Virtual view-rendering for 3D-video communication systems

Author: Tobias Lange
Matriculation number: 2517119
Course of studies: Computer- und Kommunikationstechnik
Supervisor: Prof. Dr.-Ing. Thorsten Herfet
Adviser: M. Sc. Goran Petrovic
Reviewers: Prof. Dr.-Ing. Thorsten Herfet
Prof. Dr.-Ing. Philipp Slusallek

April 30, 2013
Eidesstattliche Erklärung
Ich erkläre hiermit an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement in Lieu of an Oath
I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis.

Saarbrücken, den ___________________________ (Datum / Date) ___________________________ (Tobias Lange)

Einverständniserklärung
Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Declaration of Consent
I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken, den ___________________________ (Datum / Date) ___________________________ (Tobias Lange)
Abstract

We define the requirements for a good virtual-view rendering algorithm as follows: allows complex camera viewing scenarios, provides good image quality, adapts dynamically to an unknown scene complexity, incorporates depth information and supports real-time rendering on commodity platforms.

Our analysis of the state-of-the-art shows that we currently lack a rendering algorithm that simultaneously satisfies all these requirements. Our work is based on caview, a rendering algorithm implemented by Cha Zhang, presented at Eurographics 2004 [16]. Our contribution to the implementation of this algorithm is as follows:

We analyzed the algorithm with respect to the requirements above and defined and implemented necessary changes:

- Changed the program structure to support a wider range of input formats.
- Accelerated the rendering of intermediate frames by incorporating a modern shader-based technique.
- Implemented the depth estimation in OpenCL to make use of the highly-parallel structure of modern GPUs.

Measurements of the resulting speed increase show that we can achieve real-time performance. Our improved implementation can render intermediate views within the required time budget of 40ms. In addition, we discuss the impact of several algorithm parameters on the total rendering time and illustrate a significant improvement of rendering speed compared to the original implementation of caview in all parameter configurations (up to 50 frames per second).
Acknowledgements

First and foremost, I would like to express special thanks to Prof. Dr.-Ing. Thorsten Herfet for giving me the opportunity to write this master’s thesis at his chair, the Telecommunications Lab.

I would also like to thank M.Sc. Goran Petrovic, who always provided useful background material and for his professional advise while writing the thesis.

In particular, I would like to thank the team of the Telecommunications Lab who readily provided information at any time. Furthermore, I would like to thank all people who supported me in writing this thesis, especially Jochen Britz, Oliver Dejon and my family.
## Contents

1 Introduction ........................................ 1
   1.1 3D- and multi-view video communication systems . . . . . . . . . . 1
   1.2 Estimating scene-depth information . . . . . . . . . . . . . . . . . 4
   1.3 Related work ........................................ 5
   1.4 Requirements ....................................... 7
   1.5 Contributions ..................................... 7

2 Background .......................................... 9
   2.1 Modern GPUs ......................................... 9
   2.2 General computations using GPUs ....................... 12
   2.3 OpenCL ............................................. 15

3 Analysis of scene-complexity adaptive view rendering ........ 17
   3.1 Baseline algorithm caview .......................... 17
      3.1.1 Analysis ...................................... 21

4 Proposed fast implementation based on OpenCL ............ 24
   4.1 Changes to program structure ......................... 24
   4.2 Shading ............................................. 25
      4.2.1 Pre-computation of projection coordinates .... 25
      4.2.2 Shader structure .................................. 26
      4.2.3 Shader-friendly parameter transfer ............... 28
      4.2.4 Avoiding dynamic memory allocation ............. 29
   4.3 Parallel computation using OpenCL .................... 29
      4.3.1 Optimized data placement ......................... 30
      4.3.2 Memory alignment and pointer size ............... 30
      4.3.3 Handling dynamically-growing triangle lists .... 30
      4.3.4 Efficient data transfer between OpenCL and OpenGL shader 31
      4.3.5 Image filtering in OpenCL ......................... 32
      4.3.6 Memory differences with OpenCL on GPU and CPU . 32
      4.3.7 Precision differences with OpenCL on GPU and CPU 33
      4.3.8 Inconsistent depth values on the GPU ............ 34

5 Experiments and Results ................................ 35
   5.1 Rendering performance .............................. 35
   5.2 Performance impact of algorithm parameters ........... 37
      5.2.1 Initial grid size ................................ 37
      5.2.2 Subdivision ..................................... 38
Chapter 1

Introduction

1.1 3D- and multi-view video communication systems

3D-video systems allow a user to perceive depth in a viewed scene and to display the scene from multiple viewpoints. Stereoscopic video is a special case of 3D-video viewing, where the scene depth is rendered with the help of a specialized display device – head-mounted glasses or an auto-stereoscopic display. The application domains for 3D-video include entertainment, communications, medicine, security, visualization, and education. In each of these domains, 3D-video brings specific advantages compared to conventional, 2D-video. In general, these advantages include a heightened sense of immersion and realism of the presentation.

Multi-view video (MVV) is an emerging video-content type that introduces multiple views into a single video stream. It is therefore closely related to existing stereoscopic 3D-video systems that are limited to only two views. The multiple views correspond to different viewpoints of a natural scene, such that the scene can be displayed from different viewpoints (perspectives) or directions (angles).

Although in general, there are $N$ views embedded in every stream, the number of usable perspectives depend on the type of the output device. With a standard 2D display, each of the $N$ streams can be displayed separately and therefore allows $N$ different perspectives. The multiple-perspective viewing can be interactive, when the user selects a new viewpoint, or automatic, when his

Figure 1.1: A 3D video communication system.
movements are continuously tracked and the displayed content is adjusted. A 3D display can only show \( N - 1 \) different views, since it needs two neighboring views to display an accurate three-dimensional image.

In contrast to these widely spread display devices which only show the number of views that is directly proportional to the number of views provided in the stream, novel multi-view displays are emerging that are able to display an arbitrary number of views \( M \), limited only by the technology and the processing power of the display. The reason for that lies in the fact that if multiple neighboring views are available, it is possible to calculate novel views that lie between the positions of the original capturing cameras. The process of calculating novel views will be explained more detailed later in this thesis.

In this thesis, we focus on the applications of 3D and MVV in communications. Figure 1.1 shows a simplified view of a 3D-video communication system. It consists of the following blocks.

**Capturing.** Capturing multi-view video is more complicated than capturing normal 2D-video. Each view requires its own camera or at least its own image sensor. To simultaneously capture two views, portable cameras with multiple sensors and lens systems are currently available on the market. Capturing systems for more than two views are not commercially available, but can be built using multiple cameras in combination with a system to maintain the position and viewing direction relative to the other cameras. Figure 1.2 gives examples of such a capturing system. In addition, it is important that the capturing process of all cameras is perfectly synchronized. If this requirement is not met, especially for scenes with a lot of movement, the end result may contain artifacts or cause viewer discomfort because humans are not used to have a time delay between the images that they see with their two eyes.

**View rendering.** A viewpoint defines the position and viewing direction of a camera. Often its parameters are given as an extrinsic and an intrinsic matrix. The extrinsic parameters are contained in a rotation matrix \( R \) and a translation vector \( T \) describing the cameras position and orientation in space. Sometimes they are also combined into a single projection matrix \( P \) which is defined as the product of \( R \) and \( T \). The intrinsic matrix \( A \) consists of the internal camera parameters like focal length, aspect ratio, etc.

\[
A = \begin{bmatrix}
\alpha_x & \gamma & u_0 \\
0 & \alpha_y & v_0 \\
0 & 0 & 1
\end{bmatrix}
\]  

(1.1)

MVV systems make a distinction between viewpoints that are 'real' (corresponding to the camera positions) and viewpoints that are 'virtual' (not corresponding to any of the available cameras). The distinct new capability of MVV systems is that novel, virtual views of the scene, corresponding to virtual camera positions, can be rendered and displayed to the user. The virtual views are needed in a number of scenarios. First, novel display devices that support a fixed number of simultaneous viewpoints can display a combination of original and virtual views (e.g., positioned between the available views). Second, in user-interactive
systems (that either track the position of one or multiple users in front of the display or allow to change the perspective freely during playback), there is a requirement to create novel views in real-time so as to avoid unnatural delays between the change of the users’ position and the resulting view.

**Scene-geometry modeling.** Scene modeling for 3D-video refers to a range of approaches for reconstructing a geometric model of a real-world scene. In the context of view rendering, an important type of geometric information about the scene is the scene depth. The depth information is very important for the generation of novel views because it is the only way to differentiate the background from the foreground pixels in a view. This information is needed during rendering, as it allows a visibility resolution when switching from one scene-viewpoint to another. If this information is unavailable, depending on the scene complexity, rendering artifacts may occur as shown in Figure 1.3. In this figure, we render a virtual view while representing the scene geometry as a single plane (left) and compare it with the rendering using reconstructed depth information (right). The resulting blurring artifact in the left image is clearly visible. This artifact occurs as in the absence of accurate depth information pixels from different depth planes are erroneously combined when rendering the view.
1.2 Estimating scene-depth information

Scene-depth information represented as a depth map, conveys the distance between the camera plane and the nearest surface point in the scene for each pixel in a video frame. It is commonly represented as a gray-scale image where pixel brightness encodes the distance to the scene. Since most of the commodity cameras only capture color images without any scene-geometry information, adding the depth information is a problem that has to be solved.

One possibility to do that would be to record the depth of the scene while capturing the scene. In the robotics area, the so called LIDAR (LIght Detection And Ranging) system is used to get a depth image of the environment of robots or autonomous cars. A similar system is present in the Kinect sensor from the Microsoft Xbox. It creates a depth image to support user and gesture recognition. Although these systems automatically capture the depth information in real-time, they have a couple of flaws that prevent it from being directly used in view rendering. In general, the so-obtained depth maps cannot be used for intermediate view rendering without extensive post-processing. First, they typically have a resolution that is much smaller than the resolution of the corresponding color images. Second, they contain a lot of noise, which can cause problems in the rendering process due to depth inaccuracies. Finally, a depth map is inherently tied to a specific viewpoint that estimating the scene depth consistent with multiple viewpoints is hard.

