Multiple viewpoint rendering

by exploiting image coherence in epipolar space

Computer Graphics and Visualization
EEMCS, Delft University of Technology

Author
Kevin de Quillettes
Parallel and Distributed Systems
Master Computer Science

Supervisors
Prof. Dr. Elmar Eisemann
Leonardo Scandolo

TUDelft

Multiple viewpoint rendering by exploiting image coherence in epipolar space

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Kevin de Quillettes

Computer Graphics and Visualization Group
Department of Intelligent Systems
Faculty of Electrical Engineering, Mathematics, and Computer Science
Delft University of Technology

Delft, The Netherlands
https://www.eemcs.tudelft.nl

Author

Name:	Kevin de Quillettes
Student id:	4077482
Email:	K.A.deQuillettes@student.tudelft.nl

Title

Multiple viewpoint rendering by exploiting image coherence in epipolar space

Date of thesis defence

4 June 2018

Supervisors

Prof. Dr. Elmar Eisemann
Leonardo Scandolo

Thesis committee

Prof. Dr. Elmar Eisemann	Faculty EEMCS, Delft University of Technology
Klaus Hildebrandt	Faculty EEMCS, Delft University of Technology
Pablo Cesar	Faculty EEMCS, Delft University of Technology

Abstract

In recent years, there has been a widespread increase in the adoption of virtual reality and 3d displays which require many images as input, e.g. head mounted displays or lightfield displays. Real-time rendered imagery for such displays is commonly computed using brute-force rendering pipelines based on forward(+) or deferred rendering pipelines. This process can be sped up based on the coherence between images, and is the aim of the presented study. The result is a proof-of-concept framework meant as alternative for brute force generation of each image, whereby the developed algorithms interpolate from a set of source images based on the theory of epipolar geometry. This theory describes a relation of a surface point in 3d space between different images.

Besides advantages as better performance compared to the baseline, interpolation-based image generation also has disadvantages. Specific to the presented use case is the need for special support of commonly used view dependent properties. As such, support for specularity is demonstrated based on inclusion of the Phong reflection model in the framework.

Additionally, several techniques are included to improve the performance of the basic interpolation algorithm. Thus, further reducing the frame times of the developed interpolation-based rendering pipelines compared to the baseline rendering pipeline, which for this study is a deferred rendering pipeline. Although, the increased performance results in decreased image quality, the resulting images are of acceptable quality when compared to the baseline. All aspects considered, the results show a rendering framework based on epipolar geometry is viable, but the current implementation leaves many features used in production rendering pipelines to be ported. Moreover, the current framework implementation is built upon techniques that may have a lot of room for improvement.

Preface

Before you lies my master thesis report *"Multiple viewpoint rendering by exploiting image coherence in epipolar space"*. It is the culmination of my research towards increasing the performance of multi-view rendering. This master thesis has been done to complete the graduation requirements of the Software Technology Track at the University of Technology Delft. As such, this report marks the end of my study at the university.

The work for this master thesis was done under the Computer Graphics and Visualization group. Notably, this research was supervised by Prof. Dr. Elmar Eisemann. I would like to thank you for your guidance, insights and knowledge without which I would not have been able to conduct this research project. Furthermore, I would like to thank my daily supervisor Leonardo Scandolo who was always available to answer my questions, queries and think with me on the issues I faced.

To everybody else whom I had the pleasure to interact and work with during my time at the University of Technology Delft: I would like to thank you for your cooperation and knowledge. I am sure that I'm forgetting someone, but I would like to specially mention Ernst, Edward, Maikel, Niels and Radjino. You made the university fun and interesting for me. Finally, I want to thank my parents and brother for their support and believe in me finishing my degree. Especially, this master thesis that seemed to be endless. Thank you.

I hope you enjoy your reading.

Kevin de Quillettes

Leidschendam, The Netherlands
mei 28, 2018

Contents

1 Introduction

The current generation of graphics processing units (GPU) is capable of handling amazingly complex scenes consisting of several tens of millions of lit and textured triangles every second. Even lower-end hardware graphics chips, embedded in portable devices such as laptops and tablets, can handle realistic looking scenes in real-time.

However, consumers want more realistic looking videos and games in real-time, even to the point of photorealism. As such, the increase in fidelity of the shading models necessary to achieve this level of quality counteracts the increase in computing power of next-generation graphics cards.
To make this problem worse, even more computations are moved to the GPU since it is so fast at doing parallel computations. Examples are fluid simulation, collision detection or cloth simulation.

Another ongoing trend that diminishes the perceived increase in computing power is larger and higher fidelity displays since more pixels lead to more computations to color each of them. Besides this, there is a tendency towards multiple-viewpoint displays, which show a different viewpoint depending on the angle at which a user looks to the screen. Some examples are the new 3D TVs, head mounted displays such as the Oculus Rift or the HTC Vive, or light field displays.

The problem with multiple-viewpoint displays is the need for a multitude of input images, ranging from a couple for head mounted displays to several tens of thousands for light field displays. Fortunately, the difference between consecutive images is small. This allows for reuse of a lot of computations to generate new images.

Data reuse, or more specifically, exploiting spatio-temporal coherence is a heavily researched topic within the computer graphics research community. It has been successfully applied for improving ray tracing, increase performance of global illumination algorithms, and many more. For the purposes of this report, the interest specifically lies with decreasing frame time by reusing images. For this, several usable example algorithms are based on reprojection, image warping and multi view rendering.

1.1 Mission statement

The focus of the research presented in this report is to provide a method to efficiently, and in real-time, generate the set of images that can serve as input of multiple-viewpoint displays. As mentioned previously, one hinted possibility to achieve this is to reduce the amount of work spent on generating the image set. More specifically, can epipolar geometry be used? This is the chosen route during the currently presented research, and leads to the following main research question:

> *Can a framework be developed with the ability to generate a set of images corresponding to user controllable viewpoints by exploiting the coherence between consecutive images using epipolar space?*

The possibility for a reduced frame time likely leads to several approximations. This raises the question: "how much will the improvement be?" Also, it may also lead to an unreasonable image quality. In the end, the main research question is amended with two more research questions:

1) *How much is the performance increase over a baseline rendering pipeline? In other words, how much is the reduction in frame time when the newly developed framework is compared to a baseline rendering pipeline?*
2) *Are the resulting images of reasonable quality? More precisely, have the generated images using interpolation an image quality such that they are usable in consumer products?*

1.2 Main contributions

The research conducted for the presented thesis has resulted in the following contributions to the field of computer graphics:

1) A proof-of-concept implementation consisting of a collection of rendering pipelines which shows the viability of interpolating images based on epipolar geometry to generate additional images. The performance is such that it outperforms against the tested baseline rendering pipeline in situations that are workable, while maintaining a usable image quality.

2) The collection of rendering pipelines is combined into a framework designed and built to form the basis for future work. It includes tools to measure the performance and image quality of the used algorithms. Furthermore, a few options are included to inspect the inner workings of the algorithms and view the generated images.

3) Several optimizations and an extension on top of the basic interpolation framework have been developed and documented. The optimizations reduce the footprint of the basic framework, and show the framework is flexible and can relatively easily be amended. Additionally, specularity is added as extension since the interpolation pipelines don't natively support view dependent properties, which are commonly used. As such, the addition of specularity in the form of the Phong reflection model shows that it is possible to add support for view dependent properties.

1.3 Conventions

The following sections give detailed explanations and analyses of the research associated with this report. Most of the remaining document, the source view acquisition and epipolar view interpolation algorithms will be discussed. Herein, abbreviations and concepts are used that may be uncommon. The purpose of this section is to identify and clarify them.

Firstly, the presented rendering framework entails several rendering pipelines. Each pipeline consists of two stages. The source view acquisition stage, explained in detail in Section 3.3.1, is one of the following three variations: center view acquisition, dual view acquisition or hierarchical view acquisition. The second stage entails image interpolation and is explained in Section 3.3.2. The combination of the respective source view acquisition stages with the image interpolation stage will be abbreviated throughout the text to respectively center epipolar pipeline, dual epipolar pipeline and hierarchical epipolar pipeline.

Additionally, the basic epipolar rendering framework has several options available for activation. The different options must be tested in distinct configurations. Unfortunately, this leads to long strings signifying a configuration, and is exacerbated by the relatively long names for the individual options. To sidestep this issue for the different optimizations and extensions, the following abbreviations have been used throughout the text and figures where the full names would be too long:

- Specularity = Spec
- Primitive Coalescing = PrimCoal
- Visibility Estimation = VisEst
- Dynamic Layer Reduction = DLR

Natural language has a lot of inherent ambiguity. Mathematical notation avoids many of these issues. As such, the following sections rely on both natural and mathematical expressions to clearly communicate the ideas and inner workings of the epipolar rendering framework. However, some conventions are expected to be known.

Starting with the camera projection parameters. These parameters mathematically describe the projection of 3-dimensional geometry to 2d images and consist of the internal and external parameters of a camera. Examples are the field of view, near plane or far plane. It is commonly given in the form of one or more matrices. This also includes the viewport projection parameters. These values are used to project the scene geometry from world space to image space. The reason for aggregating these properties is that none are used separately and would clutter and obscure the main ideas.

Regarding concepts used in the pseudo code, there is an unfortunate knowledge gap between mathematical concepts and concepts implemented on actual hardware. More specifically, the epipolar rendering framework makes extensive use of textures. A widely known and well-defined concept with respect to graphics rendering APIs, such as the used OpenGL API. Unfortunately, mathematics has no such concept. So, for the remainder of this report, textures are represented as sets of tuples. Each tuple can be thought of as a pixel described by its x- and y-coordinate, and possibly other attributes such as color or depth.

A second concept used throughout the report and in the pseudo code, are so called views. They can be regarded as a special kind of textures. Explicitly, they contain images that are seen when looking through a specific camera. In other words, these textures contain the fragments[1] closest to the view position as seen from a specific camera.

The last concept that might be different from traditional uses is the frame time. That is the amount of time which is needed to render one frame. Unless otherwise specified this time is given in milliseconds, and expresses the elapsed time to render a complete set of views. More explicitly, it is the total time to render one view for each camera given as input. As an example, if ten cameras are given as input, then the frame time is the amount of time to render all ten views once.

1.4 Reading guidelines

The organization of this report is as follows. Chapter 2 starts with an overview of any previous work and scientific publications that discuss similar problems and solutions. Included in this chapter is an introduction into other important concepts needed for understanding the main contribution. Then, Chapter 3 discusses the presented framework in detail, including any optimizations and additional options. Following this, Chapter 4 starts explaining the used performance and quality measures and how these were measured. The same chapter ends with the performance and image quality measurement results. Chapter 5, explains the results as shown in Chapter 4 and discusses several limitations of the performed research. The second to last chapter concludes with answering the research question as posed in Section 1.1. Finally, Chapter 7 looks to the future and shows several possible directions of potential research.

[1] For more information about a fragment, see https://en.wikipedia.org/wiki/Fragment_(computer_graphics)

2 Background

This chapter will be the primer to the remainder of this report. To place the newly developed algorithm in context with regards to current state-of-the-art computer graphics research, Section 2.1 will provide some insight into general data reuse techniques and discuss notable research related to reprojection methods, stereo rendering and multi-view rendering. Then, Section 2.2 ends this chapter with an introduction to the main concept behind the developed view interpolation algorithm.

2.1 Related work

As mentioned in the previous chapter, the basis of the presented research is to exploit coherence between images to speed up the generation of them. Fortunately, exploiting coherence is an active area of research within the computer graphics community with study, development and documentation of numerous methods. Specifically, the interest of this report is data reuse targeted at rendering more views as fast as possible. The found methods are mostly targeted at increasing the framerate of image streams which is in line with the most common usage scenario of real-time rendering.

Temporal coherence

The rendering framework described in Section 3 exclusively exploits spatial coherence, which is one of several forms of coherence. Temporal coherence is a closely related form of coherence that is employed in many areas of computer graphics. For the purposes of this report, temporal coherence can be exploited by reusing computations when rendering views as seen by moving a single camera along a line within a static scene at different points in time. An identical situation, but viewed from the perspective of spatial coherence, is recording scene geometry from different camera viewpoints in a static scene at one point in time. This situation is the focus of the presented study.

However, temporal coherence can be exploited for many computer graphics related techniques. An interesting rendering technique which may benefit from exploitation of temporal coherence is ray tracing. Attempts have been made to use ray tracing for real-time rendering, but the rendering technique is still not heavily used in mass consumer products. Nonetheless, other uses of ray tracing still benefit from better performance. For example, rendering movie animation sequences faster or using less resources is beneficial to movie studios.

For ray tracers using a voxel spatial subdivision scheme, Jevans [22] exploited temporal coherence by only updating voxels with changed geometry. For subsequent frames, rays intersecting geometry in unchanged voxels could be ignored because the computation would be identical.

When ray tracing image sequences, temporal coherence can also be used to reduce the number of object intersection tests. Havran et al. [18] devised a method to reproject object intersections across frames for single bounce rays that pass through a pixel center. Possibly reducing the number of intersection checks to one for a set of pixels. If the reprojection fails, the normal ray tracer is used.

Temporal coherence can also be used to reduce the artifacts in ray traced image sequences. Consecutive images may exhibit flickering and popping due to sampling from wildly different directions. To aid this, Martin et al. [30] described a method to guide ray directions based on previous frames. The idea is to track ray hit points across frames using reprojection and ensure that pixels add some contribution of previously hit objects by directing new rays towards them.

