UNIVERSITÄT
DES
SAARLANDES

Telecommunications Lab

# Provision of Multi-Camera Footage for Professional Production and Consumer Environments

Dissertation

zur Erlangung des Grades
*des Doktors der Ingenieurwissenschaften (Dr.-Ing.)*
der Fakultät für Mathematik und Informatik
der Unversität des Saarlandes

vorgelegt von
**Tobias Lange**

Saarbrücken, 2021

**Öffentliches Kolloquium**

| | |
|---|---|
| Tag des Kolloquiums: | 16. Dezember 2021 |
| Dekan der Fakultät: | Prof. Dr. Thomas Schuster |

**Prüfungsausschuss**

| | |
|---|---|
| Vorsitzender: | Prof. Dr. Jan Reineke |
| Berichterstatter: | Prof. Dr. Thorsten Herfet |
| | Prof. Dr. Aljosa Smolic |
| | Dr. Joachim Keinert |
| Beisitzer: | Dr. Jan Alexandersson |

**Short Abstract**

Multi-camera footage contains much more data in comparison to that of conventional video. While the additional data enables a number of new effects that previously required a large amount of CGI magic and manual labor to achieve, it can easily overpower consumer hardware and networks. In this thesis, we explore the necessary steps to create an interactive multiview streaming system, from the cameras via the compression and streaming of the material, to the view interpolation to create immersive perspective shifts for viewers. By only using freely available consumer hardware, and making sure all steps can run in real-time, in combination with the others, the benefits of multi-camera video are made available to a wider public.

With the construction of a modular camera array for lightfield recording, we highlight the most important properties of such an array to allow for good post-processing of the recorded data. This includes a flexible yet sturdy frame, the management of computational and storage resources, as well as the required steps to make the raw material ready for further processing. The *Unfolding* scene displays the possibilities of lightfields when a good camera array is combined with the talent of professional visual effect artists for the creation of future cinematic movies. Furthermore, we explore the benefits of 5D-lightfield video for scenes with fast motion, using precisely controlled time delays between the shutters of different cameras in the capturing array.

**Zusammenfassung**

Multi-Kamera-Aufnahmen enthalten wesentlich mehr Daten als konventionelles Video. Auch wenn diese zusätzlichen Daten zahlreiche neue Effekte ermöglichen die bisher nur mit künstlichen CGI-Effekten und viel Handarbeit machbar waren, kann ihre Größe die Hardware und Netzwerke normaler Verbraucher schnell überlasten. In dieser Arbeit stellen wir die Schritte vor die nötig sind um ein interaktives Multiview-System zu erstellen. Dies beinhaltet alles von der Kamera über die Komprimierung und Übertragung des Materials, bis hin zur Ansichtsinterpolation mit der perspektivische Änderungen auf dem Empfänger berechnet werden können. Indem wir uns auf Endnutzer-Hardware beschränken und alle Schritte darauf in Echtzeit ablaufen lassen, machen wir die Vorzüge von Multi-Kamera-Aufnahmen für ein breiteres Publikum erfahrbar.

Mit der Konstruktion eines modularen Kamera-Arrays zeigen wir auf welche Eigenschaften nötig sind um das Material mit hoher Qualität verarbeiten zu können. Dabei werden der flexible aber stabile Rahmen, die Verwaltung der Speicher- und Rechner-Ressourcen, sowie die Vorverabeitung besprochen. Die *Unfolding*-Szene zeigt Möglichkeiten von Lichtfeldern für zukünftige Filmproduktionen auf, wenn gute Aufnahmen mit dem Talent von Experten für visuelle Effekte kombiniert werden. Darüber hinaus diskutieren wir die Vorteile von 5D-Lichtfeldern mit präzise kontrollierter Verzögerung zwischen den Auslösern der Kameras in Szenen mit sehr schnellen Bewegungen.

**Abstract**

Multi-camera footage contains much more data in comparison to that of conventional video. While the additional data in the added views enables a number of new effects that previously required a large amount of CGI magic and manual labor, it can easily overpower consumer hardware and networks. Those effects can provide a more immersive viewing experience, such as viewer-dependent perspective shifts, or give more artistic freedom to content creators with focus adjustments and lens changes in post-processing or resolution upscaling for single views. In this thesis, we explore the necessary steps to create an interactive multiview streaming system, from the cameras via the compression and streaming of the material, to the view interpolation to create immersive perspective shifts for viewers. Every part of that pipeline contains novel ideas or approaches to make the step faster, more efficient, or more flexible with respect to required hardware resources. By only using freely available consumer hardware, and making sure all steps can run in real-time, in combination with the others, the benefits of multi-camera video are made available to a wider public. The major steps are the distribution of the complexity of an H.264/MVC-compliant encoder over multiple simpler instances, the optimization of multiview video streaming over restricted channels by using the view interpolator's capability to reconstruct missing views, and a novel quality metric based on PSNR, which is capable of judging the perceived quality of interpolated views much better than the original PSNR. A GPU-based view interpolation algorithm that does not rely on depth maps can be used to run the pipeline in real-time.

With the construction of a modular camera array for lightfield recording, we highlight the most important properties of such an array to allow for good quality post-processing of the recorded data. This includes a flexible yet sturdy frame, the management of computational and storage resources, as well as the required steps to make the raw material ready for further processing. The *Unfolding* scene displays the possibilities of lightfields when a good camera array is combined with the talent of professional visual effect artists for the creation of future cinematic movies. Furthermore, we explore the benefits of 5D-lightfield video in scenes with fast motion, using precisely controlled time delays between the shutters of different cameras in the capturing array.

In summary, we contributed to the scientific progress of every step in a multiview video streaming pipeline. Even though not every part of this work received the same amount of publicity as the camera array, which was featured in multiple popular and industry-specific media as well as multiple scientific publications due to its features, flexibility, and the quality of results, all of them received good reviews for their respective publications. Improvements with a focus on speed increase the performance of the base algorithms at least ten-fold with comparable if not improved quality. Others present novel ideas or improve the quality of specific cases we encounter in our scenarios.

**Acknowledgments**

Even though this thesis is an original work of mine, it would not have been possible without the support of many others. Special thanks go to my thesis supervisor Prof. Thorsten Herfet who provided meaningful guidance whenever necessary, but also enough freedom to follow my own path in my research. Only his trust in my technical abilities and planning skills made the creation of the camera array and many other projects possible. I also value his patience for my lengthy writing process, especially during this thesis, which took way too long even for my taste.

Since every long-running project has good and bad times, I count myself lucky for the support I got from everyone during my time at the Telecommunications Lab. This includes my family with their never-ending emotional support, an open ear for all kinds of problems, and their efforts to keep my motivation up in trying times. In the office, I could always rely on my colleagues to be available for professional discussions or relaxed banter to clear my head between work topics. Christopher Haccius, and Andreas Schmidt were amazing office mates and always ready to bounce ideas off of. With Pablo Gil Pereira, Kelvin Chelli and Harini Hariharan I shared memorable experiences in and out of the office, at conferences, and elsewhere, which always helped to lighten the mood. Zakaria Keshta, your enthusiasm to help with any kind of occurring problem are very much appreciated.

I could also not have done it without the students I had the pleasure to advise on their way to the final theses. A big thank you goes to Alexander Blatt, Johannes Reuter, Pascal Hennen, Kim Hao Josef Nguyen, and Pascal Straub for their efforts to implement my sometimes seemingly crazy ideas and helping to make them a reality. I really enjoyed the long discourses and debugging sessions we had and hope you can still benefit from our time together. Frank Waßmuth with his special status as a student and colleague deserves a separate mention for his great expertise in practical networking and the time we shared in the university's sports programs.

Outside the office, my network of close friends was always there in case distraction, relaxation, or even technical discussions were needed. I am not going to refer to all of you by name, but you know who you are.

Thanks to everyone who volunteered to read this thesis in its unfinished state. Even though you caused me a couple of stressful days and sleepless nights, this thesis would not have been the same without your suggestions and comments.

In case I forgot to mention anyone's direct or indirect support, feel free to contact me and I am sure we will find an appropriate compensation.

# Contents

# List of Figures

# List of Tables



# List of Algorithms

# 1. Preface

## 1.1. Research Questions

While lightfield and multiview video present numerous open research topics, in this dissertation, we concentrate on the following questions:

**Q1: Do multiview and lightfield capture have the potential to replace conventional photographic capture techniques?**
Conventional photographic capture techniques have been around at least since the beginning of the 19th century. Since then, a lot of improvements have been made, including capturing quality, speed, and fidelity. The biggest steps were the switches from grayscale to color images and from film-based capture to digital image sensors. Each one was accompanied by doubts about its usefulness and its quality compared to the existing technology. Multiview and lightfield capture could be the next step in imaging technology, going from single- to multi-perspective images. However, if it should have any future success in replacing current capturing techniques, the industry with the highest demand for high-quality images must be convinced.

At the moment, that sector is undoubtedly the professional movie industry, including movie makers and broadcasters. One of the reasons they require the best possible quality is the fact that the majority of their captured material is run through multiple post-processing and compositing steps. For those, they often require accurate depth maps or other scene information which can only be acquired when the camera position is known precisely. To measure the position with the required precision, very expensive mounting rigs are needed, in addition to the cost of the professional camera equipment. With a multiview or lightfield approach, only the relative position between the cameras has to be known for the measurements and this can be achieved cheaply and repeatedly with fixed mounting frames holding the cameras. The optical systems of the cameras can also be cheaper because many visual effects, which can only be achieved with special and expensive lenses in conventional film making, can be added in a physically correct fashion during the post-processing of lightfields. Missing sensor resolution can also be recovered using lightfield-based super-resolution techniques. The combination of partially novel post-processing options and the possibility of using cheaper off-the-shelf hardware make lightfield capturing systems a valid contender for new studio installations or film productions.

**Q2: How can lightfield technology be made available for professional productions?**
Lightfield capturing equipment of a sufficiently high quality for professional productions is not currently available commercially. Building an array from available off-the-shelf components is not feasible without extensive knowledge of mechanical construction, electronics, algorithms, computer systems, and networks. Even then, the resulting systems can be unwieldy and hard to maintain and operate.

To show that this does not always need to be the case, and to test the viability of lightfields in professional productions, we devised a way of constructing a modular and expandable

lightfield capturing system with all the required storage and computing power in a form factor that makes the whole system transportable. To be able to control the independent computers in the camera array, a cluster control system is deployed in order to keep track of the different software states for camera nodes, storage nodes, and the central server. In the rare case of hardware failures, an automated deployment system guarantees that the array is back up and running within minutes of the defective hardware being replaced. The secondary hardware layer responsible for controlling the camera shutters as well as the power for the camera nodes is based on a custom design. It is capable of either synchronizing the shutters of all cameras to capture 4D lightfield video or delaying the shutter of every camera by an individual time offset to create 5D lightfields with dynamic subframing. This prototype camera array has proven its effectiveness in multiple experimental productions which resulted in impressive lightfield assets. They demonstrate the capabilities of the array and the current state of available algorithms for pre- and post-processing of lightfield material. The camera array shows that lightfield technology can be deployed in a way that makes the required effort for professional productions feasible while keeping maintenance requirements low. Additionally, it can be operated without in-depth knowledge of the inner workings of every single component.

**Q3: How can the data rate of lightfield video be handled effectively?**
Lightfield video produces a vast amount of data because the amount of pixels contained in every frame is equal to that of a conventional video frame multiplied by the number of cameras in the array. In contrast to conventional video, a good compression technique for lightfields is hard to define as the footage is rarely viewed in its raw form, and the different available post-processing algorithms require various image features to work properly.

This problem is tackled by defining different usage scenarios for the captured footage and appropriate compression schemes for each of them. To satisfy the high requirements of professional film productions. a lossless compression scheme based on OpenEXR is deployed. This enables space-efficient storage of the footage on the array-internal storage system while maintaining the possibility to apply any post-processing algorithm, if needed. Since this approach is not capable of achieving real-time performance on the array's systems, the time required for storage and processing must be included in the shooting plans. For the cases in which real-time performance is required, but some image quality can be sacrificed, we created the world's first scalable compression scheme that creates video streams compatible with the H.264/MVC standard for more than two views. These two very different approaches show that a universal approach to lightfield compression might not be possible, however they prove that lightfield compression is possible when the future use of the footage is known as the amount of data required for lightfields can then be reduced to a manageable size.

**Q4: How can the processing of lightfield footage be designed to be practically applicable?**
Lightfield footage usually needs to be processed so it can be consumed by a user. To achieve the highest possible quality from the post-processing algorithms, it is important to start from the best source material. One major factor for this is the demosaicing which creates full-color images from the raw sensor data a camera captures. While there are many different approaches for this task out there, nearly no one considers the special properties a lightfield has compared to a single image. To show the potential benefits of this, we devised a learning-based demosaicing approach that fills in the missing pixel information from neighboring images in the array, instead of just the local neighborhood in the same image. This approach ended up being significantly better than the current state-of-the-art algorithm for single image demosaicing, even though it is limited to low-resolution footage.

Another factor that keeps lightfields from being widely adopted is the fact that, due to the vast amount of pixels in a lightfield, the data rate required for a video stream can easily overwhelm an average consumer internet connection, even when the stream is already compressed using existing standards. Luckily, similar to the demosaicing, lightfields and multiview video offer additional parameters which can be tuned for an optimal streaming performance over a bandwidth restricted channel. This includes skipping complete views during encoding and reconstructing them using the remaining images. For this, an algorithm that can predict the optimal quality of the sequence at the receiver and outputs the parameter set for encoding was created. By taking into account the capabilities of the view interpolation algorithm on the client device, as well as its available hardware resources, the approach can even be deployed in heterogeneous environments. It proves the usefulness of including view interpolation techniques in the video coding chain, especially for very restricted channels.

During the research for the streaming optimization, it became apparent that the existing image quality metrics were not well suited to judging the quality of interpolated views as they pick up on minuscule errors stemming from slight miscalculations in the scene depth, which are nearly imperceptible to a a human observer. This lead to the development of a technique that eliminates the small shifts before the image quality is evaluated.

View interpolation algorithms are very resource-intensive and often require depth maps or other scene geometry information in addition to the images to achieve good results. To make them useful in a consumer environment, we devised an algorithm that runs almost completely on a middle-class consumer GPU and achieves up to 50 frames per second on footage with a resolution of 1800x1500, while only utilizing the intrinsic and extrinsic camera parameters to process the input images. It produces the view of a virtual camera from an arbitrary position in the area covered by the capturing cameras, with a quality comparable to the state-of-the-art algorithms at the time.

Overall, we showed that several shortcomings of lightfield and multiview video can be remedied today and even common consumer electronics possess enough computational power for an interactive video streaming system based on multiview or lightfield video.

## 1.2. Contributions

The first scientific contribution of this work is the creation of a flexible and modular 5D-lightfield capturing array. For preliminary tests and as a proof-of-concept for existing algorithms, we created a custom multiview video capturing rig with five cameras. Its modular design and portability allowed us to demonstrate the capabilities of our streaming and compression efforts on many occasions, and was part of the first complete multiview video real-time streaming system without specialized hardware. It provided some challenges concerning camera calibration and synchronization, due to the choice of cameras and the camera mounting solution, but, after a thorough investigation, solutions were found and implemented.

We benefited a lot from the insights gained from the five-camera multiview video capturing system when it came to up-scaling to construct the world's first 5D lightfield camera array. While a system with 64 cameras poses new challenges, mainly around the resulting amount of data and the required control, for some aspects such as camera calibration the solutions from the smaller array can be transferred. This array was designed to enable scenes with a different number of cameras, a variety of camera baselines, and layouts with a very precise control of the individual camera shutters, including frequency and phase. The challenges appearing during post-processing of the captured material have been solved using

a combination of internal developments and cooperations with other European universities. The array's effectiveness has been demonstrated with multiple capturing sessions for the EU project SAUCE[1].

The amount of raw data produced by such an array can quickly become a prohibitive factor in consumer networks and systems. With HD resolutions, the sheer amount of data that has to be transferred is simply too big for computers using common network technologies. Moreover, compression techniques for this kind of data are either not fast enough for real-time transport, rely on specialized hardware components for sender and receiver, or do not support more than two views in one stream. In order to be able to present and prove the possibilities for multiview content in streaming systems with consumer hardware, we developed a scalable real-time compression system for a variable number of views whose output is compatible with the H.264/MVC standard. It allows for real-time encoding and decoding of multiview footage on consumer hardware, with a compression performance comparable to the reference implementation of H.264/MVC. For a high number of views, it requires a powerful machine, but being able to leverage existing hardware encoding capabilities means that even low-end machines can handle more than two views. This makes it unique within the ecosystem of other compression schemes for multiview video.

With the availability of real-time compression for multiview video, it became evident that this new video format enables new parameters for streaming optimization. The adaptation options of single view video, for a given transmission channel and codec, are mostly limited to quality parameters that control the quantization of image sections and the search for optimal correspondences in other frames to minimize the amount of residual data. Multiview video also has those options, as its compression schemes are often derived from single view ones, but they also offer the possibility to not encode certain views at all and reconstruct them from the remaining views on the receiver device, if required. This lead to the development of a system capable of predicting the quality of all views after decoding and view reconstruction, based on the available data rate on the channel, sparse information about the image content, and some knowledge of the algorithm used to reconstruct non-encoded views. It shows the benefits of skipping complete views during encoding and distributing the available data rate over the remaining views to achieve a proven higher overall quality at the decoder.

The work on the streaming optimizer highlighted the fact that quality metrics, which have been used to judge the quality of coding schemes and reconstruction techniques for video, have severe problems with the quality of interpolated views. While the mean opinion score for the interpolated images was quite high, as they showed only very few visible artifacts, the resulting score was very low and did not represent the perceived quality well. One option would have been to simply use a different metric, but since the quality of the interpolated views was only a portion of the whole transmission pipeline whose quality we wanted to measure, this was not possible. Combining different metrics using different ranges for their results is nigh on impossible to do properly. This lead to the creation of a technique to measure the errors, which are the main cause of the bad quality score, but are nearly invisible to the viewer, and add them to the reference image before applying the usual quality metrics. Those quality scores had a much better correlation with the perceived quality, but were still able to be combined with older results.

The last contribution is the development of a fast multiview interpolation algorithm that does not require depth maps as input. Even though the main development of this algorithm happened during the author's master's thesis, it has been continuously updated throughout the duration of this work. Updated features include better filtering processes,

[1]SAUCE - Horizon 2020 Grant Agreement ID 780470 - `https://www.sauceproject.eu`

along with several other steps to incorporate a temporal consistency mechanism into the algorithm, making it even more time-efficient. When it was first developed, it was the fastest algorithm of its kind which was able to achieve an image quality comparable to other cutting edge approaches. Since then, new algorithms of a higher overall quality have been published, but many of them have not yet come anywhere close to the speed of our algorithm.

Overall, there are contributions to every part of a multiview streaming pipeline for real-time applications. Most contributions were either worldwide firsts or a significant jump in speed with close to state-of-the-art quality. The final big camera array was also a notable contribution to the lightfield community, as it enables the production of real-life lightfield video samples for the verification of novel algorithms, and even for completely new research areas like 5D lightfield video.

# 2. 5D Lightfield Array

One of the most important factors for the successful introduction of a new media format into widespread use is the ease of capturing it. Even though handheld devices for lightfield capture exist, for example, the Lytro Illum or the products from K-Lens[1], some of them lack the ability to capture video and all of them offer only a single view configuration. For research and professional use cases, this is not sufficient. The choice of available multiview and lightfield samples is still limited, especially for video material. Since the creation of synthetic lightfield videos requires a significant amount of artistic talent, as well as a lot of computing power, to come even close to a realistic appearance another option became necessary. Therefore, we decided to create a camera array that could be used to capture lightfield video footage of real scenes. Other camera arrays had previously been created, such as the Stanford Multi-Camera Array [14] or the Lytro Immerge 2.0, but they were either based on outdated hardware which could not produce samples with enough fidelity for modern applications or their construction details were never made public. Additionally, they are all no longer available and therefore cannot create new samples. In the following sections, we describe the steps we took to arrive at the current version of our camera array and the insights provided by the preliminary experiments.

## 2.1. Fundamentals

To give more context to the content of this chapter, some background information for the discussed topics is provided here.

Camera parameters play an integral role in the correction of visual distortions in images, due to imperfections in the optical systems as well as the projection of points between cameras. Without sufficient precision in their calculation, the processing of all multi-camera footage becomes far more complex and the quality of most existing algorithms drops significantly.

The technology behind digital cameras is assumed to be mostly clear to everyone. However, for certain special cases, details of the inner workings become increasingly important. This includes fast movements compared to the exposure time and the possibility to synchronize multiple cameras. Since this thesis includes chapters discussing multiview and lightfield content, knowing the distinction between them can be beneficial. This includes our definition of the dimensionality of visual media content. Backgrounds for some tools and frameworks which are repeatedly employed in this work are given to provide some knowledge of their capabilities and complexities.

### 2.1.1. Camera Technology

Most consumer devices with cameras, such as mobile phones and the majority of cameras in general, use a rolling shutter to control the exposure of a digital image sensor to capture photos or video. The name rolling shutter comes from analog cameras in which an opening in a shutter wheel was literally ”rolled” through the light’s path to expose the film behind,

[1] `https://www.k-lens.de`


Figure 2.1.: Differences between global (left) and rolling shutter (right) in images of a spinning airplane propeller.
Source: SmarterEveryDay - `https://youtu.be/dNVtMmLlnoE?t=150`

giving different portions of the film slightly offset exposure times. In digital cameras, where exposure is often directly controlled by the image sensor itself, it means the start and end times of light acquisition for each line of pixels on the sensor are slightly offset, creating effects similar to those of a mechanical shutter. On the other hand, sensors with global shutters expose the whole sensor at the exact same time, removing these effects.

In images from static scenes, there is no difference between the shutter types because it does not matter when certain parts of the sensor are exposed. When there is only slow movement in the scene, or the time it takes to activate the whole sensor is small compared to the chosen exposure time, the effects are hardly visible. Only for low exposure times and fast movements, especially rotations, does the difference between the shutter systems become obvious. Figure 2.1 shows two images of the same propeller of a small aircraft in flight. While the left image is shot with a global shutter and looks just like one would expect an airplane propeller to look like, the image on the right shows heavy distortions. The propeller blades do not even seem to be connected to the central hub and look more like boomerangs than straight propeller blades. This effect is caused by a short exposure time and the exposed area in the sensor moving comparably slow from the top to the bottom while the propeller keeps rotating.

Fast horizontal movements lead to less distorted, but equally visible effects which most people who have used a mobile phone to take pictures from a moving vehicle will recognize. The highlighted pole in Figure 2.2 seems to be leaning to the left, even though in reality it is perfectly perpendicular to the rail on top. That lean is created by the shutter moving down while the pole continues to go right. For the pole on the left, the effect is less severe because it is further away and, due to the perspective the image is taken from, it travels a smaller distance while the image is being taken. This dependence on speed is also the reason why the background of the image does not display any leaning effect.

Operating multiple devices in a synchronized fashion can be a substantial challenge, depending on the amount of deviation the specific use case allows for. In the range of hundreds of milliseconds, a trained team of human operators can suffice. For single to tens of milliseconds,

Figure 2.2.: Rolling shutter effects on fast straight motion. Even though it is perfectly perpendicular to the top railing, it appears to be slanted. The background shows no such effects.

triggering actions via a local network is precise enough, even when the devices are triggered sequentially. Below that range, more specialized approaches are necessary. Even when the internal clocks of the devices in question are synchronized via NTP [15] or PTP [16], and the actions are started at a predefined time in the future, the precision of time synchronization can become problematic [17]. In the microsecond range, even the scheduler in the operating system can add too much jitter [18].

When such precision is required, dedicated hardware solutions become necessary. They are used to either run all devices off of a single hardware clock or synchronize the clocks in the devices at regular intervals. In both cases, the influence of manufacturing differences of clock generators, which cause deviations from their nominal frequency, is nullified or significantly reduced. Broadcasters and other institutions working with visual media, call this approach GenLocking. Since it requires specialized circuitry to synchronize devices at the hardware level, devices with the ability to be genlocked are only available at a professional level and are quite expensive.

### 2.1.2. Camera Parameters

The camera parameters describe the position of a camera in space and the behavior of its optical system. For all cases discussed in this thesis, those parameters are split into intrinsic and extrinsic parameters.

Extrinsic parameters define the relative rotation and translation of a camera to a chosen origin. Since the extrinsic parameters are usually calculated between two or more cameras, the origin is often chosen as the position of one of these cameras. The translation vector $T$ and rotation matrix $R$ for this special camera are then defined as

$$T = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \quad R = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{2.1}$$

and the whole coordinate system is aligned with that camera. When observed from behind the camera, the x-axis points to the right, the y-axis points up and the z-axis goes through the center of the camera into the scene.

For other cameras, the translation vector is filled with the relative X-, Y-, and Z-components from the origin to the camera's position. Their rotation matrices contain the rotations around all three axes, combined into a single three-dimensional rotation matrix $R$, as shown below:

$$R = R_z(\gamma) \cdot R_y(\beta) \cdot R_x(\alpha) \tag{2.2}$$

$$= \begin{bmatrix} \cos\gamma & -\sin\gamma & 0 \\ \sin\gamma & \cos\gamma & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos\beta & 0 & \sin\beta \\ 0 & 1 & 0 \\ -\sin\beta & 0 & \cos\beta \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha \\ 0 & \sin\alpha & \cos\alpha \end{bmatrix} \tag{2.3}$$

$$= \begin{bmatrix} \cos\gamma\cos\beta & \cos\gamma\sin\beta\sin\alpha - \sin\gamma\cos\alpha & \cos\gamma\sin\beta\cos\alpha + \sin\gamma\sin\alpha \\ \sin\gamma\cos\beta & \sin\gamma\sin\beta\sin\alpha + \cos\gamma\cos\alpha & \sin\gamma\sin\beta\cos\alpha - \cos\gamma\sin\alpha \\ -\sin\beta & \cos\beta\sin\alpha & \cos\beta\cos\alpha \end{bmatrix} \tag{2.4}$$

While the extrinsic parameters form an important basis for the projection of pixels in an image into the 3D scene space, this coordinate projection is only accurate if the camera is a perfect pinhole camera. For good images, a perfect pinhole camera would need a vast amount of light in the scene or very long exposure times, due to the small opening (pinhole) through which light comes into the camera and falls onto the film or sensor. This makes them very inconvenient to use, but the resulting images are always in focus and have an unlimited depth of field. In most cameras today, the opening is much bigger and a system of lenses is added in front of the sensor to allow for dynamic focus, zoom, and aperture settings. Since those lenses are never perfect, the images on the sensors include distortions which cause certain scene points to appear in different positions in the image than where the pinhole camera model would predict.

The commonly used approaches for projecting pixels in images to scene points and back assume that the light transport in the cameras follows the pinhole model to simplify the calculations. This means when cameras behave differently, their images must be corrected first. To make those corrections possible, the introduced distortions have to be modeled and parameterized. The intrinsic camera parameters are the collection of the required parameters.

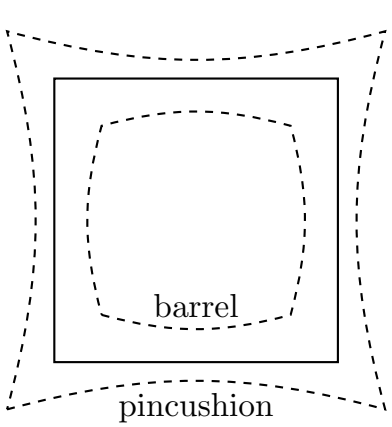
Weng *et al.* [19] describe a model which is very similar to modern calibration algorithms and describes the functionality of all parameters very clearly. The distortion parameters are coefficients for two different types of lens distortions, namely radial and tangential distortions. Radial distortions are caused by imperfections in the lens's shape. Their effects can be seen in Figure 2.3a. Depending on whether straight lines are bowing inwards or outwards, these effects are also known as pincushion or barrel distortions. The effects of these distortions can be described mathematically as:

$$x_{distorted} = x \cdot (1 + k_1 \cdot r^2 + k_2 \cdot r^4 + k_3 \cdot r^6) \tag{2.5}$$

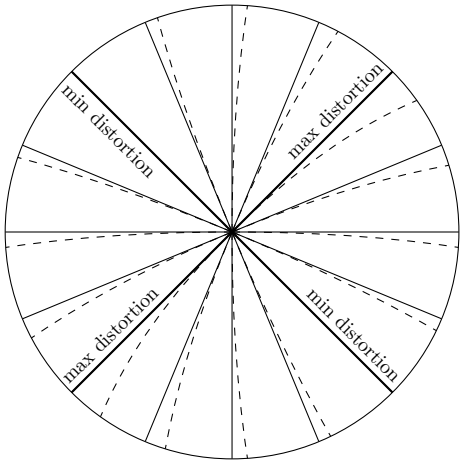
$$y_{distorted} = y \cdot (1 + k_1 \cdot r^2 + k_2 \cdot r^4 + k_3 \cdot r^6) \tag{2.6}$$

where $x$ and $y$ are the pixel positions without distortions as normalized coordinates with the origin in the optical center of the image. $k_1$, $k_2$ and $k_3$ are the radial distortion parameters for the lens which was used while recording the image, and $r^2 = x^2 + y^2$ is the square of the distance from the image center.

Tangential distortions as in Figure 2.3b appear when the sensor is not completely parallel to the lens in front of it. They bend straight lines from the optical center to the edges of the image, depending on their angle from an axis of minimal distortion, with the maximum




(a) Radial distortions




(b) Tangential distortions

Figure 2.3.: Lens distortion effects in images according to [19]. Continuous lines show the optimal lines, dashed lines are examples of the effects of different distortion classes.

on a perpendicular axis. The effect can be formulated as

$$x_{distorted} = x + [2 \cdot p_1 \cdot x \cdot y + p_2 \cdot (r^2 + 2 \cdot x^2)] \tag{2.7}$$

$$y_{distorted} = y + [p_1 \cdot (r^2 + 2 \cdot y^2) + 2 \cdot p_2 \cdot x \cdot y] \tag{2.8}$$

with $p_1$ and $p_2$ as the tangential distortion parameters. The remaining parameters are the same as for the radial distortions.

The combination of radial parameters is commonly stored as a single vector in the form of $D = [k_1, k_2, p_1, p_2, k_3]$. The original author of this model only introduced four parameters for lens correction [20]. For compatibility reasons, $k_3$ is kept separate from the other parameters. For more complicated systems, for example fisheye lenses, many approaches offer the possibility to add two more $k$ parameters which then introduce higher orders of $r$ into Equations 2.5 and 2.6.

The remaining intrinsic parameters are the x- and y-components of the focal length and the coordinates of the optical center, often called the principal point. The focal length defines the distance between the virtual pinhole in a camera and the film/sensor or image plane. With a higher focal length, distant objects appear closer, while a lower focal length creates a wider field of view. This becomes especially important for the calculation of the relative camera positions and when pixels are projected into the scene because it determines how the size of an object changes in the image depending on its distance from the camera. In most cases, the focal length is only defined by a single factor. The reason for having two factors for the focal length in the intrinsic parameters instead of one is the fact that certain lens problems or anamorphic lenses can make different focal lengths or opening angles in the horizontal and vertical direction necessary. The principle point or optical center is used to define possible deviations between the center of the lens and the center of the sensor in the camera.

The process of determining the camera parameters is commonly known as camera calibration. While one can guess how to initialize all these parameters based on the chosen

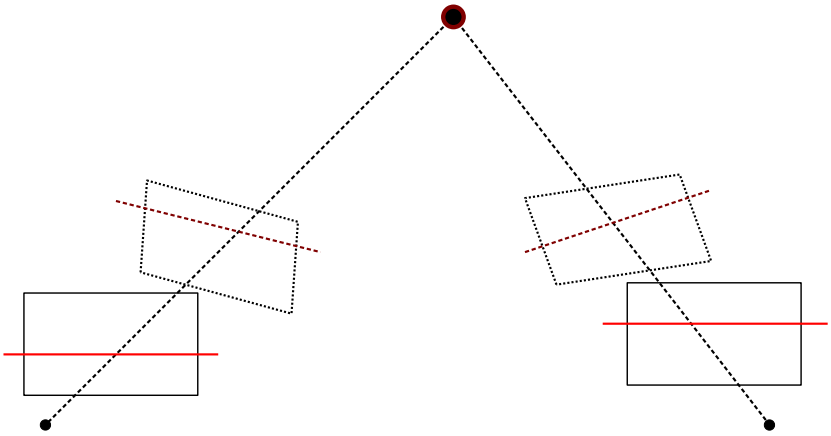


Figure 2.4.: Visualization of epipolar lines before and after rectification. In the dotted unrectified frames the epipolar lines have various angles. In the rectified frames, the epipolar lines are perfectly horizontal.

camera setup and perfect lenses, determining their precise values is more complicated. The intrinsics are usually calculated first, as they only need the input from a single camera and the calculation of the extrinsics is far easier when correct intrinsics are already known however, approaches that combine the two into a single step exist [20].

Learning-based approaches, without the need for calibration pattern in the scene, have been published in recent years. Regardless, to the best of our knowledge, none of those consider all distortions and parameters which the older pattern-based approaches can measure. They either assume no distortions at all [21] or only two radial parameters without tangential distortion or principal point offsets [22]. Apart from the choice or detection of pattern features [23], the overall process for pattern-based calibration has not changed significantly. A planar pattern with known features and precisely measured size is recorded by the camera from various angles, distances, and positions. After detecting the key points in them, their positions in the images and relative positions to each other are used to create systems of equations that can be solved for the intrinsic parameters. Due to possible outliers and insufficient precision when feature points are positioned between the pixels of an image, most approaches make use of regression algorithms or add different refinement steps like RANSAC [24] for the detected features to reach a high accuracy in the results.

Once the intrinsic parameters are known, the lens distortions can be removed by reverting their effects to make sure straight lines in the scene also appear as straight lines in the image and the optical center aligns with the center of the image. After these corrections, the footage can be treated as if it was captured by a pinhole camera. With enough features visible in at least two cameras (from patterns or features detectors), additional equations for the rotation and translation between the cameras can be created and solved. By transforming the image according to the extrinsics, the camera can be virtually rotated and shifted, such that the images seem to be shot from their intended location on the camera grid, even though the real position is slightly different.

Additionally, the data can be used to rectify the images. Rectification parallelizes the epipolar lines in the captured images as shown in Figure 2.4. Epipolar lines are virtual straight lines in images, on which a scene point travels when it changes its depth. In

(a) uncorrected (b) correction with data loss (c) correction with borders

Figure 2.5.: Lens distortion correction modes. Without black areas some parts of the original image are not visible anymore. Full data visibility requires some black areas.

non-rectified images, those lines can have any angle, depending on the camera's rotation, depicted by the dotted image borders and dark red lines. When the images are rectified, they become horizontal. This makes tracking objects with varying depth far easier, as only one dimension has to be searched. That property is very beneficial for many post-processing algorithms for multiview content or lightfields, as it decreases the complexity of searches for corresponding points between images.

The correction of images using 3D transformations and the inverse of lens distortions change one important property of the images. They are in most cases no longer rectangular, as seen in figure 2.5. If the size of the image is unimportant, it can be resized to contain the new image's shape and the empty portions of the new image are filled with a static color. Using that approach, all information contained in the uncorrected image is still present, but for every camera in a constellation, the resulting images may have a different size. In case a constant image size takes precedence or black areas are prohibited, the focal length and optical center can be adjusted in such a way that the new image only contains the parts of an image that fits the biggest rectangle, with the correct aspect ratio, into the new shape. Choosing the right balance between those two extreme points is important for every application and the optimal value is different for every use case. To minimize the amount of unused image space and discarded image data, additional constraints need to be added to the calculation, such that the required amount of rotation and translation for proper rectification is distributed over all images and kept as small as possible, instead of leaving a reference image unchanged and only modifying others.

Further details about the complexities of calibrating a large camera array and the custom solutions required for it are given in Section 2.2.4.6.

### 2.1.3. Dimensionality of Image Data

Given that we discuss multiple image and video formats in this thesis, this section is necessary to give an overview of their definitions. These days, everyone should be aware of how digital images are commonly represented, that is to say, as a uniform 2D grid of image points or pixels, with a color associated with each one. Independent of the number of color channels, they are considered to be 2D images due to their pixel structure.

A sequence of such images captured in uniform time intervals is known as video. Even though a whole new dimension is added to the 2D images in the form of time, they only form 2.5D images in the context of this work, as the third dimension can not be fully controlled and all pixels are simultaneously exposed.

Multiview, being considered the next evolution of graphical data, adds more images, called views, to every time instance. Although the version with two views can be classified

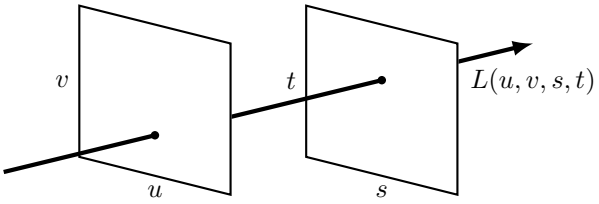



Figure 2.6.: Lightfield ray representation using two arbitrary planes proposed by Levoy [25] and Gortler [26]

as multiview footage, it is more commonly known as 3D. With such material, the images are presented separately to the left and right eye of the viewer to create the illusion of depth. This is the extent to which consumers usually come into contact with multiview content. In research, the number of views can be arbitrary and without an upper limit. In this thesis, this class of material is mostly called multiview, but the phrase '3D' may be used instead when it is limited to two views.

Contrary to the nomenclature of conventional video, lightfields are often represented as 4D data. Their basic units are not pixels, but light rays originating from the captured scene and hitting the camera's sensor. For every ray in the lightfield, the color it carries from the scene to the sensor is stored. In further processing steps, the rays and their colors are then interpreted as coarse samples of the plenoptic function [27] from the convex hull of the scene.

To fully define the ray's direction and position in space, Gortler *et al.* [26] and Levoy *et al.* [25] use intersections of the rays with two arbitrary planes as shown in Figure 2.6. The required two coordinate pairs $(u, v)$ and $(s, t)$ represent the 4 independent dimensions that make lightfield images 4D material. A sequence of lightfields captured with fixed time intervals between them is also called lightfield video.

Similar to the definition of 2D content, when all sensor pixels are exposed at the same time, lightfield video is classified as 4.5D content. Only when the exposure can be changed for certain parts of the lightfield independently from the others, it is considered to have full 5 dimensions. Ideally, the exposure time and duration would be configurable for every single pixel in the cameras, but today this is not feasible yet. For this work, it is deemed sufficient to control these parameters per camera instead of per pixel. The benefits of 5D over 4.5D lightfields are described in Section 2.6.3.

### 2.1.4. Multiview vs. Lightfields

In this thesis, topics concerning multiview and lightfield content are described. Discussions at scientific conferences have proven on multiple occasions that the distinction between those two content types is not trivial. Both can contain the footage of multiple cameras in the form of still images or video. Similarly, the image resolution, camera baseline, and the total number of views can vary.

It is clear that multiview, as well as lightfield content, exists on wide spectrums which overlap in many areas. Multiview footage can have only two views, such as on 3D BluRay disks [28], but also up to a hundred or more [29, 30] in arbitrary 1D, 2D, or 3D layouts. The multiview extension for H.264 technically supports up to 1024 [31], but in practice the usable number of views is far lower, as discussed in Section 4.4.3. HEVC/H.265 is limited to a much more realistic 64 views [32]. Every view in multiview content has a

resolution similar to that of conventional video from older 480x320 to FullHD and 4K without a particular upper limit. The distance between the views can vary from a couple of centimeters, up to nearly a meter. Depending on the camera's distance from the scene, the disparity between the cameras can be quite high, spanning several hundred pixels. This disparity makes post-processing steps like view interpolation fairly complex, as a lot of occlusions and disocclusions happen between camera pairs, which must be detected and dealt with. When these problematic areas are treated properly, the range of movement for such virtual cameras is quite large, as it spans most of the area covered by the capturing array.

Lightfields can be captured using multiple conventional cameras, plenoptic cameras, or a single camera on a precisely controlled gantry. While the first case shares view and layout characteristics with multiview content (apart from the fact that lightfields never only work with one-dimensional layouts) the latter two cases can be quite different. Most plenoptic cameras have a microlens array between the camera's main lens and the sensor. It separates the image into a high number of lenslets, each of which shows a portion of the scene from a slightly different position and angle. In the Lytro Illum, with its high-resolution sensor with 40 Megapixels, each of the approximately 200,000 lenslets only covers a single-digit number of pixels in vertical and horizontal direction. After separating the lenslets in the raw image, their pixels can be used to construct sub-aperture images that behave like the output of a camera array. The main differences are the comparably low resolution and the baseline between these virtual cameras, which is only a few millimeters at most.

Lightfield gantries move a conventional camera precisely to predefined positions and capture images from there. This technique can only capture lightfields from static scenes but offers a significantly higher resolution for each sub-aperture. The distance between the virtual cameras is highly variable but is usually chosen to be one centimeter at most.

Such a small distance between the virtual camera causes the overlap between the images to be extremely high. Combined with the high resolutions, the density of the information from all sub-aperture images is very high. This makes view interpolation much easier, because nearly no occlusion effects are visible and it even allows for advanced procedures such as refocusing [33] and super-resolution [34]. In contrast, the effects of the view interpolation are far less impressive, as it often covers only a few centimeters in all directions.

Even after this comparison, a clear line between multiview and lightfields cannot be drawn. Looking at the internal data representations, multiview treats the input images as frames from different cameras, while lightfields treat the pixels in those images as light rays from the scene. Besides that vast difference, most raw input data types can still be represented as normal images which both approaches can interpret and use in their own way.

The only distinction between them can be found when looking at the characteristics of the input material commonly used for the algorithms. Multiview content usually comes with wider camera baselines, while lightfields only use material with much smaller baselines. Since the resolution of the camera images can similarly vary in both cases, the sharpest division is found in the information or ray density of the input material. Even though both can use nearly the complete spectrum of inputs, the results of multiview algorithms lose some of their impressiveness for information-dense content and become much slower due to the fact that they put a lot of effort into the handling of occlusion effects and inpainting techniques, even though they are not as necessary for such dense content. Lightfields expect a high ray density and handle it properly, but, below a certain density, they run into the same problem as early and simple multiview algorithms, which decreases the quality of their results significantly.

Taking all of this into account, there seems to be no clear line separating input material

between multiview or lightfields. Only one in the information/ray density above which parts of multiview algorithms become ineffective/unnecessary and a second one below which lightfield algorithms lose a lot of their result quality. As those lines heavily depend on the feature set and capability of an algorithm, they are very blurry, but overall the information density seems to be the only measure separating the two main use cases for such visual content.

### 2.1.5. Network Boot / PXE

Network boot is a technique that allows a computer to boot from storage media that are not directly connected to it but can be accessed via a network. Shortly after the introduction of network protocols capable of configuring the network interface of newly started computers, like BOOTP [35], a functional predecessor of DHCP, functions were created which download boot data from a server into memory and execute it. As they were implemented in the system's BIOS, they were highly system-dependent and using them in heterogeneous environments was complicated. With the publication of the specifications of Intel's Preboot Execution Environment (PXE) [36], an intermediate layer with a unified API layer was introduced. It started as part of the network card's firmware and is today part of nearly all UEFI implementations. Even in the most recent versions, it relies on a combination of DHCP and TFTP servers to work.

The systems loaded via PXE need to be prepared for that kind of booting procedure. It is mostly a reduction in the size of the original installation medium, since the transfer via TFTP is quite slow compared to HTTP or other protocols. Fortunately, because of the popularity of network boot in data centers and environments with thin clients or diskless machines, adapted versions of all major operating systems are made available by their respective developers.

In this thesis, an open-source version of the preboot environment called iPXE is used. It is compatible with the original PXE, but also offers additional features such as more protocols for data access and scripting support. Since the original PXE resides in the UEFI firmware of all connected computers, it is always started first and configured to load and start the iPXE software via the network. The support for loading data from HTTP servers is used to decrease the transfer times of the boot media because it is much faster than TFTP. With the scripting capability, the reinstall checks and guided menus used during the boot process are implemented. More details are given in Section 2.3.6.

### 2.1.6. GStreamer

GStreamer is an open-source multimedia framework that is heavily used throughout this thesis. It is implemented in C with added support for objects via the GObject library. While the base of the framework only provides basic templates for different kinds of plugins and functions to load and chain them together, its main power comes from the vast number of freely available plugins from third parties. They include data in- and outputs from local hard drives, most network protocols, and hardware such as capture cards, cameras, or monitors. A vast number of wrappers for well-known encoder and decoder libraries can handle most existing file and container formats. For raw audio and video data, there exist format converters and basic manipulation tools for overlays, cropping, and scaling. These plugins are combined into pipelines which always start with input plugins and end in output plugins. In between, the data streams can be split up, combined, and processed by plugins an arbitrary number of times. The compatibility of consecutive plugins is specified by the capabilities assigned to the connection points.

Data transferred between the plugins only carries a minimal overhead consisting of timestamps to enable the synchronization of different data streams for encoding or playback purposes. The data type and format can only be derived from the properties negotiated between the output of the previous and the input of the current plugin. Since sometimes the exact format can only be determined once the first data packets are processed in the pipeline (for example when multimedia containers like MP4 or MKV are unpacked), the pipeline starts initially functions as it should, because the possible inputs and outputs match, but fails after the first data packet because it contains something unexpected. To handle such cases properly, the pipeline has to be created and supervised in a wrapper program capable of reacting to the current state of the pipeline and act accordingly. For such applications, bindings for a multitude of languages exist, including C++, Python, Rust, Ruby, and C#. In this thesis, the data fed into the pipelines was always tightly controlled, which meant in most cases static pipelines would suffice. When interaction with the pipeline is required, it is implemented in Python, as most of the surrounding software was also implemented in Python.

Besides the pipelines, certain steps required the creation of plugins with new functionalities. This includes dynamic multiplexers and demultiplexers for the conversion between H.264/AVC and MVC in Chapter 4, analysis plugins for cameras focus and exposure in Section 2.4.4 and outputs for shared memory to interface with external software in Section 2.2.

## 2.2. First Small Prototype Array

At first, we were looking for a camera setup that would allow us to show that our real-time H.264/MVC encoder and decoder (presented in Chapter 4) worked properly. For this use case, we devised a small camera array consisting of five cameras, each connected to a computing node for pre-coding, a more powerful computer for the final processing and video transmission, and a second computer as receiver and decoder.

### 2.2.1. Hardware

In order to be flexible with respect to the camera layout and the space the array is set up in, it was decided to mount the cameras on a modular stand made out of aluminum extrusions. We designed three versions, two made from different lengths of straight sections connected by lockable angle connectors, and one consisting of a single longer straight section for very space-constricted setups, as shown in Figure 2.7. Each of these versions can be mounted on either short or long legs, depending on whether it is to be placed on a table or directly on the ground.

The cameras can be placed freely on the aluminum extrusion using the setup shown in Figure 2.8. First, spring-loaded slot nuts with an M6 thread are inserted at the desired positions. Then a screw adapter from M6 to ⅜ inches is inserted in the slotted nut and a ball-head camera mount is screwed onto the adapter. Lastly, the camera is fastened on the camera mount.

This way of mounting the cameras allows for quick and easy changes of the camera baseline and viewing direction, especially with the camera mount[2] we chose, as it enables us to change the horizontal rotation independently from the other axes. This makes the camera arrangement very flexible and suitable for many use cases but still keeps it stable between scenes. The drawback of this flexibility is the fact that only very few constraints

[2] `https://www.novoflex.de/de/ball-serie/ball-19-p.html`

Figure 2.7.: Mounting options for the array with long and short legs and varying options for horizontal placement

Figure 2.8.: Mounting mechanism for the cameras

can be used to support the extrinsic camera calibration that is necessary for most of the advanced processing steps.

For the cameras, we chose the Logitech C920 HD Pro[3] for its capability to capture raw FullHD footage and its internal H.264 encoder. Each camera was then connected to a NUC5i5RYH[4] with a 256GB 2.5" SSD and 8GB of RAM, which was mounted in the back of the aluminum stand using the provided wall mounts. For the transfer of the captured video data, the NUCs were connected via a Gigabit switch with the sender node. This node offers two 1GBit Ethernet ports, which we used to separate the network with the camera nodes from the output or public side. On the inside, it featured an Intel Core i7 6th generation processor with 16GB of RAM for the final steps of the MVC encoding process and the other tools we required in the internal camera network to ensure that all equipment functioned as it should (see Section 2.2.2). The receiver computer featured the same hardware as the sender, the only necessary connection between sender and receiver being a standard Ethernet connection with sufficient bandwidth for the stream transfer. In the following section, we describe the software components that were required to make this real-time H.264/MVC coding demonstrator work.

### 2.2.2. Software

For the handling of all video data, we used GStreamer[5] pipelines in every device in the system, each responsible for a certain task. Figure 2.9 gives an overview of all involved steps. On the camera nodes, images coming from the camera are recorded and forwarded to the precoding part of our MVC encoder implementation presented in Chapter 4. After encoding,

[3] https://www.logitech.com/de-de/product/hd-pro-webcam-c920
[4] https://ark.intel.com/content/www/us/en/ark/products/83255/intel-nuc-kit-nuc5i5ryh.html
[5] https://gstreamer.freedesktop.org

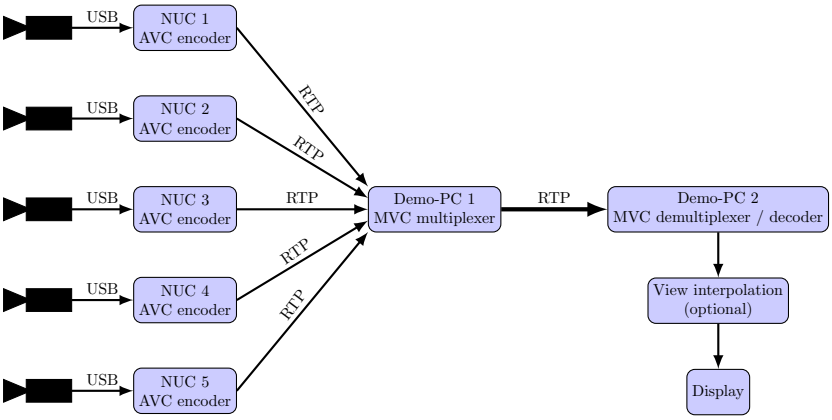



Figure 2.9.: Demonstrator pipeline overview

the resulting streams are transferred via RTP to the computer running the transcoder, where they are combined into a single H.264/MVC compliant stream. The MVC data is then transmitted to the receiver for decoding and/or further processing.

For all theses steps to work properly, the number of frames transmitted by every camera node needs to be identical. Otherwise, delays appears in the transcoder which causes the different views to go out of sync. The incoming frame rate from the camera can be configured, but its final value is ultimately determined by the internal clocks of the camera. Differences in the internal core clocks of the devices cause the frame rate of each camera to deviate slightly from the desired value and images are captured in similar, but not precisely equal time intervals. As multiview video material should be captured with perfectly synchronized cameras to fulfill the assumptions made by the H.264/MVC standard, the clock deviation in the cameras can lead to significant problems when one camera captures a frame more or less than the others. The encoding pipeline also assumes that all cameras produce the same amount of frames in a given time period so that the frame types in the precoded streams align properly. For more details see Chapter 4. The analysis of the observed deviation is discussed in Section 2.2.3. For scenes that contain movement, in addition to the frame rates of the cameras, the point in time when the cameras start capturing needs to be synchronized. If they are not, every camera starts the exposure of each frame at a slightly different point in time thereby making moving objects appear in slightly different positions in every camera, which in turn leads to problems with algorithms that try to determine the scene depth using stereo matching or plane-sweeping approaches. A hardware solution would be preferable, but the cameras we chose do not offer an input for clock or trigger synchronization, so this was not possible.

To minimize these problems, we employed two techniques. First, the camera nodes synchronize their internal clocks as precisely as possible using the PTP protocol [16]. The clock reference is placed on the sender machine, as close as possible to the nodes. Once the clocks are aligned, it is guaranteed that all frames from the cameras are tagged with the correct timestamp in the GStreamer pipeline. Furthermore, it is possible to adapt the frame rate produced by the camera using a plugin in the pipeline. By copying repeating frames when the camera is late or dropping frames when the camera is too fast, the frame rates are adjusted slightly. This way the time difference between the frames from the nodes is limited to less than a frame duration as shown in Section 2.2.3. For simple applications without fast movements in the scene, this delay is considered acceptable.

Second, to minimize the start delay between the cameras, and therefore the offset between the frames, all commands to the camera nodes were sent using clusterssh[6]. This tool offers a Linux command prompt whose inputs are forwarded to multiple machines using previously established TCP connections, as fast as possible. Using that system, multiple Linux machines can be controlled simultaneously with minimal delay. For our application, we measured a delay of less than 5ms between the execution of a command on the first and last node. This delay was deemed acceptable for this application and further efforts into the improvement of these delays were postponed until they became necessary for the next camera array. The influence of the unpredictable behavior of schedulers in non-real-time operating systems is also neglected here since their influence is usually much smaller than the jitter present on a local network.

A factor for most multiview processing steps is the knowledge of relative camera positions. There are different ways of determining the required camera matrices, but for this setup, we decided to use a pattern-based approach. Such approaches require the capture of multiple pictures of a well-known pattern by all cameras. The biggest issue is that most algorithms in libraries like OpenCV only support stereo cameras. We solved this issue by applying the algorithm to camera pairs that always contain the same reference camera. The results from that approach already required small changes to the calibration algorithms, since they usually try to minimize the unusable image area after rectification by minimizing the rotation and translation of each camera, while still getting the desired corrections. Applied to our case, this means we acquire four results for the extrinsics of the reference camera and the results for the other cameras only work with the respective result for the reference. To remedy that problem, the translation and rotation from the reference camera are added to the other camera in the pair. Therefore, the reference camera stays static and all results are compatible with each other. Even though more information is lost in the outer cameras when the images are rectified, it is acceptable when the mechanical alignment of the cameras is done carefully. Due to the flexibility of the cameras we chose, the aluminum frame itself, and the tripod mount underneath, aligning them properly is not an easy task and takes a considerable amount of time and effort to get good results. It is also important to note that the accuracy of the calibration decreased with a growing distance from the reference camera. With the center camera of the array as the reference, the results could still be good enough to be used with the view interpolation presented in Chapter 7, however, multiple calibration runs were required to achieve this.

The intrinsic parameters must be calculated separately because their result influences the calculation and results of the extrinsics. While most libraries calculate them together to save some steps and complete the camera calibration by default, they can often be performed independently. Their result is then used as an additional input parameter for the calculation of the extrinsic parameters and is not changed. This is acceptable in stereo setups, however, for setups involving more cameras, the problem described in the paragraph above occurs, resulting in multiple results for the intrinsics for one camera. Technically, this should not be the case because the properties that determine the intrinsics, such as the lens system and the exact sensor placement in the cameras, do not change and therefore the result should always be the same. Due to the number of parameters that have to be determined and the inaccuracies that occur when trying to find the reference features in the pixel grid of the images, there are often multiple sets of results that solve the given problem. For the intrinsics, the positions of the patterns in the frames also play a big role

[6] `https://github.com/duncs/clusterssh`

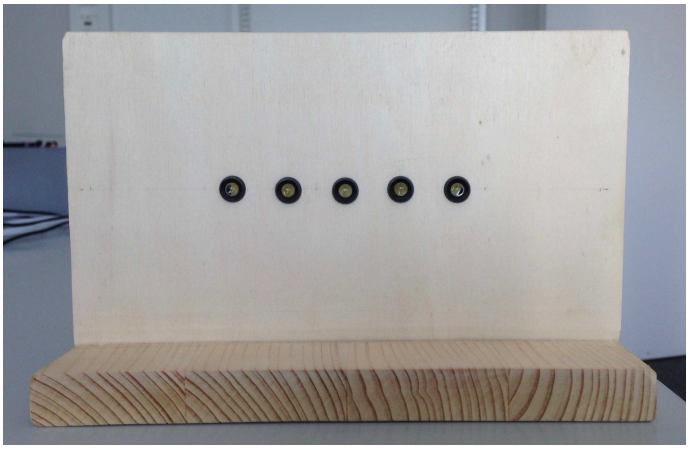


Figure 2.10.: LED testing rig

because the lens distortions are less pronounced in the center of the frame and have more influence near the borders. Therefore, the patterns should be visible in every part of the image at least once. At the same time, no area should be preferred since this causes the optimization to have a bias towards this area which usually leads to worse results overall. Especially for multi-camera setups in which the cameras do not look at the same point in space, the distribution of the patterns in the views of all cameras is hard to keep free of bias. To make the process easier, the calibration samples for the intrinsics are captured separately for each camera so that the overlap between the views can be ignored. Following these guidelines, it is possible to achieve fairly consistent results. However, the results still have to be crosschecked by hand, to avoid incorrect results caused by outliers in the feature detection or pattern coverage.

### 2.2.3. Evaluation

Since this array was mostly intended as a demonstrator for the distributed coding software and as a precursor of a bigger array, the evaluation of its capabilities was of utmost importance.

The first aspect we evaluated was the offset and drift between the different cameras. As part of Frank Waßmuth's work [37], an LED testing rig was built to measure those two important values. It consists of five LEDs embedded in a thin piece of wood, controlled by a microcontroller in the back. By displaying a known pattern on the LEDs and filming them with all cameras at once, the values we want to measure can be determined by analyzing what the cameras see. The pattern to display on these LEDs was determined by the capabilities of the cameras filming them. They have a maximum frame rate of 30 frames per second, which means there is one sample every 33ms. To detect smaller differences in the frame rate, long observations are necessary so that the errors add up and reach the observable range. The LED pattern was set to switch on an LED for two frame durations (66 ms) at the start of every second. After 33ms (or one frame), the next LED displays the same pattern, until the last LED is reached. A shorter time between the LED signal is theoretically beneficial but was not useful due to the relatively high exposure time of the cameras. That fact made the determination of the exact switch-on time much harder. Since the measurement duration was not an issue, we opted for the longer intervals.

| Sample | Rel. drift of camera $[frames/hour]$ | | | |
|---|---|---|---|---|
| | 2 | 3 | 4 | 5 |
| 1 | -0.7 | -0.4 | -0.4 | -0.5 |
| 2 | -0.7 | -0.4 | -0.4 | -0.5 |
| 3 | -0.6 | -0.4 | -0.4 | -0.5 |
| 4 | -0.5 | -0.3 | -0.3 | -0.4 |
| 5 | -0.5 | -0.3 | -0.3 | -0.3 |
| 6 | -0.6 | -0.4 | -0.4 | -0.4 |
| 7 | -0.6 | -0.4 | -0.5 | -0.6 |
| 8 | -0.6 | -0.4 | -0.4 | -0.5 |
| 9 | -0.6 | -0.4 | -0.5 | -0.6 |
| 10 | -0.5 | -0.3 | -0.4 | -0.5 |
| Average: | -0.59 | -0.37 | -0.40 | -0.48 |
| Variance: | 0.0054 | 0.0023 | 0.0044 | 0.0084 |

Table 2.1.: Relative camera drift results

| Sample | Start offset $[frames@30fps]$ | | | |
|---|---|---|---|---|
| | 2 | 3 | 4 | 5 |
| 1 | 0 | 0 | -1 | 0 |
| 2 | 0 | 0 | -1 | 0 |
| 3 | 0 | 0 | 0 | 0 |
| 4 | 0 | 1 | 0 | 1 |
| 5 | -1 | -1 | -1 | -1 |
| 6 | 0 | 0 | 0 | 0 |
| 7 | 0 | 0 | 0 | 0 |
| 8 | 0 | 0 | 0 | 0 |
| 9 | 0 | 0 | 0 | 0 |
| 10 | -1 | -1 | 0 | 0 |
| Average: | -0.2 | -0.1 | -0.3 | 0 |
| Variance: | 0.1778 | 0.3222 | 0.2333 | 0.2222 |

Table 2.2.: Relative camera startup offset results

All measurements were performed for 10 hours over night. In the morning, the recorded frames were analyzed using a MATLAB script that determined in which frames the LEDs switch from off to on and for how long they stay off and on. Starting offsets were removed before the calculation, such that they would not influence the drift measurements. Comparing the results of all cameras gave the outcome shown in Table 2.1. On average the drift between the cameras was measured at 0.48 frames per hour or one frame per 2.08 hours. For long capturing sessions, a software-based equalization of the frame rate can help to restrict the overall difference to one frame duration. As long as no fast movements are captured, this means that the system is also usable for long sessions.

The starting offset can only be determined with integer frame duration precision. To get the correct value, the position of the first completely visible pattern in the stream is compared in all the cameras. Even when the cameras switch on shortly after the pattern is complete, there is only a minuscule difference in the result, since even the highest drift we measured only changes the result by less than $0.2ms$. As we can see in Table 2.2, in most

tests the offset is less than one frame and even in the worst cases the offset never reaches two frames.

Overall, we can see that the system can be used to capture or stream short multiview sequences without fast movements, which would require a more precise frame timing. We verified this by demoing the array together with the distributed encoding system at the CeBIT 2017 at Saarland University's booth. It performed well for hours without problems on multiple days.

## 2.2.4. Considerations Learned from Small Array

Soon after the completion of the first small array, it became apparent that while the array was well suited for its intended task, it was not of much use in an extended context. Specifically the emerging lightfield research required more cameras with better frame synchronization and the possibility for a two-dimensional arrangement of the cameras. To satisfy the needs of new research in lightfield capture, the shutter release should not only be precise but also directly controllable, ideally independently for every camera. This section explains how the decision for array parameters and construction techniques were chosen.

### 2.2.4.1. Size

The first and most important decision was regarding the size of the new array, both with respect to the number of cameras and the physical dimensions or the volume that the cameras are supposed to be able to cover. While the first array was only intended to capture the torso of a person sitting at a desk, the aims of our research became more ambitious. We sought to capture a complete normal-sized human at a distance of two to four meters and to cover any two by two meter scenes at approximately the same distance. The plan also considered the difference between lightfield and multiview captures, which mostly differ in the range of camera baselines, in regard to the captured footage. The extremes are lightfield captures, for which the cameras need to have the least distance possible between them, and multiview footage, for which it can be beneficial to increase the distance between the cameras to cover a bigger area. Depending on the size of the scene, the cameras may even be positioned on an arc around the scene and be angled towards the center. To reach an acceptable camera density for the lightfield footage and to be able to have symmetric layouts with uniform distribution, the final choice was to make the number of cameras the square of an integer.

Due to budgetary and manageability considerations, it was decided to use 64 cameras. This number allows for 8x8, 16x4, or many more layouts, and even with a small distance of about 8 centimeters between the camera centers they still cover an area of 56 by 56 centimeters with an 8x8 layout.

### 2.2.4.2. Modular Design

While modularity was not a significant concern with the small array, with the much more substantial dimensions and the higher number of devices in the new array, it became very important. The main concern was that without a modular design, the whole system becomes too cumbersome to move and manage. Since the setup proved itself in the small array, every camera would be connected to a Small Form Factor (SFF)-computer which is used for both processing and caching of the recorded images. In order to be able to do this, these computers must be within the maximum length of the cable connecting them with the camera, independent of the camera layout. Large modules with many cameras can

be problematic when the cameras are spread out further because there are hard limits on the distance certain high bandwidth cables can have. The most common connection for cameras with the form factor that we were looking for is *GigE Vision*, based on Gigabit Ethernet and *USB3 Vision* based on USB 3.0. While Ethernet cables can be longer than 50 meters and still carry the required gigabit data rate, USB3 cables are more restrictive. Even though the USB3.0 standard [38] does not specify a maximum length for the cables, the maximum achievable distance using only cables is at about 5 meters. While this seems to make the Ethernet connection the obvious solution, the USB connection adds far less overhead to each module as it does not require additional switches between the cameras and the computers and no extra power connection.

The final decision was to group the cameras, the directly connected computers, and the required hardware into batches of 16. These four, more or less autarkic modules, with minimal connections to a central aggregation point, can be positioned in such a way that the cable length of either option discussed above, is sufficient for all intended camera setups.

#### 2.2.4.3. Choice of Cameras

From the small array, we have learned that camera synchronization is very important and software solutions have a limited reach. For higher precision, genlocking the cameras or a dedicated shutter release input becomes unavoidable, meaning that the camera must have one of these features. To get the smallest camera baseline, the physical size of the camera needs to be as small as possible. Fortunately, most industrial cameras with inputs for synchronization have a standardized size of 29x29 millimeters and vary only in length. Given that the inputs are located in the back of the cameras, the minimal distance between the cameras is mostly defined by the size of the camera mount used and the lens in front of the camera. When it comes to other camera features, to make proper use of the frame synchronization and shutter control from our wishlist, the sensor in the cameras needs to have a global shutter. Rolling shutters would wash out the benefits of the exact frame timing with the time difference between the top and the bottom of the frame. While having all features mentioned above, the cameras should offer a resolution high enough so the resulting ray density can compete with other existing capturing solutions. The resolution of every camera should also fall into the medium to high category according to the today's production standards. Additionally, the frame rate should be at least 30 frames per second, so medium motion can be captured properly.

