ERIK HELLMAN

PLATFORMA ANDROID

NOWE WYZWANIA

Profesjonalne programowanie w systemie Android!

Helion

Tytuł oryginału: Android Programming: Pushing the Limits

Tłumaczenie: Łukasz Piwko

ISBN: 978-83-246-9525-6

Wydawnictwo HELION
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 32 231 22 19, 32 230 98 63
e-mail: helion@helion.pl
WWW: http://helion.pl (księgarnia internetowa, katalog książek)

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
ftp://ftp.helion.pl/przyklady/plandr.zip

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/plandr
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

Spis treści

Rozdział 11.

Zaawansowane aplikacje obsługujące audio, wideo oraz kamerę

Podczas gdy do pracy z dźwiękiem, filmami i kamerą wystarczą podstawowe API, w zaawansowanych aplikacjach często potrzebne są dodatkowe funkcje. Do zwykłego nagrywania i odtwarzania mediów wystarczą dostępne w Androidzie API Javy. Niestety, opóźnienia powodowane przez te wysokopoziomowe API mogą być czasami nie do przyjęcia. Poza tym można mieć trudności w czasie pracy z aplikacjami wykorzystującymi kamerę, takimi jak aplikacje rozszerzonej rzeczywistości i inne, w których potrzebne jest reagowanie programu na bieżąco.

W tym rozdziale dowiesz, się jak maksymalnie wykorzystać różne właściwości mediów w Androidzie. W systemie tym można używać zarówno wysokowydajnych, jak i niskopoziomowych API, takich jak OpenGL ES do grafiki czy OpenSL ES do dźwięku. Nauczysz się używać tych API w kombinacji i poznasz kilka przypadków użycia zaawansowanych funkcji dźwiękowych, wideo i kamery.

> Niektóre przedstawione w tym rozdziale przykłady (m.in. ten odnoszący się do OpenSL ES) zawierają kod w języku C. Jeśli nie wiesz, jak się używa NDK Androida do budowy aplikacji przy użyciu rodzimego kodu C, możesz przed tym rozdziałem przeczytać rozdział 14.

Zaawansowane aplikacje dźwiękowe

Pierwszymi aplikacjami dźwiękowymi w Androidzie były odtwarzacze muzyki. Można było włączyć strumieniowe odtwarzanie piosenek z internetu albo odtworzyć utwory posiadane w pamięci urządzenia. Jednak w miarę jak platforma Android się rozwijała, zaczęły być potrzebne bardziej zaawansowane API do obsługi dźwięku. Dlatego firma Google postanowiła dodać API umożliwiające strumieniowanie i nagrywanie dźwięku z małym opóźnieniem.

Obecnie dostępne w Androidzie API dźwiękowe obsługują zaawansowane funkcje, których można używać w aplikacjach. Teraz już nie jest trudne zaimplementowanie aplikacji VoIP (ang. *voice over internet protocol*), zbudowanie własnego strumieniowego klienta muzycznego czy zaimplementowanie nieopóźnionych odgłosów w grach. Ponadto dostępne są API umożliwiające korzystanie z funkcji czytania tekstu na głos i rozpoznawania mowy, dzięki którym użytkownicy mogą obsługiwać aplikacje za pomocą dźwięków zamiast wizualnego interfejsu użytkownika lub technologii dotykowych.

W tym podrozdziale objaśnię, jak używać tych funkcji, oraz przedstawię przykłady kodu, dzięki którym łatwiej Ci będzie rozpocząć pracę z nimi.

Odtwarzanie dźwięków z krótkim opóźnieniem

System Android zawiera cztery API do odtwarzania dźwięków (a nawet pięć, jeśli doliczy się MIDI) i trzy do ich nagrywania. W kolejnych kilku sekcjach znajduje się przegląd tych API wraz z przykładami kodu ilustrującymi sposób użycia bardziej zaawansowanych funkcji.

API do odtwarzania dźwięku

Domyślną klasą do obsługi odtwarzania dźwięku jest `MediaPlayer`. Klasa ta zawiera podstawowe funkcje dotyczące odtwarzania dźwięku odpowiednie do użycia w odtwarzaczach muzyki (lub filmów) i udostępnia wygodne API do odtwarzania dźwięków zarówno ze źródeł strumieniowych (np. radia internetowego), jak i z lokalnych plików. Z klasą `MediaPlayer` związana jest zaawansowana maszyna stanów, którą musi śledzić aplikacja. Przy użyciu tego API można dodać do aplikacji odtwarzacz muzyki lub filmów i nie martwić się o dodatkowe przetwarzanie ani opóźnienia.

Inną możliwością w kwestii odtwarzania dźwięków jest klasa `SoundPool`, która charakteryzuje się odtwarzaniem dźwięku z bardzo małym opóźnieniem, a więc jest odpowiednia do odtwarzania odgłosów i innych krótkich dźwięków. API tego używa się np. do włączania odgłosów w grach. Brak w nim jednak obsługi strumieniowania dźwięku, więc nie nadaje się do użycia w aplikacjach odtwarzających strumienie dźwiękowe, takich jak VoIP.

Trzecią możliwość odtwarzania dźwięku reprezentuje klasa `AudioTrack`, która umożliwia przekazanie do sprzętu pojedynczego strumienia dźwiękowego w postaci surowych buforów PCM. Rozwiązanie to zapewnia niskie opóźnienia nawet przy strumieniowaniu. Klasa `AudioTrack` jest tak efektywna, że może być z powodzeniem używana w VoIP i podobnych technologiach.

Poniżej znajduje się przykład użycia klasy `AudioTrack` w typowej aplikacji VoIP:

```
public class AudioTrackDemo {

    private final AudioTrack mAudioTrack;
    private final int mMinBufferSize;

    public AudioTrackDemo() {
        mMinBufferSize = AudioTrack.getMinBufferSize(16000,
                AudioFormat.CHANNEL_OUT_MONO,
                AudioFormat.ENCODING_PCM_16BIT);
        mAudioTrack = new AudioTrack(AudioManager.STREAM_VOICE_CALL,
                16000,
                AudioFormat.CHANNEL_OUT_MONO,
                AudioFormat.ENCODING_PCM_16BIT,
                mMinBufferSize * 2,
                AudioTrack.MODE_STREAM);
    }

    public void playPcmPacket(byte[] pcmData) {
        if(mAudioTrack != null
                && mAudioTrack.getState() == AudioTrack.STATE_INITIALIZED)
{
            if(mAudioTrack.getPlaybackRate()
                    != AudioTrack.PLAYSTATE_PLAYING) {
                mAudioTrack.play();
```

```
                }
                mAudioTrack.write(pcmData, 0, pcmData.length);
            }
        }

        public void stopPlayback() {
            if(mAudioTrack != null) {
                mAudioTrack.stop();
                mAudioTrack.release();
            }
        }
    }
```

Najpierw określiłem minimalny rozmiar bufora dla strumienia audio. Aby to zrobić, trzeba znać częstotliwość próbkowania, wiedzieć, czy sygnał jest mono-, czy stereofoniczny oraz czy będzie używane 8-, czy 16-bitowe kodowanie PCM. Częstotliwość próbkowania i rozmiar próbki są parametrami metody `AudioTrack.getMinBufferSize()`, która zwraca minimalny rozmiar bufora w bajtach dla danego egzemplarza klasy `AudioTrack`.

Egzemplarz klasy `AudioTrack` tworzy się przy użyciu parametrów dostosowanych do potrzeb programisty. Pierwszy parametr dostosowuje się do typu używanego sygnału dźwiękowego. W przypadku aplikacji VoIP należy użyć `STREAM_VOICE_CALL`, a do strumieniowania muzyki — `STREAM_MUSIC`.

