Computer and Information Science; Vol. 10, No. 2; 2017
ISSN 1913-8989 E-ISSN 1913-8997
Published by Canadian Center of Science and Education

Software Design for Low-Latency Visuo-Auditory Sensory Substitution on Mobile Devices

Maxime Ambard[1]

[1] LEAD CNRS UMR 5022, Université de Bourgogne Franche-Comté, France

Correspondence: Maxime Ambard, Université de Bourgogne Franche-Comté, Dijon, France. Tel: 33-0380-393-907. E-mail: maxime.ambard@u-bourgogne.fr

Received: December 2, 2016 Accepted: December 14, 2016 Online Published: March 2, 2017

doi:10.5539/cis.v10n2p1 URL: https://doi.org/10.5539/cis.v10n2p1

Abstract

Visuo-auditory sensory substitution devices transform a video stream into an audio stream to help visually impaired people in situations where spatial information is required, such as avoiding moving obstacles. In these particular situations, the latency between an event in the real world and its auditory transduction is of paramount importance. In this article, we describe an optimized software architecture for low-latency video-to-audio transduction using current mobile hardware. We explain step-by-step the required computations and we report the corresponding measured latencies. The whole latency is approximately 65 ms with a capture resolution of 160 × 120 at 30 frames-per-second and 1000 sonified pixels per frame.

Keywords: sensory substitution, mobile device, low-latency, vision, audition, sonification

1. Introduction

Visuo-auditory sensory substitution devices (VASSDs) form a specific branch of assistive technologies for the blind that directly transduce visuo-spatial information into spatialized audio cues. These systems can be used by visually impaired people in situations where spatial information is required, such as moving into an unknown environment. The modus operandi is generally the following: acquisition of the visual information using a camera; extraction of the visual features of interest; association of each extracted visual feature with specific audio cues; and stereophonic transmission of the generated audio signal. The audio cues usually encode the spatial position of the spatial features in the Field Of View (FOV). In the basic sonification scheme, sound amplitudes are used to encode the luminosity, and pitch and stereo panning are used to encode, respectively, the vertical and horizontal positions.

Miniaturization is a key feature in VASSD mobility. This aspect constrains the whole design of the system, allowing for only small hardware systems with limited computational capacities. However, the latency between an event in the real world and its transmission in the audio output must be reduced to the minimum time possible for an optimal use of the system. Thus, implementing highly demanding multimedia computation such as video processing and on-line audio synthesis with this type of hardware, while fulfilling a strong latency constraint, requires an optimized software architecture with an efficient processing. In the following, we describe the state of the art in the current development of VASSDs, describing five other VASSDs initiatives.

VASSDs can be separated into two major families depending on their sonification procedure. The first family of systems acquires video frames at low frequency. Each frame is transformed into an audio signal that corresponds to the sonification of a horizontal, left-to-right scan of the frame. During this sonification, the image is decomposed into a sequence of columns and each column is sonified during a fraction of the whole frame period. For example, a frame acquired at 1 frame per second (fps) and decomposed into 32 columns is transduced into an audio signal lasting 1 s that is composed of 32 successive audio signals, each lasting 1/32 s.

Begun in 1992, a pioneer system called *the vOICe* remains one of the current major projects in VASSD development (Ward & Meijer, 2010; Striem-Amit et al., 2012; Brown et al., 2014; Stiles & Shimojo, 2015). It has evolved in line with computer hardware and currently proposes many different settings and implementations.

EyeMusic (Levy-Tzedek et al., 2012; Abboud et al., 2014; Maidenbaum et al., 2014A; Levy-Tzedek et al., 2014) is an initiative which is also based on periodic left-to-right horizontal scans of images. Sounds of musical instruments are produced depending on the colors of the pixels. The vertical dimension is mapped into a musical pentatonic pitch scale. The system has been slightly modified in a recent version (Maidenbaum et al., 2014b) with an increased image resolution of 50 × 30 and a hexatonic scale.

This type of sonification using horizontal scans, has proven useful for specific tasks. It has been shown that users are capable of grabbing a distant object (Levy-Tzedek et al., 2012) and that visual shapes can be differentiated (Brown et al., 2014). The orientation of letters can be recognized, allowing VASSD users to exceed the threshold for the World Health Organization definition of blindness on an adapted Snellen acuity test (Striem-Amit et al., 2012). Recent work has shown that sounds can be efficiently used as a substitute for visual and tactile textures (Stiles & Shimojo, 2015). Moreover, it has been reported that visual phenomenology can be developed within months of immersive use (Ward & Meijer, 2010). However, the low frequency of the video acquisition required by the sonification scheme prevents the system from being used for the sonification of fast moving objects.The second family of VASSDs does not use column-by-column sonification schemes but rather acquires frames at standard video frequency. The visuo-spatial information is extracted from the whole frame and sonified on-line. In comparison to the previous family, this type of sonification emphasizes the timing of a visual event at the cost of the ease of interpretating its spatial information.

The first system of this type is called Prosthesis Substituting Vision with Audition (PSVA) (Capelle et al., 1998; Arno et al., 1999, 2001). The main differences between this project and the previous one are the following: an online sonification without left-to-right scans and frequency encoding depending on both the horizontal and vertical positions of the pixels. Edges are detected from images in gray scale and the amplitude of each sound is modulated by the gray level of the corresponding pixel in the image. A finer spatial resolution is given to the center of images during the conversion.

More recently, another project called TheVibe (Auvray et al., 2005; Durette et al., 2008; Hanneton et al., 2010) has been developed. With its default configuration, this system associates a specific tone to each region of the acquired image. However, the conceptual sonification scheme, based on the coupling of sensors and receptors, allows for a much more complex sonification. As in PSVA, sounds are generated on-line from whole gray-scale images, without horizontal scans.

SeeColOr (Bologna et al., 2009a; Deville et al., 2009; Bologna et al., 2009b; Bologna et al. 2010) applies image filters to a stereoscopic colored video acquisition with a Hue-Saturation-Luminance (HSL) color format. A method, based on the combination of several conspicuity maps obtained from the extraction of different features, generates a simplified video stream that is afterwards sonified. Pre-defined pixel hue ranges are used to associate musical instruments depending on the color of the pixels. The pitch encodes the saturation and a second musical sound is added depending on the luminance. Spatial information in the FOV of the camera is rendered using Head-Related Transfer Functions (HRTF).

The ability to localize static objects with such systems has already been demonstrated (Proulx et al., 2008). It has been shown that the trajectories of moving visual targets can be differentiated following a short training period (Ambard et al., 2015). These systems have proven useful for walking (Bologna et al., 2009a; Durette et al. 2008), and they allow for the development of distal attribution (Auvray et al., 2005).

