Developing Multimedia Applications with NDK

In this chapter, we will cover the following recipes:

- Porting the ffmpeg library to Android with NDK
- Using the ffmpeg library to get media info
- Decoding and displaying the video frame
- Separating decoding and drawing with two threads
- Seeking to playback and grabbing the frames
- Optimizing the performance of multimedia apps

Introduction

Many multimedia Android applications use NDK, especially video applications. If you go through the top 50 apps under the Media and Video category, you will easily find a few apps using NDK, including MX Player, VLC for Android Beta, PPS, PPTV, and so on.

There are a few advantages for using NDK in multimedia apps. Firstly, multimedia apps, and video apps in particular, are usually CPU-intensive. Developing in native languages offers the potential to achieve better performance. Secondly, NDK allows us to detect CPU features and you can take advantage of the CPU features offered by a particular device. For example, NEON is a general-purpose **Single Instruction Multiple Data** (**SIMD**) engine available on many ARM CPUs. It can speed up some media processing tasks (for example, color conversion) several times. You can develop your apps in such a way that if NEON is available, it will use the NEON-specific code. Thirdly, multimedia apps usually require manipulation of bits and bytes, moving chunks of memory around, and so on. Native languages such as C/C++ are better at those tasks than Java. Last but not least, Android only offers support for a few video/audio codecs. There are a lot of open source codec libraries available and most of them are in native languages. You can port them to Android using NDK to enable support for additional codecs.

This chapter will discuss how to develop multimedia apps in Android NDK. As multimedia apps are usually complicated, you can only cover the development of a relatively simple frame grabber application in the chapter. But, it should be a good starting point.

The frame grabber app you're going to develop will be able to play the video frames from a video file and save the frames being displayed to pictures. It is based on the well-known `ffmpeg` library. We will first port the `ffmpeg` library to Android with NDK. After that, you will introduce the basics of reading a video file and initializing the codec context. The next step is to decode, scale, and display the video frame. We will see how to do this with the Android `SurfaceView` class, `bitmap`, and `ffmpeg`. We will also illustrate dividing the decoding, scale, and display tasks into two threads, performing random seek at playback, and saving the video frame to pictures. The last recipe will discuss the techniques to optimize the performance.

The frame grabber app uses the `ActionBarSherlock` library for the GUI. You will link to the library at all your sample projects in this chapter. The `ActionBarSherlock` library is available at `http://actionbarsherlock.com/`. The detailed instruction is covered in the second recipe.

Porting the ffmpeg library to Android with NDK

The `ffmpeg` library is not available on Android by default. You will need to port the `ffmpeg` library in order to use it.

Getting ready

This recipe uses many of the techniques covered *Chapter 8*, *Porting and Using Existing Libraries with Android NDK*. The following recipes should be reviewed first in particular:

- *Porting a library with its existing build system*
- *Using a library in multiple projects with import-module*

How to do it...

The following steps describe how to build a `ffmpeg` library for Android with NDK:

1. Go to `http://ffmpeg.org/download.html`, and download the ffmpeg source code. In this example, you used ffmpeg 1.0.1 release. Extract the downloaded archive to the sources folder of our Android NDK directory. The extracted files will be put into the `sources/ffmpeg-1.0.1` folder.

Note that the ffmpeg library is updated frequently. Hoyouver, the techniques and scripts mentioned in this recipe should work on your versions of ffmpeg. You should try it out on the latest stable release of the library.

2. Add a `build_android.sh` script under the `sources/ffmpeg-1.0.1` folder to compile the ffmpeg libraries. Note that you need to update the NDK variable in the script to point to the correct location in our computer.

```
#!/bin/bash
NDK=/home/roman10/Desktop/android/android-ndk-r8b
SYSROOT=$NDK/platforms/android-8/arch-arm/
TOOLCHAIN=$NDK/toolchains/arm-linux-androideabi-4.6/prebuilt/
linux-x86
function build_one
{
./configure \
   --prefix=$PREFIX \
   --disable-shared \
   --enable-static \
   --disable-doc \
   --disable-ffmpeg \
   --disable-ffplay \
   --disable-ffprobe \
   --disable-ffserver \
   --disable-avdevice \
   --disable-doc \
   --disable-symver \
   --cross-prefix=$TOOLCHAIN/bin/arm-linux-androideabi- \
   --target-os=linux \
   --arch=arm \
   --enable-cross-compile \
   --sysroot=$SYSROOT \
   --extra-cflags="-Os -fpic $ADDI_CFLAGS" \
   --extra-ldflags="$ADDI_LDFLAGS" \
   $ADDITIONAL_CONFIGURE_FLAG
make clean
make
make install
}
CPU=arm
PREFIX=$(pwd)/android/$CPU
ADDI_CFLAGS="-marm"
build_one
```

3. Start a command line shell, go to the `sources/ffmpeg-1.0.1` folder, and enter the following command to build ffmpeg:

   ```
   $ chmod +x build_android.sh
   $ ./build_android.sh
   ```

   The build will take quite a while to finish. After it is done, you can find the library headers under the `sources/ffmpeg-1.0.1/android/arm/include` directory, and library files under the `sources/ffmpeg-1.0.1/android/arm/lib` directory.

4. Add an Android.mk file under the `sources/ffmpeg-1.0.1/android/arm` folder:

   ```
   LOCAL_PATH:= $(call my-dir)

   #static version of libavcodec
   include $(CLEAR_VARS)
   LOCAL_MODULE:= libavcodec_static
   LOCAL_SRC_FILES:= lib/libavcodec.a
   LOCAL_EXPORT_C_INCLUDES := $(LOCAL_PATH)/include
   include $(PREBUILT_STATIC_LIBRARY)

   #static version of libavformat
   include $(CLEAR_VARS)
   LOCAL_MODULE:= libavformat_static
   LOCAL_SRC_FILES:= lib/libavformat.a
   LOCAL_EXPORT_C_INCLUDES := $(LOCAL_PATH)/include
   include $(PREBUILT_STATIC_LIBRARY)

   #static version of libswscale
   include $(CLEAR_VARS)
   LOCAL_MODULE:= libswscale_static
   LOCAL_SRC_FILES:= lib/libswscale.a
   LOCAL_EXPORT_C_INCLUDES := $(LOCAL_PATH)/include
   include $(PREBUILT_STATIC_LIBRARY)

   #static version of libavutil
   include $(CLEAR_VARS)
   LOCAL_MODULE:= libavutil_static
   LOCAL_SRC_FILES:= lib/libavutil.a
   LOCAL_EXPORT_C_INCLUDES := $(LOCAL_PATH)/include
   include $(PREBUILT_STATIC_LIBRARY)
   ```

How it works...

The previous steps show how to build ffmpeg for Android and prepare it to be used by Android applications.

The idea is to build the ffmpeg library and allow multiple projects to link to it as prebuilt libraries with import-module. You put the ffmpeg library source code under the sources folder of the Android NDK directory. As discussed in the *Using a library in multiple projects with import-module* recipe in *Chapter 8*, *Porting and Using the Existing Libraries with Android NDK*, the sources directory is appended to NDK_MODULE_PATH by NDK build system.

The build script at the second step cross compiles ffmpeg library using the NDK toolchain directly. The ffmpeg build scripts accept lots of configuration options for controlling the build process. Our script enables static libraries and disables a few components you don't want. You can execute the `./configure --help` command under the `ffmpeg-1.0.1` directory to view all configuration options and fine-tune the build process.

The compilation will produce library files and headers for six ffmpeg libraries, including libavcodec, libavfilter, libavformat, libavutil, libswresample and libswscale.

After compilation is done, you provide an Android.mk file in step 4 for an Android application project to easily import the ffmpeg libraries as static prebuilt library modules. You will see how to use the ffmpeg libraries in subsequent recipes.

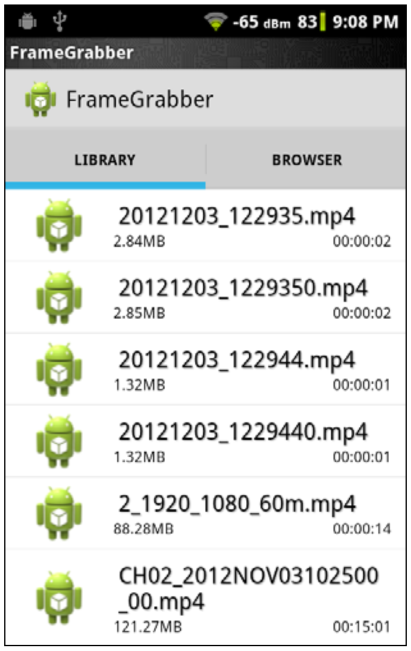
Using the ffmpeg library to get media info

Starting from this recipe, you will show how to use the ffmpeg libraries to develop multimedia applications. This recipe will discuss how to obtain various information of a video file.