Since the industrial cameras come without a lens, it was also necessary to establish the requirements for those. Starting from the size properties defined previously and given the fact that lightfield capture requires a significant portion of overlap between the cameras to be able to recover the ray direction properly, a field of view of approximately 45 degrees was determined to be optimal. Of course, the lenses must be rated for the selected capturing resolution and sensor size. Automatic focus and aperture adjustments are available but increase the cost of the lenses by a factor of at least five compared to equivalent lenses with manual focus and aperture. Given the aforementioned coverage goals and the expected sensor size between 2/3" and 1", a fixed focal length of 12.5mm was the optimal value for the lens. It results in an effective opening angle of 48 degrees in horizontal and 31 degrees in vertical direction when all sensor pixels are used. While lenses with variable focal lengths are available, they are usually bigger which increases the minimal distance between the cameras, and their higher cost is prohibitive in our given budget.

Following the considerations above, the Sony IMX249 sensor was chosen for the cameras. It features a resolution of 1920x1200 with a frame rate of 41 frames per second (even though

the manufacturer's datasheet only claims 30 frames per second). It delivers up to 12 bits of accuracy per pixel and a diagonal of 13.4mm which corresponds to a type 1/1.2 sensor. The pixel size of $5.86\mu m$ squared is quite high when compared to other sensors with similar resolutions, which makes the sensor quite efficient in darker environments or when low exposure times are used. A fitting lens with a focal length of 12.5mm, capable of covering a 1" sensor with pixels of at least $5.0\mu m$ needed to be found. After going through the lengthy ordering process for large orders of public entities in Germany, the decision fell on the combination of a FLIR BFLY-U3-23S6C-C[7] with a Kowa LM12HC lenses[8] which fulfills all requirements mentioned above.

#### 2.2.4.4. Supporting Hardware

The small array presented in Chapter 2.2 showed that the combination of a camera with a SFF-computer is a very capable and flexible design. For that reason, it was decided to retain it but adapt the computer to the new cameras. The sensor we have chosen can produce up to 176MB/s when raw sensor data is captured. This amount cannot be transferred by the usual 1GBit network connection that the SFF-computers have, at least not in real-time. While there are USB3.0 Ethernet adapters with 2.5Gbit/s available, using these would add an additional set of connectors that cannot be fixed with screws into the system and would therefore add a new layer of potential problems whenever the system is moved. Hence, we opted for a bigger cache in the computers, in the form of a bigger SSD, which is fast enough to be used as intermediate storage of the captured material. The newer SFF-computers can use NVMe M.2 SSDs which usually offer write speeds that are 5 to 10 times higher than the expected data rate from the cameras. A size of about 256GB was quite cost-efficient and with the data rate of 176MB/s, it offers enough room to cache about 20 minutes of footage with some room to spare for the operating system and the required tools. Since the array is not supposed to capture feature-length movies in a single go, this was deemed to be sufficient.

The amount of RAM in the computers of the small array was sufficient for all purposes. Therefore, the new units were also provided with 8GB of RAM. Since RAM prices were quite high when the parts were ordered, this was also a financial decision. For the rare cases when more RAM is needed, the systems can use the SSD for swapping temporarily. Due to the speed of the SSDs, this is possible without too much loss of performance.

For the CPUs, it is hard to gauge what will be required exactly, since the processing algorithms and other tasks can change in the future. Only the critical part of the capturing process was therefore taken into account when an appropriate CPU was chosen. During an active capture, the most critical part is that the frames from the cameras are stored on the SSD as fast as possible, or at least before the next frame is taken. Since the data is stored in PGM format, which is mostly raw data, not much computation power is needed for conversion, so one core is sufficient for that task. The system is still required to be responsive enough so that the capture session can be configured or stopped and the current state of the computer and the attached camera can be monitored. Ideally, the system should also be able to perform a fast and simple debayering on the captured frames, encode them as a video stream, and send this to a central controller, so a preview from all cameras is possible and quick visual checks of the camera performance and alignment can be performed. Depending on the video codec used for the stream and the hardware encoding capabilities of the CPU, this may need up to two cores when the full resolution of the camera is used. The final choice fell on a quad-core CPU with hardware support for H.264 encoding. More cores

[7] `https://www.flir.com/products/blackfly-usb3/?model=BFLY-U3-23S6C-C`
[8] `https://www.kowa-lenses.com/en/lm12hc-5mp-industrial-lens-c-mount`

and therefore more processing power would be preferred, but since the essential functionality is covered, the additional costs can not be justified with the potential gain in the future, especially since all post-processing tasks do not have to happen in real-time.

The network responsible for distributing the data between all components of the camera array is also a very important factor. Connecting all computers with full speed to a central storage or control system would be possible, but not necessary as it is not possible to transmit the complete camera footage in real-time in any case, as previously mentioned. Further processing steps that require data transfers to and from the persistent storage, do not take the same time for every file, and so the transfers do not happen concurrently, and not all computers need the bandwidth at the same time. Constructing the central storage in such a way that the data can not only be received at full speed from all cameras but also be stored with the same speed would be very costly and not feasible as long as the effects of the bottleneck are not too serious. For each module with 16 cameras, a switch with 24 Gigabit ports and two 10GBit uplink ports were chosen. At first, the uplink ports were not intended to be used, and five aggregated Gigabit Ethernet lines were planned as the connection to the central switch. This way, the bandwidth of the uplinks of all camera modules combined would be equal to the bandwidth of the central server at 20GBit/s. When all camera units transfer data at the same time, they should be able to use approximately a third of their maximum network bandwidth.

One important goal for the camera array was the precise control of the shutter timings for each camera, ideally with individual control for each camera. Choosing the right camera which offers an input for the appropriate signal is only a part of the solution as the signal for the cameras needs to come from a reliable source. Generating a master signal for the frame triggers and transferring it to the individual cameras is rather trivial since even a length difference in the communication lines of one meter should only result in a delay of about $5ns$ assuming a propagation speed of electrical signals in copper cables of about $200,000km/s$. This delay is still far from the time between the different frames which we set in steps of ten microseconds each. Therefore, this possible difference is not noticeable in the end result and we do not need to compensate for different cable lengths.

Since we want to be able to trigger sets of cameras at different time instances but still with the same frequency as the master signal, there needs to be a way to delay the signal by a configurable amount. Ideally, this delay should be configurable via software remotely. Dedicated hardware ICs for delaying signals exist, but most of them are either configured via hardware (changing the resistance or capacitor value between two pins) or only offer a maximum delay in the range of microseconds. While programmable capacitors and resistors exist, using them in addition to the delay ICs means a large footprint for the required electronics in every module that hold all devices required for 16 cameras. Finding the right combination between delay ICs and programmable components can be difficult, especially with respect to the step size with which the delay can be configured and possible repairs in the future. In addition, micro-controllers are still needed to program the delay ICs or the connected components. This led to the evaluation of software-based delays with MCUs with integrated hardware timers. Those integrated peripherals can provide delays with very high precision as long as they are used appropriately. Preliminary tests have shown that a precision of less than one microsecond can be achieved with very common and cheap MCUs. With these results, it was decided to forego the all-hardware solution in the favor of a MCU-based one. Using a microcontroller with enough GPIO pins, it would be possible to use only one controller per module, which means the overall footprint on the PCB can be fairly compact.

#### 2.2.4.5. Software

In the small array, we used software tools based on SSH to distribute the commands to the camera nodes. While this was good enough for five units, it does not scale well for more, since it opens a window for every connection. They would clutter up a desktop of any size and as they are the only way of getting feedback from the executed commands, it would be very inconvenient to check 64 windows to find out whether a command was successful or not. Systems for controlling multiple computers from a central point like SaltStack[9] or Ansible[10] usually do not have a built-in status overview for the managed machines. Such tasks have to be performed by an additional monitoring system.

For our array, we need fine-grained control over the commands being executed and real-time status information from the nodes. Simultaneous transmissions to all nodes, so they execute the commands at the exact same time, are optional, because it is only important for the control of the start of a capturing session. Since the shutter will be controlled by an external system independent from the nodes, it is sufficient that they are ready when the first frame is triggered. Even with a delay of $15ms$ between every message, which is fairly long in networking and computing terms, the 64 commands for all nodes are sent in less than a second. We decided to create a custom control system using an asynchronous message bus to deliver the messages. That way the processing time of a command is irrelevant, and commands can be emitted as quickly as possible. Depending on the exact message bus used, they might even be broadcast and therefore, be sent at the exact same time. The system should react to user interaction by sending the appropriate commands and also maintain a representation of the internal state of the nodes. Ideally, the commands and the information queries should be accessible via a REST-API, so the whole system can be controlled and monitored via a web interface.

How the system state is managed has to be changed for the large array as well. Maintaining a consistent software state over all the units manually is not feasible for the number of computers we are planning to deploy in the system. This is something also commonly found in data centers where a lot of systems require a similar software state to be able to provide large-scale services [39]. Due to this need, multiple systems to solve this problem already exist, and as they are deployed in very heterogeneous environments, they are designed to be quite flexible. Two open-source systems of this kind were already mentioned before, namely SaltStack and Ansible. Both offer the possibility to manage installed system packages, running and stopping services, and distributing necessary configuration files from a central control point.

#### 2.2.4.6. Calibration

In the small array, it was already apparent that the flexibility of the camera placement in combination with the required precision for the camera calibration poses a big challenge for the calibration algorithms and processes. In the beginning, it was planned to have as much flexibility as possible with respect to the location and orientation of the cameras in the frame of the array (i.e., five degrees of freedom). The frame which holds the cameras should keep them on a plane that fixes the z-coordinate, but the x- and y-directions have to be free to allow for layouts with different camera densities and distributions. The mounting heads between the cameras and the frame offer about 180 degrees of rotation around all three axes without steps. Because OpenCV does not allow to only fix a single dimension (only all or none can be fixed), we can only give a hint to the algorithm that the z-direction should be

[9] `https://www.saltstack.com`
[10] `https://www.ansible.com`

the same for all cameras, but in the end, it will still be estimated and the fixed dimension can only serve as a sanity check for the end result. For the intrinsics, the focal length and the principal point must be determined, both with x- and y-components. The lens distortion is modeled with up to six factors per camera. In total, 6 extrinsic and up to 10 intrinsic parameters have to be calculated from the images of the calibration patterns. For 64 cameras, 1024 parameters need to be correct, in order to make proper use of the results of a capture session. Since all negative effects of propagating the results between cameras in the array that were discussed in Section 2.2.2 only increase with more cameras, the first aspect that was investigated is the precision of the point detection from the calibration patterns. For all tests, we used generated patterns with high resolution or from vector graphics and had them professionally printed in sizes up to the A0 format, depending on the requirements of each pattern and the test parameters. To minimize deviations from the assumptions that all points in the calibration patterns are on a perfect plane, the printed versions were glued to thick foam boards or similar stiff planar materials before use, as shown in Figure 2.11.

The first approach to improving the calibration results was to improve the accuracy of the point detection in the calibration pattern, whose features were used for the input to the algorithm. Using an asymmetric circular pattern instead of a checkerboard lead to slightly better consistency and precision in the feature detection, but the overall performance was not improved much. Since the minor improvement hinted at a better sub-pixel accuracy for the detection of circles compared to lines and corners, we investigated another circular pattern [40] which includes smaller white inner circles in the center of the black circles like in the OpenCV pattern. With our camera, we achieved another small improvement, about as big as mentioned in the paper, but it was not enough to justify the added complexity of running the reference code together with the otherwise fully integrated OpenCV pipeline. Testing patterns of different sizes (see Figure 2.11) with the idea that bigger patterns contain more information about distortion effects in a single image and the bigger features should be easier to detect with sub-pixel accuracy. On the other hand, smaller patterns are easier to position such that they are completely visible in multiple cameras and with less area bias towards the center of a camera view.

The biggest problem of the larger patterns, or patterns of any size close to a camera, is the fact that all features or the complete patterns need to be visible for the detection to become valid, occasionally even with a clean white border around it. One solution to that problem is a pattern that drops the requirement to detect the full pattern to be valid. OpenCV offers patterns that consist of only ArUco markers [41] or ArUco markers in the white spaces of a normal checkerboard pattern. With these cleverly generated and distributed markers, it is possible to deduce the visible portion of the pattern and its orientation in space. Therefore, the visible portions of partially occluded patterns can be included in the calibration process for more accuracy. Unfortunately, neither the OpenCV-based nor a second version in MATLAB using AprilTags [42] did improve the calibration results. The quality of the end result still varied greatly without a discernible pattern and no clear indication of which property of the input material caused the variations.

In an attempt to combine the qualities of partial pattern detection and to extract the most information out of the available camera resolution, a fractal pattern with multiple levels of smaller patterns included in the larger pattern [43] was tested. Figure 2.12 shows a good example of the pattern. The further we zoom into the pattern, the more quadratic features appear, which can be used to refine the results. The first level also includes some additional data that can be used to identify the position of each square in the complete pattern. Even though the results in the paper looked promising, the camera we decided to use, did not offer enough resolution to detect anything but the highest level of features

Figure 2.11.: Some of the calibration patterns used for evaluation

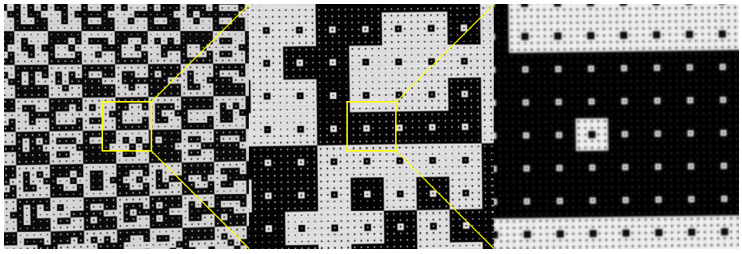


Figure 2.12.: Example of the fractal calibration pattern presented in [43]

when an A0-sized pattern was more than a few centimeters away from the camera plane. At such a small distance from the array, the overlap between the cones of vision of neighboring cameras is small, so there is only a small number of usable features for the calibration. If the pattern is moved further away, the benefit of having the fractal pattern is lost as finer features can no longer be detected reliably.

A second issue for the calibration was the task of using stereo calibration algorithms to calibrate a multi-camera array with a two-dimensional layout. The result of the calibration is always a set of parameters, which causes the epipolar lines in both images to be at least parallel, if not collinear when used for rectification of the images. While this is a desirable property in our rectified images for easier processing, the final result heavily depends on the assumptions an algorithm makes about the spatial relation of the input images. Several available algorithms assume that a stereo pair of cameras is always in a left-right configuration. This forces epipolar lines to be horizontal in the end results independent of whether this makes sense or not. In our case, this type of algorithm is not practical to use. As seen in Figure 2.13, in the case of a top-bottom camera pair, the algorithm introduces a 90-degree rotation in the extrinsics so that the epipolar lines

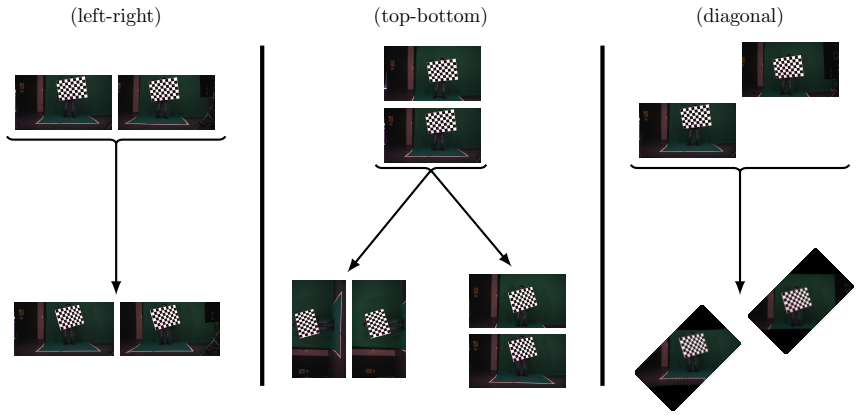



Figure 2.13.: Different behaviors of stereo calibration algorithms depending on the relative camera position

become horizontal instead of vertical. This rotation can be easily fixed, by adding an inverse rotation to the estimated rotation matrix. The principal point is harder to correct because the algorithms mainly focus on one dimension to align the epipolar lines. Calculating the exact correction is fairly complex without further calculations based on the input images.

Some algorithms do not need those fixes, as they can handle both left-right and top-bottom pairs by automatically determining whether the epipolar lines should be horizontal or vertical. Providing them with a hint about the correct alignment is only possible via preset values of the intrinsics. A separate parameter is not available in any of the algorithms we tested. Results for diagonal neighbors are almost always rotated because in those cases neither horizontal nor vertical epipolar lines can be determined properly, as they would be outside the respective other image.

To circumvent those diagonal calculations, we tested different variations of calibrating a horizontal and a vertical line of cameras centered around the chosen reference. From there, those newly calibrated cameras serve as secondary references for the other cameras, so all calibrations only happen within a row or column of cameras. The results are then related to the original reference by combining the extrinsics from the cameras calibrated against the secondary reference with the extrinsics from the secondary reference. After that calculation, every camera has extrinsic parameters relative to the center camera. Since every camera can be related to a calibrated camera in the same row and another in the same column, we can obtain two results for each camera that should be identical. Additionally, we know each camera's supposed position in the camera grid from which we can infer at least some of the expected values in the results. With this knowledge, outliers in the results can be easily identified, the respective camera recalibrated against another reference, or the whole calibration process repeated.

While checking the performed calibration tests, it became evident that the most common error we encountered was a deviation of the camera's position in the z-direction (away from the plane on which all cameras sit) while the x- and y-directions were nearly perfect. From the published results of colleagues from earlier projects, we knew that the precision achieved could already be favorably compared to other calibration efforts of multi-camera setups, even ones that use additional data in the form of captured depth data for a more accurate pattern detection and location [44]. The array used by these colleagues was a

one-dimensional layout of cameras that could be moved vertically using industrial-grade linear actuators with sub-millimeter precision for capturing static scenes as lightfields. They report a deviation from their camera mount (a straight metal rail without articulation) in z-direction of up to six centimeters in both directions with an average deviation from the line of over three centimeters. Our results showed deviations in the same direction with slightly lower but similarly formed amplitudes.

Zaharescu *et al.* [45] report similar errors in their publication with one-dimensional or arced camera arrangements and a one-dimensional calibration target with multiple reflective dots. The visible errors in their visualization of the calibration results are similar to those found in our results. They managed to improve their accuracy using an approach with iterative refinement [46], which comes very close to the ground truth. Unfortunately, it was not possible to accurately recreate the results of any type of approach which used reflective or other light-based markers [47] in the setup. The results we did manage to achieve were always significantly worse than the results of the other papers and the quality was far from that achieved in previous tests.

Our evaluation has shown that the performance of the calibration is on par with other approaches capable of dealing with the challenges of calibrating more than a stereo pair of cameras in a two-dimensional layout. There is no default solution that works in every case with the same precision, but well-performing approaches are often tailored to the array and its environment. A recent thesis [48] from Tampere University, a university with a strong background in view synthesis and holographic imagery, comes to a similar conclusion. With careful manual checks of the calibration results and good discipline during the capture of the calibration image, a good result looks similar to the visualization in Figure 2.14 which shows the result of one of our later calibration runs. While there are still some small deviations from the uniform grid, especially on the right side, most cameras are fairly close to their intended position and the deviation in z-direction is small compared to the distance between neighboring cameras. The final decision concerning the calibration was to keep the last iteration of the pattern-based approach in OpenCV, at least for the first few capturing sessions. Only then can we properly evaluate whether the current level of precision is sufficient for the algorithms the captured images are going to be used with.

## 2.3. Design Challenges

When implementing the planned array, we established an additional set of properties further to those defined in Section 2.2.4:

#### Mobility

The entire array must fit through normal doors, as well as elevators, with only minimal disassembly, and be mobile enough to capture scenes in any location with sufficient space and power.

#### Setup and Teardown

Time taken for setup and teardown should be minimal, with the number of removed parts being kept to a minimum. Components installed within other components should be sufficiently stable so as not to require extra transport security.

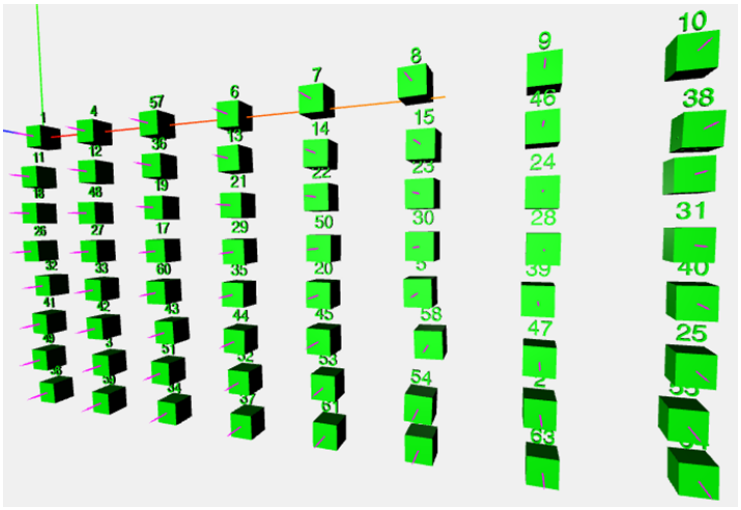


Figure 2.14.: Typical result of our OpenCV-based camera calibration approach. The actual structure is presented well in most cases, but on the right side the calculated position deviates from the intended one.

#### Independence from external infrastructure

For sufficient flexibility in the choice of capture locations, the complete system should only require minimal support infrastructure, in addition to sufficient power and stable ground.

#### Web-based control

All functions and components should be controllable through a simple web interface or a direct control mechanism somewhere in the system. Those interfaces should also provide information about the current state of the system.

The influence of these guidelines can be seen in all components of the array. If only one component violates them, setup, capture, and maintenance become more cumbersome than necessary.

### 2.3.1. Module Design

The modules are the boxes that contain the computers, which the cameras are connected to, as well as all supporting hardware. The devices must be mounted securely enough within the boxes to endure transport and continuous operation, while maintaining sufficient space between them for suitable cable routing and ventilation.

Using the known sizes of each component, first sketches and then a wooden prototype of a mounting plate for the devices in a module was created as part of the work by Johannes Reuter [49]. The overall placement of the devices stayed the same nearly from the beginning. In Figure 2.15, the components in each module and the different versions of the mounting plate are shown.

The NUCs, as the components which consume the most power and produce the most heat, are situated on the top left of the mounting plate. In that location, their hot exhaust


(a) First sketch (b) Wooden prototype (c) Final CAD model

Figure 2.15.: Design process of the mounting plates in the camera modules. After a paper and cardboard mockup a wooden prototype was evaluated before the final design was created and machined.

air does not interfere with the other electronics in the module. A second reason is that this design minimizes the smallest average distance from the computers to the planned cable outlet hole in the side panel of the module. This arrangement allows greater flexibility when setting up the array, since the length of the connected cables that can be used outside of the module is maximized. The 16 units are arranged in a 4x4 slightly rectangular pattern and are clipped onto their included wall-mounting brackets for easy maintenance. Their vertical distance is smaller than the average horizontal distance because the NUCs do not have any connections or indicators on the sides, only openings to draw in cool air. Therefore, we planned only enough space between them so that the airflow is not restricted and that one unit can be lifted from its wall-mounted adapter without having to remove the devices above it. To manage the exhaust air and all cables, all of which come out of the back of the NUCs, two vertical corridors between the first and second, as well as the third and fourth columns, are created. The holes in those corridors are used to lead the wires to the underside of the board, where they are collected and routed to their destinations. Each hole is responsible for all cables from two NUCs. Their size and location are chosen in such a way that they can bend smoothly downwards towards the opposite round edge of the hole. Airflow in that area is supported by fans on the top and the bottom of the boxes surrounding and holding the mounting plates, which draw in cool air from below the module and expel it through the top.

In the bottom left of the mounting plate, a 19” rack-mountable Ethernet switch is installed. Due to a lack of depth for a normal horizontal installation, we rotated its mounting brackets by 90 degrees so that the switch’s main body is parallel to the mounting plate, with the Ethernet ports pointing upwards. In the final design, a slot above the switches’ Ethernet ports allows the cables to run to the back of the mounting plate where they are routed to the NUCs in the top left. Our only requirements for the switch were to provide enough Gigabit ports for the NUCs of one module, some additional ports for the uplink to the central server, and enough management capabilities to combine multiple ports into a single trunk to increase the speed of the uplink. Using the Link Aggregation Control Protocol (LACP), defined in the 802.3ad and 802.1AX IEEE standards [50, 51], to configure the aggregated links, means all links behave as if they had the same MAC address and the number of bonded links can vary. In the case of a broken cable, the total bandwidth is

simply reduced until it can be replaced without packet loss. The decision of which packet goes to which link is based on a configurable set of parameters, but by default, it uses source and destination MAC or IP addresses [52, 53]. Therefore, a 1:1 connection cannot fully use the total link speed, but since we are planning to communicate between many computers and a single server this is not a problem. Single connections are also not affected because the bottleneck in those cases is the connection speed of the NUCs which is equal to the throughput of the single links at most. The first iteration of the plan assumed five ports for the uplink, such that the combined uplink speed of all four modules is equal to the Ethernet speed of the central server.

Luckily, this can be fulfilled by nearly any enterprise-grade switch with at least 24 ports. After comparing the capabilities and prices of available switches, we ordered a set of five 'HP Aruba 2530 24G 2SFP+ (J9856A)'. In the modules, they have enough ports to collect the connections from the NUCs and the uplink. The last switch is mounted next to the central server to collect the connections from all uplinks and relay the data via its two SFP+ ports to the server for storage and processing. This way, the complete network runs on uniform hardware, which makes the configuration more convenient as components from different manufacturers still have differences in their configurations and exact capabilities despite the many existing standards around Ethernet networks.

On the right side of the board, a microcontroller board and a big power supply unit are located. The microcontroller has two major functions which will be described in detail in Section 2.3.2. For this section, it is sufficient to know that it should be close to the power supply and the cable outlet for the cameras. While the cables between the cameras and the microcontroller are connected directly to the PCB with the controller on it, for the power lines, some external hardware needs to be controlled. The slot above the controller board leads the cables from the board towards the same outlet the USB cables from the NUCs go to. Below the controller, the connections from the power supply go through the slot and come back up between the NUCs.

The power supply posed its own challenges. Powering the NUCs with their included power supplies and a power strip for distribution, as it was done for the small array, was not an option due to the number of required power outlets per module. Instead, we looked into an option that allows us to power all NUCs with a single power supply. We based the total required power on the power bricks that came with the devices. Each of them is rated for 65 Watts at 19 Volts [54]. This means for every module we require a total power of $16 \cdot 65W = 1040W$. Industrial power supplies with such specifications exist, but they are often quite expensive, come in uncommon form factors, and require special connectors which are hard to get. In addition, the controller board would also need additional power regulation hardware since modern microcontrollers and most ICs cannot handle such a high voltage.

Fortunately, the NUCs do not need 19 Volts but can run on any voltage between 12 and 19V according to their data sheet [55]. This flexibility opens up several new possibilities, especially considering that 12V is most commonly used to run power-hungry devices in desktop computers, workstations, and servers. Since they have to comply with consumer electronics regulations, they are highly efficient, quiet, safe, and have well-known sizes. However, they still are available with output powers well above 1000 Watts, which nowadays usually refers to the power of the 12V rails alone. In addition to the power we require from the 12V rail, power supplies following the ATX standard [56] also offer 5V and 3.3V outputs which are also needed for our microcontroller circuit. The ATX standard also regulates the pinout and the form of the connectors, which are used to deliver the different voltages to the devices inside a

computer. With the rise of modular power supplies, which allow disconnecting unused cables from the power supply itself, manufacturers started to use many different connectors and pinouts in between the power supply and the regulated connectors. As we are technically misusing at least parts of the power supply, it does not always make sense to use the included cables when we connect to a connector that is not considered in the ATX standard, like the barrel plug in the NUCs. It was finally decided that a KOLINK Continuum 1200W would be the most appropriate solution. It was cheaper than many other power supplies in the same power range at the time and the manufacturer chose uniform connectors in the back of the unit for the connections to the single 12V rail. These eight outlets happened to be the same as standardized 12V EPS sockets. The wide availability of these connectors made it possible to make custom cables for the NUCs and use each socket to power two NUCs for optimal load distribution over the connectors. Mounting the PSU on the plate required the design and manufacture of a bracket/holder so that it could be installed vertically.

Once the components and their layout were fixed, the overall size of the mounting plate came out to be very close to 1x1 meters. With this size and the known weight of all the components, a suitable material had to be found. Construction wood, as was used for the prototype would not have been able to hold the small machine screws with threads and would have required nuts and washers on the opposite side. Since the current plan involved having the components on one side and the cables mostly running along the other with fixed guides for cable management, nuts and washers would have interfered with the components on the opposite side of the plate. Another material we evaluated was aluminum DiBond plates, a composite of a thicker plastic sheet sandwiched between two thin aluminum plates. We verified it can hold screws in cut threads reasonably well and in our tests, it provided enough stability for our application. After finding a local machine shop able to manufacture the plates, including all the required threads, we were advised that the DiBond plates would be too flexible at the size we needed, even when supported on all sides. Thicker and sturdier versions were available but would have increased the weight of each module significantly. Following this recommendation, we chose to use 3mm thick aluminum instead. To protect the cables from the hard edges on the planned through-holes and slots, they are covered by custom-made flexible inserts.

To ensure that the mounting plates fell within the previously defined design requirements, a case was designed to protect the plates and its components. For their high flexibility and our previous experience from the small array, aluminum extrusions were chosen. The base design is a box with one meter in width and height and 28cm in depth surrounding the mounting plate tightly. Behind the plate, 8cm remain for cable routing and management, thus leaving nearly 20cm of space in front of the plate for all devices and the cooling airflow. Due to the high mounting position of the NUCs, this box would have been quite top-heavy, which is the main reason why we combined two mirrored modules into a single box. The increased depth made the whole construction much more stable. For maintenance and debugging purposes, the backside is connected by hinges and removable connectors in the front. That way, it can be opened up in the front comfortably while the strain on the cable coming out of the outlets in the back is kept to a minimum.

To cover and protect the contents inside the frame, the outside is covered by customized white DiBond plates whose designs are shown in Figure 2.16. The top and bottom plate feature holes to mount the inlet and outlet fans. The back is completely closed, while the large side panels have a slot through which the cables going to the cameras leave the case. The front panel is the most complex, as it features the socket for mains power input, a panel for the network uplink, and a connector for the communication between the microcontrollers.

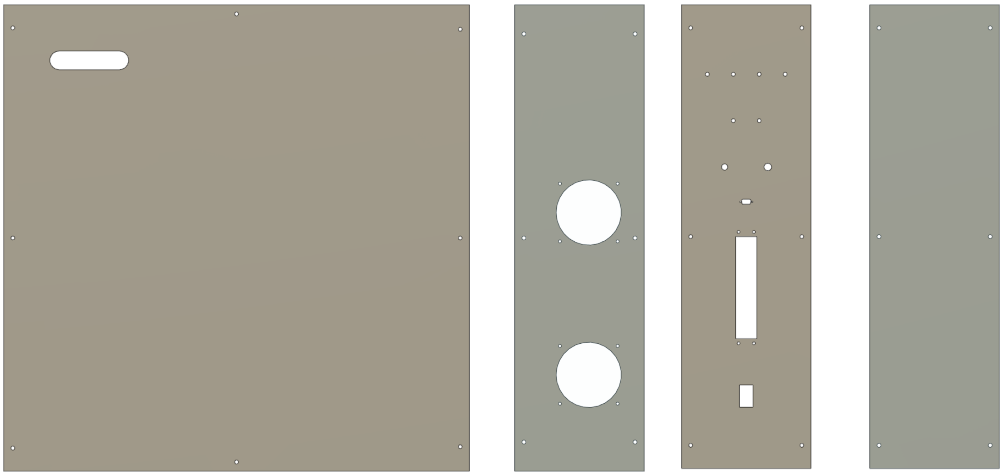


Figure 2.16.: Cover panels for the module cases (side, top/bottom, front, back)

In addition to these connectors, it has multiple LEDs to show the current status of the module, a mechanical switch to control the ATX power supply, and a push-button for fan testing.

With this construction, one of these boxes containing two modules becomes too heavy for easy handling. One mounting plate alone weighs around 8 kilograms, every NUC weighs about 0.6kg, the power supply, and the switch both weigh about 2.5kg. The aluminum profiles contribute 1.5kg per meter [57] and the outside panels about 3.8kg per square meter [58]. So even without any of the steel screws, frame connectors, and cables that are other non-removable components of the boxes, the total weight of each box already sums up to

$$23.68m \cdot 1.5\frac{kg}{m} + 4m^2 \cdot 3.8\frac{kg}{m^2} + 16 \cdot 0.6kg + 4 \cdot 2.5kg + 2 \cdot 8kg \tag{2.9}$$

$$= 35.52kg + 15.2kg + 9.6kg + 10kg + 16kg \tag{2.10}$$

$$= 86.32kg \tag{2.11}$$

To keep the whole system movable, legs with a length of 40cm are added underneath each side of the box. With two heavy-duty wheels on the front legs and a single wheel on each leg in the back, the whole box can be moved by a single person if necessary. The total width of the box of 80cm guarantees that the box with two modules can fit through the majority of normal doors. The second wheel per leg on the front was added because, when the box is opened up in front, the whole box tended to bend with only one contact point with the ground outside the main box's structure.

### 2.3.2. Electronics

The electronics boards in the top right of the mounting plates have two main functions, controlling the shutter signals to the cameras and the power to the NUCs. A secondary function is to monitor the temperature inside the module case and control the built-in fans accordingly.

First, we defined the required inputs and outputs the board needs to have to fulfill those tasks. To supply power to the board's electronics, we use the 24 pin ATX connector from

the power supply in the module. Not only does it provide every important voltage for the circuit, namely 3.3, 5, and 12 Volts, but it also has pins that can switch the power supply on and off as well as provide information about its current state.

The shutter release of the cameras can either be triggered by software via the available API, using an internal oscillator or via an external signal through an optoisolated input pin in the GPIO connector in the back, according to the technical reference [59]. Because it has the highest precision and the least steps that need to be controlled in order to achieve it, the external control via GPIO was chosen. Even though only two pins from the GPIO are needed to make this function work, the whole connector is populated in case the other pins are still needed later. Therefore, a total of six pins have to be connected to the electronics board for every camera. The search for compact connector blocks with at least six pins per port and 8 or 16 ports in one block lead to modular blocks with sockets for RJ45 connectors as found in switches or other network equipment. With the rectangular shape of the connector, the blocks are fairly compact and the cables are still easy to connect and disconnect. Further benefits are the optional LEDs next to each port that can be freely used and the widespread use of these connectors, which means we can simply buy network cables in appropriate lengths with enough shielding, replace the connector on one side with a plug that fits the GPIO port of the camera, and have a proper connection between board and camera.

#### 2.3.2.1. Power Control

Controlling the power to the NUCs requires some careful planning because the external power supply can provide up to 100A on the 12V rail to which all NUCs are connected. The main reason for having the functionality to switch the power on and off through the microcontroller is to be able to switch off or power cycle a NUC without needing physical contact when the module boxes are closed shut. Switching the power supply off while using the standby power for the microcontroller would have been an option, but would always have affected all NUCs in the module. Since this function is mostly intended for when a single NUC behaves in an unintended way, and with the NUCs being normal computers in which sudden power losses can lead to filesystem corruptions, this idea was dismissed. Instead, we decided to switch the power using MOSFETs with a very low internal resistance and integrate them into each power cable, so only a small wire for switching the MOSFET needs to be connected to the control board. The MOSFETs could also have been placed on the control board but designing connectors and copper paths that can handle 12V at 5A maximum is a challenge on its own. Because the NUCs have a lot of different ground connections in addition to the one in the power cable, such as the shielding in the network cable or the metal case on a big metal plate with connections to similarly grounded devices, it is important to switch the 12V wire in the power cable and not the ground cable. Otherwise, the power can flow back to the power supply through any other device on the mounting plate and the computer keeps running even though the MOSFET is switched off. For our circuit in Figure 2.17 we chose a p-channel MOSFET soldered into the positive power wire. A pull-up resistor between gate and source keeps the MOSFET switched off by default. Using an n-channel MOSFET on the control board, the gate can be pulled to ground to switch the cable MOSFET on and activate the power to a single NUC. The additional n-channel MOSFET is required because the control chip can not handle the 12V on its input pins.

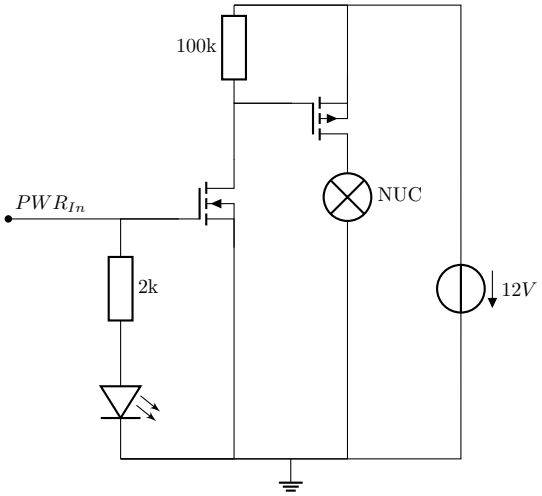



Figure 2.17.: Power switching circuit for a single NUC. The $PWR_{In}$ pin is actively pulled up and down by the GPIO controller. GND is shared between all devices in the module

The wires leading from the MOSFET gates to the control board are gathered using a simple flat cable that can be connected to a 2x8 pin connector. As this is a fairly compact connector with a trivial through-hole footprint, it can be placed nearly everywhere on the board.

#### 2.3.2.2. Fan Control

Connectors were required for the fans and temperature probes. To make the cooling of the modules affordable and quiet, we decided to use PWM-controlled PC case fans. They only need a standardized Molex KK 254 series connector and run on 12V. Although each case has four fans, only two connectors are required on the board, as the chip we planned to use is able to control two fans independently. Since the two fans on the top and bottom of the case are relatively close together, compared to their respective distance from the closest fan to the control board, we connect them using a Y-cable and then only have one cable from each pair going back to the board. While this means only the feedback from one of the fans in the pair can be measured, this is a very common way of wiring this kind of fan in computers and therefore acceptable.

The temperature probe inside the case is a simple NPN-transistor whose state the fan controller uses to determine the temperature. Again a simple 3-pin connector suffices and needs only to be close to the controller chip. For LEDs as well as the switches and buttons on the front of the case, a 2x8 pin header with a proper socket is sufficient for a good connection that can be quickly connected and disconnected.

#### 2.3.2.3. Shutter Control

The last important set of pins connecting to something off the control board are the ones that are used for connecting the control board itself to a central device for monitoring and control. One option would have been to use the integrated Ethernet port on the development board of the chosen microcontroller and connect it to the switch with the NUCs. However,

due to the hardware resources required for establishing and maintaining TCP connections and the inherently unreliable timings on the network, we continued to search for a different option.

For the master signal of the camera shutters, a dedicated line from the master to each module is the fastest and easiest solution. Most other ICs on the board are configured via the I2C protocol. Since the maximum distance between two devices on the bus is determined by the capacitance of the communication lines between them [60], in crowded environments the maximum achievable length is often limited to a couple of centimeters. But with the help of I2C buffers on both ends of a transmission line, the range can be reliably extended to multiple meters with high baud rates [61]. With such buffers, the devices on all control boards can be controlled directly from a singular I2C master in a central unit. The microcontroller can be programmed in such a way that it acts like an I2C slave device with much lower memory requirements than a full network stack. For this approach, only two pins are needed for the I2C bus, one pin for the master shutter signal, and a ground pin to equalize the potentials between the different circuits. A standard DB9 connector offers enough pins, shielding, and the option to secure the connector with screws for more stability. Since the control board does not have a side that is directly aligned with any side of the module case, a flat cable must bridge the distance to the front. Again a simple 2x5 pin header is sufficient to make proper use of the whole DB9 connector on the front of the case.

While the buffers fulfill their duties reliably in most situations, we experienced some slight interference on the bus, despite the shielding when high power conduits were running close to the connection cables. If those situations become more common, the buffers can be replaced by I2C transceivers which use differential pairs, such as the PCA9615 [62], for data transmissions that would be much more resilient against interference of this kind.

#### 2.3.2.4. Board Layout

Figure 2.18 depicts the first fully working version of the resulting board. For easier understanding, the functional areas discussed above have been highlighted and marked.

The camera connection portion includes the big connector block on top and supporting components duplicated for every single port. Each one has a transistor to boost the voltage of the trigger signal from the microcontroller to the camera and a protection resistor before one of the LEDs in the socket, to make the shutter pulses visible. The second LED is connected to a pin that provides $3.3V$ from the camera which is used to indicate proper connection and initialization of the camera. One pin is connected to a currently unused output pin of the camera in case it is needed in the future. The two remaining headers can be used to supply the camera with $12V$ power from the board, a feature that is not necessary as long as the cameras get enough power from their USB port, and to connect optional hardware to the camera's $3.3V$ supply.

The power control section consists of a 16-bit GPIO extender chip with an I2C connection. Each of these 16 pins is responsible for a single NUC. Every output pin has a LED and a current limiting resistor for visual feedback of each output's status. The transistors are responsible for activating the power cables by pulling the gate pin of the transistor in the power cable from $12V$ to ground. The connection to the cables and their transistors happens via the pin header and a flat cable on the right. Additional ground pins are not needed because both the control circuit and the cables are powered by the same power supply.

The fan area's main component is the I2C-programmable fan controller chip surrounded by a set of pull-up resistors, Zener-diodes for over-voltage protection, and the connectors for the two fan pairs and the temperature probe. A small inverter is connected to a set of status

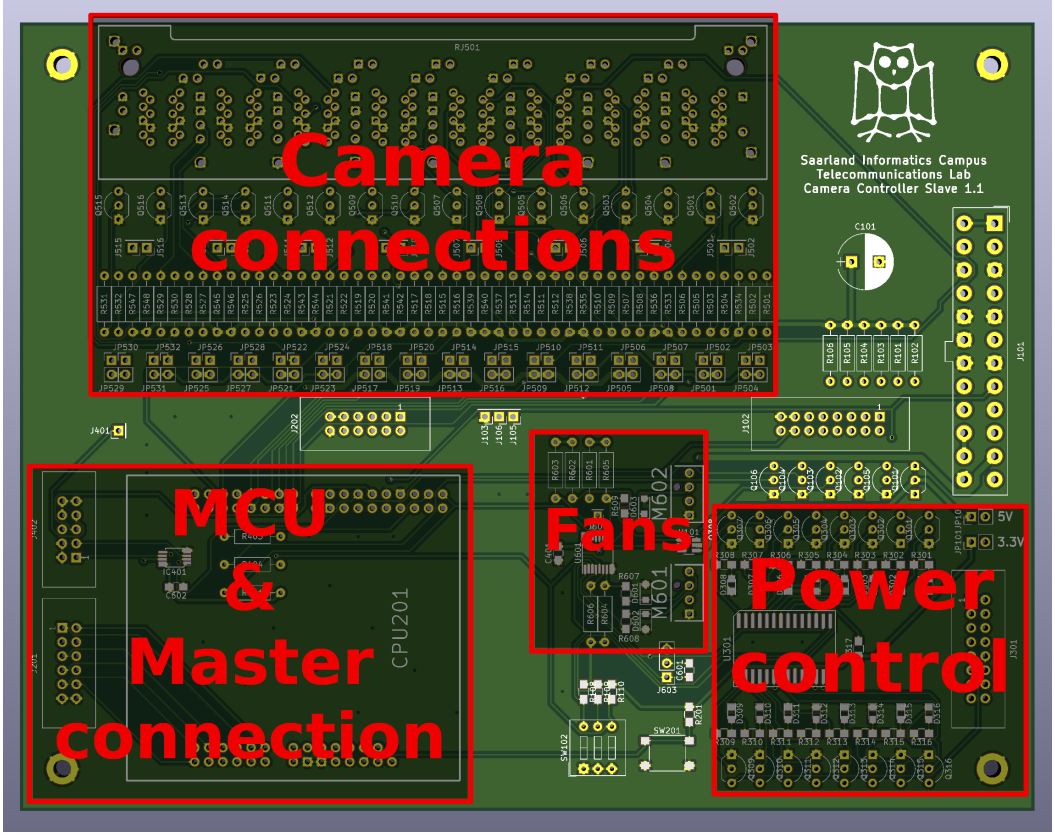



Figure 2.18.: PCB for module control version 1.1

pins from the controller to make the internal status easier to display via front panel LEDs.

The microcontroller section also contains the I2C buffer for the connection with the master board. The development board of the microcontroller is mounted upside down, simply connected using its pin headers. The software we are running on the microcontroller is mostly responsible for delaying the master trigger signal before relaying it to the connected cameras. How we ensure high precision for the configured delay is described in more detail in Section 2.4.3. Underneath sits the buffer chip with its pull-up resistors and a denoising capacitor. The top left connector takes the two I2C pins, the line for the master shutter, and a yet unused pin for arbitrary connections between the central controller and the camera control boards. The remaining pins are used for potential equalization between the grounds of the connected devices.

In the remaining areas, three major parts can be identified. On the bottom sits a dip-switch which determines the lower 3 bits of the I2C addresses of all devices on the board. The outputs are connected to the available address pins of every device on the board which talks to the I2C bus. Next to it, a push button to reset the microcontroller is located. On the top-right, we have the big 24 pin ATX power connector, a big capacitor for voltage stabilization, and the connector for the LEDs and switches on the front panel with a set of transistors and current limiting resistors.

Further details about the whole circuit are provided in the form of schematics in Appendix A.1.

The only change that was implemented after the schematics and the boards were created, is an increase in voltage for the shutter signals to the cameras. Using small upgrade boards which changed the voltage on the transistors from 5V to 12V, a complete remanufacture of the boards could be avoided. This change was necessary because most of the cameras we ordered, in contrast to their datasheets and technical references, did not trigger reliably with voltage levels of 3.3V. The exact required voltage for triggering varied from camera to camera and increased with a rising internal temperature. After a lengthy investigation from our side, this problem could be traced back to the manufacturer and faulty or insufficient optocouplers at the input pin. Since the manufacturer only agreed to repair or replace a small set of our cameras, we were forced to increase the voltage of our trigger signal to a safe level, so all cameras could be triggered within their operating temperature range.

### 2.3.3. Stand Design & Camera Mounts

The purpose of the camera stand is to hold the cameras and fix their relative position while allowing for a variety of camera layouts to be set up. Because there are not many institutions building camera arrays for multiview or lightfield capturing, there are no standardized methods or best practices for mounting the cameras.

Looking at early arrays in the literature, Zitnick *et al.* [63] at Microsoft Research used sections of square aluminum tubing mounted on normal tripods (see Figure 2.19a). All connections were made using drilled holes in predetermined positions and screws through the square tubes. The cameras themselves are mounted on camera ball mounts, allowing for limited rotation around all three axes. The overall stability of the system mainly depends on the stability of the tripods and ball mounts. Changing the relative position of the cameras would require drilling more holes or a full rebuild of the frame. Another disadvantage is the fact that the array only forms a one-dimensional arc. Adding another dimension is not trivial and would require a significant amount of support material.

Wilburn [14] uses a different approach for the mount of the famous Stanford lightfield array as shown in Figure 2.19b. He uses standing frames made from extruded aluminum

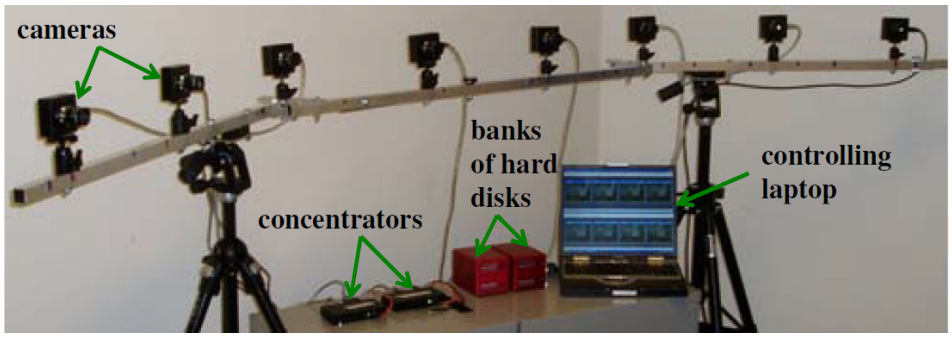
profiles. The cameras are mounted directly on horizontal struts using a small adapter made from acrylic. On these struts, the cameras can be moved freely in the groove of the profile to adjust their horizontal distance. Moving the struts in a frame closer together or spacing them out further allows for variations in the vertical distance. For dense layouts, a single frame is sufficient, but the cameras can be spread over multiple frames in case more area has to be covered or a non-planar arrangement is needed.

Newer designs such as the one from the Fraunhofer IIS [64] in Figure 2.19c or Google in Figure 2.19d are far less flexible. Their cameras are held on customized mounting plates or mounting brackets in a precisely defined position without any room for movement or rotation. While those fixed positions are easier to calibrate, as there are fewer free parameters, every time another layout is needed, the mounting plates or bracket holders have to be remade.

Our design was inspired by the Stanford design, as the newer examples had not been published when the majority of our design process happened. A modular frame design made from aluminum profiles is the base for our camera mount. Some renders of the intermediate stages during the planning phase are shown in Figure 2.20. On a height-adjustable portion with two horizontal bars, up to two camera frames, with a size of one by one meter, can be installed. To guarantee the full range of vertical movement, the vertical part is only supported towards the center and the rear of the frame. The frames are hanging off the horizontal bars for easy installation and reconfiguration but can be fixed in place with screwed connectors. Horizontal bars on the camera frames allow for a step-free adjustment of the camera's vertical distance. The cameras themselves sit on 90-degree angle pieces on the end of short extensions, with an options ball head for adjustability. The extensions are screwed into movable fixings in the horizontal bars, so moving them adjusts the camera distance in the x-direction. In their back, they have a mounting plate which guarantees they always form a right angle with the horizontal bar they sit on. These bars offset the cameras to the front of the frame and their length is used for proper cable management to minimize the cable movements directly at the cameras when the frame is manipulated or moved. The weight they shift from the center of the frame is counteracted by the weight of the cables going to the back and does not significantly influence the balance of the whole system. To increase mobility, lockable heavy-duty wheels are installed in the corners of the base rectangle. Similar to the module boxes, the shortest side of the stand's base was limited to 80cm for easy door traversal.

For the whole system, two stands with two camera frames for each were ordered. Compact layouts can fit into a single frame on either the top or bottom position on a frame. Uniform layouts that spread over multiple frames must take the thickness of the frame into account for both horizontal and vertical baselines. For multiview captures with larger camera baselines, the stands can be connected without a gap, which gives an area of two by two meters in total. Another configuration is shown in the bottom image of Figure 2.20. With lockable hinges between the camera frames, the frames can be aligned into a single row between the stand and even form an approximated arc. For additional stability of the 4x1 setup, an optional leg to support the frames in the center was added later.

Due to inconsistencies in the positioning of the planned screw holes in the aluminum profiles, the backplates of the holding arms have a gap of up to two millimeters to the strut they are supposed to touch. This means they allow for a rotation in the camera arms of up to ten degrees from the right angle they are supposed to hold. As an intermediate solution, a brace was designed to hold the arms at the correct angle while fixing their screws. It was sturdy but flexible enough to hold the arm and the back strut firmly but could be removed once the screws were tightened. A second brace was made to hold the cameras straight




(a) Array by Zitnick *et al.* [63]

(b) Stanford array by Wilburn [14]

(c) Array by Ziegler *et al.* [64]

(d) Array by Broxton *et al.* [65]

Figure 2.19.: Camera arrays with different mounting techniques

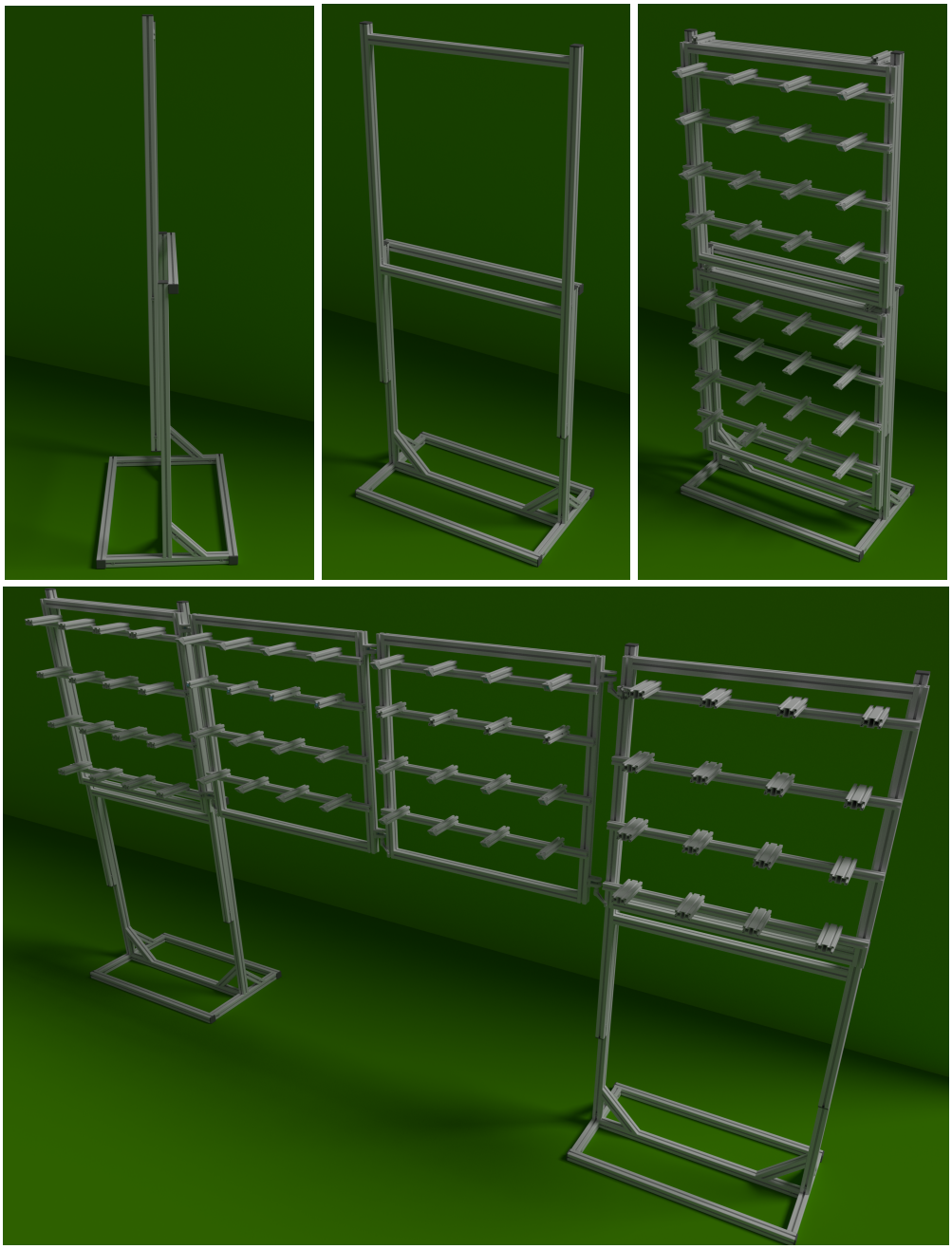
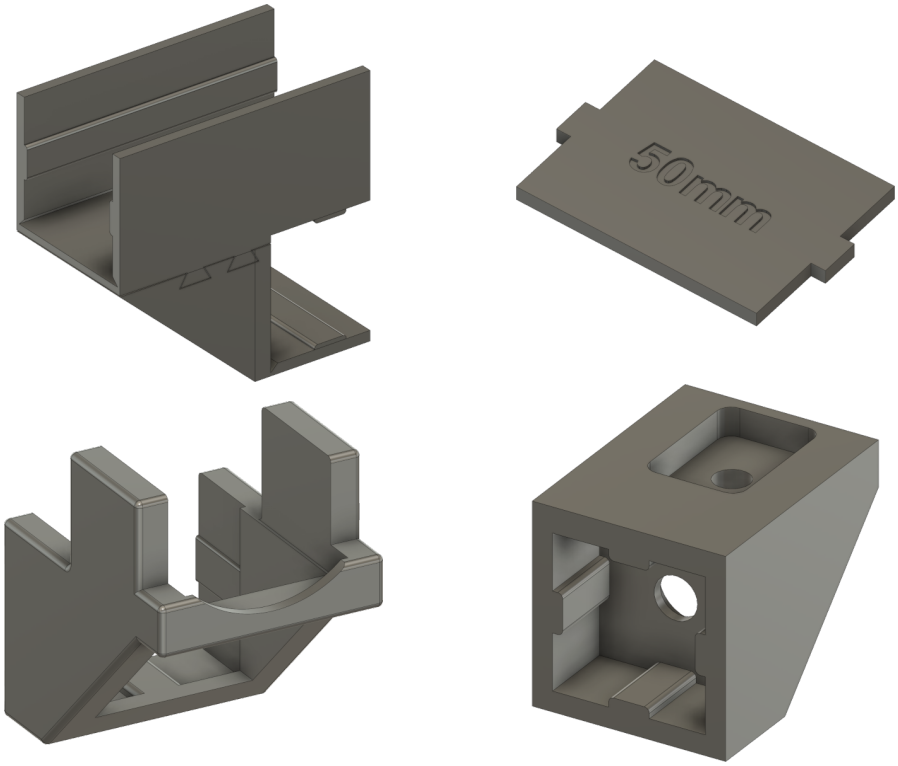


Figure 2.20.: Renders of our camera stand during planning phase




(a) Camera aligners first version (b) Camera aligners final version

Figure 2.21.: Different versions of camera alignment helpers

on top of the angle piece until they are screwed in place. It was slotted into the camera arm, held the front angle in the correct position while the extensions on top align with the sides of the camera and point it into the scene, directly in line with the camera arm. The designs in Figure 2.21a were manufactured using an FDM 3D printer in PLA and worked fine in our lab tests. Only after the first shoot under professional film production settings (details in Section 2.6.1), did it become obvious that the braces were not strong enough to hold the arms during the final tightening of the mounting screws and allowed for a few degrees of deviation from their optimal position around the y-axis. Manual fine-adjustment was required for both the camera arms and cameras on the angle pieces. Since this had to be done for each of the 64 cameras, ideally before every shoot or when the array was moved, and could only be done accurately for one camera at a time, the required time effort warranted a redesign of the adjustment helper pieces.

Making the walls of the braces thicker or manufacturing them from a harder material was not an option, as they had to stay removable so as not to interfere with neighboring cameras and cables in compact layouts. To eliminate the possible rotation of the camera at the end of the camera holder, the right-angle bracket was replaced with a custom part. At the bottom of Figure 2.21b, the result of the design process is shown. It inevitably resembles the angle piece somewhat, as it must hold the camera in the same position. Rotations around the length of the camera arm are eliminated by the back section which completely envelops the aluminum extrusion and prevents any rotation. The screw that previously held the angle bracket now prevents the new part from moving off the strut. The cutout on the

top has the same dimensions as the tripod mount on the bottom of the cameras. With an exact fit and a camera screw on the bottom, this holder fixes the rotation of the cameras around all three axes with no room for deviations. This does not allow for the use of the ball mounts underneath the cameras, however, our shoots prioritized precision and compact layouts. In any case, the holder could be replaced with the original angle piece should the ball mounts be required.

The alignment of the arms themselves was made easier by simple distance plates. They have to be remade for every new layout but due to their simplicity, the design and manufacture can be completed within minutes. For reconfiguration of the cameras, the first column is aligned by hand using a spirit level and a precise tape measure. The remaining columns are fixed relative to the first, with two spacer plates in the front and the back of the arm, maintaining the relative distance and zero rotation. Even if the arm in the first column has a rotational error, it remains the same in the whole row which makes it easier to detect, estimate its parameters and ultimately remove. Both new pieces made the setup and fine-adjustment of the mechanical components much faster and more precise.

### 2.3.4. Central Controller Case

Up to this point, the focus has been on the cameras and the modules rather than the central control unit to which they are connected. Its role is to coordinate the devices in the system, configure them, monitor their status, and provide an interface for an operator to access the system's functions. To accomplish this, two things are necessary. First, a way to collect all the cables from the modules for data aggregation and distribution. Second, a central server with sufficient network bandwidth to receive the data from all camera units, enough processing power to handle the incoming data, into a live preview for example, and hard drives long-term storage of the captured scenes.

An enterprise-grade switch with 24 ports can handle the uplinks from the modules (five Gigabit connections per module, 20 connections total). With two SFP+ ports capable of 10 GBit each, it can deliver the complete input data at full speed to the central server.

For the connections to the camera control boards and their microcontrollers, a different solution is needed. Since we have to deal with only partially standardized communication with specialized hardware over these connections, custom hardware is needed. Following a similar design process like the one for the camera controller board, the PCB in Figure 2.22 was created. The module connections on the bottom feature the same I2C buffer chip as the camera boards, to maximize the possible cable length between them. LEDs and buttons are handled by the front panel section. To control the temperature in the enclosure for all central server hardware, adaptive temperature control is implemented to activate the fans when necessary. The big connector on the left side leads to a 10-inch capacitive touch panel which can be used to control and monitor the most important features of the central controller. Additionally, its functions can be used via a simple REST interface accessible via the Ethernet port on the microcontroller development board. Here, the use of a full TCP stack was chosen since it is the easiest way to connect it to a full-size computer and because this device does not have any functions with hard timing dependencies, apart from the generation of the master shutter signal. By using a hardware timer in the microcontroller to generate the pulses and an internal interrupt signal to delegate them to a GPIO pin, this eliminates the dependency on free CPU cycles for a clean and jitter-free signal.

Further details can be found in the schematics of the board in Appendix A.2. Details about the programming on the microcontroller are given in Section 2.4.

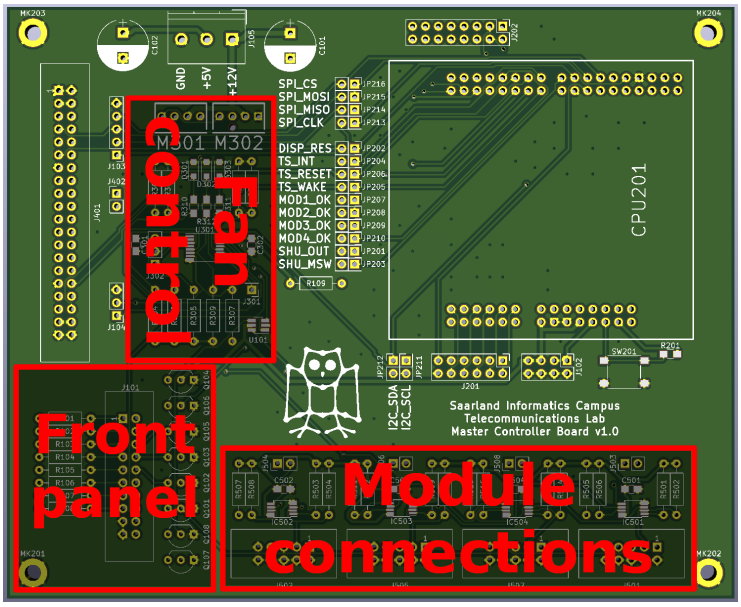



Figure 2.22.: Central controller PCB

Figure 2.23 shows the controller in its 19" enclosure working for the first time with a single module connected. The LEDs over the module connections indicate successful connections to the corresponding modules I2C bus. Above the left-most module connector is a 3.5mm audio socket which is used as an input for a button that triggers manual shutter signals. The LED next to it activates in sync with the generated shutter signal for visual confirmation. Its function can be configured in three different modes: Single, Burst, and Toggle. Based on this, the push of the shutter trigger button either generates a single pulse, a configurable number of pulses with a set delay between them, or continuous pulses with a set frequency until the button is pressed again. Next in the line are status indicators for over-temperature faults, stuck or broken fans, and the internal power supply.