Another way of acquiring the depth information is by calculating it from the available color images. Most of the depth-estimation algorithms follow the same principle:

**Depth-from-stereo** algorithms work as follows. First they calculate the projection from all points in one image to the corresponding points in another available image. For each pixel in one image, a similarity measure is applied to determine the pixel position in the other image, belonging to the the same scene point. The pixel position with the highest similarity value is then assumed to be the correct correspondence for this pixel in the source image.

**Correlation** approach is applied as follows. First, the projection from all
points in one image to the corresponding points in all other images is calculated, assuming all possible depth values. Then for each point associated with a certain depth, a similarity measure is applied to determine whether all those points belong to the same scene depth in every image. The depth with the highest similarity value is then assumed to be the correct depth for this pixel in the source image. Two of the representative techniques include cross-correlation and plane sweep. The \textit{cross-correlation} approach is one of the most precise ones as it checks every possible depth value. To do this, first the line on which a projected point travels when its assumed depth value changes, is determined. For rectified images, this simply becomes a strictly horizontal search problem. After that, a cross-correlation between the source image area and the projected area with changing depth is computed and the highest peak determines the correct depth \cite{5}. This can lead to errors if the texture along the projected line shows some repetition as then, the result shows multiple peaks from which only one is correct. Due to its high complexity, this approach is mostly used in offline stereo reconstruction algorithms.

\textit{Plane sweep} is a closely related approach. Instead of checking every possible value, the search space of plane sweep is quantized with a fixed number of depth planes. The number of planes is then fitted to the actual scene geometry. Only the depth values associated with these planes are then compared against a chosen similarity measure and the plane with the highest similarity value is taken as the correct depth \cite{2}. Due to the limited number of depth checks and the possibility to include many different similarity measures, this approach is much faster than cross-correlation and is often used when real-time performance is required.

All of the techniques described above may provide high resolution depth maps with very high accuracy in controlled settings. As for the depth-map resolution, there are even techniques available that provide depth-maps with sub-pixel accuracy. Further, these algorithms can provide a depth map for each camera in the stream and even make the depth representation consistent over different cameras. These properties make the depth-estimation approach attractive compared to the capturing approach. However, these techniques have a high complexity. This makes them hard to use in real-time systems without modifications. For example, in the case of plane sweep to obtain real-time performance, the depth accuracy or resolution can be reduced, or adapted to the scene complexity such that only a number of possible depths in scene is considered. Finally, the accuracy of depth estimation in general scenes is still an open problem \cite{10}.

1.3 Related work

A number of approaches in the field of computer graphics address the reconstruction and rendering of real-world 3D scene representations \cite{12}. The main focus of this work is geometry modeling to reconstruct an accurate volumetric model of a 3D scene such that the resulting model can be rendered using standard graphics techniques. The motivation is to automatically reconstruct such
a model from the available camera images instead of constructing it by hand.

Most commonly, a 3D scene with a single human figure is considered. Aiming at real-time rendering, Matusik et al. [7] recover an incomplete, view-dependent visual-hull model of an object offline, distributing the processing over several PCs. Rendering results show a good quality, while the hulls are extracted with as few as four cameras. However, rendering is complex and runs in parallel over four PCs to obtain real-time properties. Würmlin et al. [15] also take an approach based on visual-hull reconstruction, but use dynamic-point samples as the rendering primitive instead of triangular meshes. Experiments show convincing results for rendering of human actors captured with eight cameras. Carranza et al. [1] assume that a generic human-body model in the form of a triangular mesh is available, and focus on capturing the object’s motion in the scene by tracking this model throughout the video sequence. For rendering, view- and time-dependent textures captured by the cameras are mapped onto the model; however, the motion capture is performed offline. It is worth pointing out that the best-performing algorithms for object-geometry reconstruction (in terms of accuracy of geometry reconstruction) have long computation times, often amounting to tens of minutes per video frame [10]. The ability of these techniques to scale with the scene complexity (e.g., the number of objects in the scene) is questionable.

Instead of reconstructing volumetric 3D-scene models, a number of approaches only reconstruct local geometric models in the form of depth or disparity maps. Zitnick et al. show that high-quality rendering of virtual views is possible with accurate depth maps and stereo matting [18]. The depth maps and accurate matting information are estimated offline in this work. Taguchi et al. demonstrate real-time rendering of light fields enhanced with coarse geometry information (a small number of depth planes) [14]. However, an extension of this algorithm to scenes with a more complex depth structure is not considered in the paper. Do et al. [4] present a rendering algorithm that combines view blending with inpainting and achieves a real-time performance using GPU acceleration. However, the performance of the algorithm when rendering with inaccurate or missing depth information is not discussed in the paper. Most recently, several approaches focused on improving the speed and accuracy of depth estimation by implementing depth-from-stereo in FPGA hardware [17] or by jointly designing the color and depth imaging sensors [9]. Although these approaches significantly improve the rendering quality compared to state-of-the-art sensors (e.g., Kinect), it is unclear how to achieve a similar performance on commodity hardware or how to adapt the accuracy of depth reconstruction to the desired rendering quality. Most similar and most directly relevant to our work is the algorithm in [16]. This algorithm achieves good rendering quality and does not require accurate scene-geometry information. In addition, the algorithm allows to optimize the trade-off between the rendering quality and time budget (e.g., for scenes of different complexities). However, as this algorithm was designed as a part of a larger project to build a multi-camera capturing system, achieving real-time algorithm performance as well as specifying the exact quality/complexity trade-offs was out of the scope of the work. As our algorithm implementation builds directly on this algorithm, we will present it in more detail in the later sections of this report.
1.4 Requirements

Based on our considerations in earlier sections, we define the requirements for a virtual-view rendering algorithm as follows:

- should support a range of viewing scenarios, including translational viewpoint changes (horizontal or vertical) as well as advanced scenarios such as complex motion trajectories (mixed rotational and translational or zooming out of the scene),
- quality of interpolated (virtual) views should be comparable to that of the original views,
- should dynamically adapt the complexity of the rendering algorithm to the complexity of the underlying scene (of unknown geometry),
- should incorporate scene-depth information to improve the rendering quality, if this information is reconstructed at sufficient accuracy,
- should support a real-time rendering of virtual views on commodity platforms.

1.5 Contributions

Based on our discussion in Section 1.3, to the best of our knowledge, we currently lack a rendering algorithm that simultaneously satisfies these requirements. Our work is based on caview, a rendering algorithm implemented by Cha Zhang, presented at Eurographics 2004 [16]. Our contribution to the implementation of that algorithm is as follows:

First, we analyzed the algorithm with respect to the requirements presented above and defined necessary changes. In the sequel we modified caview to fulfill these requirements:

- We changed the program structure to support a wider range of input formats and to allow a quick change of the computation device later on.
- We accelerated the rendering of intermediate frames by substituting the old software-based rendering with a modern shader-based technique. This introduced some new requirements to the depth estimation algorithm, which we solved using a special shader structure.
- We analyzed the depth estimation part of caview, identified speed issues and addressed them by accelerating the calculations using OpenCL. We modified the code to be able to calculate the depth of each vertex in parallel. This allows us to efficiently use the highly-parallel architecture of modern GPUs.

Finally, we measured the resulting speed increase using the image sequences provided with the original code and show that real-time performance is now possible in general. With our rendering implementation, we can render intermediate views within the required time budget of 40ms. In addition, we discuss
the impact of several algorithm parameters on the total rendering time and illustrate a significant improvement of rendering speed compared to the original implementation of caview in all parameter configurations (up to 50 frames per second).
Chapter 2

Background

2.1 Modern GPUs

Modern GPUs are the logical result of hardware evolution over the last two decades. The first generation GPUs were rather simple devices with a fixed 3D-rendering pipeline with a small set of changeable parameters. Over time the steps of the pipeline became more and more flexible and allowed programmers to determine what happens in each stage in more detail with so-called shading programs. The number of computational units per stage was fixed and therefore, often created performance bottlenecks in a certain stage if its program was more complex than the ones in the other stages. In 2006, the internal architecture changed from modeling the whole rendering pipeline in hardware to using generic computational units that could be used for every stage of the pipeline [6, 8]. This unified shader architecture allows the hardware to dynamically distribute its computational performance over the stages. By shifting the computing power to different stages if needed, the hardware can adapt to the complexity of the shaders in each stage of the pipeline. Until today, the abilities of the computational units have increased further and further and are now able to perform calculations with double precision floating point numbers, access memory at much higher speeds and the rendering pipeline is fully programmable.

Although the overall architecture of GPUs has experienced extensive changes over the years and the capabilities of modern devices are close to those of a modern CPU, the graphics pipeline itself has not changed much.

The input of the rendering pipeline is a set of geometric primitives. In most cases triangles are used, but also rectangles or other polygons would be possible. They are formed from input vertices and the final primitive is assembled on the way through the pipeline. To make those primitives visible on the screen, they have to go through several stages. Figure 2.1 shows the stages of the current OpenGL pipeline. The blue boxes stand for programmable parts, while the yellow stages provide a fixed functionality. All parts with a dashed outline are optional. The intended functions for the different stages are described below.

**Vertex shader** The purpose of the vertex shader is to transform the coordi-
Figure 2.1: Rendering pipeline of OpenGL.
nates of the given vertex from its original coordinate system into the screen space. Older GPUs fully defined this stage with a 4x4 matrix which was multiplied with the coordinates of the vertex. Such matrices allow simple transformations like translation, rotation, scaling or a combination of those operations. Today this stage is defined by a vertex shader program which is able to perform arbitrary operations on the vertex. The only limit is the fact that the calculations for one vertex have to be independent from all other vertices. For each input vertex, the shader has to output exactly one output vertex. It is possible to calculate additional data for the later stages depending on the vertex’s properties.