Getting ready

This recipe and subsequent recipes are based on a project template available from the book's youbsite. Please download the source code for this book and locate the `FrameGrabberTemplate` folder in the folder `Bonus chapter 1`.

The folder contains an Eclipse Android project which implements a GUI, shown as follows:



The code loads the video files from Android media store database. This GUI is used throughout this chapter. Since it is written in pure Java code, the implementation of this GUI is out of the scope of this book and won't be discussed. However, you should at least skim the source code.

Before you start coding with the ffmpeg library, it is essential to understand some basics of video.

A video file is also called a container. The most commonly seen video container formats on Android are 3GPP (.3gp) and MPEG4 (.mp4). A video file can contain multiple media streams, including a video stream and audio stream. Each media stream is encoded by a video encoder and can be decoded by the corresponding decoder. The encoder and decoder pair is referred to as the codec. The most commonly used video codecs on Android include H.264 and MPEG-4 SP. For a detailed list of container formats and codecs supported by Android, you can refer to `http://developer.android.com/guide/appendix/media-formats.html`.

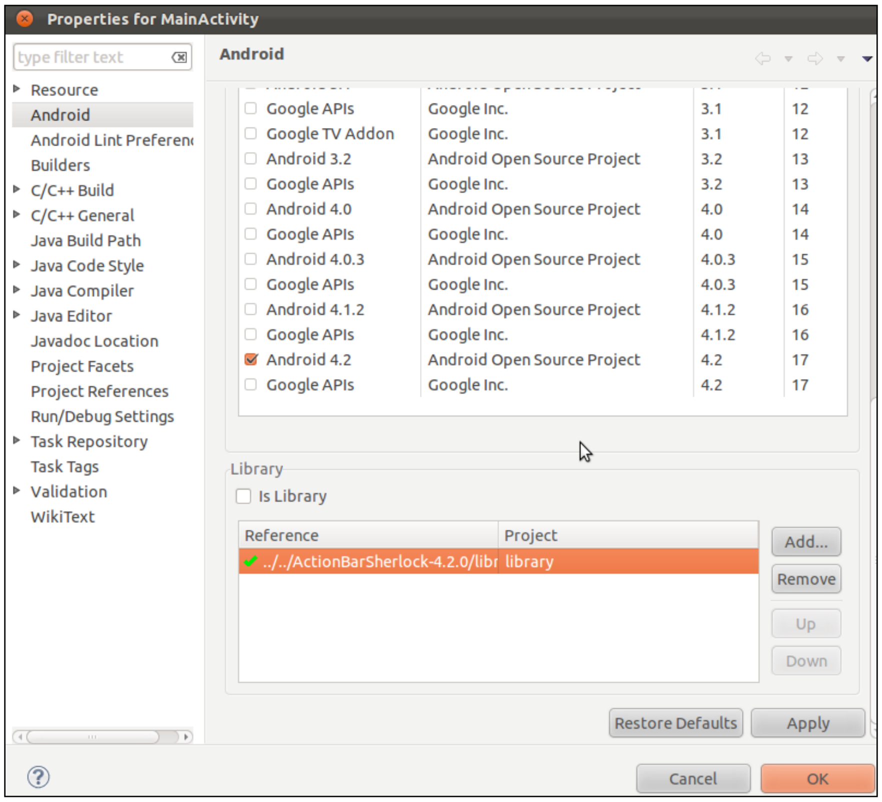
A video sequence consists of a Group of Pictures (GOP). Each GOP has a few frames. There are three types of frames, including the I-frame, P-frame, and B-frame. An I-frame can be decoded without referencing to other frames, a P-frame depends on past frames, and a B-frame depends on both past frames and future frames.

Note that you must port the ffmpeg libraries by following the instructions in the previous recipe in order to build the application in this and subsequent recipes.

How to do it...

The following steps describe how to create an Android project that uses the `ffmpeg` library to get various information of a video file:

1. Create a copy of the `FrameGrabberTemplate` folder and rename it as `FrameGrabberGetMediaInfo`. Open the copied project in Eclipse using **File** | **Import** | **Existing Projects into Workspace**.
2. Download the `ActionBarSherlock` library from `http://actionbarsherlock.com/` and extract the archive file. At the time of writing, the latest version of the library is 4.2.0, therefore the uncompressed folder name is `ActionBarSherlock-4.2.0`.
3. In Eclipse, open the library project using **File** | **Import** | **Existing Android Code Into Workspace**. Then, browse to the `ActionBarSherlock-4.2.0/library` folder to import the project.
4. Right-click on the `FrameGrabberGetMediaInfo` project and click on **Properties**. Select **Android** in the left pane, then click on **Add** at the bottom part of the right pane. Select the library project. Click on **OK** to finish:



Google APIs	Google Inc.	3.1	12
Google TV Addon	Google Inc.	3.1	12
Android 3.2	Android Open Source Project	3.2	13
Google APIs	Google Inc.	3.2	13
Android 4.0	Android Open Source Project	4.0	14
Google APIs	Google Inc.	4.0	14
Android 4.0.3	Android Open Source Project	4.0.3	15
Google APIs	Google Inc.	4.0.3	15
Android 4.1.2	Android Open Source Project	4.1.2	16
Google APIs	Google Inc.	4.1.2	16
✓ Android 4.2	Android Open Source Project	4.2	17
Google APIs	Google Inc.	4.2	17

Library

Is Library

Reference	Project
✓ ../../ActionBarSherlock-4.2.0/libr	library

5. Create a `jni` folder under the `FrameGrabberGetMediaInfo` project. Add a source file `framegrabber.c` under the `jni` folder. This is shown in the following code snippet. Note that the error checking code in the original source code is removed for simplicity:

   - `VideoState`: It's a data structure that contains the video format and codec data structure exposed by the `ffmpeg` libraries:

     ```
     typedef struct VideoState {
       AVFormatContext *pFormatCtx;
       AVStream *pVideoStream;
       int videoStreamIdx;
     }VideoState;
     VideoState *gvs;
     ```

   - `naInit`: It initializes the context for an input video:

     ```
     int naInit(JNIEnv *pEnv, jobject pObj, jstring pfilename) {
       gVideoFileName = (char *)(*pEnv)->GetStringUTFChars(pEnv,
     pfilename, NULL);
       av_register_all();
       VideoState *vs;
       vs = av_mallocz(sizeof(VideoState));
       gvs = vs;
       av_register_all();
       //open the video file
       avformat_open_input(&vs->pFormatCtx, gVideoFileName, NULL,
     NULL);
       //retrieve stream info
       avformat_find_stream_info(vs->pFormatCtx, NULL);
       //find the video stream
       int i;
       AVCodecContext *pcodecctx;
       //find the first video stream
       vs->videoStreamIdx = -1;
       for (i = 0; i < vs->pFormatCtx->nb_streams; ++i) {
         if (AVMEDIA_TYPE_VIDEO == vs->pFormatCtx->streams[i]-
     >codec->codec_type) {
           vs->videoStreamIdx = i;
           vs->pVideoStream = vs->pFormatCtx->streams[i];
           break;
         }
       }
       //get the decoder from the video stream
       pcodecctx = vs->pFormatCtx->streams[vs->videoStreamIdx]-
     >codec;
       AVCodec *pcodec;
     ```

```
  pcodec = avcodec_find_decoder(pcodecctx->codec_id);
  //open the codec
  avcodec_open2(pcodecctx, pcodec, NULL);
  return 0;
}
```

- `naGetVideoRes`, `naGetDuration` and `naGetFrameRate`: It gets the video resolution, duration in seconds, and frame rate in frames per second:

```
jintArray naGetVideoRes(JNIEnv *pEnv, jobject pObj) {
   jintArray lRes;
  AVCodecContext* vCodecCtx = gvs->pVideoStream->codec;
  lRes = (*pEnv)->NewIntArray(pEnv, 2);
  jint lVideoRes[2];
  lVideoRes[0] = vCodecCtx->width;
  lVideoRes[1] = vCodecCtx->height;
  (*pEnv)->SetIntArrayRegion(pEnv, lRes, 0, 2, lVideoRes);
  return lRes;
}

jint naGetDuration(JNIEnv *pEnv, jobject pObj) {
  return (gvs->pFormatCtx->duration / AV_TIME_BASE);
}

jint naGetFrameRate(JNIEnv *pEnv, jobject pObj) {
  int fr;
  VideoState *vs = gvs;
  if(vs->pVideoStream->avg_frame_rate.den && vs-
>pVideoStream->avg_frame_rate.num) {
    fr = av_q2d(vs->pVideoStream->avg_frame_rate);
  } else if(vs->pVideoStream->r_frame_rate.den && vs-
>pVideoStream->r_frame_rate.num) {
    fr = av_q2d(vs->pVideoStream->r_frame_rate);
  } else if(vs->pVideoStream->time_base.den && vs-
>pVideoStream->time_base.num) {
    fr = 1/av_q2d(vs->pVideoStream->time_base);
  } else if(vs->pVideoStream->codec->time_base.den && vs-
>pVideoStream->codec->time_base.num) {
    fr = 1/av_q2d(vs->pVideoStream->codec->time_base);
  }
  return fr;
}
```