Despite the multiplicity of initiatives, no recent article has described the technical details of the software implementation required by the specificities of online low-latency video-to-audio transduction on current mobile hardware. This aspect is especially important when the transduction system is dedicated for the sonification of moving objects.

In a previous work, we presented a new VASSD called LibreAudioView belonging to the second family mentioned above and dedicated to object motion sonification (Ambard et al., 2015). We emphasized the benefits of using simple video processing for motion extraction as well as several specificities in the video-to-audio mapping methods. In the present work, we describe a new version of this system, called *LibreAudioView_native* that optimizes the general hardware and software design with the hardware fully integrated into the frame of the glasses, and a specific implementation for low-latency transduction on a mobile system. Compared to the previous version, we now obtain a latency reduced by a factor of 4, decreasing from 230 ms to 65 ms.

In the next sections, we present the hardware and the experimental setup used in this work. We then describe the global software architecture and we describe more precisely some important aspects of the processing.

2. Materials and Method

The video acquisition is performed by a USB video device class (UVC) color camera module with a diagonal FOV of 67° (53.6° horizontal, 40.2° vertical), a maximal video resolution of 640 × 480 pixels at 30 fps providing an auto-exposure control. The System On a Chip (SoC) is based on a RocKship3288 featuring a Quad-core CPU

ARM Cortex-A17, 2Go DDR3 RAM and a Mali-T764 GPU. The battery is a Li-ion rechargeable 1S2P with 2000 mAh capacity providing more than 2 hours of use. A charger-booster module is dedicated to power management.

The miniature camera, the SoC, the battery and the charger-booster are all integrated into a 3D-printed Polylactic acid (PLA) eyeglass frame. Boxes of 95x45x17mm are mounted on the arms of the frame. The SoC is placed on one side and the battery and the charger-booster are placed on the other. Wires used to connect the different parts run inside the frame. The camera is situated between the eyebrows with a pith angle of 20° towards the ground. The audio output is sent using headphones especially designed for urban jogging (such as Sennheiser MX680) with integrated volume control and earbuds that do not completely block the auditory canal, allowing surrounding sounds to be heard. The integrated system containing the camera, the mini battery-powered computer, and the earphones is shown in Figure 1.

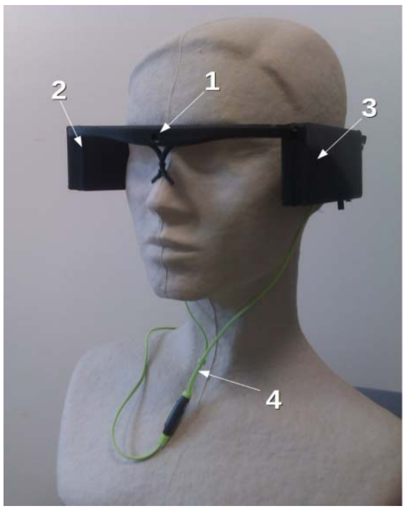


Figure 1. Photograph of the sensory substitution system integrated into the glasses frame. 1 - the camera, 2 - the mini computer, 3 - the battery and the on-off switch, 4 – earbuds

The global latency of the system was measured by clocking the latency between the light switch of an incandescent light bulb placed directly in front of the camera and the beginning of the first sound recorded at the output of the device (see Figure 2 for details).

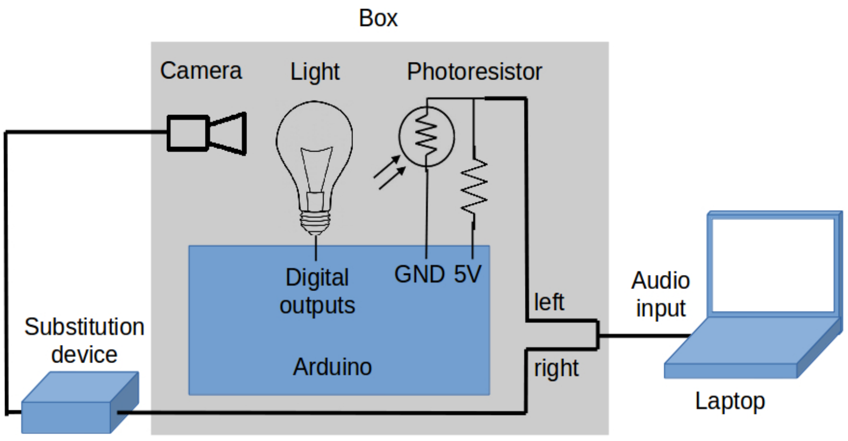


Figure 2. Sketch of the experimental setup used to measure the global system latency

An Arduino controls the on/off switch of an incandescent light bulb in periods of 2 seconds. Signals coming from the activation of a photoresistor and the sound coming from the sensory substitution system are both transmitted through a stereo audio jack that is pugged into the microphone entry of a PC for recording.

The light was successively switched on in 2 second periods by an Arduino Uno. The luminosity was measured by a photoresistor placed close to the camera. The sound output of the substitution system and the voltage of the photoresistor were each connected to one of the channels of the microphone input of a laptop. Since the system only sonifies variation in the video stream (see 3.2 for details concerning the video processing), the audio output remained silent until the light was switched. The luminosity of the light was powerful enough to set the frame rate of the camera at 30 fps after 2 seconds. The frame rate decreased to 15 fps 2 seconds after the light was switched off. Since the global latency partly depends on the frame rate, the measurement of the latency was only taken once the light was switched off. A script was used off-line to automatically measure the latency between the extinction of the light and the appearance of its associated sound in the recorded stereophonic signal at the output of the sensory substitution system.

This measure can be used as a benchmark for the whole latency between an event occurring in the real world and its transduction in the audio output signal. It is the sum of the camera latency, the video processing, the sonification, the mixing of the sound, the thread synchronization, the writing of the Pulse Code Modulation (PCM) data and the sound card latency. We measured the internal durations of specific parts of the software by repeatedly calling the CPU clock before and after the executed code. Two immediate successive calls of the clock generate a latency of 3.4 ± 7.1 µs (min = 1 µs, max = 328µs), which is negligible compared to other measured latencies. Statistics were computed on a collection of 1000 successive measurements. CPU usage was measured using the top shell command. The given percentage refers to the global CPU usage, which is composed of 4 cores. Since a single thread can only use a single core at a time, its maximum usage of the complete CPU is equal to 25%.

Except for a minimal JAVA layer, the software was developed in C. The C part of the software is composed of two major threads: the first one is dedicated to the information processing, including: video acquisition, video processing and sonification. The second one is dedicated to the audio output, including: sound mixing and sound card interactions. The general sketch of the design is presented in Figure 3.