**Primitive assembly** In this stage the primitives are assembled from the vertices coming from the vertex stage. It is not programmable and therefore, not interesting for this thesis.

**Tessellation shaders** Tessellation shaders are the newest part of the OpenGL pipeline. They are able to subdivide the primitives without interaction with the CPU. To use them it is important that all data for the new vertices in the mesh can be computed in the shader stages. Since this is not the case for our algorithm, tessellation shaders are not discussed further in this thesis.

**Geometry shader** This optional shader stage is a fairly new addition to the rendering pipeline. It receives all vertices that are associated with a geometric primitive at once. It is the only stage in the pipeline that can make decisions that are not just based on the properties of a single vertex, but on the properties of all vertices belonging to one primitive.

Additionally, it is able to destroy or create primitives based on its decisions. An instance of this shader receives one primitive represented by its vertices and outputs 0 to N primitives. The maximum value of N is limited by number of properties needed per vertex and the amount of data, the hardware is able to transfer between the stages. If necessary, this stage is also able to change the type of primitive. For example input rectangles can be transformed into triangles for the following stages.

**Clipping** The clipping stage divides partially visible primitives into their visible and invisible parts. Invisible in this stage does not mean that it is occluded by some other primitive, but only that is lies outside the area that is visible in the final frame. This process is known as clipping and reduces the workload for the following stages which normally are more complex than the ones before.

**Rasterization** This is another fixed stage in the pipeline. It maps portions of primitives to pixels on the screen. For each pixel it determines which primitives are visible in it and generates a fragment for each of these primitives. If more than one primitive is visible, their colors are combined after the next stage.

**Fragment shader** In the fragment shader stage the color for each fragment from the previous step is determined. The values it gets have the same structure like those of a vertex, but they are normally interpolated between the vertices of the primitive, depending on the position of the fragment on the primitive.
Often this stage is used to combine the colors of multiple textures by reading the color values at certain positions and adding them up.

**Composition** In this stage, the color values determined by the fragment stage for every pixel are combined into the final color of the pixel in the resulting image.

The calculations of most stages can be computed independently for each input element. This has led to the usage of parallel computation hardware in GPUs. In the beginning, there were only few parallel units available for each stage. But the advances in microprocessor technology have increased their number over the years. Figure 2.2 shows the structure of a NVIDIA GTX 680. Each green square represents a computational core. They are grouped in so-called SMX units, which are able to execute programs independently. The computing power of modern GPUs comes from it number of computational units. Although their clock frequency is significantly lower than that of a CPU (980Mhz vs. 3.2Ghz) and their cores have a much simpler design, the number of cores is a couple of orders higher (4 cores vs. 1344 cores). GPUs are also able to access their memory much faster than the CPU can access the RAM. On good hardware CPUs can read or write about 6GB per second from or to the RAM. A modern GPU can read or write more than 140GB per second. If their potential is properly used, GPUs can develop a computational power which is multiple times higher than what the fastest CPUs have to offer. This huge processing power combined with the improved programmability made GPUs more and more interesting for computational tasks that are not connected to graphics.

### 2.2 General computations using GPUs

As soon as it was possible to program the stages in the graphics pipeline, people tried to use the parallel architecture of GPUs to accelerate algorithms. In the beginning, it was important to map the different parts of an algorithm onto the stages of the graphics pipeline. The fixed number of units per stage made it complicated to use the GPUs efficiently and the output of the resulting values was only possible using textures.

The introduction of unified shader units in 2006 opened a lot of new opportunities for programmers. Without the fixed structure in the hardware itself, it became possible to use the whole computational power as needed. As the manufacturers became aware of this new use for their GPUs, they started to provide new programming frameworks for computational tasks. With those, it is possible to use the shader units without thinking about the structure of the graphics pipeline and they offered the possibility to transfer data directly to or from the GPU without using textures to transport them. The different frameworks were all limited to the graphics hardware of a certain manufacturer, but therefore, they were able to support the latest features of the hardware. Apart from the different command set and differences in the memory models they basically work the same.

The functions that are executed on the GPU are called kernels. Their param-
Figure 2.2: Internal structure of a NVIDIA GTX 680 [3].
eters can be simple constant values and/or buffers. Buffers are big chunks of memory that reside in the GPU memory and can be transferred to or from the CPU memory. They are the main way of communicating with the graphics card. To work efficiently, their data should be transferred in the fewest possible parts, because every transfer operation requires some synchronization and the connection to the GPU is optimized to transfer large chunks of data. This concept has not changed much since the beginnings of general computation on GPUs. How often a kernel is executed is defined in the CPU code. The total number of threads can be defined as a simple number \( N \) or a vector with up to three dimensions. Since the threads can be uniquely identified using a three-dimensional vector, it is easy to identify the area of data that the thread has to work on. The total number of threads is divided into work-groups of equal size. Those groups often share a special area of memory that can only be accessed by members of this work-group. It is guaranteed that all members of a work-group are executed at the same time. The maximum possible size of a work-group depends on the hardware and the amount of memory a single thread needs and is therefore fixed for each program. This means the total number of threads has to be a multiple of the work-group size. Often this leads to a couple of idle threads since the total number of threads has to be padded to become a multiple of the work-group size.

The commands that can be used inside a kernel depend on the capabilities of the hardware it is supposed to run on. Most mathematical operations are available for scalar as well as vector values. For best performance, it is recommended to use the vector functions, because the scalar calculations are performed on padded vectors and are therefore wasting computing power. Today kernels can use nearly all functions from standard C including arrays, pointers and control structures for code branching. Even custom memory structures and subroutines can be defined and used. Although control structures are available they should be avoided. The reason for that lies in the architecture of the GPUs. Their cores are not as independent as CPU cores and are not capable of individual branching. All threads that are executed in parallel have to execute the same code branch. If a thread chooses a different branch, the whole work-group has to execute again to evaluate the second branch. This means that every code branch can cut the computational performance in half. The GPU in Figure 2.2 has 8 units that can compute different programs or branches, but the 192 cores in each group have to compute the same branch of a program.

In contrast to CPU programs, kernels cannot allocate new memory during runtime. Therefore, the size of all fields and arrays has to be known before compilation and cannot be changed afterwards. Often this means that a maximum amount of memory has to be reserved, which may limit the number of threads per work-group because the amount of private memory per work-group can be very limited. As kernels are compiled during run-time for the currently active hardware, there are ways to optimize the memory requirements before compiling.
2.3 OpenCL

The early frameworks were only compatible with the hardware of a single manufacturer. The basic capabilities of all GPUs are mostly the same because they have to fulfill the requirements of common graphics frameworks like OpenGL or DirectX in consumer products. Therefore, the next logical step was to introduce computation frameworks that are able to work with GPUs from all manufacturers. For this thesis we focused on OpenCL [13] because it is based on an open standard and can be used with all modern GPUs as well as multi-core CPUs. The support for it depends on drivers from the vendors, but all major GPU manufacturers provide drivers with OpenCL support for all operating system.

Figure 2.3 shows the memory model of OpenCL. It is divided into the host space and the context space. The host memory is the RAM of the machine that controls the OpenCL device. The context contains multiple memory spaces which are mapped to the memory of the device that runs the calculations. On a GPU the global and constant memory resembles the whole graphics memory that can be accessed by the GPU. It is the biggest, but also the slowest memory available in OpenCL. In addition, each work-group has its own local memory. Often such memory has a physical counterpart in the GPU hardware which is considerably faster than the global memory. If this is not the case, this memory is mapped to the global memory. The content of the local memory is shared by all threads in one work-group. It is important to know that the content of this memory is only persistent during one run of a kernel. After the run has finished, the content is undefined. Finally, the private memory is an individual memory for each work-item or thread. Normally it is not larger than a couple of kilobytes, but it can be accessed with the highest speeds.

Since OpenCL has to work with different architectures and devices with different internal command sets, the performance may vary from device to device. If an OpenCL command has no direct counter-part in the internal command set of the device, its functionality has to be emulated using existing command. This can cause severe overheads and can limit the performance. For example on GPUs it is possible to sample textures using linear interpolation with performance loss, but if the code is run on a CPU which does not have special circuits for it, the performance drops severely compared to nearest pixel filtering. Although such considerations may complicate the coding, we use OpenCL in this thesis due to its flexibility regarding the computation hardware.

Most modern GPUs support some kind of parallel computing utilizing the highly parallel structure of those devices. Figure 2.2 shows the internal structure of a state-of-the-art graphics card. Each of the green boxes stands for a core that can be used for computations at the same time.

Our baseline algorithm has a structure that lends itself well to a parallel implementation. Several steps of the algorithm involve computations that are performed for each vertex independently and without data dependencies. In particular, this holds for Step 2, Step 3 and Step 4 of the algorithm, where neighboring camera selection, depth estimation and weight calculation, respectively, are repeated for each vertex in the mesh. These calculations are mostly
based on floating point numbers and only a small number of control structures like if-cases. Modern GPU architecture is very suitable for the implementation of our algorithm, since GPUs are designed for the type of repetitive computations inherent in our algorithm. Since those computations are repeated several hundred times for every frame, we can assume that the algorithm will run significantly faster on a GPU. We therefore believe that our baseline algorithm shows a great potential for parallelization and expect to improve the rendering speed with a GPU-accelerated implementation.
Chapter 3

Analysis of
scene-complexity adaptive
view rendering

3.1 Baseline algorithm caview

To synthesize the required novel views, this algorithm uses a 2.5D mesh i.e. a
2D mesh where each vertex has a depth value associated with it. To estimate
the depth values from the available color images, caview uses an online depth
recovery process. This estimation is implemented inside the rendering pipeline.
Importantly, the mesh and depth estimation are automatically adapting to the
scene complexity by subdividing the mesh in critical places.