Drugi, trzeci i czwarty parametr zmieniają się w zależności od sytuacji. Służą one odpowiednio do określania częstotliwości próbkowania, czy sygnał jest stereo, czy mono, oraz rozmiaru próbkowania. W aplikacji VoIP można by było zastosować 16 kHz przy 16 bitach mono, natomiast w odtwarzaczu muzyki lepsze byłoby ustawienie 44,1 kHz przy 16 bitach stereo. Wyższa częstotliwość próbkowania przy 16 bitach stereo wymaga zastosowania większego bufora oraz sprawniejszego przesyłania danych, ale pozwala uzyskać lepszą jakość. Wszystkie urządzenia z Androidem obsługują odtwarzanie sygnału PCM o częstotliwości próbkowania 8, 16 i 44,1 kHz, z rozmiarem próbki 8 i 16 bitów stereo.

Wartość parametru określającego rozmiar bufora `AudioTrack` powinna być wielokrotnością minimalnego rozmiaru bufora. Wybór konkretnej wartości zależy od potrzeb, a także od opóźnień sieciowych i innych czynników.

Nigdy nie dopuszczaj, aby bufor był pusty, bo spowoduje to powstanie zakłóceń podczas odtwarzania dźwięku.

Ostatni parametr konstruktora określa, czy dane akustyczne mają zostać wysłane raz (`MODE_STATIC`), czy jako ciągły strumień (`MODE_STREAM`). W pierwszym przypadku konieczne jest przesłanie całości. Natomiast w przypadku strumieniowania można wysyłać porcje danych PCM o dowolnym rozmiarze, co jest bardzo przydatne w aplikacjach odtwarzających strumienie muzyki i VoIP.

API do rejestrowania dźwięku

Pierwsze API do rejestrowania dźwięku (a także wideo) to `MediaRecorder`. Jest podobne do `MediaPlayer`, bo również utrzymuje wewnętrzny stan, który należy śledzić w kodzie. Za pomocą klasy `MediaRecorder` można tylko zapisywać przechwycone dane w pliku, a więc nie nadaje się ona do użytku w aplikacjach strumieniowych.

Jeśli rejestrowane dźwięki mają być przesyłane strumieniowo, można użyć API `AudioRecord`, które jest bardzo podobne do opisanego wyżej API `AudioTrack`.

Poniżej znajduje się przykład użycia klasy `AudioRecord` do nagrywania 16-bitowego sygnału monofonicznego o częstotliwości próbkowania 16 kHz:

```
public class AudioRecordDemo {

    private final AudioRecord mAudioRecord;
    private final int mMinBufferSize;
    private boolean mDoRecord = false;
    public AudioRecordDemo() {
        mMinBufferSize = AudioTrack.getMinBufferSize(16000, AudioFormat.CHANNEL_OUT_MONO,
        ↪AudioFormat.ENCODING_PCM_16BIT);
        mAudioRecord = new AudioRecord(
                MediaRecorder.AudioSource.VOICE_COMMUNICATION, 16000,
                ↪AudioFormat.CHANNEL_IN_MONO, AudioFormat.ENCODING_PCM_16BIT,
                ↪mMinBufferSize * 2);
    }

    public void writeAudioToStream(OutputStream stream) {
        mDoRecord = true;
        mAudioRecord.startRecording();
        byte[] buffer = new byte[mMinBufferSize * 2];
        while(mDoRecord) {
            int bytesWritten = mAudioRecord.read(buffer, 0, buffer.length);
            try {
                stream.write(buffer, 0, bytesWritten);
            } catch (IOException e) {
                // pominięte dla uproszczenia...
                mDoRecord = false;
            }
        }
        mAudioRecord.stop();
        mAudioRecord.release();
    }

    public void stopRecording() {
        mDoRecord = false;
    }
}
```

Jako że kod ten jest bardzo podobny do wcześniej pokazanego z użyciem klasy `AudioTrack`, można go wykorzystać w celu użycia kombinacji tych dwóch klas w aplikacjach VoIP itp.

OpenSL ES dla Androida

Wcześniej napisałem, że istnieją cztery API do odtwarzania dźwięku i trzy do jego nagrywania. Do tej pory opisałem trzy z pierwszej grupy i dwa z drugiej. Natomiast w tej sekcji znajduje się opis ostatniego z tych API, o nazwie OpenSL ES, które służy zarówno do odtwarzania, jak i rejestrowania dźwięku. OpenSL ES to standard opracowywany przez Khronos Group (tę samą grupę, która zajmuje się standardem OpenGL).

API OpenSL ES zawiera niskopoziomowe narzędzia do pracy bezpośrednio ze sprzętem, które zapewniają bardzo małe opóźnienia w odtwarzaniu i nagrywaniu dźwięku. Podczas gdy pozostałe opisywane API mają wygodne interfejsy w Javie, OpenSL ES aktualnie dostępne jest wyłącznie w języku C poprzez NDK Androida. W tej sekcji opisuję tylko rodzimą część API OpenSL ES. Jeśli chcesz dowiedzieć się, jak przy użyciu NDK Androida pisać kod JNI do współpracy z kodem w C, przeczytaj rozdział 14.

> Zakładam, że czytelnik zna podstawy programowania w języku C.

Pierwsza część przykładu użycia biblioteki OpenSL ES zawiera dyrektywy dołączające potrzebne pliki oraz deklaracje globalnych obiektów, które będą używane w dalszej części programu:

```
#include <pthread.h>

// potrzebne do obsługi OpenSL ES
#include <SLES/OpenSLES.h>
#include <SLES/OpenSLES_Android.h>

static pthread_cond_t s_cond;
static pthread_mutex_t s_mutex;

static SLObjectItf engineObject = NULL;
static SLEngineItf engineEngine;
static SLObjectItf outputMixObject = NULL;
static SLObjectItf bqPlayerObject = NULL;
static SLPlayItf bqPlayerPlay;
static SLAndroidSimpleBufferQueueItf bqPlayerBufferQueue;
```

Poniżej znajduje się funkcja zwrotna `bqPlayerCallback()`, która jest wywoływana przez OpenSL ES przy odtwarzaniu próbki:

```
static void waitForPlayerCallback()
{
    pthread_mutex_lock(&s_mutex);
    pthread_cond_wait(&s_cond, &s_mutex);
    pthread_mutex_unlock(&s_mutex);
}

SLresult enqueueNextSample(short* sample, int size, short waitForCallback)
{
    if(waitForCallback)
    {
        waitForPlayerCallback();
    }
    return (*bqPlayerBufferQueue)->Enqueue(bqPlayerBufferQueue,
                                           nextBuffer,
                                           nextSize);
}

void bqPlayerCallback(SLAndroidSimpleBufferQueueItf bq, void *context)
{
    pthread_cond_signal(&s_cond);
}
```

W tym przypadku przekazujemy bufor wyjściowy z innego wątku poprzez funkcję `enqueueNextSample()`. Do synchronizacji wątku, w którym następuje wywołanie zwrotne, z wątkiem wywołującym funkcję `enqueue` użyliśmy `pthread_mutex`. Wywołanie funkcji `waitForPlayerCallback()` blokuje wątek do czasu wywołania funkcji `pthread_cond_signal()` w wywołaniu zwrotnym (podobnie działa blokowanie metodą `Object.wait()` w Javie).