- `naFinish`: It cleans the resource and closes the file:

  ```
  int naFinish(JNIEnv *pEnv, jobject pObj) {
    VideoState* vs = gvs;
    //close codec
    avcodec_close(vs->pVideoStream->codec);
    //close video file
    avformat_close_input(&vs->pFormatCtx);
    av_free(vs);
    return 0;
  }
  ```

6. Add an `Android.mk` file with the following content:

   ```
   LOCAL_PATH := $(call my-dir)
   include $(CLEAR_VARS)
   LOCAL_MODULE    := framegrabber
   LOCAL_SRC_FILES := framegrabber.c
   LOCAL_LDLIBS := -llog -lz
   LOCAL_STATIC_LIBRARIES := libavformat_static libavcodec_static
   libavutil_static
   include $(BUILD_SHARED_LIBRARY)
   $(call import-module,ffmpeg-1.0.1/android/arm)
   ```

7. Open a command line shell, go to the `jni` folder and enter the following command to build the native code:

   ```
   $ ndk-build
   ```

   When the build is done, there should be a `libframegrabber.so` file under the `libs/armeabi` folder.

8. In `VideoLibraryFragment.java`, add the following code to define the native methods and load the native library:

   ```
   private static native int naInit(String pVideoFileName);
   private static native int[] naGetVideoRes();
   private static native int naFinish();
   private static native int naGetDuration();
   private static native int naGetFrameRate();
   static {
          System.loadLibrary("framegrabber");
   }
   ```

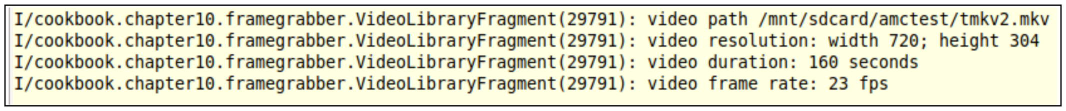
9. Append the following code to the `onListItemClick` method to obtain the video information of a clicked video file and print the information to logcat:

```
int st = naInit(filePath);
if (0 > st) {
     Log.i(TAG, "error open video file");
} else {
    int[] res = naGetVideoRes();  //width, height
    Log.i(TAG, "video resolution: width " + res[0] + "; height " +
res[1]);
    Log.i(TAG, "video duration: " + naGetDuration() + " seconds");
    Log.i(TAG, "video frame rate: " + naGetFrameRate() + " fps");
    naFinish();
}
```

10. Build the Android application and start it on a device. In addition, open a command-line shell and enter the following command to monitor the logcat output:

```
$ adb logcat cookbook.chapter10.framegrabber.
VideoLibraryFragment:I *:S -v time
```

11. Click on any of the loaded video files; you should see the resolution, duration, and frame rate in the command-line shell output. The following is a sample screenshot:



```
I/cookbook.chapter10.framegrabber.VideoLibraryFragment(29791): video path /mnt/sdcard/amctest/tmkv2.mkv
I/cookbook.chapter10.framegrabber.VideoLibraryFragment(29791): video resolution: width 720; height 304
I/cookbook.chapter10.framegrabber.VideoLibraryFragment(29791): video duration: 160 seconds
I/cookbook.chapter10.framegrabber.VideoLibraryFragment(29791): video frame rate: 23 fps
```

How it works...

The sample code demonstrates how to use the `ffmpeg` APIs to open a video file, obtain information about the video stream, and close the video file.

1. **Initialize ffmpeg and prepare to get video info**: The following steps should be followed to open a video file and initialize the context before reading the video data or retrieving media information:
    1. **Register the container formats and codecs**: This is done with the function `av_register_all`. If you are sure which format and codec to use, you can also register a specific format with `av_register_input_format` and a specific codec with `avcodec_register`.
    2. **Open the file**: This is done with `avformat_open_input`. Note that if you want the function to allocate the `AVFormatContext` for yourself, you should pass a pointer to `NULL` as the first argument.

    3. **Retrieve stream information**: The `avformat_find_stream_info` function is used here. It reads some data from the media file to get stream information.
    4. **Find the media stream**: This can be done with a loop to check the codec type of each stream. In your case, you locate a video stream by checking the codec type to see if it's `AVMEDIA_TYPE_VIDEO`.
    5. **Find the decoder for the video stream**: The `avcodec_find_decoder` function is used here. The function takes a codec ID, which can be obtained from a codec context. The codec context is partially initialized at this stage.
    6. **Initialize the codec context with the decoder**: This is done with the `avcodec_open2` function.

2. **Get the video information**: Once you have opened the video file and initialized the codec context properly, you can obtain information about the video. In your example, you retrieved the video resolution from `AVCodecContext`, duration from `AVFormatContext`, and frame rate from `AVStream`. `AVFormatContext` contains information about the file and various streams. `AVStream` contains information about a particular media stream. `AVCodecContext` includes information about a particular decoder/encoder and the data it manipulates. You can refer to the `ffmpeg` documentation for more detailed information.
3. **Close the decoder and file**: Once you are done with the video file, close the video codec by using `avcodec_close` and the video file by using `avformat_close_input`. The `avcodec_close` function frees all data associated with the codec context, and the `avformat_close_input` function closes an opened `AVFormatContext` file and frees all its contents.

Decoding and displaying the video frame

This recipe discusses how to decode a video frame and display it on an Android device with `ffmpeg` and `jnigraphics`.

Getting ready

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

How to do it...

The following steps are to create an Android project that decodes a video using the `ffmpeg` functions and render it on phone screen:

1. Create a copy of the `FrameGrabberTemplate` folder and rename it as `FrameGrabberDecodeVideo`. Open the copied project in Eclipse using **File** | **Import** | **Existing Projects into Workspace**. Follow steps 2 to 4 of the *Using the ffmpeg library to get media info to link to the ActionBarSherlock library* recipe of this chapter.
2. Add two Java files, namely `VideoEditActivity.java` and `SimpleVideoSurfaceView.java` under the `cookbook.chapter10.framegrabber` package. `VideoEditActivity.java` implements a GUI that includes `SimpleVideoSurfaceView` for video frame display, a pause/play button, and a seekbar for indicating playback progress. `SimpleVideoSurfaceView.java` extends the `SurfaceView` class and spawns a child thread for calling native methods and drawing on canvas. The following is an example of the code run by the child thread:

```
public void run() {
     while (true) {
       while (2 == mStatus) {
         //pause
         SystemClock.sleep(100);
       }
       mVideoCurrentFrame = naGetVideoFrame();
       if (0 < mVideoCurrentFrame) {
         //success, redraw
          Canvas canvas = surfaceHolder.lockCanvas();
          canvas.drawBitmap(mBitmap, mDrawLeft, mDrawTop,
prFramePaint);
          surfaceHolder.unlockCanvasAndPost(canvas);
       } else {
         //failure, break
         naFinish(mBitmap);
         break;
       }
     }
   }
```

The native method `naGetVideoFrame` decodes a video frame to the bitmap. If the decoding is successful, the child thread locks the canvas and draws the decoded frame. It then unlocks the canvas, so that the screen can be updated.