Algorithm 3.1 shows how the rendering pipeline of the baseline algorithm works.
Details about each step are described below.

```
Clear frame buffer;
Create coarse initial mesh;
foreach vertex do
    determine closest views;
end
foreach vertex do
    estimate depth value;
    calculate rendering weight;
end
while mesh is too coarse do
    subdivide triangles if necessary;
end
foreach triangle do
    render to frame buffer;
end
```

Algorithm 3.1: Pseudocode for the rendering pipeline of caview.
Step 1: Create coarse mesh.
In the first step, a coarse initial 2D mesh is created. The vertices in this mesh are placed so that they have a distance of $D$ pixels in the resulting frame. This distance $D$ can be set as a configuration parameter for each scene. In general, one attempts to minimize the number of resulting vertices, thus reducing the total number of vertices to render and the overall computation time of the rendering algorithm. On the other hand, in case of setting $D$ too high, the resulting mesh may end up being too coarse such that fine image details may be inaccurately represented or lost. This may lead to rendering artifacts.

The projection from the resulting image to 3D space is given in the formulas 3.1 and 3.2. We can see that the projection only depends on the resolution and the field of view defined in the configuration. The position and viewing direction of the virtual camera does not influence the position of the vertices at this point. They are only needed later in the rendering process.

$$x' = -\tan\left(\frac{\text{fov}}{2}\right) + x$$  \hspace{1cm} (3.1)
$$y' = -\tan\left(\frac{\text{fov}}{2}\right) * \left(\frac{\text{height}}{\text{width}}\right) + y$$  \hspace{1cm} (3.2)

Step 2: Find nearby cameras for each vertex.
In this step, the nearby cameras for each vertex are calculated. To do this, we project a light ray from the center of the virtual camera through the vertex. Then, we determine the distance of each original camera to this light ray as shown in Figure 3.1. Under the assumption that all cameras lie on a single plane, the distance to the light ray is a good approximation for the similarity of the perspective of the compared cameras. A configurable number of cameras, sorted by distance to the light ray, is chosen to be used in the later rendering steps. This step is important for a number of reasons. Firstly, it reduces the number of cameras used for each vertex in the following rendering steps significantly, which is very important for the performance since multi-view video can contain 64 or more camera streams. Another reason is that when only close-by cameras are used, in combination with the assumption that the cameras are close to each other as compared to the distance between the cameras and the scene, it is safe to assume that every considered camera sees the same region of the scene. Therefore, the algorithm does not have to handle occlusions between the views explicitly.
Step 3: Estimate depth.
To estimate the depth for the vertices, the algorithm uses a plane sweeping approach. First, the depth range of the scene is quantized by a configurable number of depth planes. Each vertex in the initial mesh is then projected into all the neighboring images (determined in the previous step) and this is repeated for each of the depth planes, as shown in Figure 3.2. The algorithm assumes that the directions of the cameras are parallel to each other. This makes the projection simpler, but not every scene can fulfill this requirement. In the next step, for each set of points belonging to one depth plane, a patch of pixels is extracted from each image. A similarity measure is applied at the patch level. The available similarity measures are sum-of-squared distances (SSD) and a mean-removed correlation based value (formula 3.3). For synthetic scenes with very little noise SSD is sufficient, but for real, captured images with more noise, the correlation-based value leads to better results but it is more complex to calculate.

\[ r_{ij} = \frac{\sum_k (I_{ik} - T_i) (I_{jk} - T_j)}{\sqrt{\sum_k (I_{ik} - T_i)^2} \sqrt{\sum_k (I_{jk} - T_j)^2}} \]  

(3.3)

Now the depth plane with the highest similarity value is chosen as the correct one for this vertex. Afterwards, the projection coordinates for this plane are saved for the last rendering step.

For the correlation-based value measure, mechanisms are implemented to accept or reject a depth plane before all planes are checked. This can increase the performance without loss of quality if the thresholds for acceptance and rejection are chosen correctly. Since those values highly depend on the scene content, there is no general rule for choosing them. For scenes with larger areas without texture or textures with a certain repetition pattern, the depth estimate may
not always be accurate. Of course this distorts the resulting mesh, but especially in textureless areas those errors will not lead to any visible rendering errors.

**Step 4: Calculate rendering weights.**
The rendering weight determines the influence of a certain image (original camera) to the appearance of a vertex in the rendered view. The weight is composed of two factors. The first is the distance to the virtual view, calculated in **Step 2**. To get a value that gets smaller with increasing distance, the inverse value of the distance is used as the weight directly. (At this point, the values do not have to be normalized yet, as will be explained more detailed later). The second factor provides a smooth transition to the image border (black region) if the projected coordinates lie to the border of the texture. Inside the image this value is always 1. Within a certain distance to the image border, it is a linear falloff from 1 to 0. Outside the image borders it is always 0. The value of this second factor is simply multiplied with the previously calculated weight. The last step is a normalization that guarantees that the sum of all weights for a vertex is equal to one. The normalization is important for the final rendering because the algorithm combines the images using alpha blending. Namely, it renders each image with a transparency value equal to its weight and stacks them on top of each other. If the weights do not sum up to one, the resulting image becomes too dark or too bright which can be seen as a rendering error.

**Step 5: Subdividing the mesh.**
Subdividing the mesh is the mechanism that allows to adapt the complexity of the algorithm to the scene geometry. For each triangle, the algorithm calculates the maximal difference in depth between its three vertices. If this difference is so big that at least one other depth plane fits in between, the triangle is subdivided since we assume an edge between foreground and background lies inside the triangle and the mesh should be more precise there. Figure 3.3 shows how we subdivide the triangles. In the middle of each side of the triangle, a new vertex is inserted and the triangle is therefore replaced by four smaller ones. We assume that this new vertex uses the same images in the final rendering as the vertices in the original triangle. The depth value for the new vertex is calculated with the same method used in **Step 3** of the algorithm. This step can be repeated multiple times depending on the scene complexity and the configuration. The effect of the subdivision can bee seen in Figure 3.4.

The amount of subdivision needed to create a sufficiently clear result depends on the scene complexity and the chosen value for the size of the initial mesh in **Step 1**. Complex scenes normally require a higher level of subdivision, but the initial mesh has still to be fine enough to capture the details with at least one vertex. Otherwise, there will be no subdivision at all.

**Step 6: Rendering the view.**
For each original image, we check whether there is a triangle that needs this image for rendering. If so, each triangle is rendered in the following way. The precalculated projected coordinates for each vertex are used to map a portion of the image onto the triangle (using standard texture mapping). The color information from the texture is then combined with an alpha value that corresponds to the weight for that image. This is repeated for every triangle and for each
used image, until the final frame is complete. Figure 3.5 shows the resulting image after rendering each weighted original image.

**Implementation.**
An implementation of the described algorithm is made available through the `caview` software provided by the author. The `caview` is freely available, open sourced, well documented and contains a number of test sequences and configuration-file examples. To synthesize the required novel views, `caview` builds a 2.5D mesh by estimating the depth values from the available color images, as described in **Step 3.** The final result is produced by a combination of the source images using alpha blending in OpenGL.

### 3.1.1 Analysis

In view of the requirements stated in Section 1.4, the baseline algorithm leaves the following issues open:

- The algorithm currently does not achieve real-time performance. Even if the resolution of the source images is just 320x240, which is at the very low end of today’s commonly used resolutions, the result is only rendered with 4 to 10 frames per second [16]. This is not nearly enough to fulfill the requirement of real-time performance in the general case. To achieve that, a frame rate of about 25 frames per second is required.

- The data rate with all views in raw format is so high that the data transfer between the loading and the rendering stages may affect real-time performance. Our experiences so far suggest that the memory speed may become the system bottleneck.

- In general, the techniques used in the algorithm implementation are at a technological state that is 8 years old. Especially for the rendering part, there are better and faster techniques available today (e.g., shading instead of repeated rendering). Because of that fact, there is a lot of potential for performance increase.
Figure 3.4: Mesh after increasing number of subdivisions.
• Most of the computationally expensive parts of the algorithm can be calculated for each vertex or triangle individually. This makes the algorithm well suited for parallel computation. Therefore, we propose the use of GPU computation for acceleration.

• Optimized quality/complexity trade-offs for this algorithm, so as to maximize the rendering quality under the given time budget, are not implemented.

• Depth estimation is only reliable on sample scenes. This can be traced back to problems in the coordinate projection and the use of the calibration data. This prevents the current software to be used with sequences where calibration data does not follow the specified format. It is also important that the cameras have not parallax and they are positioned on a plane with equal distance to the scene.

• Implementation currently does not support moving images (view rendering for video) because it only loads the images once and then continuously uses those to render the final views.
Chapter 4

Proposed fast implementation based on OpenCL

As our analysis of the baseline algorithm in Section 3.1.1 shows, caview does not fulfill our real-time requirement in its original state. To this end, we propose a fast implementation using (1) shaders and (2) OpenCL. As we will show in the sequel, the use of shaders only improves the rendering performance, while OpenCL is used to accelerate the depth estimation. During the development, we identified a number of challenges, with the mapping of the original algorithm to shader and OpenCL code. Our solutions to the issues we identified are described in the following sections.

4.1 Changes to program structure

Since the available implementation of our baseline algorithm only supports the input of scenes with a single frame per view, we had to change the input part of the program to allow for moving images and new data formats. To achieve the greatest flexibility in terms of input formats and types, we moved the code responsible for image loading into a separate process and implemented a simple interface for the data transfer and the synchronization with the main program. This allows us to load data of all kinds without having to change the computational part of the program. Figure 4.1 shows the separate processes in our proposed program structure and the hierarchy of the classes inside them. The classes Movie Loader, Image Loader and Network Loader handle the input of image sequences, image formats and devices, respectively. For this thesis we only added new image formats, the ability to load image sequences and new calibration data formats. But as soon as real-time multi-view video decoders become available, we will be able to quickly integrate them into this framework.