On the left, a touch screen displays an overview of the current system's state. In addition

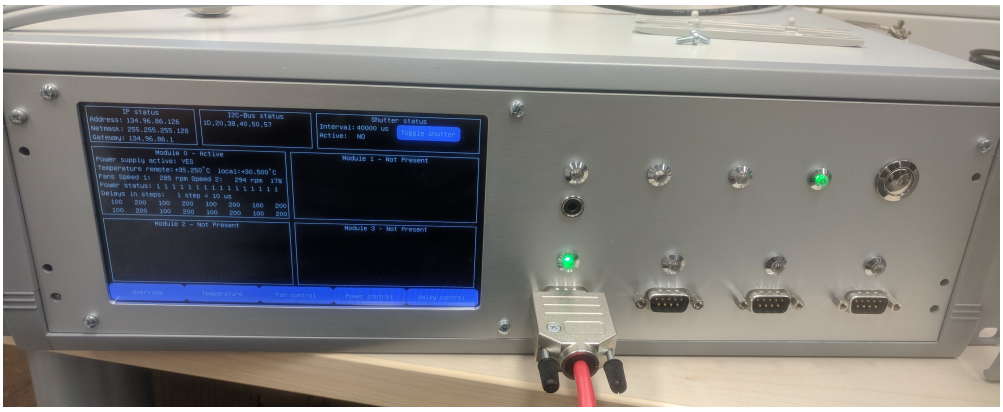


Figure 2.23.: First controller case test

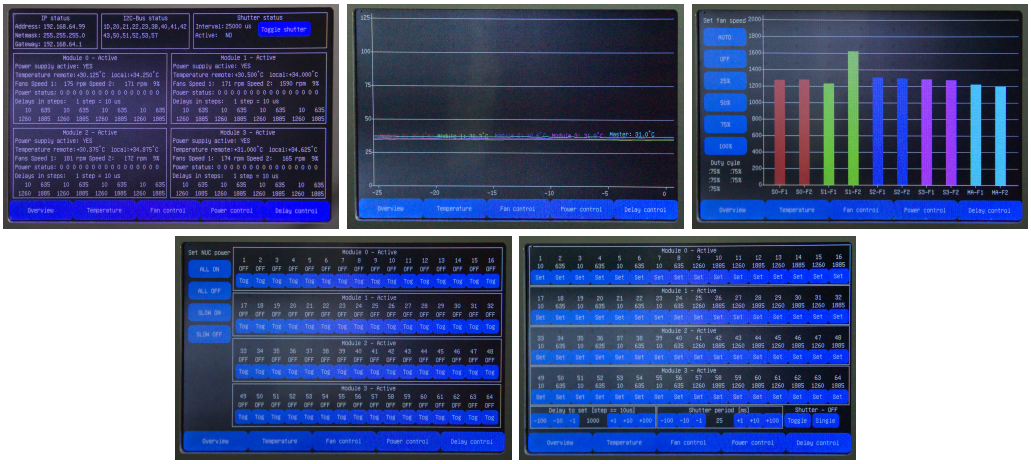


Figure 2.24.: Different screens available via the controller touchscreen

to the network status, including important IP addresses, it shows all active device addresses on the I2C bus and the current settings of the shutter generator. Underneath, for each active module, the following information is shown: status of the power supply in the module, measured temperatures in the module case, and the control chip itself, rpm speeds of the two monitored fans with the current PWM setting, the individual power status of the NUCs in the module, as well as the configured delays for each camera. With the buttons at the bottom, different views can be chosen which focus on a single aspect of the module and offer extended control options.

Extending the master controller for additional modules would only require a new front plate with more connections and the corresponding bus buffers on the internal board. Only in the case that more than 8 modules are used with this controller do more substantial changes need to be made, as then the three address lines would not be sufficient to assign distinct addresses to all devices.

#### 2.3.4.1. Server Hardware

Server hardware comes in many sizes and countless configurations. In order to have enough resources for live processing of some of the data, it needs to have at least one logical core and 4GB of RAM per camera. The total amount of 256GB of RAM was not a problem as server mainboards and processors usually support 1 to 2TB maximum. The only option which provided enough cores with sufficient speed at the time the server was built was that of using two CPUs on the same board. Even though CPUs with enough cores were available, their clock speeds were rather limited, due to the maximum amount of heat a single processor chip can dissipate, apart from being more expensive than two smaller processors. For network connectivity, a network card with two SFP+ ports for the internal network and a single Gigabit port for connecting the system to the rest of the world are necessary. The dual SFP+ cards are readily available since data centers already use speeds of 40 Gbit and beyond for internal communication and the Gigabit port is a standard feature on most mainboards. For storage, two mirrored SATA SSDs were chosen for the operating system and five 12TB hard drives in RAID5 mode for the camera data. After formatting, this results in 44TB available storage with the capability to withstand one failing hard drive. Considering that one full-length capture of about 20 minutes would result in $64 \cdot 200GB = 12.5$TB of raw


Figure 2.25.: First version of the controller case with all devices

data and storing the color version in raw format adds an additional 37.5TB, this is clearly insufficient for long-term use. This fact, in combination with the insufficient write speeds observed during the first production with the array (see Section 2.6.1), which never exceeded 30-40MB/s after the hard drive cache in the server's RAM was filled, led to the addition of the network-attached storage described in Section 2.3.7. For experiments using the view interpolation technique in Section 7, the option for a full-size GPU was included in the requirements of the server, which made a higher enclosure and a bigger power supply necessary.

The final specifications for the server are as follows: two Intel Xeon E5-2697v4[11] CPUs with 36 logical cores each which share 256GB of RAM, an Intel X520-DA2 network card, and a RAID controller based on the 9261-8i chipset. It is housed in a 4HU server case high enough to fit a GPU in the PCIe slots and with enough space for the five hard drives. The custom software we wrote so the server can fulfill its required functions is described in Section 2.4.

#### 2.3.4.2. Mobile Controller Case

In order to keep the stack of devices around the controller as mobile as the rest of the array, they are mounted in a custom flight case with built-in 19" rack rails. In Figure 2.25, the front of the completed case is shown. From top to bottom, there are power switches for all

[11] `https://ark.intel.com/content/www/us/en/ark/products/91755/intel-xeon-processor-e5-2697-v4-45m-cache-2-30-ghz.html`

devices in the case, the master control box, a small storage drawer for short cables, and other smaller peripherals during transport, a keystone patch panel with the modules for the camera module uplinks (1-20), a USB3 connection to connect peripheral devices to the server, a DisplayPort connected to the server, one Ethernet port to connect the whole system to the Internet and a USB port connected to the microcontroller in the master control box for reprogramming. The last component is the main central server. The switch is mounted in the back, opposite the power distribution unit on top. During normal operation, only the front of the case needs to be open because the only external connection required, apart from the ones coming in through the front, is the power for the case which enters through the side of the case.

### 2.3.5. Cabling

In total, the whole camera array contains over a kilometer of cables. Since they have many different purposes, lengths, and connectors, in addition to the fact that some of them are not used for their original purpose, careful planning was necessary. Additionally, they can carry a significant amount of current in some areas, making them a potential fire hazard if not chosen correctly.

The power cables which connect the power supply to the NUCs carry the most current in the whole system. With the original power supply rated at 65W [54] and us using only 12 instead of the default 19 Volts, the cables have to carry a total of $64W/12V = 5.42A$. The responsible DIN VDE standard requires a minimal wire diameter of $0.75mm^2$, which is enough for 6A continuously [66]. One end has to have a male barrel connector with a $5.5mm$ outer and $2.5mm$ inner diameter to connect to the NUCs. The other end has to plug into the power supply, for which we use a custom mini fit jr. connector from Molex, as discussed in Section 2.3.2. According to their specification, each pin is rated for up to 6A when used with a cable between 18AWG and 20AWG [67], which happens to be precisely the equivalent of a $0.75mm^2$ European wire (between 18AWG and 19AWG to be exact). Knowing the components can work together while operating within their intended parameters, the cables were manufactured from barrel connectors with 2m of attached cable, Molex connector housings, and the matching crimp terminals after the cable was shortened to the required length. The transistor circuit for the individual power switching capability was added to the finished cable once it was complete.

The cameras need two cables each: a USB cable for power, data, and configuration and a generic one for the GPIO pins including the shutter control. For the USB we chose a high-quality USB 3.0 A to micro B cable with a length of 5 meters and screws to be able to lock it in place on the camera's backside. As discussed earlier, 5m is the maximum usable length without intermittent data losses, meaning that the cable must pass directly from the NUCs to the camera, taking into consideration the approximate 1 meter of length within the module before it extends beyond the case. Considering the possible stand configurations from Section 2.3.3 and assuming the modules are always located around the center of the whole setup, the remaining four meters are more than sufficient to reach every camera in the layout. In most cases, there is still enough cable left for proper cable management.

The second cable only carries the shutter signal to the camera and optionally some low voltage, low current power. Its connectors on both ends have been decided by the cameras we chose and our electronics designs. We made the cables based on readily available 5 meter long Ethernet cables. Since crossover cables have become less and less popular in the last decades and patch cables became the standard, we can assume that the connectors on both

sides have the same pinout. In order to wire the GPIO connector correctly, the only thing we need to determine is whether the cables follow the TIA-568A or TIA-568B standard [68] because this determines the order of the colored wire pairs in the RJ45 connector. Luckily, even when the seller does not specify this explicitly, finding the order requires just a multimeter or sometimes even only a good eye, depending on the transparency of the connectors. Once the order of the pairs is known, soldering the GPIO connector to one end of the cable becomes trivial. The two unused wires are snipped off on the camera's side. Since those cables follow roughly the same path as the USB cables, they lose roughly the same amount of length inside the modules. Again this is not a problem, and even if it were to become one, these cables could be easily replaced with longer versions, because they only carry signals with very low frequencies compared to the USB.

To keep the signal between the microcontrollers in the camera modules and master controller as clean as possible, a shielded Ethernet cable was chosen for this purpose. Internal tests have shown that those cables work reliably up to 7.5 meters and become unreliable at 10 meters. To have some safety margin in extra noisy environments we chose a length of 5 meters between the camera modules and the master controller.

For the manual shutter trigger, which only connects a push button to the master controller, we chose a simple audio cable with high flexibility and damage resistance. An additional layer of protective electrical shielding reliably prevents erroneous impulses from being introduced, even when the cable runs close to heavy noise sources. With a length of 10 meters, it allows an operator to control the shutter comfortably even when standing in front of the array, away from the controller.

All remaining cables are used for their intended purpose and were simply bought off the shelf. The only modification we performed was on the power cables for the power supply and the switch in the module cases. They were changed into a y-configuration and connected to the power input socket of the module. That construction reduced the number of power cables required per module to one, which makes them more convenient to use without resorting to huge power strips. We also chose a power inlet that takes cables with a standard IEC-60320 C13 connector with a dedicated fuse and on/off switch for added safety and controllability.

### 2.3.6. Hardware Provisioning

The NUCs that control the cameras and gather the data from them are normal consumer-grade computer systems. Therefore, they require the installation and configuration of an operating system before they can do anything. In case of hardware failure or when new major releases of the operating system are required, this has to be repeated. Doing this by hand for all 64 devices, using the default installation methods, requires a significant investment of time and effort, in addition to physical access to devices without which the installation medium can not be connected to the computers and the setup can not be controlled.

To get around this, we deployed a PXE environment based on iPXE[12] in the array's network. A TFTP server [69] provides the files and the DHCP server provides further details to clients. Both services are running on the central array server and are provided to all clients on the network. All NUCs in the network are configured to perform a network boot by default. They download the iPXE executable from the server, together with a

[12] `https://ipxe.org`

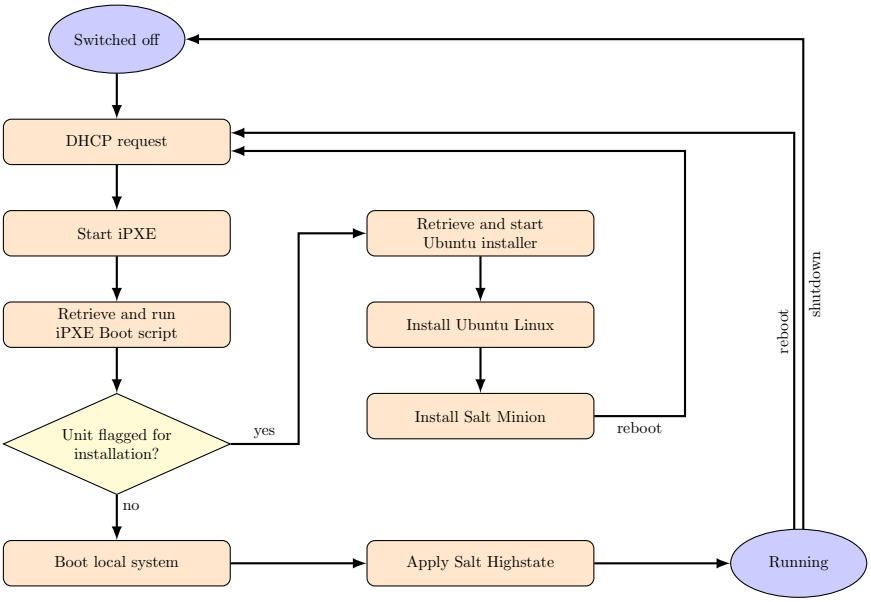



Figure 2.26.: Major steps in the boot sequence of the camera units

configuration file that provides a menu with different installation options. Without user interaction, it boots whatever is installed on the system drive. When a keyboard and monitor are connected to the computer, it allows us to start and guide the setup of our operating system of choice. For the NUCs, this is Ubuntu Server 18.04, for other machines in the system, it is CentOS 8. This setup removes the need for connecting a physical medium to the computers when they need to be re-installed.

The process of the OS setup is still a lengthy operation, as it involves unavoidable waiting times between steps and the transfer of the peripherals between the computers. Considering the fact that the setup of all NUCs should be identical apart from the hostname or some other identifying property, it can be automated further. An auto-install script provided by the TFTP server can accomplish that but using the script alone leads to completely identical installs including the hostname. To get around this problem, we generate a hostname string based on the MAC address of the device at the beginning of the PXE script and give it to the Ubuntu setup, using a kernel parameter that takes precedence during the setup. That step allows us to fully automate the setup for our purposes once it has been started. Triggering the setup still requires physical inputs for the devices because changing the default option in the startup PXE script would cause any rebooting machine to completely reinstall its system. We prevented that by adding one new function to the PXE script. After it generated the hostname for the machine, it requests a second script with the hostname in its path from the central server. By default, this request fails and it falls back to starting the menu described above. If we want to trigger a reinstall, we make the server return a minimal script that forces the PXE into the automated setup process. Since the hostname, given in the URL of the request, uniquely identifies one device, we can directly target a single device which we wish to reinstall with this approach.

Figure 2.26 shows all components of the boot sequence and how they work together. The overall setup process installs a headless system with the minimal set of packages and configuration files required to allow the unit to connect to the cluster control system

described in Section 2.4.1, which takes over once the setup is complete and takes care of the remaining required packages and configurations.

Since we use a network installer for the setup, nearly all system packages have to be downloaded from one of the official servers. In particular, when multiple machines are running the installation at the same time, the Internet connection speed quickly becomes a bottleneck. Additionally, the router of the system risks being blocked from certain package mirrors, because it requests the same data up to 64 times in short succession. The solution for this issue is to install a Ubuntu repository proxy on the central server. We modify the auto-setup script so that it accesses the proxy, instead of the external mirrors. This way, the packages required for installations or updates are only downloaded once and then distributed from the server's cache. With the reduced load on the Internet connection and the ability to make full use of the multi-Gigabit connections in the local network, the installs become much faster and their speed is nearly independent of the number of computers involved. The short installation time from reboot to full functionality, of 10-15 minutes, makes full reinstallation the tool of choice to fix situations in which the behavior of a single computer deviates from the rest after cable connection issues have been ruled out.

### 2.3.7. Storage Cluster

After the first production with the array, it became clear that the storage in the central server was insufficient for bigger productions in both capacity and speed. The transfer of the data from the camera units to the central server starts very fast until the RAM available for filesystem caching is filled. Beyond that, the speed is limited by the RAID controller's capability to store the mass of relatively small files on the available hard drives. Since the number of drives is low compared to the number of parallel instances trying to transfer data and hard drives handle random accesses relatively slowly, due to their internal moving parts, the transfer of a 15-minute sequence took multiple hours to complete. Even when known beforehand, such enforced downtimes are unacceptable for any production.

This lead to the search for a better storage solution. While being fast enough to quickly transfer the data, the data needs to be safe even when small hardware failures occur and since the system is still a research device, it should be easily extendable when the need arises. In addition, it has to be as mobile and transportable as the rest of the system. The following options were considered:

#### Cloud Storage

Rented cloud storage is now more available than ever. It can be scaled up and down dynamically to nearly unlimited sizes but the exact physical location of the data can only be determined up the continent or country in which the data center is located. The provider also has to be trusted in regard to data security, as the customers do not have any direct control over data replication or access control checks. The major drawbacks in our case are the monthly costs that can accumulate very quickly when multiple tens of Terabytes have to be rented and accessed regularly and the Internet connection speeds that would be required. The current storage in the server stores between 200 and 400 MBit per second in its slow phase after the cache is filled. Common Internet connections in Germany, apart from the ones in large corporate and university networks, do not even offer upload speeds in the same order of magnitude [70]. Even though there are no initial costs for this option, the drawbacks outweigh the great flexibility of this solution.

### Single Large Storage Server

One big storage server with many hard drives in the system can distribute the load much more evenly. A big RAID controller or special filesystems like ZFS can add redundancy to the data for protection [71]. Server cases with 60 or more hot-swappable bays for 3.5" hard drives are available from many manufacturers with a multitude of options for CPUs, RAM, and network connectivity. Full control over the data failure protection on a filesystem or hard drive level with very high achievable network speeds and a great choice of access modes make this solution a strong contender in our case. The main negative points are the existing limitations when the storage needs to be extended. With RAID controllers and ZFS, the chosen operating mode determines the granularity of how the capacity of the storage can be increased. Usually, such systems are configured with all bays occupied and if the capacity needs to change, either all hard drives or the whole system is replaced and configured again. Therefore, there is often no benefit of adding single new drives or using new drives with different capacities than the ones already in the system, either because the single drive can not be added into the active pool of drives or a bigger replacement drive is limited by the smallest drive in the system. Only a complete exchange or adding a bigger number of drives effectively increases the capacity. Decreasing the capacity to change the ratio between protected and unprotected space or changing replication parameters requires a full backup and recreation of the affected storage sections. All these facts require very careful planning of system settings, that can not be changed later without a full reinstall, which also leads to complete data loss. Creating a full backup for up to 200TB of data would require a significant amount of time and monetary effort.

### Local Storage Cluster

Storage clusters are the conglomeration of multiple smaller storage servers to form a single entity. They fall into the area of software-defined storage. The rough hardware requirements are similar to those of the big storage server, only distributed over multiple machines with fewer hard drives. Only RAID controllers are not required because redundancy, recovery, distribution, and all other data-related operations are handled by the supervising software. Hard drives are used to store the basic data blocks and new drives can be added dynamically, but only work efficiently if similar amounts of storage are added to a sufficient number of failure domains, depending on the configured replication. There are different systems with different feature sets available and a small number of these were evaluated for use in the future camera array. LizardFS[13] offers a virtual filesystem that spans all drives in the cluster, with dynamic redundancy rules that can be applied and changed on the folder level. Because of problems we had with the data recovery in case of hardware failures at the time of evaluation, we favored a different system. Tests using CEPH [72] showed more promise. Based on the Reliable Autonomic Distributed Object Store (RADOS), it offers three different modes to access the data. Data objects can be used directly, with a unique name and arbitrary sizes and contents. Those data objects can be accessed and modified using the RADOS library available for most Linux-based systems. CEPH also supports RBDs or RADOS block devices which are virtual hard drives that can be accessed remotely, integrated into a system- and then used like any other hard drive. Last is the CephFS, a filesystem that can be accessed by multiple clients at once and behaves like a network shared folder. Its redundancy and other settings are configured on a pool level, where a pool is a virtual data area which, by default, spans all available storage in the cluster and contains either raw objects, RBDs, or data for a CephFS. The performance and reliability

[13] `https://lizardfs.com`

of the system can not be proven, but its support from major industry players in the form of the CEPH foundation[14] and the fact that important research institutions with very high capacity and speed requirements like the CERN in Switzerland use it for an increasing number of parts of their storage infrastructure [73], speaks for itself.

Following the CEPH hardware recommendations, we acquired four servers with 16 logical cores, 64GB of RAM, and 240GB M.2 SSD for the operating system. Each server has 12 slots for 3.5" hard drives, two 10GBit SFP+ ports, and two Gigabit Ethernet ports. In total, we distributed 32 datacenter grade hard drives from two different manufacturers with 12TB each over the 4 servers. Each server runs a CEPH monitor, a CEPH manager, and a CEPH OSD for every installed drive. For stability and higher data security, an additional CEPH monitor is running on the central array server.

For best performance and the least amount of overhead, we decided to store our data directly as objects. It is an excellent match for our data structure, using one file per frame and camera. The name of all objects consists of three major parts. First, an arbitrary string that identifies the scene they belong to. The second and third part depends on the type of object. For frame data, it is a three-digit camera id and a five-digit continuous frame number. Calibration data for rectification has the keyword "calib" and the corresponding camera id. The color correction profiles have the keyword color instead.

Important data whose recreation requires a lot of time and effort, such as the raw data from the cameras, is stored in a replicated data pool so the data is always stored on at least two servers at all times. Other data created by processing the raw data, is stored without replication because, in case of data loss, it can be easily recreated with only very small human interaction, and replicating it would cost precious storage space. To optimize the transfer speeds after a capture session, a non-replicated cache pool is added to the pool for raw data. During normal operation, it is always kept empty to guarantee that the current version of the raw data is always in the replicated storage pool. Only when large transfers from the camera units are made is the cache pool allowed to fill briefly, and then to be drained immediately after the camera units have transferred all their data. The reasoning behind this is that in the unlikely event of a hardware failure between the time the data is captured and when it is completely replicated, the scene data either still exists in the camera units cache or can be reshot because the array and the required props and actors are still on set.

To use most of the speed of the storage cluster over the network, we upgraded the majority of the camera arrays network to 10GBit links. Each member of the cluster has one 10Gbit link to the camera array local network and one 10GBit link to a private network only for the storage devices. CEPH recommends such a setup to separate the data transfers from and to clients from the traffic created by internal replication, verification, or other maintenance tasks in the cluster. The whole cluster connects to the rest of the arrays network with two aggregated links. To accommodate the additional high-speed links in the controller box, the switch there had to be replaced with a model having more 10Gbit ports. The upgrade also affected the modules, whose five Gigabit uplinks were replaced with a single fiber 10GBit connection. The useful bandwidth for every NUC in the modules is only increased when not all of them are active at the same time, as the speed of the existing bottlenecks to the central server and the storage cluster are still limited to 20GBit/s in total.

The servers can produce a significant amount of noise due to their fans, depending on their current workload. For capturing sessions with audio recording, in which the array should be as quiet as possible, it might be necessary to place the storage cluster far away

[14] `https://ceph.io/foundation`

Figure 2.27.: Front and back views of the storage cluster case with all connections

from the rest of the array. Over relatively cheap OM3 optical multi-mode fibers, a 10GBit connection remains stable for up to 300 meters [74]. For the noise reduction effect to work, we do not need to make use of the maximum possible distance, but 100 meters of OM3 fiber are reserved for such cases.

For mobility reasons, all devices utilized in the storage cluster's operation are built into a flight case as shown in Figure 2.27. With additional fans for better cooling, the storage can be safely operated in most environments. Overall, it merely requires a single power connection to start working and an uplink to the camera array for the data to be accessed or modified.

The evaluation of the storage cluster delivered the expected results. With a total raw capacity of 349TB, we have approximately 233TB of usable storage when the raw and processed data has the same size. This approximation is slightly conservative because the processed data in the non-replicated pool will always be bigger than the raw data, since these frames have three color channels instead of one. Even when compressed, they still need more space than the raw sensor data in the raw pool. Therefore, the usable capacity is slightly higher in reality. By installing the RADOS library on the camera units, they can access the cluster directly without having to go through the central server first. With this configuration, we can now write data so fast into the persistent storage that the network uplink of the storage cluster becomes the bottleneck. This means we can write about 2GB per second into storage, but how fast it can go beyond that point has not been tested. Overall, the time required to copy captured data from the caches to the storage is now about three times as long as the capture time. This is still quite far away from real-time, which is not achievable in any case as explained in Section 2.2.4, but it is far more manageable than the situation described before. In case it ever needs to be faster in the future, another server can be added into the case or more drives can be added to the existing servers. Additionally, more links can be added to the connection to the rest of the system, but because the switch in the storage cluster is nearly full, everything beyond a small improvement requires a bigger switch.

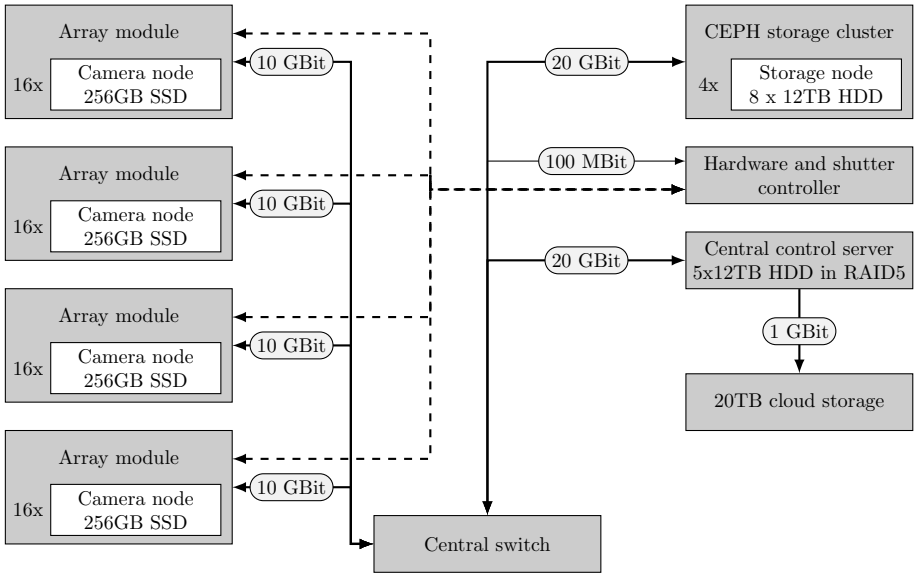



Figure 2.28.: Connections of major components in the camera array. Dashed lines represent GPIO and I2C connection. Uninterrupted lines represent Ethernet connections.

Figure 2.28 gives an overview of how the major components are connected. The only part that has not been introduced yet is the cloud storage in the bottom right. To share the data with project partners or other interested parties, direct access to the cluster is not the ideal solution. The main problem would have been the connection from the outside to the cluster. In the university network that would have been possible, but whenever the array is moved for a production and is connected to a different network, there is a downtime for everyone. Explaining the internal data structure and the numbering system for the cameras, which is based on their position in the modules and differs from the one used in most lightfield examples, would have added even more complexity. The projected effort to carry this out would have been too taxing in comparison to the benefits gained.

Instead of a direct share, we followed a different route. By financing a small extension to the Nextcloud[15] instance hosted by the computer science department of Saarland University, we reserved a share of 20TB for our purposes. Its location at the university guarantees a high availability and connection speed. To reduce the need for additional explanation, we only provided the data in a format compatible with the popular lightfield toolbox by Dansereau *et al.* [75, 76], whose way of counting cameras and representing data became the default for most new algorithms and is well known by the researchers working with lightfields. Since the procedure is very similar for every scene, the process of creating the archives with the frames in the correct order is handled by scripts as described in Section 2.4.4. Reordering hundreds of thousands of frames by hand every time something changes in any step of the processing pipeline (see Section 2.5) and project partners needing access to the new data, would simply not be feasible.

[15] `https://nextcloud.com`

## 2.4. Implementation Details

So far, this chapter has mostly covered the hardware of the camera array and its connections. Since a total of 69 computers need to be managed and controlled for normal operation, the required processes have to be fully automated or at least be supported by automated systems. To achieve this goal, a combination of already available and custom-made software was needed. The controlling instance for most of these systems is situated on the central server to keep the configuration in one place. In the following sections, the responsibilities of each system are explained.

### 2.4.1. Cluster Control

The cluster control software keeps the software and configuration on the systems up-to-date. For this, we use the Salt system[16]. Its communication is encrypted and it comes with functions to configure nearly any aspect of common Linux-based systems. All functions check the current system state before executing. When a change is not needed, it is not executed unless it is forced. In case a function is required that can not be performed using the onboard tools, new functions can be added in the form of modules written in Python. Specific clients or groups of clients can be targeted via their hostname or properties called grains which represent system properties like the network or system configuration.

With this system in place, maintenance tasks like updating packages or rebooting certain units boil down to simple short console commands. Overall, it simplifies many tasks in the day-to-day work with the array, including setting up freshly installed components.

Once the automated OS installation from Section 2.3.6 has finished, the computers connect to the master instance. They first receive the so-called high state from it, which defines the base commands to be executed whenever a client connects to the master. For us, this means the internal SSH server is activated and configured so that the central server can make connections without a password in case manual connections are necessary in the future. After that, a PTP service that synchronizes the internal clock to the central server is started. Keeping the clocks synchronized is important for the functionality of the storage cluster and delayed tasks on the camera units.

From there, the path deviates depending on whether the computer is part of the storage cluster or part of the camera modules. The camera units add additional package repositories, install the necessary drivers to be able to use the cameras, as well as the packages and services for our unit control system. Members of the storage cluster only configure a set of CentOS internals to work with the rest of the array and install the packages for the CEPH operation with the required configuration files and certificates.

Custom modules we implemented for the array include deployment of OSDs on unused hard drives on the storage nodes, upgrading the firmware of the cameras, and many tasks for transferring and processing the camera data on the camera units, most of which will be discussed in Section 2.4.4. To identify problems with the drivers or the connection of the cameras, we implemented additional grains which identify the current status of all connected cameras using the manufacturer's API, giving the camera model, serial number, and firmware version. To ensure proper functionality for the preview during capturing, another grain reports the accessible hardware encoding and decoding capabilities of the system.

Salt's file transfer function was used in the first stages of array development, to transfer

[16] `https://github.com/saltstack/salt`

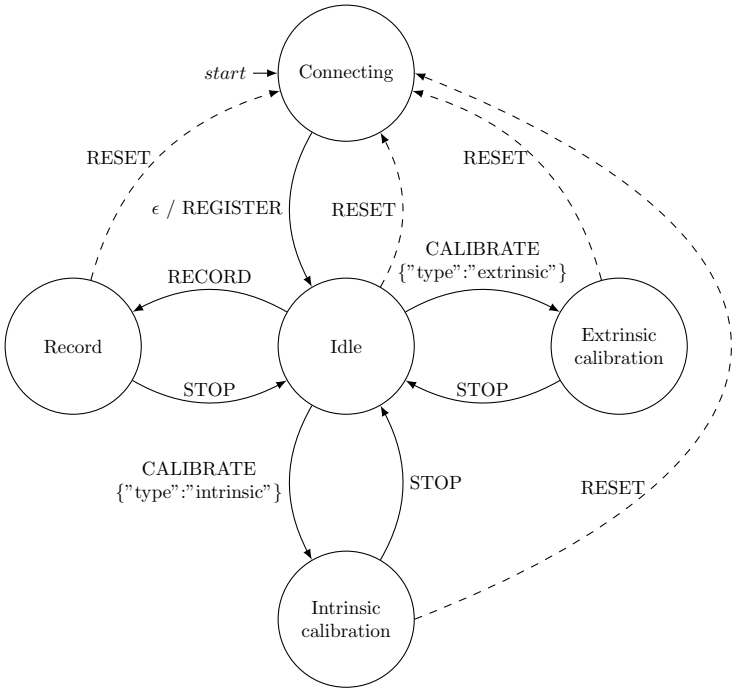



Figure 2.29.: State diagram of possible client states in the Argus control system.

the frame data from the caches to the server. It was fast enough to saturate the storage system in the central server, during the fast phase which mostly went into the filesystem cache and was limited by the network connection, and the slower phase which overwhelmed the hard drive raid, as discussed in the beginning of Section 2.3.7. The main reason why its use for that function was discontinued later, is that a transfer to the storage cluster would always have involved a now unnecessary intermediate step on the central server.

### 2.4.2. Unit System

While the computers are set up and kept up-to-date using the salt system, parameters concerning recording sessions can change many times between reboots. To make those changes possible without restarting running processes and to keep the system as responsive as possible, another system was needed.

Named after the many-eyed giant from Greek mythology, we created the Argus controller, partly in the form of the thesis by Frank Waßmuth [37]. Based on Twisted[17], an event-driven networking framework, and Msgpack[18], it is written in pure Python and is precisely tailored to our requirements. With continuous sequence numbers in every message, it allows for reliable asynchronous communication between the clients and the server.

For the different operation modes of the array, representative states exist in Argus. In Figure 2.29, the states are shown with their possible transitions. All units start in the Connecting state, in which they only try to register with the server. Once registered, they switch to the idle state, in which they wait for further commands. From there, they

[17] `https://twistedmatrix.com`
[18] `https://msgpack.org`

transition to the task-specific states which all relate to some operation using the camera. When a Stop command is issued, they perform calculations on the data they acquired, undo all changes made to the settings of the camera when the state was entered, and send the results back to the server before going back to the Idle state. All states also listen to a reset command, which forces them to go back to the connection state. It is used in case the communication with the server stopped working and the units are in different states after it has been restarted.

The Record state is most commonly used and most versatile. It starts a GStreamer pipeline that sends the output from the cameras as an H.264 stream via RTP to a chosen host and port, which usually is the central server. By default, the cameras are set to capture color images with a resolution of 1920x1200 using the internal oscillator for the shutter. This is ideal for quick previews, setting up the focus and aperture, and aligning the cameras on the stand. As the quality of the internal debayering process in the cameras was deemed unsuitable for professional production, a mode that captured the raw sensor data and only debayered them for a color preview was added. For actual capture, the frames from the camera are duplicated before encoding them for the preview and then saved as PGM or PPM files, depending on whether they contain color or not, and then stored on the internal SSD with an increasing sequence number. That format, which minimizes the overhead required before writing them to the disk, was chosen as it is fast enough to handle the data in real-time without caching. All these parameters, including the ones regulating the shutter input, are processed and handled by a custom GStreamer plugin that forms a wrapper around the camera manufacturer's Spinnaker SDK[19].

After the first couple of internal shoots, the recording state was extended by two convenience functions to make the setup faster and easier. The details about those overlays are part of Section 2.4.4.

When the Stop command is issued while recording, the GStreamer pipeline is torn down and the camera parameters, especially the ones regarding the trigger configuration, are reset. In the case that this is not done, the cameras can remain in a state which does not allow the pipeline to restart in the future. When it happens, the only way to restore normal function is to power cycle the camera by disconnecting the USB cable or removing power from the computer to which it is connected.

The calibration states behave the same while being active. Both capture color frames at full resolution using a configurable trigger. Apart from being transmitted for preview, just like for recording, the images are run through a custom GStreamer plugin which detects a chosen calibration pattern in them. The detected patterns are counted and their coordinates are collected. When either the requested number of patterns is found or the stop command is issued, the behavior of the states diverges.

The intrinsic calibration state takes all collected samples, runs them through the intrinsic camera calibration function in OpenCV, and returns the final coefficients to the server. There, they are collected and stored for future use.

Since extrinsic calibration can only be performed with the data from multiple cameras, the extrinsic state sends the detected points directly to the server without processing them. Once the data from all cameras is received, the server checks whether all cameras have a valid result from a previous intrinsic calibration. Only when this is the case does it create sets of matching points between a central camera and each remaining camera to use them in the OpenCV stereo calibration function. The results are then stored for future operations.

[19] `https://www.flir.com/products/spinnaker-sdk`

While the recording state is used in most array operations, the usefulness of the calibration states in their current version has diminished over time, mainly due to the problems discussed in Section 2.2.4. Since the more recent solutions for calibration vastly differ from the ones presented here, (see Section 2.5) and they were rapidly changing for quite some time, putting them into a format that could be run on the server inside the system was not possible yet and is left for future work.

The system controlling the camera state before, during, and after recording, is a Python service without a human accessible interface. For easy usage of the whole system, a REST API was added to the unit control system and a simple web page was created to interface with the API. Apart from the functions for state transitions, it includes functions to set and get the internal parameters of a single or all camera currently active in the system. That way, the cameras units can be monitored and controlled with any device featuring a decently modern web browser. To extend the control to the whole system, interface functions for the controller box were added. That way it is not necessary for users of the web interface to consider whether a function is controlled by Argus or the control box.

### 2.4.3. Shutter Control

The shutter delay for each camera and the power to the NUCs are controlled by a single microcontroller with only one core inside the master control case. Its software is based on FreeRTOS[20], which separates the different parts of the software into tasks with different priorities. For the master unit, this includes interfacing with the touchscreen, handling its Ethernet interface, including the simple REST API, gathering information from all I2C devices on the bus, and generating the master shutter signal.

Since even UI libraries with a very small footprint were too much of a burden on the available memory in the CPU, we designed a custom library that heavily depends on the display controller's capabilities for drawing geometric forms and text. With the help of the display's capability to only update parts of the screen, a fairly fluid user experience could be achieved with multiple data updates per second. Whenever an input on the touchscreen is detected, the coordinates are tested against all buttons on the screen and if they are inside a button, the corresponding function is executed. By generating text on buttons and labels using functions during rendering, the amount of memory reserved for caching texts was minimized at the expense of more CPU cycles, because the strings need to be created and formatted more often. Using that approach, the only things we need to store for a UI element are two 16bit coordinates for its position, two 16bit values for width and size, three bytes for its main color, and two optional function pointers. One is called to create the text visible on the element, the other is executed when the element is clicked or activated. Complex status screens still put a heavy burden on the available memory which means all other operations have to limit their memory consumption to the absolute minimum for the whole system to work properly.

Because the display uses OLED technology which tends to show burn-in effects when static contents are displayed for longer periods of time [77], a kind of screen saver was implemented. After a set idle time, the display contents are replaced with a grid of buttons filled with random colors which change every few seconds. The ever-changing display color effectively prevents the burn-in effect on the whole display. A click on any button stops the screensaver and shows the screen which was active when the screensaver was started.

[20] `https://www.freertos.org`

To supply the screen with information, it has to be gathered from the distributed devices. By polling all available addresses on the bus, the list of currently active devices is kept up-to-date. After detection, several registers are read to find out whether the device has been connected already since it was last turned on and if it is newly started, it is set up for normal operation. The information from status registers in the devices is read and stored in regular intervals, to be shown on the display. When the content of the device registers needs to be modified, this happens between the polls. In most cases, there is no special order for the communication since, after the initial setup, all regular communication is only reading values, and write operations are not time-critical, so there is no problem when the new device status is not represented in the controller's data until after the next read period. While being powered on, all devices on the bus remember their current state without regular updates and continue operating based on their current parameters. Therefore, they do not require re-initialization when the connection to the master becomes unstable or breaks completely. Once the connection is reestablished, data updates continue as if nothing had happened.

To trigger changes in the configuration of client devices at any moment other than their initialization, a simple REST API was implemented on the microcontroller. For this to work, it requests an IP address via DHCP and, once obtained, starts to listen for requests. To keep it simple and resource-friendly, it only supports two types of commands. It can either report the status of devices on the bus as seen by the master or request to change it. The exchange format between the server and controller are simple JSON objects with the least amount of decoration possible. Overall we keep the size of requests and answers below one kilobyte to be able to store the complete message in a buffer without wasting too much memory that could be used for other tasks. With that size, we are able to check the correctness of every message and return appropriate responses in case of missing parameters, for example.

The last important function of the master controller is the generation of the master shutter signal. While it only creates a sequence of short pulses, the time between them has to be exactly the same to avoid jitter in the frame rate of the captured video. Using a repeating hardware timer in the CPU in combination with an internal interrupt, the generated signal is absolutely precise with only very rare occasions where the signal is one CPU cycle late. With a clock speed of 120MHz, the deviation is measured in single-digit nanoseconds. Since our cameras can only reach 40 frames per second and the delay between the frames is never lower than 25ms, the jitter is negligible.

Using the REST API the delay between impulses can be set with microsecond precision to match the desired frame rate exactly. Even frame rates that result in periodic rational numbers for the delay, for example, 30 frames per second, can be represented with sufficient accuracy.

In the modules, the microcontrollers are configured as I2C slave devices. They receive the desired delays from the master for each camera. With these delays, they calculate a set of bitmasks that define when a camera shutter should be triggered after the master signal is received. For any step that contains a state change on the output ports, the mask with the new state is stored together with the step number in which the change occurs. When a shutter signal is received from the master, the client starts a hardware timer which creates an interrupt every $10\mu s$. The interrupt handler increments a step counter and checks whether the next change is due. If that is the case, the outputs are changed by writing the new bitmasks directly into the CPU registers responsible for the output pins and updates

the pointer to the next change. Otherwise, it does nothing and waits for the next interrupt. When the last shutter pulse is over, the hardware timer is stopped.

Since a CPU register in the microcontroller is only responsible for eight pins, setting the output pins directly via the CPU registers minimizes the difference between the first and second set of pins. Restarting the hardware timer for every instance of the master shutter signal reduces the influence of clock differences in each microcontroller by limiting the time in which the processors run unsynchronized to one frame duration maximum. Comparing the impulses from all modules after a relatively long frame duration of one-tenth of a second, using an oscilloscope with a sufficient time resolution, reveals that the time difference between the rising edges of the impulses is at most $10ns$. This is roughly one tick of the CPU's 120MHz core clock and the result of a slight deviation in the exact clock frequency of the different microcontrollers. In a system using 10-microsecond steps to set delays of multiple milliseconds, this deviation is acceptable and never led to problems, even when fast timings and high precision were required, for example in the HaToy scene (see Section 2.6.3).

To ensure that this precision is guaranteed, the microcontrollers disable the I2C buffers in their modules after they detect a master shutter pulse and the hardware timer was activated. Only when they do not receive a shutter signal for one full second, they reactivate the buffer and reconnect to the I2C bus of the system. When the I2C bus is disconnected, the microcontroller can use all its clock cycles to generate a precisely timed signal, and interrupts with higher priorities from the internal hardware, are minimized. The bus recovery feature in the buffer ICs ensures that all devices in the disconnected modules continue to work normally, even when the connection is disabled during an active data transfer. Should it be that the clock signal from the I2C master is missing for too long, the buffer generates a sequence of clock pulses followed by a stop bit so all devices on the bus can end their ongoing transmissions correctly.

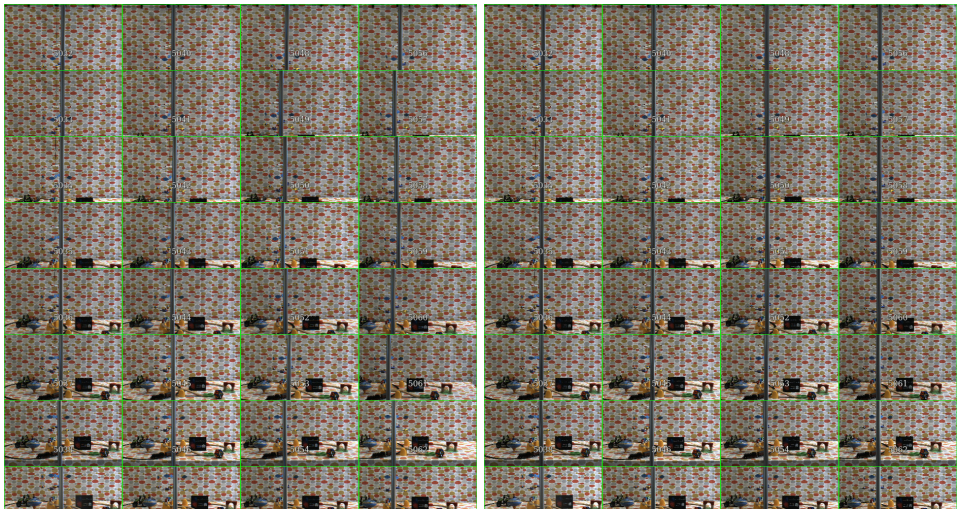
With all these optimizations for low memory consumption and load management on the microcontroller cores, the overall system is very stable and worked very reliably during all capture sessions. When more modules are added to the system, a microcontroller with more RAM would be necessary to store the status data of the new hardware. The footprint does not need to be compatible, as the master board has to be remade to accommodate the additional connections.

## 2.4.4. Repeating Tasks

During the operation of the array, there are several tasks that have to be performed at least once per shoot. Some have to be done manually while others can be at least partially automated.

### 2.4.4.1. Camera Mounting and Alignment

Firstly, it is necessary to create the camera layout by mounting the cameras on the frame. The correct vertical alignment is the simplest part because it is based on the horizontal bars in the frame, which are fixed on both sides. Careful measurement leads to near-perfect distance and parallelism to the ground. The horizontal distance is more complicated. Since they are only fixed using a single screw on the opposite end of the cameras, the final application of torque to the screw can cause a slight rotation in the camera arm and therefore, change the horizontal distance between neighboring cameras. Initially, this movement was supposed to be prevented by a plate at the end of the arm, but as described



(a) Misaligned cameras in the preview (b) Preview of a correctly aligned array

Figure 2.30.: Horizontal misalignment in the preview mosaic. When a straight vertical feature does not line up within the columns of cameras, the positioning of the cameras has to be adjusted.

in Section 2.3.3, this was made impossible by manufacturing inaccuracies. Even with the final version of alignment helpers, a slight rotation can not be prevented. The only way of detecting those is by checking the output of all cameras at once. By placing a long, thin, and straight vertical feature into the scene, any deviation on the form or rotation becomes visible as an offset between the cameras in a column of the array. Using a mosaic view, containing the preview streams from all cameras following the same layout as in the array, makes this check fast and convenient. With gentle force, the misaligned cameras can be brought to their correct location. In Figure 2.30 such a preview before and after correction of the error in horizontal rotation is seen.

The main complexity of this task lies in the creation of the combined preview. The central server was planned with enough computational resources for such a task. With the help of a shell script, a custom GStreamer pipeline is created. First, the preview streams from all cameras are extracted from their respective RTP streams and decoded. Then they are scaled to match the size of one tile in the mosaic, mirrored horizontally, given a thin border, and marked with their camera id. Those images are then composited into a single FullHD stream. For the last step, two versions exist. One that shows the composited image on the screen of the server, and a second version that reencodes the mosaic and sends it to a chosen computer on the local network. Depending on the complexity of the scene and the surrounding setup, either one of them is more practical to use.

The horizontal mirroring helps with coordination when the display is placed behind the array with the operator in front of it. In the flipped image, the operator's left and right are the same in the preview images, and the preview behaves more like a mirror. The added border and the camera id help to separate the views visually and to identify a certain camera when other adjustments are needed.

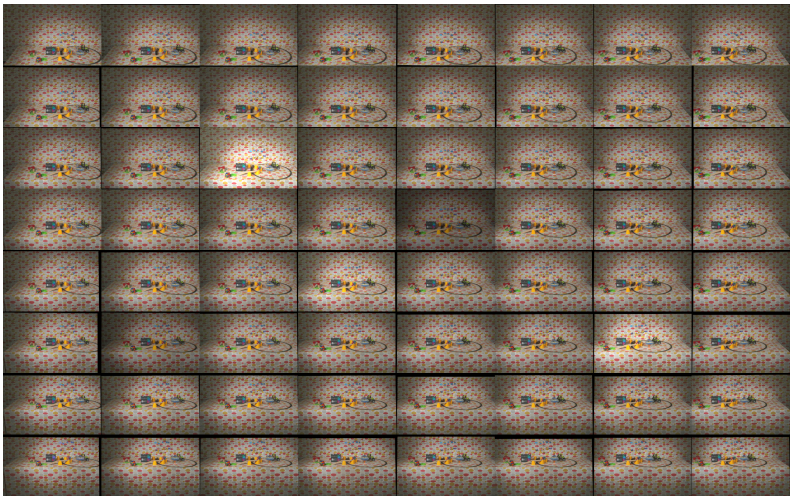


Figure 2.31.: Aperture deviations in the preview mosaic. Lower aperture number lead to a brighter image. Deviations in the other direction makes them darker.

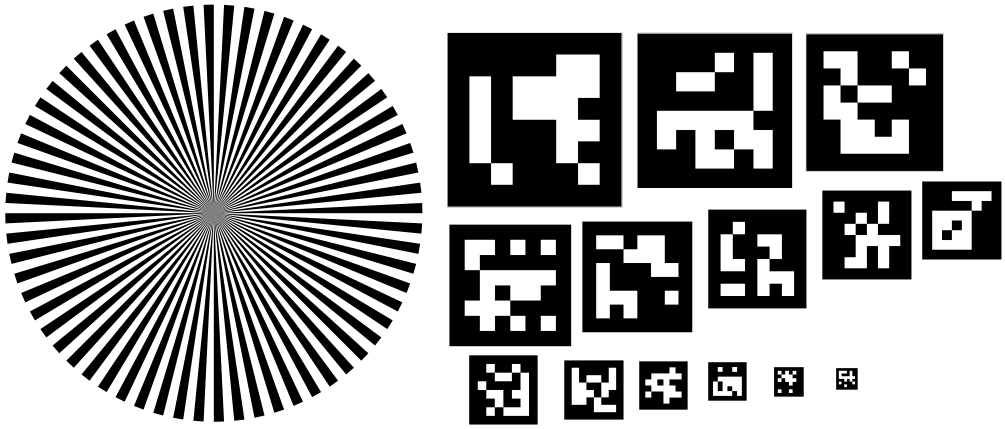
#### 2.4.4.2. Camera Lens Settings

Once the cameras are aligned, the parameters for their lenses have to be set. The aperture and the focus are set by rotating two rings on the lens and fixed using set screws. As there are no perceptible steps in the mechanism of the rings beside the end stops on either side and with only the markings on the lens's body as orientation for the operator, the settings can easily deviate from camera to camera.

The optimal setting for the aperture is mostly defined by the scene's extend in depth direction and therefore, the allowed depth of field as well as the combination of the available amount of lighting and the possible exposure time for each frame. Since the aperture setting directly affects the amount of light hitting the sensor in the camera, it's setting always influences the brightness of the whole image. As shown in Figure 2.31, those errors can be easily seen and recognized in the preview mosaic as a difference in brightness. Once identified, the apertures setting can be easily corrected. That can also be done by multiple people in parallel as long as they can see the preview screen and do not block too many cameras.

Related to the effects caused by the aperture is the influence of the exposure time. The maximum exposure time is fixed, based on movements in the scene, the allowed amount of motion blur, and the frame rate. The lower bound is only defined by the capabilities of the cameras. As it directly influences the amount of light collected by the sensor for one frame, it can be used to adjust the brightness of the image together with the aperture and the lighting in the scene. It can be often hard to detect over- or underexposed pixels in the frames by merely looking at them, especially with reflective surfaces that look different in different cameras. These areas lose some information because they cannot represent the scene's appearance properly and the missing details can not be recovered during postprocessing. For these reasons, they should be avoided at all costs.

To make their detection easier, a custom GStreamer plugin was created and added as an option into the preview pipeline. It can be configured with a lower and an upper bound for the allowed brightness values. When a certain number of channels contain values outside those bounds, they are replaced with an animated striped pattern in obvious colors to



(a) Siemens star for focus evaluation, generated with the code from [78].

(b) Markers for automatic focus evaluation.

Figure 2.32.: Different methods for manual and automatic focus evaluation

indicate danger for over- and underexposure respectively. Since these patterns are usually surrounded by very bright or very dark areas, close to the set bound, the marked areas are clearly visible, even in the previews. With that plugin, the danger of extreme color values in the recordings is significantly reduced because problems and critical areas become very visible during the setup phase in which changes to the scene to remove their causes are still possible.

The focus setting of the lens is more complicated. Firstly, the focus does not affect the whole image equally. Depending on the aperture setting and the scene geometry, only parts or even nothing at all can be out of focus when the setting is wrong. Additionally, the precise correct focus can only be determined by watching the camera feed on a pixel level. Even tiny amounts of blur caused by the focus setting can make the recorded data useless for further processing. Due to the required precision, it is not possible to use the preview mosaic while setting the focus. For easier determination of the current status of the focus, we used a high-quality printout of a Siemens star (see Figure 2.32a), whose converging lines in the center never touch. At some level, every printer or camera trying to reproduce the pattern will show aliasing patterns near the center. The area influenced by these aliasing artifacts becomes smaller when the number of accurately represented points in that area increases. When the area is out of focus, even slightly, neighboring points start to bleed into each other and the aliased area grows.

With such a pattern in the scene, where the center of the focal plane is supposed to be, the quality of the focus setting of a camera can be objectively determined by measuring the size of the aliased area in the pattern. By varying the focus around the current setpoint and watching the behavior of the pattern, it can be verified whether the setting is optimal or not. To do this properly, we still require a full-size view from the camera being set up, which means only one person can focus one camera at a time. Even for an experienced operator, it takes a significant amount of time to configure all 64 cameras correctly.

To make this process faster and parallelizable, we devised an approach to measure the focus quality automatically. By trying to detect markers of different sizes in the desired focus plane, from bigger ones (detectable from far away with only a decent focus setting) to smaller ones (requiring a very precise focus and sometimes even a small distance from the camera), the focus quality is gauged by the smallest pattern a camera can still identify

correctly. With the patterns from Figure 2.32b placed next to the Siemens star in the scene, we can combine both approaches. An optional custom GStreamer plugin in the preview pipeline detects the patterns, and depending on how many are found, indicates the quality of the focus in the pattern's region by overlaying a bar over the top of the frame. A longer bar means a better focus. Because the resolution of the cameras is limited and also due to the deliberately small size of some patterns, certain patterns can never be seen correctly. To still give a meaningful indication of the best focus, the plugin determines the maximum for the bar dynamically. A moderately quick sweep over the focus range of the lenses sets the maximum for the current setup and allows to dial in the best focus afterward. Due to the highly visible indicator on the top of the preview images, this can also be done using the preview mosaic and by multiple people at once.

After several productions, we found that the precision of the result from the marker-based approach is very close to that of the manual process. Only in very rare cases did it need to be adjusted, and those cases could be accredited to problematic lighting conditions with overexposures in the patterns.

#### 2.4.4.3. Bulk Data Transfer

After a scene was captured, the frames reside in the SSDs of the camera units. Before transferring them to permanent storage, the operator has to make sure that the expected number of frames has been captured by every camera. Differences in the number of captured frames are not common and if they occur, they are a clear sign of heterogeneous configurations or hardware failures. Such sequences can be stored, but their use is limited because the number of available cameras changes over time. Repeating the capture is usually the better solution.

These tests are performed as part of the script responsible for transferring the data to the storage cluster. First, it checks whether the expected number of camera units is active. Second, the number of images in the cache of each unit is compared. When the numbers are equal, every camera unit starts a set number of threads doing the following: listing all captured frames, distributing them between the threads, generating a unique object id for every frame, and finally sending them to the storage cluster.

The data is not removed from the caches, in case the transfer is aborted at any point due to an unexpected hardware failure. In such cases, the transfer could simply be repeated later, as the cached data stays valid until it is overwritten or deleted.

At this stage after recording, only the raw sensor data is in storage. In order to fully process it into usable images, as shown in Section 2.5, a set of images from the capture session has to be processed even though no exact correction data is available yet. Otherwise, it is not possible to provide the algorithms that calculate those correction parameters with appropriate input data.

The operator has to choose the frame to be extracted for calibration. The extraction script first generates the object IDs to be downloaded from storage. Due to their uniqueness and predictability, they can be created using the internal scene name, camera IDs, and frame number. Before writing the frames to a folder, the original camera id is converted from our internal, hardware-based, column-first system to the more common row-first order. Once the data is downloaded, a custom program is applied to them which adds minimal metadata to the raw sensor data and stores it as a digital negative (DNG) file. That file is used as input for the dcraw [79] which offers a variety of demosaicing algorithms to recover full-color images from the raw sensor data. For exchangeability reasons and to save disk space, the resulting color image is run through a minimal wrapper of the OpenEXR [80]

library to create losslessly compressed EXR files with the full-color frames.

Based on these frames, the algorithms in Section 2.5 calculate the ideal transformations, both for the image's geometry and colors. The results are saved in their original format in the redundant storage next to the frames of the scene they were calculated for. It is important to note that the order of the camera IDs has to be converted back, to make them consistent with the one used for the raw data frames.

With the parameters for geometric and color correction, all frames can be processed into their production-ready version with equalized colors and perfect camera alignment. While the main steps are described in Section 2.5, the preparations for these steps often require more effort than anticipated. Due to the constant evolution of the tools in use, we chose not to install or activate them permanently on the camera units.

Before every processing round, the required tools and their configuration files are downloaded into a temporary folder. If required, environmental variables are set temporarily during the execution of the processing pipeline. With such a temporary setup, we do not have to worry about uninstalling older tools when they are not needed anymore. After a reboot of the computers, the temporary folder and any remnants are no longer present, which also minimizes version conflicts and keeps the environment clean. We are aware that those features can also be achieved by running the processing steps in a sandboxed container, but creating a customized container for every iteration of every tool in the processing chain would have been unreasonable.

Each camera unit chooses the ID of a camera whose frames it is going to process. Usually, this is based on its position in the array and equal to the id used while uploading the data from the cache. Then each camera unit downloads the color and geometric calibration data for the camera and scene it is about to process. After that, it creates a list of all frames available in storage for these parameters. Depending on how many cores the steps in the pipeline can use on average, each camera unit starts multiple threads to process more than one frame in parallel. Each thread takes a frame's name from the global list, downloads it from the storage cluster, runs it through every step of the processing pipeline, and uploads the result to the pool of processed data in the cluster. The final result is also stored as a preview JPEG with half the resolution, next to the high-quality data. In case a previous version already exists in the cluster, it is overwritten, as it is assumed that a newer version is always of higher quality and therefore, preferred.

The material captured by the array created some interest in the research community and our partners from the SAUCE project. Making the data manageable was therefore very important. As presented in Section 2.3.7, we reserved 20TB of storage in an existing cloud storage setup on campus which we could use without restrictions. Since we were able to manage the data via internal SSH access, uploading the material as single frames would have been possible, but the workload of downloading hundreds of thousands of files using the web interface would have lessened the usefulness of the data. Creating a single file containing all the data as the other extreme would mean that any interested party would need to download more than one hundred gigabytes of data, which then needs to be extracted from the archive first. The optimal solution lies somewhere in between. Splitting the data into archives with multiple frames from all cameras with a size between 10 and 20GB was found to be the best solution for us. To allow for previews of the material, we also provide archives containing the preview JPEGs instead of the full resolution HDR images in OpenEXR format.

The automation of the archive creation was justified by the fact that it had to be repeated every time changes in the pipeline resulted in better results. The script downloads the data

for every archive into a temporary ramdisk on the central server and compresses them into an appropriately named zip file on its hard drive. Using a ramdisk for the intermediate storage was justified by the RAID's inability to handle the large number of small files quickly enough, as discussed in Section 2.3.7. Once the archive is completed, the frames are deleted and the contents of the next archive are downloaded. This process continues until a frame number is reached that is above a configurable upper limit or until none of the cameras provided any data. Later, the process was streamlined such that the data for the next archive could already be downloaded while the previous archive is still being compressed.

The uniqueness of the camera array requires a significant quantity of custom software to function properly. Distributing new versions of tools and plugins as source code to be compiled on the machines they are needed on is quite inefficient due to the high number of devices in the system. To get around this, a chain of continuous integration pipelines surrounding our version control system is employed. In the beginning, a Docker container with all required packages and non-standard components used in the camera arrays environment is created and stored locally. That container is used to compile the code of different projects into versions that can be executed on the camera units. For distribution, the results are wrapped into valid Debian packages[21] and uploaded to a locally hosted repository. From there, all devices in the camera arrays network can simply update their tools using their internal package manager orchestrated by the cluster control system. Using such a system, we also create installable packages from third-party libraries which only come as versions with install scripts, such as the drivers and tools from the camera manufacturer.

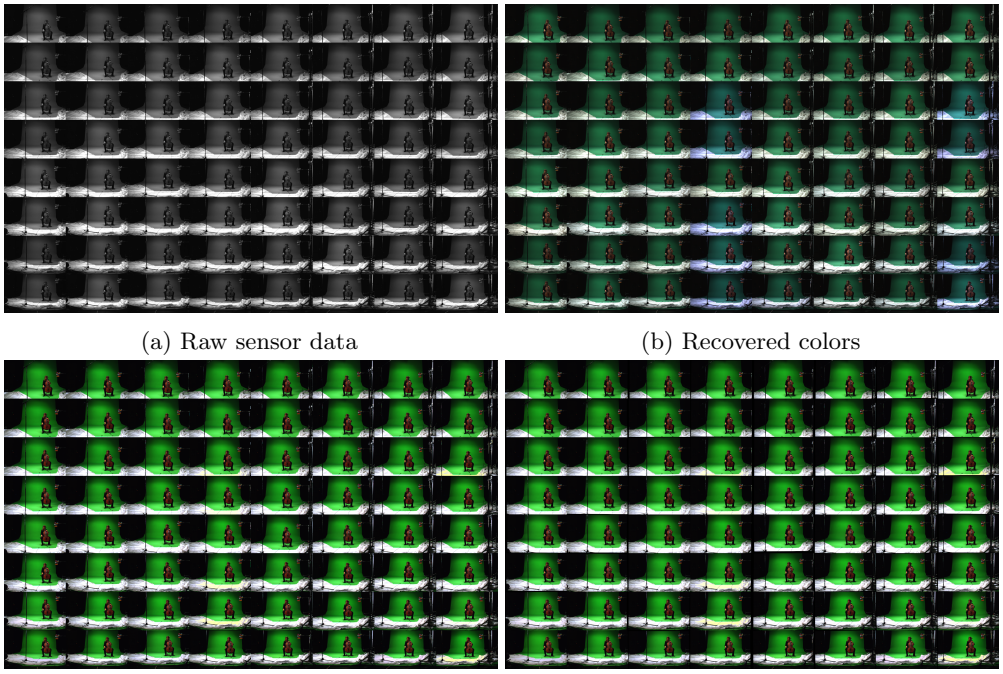
## 2.5. Processing Pipeline

The processing pipeline is one of the most important parts of the array because it contains all steps to provide high-quality color frames from raw sensor data and is used to correct slight errors in the camera configuration and layout, to provide frames from a horizontally and vertically aligned array with an equal color response from every camera.

The major complexity of this process lies in the calculation of the parameters for the processing steps. For these calculations to succeed, they must always take the state of the complete array into account and be aware of multiple constraints.

After our own approaches for geometric calibration did not yield results of sufficient quality to satisfy the requirements given by the SAUCE project partners, we restarted our search for better published calibration approaches. While the search did not provide many promising candidates, our project SAUCE partner from Brno University, namely Marek Šolony, modified one of his algorithms to work with our array. The as of yet unpublished algorithm he provided uses bundle adjustment in combination with his own SLAM framework [81] based on the work by Xu *et al.* [82] to solve our calibration optimization problem much more efficiently and precisely. The close contact with the creator of the algorithm allowed for multiple iterations of the algorithm to be tested and improved in a short amount of time. Several constraints, including the ideal array layout, the rectification in column and row direction as well as the minimization of data loss due to the required geometric transformations, were added or emphasized. Assumptions about the scene and its relation to the array, such as the main background being parallel to the camera plane, were relaxed to support more complex arrangements. In contrast to our marker-based approaches, it

[21] `https://ubuntu.com/server/docs/package-management`



(a) Raw sensor data (b) Recovered colors

(c) Corrected colors (d) Rectified frames

Figure 2.33.: Image quality after each of the processing steps.

employs different feature detection algorithms like SURF [83] and SIFT [84] to generate its calibration points.

Due to its dependence on a fairly modern CUDA-capable GPU and its UI, it is not yet integrated into the automatic workflows on the arrays server, but needs to be executed manually on a different machine. This means for each scene the chosen calibration frames must be transferred to the second computer and processed there. Only then can the final calibration parameters be transferred back and uploaded to the storage cluster.

Color calibration or equalization is required for two reasons. Due to slight manufacturing variations, no two camera sensors ever have the exact same color response, and the same applies to the transmission behavior of lenses. Choosing cameras and lenses based on their binning was not possible due to our budget constraints, therefore making it necessary to do our best with the hardware in our possession. The second reason was a software error in the interface that sets up the internal operating parameters for the cameras, which existed until after the second major recording session. This oversight caused only half of the available white balance parameters to be controllable, making it possible that some cameras had a difference in the white balance based on the first automatic setup after the startup of the camera. For color analysis and correction, we ensured that a MacBeth color pattern [85] was visible to all cameras, at least in some frames of the scene.

Our first attempts at color correction using linear regression for each channel to match the colors of the captured color patterns with their intended values produced a slight improvement but were not sufficient for final production. With the help from Mairéad Grogan at Trinity College Dublin, we were able to use her color transfer algorithm [86] to equalize the colors between all cameras. While the color alignment worked correctly out of

the box, the part which automatically detects patches of colors in all cameras that should be equal lacked some robustness and caused wrong final results. The algorithm requires representative color values from a MacBeth chart for each camera in addition to sample frames to be corrected. Detecting the pattern and selecting the colors within them, can be mostly automated, but still requires some manual supervision and correction to be accurate. For that reason, the process is currently not fully automated and has to be performed manually. The resulting 3D cube lookup tables [87] contain 32 entries in each direction of the color cube for a total of 32768 key points. This precision leads to significantly better results than our approach. After calculating the correction values, the lookup tables are stored with the frames in the storage cluster.

As soon as the correction parameters for color and geometry are uploaded to the storage cluster, all frames can be processed. The progression of a single array frame going through the processing pipeline is depicted in Figure 2.33.

