WISSENSCHAFTLICHE BEITRÄGE 2016
20. Jahrgang (2016)
ISSN: 0949-8214

Wiss Beitr TH Wildau 20:37–43
doi: 10.15771/0949-8214_2016_1_5
Januar 2016

A Modular Mobile Device for Real-Time 3D-Streaming

Ralf Vandenhouten, Richard Fiebelkorn, Andrea Funke*

Abstract

In recent years, 3D movies and streaming films have become increasingly popular. Even mobile devices, such as mobile phones and tablet computers, are becoming more popular, more powerful and have better multimedia capabilities. Nevertheless, compact mobile devices to broadcast live 3D videos in real time are barely available. In this article, we provide a modular and mobile solution that allows 3D video streaming in real time at 25 frames per second and with a resolution of 1280 × 720 pixels (720p). As operating system, we use standard hardware components combined with Android. Furthermore, we will describe the restraints of the development, and how they were solved.

Zusammenfassung

In den letzten Jahren sind 3D-Filme und das Streamen von Filmen immer populärer geworden. Auch mobile Endgeräte, wie Smartphones oder Tablet Computer, nehmen an Verbreitung zu, werden leistungsstärker und verfügen über verbesserte Multimediafähigkeiten. Trotzdem gibt es kaum kompakte mobile Geräte, um live 3D-Videos in Echtzeit zu übertragen. In diesem Artikel stellen wir eine modulare und mobile Lösung vor, die es ermöglicht, 3D-Videos in Echtzeit, bei 25 Bildern pro Sekunde und mit einer Auflösung von 1280 × 720 Pixeln (720p) zu senden. Dabei verwenden wir Standard-Hardware-Komponenten und nutzen Android als Betriebssystem. Des Weiteren beschreiben wir, welche Schwierigkeiten bei der Entwicklung auftraten und wie diese gelöst wurden.

1. Introduction

In recent years, video and live streaming have become a key consumer scenario, and mobile devices have undergone a significant development. Also, 3D-movies became more popular, and, nowadays, many mainstream TVs are able to display 3D-content, and, thus, make this technology accessible to a wider range of consumers.

In this article, we present the creation of a small, wearable mobile device for real-time 3D-video streaming applications. The aim of this project was to develop a device, which can stream 3D-videos in real-time with at least 25 frames per second (fps), and with a high definition resolution of 720p (1280 × 720 pixels). Another aim of this project was to create the device with easily available inexpensive standard hardware. A bidirectional audio streaming option is also implemented on the device for communication purposes. The said device is powered by a battery and uses wireless technology for sending the data through WiFi. The idea behind the device is to build an inexpensive system using standard components to keep it small, to make it wearable, and to make it modular for an easy adaption to new situations and tasks. For instance, two cameras are built into a frame worn on the head, having the same field of view like the person wearing it, and carrying the other parts in a small bag while sending the stream via WiFi. The person watching the stream now sees what the carrier of the device is seeing, and, because of the audio capability, both are able to communicate. One scenario for this could be a guided tour through a museum showing exhibitions. Having a 3D-able display – a 3D-ready TV or PC monitor for example – enables the receiver of the stream to watch it in 3D. It is also conceivable to use a head mounted (virtual reality) display to watch the stream.

Streaming is becoming a common technology, and with such a streaming device a lot of fields of applications are conceivable. Moreover, the availability of a high mobile bandwidth will open up even more opportunities to use streaming, where neither WiFi nor other networks are available. Streaming a video could be done with a standard mobile phone, tablet computer, or any other mobile device (Mantoro et al. 2012). Most of them either do not have the capability to generate a 3D video, cannot stream in real time with 25 fps at a resolution of 720p, or are very expensive. Google's Tango project, for example, has begun to include depth and stereo cameras into mobile devices, but mostly to reconstruct the real world into a 3D-scene and not for 3D video streaming.