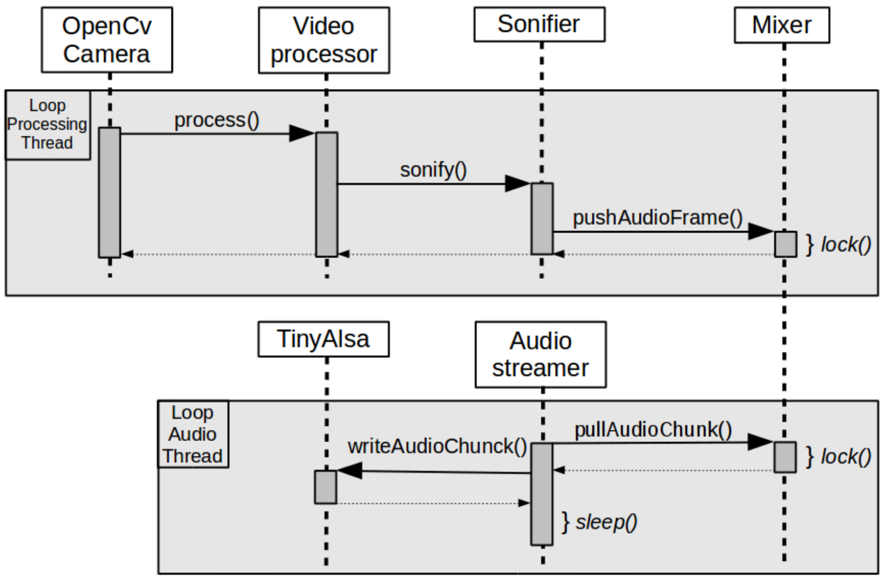


Figure 3. Sequence diagram showing the general scheme of the information processing

When a frame is received from the camera, it is transmitted to the Video processor that simplifies the visuo-spatial information. The simplified frame is then sent to the Sonifier that constructs its audio representation and pushes the data to the Mixer. The Mixer stores and provides the data to the Audio streamer. At its own rhythm, the Audio streamer pulls data from the Mixer and transmits it to the sound card.

Since the system should operate in ecological contexts with various luminosities, the camera was set with an active exposure adaptation. With these settings, the video acquisition frequency depends on luminosity and ranged from 15 fps in a very dark environment to 30 fps in a very bright environment.

Each acquired frame (called a video frame in the rest of the document) is first sent to the video processor. As a result of the video processing, each frame is composed of black and white pixels, the latter (called active pixels in the rest of the document) signaling regions of interest that should be sonified. Each processed frame is then sent to the sonifier that constructs its corresponding audio transduction. During this transduction, each active pixel is associated to its specific audio signature called an audio pixel. These pre-computed audio signatures are retrieved from a database loaded at the start of the application. The final audio transduction of the video frame (called an audio frame), resulting from the summation of all audio pixels of a video frame, is transmitted to the audio mixer for subsequent audio transmission.

On the other side, the audio output stream is running at its own constant frequency depending on the audio sampling frequency and the settings of the internal audio buffer. The mixer is devoted to the synchronization between the processing thread and the audio thread and to the smooth mix of successive audio frames for the subsequent sound production. The audio player is in charge of transmitting the data to the sound card driver.

3. Results

As presented in Figure 4, the median latency of the whole system for the sonification of a light switched off is around 48ms (min = 25.4ms, max = 76.1ms) for 100 sonified pixels. This value increases to 55.7ms (min = 27.8ms, max = 79. ms) for 500 sonified pixels and to 64.6 ms (min = 25.4 ms, max = 76.1 ms) for 1000 sonified pixels. This global latency is the sum of the latencies of the parts of the software. In the following subsections, we describe in details the processing of each part and we present the corresponding latencies.

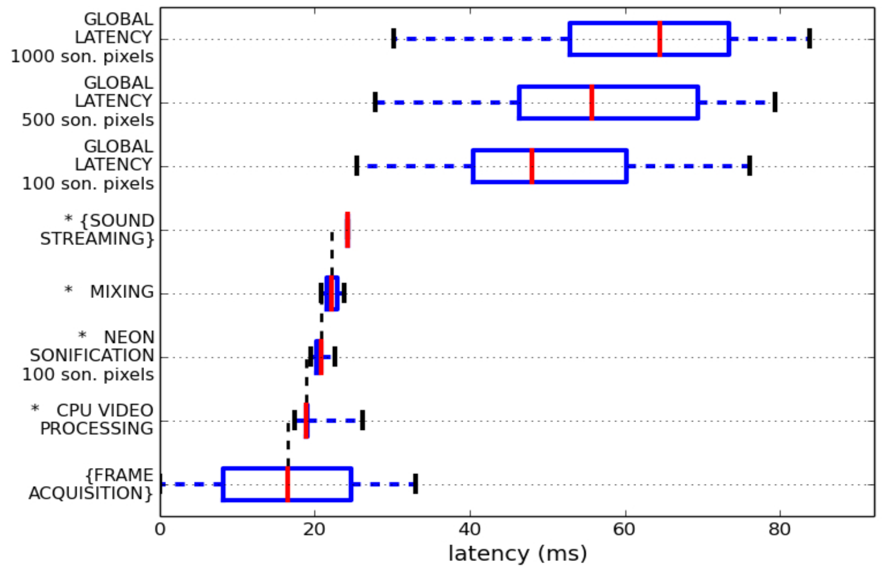


Figure 4. Whiskers plot (minimum, first quartile, median, last quartile, maximum) showing the latencies of the system

Latencies are presented for the different steps of the software: Frame acquisition, Video Processing, Sonification, Mixing, and Audio streaming. At the top are presented the global latencies for 100, 500 and 1000 sonified pixels. Box plots with the symbol * at the beginning are shifted to the right by an amount corresponding to the median latency of the previous step below marked with a vertical dashed line. Box plots with labels surrounded by the symbols { } are computed from theory and not from measurements.

3.1 Java Layer

The Java layer is intended to integrate the software with the Android operating system. At the start of the application, the native (C) part of the application is loaded as a JNI library and super user rights are granted to directly control the sound output. An audio signal is triggered to inform the user of the startup of the software. A speech recognition service (integrated into the Android OS) is activated for 5 seconds immediately after the audio signal has been triggered. Speech recognition is used to allow the selection of special operating modes that will be developed in the future. The events associated with the volume control of the headset buttons start or stop the audio output at runtime and a graphical menu is provided to offer the possibility to set the path of the audio file used as a sound database. This part of the program runs for 5 seconds at the startup of the application and afterwards remains in a sleeping state until the application is closed. It thus does not add a latency to the processing.

3.2 Video Processing

The output of the camera has a YUYV format with 160 × 120 pixels. It is first grayscaled by extracting the Y (luma) channel only. Since the system is dedicated to the sonification of a moving object, we used a frame differencing technique associated with a light blur and a thresholding to extract moving contrasts from the video stream. This frame differencing method computes an absolute difference between the current and the previous frames. The Gaussian blur was applied to a sliding 3 × 3 window and an intensity of $\sigma_x = \sigma_y = 2$. This is meant to attenuate insignificant details such as small white pebbles on the ground. The result is then thresholded, setting to 255 all pixels with a difference level above 100, and setting all other pixels to 0. The pixels set to 255 represent the active pixels that will be sonified afterward.