Starting with the grayscale raw sensor data in Figure 2.33a, the first operation is to recover full-color images using demosaicing. This has to be done first, because most image operations can destroy the Bayer pattern, thereby making a clean color reconstruction impossible. The current compromise between quality, computation time, and practicability is the dcraw tool, which implements the Adaptive Homogeneity-Directed (AHD) algorithm [88]. While not perfect, it prevents many errors that occur when the camera-internal process or simpler algorithms are used. In its results in Figure 2.33a, the problem with the difference in white balance becomes immediately visible. Most images look normal but in the center and on the right border, there are a few cameras that show a clear blue tint. Those deviations are fixed by the color correction step afterward. Because the cube files can be considered to be a professional format, most common image manipulation tools are not able to handle it properly. The only working, non-commercial solution we could find is a wrapper of OpenColorIO[22] in the OpenImageIO[23] toolbox. As OpenColorIO is intended to be a tool with system-wide influence (e.g., to correct the color representation on the connected monitors) it requires a fairly complex configuration file and environment variables to work. For this, the system is prepared by the cluster control system. After the setup, the application of the lookup table boils down to a single call to OpenImageIO. Figure 2.33c displays the uniform colors after correction. The images also appear to be far more vibrant than before. That change happens because the colors in the images are not just aligned to each other but also to the intended values of the color chart.

Subsequently, the geometry of the images is corrected. Due to of slight differences in the camera's mechanical alignment from their optimal position, objects can have a horizontal offset within a single column and vertical offsets in a row. The geometric correction must virtually move the cameras back onto a perfect grid. The extrinsic camera parameters determined earlier solve this problem, together with any relative rotation and distortions caused by the camera's lens, in a single operation. While the changes of this step are hard to spot in Figure 2.33d, most algorithms that use the images as their input benefit from them.

In the last step of the pipeline, the frames are converted from their uncompressed PPM format to OpenEXR with lossless compression. To have full control over the available parameters and to make sure an appropriate amount of bits per pixels are used, a custom program making use of the OpenEXR library was created for that task. Since this step does not change the images, its result is not included in Figure 2.33. This format, which can be

[22] `https://opencolorio.org`
[23] `https://github.com/OpenImageIO/oiio`

used in nearly all tools dealing with high-quality images, is then uploaded to the storage for processed frames.

## 2.6. Productions Using the Camera Array

A novel capture device like the 5D-capable lightfield array discussed here is only worth something if it is actually used. In this section, we present the major productions for which the array was used, beyond the smaller internal shoots for testing, refining of processing steps, and algorithm tests.

### 2.6.1. Lightfield Elements

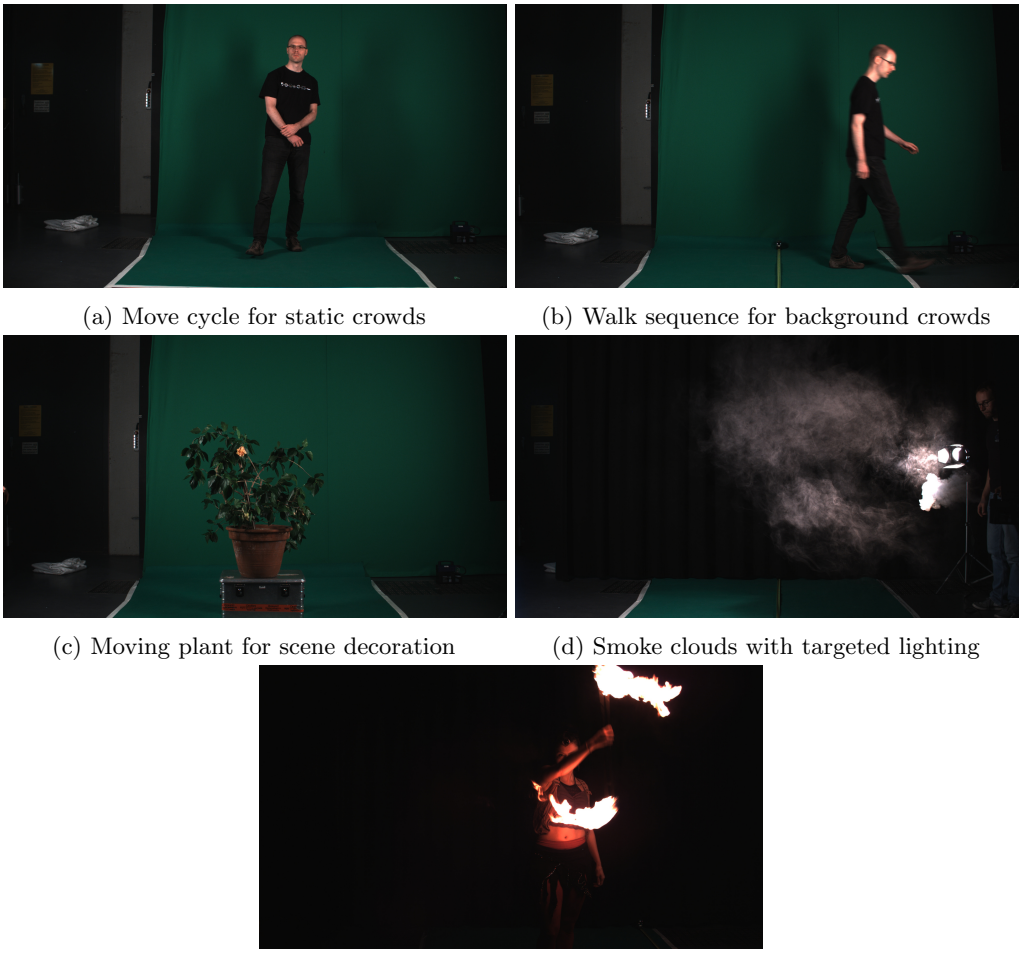
The Lightfield elements shoot was the first professionally executed shoot and simultaneously the first use of the array outside the lab. It took place on the premises of the Filmakademie Baden-Württemberg in one of their studios. The main goal was to create several move cycles, usable for background crowds, scenes with volumetric effects, and objects usable as background props in larger scenes. Figure 2.34 shows exemplary views of a selected camera for the captured sequences. The first two were intended to gauge the usefulness of lightfield footage for the creation of generated crowds in CGI when the amount of perspective shift is limited. It has the potential to make the production of such scenes with many extras much cheaper and faster to produce if short movement cycles recorded as lightfields can replace manually animated CGI models in background crowds or moving decorations. The third and fourth sequences explore a similar aspect for objects with a very complex structure or motion, such as plants and volumetric effects like smoke. The last sequence depicts the performance of a local fire dancer at night in near darkness. This was mostly an internal test to see how scenes with a high dynamic range need be set up and managed.

The complete material from that shoot is publicly available through the website of the EU-funded SAUCE project which these evaluations were a part of (`https://www.sauceproject.eu/Downloads`). This dataset also includes all calibration images captured in between the scenes.

The feedback for the captured material was mixed and highlighted a number of shortcomings the system still had at the time. Setting it up and preparing it for transport took a long time, even with multiple people working on it at the same time, mainly due to the complex storage system for the cables connecting the modules to the cameras. For subsequent shoots, this process was streamlined significantly.

The shoot also demonstrated that the state of the geometric calibration procedure at the time was not sufficient for the use cases the material was intended for. This is also the main reason why the material was never properly rectified. With increased efforts and focus on the calibration, this problem was partially resolved until the next shoot but the best solutions were only available much later. The green screen background in some and the dark background in other scenes do not give the calibration algorithm enough features to use and forced a redesign of our calibration material.

A hardware problem in the RAM of one of the camera units means some sequences do lack the footage from one camera after a certain number of frames. The exact source of that problem was pinpointed later and also caused more strict checks to be enforced in the recording procedure. This has also lead to the addition of the storage cluster (Section 2.3.7) to the array because the transfer times from the cache to permanent storage quickly started to become unacceptable within a busy day of production.



(a) Move cycle for static crowds

(b) Walk sequence for background crowds

(c) Moving plant for scene decoration

(d) Smoke clouds with targeted lighting

(e) Performance of a fire dancer at night

Figure 2.34.: Impressions from the different scenes included in the LF elements.

Figure 2.35.: Camera views from the different recorded voices in the Unfolding scene.

Last, the overexposed areas in the smoke and fire dancer sequences inspired the creation of the exposure overlay plugin, because the brightness ranges were tested before the final recording and looked fine in the preview.

The improper calibration, together with the fact that most tools for lightfield processing in professional toolchains were just reaching the prototype stage, meant that their potential for professional production could not be evaluated then, and that aspect was left open until later.

### 2.6.2. Unfolding

The *Unfolding* shoot was the key experimental production in which to emphasize the capabilities and possibilities of lightfields for professional post-production in the SAUCE project. The shoot was planned by the Filmakademie Baden-Württemberg, directed by the Director of Photography Matthias Bolliger[24] and starred the cellist Isabel Gehweiler[25] in the leading role. The shoot took place on the premises of the Saarländischer Rundfunk, a major local television and radio broadcaster, in one of their studios with the support of a full team of light and sound engineers.

For the shoot, the cellist composed a musical composition piece with four parts. She played every part separately while being filmed from different locations and angles. In post-production, the different instances of the cellist were to be combined in a single scene, showing off several effects made much simpler and more efficient by using lightfields instead of conventional single-camera footage. Figure 2.35 presents one camera's views of the different parts of the song we recorded.

[24] https://www.matthias-bolliger.de
[25] https://www.isabelgehweiler.com

The two versions of the end result show focus effects nearly unachievable with conventional means such as a focus plane following the moving bow in the hand of the cellist. Other features include the seamless combination of light field material with computer-generated objects and effects. Both versions are available on YouTube[26] [27] for normal monitors and in an adapted version for the LookingGlass[28] holographic display[29].

The production, the capturing process, and the camera array were featured in multiple industry-specific articles [89, 90, 91, 92] showing the importance of and interest in our work by the film-making industry. In addition to these publications, it triggered the interest of multiple German TV programs which covered the functions of the array and its possibilities on national TV.

### 2.6.3. HaToy

HaToy was an internal production to demonstrate the benefits of having an array with a tightly controlled timing plane for the shutter release. The scene consisted of a collection of toys and devices with very different moving patterns and speeds. In Figure 2.36 we can see the different components. The two trains drive with medium speed on their tracks while the mobile above rotates very slowly with 4 to 5 rotations per minute On the right side, the spinning top rotates at a rate of approximately 1000 to 1200 rounds per minute. The CD drive on the left is regulated at a speed of 2400 rounds per minute. With a capturing frame rate of 40 frames per second, this is equal to exactly one full revolution per frame.

To reduce the amount of motion blur in the images, the exposure time is set extremely low to $350\mu s$. Therefore, the scene had to be illuminated by the equivalent of over 5000 Watts of Halogen spotlights.

In total, we captured a sequence of traditional 4D lightfields and several versions of 5D lightfields in which we tested multiple ways to distribute different shutter timings over the array in so-called subframes. For each version, 4, 8, 16, or 64 different delays were equally distributed over the camera layout, such that ideally any square subset of the array with an appropriate number of cameras includes information from all subframes. While in most parts of the scene, these delays do not make a difference, the benefits of the sub-framing become obvious when looking at the CD drive.

In Figure 2.37a, we can see that the cameras show exactly the portion of the CD rotating in the drive. Since its speed is synchronized with the frame rate of the cameras, the following frames show no rotational change. When reconstructing the complete texture or the exact movement of the CD is the goal for this lightfield, the uniform sampling would not allow this. Even, if we assume that the rotation is in sync with the shutter, the direction of the rotation, the other half of the CD, and whether the CD rotates once or multiple times between frames, cannot be determined.

By separating the lightfield frame into 4 subframes whose shutters are distributed over one frame duration, the situation changes. Figure 2.37b shows the output of the same four cameras with the CD in four different stages of rotation. When the delay for each camera is known, one can easily determine the rotation direction by tracking the CD's texture between the subframes. In addition, every part of the CD is visible in two subframes, which makes reconstructing the complete texture from a single frame possible.

[26] Unfolding - `https://www.youtube.com/watch?v=UnsmKQjO4ro`
[27] Unfolding 2.0 - `https://www.youtube.com/watch?v=OlHnDOuf2BA`
[28] `https://lookingglassfactory.com`
[29] Unfolding 2.0 Looking Glass Edition - `https://www.sauceproject.eu/Downloads`

Figure 2.36.: HaToy scene for the demonstration of the usefulness of a precisely controlled de-synchronized shutter timings. Fine-detailed rotating mobile on top, slow moving trains below. Rotating CD drive synchronized with the camera's frame rate and an active spinning top.


(a) uniform sampling


(b) 4 subframes

Figure 2.37.: CD drive seen by the center cameras in different sampling modes. With uniform sampling, only a portion of the CD is visible. Subframes show a different section in each camera, allowing for a full reconstruction of the CD's texture.

Obviously, sub-framing reduces the number of cameras per subframe and therefore, the number of rays per time instance. For the moving parts of the scene, this trade-off between time resolution and the number of rays has to be optimized on a per scene basis. In static portions of the scene, the sub-framing can be mostly ignored, because the delay in the shutter does not change the content of the image there.

Optimizing this and exploiting the additional information gained by subframes is still an open research question. The FiDALiS project aims to solve this but at the time of writing, is still working on integrating sub-framing into the mathematical theory upon which all lightfield algorithms are based.

Most approaches for lightfield compression consider only fully synced cameras since the current state of mathematical description does not support known time offsets between cameras. One of the first approaches to take known time shifts between the camera shutter into account for compression was presented by Hariharan *et al.* [93] in 2020.

## 2.7. Conclusion

Even though there were a few problems surrounding the camera mounts and calibration process in the beginning, overall the build of the 5D lightfield array was a full success. With its capability to have different configurable delays for every single camera, it is currently unique in the world. Of course, it is clear that uniqueness is not necessarily an indicator of quality, but as part of the SAUCE project, we managed to produce some very convincing results proving the array's capability to create standard 4D lightfield images and videos.

The use of external algorithms from experts in their respective field improved our own attempts at color and geometric calibration by a huge margin and now permits us to make nearly all recordings into useful lightfield material. Some material produced by the array and the array itself has been featured in a significant number of industry-specific magazines and even in national television programs. The capabilities have also been confirmed by peer-reviewed academic publications and talks [7, 8, 9, 10, 11, 94].

While we have shown that the array is also capable of capturing 5D lightfields with up to 64 subframes, whether it is precise enough remains an open question. A lack of support for subframes in the mathematical theory currently prevents a proper evaluation of this feature. However, merely by looking at the material we produced, it becomes clear that there are benefits to having this time-controlled shutter plane.

Since lightfields are currently a hot research topic with open questions and improvement potential in many areas and 5D lightfield theory still is in its infancy, it would be accurate to say that the array will continue to be used, even if only to verify that algorithms and approaches work with real-life examples.

### 2.7.1. Future Work

With a stable frame, working calibration, and sufficiently good post-processing algorithms, the array is able to fulfill its basic functions. However, there remain several open issues which are yet to be resolved.

First and foremost, there are some usability issues for anyone without in-depth knowledge of the internal structure of the array's systems. Currently, to access all functions and properties of the hardware and cameras, two separate interfaces are required. Both are web-based and hosted on the server. The first was mainly created for testing and debugging

in the early days of development, while the second was built later with more comfort features in mind. They already share most of the functionality, but not all. Certain low-level settings can only be made by the old UI while setting others is far more comfortable in the new UI. The newer UI also gives more detailed status information about the camera units and the rest of the system.

For better usability, all functions only available in the prototype UI should be added to the new one. It should also be considered to make some scripts accessible via the web UI, which currently have to be called via the console on the central server. With such functionality, the interface can be used to guide a user through the whole process of capturing a scene, from switching on the modules over camera alignment and lighting setup to processing the final frames.

The calculation of the correction parameters for color and geometry must be calculated on computers that are not included in the array's system. That dependency makes version control and software management much harder and violates the independence property of the array. By porting the algorithms into a form that can be executed on the central array server, that can be solved. Ideally, the manual required steps would be somehow forwarded to the web UI, such that all steps in the recording process can be done using a single interface.

The currently used demosaicing is based on a fairly outdated approach. It produces acceptable results but better algorithms are already available. Even though ultimately we would like to use our array-specific solution (see Chapter 3), it may take a while until it becomes universally usable. In the meantime, algorithms like ARI [95] or DMCNN-VD [96] could be used, provided their results can be replicated. The main difficulty in this endeavor is the fact that ARI was only made available as MATLAB code and the authors of DMCNN-VD did not provide the datasets used for training or the final weights of their trained network.

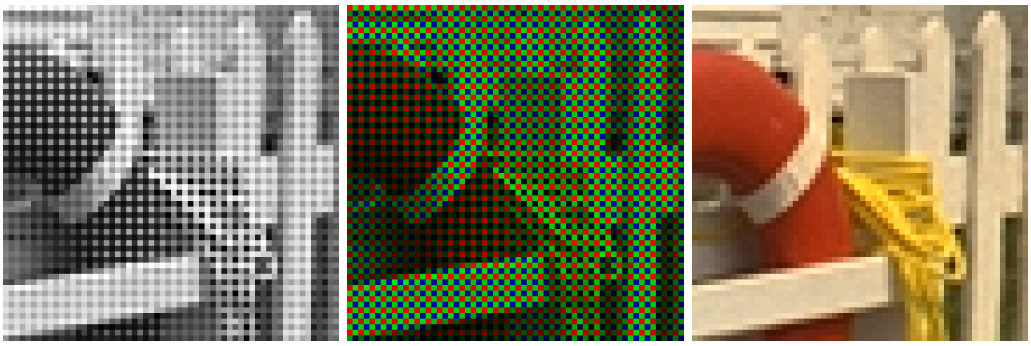
# 3. Array-Specific Demosaicing

Demosaicing or debayering is important for any image capturing system because it converts the monochrome output of a CCD or CMOS sensor into the colorful images we expect from a digital camera. Cameras with separate sensors for all color channels are very expensive and much larger than commonly used devices because they require an intricate system of prisms to split the colors into red, green, and blue and project them on three distinct sensors. On the other hand, single-sensor cameras use a color filter array (CFA), also called Bayer filters, in front of their sensor, which only allows a single base color to pass on to each pixel.

Figure 3.1 shows a simulated example of how the resulting data looks. The original color data is taken from a high-quality scan of an image captured on traditional film and provided in the Kodak Image Suite [97]. Removing the color information, following the pattern of the CFA, leaves only the data which would be captured by a single sensor camera. In Figure 3.1a, some colors can already be guessed by looking at the pattern in the pixels in some areas. When the pixels are assigned to the color channels they belong to, the image becomes clearer but it has a bias towards green. This bias appears because common CFAs include two green pixels for every single red and green pixel. The filters are built in such a way because the human visual system is more sensitive towards green than towards the other colors.

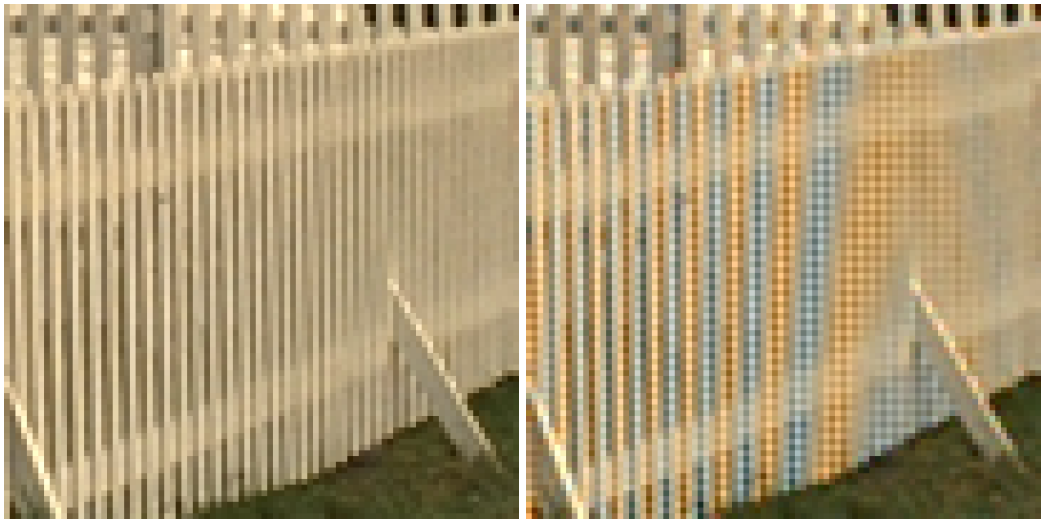
## 3.1. Basics

Demosaicing is the process of filling the gaps in the color information to get full-color images. While downsampling the image in such a way that every new pixel contains some information from all three channels, the reduction in image resolution and detail is not acceptable in most cases. Interpolation of the color data between the given data points



(a) Raw data (b) CFA colored data (c) Demosaiced data

Figure 3.1.: Example of raw sensor data created from picture 19 of the Kodak Image Suite [97].



(a) original image (b) demosaiced image

Figure 3.2.: Example of common demosaicing artifacts on picture 19 of the Kodak Image Suite [97]. Orange and blue discoloration and checkerboard-like patterns in the same regions.

maintains the original resolution but can lead to several types of artifacts, two of which are visible in Figure 3.2. The first error consists of areas of color (often light blue and orange), where there should be none. Those occur in image areas with high-frequency patterns, where edges in the texture are positioned in the CFA pattern such that the red and blue contents in neighboring pixels are overestimated. The second class of common errors is the zipper effect which places pixels from straight edges into alternating rows or columns, transforming straight lines into sequences of pixels on alternating sides of the real line.

Better and more complex algorithms reduce the amount of errors from demosaicing [98, 99, 100] but none of them are perfect. Most new algorithms rely on machine learning to improve the color reconstruction, either by creating a dictionary in which they search for the perfect solution for the current image section or use large convolutional networks to determine the missing colors based on their neighboring pixels and information learned during training.

The size of the input for convolutional networks must remain the same for all runs of the network. Since raw images can come from a variety of cameras and therefore, a plethora of resolutions, a solution is needed. Training the networks for all possible image sizes is impossible. Setting the networks up for the largest input images and padding or scaling smaller images to fit that size grows the networks to sizes impossible to manage. The problem is solved by splitting images to demosaic into small tiles, usually around 50x50 pixels in size, which are fed into the networks one after the other, and later the results are concatenated to form an image of the original size.

The demosaicing quality of the captured frames in the array has to be as high as possible at all times. Otherwise, the algorithms working with the material can run into problems when the data of the images is unpredictably changed by artifacts caused by the demosaicing algorithm.

Figure 3.3 shows some of the remaining errors in a frame from the Unfolding scene

Figure 3.3.: Frame from the Unfolding scene zoomed onto the strings of the cello with visible discoloration artifacts.

(presented in Section 2.6.2) after being demosaiced using the AHD algorithm [88]. The fine details of the cello strings shine in all colors of the rainbow instead of silver. Some green areas are expected when the green screen in the background is reflected in the shiny silver strings, however, all other colors should not be present.

Since every demosaicing algorithm working on a single image literally has to guess the missing color values, they will always create artifacts in cases where small details of specific colors fall in between two samples of a color on the sensor. By learning how to predict those small features from other images using machine learning, those situations can be reduced, although they will never be completely removed as the correct data values depend on the specific sensor and the captured scene. Since two images will never be exactly the same due to camera noise or minuscule movements of the camera or objects in the scene, finding an exact counterpart in the learned data is near impossible and therefore they still produce slight color deviations, even though these deviations are not as visible.

## 3.2. Concept

In the work of Alexander Blatt [101], we explored a novel solution that can reduce the amount of estimated data in multi-camera systems. It aims to exploit the overlap between different cameras to fill the missing color data in one camera from the data in neighboring cameras. To make it work, the correspondences for the current pixel have to be found in neighboring images, as shown in Figure 3.4. Once located, it depends on which area of the color filter it is in and whether the data is useful for filling in the data in the current image or not. In this simplified example, the chosen point in the center image only has data for the blue channel. With the matches from the top and left neighbor, the missing data for the red and green channels can be filled. In a practical application, the additional data

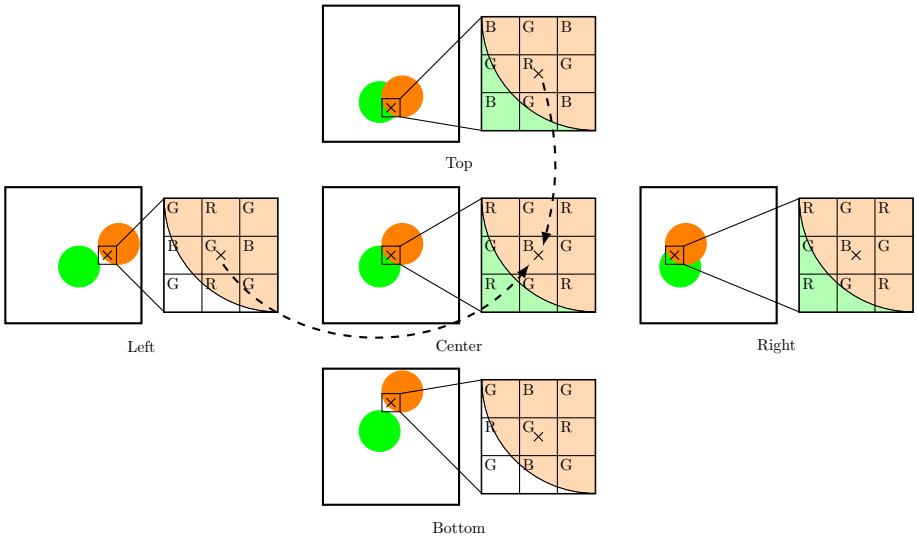



Figure 3.4.: Simplified example of our demosaicing concept for reconstructing the full color in the center image. The magnified parts of the camera views show for which color channel each channel pixel has data in the top left corner.

from the bottom and right samples could be used to reduce camera noise or other natural degradation effects. However, in this simple example, they are ignored because the data they offer is already present.

At first glance, the problem to solve seems to be quite similar to what we do for the view interpolation in Chapter 7. However, it is rather more complex, mostly due to where it is positioned in the overall lightfield pipeline and the quality and state of the material that comes with it. The view interpolation sits right at the end of the pipeline, which means it receives color-corrected and rectified full-color frames as inputs. This situation allows assumptions about the geometric relation of the images to be made and it can rely on the color response of all cameras to be the same. On the other hand, demosaicing has none of these benefits. As the first step after capturing, it has to cope with incomplete uncorrected color data and unknown translations, rotations, and lens distortions. While color calibration results could be applied to the partial color information, any geometric correction for the rectification distorts the color filter patterns on the pixel level. When the color information is smeared over multiple pixels or moved, it becomes much harder to relate a certain pixel to a color channel. Therefore, we try to avoid geometric corrections before the demosaicing. Wronski *et al.* [102] show that such an approach can work, by combining multiple raw images from a mobile phone into one image with a higher resolution.

The high complexity of the problem and its similarities with depth estimation, for which several good deep learning-based approaches exist [103, 104], led to our decision to create an approach based on deep learning. The following sections describe how the approach was designed, trained, and evaluated.

## 3.3. Network Architecture

The architecture of a neural network is, apart from the choice of the training dataset, the most important factor for its performance. As there are near infinitely many different

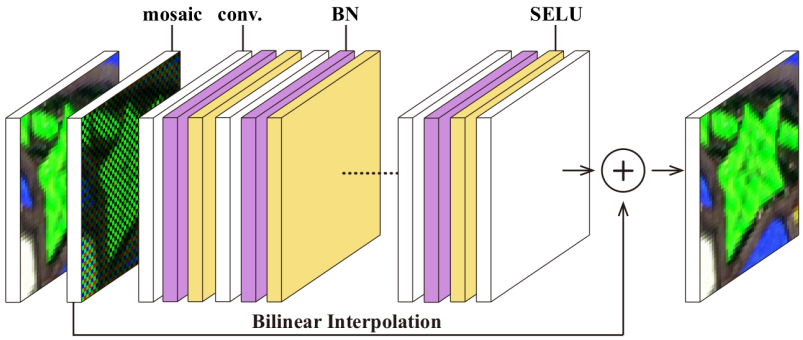



Figure 3.5.: General structure of DMCNN-VD [96] featuring 20 batches of convolutional layers and a parallel traditional biliniear interpolation step.

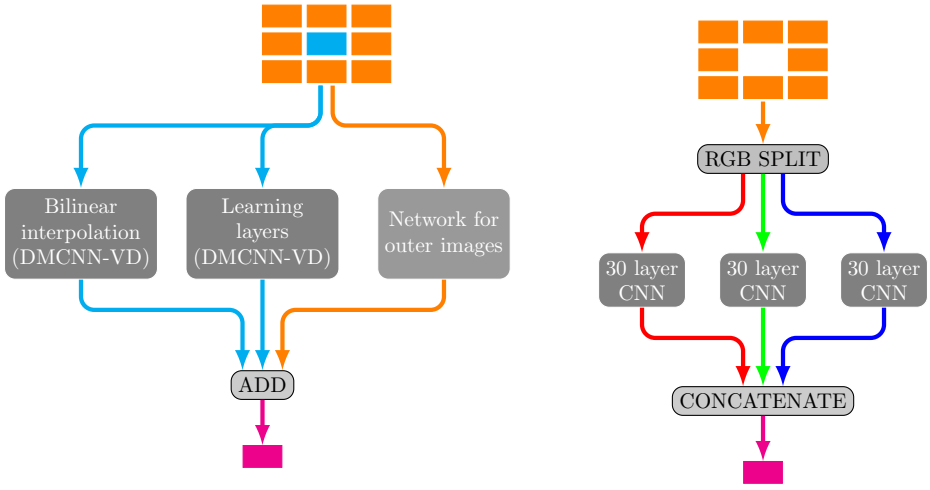
network designs and not all of them are capable of achieving a set goal, it makes sense to draw inspiration from architectures that have already proven themselves for similar or closely related problems.

After analyzing different state-of-the-art demosaicing algorithms, it was decided to use DMCNN-VD [96] as the basis for our work. Figure 3.5 shows its general structure. It consists of 20 batches of layers, each with one convolutional, one batch normalization, and one SELU [105] activation layer. In the end, the result of the trained network is added to the output of a traditional bilinear interpolation. As a result, the network only has to learn the errors a linear interpolation creates and correct those instead of performing the complete demosaicing procedure. It processes image patches of 33x33 pixels and delivers the best results compared to other state-of-the-art algorithms. To demosaic images with a higher resolution, images are cut into 33x33 pixel tiles, which are then processed and recombined later.

Since we can not rely on getting all missing information from the neighboring cameras (for example when the correspondences fall onto the same color channel or near the edges of the camera setup) we decided to keep the original DMCNN-VD network as part of the new architecture so it can serve as a fallback in case no useful data can be found in the other images. The design in Figure 3.6 was chosen, as it delivered the best results in preliminary tests.

Comparing that split network approach with an aggregated one with similar complexity to the three separate paths combined yields measurably better results for the split networks. Additionally, by separating the color channels of the outer images and processing each one using its own network, we try to emphasize that we are less interested in pseudo-demosaiced data from the outer images and want to prioritize original information. In those color-specific networks, we use 30 convolutional layers with 135 3x3 kernels, batch normalization, and SELU activation. Those parameters allow the network to draw in information for a certain pixel from an area spanning about a third of the image's width to the left and right and half the image in both vertical directions. With that range, it should be able to find matches for a pixel in most cases. Only when the object it belongs to is very close to the camera array, and therefore its disparity between cameras is very high, does it not work.

While the structure of the DMCNN-VD section of our network is very close to the original, the size of the input images is increased to 96x54, which is exactly the twentieth part of a FullHD resolution. The main reasoning for that is a property of our approach that currently




(a) Overall network

(b) Outer image network

Figure 3.6.: Network to integrate information from neighbors into the demosaicing process.

prevents it from being used on image tiles and requires full inputs as images. That problem is discussed in more detail in Section 3.5.1.

The TensorFlow [106] implementation with nearly 14 million trainable parameters in the four trainable parts of the network, is not particularly large when compared to bigger ResNet versions [107] or AlexNet [108], which both have over 60 million parameters. Nevertheless, it is already big enough to create problems during training. Not only does it require a careful choice of parameter initialization and learning rate, but also large amounts of memory during training. Compared to the original DMCNN-VD, our network takes about 42 times more input parameters and has to output over four times as many output values. For the AlexNet, the relation is slightly different; while it takes roughly 25% more inputs, our network outputs 15 times more data points. That means the data and the trainable weights are reduced towards the output. Their leaner internal structure allows it to be trained on two GPUs with 3GB of memory each when the layers are properly distributed between them. In our experiments, we ran into problems with training our network on GPUs with 16GB even with very small batch sizes of one or two because the layers have a constant size and the amount of data that needs to be kept in memory until the back-propagation step adds up quickly. Training it on a CPU-based server with more memory was not feasible as it extended the required training time to weeks or months instead of days on the GPU.

## 3.4. Training Data

Using captured lightfield footage as training data would have been ideal but is not possible due to one main reason: even though the available samples increased with the growing interest of the research community, they are all captured using single-sensor cameras. That fact disqualifies them as training data, since they already include demosaicing artifacts of various magnitudes, depending on the algorithm that was used. Rendered lightfields exist and do not require demosaicing, but even realistic scenes often do not have the same amount of challenging high-frequency textures as real images.

For our experiments, we chose a slightly different approach to generate enough training samples. In a first preparatory step, we created a collection of traditional demosaicing test datasets from scanned film or tri-sensor cameras. Then we devised a geometrically simple scene consisting of two planes, two instances of a monkey head in Blender[1], and nine cameras in an arrangement similar to that of our own camera array.

To generate samples with sufficient diversity in color and depth structure, we automated the steps seen in Figure 3.7. For every sample, random translations and rotations are added to every object in the base scene. The translations of the scene objects are chosen such that they often change their order in front of the background, so the occlusions are very varied. The part of the applied texture that is seen by the camera can also be changed depending on which side of an object is facing the camera. In order to make the samples more realistic, every camera is rotated between -2 and 2 degrees around all axes to simulate alignment deviations as they often occur in real camera arrays. For every object, a random image from the demosaicing test image collection is assigned as texture before rendering, to increase the amount of high-frequency features in the final image. The final render is performed in the resolution our network accepts as input for all nine cameras.

One sample for the training of the network then consists of the nine rendered images, whose color information is artificially reduced to simulate the presence of a color filter, similar to a real camera as input. The label for that input data is the original center image. For our training, we created a total of 20 thousand of these samples, of which one thousand samples were used as test samples, one thousand for validation, and the remaining ones for training.

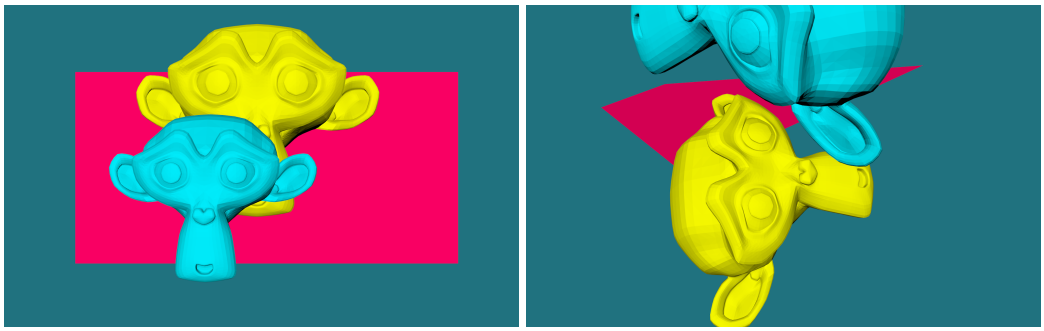
## 3.5. Evaluation

To evaluate the performance of our approach, we used the samples from the test partition of our dataset. For comparison, we also trained a pure DMCNN-VD version on our dataset. This was necessary because of two reasons: we wanted it to work with images of the same size as our algorithm, but primarily due to the fact that the authors of that approach did not publish a trained model for direct comparison.

For our final quality values, the test set was processed by our network as well as the DMCNN-VD implementation and their CPSNR[2] scores were measured against their respective label. In those measurements, the DMCNN-VD scored at $35.42dB$ while our best approach reached $37.87dB$. More details about the other variants and their performance can be found in [101].

DMCNN-VD claims a value of $41.05dB$ in their paper [96] therefore it was necessary to investigate where this discrepancy in the results was coming from, as testing our approach against an inferior version of the state-of-the-art algorithm would be worthless. Since the trained network for DMCNN-VD is not publicly available, we compared the performance of ARI [95] on our dataset, as it is available as a MATLAB implementation and its performance values are mentioned in the DMCNN-VD paper. According to the paper's results, ARI achieves $39.00dB$. Applied to our dataset, its performance drops to $33.91dB$. We concluded that our dataset must be more complicated than the one they were previously tested on since both algorithms performed worse by a comparable value.

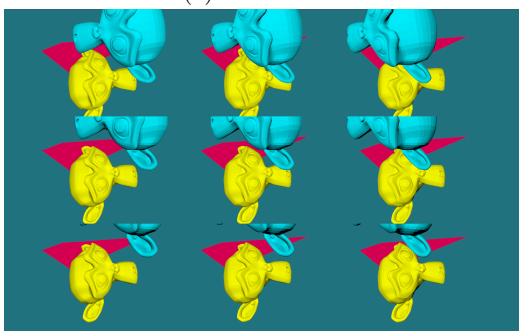
[1] `https://www.blender.org`
[2] Averaged value of the PSNR score from all color channels



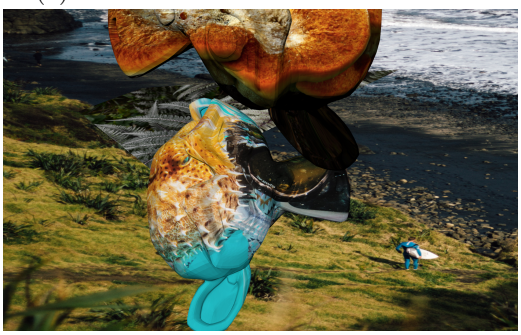
(a) Base scene



(b) Scene with added random movements



(c) Scene as seen from all cameras



(d) Scene with random textures

(e) Final image

Figure 3.7.: Generation process of training samples.

This finding legitimizes our result, such that we can conclude that the additional information from the neighboring images in a camera array improves the results on average by $2.45dB$. The main drawback of the current version is the problem that it can only process images with a resolution of 96x54, which is hardly enough to be used on real captured footage.

### 3.5.1. Open Issues

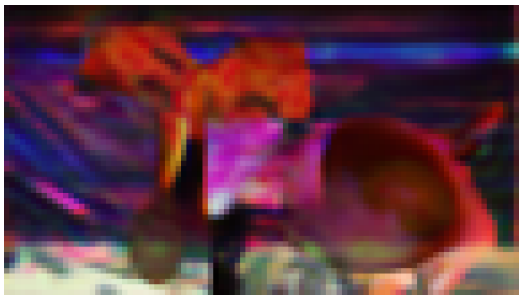
With an input resolution of 96x54, the network can not be used for any practical application. Scaling the resolution up to the default resolution of the array would theoretically be possible but would require additional changes. With the 20 layers of 3x3 kernels in the outer image network, matches in the images can only be found 20 pixels away in every direction from the original location. For small resolutions, this is a significant portion of the image, but at a resolution of 1920x1200, it is too small to catch most instances of a scene point in the other images. To increase the detection range, more layers can be added to the network or bigger kernels can be used. Both possibilities significantly increase the amount of memory required for training, in addition to the factor introduced by the higher resolution.

Just with the resolution adjustment, the amount of input and output data is increased by a factor of 400. With more layers and bigger kernels, that factor could easily reach three or more orders of magnitude. Since we already used a significant portion of our available GPU memory during the training for the current parameters, this becomes simply unfeasible.

Tiling the input images would solve the issue but for array material, it is much more complicated. For every tile from the image, the matching tiles in the neighboring images have to be found. Since their locations depend heavily on the depth values in the center tile, this is not straightforward. Training a new network that can determine the position of matches for a certain pixel in the other images would be possible, proven by learning-based stereo depth estimators like the one by Yao *et al.* [109]. As the image tiles we are looking for are bigger than a single pixel and can contain parts of multiple objects with different distances from the camera, occlusion effects may require a variable amount of additional tiles from the neighbors to find the matches for pixels in the foreground, background, and all depths in between.

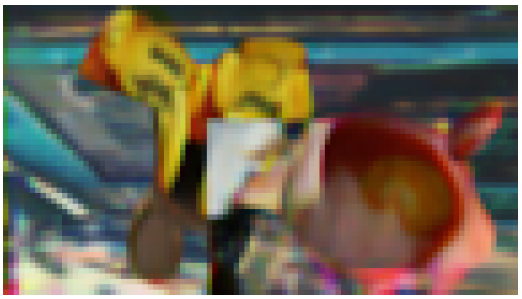
The variable amount of outputs of such a network and the variable number of inputs of the following network currently cannot be handled by our architecture. Assuming a fixed amount of candidates and padding the data in cases with less data is an option but still requires a more complex network. Preliminary tests in [101], for how the current version behaves on the borders of the camera array, show that there exists a heavy bias for certain neighbors. In Figure 3.8, the different effects of missing neighbors become very apparent. While on the left edge of the array only very minor errors appear in the results of the network, on the top edge the results are completely unusable. With more kernels and more training samples with missing neighbors, the issues on the edges could be resolved. However, those issues are still present in the network and the effort required for the addition of more choices per direction becomes harder and harder to determine.

The most pressing issues of our approach can very likely be solved with more hardware resources or more efficient architectures, but for now, a practically usable solution is out of reach. Since we have proven the effectiveness of our network architecture, when it comes to increasing the demosaicing quality with the information from other cameras in an array, we are confident it will still be relevant when more potent hardware or new deep learning methods become available.



(a) Sample without the top row of neighbors.

(b) Reference image with all neighbors available.



(c) Sample without the left column of neighbors.

Figure 3.8.: Influence of missing neighbors onto the current version of our demosaicing network. Missing top neighbors lead to massive errors, left neighbors seem to have only minimal influence on the final result.

# 4. Real-Time Multiview Coding

Multiview and lightfield video have one major difference in comparison to conventional video with regard to the raw output. While conventional video only consists of the output of a single camera, the new formats always contain the footage from multiple cameras. With multiple cameras, the overall data rate of those videos is equal to that of a single camera multiplied by the number of cameras used to capture the scene. Depending on the respective pixel format, the resulting data rates can quickly overwhelm a consumer's Internet connection or even the processing capabilities of professional production equipment. Fortunately, there is a lot of similar content in the images of different cameras, which could be exploited for efficient compression. There are even additions for well-known and widely used video compression standards like H.264/MVC [31] or H.265 [32] for the support of multi-camera footage and emerging new formats, specially designed for this kind of footage such as JPEG Pleno [110]. Most of these standards are only implemented in their respective reference implementations. For real-time applications with multiview video, none of these implementations are usable, as their complexity is quite high and compressing a frame takes significantly longer than a usual frame duration.

In this chapter, we present a scalable solution for real-time compression of multiview footage based on the H.264/MVC standard. It exploits the similarities between H.264/AVC and H.264/MVC to accelerate the compression. We show that in practical examples, it can reach real-time performance and can be used for interactive systems with multiple cameras.

## 4.1. Background

The content of this chapter describes fine details about the inner workings of video codecs with multiview support. In this section, further information is given to help understand the complexities of this work and underline its importance based on the available real-time encoders and decoders.

### 4.1.1. Standards Supporting Multiview Content

When most of the work for this chapter was done, not many video codecs with multiview support existed. The most mature was H.264 with its Annex H adding support for multiple views in one stream, while maintaining backward compatibility, such that a decoder with no multiview support can at least decode the base view. All data only used for multiview content is added in the form of data sections which an unaware decoder can safely skip, such that a single "normal" view remains. The main approach to exploit the similarities between different views for a more efficient encoding is to apply the same prediction scheme, already present in the time domain for a single view, to the different views at the same time instance. Figures 4.1 and 4.2 in Section 4.2 visualize this idea.

At the time, implementations in hard- and software capable of encoding and decoding multiview video existed, but all of them were limited to a maximum of two views. The main reason for this limitation is the use of H.264/MVC on BluRay discs with 3D content, which

does not require more views. Beyond that, only the reference implementation is available but is far from being able to achieve real-time performance [111] (see Section 4.4.1).

In recent years, there have been efforts to implement certain parts of a full H.264/MVC encoder on FPGAs, but not a whole encoder or decoder [112]. The memory requirements, which increase linearly with the number of views, and the quadratically increased complexity for the parameter search, are the most likely reasons for this, as video encoders are already quite complex applications to implement in pure hardware, even for single view versions.

The successor to H.264, H.265/HEVC already has multiview support in the base standard. It mainly follows the same principles as H.264 when it comes to extending the format to carry multiple views at once. Even though it can improve the encoding efficiency by up to 70% compared to H.264/MVC [113], apart from the reference implementation, nothing is available yet. For the 3D version, a small number of scientific efforts to create faster encoders can be found, but they do not state any absolute numbers about their speed [114]. The main reason why for this work H.264/MVC was preferred over H.265, was the missing widespread availability of hardware encoders and decoders in consumer hardware at the time, as they were planned to be an important building block of the work in Section 4.3.1.

Recently, due to the enormous popularity of lightfields in the research community, new standards for the coding of immersive multiview content have emerged.

JPEG Pleno [110, 115] is the effort of the Joint Photographic Experts Group to provide an efficient means to encode material with multiple views captured by camera arrays or plenoptic cameras. It requires high-quality disparity maps for the encoding and uses those in combination with a few fully encoded views in key locations, to reconstruct the complete lightfield. Calculating such disparity maps with sufficient quality within a reasonable time frame and still making the whole encoding process real-time capable, is impossible. Additionally, JPEG Pleno was never intended to be used with video footage. That fact disqualified it for use in this chapter.

MPEG-I [116] is intended to encode multiview video to provide interactive immersive experiences. It requires a depth for every view and frame in the stream to quickly and accurately determine corresponding areas. By choosing "base views" with the most correspondences with neighboring views, the amount of data added by the other views is minimized. The remaining data is arranged together with the base views into atlases of rectangular image sections, for both the color data and depth information. After encoding, the depth and color atlases for every frame in the sequence using normal HEVC, some metadata is added for reconstruction and the MPEG-I stream is complete. For decoding, the information from the atlases is projected to the desired view location, which recreates the view at that location. As of now, only the reference implementation of a depth map generator is available. Due to its dependence on depth maps and it being in the very early planning phase at the time most of the work for this chapter was done, it was not considered as a basis for this work.

### 4.1.2. Frame Coding in H.264 and HEVC

H.264 and its successor HEVC both follow the same principle for encoding video data. Every image is represented as a set of slices which represent connected sections of the image. There are three types of slices, I-slices, P-slices, and B-slices. The type determines how the image data in the slice can be predicted later.

Data contained in the slices is split up further into smaller elements. H.264 calls them

macroblocks (MBs), in HEVC they are named Coding Tree Units (CTUs). While H.264 uses a fixed size for the macroblocks of 16x16, the equivalent of HEVC allows different sizes between 16x16 and 64x64 pixels. In both cases, the image sections can be subdivided multiple times if it improves the encoding performance.

In general, the data in a macroblock is predicted by a so-called motion vector that points to a very similar image section in the same or other already decoded frames. The information from that image section is then used as a basis for the prediction. The difference between this base data and the original data is stored as residual data and added during the decoding process. Assuming that the motion vector and the residual data are fairly similar for neighboring blocks, their base values are predicted from the neighbors and only the difference is stored.

When the desired size for a macroblock is reached, the prediction mode is determined based on the slice type. I-slices only allow for Intra prediction. In that mode, the encoder can only use previously encoded macroblocks from the same image to represent the data. Those blocks only contain the residual and no motion vectors. They do not store the full residual but again only the difference, which is added to a base derived from the neighboring blocks. A special type of I-slices is the IDR-slice, which stands for Instantaneous Decoding Reset. At the start of such a slice/frame, all previously decoded pictures still in the decoders buffer are deleted and the internal states of the decoder are reset to the start configuration. All following operations referencing frames before an IDR-slice produce undefined results.

P-slices can also use intra-coded blocks, but use it only as a fall-back in extreme cases. Normally, they use inter-coding for block prediction. By referencing a previously decoded frame, the coding becomes more efficient, because there is a high chance that the object a macroblock belongs to has only moved slightly since the last frame and therefore, a nearly identical copy of the data should be available there. In addition to the residual data, a P-type macroblock has a motion vector pointing to the best match in the other image, relative to its current location, and an index referencing the image it wants to use. That index points to an entry in a sorted list of available references, which is generated for every slice and used for all macroblocks within it but it can be modified for a single block, if necessary. In such a case, arbitrary indices can be added, removed, or modified in the list. There is no maximum number of allowed operations, such that nearly all modifications one can think of are possible.

A special property of predicted blocks is the ability to skip them. Skipped blocks only contain a very short header and a skip flag. All other data is derived from the previous blocks without modification or initialized as zero.

B-slices are the most efficiently encoded slice type. Their macroblocks can be I-type, P-type, or B-type. The B-type is very similar to the P-type, the main difference being a second list of reference images in reverse order of the first, as well as a second reference and motion vector. The two sections of the two images to which the references point, are combined with an optional weight before the residual is added. For special prediction cases in B-slices, the final result depends on a field called the co-located picture. In all cases discussed here, it points to the first entry in the second reference list.

If the data in slices would be stored simply as raw bits, the encoding would not be very efficient. Most fields in the standard are defined such that the most common values map to zero and the amount of entropy in the final data is minimized overall. The standard defines two methods for encoding the bits further. Context Adaptive Variable Length

Coding (CAVLC) is the simplest approach, which assigns bit sequences to values based on their probability of appearing in the stream, with a clear focus on values between -1 and 1.

The second approach is on average 10-20% more efficient. The Context Adaptive Binary Arithmetic Coding (CABAC) uses a large set of context objects holding the coder's state for more efficient encoding. By choosing the correct context based on the current state and data field being read or written, it can adapt to the characteristics of the expected data and increase its efficiency.

Which of the features described above are actually used for the coding of a video is determined by the encoder, but the choice can be limited by the selected profile and level. The defined profiles and levels determine feature sets and the overall performance a decoder has to have in order to be able to decode a video without interruptions. In Section 4.3.1.1 the most important profiles and their supported features are discussed. Details about the available profiles and their influence on the scalability of the presented approach are given in Section 4.4.3.

### 4.1.3. Stream Structure in H.264

In an H.264 stream, all data is contained in Network Abstraction Layer (NAL) units. Every H.264 encoded video contains at least a Sequence Parameter Set (SPS), a Picture Parameter Set (PPS), and one slice. The SPS holds important parameters for the whole video. It includes the profile and level required to decode the video and the size of the encoded frames. In the PPS, an SPS is referenced and the length of the reference lists for P- and B-slices are defined together with several quantization and quality parameters. With those two NAL units, the decoder can start to recreate images from slices.

The chain of slices following the initial parameter sets can contain an infinite number of I-, P- and B-slices in a nearly arbitrary order. It is not completely free because IDR slices can destroy the prediction mechanism of following P- and B-slices, as described in the previous section. Usually, a certain pattern of different slices is repeated over and over, separated by IDR slices in key locations. Such a group of slices is known as a GOP or Group Of Pictures. Different GOPs do not interact with each other because the decoder might be reset between them. This fact becomes important when multiple independently encoded streams are combined, as in Section 4.3.1.

In addition to the minimal required NALs, many more are defined in the H.264 standard. The most important ones for this chapter are the Supplemental Enhancement Information (SEI), which can hold arbitrary and optional data not necessary for decoding in most cases, the subset sequence parameter set (SSPS), and the prefix NAL unit. The SSPS contains a complete SPS but adds information required for the handling of multiple views within one stream. Prefix NAL units are closely related to SSPSs because they hold the required multiview information for the base view of the stream, which has to use SPSs. Access Unit Delimiters (AUD) help to identify the start and end of GOPs but are only mandatory for dedicated devices such as BluRay players. Video Usability Information (VUI) contains non-essential information about the encoded video, such as aspect ratio, overscan, and color formats.

For HEVC, the names for the NAL units have changed, but apart from small differences, the same principles apply.

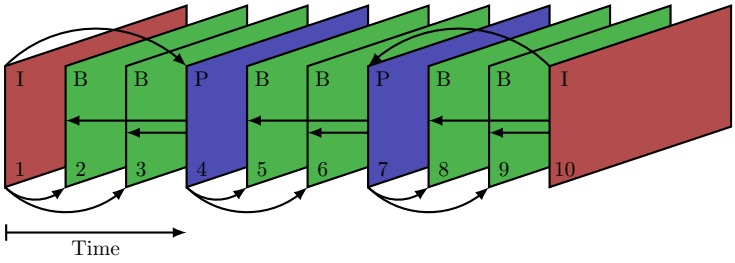



Figure 4.1.: Example for an AVC prediction scheme. Arrows point to image which uses the source as a reference for their prediction.

## 4.2. Concept

Looking at the compression schemes of H.264/AVC in Figure 4.1 and H.264/MVC in Figure 4.2, it becomes obvious that there exist many similarities. Even though the MVC follows a two-dimensional pattern instead of a one-dimensional one, the prediction pattern of the single views stays the same. The inter-view prediction can be seen as a tacked-on feature, which it technically is since the multiview feature set was defined in the Annex H of the H.264 standard. The inter-view prediction is defined nearly identically to the inter-frame prediction with only minimal changes to prerequisites and limitations to be able to differentiate between the two prediction modes. This similarity, combined with the fact that the main complexity of H.264/MVC encoding stems from the number of images that have to be kept in memory and need to be analyzed to achieve the best encoding, lead to the idea to distribute the overall complexity to multiple computational units and also leverage the availability of H.264 hardware encoding features in modern CPUs. The possible improvements in encoding efficiency by replacing I-frames of non-base views with P- or B-frames and inter-view prediction have already been discussed and optimized by Merkle *et al.* [117]. Therefore, our efforts were mainly focused on an approach for faster encoding and not the search for the perfect prediction pattern.

The details of the steps we have taken to distribute the complexities and MVC encoding to achieve much faster encoding times are explained in Section 4.3. The main idea is to treat the frames from every view independently at first, encoding them as H.264/AVC with the same parameters and, after gathering them in a central location, combine them into a single stream with different levels of inter-view prediction and minimal changes to the encoded data.

## 4.3. Implementation

The implementation of these ideas seems simple at first glance, but requires a significant amount of analysis of the internal structure of H.264 streams and their extensions. There were multiple implementations of the intended changes to the encoding process, each focusing on a certain feature in the list of changes and building upon the earlier stages whenever possible. Most of the implementation work in this chapter was part of Josef Nguyen's work [118, 119]. The stream multiplexer without proper inter-view prediction is described in Section 4.3.1. Improved versions are presented in Sections 4.3.2 and 4.3.3.

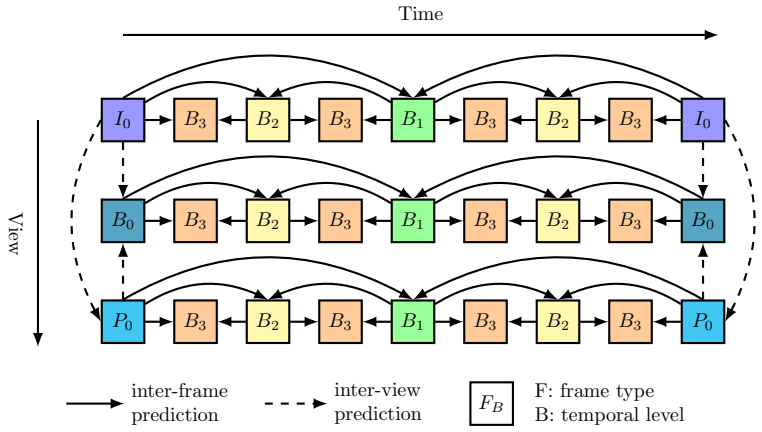



Figure 4.2.: Example of a MVC prediction scheme. Arrows point to image which uses the source as a reference for their prediction.

### 4.3.1. Stream Multiplexer

The basis for all approaches is the assumption that pre-coded H.264/AVC streams are compatible with H.264/MVC or can be made compatible with minimal changes and without re-encoding of the contained footage. The proof-of-concept implementation starts with an analysis of the different feature sets available in the different profiles of H.264 and their stream structure. In the following sections, we discuss non-obvious problems that occur with this approach and show how they can be remedied.

#### 4.3.1.1. Encoder Features

To combine multiple H.264/AVC streams into a single H.264/MVC stream, the first issue to be fixed is the feature set used by the encoders. This is important because the multiview profiles do not support all features the single view profiles offer, as shown in Figure 4.3. In order to have the largest common feature set between the single view profile and the multiview profile, it was decided to choose the *High Profile* and the *Multiview High Profile*. Those profiles give a good selection of features for efficient coding and only few incompatible features. The main features which cannot be used are *Field coding* and the closely connected *MBAFF (MacroBlock-Adaptive Frame-Field) coding*. Both of these features are only required for the efficient coding of interlaced video, a format whose use has been gradually diminishing in recent years, in favor of progressive video with higher spatial resolution because of the higher perceived quality [120, 121]. Nowadays, these features are mostly used in bandwidth-limited cases in which the frame rate has absolute priority over the resolution. As video conferencing and comparable applications usually only contain slower movements than sports broadcasts, the loss of interlaced coding support is not a problem. Without the need for interlaced coding, the pre-coded views are compatible based on the feature set but the structures of the streams are not the same.

#### 4.3.1.2. Stream Structure

Looking at the structures for both types of streams, it is clear that one stream, namely the base view, remains unchanged while the others have to carry additional information.

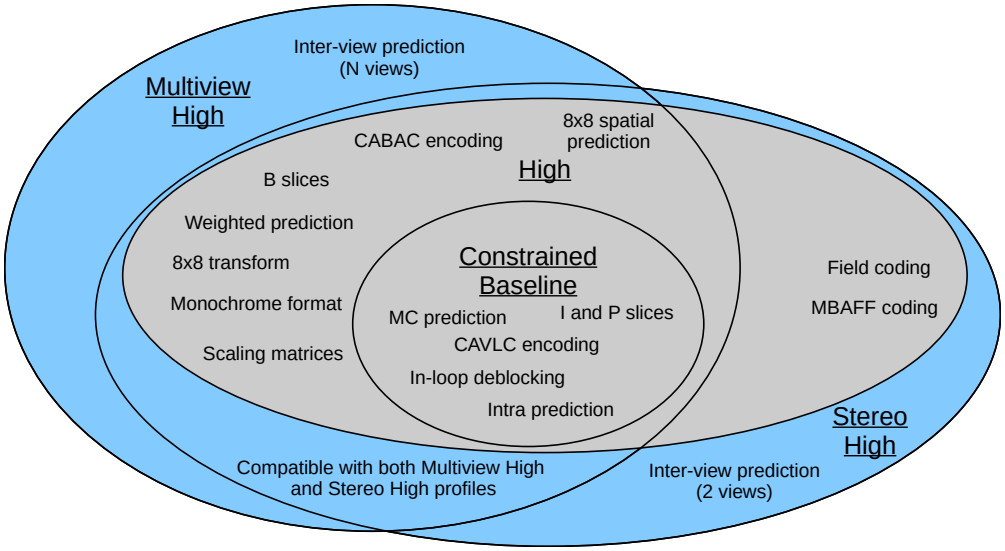



Figure 4.3.: H.264 supported feature cloud including Constrained Baseline, High, Stereo High and Multiview High profiles according to the current standard [31].

The Sequence Parameter Sets (SPSs) of the other views are replaced by a Subset Sequence Parameter Set (SSPS) which consists of an SPSs and some additional information to identify which other views are required to decode it. The required additional data in the SSPS can be easily deduced from the order and number of input streams of the multiplexer:

**Number of views** The number of views is equal to the number of input streams.

**View order** The order of views is not important as long as no inter-view prediction is used, but for later use cases we define the order of views as the order of inputs to the multiplexer.

**View dependencies** The dependencies are left empty for now, as there is not going to be any inter-view prediction and therefore, no dependencies between the views.

**Operation points** The operation points reference levels to define the required decoding capabilities to decode a view. It also includes the requirements of its dependencies. Since there are no dependencies here, we define them as the level from the input stream.

**Display options** For the display options, the default values are used since this functionality is not used in this scenario.

As the parameter set IDs are identical in every input stream, it is not possible to differentiate between when left unaltered. For this reason, a counter for every type of parameter is used to adapt the parameter set ID when first encountered. Subsequently, the process of mapping from an input stream and original ID to a new ID is saved, ensuring that other references in the stream can later be adapted to the new IDs.

Since all input streams are encoded by similar encoders and equal parameters, it is very likely that at least some of the SPSs and therefore, the SSPSs contain identical content, meaning that any repetition would be an unnecessary waste of data rate. The multiplexer

compares every new SPS with those it has already encountered. When an SPS with identical parameters is found, it is discarded and only ID mapping is stored.

What follows the sequence parameter set in the input streams is a number of picture parameter sets. Most AVC encoders create multiple sets for every stream, even when only one of them is used. As it is hard to find out which parameter sets are really used in the stream that follows without analyzing every single NALU in it, only incoming sets with different contents are incorporated in the final stream. In order to do this, a similar approach to that of the SPSs is applied where the minimal set of unique parameters sets is determined, new IDs are assigned and the mapping from old to new IDs is stored and used for the rest of the stream.

For both the SPSs and the PPSs, the content of the parameter sets is left mostly unchanged but the ID is adapted as described above. Even though it is not important for the decoding with an H.264/MVC compatible decoder, we create a SEI unit that includes the mappings of the parameter sets and embed it into the head of the stream. Since there is no type of SEI message that directly corresponds with this data, we use the unregistered message type which allows for arbitrary content in the message. We use this feature to be able to reverse the multiplexing process, as the demultiplexing of the streams can be used to parallelize the decoding process at the receiver. Other SEIs created by the original encoders are dropped because they often are unregistered messages as well and they are not necessary for the decoding process. Leaving them in would require a restructuring of their internal data, so they can be clearly differentiated from the messages the multiplexer creates.

The next difference that must be addressed is the MVC extension for the NAL unit header. It is used in slices and mainly defines the view a slice belongs to, how important it is in the stream, and lastly for which kinds of references it can be used. Since this information is also needed for the base view, but it cannot use the header extension to maintain backward compatibility with non-multiview aware decoders, a prefix NAL unit, which contains the additional information, is added before every slice. The content of the fields is set as follows:

**IDR flag** Since an IDR frame causes all decoder buffers to be cleared, only the IDR frame from the base view is kept. For the IDR frames of the other views, this flag has to be set to false. Only then is it guaranteed that the buffers from lower views are maintained and the following frames can be decoded correctly.

**Priority** The priority is assigned based on how they are referenced later. I-slices get the highest priority of 0 as they are most likely to be referenced later. Other slice types used as references by other frames are assigned priority 1. All slices belonging to non-referenced frames get the lowest priority 2.

**View ID** The view ID defines the position of the view in the input order of the multiplexer.

**Temporal ID** Temporal IDs are only required in combination with operation points from the SSPSs which are not used in this scenario. Therefore, the values are assigned as follows: I-slices and P-slices get the temporal ID 0. Referenced B-slices get the ID 1 and non-references B-slices get the ID 2.

**Anchor pic flag** The anchor frames denote borders for the prediction process. That is why all the I-slices which have not been designated as belonging to IDR frames are set as anchor pictures.

**Inter-view flag** No inter-view prediction is used in this scenario therefore, this flag is set to false for every frame.

Any non-essential NALUs, such as Video Usability Information (VUI), Access Unit Delimiter (AUD), and other Supplemental Enhancement Information (SEI) are removed from the input streams. The main reason being the fact that the application context of the stream is known and does not need the additional data for the compatibility with BluRay players or other special devices. Some types of NALUs, especially VUIs, can even crash the H.264/MVC reference implementation as soon as it encounters them in the stream because it does not ignore them as it is supposed to, according to the standard.

#### 4.3.1.3. Software Structure

Implementing the required stream modifications as described in the previous section requires partial decoding of the stream. As H.264/AVC defines different data coding schemes which are used to encode different parts of the NAL units, even reaching the required data can be quite complex. In addition, determining the differences between the stream structure defined in the standard and the structure a commonly available encoder actually creates, was crucial for the creation of a working prototype.

To achieve this, two versions of the multiplexer were created: a partial encoder and decoder pair, written from scratch in C++, which sticks very closely to the definitions in the standard and focuses on readability and comparability with the standards definitions while being easy to modify and adapt to the needs of the multiplexer. It also skips most parts of the standard which interpret the encoded parameters, especially those that create images from the raw data, as these are not important for this implementation. It decodes the stream to such an extent that the data structures mentioned in the standard are accessible and modifiable, but only evaluates them as far as it is required to get to this point. To recreate the stream from the decoded data, the encoding is also implemented starting from the stage where the decoding ended. An overview of the implemented features is shown in Figure 4.4. The resulting implementation was used for stream analysis and to determine all required changes to achieve the intended result. Since it was created for readability and comparability with the standard, it was quite slow and was not able to achieve real-time performance in any realistic scenario. To show that the solution was able to transcode live material with sufficient speed, a more optimized solution was required.