3. Add the layout file `activity_video_edit.xml` under the `res/layout` folder. The layout file defines the GUI for `VideoEditActivity.java`.
4. Create a `jni` folder under the project. Add a source file `framegrabber.c` under the `jni` folder. `framegrabber.c` is updated from the `framegrabber.c` file seen in the previous recipe. Let's highlight a few changes as follows:
   - `VideoState` and `VideoDisplayUtil`: These two data structures contain the information and data structure used to decode and scale the video frame:

```
typedef struct VideoState {
   AVFormatContext *pFormatCtx;
   AVStream *pVideoStream;
   int videoStreamIdx;
   AVFrame* frame;  //to store the decoded frame
   int fint;
   int64_t nextFrameTime;//track next frame display time
   int status;
}VideoState;

typedef struct VideoDisplayUtil {
   struct SwsContext *img_resample_ctx;
   AVFrame *pFrameRGBA;
   int width, height;
   void* pBitmap;//use the bitmap as the pFrameRGBA buffer
   int frameNum;
} VideoDisplayUtil;

VideoState *gvs;
VideoDisplayUtil *gvdu;
```

   - `naPrepareDisplay`: It accepts the bitmap from the Java code, locks the pixels and sets them as the data buffer for `pFrameRGBA`, which is used to hold the frame ready to be displayed (decoded, color converted, and scaled):

```
int naPrepareDisplay(JNIEnv *pEnv, jobject pObj, jobject pBitmap,
jint width, jint height) {
   VideoState* vs = gvs;
   VideoDisplayUtil* vdu = av_mallocz(sizeof(VideoDisplayUtil));
   gvdu = vdu;
   vs->frame = avcodec_alloc_frame();
   vdu->frameNum = 0;
   vdu->width = width;
   vdu->height = height;
   vdu->pFrameRGBA = avcodec_alloc_frame();
   AndroidBitmapInfo linfo;
   int lret;
   //1. retrieve information about the bitmap
   AndroidBitmap_getInfo(pEnv, pBitmap, &linfo);
   //2. lock the pixel buffer and retrieve a pointer to it
   AndroidBitmap_lockPixels(pEnv, pBitmap, &vdu->pBitmap);
   //you use the bitmap buffer as the buffer for pFrameRGBA
   avpicture_fill((AVPicture*)vdu->pFrameRGBA, vdu->pBitmap, PIX_
FMT_RGBA, width, height);
   vdu->img_resample_ctx = sws_getContext(vs->pVideoStream->codec-
>width, vs->pVideoStream->codec->height, vs->pVideoStream->codec-
>pix_fmt, width, height, PIX_FMT_RGBA, SWS_BICUBIC, NULL, NULL,
NULL);
   vs->nextFrameTime = av_gettime() + 50*1000;    //introduce 50
milliseconds of initial delay
   return 0;
}
```

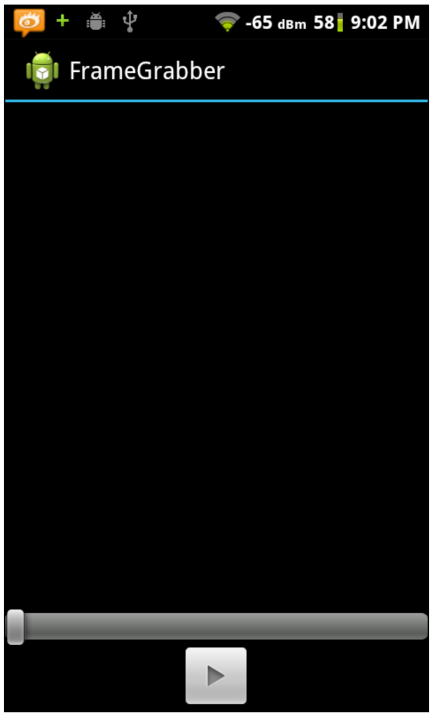
- `naGetVideoFrame`: It reads a video frame, decodes it, converts the color space to RGBA, and scales the frame:

```
int naGetVideoFrame(JNIEnv *pEnv, jobject pObj) {
  VideoState* vs = gvs;
  VideoDisplayUtil *vdu = gvdu;
  //read frames and decode them
  AVPacket packet;
  int framefinished;
  while ((!vs->status) && 0 <= av_read_frame(vs->pFormatCtx,
&packet)) {
    if (vs->videoStreamIdx == packet.stream_index) {
      avcodec_decode_video2(vs->pVideoStream->codec, vs->frame,
&framefinished, &packet);
      if (framefinished) {
          sws_scale(vdu->img_resample_ctx, vs->frame->data,
vs->frame->linesize, 0, vs->pVideoStream->codec->height, vdu-
>pFrameRGBA->data, vdu->pFrameRGBA->linesize);
        int64_t curtime = av_gettime();
        if (vs->nextFrameTime - curtime > 20*1000) {
usleep(vs->nextFrameTime-curtime);
        }
        ++vdu->frameNum;
        vs->nextFrameTime += vs->fint*1000;
        return vdu->frameNum;
      }
    }
    av_free_packet(&packet);
  }
  return 0;
}
```

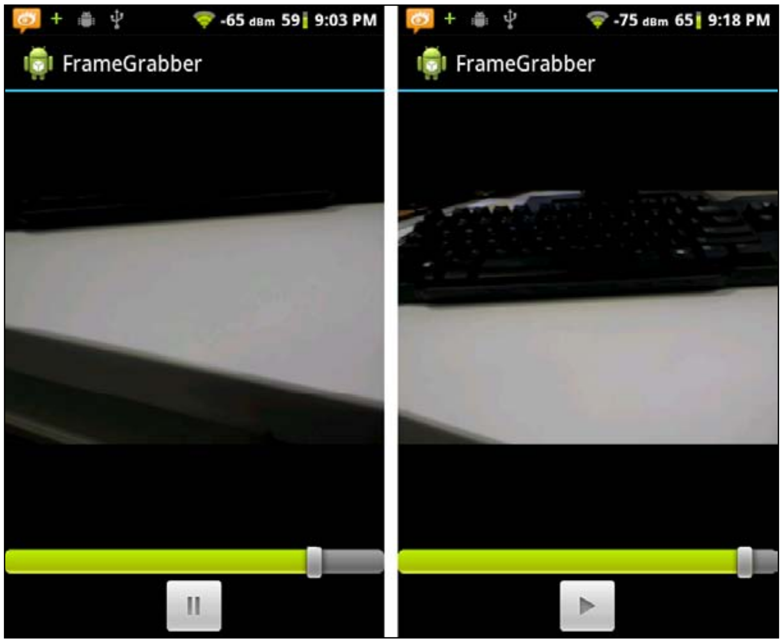
5. Add an `Android.mk` file under the `jni` folder with the following content:

```
LOCAL_PATH := $(call my-dir)
include $(CLEAR_VARS)
LOCAL_MODULE  := framegrabber
LOCAL_SRC_FILES := framegrabber.c
LOCAL_LDLIBS := -llog -ljnigraphics -lz
LOCAL_STATIC_LIBRARIES := libavformat_static libavcodec_static 
libswscale_static libavutil_static
include $(BUILD_SHARED_LIBRARY)
$(call import-module,ffmpeg-1.0.1/android/arm)
```

6. Build the Android project, and start the application on an Android device. The GUI will list the videos available in the Android media store. Click on a video to start the `VideoEditActivity` GUI, as follows:



7. Click on the play button to start playing the video and click on the button again to pause the playback. This is shown in the left and right screenshots respectively:



How it works...

The sample project shows how to read video data, decode a frame, convert the frame to an RGBA color space, scale the frame, and finally render it on the phone screen. Perform the following steps in your project:

1. **Open the video file and initialize the codec context**: This is described in detail in the previous recipe.
2. **Read a video frame from video stream**: This is done by the `av_read_frame` function. This function takes a pointer to `AVFormatContext` and a pointer to `AVPacket` as input an argument and adds the data to `AVPacket`. For the video stream, it always returns data for one video frame. The function returns zero on success.

[You should call `av_free_packet` to release the data associated with `AVPacket` once you're done with the data.]

3. **Decode the video frame**: This is done with the `avcodec_decode_video2` function. This function has the following prototype:

   ```
   int avcodec_decode_video2(AVCodecContext *avctx, AVFrame *picture,
   int *got_picture_ptr, const AVPacket *avpkt);
   ```

   It takes a pointer to the codec context, a pointer to `AVFrame`, an integer pointer, and a pointer to `AVPacket`, which has the encoded video data. When the function returns, the integer pointed by `got_picture_ptr` will set to a non-zero value if there are frames decoded, and the decoded video frame data will be saved to the `AVFrame` data structure pointed by `picture`.

4. **Convert the color space and scale the frame**: Color conversion and scaling are done by a single function `sws_scale`. However, before calling this function, you should get `SwsContext` by calling another function `sws_getContext`. The `sws_getContext` function initializes the conversion parameters, including the source color space and converted color space, the source video frame dimension and the converted video frame dimension, what sampling algorithm to use, and so on. On the other hand, the `sws_scale` function takes the actual data, does the conversion, and saves the data to a data buffer.

   In your example, you called `sws_getContext` to get `SwsContext` in the `naPrepareDisplay` function and `sws_scale` for every frame in `naGetVideoFrame`.

5. **Display the frame**: Displaying the frame is usually system dependent. On an Android platform, you can use a bitmap as the data buffer to hold the output from `sws_scale` and then draw the bitmap on a canvas.

   This is shown in our example. In the `naPrepareDisplay` function, take a bitmap object from Java code, lock it, and set it as the data buffer for `VideoState->pFrameRGBA`. Every time you call `sws_scale` in `naGetVideoFrame`, the bitmap data will be updated with the decoded, color converted and scaled video frame data. In the Java code, you can simply draw the bitmap as shown in the `run` method of `SimpleVideoSurfaceView.java`.

The functions described in this section are defined in the header files under the `sources/ffmpeg-1.0.1/ffmpeg-1.0.1/android/arm/include/` folder. The `ffmpeg` header files contain very detailed explanations of how each function should be used and are an excellent reference for us.