Streaming a live 3D-video feed is more complex and computationally more intensive than just sending a 2D-stream or a video file. One of the problems to be solved is to stream

* corresponding author

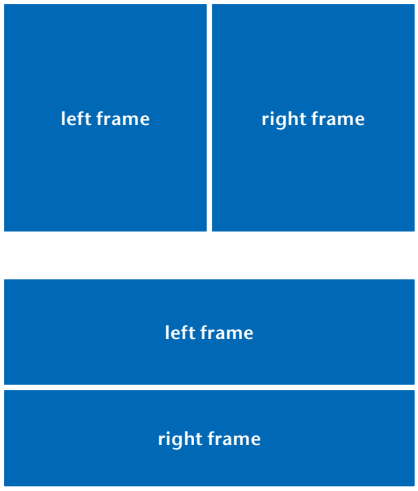

Fig. 1) 3D formats: top: side-by-side; bottom: top-bottom

live 3D at 720p with a low latency at 25 fps. It takes a lot of computing capacity for the hardware to convert and encode a live video. The computer must be able to control two cameras and to process their video data. It also converts the data to create a 3D-video, encodes the result into a 3D-stream, and sends it wirelessly to the receiver.

In the next section, we give some information about 3D-videos related to our work. In section 3, we present our solution in detail, particularly the used hardware and software components. Section 4 describes restraints we encountered during the development. Finally, section 5 concludes this paper and outlines future work.

2. 3D-Video

Existing 3D-video systems are based on stereoscopic technology, which means they use two cameras with a small distance between them – similar to the eyes of a human. To create a 3D-video, both video cameras capture images synchronously, and, depending on the format and video codec used, both streams are combined into a 3D-video. There are different formats for 3D-videos. The difference between them is how they use and store two synchronous video frames to build the 3D-frame while displaying. They can be separated into two types, which give different results in the final resolution. With the first type, both frames are put into one 3D-frame with the same resolution as the source frames. This means the two frames are reduced in a resolution to fit into one frame. This can be done in different manners, and the most common are called *side-by-side* and *top-bottom* (figure 1). With the *side-by-side* method, the horizontal resolution of both frames is reduced to half in order to store the frames for the left and the right sides next to each other horizontally in the final 3D-frame. This means that the two sub-frames, one for each eye, are stored side by side (giving the method its name). This method will lead to a half horizontal resolution for both sub-frames while displaying them. Just like the *side-by-side* format, the *top-bottom* format also does not have the full resolution in the final 3D-video, but instead of reducing the horizontal resolution, the vertical resolution is reduced, and the frames are stored at the top and the bottom of the 3D-frame.

While displaying these formats, the receiver has to recreate full frames from the sub-frames and display them separately for each eye, respectively. Therefore, the player has to extract the frames from the 3D-frame and rescale them, for example by filling the missing columns or rows by using algorithms that interpolate the missing pixels. The advantage of these formats is that they use standard video resolutions (provided the source does, too), and have the same size as one 2D-frame. The required bandwidth is the same as with a normal 2D-video. The disadvantage is the leakage of resolution for each sub-frame. The other format type includes sub-frames with full resolution for each frame. Such formats use both sub-frames with their original resolution to generate a 3D-frame. The result is a frame that has a double height if using the *top-bottom* method or double width if using *side-by-side* method. Also other formats exist, as well as an extension for the H.264/MPEG-4 AVC video compression, or the so-called multi-view video coding (MVC). The MVC format also includes depth of information, and allows for generating new views from the data. After the reconstruction of the two frames, different display technologies are used to display the frames to the eyes separately. The most common ones are:

- **Active Shutter:** In an active shutter system, the frames for the left and for the right eye are displayed in an alternate sequence, i.e. a system – glasses with LCD shutters – blocks the left eye while the frame for the right one is presented and vice versa. This is repeated at high speed in order to make the interruptions invisible for the viewer.
- **Polarization:** This system uses polarization to ensure that the frame for the left eye is only seen by the left eye. This is done by using display technology with different polarizations for the left and the right frame, and glasses using polarizing filters fitting to the corresponding frame.
- **Anaglyph 3D:** With this method, the frames for each eye are encoded by using filters of different colors. Typically red and cyan are used. With glasses using the right filters, it is ensured that each frame only reaches the appropriate eye. An example is shown in figure 2.