The first impulse of using a fully optimized encoder like x264 as the basis for the implementation, to get the best possible speed, was never fully executed. The main reason for this being that the encoder side (the byte stream writer) in particular is very closely intertwined with the parameter prediction, and the input it requires is quite hard to create without starting the encoding at the very beginning. Another drawback was the missing MVC awareness of the encoder, which would have meant that adding all required extensions in a fully optimized fashion was required before it would be of any use. Since the software was still a work in progress at this point, with an unknown number of changes still being required as the feature set of the multiplexer was extended, this was not feasible. With this inherent inflexibility and the problems with isolating the byte stream coding from the rest of the encoder, this option lost a lot of its viability.

As a compromise between speed and usability, the byte stream encoding and decoding portion of the JMVC reference software was extracted and restructured to be usable as an independent library that can be easily combined with the multiplexer. Such a library makes the logic for changing the stream independent from the basic reading and writing of the byte stream. If the need arises for a more potent byte stream handler, it can be easily replaced with a new version without touching the multiplexer's algorithms.

The overall structure of a complete system based on this multiplexer is shown in Figure 4.5. The first step is the pre-coding of the camera footage in the camera nodes. It is very important

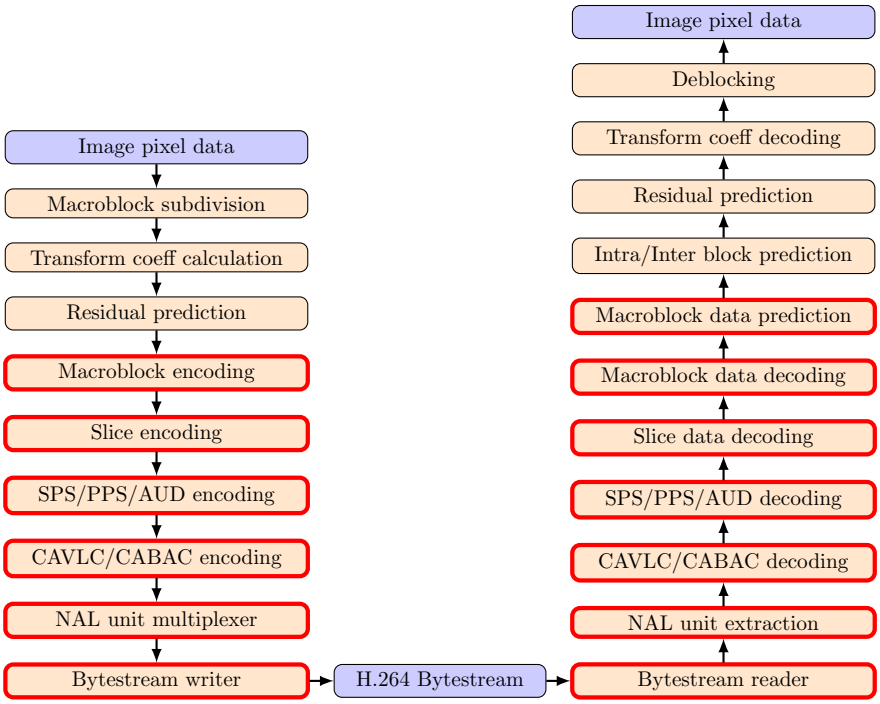



Figure 4.4.: Abbreviated list of steps in a H.264 encoder/decoder pair. Steps with a red border are implemented in the experimental transcoder.

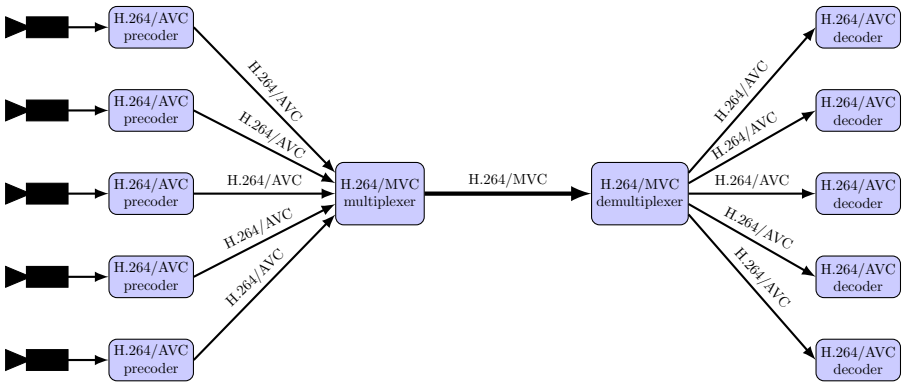



Figure 4.5.: Structure of the multiplexer system. After the individual precode as normal AVC streams, the multiplexer combines them in a single stream. At the receiver the process can be reversed.

that all nodes use the same parameters for the encoders as deviating GOP sizes can lead to severe decoding artifacts due to how IDR frames are used in MVC. Many parameters of the encoder are only there to make the AVC encoder only use features that also exist in the MVC profiles and to disable certain automatic parameter calculations so that the frame structure is identical for every node, even though the images in the streams differ slightly.

Below, the options and parameters for the GStreamer plugin which handles the pre-coding in the camera nodes is given:

```
x264enc byte-stream=true option-string="bframes=0:min-keyint=16:
keyint=16:no-scenecut=true:force-cfr:no-mbtree: sync-lookahead=0:
rc-lookahead=0"
```

First, "byte-stream=true" forces the resulting byte stream into the format specified in the H.264 standard. "bframes=0", "force-cfr", "no-mbtree", "sync-lookahead=0" and "rc-lookahead=0" reduce the overall time required for encoding by minimizing the number of frames being sent out of order and therefore, the overall transmission delay. This is done by disabling B-frames altogether. "min-keyint=16", "keyint=16" and "no-scenecut=true" set the minimum and maximum length for GOP sizes in the stream and disable the detection of scene cuts which would trigger the start of a new GOP out of sequence.

The pre-coded streams are then sent to the multiplexer via RTP to a predefined port. On the sender machine, they are fed into the multiplexer which ingests their contents until all input streams reach a valid state. A valid state means that at least one SPS and one PPS have been read. As soon as this is the case, the multiplexer analyses their content and creates the multiview correspondences as well as the mapping as described in Section 4.3.1.2. The mapping and the new parameter sets are then written to the output byte stream, which concludes the multiplexer initialization. During normal operation, the data from an input is read until a slice unit is found. Its header is adjusted to match the new parameter set IDs, the new profile, and level values. If it belongs to the base view, the required additional NAL units are created before they are written to the output. This procedure is then repeated for all other inputs in a round-robin fashion until the first input ends.

The output stream is sent to the receiver machine using an arbitrary transmission method. On the receiver, there are two options to decode the stream. If a sufficiently fast decoder with H.264/MVC support is available, the stream can simply be fed into it and be decoded as the standard intends. If no such decoder can be used, a demultiplexer can be used to recover the single view streams which can then make use of widely available hardware decoding support. To do this, the previously described process is reversed, the original IDs of parameter sets are restored and the fields added by the multiplexer are removed. After the reversal process, it is guaranteed that all streams have only the required number of parameter sets and only valid fields according to the H.264/AVC standard, which can be used by nearly any common video decoder. With the hardware decoding support of modern CPUs and GPUs, it is possible to decode multiple H.264/AVC streams in parallel. Independent of which approach was used for the decoding of the streams, the contained frames from all cameras can now be used or processed further.

The viability of this approach was demonstrated at CeBIT 2017 (Figure 4.6), a prominent German tech expo, and acknowledged by the research community [5]. For that event, the functionality of the multiplexer was implemented as a GStreamer plugin, including multiplexing and demultiplexing with automatic parameter detection, so that there was no need for configuration files. Using five cameras capturing 1080p video at 25 frames per second, the system created a multiview stream from the live footage, streamed it over a small local network, and decoded it again to show a mosaic with the views from all five

Figure 4.6.: Multiplexer demonstrator with five cameras at CeBIT 2017.

cameras. The system performed nearly perfectly for multiple hours at a time, with a total system delay of less than one second.

Since this approach only multiplexes multiple H.264/AVC views, the encoding performance does not reach the level that can be achieved with the reference implementation. This is easily explainable since it does not make use of the inter-view prediction, which is the main new feature of H.264/MVC. Making use of this feature and improving the encoding efficiency was the focus of the following extended implementations.

### 4.3.2. Towards Inter-View Predictions in Multiplexer

In the presented approach for a scalable encoder, the first encoding steps and the final stream construction do not necessarily have to be performed on the same device. Therefore, the different stages might not have access to every part of the input material. For the traditional approach, such access is absolutely necessary to achieve the best encoding performance, since it requires a thorough analysis of the whole range of inputs. Since an optimal solution is not possible, even for highly optimized encoders without significant delays, we focus first on implementing a non-optimal solution that still performs significantly better than the version from the previous section.

One major source of bitrate in the previous solution is the I-frames in views two and higher. In H.264, the I-frames are usually more than three times larger than P-frames, which in turn are about twice the size of B-frames [122]. For five views, this means the equivalent of 20 B-frames is wasted in every GOP simply by having an I-frame for every view in the beginning. Considering that in interactive applications quite short GOPs of length 8 or 16 are used, depending on the frame rate and delay requirements, there is a lot of potential gain in replacing the I-frames.

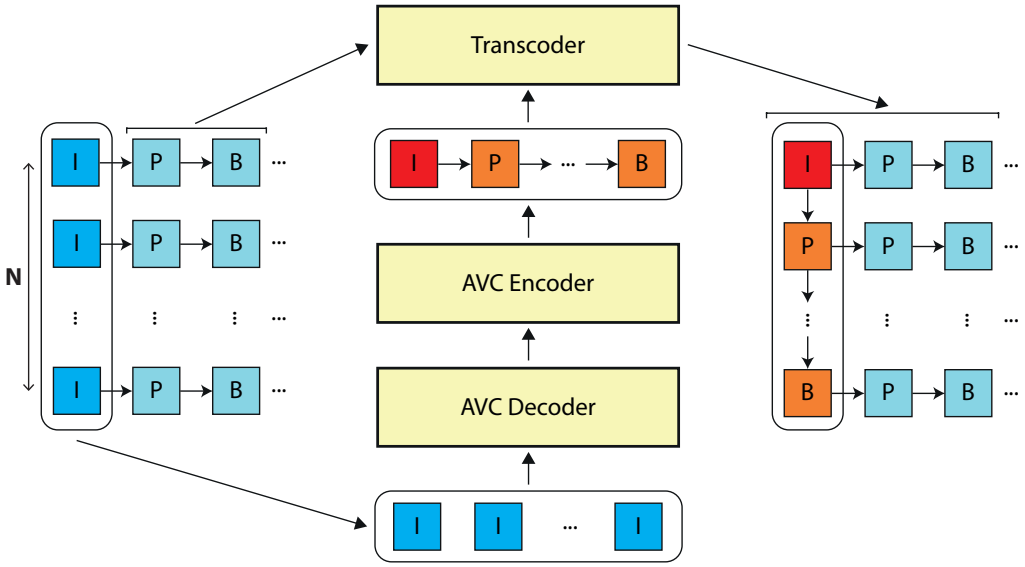



Figure 4.7.: Multiplexer scheme with added inter-view prediction. I-frames are copied from the input and encoded separately with a GOP length equal to the number of views. Their encoded versions are reintroduced into the stream in the transcoder.

In order to implement this approach, steps to introduce a kind of inter-view prediction are added in the multiplexer. By looking at how the inter-view prediction is defined in H.264/MVC, it can be seen that it functions like the inter-frame prediction, but accesses different views as references.

An overview of the whole encoder is given in Figure 4.7. The main idea here was to extract the I-frames from the incoming streams and decode them. This is possible because I-frames only use intra-frame prediction which makes them independent from the rest of the stream they belong to. Once they are decoded, a slightly modified conventional H.264/AVC encoder is configured to produce a stream with a GOP size equal to the number of input views. We elaborate on the required modifications to the encoder later in this section. The decoded frames are then fed into the encoder to produce a re-encoded version of the I-frames. The transcoder then takes this sequence and adds it to the output stream, in place of the I-frames. This process drastically increases the encoding efficiency, but lowers the overall quality slightly because the other frames of the views have been encoded with the original I-frames as reference. Since the re-encoded views are slightly different from the I-frames that were in their place before, the macroblocks which referred to the I-frames now produce a slightly different result than before. These errors propagate through the whole sequence, but even though they are usually too small to be noticeable, they can still be measured. While this idea seems to be trivial at first glance, issues become apparent upon closer inspection.

### Fixing Encoder Predictions

When the new frame sequence is transplanted back into the sequence, at first the references are all wrong because they point to other frames of the same view, instead of to the same frame in other views as intended. To correct this, reference list modification commands are added to the header of every slice of the transplanted sequence, to modify the reference

Figure 4.8.: Prediction errors after adding inter-view prediction.

list a decoder would create in such a way that it contains the correct frames in the right positions. Post-modification, it seems that the output is initially functioning as is should, however, there are some frames in the stream which present clusters of macroblocks showing severe errors (See Figure 4.8), even though the input streams can be decoded correctly. It is evident that those blocks reference the correct image but the motion vector is off by a rather large amount. Further investigation reveals that this only happens in B-frames, P-frames are never affected. The reason for this can be found in Section H.8.4 in the H.264 standard [31]:

> For the invocation of the MVC inter prediction and inter-view prediction process as specified in this clause, the inter-view reference components and interview only reference components that are included in the reference picture lists are considered as not being marked as "used for short-term reference" or "used for long-term reference".

Since the affected frames reference the newly encoded inter-view predicted frames, which are marked as inter-view and therefore neither as short-term nor long-term reference frames, this clause affects all B-frames in the replaced sequence. The different frame marking causes a decision in the motion vector derivation of skipped macroblocks in B-frames to go in the wrong direction and creates the artifacts visible in the output. Even though some short-term solutions (such as not using B-frames at all or disabling skipped macroblocks for B-frames in the encoder) were considered and implemented, they massively reduced the maximum achievable encoding performance of our encoding system as B-frames are more efficient than any other frame type and skipped macroblocks are the main reason for it.

The problem is caused by the *spatial direct luma motion vector prediction mode* applied to skipped macroblocks. Section 8.4.1.2.2 of the H.264 standard lists a set of conditions that have to be fulfilled to set the *colZeroFlag* to 1. This flag later determines whether the starting point of the motion vector is set to 0, or whether it is derived from the neighboring macroblock's motion vectors. The first condition is that the first image in the second

reference list is marked as a short-term reference. As discussed before, this can never be the case for our replaced frames but is possible in the encoder and this creates the discrepancy visible in the decoded frames. To fix this, the critical part of the encoder, which is used for the re-encoding, is identified and modified in such a way that it makes the same decision as the decoder which is going to process the final stream. It is a simple fix since it only involves a single decision, but fixes the problem for the replaced frames.

After this modification, the new inter-view predicted frames are decoded correctly, but there are still errors in non-modified frames, which reference the new frames. These errors are not caused by a discrepancy between AVC and MVC decoding, but by the fact that the re-encoded frames are not I-frames anymore and there are differences in the motion vector derivation when the co-located macroblock is intra-coded or not. In I-frames, all macroblocks are intra-coded, while in all other types there is an option for intra-coding, which is only used in rare cases. Modifying the encoder to use the correct macroblock prediction in critical places requires major work in the encoder as well as a very thorough analysis of all macroblocks in all frames of all views. The high complexity involved makes such an approach unfeasible for real-time applications. An easier solution with only a small impact on the encoding efficiency is to slightly adapt the GOP structure. As P-frames are not affected by the problem because they do not have a second reference list, they are used as anchor frames in the center of the GOP. Then the encoder is modified such that the frames which are going to be replaced are never in the position of the co-located frame. This is achieved with a simple modification of the second reference list of the slices in the encoder. For a GOP size of 8, the decoding order might be IPBbbBbb (the small b denotes B-frames which only reference other B-frames). Therefore, most frames are still B-frames which keeps the prediction efficient. With this change, all the errors disappear and the decoding is successful.

This implementation of the distributed encoder introduces a first version of inter-view prediction in the final output stream. The inter-view prediction is added to the stream during multiplexing and the implementation is based on the multiplexer presented in Section 4.3.1. Due to its implementation, there is a small decrease in overall quality but a significant increase in encoding efficiency. The complexity of the required operations is low enough to keep it feasible for interactive applications without specialized hardware.

### 4.3.3. Real Inter-View Prediction with Distributed Coding

In this section, a different approach to fast inter-view prediction is presented. While the previous attempt added inter-view prediction during multiplexing, it came with a slight drop in quality. To prevent this, the way to add the inter-view prediction is modified in such a way that transplanting the inter-view coded frames does not change the prediction in each view. Figure 4.9 gives an overview of the new structure. As a first step, the first frames of the GOPs from every view are collected and encoded. The encoded frames are distributed to the respective encoders for the rest of the frames and injected into the encoder state, so the following frames reference the correct encoded frame. The following frames are encoded normally and the resulting video streams are multiplexed with the method presented in Section 4.3.1. The inter-frame prediction is turned into inter-view prediction as described in Section 4.3.2. Those steps result in an H.264/MVC compatible video stream with real inter-view prediction.

Due to the complexities involved in extracting the encoded frames and the associated buffer from one encoder and injecting it into another, this version was never fully implemented. It requires an encoder-specific data dumper that extracts the frame from the decoded picture buffer, along with all associated frame data like frame type and saved information about

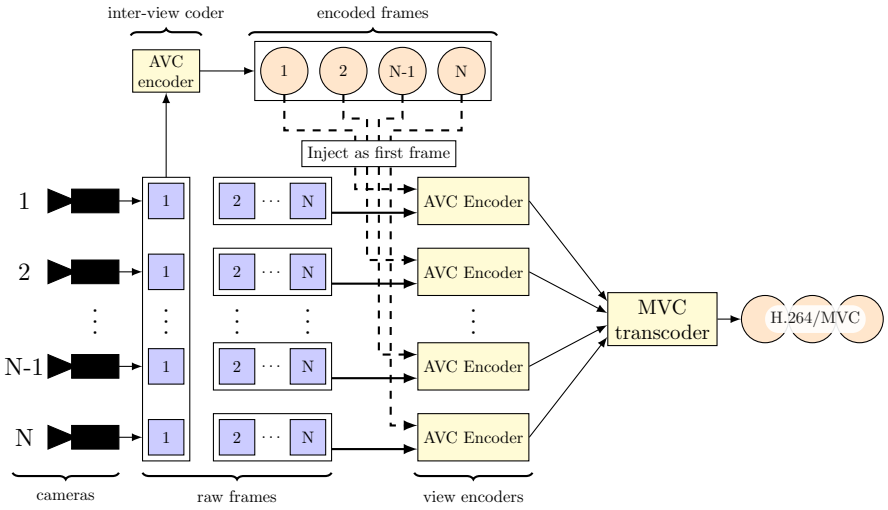



Figure 4.9.: Proposed structure for full inter-view injection. Future I-frames are intercepted and coded separately. Their encoded versions are injected into the view encoders as a virtual first frame, then they work normally.

the macroblocks. In the view pre-coders the frame data must be injected by replacing the respective data from the first frame with the data from the inter-view encoder, before the following frames are processed. If this is done correctly, the need for the second fix described in Section 4.3.2 becomes obsolete because all frames reference the correct data.

The major drawback of this approach is the requirement to transfer raw uncompressed frames from one encoder to another. On the other hand, there is no need to re-encode any frame which negates the quality drop from the earlier attempt. That procedure also removes the need for the mandatory P-frame in the GOP, which increases the encoding performance.

## 4.4. Evaluation

In the previous section, multiple approaches for the design of a distributed or hierarchical real-time encoder for multiview content were presented. In this section, the performance of these approaches is evaluated with respect to their speed, encoding performance and quality, and scalability.

### 4.4.1. Speed

It is important to note that, even though some more optimized solutions exist, especially for the multiplexing version, all measurements were taken using an implementation based on modified versions of the JM and JMVC reference software for better comparability. Every measurement was taken by applying the corresponding implementation to five views of the Ballet sequence [63], each with 100 frames and a resolution of 1024x768. The main advantage of our approach is the significant difference in the time that is required to create a standard-compliant H.264/MVC stream from a set of input views. Table 4.1 shows the result of the performance evaluation. It becomes immediately evident that the reference implementation of the H.264/MVC standard is by far the slowest solution, even though the

| | Encoding [s] | Transcoding [s] | Total [s] |
|---|---|---|---|
| **JMVC** | 27793 | 0 | 27793 |
| **1. Transcoder (no IV)** | 521 | 281 | 802 |
| **2. Transcoder (P only)** | 368 | 534 | 902 |
| **3. Transcoder** | 521 | 567 | 1088 |

Table 4.1.: Measured encoding times. The first transcoder is the multiplexer without any inter-view coding. The second transcoder has inter-view coding, but only uses I- and P-frames. The third transcoder is the final prototype with full frame-type and inter-view coding support.

| | View0 | View1 | View2 | View3 | View4 | Frame0 |
|---|---|---|---|---|---|---|
| **JMVC** | 447.52 | 431.68 | 457.47 | 415.82 | 433.64 | 799.67 |
| **1.Transcoder (no IV)** | +9% | +10% | +9% | +12% | +8% | +6% |
| **2.Transcoder (P only)** | +28% | +30% | +29% | +32% | +28% | -16% |
| **3.Transcoder** | +7% | +4% | +8% | +5% | +5% | -16% |

Table 4.2.: Average frame sizes in bytes for different views and transcoder versions.

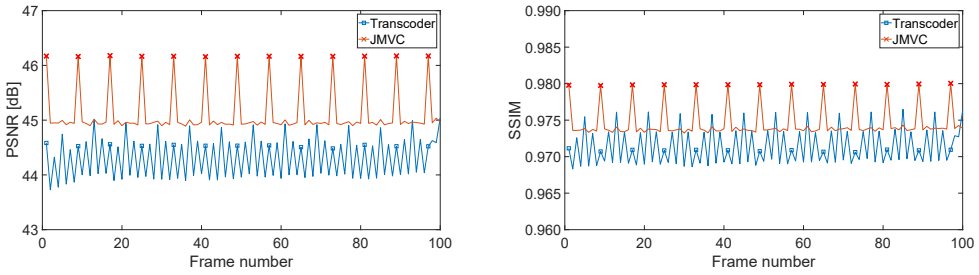
presented solutions are based on the same code base. Even the slowest transcoder version only requires 3.2% of the JMVC's time.

The significantly lower encoding time for the transcoder version, which does not use B-frames, stems from the fact that encoding P-frames is less complex than encoding B-frames because they have fewer references and therefore, it is not necessary for the encoder to consider so many neighbors for the prediction of the macroblocks. It is also interesting to see that the modified encoder in the last version requires the same amount of time as the encoder in the first version, despite the change introduced in Section 4.3.2. The transcoding time increases with every new iteration of the transcoder, the largest step being between the first and the second version. That can be explained by the addition of the re-encoding step in the second version. Decoding the frames is quite fast since only I-frames are decoded and they are the least complex to decode. Encoding involves all frame types which makes it more complex. Another portion of the transcoding time goes into the recalculation of the reference list of the new inter-view predicted frames, as explained earlier.

Overall, due to the fact that the sequence to encode is only 10 seconds long and has a frame rate of 10 frames per second, the results do not look very impressive and are still far off the real-time target. For the transcoder without inter-view interpolation, we have shown in [4] that switching to a more efficient implementation for the computationally expensive parts allows the approach to reach real-time speeds with performance to spare on hardware that is considered quite outdated by now. Since the transcoding was only slightly optimized and still used the code from JMVC to read and write the byte stream, there is even more potential for future improvements. Overall, it is very feasible that all presented approaches reach real-time performance when implemented using a more optimized platform for a moderate number of views. For a high number of views, new bottlenecks emerge, as discussed in Section 4.4.3.

### 4.4.2. Encoding Efficiency

Speed is the main focus in our approach, but it is worthless if the compression is not efficient. Table 4.2 compares the average frame sizes of the JMVC output to that of our transcoders.




(a) PSNR scores (b) SSIM scores

Figure 4.10.: Quality results for the encoded Ballet sequence [63] with interview support

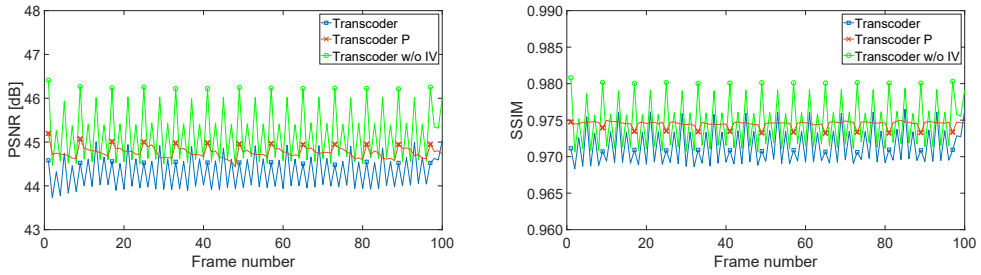
It shows that the frame sizes with the latest transcoder are only 4 to 8% bigger than those created by JMVC, even though our transcoder is forced to add a P-frame in the center of the GOP (see Section 4.3.2). The bigger size of the P-frames becomes obvious in the results for the second transcoder, whose frames are between 28 and 32 percent bigger than those of the reference implementation. The main reason for the lack of notable difference is the more efficient encoding of the first frames in the GOPs. The JM encoder encodes the frames in a way that they are up to 16 percent smaller in the byte stream. As the first frames in a GOP always have the best quality in the GOP, they are always bigger than the other frames, which makes their size difference even more significant.

The first transcoder with inter-view coding, which we introduced in Section 4.3.2, does not influence the size of the stream a lot. A comparison with the pure multiplexer using the exact same settings, only reduced the frame sizes by 40 bits on average.

Figure 4.10 shows the quality comparison of JMVC and the last transcoder for one of the views, which is the last in the prediction chain. Their frames are therefore most affected by the prediction errors, which we introduced by the re-encoding of the frames at the beginning of the GOPs. The quality loss we expected from the re-encoding of the GOP start frames and the prediction change becomes evident here. The transcoded stream has a PSNR value approximately one dB lower than the JMVC stream and an SSIM value which is approximately 0.05 lower. Considering the large speedup and the improvement in compression efficiency, such a drop in quality is acceptable. The result of the re-encoding itself can also be seen here, as the quality of the first frame in the GOP which usually has the highest quality, drops below the one of the center P-frame we introduced. In SSIM, it is even more evident where the first frame is just slightly better than the frames with the most prediction instances. This discrepancy can be explained by the blocky characteristic of encoding artifacts introduced by all MPEG video codecs, as they change the structure of the image and this is picked up by the SSIM.

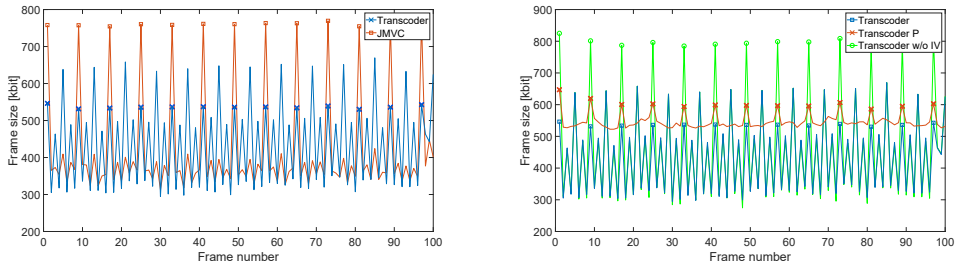
When comparing the different transcoders with each other in Figure 4.11, the version without inter-view prediction and the version with B-frames show a similar structure, which is simply shifted. This similarity in the graphs can be traced back again to the similar GOP structure and the added re-encoding step. The transcoder limited to P-frames shows a different pattern. The first frame has the highest quality (at least when measured with PSNR) and then the quality continually drops for the rest of the GOP because every new frame is predicted using the previous one. The low-quality value at the beginning of the GOP is again an effect of the slight block artifacts added by the re-encoding.

Looking at a comparison of the frame sizes in Figure 4.12, the re-encoded frames are




(a) PSNR scores (b) SSIM scores

Figure 4.11.: Quality results for the encoded Ballet sequence with different encoder versions.




(a) B-frames vs. JMVC (b) Encoder comparison

Figure 4.12.: Frame sizes for the encoded Ballet sequence.

about 30 percent smaller than their equivalent from the JMVC stream. The main reason is the loss of details in the second encoding step, which makes the frames easier to compress. The P-frames are approximately 50% bigger than their equivalent B-frames, as expected. The difference gives the appearance of the GOP from the transcoder being shifted by half a GOP size, but this is not the case. The size of the B-frames from the transcoder varies much more than those from the JVMC. This only shows that different encoders distribute the bits differently over the frames of a GOP. JM gives more bits to frames higher in the prediction hierarchy, while JMVC attempts to maintain a more static number of bits per frame.

A comparison between the different transcoders again confirms what we have seen before. The re-encoded frames are bigger than the original ones, while the other frames maintain their size after the transcoding. The P-frame-only version displays a higher average bitrate per frame, but also less variance, similar to the behavior in the quality.

### 4.4.3. Scalability

Given that the motivation behind the procedures presented in this chapter was the lack of available implementations of H.264/MVC for more than two views, it is essential to note how well the new approach scales with regard to the number of views to encode. The sequence used for evaluation only has a resolution of 1024x768. At this point in time, this resolution is considered very low, since commonly consumed videos currently shift slowly from FullHD to 4K resolutions and 8K is already on the horizon. Even though multiview codecs do not see a lot of use nowadays, this can change quickly, with the rise of lightfields. So here we

discuss what the limits are for the number of views, either defined by the standard or the capability to keep up with real-time speeds.

Table 4.3 is an extract of Table A-1 from the H.264 standard [31]. It defines the number of macroblocks per second a decoder must be able to process, so it can decode the video stream at the speed it is intended to be viewed. Additionally, it defines the memory a decoder requires (in macroblocks), so it can decode the stream. Inversely, those limits must be considered by the encoder to make sure a decoder capable of the respective level can process it.

To make the distributed encoder work properly with the camera array presented in Chapter 2, we have to check how many views with a resolution of 1920x1200 at 40 frames per second the encoder can handle. First, the number of macroblocks in one frame has to be calculated as

$$\left\lceil \frac{1920}{16} \right\rceil \cdot \left\lceil \frac{1200}{16} \right\rceil = 120 \cdot 75 = 9000 \tag{4.1}$$

This already forces us to use at least level 5 (see Table 4.3), since it is the first one that supports the required number of macroblocks per frame. As the current state of the implementation is still based in parts on JMVC, we are limited by its highest supported level, which is 5.1. The maximum number of views from our camera array that can be decoded in real-time with a level 5.1 capable H.264 decoder is

$$\left\lfloor \frac{983040 \frac{macroblocks}{s}}{9000 \frac{macroblocks}{frame} \cdot 40 \frac{frames}{s}} \right\rfloor = \lfloor 2.73 \rfloor = 2 \tag{4.2}$$

The effort for two views seems impractical, since the MVC coding becomes more efficient with more views.

In case real-time decoding is not crucial, or a faster decoder is available, another value in the table becomes the limiting factor, namely the *maximum decoded picture buffer size*. In Section H.10.2.1 of the H.264 standard, the maximum size of the decoded picture buffer is given as:

$$max\_dpb\_size = \min(dpb\_size(MaxDpsMbs), dpb\_size(NumViews)) \tag{4.3}$$

where

$$dpb\_size(MaxDpsMbs) = \frac{2 \cdot MaxDpbMbs}{PicWidthInMbs \cdot FrameHeightInMbs} \tag{4.4}$$

$$dpb\_size(NumViews) = \max(1, \lceil \log_2(NumViews) \rceil) \cdot 16 \tag{4.5}$$

Calculating those with the values from the camera array and level 5.1 gives:

$$dpb\_size(MaxDpsMbs) = \frac{2 \cdot 184320}{120 \cdot 75} = 40.96 \tag{4.6}$$

Since this value is independent of the number of views and the result of Equation 4.5 surpasses it for three views and more, Equation 4.3 is only defined by the result of Equation 4.6. Therefore, the maximum amount of pictures that can be in the decoded picture buffer at the same time is 40. Considering a common three-stage multiview prediction scheme with GOP size 8, the two previous frames of all views and the one currently being decoded have to be held in the buffer. This means the maximum frames in the buffer are $2 \cdot NumViews + 1$. Figure 4.2 in Section 4.2 might be helpful to understand this. In case the decoding order diverges from the display order and the decoded frame can not be displayed immediately, it

| Level number | Max macroblock processing rate (MB/s) | Max frame size (MBs) | Max decoded picture buffer size (MBs) |
|---|---|---|---|
| 1 | 1 485 | 99 | 396 |
| 1b | 1 485 | 99 | 396 |
| 1.1 | 3 000 | 396 | 900 |
| 1.2 | 6 000 | 396 | 2 376 |
| 1.3 | 11 880 | 396 | 2 376 |
| 2 | 11 880 | 396 | 2 376 |
| 2.1 | 19 800 | 792 | 4 752 |
| 2.2 | 20 250 | 1 620 | 8 100 |
| 3 | 40 500 | 1 620 | 8 100 |
| 3.1 | 108 000 | 3 600 | 18 000 |
| 3.2 | 216 000 | 5 120 | 20 480 |
| 4 | 245 760 | 8 192 | 32 768 |
| 4.1 | 245 760 | 8 192 | 32 768 |
| 4.2 | 522 240 | 8 704 | 34 816 |
| 5 | 589 824 | 22 080 | 110 400 |
| 5.1 | 983 040 | 36 864 | 184 320 |
| 5.2 | 2 073 600 | 36 864 | 184 320 |
| 6 | 4 177 920 | 139 264 | 696 320 |
| 6.1 | 8 355 840 | 139 264 | 696 320 |
| 6.2 | 16 711 680 | 139 264 | 696 320 |

Table 4.3.: Level limits for H.264 [31].

has to be stored until it is displayed. That increases the number of frames in the buffer to at most $3 \cdot NumViews + 1$. Solving the equation for the number of views gives the maximum number of views that can be included in the stream without violating the limits of the chosen level:

$$NumViews = \left\lfloor \frac{dpb - 1}{3} \right\rfloor = \left\lfloor \frac{40 - 1}{3} \right\rfloor = 13 \tag{4.7}$$

Consequently, to transmit the 64 views of the camera array, at least 5 streams are required. Depending on the application that uses the streams, a line-wise or column-wise grouping with 8 views each might be more efficient. For two-dimensional usage, a 3x3 or 3x4 subset can also be used, but in those cases, the views can not be distributed evenly over the streams. A different application-specific camera layout could solve the issue. A higher level, such as 6.2 which does not have encoder or decoder support at the moment, would increase the possible number of views to 51. While this is still not enough to include all cameras in the stream at once, it gives more possible options for view distribution.

Another limit to consider is the speed of the re-encoding step or the first encoding step required in Section 4.3.3. Since those are only handling one frame in time direction, independent of the number of frames in view direction, all their operations have to be performed within one frame duration to maintain real-time performance. For a frame rate of 40 frames per second and 64 views, the time budget for each frame is $\frac{1}{40 \cdot 64} s = 390 \mu s$ or the equivalent of 2560 frames per second at 1920x1200 resolution. Such a value is currently not

achievable by any CPU with the most optimized encoders or even the fastest GPUs available. Depending on the delay requirements of the final application, the timing requirements can be relaxed, if we assume that the encoders for every view are also running faster than real-time. In this case, the transcoder may take longer than one frame duration and as long as it finishes within a fraction of the GOP duration, the view precoders can still catch up. Hence, one GOP can still be processed in one GOP duration. This approach makes the limit more feasible, especially for longer GOPs, but still requires very powerful CPUs or GPUs. Only combining this relaxed timing requirement with a lower number of views, brings real-time performance into the reach of affordable hardware.

## 4.5. Open Issues

Even though our approach produces standard-compliant streams and real-time performance has been demonstrated at least for some of the implementations, there is still a number of issues that need to be addressed in the future if the system presented here is to be used in production environments:

### Optimized implementation

The biggest hindrance to better performance is the fact that all approaches are mostly based on unoptimized reference implementations of the respective standards. The simplicity of these implementations was quite useful for getting a grip on what needed to be changed to make them work and ease the implementation of those changes. Now that the changes are known, adding them to a more optimized encoder or decoder respectively would give much better performance with respect to speed. Since the changes we propose are rather simple, this could be done quickly if an optimized encoder/decoder pair with full multiview support was available. However, with the current state of available decoders, in addition to our modifications, the whole functionality required for MVC, needs to be added as well. Adding the functions to a highly optimized system while making sure the additions are as optimized as their environment would be a very lengthy process.

### Porting to newer standards

The approaches presented in this chapter are based on an extension to the original H.264 standard. Even though it has been updated until recently, it is foreseeable that most systems will move on to newer standards such as HEVC (H.265) or AV1 [123]. They promise a better encoding performance than H.264, namely requiring about half the bitrate for the same quality. While AV1 does not support multiple views in one stream yet, HEVC includes this functionality in the base standard. Since the overall structure of how the prediction works has not changed much when compared with H.264, porting the presented approaches to HEVC may make it more efficient and give it better chances of being adopted in the future.

### Optimize inter-view coding

All presented approaches are currently limited to using the first frame in the GOP for inter-view prediction. While this gives the biggest gain because the largest frames are replaced, using additional inter-view predicted frames in a GOP can increase the overall performance. Due to the structure of our approach, the position of these frames can not be determined dynamically. So to make the optimal choice for additional inter-view frames, a detailed analysis of the data rate distribution over the frames in the GOP, is required.

Additionally, more inter-view frames add harder limits to the speed requirements of each component, as they are required for certain prediction steps.

# 5. Optimized Streaming of Multiview Content

Even though multiview content can be compressed quite efficiently and even in real-time under certain conditions, as shown in Chapter 4, the required data rate can still be much higher than that of single view content. This increase in data rate makes it still prohibitive for live-streaming scenarios when the available bandwidth to the clients is limited. Unlike other approaches which only aim to have a single view at the receiver to show and only want to switch between the views within the stream [124], our goal is to deliver all views to the client to enable further processing such as view interpolation or depth estimation. Therefore, a solution that can switch quickly between views but only transmits one at a time does not work here and we must rely on a video codec with multiview support. These codecs are not supported by most common video players and encoder suites, which means they often only exist as slow reference implementations without automatic data rate or quality adjustments. Single view streaming codecs can only lower the required data rate for the transmission of a video by degrading the video quality, either by reducing the temporal or spatial resolution, so there is less content to be transmitted, or by increasing the quantization of the image contents so every image is represented by less data. Which one of these parameters takes precedence over the others heavily depends on the content and the viewing scenario [125]. While content with a lot of fast movements such as sports footage needs to retain a high frame rate, otherwise the footage will look choppy, such material can cope with less overall image detail because small details can not be perceived in fast-moving parts of the images. On the other hand, slower-moving footage needs the fine details while the lower frame rate might be missed by the viewer.

In this chapter, we explore how to optimize a video stream containing multiple views under the assumption that all views must be present at the receiver at the same time, that the available bandwidth on the channel is limited but known, and some kind of view interpolation algorithm is available for view reconstruction.

## 5.1. Concept

Multiview footage opens up a new dimension of parameters for data rate optimization. In combination with a real-time view interpolation algorithm, such as the one described in Chapter 7, complete views can be dropped from the transmission and the saved bandwidth can be redistributed over the remaining views. On the receiver, the missing views can be reconstructed by using a view interpolation algorithm if necessary. In case the quality or speed of the available view interpolation algorithms is insufficient, falling back to equally distributing the data rate between the views and transmitting them all is always an option. Figure 5.1 gives an overview of the steps included in such a streaming pipeline.

Choosing the optimal parameters is already complex for common encoders [126], so adding new parameters adds even more complexity. Since every step in the pipeline has its own set of parameters and output requirements, each one has to be set up to maximize the

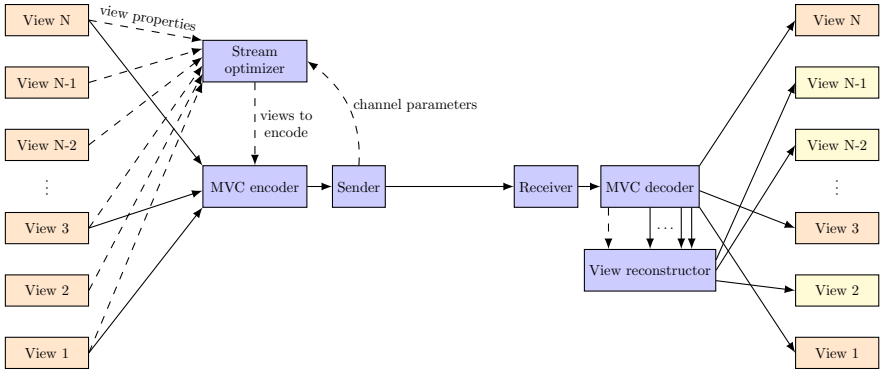



Figure 5.1.: Proposed multiview streaming pipeline.

respective output quality while fulfilling the requirements of the next step. In order to be able to predict the optimal parameters for every step, the influence of the parameters on the quality at the end of the pipeline has to be determined. To be able to efficiently combine the quality gains and losses in each step, this influence has to be measured using a single metric, since the scales and measurements of different metrics are almost always incompatible [127]. The step with the most complex analysis of quality vs. data rate is the video encoding step at the beginning of the pipeline. Fortunately, this work has already been done for the common video encoders [128] during their development and shortly after, but only with respect to the PSNR. While the choice of PSNR as their metric is very common for video and image decoders which try to keep as much image information as possible, it is not ideal for view interpolation applications as shown in Chapter 6. For the other steps, the evaluation of the influence of a change in the used data rate on the quality of the final result needs to be done by us. Once the influence is known for all steps, the final quality can be predicted for a given available data rate and a set of parameters for the intermediate steps. From there, it becomes an optimization problem to find the parameters which maximize the quality of the final images.

View interpolation results often fail to score high PSNR values even for images that look good to human observers. Even though we created and published a novel quality metric that tries to solve this issue while staying mostly compatible with PSNR (See Chapter 6), it was decided to mostly use SSIM [129], because it is far more well known and better accepted in the research community.

Starting at the end of the pipeline, the average quality $Q$ of the views received on a client device can be defined as

$$Q = \frac{Q_E \cdot m + Q_R \cdot (n - m)}{n} \tag{5.1}$$

where $Q_E$ is the quality of an encoded view in the stream, $m$ is the number of encoded views and $n$ is the number of views expected at the receiver. $Q_R$ denotes the expected quality of a reconstructed view. Since the view interpolation algorithms used for reconstruction only have the encoded views available as input, it is safe to assume that the quality of their output is never higher than the quality of the encoded views and strongly influenced by it. $Q_R$ in Formula 5.1 can be substituted with $c \cdot Q_E$ where $c$ is an arbitrary term describing a characteristic function which describes the view interpolation algorithms ability to recreate

the missing views from the input material, resulting in

$$Q = \frac{Q_E \cdot m + c \cdot Q_E \cdot (n - m)}{n}. \tag{5.2}$$

In Pascal Straub's thesis [130], we found that the most influential parameter for this function is the ratio of pixels in the view to be reconstructed, which are also visible in the remaining encoded views. Following Zhang *et al.* [13], who used that coverage to optimize the distribution of the movable cameras in their camera array for maximum image quality, we used the same definition from Mavrinac *et al.* [131]. They define the covered points in multi-camera systems $C$, as the points that are included in four sets: $C_V$ is the set of fully or partially visible points, $C_R$ denotes the points covered by a sufficient number of pixels, $C_F$ includes all points in focus and $C_D$ contains visible points based on the direction of the camera relative to the scene.

$$C = C_V \cap C_R \cap C_F \cap C_D \tag{5.3}$$

Based on the image data and common assumptions about camera arrays, it is possible to simplify the function. Since we are only interested in scene points with the size of a pixel and camera arrays usually consist of similar cameras which are positioned roughly at the same distance from the scene, all important points in the scene should have sufficient resolution coverage, therefore, $C_R$ contains all scene points and can be ignored. The same holds true for the $C_F$ as the focus is highly dependent on the camera's distance to the scene, the focus setting of the lens, and the chosen aperture. If those are chosen well, the whole scene should be in focus for all cameras and $C_F$ does not influence the outcome of the formula anymore. As a result, without those two sets, the content of the covered points is only based on the direction of the camera $C_D$ and obstructions in the scene which dictate the visibility of point $C_V$. This coverage can be measured by determining whether the points in a single image have correspondences in color and depth in the neighboring images, similar to what the plane-sweeping approach does in Section 7.3.

While the relative positions and rotations of the cameras are static during a scene, the visibility can vary greatly depending on how the objects in the scene move. An analysis of the coverage for every single frame is possible, but it is not feasible for longer scenes or large material libraries. Additionally, the parameters of a transmission stream should not be changed after every single frame, and even if it would be done, the encoding would be highly inefficient, because the prediction structures in the codec can not be used properly (compare with Chapter 4). To allow for a high efficiency in the encoder, uniform parameters for a whole sequence are required. By choosing a fitting characteristic function for the relationship between the coverage and the expected quality of reconstruction, it becomes possible to determine the views with the best quality in case of a reconstruction based on the coverage from the remaining views. Since no reconstruction algorithm can avoid artifacts in every possible case, there will always be certain cases and scene setups where it behaves sub-optimally. Considering that the characteristic curve is probably determined for a favorable setup and content, we can assume the approximation of $Q_R$ using the characteristic polynomial $c$ in $c \cdot Q_E$ is merely an upper bound. To get a more precise value for the actual quality, more parameters have to be added to the characteristic function, such as the relative angle and the overall distance to the virtual view.

Under the assumption that neither the number of available views at the end of the pipeline nor the resolution of these images can change, we can determine the minimal quantization parameter (QP) for the encoder to make all encoded views fit into the limited data rate

**Input:** views
**Input:** available data rate on channel
**Input:** characteristic function for view interpolation

Determine QP for pure encoding
Calculate quality of pure encoding
new_config = Config for pure encoding
**repeat**
  best_config = new_config

  Calculate coverage for all remaining views
  Choose view with best coverage for removal

  Determine QP for remaining views
  Calculate quality of encoding
  Calculate quality of reconstructed views

  new_config = Config with reconstruction
**until** $Quality(new\_config) \leq Quality(best\_config)$

**Output:** best_config

**Algorithm 5.1:** Optimization procedure for the streaming parameters for a known video sequence and channel characteristics.

on the channel. From these parameters, we can get a base value for the received quality when all views are encoded and none are reconstructed. By determining the views with the highest coverage and therefore the highest expected quality after reconstruction, we choose the views to drop from the transmission. Then we recalculate the possible QP for the remaining views and their respective quality. Based on that, the characteristic function gives the quality for the reconstructed view and with Equation 5.2 we get the overall average view quality. Due to the potentially lower QP for the encoded views, their higher quality can compensate for the quality drop caused by the reconstruction and the average quality can become higher than before.

If the quality is higher, we again remove the views with the highest coverage and repeat the calculations. We do this until the quality value is lower than the one before, meaning that the reachable peak quality has been found as well as the required parameters to achieve it. The whole procedure is summarized in Algorithm 5.1.

## 5.2. Implementation

To evaluate the concept, we created a simple view interpolation algorithm based on Zhang's algorithm [13]. We call it simple because it is only intended to be used with material that provides depth maps for all views and it can only recreate views in positions of existing cameras. With good depth maps, it is still capable of producing good results without too much complexity for depth estimation.

Determining the characteristic curve for this algorithm was the next step. For analy-

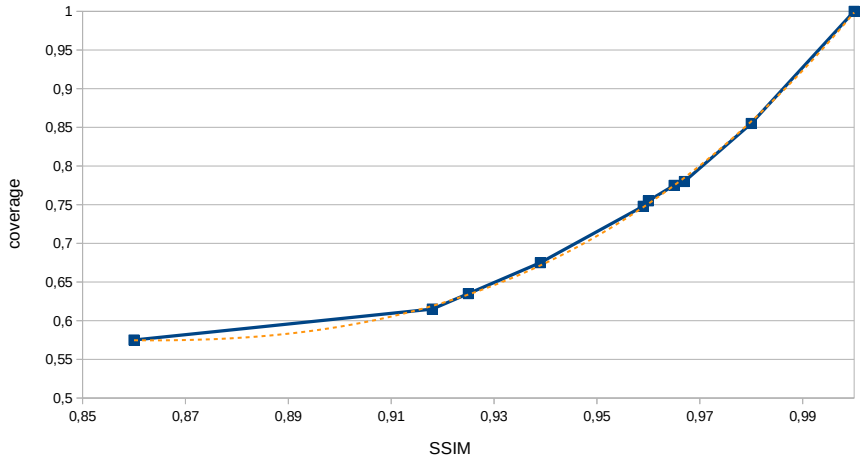



Figure 5.2.: Characteristic curve for the tested view interpolation with polynomial regression for the Ballet scene [63].

sis, we used the Ballet scene [63], removed different cameras and measured the coverage of a chosen view as well as the achieved quality compared to an encoded version of that view for every frame in the sequence, and averaged the results. The points with the coverage on one axis and the quality on the other axis are plotted in Figure 5.2. The dotted line represents a polynomial regression of degree three for the given data. Even though it is a good fit, it is neither optimal nor sufficient for large spread use, as discussed in Section 5.3, but serves as a good example for the following steps.

For the calculation of the optimal QP for a given number of views, we assume that the average data rate required of an additional view is a fixed fraction of the data rate required for the base view. This assumption simplifies the data rate estimation of the H.264/MVC stream down to that of a single view H.264/AVC stream and a well-chosen multiplication factor. As many heavily optimized encoders are available for that kind of stream, the optimal solution can even be found by a greedy search for a reasonable range of values for QP. Using that approach, we overestimate the real data rate consistently by 10-15% for QP values between 20 and 40, which are the most interesting for our scenario. Since it is fairly consistent, it can be easily compensated for with an additional scaling factor.

Combining that way of finding the best possible QP with the quality assessment formula from Section 5.1, finalizes our basic reference implementation of the concept. Table 5.1 lists the results for a set of scenarios in which the Ballet scene must be transmitted using varied data rate limits. From the SSIM values of the final quality, we can see that for low data rates up to 1Mbit/s there are slight improvements in average quality when a number of views are reconstructed at the receiver instead of being transmitted. For the higher data rates, the algorithm chooses to transmit all views, because quality drop from reconstructed views compared to encoded ones with a lower QP would be too high.

## 5.3. Evaluation

After the first experiments showed that dropping views from a multiview sequence and reconstructing them can be beneficial for the overall stream quality, further evaluations were

| $d_{target}[kbits/s]$ | $d[kbits/s]$ | $m$ | $QP$ | $Q$ | $Q_E$ |
|---|---|---|---|---|---|
| 500 | 468 | 5 | 36 | 0.920 | 0.910 |
| 800 | 767 | 6 | 33 | 0.929 | 0.928 |
| 1000 | 990 | 7 | 32 | 0.934 | 0.933 |
| 2000 | 1998 | 8 | 27 | 0.946 | 0.946 |
| 5000 | 4276 | 8 | 22 | 0.952 | 0.952 |

Table 5.1.: Results of the optimization procedure listing the target data rate $d_{target}$, the used data rate $d$, the number of encoded views $m$, the QP used for encoding, the achieved average quality after optimization $Q$ and the quality achieved by complete encoding $Q_E$.

| $d_{target}[kbits/s]$ | $d[kbits/s]$ | $m$ | $QP$ | $Q$ | $Q_E$ |
|---|---|---|---|---|---|
| 500 | 478 | 4 | 36 | 0.895 | 0.883 |
| 800 | 784 | 6 | 35 | 0.902 | 0.898 |
| 1000 | 979 | 6 | 33 | 0.907 | 0.905 |
| 2000 | 1869 | 8 | 30 | 0.921 | 0.921 |

Table 5.2.: Results of the optimization procedure for the Breakdance scene with the characteristic function from the Ballet scene.

done. In the previous section, we only tested the process using the same scene which we also used to determine the characteristic function of the interpolation algorithm. Validating it against different scenes is necessary to determine whether one set of parameters can be applied to multiple scenes or whether it has to be recalculated for every input.

Table 5.2 shows the experimental results when the characteristic function determined for the Ballet Sequence is applied to the Breakdance sequence with more dynamic movement. The results for the same data rates as in Table 5.1 are different, even though the same algorithm and the same parameters are applied, indicating the different coverage has a significant influence on the overall outcome of the optimization. That it works nearly as expected comes as little surprise, as the two scenes are closely related, use the same camera setup and the scene composition is fairly similar, apart from the different speeds and amounts of motion.

We were aware that this type of verification only has limited value, but at the time the search for further viable test scenes yielded no results. Most examples only consisted of a single frame, unsuitable for proper video encoding. Others only had very low-resolution images, far below the Ballet scenes 1024x768, or only two or three views. Those did not require low-quality encoding even for very restricted channels or did not offer any meaningful choice for the views to reconstruct. The few remaining candidates, such as the sequences from the MPEG-FTV project from Nagoya university[1], did not provide any or enough depth data for our test algorithm to work properly. The scarcity of available samples was one of the major reasons for the construction of our own camera array from Chapter 2 to be able to create our own test sequences.

Until now all tests only show slight improvements in data rate ranges far below anything that would be deemed suitable for transmitting eight views containing 6.3 Megapixels per frame. Simulating a better view interpolation algorithm by adjusting the characteristic

[1] `https://www.fujii.nuee.nagoya-u.ac.jp/multiview-data`

| $d_{target}[kbits/s]$ | $d[kbits/s]$ | $m$ | $QP$ | $Q$ | $Q_E$ |
|---|---|---|---|---|---|
| 5000 | 4743 | 6 | 20 | 0.953 | 0.952 |
| 8000 | 7268 | 7 | 19 | 0.954 | 0.953 |
| 10000 | 9668 | 6 | 17 | 0.956 | 0.955 |

Table 5.3.: Results of the optimization procedure for a simulated better reconstruction algorithm applied to the Ballet scene.

function to report slightly higher values, we achieved the results in Table 5.3. It shows that with an improved view interpolation, the proposed pipeline can provide quality improvements beyond 10 Mbit/s which is far more reasonable for the amount of data to be transmitted.

The characteristic function we have chosen is too simple for more general approaches. It only considers the coverage of the views, which already integrates the relative angle and camera baseline to a certain degree due to the way it is calculated, but does not integrate other parameters such as the overall complexity of the scene or the amount of visible movement. Since calculating the function for every transmitted scene can become very expensive, depending on how many input parameters are used for the function and how many scenes it has to be done for, we propose the following in case of a wide-spread use: the scenes should be sorted into categories for a set of different complexity classes from slow-moving near-planar scenes to fast-moving scenes with a long distance between the closest and the farthest objects in the scene. For each category, a representative scene is chosen for which the characteristic function is calculated. When a scene is transmitted, its similarity to the available categories is determined. Then either the function from the closest category is used or an interpolated version between the closest categories is created, depending on the sparsity of the chosen categories. With this approach, a function close to the actual behavior can be used in each transmission, without having to do a complete recalculation for each scene. The quality of that prediction can be controlled by the size of the category set and how its members are chosen.

While this work proves that there are benefits to having a view interpolation step in the transmission pipeline for multiview video and provided the foundation for a well-accepted publication [6], its overall impact is rather small until now. The combination of view interpolation [132, 133] and compression using existing video codecs [134, 135] has already been attempted and published multiple times [136, 137], but the research community for multiview video transmission was never huge and its main interest moved to algorithms for lightfields since then.

In our opinion, a likely reason why this approach became more popular in lightfields than in multiview video is based on the fact that the camera baselines in lightfields are commonly far smaller than those of multiview video. This camera layout automatically leads to a higher coverage between views and therefore, a higher base reconstruction quality, which makes the approach usable for a wider range of data rates. Additionally, lightfield processing often requires a large amount of computational power, in which an added view interpolation step does not play a significant role. The higher number of views and the additional amount of data to transfer also makes efficient transmission schemes more relevant than for multi-video which could still be transferred with already available approaches over common network connections (as shown in Chapter 4).

# 6. Quality Metrics for Interpolated Views

Whenever the quality of an image needs to be measured, an appropriate metric is required. Even though there is a plethora of metrics available for many different applications and image types [138, 139, 140], sometimes it is necessary to use a specific metric for a case in which it behaves sub-optimally. An example for such an application is described in Chapter 5, in which results from earlier studies were combined with new findings to predict the output quality of a novel multiview streaming pipeline. Specifically, existing video encoders have to be coupled with novel view interpolation techniques, and their combined performance has to be predicted to choose a set of optimal encoding parameters. Most media encoder/decoder combinations measure the quality of the reconstructed data using PSNR. The resulting value tells them how accurate the original data is being reconstructed from the encoded data stream. While those measurements help to quantify the overall loss of precision in the data, it does not necessarily represent the perceived quality as judged by a human observer [141]. That is especially true for certain image generation or reconstruction methods, as shown in Section 6.1. If the observed quality is more important than the data accuracy, different metrics have to be used, but combining the results from different metrics into a single coherent result is often impossible due to different scales and units in the results. Therefore, it is necessary to use a single metric throughout the whole system, which means the results are not representing the real quality for certain steps.

In this chapter, we describe an application-specific way to adapt existing quality metrics to remedy this discrepancy. By analyzing the causes for the low scores for one application, we compensate for them so that the results of the metric are closer to the Mean Opinion Score(MOS) while maintaining its other characteristics. Sections 6.2 and 6.3 describe how we achieve this feat without modifying the original metric itself and the implementation in our particular case.

## 6.1. Background

In this chapter, the advantages and disadvantages of quality metrics are discussed. They are often compared to those of PSNR or Peak Signal to Noise Ratio, which is a common metric for all situations that need to measure the precision of a transmission or reconstruction of data. Here, PSNR is only discussed in the context of images and video. For an image with a single channel, it is defined as

$$\text{PSNR} = 10 \cdot \log_{10} \left( \frac{\text{MAX}^2}{MSE} \right) dB \tag{6.1}$$

$$= 20 \cdot \log_{10} \left( \frac{\text{MAX}}{\sqrt{MSE}} \right) dB \tag{6.2}$$

where MAX is the maximum value a pixel can represent and

$$MSE = \frac{1}{x \cdot y} \cdot \sum_{i=0}^{x-1} \sum_{j=0}^{y-1} [I(i,j) - N(i,j)]^2 \tag{6.3}$$

(a) Reference image (no distortion)

(b) Salt&Pepper noise 3% (29.03dB)

(c) Gaussian noise Variance=10 (26.56dB)

(d) JPEG quality=10 (26.78dB)

(e) JPEG quality=40 (31.24dB)

(f) JPEG quality=80 (35.75dB)

Figure 6.1.: Noise and compression influence on PSNR score for a photograph with 640x480 pixels.

with $x$ and $y$ as the resolution of the tested image in x- and y-direction, $I$ as the reference image and $N$ as the noisy or processed image to test. $I(i,j)$ stands for the pixel value of image $I$ at the position $x, y$. $N(i,j)$ is the equivalent for the tested image.

For color images, which usually consist of three distinct color channels, there are two options for applying PSNR to them. The first and more common one is to apply the metric to a grayscale version of the image or just the Y-channel in case the image is in the YUV format. Since the quality of the color components is usually higher than that of the gray/brightness component, only analyzing the Y-channel gives a good lower bound of the overall quality. The second option is to combine the measurements as follows:

$$\mathrm{PSNR}_{\mathrm{COMB}} = \frac{\mathrm{PSNR}_1 + \mathrm{PSNR}_2 + \mathrm{PSNR}_3}{3} \tag{6.4}$$

Because this formula weighs all channels equally, it is only used for image in the RGB or CIE color space.

Due to its simplicity, it can be calculated very quickly but does not always match the mean opinion score closely [142, 127]. In Figure 6.1, a set of images with different noise types and levels of JPEG compression are shown with their respective PSNR scores. There, the disparity between the scores and the perceived quality starts to become evident. Even though the images in Figures 6.1d and 6.1e have nearly the same difference in measured quality as in Figures 6.1e and 6.1f, the change is far more visible for the first pair than for the second. With the Mean Square Error (MSE) as the main component, the result of PSNR is only based on a direct pixel comparison, and whether those differences are visible or not is not considered. The mapping given in Table 6.2 between PSNR and MOS can be used to somewhat determine how the results of PSNR can be interpreted.

| PSNR [dB] | MOS | Impairment |
|---|---|---|
| $> 37$ | 5 (Excellent) | Imperceptible |
| 31 - 37 | 4 (Good) | Perceptible (not annoying) |
| 25 - 31 | 3 (Fair) | Slightly annoying |
| 20 - 25 | 2 (Poor) | Annoying |
| $< 20$ | 1 (Bad) | Very annoying |

Figure 6.2.: Mapping between PSNR and MOS according to [143] and impairment descriptions from [144].

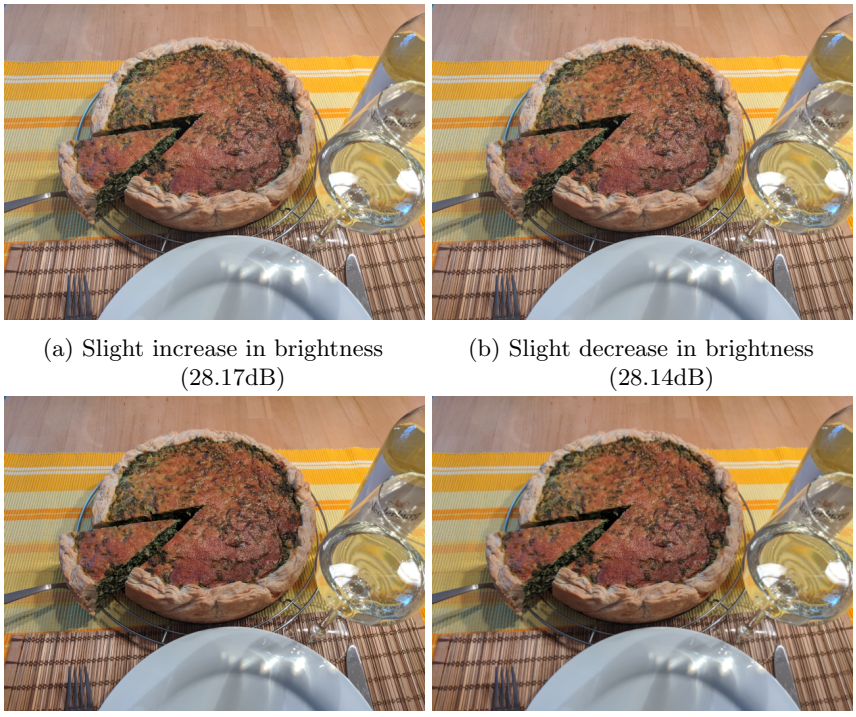
The errors introduced into images by reconstructing them with view interpolation algorithms look different to common noise patterns or the artifacts from block-based compression algorithms. Noise errors mostly play a role when natural camera noise existing in the input data is reduced by averaging the color data from more than one image before using it for reconstruction. Due to its inherent randomness, the noise in the reference image can never be reconstructed properly and therefore, lowers the measured quality.

For reconstruction, information of other images is combined based on some form of scene geometry, often depth or disparity, by projecting it into the scene space and back to the camera in question. Errors in depth estimation or even just quantization steps can move image sections away from their intended position. Image portions with the same depth move in a similar fashion, but the direction and distance of the movement can change depending on the value of the error. Since this type of error involves large sections of an image, even small shifts of less than a pixel can have a severe influence on the PSNR score. The example in Figure 6.3c is shifted by just one pixel to the right. Such a shift is nearly invisible to a human observer, especially in high-resolution images, as long as the images are not directly shown after each other in the same position. Nevertheless, the quality score assigned by the PSNR is mediocre at best.

A second common problem comes from the camera parameters used in the projections. An offset in the principal point can move the whole image, similar to the depth errors, but moves the whole image equally. When the field of view is off by even a small fraction, the projected image can be too small or too big for the image area of the virtual camera. Smaller images result in easily visible borders of empty image space. On the other hand, the effects of larger images are nearly invisible. The slight zoom effect grows from the focal point towards the edges. Again, the impact of these effects on the PSNR score is significant, proven by Figure 6.3d.

If the reconstruction algorithm uses data from multiple images to reconstruct a certain portion of the image, it is very important to weigh the different parts properly. Errors in the sum of the weighting factors can cause the result to become slightly brighter or darker. Since the precision for the calculation of the weighting factors is commonly more than high enough to keep the error below one step in the available color values, the calculation of the weighting factors usually does not cause measurable errors. Much more probable is a color or brightness mismatch between the different views in the input data. When such errors appear, the whole image might be too bright or too dark compared to its reference. The minimal, nearly undetectable brightness changes in Figures 6.3a and 6.3b give a feeling of how severe the influence is on the final quality score.