On a different note, temporal coherence can also be exploited with photon mapping based global illumination techniques. Photon mapping is typically implemented as two passes [21]. First, the photon map is constructed by tracing the paths of photons emitted from light sources. Photon path tracing is identical to ray tracing, except that the interaction with a material is different. A photon can either be reflected, transmitted, or absorbed, which is probabilistically determined. Each photon bounce on a non-specular surface is stored in a map, which contains the intersection point, incoming photon power and incident direction. The second pass, gathers the radiance values for every pixel in the resulting image from the photon map.

Tawara et al. [43] described several possibilities to exploit the temporal coherence. Among other possibilities, gathering the radiance values for multiple frames can be done in a single pass due to the coherence between image samples in the temporal domain. Another possibility for data reuse amounts to reusing photon hit points for generating photon maps. A similar temporal coherence method is employed by Wang et al. [46].

Spatial coherence

Exploitation of temporal coherence, or the study of signals that correlate at different points in time, is one of many methods to reduce GPU workload. The subject of the presented study exploits a form of coherence known as spatial coherence and is concerned with the correlation of different points in space.

Intuitively, reducing the amount of data to produce in combination with a reconstruction filter could reduce the GPU workload. Yang et al. [52] have documented a method whereby images are produced at lower than desired resolutions. To get the correct image resolution, a geometry-aware reconstruction filter is applied to the low-resolution image. Specifically, the filter is a bilateral filter modified to reduce artifacts on depth boundaries and near high-frequency image features by adapting the reconstruction kernel to each pixel such that data integration across region boundaries is avoided.

Another geometry-aware upsampling algorithm was developed by Herzog et al. [20]. The spatial upsampling is computed as a weighted average with the weights being a function of sample orientation, linear depth and image space filter. This is combined with temporal coherence to increase image quality. More importantly, this method is suitable for execution on GPUs.

Besides reducing the amount of computation, it is also possible to reduce fragment shader invocations. Yong et al. [19] proposed a rendering pipeline which is prepended to traditional pipelines with a coarse processing step for triangles. Instead of processing fragments at pixel level, blocks of 2x2 or 4x4 pixels are processed under the assumptions that individual pixels within a block are identical. However, if the quality is deemed not enough, the data can be passed on to the original shader and be processed at individual pixel level.

But spatial coherence is not limited to reducing the amount of computation. it can also be exploited to improve performance of ray tracing. Kim et al. [24] explored the usage of ray tracing on mobile hardware and present an adaptive undersampling technique to reduce the number of rays that need to be traced. The method adaptively decides whether rays need to be traced or can be approximated based on similarity of neighboring samples, whereby ray tracing of missing samples is replaced by cheaper linear interpolation of geometric attributes at the first hit points.

Reprojection

Another aspect of the presented research is rendering frames as fast as possible by exploiting coherence. An intuitive idea to increase the framerate when generating images is reusing previously generated frames by mapping the corresponding pixels in the source image to the target image. In this case, the source image would be computed frames based on the scene geometry, and the target frames are extra frames that are generated from the source images by means of copying pixels. In literature this idea is referred to as reprojection. Now the question remains: What mapping can be used to efficiently reproject source information to the target?

An intuitive possibility would be to reuse computation results from existing images to generate new images. This is the basic idea behind the method developed by Nehab et al. [32]. Practically, the proposed method is implemented as an addition to exisiting fragment shaders. Augmented with a cache that stores values associated with visible surface points in viewport-sized, off-screen buffers. To support this method, fragments shaders are modified to fetch a value from the cache if possible. Otherwise, the original shader is evaluated.

The accuracy of values stored in the cache will degrade over time. The authors counteract this by refreshing the cache. Two possible refresh policies are discussed. With the screen partitioned into a grid of non-overlapping tiles, each tile could be refreshed in turn. Or, tiles could be randomly selected to be refreshed.

To find a correspondence between the values stored in the cache and the new frame, the authors supply the parameters of the source frame alongside the necessary parameters for the current frame. This allows the vertex shader to attribute the vertices with the projection-space coordinates of the same vertex in the cache, which is interpolated by the hardware such that each pixel knows it correspondence to the location in the cache. This is augmented with a comparison of the depth, stored in the depth map of the source frame to filter out occlusions by unrelated surface points.

The scene-assisted interpolation method by Yang et al. [53] uses an identical method to track the correspondence between pixels in different frames. The difference being that scene-assisted interpolation uses the previous and next frame to generate one or more in-between frames. For reprojected frames, only the depth buffer is rasterized. The corresponding color is fetched from the source frames whereby the color is merged using a few simple rules. If a surface point is visible in both source frames, then the resulting color is a linear interpolation. If a surface point is visible in one source frame, then that color is used. Otherwise, the authors give the option to evaluate the original shader to compute the correct result, or, the pixel in either source frames that is closest to the camera can be used.

The previously mentioned ideas required manual modification of exisiting rendering pipelines to add reprojection capabilities. The paper “Automated reprojection-based pixel shader optimization” by Sitthi-amorn et al. [41] provides an offline method with the ability to automate this process, and builds upon the reprojection cache introduced by Nehab et al [32].

As input, a pixel shader and representative rendering session are expected. The modification pipeline operates on the abstract syntax tree (AST) of the pixel shader where leaf nodes represent variables, internal nodes represent calculations and the root node is the output color of the pixel shader. The goal is to find the optimal internal nodes, which are expected to be expensive to evaluate, and replace them with cache fetch instructions.

To guide the node selection process, a performance and error model are trained based on a representative rendering session. The performance model is a prediction of the average time to render a single pixel. This model is trained based on the render time, cache hits and misses of several randomly picked frames from the sample rendering session. The error model is trained for different refresh periods, and trained on a set of pre-generated shaders where each permutation caches a different AST node. The error model gives an estimate of the average pixel error over a set of rendered frames. To select a specific shader permutation for usage, the user must provide an error threshold. The system selects the shader which is below the specified threshold and has the best performance.

Image warping

A similar idea to reprojection is referred to as image warping in literature. The contrast with reprojection being that image warping operates on image data, whereas reprojection is more general and can be applied to parameters that are internal to the rendering pipeline.

A method developed by Didyk et al. [10] uses image warping to upsample an image stream. In other words, a stream of rendered images is interspersed with warped images to increase the framerate.

Along with the source images that will be warped, the pipeline assumes a depth map and motion flow map are given as input. The additional maps are either a byproduct of the rendering process or can easily be obtained.

To extrapolate images, a regular grid is placed over a source image. The grid is adjusted by moving non-edge grid vertices, that are close to a discontinuity in the motion flow map, to the discontinuity edge. The irregular grid warps the source images by applying the motion flow to the grid vertices since the vertices have the original texture coordinates associated with them. Additionally, the warped images are selectively blurred to hide small imperfections due to disocclusions. However, this results in loss of high-frequency image content and is compensated for by subtracting a blurred version of the source image to increase the high-frequency content. In turn, the modified source image could exceed the display's dynamic range. To ensure this does not happen, some high-frequency content from the modified source image is moved to the warped images and a gamma correction is applied.

Another method, developed by Yang et al. [53], is bidirectional image warping to increase the framerate of a real-time rendered image stream. The input to the algorithm is two traditionally generated images I_t and I_{t+1} rendered at time t and $t+1$, along with the corresponding depth maps. Furthermore, the forward motion flow map V_t^f and backward motion flow map V_{t+1}^b which respectively encodes the motion of every pixel from image I_t to I_{t+1}, and the motion from every pixel from image I_{t+1} to I_t are expected as input. These might differ due to occlusions.

To generate an intermediate frame at time $t+\alpha$, the iterative greedy search algorithm described for the forward direction by the following equations is applied to each pixel:

$$p_{t,0} = p_{t+\alpha}$$

$$p_{t,i} = p_{t+\alpha} - \alpha * V_t^f[p_{t.i-1}].xy$$

In these equations, $p_{x,y}$ is the image-space pixel location at time x in iteration y. Note that the authors improve on these basic equations by using a different initialization for the greedy search algorithm.

Furthermore, the clip-space depth z^f and a measure of the screen-space error e^f are computed using these equations:

$$z^f = Z_t\left[p_{t,i}\right] + \alpha * V_t^f\left[p_{t,i}\right].z$$

$$e^f = ||\,(p_{t,i} + \alpha * V_t^f[p_{t,i}]) - p_{t+\alpha}\,||$$

In parallel, a similar iterative search is done for the backward direction using similar equations.

The results of both iterative searches are combined based on the computed depth values and screen-space errors. For each pixel, if the error measure is below a set threshold and they have similar depth, then a linear interpolation between the color of the source pixel with the lowest error and the color of that pixel projected to the other source image is used. Otherwise, the color of the pixel closest to the camera is used.

Multiview rendering

Besides increasing the frame rate of a real-time rendering pipeline by interpolating images, a small amount of research is also targeted at efficiently rendering a scene from multiple viewpoints. But, how does this relate to the previously mentioned techniques? Imagine each reprojected or warped image being a different viewpoint, then it becomes more obvious that reprojection and image warping are targeting similar problems. However, the previously mentioned techniques operate under different assumptions. The following methods are more closely related to the problem that is subject of the presented research.

J. Hasselgren and T. Akenine-Möller [17] researched a method to efficiently render up to sixteen views. The authors based their method on two observations. Texturing dominates the cost when considering memory accesses, and increased pixel shader complexity led to many applications where the performance is bound by pixel shader throughput. The novelty of this method is to rasterize the scene to all views simultaneously. The authors claim this will maximize the texture sampling cache hit-ratio since each view uses roughly similar texels. In turn, this should reduce the impact of texturing.

Rasterization of each triangle to all the views is done in normalized, perspective-correct barycentric coordinates. As such, the color of a point within a triangle can be expressed in one set of barycentric coordinates, and several constants that depend on the view. Combined with the deterministic nature of pixel shaders, the conclusion can be drawn that roughly sorted barycentric coordinates will lead to roughly sorted texture accesses with a maximized cache hit-ratio. The only requirement being that texture accesses are view independent.

The basic rasterization algorithm operates on scanlines within the bounds of a triangle, where the samples are generated ordered by view based on an efficiency measure. But Hasselgren and Akenine-Möller found that this can be optimized by using a tile-based approach. It is expected to improve the texture cache hit-ratio since the projection of a tile overlaps in two consecutively traversed views.

A second improvement to the basic rasterization algorithm is based on the observation that a pixel shader with the same input will result in the same color, which implies that the results of the pixel shader might be reused. The authors achieve this by storing and reusing color information in a cache. The total set of views to be generated is split up in sets such that the divergences in each set are small. From each set one view is picked to form the basis for the cache, for which the pixel shader is fully evaluated. The fragments in the remaining views are partly based on a full evaluation of the pixel shader, and partly on the cache.

A second method directed at multiview rendering was described by M. Halle [15]. A three-step pipeline is used to render several hundred views for a single static scene.

The first step consists of preprocessing the scene data. More specifically, this entails doing view independent lighting calculations for each vertex in the scene. Furthermore, the grid of cameras corresponding with the views are decomposed in rows. The benefit hereof is that each vertex, as seen from a row of cameras, only differs in its x-coordinate. As such, each vertex can be described by a five-dimensional vector in homogeneous screen space. Coincidentally, the last preprocess step is to transform the vertices to homogeneous screen space as seen from the left- and rightmost cameras in each row.

The next step in the multiview rendering pipeline is to decompose the geometry into line segments such that a segment aligns with a scanline in a final image. The author mentions that this process is comparable with conventional scan conversion, except that the line segment preserves its 3-dimensional continuity. Each line segment is stored such that its correspondence with the intersecting scanline is preserved.

The last step in the rendering pipeline consists of rasterization of the captured line segments into images that form epipolar planes and composition of the final images from the epipolar planes. One horizontal slice, or pixel row, of the volume created by stacking each final image on top of each other is called an epipolar plane. This also provides the method to compose the final images from the epipolar planes. Included in the process of rasterizing each line segment is any view dependent calculations, texturing and/or hidden surface removal.

2.2 Epipolar geometry

Part of the mission statement of the presented research, as given in Section 1.1, was to exploit coherence between images. To sustain this idea, a method relating a point in world space with its coinciding image locations in different views is needed. This relation is described by an idea referenced as epipolar geometry in literature [16].

Given two cameras looking at a 3-dimensional scene from two distinct positions and orientations. The 2D projections, as seen from the cameras, and their 3D counterparts have several geometric relations. Fortunately, the geometric relations lead to constraints between image points that relate to the same origin in the 3D scene. The view interpolation framework described in the following chapters uses a simplified case hereof.

For the general case regarding epipolar geometry, consider two cameras with center of projections O_l and O_r, and a point X in 3d space. The projection of X onto the image planes of the two cameras is given by X_l and X_r. Furthermore, imagine a virtual line segment L_v between O_l and O_r. The intersection of the two camera image planes and the line L_v gives the *epipoles* of both image planes.

Now, consider the line E_l going through O_l and X. Any point along E_l will always project to the point X_l onto the image plane corresponding with center of projection O_l. But line E_l projected onto the image plane corresponding with center of projection O_r results in a line which is called an *epipolar line*. Furthermore, the plane formed by O_l, O_r and X is named the *epipolar plane*. For a visual example setting, see Figure 2.1.