We compared the time required by the video processing using two libraries: the standard OpenCV library using the CPU and the OpenCL-OpenCV (OpenCV1.1 with full support) library that uses the GPU. Code for the two solutions is presented in Annexe 5.2 and the comparison of the performances is shown in table 1. For a frame resolution of 160 × 120, the time taken by the standard OpenCV library is approximately 2.5 ms per frame whereas it is approximately 7.5 ms with OpenCV-OpenCL. We tested the same code with a 1920 × 1080 and 3840×2160 frame resolution and we consistently found a seed-up factor of over 1.5 in favor of the standard library. It seems that the video-processing required by the system does not allow the strong GPU acceleration normally provided by OpenCV-OpenCL. Thus, using standard OpenCV seems preferable at this time. However, for a high resolution with the standard library, the processing thread reaches its maximum (25%) of the CPU usage (usage of one complete core among four) whereas the CPU usage remains stable with OpenCL, suggesting that the GPU technology might be a better option for heavy computations.

Table 1. Latency and CPU usage required by the video processing

		Image resolution		
		160x120	1920x1080	3840x2160
time (ms)	Standard OpenCV	2.560	51.555	138.388
	OpenCL-OpenCV	7.118	85.406	278.048
CPU usage	Standard OpenCV	8%	20%	25%
	OpenCL-OpenCV	10%	10%	8%

Comparison of the time and the CPU required by the video processing using the standard OpenCV implementation and the OpenCL-OpenCV implementation according to the image resolution.

3.3 Sonification

The sonification consists in scanning all the active pixels of the processed video frame. For each active pixel (i.e. white pixel) characterized by its [ID_column , ID_row] position in the video frame, the corresponding audio pixel is retrieved from the sound database loaded at the startup, as explained in the next sub-section.

3.3.1 Synthesizing and Storage of a Sound Database

Synthesizing online all the sounds corresponding to all active pixels during signal transduction would require far too much computation for our hardware. For this reason, we used pre-computed sounds stored in a file and loaded in the Random Access Memory (RAM) at the startup of the application. In contrast to many other visuo-auditory sensory substitution devices, we completely decouple the sound database from the processing methods in charge of the sonification scheme.

The metadata required to correctly load the data file is stored in an XML format in the author field of the wave file, as presented in appendix A. A special flag indicating the type of file structure is situated at the beginning of the metadata. In the following, we present a sound database corresponding to the type “LAV”.

The audio file is a single f loat32 (4 bytes) stereophonic WAV file containing a succession of 160 × 120 = 19200 audio pixels of stereophonic PCM data. Each audio pixel is composed of a succession of 8 audio chunks, each containing 128 samples, totaling $128 * 8 = 1024$ stereophonic samples (2048 values, 8192 bytes) and corresponding to 23.2 ms for an audio sampling frequency of 44100 Hz (see Figure 5 for an illustration). For a video frame resolution of 160 × 120 = 19200 pixels, the total amount of memory to be loaded is thus equal to

$19200 * 8192 = 157,286,400$ bytes which is small enough to be stored with standard RAM capacities. At the launch of the application, the time required to load the data from the file is around 0.5 s.

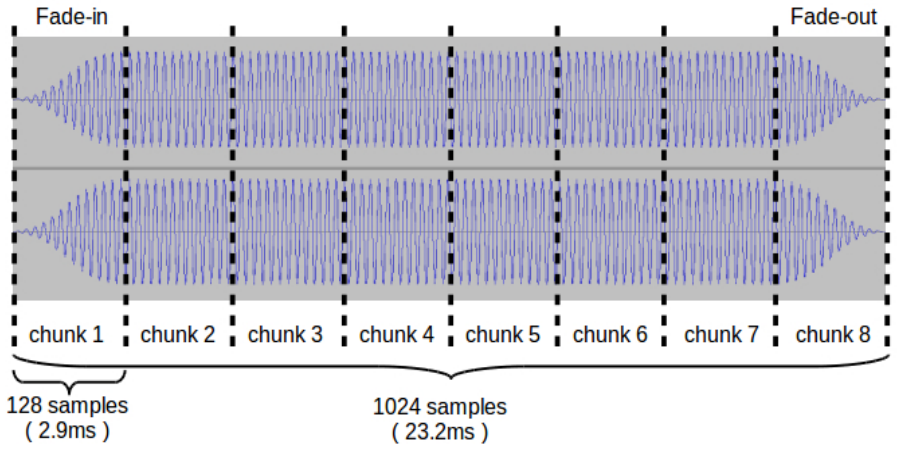


Figure 5. Structure of an audio pixel. An audio pixel is a stereophonic sound lasting 23.2 ms and composed of 8 contiguous data chunks each containing 128 samples. A fade-in and a fade-out modulate, respectively, the first and the last data chunk

$$f(x,y) = 1960\frac{0.53 + z(x,y)}{26.28 - z(x,y)}$$
$$z(x,y) = z_{min} + l(x,y)(z_{max} - z_{min})$$
$$l(x,y) = \frac{2yW + x}{2WH}$$

Compared to our previous work (Ambard et al., 2015), the default sound database has significantly changed. We reused the methods associating a given frequency to each pixel of the image as follows:

where x and y are, respectively, the column ID and the row ID of the pixel, W = 160 and H = 120 are, respectively, the total number of columns and rows in the image, f is the frequency of a sinusoid signal, and $z_{min} \approx 2.5$ and $z_{max} \approx 14.5$ are variables setting the minimal and the maximal used frequencies to 250Hz and 2500Hz (see Ambard et al., 2015 for more details).

Each pixel is then associated with a pure tone defined by the following formula:

$$tone(x,y) = sin(2\pi f(x,y)t + \varphi_{rand})$$

where φ_{rand} is a randomly chosen phase added to limit sinusoid interferences when multiple sounds are added.

Instead of using equations to compute the Inter-aural level difference (ILD) and Inter-aural time difference (ITD) as presented in our previous work, the sounds are now spatialized using the compact Head Related Transfer Function (HRTF) database from the MIT. This eliminates incoherences in sounds when ILD and ITD do not exactly correspond to the same spatial position.

To spatialize the sound for each pixel, all the pixel coordinates are first remapped on a regular spherical grid with azimuth angles α_x ranging from −60° to 60° and elevation angles θ_y ranging from −40° to 40°. For each pixel localization, an interpolated HRTF is computed based on the inverse distance weighting of the four most proximate HRTF database nodes.

