CODEN:LUTEDX/(TEIE-3084)/1-47/(2019)

Industrial Electrical Engineering and Automation

Prototyping a composite view for park assist cameras in automotive vehicles

Kevin Eriksson
Johan Hermansson

Division of Industrial Electrical Engineering and Automation
Faculty of Engineering, Lund University

Prototyping a composite view for park assist cameras in automotive vehicles

Kevin Eriksson & Johan Hermansson

Bachelor thesis
Course Code: EIEL05

4th June 2019

Printed in Sweden
Lunds universitet
Lund 2019

Sammanfattning

Syftet med examensarbetet är att undersöka möjligheten att presentera en kompositvy där den ena delen av vyn visar bilens omgiving, sett ur ett fågelperspektiv, och den andra visar bilens backkamera. För att undersöka denna möjlighet byggdes två prototyper baserade på två separata implementationer av parkeringskameran. Den första prototypen baseras på Volvo Car Corporations egen version av Android OS, och är begränsad av en enhet som kontrollerar samt processar den kameraström som visas på bilens skärm. Den andra prototypen är baserad på en dator med operativsystemet Linux och utnyttjar OpenCV för att hantera de fyra kameraströmmarna på grund av avsaknad av tidigare nämnd kontrollenhet.

För att bedöma kvaliteten på respektive prototyp samlades mätdata in av upplevd latens samt hur många bilder prototypen kan strömma per sekund. De båda prototyperna uppvisade snarlika resultat med avseende på upplevd latens, dock var antalet bilder per sekund lägre än väntat i den andra prototypen. På grund av tidsbegränsingar gjordes aldrig någon grundlig undersökning av vad som orsakade problemet.

Nyckelord

Parkeringskamera, Videoströmning, Inbyggda system, Android, Prototyp, C++ , Java.

Abstract

In this thesis two prototypes, based on separate implementations of a Park Assist Camera, was developed. The purpose was to investigate the possibility of presenting a composite view, where the first view shows the surroundings of the car from a bird's-eye view and the other view shows the rear camera. The first prototype was based on Volvo Car Corporations flavour of Android OS, where the performance of the prototype was limited by a component responsible for processing frames from the four cameras into a single, continuous, stream. The second prototype, based on a computer running Linux, used OpenCV to process the camera streams due to the absence of aforementioned component in the first prototype.

To evaluate the performance of each prototype, the amount of frames being rendered each second was measured as well as the perceived latency. Both prototypes presented almost identical results regarding perceived latency, while the first prototype performed better in regard to frames per second. However, the second prototype was unexpectedly suffering from a low frame rate which was not thoroughly investigated due to time constraints.

Keywords

Park assist camera, Video streaming, Embedded system, Android, Prototyping, C++ , Java

Acknowledgments

We want to thank everyone involved making this thesis possible. Especially Robert Fekete, Staffan Rydén and Simon Sjölander for their guidance, support and for sharing their knowledge. We also want to show our gratitude to Volvo Car Corporation for their collaboration and the opportunity to perform this project. Finally, a big thank you to our supervisor Christian Nyberg, for his guidance throughout the project, and our examiner Mats Lilja.

Kevin Eriksson & Johan Hermansson

Contents

1 Introduction

The following section is dedicated to give a brief overview of the projects preconditions.

1.1 Background

Volvo Car Corporation (VCC) is a Swedish car manufacturer who, in recent times, has decided to handle a larger part of their software development internally. This in contrast to outsourcing the development to a third party supplier, which is the traditional approach in the automotive industry. This thesis was requested by a team at VCC's office in Lund. The team focuses on the Park Assist Camera (PAC) in the car and were interested in the possibility of presenting a rear camera view simultaneously with a Bird's-Eye View (BEV). The goal is to present a combined view to the user/driver to eliminate the need to manually switch between the two separate views. The idea behind presenting both views simultaneously is their contrasting usefulness in different situations. The bird's-eye view provides an overview of close proximity surroundings in every direction, whereas the rear view provides more depth in the image though only in the backwards direction. Thus, showing both views together may be desirable for the driver.

VCC's PAC-model consists of four wide angle cameras connected to a camera module, in this thesis denoted the Camera Control Module (CCM) for proprietary reasons. The cameras are placed one on each side of the car and each one produces a camera stream which is received by the CCM.

The camera streams are presented to the user/driver through the car's infotainment display. This display is controlled by a computer, called the Infotainment Head Unit (IHU), running the Android operating system and is hence a so called "Android-based In-Vehicle Infotainment".

This system includes a Hardware Abstraction Layer (HAL) which includes an Exterior View System (EVS) stack. This stack provides the system with an interface for controlling the camera streams and what is shown on the infotainment display.

In addition, VCC have begun development on the next generation hardware platform. This platform benefits from being able to replace hardware components, like the CCM, with software-based solutions achieving the same result. Thus, expensive hardware can be omitted with the implication of an overall simpler hardware structure. Furthermore, the long lead times associated with outsourcing development to third parties can be avoided.

1.2 Purpose

The purpose of this thesis is to investigate the possibility of presenting a composite view for the driver, meaning both the bird's-eye view and rear view being rendered on the car's infotainment display simultaneously. This will be achieved by developing two prototypes, based on current hardware and next generation hardware respectively.

1.3 Goal

The goal for this thesis is to create two prototypes showing a composite view of the PAC and comparing their performance in regards to frames per second and perceived latency. Also their respective shortfalls will be identified. The first prototype will be designed for VCC's current hardware and extend already existing software. The second prototype will be developed on VCC's next generation hardware with fewer constraints in respect to hardware components like the CCM. As a result of omitting these hardware components, more processing must be done by the IHU, for example stitching and post processing of the images generated by the four cameras, resulting in a bird's-eye view.

1.4 Problem

To achieve the goal of this thesis, the following questions are meant to be answered:

1. How is the first prototype limited by the CCM?
2. What deficiencies are present in the prototype based on current generation hardware?
3. How does the performance compare between current generation hardware and next generation hardware?

1.5 Justification

The justification of this thesis is our interest in acquiring a deeper knowledge in the following areas:

- How complex systems in the automotive industry are built.
- How to develop graphic applications with OpenGL/OpenCV.
- Development in multiple abstraction layers in Android.
- Cross language implementation (Java, JNI, C++).
- Development on/for embedded systems.

VCC's justification for requesting the thesis is their interest in researching the possibility to render multiple camera streams simultaneously. This is a feature requested by customers and is available in cars from multiple rivalling car manufacturers. Furthermore, to comply with the requirements of US legislation, car manufacturers are bound to display the rear view camera when a backing event have been initiated by the driver[1, p. 69]. Thus the initial view presented to the driver can not solely be the BEV. One solution to this problem is to present a composite view where one of the fragments consist of the rear view.

1.6 Delimitations

As a result of conducting this thesis at VCC, proprietary information must either be omitted or modified.

Regarding the first prototype, the camera stream is treated by different systems depending on if the OS has booted or not. The prototype will be limited to the system running after the OS has booted. Furthermore, the prototype must be based on VCC's existing code base and support already existing features provided by VCC. It shall also use the already existing hardware structure, including the CCM, for acquiring camera streams. Thus the prototype will also be limited by the nature of the CCM.

The second prototype is of secondary priority and is therefore limited by the time invested into the first prototype.

2 Technical background

This section is devoted to clarifying the different hardware and software structures used in this thesis. Furthermore, the interaction between the different hardware modules related to the PAC will be described as well as their individual behavior. Also, since the first prototype in this thesis is developed in both native C++ and Java, the connection between the two is explained.

2.1 Hardware components of the PAC

The hardware components involved in the PAC can be simplified into the components in *fig. 1*. The IHU is an embedded system, running Android, responsible for coordinating other components in the car as well as managing presentation on the infotainment screen. Regarding the PAC, the role of the IHU is to acquire a raw video stream from the CCM and present it on the infotainment screen while performing required modifications and additions to the stream. Hence, the prototype developed for this thesis will solely be developed on this component in the stack. The CCM will be further described in section 2.4.

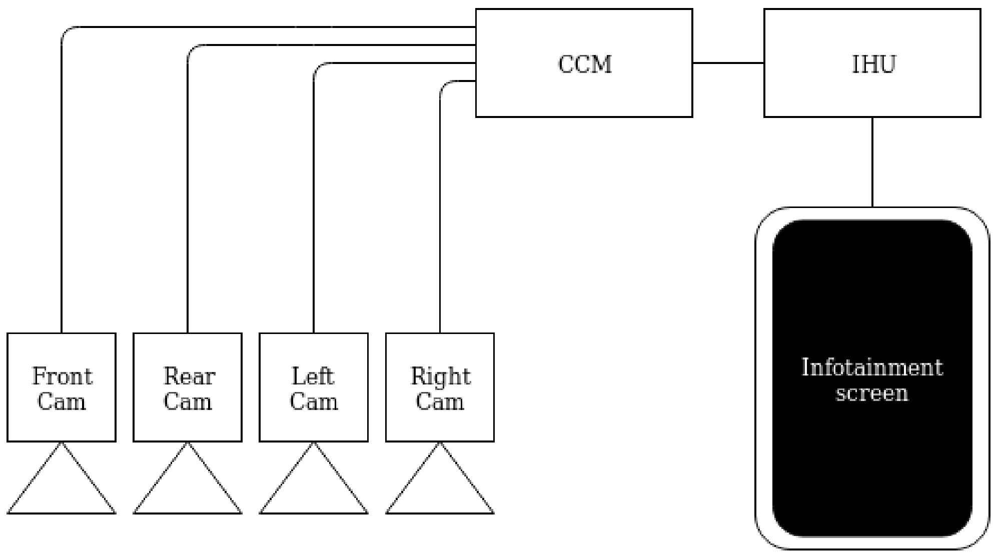


Fig. 1 – Concept of PAC hardware structure

2.2 Software components of the PAC stack

VCC's specific components of the PAC stack that are developed and maintained by VCC is proprietary. However, parts of the PAC structure follow the EVS structure in *fig. 2*. At the top layer a PAC service corresponding to the car service in *fig. 2* is running after the Android OS start-up has finished. The service is responsible for inflating the PAC layout on the infotainment screen whenever it detects that the reverse gear signal has been actuated by the gearbox. The inflated layout contains the output camera stream from the CCM, further explained in *section 2.4*. Rendering of the camera stream is handled in the native layer (EVS Application in *fig. 2*) of the PAC stack communicating with the hardware through the hardware abstraction layer (HAL).