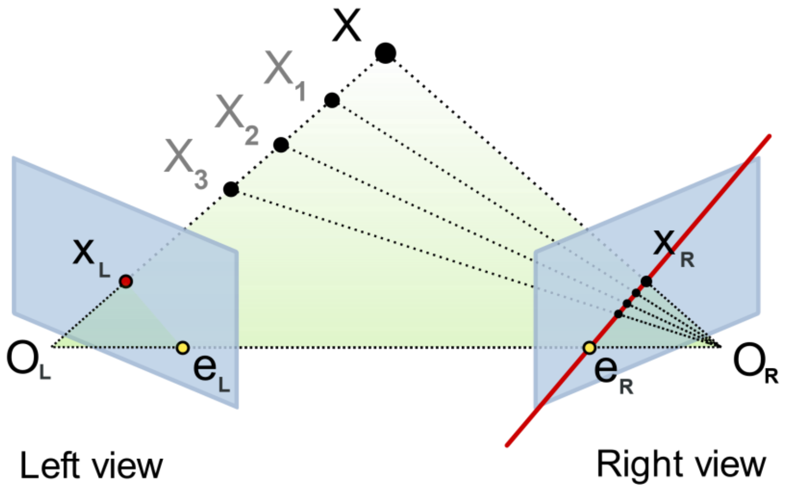

Figure 2.1. *Example setting showing an epipolar plane and epipolar line corresponding to the center of projections O_l and O_r, and point of interest X. The epipolar plane is formed by the triangle $O_L O_R X$. The epipolar line of point X in the right view is from e_R to X_R. (Source: https://en.wikipedia.org/wiki/Epipolar_geometry. Accessed: 27-06-2017)*

This notion is commonly used by computer stereo vision to recreate the depth from a pair of images. However, the same concepts can be adapted to create an image-based view interpolation algorithm. Since the camera configurations are controlled by the application, and thus known, and the depth of 3D points is also known, it is possible to warp image points from one image plane to another.

However, due to limitations in current generation graphics processors, it isn't possible to do this with unrestricted epipolar geometry. Instead, a simplified case is used with view interpolation. The camera placement is restricted such that the image planes coincide. This leads to the situation that the epipolar lines, and planes, are parallel to the horizontal axis of the image.

3 Epipolar based view interpolation

The research conducted to answer the research question stated in Chapter 1, has resulted in a framework which provides several different methods to generate a set of images corresponding with a set of given viewpoints. This framework is fundamentally built upon the theory of epipolar geometry, which is used to correlate pixels across different views. Over the course of the study, the sourcing and composition of the epipolar geometry led to three rendering pipelines.

This chapter provides a detailed insight into the inner workings of the three epipolar rendering pipelines. The remainder of the chapter is structured as follows. Section 3.1 introduces the restrictions on the positioning of the view corresponding cameras. Next, Section 3.2 provides a conceptual overview of the rendering framework. Then, Section 3.3 covers the internals of the basic algorithms, discussing three source view acquisition methods and the epipolar view interpolation method. Section 3.4 introduces several optimizations that were implemented to reduce the frame time. Lastly, Section 3.5 glosses over support for view dependent properties, particularly discussing support for specular highlights based on the (Blinn-)Phong reflection model.

3.1 Camera placement restrictions

Current generation graphics pipelines are only capable of efficiently rasterizing 3-dimensional geometry to 2d images. Unfortunately, view interpolation in the epipolar plane without any restrictions would require rasterization of volumetric data because the epipolar geometry could have any orientation in 3d space. To sidestep this inherent issue, the camera orientation and placement has been restricted for the presented view interpolation methods.

As part of the input, the epipolar rendering framework expects a set of cameras. Herein, each camera corresponds with a single view that a user wants to be rendered by the framework. Since the presented method relies on coherence between distinct views, there must be some amount of overlap between them. Otherwise, there would be no possibility to reuse image sections for generating one or more images. Furthermore, each camera must have identical internal parameters and orientation. In other words, all the cameras must have an identical viewing direction, field of view and near and far plane. The last two conditions are: the cameras must be positioned such that they all lie on a line segment with two neighboring cameras positioned equidistant to any other pair of cameras, and the line segment of cameras must be perpendicular to the viewing axis. See Figure 3.1 for a graphical depiction of the described situation.

Due to the camera placement restrictions, the epipolar geometry describing a point sampled from any scene will be on the same scanline in every given input camera. Thus, simplifying the capture process of epipolar geometry from a 3d problem to a 2d problem. Lastly, the restriction on spacing between cameras ensures that the points coinciding with the image location for the distinct cameras are also evenly spaced, assuring that the epipolar geometry can be captured in textures by means of interpolation.

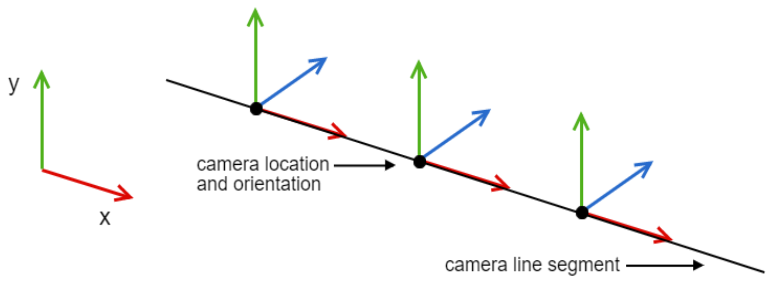

Figure 3.1. *Depiction of camera positioning on a line segment. All cameras have an identical view direction. Furthermore, the up vectors are perpendicular to the optical axes of the cameras and the line segment. Lastly, the distance between each pair of cameras is identical.*

3.2 Overview of rendering framework

In broad terms, the fulfillment of the goal as stated in Chapter 1 has led to a two-stage framework. A depiction of the developed render pipeline from input to output via the two processing stages is shown in Figure 3.2. The input to the rendering pipeline consists of a set of triangles in combination with a set of cameras describing each of the views expected as output and the inter-camera relation. Stage one processes the geometry data into a format usable for interpolation into multiple views. The incremental improvement of this process has resulted in three distinct rendering pipelines that are collectively referred to as source view acquisition. As shown in Figure 3.2, the specific pipelines are named: center view acquisition, dual view acquisition and hierarchical view acquisition. The internal specifics of the different source view acquisition rendering pipelines are discussed in Section 3.3.1.

The output of the source view acquisition stage is a set of rendered views tagged with information about the corresponding camera, and forms the input for the view interpolation stage and is considered the second stage. The goal of the view interpolation stage is generation of a relatively large set of views compared to the sparse set of rendered views given as input. The algorithm within this stage executes on the GPU and operates based on the theory of epipolar geometry to efficiently generate all output views. See Section 3.3.2 for a more detailed description.

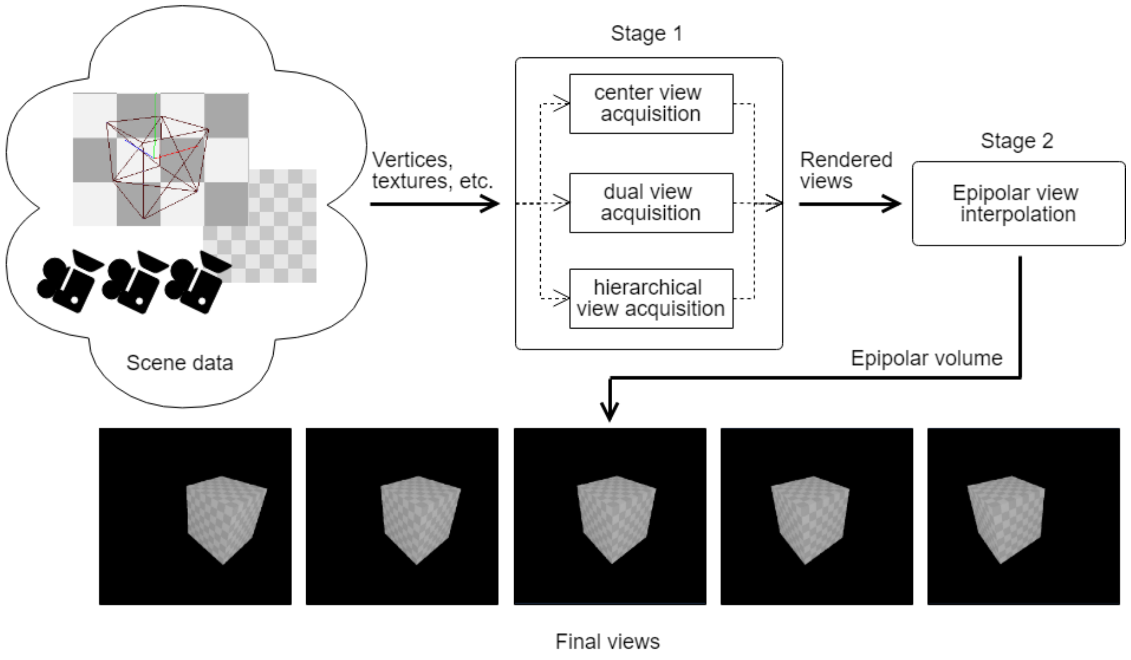

Figure 3.2. *High-level overview of the newly developed multiview rendering pipeline, including input and output. Additionally, a description of the inter-stage data is given.*

The output of the view interpolation, and thus the output of the rendering framework, is the set of generated views packed in a three-dimensional texture. A depiction of this is shown in Figure 3.3. The views are stacked one after another such that each vertical slice corresponds with an individual view. Incidentally, a row of pixels in the 3d volume is considered an epipolar slice in this configuration, whereby the epipolar view interpolation stage uses each epipolar slice as a distinct render target. This decomposition of epipolar volume in combination with the camera placement restrictions is one of the reasons to having the view interpolation algorithm run efficiently on a GPU.

Lastly, it should be noted that the framework provides the three-dimensional texture as final product instead of individual images. An additional copy step would increase computation time without any benefit since usage of 2d textures is comparable to the packed epipolar volume in terms of performance and programmer convenience.

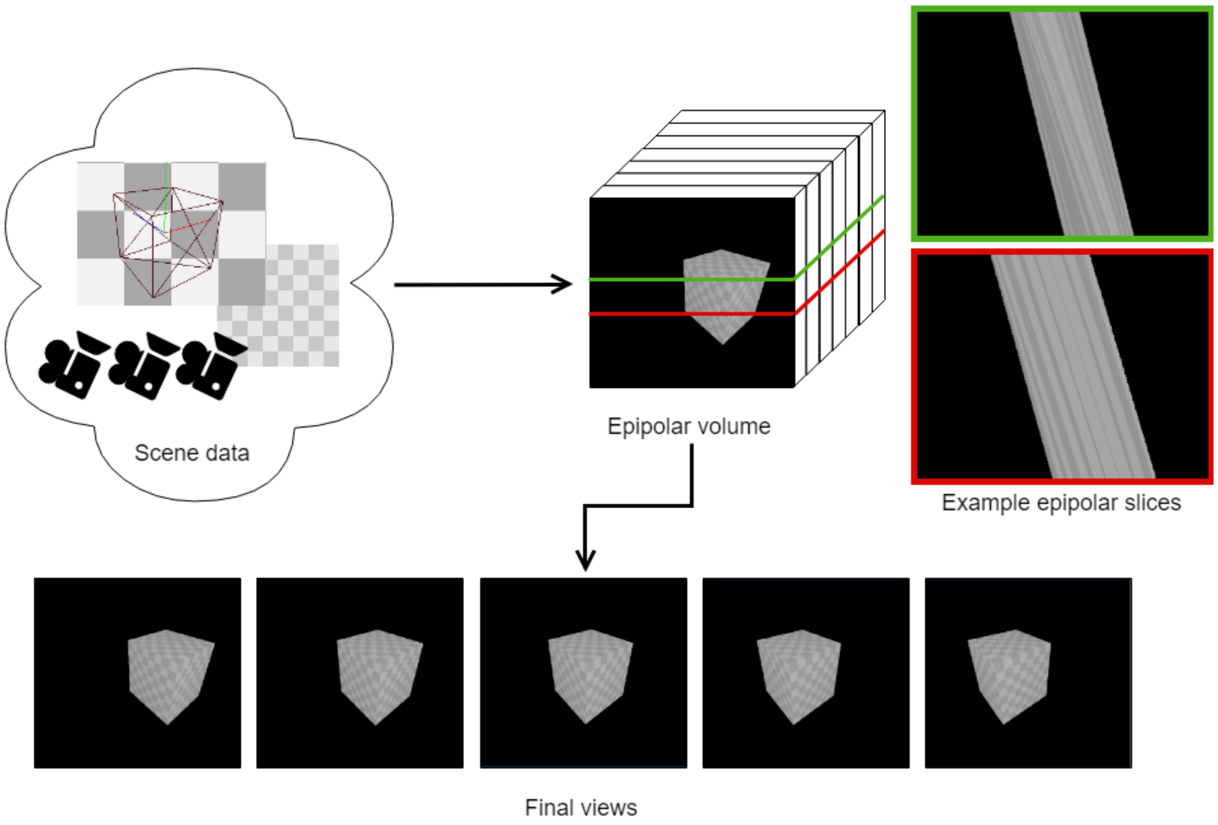

Figure 3.3. *Depiction of the epipolar volume and how it fits in the rendering framework. The implementation stores it as a 3d texture, whereby views are stacked one after another. As such, each row of pixels is considered an epipolar slice (two examples corresponding to the shown views are included in the figure). Lastly, it should be noted that the framework does not separate the epipolar volume into individual views. Usage of individual views can be done by reading vertical slices of the epipolar volume.*

3.3 Basic rendering framework

The general framework overview of the previous section provided an idea of the architecture. Moreover, it places the specific algorithmic details in a frame of reference. But, the generality of the previous section still leaves many options open regarding the details within the distinct stages. Filling this knowledge gap is the goal of the current section. Specifically, Section 3.3.1 disseminates the information on stage 1 of rendering pipeline. Discussing details of the different source view acquisition methods. Then, Section 3.3.2 will continue with the specifics of epipolar view interpolation.