The last class of errors commonly appearing in reconstructed images are "obvious" image artifacts. Those appear when the coordinate projection goes completely wrong in at least one of the input images due to failed depth estimation or other causes. In the final image,



(a) Slight increase in brightness (28.17dB)

(b) Slight decrease in brightness (28.14dB)

(c) Shift by one pixel horizontally (23.30dB)

(d) Zoom shifting one pixel over the image border (28.82dB)

Figure 6.3.: Influence of different error types to the PSNR score.

these are visible as sections of colors appearing in places where they should not be and are often clearly separated from the surrounding areas by distinct borders. Since they do not occupy large portions of the image in general, their influence on the quality score depends heavily on their size and content. Unlike the error types discussed before, these errors can be considered to be preventable as they can often be traced back to errors in depth or camera calibration.

## 6.2. Concept

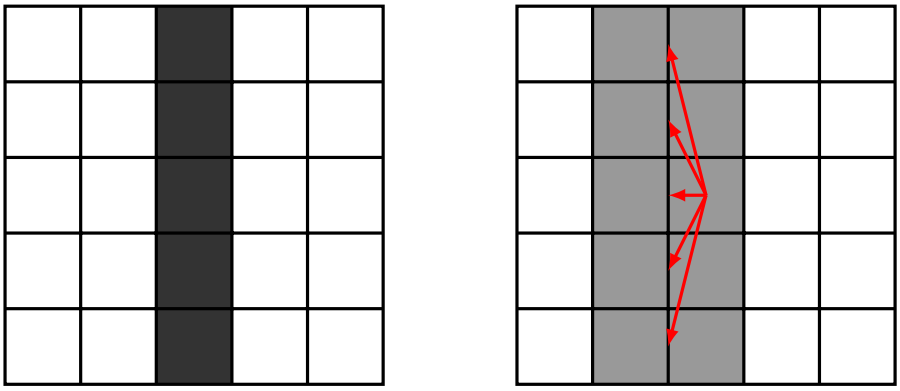
In addition to a better match of the resulting score with the MOS, the main requirement for this new metric is its compatibility with the traditional PSNR. To achieve this, the new metric needs to return values in the same range and on the same scale as PSNR. Our solution is to use an existing implementation of the metric and adapt it, in such a way that it ignores the existing but invisible errors while judging everything else as before. As shown in Section 6.1, the inner workings of PSNR are rather simple and therefore, do not offer many possibilities for adaptation without changing it completely. Due to the low complexity of the metric and the high amount of details in the tested images, we identified the input side of the metric as the obvious place for our changes.

As a full reference metric, PSNR requires two input images: one image to test and a reference image to test against. Since we assume the tested image has a high perceived quality, we know there is a high similarity between the two images and the tested image only contains the different error types the view interpolation produced. It is obvious in Figure 6.3 that the different kinds of errors the view interpolation typically causes have different effects on the PSNR score and those results do not necessarily correlate with their influence on the perceived quality. The image artifacts are a type of error that can be traced back to wrong texture assignment or false depth values. Artifacts in mostly untextured areas have nearly no impact on the end result, whereas the same errors in textured areas have an impact roughly proportional to their size and the perceived quality. Their influence should still be represented in the result of the new metric.

Zoom and shift errors are much less visible than the artifacts but degrade the measured quality even more. They are usually caused by very small deviations from the correct values for the field of view or depth. Even though it can be difficult for a human observer to detect them, the fact that they often cover large areas of the image means they degrade the measured quality a lot.

Sub-pixel shifts smear the information of a single pixel over adjacent ones. Detecting the direction and distance of a shift based on a single line can be quite hard. Figure 6.4 shows only a subset of possible shifts that create the given result from the original data. Determining the correct one is impossible without taking information from the surrounding area.

Now that the error types with the biggest impact on PSNR, but the smallest deviation in the underlying data, are identified, the remaining open question is how to make the applied quality metric ignore those errors. Removing them from the image to test, is a very complex task since the different error types can overlap, and deciding what effect comes from which error type is nearly impossible. Adding the error of a certain type to the reference is far simpler because in this case, we can clearly define the region and the error type beforehand so there is no guesswork while manipulating the image. Therefore, detecting those parameters in the tested image becomes the hardest task. Since shift and zoom effects both move pixels within a region, either by a nearly constant amount or by a



(a) original line (b) line after half-pixel shift

Figure 6.4.: Effect of sub-pixel shifts on borders. The exact shift can not be easily defined without taking a larger neighborhood into account.

continuously increasing value towards the edges of the image, they differ clearly from the other image artifacts that copy image contents from neighboring regions and often show clear edges around the affected region. How we implemented the detection and distinction of these errors is described in Section 6.3.

When the parameters of the errors are determined, we apply them to the reference image to create a new reference. As this new reference now includes the errors which we do not want to influence the measured quality, and they are the same as in the tested image, the errors are now effectively ignored by the image metric. This way the whole approach becomes fairly flexible with respect to problematic error types and even the metric to be used. The influence of the error types needs to be re-evaluated for different metrics but the main approach stays the same. In this thesis, we mainly focus on PSNR.

## 6.3. Implementation

The shifted regions and zoom effects can all ultimately be reduced to a set of connected pixels which are moved in the same direction compared to the original. To make the approach work, we need to know exactly which pixels contain the errors and how far in which direction they moved. In Section 6.1 we found that these shifts do not necessarily happen in full pixel steps, in most cases it is between two integer values and often even less than a single pixel. One approach to detect those continuous and minute shifts between images is optical flow [145]. Algorithms to calculate the optical flow between images are readily available due to their numerous uses in motion detection [146, 147], object separation [148, 149], and robotics [150]. Optical flow algorithms are grouped into two categories, namely sparse and dense algorithms. Since we need shift information for every pixel in the image, the only choice for our approach are the dense algorithms.

After evaluating multiple algorithms, we decided to use the "Dual TV L1" [151, 152] algorithm. When used with appropriate parameters, its results show clear object borders, comparably uniform movements within an object, and sufficient precision for our purposes. The parameters that work well for our application are $\tau = 0.2$, $\lambda = 0.2$, $\theta = 0.2$, $\epsilon = 0.006$, using 5 scales, 5 warps and 500 iterations. In Figure 6.5, we show the performance of the shift detection on a number of scenes often used in view synthesis experiments. For testing,

(a) Reference images (b) Shift regions (c) Detected shifts

Figure 6.5.: Examples for shift detection with random synthetic shifts.

every color in the hand-drawn masks was assigned a random shift in x- and y-direction between $-2$ and 2. Additionally, a random zoom such that a maximum of two pixels were cut from the image around the borders could also be added, but for better visibility of the shifted regions, the zoom effect was disabled in the examples presented here. The pixels in the reference image are then shifted according to the mask, to create a warped image. Using the optical flow algorithm to find the differences between the warped image and the reference, results in the detected shifts in the right-most column.

When it comes to errors, which we do not want to include in the new reference, the results from the optical flow algorithm are nearly perfect for our application. The center image of Figure 6.6 contains a number of typical artifacts that we often see in the results of our view interpolation algorithm. They look like parts of a leaf floating between the actual leaves and they also get their texture from the neighboring regions. In the image on the right, we can see that the first two artifacts are clearly depicted in the calculated flow map. From the third artifact, only small portions are visible in the flow map, while the remaining area shows values in the range of the surrounding area. Since all flow values for the artifacts greatly differ from their neighbors in amplitude as well as direction, they can be easily detected. For our implementation, we chose a simple threshold for the detection and set the shift in the affected regions to zero when creating the new references. Since the regions covered by the artifacts are already far off compared to the reference, it was decided that setting their shift to zero instead of a value similar to the surrounding area does not make a difference in the end result big enough to warrant the increased effort for proper inpainting.

Even though the detection is not perfect and sometimes the regions do not match up exactly with the mask and show some variations in shift amplitude, using it to create our

(a) Reference (b) Interpolated view (c) Artifacts in flow

Figure 6.6.: Optical flow results for interpolation artifacts.

| Scene name | PSNR [dB] | | MSE | | SSIM | |
|---|---|---|---|---|---|---|
| | old | new | old | new | old | new |
| Teddy | 36.916 | 51.226 | 13.331 | 0.4942 | 0.9537 | 0.9983 |
| Cones | 32.315 | 46.872 | 38.462 | 1.3468 | 0.9214 | 0.9968 |
| Breakdance | 38.198 | 51.303 | 9.923 | 0.4854 | 0.9443 | 0.9977 |

Table 6.1.: Metric results for the examples from Figure 6.5.

new references still produces expected results, namely a higher but not perfect quality score representing the perceived quality more closely. Precise values can be found in Table 6.1.

In the end, a small executable was created that gets the image to test and the original reference image as inputs and returns a new reference that contains the changes discussed above. It is programmed in C++ and is mostly based on OpenCV, which allows it to be compiled quickly on many operating systems. Applying the presented approach to any image-based full reference quality measurement is straightforward. First, the image to test and the reference are processed through the executable and the new reference is saved. Second, the original unchanged quality metric is used to measure the quality between the image to test and the new reference. The overall approach can be applied to a wide variety of metrics if needed, but the creation of the new reference has to be adapted to the new metric, as it is currently only optimized for PSNR.

## 6.4. Evaluation

For the evaluation of the approach, we compared a number of interpolated views with high perceived quality. Figure 6.7 shows a couple of real interpolated images from the set we tested. Apart from some fine details, such as the wooden sticks and a small number of cone tips in the Cones scene and small deviations on the grid pattern in the background of the Teddy scene, the reconstruction has nearly no visible errors. Overall, it performs similarly to the synthetic tests we performed during the development of the new measurement approach, but the margin of improvement is smaller. This difference can be explained by the other subtle differences which appear in real interpolated views, such as camera noise. In the synthetic tests, the noise pattern was exactly the same and was only shifted with the added errors. Since the error was then corrected almost perfectly when using the new reference, the result showed a very high quality. In real applications, the camera noise in the reference


(a) Cones reconstruction
Old: 29.65dB - New: 36.51dB

(b) Teddy reconstruction
Old: 32.96dB - New: 38.16dB

Figure 6.7.: Excerpt from tested images (bottom) with reference (top).

image of a reconstructed view is unknown and, depending on how the view synthesis works, we either get the noise pattern of a single neighboring camera or a combination of the noise patterns from all considered neighbors. In the first case, the pattern can be anything between identical and completely opposite, and therefore its influence on the final quality value can differ greatly. In the second case, the combination of pixels from multiple images means it is likely that the noise is averaged out. Even though the reconstructed views might actually be closer to the reality of the scene because there is less noise overall, the quality is still lowered because the noise still exists in the reference image and the metric expects it to be in the tested image as well.

The research community seemed to agree and a paper presenting the metric was published [2] with favorable reviews. The audience at the conference agreed on the importance of having comparable metrics when combining different quality-changing steps and welcomed our new approach to the problem. But this was the end of the general interest of the research community as according to the discussions after the presentations, the major players in the field are focusing their efforts on creating a universal subjective metric, ideally as a non-reference metric. Since then, many metrics trying to achieve this goal have been published [153, 154], but they always cover a very small portion of image-based media E.g., portraits, nature photography, or HDR images of inside architecture, and do not work very well on images of other types. This development does not make our approach more meaningful, but it serves its purpose and in its niche, it has not been replaced by universal metrics yet.

# 7. Real-Time View Interpolation

Multiview video needs to be processed before viewing, in order to experience the benefits this new format can provide. One of these benefits is the ability to change the viewpoint of a video interactively, for example, to adjust it to a person's position in relation to the screen for a more immersive experience or to introduce interactivity in a pure broadcasting scenario. For any interactive application, real-time performance is mandatory for every step from the sender or broadcaster to the screen at the client. Throughout this work, real-time performance is defined as the fact that the data of one frame can be processed within the time interval, i.e., before the data for the next frame arrives. At the time of writing, many existing *image-based rendering* (IBR) algorithms that claim to be fast or interactive are only capable of reaching single-digit frame rates [155, 156], which is too slow, if we assume a nominal frame rate of 25 frames per second. Faster algorithms often follow a *depth image-based rendering* (DIBR) approach, which requires exact depth maps in addition to the color images from the capturing cameras [63, 157, 158]. Unfortunately, high-quality depth maps are not always available. Especially when live events are streamed and there is no time to compute a depth map, DIBR based approaches are not viable. Even though a lot of effort has gone into the development of devices that can capture scene geometry information together with the color information, they all have certain drawbacks such as low resolution, high amount of noise, or limited range [159, 160]. Improvement of such depth maps of lower quality is possible but still a very complex task [161, 162, 163]. Since we were looking for a view-interpolation algorithm that could be used in highly interactive scenarios such as video conferencing, we restricted our input data to the color images captured by the cameras as well as intrinsic and extrinsic calibration data for every input camera.

## 7.1. Background

This section explains important concepts for the understanding of the rest of this chapter. Understanding the different types of view interpolation approaches helps to put the focus of the algorithm presented here into context.

Since not everyone is familiar with the different ways of how GPUs can be utilized for different tasks, we provide a quick overview of 3D rendering with OpenGL and general-purpose computing.

### 7.1.1. View Interpolation Algorithm Types

In the following sections, the development of an image-based rendering approach for view interpolation is discussed. That class of algorithms only uses color images from a camera or other sources in combination with some calibration or metadata to create novel views. Since the geometric projection between different views and the scene requires some kind of scene geometry information, the algorithm has to recover or estimate it from the input images first. As shown in Section 7.5.1, the depth estimation takes the majority of the time budget allotted for each frame for most interactive applications. For image post-processing or other steps that can improve image quality, not a lot of time remains.

Depth Image Based Rendering (DIBR) algorithms solve this issue by adding depth or disparity maps as a mandatory input. For recorded content, the generation of depth maps can then take an arbitrary amount of time, which leads in general to a higher quality. In addition to the better scene geometry information, there is much time left over to implement steps for even better results.

On the other hand, with live content, they have to deal with the noisy output of depth sensors and their offset from the location of the color images. That is why those algorithms usually prefer recorded content with a lot of time for preparation.

The approaches discussed above share one common property: all the complexity of creating the desired views is situated purely on the client. Client devices with sufficient computation power can quickly become too expensive for widespread deployment. Since creating individual views has a complexity that scales linearly with the number of views to create, it is not well suited for systems catering to multiple clients. To make cloud-based view interpolation possible in a broadcasting setting, Collet *et al.* [164] propose an intermediate representation. From the input images, they create textured 3D meshes for the scene objects and the background. Depending on the number of input views and the resolution of the final mesh, such 3D representations can significantly reduce the amount of data to be transmitted. Additionally, from these meshes, it is far easier for the clients to recreate the desired perspectives as they combine color and depth data. The only drawback is the vast amount of computational resources required to achieve fast computation times even for low-resolution meshes.

### 7.1.2. 3D Rendering with OpenGL

OpenGL is a framework for the rendering of 3D content which is available for most operating systems and GPUs [165]. Most of its operations directly interact with the GPU and the data residing in its memory.

In general, everything rendered in OpenGL consists of a collection of vertices in 3D space which are connected to form primitives. Those primitives can be points, lines, triangles, or quads that form the surfaces visible in the scene later. Since triangles are the most common type of primitive, we use them as a generalization for all primitives in the remainder of this chapter.

3D scenes can be created by adding single triangles to them until a scene is complete. Creating them one by one is quite inefficient as the data for every triangle is transferred to the GPU in a separate operation and requires some synchronization time between the CPU and GPU every single time. Collecting the data for all triangles in a single data structure and transferring that in a single operation is much quicker, especially in scenes with a high number of triangles.

The real power of OpenGL comes from the flexibility added by its shader pipeline, which is shown in Figure 7.1. All vertices in a scene are processed through its stages which can perform calculations on the vertex data and even add or remove primitives before the scene is rendered. How many parameters a vertex can carry through the pipeline depends mostly on the capabilities of the hardware whose shader units are used.

The first processing stage after the initialization of the data is the Vertex Shader. It receives a single vertex and all its assigned parameters as input, can perform any operations on the data, and has to return exactly one vertex, but can assign arbitrary data to it. The most common things done in this stage are the projection of the vertex's position into its

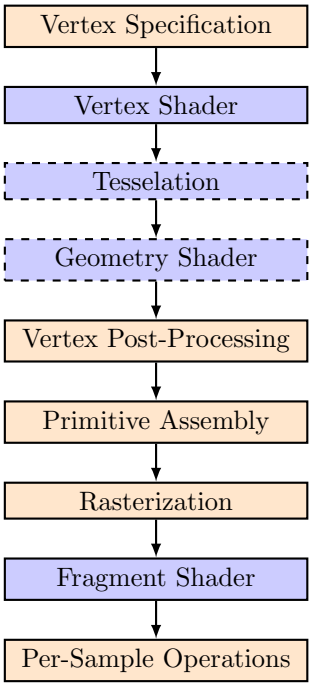



Figure 7.1.: OpenGL rendering pipeline. Orange stages have a fixed functionality, blue ones can be programmed. Only stages with a dashed outline are optional.

final screen location and the preparation of data for the later shader stages.

The Tesselation phase is the newest addition to the pipeline and completely optional. It consists of three substages that subdivide a surface into smaller pieces with their own parameters. A Tesselation control shader sets the amount of tesselation to be done, the primitive generator creates the new intermediate vertices based on the results of the controller, and the evaluation shader computes the attributes of those new vertices based on the state of the original ones.

In the geometry shader, one primitive goes in and an arbitrary number of primitives comes out. The type of primitive can be different but needs to be fixed for each instance of the shader. Duplicating primitives for layered rendering or just separating parameters from each other are its main use case, but in theory, it can be used to replace the tesselation stage to some degree. A specialty of this stage is its possibility to be run multiple times during one pipeline iteration. This feature allows multiple different parallel calculations on the same primitive.

Vertex processing prepares the data for the later stages, which transform the geometric data into screen pixels. The vertex data is converted to a list of primitives, the ones outside the set viewing range are discarded and the remaining ones are transformed according to the current view setup. The following primitive assembly sorts the primitives again for later and is capable of discarding primitives whose surface faces away from the camera, also known as face culling.

Rasterization describes the process of creating fragments from the primitives. A fragment is a portion of the primitive that coarsely correlates with the area that covers a pixel in the final image. The parameters a fragment receives, are interpolated from the vertices defining the original primitive, based on the fragment's location within it.

For each fragment, the fragment shader is executed. Its task is to transform the parameters a fragment inherited from the vertices into color and depth data. How many operations are used to achieve this and what they do exactly can be freely defined by the user.

In the last stage, it is decided which fragment is going to be visible in the final image with occlusion tests based on the depth and the transparency of each fragment. If it is configured correctly, this step can perform antialiasing to reduce staircase patterns at the edges of objects.

This architecture with its programmable stages provides many possibilities for the implementation of quite complex processes. When the vertex parameters are provided at least partially from the frameworks described in the following section, even more possibilities become available.

### 7.1.3. Computation on GPUs

When algorithms for servers or PCs are discussed, most people assume the only component in these machines capable of performing arbitrary computation is the CPU. Applications that can be parallelized or vectorized well, are also suited to be executed on GPUs. There they can make use of the many thousands of cores usually reserved for rendering 3D graphics.

Since the introduction of the OpenGL Shading Language in OpenGL 2.0, researchers have been using GPUs to perform complex computations [166], calling it General-Purpose Computation on Graphics Processing Units (GPGPU). By putting input data into images used as textures, processing them through the shaders of the GPU, and analyzing the resulting display output to get the results back, arbitrary computations could be performed. The resources of the device could sometimes not be used efficiently because the function of the shaders was fixed, could only be changed slightly, and the number of units for every shader type was determined by the hardware. In 2007, with the introduction of the NVIDIA GeForce 8000 and the Radeon HD 2000 families of graphics cards, this changed significantly with so-called unified shaders. Those processing units can be freely configured to fulfill any arbitrary function. With them, the speed of calculations could be increased, because they could be more freely programmed. Additionally, all shaders could be used for data processing since they are not locked into a certain stage of the shader pipeline anymore.

Shortly after the unified shaders came to the market, programming frameworks dedicated to data processing on GPUs were created. The most famous ones being CUDA [167] and OpenCL [168]. CUDA was only made for NVIDIA GPUs while OpenCL aimed to offer a unified interface for devices from all manufacturers. In both frameworks, programs are written in a dialect of C with special functions for vector and matrix operations. They differ mostly in their available functions for data transfer and memory management. Both frameworks got rid of the necessity to input and output data via textures but maintain the option to do so, for example to use the calculation result for rendering. They introduce the possibility to arrange the data in N-dimensional arrays instead, which can be written and read independently from rendering operations. When used properly, they achieve comparable performance on the same hardware [169]. Even though CUDA became more popular in recent years, OpenCL was the framework of choice in this chapter due to its versatility with respect to usable hardware.

## 7.2. Concept

An increase in speed of a view interpolation algorithm usually comes with a lower quality or a restriction of possible inputs. Grau *et al.* [170] and Hilton *et al.* [171] achieve good quality and speed by limiting the input material to footage from stadium cameras capturing football matches and similar sports events. In those cases, the non-static parts of the image (players and balls) are usually filmed in front of the green grass which simplifies the separation of foreground and background. Since we were more focused on interactive scenarios, such as video conferences which do not guarantee any known structure in the background, such an approach was not an option.

The algorithm published by Zhang *et al.* [13] did not make any assumptions about the contents of the scene and offered a configurable trade-off between image detail and speed. This trade-off was possible because the algorithm was based on a plane-sweeping approach and uses an adaptive 2.5D mesh to reconstruct the novel views. For reasonable image qualities, their implementation only achieved between 5 and 10 frames per second on very low-resolution input material. Further, their implementation only used a single core of the CPU for the calculations and an old version of OpenGL[1] for rendering. Thus, there exists a lot of potential for speed increase simply by making use of the full capabilities of the available hardware at the time. Especially due to the availability of multi-core CPUs in consumer PCs and the emergence of unified shaders [172], even common consumer devices have a lot of computational power, most of which was not used by the original algorithm.

## 7.3. Analysis

The first step towards an optimized version of an existing algorithm is always the thorough analysis of its structure. Algorithm 7.1 shows the major steps the program performs during runtime. The first two steps can have a big influence on the performance of all following steps, both in quality and speed, as they determine how all other steps operate, but optimizing them does not make a big difference as they are only executed once at the beginning of the program. Their way of influencing the following steps, can not be changed by optimization. Resetting the framebuffer between frame calculations is important, but does not offer a lot of optimization potential since it is a simple step in every graphics framework which is already highly optimized. It could be argued that, in certain cases, this step would be unnecessary as the framebuffer is overwritten completely in every step and therefore, would not need to be reset, but its runtime compared to the rest of the algorithm is minuscule and it is considered good practice to invalidate the previous buffer contents before starting to work on the next frame [173], so it is left in.

### Step 1: Create initial mesh

The creation of the initial mesh is the first step that can be slightly optimized. Since all vertices of the basic first mesh are always in the same position, which only depends on the configuration parameters, it could be created once and then always be reused for all frames. The vertices are uniformly distributed over the frame for the virtual view, with a configurable horizontal and vertical distance $D$ between them. To save time for computation and memory allocation later on, in this step the vertices for all possible subdivision levels are allocated and associated with their positions. This precomputation of the vertices is possible because we start with a uniform grid and subdivide triangles by halving the

[1] `https://www.opengl.org`

```
Read configuration file
Calculate parameters for rendering
while output window exists do
    Reset framebuffer
    Create initial mesh (Step 1)
    Load input images (Step 2)
    foreach vertex in the mesh do
        Find closest views (Step 3)
    end
    foreach vertex in the mesh do
        Estimate depth (Step 4)
        Calculate rendering weights (Step 5)
    end
    while Mesh is too coarse do
        foreach Triangle do
            if vertices have different depth then
                Subdivide triangle (Step 6)
                foreach new vertex do
                    Estimate depth (Step 4)
                    Calculate rendering weights (Step 5)
                end
            else
                Keep triangle unchanged
            end
        end
    end
    Complete mesh (Step 7)
    foreach Triangle do
        Render to frame buffer (Step 8)
    end
end
```

**Algorithm 7.1:** Major steps of Zhang's algorithm [13].

distance between the vertices in the corners of the triangle resulting in up to four smaller triangles occupying the space of the old triangle. The details of this process are explained in Step 6. Due to the simplicity of the calculations in this step, the gain is not very big compared to a good parallel implementation on multiple CPU cores or directly on the GPU. Furthermore, keeping the initialization step static for the whole scope of a program execution also limits the possibilities for modifications to the overall flow of the program with respect to features that require an initialization based on the result of previous frames. Due to of these limitations and the small expected gain, we opted to forgo the reuse of a single result and maintain the recalculation per frame, but use proper parallelization techniques to speed up the calculations. The loading step of the input images can not be optimized a lot either.

### Step 2: Load input images

In the original algorithm, the pixel data is already copied efficiently from an input buffer where the images are decompressed to the memory area where the following calculations are happening. Apart from employing some kind of lossless compression to the image data in order to reduce the amount of data that has to be copied, there is no optimization potential.

The next steps have the most potential to make the software faster. All of these steps have to be performed either per vertex or per triangle in the mesh. Depending on the configuration parameters and the input resolution of the images, each of the steps has to be performed thousands to millions of times per frame. Since the original implementation only performs these calculations in serial using a single CPU core, the performance gain here could be enormous.

### Step 3: Find closest views

First, the set of nearest cameras are determined for every vertex. This is done by calculating the distances from every input camera's center of projection to the light ray from the center of the virtual camera through the considered vertex. As shown in Figure 7.2, this approximation is a good estimate for the closeness of the cameras to the virtual camera when the cameras are roughly positioned on a plane and look in a similar direction. The results of all cameras are sorted in ascending order and the *num_interp* closest cameras are saved for future use. *num_interp* is a configurable parameter in the configuration file which ranges between 1 and 4. Since the calculations for one vertex, required to find the closest cameras, are completely independent of the results of other vertices, all vertices can be processed in parallel and at the same time, at least in theory. In practice, this might not be possible because the number of vertices can become quite large, especially for high-resolution input images and a fine grid. Based on the parameters, the number of vertices can become higher than the number of available computational units in a CPU or GPU. Therefore, efficient queuing is required to get the most out of the parallel execution. Nevertheless, a significant speedup can the expected as there is no need for barriers in the execution due to the independence of the vertices in this step.

### Step 4: Estimate depth

Once the views to consider for every vertex are determined, the depth estimation step begins. Following an adapted version of the plane-sweep algorithm [174], a set of depth planes is placed in the scene space. The position of these planes is configured by three configuration

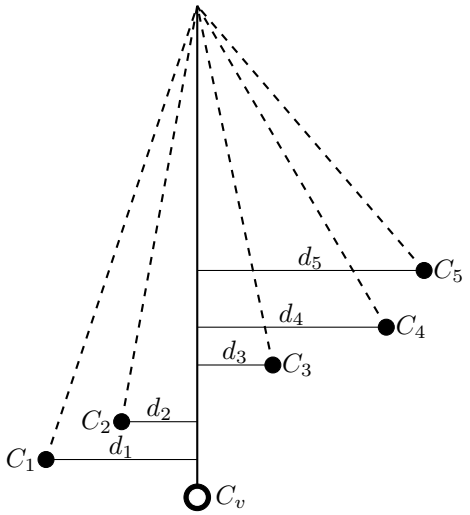



Figure 7.2.: Nearest camera calculation. Distance of cameras is calculated as the distance to a ray through the center of the virtual camera.

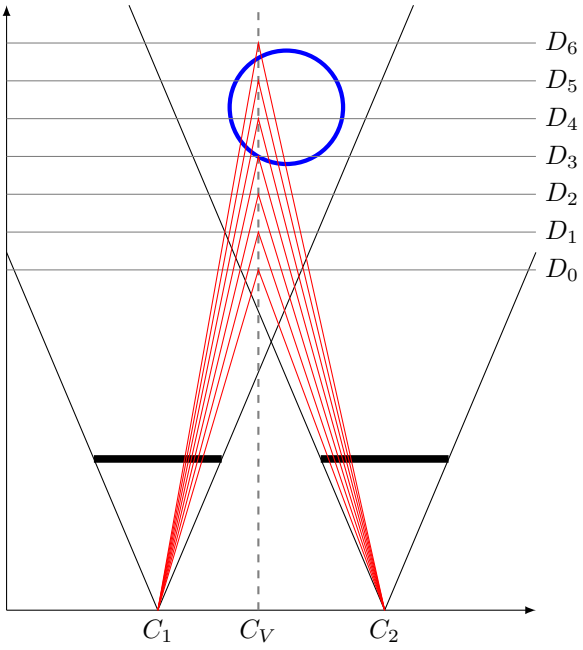



Figure 7.3.: Depth estimation via plane sweeping. A point is tested with depths from all depth planes. The depth with the highest similarity in neighboring images is chosen as the correct value.

parameters; *min_depth*, *max_depth* and *num_depth*. The algorithm uniformly distributes *num_depth* planes between *min_depth* and *max_depth* inclusive, parallel to the image plane of the virtual view. Figure 7.3 illustrates the process of estimating the depth values for the virtual view. First, a vertex is projected from the virtual camera into the scene. This gives a vector of indeterminate length. One after the other, every length corresponding to a depth plane is considered to be the correct one. The scene point at the matching depth is then projected onto the image plane of every image in the closest set determined in the previous step and an image patch surrounding the intersection is extracted. Next, the extracted patches are compared and a similarity value is computed. The base implementation offers two different comparison functions: a sum-of-squared distances (SSD) function (Equation 7.1) and a mean-removed correlation-based (MRC) function (Equation 7.2).

$$D_{SSD} = \sum_{i \in N} \sum_{P \in W} \frac{(I_{i,P} - \overline{I}_P)^2}{C_P} \tag{7.1}$$

$$D_{MRC} = \sum_{i \in N} \sum_{j \in N, j > i} \frac{\sum_{P \in W} (I_{i,P} - \overline{I}_i) \cdot (I_{j,P} - \overline{I}_j)}{\sqrt{\left[\sum_{P \in W} (I_{i,P} - \overline{I}_i)^2\right] \cdot \left[\sum_{P \in W} (I_{j,P} - \overline{I}_j)^2\right]}} \tag{7.2}$$

$N$ is the set of nearest or closest images, $W$ is the window or patch of extracted pixels and $P$ is the position of a certain pixel within the window. $I_{i,P}$ denotes the pixel value of image $i$ at position $P$, whereas $\overline{I}_P$ is the average pixel value at position $P$ of all images in $N$. $\overline{I}_i$ is the average value of all pixels in the window $W$ extracted from image $i$.

Since the optimal choice depends on the type and quality of the input material, which of these functions is used is determined by another configuration parameter. For noise-free synthetic content, the SSD produces good results, but for noisy camera-captured material, the MRC fairs much better. The biggest drawback of the MRC is its higher complexity which is evident when comparing both formulas. This step is then repeated for every depth plane in the scene. Once the similarity value for all depth planes is known, they are sorted and the depth plane with the highest similarity is chosen as the correct one and saved in the current vertex.

## Step 5: Calculate rendering weights

The rendering weights in each vertex determine the contributions of every image in the set of closest images to the final rendering result for that vertex. They are calculated using two factors. The first one is the inverse camera distance from Step 2. The second one is a factor which is 1 when the result of the projection from the scene point lies within the respective image. Near the edges of the image, when parts of the image patch, which has to be extracted, fall outside the image, the factor decreases, representing the number of valid pixels. When the image patch lies completely outside the image, the factor is zero. This means the weight of an image whose information becomes less reliable towards its edges is lower. The weight for each image is the product of these two factors. In a final step, the weights are normalized so they add up to 1. This is important since the images are combined later on using alpha blending, which would lead to deviations in brightness if the sum of the weights is higher or lower than one.

Again, the calculations in this step do not require any information from the current step, but only need results from earlier steps. This makes it easy to parallelize this step, but the gain is not significant, due to the simplicity of the performed computations.

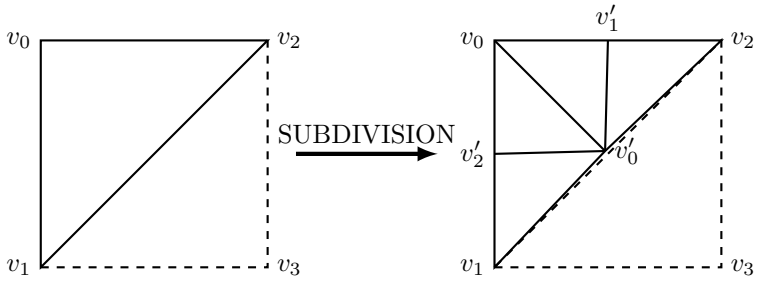



Figure 7.4.: Subdivision process of a single triangle with coordinate deviation in the end result.

## Step 6: Subdivide triangles

This step is triggered when a discrepancy between the estimated depth values of the vertices in a triangle is detected. Such a discrepancy means that, under the assumptions that the estimation returned the correct depth value, at least one side of the triangle crosses a border between two objects with different depths. It is first checked how often the triangle has already been subdivided. In case the configured limit has not been reached, the subdivision is performed. To get a better approximation of the edge, that the triangle crosses, a new vertex is introduced in the middle of every edge of the triangle, as shown in Figure 7.4. The original triangle is replaced by four smaller triangles that occupy the same area. Next, it is checked whether the new vertices have already been processed or not. This test is necessary when a certain vertex already belongs to the subdivided version of a neighboring triangle that has been processed earlier. If a vertex is completely new, it is processed as follows: i) their nearest cameras are determined following the process described in step 3, ii) the depth of the vertex is estimated like in step 4, and iii) the rendering weights are determined as described in step 5.

Once all new vertices and triangles are processed, it is determined whether any triangles in the mesh have been subdivided in the current step. If that is the case, the step is repeated until the mesh does not change anymore, which either means no triangle has a depth difference in its vertices anymore, or the subdivision limit has been reached.

The optimization of this step is critical for the success of the whole process. The distribution over multiple CPU threads should not pose a significant challenge, as long as the access to the newly created vertices is controlled properly to prevent race conditions and redundant calculations of data. The efficient use of GPUs for the calculations of this step is more complex. The step involves many if/else instructions, which lead to code branching and immense performance losses on GPUs due to the simpler architecture of their cores compared to CPUs and their way to handle code branching [175]. A restructuring of this step is required to make each substep more uniform and to reduce the number of decisions at runtime as much as possible.

## Step 7: Complete mesh

The last step of the mesh construction is the mesh completion. In this particular case, it means finding triangles of all subdivision levels which have neighbors of a higher subdivision level. One example of such a triangle would be the right triangle in Figure 7.4. While the edge between $v_1$ and $v_2$ has been split into two edges with $v_0'$ in the center for the left triangle, the right triangle still relies on the direct edge between $v_1$ and $v_2$. In a perfect

world, this would not matter since $v_0'$ is supposed to lie exactly on that edge and the behavior, therefore, does not change. In reality, it is a bit different. Due to rounding errors and the limited number of decimal digits in the definition of the position of $v_0'$, it might not lie directly on the edge and a gap between the neighboring triangles might appear, which can lead to cracks or other rendering artifacts, depending on which side of the edge the new vertex falls. To remedy this problem, the current step was introduced. It redefines the space occupied by the problematic triangles using smaller triangles making use of the newly created vertices. When all three sides have new, unused vertices, the result is similar to that of the subdivision step. It is still kept separated as it is much simpler with respect to the required complexity. The main reason is that the process is only executed once and all vertices used in this step have already been processed before and no further depth estimation or similar calculation is necessary.

Considering the optimization potential, we encounter the same problems we have already seen in the previous step, namely a lot of code branching due to the high number of tests and decisions to find and process the problematic triangles. In comparison with the previous step, those problems are not as critical since the step is only performed once for every frame.

### Step 8: Render to frame buffer

In this step, the data which was calculated in the earlier steps is used to render the virtual view from the position of the virtual camera. The original implementation uses the software rendering of OpenGL to do this. For every image that has been used in the previous steps, every triangle which has the current image in its list of nearest cameras is rendered separately. All vertices of the triangle are projected onto the current image and the resulting coordinates are then used for standard texture mapping. To combine all used textures, the triangles are rendered with an alpha value equal to the rendering weight. Once all triangles have been rendered for every input image, the virtual view is complete.

Since every triangle is sent to the GPU and rendered individually, the whole process is not very fast and the repeated small data transfers between the CPU and the GPU add a significant overhead to the whole process. Making use of batch transfers to get the data to the GPU would be much faster but requires a reorganization of the rendering step to have all required data available at the same time in the correct order. With the use of custom shader programs in the rendering pipeline, it would even be possible to move some calculations, especially the ones responsible for the final projections, to the GPU which is very efficient when it comes to vector and matrix calculations.

## 7.4. Improvements

During the thorough analysis of the base algorithm in the previous section, potential for performance improvements in multiple parts has been identified. A subset of the implemented changes was described in [1]. The major improvements are explained in detail in this section and their influence with respect to the overall rendering time is analyzed in the next section.

### 7.4.1. Shader-Based Rendering

While the software rendering technique in the original implementation is quite flexible when it comes to which data is used for which triangle or vertex, it is also quite slow since every triangle that is rendered has to be processed individually. Modern rendering techniques rely on batched data transfers and programmable shaders for fast rendering with individual

settings for the triangles [165]. Since the use of shaders for computations has only become viable with the introduction of the unified shader architecture in GPU at the end of 2006, and therefore, two years after the publication of [13], with the GeForce 8 series GPUs the base implementation of our algorithm does not consider the requirements for their use in its structure. For this reason, the reorganization of the rendering data and the transfer of certain parts of per-triangle calculations from the CPU to appropriate shaders of the OpenGL rendering pipeline are the main topics discussed in this section.

#### 7.4.1.1. Data Pre-Computation

As mentioned before, one major difference between software rendering and shader-based rendering is when the required data has to be ready. In software rendering, missing pieces of data can be calculated just before the single triangle is sent to the GPU. To use shaders most efficiently, all data for a batch of triangles needs to be ready before rendering starts.

While this seems not to be an issue when looking at the description of the original algorithm in Section 7.3, rendering artifacts appear in certain parts of the resulting image when just the data that has been computed before the rendering step is being used. The main reason being the projected coordinates from the depth estimation step. If the vertices of a triangle use different sets of nearest images, there is a problem during rendering. In a triangle with the vertices A and C with the closest cameras $\{1, 3, 4, 5\}$ and B which is closest to $\{2, 3, 5, 6\}$, the projection coordinates are only calculated for the images in the set of closest images. Since OpenGL interpolates the projection coordinates and the color values for every pixel inside a triangle from the vertices in the corners, in the problematic triangle, there are either missing coordinates for some textures or the system is trying to interpolate between two different textures, depending on how it treats a texture mismatch in the pipeline. To fix the occurring errors, the coordinates have to be known for the combined set of images $D = A \cup B \cup C = \{1, 2, 3, 4, 5, 6\}$.

Ideally, we would determine the combined set of all images occurring in the triangles which use a given vertex and calculate the projections for every image in the set. As there is no internal mapping from vertices to triangles and the subdivision step can change the number of triangles per vertex dynamically, this is not an easy task. In addition to those factors, there is technically no upper limit of how many triangles a vertex can belong to, and with high-quality settings, we are dealing with a huge number of triangles and vertices. Even with an efficient algorithm to determine the accurate set for every vertex, it is still faster to calculate the projection coordinates for all input images once the depth has been estimated and transfer all of the coordinates to the GPU. This procedure aids the performance with GPU computation as there are fewer checks and therefore, fewer code branches required. Instead, the projections must be calculated a fixed number of times, which is easily parallelizable. For a scenario with 16 input images, in which the projection coordinates are stored using a 32 bit float value, the memory required for the projection coordinates of every one million vertices is

$$MEM_P = 1000000 \cdot 16 \cdot 2 \cdot 4bytes = 128000000bytes \approx 122MB \tag{7.3}$$

While this might seem like a lot, modern GPUs usually feature multiple gigabytes of graphics memory, so this is not a limiting factor. Additionally, one million vertices is a value that is only encountered when very high-quality settings are applied to high-resolution input images. Even if we reserve the highest rendering modes for GPUs near the upper end of the performance spectrum, we do not consider this a problem since these devices also have enough computing power to perform the calculations required for one frame fast enough for a good overall experience.

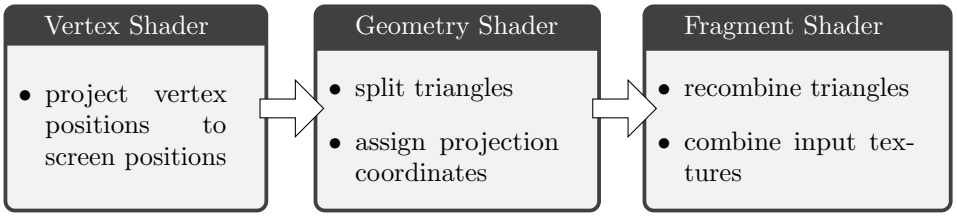



Figure 7.5.: Implemented shader structure with assigned tasks.

#### 7.4.1.2. Shader Structure

In Section 7.1.2 it is shown that a model OpenGL rendering pipeline contains many different shader stages and only some of which are freely programmable. These programmable stages can perform arbitrary calculations but are limited with respect to what they receive as input and what they can output. Figure 7.5 shows the simple structure we have chosen to adopt for this implementation. The vertex shader only projects the vertex positions to screen coordinates and passes through all other input parameters. This minimal functionality is necessary because there is no default implementation of vertex shaders and they are mandatory for every rendering pipeline.

The geometry shader with the capabilities to create new vertices and triangles is the first step that is used to make the rendering of the triangles, which may have up to three sets of textures that should be used for rendering. As explained in Section 7.4.1.1, the projection coordinates for all input textures are available and only need to be properly used in accordance with the nearest image sets and the vertex coordinates. Here, the properties of the geometry shader come into play. It takes a triangle as input and has access to all vertices the triangle consists of as well as their respective parameters. Internally, it is invoked three times for every triangle. Each invocation creates copies of the original vertices with the same coordinates and the nearest image set of only one of the vertices. It then also reads the projection coordinates of the respective nearest image set and forwards them to the next shader stage. After this step, there are three times as many triangles as before, but it is guaranteed that all the vertices in a triangle access the same textures and all have the correct projection coordinates.

The fragment shader is called for every fragment which contributes to the value of a pixel to the final image, receives interpolated coordinates from the corner vertices, and has to return a color value for the fragment in the final image. For this implementation, it extracts the color value from every image in the given nearest image set, multiplies them with their respective rendering weights, and adds them up. As a final step, the alpha value of the resulting color is multiplied by 1/3. It recombines the three triangles that share the same position which were created in the geometry shader. With these steps, the problem with the different texture sets of the vertices in a triangle is solved and the shaders produce the intended result.

##### Shader Parameter Transfer

Now that we have determined the parameters we need for a correct shader-based rendering, there is another hard limitation we have to work around in order to make the approach as flexible as possible. To render a vertex, the following data has to be provided to the shader pipeline: one position, one vertex index, $N$ image IDs, $N$ weights and $I$ projection coordinates. $N$ is the number of considered nearest neighbors and $I$ the number of input

images. For footage from a camera array with 16 cameras and 4 images used for interpolation, this equals

$$Params = 1 + 1 + 2 \cdot N + I = 1 + 1 + 2 \cdot 4 + 16 = 26 \tag{7.4}$$

parameters for every vertex. Parameters with multiple dimensions such as the position or the projection coordinates, count as a single one because they are defined as vectors with the required number of dimensions.

Since OpenGL restricts the number of parameters per vertex, the amount of images that can be used with this approach is limited. The exact limit is defined by the GPU and is given by the OpenGL context property *GL_MAX_VERTEX_ATTRIBS*. Even the most recent consumer GPUs only allow a maximum of 16 parameters. Only professional grade GPUs and several AMD GPUs offer up to 29 parameters [176]. This upper bound severely limits the number of images we can use. As this approach is supposed to work with our 64 camera array, a solution is needed.

The most flexible solution we found is the use of a *buffer texture*. A buffer texture can be used to access big chunks of memory from the shader pipeline. In the shaders, it looks like a large one-dimensional array and can be accessed as such. The location of the data for a specific vertex can be derived from the vertex ID, which enumerates all vertices to be rendered. Every entry in the texture can be one of the basic data types in OpenGL as well as vectors with up to four entries of the basic types. Even though the vectors are supposed to represent the color channels R,G,B and A, they can be freely accessed and interpreted. Therefore, they can be used for all kinds of data as long as they can be represented by the basic data types or their vectors.

While the size of these textures buffers is only limited by the size of the buffer they get their data from, and therefore the size of memory on the GPU, the highest index that can be accessed by the shaders is defined in *GL_MAX_TEXTURE_BUFFER_SIZE*. Luckily, all modern dedicated GPUs support values of at least 130 million, many AMD GPUs even reach into the billions and support the maximum value of a 32-bit integer, so this limit is not a problem [176]. In case we come close to the limit, using multiple buffer textures and splitting the data between them is possible. This measure reduces the parameters for the example values above to

$$Params = 1 + 1 + 2 \cdot N = 10 \tag{7.5}$$

That amount is unproblematic for most GPUs and is only dependent on the number of images considered for interpolation.

Another source of rendering errors was identified as the number of textures used for rendering. Since every vertex needs to be able to access every input image for the final rendering step, reducing the number of images on the GPU is not an option. The main reason for the problem is the limited number of texture sampling units that are allowed to be used at once. To remain flexible with respect to the number of input images for a scene, an array texture is used. Its structure is similar to that of mipmapped textures but instead of increasingly smaller representations in the third dimension, they contain different images of the same size. With such an array texture, all input images can be grouped together and can then be handled by a single texture sampling unit. There are again hardware limits for the maximum number of different images in a single array texture but most GPUs support 2048 images, which is far more than what is needed for the footage we plan to use with this view interpolation software.

### Dynamic Memory Allocation

The shader software for the rendering of the interpolated views contains several parameters that can change depending on the specific configuration used for a certain software execution. Most of these parameters also influence the number of input and output parameters of the different shader stages as well as how much memory is required for intermediate results or how often certain loops have to be executed. While unknown loop iterations only influence how well the code can be optimized by the compiler, varying amounts of required memory are more problematic as code on GPUs can not allocate memory dynamically, when more is required. Always allocating the highest amount of memory that is ever going to be needed, would limit the choice of GPUs to run this algorithm on, since not all can support the highest settings. With the maximum allocation approach, GPUs that can not handle the highest settings can not run any settings at all. Since the range of quality and speed settings in this algorithm can span a vast parameter range and it is supposed to adapt to different performance levels of the hardware it is running on, this is not acceptable.

Our solution for the problem is mainly enabled by the fact that shader programs can be compiled during runtime. This feature exists because usually it is not known on which hardware a piece of software is going to be running, and it might even change between runs. Since shaders and other GPU code are highly hardware-specific, the only solution to make them compatible with a wide range of hardware is to compile and ship the GPU-specific code for every available platform or to compile the code during runtime for the one hardware configuration, which is currently present. While the first approach is faster when the software is executed, since it is enough to merely load everything from storage, it does not support hardware if it has not been considered during the compilation step, especially if it has been released after the software. The second approach can be used with any compatible hardware independent of its release date or availability to the developer. Technically, the compiled code can be stored and reused without recompilation, but then it requires checks in every software execution to know whether any parameters that influence the compilation process have changed and a re-compilation is required. As described above, we plan to change parameters rather often using the configuration of the software. Since the kernels have to be recompiled for each parameter change and therefore, there is not a lot to gain from saving the shader software, we decided to compile the shaders in every run. The compilation process only takes a couple of seconds for very complex programs, which is acceptable at the beginning of the software execution.

The parameters that influence the amount of required memory are represented by placeholders in the GPU code. Before compilation, the placeholders are replaced with the current value of a parameter or a calculated value. Since the size of arrays, the number of loop executions, and other parameter-dependent values are not variables but literals, they are hard-coded into the program. With fewer variables in the code and the size of all arrays fixed, the automatic optimization in the compiler can work more efficiently and dynamic memory allocation is not required anymore.

This approach is applicable to every piece of code that does not have access to dynamic memory allocation but can be compiled during runtime. In the context of this thesis, this means any piece of code executed by a GPU. This procedure is important because it allows the transfer of even more complex computations from the CPU to a GPU with general computation capabilities.

### 7.4.2. Parallel Execution Using GPGPU

The analysis of the base algorithm in Section 7.3 shows that there is a lot of potential for speed increase using parallel execution. Due to the high number of elements that have to be processed and the independence of the different steps in the algorithm, the use of GPUs instead of multiple CPU cores was considered. Modern GPUs offer the possibility to use their processing units, which usually handle the conversion of 3D data to pixels on a screen, for arbitrary computations that do not have to result in something on screen necessarily. With GPUs being clocked far slower than CPUs at only slightly over 1.5GHz, they are about two times slower than most modern desktop CPUs. Their main benefit is the higher number of cores. While modern consumer or workstation CPUs can feature 32 or more logical cores, GPUs have over 10000 cores on the upper end of the performance spectrum.

To perform computations on a GPU, there were two major frameworks available. Arguably, the most common one was and is CUDA by Nvidia[2], but using it limits the algorithm to GPUs of this manufacturer. Due to the lab's affiliation with the Intel Visual Computing Institute (IVCI) at the time, it was important to keep the algorithm compatible with their devices. Therefore, we chose OpenCL[3] to execute the code on the GPU. That framework is supported by nearly all GPU and CPU manufacturers with varying levels of support and can achieve the same performance as CUDA when given a fair comparison [169].

#### 7.4.2.1. Preparation

In order to make the transition from CPU to GPU more manageable, the original algorithm was split into four main parts. Its functionality with respect to the steps is described in Section 7.3.

**Scene Loader** Reads and processes the configuration file for a scene. Furthermore, it loads and decodes the input images from the source given in the configuration. The raw image data and all important parameters are placed in a location where they can be accessed by the next step.

**Depth estimator** Uses the data provided by the loader to estimate the depth for the virtual view. Consisting of the creation of the initial mesh, coarse depth estimation, mesh refinement, and mesh finishing. The resulting depth information is stored in the form of a depth map for the next step to use.

**View processor** Combines the input images with the recovered depth information into a complete virtual view. To maintain independence from the previous part, the mesh is recreated from the depth data. Following that, the vertices are projected to the respective input images and the images are blended together to create the final virtual view, which is stored in a texture for further use.

**Renderer** This part is responsible for outputting the final image. To keep the full functionality of the original version, it also handles the window in which the image is displayed and the interactive inputs to control the position of the virtual view.

The separation of functionality allows the replacement of parts without disturbing the functionality of the rest of the program.

To allow for a flexible exchange of the program parts, they are implemented as dynamic libraries with a simple interface for initialization, processing, and cleanup. The main program

[2] `https://developer.nvidia.com/cuda-zone`
[3] `https://www.khronos.org/opencl`

searches for the libraries with the original functionality by default, but with command-line parameters, the use of different libraries can be forced.

The data between the different parts is exchanged using a memory structure, whose address is shared between all program parts. It contains the configuration parameters, camera calibration data for the input cameras, the current camera position and direction as well as the OpenCL context used by the program and pointers to the intermediate frames containing the main shared data between the steps. Every frame is accompanied by a position flag which indicates whether the frame resides in RAM or GPU memory. With those flags, the individual libraries know where the previous step placed the frame and can copy it to their preferred location, if necessary. Therefore, the choice of libraries can influence the overall performance of the program negatively if the frame data is copied from RAM to GPU memory multiple times during the processing of a frame, but it guarantees the interoperability between different libraries, independently from the device they mainly rely on for their computations.

#### 7.4.2.2. Porting Code to OpenCL

OpenCL kernels use OpenCL C as their programming language. It is a dialect of C99 with some features, such as dynamic memory allocation missing, but extended by data types for vectors and matrices as well as keywords for memory placement. Given that the original implementation of the algorithm to be ported is a pure C program, the conversion is quite straightforward. The original implementation does not rely on any external libraries for its computation, so converting it to valid OpenCL C requires only small changes.

The main changes that are required for a first runnable version involve the prevention techniques for dynamic memory allocation, as described in Section 7.4.1.2. Applying them to every piece of code that is supposed to be run on the GPU eliminates any arrays of variable size and replaces it with a fixed one. Furthermore, pieces of code whose execution purely depends on the value of a configuration parameter can be optimized away when they are not needed. Since the if-clauses controlling the execution of these code areas only contain checks on literals after the replacement of the placeholders, the compiler can automatically remove them. The reduction of possible code branches is the first step towards an optimized GPU version of the code.

Another mandatory change to the code is the definition of the memory locations for inputs and outputs of the kernels. OpenCL differentiates five locations for data on the GPU:

**Host memory** Represents the memory or RAM of the machine the OpenCL devices reside in. When OpenCL is used in combination with a dedicated GPU, this portion of memory is the slowest by a fairly large margin. The first reason for this is the fact that the memory itself is slower than that of GPUs, the second reason is that every data transfer requires some synchronization between the host and the GPU.

**Global memory** Describes the dedicated graphics memory on the GPU. Its size is given by the memory size in the GPU's datasheet. It is the slowest of all accessible memory areas on the GPU and its speed can vary greatly depending on the type of memory used on the GPU. Compared to the RAM of the host system, its bandwidth is still several times faster. Modern GPUs with HBM2 can reach up to 1TB/s of data throughput while DDR4 RAM in a quad-channel configuration can only deliver about 80GB/s.

**Constant memory** Is part of the global memory and therefore, shares its speed and size properties. The main difference is that, from a kernel perspective, it can only be read and its content is constant.

**Local memory** Is a section of internal on-chip memory that can only be used by a set of work units, a so-called work-group. The size of this memory is quite small, but it is much faster due to its proximity to the work units on the chip. The minimum size of this memory is at least 16KB, as given in the OpenCL standard, but the exact size is GPU-specific.

**Private memory** Is memory that can only be accessed by a single work unit. There are no guarantees for its size by the OpenCL standard and its whole implementation is up to the manufacturers.

Looking at these choices, it would seem that local memory is the best choice for the position of parameters and intermediate results due to its speed. Two important factors that make its explicit use more complicated, are the persistence of local memory between kernel calls which cannot be guaranteed. Even with predictable persistence, calling the kernel with a different number of work items can move items to a different workgroup which means they cannot access their previous intermediate results anymore. Although global memory is considerably slower, it is the only practical choice for our algorithm. With its comparably large size, global accessibility, and defined persistence, it offers the biggest flexibility of all memory spaces. Since the exact memory area required for the calculations of a certain vertex or triangle can vary greatly, as shown in the algorithm analysis above, global memory becomes the only choice because calculating the optimal data to fit into local or private memory would be too complex for real-time performance. The compiler or the OpenCL runtime environment are still able to use local and private memory as a cache when data is accessed often.

It is important to ensure that the alignment of all the variables is identical, particularly for debugging, but also for all data which is transferred between the RAM in the GPU in the form of structs[4]. The requirements for the placement of variables are usually stricter on GPUs than in RAM. For example, while in standard C/C++ variables only need to be aligned at byte boundaries, in OpenCL C the start of the variables must start at an address divisible by powers of two. The exact value varies between data types. If the alignment is not properly checked, the structs in the RAM have a different size to what the code assumes on the GPU. Depending on the data in the struct, it is also possible that these data fields in the struct have the same alignment, but there are added bytes at the end of the struct itself to align the struct with the next. Both cases lead to problems since the OpenCL code interprets the wrong memory areas as the content of its parameters.

To check whether the alignment is correct, a duplicate of the struct is defined in the host code, in which all primitive data types like *int* or *float* are replaced by their OpenCL equivalent *cl_int* or *cl_float*. Those special types adhere to the alignment rules required on the GPU. Even though it is deactivated by default, most common compilers have a warning that activates when padding bytes are added to structs to align their contents or the struct itself. Those warnings, in combination with the CL data types, help to order the contents in a struct more efficiently and to make sure that the structs can be transferred from RAM to the GPU properly.

[4] Combination of multiple simple data types into a single structure.

The last important conversion step is to handle parts of the algorithm that have to be run a variable number of times for every frame using the OpenCL kernels whose number of iterations have to be defined before they are executed. For the subdivision and mesh completion steps, this is especially important. The iterations of those steps are heavily dependent on the contents of the input images and are, therefore, hard to predict. Always setting the largest possible number of iterations and then only working on the valid items is a possible solution, but due to the large number of work items that can be set in the kernels and the rather low number of required work items, especially in the later subdivision steps, this approach would be a very slow and inefficient solution.

The optimal solution would be to run the first iteration of these steps and, if required, have a kernel start the kernel for the next step or stop the execution. Unfortunately, neither CUDA nor OpenCL offer this functionality. In both frameworks, every kernel execution has to be queued from the host code. Having a kernel that can adjust the required number of iterations or steps of other kernels is not possible.

In order for the host to determine the next step, it needs to know what the outcome of the last step is. Ideally, this can be done without a thorough analysis of the data created on the GPU. Our solution to this problem is a simple counter containing the number of newly created triangles in the previous step. This counter is updated by the work items using atomic functions at the end of their kernels. Only the value of this counter is downloaded into RAM after the kernel execution is finished and, based on that value, either another subdivision step is triggered or the finalization of the mesh is started.

Due to the fact that the host program must wait for a kernel to finish and must also download a piece of memory from the GPU before a new kernel can be started, a significant amount of idle time is added to the overall rendering process. Combined with the heavy code branching in the respective kernels, this mandatory synchronization makes the subdivision and mesh finishing into the least efficient parts of the algorithm. On certain platforms, it is even faster to use an initial mesh that is finer than the highest subdivision level and deactivate subdivision completely, instead of using a coarser start mesh with two or three subdivision steps.

While we did not find an optimal solution for this problem, the recreation of these two steps on the GPU gave a lot of insight into the differences between the hardware architectures of CPUs and GPUs. Long tasks with many memory accesses and repeated calculations can be handled very efficiently on the GPU when the number of iterations in a loop is known before the execution starts and the processed items can be processed independently. On the other hand, even fairly simple instructions can become very inefficient on a GPU platform when many decisions leading to code branching have to be made per item. The cases in which the exact number of kernel executions is not known also add to the inefficiency because waiting for a kernel to finish, analyzing data from the GPU, and then queuing the next kernel, is far slower than using the internal dependency mechanisms for kernels in the queue and let the GPU decide when a queued kernel can be executed.

This observation concludes the basic transfer of the base algorithm from the CPU to the GPU. We have shown the major steps required to move as much of the functionality of the algorithm from the CPU to a GPU to make use of its highly parallel execution capabilities. The general methods for removing the need for dynamic memory allocation can be applied to many different algorithms which want to make use of GPU parallelism. Even if not all parts could be ported to the GPU with the highest efficiency, the resulting increase in speed is still significant, as shown in Section 7.5. Of course, our development and improvement of the algorithm did not stop there. In the following section, modifications for more quality and speed are introduced.

### 7.4.3. Further Performance Improvements

While porting the base algorithm to make use of GPUs took a significant amount of time, we did not stop there. The original algorithm did not use any special techniques with respect to image sampling or filtering. GPUs offer filtering options for image sampling operations, implemented in hardware, which have nearly no influence on the speed of the operation. Since the coordinate projection onto the input images often results in non-integer coordinates, which can benefit from filtered image sampling, we estimated that the benefits of attempting this would potentially be worthwhile.

#### 7.4.3.1. Fast Linear Texture Filtering

Bi-linear interpolation for image sampling is not new. It has been around since the 1980s and in most situations where visual quality is required, it has already been replaced by anisotropic or even more sophisticated interpolation schemes. However, for our purposes the quality improvement we achieve with bilinear interpolation over the nearest-neighbor interpolation is sufficient. To gauge the performance and quality impact, the bilinear filter was implemented for the CPU as well as for the GPU. Both initial implementations handle the filtering in code because the hardware support for that filter was not available in the hardware we used at the time.

While nearest-neighbor sampling consumes only two rounding operations to get the closest coordinates and one sampling operation to get the final color values, bilinear interpolation is more expensive. First, the pixel coordinates surrounding the sampling point have to be calculated, which takes two rounding operations per coordinate dimension and two additions/subtractions depending on the implementation. Second, all four neighbors have to be sampled and the color values have to be combined. This calculation requires six floating-point multiplications to get the final value, for each color channel.

For a two-dimensional image with 3 color channels, nearest neighbor filtering requires a total of two rounding and one sampling operation. Bilinear filtering needs 4 rounding operations and 2 additions or subtractions to get the neighboring coordinates, in addition to 4 sampling operations, a total of 18 floating-point multiplications and 9 additions to retrieve all colors of a pixel, as depicted in Figure 7.6. While this does not seem like a lot, considering the speed of modern CPUs, it is important to remember that pixel sampling with filtering is one of the most used functions in the algorithm. For every vertex, whose depth needs to be estimated, an area around the projection coordinate has to be extracted pixel by pixel from every image in the closest neighbors set. This operation is then repeated for every depth plane. In a fairly normal scenario with 4 considered cameras, 20 depth planes, and a window size of 7x7, the estimation adds up to $7 \cdot 7 \cdot 20 \cdot 4 = 3920$ sampling operations per vertex.

Using the filter functionality of the hardware samplers on a GPU mitigates this difference because their implementations are highly optimized and their access to the texture data is very efficient. Figure 7.7 shows the impact filtering can have on the quality of the estimated depth values. The image shows a tree standing next to a hill whose slope is in the bottom right portion of the frame. While the left image shows periodic errors in the depth, the right image has a much smoother transition from foreground to background in the same area. This improvement can also be observed in the canopy of the tree and its stem. The noisy artifacts next to the tree, are only reduced slightly by this step. The disappearance of the steps in the depth map reduces the amount of subdivisions triggered at those locations and consequently lowers the overall complexity of the depth estimation. These periodic errors only appear when the camera layout of the array, the distance from the cameras,

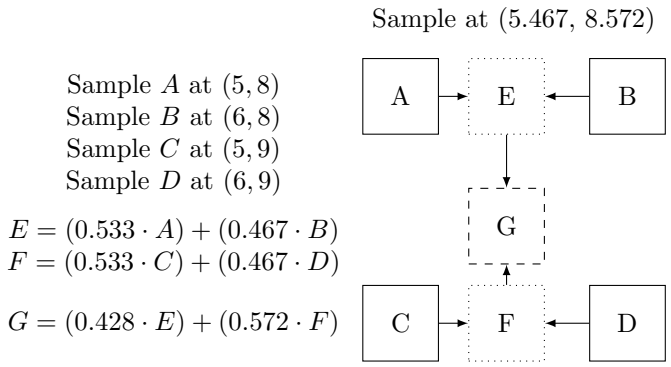
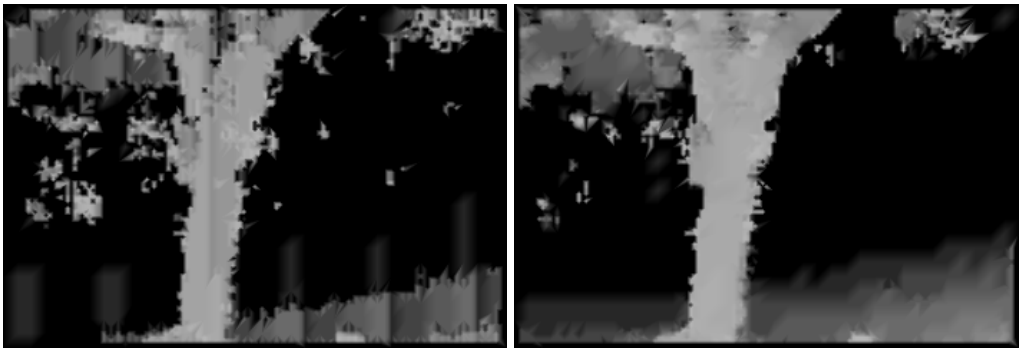



Figure 7.6.: Bilinear interpolation calculation example per channel.



(a) nearest-neighbor interpolation (b) bilinear interpolation

Figure 7.7.: Influence of different texture interpolation techniques in OpenCL.

and the properties of the texture line up correctly. Their removal without a big impact on the overall performance is important within the context of the overall improvement of the algorithm.

#### 7.4.3.2. Depth Filtering

The filtering of the texture during the sampling operations only has a small impact on the overall performance of the algorithm. The filtering of the depth estimate itself has greater benefits. Figure 7.8a shows the recovered depth map of the well-known Teddy scene from the Middlebury stereo dataset [177]. All over the image, but especially in areas where the depth should be fairly continuous or flat, artifacts reminiscent of salt-and-pepper noise can be found. Those artifacts stem from the fact that the depth estimation often defaults to the highest or lowest possible depth value when the analysis of the image similarity at the tested depth planes is not conclusive. While most of them are invisible in the final view, as they often coincide with uniformly textured areas in the scene, they have a big influence on the subdivision steps. As they create artificial edges where none should exist, they trigger unnecessary subdivisions and the estimation steps that come with them.

To remove salt-and-pepper noise without blurring edges and other structures in an image, median filters have been the go-to method for a long time. Since it is a non-linear filter, and apart from some recent implementations requires sorting [178], it is fairly expensive

to implement, especially on GPUs. Therefore, the filter can only be applied to a small neighborhood of vertices or the cost of the filter outweighs its benefits.