Fig. 2) Example of an anaglyph 3D frame (Movie: Big Buck Bunny, 3D version)

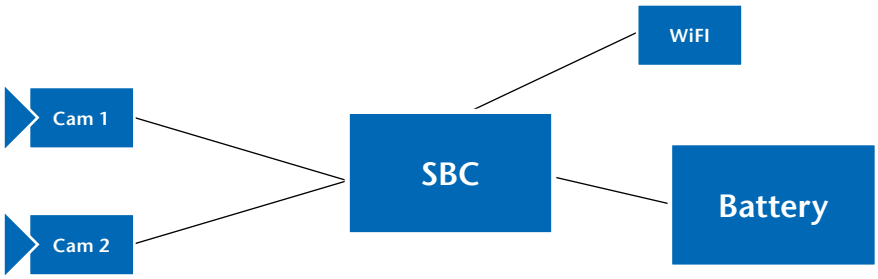

Fig. 3) Sketch of the streaming device

These three methods all use glasses to create the stereoscopic 3D-effect, but there are also some other technologies available, for example head-mounted displays – having a separate display for each eye, or auto-stereoscopic displays. Other important aspects for generating 3D-videos are the position and the orientation of the cameras used. As mentioned before, we use two cameras running synchronously, and in order to obtain a good result for a 3D-video, the position and orientation of the two cameras in relation to one another is important. To create a good 3D-effect for the user, the distance between the cameras is crucial. If the two cameras are too close to each other, there will be no 3D-effect, and if the distance is too long, the impression will not be realistic. A bad camera setup will cause headaches and dizziness because the brain cannot interpret the two views and will get confused while watching the video. In addition to the distance, the orientation is important as well. Both image planes have to be in the same plane and the optical axes have to be parallel, which means the view direction needs to be more or less the same. Moreover, any rotation of the optical axes should be avoided. In addition to the mechanical parameters, also the transmission and the video quality of a 3D-video and displaying it can influence the 3D-user experience (Hewage & Martini 2013).

3. A Modular Wireless Mobile Streaming Device

For the main component of the system, we chose a small computing device – a single board computer (SBC), normally used technology made for mobile applications like mobile phones or tablet computers. For the past few years, the technology has been subject to a high pace of development that results in powerful systems on a chip (SoC) with multicore processors and specialized sub-processors for different tasks like GPU for graphic tasks, or encoder/decoder for different audio and video codecs. We have decided to use a SBC for our project, based on a SoC with a multicore processor and with two USB 2.0 cameras for video capturing. Therefore, the chosen SBC should support at least two independent USB 2.0 hosts. For the WiFi interface, we use a USB-WiFi-Adapter. To run the device wirelessly, we use a Lithium-polymer-battery combined with a step-down-converter as a mobile power-supply. All these components are replaceable, making our device modular and adaptable. A sketch of the device is depicted in figure 3.