Note that the descriptions in this document with respect to the distinct parts of the different epipolar rendering pipelines give an in-depth view of the inner workings, the necessary hardware interfacing, graphics API setup, and asset and resource housekeeping is omitted for conciseness and brevity. However, a concrete and working implementation of the ideas given in this chapter is used to present real-world results on the performance and image quality compared to the baseline. It is built upon the C++ programming language with OpenGL 4.3 as the graphics API.

3.3.1 Acquisition of source views

The epipolar geometry-based view interpolation to generate the final image set is done in image space. Unfortunately, the scene geometry is given as a set of triangles in 3-dimensional space. Thus, some processing needs to be done to transform the scene geometry to image space. This is stage 1 of the two-stage description given in Section 3.2. Unfortunately, a straightforward projection of 3-dimensional data to 2-dimensional images can lead to missing information due to occlusion. In the resulting image set this might manifest as holes appearing because traditional rendering pipelines discard geometry that is occluded by objects that appear in front.

Fortunately, capturing hidden geometry is a well-studied problem. A class of rendering techniques which use these type of methods is known as order-independent transparency [3, 12, 31]. One of the solutions used to solve it is known as depth peeling and provides a starting point for capturing the 3d scene geometry in 2d images, while retaining the necessary information.

Center view acquisition

As discussed in Section 3.2.1, the camera array is configured such that the cameras form a line. The first method for capturing the geometric information uses depth peeling and does so from the center view.

Depth peeling [12] is a multi-pass rendering technique, whereby the geometry is rendered n times to get n depth layers. During the rendering process, the geometry rasterizer produces a set of fragments for each pixel. Considering only depth peeling, the fragments consist of the image location and a corresponding depth value. The depth value in combination with standard depth testing gives the fragments closest to the camera. To get the second closest fragments it is necessary to disregard the closest fragments. Fortunately, this can be done with the modern graphics pipeline by means of discarding fragments in the fragment shader based on the depth value of the previously captured layer. This process is continued until all fragments are captured. See Code listing 3.1 and Code listing 3.2 for a mathematical overview of the depth peeling algorithm.

The concrete implementation of the algorithm stores the depth values in 10 distinct viewport-sized 2d textures, in which each depth value is stored as a 32-bit floating-point value to make sure no precision is lost. To capture the necessary geometry, each pass renders to an unused texture, whereby the standard depth testing is used to acquire the fragment closest to the camera. The actual depth peeling is implemented using a texture fetch of the previous depth layer and a conditional discard. Since each texture holds the depth values for a single depth layer, a maximum of 10 depth layers can be captured. However, this should be enough to do the epipolar view interpolation in combination with the extended depth offset which will be discussed in Section 3.4.1. To make sure the depth peeling stops doing extra passes when it has captured all the depth layers, every pass keeps track of the number of fragments that have been processed. This is done with OpenGL occlusion queries. When the counted fragments reach zero, the depth peeling is stopped because there are no more fragments left to process.

Input: Center camera projection parameters MVP. Furthermore, scene geometry G.
Output: A partially ordered set of textures V, such that V_i corresponds with depth layer i and where $0 \leq i \leq n$. $i = 0$ is the first depth layer and $i = n$ coincides with the farthest depth layer as seen from the center camera.

```
AcquireCenterSourceViews($MVP$, $G$):

$V$ := {$\emptyset$}

$F$ := RasterizeGeometry($MVP$, $G$)
$F_{v,0}$ := DetermineVisibleFragments($F$, {$\emptyset$})
$V_0$ := CalculateLighting($F_{v,0}$)
$V$ := $V \cup \{V_0\}$

$i$ := 1
while true:
        $F_{v,i}$ := DetermineVisibleFragments($F$, $F_{v,i-1}$)
        if $F_{v,i} = \{\emptyset\}$:
            break
        end if

        $V_i$ := CalculateLighting($F_{v,i}$)
        $V$ := $V \cup \{V_i\}$

        $i$ := $i$ + 1
end while

return $V$
```

Code listing 3.1. *High-level mathematical overview of the depth peeling algorithm. This refers to several helper functions that describe the standard rendering pipeline. They can be found in Code listing 3.2.*

The current implementation of the view interpolation framework captures the geometric information of a single depth layer using a method known as deferred rendering augmented with screen-space ambient occlusion (SSAO). The process to capture each depth layer consists of three render passes.

The first pass processes the geometry and stores the albedo, depth and normals in textures. The used formats for the textures are respectively 8-bit RGB, 32-bit floating-point depth component and 32-bit floating-point RGB. A simple optimization to reduce memory usage and bandwidth is to pack the different properties together. For example, the albedo and depth could be stored in a single 32-bit floating-point RGBA textures instead of separate textures. However, this was not done as it is highly dependent on the use case.

The SSAO render pass takes the depth texture as input and computes the ambient occlusion in screen-space. The ambient occlusion is stored in a 32-bit floating-point single channel texture. Refer to Section 6.1 in ShaderX 7 [11] for information on implementing SSAO as that is outside the scope of this report.

In traditional deferred renderers, the third render pass would compose the final color based on the ambient, diffuse and specular light components. However, for the presented purpose, only the albedo of the geometry, ambient contribution and ambient occlusion are gathered into a single color. The reason being that these components don't depend on the position of the camera.

For each depth layer, the textures containing the depth, normals and composed color are saved and passed onto the next stage. Any other textures are cleared and reused to reduce memory usage.

Note that the source view acquisition is not inherently reliant upon deferred rendering to get the ambient and diffuse color information. Deferred rendering is used because it is a renowned and widely used rendering technique, implemented and shipped by several triple-A games such as Gears of War 4, Battlefield 4, FIFA 17 and Grand Theft Auto V. Fortunately, the usage of the deferred rendering method is self-contained and can easily be replaced with other rendering techniques. Alternative methods are forward rendering, forward+ rendering, or a mix of rendering methods.

```
RasterizeGeometry(Projection parameters $MVP$, Geometry $G$):

returns a set of tuples $(x,y,z)$ such that the geometry $G$ projected in accordance
with the projection matrix MVP intersects with $(x,y,z)$, i.e. the geometry is
discretized into fragments.

DetermineVisibleFragments(Fragments $F$, Occluding Fragments $F_o$):

if $F_o = \{\emptyset\}$:
      return $F$
end if

$F_v$ := $\{(x,y,1) \mid (x,y,\_) \in F\}$
foreach fragment $(x,y,z) \in F$ do:
    if $F_v$ contains $(x,y,z_d)$ such that $z_d < z$ or
       $F_o$ contains $(x,y,z_o)$ such that $z_o > z$:
        continue
    end if

    $F_v$ := $(F_v \setminus \{(x,y,\_)\}) \cup \{(x,y,z)\}$
end foreach

return $F_v$

CalculateLighting(Fragments $F$):

$P$ := $\{\emptyset\}$
foreach fragment $(x,y,z) \in F$ do:
      $c$ := color of $(x,y,z)$, computed according to lighting model
      $p$ := $(x,y,z,c)$
      $P$ := $P \cup \{p\}$
end foreach

return $P$
```

Code listing 3.2. *Pseudocode for several support functions to complete the mathematical descriptions of the different pipelines. The functions shown in this Code listing are a description of the standard graphics pipeline.*

Runtime complexity

An indication of the runtime of an algorithm can be given by its time complexity. As a measure of the amount of work for a given input size, it provides a hint about the performance of an algorithm when compared to a baseline. However, it isn't perfect because certain aspects of an implemented algorithm are hidden or not considered. An example is the overhead incurred due to resource management.

For the center view acquisition algorithm, this is determined by derivation of the pseudocode for *AcquireCenterSourceViews* in Code listing 3.1. The time complexity of the referenced functions, shown in Code listing 3.2, are respectively shown in equation 3.1, 3.2 and 3.3 for *RasterizeGeometry*, *DetermineVisibleFragments* and *CalculateLighting*. In the equations, v refers to the number of vertices in the input and p is the number of pixels. Furthermore, l represents the time complexity of computing the color for a single fragment based on a lighting model. However, the lighting model is not considered as part of this thesis, and thus treated as a constant.

$$\text{Runtime RasterizeGeometry} = O(v) \tag{3.1}$$
$$\text{Runtime DetermineVisibleFragments} = O(p) \tag{3.2}$$
$$\text{Runtime CalculateLighting} = O(p) \tag{3.3}$$

For *AcquireCenterSourceViews*, the overall time complexity would then lead to equation 3.4. For the detailed derivation of the time complexity see Appendix B.2.

$$\text{Runtime AcquireCenterSourceViews} = O(v + p) \tag{3.4}$$

Algorithm pros and cons

The advantages of the described depth layer capturing method is that it is a simple and well-known method. Moreover, it operates independent of the rendering method, and as such it can work nicely with any deferred rendering pipeline. Thus, making it easily integrable in most current-generation graphics rendering engines.

Unfortunately, center view acquisition has several (severe) disadvantages making it undesirable to use in practice. Starting with the choice of depth peeling to capture the necessary information. A GPU works most efficiently if it can be constantly fed with work. Unfortunately, depth peeling requires the use of occlusion queries to detect when all depth layers are captured. This leads to stalls in the graphics processing pipeline because the results of the occlusion queries aren't immediately available, and thus, need to be waited on.

Another downside of center view acquisition is due to capturing the geometric information from a single camera. There is a limit to the area that is seen from a single camera. The camera array for which the views are expected as output cover a larger area. As such, some geometric information is missing and needs to be extrapolated or the final set of views need to be cropped.

A third disadvantage inherent to center view acquisition is missing information for geometry running parallel to the camera view direction. An example might be a wall on the center axis of the center camera standing parallel to its view direction. From a different camera, the missing geometry might be visible but isn't captured. Thus, the interpolation misses this information. Ultimately, this leads to holes in the final set of images.

Dual view acquisition

To counteract the two problems related to usage of a single camera as the source and the placement of that camera in the center, a slightly modified variant was developed. Instead of depth peeling from the center camera, dual view acquisition uses the same process as center view acquisition to capture source views. However, from both the leftmost and rightmost cameras. This ensures that the entire visible region is covered. Furthermore, the idea is that any missed geometry running parallel to the view direction of the leftmost camera is covered by the rightmost camera, and vice versa. The mathematical overview of the algorithm can be found in Code listing 3.3.

Input: Camera projection parameters MVP_l and MVP_r corresponding respectively with the leftmost and rightmost camera projection parameters. Furthermore, scene geometry G.
Output: Two partially ordered sets V_l and V_r, such that $V_{x,i}$ corresponds with the projection parameters for camera x, where $x \in \{ l, r \}$ and depth layer i, with $0 \leq i \leq n$. $i = 0$ is the frontmost depth layer and $i = n$ coincides with the farthest depth layer.

```
AcquireDualSourceViews($MVP_l$, $MVP_r$, $G$):

$V_l$ := AcquireDepthLayers($MVP_l$, $G$)
$V_r$ := AcquireDepthLayers($MVP_r$, $G$)

return ($V_l, V_r$)

AcquireDepthLayers(Projection parameters $MVP$, Geometry $G$):

$V$ := $\{\emptyset\}$

$F$ := RasterizeGeometry($MVP$, $G$)
$F_{v,0}$ := DetermineVisibleFragments($F$,$\{\emptyset\}$)
$V_0$ := CalculateLighting($F_v$)
$V$ := $V \cup \{V_0\}$

$i$ := 1
while true:
    $F_{v,i}$ := DetermineVisibleFragments($F$, $F_{v,i-1}$)
    if $F_{v,i} = \{\emptyset\}$:
        break
    end if

    $V_i$ := CalculateLighting($F_{v,i}$)
    $V$ := $V \cup \{V_i\}$

    $i$ := $i$ + 1
end while

return $V$
```

Code listing 3.3. *Mathematical overview of dual view acquisition. Any function not listed in this section and used by AcquireDepthLayers is identical to the function as used in the section on center view acquisition.*

The mentioned change won't increase the complexity classification of the overall algorithm since the previously determined complexity will only be multiplied with a constant. But, note that the measured runtime of dual view acquisition will increase noticeably due to capturing twice the number of depth layers. Furthermore, double the number of images need to be processed by the epipolar interpolation stage explained in subsection 3.3.2.

Hierarchical view acquisition

During the development of the presented framework, a third method for acquiring source views was developed. The reason is that edge cases exist in which the left- and rightmost view are not sufficient to capture all necessary geometry. Therefore, hierarchical view acquisition uses a heuristic to attempt to obtain a minimum set of views that capture the necessary information to feed the epipolar interpolation stage. Opposed to the operating assumption of the previous methods to capture all geometric information, which leads to a time complexity as a function of the depth complexity of a scene.

Hierarchical view acquisition tries to take advantage of the relatively small distance between the leftmost and rightmost cameras. The idea is to render the two views corresponding to the outermost cameras using standard deferred rendering, then check if the visual quality of the views between the two rendered views is good enough using a heuristic. The quality check doesn’t use view interpolation, instead a measure is used to predict if the quality of the output will be good enough when the acquired views are interpolated. For sufficient quality, the hierarchical view acquisition is done. Otherwise, the middle view is captured. This process is recursively done with both ranges of half the views, and is continued until the heuristic is satisfied with each branch. In addition to the image quality heuristic, a time-based rendering budget is used as stopping criterion to keep the time spent on acquiring views within reasonable bounds since the heuristic could theoretically continue to an extreme number of source views. The time to generate each view is tracked and accumulated. Views are continuously generated if the accumulated time is less than a user-specified threshold, and the heuristic based stop criterion isn’t fulfilled.