W poniższym kodzie następuje inicjacja silnika OpenSL ES:

```
SLresult initOpenSLES()
{
    // Służy do sprawdzania wyniku każdej operacji...
    SLresult result;

    int speakers;
```

```
    int channels = 2;

    // Najpierw tworzymy muteks, który będzie potrzebny do odtwarzania później.
    pthread_cond_init(&s_cond, NULL);
    pthread_mutex_init(&s_mutex, NULL);

    // utworzenie i realizacja silnika
    result = slCreateEngine(&engineObject, 0, NULL, 0, NULL, NULL);
    if(result != SL_RESULT_SUCCESS) return result;
    result = (*engineObject)->Realize(engineObject, SL_BOOLEAN_FALSE);
    if(result != SL_RESULT_SUCCESS) return result;
    result = (*engineObject)->GetInterface(engineObject, SL_IID_ENGINE, &engineEngine);
    if(result != SL_RESULT_SUCCESS) return result;

    // utworzenie i realizacja miksera wyjściowego
    const SLInterfaceID outputIds[1] = {SL_IID_VOLUME};
    const SLboolean outputReq[1] = {SL_BOOLEAN_FALSE};
    result = (*engineEngine)->CreateOutputMix(engineEngine, &outputMixObject, 1, outputIds,
    ↪outputReq);
    if(result != SL_RESULT_SUCCESS) return result;
    result = (*outputMixObject)->Realize(outputMixObject, SL_BOOLEAN_FALSE);
    if(result != SL_RESULT_SUCCESS) return result;

    // utworzenie bufora wyjściowego i ujścia
    SLDataLocator_AndroidSimpleBufferQueue bufferQueue =
    ↪{SL_DATALOCATOR_ANDROIDSIMPLEBUFFERQUEUE, 2};
    speakers = SL_SPEAKER_FRONT_LEFT | SL_SPEAKER_FRONT_RIGHT;
    SLDataFormat_PCM formatPcm = {SL_DATAFORMAT_PCM, channels, SL_SAMPLINGRATE_44_1,
    ↪SL_PCMSAMPLEFORMAT_FIXED_16, SL_PCMSAMPLEFORMAT_FIXED_16, speakers,
    ↪SL_BYTEORDER_LITTLEENDIAN};
    SLDataSource audioSource = {&bufferQueue, &formatPcm};
    SLDataLocator_OutputMix dataLocOutputMix = {SL_DATALOCATOR_OUTPUTMIX, outputMixObject};
    SLDataSink audioSink = {&dataLocOutputMix, NULL};

    // utworzenie i realizacja obiektu odtwarzacza
    const SLInterfaceID playerIds[] = {SL_IID_ANDROIDSIMPLEBUFFERQUEUE};
    const SLboolean playerReq[] = {SL_BOOLEAN_TRUE};
    result = (*engineEngine)->CreateAudioPlayer(engineEngine, &bqPlayerObject, &audioSource,
    ↪&audioSink, 1, playerIds, playerReq);
    if(result != SL_RESULT_SUCCESS) return result;
    result = (*bqPlayerObject)->Realize(bqPlayerObject, SL_BOOLEAN_FALSE);
    if(result != SL_RESULT_SUCCESS) return result;
    result = (*bqPlayerObject)->GetInterface(bqPlayerObject, SL_IID_PLAY, &bqPlayerPlay);
    if(result != SL_RESULT_SUCCESS) return result;

    // pobranie obiektu kolejki bufora odtwarzacza
    result = (*bqPlayerObject)->GetInterface(bqPlayerObject, SL_IID_ANDROIDSIMPLEBUFFERQUEUE,
    ↪&bqPlayerBufferQueue);
    if(result != SL_RESULT_SUCCESS) return result;

    // rejestracja funkcji zwrotnej
    result = (*bqPlayerBufferQueue)->RegisterCallback(bqPlayerBufferQueue, bqPlayerCallback,
    ↪NULL);
    if(result != SL_RESULT_SUCCESS) return result;

    return SL_RESULT_SUCCESS;
}
```

Wszystkie inicjacje w tym kodzie są wykonywane wg jednej podstawowej zasady: najpierw **tworzy** się obiekt (np. silnik), a następnie się go **realizuje**. Potem pobiera się **interfejs** do sterowania tym obiektem.

W powyższym przykładzie najpierw zostały utworzone obiekty silnika i odtwarzacza. Ponadto powstał obiekt bufora na próbki dźwiękowe przeznaczone do odtworzenia. Odtwarzacz jest przystosowany do odtwarzania 16-bitowego stereofonicznego (dwa kanały) dźwięku o częstotliwości 44,1 kHz. Na końcu znajduje się funkcja zwrotna dla odtwarzacza próbek, która zostanie wywołana, gdy odtwarzacz będzie gotowy do odbioru nowej próbki do odtworzenia.

Zwróć też uwagę, że na początku tej funkcji tworzone są obiekty związane z muteksem użytym w funkcji zwrotnej i w funkcji dodającej nowe próbki.

Ostatnia część tego przykładu zawiera dwie funkcje. Jedna służy do kontrolowania stanu obiektu odtwarzacza (odtwarzanie i wstrzymanie), a druga do jego zamykania i porządkowania zasobów.

Poniżej znajduje się przykład użycia biblioteki OpenSL ES do odtwarzania dźwięków z bardzo małym opóźnieniem. API to jest niezastąpione, gdy wymagane są ekstremalnie krótkie opóźnienia oraz gdy większość kodu jest już gotowa i napisana w C. Do nagrywania używa się tego samego API w bardzo podobny sposób jak do odtwarzania, z tym że należy posługiwać się innymi interfejsami.

```
SLresult pausePlayback()
{
    return (*bqPlayerPlay)->SetPlayState(bqPlayerPlay, SL_PLAYSTATE_PAUSED);
}

SLresult startPlayback()
{
    return (*bqPlayerPlay)->SetPlayState(bqPlayerPlay, SL_PLAYSTATE_PLAYING);
}

void shutdownOpenSLES()
{
    if (bqPlayerObject != NULL) {
        (*bqPlayerObject)->Destroy(bqPlayerObject);
        bqPlayerObject = NULL;
        bqPlayerPlay = NULL;
        bqPlayerBufferQueue = NULL;
    }

    if (outputMixObject != NULL) {
        (*outputMixObject)->Destroy(outputMixObject);
        outputMixObject = NULL;
    }

    if (engineObject != NULL) {
        (*engineObject)->Destroy(engineObject);
        engineObject = NULL;
        engineEngine = NULL;
    }

    pthread_cond_destroy(&s_cond);
    pthread_mutex_destroy(&s_mutex);
}
```

Czytanie tekstu na głos

Elementy wizualne pozwalają najszybciej dostarczyć informacje do użytkownika, wymagają one jednak jego pełnej uwagi. Jeśli użytkownik nie może patrzeć na urządzenie, trzeba znaleźć inny sposób, aby się z nim porozumiał. W systemie Android dostępne jest API do odczytywania tekstu na głos (TTS), za

pomocą którego można zastosować w aplikacji głosowe powiadomienia, niewymagające od użytkownika spoglądania na ekran.

