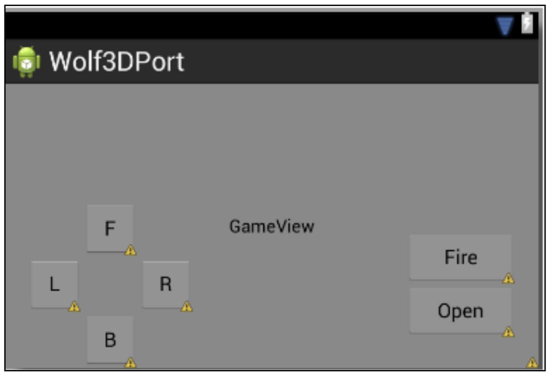
1. If you haven't gone through the previous recipe, please do so now. After that, copy or rename the `Wolf3DPort` project folder to `Wolf3DPortStep2`. Open the project in Eclipse.
2. Update `jni_wolf3dport.c` under the `jni` folder to add the `keypress` and `keyrelease` functions:

```
extern void keyboard_handler(int code, int press);
jint keyPress(JNIEnv * pEnv, jobject pObj, jint scanCode) {
  keyboard_handler(scanCode, 1);
  return scanCode;
}
jint keyRelease(JNIEnv * env, jobject pObj, jint scanCode) {
  keyboard_handler(scanCode, 0);
  return scanCode;
}
```

3. Under `cookbook.chapter11.wolf3dport` package, add a new Java file `ScanCodes.java`, which defines the scan code for keyboard keys used in the Wolf3D game. The class is created by referring to the `vi_comm.h` file under the `jni` folder:

```
public class ScanCodes {
  public static final int sc_Escape = 0x01;
  public static final int sc_UpArrow = 0x48;
  public static final int sc_DownArrow = 0x50;
  public static final int sc_LeftArrow = 0x4b;
  public static final int sc_RightArrow = 0x4d;
  public static final int sc_Control = 0x1d;
  public static final int sc_Space = 0x39;
  public static final int sc_LShift = 0x2a;
  public static final int sc_Return = 0x1c;
  public static final int sc_Y = 0x15;
  public static final int sc_N = 0x31;
}
```

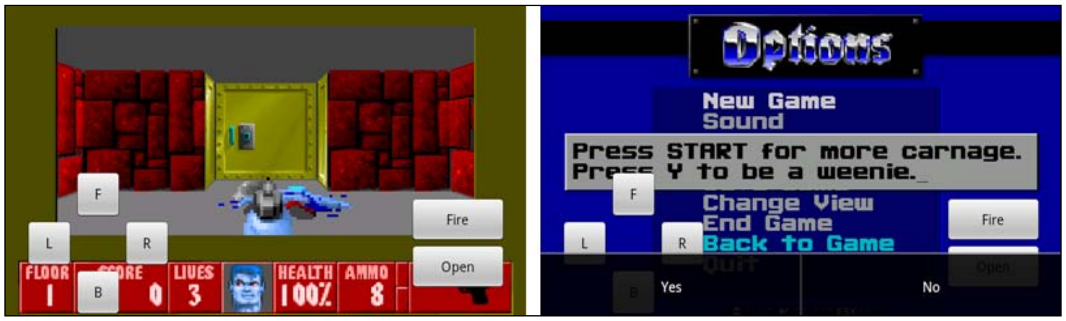
4. Update the layout file `activity_main.xml` under the `res/layout` folder to get the following graphical layout. The content of the file can be referred from the source code downloadable from the book's website:



5. Update `MainActivity.java` to set up touch listeners for the buttons. In addition, we set up two option menu items to answer "Yes" or "No" when there's a pop-up dialog box, and event handlers for hard key press. Let's see the code that sets up touch listeners for the **Fire** button:

```
Button btnFire = (Button) findViewById(R.id.buttonFire);
btnFire.setOnTouchListener(new View.OnTouchListener() {
  @Override
  public boolean onTouch(View v, MotionEvent event) {
    int action = event.getAction();
    if (action == MotionEvent.ACTION_DOWN) {
      keyPress(ScanCodes.sc_Control);
    } else if (action == MotionEvent.ACTION_UP) {
      keyRelease(ScanCodes.sc_Control);
    }
    return false;
  }
});
```

6. Build the project and run it on an Android device or emulator. We can now play the Wolf3D game on Android:



How it works...

This recipe demonstrates how to add a poor man's game control to the Wolf3D game.