The final signal for an audio pixel is generated as follows:

$$s_r(x,y,t) = HRTF_r(\alpha_x, \theta_y, tone(x,y))$$
$$s_l(x,y,t) = HRTF_l(\alpha_x, \theta_y, tone(x,y))$$

where $HRTF_r$ (resp. $HRTF_l$) stands for the HRTF transformation function for the right (resp. left) channel and s_r (resp. s_l) stands for the signal for the right (resp. left) channel of a synthesized audio pixel.

In order to decrease the computation required by the mix of successive audio frames (see section 3.4), the first audio chunk of each synthesized audio pixel is multiplied by a sigmoid fade-in and the last audio chunk is multiply by a sigmoid fade-out, both computed by means of the following formulas:

$$A_{fadein}(t) = haversine(\pi t/T_{max})$$
$$A_{fadeout}(t) = 1 - A_{fadein}(t)$$

with T_{max} = 128/44100 sec.

3.3.2 Generation of an Audio Frame

The summation of all the audio pixels associated with all active pixels extracted from the video frame results in an audio frame which is the audio signature of the whole processed video frame. Thus, an audio frame is composed of the same number of samples as an audio pixel (i.e 128 $*$ 8 = 1024 stereophonic samples) and has the same fade in-fade out amplitude modulation (see Figure 6 for an illustration). After an audio frame has been computed, the samples are converted from float32 into int16 to fit the settings of the audio output (see section 3.5).

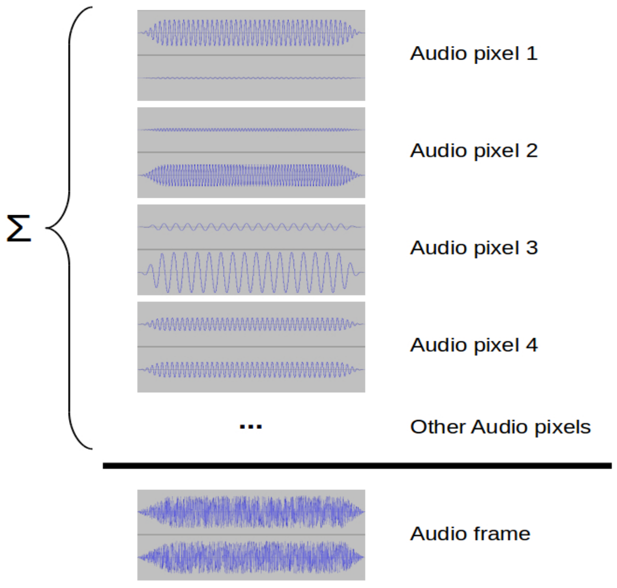


Figure 6. Generation of an audio frame based on a collection of audio pixels

The audio pixels corresponding to the sonified active pixels of the processed video frame are added to each other to generate an audio frame of the same duration that represents the audio transduction of the current video frame

Even using a precomputed sound database, the summation of the audio pixels is one of the heaviest calculations of the software. This can generate a long latency if the calculation is not optimized. We therefore took advantage of the Single Instruction, Multiple Data (SIMD) architecture of the Cortex − A17 CPU. We compared the time taken to calculate the sonification of 100, 500 and 1000 active pixels by means of a standard C loop and a loop based on NEON instructions (see Annexe 5.3 for details). As presented in Table 2, we measured a speed-up factor of 3 in favor of the NEON computations. The associated CPU load is also lower with NEON optimization than with a standard C loop.

Table 2. Latency and CPU usage required by the sonification

		Number of sonified pixels		
		100	500	1000
time (ms)	C	5.866	17.082	22.186
	NEON	1.702	6.005	8.896
CPU usage	C	9%	16%	19%
	NEON	8%	9%	10%

Comparison of the time required by the sonification and the CPU usage of the whole software for 100, 500, and 1000 sonified pixels with a standard C loop and a NEON optimization.

3.4 Mixing of the Audio Signal

After sonification, the computed audio frame is sent to the mixer that carries out the synchronization between the processing and the audio threads. The mixer receives multiple calls from the processing thread to push new audio data, and from the audio thread to pull data in order to send it toward the audio output. For this reason, the mixer memory has to be protected by a mutex to avoid race conditions when data is simultaneously read and written. The mixer aims to generate a continuous audio stream despite an irregular reception of audio frames.

Audio data are pushed to the mixer audio frame by audio frame (1024 samples, 23 ms). On the contrary, audio data are pulled from the mixer audio chunk by audio chunk (128 samples, 2.9 ms). Pulling audio chunks instead of audio frames allows the system to frequently check whether a new audio frame is available, and if so, to rapidly update the audio output. Moreover, it allows the mixer to adapt the successive audio chunks transmitted to the audio streamer in order to smooth the transitions between successive audio frames. To do so, two successive audio frames are always superposed and mixed on a small portion of the signal. This is achieved by summing a faded out audio chunk of the current audio frame with a faded in audio chunk of the following audio frame.

As mentioned in section 3.3, the sonification of a video frame generates an audio frame that lasts a constant duration of ≈ 23 ms. This duration is not long enough to cover the period between two acquired video frames at 30 fps. Thus, audio frames are consumed by the audio streaming more quickly than they are produced by the sonification. In the case in which all the audio chunks of the current audio frame have been transmitted before a new audio frame is available, the current audio frame is repeated and the last chunk of the current audio frame is mixed with its first chunk as indicated by the label ”A” in Figure 7.