The threshold isn’t directly specified by a user because of the drawbacks. Even with identical input, the time to render a single view is not constant due to external factors. Moreover, the time it takes to render a single view can vary heavily from GPU to GPU since not every GPU has the same processing power. So, if the threshold is specified by the user as the maximum time, the value must be tweaked for every GPU the algorithm runs on. To overcome the user-unfriendliness and consistency issues, the maximum number of views is set by the user. Technically, the number of views can be directly used as the threshold in the proof-of-concept implementation since the threshold is static. However, in production implementations this value could be supplied by a dynamically adapted render budget. So, to make the system more integrable in existing render pipelines, a transformation is applied to get a time-based threshold. This is done by multiplication with the average time to render a view. In addition to cross-platform compatibility, the platform can change at runtime. For example, modern GPUs can modulate the available processing power based on its temperature. To take the changing conditions of the used platform into account, the frame time is tracked as the running average of the last five views.

The implementation used for the performance and quality assessment is implemented iteratively, instead of recursively. A mathematical overview of the iterative algorithm is shown in Code listing 3.4.

Input: An ordered set of camera projection parameters MVP_i, where $0 \leq i \leq n$ and it is the index of the camera. This set is ordered such that $i = 0$ corresponds with the leftmost camera parameters and $i = n$ corresponds with the rightmost camera parameters. Furthermore, quality threshold H_t, max renderbudget B and scene geometry G are given as input.
Output: A set of textures V, such that V_i corresponds with the i^{th} source view.

```
AcquireRangedSourceViews($MVP$, $G$, $H_t$, $B$):

$T$ := 0
RenderQueue := empty queue

$(V_0, T_0)$ := RenderView(0, $MVP_0$, $G$)
$(V_1, T_1)$ := RenderView($n$, $MVP_n$, $G$)
$V$ := $\{V_0\} \cup \{V_1\}$

RenderQueue.Push($(V_0, V_1)$)
$T$ := $T + T_0 + T_1$

while RenderQueue not empty and $T < B$:
        $(V_l, V_r)$ := RenderQueue.Pop()
        $Index$ :=

        $V_{vis}$ := ComputeVisibleSet($T_l$, $MVP_l$, $T_r$, $MVP_r$, $MVP_{Index}$)
        $H$ := CountHoles($V_{vis}$)
        if $H < H_t$:
                $(V_i, T_i)$ := RenderView($Index$, $MVP_{Index}$, $G$)
                $V$ := $V \cup \{V_i\}$

                RenderQueue.Push($(V_l, V_i)$)
                RenderQueue.Push($(V_i, V_r)$)
                $T$ := $T + T_i$
        end if
end while

return V

RenderView(ViewIndex $I$, Projection parameters $MVP$, Geometry $G$):

$Time_s$ := start time

$F$ := RasterizeGeometry($MVP$, $G$)
$F_v$ := DetermineVisibleFragments($F$, $\{\emptyset\}$)
$V$ := CalculateLighting($F_v$)

$Time_e$ := end time

return $(V, Time_e - Time_s)$
```

Code listing 3.4. Pseudocode for the hierarchical view acquisition pipeline. The shown functions refer to several helper functions. These are shown in *Code listing 3.2*. Furthermore, the algorithmic description of the erosion based image quality metric is given in *Code listing 3.5* and *Code listing 3.6*.

```
ComputeVisibleSet(Left view $V_l$, $MVP_l$, Right view $V_r$, $MVP_r$, $MVP_m$):

$V_{l,e}$ := ErodeView($V_l$, $MVP_r$, $MVP_l$)
$V_{r,e}$ := ErodeView($V_r$, $MVP_l$, $MVP_r$)

return $V_{l,e} \cup V_{r,e}$

ErodeView(Target view $V_t$, Projection parameters $MVP$, Projection parameters $MVP_u$):

$EdgeMap$ := $\{\emptyset\}$
foreach fragment $(x, y, z) \in V_t$ do:
        if $p$ corresponds with an edge of object in scene geometry:
                $EdgeMap$ := $EdgeMap \cup \{(x, y, z, 1)\}$
        else
                $EdgeMap$ := $EdgeMap \cup \{(x, y, z, 0)\}$
        end if
end foreach

$PrefixSumMap$ := $\{\emptyset\}$
foreach tuple $(x, y, z, mark) \in EdgeMap$ do:
        $PrefixSum$ := $\sum_{i=0}^{x} mark$, where $(i, y, z, mark) \in EdgeMap$
        $PrefixSumMap$ := $PrefixSumMap \cup \{(x, y, z, PrefixSum)\}$
end foreach

$ErodedView$ := $\{\emptyset\}$
foreach tuple $(x, y, z, prefix\_sum) \in PrefixSumMap$ do:
        $x_w$ := unproject $x$ to world space with parameters $MVP_u$
        $x_r$ := reproject $x$ to image space with parameters $MVP$

        if $(x_r, y, z, eroded\_sum) \in PrefixSumMap$ and $prefix\_sum \neq eroded\_sum$:
                $ErodedView$ := $ErodedView \cup \{(x, y, 1)\}$
        else
                $ErodedView$ := $ErodedView \cup \{(x, y, 0)\}$
        end if
end foreach

return $ErodedView$
```

Code listing 3.5. Pseudocode for the image-based erosion of geometry to compute the potentially visible geometry when looking from camera positions other than the provided MVP_l and MVP_r.

As mentioned, the hierarchical view acquisition algorithm uses a heuristic as part of the stopping criterion. The idea of the heuristic is to provide a guestimate of the image quality using less resources than performing actual interpolation since it is very hard to exactly measure the image quality. As such, it exploits the ability of the human visual system to compensate for some amount of noise. The idea is to use a filtered version of a captured view as a mask to compute the information which is lost if the source view capture process were to stop. The decision would be to decide if the lost information leads to an acceptable result.

Filtering the view to form the mask is like a problem known in literature as finding the potentially visible set (PVS) since the image space mask should give all geometry which might be visible. In general, computing the exact PVS is a very difficult problem. However, Décoret et al. [9] have shown that it is possible to efficiently compute a conservative estimate given that the viewpoint origin region, denoted as viewcell, is convex. The authors could modify the determination of a region's visibility from a viewer region to a point-to-point query.

The idea for computing the PVS is to erode the viewcell and area of which the visibility needs to be computed using the viewcell as the structuring element. This reduces the two regions to a pair of points. Furthermore, all occluders need to be eroded using the viewcell as the structuring element, which is done as a preprocessing step. The visibility of a region is determined by checking if any occluder intersects the line segment running between the pair of points acquired from the eroded viewcell and eroded geometry.

The used heuristic for conservative visibility estimation is based on the mentioned principles for PVS computation but operates in image-space and removes the need for the preprocessing step. The goal is to get an estimate for the visible geometry after subtraction of the already captured geometry. The computation entails a three-step process, with the depth map, a by-product of the deferred rendering process, expected as input. The output is a visibility map for the pixels which contain visible geometry after subtraction of the already captured geometry.

The first step consists of creating a discontinuity map with pixels set to 1 if they correspond with a discontinuity, and all other pixels set to 0. Before erosion of the objects can take place, the edges of the distinct objects need to be found. To this end, a discontinuity map is formed using a mean-based local threshold. In other words, the depth of a pixel is compared with the geometric mean of several surrounding pixels. In the current implementation, the mean is computed based on a 5-pixel wide one-dimensional window. This is justified because the epipolar planes are generated per scanline. As such, the visibility estimation, and thus also the thresholding, can be done per scanline. An advantage of a 1-dimensional window, compared to a 2d window, is the lower number of depth texture fetches per pixel.

The second step is to apply a prefix sum to the scanlines of the discontinuity map. This will result in ranges of pixels with the same value if no edge was detected in the previous step since the discontinuity map consists only of binary values.

With the processed discontinuity map, the objects in the depth map can be easily eroded to form a visibility map. Remember, the structuring element is equal to the viewcell. In this case it's equal to a line because the viewpoints all lie on a single line. As such, given a pixel p with image location (x, y), the erosion is done by reprojecting p to a pixel p' with image location (x', y). The reprojection entails an unprojection to view space, addition or subtraction of the distance between the leftmost and rightmost viewpoints, and a projection to image space. The addition or subtraction is dependent on the side of the view in the current range being eroded. To check if p is within the set of visible geometry, compare the value of p in the summed discontinuity map against the value of p'. If the values differ, then an edge of an object is crossed and thus is pixel p marked as visible.

The erosion leaves holes in the visibility map where the captured views possibly miss information for all cameras in between the cameras that belong to the captured views. The holes are pixelwise counted, and the decision of an acceptable result is done against a user supplied threshold. The user sets the threshold as a percentage of the total number of pixels in a texture to make the heuristic more robust against textures size differences. The mathematical equivalent of counting the pixels is shown in Code listing 3.6.

```
CountHoles(Eroded view $V_e$):

Count := 0
foreach pixel $(x, y, mark) \in V_e$ do:
        if $mark \neq 1$:
                Count := Count + 1
        end if
end foreach

return Count
```

Code listing 3.6. Pseudocode for counting the number of holes in an eroded image. The image quality decision-rule is implemented as a comparison of the count returned by this function and a user-provided threshold.

Runtime complexity

Determining the time complexity of the hierarchical view acquisition algorithm is a little more involved than the previous two cases. The basis for the time complexity analysis is the pseudocode shown in Code listing 3.4. For the previously determined time complexity of *RasterizeGeometry*, *DetermineVisibleFragments* and *CalculateLighting* refer to equations 3.1, 3.2 and 3.3. Additionally, *AcquireHierarchicalSourceViews* references the functions *RenderView*, *CountHoles, ErodeView* and *ComputeVisibilitySet*. Their respective time complexities are shown in equations 3.5, 3.6, 3.7 and 3.8. Herein, v refers to the amount of geometry contained in a scene and p is the number of pixels contained in an image.

$$\text{Runtime RenderView} = O(v + p) \tag{3.5}$$

$$\text{Runtime CountHoles} = O(p) \tag{3.6}$$

$$\text{Runtime ErodeView} = O(p^2 + p) \tag{3.7}$$

$$\text{Runtime ComputeVisibilitySet} = O(p^2 + 2 * p) \tag{3.8}$$

The combined time complexity of the helper functions and the function *AcquireHierarchicalSourceViews* results in equation 3.9. Herein, two additional variables are referenced. T refers to the time needed to render a single view. This is the time reported by *RenderView*. B refers to the render budget and is given as input to the algorithm. The time complexity is dependent upon these variables since the number of executions of the while-loop is parameterized by these variables. For the full derivation of the time complexity see Appendix B.3.

$$\text{Runtime AcquireHierarchicalSourceViews} =$$
$$2 * O(v + p) + O(1) + 9 + \frac{T}{B} * (O(p^2 + 2p) + O(v + p) + O(p) + 3 * O(1) + 11) + 1 \tag{3.9}$$

Algorithm pros and cons

One advantage which favors usage of the hierarchical view generation algorithm is the ability to target a performance constraint by limiting the number of views that may be generated, or the ability to target image quality by leaning towards a higher image quality threshold and limit the generated views at infinity.

A second advantage is the modularity regarding the quality measure. It is easy to swap in different quality measures. As an example, and to possibly improve performance, a second image quality heuristic was implemented. The second heuristic performed a pixel wise reprojection of the captured images in a range of views to its center image. If the resulting image has too many holes, then the algorithm would continue; otherwise the recursion stops. However, more clever heuristics may improve the performance further, and can be the subject of future research projects.

Unfortunately, the hierarchical view generation algorithm also has several undesired properties. Starting with the interdependency between the generated views. The decision to continue generating source views is dependent upon already generated views. As such, the CPU can’t generate new graphics rendering commands and send them to the GPU without waiting for previous results. Like the depth peeling based methods, this leads to stalls in the graphics pipeline and to underutilization of the GPU.

A second disadvantage is the usage of a conservative visibility estimation. The conservative nature of the visibility estimation can lead to extra, and unneeded, generation of source views. This may benefit the quality of the resulting images because more information is captured, but there is also a negative impact on the runtime as the epipolar rendering stage must perform more work.

The last undesired property is the potential for erratic behavior. Although the hierarchical view generation algorithm is limited by a render budget, the visibility estimation can still result in large fluctuations in the number of generated views depending on slight changes in the camera array configuration or the visible geometry. An extreme example of visible geometry that might lead to erratic behaviour is looking at a single curtain versus looking at the leaves of a bush.

3.3.2 Epipolar view interpolation

The source view acquisition methods, described in the previous section, were the first step towards the goal stated in Section 1.1. They provide the necessary geometric information in image form.

This section documents the specific details of Stage 2 from the overview given in Section 3.2. Explicitly, it presents the multi-pass, image-space interpolation algorithm that transforms the image-based geometry information to the final set of images by means of interpolation. Conceptually, the presented algorithm operates in epipolar space. In this context, the epipolar space is a 3-dimensional space meaning that every point can be characterized by three coordinates. Additionally, each point can have additional attributes such as color or depth. As was briefly introduced in Section 2.2, epipolar space provides a structured link of a geometry sample across different views given the previously described camera placement constraints. The idea behind the interpolation algorithm is to transform each geometry sample into the epipolar space equivalent, and let the GPU interpolate the sample to each view.