6. **Video playback the frame rate**: The video frames are displayed on the phone screen at a certain rate, called the **frame rate**. In our example, we adopt a simple approach to control the playback frame rate.

   We first retrieve the frame rate value encoded in the video stream at the `naInit` function. We take 1/frame rate as the display interval between two consecutive frames. A variable `nextFrameTime` is defined in the `VideoState` data structure to keep a track of the targeted display time for the next frame. If the time is not reached when you're ready to display the next frame, we wait for a while as shown in the `naGetVideoFrame` function. If you cannot decode fast enough to meet the targeted time, then we display the frame immediately.

   Note that this is not how the video playback frame rate is controlled typically. As you're only dealing with video data for your final frame grabber application, the video/audio synchronization is not an issue for us. But, a real video player cannot ignore this and must use more advanced techniques for frame rate control and audio-video synchronization. Interested readers can read more about **Presentation Time Stamp** (**PTS**) and **Decoding Time Stamp** (**DTS**) and refer to the popular *How to Write a Video Player in Less Than 1000 Lines* tutorial at `http://dranger.com/ffmpeg/`.

7. **Video playback the progress update**: We used a seek bar with a maximum progress value of 1000 for a progress update in our example.

   At the initialization, we call the `naGetDuration` and `naGetFrameRate` native methods to get the video length in seconds and video frame rate in frames per second. You can get an estimated total number of frames in the video by multiplying the two values.

   Every time you want to display a frame, call the `naGetVideoFrame` native method, which returns the current frame number on success. In this way, you can track what frame is displayed easily.

   Once you know the frame number being displayed, and total number of frames, you can easily scale the value to the range of `[0, 1000]` and set the seek bar progress value. In the Java code, `SimpleVideoSurfaceView.java` keeps track of the frame numbers and `VideoEditActivity.java` updates the seek bar progress with a handler.

8. **Pause the playback**: Pausing the playback can be easily added in your design. You're using a single thread to decode, scale, and draw the frame continuously, therefore you can simply set a flag to put the thread to sleep when you want to pause it. This is done in `SimpleVideoSurfaceView.java` by setting `mStatus` to `2`.

Separating decoding and drawing with two threads

The sample project in previous recipe decodes the frame and draws the frame on canvas within a single thread. It is generally a better idea to separate the two with two threads. Firstly, the processing time for each frame differs. If a particular frame takes a longer time to decode, its previous frame is going to stay longer on screen if a single thread is used. Secondly, you can take advantage of the multi-core CPU available in many Android devices to achieve a smoother video playback.

Getting ready

The sample project uses multithreading, which is discussed in the *Chapter 6*, *Android NDK Multithreading*, in detail. Readers who are not familiar with multithreading are recommended to go through the following recipes first:

- *Creating and terminating native threads in Android NDK*
- *Synchronizing native threads with Semaphore in Android NDK*

How to do it...

The following steps create an Android project that decodes video using `ffmpeg`:

1. Create a copy of the `FrameGrabberTemplate` folder and rename it as `FrameGrabberDecodeAsync`. Open the copied project in Eclipse using **File** | **Import** | **Existing Projects into Workspace**. Follow steps 2 to 4 of the *Using the ffmpeg library to get media info to link to the ActionBarSherlock library* recipe of this chapter.
2. Follow steps 2 and 3 of the *Decoding and display the video frame* recipe of this chapter to add the Java source files `VideoEditActivity.java` and `SimpleVideoSurfaceView.java`, and XML file `activity_video_edit.xml`.
3. Create a `jni` folder under the project. Add a source file `framegrabber.c` under the `jni` folder. `framegrabber.c` is updated from the `framegrabber.c` file seen in the previous recipe. Let's highlight a few changes:
    - `VideoState`: The `VideoState` data structure is updated. An array of `AVFrame` pointers is added to buffer the decoded frames. Two semaphores, `fqfullsem` and `fqemptysem`, are provided to synchronize access to the array:

      ```
      #define VIDEO_FR_QUEUE_SIZE 5
      typedef struct VideoState {
        AVFormatContext *pFormatCtx;
        AVStream *pVideoStream;
      ```

```
  int videoStreamIdx;

  AVFrame* frameq[VIDEO_FR_QUEUE_SIZE];
  int frameqDequeueIdx, frameqEnqueueIdx;

  int fint;
  int64_t nextFrameTime;

  sem_t fqfullsem, fqemptysem;
  int status;
}VideoState;
```

- `decode_video_thread`: It is a function run by a separate thread. It reads and decodes video frames continuously:

```
void *decode_video_thread(void *arg) {
  VideoState* vs = (VideoState*)arg;
  AVPacket packet;
  int framefinished;
  while ((!vs->status) && 0 == av_read_frame(vs->pFormatCtx,
&packet)) {
      if (vs->videoStreamIdx == packet.stream_index) {
          sem_wait(&vs->fqemptysem);
          AVFrame *pframe = vs->frameq[vs->frameqEnqueueIdx];
          avcodec_decode_video2(vs->pVideoStream->codec,
pframe, &framefinished, &packet);
          if (framefinished) {
              vs->frameqEnqueueIdx++;
              if (VIDEO_FR_QUEUE_SIZE == vs-
>frameqEnqueueIdx) {
                  vs->frameqEnqueueIdx = 0;
              }
              sem_post(&vs->fqfullsem);
          }
      }
      av_free_packet(&packet);
  }
  vs->status = 2; //end of decoding
  return 0;
}
```

- `naStartDecodeVideo`: It starts the thread to run the `decode_video_thread` function:

  ```
  int naStartDecodeVideo(JNIEnv *pEnv, jobject pObj) {
    VideoState *vs = gvs;
    init_queue(vs);
    vs->status = 0;
    pthread_t decodeThread;
    pthread_create(&decodeThread, NULL, decode_video_thread,
  (void*)vs);
    return 0;
  }
  ```

- `naGetVideoFrame`: It takes a decoded frame from the `vs->frameq` buffer, converts its colorspace, and scales it. The function returns the frame number of the scaled frame, or zero if there are no more frames to display:

  ```
  int naGetVideoFrame(JNIEnv *pEnv, jobject pObj) {
    VideoState* vs = gvs;
    VideoDisplayUtil *vdu = gvdu;
    if (!vs->status || (vs->frameqDequeueIdx != vs-
  >frameqEnqueueIdx)) {
       int filledSlots = 0;
       sem_getvalue(&vs->fqfullsem, &filledSlots);
       sem_wait(&vs->fqfullsem);
       AVFrame *pframe = vs->frameq[vs->frameqDequeueIdx];
       sws_scale(vdu->img_resample_ctx, pframe->data,
  pframe->linesize, 0, vs->pVideoStream->codec->height, vdu-
  >pFrameRGBA->data, vdu->pFrameRGBA->linesize);
       int64_t curtime = av_gettime();
       if (vs->nextFrameTime - curtime > 20*1000) {
           usleep(vs->nextFrameTime-curtime);
       }
       ++vdu->frameNum;
       vs->nextFrameTime += vs->fint*1000;
       vs->frameqDequeueIdx++;
       if (VIDEO_FR_QUEUE_SIZE == vs->frameqDequeueIdx) {
           vs->frameqDequeueIdx = 0;
       }
       sem_post(&vs->fqemptysem);
       return vdu->frameNum;
    } else {
       vs->status = 3; //no more frame to display
       return 0;
    }
  }
  ```

4. Add an `Android.mk` file under the `jni` folder with the following content:

   ```
   LOCAL_PATH := $(call my-dir)
   include $(CLEAR_VARS)
   LOCAL_MODULE  := framegrabber
   LOCAL_SRC_FILES := framegrabber.c
   #LOCAL_CFLAGS := -DANDROID_BUILD
   LOCAL_LDLIBS := -llog -ljnigraphics -lz
   LOCAL_STATIC_LIBRARIES := libavformat_static libavcodec_static
   libswscale_static libavutil_static
   include $(BUILD_SHARED_LIBRARY)
   $(call import-module,ffmpeg-1.0.1/android/arm)
   ```

5. Build the Android project and start the application on an Android device. The application behaves similar to the sample project in the previous recipe.

How it works...

The sample project uses two threads to handle the decoding, scaling, and drawing. The design of the application can be illustrated as follows.

The UI thread starts two threads. Let's call them **decoding thread** and **display thread**. The decoding thread is started by `naStartDecodeVideo` and runs the function `decode_video_thread`. It continuously reads and decodes frames and puts the decoded frames into the frame buffer queue `VideoState->frameq`. The display thread is started at the Java code, it calls the native method `naGetVideoFrame` to retrieve the decoded frame from the buffer, scales it, and draws the pixels on a canvas. The thread then notifies the UI thread to update the display.