As the operating system of the device, we opted for Android. The manufacturer of the single board computer delivers an image file with *Android KitKat* in version 4.4.4. This allows us to use the Android application-programming interface (API) at level 19 of the Android software development kit (SDK) (Android Developers 2015b). This Android image file comes with a kernel that is configured to support V4L2, which makes it possible to access USB cameras through this API, as long as the connected cameras have UVC support. With this setup, we programmed an Android application using the native development kit (NDK) for Android (Android Developers 2015a). We access the two cameras using the native-code languages C and C++. We used the Java Native Interface (JNI) to build a c-library file to use the native code with Java (Oracle 2014). The library file encapsulates the code for controlling the two cameras and for pulling the frames by using the V4L2 API. The cameras deliver MJPEG encoded frames, which is a basic requirement for 25 fps because of bandwidth limitations of USB 2.0. Thus, the frames have to be decoded. Decoding is done in our application by using the free JPEG library distributed by the Independent JPEG Group. All of these software components are encapsulated into a native library that is loaded by our Android application. In the Java part of the application, the frames are pulled simultaneously by using methods offered by the native library and stored on the Java side. In the next step, two corresponding frames, one of each camera, are combined to a 3D-frame. This is done by using the *side-by-side* format, by reducing the horizontal resolution of each frame, and by copying both into a new frame side by side. Using the *side-by-side* format keeps the bandwidth requirements low. Now that the 3D-frame is created, it will be encoded to H.264. Modern SoC for mobile applications come with a myriad of smaller co-processors, usually including one for encoding videos. In general, video streaming is associated with a high-energy consumption (Trestian et al. 2012). Using a more efficient co-processor for encoding the video can reduce the power demand and the CPU load. The Android SDK offers the *MediaCodec* API, which is included in the Android SDK of level 16. This API encapsulates all the low-level media codecs of such a co-processor. This functionality is an advantage of the Android operating system because under Linux, there is often no proper driver support available for using the hardware encoder. We stream the encoded video using the *Real-time Transport Protocol* (RTP) (Schulzrinne et al. 2003) and the *User Datagram Protocol* (UDP) (Eggert & Fairhurst 2008). In the Android SDK, there is no known implementation for streaming video data with these protocols. Accordingly, we implemented our own solution based on an open source library for RTP and UDP streaming. However, since this library did not completely fit our needs, we had to implement additional features and changes. The flowchart in figure 4 gives an overview