Each pass of the interpolation algorithm takes one source view as input and transforms each pixel into a line segment in epipolar space. If GPUs can rasterize 3d geometry to a 3d texture, it would be possible to directly generate the epipolar representation of the scene geometry. Unfortunately, this is not (yet) possible. However, due to the restrictions on the camera placement, the line segment in epipolar space stays within the same scanline. As such, it is possible to decompose the 3-dimensional epipolar volume of all views into a set of distinct 2-dimensional planes. Each plane links one row of pixels in the source view to its corresponding location in the epipolar volume.

Transforming a source view pixel to a line segment in epipolar space entails calculating the coordinates for the endpoints and generating a vertex for each endpoint. To transfer the color of the source pixel to the epipolar line segment, set the color and depth of the endpoints to be identical to the source pixel. This can then be handed to the rasterizer to be placed into the correct epipolar plane.

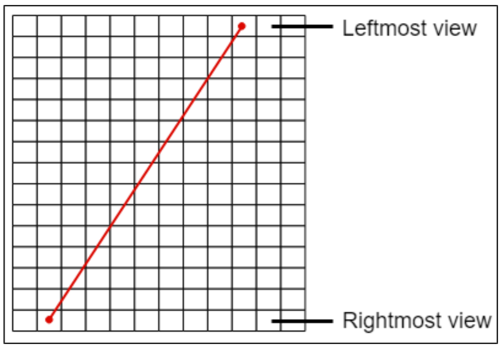

Figure 3.4. *Example diagram of a single epipolar plane. Each row of pixels corresponds with a pixel row for a different view. The red line signifies one source pixel that has been transformed to its counterpart in epipolar space.*

Calculating the endpoint coordinates of a line, for one source pixel, amounts to finding the coordinates (x, y) of both vertices that describe a line since the epipolar space has been reduced to a plane.

Looking at Figure 3.4, computing the x-coordinate of one endpoint equates to calculating the x-coordinate in the final views, and can be done in several manners. The proof-of-concept implementation settled for unprojection of the source pixel coordinates to view space, adding the disparity only to the x-coordinate, and reprojecting the modified coordinates to image space. The reason for choosing this method was that world space coordinates are needed for adding specularity to the final images. Section 3.5 contains more information on support for specularity. Unprojection is done by multiplication of the source pixel coordinates with the inverse of the projection matrix followed by a perspective division. Reprojection entails a multiplication of the modified view space coordinates by the projection matrix again followed by a perspective division. The disparity is equal to the distance from the position of the source view camera to the position of the target camera, which is either the camera corresponding with the leftmost or rightmost view. For mathematical equations to compute the disparity see the pseudocode for *DrawDepthLayer* in Code listing 3.9.

What remains is calculating the y-coordinate. Looking to the example in Figure 3.4, a line must always start in the center of the bottom row of pixels and end in the center of the top row of pixels since the algorithm wants to know where each source pixel lands in every view. By definition, the image space y-coordinate is fixed to 0.5 for the start vertex, and $0.5 + n - 1$ for the end vertex, in a situation with n views and where the pixel centers fall on half integers. Moreover, the y-coordinate is identical for each line in epipolar space.

Note that the described epipolar line generation algorithm is sufficient for the basic view interpolation algorithm. However, this does not allow for view dependent properties such as specularity. The addition discussed in Section 3.5 requires the ability to compute properties on arbitrary locations along a line in epipolar space. This is possible by generating multiple connected line segments, where the end of a line segment, and conjointly the beginning of the next line segment, is determined by a vertex. The coordinates of the additional vertices could be computed as a linear interpolation between the computed end points in epipolar space. However, the world space coordinates are needed for view dependent properties. As such, the needed coordinates are computed as a linear interpolation of the world space coordinates. Any attributes such as color, depth or specular intensity are copied from the source pixel.

However, doing a verbatim transformation of each source pixel to a epipolar line is not enough. A method is needed to resolve crossing epipolar lines. The chosen method is to superimpose the epipolar lines and let the depth of a fragment decide which is kept in the composed result. This method works because only fragments closest to the camera are visible. As such, this resolution method can be implemented by means of depth testing.

The same principle is also used to combine the epipolar planes from different views. For example, each depth layer for the center view acquisition method has several epipolar planes that occupy the same space in the epipolar volume. Again, the correct pixels are resolved by depth testing. An advantage of depth testing as conflict resolution is nice integration with the conceptual interpolation algorithm. But, more importantly, depth testing is commonly used, and thus implemented as a hardware operation.

Unfortunately, depth testing by itself does not deliver the best image quality. For the human eye, the quality difference between the composed views using only depth testing and views generated using the baseline algorithm is barely noticeable. However, the described composition method suffers from the finite precision of GPUs. Since the transformation and resolution of multiple epipolar planes cannot be done in one operation, the intermediate result needs to be stored. Unfortunately, this leads to precision loss and means that geometry which originally had different depths get treated as having an identical depth. To work around this problem, the proof-of-concept implementation adds a depth bias to the generated quads in epipolar space. The depth bias is a fixed value multiplied by the distance from the camera corresponding to the source view. The distance is expressed in the interval $[0; 1]$, where 1 is equal to the distance between the camera used for the source view and the camera farthest away from the source view camera.

Besides problems with precision, the proof-of-concept implementation has uncovered some more shortcomings with the conceptual interpolation algorithm. Firstly, the lines were exchanged with narrow quads composed of two triangles. Ideally, the quads have a width equal to the width of one pixel. However, in practice, the quads needed to be slightly wider than one pixel to compensate for the difference in rasterization between lines and triangles. More specifically, OpenGL has different rules for triangles covering a pixel versus coverage of a pixel by lines. The reason for generating quads is that one line per source view pixel results in a very large number of primitives that need to be rasterized. As such, one pursued improvement was to coalesce several adjacent lines in a epipolar plane into a single quad. More details about this optimization are given in Section 3.4.3.

In addition, the replacement of lines with quads resulted in a welcome side effect. Small, pixel sized holes are filled by the quads. It is possible for the source views to not provide the necessary information that can be seen from other views. The underlying reason is that the source views are generated by sampling the geometry data. As such, small triangle sections may be missed. Fortunately, it is very likely that the color information of the missed areas is like surrounding geometry. As such, slightly overestimating the area that is covered by a quad in epipolar space contributes to a higher image quality.

Note that the order of the emitted vertices became important when the lines were exchanged with quads. See Figure 3.5 for an example quad in an a epipolar plane, which is split in the middle. The order in which the vertices are generated is $0, 1, 2, 3, 4, 5$. The order for the vertices in the four triangles is $(0, 1,2), (1,3,2), (2,3,4), (3,5,4)$. Every second triangle has a pair of vertices that should be swapped. This is reflected by the pseudocode for *DrawDepthLayer* in Code listing 3.9.

The proof-of-concept implementation uses a geometry shader for the generation of the vertices. The order in which the vertices are computed and emitted is from vertex 0 to vertex 5. The triangle assembly including the vertex swap in every second triangle are automatically done by the rasterizer.

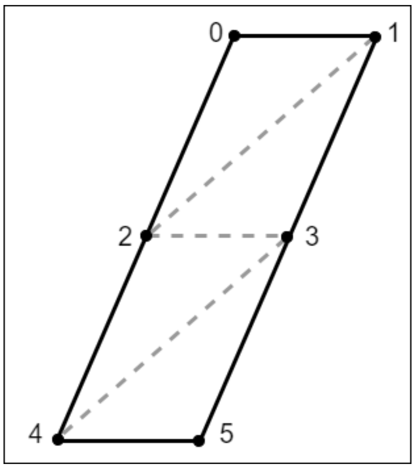

Figure 3.5. *Example of a quad in an epipolar plane. The quad is split in the middle (between vertices 2 and 3). Additionally, the triangles constituting the quad are visualized.*

A second issue that led to a change of the conceptual algorithm is a consequence of how current generation GPUs operate. In theory the source views can directly be transformed into epipolar geometry. In practice, this leads to underutilization of the GPU since the source views can contain pixels which don’t contribute to the final views. An example situation would be a pixel that is not covered by geometry. The branching to filter the offending pixels leads to execution divergence. Branching on GPUs is implemented as computing all branches and masking out the incorrect answer. So, execution divergence would have a small impact if the mapping of a pixel to epipolar geometry was relatively simple. Unfortunately, this is not the case.

This performance penalty was solved by adding an extra pass to filter out non-contributing pixels and storing contributing pixels in an intermediate buffer. The proof-of-concept implementation captures meaningful pixels using Transform Feedback [6, 27, 39]. The content of the intermediate buffer is then used as the input for the generation of geometry in epipolar space. This solves the issue because the branches in the intermediate pass are relatively short.

The last change made to the conceptual algorithm is a consequence of the interpretation of OpenGL rasterization rules by the different GPU manufacturers. The OpenGL specification contains several rules which dictate the pixels that are covered by a triangle. Unfortunately, GPU manufacturers have some freedom in the implementation, and this leads to differing outcomes on different GPUs using identical code. An example situation of the encountered problem can be seen in Figure 3.6 (a) and (b). Different vendors pick different triangle edges of which fragments are discarded to account for shared edges between triangles. If this happens in the epipolar plane, then a single view is missing from the result.

The chosen solution for this problem was to add two additional virtual views. In other words, the epipolar coordinates where computed such that two additional views are added, but the endpoints lie outside the rendering viewport. So, these views will be clipped out by the rasterizer. As such, it does not matter which triangle edge has discarded fragments since they won't be visible.

Note that the changes regarding generating quads instead of lines and the filter-pass for inactive pixels are reflected in Code listing 3.7, Code listing 3.8 and Code listing 3.9. Additionally, triangle assembly including vertex swapping is also included.

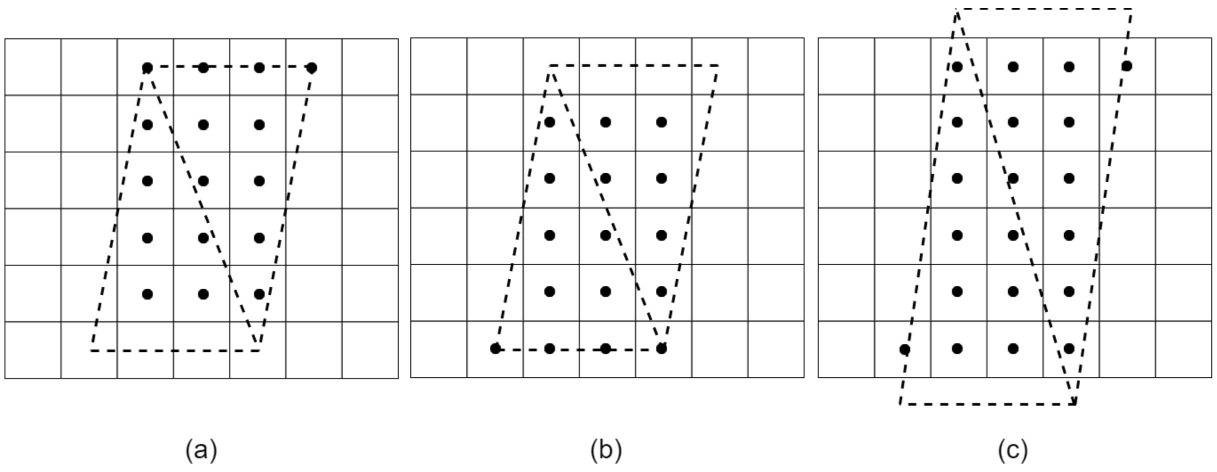

Figure 3.6. *Different GPU hardware vendors wield different rules to handle overlapping triangle edges. This leads to situations such as shown in (a) or (b), where the pixels covered by different triangle edges are discarded. As such, computing the theoretically ideal epipolar coordinates leads to different behavior. For geometry in the epipolar plane this results in a single missing view. The work around is to add to additional "virtual" views, as shown in (c).*

Input: A set of n source views V and corresponding projection parameters MVP_v for each of the source views, indexed such that V_i is the i^{th} source view, and $MVP_{v,i}$ are its corresponding projection parameters. A single source view is either a depth layer, in case of center view acquisition and dual view acquisition, or an actual view, in case of hierarchical view acquisition. Furthermore, a pair of camera projection parameters MVP_l, corresponding with the leftmost camera, and MVP_r corresponding with the rightmost camera, are given as input.
Output: A partially ordered set $EpipolarVolume$ containing tuples of the form (x, y, z, d, c). This gives the color c at location (x, y, z) with depth value d. The pixels of a single view view can be retrieved by choosing a fixed z-coordinate and varying the x and y. $z = 0$ gives the view corresponding with the leftmost view and $z = k$ gives the rightmost view, where k is at most the number of views.

```
AcquireInterpolatedViews(Source views $V$, Projection parameters $MVP_v$; $MVP_l$;
$MVP_r$, $n$):

$EpipolarVolume$ := { $\emptyset$ }
for $i = 1$ to $n$:
    $ActivePixels$ := CaptureActivePixels($v_i$, $MVP_{v,i}$)
    $ViewFragments$ := DrawDepthLayer($ActivePixels$, $MVP_{v,i}$, $MVP_l$, $MVP_r$, $n$)

    foreach tuple $(x,y,z,d,c) \in ViewFragments$:
        if $EpipolarVolume$ contains $(x,y,z,d_{ev},\_)$ such that $d < d_{ev}$:
            $EpipolarVolume$ := $EpipolarVolume \setminus \{ (x,y,z,d_{ev},\_) \}$
            $EpipolarVolume$ := $EpipolarVolume \cup \{ (x,y,z,d,c) \}$
        end if
    end foreach
end for

return $EpipolarVolume$
```