Poniżej znajduje się przykład użycia API TTS:

```
public class TextToSpeechDemo implements TextToSpeech.OnInitListener {
    private final TextToSpeech mTextToSpeech;
    // do kolejkowania wiadomości zanim nastąpi inicjacja silnika TTS...
    private final ConcurrentLinkedQueue<String> mBufferedMessages;
    private Context mContext;
    private boolean mIsReady;

    public TextToSpeechDemo(Context context) {
        mContext = context;
        mBufferedMessages = new ConcurrentLinkedQueue<String>();
        mTextToSpeech = new TextToSpeech(mContext, this);
    }

    @Override
    public void onInit(int status) {
        if (status == TextToSpeech.SUCCESS) {
            mTextToSpeech.setLanguage(Locale.ENGLISH);
            synchronized (this) {
                mIsReady = true;
                for (String bufferedMessage : mBufferedMessages) {
                    speakText(bufferedMessage);
                }
                mBufferedMessages.clear();
            }
        }
    }

    public void release() {
        synchronized (this) {
            mTextToSpeech.shutdown();
            mIsReady = false;
        }
    }

    public void notifyNewMessages(int messageCount) {
        String message = mContext.getResources().getQuantityString(R.plurals.msg_count,
        ↪messageCount, messageCount);
        synchronized (this) {
            if (mIsReady) {
                speakText(message);
            } else {
                mBufferedMessages.add(message);
            }
        }
    }

    private void speakText(String message) {
        HashMap<String, String> params = new HashMap<String, String>();
        params.put(TextToSpeech.Engine.KEY_PARAM_STREAM, "STREAM_NOTIFICATION");
        mTextToSpeech.speak(message, TextToSpeech.QUEUE_ADD, params);
        mTextToSpeech.playSilence(100, TextToSpeech.QUEUE_ADD, params);
    }
}
```

Dzięki temu, że silnik TTS jest inicjowany asynchronicznie, można kolejkować wiadomości zanim wykona się operację odczytu tekstu na głos.

Do silnika TTS można wysłać kilka parametrów. W powyższym przykładzie pokazałem, jak wybrać strumień dźwiękowy dla wiadomości głosowej. W tym przypadku użyto tego samego strumienia co dla dźwięków powiadomień.

Jeśli chcesz „wygłosić" kilka wiadomości, to po każdej z nich powinna następować krótka przerwa. Umożliwi to użytkownikowi rozróżnienie poszczególnych komunikatów.

Rozpoznawanie mowy

Oprócz syntezy mowy Android umożliwia także jej rozpoznawanie. Ta funkcja jest nieco bardziej skomplikowana i jest dostępna w mniejszej liczbie języków niż API TTS, ale i tak umożliwia zastosowanie alternatywnego sposobu komunikacji z programem, gdy funkcje dotykowe są ograniczone.

Uwaga: aby używać funkcji rozpoznawania mowy, należy zadeklarować użytkownika z uprawnieniem `android.permission.RECORD_AUDIO`.

Poniżej znajduje się kod, na początku którego tworzony jest obiekt `SpeechRecognizer`, po czym zostaje ustawiona procedura nasłuchująca dla wywołania zwrotnego. Gdy zostaje wywołana procedura nasłuchująca kliknięć `doSpeechRecognition()`, następuje inicjacja rozpoznawania mowy w języku określonym parametrem i z flagą oznaczającą, że w czasie przetwarzania chcemy otrzymywać częściowe wyniki.

```
public class SpeechRecognitionDemo extends Activity {

    private SpeechRecognizer mSpeechRecognizer;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.speech_recognition_demo);
        mSpeechRecognizer = SpeechRecognizer.createSpeechRecognizer(this);
        mSpeechRecognizer.setRecognitionListener(new MyRecognitionListener());
    }

    @Override
    protected void onDestroy() {
        super.onDestroy();
        mSpeechRecognizer.destroy();
    }

    public void doSpeechRecognition(View view) {
        view.setEnabled(false);
        Intent recognitionIntent = new Intent(RecognizerIntent.ACTION_RECOGNIZE_SPEECH);
        recognitionIntent.putExtra(RecognizerIntent.EXTRA_PARTIAL_RESULTS, true);
        recognitionIntent.putExtra(RecognizerIntent.EXTRA_LANGUAGE, "en-US");

        mSpeechRecognizer.startListening(recognitionIntent);
    }

    private class MyRecognitionListener implements RecognitionListener {
        @Override
        public void onReadyForSpeech(Bundle bundle) {
    }
```

```
        @Override
        public void onBeginningOfSpeech() {
             ((TextView) findViewById(R.id.speech_result)).setText("");
        }

        @Override
        public void onRmsChanged(float rmsdB) {
            // nieużywana
        }

        @Override
        public void onBufferReceived(byte[] bytes) {

        }

        @Override
        public void onEndOfSpeech() {
            findViewById(R.id.do_speech_recognition_btn).setEnabled(true);
        }

        @Override
        public void onError(int i) {
            // Coś się nie udało...
            findViewById(R.id.do_speech_recognition_btn).setEnabled(true);
        }

        @Override
        public void onResults(Bundle bundle) {
            ArrayList<String> partialResults =
            ↪bundle.getStringArrayList(SpeechRecognizer.RESULTS_RECOGNITION);
            if (partialResults != null && partialResults.size() > 0) {
                String bestResult = partialResults.get(0);
                 ((TextView) findViewById(R.id.speech_result)).setText(bestResult + ".");
            }
        }

        @Override
        public void onPartialResults(Bundle bundle) {
            ArrayList<String> partialResults =
            ↪bundle.getStringArrayList(SpeechRecognizer.RESULTS_RECOGNITION);
            if (partialResults != null && partialResults.size() > 0) {
                String bestResult = partialResults.get(0);
                 ((TextView) findViewById(R.id.speech_result)).setText(bestResult);
            }
        }

            @Override
            public void onEvent(int i, Bundle bundle) {
                // nieużywana...
            }
        }
    }
```

Procedura nasłuchująca otrzymuje wywołanie każdej metody po kolei. W tym przykładzie wyświetlamy częściowy wynik rozpoznawania (najlepsze dopasowanie) w metodzie `onPartialResults()` i kontynuujemy to aż do otrzymania ostatecznego wyniku w `onResult()`.

Bardziej zaawansowana aplikacja mogłaby interpretować słowa i nasłuchiwać konkretnych poleceń. Mogłaby rozpoznawać mowę, aż użytkownik kazałby jej przestać. Mogłoby to być przydatne w programie do dyktowania, w którym użytkownik mógłby powiedzieć „stop”, aby przerwać na chwilę dyktowanie zdań.

Przetwarzanie filmów przy użyciu OpenGL ES 2.0

System Android obsługuje akcelerację przetwarzania grafiki przy użyciu bibliotek OpenGL ES 2.0 i 3.0. Mimo że biblioteka OpenGL jest już standardowo używana do renderowania i składania interfejsu użytkownika w Androidzie, API to jest bardziej znane z tego, że służy do tworzenia silników dwu- i trójwymiarowych gier. Skoro jednak OpenGL ES 2.0 i nowsze wersje tej biblioteki wymagają do przetwarzania dedykowanego GPU, można jej używać do przetwarzania na bieżąco danych wideo lub z kamery. W tym podrozdziale pokazuję, jak dodawać efekty za pomocą biblioteki OpenGL ES 2.0. Zaczniemy od przykładu z wideo.

Zakładam, że czytelnik zna podstawy OpenGL ES 2.0.

W nowszych wersjach Androida (poczynając od API 11) można używać rozszerzenia o nazwie **tekstura strumieniowa** (ang. *streaming texture*) do przesyłania ciągłych strumieni obrazów, np. wideo, do kontekstu OpenGL ES 2.0 jako tekstury. Z pewnością przykład przedstawiający film odtwarzany na obracającym się sześcianie byłby bardzo ciekawy, ale w praktyce opisywanego API częściej używa się do dodawania efektów wizualnych do filmów. W poniższym kodzie uwzględniłem tylko te części, które dotyczą używania tekstury strumieniowej. Pełne przykłady użycia biblioteki OpenGL ES 2.0 można znaleźć w źródłach wymienionych w sekcji „Dodatkowe źródła informacji" oraz w katalogu z przykładami w SDK Androida.