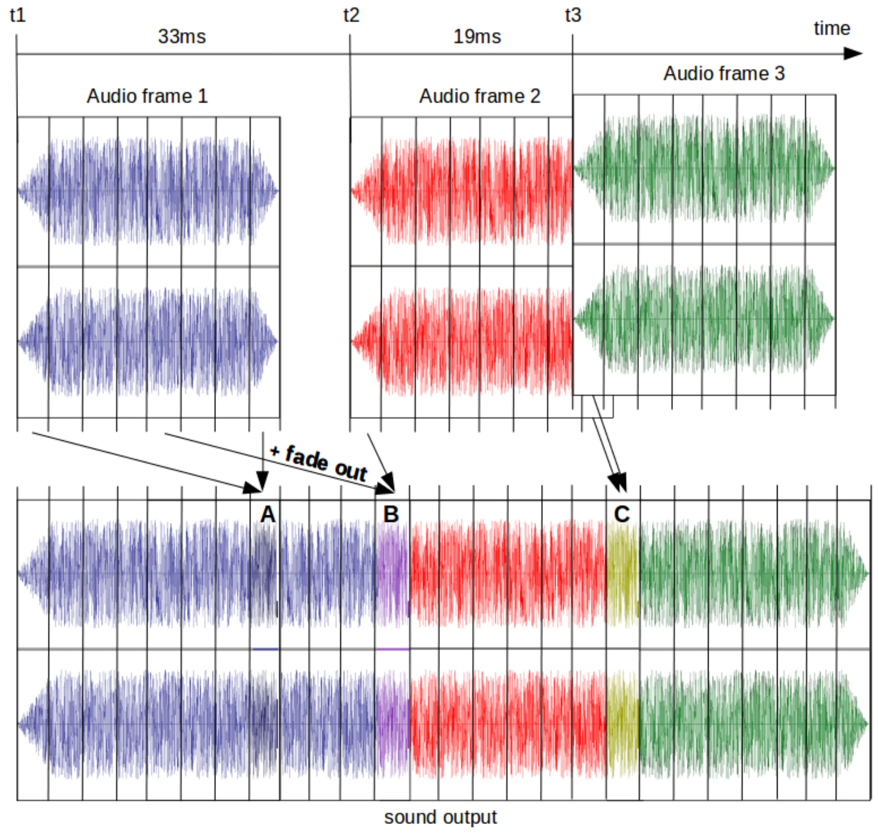


Figure 7. Illustration of the different scenarios for the mixing of successive audio frames

Three audio frames arrive at times t1, t2, and t3. A. The delay (33 ms) between t1 and t2 is more than the duration of an audio frame (22.9 ms). For this reason, the first audio frame is repeated and the first audio chunk is mixed with the last audio chunk to obtain a smooth transition. B. The second audio frame arises while the repetition of the first audio frame is still in progress. The first audio frame is interrupted, the next audio chunk of the current audio frame is faded out and mixed with the first audio chunk of the second audio frame. C. The third audio frame arises right before the end of the second one. In this case, the last audio chunk of the second audio frame is simply mixed with the first audio chunk of the third audio frame.

On the contrary, a new audio frame may be available before all the audio chunks of the repeated current audio frame have been transmitted to the audio streamer as indicated by the label ”B” in Figure 7. In this scenario, and in order to limit the latency between a visual event and its auditory transduction, priority has to be given to the new incoming audio frame and the transmission of the current audio frame has therefore to be interrupted to make way for for new one. In this case, the current audio chunk of the previous audio frame is faded out on-line and mixed with the first audio chunk of the newly arrived audio frame.

Finally, in the simplest scenario, when the arrival times of two successive audio frames are such that the two audio frames have simply to be concatenated, the last chunk of the previous audio frame is mixed with the first chunk of the new one as indicated by the label ”C” in Figure 7.

The latency of the mix between the push of a new audio frame and the pull of its first audio chunk is thus theoretically a uniform distribution with a minimum of 0 ms and a maximum of 2.9 ms (the duration of an audio chunk). We measured this latency and found a mean of 1.412 ms with a minimum of 3 μs and a maximum of 3.063 ms, in accordance with the theory.

3.5 Low-Latency Audio Streaming

The android operating system is known for poor its performance in terms of sound streaming latency. In our previous work, we presented a system with a total latency around 0.23s (Ambard et al., 2015). In order to reduce the latency, we conducted tests with a library based on the Open Sound Library for Embedded Systems (OpenSL ES) that is provided with the Android Native Development Kit (NDK) to improve the sound latency (Patra et al., 2014). Unfortunately, we did not obtain significant improvement. We then decided to root our Android system in order to directly access the sound card with superuser permissions. We used a library called Tinyalsa (Tinyalsa Project, 2011) to perform direct PCM streaming.

The audio output was configured with a 16bit sample format and a sampling frequency equal to 44100 Hz. The buffer of the alsa PCM player was composed of three buffers of 64 samples corresponding to 3×1.45ms = 4.35ms of sound. The audio thread aims to pull and copy, in an infinite loop, successive audio chunks from the mixer toward the audio driver.

4. Discussion

In this work we have presented a software architecture optimized for mobile computing that provides a low latency between an acquired video stream and its sonified transduction. This characteristic is of paramount importance for this type of sensory substitution system whose aim is to transduce online the trajectory of moving objects into sounds. Compared to our previous system presented in (Ambard et al., 2015), we reduced the overall latency from 230ms to 65ms. Running the new software presented in this work on our previous hardware, gives a median latency of 88ms for 1000 sonified pixels which indicates that ~14% of the system speed-up is due to the new hardware and ~86% is due to the new software.

The selection of audio cues used to render the spatial information has been externalized from the inner design of the software. The synthesized sounds are recorded in a file loaded at the startup of the application. This allows for the separation between the technical and the psychoacoustic aspects of the system. Different sonification schemes can thus be shared and tested easily. In order to limit the video processing latency, the frames are acquired at a low resolution and the video processing is very simple but efficient. The sonification process uses NEON optimization to speed up the processing. The successive sounds are asynchronously and continuously mixed to limit the audio artifacts created by non-contiguous signals. The decomposition of sounds into small fractions allows the audio stream to be quickly updated when a new frame is acquired, even with variable video acquisition frequencies. The audio output bypasses the Android layer to directly control the sound card in order to reduce the latency.

Thanks to the association of all these techniques, it is possible to obtain a total latency of approximately 48ms (for 100 sonified pixels), largely due to the camera acquisition (≈ 16.5 ms). The information processing duration (video processing + sonification + thread synchronization) is approximately 7 ms (for 100 sonified pixels) and the audio buffer requires an additional 4 ms of latency. The difference between the observed 48 ms and the theoretical $16.5 + 7 + 4 = 27.5$ ms is probably due to additional hardware latencies (camera and sound card). It has been shown that musicians and non-musicians can notice the latency between a simple action (keystroke) and a resulting sound in the ranges of, respectively, 102 ± 65 ms and 180 ± 104 ms (van Vugt & Tillmann, 2014).

The system latency of 48 ms or even that of 65 ms (for 1000 sonified pixels) thus seems at the limit of the detection capacities of human beings.

Further reduction in the video acquisition latency would involve the use of a high-speed camera, since using a 60 fps camera could theoretically decrease the latency between a real event and its appearance on an acquired video frame from 16.5 ms to 8.3 ms. Reducing the audio latency would probably involve the use of real-time hardware for sound streaming. Using a real-time linux kernel might be an option, but this would require abandonning the Android OS and all its applications that are dedicated to visually impaired people. Moreover, this would probably allow a reduction from 4 ms to ≈ 2 ms and thus not drastically change the overall latency.

The software architecture described in this work is not well suited to sonification methods in which video frames are acquired at a low frequency and in which each of them is sonified during a column-by-column, left-to-right scan, as in the first family of VASSDs presented in the introduction. However, the video processing can be adapted to construct from each acquired frame a sequence of processed images, each containing a specific column of the current frame. Each of these processed images is then sent for sonification. Since the audio streaming is updated at each new incoming image, this would generate a sound-scape similar to that produced by the original sonification methods.