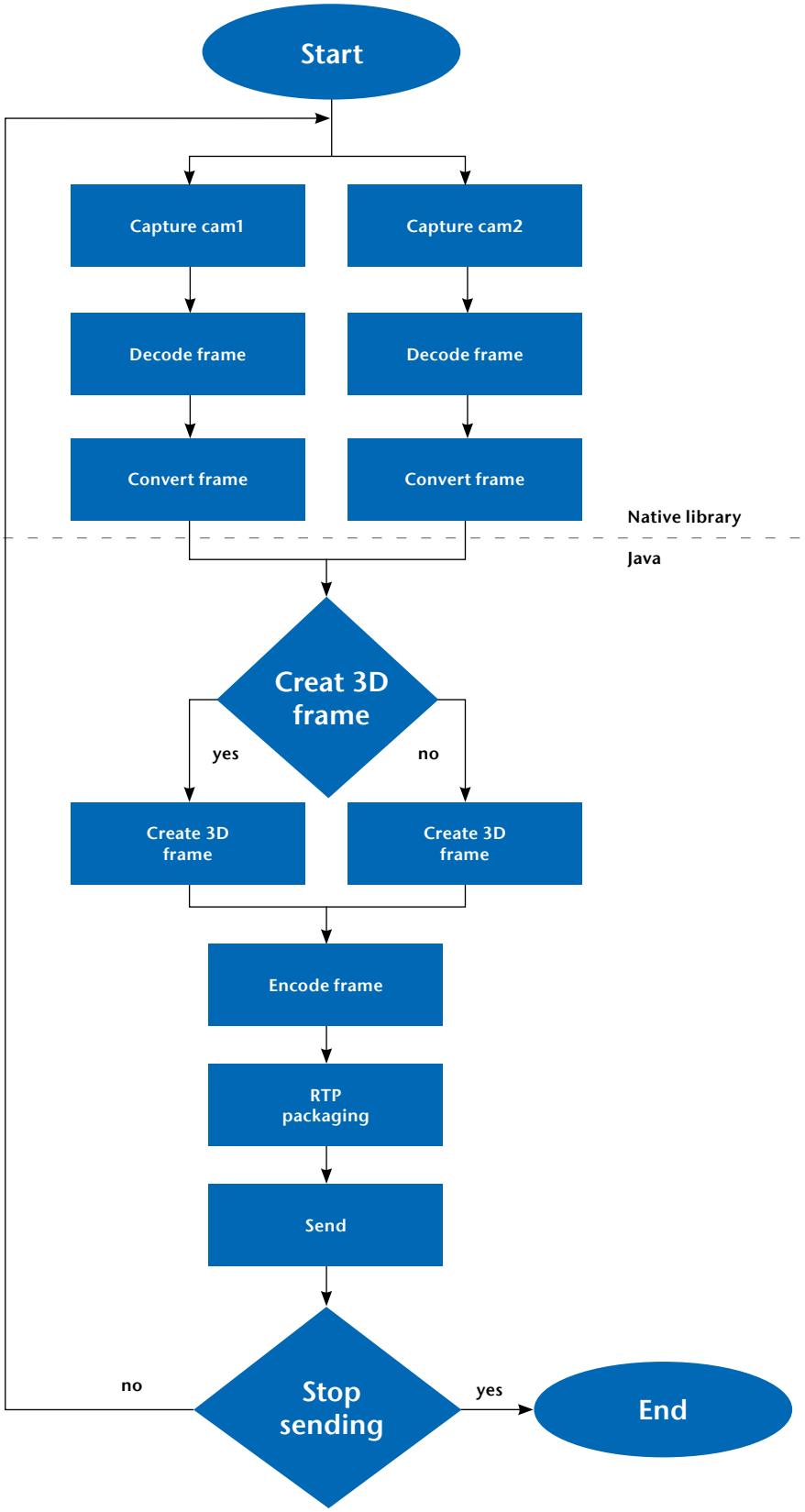

Fig. 4) Creating and sending 3D video

of how the 3D-frame is generated and handled. Figure 5 shows the application and the Android layers.

For the audio capability of our device, we use a connected microphone for recording and a connected speaker for playing sound. To record the audio stream with a microphone and to play the incoming audio-stream, we use the Android SDK. For encoding and decoding audio, we used the Audio Stream package, which is included in the RTP package of the Android SDK. This package supports streaming audio via RTP using different audio codecs.

One important part of streaming is the exchange of ports and addresses used by the sender and by the receiver of the streams. We use the session description protocol (SDP) for this task (Handley et al. 2006). Port assignment for receiving is often done randomly, so we have to make sure that the incoming and outgoing streams are addressed to the right ports. For this reason, it is necessary to tell the sender of the audio stream on which port our device is listening for the audio stream. We also need to know to which port we have to send the video and the outgoing audio stream. This problem is solved by creating an SDP file (figure 6), which includes all the necessary information about the streams, including the ports for sending video data and the used format, receiving and sending audio ports and formats and codecs for all included streams. This file can be used for exchange with a server to build up all the connections, or it can be used by a media player client software, for instance the VLC media player.

Because we aim at existing standards for the 3D video streams, a high compatibility can be guaranteed, and many different displays are supported. A 3D-able display, for example a 3D-ready TV or PC monitor, makes it possible to watch the video in 3D. If no 3D-display is available, the VLC media player can be used to generate an anaglyph 3D-video from the stream.

We configure the device to run automatically as soon as it gets powered up, and after booting Android, our application starts automatically and begins streaming the live 3D-video and audio. It is also possible to switch between the 3D- and the 2D-mode, when only one camera is selected. For the 2D-mode, there is an option to choose the camera to be used. When running the device without any input device or monitor connected, one camera is selected by default. To signal the start of the streaming, the device gives a sound notification.

4. Restraints During Development

To find the right computing device for our task, we had to determine the required specifications first. We wanted to create 3D-video; therefore it was essential that the single board computer had sufficient interfaces to control at least two cameras with the desired resolution and frame rate. To send the data wirelessly, a WiFi interface was also necessary. To have enough computing resources, and to be able to handle all the different tasks simultaneously, a multicore CPU with at least four cores seemed to be appropriate. To connect the cameras to the SBC, different types of interfaces came into consideration. A common type of interface for mobile application is the CSI (camera serial interface) of the MiPi Alliance. There are common devices, like *Rasberry-Pi, PandaBoard,* and *HummingBoard,* which support this interface. Many different boards are available, but the majority of them offer only one or no CSI interface. Another possibility would be to use FPGAs to combine the two camera frames before sending the data to the computing device using only one interface. Lattice Semiconductor offers a solution, which combines the frames of two camera sensors into one and sends the data as if it is only one camera (Lattice Semiconductor 2015). Even though this would be a powerful solution, it does not fit our project's aim to build the device from low-cost standard hardware. Normally, FPGAs are expensive, and using them would increase the development time for programming the FPGA and for designing the electronic parts to use the FPGA.