The original baseline algorithm only supports the single-threaded computation on the CPU. Since we wanted to introduce new computation devices like GPUs,
we designed a class hierarchy for the computational part to keep the old algorithm mostly untouched while being able to implement and test new ways of rendering our frames. Which class is responsible for calculating the result is defined by a set of command-line arguments. This enables us to run the same image sequences on different devices without any changes. Our final implementation contains a single-threaded processing class which is very similar to the original implementation of caview with optional shader support. The class OpenCLProcessor contains the host code for our OpenCL implementation, which is described in detail in Section 4.3.

4.2 Shading

One of the reasons for the low performance of the algorithm was identified as the way the resulting intermediate frames are rendered. When the algorithm was published in 2004, it might have been state-of-the-art, but is considered deprecated by today’s standards. To this end, we propose to use OpenGL shaders instead of the old fixed pipeline rendering. A shader-based implementation makes our rendering much more efficient. However, the large number of parameters and images needed to render a single frame poses specific problems. In the sequel, we detail on the challenges in developing a shader-based implementation and our solutions.

4.2.1 Pre-computation of projection coordinates

The first problem that required changes to the original algorithm was the fact that for efficient rendering with shaders, we need all the rendering data beforehand. The original implementation does not have this requirement. Under certain conditions, there are projection coordinates missing which are then calculated during rendering in the original algorithm. This happens when the three vertices of a triangle do not rely on the same subset of input images to render the final image. For example (Fig. 4.3), if the first two vertices use the image subsets $A = B = [1, 2, 3, 4]$ and the third uses the subset $C = [1, 2, 5, 6]$ then the projection coordinates for the combined set $D = A \cup B \cup C = [1, 2, 3, 4, 5, 6]$ are needed for an error-free rendering result.

If only the available information is fed to the OpenGL shaders, it leads to clearly visible rendering artifacts in the form of discolored lines. This happens because
OpenGL only gets the texture coordinates for the corners of a triangle and the values for all the pixels inside the triangle are interpolated from those. If we look at the example above, the rendering with the first two images (1 and 2) works fine, since they are contained in each of the subsets. However, the last two images in each subset (3, 4 and 5, 6) are different. Depending on the way we render the triangle, OpenGL either is missing information for at least one vertex or needs to interpolate between two different textures. Both cases lead to undefined behavior of the graphics hardware and therefore produces rendering artifacts.

Our solution strategy is as follows. The algorithm computes the projection coordinates and weights for the subset assigned to each vertex. To find out which set of images is needed for correct rendering, we first have to calculate the combination of the image sets for each triangle. In the second step, we have to determine the combination of the resulting sets of all triangles that a vertex belongs to. However, the difficulty with implementing the second step is that the number of triangles which use a certain vertex is not known, can change quickly in the algorithm and there is no real upper limit. This means it is hard to track the connections between vertices and triangles. We could not find an efficient way to calculate and store the combined set and the naive computation is so inefficient that we cannot calculate the combined set for each vertex in a reasonable amount of time.

We circumvented this complex problem by calculating and saving the projection coordinates for every source image as soon as we determined the depth of a vertex. We then give all these coordinates to the shaders and rely on the OpenGL internals to solve the subset problem. The exact solution with OpenGL shaders is described below.

### 4.2.2 Shader structure

In the final version of our implementation, we use an OpenGL pipeline consisting of a vertex shader, a geometry shader and a fragment shader. Figure 4.2 shows our proposed shader structure and a high-level description of the tasks of each shader stage.
All the data we supply to OpenGL goes into the vertex shader. In our implementation, all the processing that needs to be done per vertex is already done in the depth estimation part of our algorithm. Therefore, the vertex shader only transforms the positions of the vertices so that they are rendered in the correct position on the screen and gives all data to the next level of shaders.

The geometry shader is called for every triangle we want to render. Figure 4.3 shows how we use the geometry shader to solve our subset problem (Section 4.2.1). Each input triangle contains up to 3 different sets of images, which we have to combine into one. We separate every triangle into three new triangles with the same spatial coordinates. The shader assigns one of the image sets of the original triangle to each of these new triangles. Considering the example above, the first and the second triangle will be rendered using the subset $A = B = [1, 2, 3, 4]$ and the third triangle will use the set $C = [1, 2, 5, 6]$. Since we have pre-computed the projection coordinates for each vertex onto every input texture, the shader is able to determine the coordinates that correspond to each image in a set. The image weights are only calculated for the images in the set. If a vertex does not have a weighting factor for an image, because the image comes from a different vertex, we set the corresponding weight to zero. This guarantees that the calculated weights are preserved and we get a smooth transition between the images. The three new triangles can be rendered without errors, because each of their vertices shares the same image subset.

The final step in the rendering pipeline is the fragment shader. It is called for each part of a triangle that can be seen in a pixel of the final view. In this step, we combine all the images from the image set. For each texture that is listed in an image set, we read the color from the position indicated by the projection coordinates. These color values are then combined using the weights, which have been calculated by the geometry shader. The alpha value of the resulting color is set to $1/3$ to blend the three triangles together that were created in the geometry shader. This way, the triangles with different image sets $A$, $B$ and $C$
behave like one single triangle with the combined image set $D = A \cup B \cup C$. This solves our subset problem because we do not have to know the combined set before the rendering process starts and still get a result which behaves exactly like a triangle with the combined image set.

### 4.2.3 Shader-friendly parameter transfer

The attributes we need to render each vertex, consist of image IDs and weights (1-4 IDs and weights, depending on the number of images used for interpolation), one position in space and one projection coordinate per input texture. Since all shader parameters in OpenGL are represented as vectors with four components, a three-dimensional rendering position needs the same space as a single weight. For large scenes with big camera arrays, it can easily happen, that we already require 16 or more attributes for just the projection coordinates. Since the maximum number of attributes we can give to one vertex is limited by the graphics hardware and even modern GPUs only allow 16 attributes per vertex, this is a problem.

To solve this problem, we used OpenGL buffer texture to transport the data to the shaders. Our algorithm stores the data in a buffer which can be seen as a large one-dimensional texture from the shader perspective. Instead of the large number of coordinates, we now just give an internal vertex index to the shader. The geometry shader uses this index to calculate an offset which indicates where it can find the coordinates for the current vertex in the buffer texture. With the image IDs, it determines the coordinates it currently needs and sends them as attributes to the fragment shader.

Since our algorithm limits the number of images used per vertex to 4, which is sufficient for most camera array architectures, we do not have more than 16 vertex attributes between each shader stage. This allows our algorithm to render intermediate views from multi-view video material with a nearly unlimited number of input views. The only limiting factor is the number of entries we can store in one buffer texture. We can increase this number by using multiple buffer textures instead of just one. Then we are only limited by the available memory on the GPU.
In the first versions of our implementation, we used a custom array of 2D textures to access the source images in the shaders. Although our available GPU allowed 192 textures at the same time, and the number of textures we use for rendering in the last step cannot be larger than 4, we experienced severe rendering errors using this method. Because of that, we switched to the 2D_TEXTURE_ARRAY type of OpenGL to store the source images. This allows us to use up to 1024 textures of the same size as a single texture with an additional dimension to choose one of the textures. This resolved our rendering problems, but since the 2D_TEXTURE_ARRAY type is not supported in OpenCL 1.1, this introduced difficulties in our OpenCL implementation, which will be detailed later.

4.2.4 Avoiding dynamic memory allocation

Inside the shaders, it is not possible to allocate memory during run-time. This means that the size of all arrays and other structures has to be known before compiling the code. There are a couple of factors like the number of views in the scene or the number of views that are used to create the new image, that are only known after our program has got the information from the configuration file. Always using the maximum possible size for every array would be highly inefficient and may cause problems with the hardware because our algorithm does not limit the range of most parameters. Luckily, the source code for the shading programs can be compiled while our program is already running. Since we want to be flexible when it comes to the hardware we use, such an online compilation is even recommended because all manufacturers provide their own shader compilers. For optimal usage of the available memory, we use placeholders in the shader source code where the size depends on configuration values. After the shader source code is read from the file, we replace the place-holders with the correct values. As this happens after the configuration and therefore after all the critical parameters are known, we can allocate the exact amount of memory needed for the calculations with the least overhead possible.

4.3 Parallel computation using OpenCL

Our analysis of the baseline algorithm (Section 3.1.1) showed that most of the calculations are done for every vertex in the scene mesh independently. Since there is a large number of vertices present, even in small scenes, we propose to use the highly-parallel architecture of modern GPUs for acceleration. We decided to use the OpenCL framework to perform this acceleration. Although there are other frameworks available which might achieve a higher performance since they are optimized for the hardware capabilities of devices of a certain vendor, we chose OpenCL since it does not limit the hardware to use to the products of a single manufacturer. We now detail on our OpenCL implementation.

Converting standard C to OpenCL In a first step, we transferred the parts of the caview code which are repeated for every vertex or every triangle from standard C code into OpenCL kernels. Since OpenCL allows the use
of most standard C commands, including array handling and control structures for branching, this is mostly a 1:1 conversion.

### 4.3.1 Optimized data placement

Next, we analyzed how the memory that our algorithm uses has to be mapped to the OpenCL memory architecture for optimal results. All the information we need for rendering or intermediate results between the algorithm steps is stored in global memory. We justify this decision as follows. The rendering information has to be placed in the global memory space, because it contains large amounts of data which have to be available to all instances of our kernels at all times. The intermediate results could be stored in faster local memory, but the persistence of those memory spaces cannot be guaranteed over multiple kernel calls. Additionally, we cannot make sure that a second kernel call with possibly different working dimensions and offsets operates on the same local memory as before. Those two uncertainties make the local memory impractical for our algorithm. Although access to the global memory space is considered slow and should only be used to stream large chunks of data in or out of it, we do not have another choice, because we cannot predict which parts of the memory are needed by a certain thread and the overall size of all data is often too big for the faster memory spaces. Additionally, we need the persistence of the global memory between different kernel calls to ensure that the algorithm works correctly. Therefore, we use the global memory to store the intermediate results. All other variables which are only used during one kernel call, are either stored in private or constant memory, depending on how they are used.