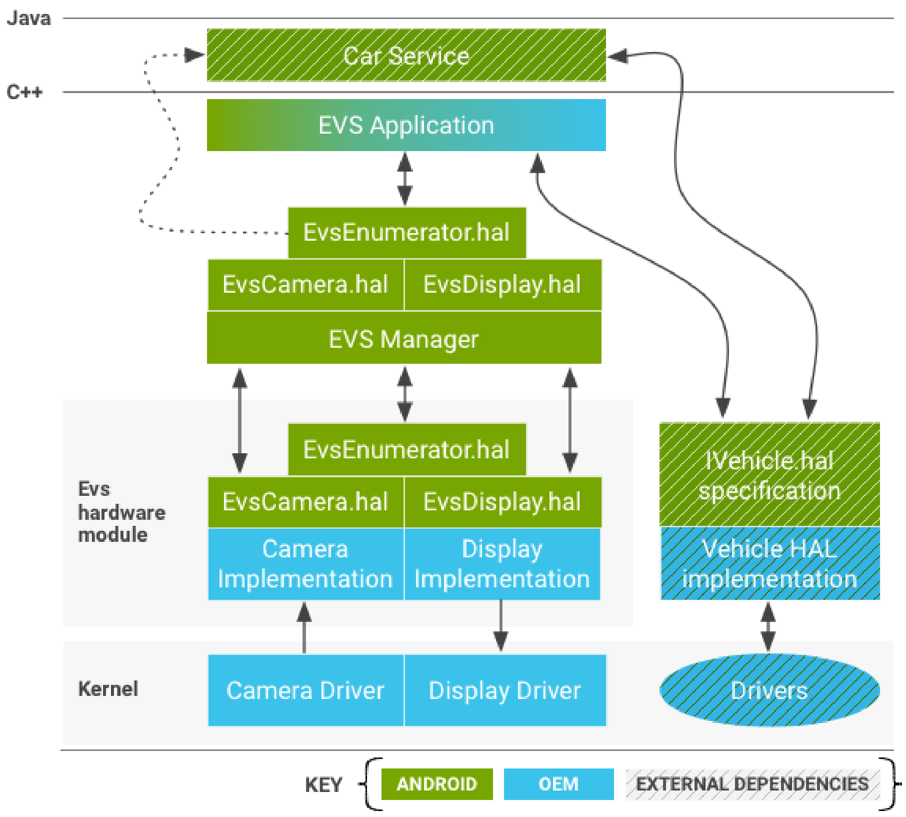


Fig. 2 – Overview of the EVS software system components in the Android Vehicle Camera HAL. Source: [2]

2.3 Communication between native code and Java

Android is an open source operating system available for third parties to develop and customize to their own variant [3]. The source and it's documentation is available in the Android Open Source Project (AOSP) repository. The infrastructure of the AOSP stack provides abstractions for communicating with the manufacturer's hardware e.g. the EVS (see *fig. 2*). Third parties are responsible for supporting hardware and all functionality involved. Native implementation is done in the C++ language and Android's top layer is developed in Java. To be able to communicate between the layers, AOSP uses the Java Native Interface (JNI) which is a framework with the main purpose of allowing Java code running in a Java VM to interoperate with code developed natively in C++ [4].

2.4 Camera Control Module (CCM)

The CCM is a component with the purpose of receiving input data from all four park assist cameras as well as specifying a certain camera stream, from the other PAC components in the car. This image data is processed within the CCM and output on a single video stream. The output video stream can only show one of the cameras at a time or all four cameras simultaneously through a BEV, further explained in *section 2.5*. That is, it is not possible to show both the BEV and the rear view at the same time. The resolution of the output stream is a constant $768 \cdot 1024$ pixels even though the resolution of the camera frame is smaller. This means that parts of the video stream consists solely of black pixels. The top left corner of the video stream is always the top left corner of the camera frame regardless of the size of the frame as illustrated by *figs. 3 and 4*. The image can later be centered by offsetting the video stream along the x- and y-axis.

Communication with the CCM is done through a signal framework which is managed by the native PAC components. To send a signal from the Java layer to the CCM, a function must be constructed in the JNI. This function can then use the signal framework to send a request for a specific view to the CCM. The CCM will process the request and send the specified camera stream back to the IHU. The amount of time required by the CCM to process the request is not specified. After the correct video stream has been sent to the IHU, an acknowledgement signal is sent which will indicate the current state of the video stream.

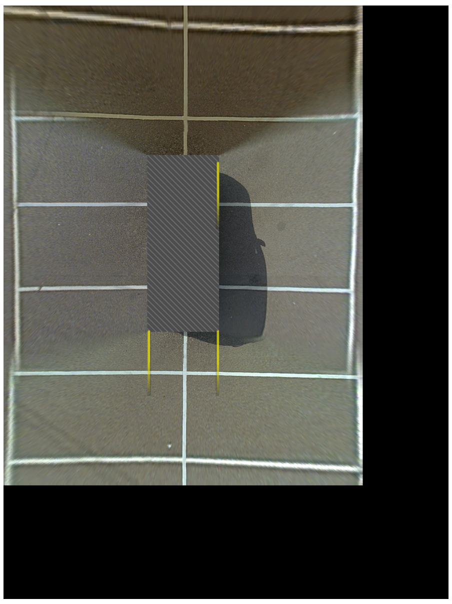


Fig. 3 – Original BEV stream

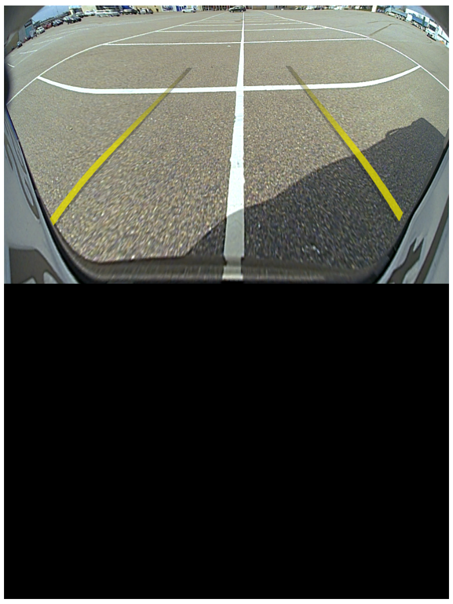


Fig. 4 – Original rear view stream

2.5 Bird's-eye view (BEV)

BEV is a term describing viewing something from above. The term is commonly used in the car industry, describing a view showing the cars surroundings from above. An example of a BEV is presented in *fig. 3.* This view is achieved using a technique where one camera on each side of the car is combined into a single view using algorithms modifying each camera stream. The cameras on the car are wide angle cameras with a field of view (FoV) of almost 180°. Since these cameras are pointing horizontally on the car, some modifications of the camera stream are required to achieve the desired vertical view required for the BEV. This is only possible with a camera with a large FoV since is is not possible to change the physical angle of the camera.

Cameras with a large FoV causes inaccuracies in the picture called curvilinear perspective [5], or fish-eye effect, where straight lines at the edges of the frame appear bent. The countermeasure to this problem is called de-warping and is one of the modifications that has to be made to the PAC to achieve an accurate vertical view of the camera and, thereby, an accurate BEV.

2.6 OpenCV

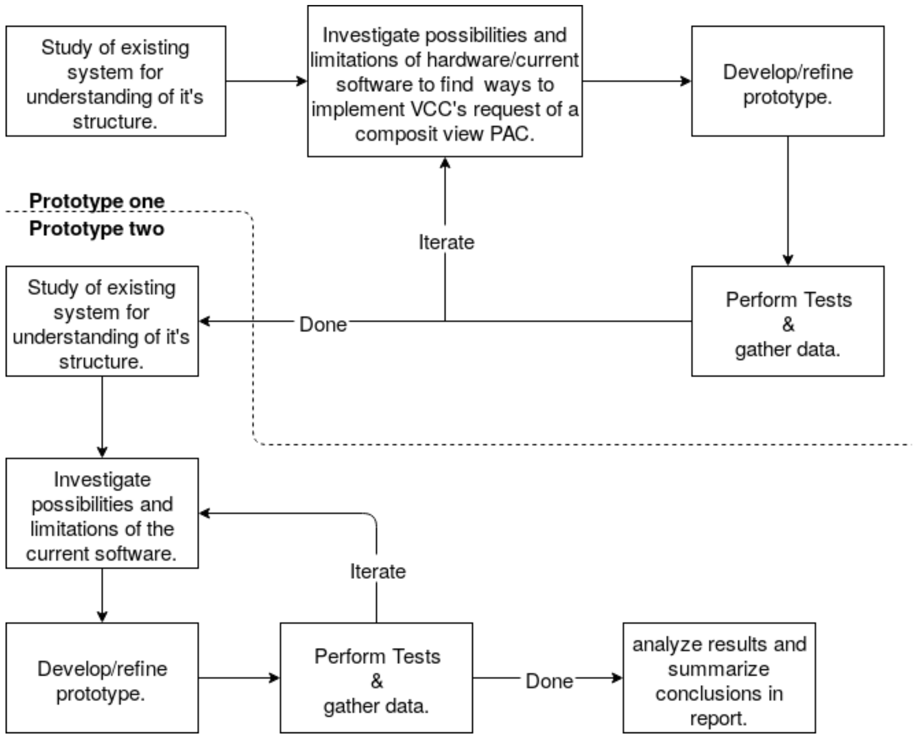
OpenCV is an open source library intended for computer vision and machine learning [6]. OpenCV also provide algorithms for stitching and undistorting images, as well as other functions for manipulating images, useful in development of the second prototype in this thesis.

2.7 OpenGL

OpenGL is a graphics API, maintained by Khronos Group, for developing 2D and 3D applications. It provides functionality for rendering and texture mapping, among many other functions and is often used to communicate with a GPU [7]. OpenGL is primarily used in the first prototype in this thesis.

3 Methodology

This section describes how the thesis work was conducted. Since the thesis is based on the development of prototypes, the development process along with how data was gathered for performance analysis is described in *fig. 5*.



Study of existing system for understanding of it's structure.
Investigate possibilities and limitations of hardware/current software to find ways to implement VCC's request of a composit view PAC.
Develop/refine prototype.
Perform Tests & gather data.
Iterate
Done
Prototype one
Prototype two
Study of existing system for understanding of it's structure.
Investigate possibilities and limitations of the current software.
Develop/refine prototype.
Perform Tests & gather data.
Iterate
Done
analyze results and summarize conclusions in report.

Fig. 5 – Illustration of how the thesis work was conducted.

3.1 Study of current documents

To identify a suitable breakdown structure of the problem, the first step was to acquire knowledge from similar code structures in VCC's code base and study internal documentation of the CCM and the IHU embedded system. One vital part is the understanding of how signals are passed, via the internal data signaling framework (DSF), and understanding how the camera streams are propagating through the system.