Due to lack of other options among all the single board computers and camera interfaces, we decided to use USB 2.0 as the camera interface. With USB 2.0, we can use almost any camera available, as long as it supports the required resolution and frame rate – even inexpensive webcams. The camera also had to support UVC for using it with the V4L2 API. Because of the afore-mentioned facts, we selected a single board computer with at least two independent USB hosts. The advantage of the chosen one is that it has at least three independent

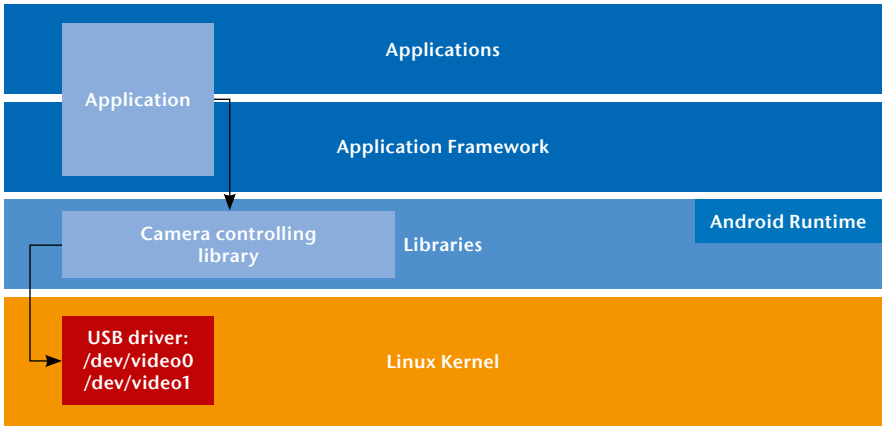

Fig. 5) The application and the Android layers

USB hosts, two USB 3.0 hosts and one USB 2.0. We use two USB hosts for the two cameras, and the third USB host for other peripheries, like the required WiFi receiver or input devices. As mentioned before, the device should run wirelessly; therefore, it also needs a wireless power-supply. The selected computing device needs a high current, so the power-supply delivered with it has an output of 5.0 V and 4.0 A. Thus, we decided to use rechargeable Lithium-polymer batteries. They can supply and sustain high currents, and the handling is uncomplicated. In addition to that, we use a step-down-converter to convert the output voltage of the battery from 7.9 V to 5.4 V.

As we started developing, we used Linux as an operating system, an Ubuntu 14.04.01 Linux image. For our first tests on implementing all the features for streaming, we used the *GStreamer* framework, a useful tool for multimedia tasks in Linux. With *GStreamer* we built different pipelines to test different ways for generating and encoding the 3D-video. We also tested different ways for streaming with *GStreamer* and checked the synchronicity of the two USB cameras with respect to generating a 3D-video stream. Since we used cameras without any hardware trigger that could be used for hardware synchronization, we developed software techniques to synchronize them. The results were positive. The two cameras worked without any issues and were synchronized sufficiently by the software. Only the targeted frame rate and latency could not be reached. While encoding the video using a software encoder included in the *GStreamer* framework, the CPU load was nearly 100 %, and we had a big latency in the video stream (several seconds). The SoC of our single board computer has a co-processor for hardware encod-

```
V=0
O=- 1188340656180883 1 IN IP4 10.220.1.161
S=STREAMINGDEVICE
I=N/A
C=IN IP4 10.220.1.161
T=0 0
A=RECVONLY
M=AUDIO 40004 RTP/AVP 0
A=RTPMAP:0 PCMU/8000
A=CONTROL:TRACKID=0
M=VIDEO 50002 RTP/AVP 96
A=RTPMAP:96 H264/90000
A=FMTP:96 PACKETIZATION-MODE=1;PROFILE-LEVEL-
ID=42801F;SPROP-PARAMETER-SETS=Z0KAH+KBKCB8QA==,AM4G8G==;
A=CONTROL:TRACKID=1
```