Poniżej znajduje się początek implementacji `Renderer`, która będzie używana przez `GLSurfaceView`.

```
public class MyVideoRenderer implements GLSurfaceView.Renderer,
↪SurfaceTexture.OnFrameAvailableListener {
private static int GL_TEXTURE_EXTERNAL_OES = 0x8D65;
private MediaPlayer mMediaPlayer;
private float[] mSTMatrix = new float[16];
private int muSTMatrixHandle;
```

Użyte w tym przypadku rozszerzenie GL ma nazwę `GL_TEXTURE_EXTERNAL_OES` i nie jest zdefiniowane w SDK Androida. Ale jest to tylko stała, którą definiuje się w kodzie, tak jak pokazano. Renderer nasz zawiera też odwołanie do obiektu `MediaPlayer`, który będzie używany do odtwarzania wideo. Na końcu znajdują się definicje macierzy przekształcenia i odpowiedniego uchwytu shadera. Macierz zostanie później wykorzystana do odwzorowania współrzędnych tekstury otrzymanych z wideo lub kamery w shaderze fragmentów.

W widocznej poniżej metodzie `onSurfaceCreated()` tworzona jest tekstura, która będzie używana na wejściu.

```
public void onSurfaceCreated(GL10 glUnused, EGLConfig config) {
    ...fragment opuszczony dla uproszczenia...

    muSTMatrixHandle = GLES20.glGetUniformLocation(mProgram, "uSTMatrix");
    checkGlError("glGetUniformLocation uSTMatrix");
    if (muSTMatrixHandle == -1) {
        throw new RuntimeException("Nie można pobrać uSTMatrix");
    }

    // tworzenie tekstury
    int[] textures = new int[1];
    GLES20.glGenTextures(1, textures, 0);
    mTextureID = textures[0];
    GLES20.glBindTexture(GL_TEXTURE_EXTERNAL_OES, mTextureID);
```

```
        GLES20.glTexParameterf(GL_TEXTURE_EXTERNAL_OES,
                               GLES20.GL_TEXTURE_MIN_FILTER,
                               GLES20.GL_NEAREST);
        GLES20.glTexParameterf(GL_TEXTURE_EXTERNAL_OES,
                               GLES20.GL_TEXTURE_MAG_FILTER,
                               GLES20.GL_LINEAR);

        // definicja obiektu SurfaceTexture i przypisanie go do MediaPlayer
        mSurface = new SurfaceTexture(mTextureID);
        mSurface.setOnFrameAvailableListener(this);
        Surface surface = new Surface(mSurface);
        mMediaPlayer.setSurface(surface);
        surface.release();

        synchronized (this) {
            updateSurface = false;
        }

        mMediaPlayer.start();
    }
```

Zamiast `GLES20.GL_TEXTURE2D` użyłem wcześniej zdefiniowanej stałej. Następnie utworzyłem obiekt klasy `SurfaceTexture` i dodałem go do obiektu klasy `Surface`, który to obiekt jest używany jako powierzchnia renderingu przez `MediaPlayer`. Dzięki temu każda klatka filmu będzie renderowana w teksturze strumieniowej, a nie na ekranie. Ponadto dodałem wywołanie zwrotne aktualizujące zmienną `updateSurface`, gdy pojawia się nowa klatka. Na końcu włączyłem odtwarzanie filmu w `MediaPlayer`.

W widocznej poniżej metodzie `onDrawFrame()` najpierw sprawdzam, czy dostępna jest nowa klatka. Jeśli tak, następuje aktualizacja `SurfaceTexture` i macierz przekształcenia odpowiednia dla tej klatki zostaje zapisana w `mSTMatrix`. Zostanie ona użyta do przekształcenia współrzędnych tekstury na odpowiednią lokalizację próbkowania w klatce. Zwróć uwagę, że także wiązanie tekstury zostało zmienione na stałą `GL_TEXTURE_EXTERNAL_OES`.

```
    public void onDrawFrame(GL10 glUnused) {
        synchronized (this) {
            if (updateSurface) {
                mSurface.updateTexImage();
                mSurface.getTransformMatrix(mSTMatrix);
                updateSurface = false;
            }
        }

        ...fragment opuszczony dla uproszczenia...

        GLES20.glActiveTexture(GLES20.GL_TEXTURE0);
        GLES20.glBindTexture(GL_TEXTURE_EXTERNAL_OES, mTextureID);

        GLES20.glUniformMatrix4fv(muMVPMatrixHandle, 1, false, mMVPMatrix, 0);
        GLES20.glUniformMatrix4fv(muSTMatrixHandle, 1, false, mSTMatrix, 0);

        GLES20.glDrawArrays(GLES20.GL_TRIANGLE_STRIP, 0, 4);
        checkGlError("glDrawArrays");
        GLES20.glFinish();
    }
```

Poniżej znajduje się kod źródłowy użytego w tym przykładzie shadera wierzchołków. W shaderze tym przy użyciu zmiennej `uniform uSTMatrix` w połączeniu ze zwykłą macierzą przekształcenia i pozycją wierzchołka obliczana jest ostateczna wartość `vTextureCoord`.

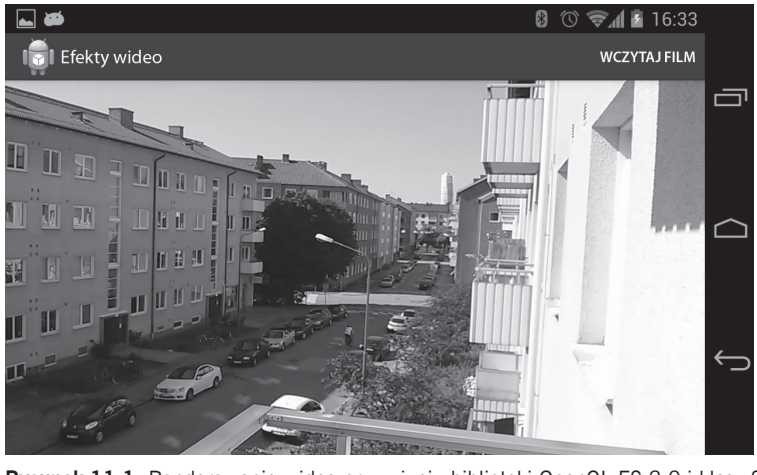
```
uniform mat4 uMVPMatrix;
uniform mat4 uSTMatrix;
attribute vec4 aPosition;
attribute vec4 aTextureCoord;
varying vec2 vTextureCoord;
void main() {
    gl_Position = uMVPMatrix * aPosition;
    vTextureCoord = (uSTMatrix * aTextureCoord).xy;
}
```

Shader fragmentów rozpoczyna się od zaznaczenia, że do jego działania potrzebne jest rozszerzenie `GL_OES_EGL_image_external`. Następnie `sTexture` zmienia typ na `samplerExternalOES`.

```
#extension GL_OES_EGL_image_external : require precision mediump float;
varying vec2 vTextureCoord;
uniform samplerExternalOES sTexture;

void main() {
    gl_FragColor = texture2D(sTexture, vTextureCoord);
}
```

Powoduje to, że obraz wideo jest renderowany na ekranie w pierwotnym stanie, tzn. bez żadnych zmian, jak widać na rysunku 11.1.