After evaluating multiple possible filter sizes and application variants, we settled on the following: a 5x5 median filter is applied only to the coarsest mesh after the first round of depth estimation. The other estimation rounds are left untouched. The influence of this step on the depth map can be seen in Figure 7.8b. The reduction of the overall noise is immediately visible. While the extreme values in the noisy areas are suppressed, the edges are mostly maintained. In regions where the noise comes close to or mixes with the edges, spiky artifacts from the edges are visible but their extent is much smaller than the errors in the original version. Overall, the estimated depth is much more uniform than before.

In the final views based on the depth maps (Figures 7.8c and 7.8d), the differences are much more subtle. The red circles in the left image highlight two areas where small errors disappear because of the applied filter. Even though the visual impact is fairly small, the overall computation time spent on the depth estimation is reduced by up to 50%. The reduction in false edges means fewer subdivisions are triggered in the later steps and therefore, fewer computations are performed overall. The exact speedup depends on the scene content, the current camera position, and the configured scene parameters, but 20-30% are very common and values of over 40% can often be observed in complicated scenes.

The reduction in complexity offered by the introduction of the filtering step is a very important step towards a consistent real-time performance of the complete view interpolation process. Since the depth estimation step takes up most of the available time budget, as shown in Section 7.5.1, the impact of this filtering step in this view interpolation system is only second to the transition from CPU to GPU.

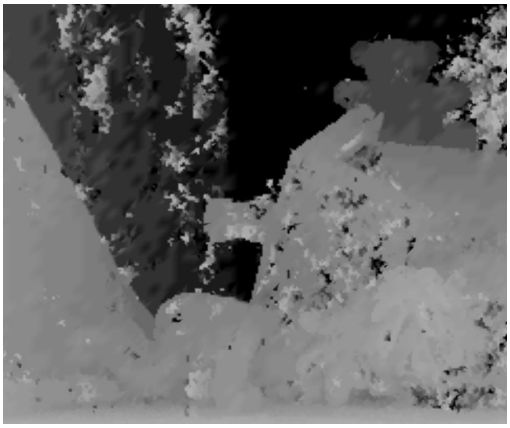
#### 7.4.3.3. Temporal Consistency

The improvements in the previous sections changed the overall concept of the original algorithm only slightly and simply add or change single steps. This section is different as it adds a dependency on earlier frames to reduce the rendering time and the flickering between consecutive frames. Up to now, the calculations for every frame were completely independent of earlier or later results. For rapidly changing scenes or camera positions, it can be beneficial when consecutive frames do not share a lot of content, since searching for matches between the frames takes some time and might not yield useful results.

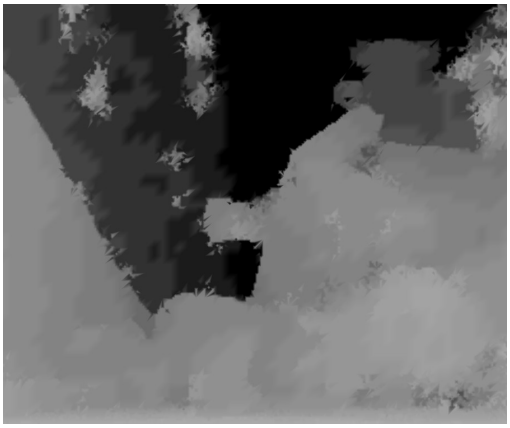
Considering scenes with smooth camera movements and no scene changes, it can be useful to take the results from earlier frames into account to reduce the overall number of required computations per frame. The introduction of temporal consistency into the view interpolation process can be used to achieve that.

Temporal consistency generally describes the fact that objects in the real world do maintain certain properties within a reasonable amount of time. In the context of this chapter, it means that static objects in one frame are still in the same location in the next frame, and moving objects only change their position slightly between frames. When optimally used, the depth values of static objects can be reused without any recomputation of data or loss of quality in the final image. Even the data of moving objects could be reused if the movement is tracked and extrapolated to the current frame. While this procedure seems straightforward, the tricky part is how to identify and track the static and dynamic parts of the scene without using all the time that can be gained by partially reusing the older depth values.

Different methods and implementations for adding temporal consistency into the existing algorithm have been evaluated in the preparatory work for [49]. Drawing inspiration from Kauff *et al.* [180] and Riechert *et al.* [179] from the Fraunhofer HHI, steps to exploit the



(a) depth without filter



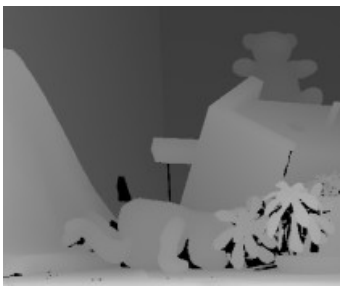
(b) depth with median filter

(c) result without filter

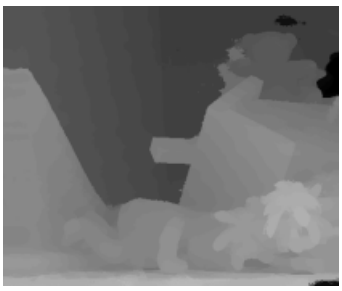
(d) result with median filter

Figure 7.8.: Depth filtering results with and without the new filter.

(a) left image



(b) ground truth



(c) estimated disparity

Figure 7.9.: Disparity results of LHRM [179].

temporal consistency, were added to the system. Their HRM (Hybrid Recursive Matching) and LHRM (Line-wise Hybrid Recursive Matching) algorithms provide impressive quality in the estimated depth representations, as shown by the example in Figure 7.9. The computations can be performed in a reasonable time and they scale well with the number of used CPU cores. A direct transfer of the presented algorithms from the papers was not possible, as they do not fully disclose some information about the metrics used in the iterative part of their algorithm and their overall structure is different from the one we have. Even though the structure of the HRM and LHRM algorithms is mostly incompatible with our algorithm, the integration of samples from earlier frames into its computations for temporal stability and complexity reduction can be used with our existing code. By adding disparity values from neighboring pixels and previous frames into the pool of considered values, enough knowledge is propagated to ensure spatial and temporal stability in the final disparity maps.

Even though our algorithm was already temporally stable because of how the results of the parallel computations on the GPU are ordered, testing whether previous results are still valid and skipping the further computations can reduce the overall complexity if the initial test is not too expensive. A first implementation stored the depth as well as the image patches used during the computation for consistency checks. Since the checks using the previous image content were too specific and finding an actually consistent point was only possible when neither the input frames nor the virtual camera position changed, it was quickly dismissed. The revised version only stores the depth value and the similarity score from the previous frame. Before the regular plane-sweeping part in the algorithm, the current vertex is projected onto the neighboring images using the saved depth and the similarity score for the new frames and camera position is calculated. If it is equal to or higher than the saved value from the earlier frame, the rest of the depth estimation is skipped for this vertex. In case that the similarity score is lower than before, the depth estimation is performed normally but the depth plane corresponding to the stored depth value is skipped because its similarity score was already calculated in the testing step. This approach can still produce sub-optimal results when the saved similarity score is low. In these situations, it is possible that a comparably poor depth value is propagated through multiple frames even though a better one would exist. Such bad estimation results can be easily matched and other depth values are therefore not tested. This procedure reduces the achievable quality compared to the unmodified algorithm, but its impact can be dampened by the introduction of a threshold that the similarity score has to surpass for the associated depth to be considered in future frames. Overall, this limits the false positives for the temporal consistency to high-quality matches, which have a far less negative impact on the final quality than the low similarity matches because of the better match between the areas surrounding the projected coordinates.

While these temporal false positives reduce the quality of the end result but improve the speed of the calculations, false negatives have no impact on the quality but decrease the performance gain. The main source of false negatives in the samples we tested is the natural camera noise, which adds a different amount of white noise to every frame. Since this noise is different in every camera and is included in the calculation of the similarity scores, it can increase or decrease the similarity value between frames, even in completely static areas of the scene. An improved score means the vertex is correctly assumed to be temporally consistent but makes it more likely that the score will decrease in the next frame. The decreasing scores are problematic, since a lower score triggers a reevaluation of all depth planes, even though the scene has not changed, and therefore, lowers the efficiency of the depth estimation. The introduction of a second threshold, which allows a worse similarity score to be accepted as "good enough" to be temporally consistent, solved this problem.

This threshold has to be tuned to the noise present in the scene, so it is high enough to cover the effects of the noise but still low enough that actual changes are still detected as changes.

The performance gains of this step were only evaluated in comparison to the CPU-based version but since the steps in the algorithm are the same and the depth estimation takes the majority of computation time in both implementations, the impact of the improvement is comparable. The most notable time changes are achieved in static scenes without camera movement. Depending on the configured parameters of the depth estimation, up to 82% reduction in rendering time per frame can be achieved, even though a reduction of approximately 50% is more realistic. This improvement can be considered a good result but this special test scenario is fairly unrealistic with respect to any practical use case.

Looking at a static scene with a moving virtual camera or having a scene with movement between the frames is much closer to reality. As movements in either the scene or the camera cause the vertices in the depth mesh to lose temporal consistency, the expected increase in speed is lower than in the static cases. With small steps between the frames, the chance for a vertex to fall onto the same depth plane is higher than for bigger steps. This relation is being reflected in our measurements as the gains drop to about 20% in these cases. With higher threshold values for considering worse similarity values as still temporally consistent, that value can be increased but only at the cost of artifacts in the depth map which trail moving objects in the scene.

When it comes to the impact the temporal consistency step has on the visual quality, it was found to be insignificant. Compared to the original unmodified algorithm, there was no measurable difference for the completely static scenes, neither in the PSNR nor the SSIM scores. When the static scene is combined with camera movement, the quality even goes up slightly due to a small smoothing effect the temporal consistency checks have onto the depth map in those cases.

For the tested dynamic scenes, there was no good ground truth available to test against, and removing frames from the input material to use as ground truth was not feasible because small errors in the camera calibration data led to visibly worse results compared to the unmodified input. Since the PSNR metric does not reflect the perceived quality very well anyways, as shown in Section 6, the quality of the view interpolation for the dynamic scenes was compared by multiple observers. They all judged it to be at least of similar quality, in some cases even better. The main reason for higher quality is the problematic areas in which the similarity score is hardly conclusive. In those, the temporal consistency removes some random flickering when the depth would switch between different depth planes in the original implementation.

Even though this improvement is not integrated into the high-speed GPU-based version of the algorithm, it still shows some of the potential that the algorithm has for future use cases and projects.

## 7.5. Evaluation

Most changes we made to the base algorithm were focused on lowering the amount of time needed for the computation of each frame. In this section, the impact of these changes with respect to the overall speed as well as the quality is measured. It shows that the improvements increase the processing speed so far that frames can be rendered much faster, even when more high-resolution input images are processed with more demanding settings.

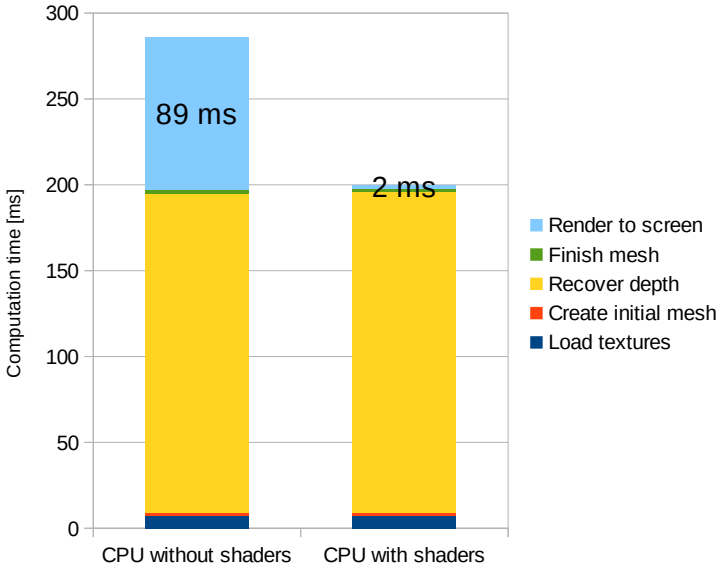



Figure 7.10.: Influence of shader-based OpenGL on the computation time of different algorithm steps.

### 7.5.1. Rendering Performance

For the algorithm this portion of the thesis is based on, the authors state a performance of 4-10 frames per second with an image resolution of 320x240 [13]. Even on current CPUs, at the time of writing, the performance was not significantly better. Nevertheless, the influence of every improvement was measured against the original algorithm on the same hardware platform. The measurements were taken on a machine with a 3rd generation Core i5 processor with an AMD Radeon 7970 as the main GPU. The default configuration for all following measurements are as follows: 9 input views with a resolution of 900x750, 10 depth planes, 10 pixels initial vertex distance, 2 levels of subdivision, and a window size for similarity tests of 5 pixels.

The first major change was the step from pure software-based OpenGL to shader-based OpenGL. Figure 7.10 shows how the time used for the different parts of the algorithm changes when the shaders are used. Since the shaders only influence the last step of the algorithm, the execution times of the other parts are unaffected. It is clearly visible that the shaders, in combination with the batch execution for all the triangles, are far more efficient than the old approach which renders triangles one after the other. The overhead produced by the original rendering step where the data for every vertex was transferred to the GPU individually, takes up most of the rendering time. Therefore, the capabilities of a modern GPU can not be used efficiently. One indicator for this is the fact that modern well-known GPU benchmarks, like 3DMark, render up to 14 million triangles per frame with up to 60 frames per second and more [181], while the view interpolation algorithm discussed here only needs to render approximately two million triangles, even with a FullHD input, a dense initial grid and one subdivision for every triangle. Additionally, it must be considered that the shader complexity in the benchmark is more complex with reflections and other lighting effects than the ones we use here. Therefore, it can be assumed that the

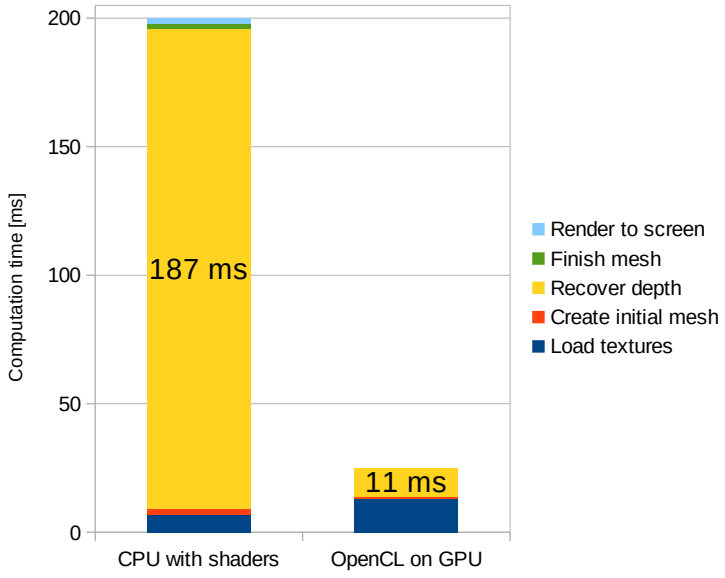
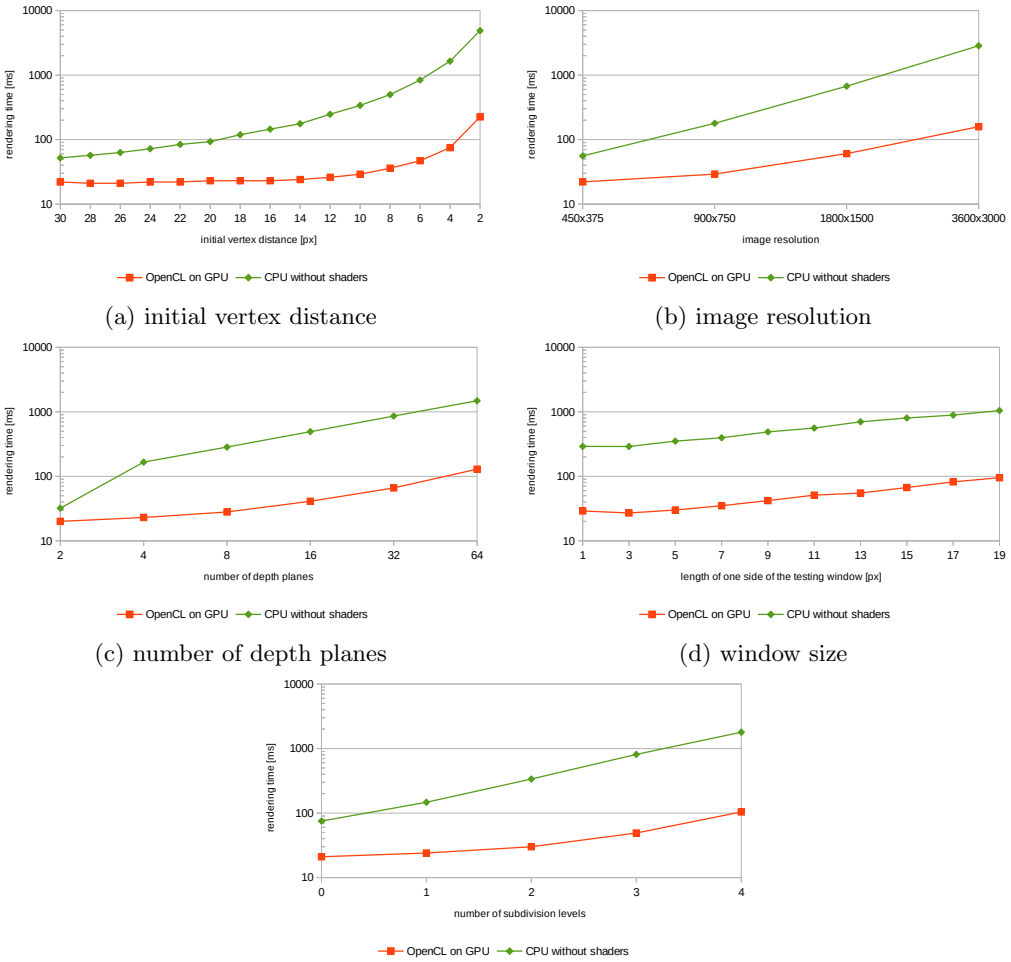



Figure 7.11.: Influence of OpenCL-based computations on the computation time of different algorithm steps.

efficient rendering of triangles with associated shaders is much faster than what we see in the original implementation. The reduction in time needed for the rendering supports this assumption.

The remaining time in the rendering step mostly consists of the time required to transfer the data for the shaders from RAM to GPU memory in the correct order. This fact becomes apparent in Figure 7.11 where the render time completely disappears from the graph because the data is already present on the GPU and the rendering itself takes less than 0.5ms and could not be measured anymore since the time resolution was limited to 1ms. Looking at the total computation times in Figure 7.10, the use of modern OpenGL rendering techniques alone reduces the total computation time by nearly 30% in this test scenario. In configurations with simple scene geometry but more triangles, the influence can be even more pronounced.

The next round of improvements included the switch from pure CPU computation to General Purpose Computation on Graphics Processing Unit (GPGPU) to make use of the fact that most computations in the algorithm are performed per vertex and are independent of each other. The difference in runtime is immediately visible in Figure 7.11. While the time used for the estimation of the depth map drops from 187ms to 11ms, the time it takes to load to input images or textures nearly doubles. The main reason for the increase in loading times is the fact that, in addition to loading the images from the disk and decoding the images, a transfer from the RAM to the GPU is required. Fortunately, this is balanced out by the lesser time taken by all other steps before and after the depth estimation. In these simpler steps of the algorithm, the faster memory throughput, combined with the highly parallel computation power of the GPU together reduce the time required for these steps to under 1ms and in some cases even under 0.5ms, which makes them disappear from the graph. In total, the use of a medium-tier GPU reduces the amount of time required for the complete render of a single frame by a factor of 8 for the default parameters chosen here.




(a) initial vertex distance

(b) image resolution

(c) number of depth planes

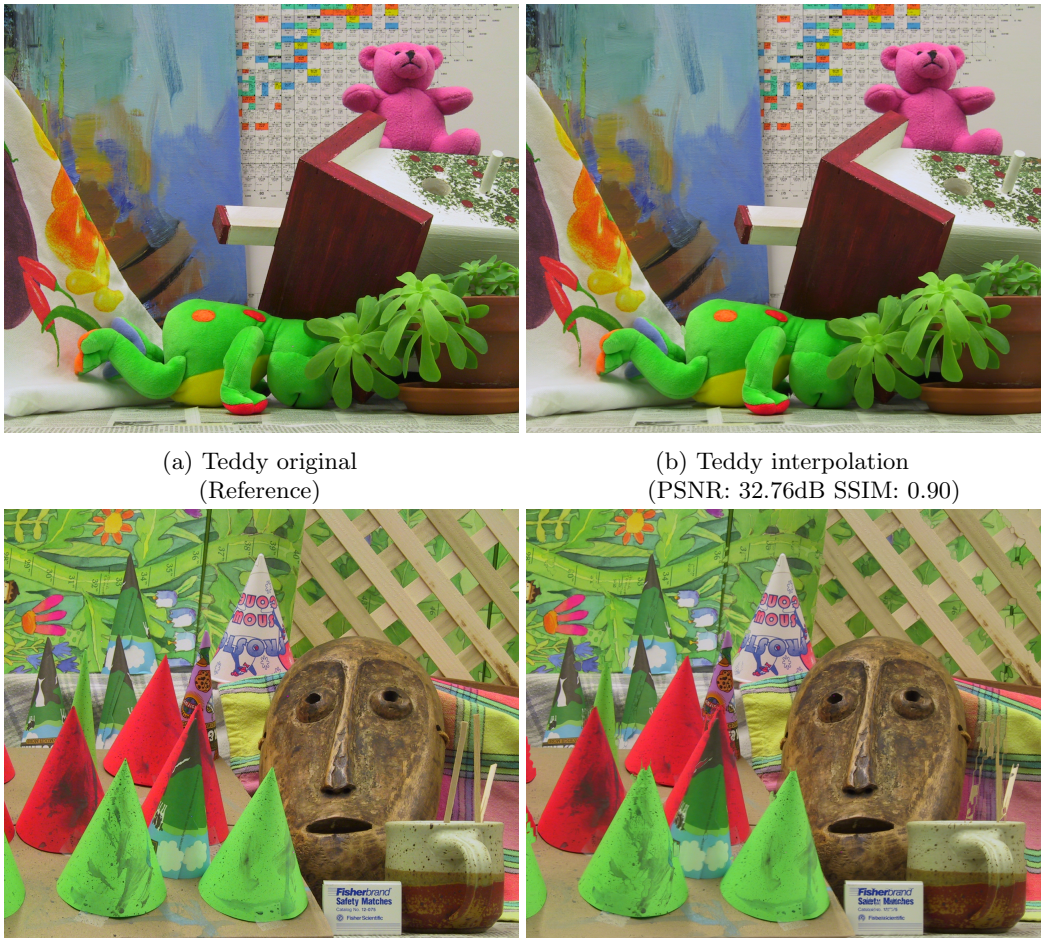
(d) window size

(e) levels of subdivision

Figure 7.12.: Influence of different parameters on the rendering time.

It is important to note that this factor is not the best one that can be achieved since the different configuration parameters have a different impact on the runtime. However, it gives a good indication of the improvement we achieved. The graphs in Figure 7.12 show how the rendering time changes when one configuration parameter is modified. They show a similar shape for the CPU and the GPU-based computations, even though they are approximately an order of magnitude apart from each other. For the initial vertex distance where this is not the case in the beginning, the number of vertices in the frames is not high enough to use all hardware resources of the GPU with full efficiency. This is also visible in the nearly horizontal line of the GPU times in the left half of the graph, as the GPU copes with the increased number of vertices by utilizing resources that have been idle before. Once all of the computational units are in use, the times increase as expected.

All these graphs only show the change for one parameter. When a configuration is adapted for a given scene, most likely more than one parameter needs to be changed to achieve



(a) Teddy original (Reference)

(b) Teddy interpolation (PSNR: 32.76dB SSIM: 0.90)

(c) Cones original (Reference)

(d) Cones interpolation (PSNR: 27.22dB SSIM: 0.84)

Figure 7.13.: Examples of interpolated images with metric values.

the best results. In those cases, it is important to know that the parameters are combined multiplicatively. When one parameter increases the number of vertices in the mesh and a second increases the number of steps that need to be taken per vertex, their influence does not just add up, it has to be multiplied together. With a clever combination of these parameters, even higher gains can be achieved on the GPU but they are not necessarily realistic, since the requirements for an optimal result are different from scene to scene and not all can achieve the same improvements.

### 7.5.2. Rendering Quality

As we have shown, the execution speed of our view interpolation algorithm has been greatly increased in comparison to the original baseline implementation. However, as with any other media application, speed is worth nothing without good quality. As we can see from the images in Figure 7.13, the visual quality of the interpolated frames is so high that nearly no differences to the reference can be seen. Very small shifts of image regions due to

slight inaccuracies in the depth reconstruction are not directly visible but influence common quality metrics significantly. Those shifts can be caused by a wrong depth chosen by the estimation, especially in regions with repeating patterns of matching frequencies, or simply the depth plane not matching the exact depth of an object perfectly due to the discrete way the space is sampled in the depth direction. The error in depth causes small shifts, which are less than a pixel away from the correct location most of the time, given an appropriately high number of depth planes is tested and the one closest to the real depth is chosen as the correct one. Nevertheless, the measured PSNR values are fairly low and go higher than the lower 30s in very rare cases. How to cope with this behavior in situations where more appropriate PSNR scores are required is investigated in detail in Chapter 6. On the other hand, the SSIM results are better and closer to the visual impression with values from 0.84 to 0.97. As it compares image structures and not just single pixels, the sub-pixel shifts do not cause such a significant decrease in the metric's final value.

Those quality results are slightly better than what the original implementation could achieve.The main reason for this is the newly introduced filtering step, which helps to reduce the number of wrong depth values in areas where the similarity scores are inconclusive due to occlusions or camera noise. In those regions, the depth map is now smoothed between edges, which is mostly correct in textureless areas or a better result than the extreme values which occurred there in the original implementation.

In summary, the changes made to the original algorithm do not significantly change the maximum achievable image quality. Considering this was not the goal of this thesis, that is not surprising. Even then, algorithms achieving better image qualities existed but they either relied on a much longer time budget, the existence of detailed depth maps in addition to the input frames, or hard assumptions about the scene contents [182]. With respect to other IBR algorithms of the time, the quality is comparable to the upper end of the spectrum [183, 184], especially considering the small time budget we have set for each frame.

### 7.5.3. Relevance for New Projects

Given the age of the original algorithm and the implementation discussed here, it is easy to question its value for this thesis. Since we published the last paper about this topic, there have been many developments. Faster algorithms for view interpolation from multiple cameras have been developed [185, 186, 187], neural nets have become even better at calculating depth maps [188, 189] but most importantly, interpreting multiview material as the views from distinct cameras has become slightly outdated. Instead, they are treated as sub-apertures of a lightfield and are often used as real-world samples for new, more efficient lightfield representations and rendering algorithms. Notable examples for this are the DeepView algorithm by Flynn *et al.* [190] and the follow-up work from Broxton *et al.* [191].

Even though the approaches they are describing in their publications look completely different from what we have done, their goal to create new dynamic views in the area covered by the cameras is essentially the same as ours. Additionally, there are clear parallels between their approaches and ours. The multiplane images in [190] represent different depths or distances from the cameras, similar to our depth planes. For their representation, they assign pixels from the cameras to these planes, where overlaying the contents of the input images result in sharp images. Therefore, the plane represents the correct depth for this portion of the image. The general approach up to this point is nearly identical, the main difference being that they use a sophisticated neural net to determine the correct depth and to choose the appropriate pixels to assign to each plane, whereas we use fairly simple image metrics for determining the depth and only indirectly assign pixels via the coordinates in the

virtual view. In [191], their process is refined by moving from flat to spherical surfaces and add a dictionary-based compression, which allows for real-time streaming of the compressed content and interactive viewpoint choice within modern browsers. While their results look impressive, even with very challenging scenes, and their way of creating novel views from the processed material is fast and accessible, it would not be possible to use them in the scenario we tried to tackle in this thesis. The main reason being that the processing of the input material takes very long compared to our time budget, namely about 50 seconds per frame with 4 cameras on a high-end professional GPU for DeepView and over 24 CPU hours per frame for the layered mesh representation. Nevertheless, their continued interest in the problem we set out to solve, and the fact that they are still relying on a related set of tools, albeit more advanced in the way they apply them, makes this work still somewhat relevant today. In particular, under the aspect that our approach allowed the usage of live material for the view interpolation without any preprocessing apart from the camera calibration.

# 8. Conclusion

This thesis tackles many topics related to multi-camera footage, from the capture of multiview video and 5D lightfields using a custom-built, modular camera array, via the compression and transmission of multiview video to real-time view interpolation on GPUs.

The main focus of this work lies in the camera array. We describe the reasoning behind construction details and the different steps the array went through before it reached its final state. By discussing the major productions performed using the array, we show that it can capture lightfield video with sufficient quality for use in professional production and that there are benefits to having a controllable fifth dimension in lightfields.

To increase the quality of the captured footage further, we presented our work-in-progress deep learning approach for better demosaicing of camera array footage. Even though it can only process footage with very low resolutions, it shows clear improvements over state-of-the-art single image algorithms.

With the amount of data lightfields and multiview video contain in their raw form, networks quickly become the bottleneck in interactive applications. Since most available encoding solutions are either very hardware-intensive or incredibly slow, we present our own approach. Based on the H.264/MVC standard, we created a solution that cleverly distributes the complexity over multiple devices to achieve significant increases in speed with only minimal losses in quality. At the time, and to the best of our knowledge, up to this day, it is the fastest encoding system for more than two views in one stream. In addition, there is still potential for further speed increase since many steps in our implementation are not particularly optimized compared to available single-stream encoders.

In order to unlock the immersive properties of the transmitted multiview material, we developed a view interpolation algorithm capable of performing view interpolation in real-time. By leveraging the computational power of modern GPUs, it is able to achieve more than 25 frames per second with resolutions higher than FullHD. The obtained quality is at least up to par with other fast algorithms which do not rely on depth maps as input.

Combining the view interpolation with the multiview encoder lead to a novel approach to the optimization of multiview streaming. We have shown that removing views from the stream, redistributing the available data rate, and reconstructing the missing frames after decoding using view interpolation can lead to a higher overall quality compared to all views sharing the available data rate.

To make reliable predictions about the quality of images after their transmission and reconstruction, we presented an approach that helps to align PSNR scores with the mean opinion score when used with images created by view interpolation. After analyzing the effects of typical errors caused by view interpolation, the least visible errors with the highest impact on the score are transferred from the tested image to the reference, and therefore, effectively made invisible to the applied metric. This approach maintains most characteristics of the original metric, such that its results can still be combined with measurements of the original metric.

While most of our approaches were a clear improvement over the state-of-the-art in at least one aspect at the time of their publication, they were not perfect. As part of ongoing research, there will always be cases in which they behave sub-optimally. With the continuous

development of new hardware, new features, and novel ways for software acceleration, it is never possible to support everything, and execution times could always be lower somehow. Since the other points we identified for improvement are highly dependent on the contribution in question, they are discussed in the respective chapters.

Overall, we contributed to the scientific progress of every step in a multiview video streaming pipeline. By restricting us to the use of consumer hardware in most topics, we presented ways in which complex media can be made accessible for consumers without special devices. Even though all results have been published at international conferences, not all have had the same impact on the research community. In particular, the interest in multiview-specific algorithms has decreased due to the current popularity of lightfields. While the format of the input material is very similar, the interpretation of the data is very different. Nevertheless, parts of the ideas presented in our approaches can be found in state-of-the-art lightfield processing and compression algorithms.

On the other hand, the lightfield array and the material created with it were featured at multiple international conferences as well as public and industry-specific media. With the help of multiple renowned visual effects companies, we showed that lightfields can be used in professional video editing suites, even though their support is still in its infancy. This experiment proves that lightfields work well with computer-generated content and has the potential to revolutionize the film industry when lightfield cameras and editing tools become more easily available.

# Own publications


[1] Tobias Lange, Goran Petrovic, and Thorsten Herfet. "Real-time virtual view rendering for video communication systems using OpenCL". In: *2014 IEEE International Symposium on Broadband Multimedia Systems and Broadcasting*. IEEE. 2014, pp. 1–5. DOI: 10.1109/bmsb.2014.6873485.

[2] Tobias Lange and Thorsten Herfet. "Compensation for sub-pixel image shifts in interpolated images when using common quality measures". In: *2015 9th International Workshop on Video Processing and Quality Metrics for Consumer Electronics (VPQM)*. 2015. eprint: https://www.nt.uni-saarland.de/wp-content/uploads/2019/05/2015_VPQM_Lange.pdf.

[3] Andreas Schmidt, Tobias Lange, and Thorsten Herfet. "Low-latency multimedia streaming using Open Networking Environments". In: *2016 2nd IEEE International Conference on Computer and Communications (ICCC)*. IEEE. 2016, pp. 2094–2098. DOI: 10.1109/compcomm.2016.7925069.

[4] Tobias Lange and Thorsten Herfet. "A complete multi-view video streaming system". In: *2017 IEEE 7th International Conference on Consumer Electronics-Berlin (ICCE-Berlin)*. IEEE. 2017, pp. 19–21. DOI: 10.1109/icce-berlin.2017.8210578.

[5] Tobias Lange and Thorsten Herfet. "A distributed real-time multi-view video encoder on consumer hardware". In: *2017 IEEE International Symposium on Broadband Multimedia Systems and Broadcasting (BMSB)*. IEEE. 2017, pp. 1–3. DOI: 10.1109/bmsb.2017.7986173.

[6] Tobias Lange and Thorsten Herfet. "Towards an optimized multiview streaming system with view interpolation". In: *2017 IEEE International Conference on Consumer Electronics (ICCE)*. IEEE. 2017, pp. 61–63. DOI: 10.1109/icce.2017.7889230.

[7] Harini Priyadarshini Hariharan, Tobias Lange, and Thorsten Herfet. "Low complexity light field compression based on pseudo-temporal circular sequencing". In: *2017 IEEE International Symposium on Broadband Multimedia Systems and Broadcasting (BMSB)*. IEEE. 2017, pp. 1–5. DOI: 10.1109/bmsb.2017.7986144.

[8] Thorsten Herfet, Tobias Lange, and Harini Priyadarshini Hariharan. "Enabling Multiview-and Light Field-Video for Veridical Visual Experiences". In: *2018 IEEE 4th International Conference on Computer and Communications (ICCC)*. IEEE. 2018, pp. 1705–1709. DOI: 10.1109/compcomm.2018.8780991.

[9] Thorsten Herfet, Tobias Lange, and Kelvin Chelli. "5D Light Field Video Capture". In: *Proceedings of the 16th ACM SIGGRAPH European Conference on Visual Media Production*. 2019. eprint: https://www.cvmp-conference.org/files/2019/short/33.pdf.

[10] Jonas Trottnow, Simon Spielmann, Tobias Lange, et al. "The Potential of Light Fields in Media Productions". In: *SIGGRAPH Asia 2019 Technical Briefs*. 2019, pp. 71–74. DOI: 10.1145/3355088.3365158.

[11] Kelvin Chelli, Tobias Lange, Thorsten Herfet, et al. “A Versatile 5D Light Field Capturing Array”. In: *NEM Summit 2020*. NEM. 2020. eprint: `https://nem-initiative.org/wp-content/uploads/2020/07/4-4-nem2020_kc_tl_th_camready.pdf?x98588`.

[12] Thorsten Herfet, Kelvin Chelli, Tobias Lange, et al. “Fristograms: Revealing and Exploiting Light Field Internals”. In: (July 22, 2021). arXiv: `2107.10563 [eess.IV]`.

# Bibliography


[13] Cha Zhang and Tsuhan Chen. "A self-reconfigurable camera array". In: *ACM SIGGRAPH 2004 Sketches*. 2004, p. 151. DOI: 10.1145/1186223.1186412.

[14] Bennett Wilburn, Neel Joshi, Vaibhav Vaish, et al. "High performance imaging using large camera arrays". In: *ACM SIGGRAPH 2005 Papers*. 2005, pp. 765–776. DOI: 10.1145/1186822.1073259.

[15] IETF. *Network Time Protocol (Version 3) Specification, Implementation and Analysis*. RFC 1305. Mar. 1992. DOI: 10.17487/RFC1305. URL: https://rfc-editor.org/rfc/rfc1305.txt.

[16] IEEE. "Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems". In: *IEEE 1588-2008* (July 2008). URL: https://standards.ieee.org/standard/1588-2008.html (visited on 08/18/2021).

[17] Teodor Neagoe, Valentin Cristea, and Logica Banica. "NTP versus PTP in Computer Networks Clock Synchronization". In: *2006 IEEE International Symposium on Industrial Electronics*. IEEE, July 2006. DOI: 10.1109/isie.2006.295613.

[18] Arthur Toussaint, Mohammed Hawari, and Thomas Heide Clausen. "Chasing Linux Jitter Sources for Uncompressed Video". In: *14th International Conference on Network and Service Management (CNSM)*. 2018 14th International Conference on Network and Service Management (CNSM). IEEE. Rome, Italy, Nov. 2018. URL: https://hal-polytechnique.archives-ouvertes.fr/hal-02263380.

[19] J. Weng, P. Cohen, and M. Herniou. "Camera calibration with distortion models and accuracy evaluation". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 14.10 (1992), pp. 965–980. DOI: 10.1109/34.159901.

[20] Z. Zhang. "A flexible new technique for camera calibration". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 22.11 (2000), pp. 1330–1334. DOI: 10.1109/34.888718.

[21] Yannick Hold-Geoffroy, Kalyan Sunkavalli, Jonathan Eisenmann, et al. "A Perceptual Measure for Deep Single Image Camera Calibration". In: *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*. IEEE, June 2018. DOI: 10.1109/cvpr.2018.00250.

[22] Manuel Lopez, Roger Mari, Pau Gargallo, et al. "Deep Single Image Camera Calibration With Radial Distortion". In: *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE, June 2019. DOI: 10.1109/cvpr.2019.01209.

[23] Zhen Liu, Qun Wu, Suining Wu, et al. "Flexible and accurate camera calibration using grid spherical images". In: *Optics Express* 25.13 (June 2017), p. 15269. DOI: 10.1364/oe.25.015269.

[24] Martin A. Fischler and Robert C. Bolles. "Random sample consensus". In: *Communications of the ACM* 24.6 (June 1981), pp. 381–395. DOI: 10.1145/358669.358692.

[25] Marc Levoy and Pat Hanrahan. "Light field rendering". In: *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques - SIGGRAPH '96*. ACM Press, 1996. DOI: 10.1145/237170.237199.

[26] Steven J. Gortler, Radek Grzeszczuk, Richard Szeliski, et al. "The lumigraph". In: *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques - SIGGRAPH '96*. ACM Press, 1996. DOI: 10.1145/237170.237200.

[27] Michael Landy and J. Anthony Movshon. "The Plenoptic Function and the Elements of Early Vision". In: *Computational Models of Visual Processing*. MIT Press, 1991, pp. 3–20.

[28] Michael Zink. "Blu-ray 3D™". In: *SMPTE International Conference on Stereoscopic 3D for Media and Entertainment*. SMPTE. 2010, pp. 1–8. DOI: https://doi.org/10.5594/M001412.

[29] Masayuki Tanimoto, Toshiaki Fujii, and Norishige Fukushima. "1D Parallel Test Sequences for MPEG-FTV Status: For discussion Source: Nagoya University". In: (Apr. 2008). URL: http://www.fujii.nuee.nagoya-u.ac.jp/multiview-data/mpeg2/mpegCamPara/m15378.doc.

[30] Masayuki Tanimoto. "FTV (free-viewpoint television)". In: *APSIPA Transactions on Signal and Information Processing* 1 (2012). DOI: 10.1016/j.image.2012.02.016.

[31] ITU. *H.264 - Advanced video coding for generic audiovisual services*. 06/19. International Telecommunications Union. June 2019. URL: https://www.itu.int/rec/T-REC-H.264-201906-I/en.

[32] ITU. *H.265 - High efficiency video coding*. 11/19. International Telecommunications Union. Nov. 2019. URL: https://www.itu.int/rec/T-REC-H.265-201911-I/en.

[33] Sakila S. Jayaweera, Chamira U. S. Edussooriya, Chamith Wijenayake, et al. "Multi-Volumetric Refocusing of Light Fields". In: *IEEE Signal Processing Letters* 28 (2021), pp. 31–35. DOI: 10.1109/lsp.2020.3043990.

[34] Sven Wanner and Bastian Goldluecke. "Spatial and Angular Variational Super-Resolution of 4D Light Fields". In: *Computer Vision – ECCV 2012*. Springer Berlin Heidelberg, 2012, pp. 608–621. DOI: 10.1007/978-3-642-33715-4_44.

[35] IETF. *Bootstrap Protocol*. RFC 951. Sept. 1985. DOI: 10.17487/RFC0951. URL: https://rfc-editor.org/rfc/rfc951.txt.

[36] Mike Henry, Eric Dittert, Vish Viswanathan, et al. *Intel Preboot Execution Environment*. Internet-Draft draft-henry-remote-boot-protocol-00. Work in Progress. Internet Engineering Task Force, July 1999. 16 pp. URL: https://datatracker.ietf.org/doc/html/draft-henry-remote-boot-protocol-00.

[37] Frank Waßmuth. "Adaptive Control Infrastructure for Scalabale Multi-View Camera Arrays". Master's Thesis. Saarland University, Feb. 2018.

[38] Hewlett-Packard Company, Intel Corporation, Microsoft Corporation, et al. *Universal Serial Bus 3.0 Specification*. Whitepaper. USB-IF, Nov. 2008.

[39] James O Benson, John J Prevost, and Paul Rad. "Survey of automated software deployment for computational and engineering research". In: *2016 Annual IEEE Systems Conference (SysCon)*. IEEE. 2016, pp. 1–6. DOI: 10.1109/syscon.2016.7490666.

[40] Ankur Datta, Jun-Sik Kim, and Takeo Kanade. "Accurate camera calibration using iterative refinement of control points". In: *2009 IEEE 12th International Conference on Computer Vision Workshops, ICCV Workshops* (2009), pp. 1201–1208. DOI: 10.1109/iccvw.2009.5457474.

[41] S. Garrido-Jurado, R. Muñoz-Salinas, F.J. Madrid-Cuevas, et al. "Automatic generation and detection of highly reliable fiducial markers under occlusion". In: *Pattern Recognition* 47.6 (2014), pp. 2280–2292. ISSN: 0031-3203. DOI: 10.1016/j.patcog.2014.01.005. URL: https://www.sciencedirect.com/science/article/pii/S0031320314000235.

[42] Edwin Olson. "AprilTag: A robust and flexible visual fiducial system". In: *2011 IEEE International Conference on Robotics and Automation*. IEEE. 2011, pp. 3400–3407. DOI: 10.1109/icra.2011.5979561.

[43] Hendrik Schilling, Maximilian Diebold, Marcel Gutsche, et al. "A fractal calibration pattern for improved camera calibration". In: *Forum Bildverarbeitung 2016*. 2016. DOI: 10.5445/KSP/1000059899. URL: https://doi.org/10.5445/KSP/1000059899.

[44] Sandro Esquivel, Yuan Gao, Tim Michels, et al. *Synchronized Data Capture and Calibration of a Large-Field-of-View Moving Multi-Camera Light Field Rig*. Talk at 3DTV-CON 2016, Workshop on Light Field Capture and Processing. July 2016. eprint: https://www.informatik.uni-kiel.de/~sae/docs/3DTV16-Workshop-Esquivel.pdf.

[45] Andrei Zaharescu, Radu Horaud, Rémi Ronfard, et al. "Multiple Camera Calibration using Robust Perspective Factorization". In: *3D Data Processing, Visualization and Transmission*. Ed. by IEEE. Chapel Hill, United States, 2006, pp. 504–511. DOI: 10.1109/3dpvt.2006.100. URL: https://hal.inria.fr/inria-00545155.

[46] Andrei Zaharescu and Radu Horaud. "Robust factorization methods using a gaussian/uniform mixture model". In: *International Journal of Computer Vision* 81.3 (2009), pp. 240–258. eprint: https://arxiv.org/abs/2012.08243.

[47] Tomáš Svoboda. "Quick guide to multi-camera self-calibration". In: *ETH, Swiss Federal Institute of Technology, Zurich, Tech. Rep. BiWi-TR-263, http://www. vision. ee. ethz. ch/svoboda/SelfCal* (2003). eprint: https://cmp.felk.cvut.cz/~svoboda/SelfCal/Publ/selfcal.pdf.

[48] Peter Todorov. "Multi-camera calibration". Bachelor's Thesis. Tampere University, May 2019. eprint: https://trepo.tuni.fi//handle/123456789/27414.

[49] Johannes Reuter. "Adding temporal consistency to an existing view interpolation algorithm". Master's Thesis. Saarland University, July 2019.

[50] Howard Frazier–Broadcom, Schelto Van Doorn–Intel, Robert Hays-Intel, et al. "IEEE 802.3 ad Link Aggregation (LAG)". In: (2007).

[51] Jessy Rouye Stephen Haddock. "IEEE 802.1AX-2020 – Link Aggregation". In: (2020).

[52] Huawei. *What Is LACP? How Does LACP Work?* URL: https://support.huawei.com/enterprise/en/doc/EDOC1100086560#EN-US_TOPIC_0169439602 (visited on 08/18/2021).

[53] HP Enterprises. *Trunk load balancing using port layers*. URL: https://techhub.hpe.com/eginfolib/networking/docs/switches/WB/15-18/5998-8162_wb_2920_mcg/content/ch04s11.html (visited on 08/18/2021).

[54] Intel Corporation. *Power Adapter and Power Cord Specifications for Intel® NUC Products*. URL: https://www.intel.com/content/www/us/en/support/articles/000007053 (visited on 08/18/2021).

[55] Intel Corporation. *Intel® NUC Kit NUC6i5SYK*. URL: https://ark.intel.com/content/www/us/en/ark/products/89188/intel-nuc-kit-nuc6i5syk.html (visited on 08/18/2021).

[56] Intel Corporation. *ATX Multi Rail Desktop Power Supply Design Guide*. June 2020. URL: https://cdrdv2.intel.com/v1/dl/getContent/336521 (visited on 08/18/2021).

[57] Bosch Rexroth AG. *Einführung Strebenprofile*. URL: https://www.boschrexroth.com/ics/cat/content/assets/Online/do/Strut_profiles_MGE_14_DE_2019-07_20191202_155700.pdf (visited on 08/18/2021).

[58] 3A Composites GmbH. *DIBOND product information*. Jan. 2020. URL: https://media.3acomposites.com/pdf/dibond/EN_DIBOND_ProductGuide_01-2020.pdf (visited on 08/18/2021).

[59] FLIR Systems, Inc. *Blackfly USB3 Technical Reference*. Aug. 30, 2018. URL: https://flir.app.boxcn.net/s/jw17hga6i36z7cfifd6l0rw3jma9ow9g/file/418588576484 (visited on 08/18/2021).

[60] NXP Semiconductors. *UM10204 - I2C-bus specification and user manual*. Apr. 4, 2014. URL: https://www.nxp.com/docs/en/user-guide/UM10204.pdf (visited on 08/18/2021).

[61] Linear Technology Corporation. *LTC4307 - Low Offset Hot Swappable 2-Wire Bus Buffer with Stuck Bus Recovery*. 2018. URL: https://www.analog.com/media/en/technical-documentation/data-sheets/4307f.pdf (visited on 08/18/2021).

[62] NXP Semiconductors. *PCA9615 - 2-channel multipoint Fast-mode Plus differential I2C-bus buffer with hot-swap logic*. May 10, 2016. URL: https://www.nxp.com/docs/en/data-sheet/PCA9615.pdf (visited on 08/18/2021).

[63] C Lawrence Zitnick, Sing Bing Kang, Matthew Uyttendaele, et al. "High-quality video view interpolation using a layered representation". In: *ACM Transactions on Graphics (TOG)*. Vol. 23. 2004, pp. 600–608. DOI: 10.1145/1186562.1015766.

[64] Matthias Ziegler, Joachim Keinert, Nina Holzer, et al. "Immersive virtual reality for live-action video using camera arrays". In: *IBC Conference, Amsterdam, Netherlands*. 2017, pp. 1–8.

[65] Michael Broxton, Jay Busch, Jason Dourgarian, et al. "A low cost multi-camera array for panoramic light field video capture". In: *SIGGRAPH Asia 2019 Posters*. 2019, pp. 1–2. DOI: 10.1145/3355056.3364593.

[66] VDE. *DIN VDE 0298-4 - Verwendung von Kabeln und isolierten Leitungen für Starkstromanlagen*. Norm. June 2013. URL: https://www.vde-verlag.de/normen/0298016/din-vde-0298-4-vde-0298-4-2013-06.html.

[67] LLC MOLEX. *MINI-FIT JR. product specification*. Nov. 12, 2020. URL: https://www.molex.com/pdm_docs/ps/PS-5556-001-001.pdf (visited on 08/18/2021).

[68] TIA. *TIA-568 - Commercial Building Telecommunications Cabling Standards*. standard. Dec. 2015.

[69] K Sollins. *The TFTP protocol (revision 2)*. Tech. rep. STD 33, RFC 1350, MIT, 1992.

[70] nPerf SAS. *Barometer von festen Internet-Verbindungen in Deutschland*. July 17, 2019. URL: https://media.nperf.com/files/publications/DE/2019-07-17_Barometer-festen-internet-verbindungen-2019-S1.pdf (visited on 08/18/2021).

[71] Yupu Zhang, Abhishek Rajimwale, Andrea C Arpaci-Dusseau, et al. "End-to-end Data Integrity for File Systems: A ZFS Case Study." In: *FAST*. 2010, pp. 29–42. eprint: https://www.usenix.org/legacy/event/fast10/tech/full_papers/fast10proceedings.pdf#page=37.

[72] Sage A Weil, Scott A Brandt, Ethan L Miller, et al. "Ceph: A scalable, high-performance distributed file system". In: *Proceedings of the 7th symposium on Operating systems design and implementation*. 2006, pp. 307–320. eprint: https://www3.nd.edu/~dthain/courses/cse40771/spring2007/psnowber-ceph.pdf.

[73] Hervé Rousseau, Belinda Chan Kwok Cheong, Cristian Contescu, et al. "Providing large-scale disk storage at CERN". In: *EPJ Web of Conferences*. Vol. 214. EDP Sciences. 2019, p. 04033. DOI: 10.1051/epjconf/201921404033.

[74] Emulex Inc. / Broadcom Inc. *Cabling Guide for 10GbE Network Adapters*. 2012. URL: https://docs.broadcom.com/doc/12356169 (visited on 08/18/2021).

[75] Donald G Dansereau, Oscar Pizarro, and Stefan B Williams. "Decoding, Calibration and Rectification for Lenselet-Based Plenoptic Cameras". In: *Proceedings of the 2013 IEEE Conference on Computer Vision and Pattern Recognition*. 2013, pp. 1027–1034. DOI: 10.1109/cvpr.2013.137.

[76] Donald G Dansereau, Oscar Pizarro, and Stefan B Williams. "Linear Volumetric Focus for Light Field Cameras." In: *ACM Trans. Graph.* 34.2 (2015), pp. 15–1. DOI: 10.1145/2665074.

[77] Antti Laäaäperi, Ilkka Hyytiäinen, Terhi Mustonen, et al. "OLED Lifetime Issues in Mobile Phone Industry". In: *SID Symposium Digest of Technical Papers*. Vol. 38. 1. Wiley Online Library. 2007, pp. 1183–1187.

[78] Hans Strasburger. *Siemens star (128 spokes) & Matlab code*. 2018. URL: https://commons.wikimedia.org/wiki/File:Siemens_star_(128_spokes)_%26_Matlab_code.svg (visited on 08/18/2021).

[79] Dave Coffin. *DCRAW: Decoding raw digital photos in linux*. 2008. URL: http://www.dechifro.org/dcraw (visited on 08/18/2021).

[80] Florian Kainz, Rod Bogart, and Piotr Stanczyk. "Technical introduction to OpenEXR". In: *Industrial light and magic* (2009), p. 21. eprint: https://www.openexr.com/documentation/TechnicalIntroduction.pdf.

[81] Viorela Ila, Lukas Polok, Marek Solony, et al. "SLAM++-A highly efficient and temporally scalable incremental SLAM framework". In: *The International Journal of Robotics Research* 36.2 (2017), pp. 210–230. DOI: 10.1177/0278364917691110.

[82] Yichao Xu, Kazuki Maeno, Hajime Nagahara, et al. "Camera array calibration for light field acquisition". In: *Frontiers of Computer Science* 9.5 (2015), pp. 691–702. DOI: 10.1007/s11704-015-4237-4.

[83] Herbert Bay, Tinne Tuytelaars, and Luc Van Gool. "Surf: Speeded up robust features". In: *European conference on computer vision*. Springer. 2006, pp. 404–417. DOI: 10.1007/11744023_32.

[84] David G Lowe. "Distinctive image features from scale-invariant keypoints". In: *International journal of computer vision* 60.2 (2004), pp. 91–110. DOI: 10.1023/b:visi.0000029664.99615.94.

[85] Danny Pascale. "RGB coordinates of the Macbeth ColorChecker". In: *The BabelColor Company* 6 (2006). eprint: https://www.babelcolor.com/index_htm_files/RGB%20Coordinates%20of%20the%20Macbeth%20ColorChecker.pdf.

[86] Mairead Grogan and Aljosa Smolic. "L2 based Colour Correction for Light Field Arrays". In: *Proceedings of the 16th ACM SIGGRAPH European Conference on Visual Media Production*. 2019. DOI: https://v-sense.scss.tcd.ie/wp-content/uploads/2019/12/CVMP2019_mairead2.pdf.

[87] Adobe Systems Incorporated. *Cube LUT Specification Version 1.0*. 2013. URL: https://wwwimages2.adobe.com/content/dam/acom/en/products/speedgrade/cc/pdfs/cube-lut-specification-1.0.pdf (visited on 08/18/2021).

[88] Keigo Hirakawa and Thomas W Parks. "Adaptive homogeneity-directed demosaicing algorithm". In: *Ieee transactions on image processing* 14.3 (2005), pp. 360–369. DOI: 10.1109/tip.2004.838691.

[89] Paul Hellard. "The SAUCE". In: *VFX Science* (Apr. 2019). URL: https://vfxscience.com/2019/04/22/the-sauce.

[90] Ian Failes. "Light Fields and the Future of VFX". In: *VFXVoice - Tech & Tools Fall 2019* (2019), pp. 24–28. URL: https://www.vfxvoice.com/light-fields-and-the-future-of-vfx.

[91] Dan Ring. "Quest for Reality - A primer on Light Field technology". In: *CGW 02.2020* (Sept. 2020), pp. 63–64. URL: http://digital.copcomm.com/i/1277231-edition-2-2020/63?.

[92] Matthias Bolliger. "Bewegtbild 5.0". In: *Film & TV KAMERA* (Oct. 2019), pp. 38–41.

[93] Harini Priyadarshini Hariharan and Thorsten Herfet. "Optimal Predictive Coding of 5D Light Fields". In: *2020 IEEE International Symposium on Broadband Multimedia Systems and Broadcasting (BMSB)*. IEEE. 2020, pp. 1–3. DOI: 10.1109/bmsb49480.2020.9379917.

[94] David Bařina, Tomáš Chlubna, Marek Šolony, et al. "Evaluation of 4D Light Field Compression Methods". In: *International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision (WSCG), Part I*. Union Agency: Union Agency, May 2019. Chap. 159971, pp. 55–61. DOI: 10.24132/CSRN.2019.2901.1.7.

[95] Yusuke Monno, Daisuke Kiku, Masayuki Tanaka, et al. "Adaptive residual interpolation for color image demosaicking". In: *2015 IEEE International Conference on Image Processing (ICIP)*. IEEE. 2015, pp. 3861–3865. DOI: 10.1109/icip.2015.7351528.

[96] Nai-Sheng Syu, Yu-Sheng Chen, and Yung-Yu Chuang. "Learning deep convolutional networks for demosaicing". In: *arXiv preprint arXiv:1802.03769* (2018). eprint: https://arxiv.org/abs/1802.03769.

[97] Kodak. *Kodak Lossless True Color Image Suite - PhotoCD PCD0992*. URL: http://r0k.us/graphics/kodak (visited on 08/18/2021).

[98] Edward Chang, Shiufun Cheung, and Davis Y Pan. "Color filter array recovery using a threshold-based variable number of gradients". In: *Sensors, Cameras, and Applications for Digital Photography*. Vol. 3650. International Society for Optics and Photonics. 1999, pp. 36–43.

[99] Chuan-kai Lin. *Pixel grouping for color filter array demosaicing*. 2003.

[100] Joan Duran and Antoni Buades. "A demosaicking algorithm with adaptive inter-channel correlation". In: *Image Processing On Line* 5 (2015), pp. 311–327. DOI: 10.5201/ipol.2015.145.

[101] Alexander Blatt. "Extending Deep Convolutional Demosaicing to Camera Arrays". Master's Thesis. Saarland University, Dec. 2019.

[102] Bartlomiej Wronski, Ignacio Garcia-Dorado, Manfred Ernst, et al. "Handheld multi-frame super-resolution". In: *ACM Transactions on Graphics* 38.4 (July 2019), pp. 1–18. DOI: 10.1145/3306346.3323024.

[103] Clement Godard, Oisin Mac Aodha, Michael Firman, et al. "Digging Into Self-Supervised Monocular Depth Estimation". In: *2019 IEEE/CVF International Conference on Computer Vision (ICCV)*. IEEE, Oct. 2019. DOI: 10.1109/iccv.2019.00393.

[104] Zhenyu Zhang, Stéphane Lathuilière, Andrea Pilzer, et al. "Online Adaptation through Meta-Learning for Stereo Depth Estimation". In: (Apr. 17, 2019). arXiv: 1904.08462 [cs.CV].

[105] Günter Klambauer, Thomas Unterthiner, Andreas Mayr, et al. "Self-Normalizing Neural Networks". In: *Advances in Neural Information Processing Systems 30 (NIPS 2017)* (June 8, 2017). arXiv: https://arxiv.org/abs/1706.02515 [cs.LG].

[106] Martin Abadi, Paul Barham, Jianmin Chen, et al. "Tensorflow: A system for large-scale machine learning". In: *12th USENIX symposium on operating systems design and implementation (OSDI 16)*. 2016, pp. 265–283. eprint: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi.

[107] Kaiming He, Xiangyu Zhang, Shaoqing Ren, et al. "Deep Residual Learning for Image Recognition". In: (Dec. 10, 2015). DOI: 10.1109/cvpr.2016.90. arXiv: 1512.03385 [cs.CV].

[108] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "Imagenet classification with deep convolutional neural networks". In: *Advances in neural information processing systems* 25 (2012), pp. 1097–1105. DOI: 10.1145/3065386.

[109] Yao Yao, Zixin Luo, Shiwei Li, et al. "MVSNet: Depth Inference for Unstructured Multi-view Stereo". In: *Computer Vision - ECCV 2018*. Springer International Publishing, 2018, pp. 785–801. DOI: 10.1007/978-3-030-01237-3_47.

[110] Touradj Ebrahimi, Siegfried Foessel, Fernando Pereira, et al. "JPEG Pleno: Toward an Efficient Representation of Visual Reality". In: *IEEE MultiMedia* 23.4 (Oct. 2016), pp. 14–20. DOI: 10.1109/mmul.2016.64.

[111] Péter Tamás Kovács, Zsolt Nagy, Attila Barsi, et al. "Overview of the applicability of H.264/MVC for real-time light-field applications". In: *2014 3DTV-Conference: The True Vision-Capture, Transmission and Display of 3D Video (3DTV-CON)*. IEEE. 2014, pp. 1–4. DOI: 10.1109/3dtv.2014.6874744.

[112] NA Bahran, W El-Shafai, A Zekry, et al. "An FPGA design and implementation of EPZS motion estimation algorithm for 3D H. 264/MVC standard". In: *Multimedia Tools and Applications* 78.16 (2019), pp. 22351–22396. DOI: 10.1007/s11042-019-7562-z.

[113] Seif Allah El Mesloul Nasri, Abdul Hamid Sadka, Noureddine Doghmane, et al. "Multiview Video Coding: A Comparative Study Between MVC and MV-HEVC". In: *Recent Trends in Computer Applications*. Springer International Publishing, 2018, pp. 99–112. DOI: 10.1007/978-3-319-89914-5_7.

[114] Qiuwen Zhang, Zhifeng Zhang, Bin Jiang, et al. "Fast 3D-HEVC encoder algorithm for multiview video plus depth coding". In: *Optik* 127.20 (Oct. 2016), pp. 8864–8873. DOI: 10.1016/j.ijleo.2016.06.091.

[115] Pekka Astola, Luis A da Silva Cruz, Eduardo AB da Silva, et al. "JPEG Pleno: Standardizing a coding framework and tools for plenoptic imaging modalities". In: *ITU Journal: ICT Discoveries* (2020). eprint: http://urn.fi/URN:NBN:fi:tuni-202101211570.

[116] Jill M. Boyce, Renaud Dore, Adrian Dziembowski, et al. "MPEG Immersive Video Coding Standard". In: *Proceedings of the IEEE* (2021), pp. 1–16. DOI: 10.1109/jproc.2021.3062590.

[117] P. Merkle, A. Smolic, K. Muller, et al. "Efficient Prediction Structures for Multiview Video Coding". In: *IEEE Transactions on Circuits and Systems for Video Technology* 17.11 (Nov. 2007), pp. 1461–1473. DOI: 10.1109/tcsvt.2007.903665.

[118] Kim Hao Josef Nguyen. "A fast H.264/MVC transcoder". Bachelor's Thesis. Saarland University, June 2016.

[119] Kim Hao Josef Nguyen. "A hierarchical H.264/MVC encoder with full inter-view support". Master's Thesis. Saarland University, Nov. 2018.

[120] Christian Keimel, Arne Redl, and Klaus Diepold. "Comparison of HDTV formats in a consumer environment". In: *Image Quality and System Performance VIII*. Ed. by Susan P. Farnand and Frans Gaykema. SPIE, Jan. 2011. DOI: 10.1117/12.876781.

[121] Joshan Meenowa, David S. Hands, Rhea Young, et al. "Subjective assessment of HDTV content: comparison of quality across HDTV formats". In: *Human Vision and Electronic Imaging XV*. Ed. by Bernice E. Rogowitz and Thrasyvoulos N. Pappas. SPIE, Feb. 2010. DOI: 10.1117/12.838809.

[122] Jeroen Doggen and Filip Van der Schueren. "Design and simulation of a H.264 AVC video streaming model". In: *2008 European Conference on the Use of Modern Information and Communication Technologies*. 2008.

[123] Yue Chen, Debargha Murherjee, Jingning Han, et al. "An overview of core coding tools in the AV1 video codec". In: *2018 Picture Coding Symposium (PCS)*. IEEE. 2018, pp. 41–45. DOI: 10.1109/pcs.2018.8456249.

[124] Niklas Carlsson, Derek Eager, Vengatanathan Krishnamoorthi, et al. "Optimized adaptive streaming of multi-video stream bundles". In: *IEEE transactions on multimedia* 19.7 (2017), pp. 1637–1653. DOI: 10.1109/tmm.2017.2673412.

[125] Mylène C. Q. Farias and Sanjit K. Mitra. "Perceptual contributions of blocky, blurry, noisy, and ringing synthetic artifacts to overall annoyance". In: *Journal of Electronic Imaging* 21.4 (Nov. 2012), p. 043013. DOI: 10.1117/1.jei.21.4.043013.

[126] Xin Zhao, Jun Sun, Siwei Ma, et al. "Novel statistical modeling, analysis and implementation of rate-distortion estimation for H. 264/AVC coders". In: *IEEE Transactions on Circuits and Systems for Video Technology* 20.5 (2010), pp. 647–660. DOI: 10.1109/tcsvt.2010.2045803.

[127] Alexander Tanchenko. "Visual-PSNR measure of image quality". In: *Journal of Visual Communication and Image Representation* 25.5 (2014), pp. 874–878. DOI: 10.1016/j.jvcir.2014.01.008.

[128] Piotr Romaniak, Lucjan Janowski, Mikolaj Leszczuk, et al. "Perceptual quality assessment for H.264/AVC compression". In: *2012 IEEE Consumer Communications and Networking Conference (CCNC)*. IEEE, Jan. 2012. DOI: 10.1109/ccnc.2012.6181021.

[129] Zhou Wang, Alan C Bovik, Hamid R Sheikh, et al. "Image quality assessment: from error visibility to structural similarity". In: *IEEE transactions on image processing* 13.4 (2004), pp. 600–612. DOI: 10.1109/tip.2003.819861.

[130] Pascal Straub. "Optimized dynamic multi-view video streaming using view interpolation". Master's Thesis. Saarland University, Feb. 2016.

[131] Aaron Mavrinac, Jose L Alarcon Herrera, and Xiang Chen. "A fuzzy model for coverage evaluation of cameras and multi-camera networks". In: *Proceedings of the Fourth ACM/IEEE International Conference on Distributed Smart Cameras*. 2010, pp. 95–102. DOI: 10.1145/1865987.1866003.