Fig. 6) SDP file

ing H.264, but is not supported by the *GStreamer* framework. Trying to implement a proprietary solution or plug-in for *GStreamer* failed as a consequence of lack of documentation and driver support. We were able to use the co-processor to encode frames, but the result did not meet our needs. We could only achieve a latency of about four seconds, and the solution could not be configured.
All these facts led us to the decision to use Android as the operating system. However, while using Android, one issue needed to be solved: accessing the two USB cameras simultaneously. At the beginning we had some problems using the two USB cameras with standard Android camera applications. Then we did some tests with the Android SDK – the Android Camera API. Through this API, only one of the cameras could be accessed because from the design, Android is made for reading only one camera at the time. Hence, we decided to write a native low-level code for accessing and controlling the cameras. Therefore, we implemented a proprietary solution by using the V4L2 API to get both cameras working at the same time, and making them employable for multi-threading applications. In addition, however, a few more adjustments were necessary to control the cameras under Android. To use the cameras with V4L2, we had to change the permissions for the device files corresponding to the cameras. Due to the security features in Android, we needed root access to change any permission of these files. The Android image we use comes with root rights already enabled. Therefore we only had to remount the file system of our Android device to get write access to a system file. To remount the file system, we used the USB-debugging functionality of Android and connected through the Android Debug Bridge (ADB). After establishing the connection with a host computer via USB cable, we could run the ADB from a terminal and used the command:

- *adb shell mount -o remount,rw /system*

to remount the file system. Now that we had write access to the file system, we could edit the system file *ueventd.devicename.rc*, where we had to change two lines – one for each device file. For the first one, we changed the line,

- */dev/video0 0660 system camera* to */dev/video0 0666 system camera*

and accordingly, for the second one:

- */dev/video1 0660 system camera* to */dev/video1 0666 system camera*

Changing the permission from *660* to *666* gives read and write access not only to the owner and group, but also to all other users. The names of the device files can differ on other devices, depending on the operating system and the SoC used. Some SoCs assign a different amount of V4L2 devices.
In some use cases it could be important to secure or encrypt the video stream. Instead of encrypting the video data, as tested in Massandy & Munir (2012), we investigated a VPN connection to secure the video stream. The problem with using VPN connections in Android 4.4 is that the user has to confirm the connection to the VPN through a pop-up dialog. For using our device without a connected display or any input device, we use the application *OpenVPN für Android* by Arne Schwabe to establish the VPN connection (Schwabe 2015). To confirm the VPN connection and the pop-up dialog without user interaction, we use the *Xposed* framework for Android, which includes a feature to automatically confirm a pop-up dialog (Xposed Module Repository 2014). This allowed us to run the device without any connected display or input device while using a secure VPN connection for streaming.

5. Conclusions and Future Work

During this project, we created a modular mobile device based on a single board computer with the capability to stream 3D-HD-videos (1280 × 720 pixels) in real-time with 25 frames per second. Our device consists of two USB cameras, a SBC, a WiFi module, and a mobile power supply. All parts can be included in one small case, or the two cameras can be separated from the rest, connected to the computing device by their USB cables. Since we use standard hardware components, all parts are easily replaceable and adaptations for new requirements and tasks can be done easily.
While our application is running and streaming video and audio data, the CPU load is still less than 50 %. With some more optimization, this value could be decreased to an even lower value. Thus, there are still enough resources left on the computing device for other tasks. In the future, we plan to implement a solution to access the cameras without root rights. In this regard, we investigate a library for accessing USB cameras under Android called *UVCCamera*, which does not need root rights (GitHub 2015). However, during testing we could not reach the target frame rate so far. We are also thinking about implementing a camera calibration to remove any distortion of the camera lens. The aim would be to stream undistorted frames with the full frame rate. A possible extension for our device would be augmented reality to give additional context information to the viewer of the stream.