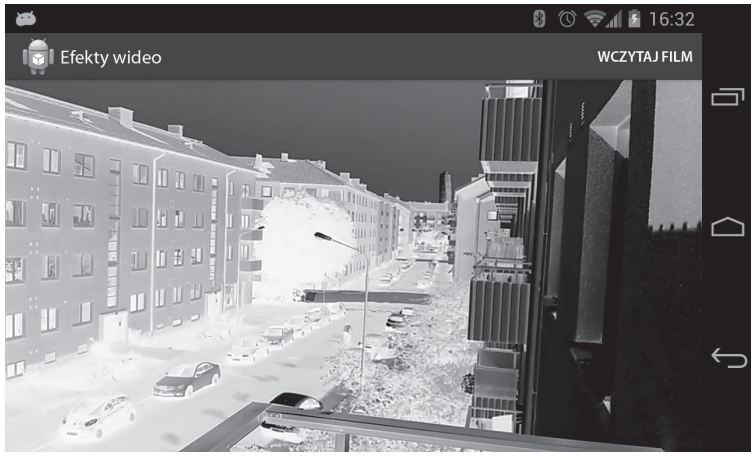
Rysunek 11.1. Renderowanie wideo przy użyciu biblioteki OpenGL ES 2.0 i klasy SurfaceTexture

Teraz wystarczy tylko zmienić coś w shaderze fragmentów, aby uzyskać różne ciekawe efekty wizualne.

Poniższy kod stosuje na bieżąco do filmu efekt **negatywu** (rysunek 11.2):

```
#extension GL_OES_EGL_image_external : require precision mediump float;
varying vec2 vTextureCoord;
uniform samplerExternalOES sTexture;
uniform float uResS;
uniform float uResT;

void main() {
    vec2 onePixel = vec2(1.0 / uResS, 1.0 / uResT);
    float T = 1.0;
    vec2 st = vTextureCoord.st;
```



Rysunek 11.2. Renderowanie wideo przy użyciu biblioteki OpenGL ES 2.0 i filtru dającego efekt negatywu

```
        vec3 irgb = texture2D(sTexture, st).rgb;
        vec3 neg = vec3(1., 1., 1.)-irgb;
        gl_FragColor = vec4(mix(irgb, neg, T), 1.);
    }
```

Ten shader pobiera dwa dodatkowe parametry określające rozmiar tekstury. Parametry te są przydatne także w innych rodzajach filtrów. Więcej informacji na ten temat można znaleźć w „Dodatkowych źródłach informacji" wymienionych na końcu tego rozdziału.

Przetwarzanie obrazu z kamery przy użyciu OpenGL ES 2.0

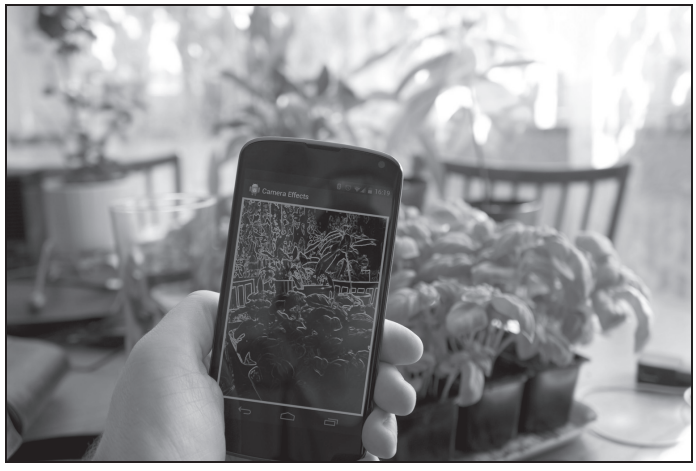
Przy użyciu biblioteki OpenGL ES 2.0 można przetwarzać obraz z kamery w taki sam sposób jak dane wideo. Można podłączyć strumień klatek podglądu do tego samego rodzaju tekstury strumieniowej i przetwarzać te klatki na bieżąco. W ten sposób tworzy się aplikacje rzeczywistości rozszerzonej (ang. *augmented reality* — AR), stosuje się na bieżąco filtry oraz wykonuje skomplikowane obliczenia na obrazach, których wykonanie przy użyciu CPU zajęłoby znacznie więcej czasu.

```
mSurface = new SurfaceTexture(mTextureID);
mSurface.setOnFrameAvailableListener(this);

try {
    mCamera.setPreviewTexture(mSurface);
    mCamera.startPreview();
} catch (IOException e) {
    e.printStackTrace();
}
```

Jedyna różnica w stosunku do poprzedniego przykładu znajduje się w metodzie `onDrawFrame()`, w której nie jest używany obiekt klasy `MediaPlayer`, ale bezpośrednio przekazuje się `SurfaceTexture` do `Camera` jako teksturę podglądu.

Użyty w tym przykładzie shader fragmentów wykrywa krawędzie klatek podglądu. Wynik tego działania jest pokazany na rysunku 11.3.



Rysunek 11.3. Efekt wykrywania krawędzi w klatkach podglądu z kamery w telefonie Nexus 4

Poniżej znajduje się kod źródłowy opisywanego shadera.

```
#extension GL_OES_EGL_image_external : require precision mediump float;
varying vec2 vTextureCoord;
uniform samplerExternalOES sTexture;
uniform float uResS;
uniform float uResT;

void main() {
    vec3 irgb = texture2D(sTexture, vTextureCoord).rgb;
    float ResS = uResS;
    float ResT = uResT;
    vec2 stp0 = vec2(1./ResS, 0.);
    vec2 st0p = vec2(0., 1./ResT);
    vec2 stpp = vec2(1./ResS, 1./ResT);
    vec2 stpm = vec2(1./ResS, -1./ResT);
    const vec3 W = vec3(0.2125, 0.7154, 0.0721);
    float i00 = dot(texture2D(sTexture, vTextureCoord).rgb, W);
    float im1m1 = dot(texture2D(sTexture, vTextureCoord-stpp).rgb, W);
    float ip1p1 = dot(texture2D(sTexture, vTextureCoord+stpp).rgb, W);
    float im1p1 = dot(texture2D(sTexture, vTextureCoord-stpm).rgb, W);
    float ip1m1 = dot(texture2D(sTexture, vTextureCoord+stpm).rgb, W);
    float im10 = dot(texture2D(sTexture, vTextureCoord-stp0).rgb, W);
    float ip10 = dot(texture2D(sTexture, vTextureCoord+stp0).rgb, W);
    float i0m1 = dot(texture2D(sTexture, vTextureCoord-st0p).rgb, W);
    float i0p1 = dot(texture2D(sTexture, vTextureCoord+st0p).rgb, W);
    float h = -1.*im1p1 - 2.*i0p1 - 1.*ip1p1 + 1.*im1m1 + 2.*i0m1 + 1.*ip1m1;
    float v = -1.*im1m1 - 2.*im10 - 1.*im1p1 + 1.*ip1m1 + 2.*ip10 + 1.*ip1p1;
    float mag = length(vec2(h, v));
    vec3 target = vec3(mag, mag, mag);
    gl_FragColor = vec4(mix(irgb, target, 1.0),1.);
}
```

Kodowanie mediów

Android 4.3, a dokładniej API 18, zawiera kilka udoskonaleń w API mediów. Dwie ważne klasy tego API to `MediaCodec` i `MediaMuxer`. Pierwsza z nich, wprowadzona we wcześniejszej, bardziej ograniczonej wersji

Androida 4.2 (API 16), umożliwia dostęp do niskopoziomowych funkcji kodowania mediów. W API 18 klasa `MediaCodec` dodatkowo obsługuje kodowanie z `Surface`, co oznacza, że można przy jej użyciu rejestrować sceny OpenGL ES 2.0 do strumienia wideo.