To decode a video frame, the code reads the frame, decodes it, converts the color and scales it, draws it on canvas, and then renders it on the phone display. Our design divides the tasks in such a way that the decoding thread reads the frame and decodes it, the display thread converts the color and scales it, draws it on canvas, and the UI thread finally renders the pixels on the phone display. There are alternative designs. For example, you can separate the reading and decoding by adding another thread, or move the tasks of color conversion and scaling to decoding the thread. Interested readers can implement those designs and compare the performance.

Queue management and thread synchronization

You used an array of `AVFrame` as the queue for decoded (but not scaled) frames.
The decoding thread saves the decoded frame into the queue. An integer variable `frameqEnqueueIdx` in the `VideoState` data structure is used to keep a track of the next `AVFrame` slot to use. The display thread takes the decoded frame from the queue. An integer variable `frameqDequeueIdx` indicates the next `AVFrame` to take.

The two threads need to be synchronized in such a way that the decoding thread will block when there is no empty `AVFrame` slot to use, and the display thread will block when there is no filled `AVFrame` to scale and display. This is done with two semaphores, `fqfullsem` and `fqemptysem`, which indicate the number of filled and empty `AVFrame` slots in the queue respectively.

In the decoding thread, we first decrease `fqemptysem` by one (`sem_wait`) to indicate that you just reserved `AVFrame` located by `frameqEnqueueIdx`. We then decode a frame and save the data to the reserved frame. Finally, we increase `fqfullsem` by one (`sem_post`) to signal that a new frame is available for display. In the display thread, we first decrease `fqfullsem` by one to indicate that we obtained `AVFrame` at `frameqDequeueIdx` to process. After we scale the frame, we increase `fqemptysem` by one to signal that this `AVFrame` slot can now be reused. In this way, the two threads can operate independently and only when the queue is empty or full, awill a thread block to wait for the other thread.

Seeking to playback and grabbing the frames

In this recipe, we will complete the frame grabber application by adding seeking at video playback and saving the video frame displayed to pictures.

How to do it...

The following steps are to create an Android application that allows seeking at video playback and saves video frames to pictures:

1. Create a copy of the `FrameGrabberTemplate` folder and rename it as `FrameGrabber`. Open the copied project in Eclipse using **File** | **Import** | **Existing Projects into Workspace**. Follow steps 2 to 4 of the *Using ffmpeg library to get media info* recipe of this chapter to link to the `ActionBarSherlock` library.
2. Follow steps 2 and 3 of the *Decoding and display the video frame* recipe of this chapter to add the Java source files `VideoEditActivity.java` and `SimpleVideoSurfaceView.java`, and XML file `activity_video_edit.xml`.
3. Modify the `activity_video_edit.xml` file to include a **Gallery** widget for displaying the frames grabbed. Update `SimpleVideoSurfaceView.java` by adding a `grab` button and the code to save the frame being displayed:

```
public void saveCurrentFrame() {
   try {
     File extDir = Environment.getExternalStorageDirectory();
     String filePath = extDir.getAbsolutePath() + "/framegrabber/"
+ mVideoFileName + "_" + mVideoCurrentFrame + ".png";
     FileOutputStream out = new FileOutputStream(filePath);
```

```
      mBitmap.compress(Bitmap.CompressFormat.PNG, 90, out);
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
```

4. Create a `jni` folder under the project. Add a source file `framegrabber.c` under the `jni` folder. `framegrabber.c` is updated using the `framegrabber.c` file seen in the previous recipe. Let's highlight a few changes:
   - `decode_video_thread`: This function is run by the decoding thread. It seeks to proper key frame when seeking is performed:

```
void *decode_video_thread(void *arg) {
  VideoState* vs = (VideoState *)arg;
  AVPacket packet;
  int framefinished;
  while ((!vs->status) && 0 <= av_read_frame(vs->pFormatCtx,
&packet)) {
    if (vs->ifSeek) {
      int64_t targetTime = (int64_t)(vs->seekTargetTime *
(AV_TIME_BASE / 1000));
      targetTime = av_rescale_q(targetTime, AV_TIME_BASE_Q,
vs->pVideoStream->time_base);
      av_seek_frame(gvs->pFormatCtx, vs->videoStreamIdx,
targetTime, 0) < 0);
      vs->frameqEnqueueIdx = 0;
      pthread_mutex_lock(&vs->mux);
      while (2!=vs->ifSeek) {
        pthread_cond_wait(&vs->cond, &vs->mux);
      }
      vs->ifSeek = 0;
      pthread_mutex_unlock(&vs->mux);
    } else if (vs->videoStreamIdx == packet.stream_index) {
      sem_wait(&vs->fqemptysem);
      AVFrame *pframe = vs->frameq[vs->frameqEnqueueIdx];
      avcodec_decode_video2(vs->pVideoStream->codec, pframe,
&framefinished, &packet);
      if (framefinished) {
        vs->frameqEnqueueIdx++;
        if (VIDEO_FR_QUEUE_SIZE == vs->frameqEnqueueIdx) {
          vs->frameqEnqueueIdx = 0;
}
        sem_post(&vs->fqfullsem);
      }
    }
    av_free_packet(&packet);
  }
  vs->status = 2;  //end of decoding
  return 0;
}
```

- `naGetVideoFrame`: This function is run by the display thread. It resets the frame buffer queue when seeking is performed:

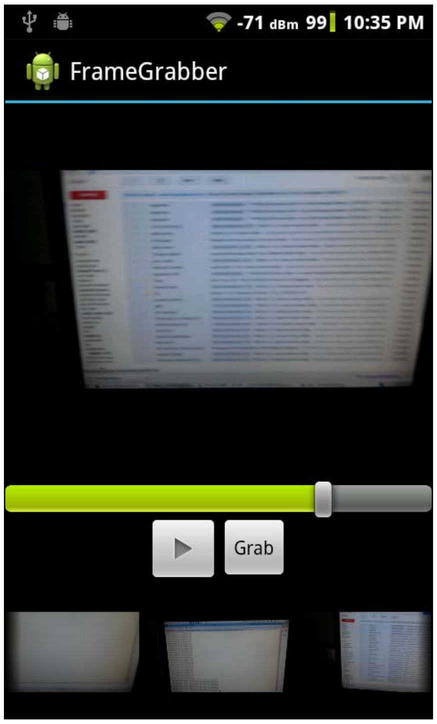
```
int naGetVideoFrame(JNIEnv *pEnv, jobject pObj) {
  VideoState *vs = gvs;
  VideoDisplayUtil *vdu = gvdu;
  if (!vs->status || (vs->frameqDequeueIdx != vs-
>frameqEnqueueIdx)) {
    if (vs->ifSeek) {
      int numOfItemsInQueue = 0;
      sem_getvalue(&vs->fqfullsem, &numOfItemsInQueue);
      while (numOfItemsInQueue--) {
        sem_wait(&vs->fqfullsem);
        sem_post(&vs->fqemptysem);
      }
      while (vs->ifSeek) {
        pthread_mutex_lock(&vs->mux);
        vs->ifSeek = 2;
        pthread_cond_signal(&vs->cond);
        pthread_mutex_unlock(&vs->mux);
      }
      vs->frameqDequeueIdx = 0;
      vdu->frameNum = vs->seekTargetTime / vs->fint;
      vs->nextFrameTime += vs->fint * 1000;
    } else {
      sem_wait(&vs->fqfullsem);
      AVFrame *pframe = vs->frameq[vs->frameqDequeueIdx];
             sws_scale(vdu->img_resample_ctx, pframe->data,
pframe->linesize, 0, vs->pVideoStream->codec->height, vdu-
>pFrameRGBA->data, vdu->pFrameRGBA->linesize);
      int64_t curtime = av_gettime();
      if (vs->nextFrameTime - curtime > 20*1000) {
        usleep(vs->nextFrameTime-curtime);
      }
      ++vdu->frameNum;
      vs->nextFrameTime += vs->fint*1000;
      vs->frameqDequeueIdx++;
      if (VIDEO_FR_QUEUE_SIZE == vs->frameqDequeueIdx) {
        vs->frameqDequeueIdx = 0;
      }
```

```
                sem_post(&vs->fqemptysem);
              }
              return vdu->frameNum;
            } else {
              vs->status = 3;  //no more frame to display
              return 0;
            }
          }
```

5. Add an `Android.mk` file under the `jni` folder. The file is exactly the same as the `Android.mk` file shown in step 4 of the previous recipe.
6. Add the `WRITE_EXTERNAL_STORAGE` permission after `<uses-sdk />` in the `AndroidManifest.xml` file:

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

7. Build the Android project and start the application on an Android device. You can now seek to a specific position by dragging the seek bar handle or clicking on a position in the seek bar. You can also grab the frame by clicking on the **Grab** button. The grabbed frames will be shown in the gallery at the bottom of the screen. This is shown in the following screenshot:



How it works...