### 4.3.2 Memory alignment and pointer size

OpenCL uses a certain memory alignment to get optimal performance from SIMD units on CPUs and GPUs. Specifically, it means that all variables have to be aligned to a power of two bytes. All the structures that are copied directly to/from OpenCL have to adhere to the same alignment, otherwise the values in the host and the OpenCL device are not consistent. For structures that contain pointers, it is also important to know that the size of a pointer might be different on the host and on the OpenCL device.

### 4.3.3 Handling dynamically-growing triangle lists

In our algorithm, we have some parts of code, namely the subdivision and complete mesh methods, which require the kernel to work on each triangle in a certain list and the kernel is able to add new entries to this list. This means we had to find a way to adapt the number of OpenCL threads to the current size of the list. We decided to store the size of the list before the kernel worked on it and to compare that value with the size after the kernels have completed their work. If these differ, we additionally run the kernels on the difference between the old and the new size so that all the newly created items are processed. This is repeated until the number of items in the list does not change anymore.
This requires some synchronization and data transfers between the host and OpenCL because we always have to wait until the kernels are done, then read the current size of the list we are working on from the OpenCL memory and issue another run of the kernel if needed. This means we cannot fully utilize the GPU’s computation power, but since we cannot run additional kernels from inside a running kernel and we do not know how many new items will be generated during a certain run, this solution is nearly optimal. Simply running more threads is not possible, because the additional threads will immediately finish working before the other threads can add something to the list.

4.3.4 Efficient data transfer between OpenCL and OpenGL shader

In order to save as many data copy operations as possible, we made use of the OpenCL-OpenGL interoperability feature which allows us to use OpenGL events for OpenCL synchronization and to access buffers from OpenGL as memory in OpenCL. By writing the results of our depth estimation directly into the buffers that we are rendering from, we do not need to copy the data in the end and we can save some time. Although the OpenCL specification clearly states that the developer of the program has to take care of the synchronization between OpenGL and OpenCL to maintain buffer consistency during rendering and OpenCL computation, the drivers are allowed to do implicit synchronization as they see fit. In our case the driver introduced a 20ms waiting period after the rendering commands were issued before it allowed us to start with the calculations for the next frame. This was not tolerable since this waiting period took half of the time budget we had for every frame and was not needed, since we synchronized everything correctly. To reduce this implicit synchronization period we tried to implement a round-robin buffer system so that a buffer that was used for rendering in one frame was not used for calculation in the next
frame. This approach was not successful and the synchronization problem remained. In the end, we decided to separate the computation and the rendering buffers and to copy the data from OpenCL to OpenGL just before rendering. The data paths of our OpenCL implementation can be seen in Figure 4.5. Since this is a GPU internal memory copy operation, it makes use of its high memory bandwidth and is therefore a very quick operation.

4.3.5 Image filtering in OpenCL

In chapter 4.2 we mentioned that the change of the internal representation of our textures from array of 2D textures to one TEXTURE_2D_ARRAY is challenging, since 2D_TEXTURE_ARRAY type is not supported in OpenCL 1.1. In more detail, due to the current state of available OpenCL drivers for our hardware, we were limited to OpenCL 1.1 in this thesis. The texture type TEXTURE_2D_ARRAY is first supported in OpenCL 1.2. Fortunately, it is possible to treat a TEXTURE_2D_ARRAY, which combines multiple 2D textures of the same size, as a 3D texture in OpenCL. This approach is directly applicable if we are sampling the textures with nearest-pixel filtering, which gives the color value of the pixel closest to the requested coordinates. However, if we try to use linear interpolation to sample the texture, this approach performs the interpolation not only in the x and y direction within one image but also in the z direction between the images. This leads to errors in the depth estimation and causes rendering artifacts. In our case, linear filtering is attractive because our projected coordinates often lie in between two pixels and a linear interpolation would provide us with a higher image quality for depth estimation. In addition, because modern GPUs contain special hardware for this filtering, we would not experience any performance loss by using linear interpolation.

Since the limiting factor in this case is the current driver support for our graphics card, we were not able to solve that problem. However, our experimental results show that the impact of this limitation on the rendering quality is highly scene dependent. For example, Figure 4.6 compares two depth maps for the 'flowg' sample scene. The depth map on the left was calculated using nearest pixel filtering and the right depth map is computed using linear texture filtering in OpenCL. It is clearly visible that the right depth map contains less artifacts, especially for image parts in the background or transitions from foreground to background. In the bottom right corner the improvements can be seen best. The nearest pixel filtering causes artifacts while linear filtering models the slope into the background pretty well. Thus, for this scene, there is a significant improvement from the linear interpolation. Since we cannot use this interpolation on all scenes yet, we do not analyze the improvements in detail, but our tests suggest that the visual quality is a lot better for scenes with a lot of details in the background.

4.3.6 Memory differences with OpenCL on GPU and CPU

OpenCL allows to make calculations on a lot of different devices and GPUs are just one of them. So far we have assumed that OpenGL and the OpenCL device
work on the same memory, but this is not always the case. For example, if the computer has a GPU with dedicated memory, but uses its CPU for calculations, they are working in separate memory spaces. In such cases, it is not possible to make use of the interoperability feature for efficient data transfer between OpenCL and OpenGL, at least not for direct OpenGL buffer access. We then have to fall back to fully separated buffers and use only OpenGL commands to fill the rendering buffers. Since this decreases our overall performance, we created a function that decides whether this is necessary. Although there are no functions in OpenGL that can help us determine whether the graphics card uses the host memory or its own dedicated memory, OpenCL offers an extension that can check the compatibility of an OpenCL device with a certain OpenGL context. Namely, once we have established our output window, we know the OpenGL context we want to connect to. We then choose the OpenCL device to use for our calculations. If the two are compatible, we use the faster copy mechanisms. Otherwise, our algorithm automatically switches to the appropriate compatibility functions.

4.3.7 Precision differences with OpenCL on GPU and CPU

When making calculations on a GPU, it is important to know that their cores work differently than the ones on a CPU. Although modern graphics cards are able to calculate floating-point operations with double precision, they still only use single precision in default mode. The main reason for that is the fact that the double precision units are shared between multiple cores and therefore, the performance is a lot lower than in single precision. CPUs on the other hand, normally perform floating point calculations with maximum precision and round the result if necessary. During the debugging process, we noticed that the results for the floating point parameters for our vertices differ in the 5th decimal place when comparing the results from the CPU and the GPU. For some applications that may be a problem, but in our case it just means that the calculated coordinates are up to 0.00001 pixels off. Since we are not able to use the linear filtering for our textures (as mentioned above), these differences only cause
minimal errors in very rare cases (e.g., different rounding to the nearest pixel). Therefore, these errors can easily be tolerated by our program.

4.3.8 Inconsistent depth values on the GPU

When a triangle is subdivided, a new vertex has to be used in the middle of the edge. If the vertex has not been used yet for another triangle, its depth is recovered. The search is limited to the depth range covered by the original triangle. In the case that the new vertex is already used and its depth has been determined, nothing is done. Since the original \
\texttt{caview} implementation calculates the subdivision in a single thread the result is deterministic, because the triangle that first uses a certain vertex is always the same. Therefore, the recovered depth does not change as long as the input textures and and the camera position stay the same. For our proposed GPU implementation this is not guaranteed. Due to the huge number of parallel threads the order of execution cannot be determined. This means the first triangle for a vertex can change and with it the search range for the depth estimation. During debugging we noticed that this behavior is changing the recovered depth and can lead to artifacts in the depth map. To prevent this from happening, we decided not to limit the search range for new vertices. Therefore, the order in which the triangles are processed does not matter anymore and the result is stable. Obviously we need more calculations for the new vertices, but the GPU has enough resources to compensate for that.
Chapter 5

Experiments and Results

We tested our algorithm against the image sequences that were provided with the original implementation of caview. Our implementation can operate in one of four different modes: the original CPU implementation without shaders, the original CPU implementation with shaders, the OpenCL implementation running on the CPU and finally the OpenCL implementation running on the GPU. Our experimental results show that we can render intermediate frames with up to 50 frames per second, while using the same parameters as in the original implementation. In addition, we show that we are even able to use parameters which achieve a better visual quality than before, without losing the real-time property. A detailed analysis of the overall performance and the impact of the parameters on the framerate and image quality follows below.

5.1 Rendering performance

In Section 3.1.1 we found that the original implementation uses outdated technology to render the computed frames. Especially for scenes with a high number of vertices or input textures, the rendering can take a lot of time. The old rendering technique often consumes the whole time budget on its own, without leaving any time for depth estimation to complete. With the help of shaders, we were able to shift the work required to render the triangles from the CPU to the graphics card.

Figure 5.1 shows the effect of utilizing shaders instead of software-based rendering. Both measurements were taken with our single-threaded CPU-based implementation of the depth estimation. While the other parts of the algorithm need exactly the same time as before, the rendering time has been reduced significantly. Our shading approach only needs 2ms of CPU time compared to 89ms without shaders. The rendering performance is therefore increased by a factor of 45. For small scenes, the original implementation used around 400K OpenGL calls to render a single frame. With shaders, only 70 OpenGL commands are called to get the same visual result using the same scene parameters as before. In fact, the execution time of these commands is nearly negligible, such that our measured 2ms mostly refers to transferring the data from the RAM to the GPU. Since the data is transferred in big chunks instead of indi-
individuals for every vertex, the PCI-Express interface that connects the GPU to the rest of the PC can be used with higher efficiency. This also increases the performance of the rendering part of our algorithm.

In the above comparison, we chose to compare against the original single-threaded implementation without shaders. We justify the decision as follows. For our OpenCL implementation of the depth estimation on the GPU, the comparison is not as straightforward. Since the data we want to render is already present on the GPU, we can utilize the high transfer rates of the internal GPU memory. Therefore, the transfer only needs a fraction of the time that was needed before. Additionally, the transfer command itself is only queued for execution and does not wait for completion. These factors reduce the time that is needed for rendering to less than 0.5ms. As our timer has a precision of 1ms, we cannot get an accurate measurement of the time taken to render a frame.