Klasa `MediaMuxer` służy do multipleksowania surowych strumieni medialnych do plików przeznaczonych do odtwarzania. Przy użyciu tych dwóch klas można np. dodać do gry możliwość nagrywania rozgrywki albo, w połączeniu z poprzednimi przykładami przetwarzania wideo i obrazu z kamery, zapisać wynik w pliku MP4.

Nagrywanie sceny OpenGL

Poniżej znajduje się prosty przykład użycia klas `MediaCodec` i `MediaMuxer` do nagrywania scen OpenGL ES 2.0 do pliku MP4. Metoda ta powinna działać z każdym rodzajem treści OpenGL ES 2.0. W kodzie tym tworzone są koder i multiplekser. Ponadto został w nim użyty obiekt typu `MediaFormat` do określenia parametrów kodera:

```
private void prepareEncoder() {
    mBufferInfo = new MediaCodec.BufferInfo();
    MediaFormat format = MediaFormat.createVideoFormat("video/avc", VIDEO_WIDTH,
    ↪VIDEO_HEIGHT);
    format.setInteger(MediaFormat.KEY_COLOR_FORMAT,
    ↪MediaCodecInfo.CodecCapabilities.COLOR_FormatSurface);
    format.setInteger(MediaFormat.KEY_BIT_RATE, 6000000); // 6 Mbps
    format.setInteger(MediaFormat.KEY_FRAME_RATE, 30);
    format.setInteger(MediaFormat.KEY_I_FRAME_INTERVAL, 10);

    // utworzenie kodera MediaCodec i konfiguracja go przy użyciu naszego formatu
    mEncoder = MediaCodec.createEncoderByType(MIME_TYPE);
    mEncoder.configure(format, null, null, MediaCodec.CONFIGURE_FLAG_ENCODE);

    mMuxer = new MediaMuxer(mOutputFile.getAbsolutePath(),
    ↪MediaMuxer.OutputFormat.MUXER_OUTPUT_MPEG_4);
    mSurface = mEncoder.createInputSurface();
}
```

W powyższym przykładzie na wyjściu użyto formatu H.264, z szybkością transmisji bitów 6 Mbps, częstotliwością klatek 30 na sekundę oraz 10 klatkami między każdą parą klatek typu i-frame. Ponadto został utworzony obiekt `Surface`, który będzie przekazywany na wejściu do kodera.

Po utworzeniu kodera i multipleksera kolejnym krokiem jest utworzenie kontekstu EGL, który zostanie użyty do nagrywania:

```
private static final int EGL_RECORDABLE_ANDROID = 0x3142;

private void recorderEglSetup() {
    mEGLDisplay = EGL14.eglGetDisplay(EGL14.EGL_DEFAULT_DISPLAY);
    if (mEGLDisplay == EGL14.EGL_NO_DISPLAY) {
        throw new RuntimeException("Nieudany dostęp do ekranu EGL!");
    }

    int[] version = new int[2];
    if (!EGL14.eglInitialize(mEGLDisplay, version, 0, version, 1)) {
        mEGLDisplay = null;
        throw new RuntimeException("Błąd inicjacji EGL!");
    }
```

```
        int[] attribList = {
                EGL14.EGL_RED_SIZE, 8,
                EGL14.EGL_GREEN_SIZE, 8,
                EGL14.EGL_BLUE_SIZE, 8,
                EGL14.EGL_RENDERABLE_TYPE, EGL14.EGL_OPENGL_ES2_BIT,
                EGL_RECORDABLE_ANDROID, 1,
                EGL14.EGL_NONE
        };
        EGLConfig[] configs = new EGLConfig[1];
        int[] numConfigs = new int[1];
        if (!EGL14.eglChooseConfig(mEGLDisplay, attribList, 0, configs, 0, configs.length,
                                   numConfigs, 0)) {
            throw new RuntimeException("Błąd konfiguracji EGL!");
        }

        int[] glAttribs = {
                EGL14.EGL_CONTEXT_CLIENT_VERSION, 2,
                EGL14.EGL_NONE
        };
        mEGLContext = EGL14.eglCreateContext(mEGLDisplay, configs[0],
        ↪EGL14.eglGetCurrentContext(), glAttribs, 0);
        int[] surfaceAttribs = {
                EGL14.EGL_NONE
        };
        mEGLSurface = EGL14.eglCreateWindowSurface(mEGLDisplay, configs[0], mSurface,
        ↪surfaceAttribs, 0);
    }

    public void releaseRecorder() {
        mEncoder.stop();
        mEncoder.release();
        mEncoder = null;

        mMuxer.stop();
        mMuxer.release();
        mMuxer = null;

        EGL14.eglDestroySurface(mEGLDisplay, mEGLSurface);
        EGL14.eglDestroyContext(mEGLDisplay, mEGLContext);
        EGL14.eglReleaseThread();
        EGL14.eglTerminate(mEGLDisplay);

        mSurface.release();
        mSurface = null;

        mEGLDisplay = null;
        mEGLContext = null;
        mEGLSurface = null;
    }
```

Dwie najważniejsze części tego kodu zostały oznaczone pogrubieniem. Zdefiniowana przed metodą stała informuje Androida, że używany jest kontekst z możliwością rejestracji. Zmienna `mSurface` jest taka sama jak utworzona w metodzie `prepareEncoder()`.

Metoda `release()` musi zostać wywołana po zakończeniu nagrywania, aby uporządkować i zwolnić zajmowane przez aplikację zasoby. Zauważ, że zwolnienie omawianego kontekstu EGL nie powoduje zwolnienia tego samego kontekstu używanego do renderowania na ekranie.

Jako że nagranie sceny wymaga dwóch przebiegów renderowania (raz na fizycznym wyświetlaczu i raz w obiekcie `Surface` używanym do kodowania), potrzebne są dwie metody do zapisywania i przywracania stanu renderingu OpenGL. Poniżej znajduje się kod źródłowy tych metod.

```
private void storeRenderState() {
    System.arraycopy(mProjMatrix, 0, mSavedProjMatrix, 0, mProjMatrix.length);
    mSavedEglDisplay = EGL14.eglGetCurrentDisplay();
    mSavedEglDrawSurface = EGL14.eglGetCurrentSurface(EGL14.EGL_DRAW);
    mSavedEglReadSurface = EGL14.eglGetCurrentSurface(EGL14.EGL_READ);
    mSavedEglContext = EGL14.eglGetCurrentContext();
}

private void restoreRenderState() {
    if (!EGL14.eglMakeCurrent(mSavedEglDisplay,
            mSavedEglDrawSurface,
            mSavedEglReadSurface,
            mSavedEglContext)) {
        throw new RuntimeException("Nieudane wywołanie eglMakeCurrent!");
    }
    System.arraycopy(mSavedProjMatrix, 0, mProjMatrix, 0, mProjMatrix.length);
}
```