The design of such mobile systems is a tradeoff between miniaturization, information resolution, and latency. Dedicated miniature printed circuit boards would have the advantage of being integrable in eyeglass frames while providing low latency and good sound quality.

Acknowledgments

This study was carried out in part with the support of the Burgundy region (contract 2014-9201AAO047S05164), the University of Burgundy and the CNRS (Centre national de la recherche scientifique). The author would like to thank Philippe Pfister, the electronic engineer at the LEAD laboratory. He assembled the mobile device and the system used to measure the global latency of the sensory substitution system.

References

Abboud, S., Hanassy, S., Levy-Tzedek, S., Maidenbaum, S., & Amedi, A. (2014). Eyemusic: Introducing a "visual" colorful experience for the blind using auditory sensory substitution. *Restor. Neurol. Neurosci., 32*(2), 247–257. http://doi.org/10.3233/RNN-130338

Ambard, M., Benezeth, Y., & Pfister, P. (2015). Mobile video-to-audio transducer and motion detection for sensory substitution. *Front. In ICT, 2*(20). http://doi.org/10.3389/fict.2015.00020.

Arno, P., Capelle, C., Wanet-Defalque, M. C., Catalan-Ahumada, M., & Veraart, C. (1999). Auditory coding of visual patterns for the blind. *Perception, 28*, 1013–1029. http://doi.org/10.1068/p2607

Arno, P., Vanlierde, A., Streel, E., Wanet-Defalque, C., Sanabria-Bohorquez, S., & Veraart, C. (2001). Auditory substitution of vision: Pattern recognition by the blind. *Appl. Cognit. Psychol., 15*, 509–519. http://doi.org/10.1002/acp.720

Auvray, M., Hanneton, S., Lenay, C., & O'Regan, K. (2005). There is something out there: distal attribution in sensory substitution, twenty years later. *J. Integr. Neurosci.*, *4*(4), 505–521. http://doi.org/10.1142/s0219635205001002

Bologna, G., Deville, B., & Pun, T. (2009a). Blind navigation along a sinuous path by means of the see color interface. *Bioinspired Applications in Artificial and Natural Computation*, *5602,* 235–243. http://doi.org/10.1007/978-3-642-02267-8 26

Bologna, G., Deville, B., & Pun, T. (2009b). On the use of the auditory pathway to represent image scenes in real-time. *Neurocomputing, 72*, 839–849. http://doi.org/10.1016/j.neucom.2008.06.020

Bologna, G., Deville, B., & Pun, T. (2010). Sonification of color and depth in a mobility aid for blind people. *In Proceedings of the 16th International Conference on Auditory Display (ICAD2010), Washington, DC, USA.*

Brown, J. D., Simpson, A. J. R., & Proulx., M. J. (2014). Visual objects in the auditory system in sensory substitution: How much information do we need? *Multisensory Research, 27*, 337–357,. http://doi.org/10.1163/22134808-00002462

Capelle, C., Trullemans, C., Arno, P., & Veraart, C. (1998). A real-time experimental prototype for enhancement of vision rehabilitation using auditory substitution. *IEEE Transactions On Biomedical Engineering, 45*(10), 1279–1293. http://doi.org/10.1109/10.720206

Deville, B., Bologna, G., Vinckenbosch, M., & Pun, T. (2009). Seeing colours with an orchestra. *Human*

Machine Interaction, 5440, 235–243. http://doi.org/10.1007/978-3-642-00437-7 10

Durette, B., Louveton, N., Alleysson, D., & Hérault, J. (2008). Visuo-auditory sensory substitution for mobility assistance: testing TheVibe. *Workshop on Computer Vision Applications for the Visually Impaired*, pages Marseille, France.

Hanneton, S., Auvray, M., & Durette, B. (2010). The vibe: a versatile vision-to-audition sensory substitution device. *Applied Bionics and Biomechanics, 7*(4), 269–276. http://doi.org/10.1080/11762322.2010.512734.

Levy-Tzedek, S., Hanassy, S., Abboud, S., Maidenbaum S., & Amedi, A. (2012). Fast, accurate reaching movements with a visual-to-auditory sensory substitution device. *Restor. Neurol. Neurosci., 30*, 313–323. http://doi.org/10.3233/ RNN-2012-110219

Levy-Tzedek, S., Riemer, D., & Amedi, A. (2014) Color improves 'visual' acuity via sound. *Frontiers in neuroscience*, 8. http://doi.org/10.3389/fnins.2014.00358

Maidenbaum, S., Abboud, S., & Amedi, A. (2014a) Sensory substitution: closing the gap between basic research and widespread practical visual rehabilitation. *Neurosci. & Biobehav. Reviews, 41*, 3–15. http://doi.org/10.1016/j.neubiorev.2013.11.007

Maidenbaum, S., Arbel, A., Shapira, S., Buchs, G., & Amedi, A. (2014b) Vision through other senses: practical use of sensory substitution devices as assistive technology for visual rehabilitation. I*n Proceedings of the 22nd Mediterranean Conference of Control and Automation (MED)*, Palermo, Itali. http://doi.org/10.1109/med.2014. 6961368

Patra, S., Velisetty, K., Patel, P., & Kolhe, A. (2014). A study on the opengl es and the opensl es. *Android ndk Paradigm, 10*(4). The Android Open Source Project.

Proulx, M. J., Stoerig, P., Ludowig, E., & Knoll, I. (2008). Seeing 'where' through the ears: Effect of learning-by-doing and long-term sensory deprivation on localization based on image-to-sound substitution. *Plos One, 3*(3). http://doi.org/10.1371/journal.pone.0001840

Stiles, N. R. B., & Shimojo, S. (2015). Auditory sensory substitution is intuitive and automatic with texture stimuli. *Sci. Rep.*, 5. http://doi.org/10.1.1/srep15628

Striem-Amit, E., Guendelman, M., & Amedi, A. (2012). 'Visual' acuity of the congenitally blind using visual-to-auditory sensory substitution. *Plos One, 7*(3). http://doi.org/10.1163/187847612x648206

Tinyalsa-ndk (2011). https://github.com/ykasidit/tinyalsa-ndk

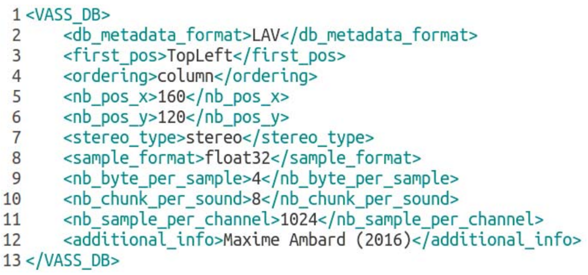
van Vugt, F. T., & Tillmann, B. (2014). Thresholds of auditory-motor coupling measured with a simple task in musicians and non-musicians: Was the sound simultaneous to the key press? *Plos One, 9*(2). http://doi.org/10.1371/ journal.pone.0087176