There are three categories of controls we have implemented:

- **On-screen buttons**: The navigation buttons **F**, **B**, **L**, and **R** are used to control the game movement. The **Fire** button triggers a gun fire and the **Open** button opens a door.
- **Option menu**: The two option menu items "**Yes**" and "**No**" are used to answer a pop-up dialog box.
- **Hard key**: We only implemented the back hard key control, which will bring the game to its previous screen.

The technique we used is that we sent the key press and key release events to the native methods `keyPress` and `keyRelease`, to achieve a proper response. For example, when the **Fire** button is pressed, we send the keyboard control key press event to native code.

Note that the control we implemented needs lots of polish and improvement. The button should be translucent; only a limited number of key events are simulated, and so on. However, it is simple and enough to illustrate the idea.

Handling audio with MediaPlayer and SoundPool through JNI

We can play the game with the simple game control added in the previous recipe. However, there's still one important piece missing from the video game—audio. This recipe adds audio to the Wolf3D game on Android. We will use MediaPlayer for background music and SoundPool for short audio effects, such as gun fire.

Getting ready

Readers are expected to know how MediaPlayer and SoundPool work. You can refer to the Android documentations at `http://developer.android.com/reference/android/media/MediaPlayer.html` and `http://developer.android.com/reference/android/media/SoundPool.html`.

How to do it...

The following steps describe how to add audio for Wolf3D on Android:

1. If you haven't gone through the previous recipe, please do so now. After that, copy or rename the `Wolf3DPortStep2` project folder to `Wolf3DPortStep3`. Open the project in Eclipse.
2. Download the game audio files from `http://www.wolfenstein3d.co.uk/music.htm`. We only use two audio files in this sample project. `gunfire.wav` is renamed from the file at `http://www.wolfenstein3d.co.uk/gunjm.zip` and `search.mp3` is renamed from the file at `http://www.mediafire.com/?3r6durepb206lps`. Add `gunfire.wav` under the `assets` folder and `search.mp3` under the `res/raw` folder.
3. Update `jni_wolf3dport.c` under the `jni` folder to add the `jni_playSound` and `jni_playMusic` functions. These two functions invoke the `playSound` and `playMusic` methods in `MainActivity.java` to play sound effects and background music respectively:

```
...
playSoundMID = (*pEnv)->GetMethodID(pEnv, jMainActCls,
"playSound", "(I)V");
playMusicMID = (*pEnv)->GetMethodID(pEnv, jMainActCls,
"playMusic", "(I)V");
WolfMain(argc, argv);
```

```
...
void jni_playSound(int idx) {
  JNIEnv *pEnv;
  (*cachedJvm)->AttachCurrentThread(cachedJvm, &pEnv, NULL);
  (*pEnv)->CallVoidMethod(pEnv, cachedMainActObj, playSoundMID, 
idx);
}
void jni_playMusic(int idx) {
  JNIEnv *pEnv;
  (*cachedJvm)->AttachCurrentThread(cachedJvm, &pEnv, NULL);
  (*pEnv)->CallVoidMethod(pEnv, cachedMainActObj, playMusicMID, 
idx);
}
```

4. Add a new file `sd_android.c` under the `jni` folder and copy the content of `sd_null.c` to `sd_android.c`. Update the `SD_PlaySoundWL6` and `SD_StartMusic` functions to invoke `jni_playSound` and `jni_playMusic`.

```
boolean SD_PlaySoundWL6(soundnamesWL6 sound)
{
  jni_playSound(sound);
  return true;
}

void SD_StartMusic(int music)
{
  jni_playMusic(music);
  SD_MusicOff();
}
```

5. Under `src`, add a new Java package `cookbook.chapter11.wolf3dport.audio`. Add the following Java files under the package:
   - `Audiowl6.java`: It defines the sound names and music resource IDs.
   - `GameMusic.java`: It controls the playback of background music:

```
public class GameMusic implements OnCompletionListener {
   private static final String TAG = "GameMusic";
   private MediaPlayer mediaPlayer;
   private boolean isPrepared = false;
   public GameMusic(Context mContext, int resId) {
       try {
         mediaPlayer = MediaPlayer.create(mContext, resId);
           isPrepared = true;
           mediaPlayer.setOnCompletionListener(this);
       } catch (Exception e) {
           e.printStackTrace();
```

```
            }
        }
        public void destroy() {
            if (mediaPlayer.isPlaying()) {
                mediaPlayer.stop();
            }
            mediaPlayer.release();
        }
        public void play() {
            if (mediaPlayer.isPlaying()) {
                return;
            }
            try {
                synchronized (this) {
                    if (!isPrepared) {
                        mediaPlayer.prepare();
                    }
                    mediaPlayer.start();
                }
            } catch (IllegalStateException e) {
                e.printStackTrace();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
        public void pause() {
            if (mediaPlayer.isPlaying()) {
                mediaPlayer.pause();
            }
        }
        ...
    }
```