Następnie potrzebna będzie też metoda do przesyłania zakodowanego strumienia wideo do multipleksera służącego do zapisywania danych w pliku MP4:

```
private void drainEncoder(boolean endOfStream) {
    if (endOfStream) {
        mEncoder.signalEndOfInputStream();
    }

    ByteBuffer[] encoderOutputBuffers = mEncoder.getOutputBuffers();

    while (true) {
        int encoderStatus = mEncoder.dequeueOutputBuffer(mBufferInfo, 0);
        if (encoderStatus == MediaCodec.INFO_TRY_AGAIN_LATER) {
            break;
        } else if (encoderStatus == MediaCodec.INFO_OUTPUT_BUFFERS_CHANGED) {
            encoderOutputBuffers = mEncoder.getOutputBuffers();
        } else if (encoderStatus == MediaCodec.INFO_OUTPUT_FORMAT_CHANGED) {
            MediaFormat newFormat = mEncoder.getOutputFormat();
            mTrackIndex = mMuxer.addTrack(newFormat);
            mMuxer.start();
            mMuxerStarted = true;
        } else {
            ByteBuffer encodedData = encoderOutputBuffers[encoderStatus];

            if ((mBufferInfo.flags &
                            MediaCodec.BUFFER_FLAG_CODEC_CONFIG) != 0) {
                mBufferInfo.size = 0;
            }

            if (mBufferInfo.size != 0) {
                encodedData.position(mBufferInfo.offset);
                encodedData.limit(mBufferInfo.offset + mBufferInfo.size);

                mMuxer.writeSampleData(mTrackIndex, encodedData, mBufferInfo);
            }

            mEncoder.releaseOutputBuffer(encoderStatus, false);
```

```
            if ((mBufferInfo.flags
                & MediaCodec.BUFFER_FLAG_END_OF_STREAM) != 0) {
                break;
            }
        }
    }
}
```

Użyta w tym kodzie pętla `while` działa, dopóki na wyjściu kodera dostępne są dane.

Poniżej znajduje się typowy przykład implementacji metody `onSurfaceChanged()`, służącej do obsługi sceny OpenGL ES, w której dodatkowo ustawiane są koder i kontekst EGL do nagrywania sceny:

```
public void onSurfaceChanged(GL10 gl10, int width, int height) {
    GLES20.glViewport(0, 0, width, height);
    float ratio = (float) width / height;
    Matrix.frustumM(mProjMatrix, 0, -ratio, ratio, -1, 1, 3, 7);

    prepareEncoder(mContext);
    if (mEncoder != null) {
        storeRenderState();
        recorderEglSetup();
        mEncoder.start();
        if (!EGL14.eglMakeCurrent(mEGLDisplay, mEGLSurface,
                                  mEGLSurface, mEGLContext)) {
            throw new RuntimeException("Błąd w metodzie eglMakeCurrent");
        }
        restoreRenderState();

        mFrameCount = 0;
    }
}
```

Przedstawiona poniżej metoda `onDrawFrame()` najpierw rysuje scenę na głównym wyświetlaczu, a następnie sprawdza, czy koder jest poprawny. Później zmienia kontekst EGL, konfiguruje obszar widoku dla drugiego renderowania i rysuje scenę w tym obszarze.

```
public void onDrawFrame(GL10 gl10) {
    drawFrame();

    if(mEncoder != null) {
        storeRenderState();
        if (!EGL14.eglMakeCurrent(mEGLDisplay, mEGLSurface,
                                  mEGLSurface, mEGLContext)) {
            throw new RuntimeException("Nieudane wywołanie eglMakeCurrent");
        }
        Matrix.orthoM(mProjMatrix, 0, 0, mWidth, 0, mHeight, -1, 1);
        GLES20.glViewport(mViewportXoff, mViewportYoff,
                          mViewportWidth, mViewportHeight);
        drawFrame();
        drainEncoder(false);
        long when = System.nanoTime();
        EGLExt.eglPresentationTimeANDROID(mEGLDisplay, mEGLSurface, when);
        EGL14.eglSwapBuffers(mEGLDisplay, mEGLSurface);
        restoreRenderState();
    }
}
```

Po zakończeniu drugiego rysowania koder jest pusty.

Poniższa metoda ilustruje sposób sygnalizowania, aby zatrzymać koder i zwolnić wszystkie jego zasoby:

```
public void stopRecording() {
    drainEncoder(true);
    releaseRecorder();
}
```

Metodę tę powinno się wywoływać podczas opuszczania aplikacji przez użytkownika albo w momencie, gdy ma nastąpić zatrzymanie nagrywania.

Podsumowanie

W nowszych wersjach Androida znacznie wzbogacono asortyment narzędzi do zaawansowanej obróbki mediów. Przedstawione w tym rozdziale przykłady ilustrują tylko niektóre z tych zaawansowanych API. Można zbudować kompletną aplikację z wybranego pojedynczego przykładu albo połączyć kilka przykładów, aby uzyskać bardziej złożony program.

Wprawdzie przedstawione w tym rozdziale przykłady nie są kompletnymi aplikacjami, ale doświadczony programista powinien sobie z nimi poradzić.

Dodatkowe źródła informacji

Dokumentacja

Dokumentacja API OpenSL ES na stronach Khronos Group: *www.khronos.org/opensles*.

Specyfikacja rozszerzenia OpenGL GL_OES_EGL_image_external:
http://www.khronos.org/registry/gles/extensions/OES/OES_EGL_image_external.txt.

Wpis Yu Lu LittleCheeseCake na temat shaderów OpenGL ES 2.0:
http://littlecheesecake.me/blog/13804700/opengles-shader.

Zastosowania `MediaCodec` i `MediaMuxer` w Androidzie: *http://bigflake.com/mediacodec/*.

Skorowidz

F

G

H

I

J

K

L

Ł

M

N

O

P

R

S

Ś

T

U

V

W

X

Y

Z

Ż

PLATFORMA ANDROID

NOWE WYZWANIA

Android to lider wśród systemów operacyjnych dla telefonów i tabletów. Spotkasz go w większości tego typu urządzeń. Ciągły rozwój Androida sprawił, że jego obecne wersje zapewniają niesamowite możliwości, a przy tym są przyjazne dla programistów. Jeżeli poznałeś już ten system i marzysz o stworzeniu bardziej zaawansowanej aplikacji, to trafiłeś na doskonałą książkę.

W trakcie jej lektury wzbogacisz swoją wiedzę o cenne informacje. Dowiesz się, jak skonfigurować środowisko programistyczne oraz efektywnie wykorzystać język Java do tworzenia aplikacji. Po tym wstępie zaczniesz poznawać bardziej zaawansowane elementy platformy. Nauczysz się korzystać z zasobów, projektować interfejs użytkownika, tworzyć usługi i zadania pracujące w tle oraz odbierać komunikaty i dokonywać zmian w konfiguracji. Ponadto Twoją uwagę powinny przykuć rozdziały poświęcone testom automatycznym, geolokalizacji oraz bezpieczeństwu tworzonych aplikacji. Na koniec został gwóźdź programu — hakowanie platformy Android oraz wykorzystanie ukrytego API. Brzmi intrygująco? I tak właśnie jest!

Dzięki tej książce:

- błyskawicznie zaznajomisz się z podstawami platformy Android
- wykorzystasz potencjał języka Java oraz składników Androida
- zapanujesz nad zasobami
- zbudujesz zaawansowany interfejs użytkownika
- poznasz ukryte API platformy
- opublikujesz aplikację w Google Play

Twoja przepustka do zaawansowanego świata Androida!

helion.pl
księgarnia internetowa

Nr katalogowy: 23913

Księgarnia internetowa:
http://helion.pl

Zamówienia telefoniczne:
0 801 339900
0 601 339900

Helion

Sprawdź najnowsze promocje:
http://helion.pl/promocje
Książki najchętniej czytane:
http://helion.pl/bestsellery
Zamów informacje o nowościach:
http://helion.pl/nowosci

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel.: 32 230 98 63
e-mail: helion@helion.pl
http://helion.pl

Informatyka w najlepszym wydaniu

ISBN 978-83-246-9525-6

9 788324 695256

Cena: 67,00 zł