3.2 Developing the first prototype

The process of developing the first prototype was based on an iterative work model, where the main goal was divided into several smaller and more comprehensible tasks extending the functionality of the prototype step by step. Another benefit of this model was that the prototype could be kept in a working state during the entire development process. Working this way proved useful since it allowed for continuous testing of the prototype at any point in time of the development phase. Furthermore, continuous testing of the prototype allowed for easier detection of bugs and introduction of erroneous code. However, this made it more difficult to work on different parts of the prototype simultaneously. To circumvent this obstacle, the process was influenced by one main feature from Extreme Programming [8]; Pair programming. By using pair programming in the development process it was easier to detect mistakes in the code early which turned out to have a bigger impact on the development process than expected. Since the the prototype is a full stack implementation, every new version of the source code had to be compiled and built followed by flashing/writing the new binaries to the IHU unit. Otherwise it would be sufficient to only replace the Java application. The whole process took an average time of five minutes presuming no errors were detected in the code and ccache was used. Ccache is a tool for the GCC compiler [9] making caching of the compilation possible, thus decreasing the amount of code that needs to be rebuilt. In the end of the development phase of prototype one it was realized that the Android environment setup script provides a tool chain including different build tools. One such tool provided is mm and mma [10] which allows for building sub modules of the source tree. These tools would greatly improve development speed because building the whole source tree could have been avoided for all builds except the first.

3.2.1 Creating two render surfaces

The first milestone of the prototype was to be able to render two instances of the same video stream in two separate OpenGL-surfaces on the screen. The development started with creating a custom view with the maximum allowed width and height, which is decided by the parent layout. The layout was a LinearLayout [11] which separates child elements vertically, resulting in the child views being stacked on top of each other. To decide how much vertical space to assign to each view, a weight sum was declared in the parent view. This allows the child views to allocate a certain percentage of the screen dynamically by declaring their individual weight in the parent view. This may be found useful later since the two different camera streams have different aspect ratios and therefore may require different amounts of screen real estate. Inside the parent view, two child views were placed both with an equal weight leaving them with half the vertical space each. Both child views were then connected to the same render manager resulting in this render manager giving both views the same data to render. This solution worked as expected apart from the bottom part of the video stream not being visible when rendering the BEV, as a result of the real estate being divided equally. This, however, was decided to be solved later when the two render surfaces was showing different video streams.

3.2.2 Switching between camera streams

After being able to show the same stream in both views simultaneously, the next step was to periodically request the different views. However, still showing the same view in both rendering surfaces. The initial idea was to create a callback-based structure in a designated thread. This thread was supposed to request a view from the CCM, wait for the CCM to process the request, and send its acknowledgement signal as a callback. When receiving the callback signal the thread was supposed to request the other camera stream and, in this manner, keep requesting alternate views when receiving the callback from the previous request. This solution was later abandoned due to, what was believed to be, infrastructure problems in the DSF causing acknowledgement signals to never reach the calling thread. This problem was later found to only affect the IHU when connected to the test equipment, used to simulate signals of a real car in a synthetic manner. When connected to a real car the signals were received in accordance with the documentation of DSF. However, even when the signals were propagating correctly, the time required by the CCM to process the request was believed, by colleagues, to be around 200 ms. As a result, the next request would not be sent any less than 200 ms after the previous request, meaning only a maximum of five requests could be sent every second. Not being able to switch views more frequently than five times per second would create a bottleneck in the system, thus limiting the performance of the prototype. Instead a solution based on a controller thread in the PAC service was implemented. The solution built on the same philosophy as the previous solution but using a periodic request instead of waiting for a callback from the CCM. This design was realized with a separate thread controlled by the PAC Service. The sole purpose of this thread was to request the two camera streams alternately, while sleeping a specified amount of time between each request, indefinitely until it is interrupted by the PAC Service. By running this functionality in a separate thread, halting the main thread could be avoided while assuring time consistency in the CCM requests. This consistency was preserved by the thread only being responsible for a single task and providing no direct communication with other threads.

3.2.3 Selective rendering depending on camera stream

After achieving periodic switching between camera streams, the logical step to proceed was to implement functionality that allows each rendering surface to only render one of the incoming camera streams. The first obstacle encountered was how to perform the selective rendering accurate enough to only render correct frames. The initial solution to this problem was to listen for the signal confirming a change of camera stream from the CCM. As mentioned in *section 3.2.2*, there was an issue with the data signaling preventing this solution. This required rethinking the structure of the prototype and to come up with another solution for the selective rendering.

The first idea that came to mind was to construct a solution that avoids switching which surface is allowed to render for a given time span. As previously mentioned this time span was estimated to be ≈ 200 ms. Therefore a first version of the prototype was constructed, waiting 200 ms after the request was sent before switching the rendering surface that was allowed to render. This however was performing poorly, with no consistency with respect to which frame it was rendering. This was the first indication of fluctuations in the CCM delay.

The prototype was updated by introducing a span of 50 ms in which neither of the rendering surfaces were allowed to render. This improved the result but the rendering surfaces still occasionally rendered the wrong frame. At this point it was clear that the delay in the CCM had to be measured. These measurements are presented in *fig. 10*, page 43.

The results vary 149 ms regarding the time it takes for the CCM to provide the new video stream. Since the CCM can change the video stream at any point in the given time span, the only solution was to allow neither surface to render during this period. This proved problematic since increasing the time span, allowing neither surface to render, would lead to several dropped frames, thereby decreasing the quality of the prototype. This led to the realization that the solution would perform too poorly, if it was even possible to make it perform correctly. Thus, the solution was discarded.

Instead, the problem was approached from a different angle, where each image arriving in the graphic buffer for image processing is analyzed. By analyzing the incoming frame it is possible to identify it based on the fact that the different camera views sent by the CCM have different aspect ratios, where black pixels fill the remaining gap on the screen. By analyzing an area known to be either black or colour depending on the view, it's possible to identify what view the CCM is currently streaming. The algorithm for identifying frames is presented in *appendix B* and the analyzed pixels are illustrated by a red line in *figs. 6 and 7.* Based on this information the correct rendering surface is allowed to render. As a result, the other rendering surface is blocked and the surface is frozen at the last frame rendered before the video stream changed.

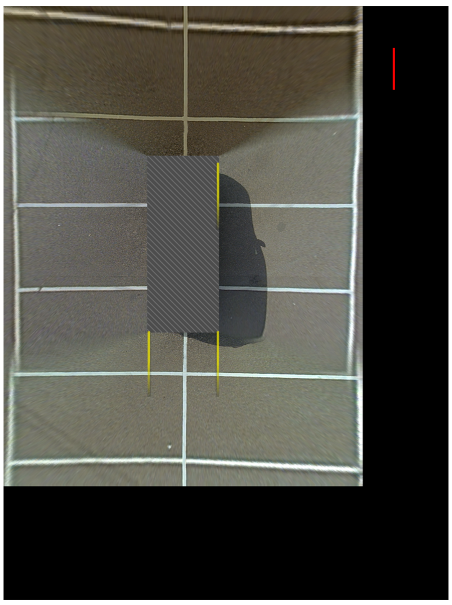


Fig. 6 – Original BEV with a red line illustrating the pixels being analyzed

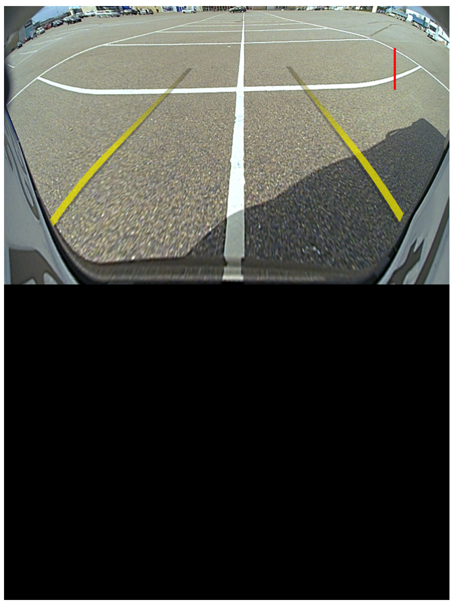


Fig. 7 – Original rear view with a red line illustrating the pixels being analyzed

A collection of measurements was acquired and showed significant deficiencies with respect to delivered frame rate. The BEV and rear view individually measured 5 and 13 fps respectively, together reaching a total of 18 fps. This result equals a loss of approximately 40% of the frames delivered by the CCM, which delivers a continuous stream at 30 fps. The initial thought was that the CPU was choked, and therefore throttling the process. However, this idea was rejected after measurements of the CPU load was gathered. The gathered measurements did not explain why the problem occurred and only showed a minor increase of the the CPU load when switching camera stream every 10 ms compared to when utilizing a fixed camera stream.

After analyzing the manager responsible for rendering both surfaces, located in the Java layer of the application, it became clear that both managers call the method glBind-VideoTextureCompositeView() shown in *appendix A*. Every call to the method resulted in retrieval of the latest frame in a series of buffered frames. Thereafter the frame is analyzed and discarded (deallocated), independently of the result of the pixel analysis. This turned out to be the source of the issue with dropped frames, since it prevents other callers from analyzing the same frame, thus leading to the frame never being rendered.

With this theory in mind a new possible solution was created. It was built on the notion that it might be acceptable to occasionally loose a frame from the spectators point of view. The code was again modified to keep track of a state, updated by the outcome of the pixel analysis in the glBindVideotextureCompositeView method *appendix A*. If the outcome differed from the expected outcome, the frame was discarded and the state indicating a hand over to render the next view was set.

The result showed a minor increase from 18 to 20 fps but with the unexpected outcome of 5 and 15 fps respectively for the BEV and the rear view. It is clear that the rear view shows the desired result of 15 fps but for some reason the result for the BEV deviates. As a result, the prototype was further investigated and the cause of the problem was linked to the previous issue causing dropped frames.

The solution was to avoid discarding a frame unless the caller had rendered it. This allowed for other callers to analyze that same frame. I.e. a caller is only allowed to discard a frame it has successfully rendered, since no other caller should render the same frame again. Note that a vital part for this solution to work is that every frame must be rendered by exactly one view. Otherwise the frame will never be discarded, and thereby block all future frames from being rendered.

After the issue was resolved, the result (see *section 4.1*) improved to 13 and 17 fps respectively for the BEV and rear view, thus fully utilizing the CCM's output capacity of 30 fps.

3.3 Testing and Measurements

In this section the different measurements performed on the prototypes are explained, covering both how fps and perceived latency were calculated.

3.3.1 Frames per second

The number of frames per second was measured by creating a class in the native layer of the application, containing a counter intended to be increased every time a frame was drawn. By using two instances of this debug class, one for each rendering surface, it was possible to count the individual number of rendered frames. The debug tool also kept track of the time stamp of the first and last rendered frame respectively. By using the following equation it was possible to calculate the average number of rendered frames per second for a single rendering surface:

$$Average\, fps \;=\; \frac{n}{t_n - t_0}$$

$n \;=\; number\, of\, rendered\, frames$
$t_n \;=\; time\, of\, last\, frame$
$t_0 \;=\; time\, of\, first\, frame$

3.3.2 Latency of camera pipeline

Every camera pipeline suffers from latency due to the steps involved in producing the final image, transformed from data captured by the CMOS sensor. To measure the latency introduced by the prototype, the following equipment was used:

- External camera capable of recording in 120 fps.
- LED light.
- Stopwatch with millisecond accuracy run on a computer screen with 120 Hz refresh rate.

The equipment was set up in the following way:

1. External camera recorded:
    - Infotainment display.
    - LED light.
    - Stopwatch.
2. PAC camera recorded:
    - LED light.