Code listing 3.7. *Mathematical description of the core-loop in the epipolar based view interpolation algorithm. Several functions are referenced from Code listing 3.8 and Code listing 3.9.*

```
CaptureActivePixels(Source view $v$, Projection parameters $MVP$):

$ActivePixels$ := { $\emptyset$ }
foreach fragment $(x,y,z,c) \in v$:
    if $z = 0$:
        continue
    end if

    $Stride$ := 1
    $ActivePixels$ := $ActivePixels \in \{ (x,y,z,c,Stride) \}$
end foreach

return $ActivePixels$

EmitVertex($Coords$, Projection parameters $MVP$, $\alpha$, $n$, Color $c$):

$x_{epi}$ := $x$-coordinate from $Coords$ after projection to clip space using $MVP$
$y_{epi}$ := $\frac{((1-\alpha)*0.5+\alpha*(n-0.5))}{n} * 2 - 1$
$z_{epi}$ := $z$-coordinate from $Coords$

return $(x_{epi}, y_{epi}, z_{epi}, c)$
```

Code listing 3.8. Mathematical description of several support functions used by the view interpolation algorithm. Note that the computation of y_{epi} assumes that pixel centers are positioned at half-integer locations.

```
DrawDepthLayer($ActivePixels$, Projection parameters $MVP_{v,i}$; $MVP_l$; $MVP_r$, $n$):

$CameraPosition_l$ := Get position from $MVP_l$
$CameraPosition_r$ := Get position from $MVP_r$
$CameraPosition_{v,i}$ := Get position from $MVP_{v,i}$

$MaxViewOffset$ := $|| CameraPosition_r - CameraPosition_l ||$
$ViewDistance$ := $\frac{||CameraPosition_{v,i} - CameraPosition_l||}{MaxViewOffset}$

$Disparity_l$ := $ViewDistance * MaxViewOffset$
$Disparity_r$ := $(1 - ViewDistance) * MaxViewOffset$

$EpipolarTriangles$ := $\{\emptyset\}$
foreach fragment $(x, y, z, c_{l,d}, Stride) \in ActivePixels$:

    // Compute epipolar plane end points corresponding to current fragment
    $Coords_{l,v}$ := Unproject $(x, y, z)$ to view space using $MVP_{v,i}$
    $StartCoords_{l,v}$ := $Coords_{l,v} + (Disparity_l, 0, 0)$
    $EndCoords_{l,v}$ := $Coords_{l,v} - (Disparity_r, 0, 0)$

    Get color $c_{r,d}$ proper for $x + Stride$
    $Coords_{r,v}$ := Get coordinates proper for $x + Stride$ and
                  unproject to view space using $MVP_{v,i}$
    $StartCoords_{r,v}$ := $Coords_{r,v} + (Disparity_l, 0, 0)$
    $EndCoords_{r,v}$ := $Coords_{r,v} + (Disparity_r, 0, 0)$

    // Loop over geometry splits and emit corresponding vertices
    $GeometrySplits$ := $[0, 1]$
    $Vertices$ := $[\,]$
    $VertexCount$ := $0$
    for $Index$ := 0 to length($GeometrySplits$):
        $\alpha$ := $GeometrySplits[Index]$
        $Coords_{l,v}$ := $(1 - \alpha) * StartCoords_{l,v} + \alpha * EndCoords_{l,v}$
        $Coords_{r,v}$ := $(1 - \alpha) * StartCoords_{r,v} + \alpha * EndCoords_{r,v}$

        $ModelMatrix$ := $I$
        $ViewMatrix$ := Get view matrix from $MVP_l$ and modify such that position
                        corresponds with
                        $CameraPosition_l + \alpha * (CameraPosition_r - CameraPosition_l)$
        $ProjectionMatrix$ := Get projection matrix from $MVP_l$
        $MVP$ := $ProjectionMatrix * ViewMatrix * ModelMatrix$

        $Vertices[VertexCount]$ := EmitVertex($Coords_{l,v}$, $MVP$, $\alpha$, $n$, $c_{l,d}$)
        $VertexCount$ := $VertexCount + 1$
        $Vertices[VertexCount]$ := EmitVertex($Coords_{r,v}$, $MVP$, $\alpha$, $n$, $c_{r,d}$)
        $VertexCount$ := $VertexCount + 1$
    end for

    // Loop over vertices to assemble create triangles
    for $TriWindowStart$ := 0 to $VertexCount - 3$ step 2:
        $EpipolarTriangles$ := $EpipolarTriangles \cup$
 $\{(Vertices[TriWindowStart], Vertices[TriWindowStart + 1], Vertices[TriWindowStart + 2])\}$
        $EpipolarTriangles$ := $EpipolarTriangles \cup \{(Vertices[TriWindowStart +$
 $1], Vertices[TriWindowStart + 3], Vertices[TriWindowStart + 2]\}$
    end for
end foreach

$Fragments$ := RasterizeGeometry($I$, $EpipolarTriangles$)
return $Fragments$
```

Code listing 3.9. DrawDepthLayer transforms the geometry, gathered by the previously described view acquisition stages, into epipolar geometry.

However, a conceptual description of an algorithm gives no information about the mapping to actual hardware. To aid the reproducibility of the presented work, several important implementation details are given. The interpolation algorithm is implemented as two render passes. One corresponds with the theoretical *CaptureActivePixels* function, and one implements the *DrawDepthLayer* function. The order of operations is such that the active pixels in all source views are captured and stored in intermediate buffers, before all intermediate buffers are pushed through the interpolation pass corresponding with *DrawDepthLayer*. One advantage of the chosen order is less state changes than interleaving the *CaptureActivePixels* and *DrawDepthLayer* pipelines, which is relatively costly. However, a major downside is increased memory cost which leads to increased memory bandwidth usage. Additionally, a tight upper bound on the memory usage is not known beforehand because the number of active pixels cannot easily be predicted. Thus, the maximum possible memory usage needs to be preallocated. This is possible since discrete GPUs mostly have enough memory available. However, if not enough memory is available, the two render pipelines can be interleaved to reduce the memory usage.

The render pass associated with *CaptureActivePixels* runs once for each source view. It consists only of a vertex and geometry shader. The input of the vertex shader is one vertex, without any attributes, per pixel in the given source view. Associating a vertex to a source view pixel is done based on gl_VertexID[2]. The second stage of the render pipeline consist of a geometry shader. This shader filters the inactive pixels by checking the depth value. The attributes emitted the originating image location, and corresponding depth value. Additionally, the stride is emitted. This attribute is specific to the primitive coalescing optimization discussed in Section 3.4.3. The attributes are directed to a bound buffer using Transform Feedback.

The stage associated with the theoretical *DrawDepthLayer* function, is a multi-pass process whereby the contents of one intermediate buffer is processed per pass. The graphics pipeline is configured as a graphics pipeline consisting of a passthrough vertex shader, a geometry shader and a fragment shader. The vertex shader reads the input from the intermediate buffer and passes that to the geometry shader without any modifications. The geometry shader transforms its input from a source pixel into one or several connected quads that reside in epipolar space. This is done by generating vertices of triangles that form the quad(s). To direct a quad to its corresponding epipolar plane, layered rendering[3] is used. Explicitly, this means that an invocation of the geometry shader sets the gl_Layer variable to the source pixel y-coordinate. This ensures that a quad is directed to the correct epipolar plane. A side effect is that all epipolar planes must be stored in a 3d texture that is attached as a layered render target. The overlapping geometry resolution is implemented by activating depth testing and configuring the depth function to be GL_LESS. As such, the hardware automatically resolves any conflicts with the emitted geometry.

The last stage consists of a relatively simple fragment shader which writes the interpolated color to the output.

[2] For more information on gl_VertexID, see https://www.khronos.org/registry/OpenGL-Refpages/gl4/html/gl_VertexID.xhtml.

[3] For more information on Layered rendering, see https://www.khronos.org/opengl/wiki/Geometry_Shader#Layered_rendering. More information on layered framebuffers, necessary for layered rendering, can be found here: https://www.khronos.org/opengl/wiki/Framebuffer_Object#Layered_Images.

Note that, technically, it would be possible to move some of the computations from the geometry shader to the vertex shader, but the downside is that more information has to pass through slower memory, whereas calculating everything in the geometry shader keeps the information in registers. As such, it is more performant to keep the calculations in the geometry shader.

Runtime complexity

The theoretical performance analysis of the epipolar space based view interpolation algorithm is based on the pseudocode shown in Code listing 3.7, Code listing 3.8 and Code listing 3.9. Note that *DrawDepthLayer* references *RasterizeGeometry*. The time complexity of this function was earlier determined to be $O(v)$. The respective time complexities of *EmitVertex*, *CaptureActivePixels* and *DrawDepthLayer* were determined to be as shown in equations 3.10, 3.11 and 3.12.

$$\text{Runtime EmitVertex} = O(1) \tag{3.10}$$

$$\text{Runtime CaptureActivePixels} = O(p) \tag{3.11}$$

$$\text{Runtime DrawDepthLayer} = O(p) \tag{3.12}$$

The pseudocode showing the main loop of the view interpolation algorithm referred to the function *AcquireInterpolatedViews* in Code listing 3.7. Its time complexity is shown in equation 3.13, where n is the number of source views given by the source view acquisition algorithms and p is the pixel count in a single view. For source view acquisition based on depth peeling a constant number of source views are given. In such cases, the time complexity for view interpolation simplifies to $O(p)$. See Appendix B.4 for a more detailed derivation of the time complexity.

$$\text{Runtime AcquireInterpolatedViews} = O(n^2 p) \tag{3.13}$$

Algorithm pros and cons

One advantage of the epipolar view interpolation pipeline is the ability to generate a large set of views by interpolation from a small set of correctly chosen source images.

Another advantage is, the interpolation has no need for extra hole filling logic. To a certain extent small holes are covered since the area occupied by a pixel is overestimated. This is done to counteract some corner cases with the implementations for the OpenGL rasterization rules by the different hardware vendors.

The third advantage is that epipolar based view interpolation operates in image-space. As such, the algorithm scales independently of the scene complexity. Moreover, it can easily integrate into existing rendering pipelines as an extra pass and the source view acquisition stages can evidently be based on commonly used, and existing technology.

The last, discussed, advantage is having no dependency on a predefined scene data format. As such, this algorithm is independent of currently used digital content creation pipelines and model data.

But, the presented interpolation algorithm has its share of disadvantages. One disadvantage is the reliance on an intermediate pass to reduce frame time. The data from the source view acquisition stage can contain superflouos data which leads to execution divergence when interpolating. More concretely, some shader processing units remain idle while other perform computations. This leads to underutilization of the available processing power and is solved in an intermediate pass by filtering out pixels which don’t contribute to the result. A side effect of the intermediate pass is the higher memory usage since the data transfer from the intermediate pass to the interpolation pass needs to be passed through some form of a buffer.

Another disadvantage of the epipolar based view interpolation is its memory demand. It needs a depth texture which has equal dimensions as the combined set of interpolated views since the rasterization of epipolar geometry needs to have depth testing activated.

The last, mentioned, disadvantage is the extra effort involved to add support for view dependent properties. As will be shown in Section 3.5, it is possible to add support for view dependent properties. However, existing reflection models must be modified to fit the presented interpolation method, while not leading to an exorbitant increase of the frame time. Furthermore, the proposed method of adding view dependent properties by piecewise approximation might not fit every view dependent property.

3.4 Optimizations

The rendering framework described in the previous section can generate the required views based on interpolation. However, several amendments can be made to improve on the basic rendering pipelines. During the development of the framework, several possible optimizations have been explored and were included in the final implementation.

3.4.1 Extended depth offset

The previous section discussed the process to capture scene geometry by depth peeling, and the interpolation to the different views. However, it was not mentioned that captured geometry not in the first depth layer may be occluded. As such, it may not be seen from any viewpoint. Thus, leading to inefficient and superfluous use of computational resources. More specifically, extended depth offset focusses on small units of geometry in very close proximity to its occluder.

Until now, the depth layer extraction only considers the unmodified depth of captured geometry. Specifically, consider a pixel p_d which corresponds with a fragment at image location (x, y) in depth layer d and pixel p_{d+1} at the same location (x, y) but in depth layer $d+1$. Pixel p_{d+1} only captures the fragment closest to the viewer such that $depth(p_{d+1}) > depth(p_d)$. To diminish the impact of capturing always hidden geometry, the depth layer extraction can be modified to $depth(p_{d+1}) > depth(p_d) + \delta$, with $\delta > 0$.

Lee et al. [25] formulated a function that computes the offset δ such that the skipped area is guaranteed to not be visible, and should bound the number of depth layers to maximum of ten layers. Since the depth layers are stored in textures, the authors observed that advantage can be taken of the pixel center, point-sampled geometry representation. The idea is to project the resulting pixels onto the scene and skip a distance behind the projected pixel such that the area is not capturable by other pixels. Comparable to the area covered by the umbra, if the viewer was a lightsource and the projected pixel an occluder. To increase the skipped distance, the projection of an area almost equals two times the pixel size projected onto the scene. This is justified because the geometry is sampled from the pixel centers. As such, the geometry as seen from the area between pixel centers won't be included in the depth layers and will not lead to information loss.

The mathematical formulation is as follows:

$$\delta = \frac{d * s}{E - s} \tag{3.1}$$

with depth d of a fragment, pixel size s and lens radius E. For the purposes in this report, E is the distance between the left- and rightmost view position.

Note that extended depth offset is portrayed as an optimization. But for the performance measurements it is considered an integral part of the basic epipolar interpolation pipeline, since the overhead of the interpolation method is exorbitantly high without this feature. As such, when the performance measurements mention the "basic center/dual epipolar pipeline", then this optimization is considered to be included.