The preceding example illustrates how to enable seeking to a specific time and save the frame displayed to a picture.

How to perform a seek at ffmpeg: The function `av_seek_frame` is provided by `ffmpeg` to perform seeking on a stream. It has the following prototype:

```
int av_seek_frame(AVFormatContext *s, int stream_index, int64_t
timestamp, int flags);
```

The function seeks to the key frame near the time specified by the argument timestamp in the media stream specified by `stream_index`. Note that the timestamp should be in `AVStream.time_base` units. The function returns a value greater than or equal to `0` on success.

In the `decode_video_thread` function, we first convert the seek target time from milliseconds to `AV_TIME_BASE`. We then call the `av_rescale_q` function from `ffmpeg` to convert the time from `AV_TIME_BASE` to `vs->pVideoStream->time_base`.

The `avformat_seek_file` function is the new seeking function provided by `ffmpeg`. However, it is not stable on `ffmpeg` 1.0.1 yet. It is recommended that you use `av_seek_frame` for now.

Thread synchronization at seeking: We want to skip the frames left in the queue. To keep things consistent (a decoding thread saves frames to the queue, and a display thread consumes it), the cleanup is performed in the display thread. Therefore, the decoding thread needs to wait for the display thread to clear the queue before it can decode a new frame and add it to the queue.

You used a mutex and a conditional variable to synchronize the two threads. The decode thread will perform the seeking on the video stream and will wait on the conditional variable (`pthread_cond_wait`) while `vs->ifSeek` is not equal to `2`. The display thread will clear the queue, set `vs->ifSeek` to `2`, and signal the decode thread (`pthread_cond_signal`). It keeps sending signals as long as `vs->ifSeek` is not set to `0`. This is to prevent the situation that the display thread calls `pthread_cond_signal` before the decode thread is blocked by `pthread_cond_wait`. Once the decode thread receives the signal, it wakes up, sets `vs->ifSeek` to `0` to indicate the seeking is completed, and continues decoding the frames at the new position. The display thread will get out of the signal loop, set `frameqDequeueIdx`, `frameNum`, and `nextFrameTime` properly.

Save Frame to Pictures: You save the frame displayed to pictures at SimpleVideoSurfaceView.java. Since the frame data is stored at the bitmap object, you can easily compress the bitmap to the formats supported by Android bitmap class. In our code, you save the bitmap to the PNG file.

Optimizing the performance of multimedia apps

We have our frame grabber app up and running. However, the video playback is slow for some high resolution videos because the time it takes to decode, convert color, and scale is longer than the targeted delay between two frames. In other words, the code is not fast enough to achieve the proper frame rate. This recipe discusses the optimizations you can do to improve the performance. As an example, we will build `ffmpeg` for different CPU architectures to see if there's any performance improvement.

Getting ready

It is helpful that you read the *Building Android NDK applications for different CPU features* recipe in *Chapter 3*, *Build and Debug NDK Application*.

This recipe assumes that you have followed the first recipe of this chapter to build `ffmpeg-1.0.1` at the `sources` folder of the NDK directory.

How to do it...

The following steps are to build `ffmpeg` for different CPU architectures, recompile our frame grabber application, and profile its performance:

1. Update the `build_android.sh` script as follows to build six different versions of the `ffmpeg` libraries:

```
#!/bin/bash
#modify the NDK path to point to the Android NDK path
NDK=/home/roman10/Desktop/android/android-ndk-r8b
SYSROOT=$NDK/platforms/android-8/arch-arm/
TOOLCHAIN=$NDK/toolchains/arm-linux-androideabi-4.6/prebuilt/
linux-x86
function build_one
{
./configure \
    --prefix=$PREFIX \
    --disable-shared \
    --enable-static \
    --disable-doc \
    --disable-ffmpeg \
    --disable-ffplay \
    --disable-ffprobe \
    --disable-ffserver \
    --disable-avdevice \
    --disable-doc \
    --cross-prefix=$TOOLCHAIN/bin/arm-linux-androideabi- \
    --target-os=linux \
    --arch=arm \
    --disable-symver \
    --enable-cross-compile \
    --sysroot=$SYSROOT \
    --extra-cflags="-Os -fpic -DANDROID $ADDI_CFLAGS" \
    --extra-ldflags="$ADDI_LDFLAGS" \
    $ADDITIONAL_CONFIGURE_FLAG
make clean
make -j 2
make install
}

#arm
CPU=armv5te
ADDI_CFLAGS="-marm"
ADDITIONAL_CONFIGURE_FLAG="--disable-neon --enable-armv5te
--disable-armv6 --disable-armv6t2 --disable-armvfp"
PREFIX=$(pwd)/android/$CPU
build_one

#armv6
CPU=armv6
ADDI_CFLAGS="-marm -march=armv6"
ADDI_LDFLAGS="-Wl,--fix-cortex-a8"
PREFIX=$(pwd)/android/$CPU
ADDITIONAL_CONFIGURE_FLAG="--disable-neon --disable-armv5te
--enable-armv6 --disable-armv6t2 --disable-armvfp"
build_one

#armv6vfp
CPU=armv6vfp
ADDI_CFLAGS="-mfloat-abi=softfp -mfpu=vfp -marm -march=armv6"
ADDI_LDFLAGS="-Wl,--fix-cortex-a8"
PREFIX=$(pwd)/android/$CPU
ADDITIONAL_CONFIGURE_FLAG="--disable-neon --disable-armv5te
--enable-armv6 --disable-armv6t2 --enable-armvfp"
build_one

#armv7
CPU=armv7
```

```
ADDI_CFLAGS="-mfloat-abi=softfp -mfpu=vfp -marm -march=armv7-a"
ADDI_LDFLAGS="-Wl,--fix-cortex-a8"
PREFIX=$(pwd)/android/$CPU
ADDITIONAL_CONFIGURE_FLAG="--disable-neon --disable-armv5te
--disable-armv6 --disable-armv6t2 --enable-armvfp"
build_one

#armv7 neon
CPU=armv7-neon
ADDI_CFLAGS="-mfloat-abi=softfp -mfpu=neon -marm -march=armv7-a"
ADDI_LDFLAGS="-Wl,--fix-cortex-a8"
PREFIX=$(pwd)/android/$CPU
ADDITIONAL_CONFIGURE_FLAG="--enable-neon --enable-armv5te
--enable-armv6 --enable-armv6t2 --enable-armvfp"
build_one

#armv7 neon 2
CPU=armv7-neon-2
ADDI_CFLAGS="-mfloat-abi=softfp -mfpu=neon -marm -march=armv7-a"
ADDI_LDFLAGS="-Wl,--fix-cortex-a8"
PREFIX=$(pwd)/android/$CPU
ADDITIONAL_CONFIGURE_FLAG="--enable-neon --disable-armv5te
--disable-armv6 --disable-armv6t2 --disable-armvfp"
build_one
```

We commented out `build_one` for arm CPU as you have already built the libraries in first recipe of this chapter.

2. Start a command-line shell, and go to the `sources/ffmpeg-1.0.1` directory. Enter the following command to build the libraries:

```
$ ./build_android.sh
```

The build will take a while. After it is done, you should be able to find the library files and headers in the subdirectories of the `sources/ffmpeg-1.0.1/android` folder.

3. At the six subdirectories of the `sources/ffmpeg-1.0.1/android` folder, `armv5te`, `armv6`, `armv6vfp`, `armv7`, `armv7-neon`, and `armv7-neon-2`, add an `Android.mk` file:

```
LOCAL_PATH:= $(call my-dir)
#static version of libavcodec
include $(CLEAR_VARS)
LOCAL_MODULE:= libavcodec_static
LOCAL_SRC_FILES:= lib/libavcodec.a
LOCAL_EXPORT_C_INCLUDES := $(LOCAL_PATH)/include
include $(PREBUILT_STATIC_LIBRARY)
```

```
#static version of libavformat
include $(CLEAR_VARS)
LOCAL_MODULE:= libavformat_static
LOCAL_SRC_FILES:= lib/libavformat.a
LOCAL_EXPORT_C_INCLUDES := $(LOCAL_PATH)/include
include $(PREBUILT_STATIC_LIBRARY)

#static version of libswscale
include $(CLEAR_VARS)
LOCAL_MODULE:= libswscale_static
LOCAL_SRC_FILES:= lib/libswscale.a
LOCAL_EXPORT_C_INCLUDES := $(LOCAL_PATH)/include
include $(PREBUILT_STATIC_LIBRARY)

#static version of libavutil
include $(CLEAR_VARS)
LOCAL_MODULE:= libavutil_static
LOCAL_SRC_FILES:= lib/libavutil.a
LOCAL_EXPORT_C_INCLUDES := $(LOCAL_PATH)/include
include $(PREBUILT_STATIC_LIBRARY)
```

4. Update the source code of `framegrabber.c` to include additional log statements for profiling. Let's see the log statements for logging the decoding time and scaling time:

```
LOGI(2, "---avcodec_decode_video2 ST");
avcodec_decode_video2(vs->pVideoStream->codec, pframe,
&framefinished, &packet);
LOGI(2, "---avcodec_decode_video2 ED");...
...
LOGI(2, "---sws_scale ST");
sws_scale(vdu->img_resample_ctx, pframe->data, pframe->linesize,
0, vs->pVideoStream->codec->height, vdu->pFrameRGBA->data, vdu-
>pFrameRGBA->linesize);
LOGI(2, "---sws_scale ED");
...
```

5. You will use a Python script `profile.py` for parsing the logout output and calculating the average time for decoding, scaling, and color conversion. The script can be found under the `profile` folder of the source code for this chapter.