To measure the perceived latency of the prototype a test setup was constructed according to *fig. 8*. The purpose of the test was to record the elapsed time between the real world event taking place, and the event being rendered on the infotainment screen. For the purpose of the test a LED light was used as a distinct way of indicating the occurrence of a real world event which is also easy to distinguish in the rendered frame. To be able to extract a time stamp of the actual event and the rendered event respectively, a stopwatch had to be visible in the recording at all times. The perceived latency was calculated as the difference between the time stamps acquired by analyzing the recorded video frame by frame.

The accuracy of the perceived latency was found to be approximately 8 ms (1000 ms/120 $\approx$ 8 ms) due to the time resolution of the stopwatch and the recording device. It is also worth noticing that recording in a higher frame rate than 120 fps would not improve the result as the computer screen presenting the stopwatch had a maximum refresh rate of 120 Hz.

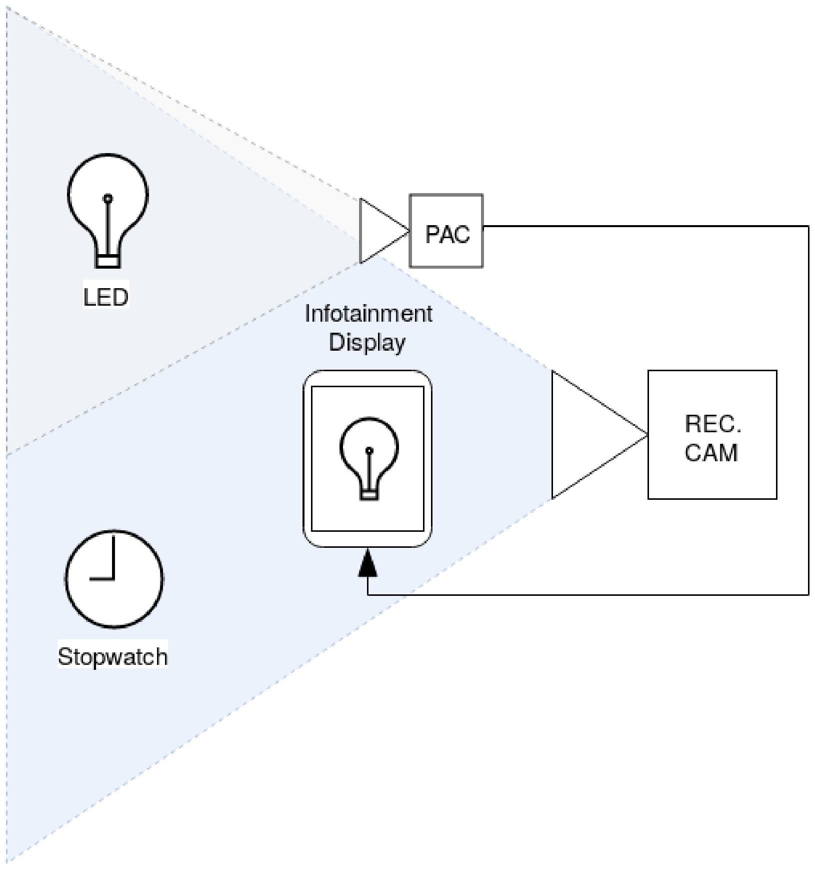


Fig. 8 – Illustration of the test setup used to measure perceived latency in the PAC

3.4 Developing the second prototype

In this section, development of the second prototype will be explained.

3.4.1 Examining the prerequisites

For development of the second prototype a computer running Linux, with hardware connections for the four cameras, was provided. This computer was equipped with an existing program capable of rendering the camera streams. The image processing and rendering was executed solely with OpenCV, a framework explained in *section 2.6*. The entire program was written in C++ and was created in a fashion where an initial setup was executed when running the program. Thereafter an event loop would run until interrupted by the user. This loop primarily fetched, altered, and rendered frames from all four cameras while also handling input from the keyboard to toggle certain options on or off. Aforementioned alterations include the process of creating a BEV from the four camera streams, a process handled by the CCM in the first prototype. The view rendered to the screen contained a BEV, as well as the raw, distorted, stream from each camera. The goal was to modify this view in line with the layout on the first prototype to only show the BEV and the rear view, however undistorted. A problem encountered with this prototype was the limited frame rate of only 8 fps. This frame rate is surprisingly low considering the absence of a CCM, since the CCM was the bottleneck in the first prototype. Without this limiting component it should be possible to render much higher frame rates.

3.4.2 Testing and measurements

To measure the average fps on the second prototype, the same measurement class mentioned in *section 3.3.1* could be used since both programs were written in C++. In line with the measurement of fps in the first prototype, a frame was added to the the measurement whenever a frame was rendered onto the screen.

Regarding measurements of the perceived latency, the corresponding setup used for measurements of the first prototype (see *fig. 8*), could also be applied to the second prototype. By measuring performance factors in the same manner on both prototypes, the trustworthiness of the measurements increases since it decreases the margin of error.

3.5 Criticism of sources

In this thesis the following two forums have been used during the development to support choices: [12, 13]. However, it is important to keep in mind that discussions and statements in forums can not be considered a reliable source of information since the credibility of the author cannot be determined. Thus, statements in this thesis based on information from forums has either been confirmed by testing, or has only been part of a speculation.

[2–4, 6, 7, 9–11, 14, 15] are documentation of libraries, API's, development environments and tools used for the purpose of developing the prototypes and can be considered reliable since they are either written by the maintainers/developers or by a well known organization that is considered trustworthy and have nothing to gain from providing incorrect information regarding technical documentation. This also applies to sources [5, 8] which are mainly used for explanatory purpose of terms.

[1] a document provided by U.S. National Highway Traffic Safety Administration (NHTSA). Therefore it can be considered a reliable source of information concerning Park Assist Camera regulations.

4 Result

In this section the results of all measurements are presented as well as an image of the final prototype. The results are categorized in line with the two prototypes to favour comparison between the two. All graphs are collected in *appendix D*.

4.1 First prototype

A view of the prototype is shown in fig. 9.

Performance measurements of the prototype are as follows:

- Average fps only rendering one view: 30 fps.
- Average fps in BEV: 13 fps.
- Average fps in rear view: 17 fps.
- Perceived latency in BEV when rendering both views (switching every 10 ms):
  - Average: 314 ms.
  - Min: 267 ms.
  - Max: 366 ms.
  - Graph: fig. 11.
- Perceived latency in BEV when rendering both views (switching every 150 ms):
  - Average: 323 ms.
  - Min: 267 ms.
  - Max: 434 ms.
  - Graph: fig. 12.
- Perceived latency when only rendering BEV:
  - Average: 203 ms.
  - Min: 184 ms.
  - Max: 216 ms.
  - Graph: fig. 13.
- Perceived latency in rear view when rendering both views (switching every 10 ms):
  - Average: 280 ms.
  - Min: 234 ms.
  - Max: 324 ms.
  - Graph: fig. 14.

- Perceived latency in rear view when rendering both views (switching every 150 ms):
  - Average: 309 ms.
  - Min: 234 ms.
  - Max: 416 ms.
  - Graph: fig. 15.
- Perceived latency when only rendering rear view:
  - Average: 173 ms.
  - Min: 150 ms.
  - Max: 185 ms.
  - Graph: fig. 16.
- Latency in the CCM when switching to BEV:
  - Average: 186 ms.
  - Min: 130 - 139 ms.
  - Max: 260 - 269 ms.
  - Graph: fig. 10.
- Average latency in the CCM when switching to rear view:
  - Average: 153 ms.
  - Min: 60 - 69 ms.
  - Max: 200 - 209 ms.
  - Graph: fig. 10.

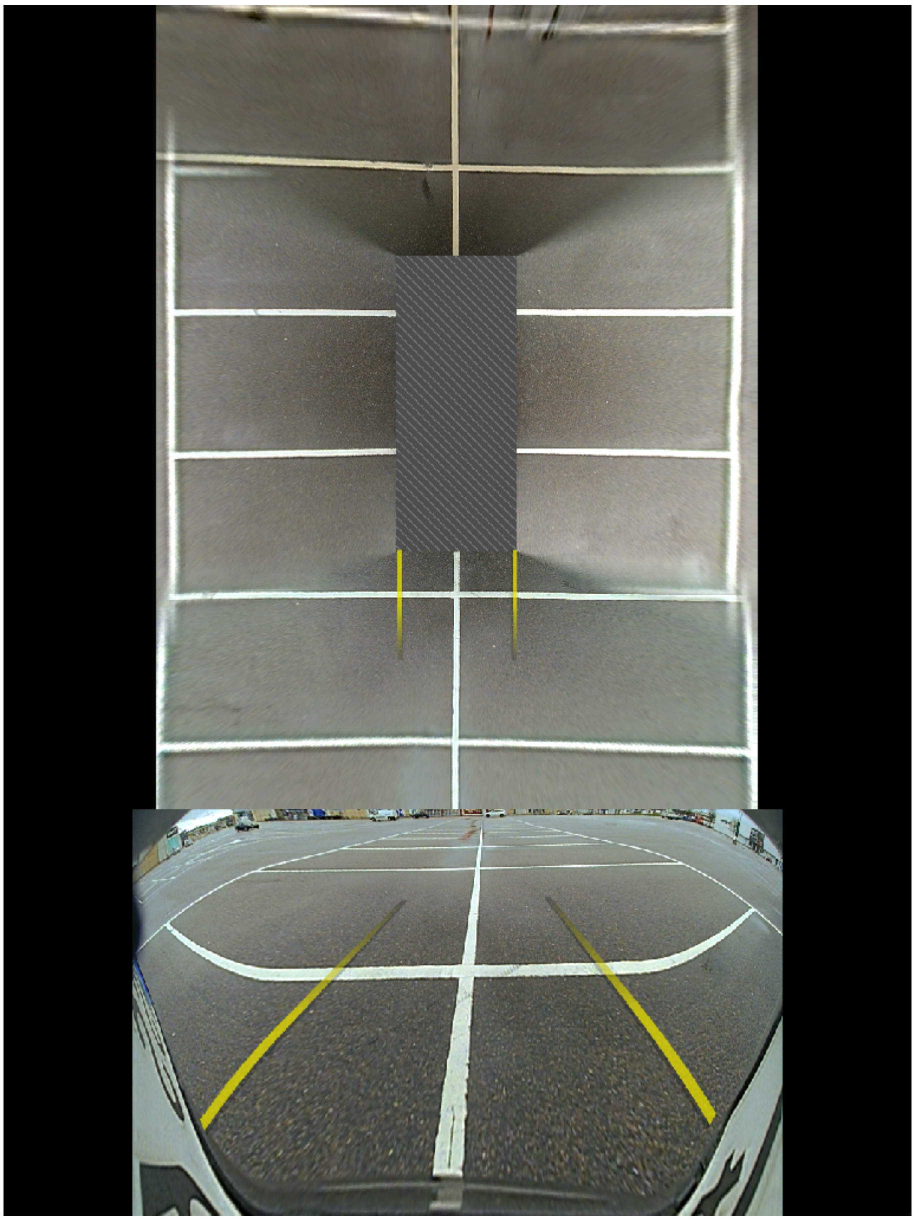


Fig. 9 – Final version of the first prototype

4.2 Second prototype

Performance measurements of the prototype are as follows:

- Average fps in BEV: 8 fps.
- Average fps in rear view: 8 fps.
- Perceived latency in BEV:
  - Average: 267 ms.
  - Min: 231 ms.
  - Max: 308 ms.
  - Graph: fig. 17.
- Perceived latency in rear view:
  - Average: 267 ms.
  - Min: 231 ms.
  - Max: 308 ms.
  - Graph: fig. 18.