REFERENCES

Android Developers (2015a) Android NDK. https://developer.android.com/ndk/index.html. Accessed 07 Sep 2015

Android Developers (2015b) Android SDK Reference. Package Index. https://developer.android.com/reference/packages.html. Accessed 08 Sep 2015

Eggert L, Fairhurst G (2008) Unicast UDP Usage Guidelines for Application Designers. IETF RFC 5405. https://tools.ietf.org/html/rfc5405. Accessed 04 Sep 2015

GitHub (2015) UVCCamera. https://github.com/saki4510t/UVCCamera/tree/master. Accessed 11 Sep 2015

Handley M, Jacobson V, Perkins C (2006) SDP: Session Description Protocol. IETF RFC 4566. https://tools.ietf.org/html/rfc4566. Accessed 09 Sep 2015

Hewage CTER, Martini MG (2013) Quality of experience for 3D video streaming. IEEE Comm Mag 51(5):101–107. doi: 10.1109/MCOM.2013.6515053

Lattice Semiconductor (2015) MachXO2 Dual Sensor Interface Board. http://www.latticesemi.com/en/Products/DevelopmentBoardsAndKits/MachXO2DualSensorInterfaceBoard.aspx. Accessed 03 Sep 2015

Mantoro T, Ayu MA, Jatikusumo D (2012) Live video streaming for mobile devices: An application on android platform. In: Proc 2nd Int Conf Uncertainty Reasoning and Knowledge Engineering (URKE), 14–15 Aug 2012, Jalarta, ISBN: 978-1-4673-1459-6, pp 119–122. doi: 10.1109/URKE.2012.6319523

Massandy DT, Munir IR (2012) Secured video streaming development on smartphones with Android platform. In: Proc 7th Int Conf Telecomm Systems Services Applications (TSSA), 30–31 Oct 2012, Denpasar-Bali, ISBN: 978-1-4673-4549-1, pp 339–344. doi: 10.1109/TSSA.2012.6366079

Oracle (2014) Java Native Interface Specification. http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/jniTOC.html. Accessed 08 Sep 2015

Schimek MH, Dirks B, Verkuil H, Rubli M, Walls A, Carvalho Chehab M (2009) Video for Linux Two API Specification. Revision 2.6.32. http://www.linuxtv.org/downloads/legacy/video4linux/API/V4L2_API/spec-single/v4l2.html. Accessed 05 Sep 2015

Schulzrinne H, Casner S, Frederick R, Jacobson V (2003) RTP: A Transport Protocol for Real-Time Applications. IETF RFC 3550. https://tools.ietf.org/html/rfc3550. Accessed 03 Sep 2015

Schwabe A (2015) OpenVPN für Android 0.6.43. https://play.google.com/store/apps/details?id=de.blinkt.openvpn&hl=de. Accessed 11 Sep 2015

Trestian R, Moldovan AN, Ormond O, Muntean G (2012) Energy consumption analysis of video streaming to Android mobile devices. In: Proc IEEE/IFIP Network Operations Management Symp (NOMS), 16–20 Apr 2012, Maui, ISBN: 978-1-4673-0267-8, pp 444–452. doi: 10.1109/NOMS.2012.6211929

Xposed Module Repository (2014) Xposed Installer. http://repo.xposed.info/module/de.robv.android.xposed.installer. Accessed 11 Sep 2015

AUTHORS

Prof. Dr. rer. nat. Ralf Vandenhouten
Dipl.-Ing. (FH) Richard Fiebelkorn
Technische Hochschule Wildau
Forschungsgruppe Telematik

Andrea Funke
eayse GmbH

E-mail for correspondence:
ralf.vandenhouten@th-wildau.de