- `GameSound.java`: It controls the playback of sound effects:

```
public class GameSound {
   private int soundId;
   private SoundPool soundPool;

   public GameSound(SoundPool pSoundPool, int pSoundId) {
       soundId = pSoundId;
       soundPool = pSoundPool;
   }
   public void play(float volume) {
       soundPool.play(soundId, volume, volume, 0, 0, 1);
```

```
    }
    public void unload() {
        soundPool.unload(soundId);
    }
}
```

- `MyAudioManager.java`: It manages the background music and sound effects playback:

```
public synchronized void playSound(int soundIdx) {
  if (0 == Audiowl6.SOUNDS[soundIdx].compareTo("")) {
    return;
  } else {
    if (preloadSounds.containsKey(Audiowl6.
SOUNDS[soundIdx])) {
      preloadSounds.get(Audiowl6.SOUNDS[soundIdx]).play(50);
    } else {
      AssetFileDescriptor assetDescriptor;
      try {
        assetDescriptor = assetMgr.openFd(Audiowl6.
SOUNDS[soundIdx]);
        int soundId = mSoundPool.load(assetDescriptor, 0);
        GameSound newSound = new GameSound(mSoundPool,
soundId);
        preloadSounds.put(Audiowl6.SOUNDS[soundIdx],
newSound);
        newSound.play(50);
      } catch (IOException e) {
        e.printStackTrace();
      }
    }
  }
}
public synchronized void playMusic(int musicIdx) {
  if (-1 == Audiowl6.MUSICS[musicIdx]) {
    return;
  } else {
    AssetFileDescriptor assetDescriptor;
    if (null != mMusic) {
      mMusic.destroy();
    }
    mMusic = new GameMusic(mContext, Audiowl6.
MUSICS[musicIdx]);
    mMusic.setVolume(50);
    mMusic.setLooping(true);
    mMusic.play();
  }
}
```

6. Update `MainActivity.java` to create an instance of `MyAudioManager` and implement the `playSound` and `playMusic` methods:

```
private MyAudioManager mAudioManager;
@Override
protected void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  …
  mAudioManager = new MyAudioManager(this.
getApplicationContext());
  …
}
@Override
protected void onResume() {
  super.onResume();
     mAudioManager.resumeMusic();
}
@Override
protected void onPause() {
  super.onPause();
     mAudioManager.pauseMusic();
}
@Override
protected void onDestroy() {
  super.onDestroy();
  mAudioManager.unloadSounds();
  mAudioManager.stopMusic();
  System.exit(0);  //to kill the bg thread
}
private void playSound(int soundIdx) {
  mAudioManager.playSound(soundIdx);
}
private void playMusic(int musicIdx) {
     mAudioManager.playMusic(musicIdx);
}
```

7. Build the project and run it on an Android device or emulator. We can now play the Wolf3D game on Android with gunfire effects and background music.

How it works...

This recipe goes through the steps to enable audio effects in the Wolf3D game on Android.

The native code is straightforward. Every time a sound effect is played, the `SD_PlaySoundWL6` function in `sd_android.c` is called. In the function, we call `jni_playSound` to invoke the Java method through JNI, which does the actual playback. When you want to play some background music, call the `SD_StartMusic` function in `sd_android.c`. In the function, we call `jni_playMusic` to execute the Java method through JNI, which plays the music using MediaPlayer.

Sound effects are usually short audio clips, and the Android `SoundPool` class is designed for this. `SoundPool` is able to keep the decompressed audio in memory for quick access. This is implemented in `GameSound.java`. Note that in `MyAudioManager.java`, we preload audio effects in the `preloadSounds` method.

Background music is usually longer and the Android `MediaPlayer` class is good at handling such audio files. We can easily play, pause, and reset background music with `MediaPlayer`. This is implemented in `GameMusic.java`.

We manage both sound effects and background music through the `MyAudioManager` instance created in `MainActivity.java`.