Ward, J., & Meijer, P. B. L. (2010). Visual experiences in the blind induced by an auditory sensory substitution device. *Consciousness and Cognition, 19*, 492–500. http://doi.org/10.1016/j.concog.2009.10.006

Appendix A

XML for the metadata of a sound database

The code in Figure 8 shows the structure of the metadata inserted into the author field of the wave file. VASSDB is the root tag used to declare that the wav file should be interpreted as a visuo-auditory sound database. The next tag db_metadata_format specifies that the next tags are related to a LibreAudioView sound database. The next specific tags of a LibreAudioView sound database are described in table A1.



```
<VASS_DB>
    <db_metadata_format>LAV</db_metadata_format>
    <first_pos>TopLeft</first_pos>
    <ordering>column</ordering>
    <nb_pos_x>160</nb_pos_x>
    <nb_pos_y>120</nb_pos_y>
    <stereo_type>stereo</stereo_type>
    <sample_format>float32</sample_format>
    <nb_byte_per_sample>4</nb_byte_per_sample>
    <nb_chunk_per_sound>8</nb_chunk_per_sound>
    <nb_sample_per_channel>1024</nb_sample_per_channel>
    <additional_info>Maxime Ambard (2016)</additional_info>
</VASS_DB>
```

Figure 8. Example of metadata used for a sound database.

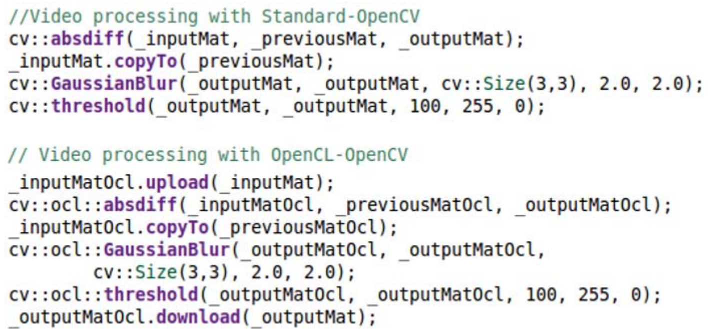
Table A1. Explanation of the meaning of the tags used in the XML metadata integrated in the WAV file used as sound database.

first_pos	position of pixel related to the first sound in the data base
ordering	indicates if sounds are ordered column-by-column or line-by-line.
nb_pos_x	number of pixels on a line.
nb_pos_y	number of pixels on a column.
stereo_type	indicates if the database is stereo or mono.
sample_format	indicates the sample format. Could be int8, int16, int32, float32...
nb_byte_per_sample	indicates the number of bytes per sample.
nb_chunk_per_sound	indicates the number of audio chunk for an audio pixel.
nb_sample_per_channel	indicates the number of sample per channel in an audio pixel.
additional_info	tag used to add more info.

Appendix B

OpenCv video processing

We measured the performances of the video processing using the CPU compared to the GPU (using OpenCL). Figure 9 shows the code related to the two different methods, with the CPU computing at the top and the GPU computing at the bottom. For the CPU, the first line computes the differences in absolute values between the current frame and the previous one, the second line stores the current frame in order to be used with the next incoming frame. The third line blurs the image to mask small details and the last line thresholds the values by setting to 255 all the pixels above 100 and setting to 0 all the others. The OCL method does the same, except for an upload of the incoming image in the GPU memory and a download of the resulting image toward the CPU memory for the further processing.



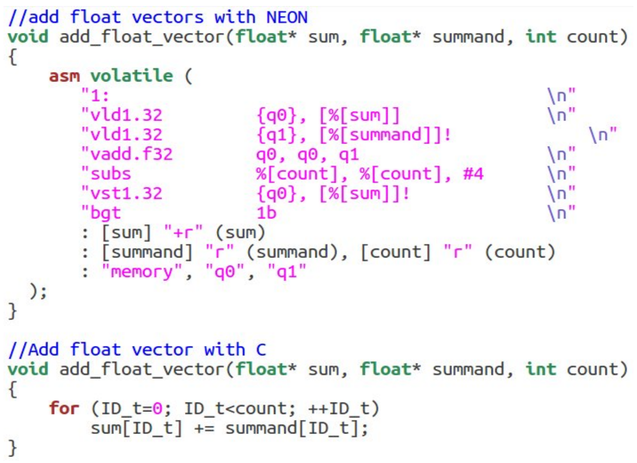
```
//Video processing with Standard-OpenCV
cv::absdiff(_inputMat, _previousMat, _outputMat);
_inputMat.copyTo(_previousMat);
cv::GaussianBlur(_outputMat, _outputMat, cv::Size(3,3), 2.0, 2.0);
cv::threshold(_outputMat, _outputMat, 100, 255, 0);

// Video processing with OpenCL-OpenCV
_inputMatOcl.upload(_inputMat);
cv::ocl::absdiff(_inputMatOcl, _previousMatOcl, _outputMatOcl);
_inputMatOcl.copyTo(_previousMatOcl);
cv::ocl::GaussianBlur(_outputMatOcl, _outputMatOcl,
        cv::Size(3,3), 2.0, 2.0);
cv::ocl::threshold(_outputMatOcl, _outputMatOcl, 100, 255, 0);
_outputMatOcl.download(_outputMat);
```

Figure 9. Two different methods to perform video processing. On top, the method using the standard OpenCV library. On the bottom, the method using the OpenCL-OpenCV library

Appendix C

NEON assembly code for float vector summation

We used neon assembly functions to take full advantage of the ARM NEON capabilities of the RK3288 SoC. We used http://pulsar.webshaker.net/ccc/index.php?lng=us to calculate the ARM cycles required by the computation of the code presented in Figure 10. The computation reached a total of 10 cycles for the addition of 4 floats, which is 2.5 cycles per float.



```
//add float vectors with NEON
void add_float_vector(float* sum, float* summand, int count)
{
    asm volatile (
       "1:                                          \n"
       "vld1.32         {q0}, [%[sum]]              \n"
       "vld1.32         {q1}, [%[summand]]!             \n"
       "vadd.f32        q0, q0, q1                  \n"
       "subs            %[count], %[count], #4      \n"
       "vst1.32         {q0}, [%[sum]]!             \n"
       "bgt             1b                          \n"
       : [sum] "+r" (sum)
       : [summand] "r" (summand), [count] "r" (count)
       : "memory", "q0", "q1"
  );
}

//Add float vector with C
void add_float_vector(float* sum, float* summand, int count)
{
    for (ID_t=0; ID_t<count; ++ID_t)
        sum[ID_t] += summand[ID_t];
}
```

Figure 10. NEON and C methods used to sum two float vectors. Sum and summand should contain arrays of values