5 Analysis

This chapter is dedicated to analysis and discussion with respect to the extracted measurements in *section 4*, page 24. This with the purpose of answering the problems in *section 1.4*.

5.1 Prototype one

In this section the result of the first prototype will be analyzed and evaluated in order to justify the conclusions, regarding the first prototype, stated in this thesis.

5.1.1 Frames per second

As the results show, the final version of the first prototype managed to output an average of 17 and 13 fps for the rear view and BEV respectively. Both views sum up to 30 fps which is the maximal frame rate the CCM can output. Thus, the prototype manages to handle all frames delivered by the CCM and causes no loss of efficiency. The reason for the difference in frame rate between the rear view and BEV can be explained by the measurements performed on the CCM, as shown in *fig. 10*. The result of this measurement shows that the switch from rear view to BEV takes approximately 33 ms longer than the switch from BEV to rear view. When rendering a video in 30 fps, as the CCM does, one frame is outputted every 1000 ms/30 $\approx$ 33 ms which shows the CCM requiring one additional frame to switch to the BEV. Since it takes one additional frame to switch to the BEV, the rear view is going to be allowed to render one additional frame for each switch. Thus the rear view is going to provide a higher frame rate than the BEV. A potential solution to this would be to alter the thread requesting views from the CCM, to make it wait an additional 33 ms before requesting the rear view.

5.1.2 Perceived latency

After comparing the results regarding perceived latency with the reference values, it is clear that the first prototype does increase the latency in the system. The average latency in of the rear view is 280 ms compared to 173 ms in the reference measurement. As a result, the average increase in latency is 107 ms. The increase is approximately equal in the BEV, where the latency went from 203 ms in the reference measurement to 314 ms in the prototype, an increase of 111 ms. Additionally, the fluctuation in the latency increase in line with the time between each request from the thread requesting views from the CCM. This is illustrated in *figs. 11 and 12* where the time between each request is 10 ms and 150 ms respectively. The same behavior can be observed in *figs. 14 and 15*. Aforementioned behavior is a result of the increasing time the rendering surfaces are being allowed to render since the presently rendering surface will have a shorter delay closer to the reference. However, at the expense of increasing the latency in the currently frozen surface. In the case of a system such as a PAC, predictability is important since the contrary can give the driver a false sense of security. Moreover, a false sense of security can lead to accidents in the form of the driver assuming their path is clear. Thus, consistency of the perceived latency can be considered an important factor when evaluating the quality of a PAC. With this in mind, it can be concluded that using a shorter interval between each request to the CCM will increase the quality of the prototype.

The importance of keeping the latency in the system to a minimum is illustrated in the following example, using values from the first prototype: A car reversing in a parking lot at ten kilometers per hour moves approximately 2,78 meters per second. Considering a latency of 203 ms, the car will move $2,78 \cdot 0,208 \approx 0.578$ m before the driver has a chance to react. By increasing the latency by 111 ms, in line with the first prototype, the car will under the same conditions move $2,78 \cdot 0,319 \approx 0.886$ m. This would be an increase of $0,308$ m, or 53%. Thus the question arose: does the benefit from a better overview of the car's surroundings exceed the cost of additional latency introduced into the system?

5.1.3 Analyzing the frequency of CCM requests

The frequency of how often requests were performed proved to affect the stability of the perceived latency. Two different frequency intervals were chosen: 10 ms and 150 ms. As a reference, the perceived latency when only rendering one of the views was measured. The acquired test results are illustrated in *figs. 11 to 16* in *appendix D*. As shown in the diagrams a higher frequency is preferable to a lower frequency in regards to stability of perceived latency. The reference diagrams *figs. 13 and 16* fluctuate the least of all comparisons due to the frame being updated at a constant 30 fps, whereas switching causes the two views to share the data bus which is limited to 30 fps. However, after observing *fig. 10*, page 43 it is apparent that the time it takes to actuate the requested signal is considerably longer than the frequency requests are sent in since the request are sent every 10 ms. Thus it seems that the CCM is capable of either queuing requests or handling multiple request concurrently, otherwise there would be no such differences between longer and shorter request intervals. One thing to keep in mind is that a higher request frequency requires more CPU resources, to a point where it's not viable. What is viable or not is a question in which the whole system must be considered and which is outside the scope of this thesis.

5.2 Prototype two

When analyzing the results from the second prototype, it is clear that the performance is inferior to the first prototype with regards to fps. The final version of the second prototype only renders each view with an average of 8 fps which is lower than the 17 and 13 fps respectively in the first prototype.

5.2.1 Lack of time

The second prototype was built on an already functioning example project implemented by VCC. However, this existing program was misunderstood to be rendering at a frame rate of 30 fps, when it was in fact only rendering at 8 fps. As a result, the time it would take to achieve a composite view prototype was underestimated. When realizing this fact, it became clear that the time left in the project would not be sufficient to make the prototype perform according to the expectations. Despite this, the development proceeded to analyze the cause of the limited performance with the goal of being able to resolve the issue.

5.2.2 Frame rate issues

The reason for the relatively poor frame rate remains unclear and further investigation was not possible due to the lack of time left in the project. However, what was found is that some users experience similar problems when utilizing OpenCV's built in functions for rendering. The functions in question are imshow() and waitKey() [14], which according to discussion at following forums [12] [13] could be the reason for poor performance in versions prior to OpenCV 4.0. The problems described coincide with the low frame rate associated with the prototype.

A minimal test setup was made to compare the performance of OpenCV's rendering functionality with a dedicated graphics library. The graphics library chosen for comparison is OpenGL [7] and the number of samples the comparison is based on is 2000 rendered frames. What was learned was that OpenCV took approximately 2.59 ms, on average, to render a frame while the solution with OpenGL took approximately 1.27 ms, on average, to render a frame under the same conditions. For the purpose of measurements, chrono [15] part of the C++ standard library header for date and time, was used to measure the duration of time it took for both solutions to render each frame. To achieve the most accurate timing available to the platform, chrono's high resolution clock was used.

The test showed OpenGL performing better than OpenCV for rendering. However, it could not be determined to be the sole issue of limited frame rate in the second prototype. Other possible limiting factors could be the image processing required to produce the BEV being inefficiently implemented or issues related to the retrieval of individual camera frames from the buffers.

5.2.3 Perceived latency

The perceived latency in the second prototype is 267 ms, on average, in both the BEV and the rear view. This latency is similar to the latency of the first prototype and is likely a result of the limited frame rate of the prototype. Since the prototype only renders at 8 fps, the view of the surroundings are only being updated eight times each second. This results in a span of 1000 ms/8 $\approx$ 125 ms during which the event may have occurred, thus increasing the perceived latency. In contrast to the first prototype, the perceived latency is equal in both views, since both views render the same camera frames. This means that if the prototype was rendering in 30 fps as expected, both views would have a decreased latency.

5.2.4 Value of the prototype

As mentioned in *section 3.4.1*; Due to the lack of a CCM, all processing of camera streams had to be handled manually. The four camera streams had separate frame buffers from which it was possible to fetch the most recent frame. As a result, it was possible to create a view containing both the BEV and the rear view using the same frame in both views. Thus the second prototype was not limited to render a frame in either the BEV or the rear view. This implied that both views would theoretically be able to achieve a frame rate of 30 fps. However, there were still restrictions on how many operations that could be performed without exceeding the time window of ≈ 33 ms when rendering in 30 fps. To clarify, since frames were arriving from the cameras every 33 ms, the frame had to be rendered in this time window, otherwise frames would arrive at a higher rate than they were rendered, leading to the frames either being dropped or the camera stream having an increased perceived latency due to frames being queued.

Even though the prototype was not performing as expected, in regards to frame rate, the results still provided valuable information. Since the existing program was only rendering at 8 fps to begin with, the implementation of the composite view did not affect the performance negatively. Considering the CCM being the bottleneck in the camera pipeline of the first prototype, it was likely that the absence of a CCM would allow the second prototype to render multiple camera streams without losing performance, assuming this fact applied to any frame rate. However, this could not be confirmed in this thesis.

5.3 Choice of measurements

This section is devoted to clarifying the choice of measurements.

5.3.1 Frames per second

To evaluate the performance of the two prototypes developed for this thesis, two different measurements have been conducted on each prototype. The first measurement was the average frames per second output to the display which was measured as explained in *section 3.3.1*. Worth noticing is the adaptability of the methods used to measure fps and perceived latency. The only requirement to be able to use the same fps measurement class is that the rendering has to be done in C++. Furthermore, since the measurement of perceived latency is performed independently from the system it is measuring, it can be performed in the same manner with equally accurate result on any system, provided the same equipment is used.

Also worth noticing is that when calculating the average fps in the manner explained in *section 3.3.1*, the frames are not guaranteed to be divided evenly over time. Since only one surface can render at any given point in time, the other surface is frozen in the meantime. Thus when one surface is rendering, it is rendering at a higher frame rate than the calculated average. For example; A surface rendering in 30 fps for 5 seconds, and then stopping for 5 seconds, will have an average frame rate of 15 fps even if it is frozen for 5 seconds. This scenario, however, could not be considered a viable solution for a PAC since it would not fulfill the goal of providing the driver with a real time view of the surroundings. As a result, this measurement was primarily used to determine if the prototype made use of all the delivered frames. Since the performance of the prototype is already being limited by the nature of the CCM, it is important to make use of as much resources as possible and not to introduce further restrictions to the performance. By measuring the sum of the average fps for both rendering surfaces one can determine if the sum equals the amount of frames per second being delivered by the CCM. However, measuring the average fps cannot provide enough information to determine how the other performance factors like latency, is affected by the prototypes in this thesis. Thus, it is important to not only calculate the average fps, but also calculate the latency the prototype introduce into the system.

5.3.2 Perceived latency

As explained in *section 3.3.2*, the measured latency of the prototype is the latency the driver experience from an event occurring until it is rendered on the screen. This measurement was chosen as a result of the prototypes application, as explained in *section 1.2.* For the prototype to be viable as a Park Assist Camera, the delay has to be kept to a minimum. A PAC is a time critical system, since latency in the system can cause accidents in traffic as the driver may not be able to take action to an event they have not yet been able to see. As a result, perceived latency could be considered a suitable measurement to evaluate the performance of the prototype.