[132] Mikael Le Pendu, Christine Guillemot, and Aljosa Smolic. "A fourier disparity layer representation for light fields". In: *IEEE Transactions on Image Processing* 28.11 (2019), pp. 5740–5753. DOI: 10.1109/tip.2019.2922099.

[133] David C Schedl, Clemens Birklbauer, and Oliver Bimber. "Optimized sampling for view interpolation in light fields using local dictionaries". In: *Computer Vision and Image Understanding* 168 (2018), pp. 93–103. DOI: 10.1016/j.cviu.2017.06.009.

[134] Harini Priyadarshini Hariharan and Thorsten Herfet. "On The Implication Of Light Field Compression On Post-Processing Algorithms". In: *2019 IEEE International Symposium on Broadband Multimedia Systems and Broadcasting (BMSB)*. IEEE. 2019, pp. 1–4. DOI: 10.1109/bmsb47279.2019.8971923.

[135] Caroline Conti, Paulo Nunes, and Luis Ducla Soares. "HEVC-based light field image coding with bi-predicted self-similarity compensation". In: *2016 IEEE International Conference on Multimedia & Expo Workshops (ICMEW)*. IEEE. 2016, pp. 1–4. DOI: 10.1109/icmew.2016.7574667.

[136] Fatma Hawary, Christine Guillemot, Dominique Thoreau, et al. "Scalable light field compression scheme using sparse reconstruction and restoration". In: *2017 IEEE International Conference on Image Processing (ICIP)*. IEEE. 2017, pp. 3250–3254. DOI: 10.1109/icip.2017.8296883.

[137] Xiaoran Jiang, Mikael Le Pendu, and Christine Guillemot. "Light field compression using depth image based view synthesis". In: *2017 IEEE International Conference on Multimedia & Expo Workshops (ICMEW)*. IEEE. 2017, pp. 19–24. DOI: 10.1109/icmew.2017.8026313.

[138] Weisi Lin and C.-C. Jay Kuo. "Perceptual visual quality metrics: A survey". In: *Journal of Visual Communication and Image Representation* 22.4 (May 2011), pp. 297–312. DOI: 10.1016/j.jvcir.2011.01.005.

[139] Emin Zerman, Giuseppe Valenzise, and Frederic Dufaux. "An extensive performance evaluation of full-reference HDR image quality metrics". In: *Quality and User Experience* 2.1 (Apr. 2017). DOI: 10.1007/s41233-017-0007-4.

[140] Vamsi Kiran Adhikarla, Marek Vinkler, Denis Sumin, et al. "Towards a quality metric for dense light fields". In: *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2017* (Apr. 25, 2017). DOI: 10.1109/CVPR.2017.396. arXiv: 1704.07576 [cs.CV].

[141] Quan Huynh-Thu and Mohammed Ghanbari. "Scope of validity of PSNR in image/video quality assessment". In: *Electronics letters* 44.13 (2008), pp. 800–801. DOI: 10.1049/el:20080522.

[142] Olivia Nemethova, Michal Ries, Markus Rupp, et al. "Quality Assessment for H.264 Coded Low-rate and Low-resolution Video Sequences". In: Jan. 2004, pp. 508–512. eprint: https://publik.tuwien.ac.at/files/pub-et_8787.pdf.

[143] Jirka Klaue, Berthold Rathke, and Adam Wolisz. "EvalVid – A Framework for Video Transmission and Quality Evaluation". In: *Computer Performance Evaluation. Modelling Techniques and Tools*. Springer Berlin Heidelberg, 2003, pp. 255–272. DOI: 10.1007/978-3-540-45232-4_16.

[144] ITU. *BT.500 - Methodologies for the subjective assessment of the quality of television images*. 10/19. International Telecommunications Union. Oct. 2019.

[145] James J Gibson. *The perception of the visual world.* Houghton Mifflin, 1950. DOI: 10.2307/1419017.

[146] Sandeep Singh Sengar and Susanta Mukhopadhyay. "Motion detection using block based bi-directional optical flow method". In: *Journal of Visual Communication and Image Representation* 49 (2017), pp. 89–103. DOI: 10.1016/j.jvcir.2017.08.007.

[147] Amir Akramin Shafie, Fadhlan Hafiz, MH Ali, et al. "Motion detection techniques using optical flow". In: *World Academy of Science, Engineering and Technology* 56 (2009), pp. 559–561. DOI: 10.5281/zenodo.1071466.

[148] Jens Klappstein, Tobi Vaudrey, Clemens Rabe, et al. "Moving object segmentation using optical flow and depth information". In: *Pacific-Rim Symposium on Image and Video Technology*. Springer. 2009, pp. 611–623. DOI: 10.1007/978-3-540-92957-4_53.

[149] Yi-Hsuan Tsai, Ming-Hsuan Yang, and Michael J Black. "Video segmentation via object flow". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 3899–3908. DOI: 10.1109/cvpr.2016.423.

[150] Xiaojing Song, Lakmal D Seneviratne, and Kaspar Althoefer. "A Kalman filter-integrated optical flow method for velocity sensing of mobile robots". In: *IEEE/ASME Transactions on Mechatronics* 16.3 (2010), pp. 551–563. DOI: 10.1109/tmech.2010.2046421.

[151] Javier Sánchez Pérez, Enric Meinhardt-Llopis, and Gabriele Facciolo. "TV-L1 optical flow estimation". In: *Image Processing On Line* 2013 (2013), pp. 137–150. DOI: 10.5201/ipol.2013.26.

[152] Christopher Zach, Thomas Pock, and Horst Bischof. "A duality based approach for realtime tv-l 1 optical flow". In: *Joint pattern recognition symposium*. Springer. 2007, pp. 214–223. DOI: 10.1007/978-3-540-74936-3_22.

[153] Manish Narwaria, Matthieu Perreira Da Silva, and Patrick Le Callet. "HDR-VQM: An objective quality measure for high dynamic range video". In: *Signal Processing: Image Communication* 35 (July 2015), pp. 46–60. DOI: 10.1016/j.image.2015.04.009.

[154] Louis Kerofsky, Rahul Vanam, and Yuriy Reznik. "Adapting objective video quality metrics to ambient lighting". In: *2015 Seventh International Workshop on Quality of Multimedia Experience (QoMEX)*. IEEE, May 2015. DOI: 10.1109/qomex.2015.7148135.

[155] Benjamin Meyer, Christian Lipski, Björn Scholz, et al. "Real-time free-viewpoint navigation from compressed multi-video recordings". In: *Proc. 3D Data Processing, Visualization and Transmission (3DPVT),(May 31, 2010)* (2010), pp. 1–6. eprint: https://graphics.tu-bs.de/upload/publications/meyer2010realtime.pdf.

[156] Tatsuro Mori, Keita Takahashi, and Toshiaki Fujii. "Real-Time Free-Viewpoint Image Synthesis System Using Time Varying Projection". In: *ITE Transactions on Media Technology and Applications* 2.4 (2014), pp. 370–377. DOI: 10.3169/mta.2.370.

[157] Karsten Mueller, Aljoscha Smolic, Kristina Dix, et al. "View synthesis for advanced 3D video systems". In: *EURASIP Journal on image and video processing* 2008 (2009), pp. 1–11. eprint: https://link.springer.com/content/pdf/10.1155/2008/438148.pdf.

[158] Patrick Ndjiki-Nya, Martin Koppel, Dimitar Doshkov, et al. "Depth image-based rendering with advanced texture synthesis for 3-D video". In: *IEEE Transactions on Multimedia* 13.3 (2011), pp. 453–465. DOI: 10.1109/tmm.2011.2128862.

[159] Jaesik Park, Hyeongwoo Kim, Yu-Wing Tai, et al. "High quality depth map upsampling for 3D-TOF cameras". In: *2011 International Conference on Computer Vision*. IEEE, Nov. 2011. DOI: 10.1109/iccv.2011.6126423.

[160] Ying He, Bin Liang, Yu Zou, et al. "Depth Errors Analysis and Correction for Time-of-Flight (ToF) Cameras". In: *Sensors* 17.1 (Jan. 2017), p. 92. DOI: 10.3390/s17010092.

[161] Krishna Rao Vijayanagar, Maziar Loghman, and Joohee Kim. "Refinement of depth maps generated by low-cost depth sensors". In: *2012 International SoC Design Conference (ISOCC)*. IEEE, Nov. 2012. DOI: 10.1109/isocc.2012.6407114.

[162] Shi Yan, Chenglei Wu, Lizhen Wang, et al. "DDRNet: Depth Map Denoising and Refinement for Consumer Depth Cameras Using Cascaded CNNs". In: *Computer Vision – ECCV 2018*. Springer International Publishing, 2018, pp. 155–171. DOI: 10.1007/978-3-030-01249-6_10.

[163] Dawid Mieloch and Adam Grzelka. "Segmentation-based Method of Increasing The Depth Maps Temporal Consistency". In: *International Journal of Electronics and Telecommunications*. Polish Academy of Sciences Committee of Electronics and Telecommunications, 2018. DOI: 10.24425/123521.

[164] Alvaro Collet, Ming Chuang, Pat Sweeney, et al. "High-quality streamable free-viewpoint video". In: *ACM Transactions on Graphics* 34.4 (July 2015), pp. 1–13. DOI: 10.1145/2766945.

[165] Joey de Vries. *Learn OpenGL*. Kendall & Welling, June 17, 2020. 522 pp. ISBN: 9090332561. URL: https://www.ebook.de/de/product/39272964/joey_de_vries_learn_opengl.html.

[166] Mark Harris. "GPGPU: General-purpose computation on GPUs". In: *SIGGRAPH 2005 GPGPU COURSE* (2005), pp. 1–51. URL: http://download.nvidia.com/developer/presentations/2005/GDC/OpenGL_Day/OpenGL_GPGPU.pdf.

[167] John Cheng, Max Grossman, and Ty McKercher. *Professional CUDA C Programming*. John Wiley & Sons Inc, Oct. 7, 2014. 528 pp. ISBN: 1118739329. URL: https://www.ebook.de/de/product/22064208/john_cheng_max_grossman_ty_mckercher_professional_cuda_c_programming.html.

[168] Benedict Gaster. *Heterogeneous computing with OpenCL*. Amsterdam Boston: Elsevier/Morgan Kaufmann, 2013. ISBN: 0128016493.

[169] Jianbin Fang, Ana Lucia Varbanescu, and Henk Sips. "A comprehensive performance comparison of CUDA and OpenCL". In: *2011 International Conference on Parallel Processing*. IEEE. 2011, pp. 216–225. DOI: 10.1109/icpp.2011.45.

[170] Oliver Grau, Graham A Thomas, A Hilton, et al. "A robust free-viewpoint video system for sport scenes". In: *2007 3DTV conference*. IEEE. 2007, pp. 1–4. DOI: 10.1109/3dtv.2007.4379384.

[171] Adrian Hilton, Jean-Yves Guillemaut, Joe Kilner, et al. "Free-viewpoint video for TV sport production". In: *Image and Geometry Processing for 3-D Cinematography*. Springer, 2010, pp. 77–106. DOI: 10.1007/978-3-642-12392-4_4.

[172] Victor Moya, Carlos Gonzalez, Jordi Roca, et al. "Shader performance analysis on a modern GPU architecture". In: *38th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'05)*. IEEE. 2005, 10–pp. DOI: https://doi.org/10.1109/MICRO.2005.30.

[173] Randima Fernando. "GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics". In: Pearson Higher Education, 2004. Chap. 28. ISBN: 0321228324.

[174] M. I. Shamos and D. Hoey. "Geometric intersection problems". In: *17th Annual Symposium on Foundations of Computer Science (sfcs 1976)*. 1976, pp. 208–215. DOI: 10.1109/sfcs.1976.16.

[175] Tianyi David Han and Tarek Abdelrahman. "Reducing branch divergence in GPU programs". In: Jan. 2011, p. 3. DOI: 10.1145/1964179.1964184.

[176] Sascha Willems. *OpenGL hardware database*. URL: https://opengl.gpuinfo.org/listcapabilities.php (visited on 08/18/2021).

[177] Daniel Scharstein and Richard Szeliski. "High-accuracy stereo depth maps using structured light". In: *2003 IEEE Computer Society Conference on Computer Vision and Pattern Recognition, 2003. Proceedings.* Vol. 1. IEEE. 2003, pp. I–I. DOI: 10.1109/CVPR.2003.1211354.

[178] Weiping Yang, Zhilong Zhang, Xinping Lu, et al. "A novel fast median filter algorithm without sorting". In: *Real-Time Image and Video Processing 2016*. Vol. 9897. International Society for Optics and Photonics. 2016, 98970A. DOI: 10.1117/12.2219847.

[179] Christian Riechert, Frederik Zilly, Marcus Müller, et al. "Real-time disparity estimation using line-wise hybrid recursive matching and cross-bilateral median upsampling". In: *Proceedings of the 21st International Conference on Pattern Recognition (ICPR2012)*. IEEE. 2012, pp. 3168–3171. eprint: https://projet.liris.cnrs.fr/imagine/pub/proceedings/ICPR-2012/media/files/0914.pdf.

[180] Peter Kauff, Nicole Brandenburg, Michael Karl, et al. "Fast hybrid block-and pixel-recursive disparity analysis for real-time applications in immersive tele-conference scenarios". In: University of West Bohemia, 2001. eprint: http://wscg.zcu.cz/wscg2001/Papers_2001/R132.pdf.

[181] UL benchmarks. *3DMark Technical Guide*. URL: https://s3.amazonaws.com/download-aws.futuremark.com/3dmark-technical-guide.pdf (visited on 08/18/2021).

[182] Aljoscha Smolic, Karsten Muller, Kristina Dix, et al. "Intermediate view interpolation based on multiview video plus depth for advanced 3D video systems". In: *2008 15th IEEE International Conference on Image Processing*. IEEE, Oct. 2008. DOI: 10.1109/icip.2008.4712288.

[183] Sergey Smirnov, Mihail Georgiev, and Atanas Gotchev. "Comparison of cost aggregation techniques for free-viewpoint image interpolation based on plane sweeping". In: *Ninth International Workshop on Video Processing and Quality Metrics for Consumer Electronics*. 2015. eprint: https://researchportal.tuni.fi/en/publications/comparison-of-cost-aggregation-techniques-for-free-viewpoint-imag.

[184] Sergi Pujades, Frederic Devernay, and Bastian Goldluecke. "Bayesian View Synthesis and Image-Based Rendering Principles". In: *2014 IEEE Conference on Computer Vision and Pattern Recognition*. IEEE, June 2014. DOI: 10.1109/cvpr.2014.499.

[185] A. Khatiullin, M. Erofeev, and D. Vatolin. "FAST OCCLUSION FILLING METHOD FOR MULTIVIEW VIDEO GENERATION". In: *2018 - 3DTV-Conference: The True Vision - Capture, Transmission and Display of 3D Video (3DTV-CON)*. IEEE, June 2018. DOI: 10.1109/3dtv.2018.8478562.

[186] Jun Chen, Ryosuke Watanabe, Keisuke Nonaka, et al. "Fast Free-viewpoint Video Synthesis Algorithm for Sports Scenes". In: *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, Nov. 2019. DOI: 10.1109/iros40897.2019.8967584.

[187] Lingjie Liu, Jiatao Gu, Kyaw Zaw Lin, et al. "Neural Sparse Voxel Fields". In: (July 22, 2020). arXiv: 2007.11571 [cs.CV].

[188] Nikolai Smolyanskiy, Alexey Kamenev, and Stan Birchfield. "On the Importance of Stereo for Accurate Depth Estimation: An Efficient Semi-Supervised Deep Neural Network Approach". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*. June 2018. DOI: 10.1109/cvprw.2018.00147.

[189] Hamid Laga, Laurent Valentin Jospin, F. Boussaid, et al. "A Survey on Deep Learning Techniques for Stereo-based Depth Estimation". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* (2020), pp. 1–1. DOI: 10.1109/tpami.2020.3032602.

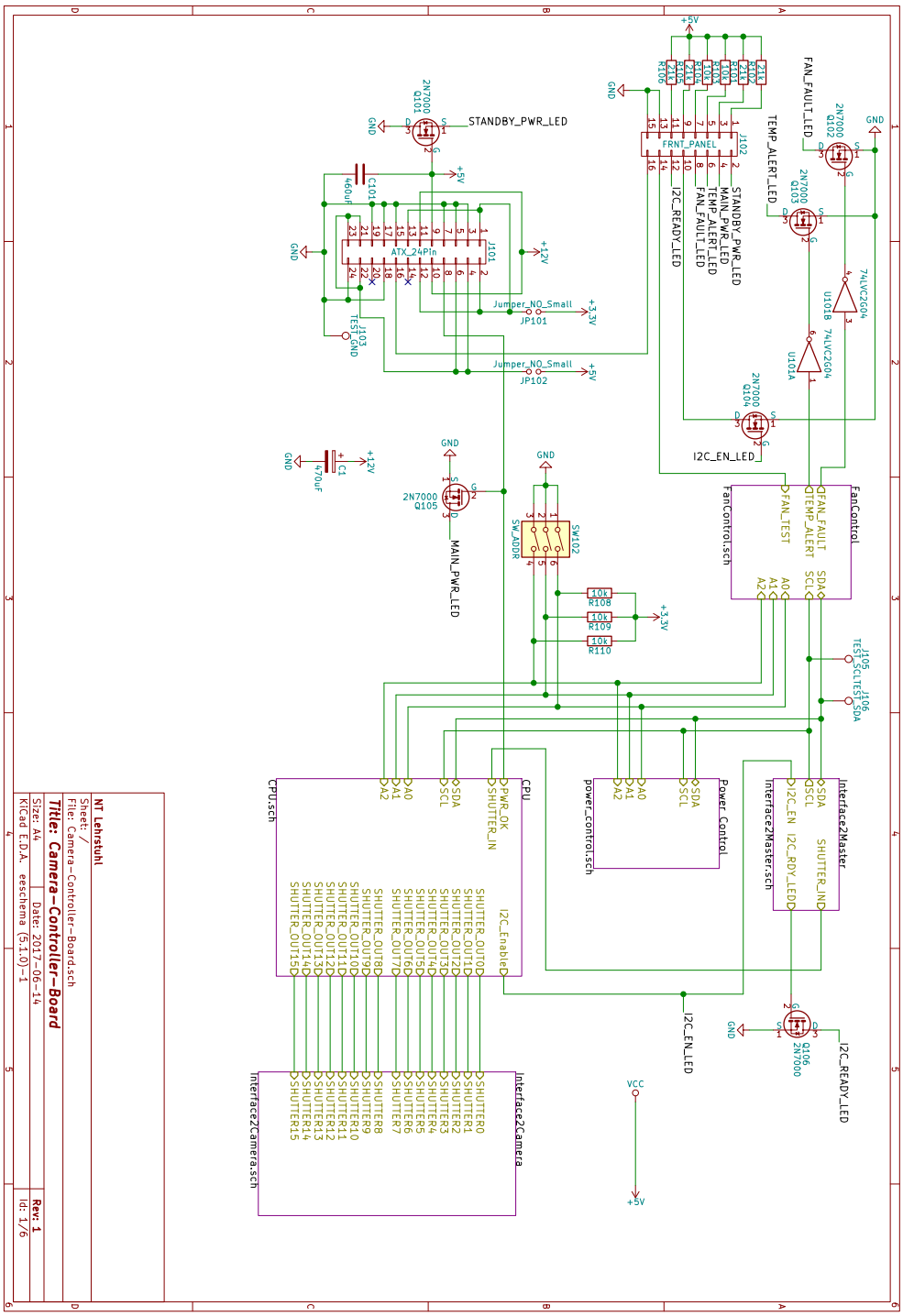
[190] John Flynn, Michael Broxton, Paul Debevec, et al. "Deepview: View synthesis with learned gradient descent". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2019, pp. 2367–2376. DOI: 10.1109/cvpr.2019.00247.

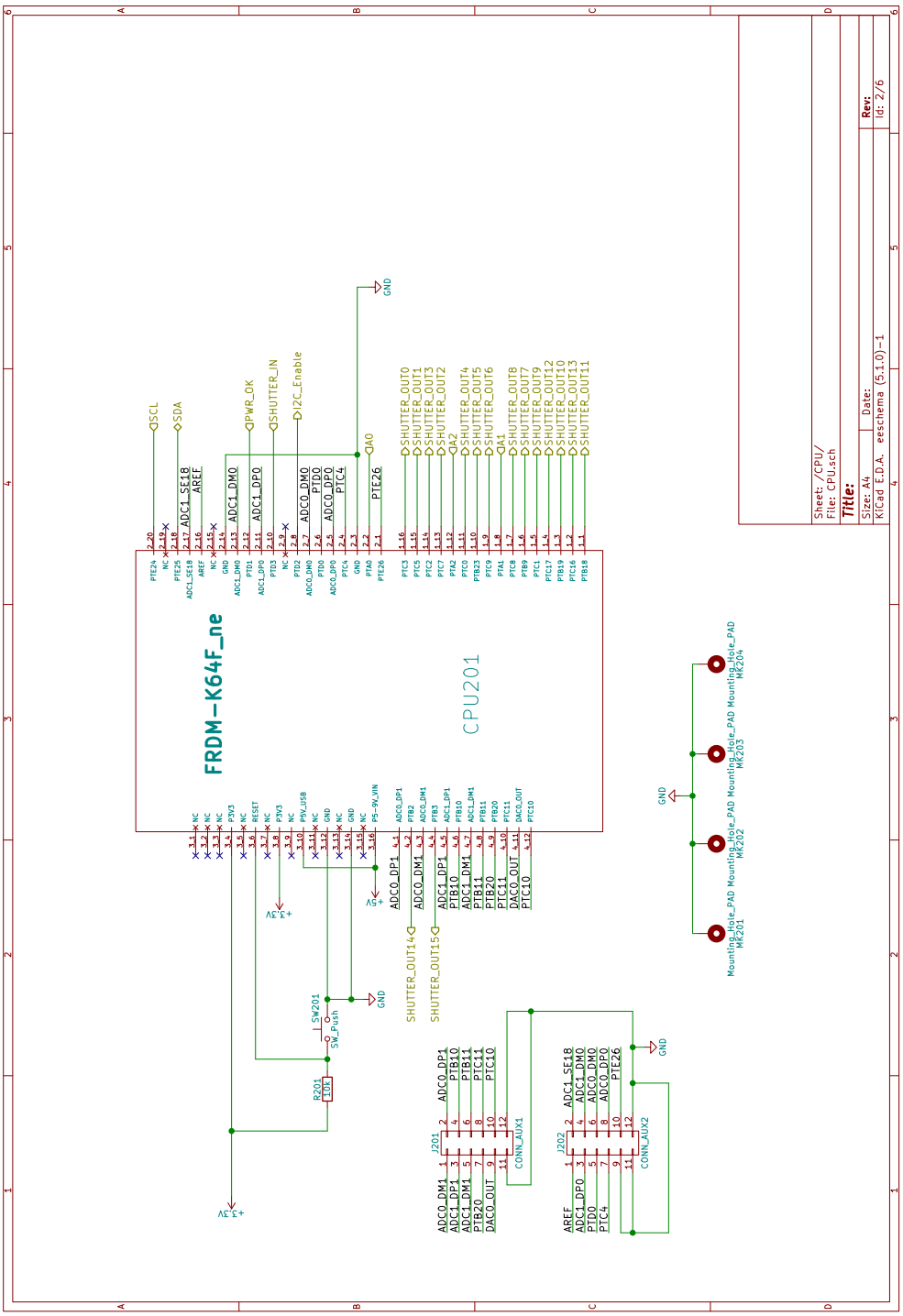
[191] Michael Broxton, John Flynn, Ryan Overbeck, et al. "Immersive light field video with a layered mesh representation". In: *ACM Transactions on Graphics (TOG)* 39.4 (2020), pp. 86–1. DOI: 10.1145/3386569.3392485.

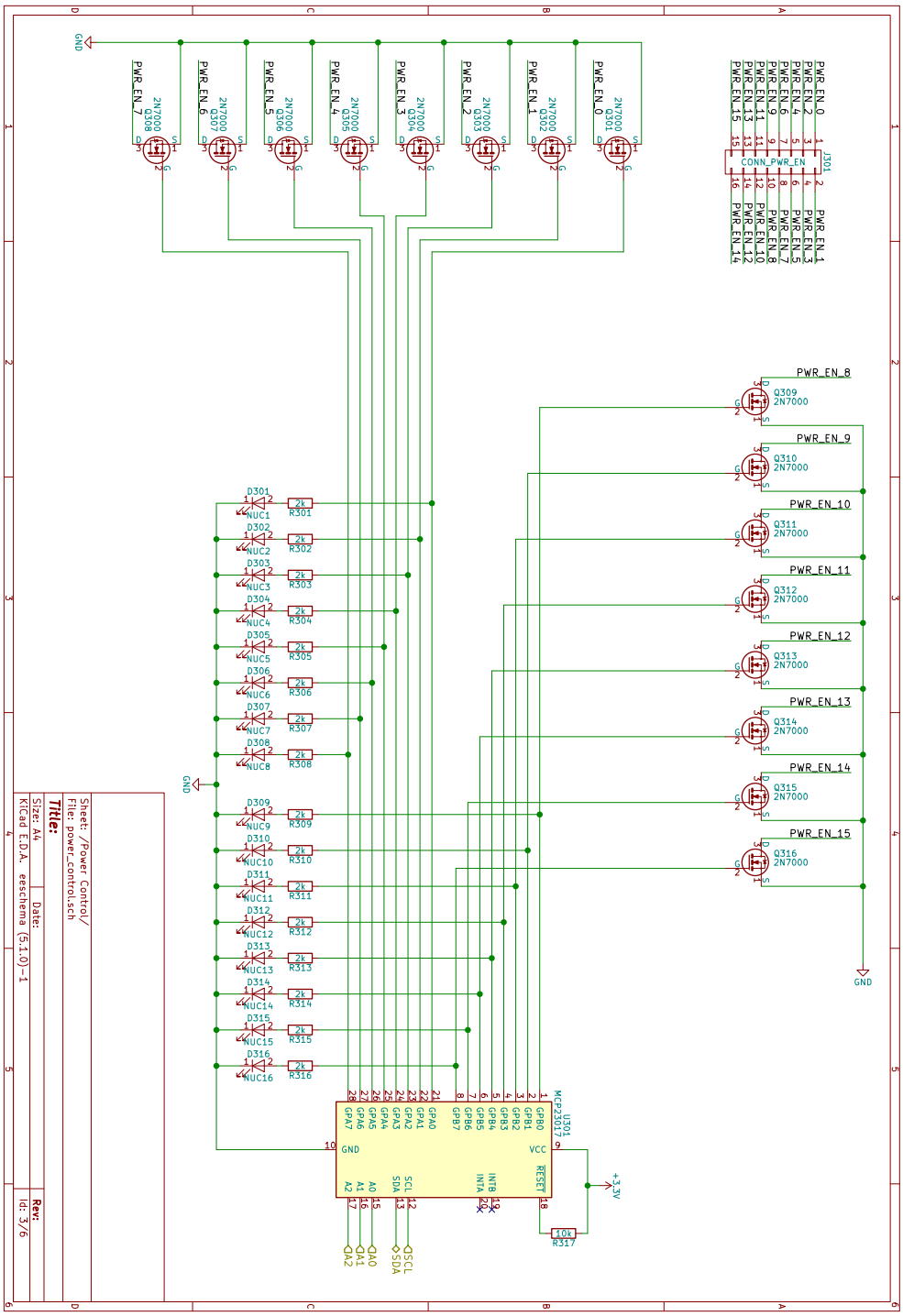
# A. Schematics

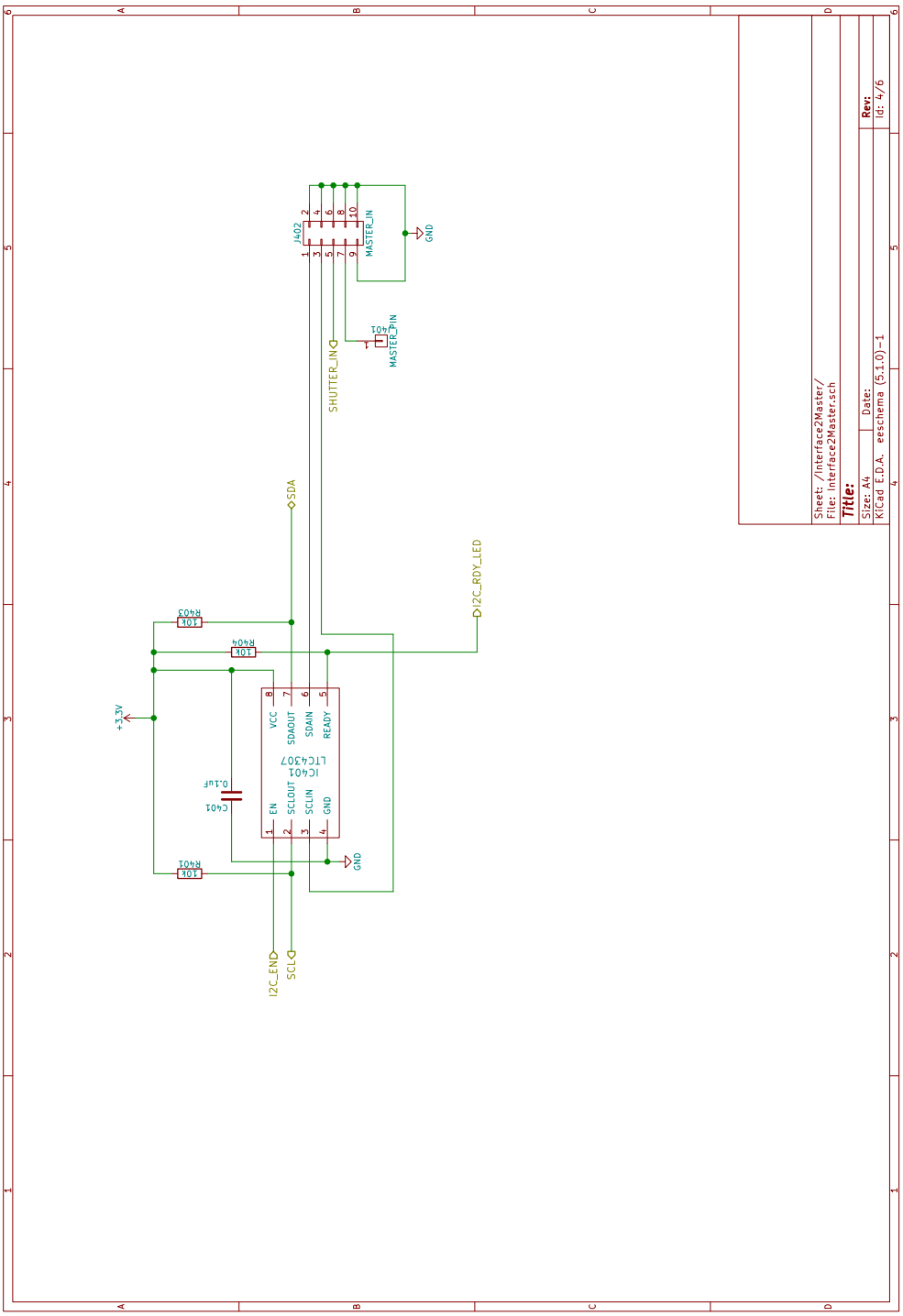
This part of the thesis provides the schematics for the custom PCBs currently in use in the camera array. The first page for each board provides an overview of the system with the connections between the major building blocks. On the remaining pages the details for those blocks are given. Each sheet's name is equivalent to the description of the block whose components it contains.

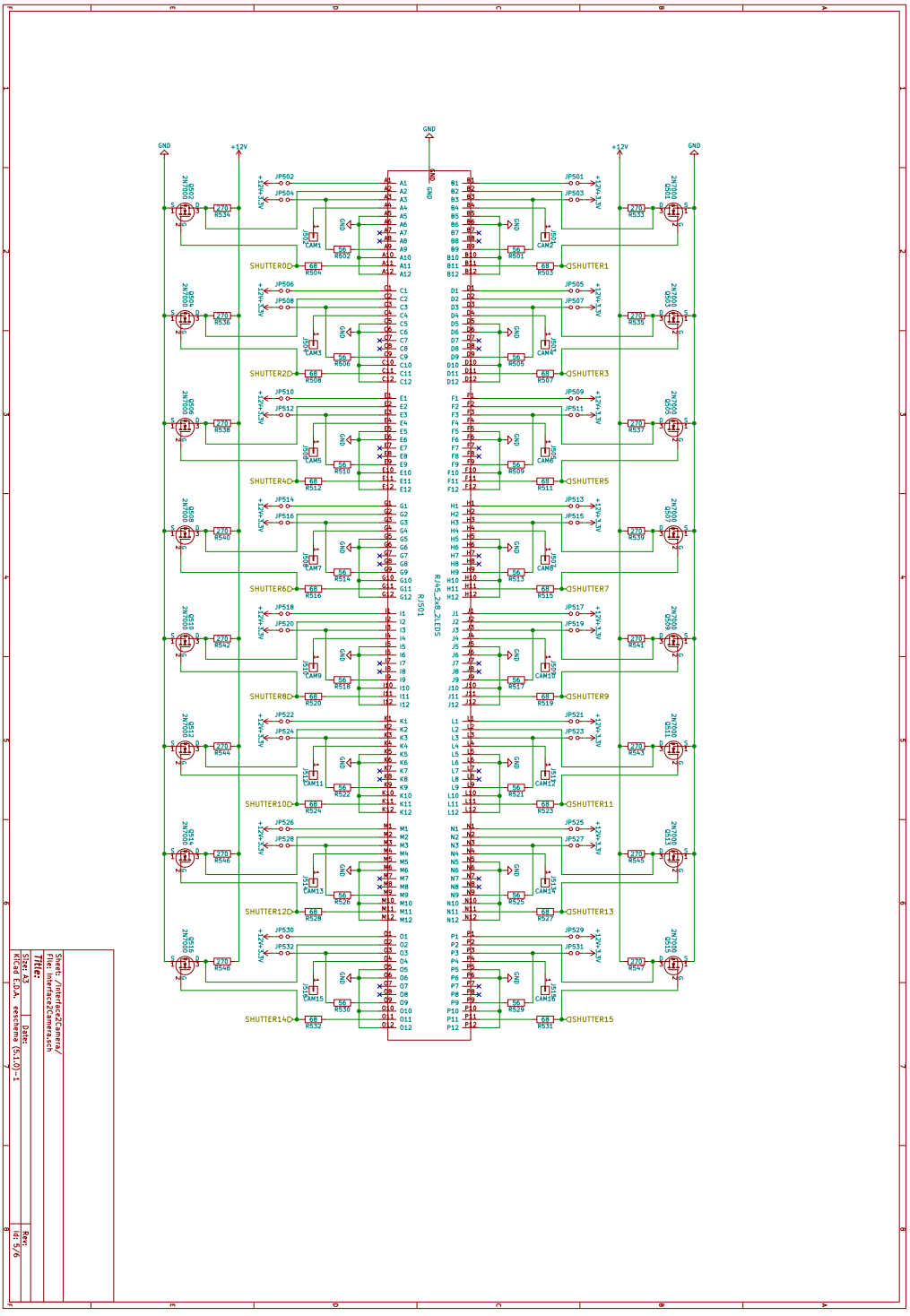
# A.1. Camera controller board

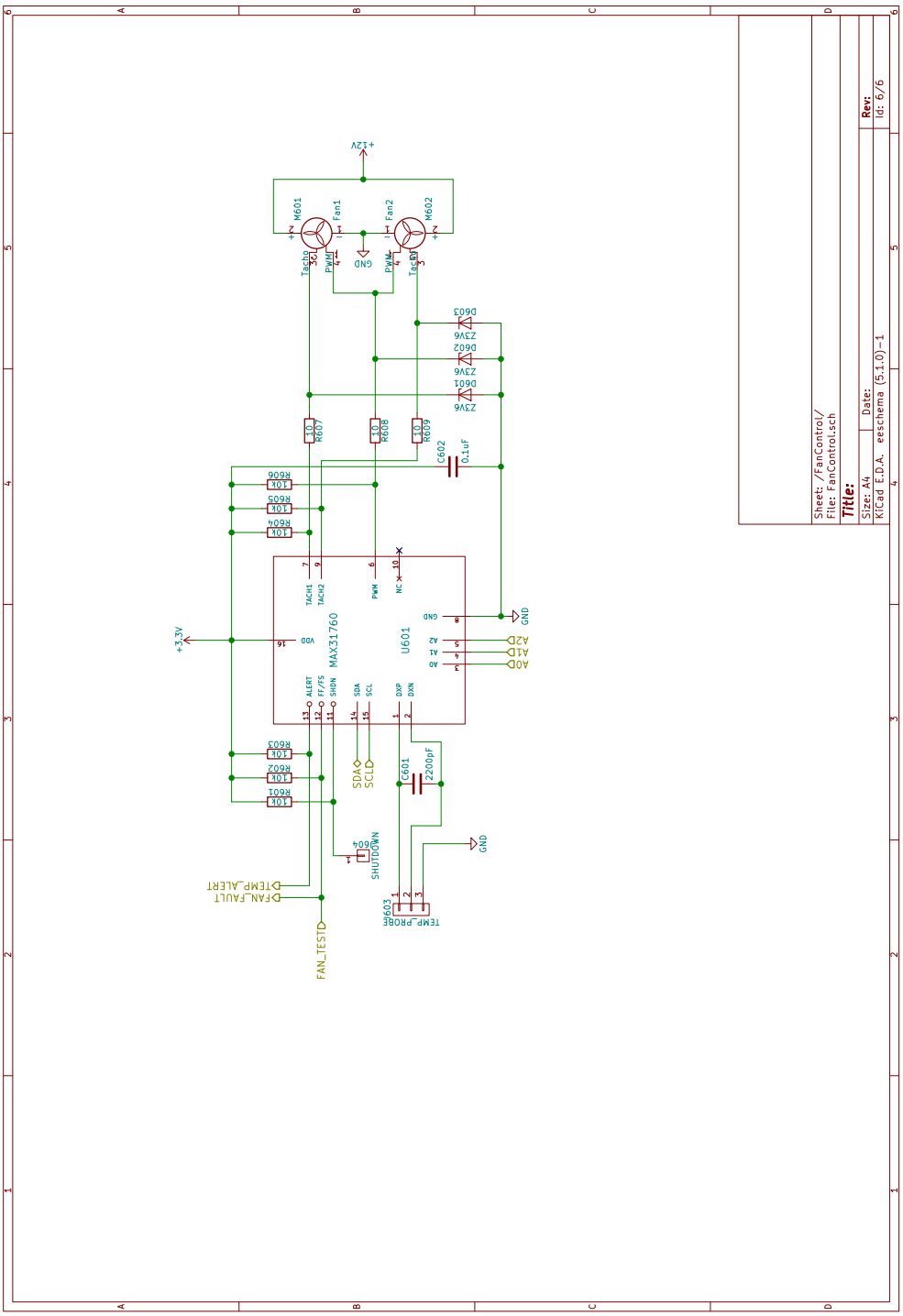



FRDM-K64F_ne
CPU201
SCL
SDA
PWR_OK
SHUTTER_IN
I2C_Enable
SHUTTER_OUT0
SHUTTER_OUT1
SHUTTER_OUT3
SHUTTER_OUT12
SHUTTER_OUT4
SHUTTER_OUT5
SHUTTER_OUT6
SHUTTER_OUT8
SHUTTER_OUT7
SHUTTER_OUT9
SHUTTER_OUT12
SHUTTER_OUT10
SHUTTER_OUT13
SHUTTER_OUT11
SHUTTER_OUT14
SHUTTER_OUT15
ADC0_DP1
ADC0_DM1
ADC1_DP1
ADC1_DM1
PTB10
PTB11
PTB20
PTC11
DAC0_OUT
PTC10
ADC1_SE18
AREF
ADC1_DM0
ADC0_DP0
PTD0
PTC4
PTE26
A0
A1
A2
GND
+3V3
SW201
SW_Push
R201
100
J201
J202
CONN_AUX1
CONN_AUX2
Mounting_Hole_PAD
MK201
MK202
MK203
MK204
Sheet: /CPU/
File: CPU.sch
Title:
Size: A4
Date:
Rev:
KiCad E.D.A. eeschema (5.1.0)-1
Id: 2/6

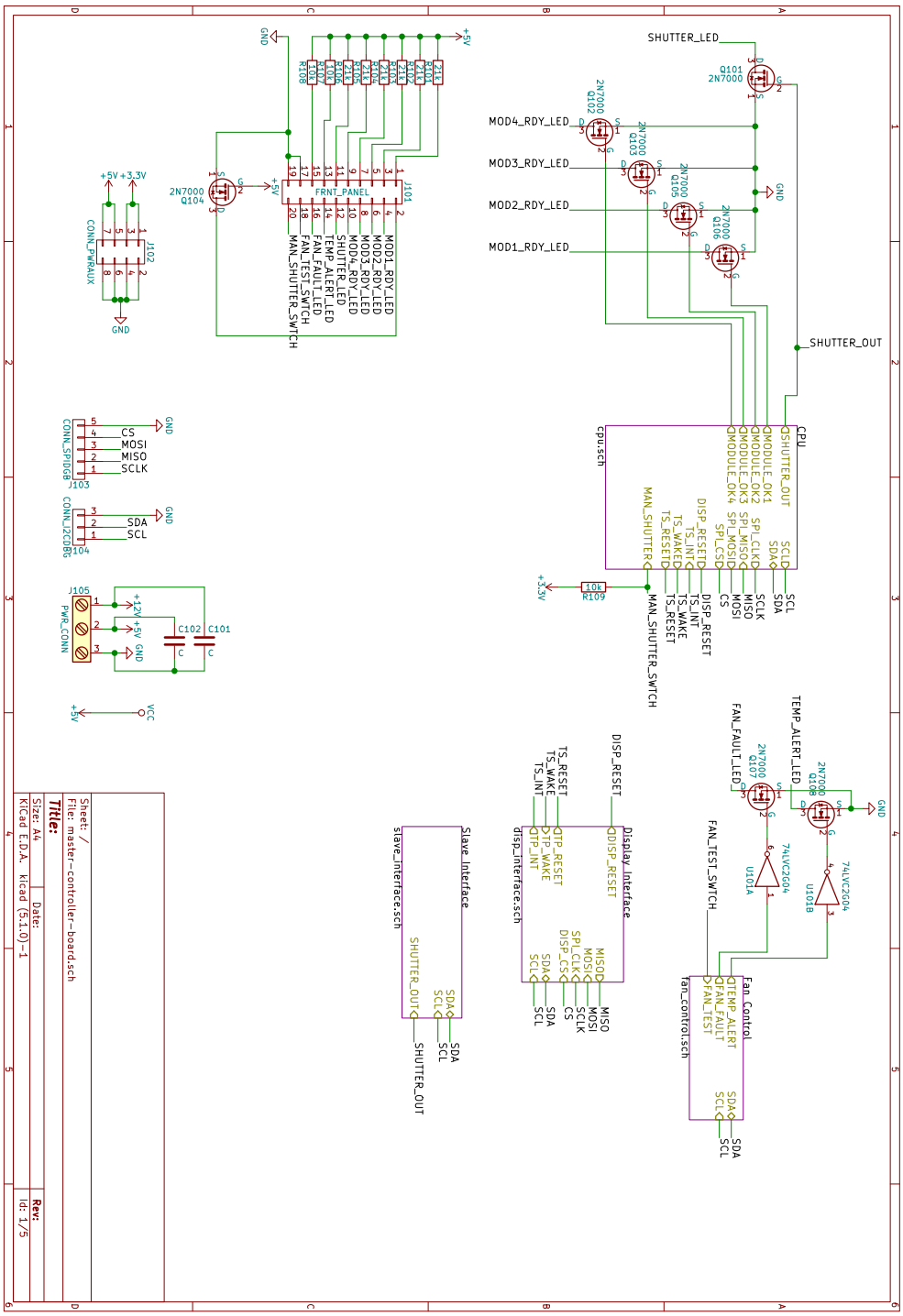

GND
PWR_EN_7
PWR_EN_6
PWR_EN_5
PWR_EN_4
PWR_EN_3
PWR_EN_2
PWR_EN_1
PWR_EN_0
2N7000
Q308
Q307
Q306
Q305
Q304
Q303
Q302
Q301
CONN_PWR_EN
J301
PWR_EN_8
PWR_EN_9
PWR_EN_10
PWR_EN_11
PWR_EN_12
PWR_EN_13
PWR_EN_14
PWR_EN_15
Q309
Q310
Q311
Q312
Q313
Q314
Q315
Q316
D301
D302
D303
D304
D305
D306
D307
D308
D309
D310
D311
D312
D313
D314
D315
D316
NUC1
NUC2
NUC3
NUC4
NUC5
NUC6
NUC7
NUC8
NUC9
NUC10
NUC11
NUC12
NUC13
NUC14
NUC15
NUC16
2k
R301
R302
R303
R304
R305
R306
R307
R308
R309
R310
R311
R312
R313
R314
R315
R316
U301
MCP23017
GPA7
GPA6
GPA5
GPA4
GPA3
GPA2
GPA1
GPA0
GPB7
GPB6
GPB5
GPB4
GPB3
GPB2
GPB1
GPB0
VCC
RESET
INTB
INTA
SCL
SDA
A0
A1
A2
+3.3V
10k
R317
Sheet: /Power Control/
File: power_control.sch
Title:
Size: A4
Date:
KiCad E.D.A. eeschema (5.1.0)-1
Rev:
Id: 3/6

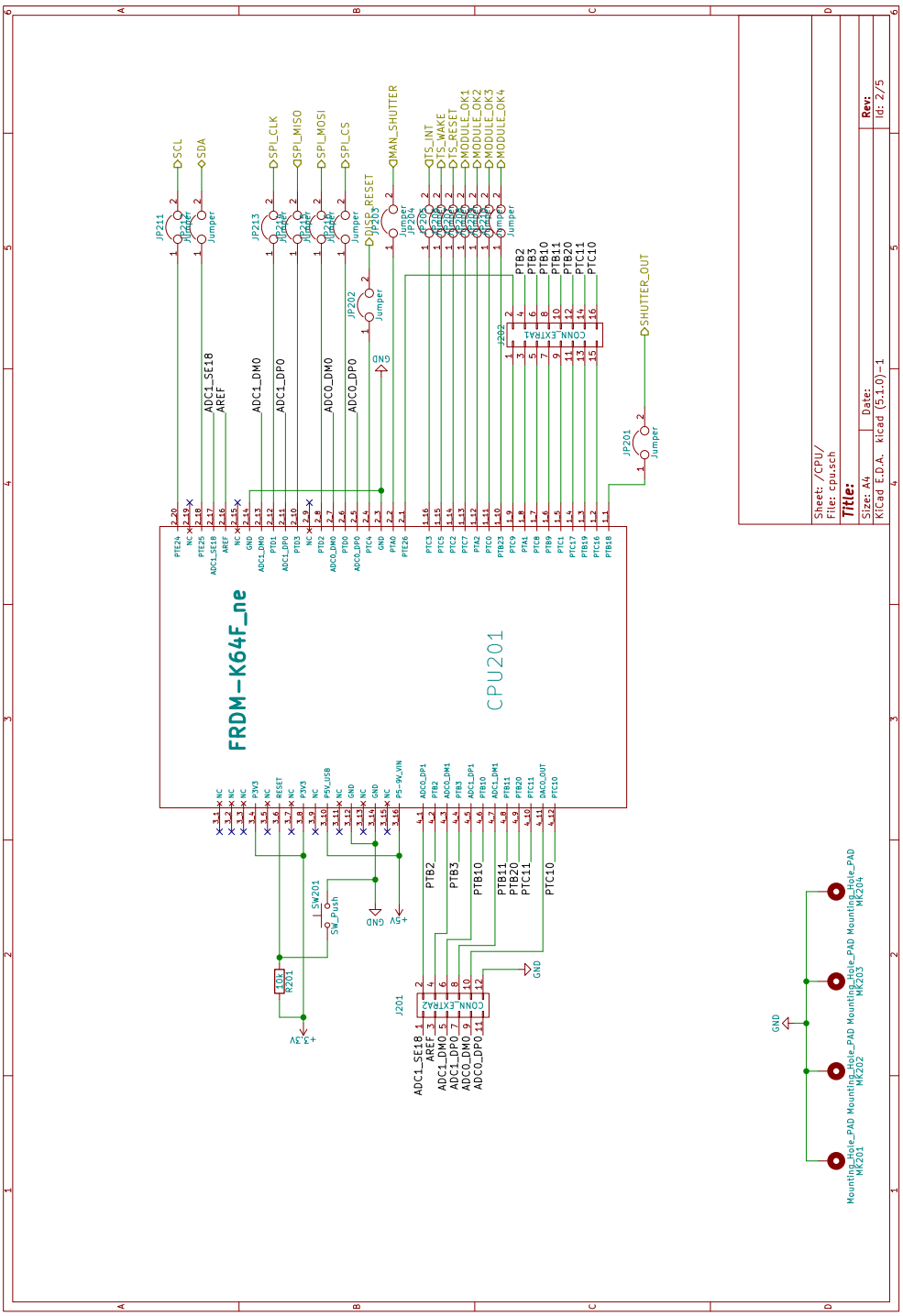

J402
MASTER_IN
GND
R401
MASTER_PIN
SHUTTER_IN
SDA
I2C_RDY_LED
R403
10K
R404
10K
+3.3V
IC401
LTC4307
VCC
SDAOUT
SDAIN
READY
EN
SCLOUT
SCLIN
GND
C401
0.1uF
R401
10K
GND
I2C_END
SCL
Sheet: /Interface2Master/
File: Interface2Master.sch
Title:
Size: A4
Date:
Rev:
KiCad E.D.A. eeschema (5.1.0)-1
Id: 4/6

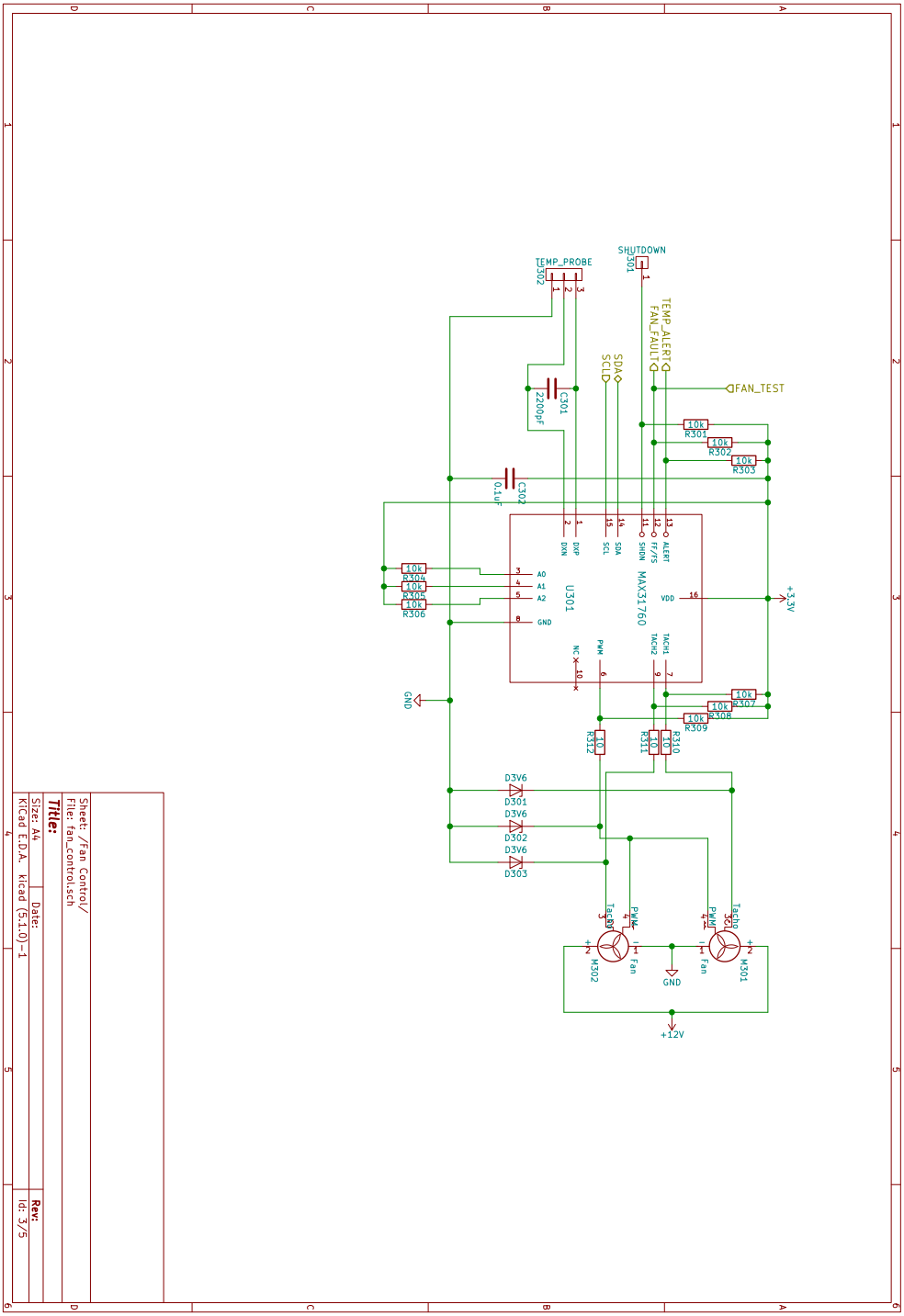

GND
+12V
RJ501
RJ45_2x6_2LEDS
SHUTTER0D
SHUTTER1
SHUTTER2D
SHUTTER3
SHUTTER4D
SHUTTER5
SHUTTER6D
SHUTTER7
SHUTTER8D
SHUTTER9
SHUTTER10D
SHUTTER11
SHUTTER12D
SHUTTER13
SHUTTER14D
SHUTTER15
Sheet: /Interface2Camera/
File: Interface2Camera.sch
Title:
Size: A3
Date:
KiCad E.D.A. eeschema (5.1.0)-1
Rev:
Id: 5/6

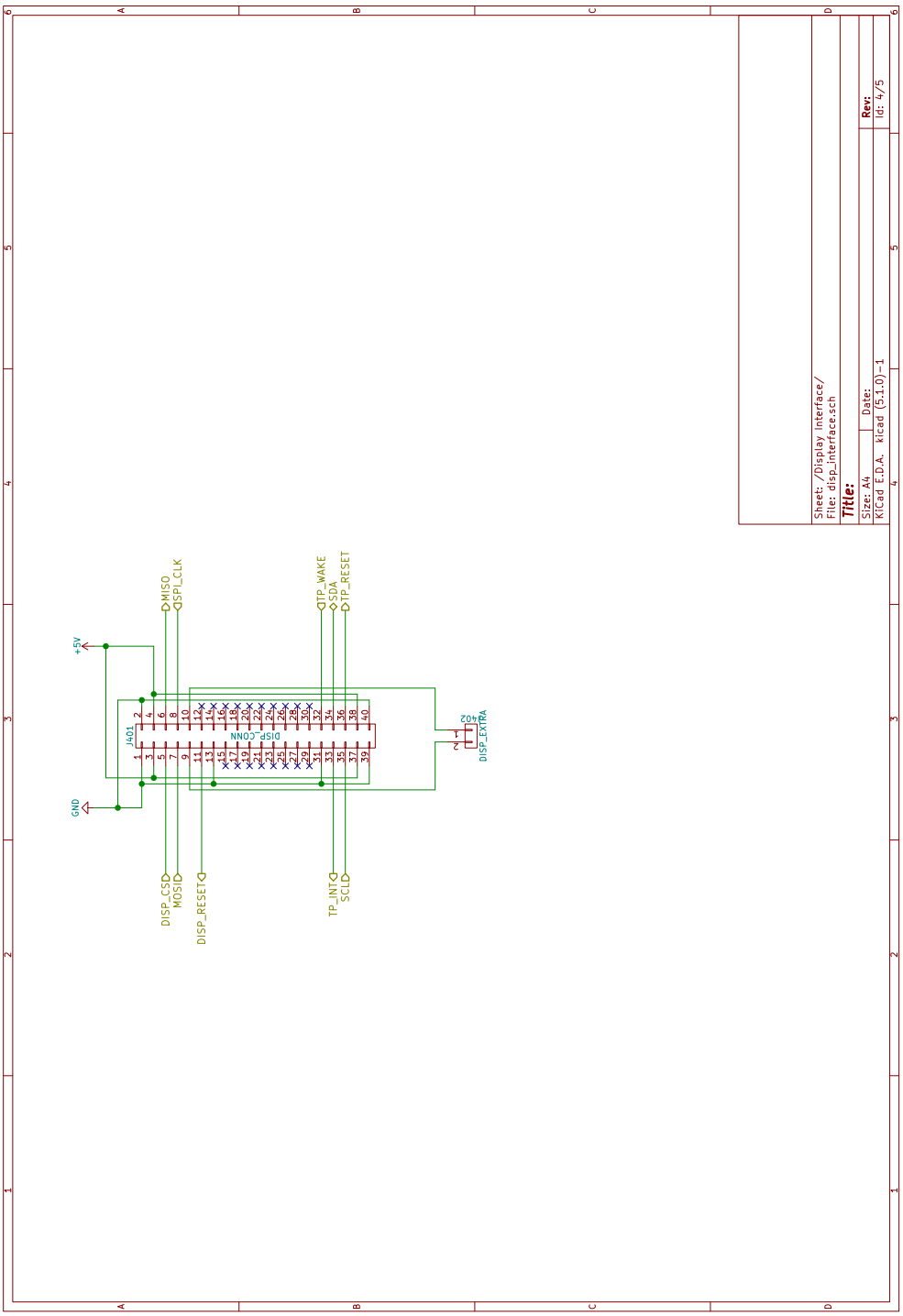

+12V
M601
Fan1
Fan2
M602
GND
PWM
Tach
D603
D602
D601
Z3V6
R607
R608
R609
10
C602
0.1uF
R606
R605
R604
10K
TACH1
TACH2
NC
MAX31760
U601
VDD
+3.3V
A2
A1
A0
ALERT
FF/FS
SHDN
SDA
SCL
DXP
DXN
R603
R602
R601
C601
200pF
SDA
SCL
Q604
SHUTDOWN
TEMP_PROBE
TEMP_ALERT
FAN_FAULT
FAN_TEST
Sheet: /FanControl/
File: FanControl.sch
Title:
Size: A4
Date:
Rev:
KiCad E.D.A. eeschema (5.1.0)-1
Id: 6/6

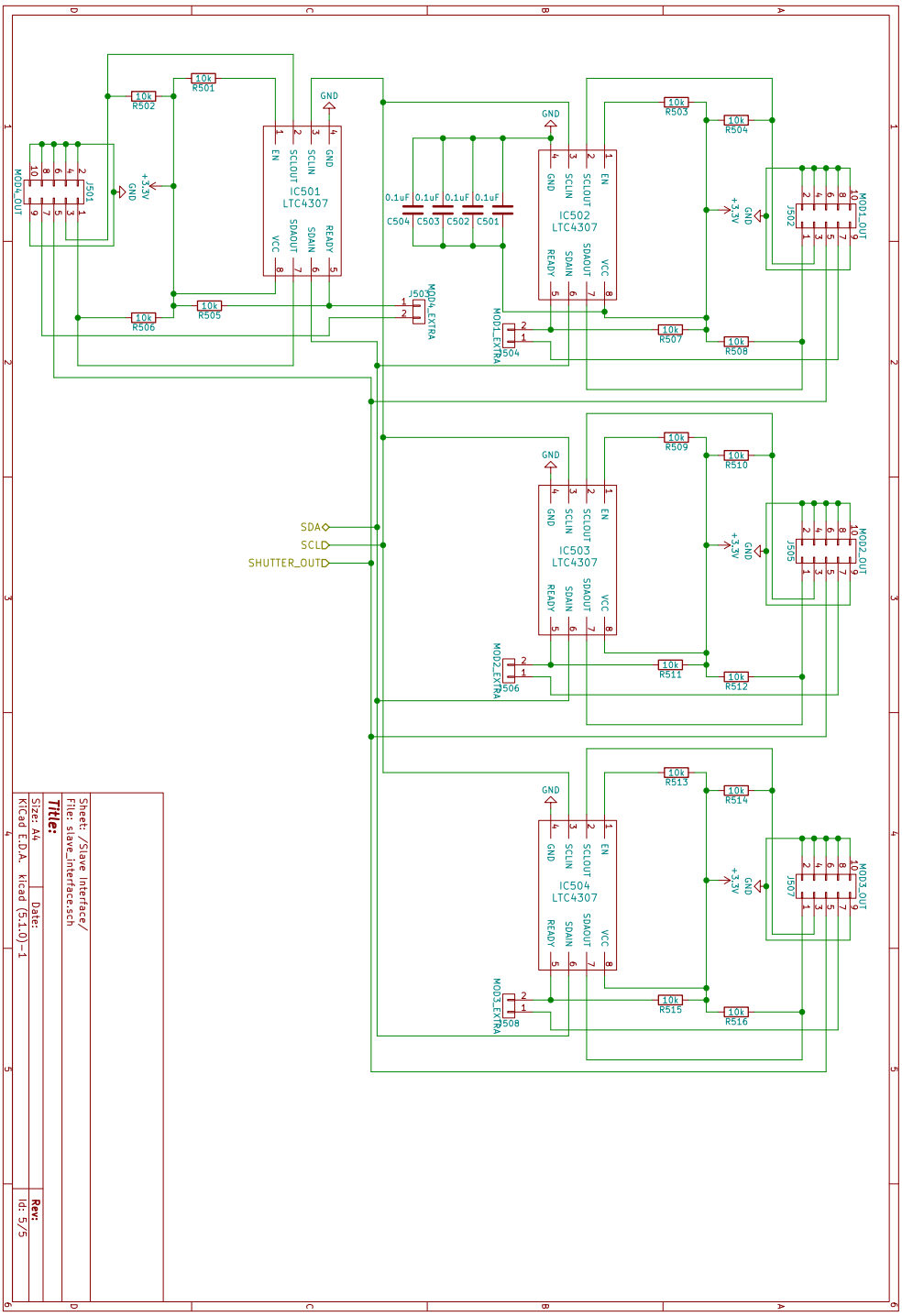
## A.2. Master controller board




FRDM-K64F_ne
CPU201
Sheet: /CPU/
File: cpu.sch
Title:
Size: A4
Date:
Rev:
KiCad E.D.A. kicad (5.1.0)-1
id: 2/5
SCL
SDA
SPI_CLK
SPI_MISO
SPI_MOSI
SPI_CS
MAN_SHUTTER
TS_INT
TS_WAKE
TS_RESET
MODULE_OK1
MODULE_OK2
MODULE_OK3
MODULE_OK4
SHUTTER_OUT
ADC1_SE18
AREF
ADC1_DM0
ADC1_DP0
ADC0_DM0
ADC0_DP0
PTB2
PTB3
PTB10
PTB11
PTB20
PTC11
PTC10
GND
+3V3
+5V
SW201
SW_Push
R201
10k
J201
J202
CONN_EXTRA1
JP201
JP202
JP203
JP204
JP205
JP206
JP207
JP208
JP209
JP210
JP211
JP212
JP213
Jumper
Mounting_Hole_PAD
MK201
MK202
MK203
MK204


TEMP_PROBE
J302
SHUTDOWN
J301
TEMP_ALERT
FAN_FAULT
SDA
SCLD
FAN_TEST
2200pF
C301
0.1uF
C302
10k
R301
10k
R302
10k
R303
10k
R304
10k
R305
10k
R306
U301
MAX31760
DXP
DXN
SCL
SDA
SHDN
FF/FS
ALERT
A0
A1
A2
GND
VDD
+3.3V
NC
PWM
TACH2
TACH1
10k
R307
10k
R308
10k
R309
10
R312
10
R311
10
R310
D3V6
D301
D3V6
D302
D3V6
D303
Tach
PWM
Fan
M302
M301
GND
+12V
Sheet: /Fan Control/
File: fan_control.sch
Title:
Size: A4
Date:
KiCad E.D.A. kicad (5.1.0)-1
Rev:
Id: 3/5


Sheet: /Display Interface/
File: disp_interface.sch
Title:
Size: A4
Date:
Rev:
KiCad E.D.A. kicad (5.1.0)-1
Id: 4/5
+5V
GND
MISO
SPI_CLK
TP_WAKE
SDA
TP_RESET
DISP_CS
MOSI
DISP_RESET
TP_INT
SCLD
DISP_EXTRA
J401
DISP_CONN
R402


IC501
LTC4307
IC502
LTC4307
IC503
LTC4307
IC504
LTC4307
SDA
SCLD
SHUTTER_OUT
MOD1_OUT
MOD2_OUT
MOD3_OUT
MOD4_OUT
Sheet: /Slave Interface/
File: slave_interface.sch
Title:
Size: A4
Date:
KiCad E.D.A. kicad (5.1.0)-1
Rev:
Id: 5/5