If the shaders are disabled, the data that was calculated on the GPU has to be transferred to the RAM before it can be used for software rendering. Since the data we need is contained in larger structures in the GPU memory, we have to copy some data that is not directly necessary for rendering. As the transfer to the RAM is much slower than an internal copy operation, it requires an amount of time that is not negligible. Since this data transfer is only needed because the GPU and the CPU work in different memory spaces, it skews the overall measurement results. This is the reason why we chose the original single-threaded implementation to show the improvements of shader usage. The fact that both measurements are significantly above our time budget of 40ms per frame, does
not make a difference.

5.2 Performance impact of algorithm parameters

For every scene there are a number of parameters that can be used to change the behavior of our algorithm. They include the number of images used, the resolution of the input images, the initial grid size, the maximum number of subdivision levels, the number of depth planes in the scene and the size of the window that is checked for image correspondence during the depth recovery. We discuss the influence of each parameter on the overall performance and the image quality in the following sections. All the measurements were taken while rendering the teddy scene with 9 views. Unless stated differently, we used an up-scaled version of this sequence with a width of 900 pixels and a height of 750 pixels. The following default parameters were used for the calculations: 10 depth planes, initial vertex distance of 10 pixels, 2 levels of subdivision and a window size of 5 pixels. For each measurement, 30 frames were rendered and we averaged the obtained values for the last 25 frames. We do this because the first frame always contains some initialization procedures which need some time, but are only executed for the first frame. To be sure that we get stable values, we started the measurements after the fifth frame. In each experiment, we take 4 measurements: for the original CPU implementation without shaders, for the original CPU implementation with shaders, for OpenCL calculations on the CPU and finally for the OpenCL calculations on the GPU.

Figure 5.2 shows how much our OpenCL implementation on a GPU improves the rendering performance compared to the single-threaded CPU computation. In comparison with Figure 5.1, we see a significant reduction in the time needed for depth estimation (a 15-fold reduction is observed in this experiment). In addition, we see that the rendering and finish mesh stages do not appear in the right graph. This is because these parts are calculated so fast on the GPU that the computation time falls below our time measurement precision and therefore appears as 0ms in our measurements.

5.2.1 Initial grid size

The initial grid size determines how many vertices we use to cover the image in the initial step of our algorithm. It defines the distance in x and y-direction to the next neighbors of a vertex. Therefore, a linear decrease of the initial grid size leads to a quadratic increase of the number of initial vertices. Since most operations in our algorithm have to be performed for every vertex individually, the initial grid size has a great impact on the overall performance.

Table 5.1 shows the time needed to render one frame in each operation mode with different values of the initial grid size. All the calculation modes show the expected quadratic decrease in computation time. They do not fit the curve exactly, but this is due to the fact that not all parts of the algorithm are directly related to the number of vertices. For example, the time it takes to load the textures is constant in each mode, regardless of the initial grid size. The
impact of the number of vertices on rendering time is most visible for CPU computations without shaders. As we have discussed in Section 5.1, the impact is marginal for all other modes.

The increase of computation time with decreasing grid size is much slower for our OpenCL computation on the GPU compared to other modes (Figure 5.3). The main reason is that the structure of our algorithm is optimized to work with a very high number of vertices. In general, the architecture of a GPU is most efficiently used if it works with a very high number of threads in parallel. For the teddy scene, the GPU gets saturated for an initial grid size of around 12 pixels. At this point, the time per frame starts to increase significantly because all cores of the GPU are used and additional computational operations can only be performed by using some cores for more than one thread. It can also be seen that our implementation can handle parameters with higher computational requirements while staying within the time budget. Compared to the original CPU implementation, in this operational mode, we achieved an increase in performance of up to 20 times as a result of our optimizations.

5.2.2 Subdivision

The subdivision parameter determines how often the sides of a triangle can be subdivided. Since the calculation of this operation contains a lot of testing operations and branching, it is not ideal for GPU computations. Additionally, the original algorithm uses loops with dynamic limits that require additional
Table 5.1: Rendering times for different initial grid sizes (in ms).

<table>
<thead>
<tr>
<th>Computation mode</th>
<th>2px</th>
<th>6px</th>
<th>12px</th>
<th>18px</th>
<th>24px</th>
<th>30px</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU, no shaders</td>
<td>4880</td>
<td>837</td>
<td>247</td>
<td>119</td>
<td>72</td>
<td>52</td>
</tr>
<tr>
<td>CPU with shaders</td>
<td>4123</td>
<td>684</td>
<td>208</td>
<td>103</td>
<td>65</td>
<td>48</td>
</tr>
<tr>
<td>OpenCL on CPU</td>
<td>2059</td>
<td>406</td>
<td>128</td>
<td>71</td>
<td>49</td>
<td>38</td>
</tr>
<tr>
<td>OpenCL on GPU</td>
<td>225</td>
<td>47</td>
<td>26</td>
<td>23</td>
<td>22</td>
<td>22</td>
</tr>
</tbody>
</table>

Figure 5.3: Rendering times for different initial grid sizes.
synchronization, as mentioned in Section 4.3.

As subdivision adapts the mesh to the scene geometry, it can help to reduce the needed computation effort by using a coarser initial mesh. The optimal value of subdivision depends on the scene, but often 2 to 3 subdivision levels are enough to get a good adaptation. A vertex distance of less than one pixel in the maximally subdivided mesh is not useful, since the textures we use for depth recovery do not have sub-pixel accuracy and therefore, a finer mesh will not provide visible improvements. Such a fine mesh just adds unnecessary computation efforts due to the increased number of vertices.

The influence of the number of subdivision levels on the total rendering time is shown in Table 5.2 and in Figure 5.4. Similar to the initial grid size, it displays a quadratic increase for a linear change in the number of subdivision levels. Since every subdivision step may divide the previous grid in two dimensions, the number of vertices can be doubled in each step. The number of tests that have to be done directly depends on the number of vertices in the scene. Such a test would have to be repeated for every subdivision step. This makes subdivision an expensive operation, but it is still more effective than calculating the depth for every vertex of a finer initial grid.

Our results show that our GPU implementation is able to handle the effort of subdivision much better than the original CPU implementation. We can see that the optimal usage of the GPU cores is between the first and the second subdivision step. In contrast to other computation modes, our OpenCL mode on the GPU can handle up to two subdivision steps within the time budget of 40ms. In this test, we have achieved an acceleration of more than 17 times over the original implementation.

### 5.2.3 Number of depth planes

The number of depth planes determines how many equally-spaced planes we use to divide the depth range of the scene. Therefore, it directly influences the number of tests we have to perform during the depth estimation for each vertex. Because of this fact, we expect a linear dependence of the total time to this parameter.

The results of our tests for different amounts of depth planes can be found in Table 5.3. All modes show a linear increase we were expecting (Figure 5.5). As this parameter directly influences the complexity of per-vertex calculations...
Figure 5.4: Rendering times for different subdivision levels.

<table>
<thead>
<tr>
<th>Computation mode</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU, no shaders</td>
<td>32</td>
<td>166</td>
<td>284</td>
<td>492</td>
<td>859</td>
<td>1483</td>
</tr>
<tr>
<td>CPU with shaders</td>
<td>26</td>
<td>129</td>
<td>233</td>
<td>427</td>
<td>782</td>
<td>1411</td>
</tr>
<tr>
<td>OpenCL on CPU</td>
<td>27</td>
<td>65</td>
<td>137</td>
<td>314</td>
<td>781</td>
<td>1601</td>
</tr>
<tr>
<td>OpenCL on GPU</td>
<td>20</td>
<td>23</td>
<td>28</td>
<td>41</td>
<td>66</td>
<td>129</td>
</tr>
</tbody>
</table>

Table 5.3: Rendering times for different number of depth planes (in ms).

(which can make up more than 90 percent of the total rendering time), doubling the value of this parameter nearly doubles the total rendering time. Our GPU method is optimally used with a value between 8 and 16, where all the cores work are fully utilized. The time budget allows about 16 planes while keeping the total rendering time under 40ms. Overall, with respect to the number of depth planes, our improved implementation is 10 times faster than the original algorithm.

5.2.4 Window size

The window size determines how many pixel are checked horizontally and vertically around the projection position of a vertex during depth estimation. This should lead to a quadratic connection between the window size and the total rendering time, because it is directly related the number operations needed per vertex.

Figure 5.6 shows that values for the non-OpenCL methods do not exactly match the expected quadratic increase. The reason is that small windows produce a lot of wrong depth values, which trigger subdivision and therefore create more vertices. An increase in the window size makes the depth estimation more precise, which reduces the amount of subdivision. For very small window sizes,
the increased complexity is amortized by the smaller amount of subdivision. As the window size continues to increase, the amount of subdivision converges to a stable value and the increase of the overall rendering time becomes quadratic. Although OpenCL implementations are similarly affected, they can handle the increased workloads better, which makes the rendering times appear nearly constant for small to medium window sizes. With respect to the window size, we have achieved a ten-fold performance increase (Table 5.4).

5.2.5 Image resolution

The resolution of the source images is an important factor of the subjective image quality. As we only change the image resolution, and keep all other parameters constant, the number of vertices in the scene increases quadratically. The total rendering time should show the same behavior. In addition to the increased number of vertices, it also increases the time needed to provide the images for all views to OpenGL and OpenCL. Depending on the number of views in a scene, the transfer of the raw data images can require a significant portion of the total time budget, although we are trying to copy the image in the background.

Table 5.5 and Figure 5.7 show the computation times for 3 different resolutions.
The rendering time follows a quadratic curve as expected. The main increase can be traced back to the higher amount of vertices in the scene. The OpenCL implementation suffers most from the increased image size with growing resolution, because the data also has to be provided to OpenCL.