6 Conclusion

To conclude this thesis, the results have been analyzed in order to evaluate the accomplishment of the goal defined in *section 1.3*.

1. **How the first prototype is limited by the CCM**

   The limitation of the CCM is a lack of the ability to customize the frames being outputted to contain both the BEV and the rear view. This limitation requires partitioning of the output stream by alternately switching between either the BEV or rear view. Therefore, the BEV and rear view cannot both achieve a frame rate of 30 fps.

2. **Deficiencies in the first prototype**

   The prototype introduces two deficiencies to the system. Firstly, the frame rate is reduced to 13 and 17 fps for the BEV and rear view respectively, as mentioned in *section 4*. This, in contrast to only rendering a single view, causes a performance drop of 56% for the BEV and 43% for the rear view when comparing with the single view with update frequency of 30 fps. Secondly the perceived latency is increased, partly as a result of the reduced frame rate.

3. **How the performance compares between the first and second prototype**

   The first prototype provides a higher frame rate than the second prototype. However the perceived latency is similar in both prototypes.

6.1 Future work

Considering the first prototype, an improved result is not achievable without modifications to the CCM to make it possible for frames on the output stream to contain both views simultaneously. Therefore, there is no apparent further development to be made on the first prototype until this issue has been resolved.

Since the cause of the limited frame rate in the second prototype could not be determined, it would be prudent to further investigate the bottleneck of this prototype. A suggestion for further development is to perform the rendering using OpenGL instead of OpenCV since OpenCV might be the bottleneck considering the measurements performed in *section 5.2.2*.

6.2 Reflection of ethical aspects

The first aspect concerning public utility for the prototypes developed during this thesis is that they both aim to give the driver a more complete overview of the cars surroundings. One can argue that this will increase the safety, both for the driver and for those in the vicinity. However, as the example in *section 5.1.1* shows, the prototype can give the driver a false sense of security due to the increased latency in the system. The increased risk of inflicting damage is an indication that the system is not fit for use on it's own.

The second aspect is that it is an aid for individuals with limited mobility and injury, who find it problematic to look over their shoulder in parking situations. Utilities like this can be helpful even for those without injury or mobility issues. In some situations it can be hard to observe everything ongoing in the cars surroundings and in those situations utilities of this kind can be of great help. The third aspect is that the driver does not need to manually switch between the rear view and BEV which can cause a shift in focus for the the driver, assuming the car is not stopped during the change of view. The shift in focus means that the driver is not aware of the car's surroundings in that moment, thus the risk of an accident happening increases. As good as any feature might be, security is the one thing manufacturers can't overlook. Thus, meticulous testing and evaluation is needed before features like these go into production.

7 Terminology

1. IHU - Infotainment Head Unit; An embedded system responsible for coordination and presentation of information.
2. EVS - Exterior View System; A part of the HAL which provides an interface for access to the onboard cameras of the car.
3. CCM - Camera Connection Module; The module explained in *section 2.4*.
4. BEV - Bird's-eye view; Explained in *section 2.5*
5. PAC - Park Assist Camera; Module consisting of the four cameras on the car as well as the software interacting with the cameras.
6. DSF - Data Signaling Framework; A term describing several proprietary components of the internal framework used for communicating signals in the car.
7. Performance - The combination of perceived latency & fps.
8. Fps - frames per second; The average amount of frames rendered over a finite amount of time.
9. Composite view - The combination of BEV & rear camera view displayed simultaneously on the screen.

8 References

[1] *Laboratory test procedure for FMVSS 111.* Tech. rep. National Highway Traffic Safety Administration, 2018. URL: https://www.nhtsa.gov/sites/nhtsa.dot.gov/files/documents/tp-111-v-01-final.pdf (visited on 16/05/2019).

[2] *Vehicle Camera HAL.* Android Open Source Project. URL: https://source.android.com/devices/automotive/camera-hal (visited on 07/05/2019).

[3] *Android Open Source Project.* URL: https://source.android.com/ (visited on 06/05/2019).

[4] *Overview of JNI.* 24th Oct. 2014. URL: https://www.ibm.com/support/knowledgecenter/en/SSYKE2_8.0.0/com.ibm.java.vm.80.doc/docs/jni_overview.html (visited on 09/04/2019).

[5] *Curvilinear perspective - Oxford Reference.* DOI: 10.1093/oi/authority.20110803095654622. URL: http://www.oxfordreference.com/view/10.1093/oi/authority.20110803095654622 (visited on 26/04/2019).

[6] *OpenCV: About.* URL: https://opencv.org/about/ (visited on 05/05/2019).

[7] *OpenGL - The Industry Standard for High Performance Graphics.* URL: https://www.opengl.org/ (visited on 02/05/2019).

[8] *Extreme Programming Rules.* URL: http://www.extremeprogramming.org/rules.html (visited on 03/05/2019).

[9] *ccache - ArchWiki.* URL: https://wiki.archlinux.org/index.php/ccache (visited on 02/05/2019).

[10] *Android Build System - eLinux.org.* URL: https://elinux.org/Android_Build_System (visited on 02/05/2019).

[11] *LinearLayout.* Android Developers. URL: https://developer.android.com/reference/android/widget/LinearLayout (visited on 03/05/2019).

[12] *imshow() very slow - OpenCV Q&A Forum.* URL: https://answers.opencv.org/question/96608/imshow-very-slow/ (visited on 01/05/2019).

[13] *waitKey(1) timing issues causing frame rate slow down - fix? - OpenCV Q&A Forum.* URL: https://answers.opencv.org/question/52774/waitkey1-timing-issues-causing-frame-rate-slow-down-fix/ (visited on 01/05/2019).

[14] *OpenCV: High-level GUI.* 2018. URL: https://docs.opencv.org/3.4.5/d7/dfc/group__highgui.html\#ga453d42fe4cb60e\\5723281a89973ee563 (visited on 01/05/2019).

[15] *Standard library header <chrono> - cppreference.com.* URL: https://en.cppreference.com/w/cpp/header/chrono (visited on 02/05/2019).

Appendix

A First version of pixel analysis algorithm

The following code is an extract from the first working version of prototype one, suffering from dropped frames. To avoid infringing the non-disclosure agreement with VCC, proprietary parts of the code have been omitted. However, the content is a valid representation of the functionality.

```
constexpr uint32_t BLACK = 0;

JNIEXPORT
jint JNICALL
    Java_com_volvocars_pactestapp_Evs_glBindVideoTextureCompositeView(
    JNIEnv* /*env*/,jobject /*caller*/, jint image) {

    /*Omitted code due to confidentiality*/

    sp<GraphicBuffer> client_buffer;
    int result = pac_client->GetClientBuffer(&client_buffer);

    uint32_t* pixels = nullptr;
    client_buffer->lock(client_buffer->getUsage(), (void**) &pixels);
    //pixels now points to the first pixel of the frame.

    //stride is the amount of pixels in a row.
    uint32_t stride = client_buffer->getStride();

    //The first index is the pixel at {640,100}
    int index = 640 + 100*stride;


    //Assume that the column is black
    bool isBlackColumn = true;


    //Check if 100 pixels in a column are all black
    for (int i = 0; i < 100; ++i){
        index += stride;
        //Incrementing index by one stride will move it to the next
    pixel in the same column

        uint32_t pixelValue = pixels[index];
        if (pixelValue != BLACK) {
            //As soon as a pixel with color is detected. Break.
            isBlackColumn = false;
            break;
        }
    }

    if(isBlackColumn){
        if(image == BEV){
            // Refresh frame.
        }
        else{
            // Keep previous frame.
        }
    }
    else {
        if(image == BEV){
            // Keep previous frame.
        }
        else {
             // Refresh frame.
        }
    }
    return result;
}
```

B Second version of the pixel analysis algorithm

The following code is an extract from the second working version of code. To avoid infringing the non-disclosure agreement with VCC, proprietary parts of the code have been omitted. However, the content is a valid representation of the functionality.

```
constexpr uint32_t BLACK = 0;

JNIEXPORT
jint JNICALL
    Java_com_volvocars_pactestapp_Evs_glBindVideoTextureCompositeView(
    JNIEnv* /*env*/, jobject /*caller*/, jint visnImg) {

    //Omitted code due to confidentiality

    sp<GraphicBuffer> client_buffer;
    int result = pac_client->GetClientBuffer(&client_buffer);

    if (result != ClientResult::SUCCESS) {
        return result;
    }

    uint32_t* pixels = nullptr;
    client_buffer->lock(client_buffer->getUsage(), (void**)&pixels);

    // pixels now points to the first pixel of the frame.

    // stride is the amount of pixels in a row
    uint32_t stride = client_buffer->getStride();

    //The first index is the pixel at {640 ,100}
    int index = 640 + 100 * stride;

    // Assume  that  the  column  is  black
    bool isBlackColumn = true;

    // Check if 100 pixels in a column are all black
    for (int i = 0; i < 100; ++i) {
        /*Incrementing  the  index by one stride will move it to the
    next pixel in the column */
        index += stride;
        uint32_t pixelValue = pixels[index];

        //As soon as a pixel  with  color is  detected. Break.
        if (pixelValue != BLACK) {
            isBlackColumn = false;
            break;
        }
    }

    if (isBlackColumn) {
        if (visnImg == BEV) {
                // Refresh and discard frame.
        } else {
            result = ClientResult::NO_NEW_FRAMES;
        }
    } else {
        if (visnImg == BEV) {
            result = ClientResult::NO_NEW_FRAMES;
        }else {
            //Refresh and discard frame.
        }
    }

    return result;
}
```

C Rendering the second prototype

The following code is an extract from the final version of the second prototype. To avoid infringing the non-disclosure agreement with VCC, proprietary parts of the code have been omitted. However, the content is a valid representation of the functionality.

```
int main
{
    cv::Mat generateCameraControlView(gen_BEV & gb, cv::Mat back, cv::Mat bev)
    {
        int singleWidth = 600;
        int singleHeight = 500;
        int bevHeight = singleHeight * 2;
        int bevWidth = singleWidth;
        int outputWidth = bevWidth;
        int outputHeight = bevHeight + singleHeight;

        //Undistort the rear view
        cv::cuda::GpuMat cuda3 = gb.undistort_fisheye_img(back, "back");
        cv::cuda::GpuMat cudaBev(bev);
        cv::cuda::GpuMat cudaOutputImage = cv::cuda::GpuMat(outputHeight, outputWidth, CV_8UC3, Scalar::all(0));

        //Create a cv::mat which will contain the BEV
        cv::cuda::GpuMat roi = cudaOutputImage(Rect(0, 0, bevWidth, bevHeight));
        cv::cuda::resize(cudaBev, roi, cv::Size(bevWidth, bevHeight));

        //Update cv::mat to be able to also contain the rear view
        roi = cudaOutputImage(Rect(0, bevHeight, singleWidth, singleHeight));
        cv::cuda::resize(cuda3, roi, cv::Size(singleWidth, singleHeight));

        return cv::Mat(cudaOutputImage)
    };

    while (key != 'q')
    {
        //Wait for at least 1 ms to allow time for rendering image
        key = cv::waitKey(1);

        //Check if any new frames are available.
        if (areFramesAvailable(frameProducer, NUM_OF_CAMERA))
        {
            //Retrieve the most recent frame from each camera.
            frameProducer->getFrame(0, &leftImage, &timestampLeft);
            frameProducer->getFrame(1, &rightImage, &timestampRight);
            frameProducer->getFrame(2, &frontImage, &timestampFront);
            frameProducer->getFrame(3, &backImage, &timestampBack);

            //Create the BEV from the four individual images.
            bev = GEN_BEV.input4_to_BEV(leftImage, rightImage,
    frontImage, backImage);

            //Create a view to render to the display.
            view = generateCameraControlView(GEN_BEV, backImage, bev);

            //Output view for OpenCV to render.
            cv::imshow(birdsEyeViewName, view);
        }
    }
}
```