6. Open the frame grabber application you developed in the previous recipes and update the `Android.mk` file under the `jni` folder as follows. The changed part is highlighted as follows:

```
LOCAL_PATH := $(call my-dir)
include $(CLEAR_VARS)
LOCAL_MODULE  := framegrabber
LOCAL_SRC_FILES := framegrabber.c
#LOCAL_CFLAGS := -DANDROID_BUILD
LOCAL_LDLIBS := -llog -ljnigraphics -lz
LOCAL_STATIC_LIBRARIES := libavformat_static libavcodec_static
libswscale_static libavutil_static
include $(BUILD_SHARED_LIBRARY)
$(call import-module,ffmpeg-1.0.1/android/armv5te)
```

7. Add an `Application.mk` file under the `jni` folder with the following content:

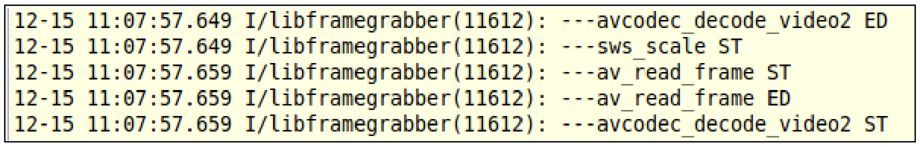
```
APP_ABI := armeabi
```

8. Build the application again and run it on an Android device. Start a command-line shell and create a directory named `profile`, go to the folder, and enter the following command:

```
adb logcat -c
```

```
adb logcat -v time libframegrabber:I SimpleVideoView:I *:S | tee
armv5te.txt
```

9. Click on a video and then click on the **Play** button. You should be able to see the logcat output as follows:



```
12-15 11:07:57.649 I/libframegrabber(11612): ---avcodec_decode_video2 ED
12-15 11:07:57.649 I/libframegrabber(11612): ---sws_scale ST
12-15 11:07:57.659 I/libframegrabber(11612): ---av_read_frame ST
12-15 11:07:57.659 I/libframegrabber(11612): ---av_read_frame ED
12-15 11:07:57.659 I/libframegrabber(11612): ---avcodec_decode_video2 ST
```

10. Repeat steps 6 to 9 for all six different builds of the `ffmpeg` library. Note that you need to update the last line of `Android.mk` in step 6, the `APP_ABI` in step 7, and the logcat command in step 8 according to the following table:

CPU type	Android.mk last line	APP_ABI	Logcat command

`armv5te`	`$(call import-module,ffmpeg-1.0.1/android/armv5te)`	`APP_ABI := armeabi`	`adb logcat -c`
			`adb logcat -v time libframegrabber:I SimpleVideoView:I *:S | tee armv5te.txt`

`armv6`	`$(call import-module,ffmpeg-1.0.1/android/armv6)`	`APP_ABI := armeabi`	`adb logcat -c`
			`adb logcat -v time libframegrabber:I SimpleVideoView:I *:S | tee armv6.txt`

`armv6vfp`	`$(call import-module,ffmpeg-1.0.1/android/armv6vfp)`	`APP_ABI := armeabi`	`adb logcat -c`
			`adb logcat -v time libframegrabber:I SimpleVideoView:I *:S | tee armv6vfp.txt`

`armv7`	`$(call import-module,ffmpeg-1.0.1/android/armv7)`	`APP_ABI := armeabi-v7a`	`adb logcat -c`
			`adb logcat -v time libframegrabber:I SimpleVideoView:I *:S | tee armv7txt`

`armv7-neon`	`$(call import-module,ffmpeg-1.0.1/android/armv7-neon)`	`APP_ABI := armeabi-v7a`	`adb logcat -c`
			`adb logcat -v time libframegrabber:I SimpleVideoView:I *:S | tee armv7neon.txt`

`armv7-neon-2`	`$(call import-module,ffmpeg-1.0.1/android/armv7-neon-2)`	`APP_ABI := armeabi-v7a`	`adb logcat -c`
			`adb logcat -v time libframegrabber:I SimpleVideoView:I *:S | tee armv7neon2.txt`

11. Get the profile result with the following command. The result will be stored at `res_armv5te.txt`, `res_armv6.txt`, `res_armv6vfp.txt`, `res_armv7.txt`, `res_armv7neon.txt`, and `res_armv7neon2.txt`:

```
$ ./profile.py armv5te.txt
$ ./profile.py armv6.txt
$ ./profile.py armv6vfp.txt
$ ./profile.py armv7.txt
$ ./profile.py armv7neon.txt
$ ./profile.py armv7neon2.txt
```

You can summarize the profiling results as follows.

	Read packet	Decode	Scale and color conversion	Draw on canvas
Armv5te	0.58	169.79	138.46	2.69
Armv6	0.19	173.75	138.73	4.78
Armv6vfp	0.11	169.31	137.91	2.81
Armv7	0.10	157.31	123.82	2.96
Armv7neon	0.14	155.83	122.36	2.94
Armv7neon2	0.11	156.30	121.61	2.72

How it works...

The preceding example demonstrates building different versions of the `ffmpeg` libraries for different CPU architectures and profiling for performance comparison.

- **Building the ffmpeg libraries**: `ffmpeg` build scripts accept many of the configuration options, which allows us to control how the libraries are built. In our example, we enabled different optimizations for different CPU architectures. Interested readers can execute `./configure --help` to view all the configuration options supported by the build scripts and try out more options.
- **Profiling**: We use a simple Python script to parse the logcat output for profiling. We inserted the logcat output statement before and after calling a few functions, including the reading packet, decoding the packet, scaling and color conversion, and drawing on canvas. The Python script looks for the start time and end time every time a function is called and gets the difference as the processing time for the function. We used a 60 second video of 1280x920 pixels for testing, and the average processing time for each function is produced as the output.

  Note that you used the final frame grabber for this recipe. To eliminate the effect of multithreading, you can use the single-threaded frame grabber application in the *Decoding and display the video frame* recipe.

- **Other optimizations**: This recipe can only be a start for application performance optimization based on `ffmpeg`. There are a few directions that you can explore. Firstly, the profiling results tell us that the color conversion and scaling are time-consuming. In fact, the color conversion and scaling code is not optimized to run on ARM-based device at the time of writing. There are other open source projects that provide color conversion and scaling, such as the `libyuv` project. You can replace the `ffmpeg` color conversion and scaling with other projects' code and compare the performance. Secondly, the compiler matters. Different compilers can compile the same source code to significantly different binaries. The latest Android NDK r8d offers GCC 4.4.3, GCC 4.6, GCC 4.7, and CLANG 3.1 toolchains. It's worth the effort to try building applications based on `ffmpeg` with different toolchains and compare the performance. Thirdly, the compilers and `ffmpeg` offer far more configurations than what you have seen so far; you should examine more options and look for the options that can result in binaries with better performance.

There's more...

You have developed and partially optimized the frame grabber application. Many of us may want to further develop this to a video player. A video player is generally much more complicated, especially when you want to support lots of codecs, file formats, color spaces, and so on. The `ffmpeg` library can handle a lot for us. But, there are still many things that you need to do yourself.

The first thing to consider is audio decoding. You can use `ffmpeg` APIs to decode the audio. Once you have both video and audio, you need to synchronize the two. There are three approaches, synchronize audio to video, synchronize video to audio, and synchronize both video and audio to an external clock. In addition, you used Android `SurfaceView` and bitmap for rendering, so that you can easily grab the frame by saving the bitmap pixels. However, this may not be the optimal approach for rendering. You can explore options, such as extending a view class and using OpenGL, and compare their performance. Once you have the basic video playback functions working smoothly (it is not easy!), you can look into things such as subtitle display and frames skipping.

A good reference to writing a video player with `ffmpeg` is available at `http://dranger.com/ffmpeg/`.