Our GPU implementation suffers the most from image loading times, since the computation time is so short that the image loading is responsible for nearly 50% of the total rendering time. Nevertheless, we achieved a performance increase by a factor of 12.

### 5.3 Impact on rendering quality

Our informal visual evaluations showed that our implementation achieves a good rendering quality with all tested sequences. Since we lack a widely accepted metric to quantify the quality of intermediate views [11], in the sequel, we briefly discuss the potential impact of algorithm parameters on the visual quality.

**Initial grid size.** The impact of different initial grid sizes on the image quality cannot be easily quantified. Scenes without small details can tolerate much coarser initial grid sizes without compromising the resulting image quality. Even
if the grid is too coarse to capture the edges in depth exactly, active subdivision is able to adapt the mesh in those places. Only if there are small details which correspond to changes in depth, a tighter mesh is required to recover their depth correctly, but this also increases the required computations significantly. Therefore, there is no generic rule to get a good trade-off between speed and image quality. The optimal value has to be determined for every scene individually. But in general, we suggest to choose a finer grid if the time budget allows it, to be able to capture as many details as possible.

Subdivision. Subdivision is very important for the image quality, because it allows our algorithm to adapt to the scene geometry, but its effectiveness also depends on other parameters. For example, subdivision is only performed if the difference in depth in a triangle is high enough. This means, it is important that the initial grid is fine enough to cover every important detail in the scene with at least one vertex. Otherwise, it may create errors in the depth estimation that often create artifacts in the final frame. The optimal number of subdivision steps depends on the complexity of the scene geometry, since it has to make the triangles small enough to follow the edges in depth as closely as possible. Too many subdivision levels do not decrease the image quality, but reduce the overall rendering speed.

Number of depth planes. The optimal value for this parameter is highly dependent on the scene complexity and the range of depth that it covers, but in general a higher value is better. This is the case because the algorithm only considers the depth values of discrete planes during depth estimation. Since a good depth estimation is the key to a good rendering result, we always want to have as many choices as possible to get exact results. If an object lies directly in between two planes, it can easily happen that the test scores of those neighboring planes are very bad and the vertex is assigned a depth value that is completely wrong. If this happens close to a color change in the texture, it
results in a clearly visible artifact. A higher number of planes can help to avoid such problems.

**Window size.** Intuitively, comparing more pixels leads to stable and reliable depth values. But in caview, as well as our improved algorithm, large window sizes also lead to rendering errors. A larger window detects the depth more precisely, but especially for scenes with a small camera distance, it is possible that the window also catches the shifted background behind an edge. Often this results in a high similarity value at a wrong depth, especially around edges between the foreground and background (depth discontinuity). This causes a certain area of the scene around foreground objects to be classified as foreground pixels as well. This causes parts of the background to follow the foreground object when the camera position is moved. Therefore, we normally limit the window size to 5 or 10 pixels, depending on the camera layout and image quality.
Chapter 6

Conclusion

In this final section of the thesis, we want to summarize our contributions to the original algorithm, discuss the results and identify remaining issues for future work.

Our initial analysis of caview revealed that it delivers images with a good visual quality, but does not achieve a real-time performance. The slowest parts in the algorithm are the software-based rendering based on outdated OpenGL specifications and the depth estimation process.

First, we proposed to use an efficient rendering approach based on OpenGL shaders. With a combination of vertex shader, geometry shader and fragment shader, we were able to render the intermediate frames in a fraction of the previously needed time, while maintaining the good image quality.

Second, as a deeper analysis of the depth estimation process revealed, most calculations can be performed independently for each vertex or triangle. This is a great basis for parallel execution. Due to the high number of vertices in a scene, we proposed to use GPU computations via OpenCL. To transfer the depth estimation algorithm to the GPU, we have successfully adapted the standard C code to work within the limitations of OpenCL.

Our experiments have shown that our proposed methods to increase the performance of caview were successful. Although not all parts of the original implementation are well-suited for computation on a GPU, we increased the rendering performance at least by a factor of 10. This enables our algorithm to render intermediate frames in real time. Even with parameter settings for higher image qualities, the total rendering time stays below 40ms for each frame.

6.1 Open issues

Although we improved the rendering speed of caview, our implementation still has some issues that have to be addressed in the future.
Poor depth recovery. For captured scenes, the depth information recovered by our algorithm is not very precise. Especially in scene sections with a repeated pattern or in texture-less areas, our depth maps contain a lot of noise. Most of these errors are not directly visible in the final image, but if they appear near an edge in the texture, the artifacts are decreasing the image quality. We conjecture that the quality of depth estimation can be improved with an additional step to smooth the depth values in the grid while preserving the detected edges (depth discontinuities).

As described in Section 5.3, a large window size can produce artifacts around small foreground objects because of the simple comparison metric used in our depth estimation. Incorporating edge information from the textures might help with the exact recovery of edges in scene depth and therefore improve the image quality around small scene details in the foreground.

High input data rate. In the current implementation, the input images are transferred as raw data from the input process to the computational part of our program. Especially for larger scenes with many cameras, this results in a huge amount of data that has to be transferred per frame. Since the data still has to be copied into OpenCL and/or OpenGL memory, it takes so much time, that for higher resolutions it can consume the whole time budget we have for a frame.

A solution could be to provide the images in a compressed format which is directly decoded in the GPU memory. This would reduce the transfer time, as the slow memory bandwidth in RAM only has to handle the compressed data and the raw data is only present in the faster GPU memory.

For this thesis, there was no decoder available, capable of decoding multi-view video with a sufficient number of views in real time. Caching the whole sequence in GPU memory can be an intermediate solution, but it is only possible for smaller image sequences with a limited number of views. For larger scenes with many frames for each camera, it already required more than 4GB of RAM in our case. Currently, only a few high-end consumer-grade GPUs can store such an amount of data while handling the additional data needed during the depth recovery. This makes a real-time decoder the preferred solution.

Restriction for camera layouts and image sequences. Our algorithm has the same restrictions for an input sequence as caview. It requires that the distance between the cameras is small compared to their distance to the scene. They also have to be calibrated in a specific way for reliable depth estimation. This prevents our algorithm from being used with some well-known samples like the ballet or breakdancers sequences. As those scenes are often used for algorithm comparisons, these restrictions should be removed. To remove these restrictions, we may have to drop some of the assumptions on coordinate projection. It should be possible to exchange our projection calculations with more complex operations without changing the rest of the program. Since this would increase the complexity of one of the most used operations, it has to be determined how this effects our overall performance and a balance between camera layout restrictions and calculation complexity has to be found.
Bibliography


### List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>A 3D video communication system.</td>
<td>1</td>
</tr>
<tr>
<td>1.2</td>
<td>Systems for capturing multi-view video streams.</td>
<td>3</td>
</tr>
<tr>
<td>1.3</td>
<td>Rendering results without (left) and with depth information (right).</td>
<td>4</td>
</tr>
<tr>
<td>2.1</td>
<td>Rendering pipeline of OpenGL.</td>
<td>10</td>
</tr>
<tr>
<td>2.2</td>
<td>Internal structure of a NVIDIA GTX 680 [3].</td>
<td>13</td>
</tr>
<tr>
<td>2.3</td>
<td>Memory model of OpenCL.</td>
<td>16</td>
</tr>
<tr>
<td>3.1</td>
<td>Distance to virtual view.</td>
<td>18</td>
</tr>
<tr>
<td>3.2</td>
<td>Plane-sweeping for two cameras.</td>
<td>19</td>
</tr>
<tr>
<td>3.3</td>
<td>Triangle subdivision.</td>
<td>21</td>
</tr>
<tr>
<td>3.4</td>
<td>Mesh after increasing number of subdivisions.</td>
<td>22</td>
</tr>
<tr>
<td>3.5</td>
<td>Final frame after rendering each image.</td>
<td>23</td>
</tr>
<tr>
<td>4.1</td>
<td>Program structure of our implementation.</td>
<td>25</td>
</tr>
<tr>
<td>4.2</td>
<td>Proposed shader structure.</td>
<td>26</td>
</tr>
<tr>
<td>4.3</td>
<td>Illustration how the subset problem is solved.</td>
<td>27</td>
</tr>
<tr>
<td>4.4</td>
<td>Software rendering vs. hardware shading.</td>
<td>28</td>
</tr>
<tr>
<td>4.5</td>
<td>Data paths of the data needed for rendering.</td>
<td>31</td>
</tr>
<tr>
<td>4.6</td>
<td>Difference in depth estimation for linear and nearest pixel texture filtering in OpenCL.</td>
<td>33</td>
</tr>
<tr>
<td>5.1</td>
<td>Reduction of rendering time using shaders.</td>
<td>36</td>
</tr>
<tr>
<td>5.2</td>
<td>Reduction of rendering time using OpenCL on a GPU.</td>
<td>38</td>
</tr>
<tr>
<td>5.3</td>
<td>Rendering times for different initial grid sizes.</td>
<td>39</td>
</tr>
<tr>
<td>5.4</td>
<td>Rendering times for different subdivision levels.</td>
<td>41</td>
</tr>
<tr>
<td>5.5</td>
<td>Rendering times for different numbers of depth planes.</td>
<td>42</td>
</tr>
<tr>
<td>5.6</td>
<td>Rendering times for different window sizes.</td>
<td>43</td>
</tr>
<tr>
<td>5.7</td>
<td>Rendering times for different image resolutions.</td>
<td>44</td>
</tr>
</tbody>
</table>
List of Tables

5.1 Rendering times for different initial grid sizes (in ms) . . . . . . . 39
5.2 Rendering times for different subdivision levels (in ms) . . . . . . 40
5.3 Rendering times for different number of depth planes (in ms) . . . 41
5.4 Rendering times for different window sizes (in ms) . . . . . . . . 42
5.5 Rendering times for different image resolutions (in ms) . . . . . . 43