D Graphs

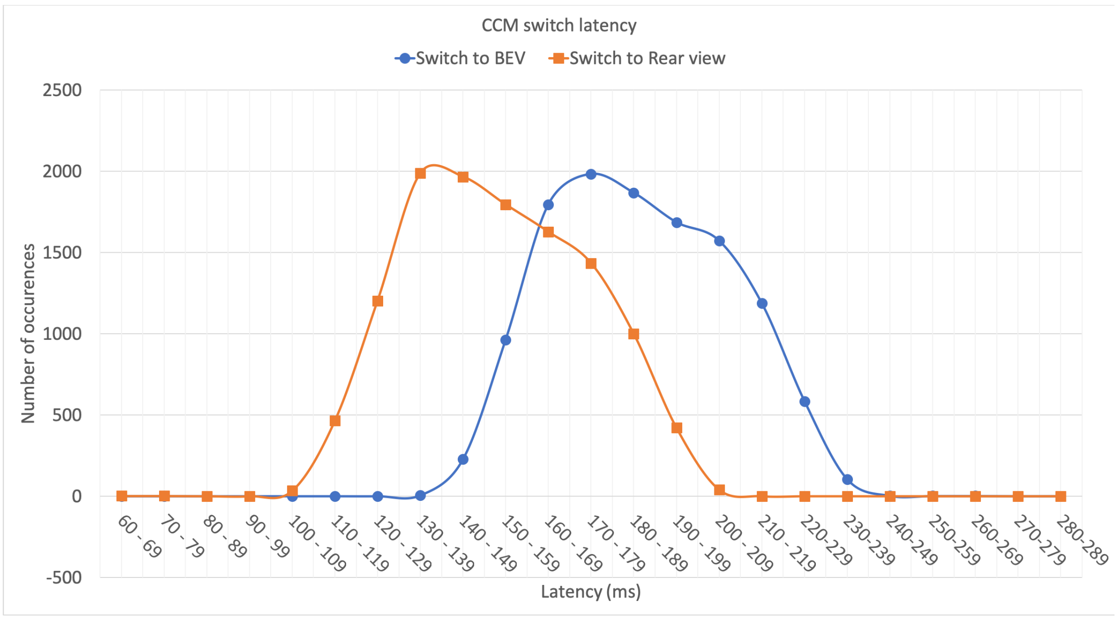


Fig. 10 – Statistics over the latency from request of video stream until first frame is delivered by the CCM.

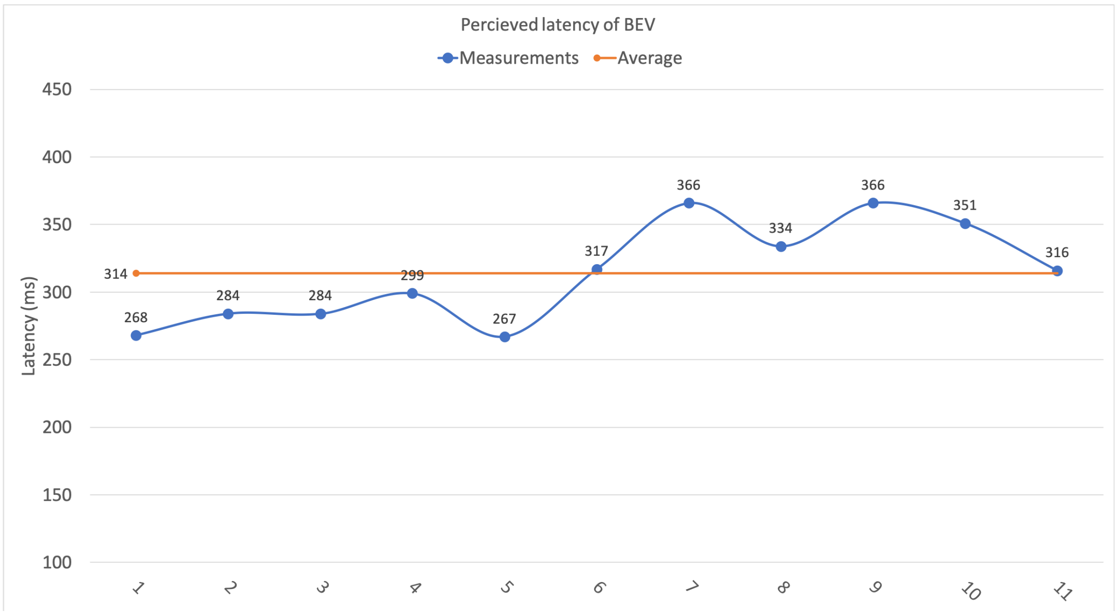


Fig. 11 – Perceived latency of BEV when switching camera stream every 10 ms

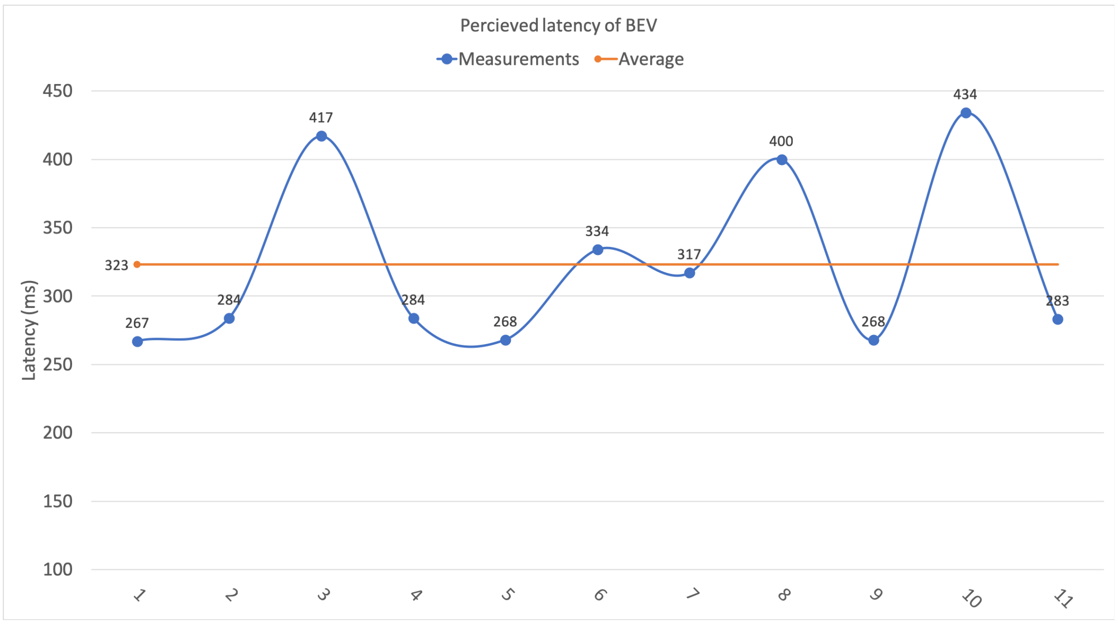


Fig. 12 – Perceived latency of BEV when switching camera stream every 150 ms

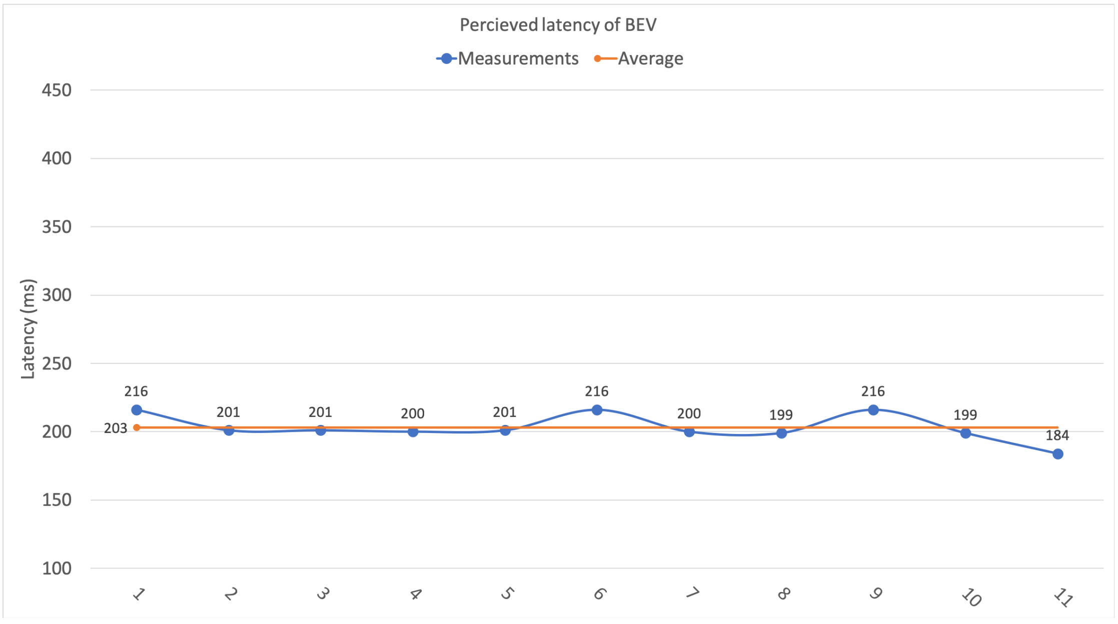


Fig. 13 – Perceived latency of BEV when not switching camera stream

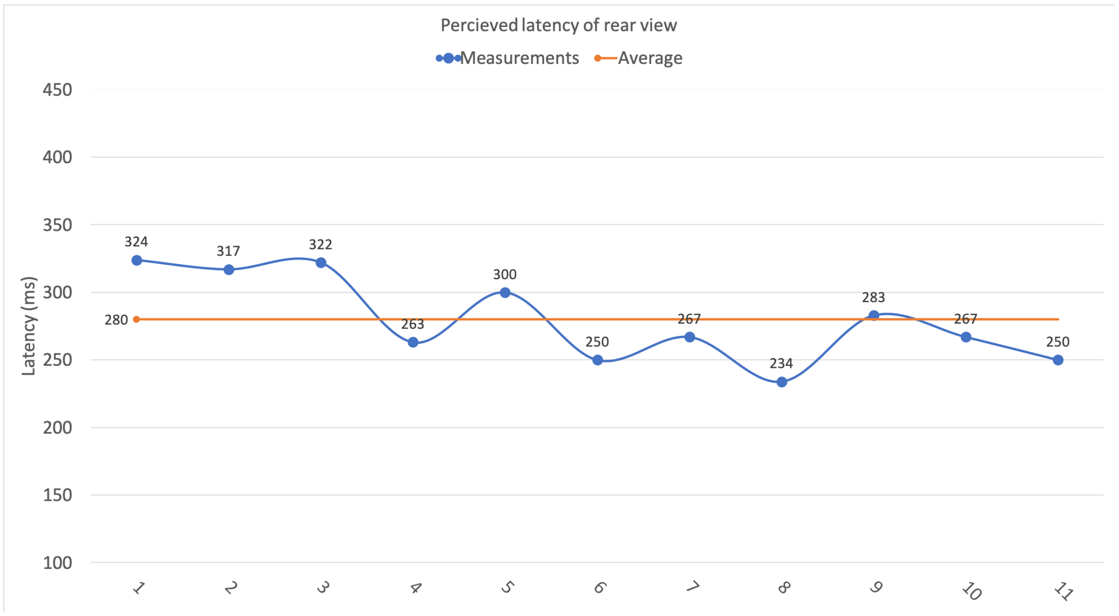


Fig. 14 – Perceived latency of rear camera when when switching camera stream every 10 ms

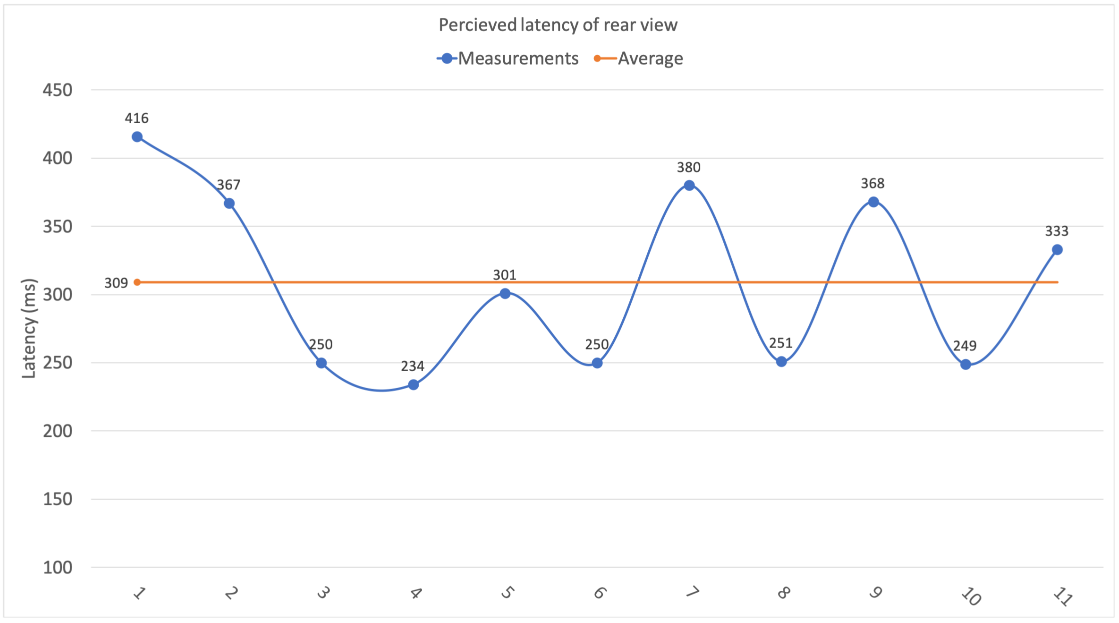


Fig. 15 – Perceived latency of rear camera when when switching camera stream every 150 ms

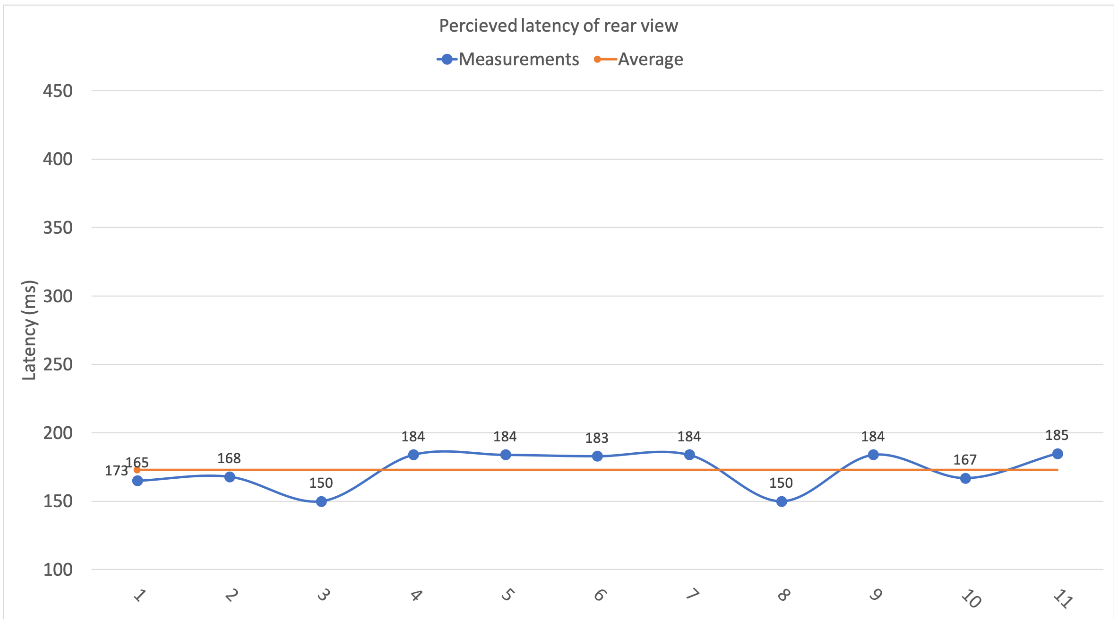


Fig. 16 – Perceived latency of rear camera when not switching camera stream

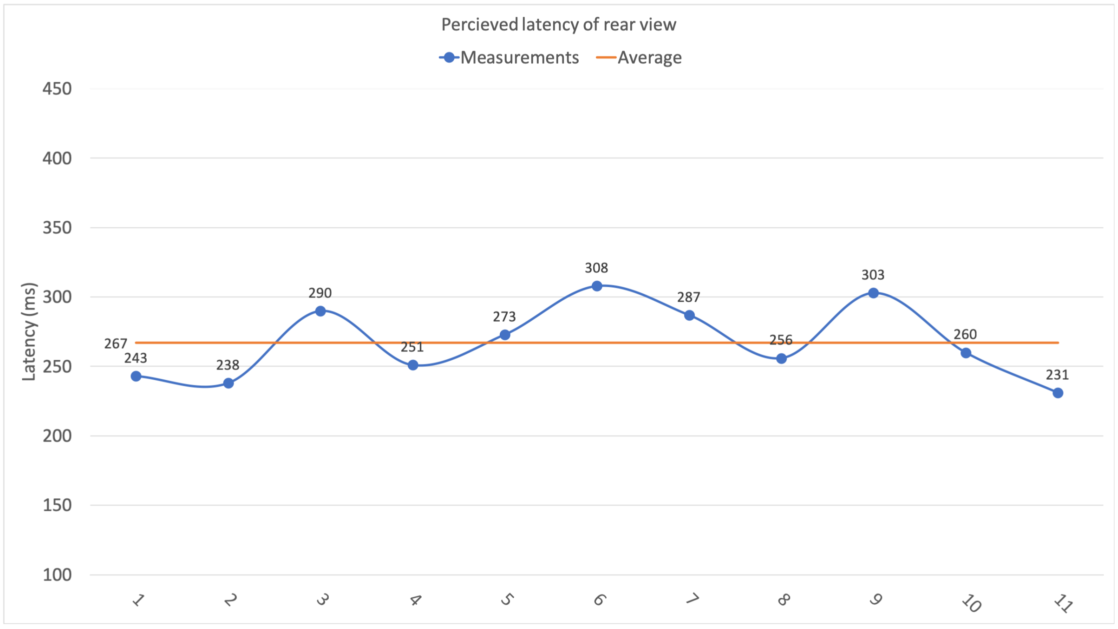


Fig. 17 – Perceived latency of rear camera in the second prototype

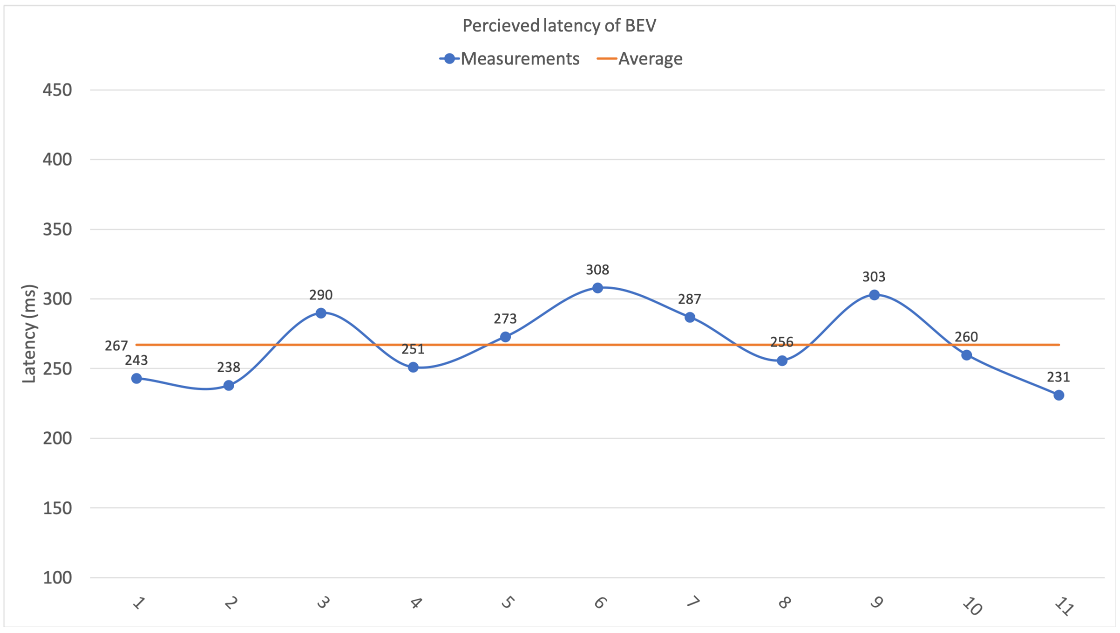


Fig. 18 – Perceived latency of BEV in the second prototype