3.4.2 Conservative visibility estimation

The extended depth offset is great for reducing the amount of work by decreasing the number of depth layers. However, the individual depth layers still contain geometry that won't contribute to the result because it is occluded. As an example, consider a wall with a decorative curtain some small distance in front of it. But far enough behind the curtain to not be considered by the virtually projected pixels. When looking straight ahead at the curtain, some amount of the wall will never be seen. No matter how far the viewer strafes left or right.

Solving this issue equates to computing the potentially visible region. The same problem is encountered by the heuristic used in the hierarchical view acquisition algorithm. As such, the computation of the visibility map is reused for this optimization. A mathematical overview of the generation of the visibility map is identical to the *ErodeView* function as shown in Code listing 3.5. The visibility map is computed for each depth layer and is used as a mask for the next depth layer. The holes in the visibility masks match the sections for which more information is needed. As such, only those sections are captured during the next depth peeling pass.

Note that the masks of all previous depth layers are accumulated during the generation of a new visibility mask. Otherwise, already masked out sections may be reincluded in later visibility masks and would lead to unwanted extra work for the interpolation stage.

Usage of conservative visibility estimation by depth peeling based source view acquisition methods requires modifications of the basic algoritm. Integration of conservative visibility estimation in the center view acquisition is shown in Code listing 3.10. Necessary changes for the dual view acquisition algorithm can be seen in Code listing 3.11. Both code listings only show modified functions; any other referenced functions remain the same.

```
modified AcquireCenterSourceViews(Projection parameters $MVP_c$, Projection parameters $MVP_l$, Projection parameters $MVP_r$, Geometry $G$):

$V$ := $\{\emptyset\}$

$F$ := RasterizeGeometry($MVP$, $G$)
$F_{v,0}$ := DetermineVisibleFragments($F$, $\{\emptyset\}$)
$V_0$ := CalculateLighting($F_{v,0}$)
$V$ := $V \cup \{V_0\}$

$i$ := 1
while true:
    $VisMask_l$ := ErodeView($F_{v,i-1}$, $MVP_c$, $MVP_l$)
    $VisMask_r$ := ErodeView($F_{v,i-1}$, $MVP_c$, $MVP_r$)
    $VisMask$ := $\{(x,y,1) \mid (x,y,1) \in VisMask_l \text{ or } (x,y,1) \in VisMask_r\}$

    $F_{v,i}$ := DetermineVisibleFragments($F$, $F_{v,i-1}$, $VisMask$)
    if $F_{v,i} = \{\emptyset\}$:
        break
    end if

    $V_i$ := CalculateLighting($F_{v,i}$)
    $V$ := $V \cup \{V_i\}$

    $i$ := $i + 1$
end while

return V
```

Code listing 3.10. *Modified functions to show integration of conservative visibility estimation in the center view acquisition algorithm. Note, DetermineVisibleFragments also requires modification. The modified function is shown in Code listing 3.11. The remaining referenced functions are identical to previously shown versions.*

```
modified AcquireDepthLayers(Projection parameters $MVP$, Projection parameters
$MVP_o$, Geometry $G$):

$V$ := { $\emptyset$ }

$F$ := RasterizeGeometry($MVP$, $G$)
$F_{v,0}$ := DetermineVisibleFragments($F$, { $\emptyset$ })
$V_0$ := CalculateLighting($F_v$)
$V$ := $V \cup \{ V_0 \}$

$i$ := 1
while true:
    $VisMask$ := ErodeView($F_{v,i-1}$, $MVP$, $MVP_o$)

    $F_{vi}$ := DetermineVisibleFragments($F$, $F_{v,i-1}$, $VisMask$)
    if $F_{v,i} = \{ \emptyset \}$:
       break
    end if

    $V_i$ := CalculateLighting($F_{v,i}$)
    $V$ := $V \cup \{ V_i \}$

    $i$ := $i + 1$
end while

return $V$

modified DetermineVisibleFragments(Fragments $F$, Occluding Fragments $F_o$, Visibility
Mask $V$):

if $F_o = \{ \emptyset \}$:
    return $F$
end if

$F_v$ := $\{(x,y,1) \mid (x,y,\_) \in F \}$
foreach fragment $(x,y,z) \in F$ do:
    if $F_v$ contains $(x,y,z_d)$ such that $z_d < z$ or
       $F_o$ contains $(x,y,z_o)$ such that $z_o > z$ or
       $V$ not contains $(x,y,1)$:
         continue
    end if

    $F_v$ := $(F_v \setminus \{ (x,y,\_) \}) \cup \{ (x,y,z) \}$
end foreach

return $F_v$
```

Code listing 3.11. *Modified functions to show integration of conservative visibility estimation in the dual view acquisition algorithm. Unless otherwise specified, any referenced functions are the same as shown in earlier sections.*

3.4.3 Primitive coalescing

The basic transformation of every pixel to its corresponding epipolar line, as described in Section 3.3.2, is suboptimal with respect to the performance. Neighboring source pixels are likely to have similar depth values. As such, the corresponding epipolar lines are likely to follow a similar direction. But, are spatially shifted by one pixel. This can be taken advantage of by approximating the consecutive epipolar lines with a single quad in epipolar space.

So, instead of transforming single pixels to epipolar lines, ranges of consecutive pixels can be transformed to quads. Furthermore, since the slopes of the epipolar lines in a range of pixels are similar, only the endpoints of a range of pixels needs to be transformed to the four corner points of an epipolar quad. This should result in approximately the same epipolar area being covered as the consecutive epipolar lines. The method to perform the transformation of the endpoints is the same as the basic algorithm.

The ranges of consecutive pixels with similar depth values are computed based on the depth map. These are a by-product of the source acquisition stage. A custom 1-dimensional mipmap is generated for a given depth map, where the finest miplevel is initialized with the depth values acquired during source acquisition. Each pixel in the mipmap stores the lowest and highest value in the range it belongs to, or it is marked as not continuing a pixel range if the range between the minimum and maximum is too large. The maximum allowed delta between the minimum and maximum depth value is a user provided fixed value.

To extract the pixel ranges, the mipmap is traversed once for each source pixel from the coarsest level to the finest level. For a given pixel p at image location $s = (x_s, y_s)$, its corresponding location $t = (x_t, y_t)$ in mipmap level m can be computed using Equation 3.2.

$$t = \left(\left\lfloor \frac{x_s}{2^m} \right\rfloor, y_s\right) \tag{3.2}$$

Only if pixel p is at the beginning of a range is it transformed into an epipolar quad. Mathematically this corresponds with $x_s \; mod \; 2^m = 0$, where m is the coarsest mipmap level for which the range of depth values is still similar.

The pseudocode for acquiring the mipmap necessary for primitive coalescing is shown in Code listing 3.12. Any modifications to the original interpolation algorithm functions can be seen in Code listing 3.13.

Unfortunately, a naïve primitive coalescing implementation, with one source pixel being processed by a single thread on the GPU, leads to branch divergence because only pixels at the start of a range are processed into quads. The threads processing the other pixels are performing superfluous calculations or remain dormant.

To overcome this problem an intermediate step needs to be added. Collecting the pixels that form a range in an intermediate buffer, instead of directly processing each source pixel. This leads to better utilization of hardware because the branches are relatively short. To create the epipolar geometry, the necessary data is read from the intermediate buffer. Fortunately, such a pass was already included in the basic interpolation pipeline. As such, this optimization can be integrated into the conceptual *CaptureActivePixels* function.

```
AcquirePrimitiveMipMap(Source view $v$, $MaxMipMapDepth$, $DepthDifferenceThreshold$):

$DimX$ := Get largest $x$-coordinate in $v$
$PrimitiveMipMap\_DimX$ := 0
for $MipMapLevel$ := 1 to $MaxMipMapDepth$:
    $PrimitiveMipMap\_DimX$ := $PrimitiveMipMap\_DimX + \left\lceil \frac{DimX}{2^{MipMapLevel}} \right\rceil$
end for

// Copy depth values to primitive mipmap
$PrimitiveMipMap$ := $\{\emptyset\}$
foreach fragment $(x, y, z, \_) \in v$:
    $PrimitiveMipMap$ := $PrimitiveMipMap \cup \{(PrimitiveMipMap\_DimX - DimX + x, y, z, z)\}$
end foreach

// Generate primitive mipmap
$WriteBaseIndex$ := $PrimitiveMipMap\_DimX - DimX$
for $MipMapLevel$ := 1 to $MaxMipMapDepth$:

    $ReadBaseIndex$ := $WriteBaseIndex$
    $WriteBaseIndex$ := $WriteBaseIndex - \left\lceil \frac{DimX}{2^{MipMapLevel}} \right\rceil$
    $PreviousReadBound$ := $\left\lceil \frac{DimX}{2^{MipMapLevel} - 1} \right\rceil$

    foreach fragment $(x, y, \_, \_) \in v$:

        // Read depth values
        Read $DepthLeft\_Min$ and $DepthLeft\_Max$ from
        $(ReadBaseIndex + 2 * x, y, DepthLeft\_Min, DepthLeft\_Max) \in PrimitiveMipMap$

        $DepthRight\_Min$ := 1
        $DepthRight\_Max$ := 1
        if $2 * x < PreviousReadBound$:
            Read $DepthRight\_Min$ and $DepthRight\_Max$ from
            $(ReadBaseIndex + 2 * x + 1, y, DepthRight\_Min, DepthRight\_Max) \in PrimitiveMipMap$
        end if

        // Check for continuation of primitive coalescing in pixel range
        $MinDepth$ := $min(DepthLeft\_Min, DepthRight\_Min)$
        $MaxDepth$ := $max(DepthLeft\_Max, DepthRight\_Max)$
        if $MinDepth = 1$ and $MaxDepth = 1$ or $MaxDepth - MinDepth > DepthDifferenceThreshold$:
            $MinDepth$ := 1
            $MaxDepth$ := 1
        end if

        $PrimitiveMipMap$ := $PrimitiveMipMap \cup \{(WriteBaseIndex + x, y, MinDepth, MaxDepth)\}$
    end foreach
end for

return $PrimitiveMipMap$
```

Code listing 3.12. *Pseudocode for generation of a primitive coalescing mipmap. Mip level 0 stores ranges of single pixels, mip level 1 stores ranges of 2 pixels wide, mip level 2 stores ranges of 4 pixels wide, etc. The mipmap is generated based on the depth map and a user provided threshold. The pixels store minimum and maximum depth values in a range of pixels.*

```
modified CaptureActivePixels(Source view $v$, Projection parameters $MVP$, $MaxMipMapDepth$, $DepthDifferenceThreshold$):

$PrimitiveMipMap$ := AcquirePrimitiveMipMap($v$, $MaxMipMapDepth$, $DepthDifferenceThreshold$)

$DimX$ := Find largest $x$-coordinate in $v$
$ActivePixels$ := $\{\emptyset\}$
foreach fragment $(x, y, z, c) \in v$:
    if $z = 0$:
        continue
    end if

    // Calculate Stride for current pixel from primitive mipmap
    $Stride$ := $0$
    $DepthMinMax$ := $(0, 0)$
    $ReadIndex$ := $0$
    $MipMapLevel$ := $MaxMipMapDepth$
    while $MipMapLevel \geq 0$:
        $Stride$ := $2^{MipMapLevel}$

        $ReadX$ := $ReadIndex + \left\lfloor \frac{DimX}{Stride} \right\rfloor$
        Read $(ReadX, y, Min, Max) \in PrimitiveMipMap$

        $DepthMinMax$ := $(Min, Max)$
        if $Min < 1$ and $Max < 1$:
            break
        end if

        $ReadIndex$ := $ReadIndex + \left\lceil \frac{DimX}{Stride} \right\rceil$
        $MipMapLevel$ := $MipMapLevel - 1$
    end while

    // Check for pixel validity and if it's within viewport range
    $(Min, Max)$ := $DepthMinMax$
    if $Min \geq 1$ and $Max \geq 1$ or $x \,\%\, Stride > 0$:
        continue
    end if

    $ActivePixels$ := $ActivePixels \cup \{(x, y, z, c, Stride)\}$
end foreach

return $ActivePixels$
```

Code listing 3.13. *Shows the modified CaptureActivePixels function with support for primitive coalescing. MaxMipMapDepth and DepthDifferenceThreshold are user-provided parameters, where MaxMipMapDepth is the maximum number of mipmap levels in the primitive mipmap and DepthDifferenceThreshold is the maximum allowed distance between the minimum and maximum depth value in a range of pixels for them to be coalesced.*

3.4.4 Dynamic layer reduction

The extended depth offset and visibility estimation optimizations reduced the amount of work for the epipolar interpolation stage. However, during the development of the center and dual view acquisition algorithms, the observation was made that the resulting views mostly relied on the first several depth layers. So, to further reduce the number of unnecessary computations, a lazy layer reduction step was added to the view interpolation stage.

This process is lazy because there is a one frame delay in the acquired fragment counts. The concrete implementation of the algorithm uses occlusion queries to count the number of fragments. To ensure the graphics pipeline doesn't stall when a frame is rendered, the results of the occlusion queries for the previous frame are retrieved and used to determine whether to add an extra depth layer or discard the last depth.

The actual layer reduction decides each frame to include or discard the last depth layer. The input to the decision is the number of fragments that pass the depth test during the epipolar plane rendering and two fixed, user-specified thresholds. The thresholds are a minimum and maximum percentage of fragments. If the number of passed fragments is below the lower limit, then the last layer is dropped from the result during rendering of the next frame. If the number of passed fragments is higher than the upper limit another depth layer is added, if more layers are available.

Note that the fixed user-specified thresholds are not ideal, and probably needs to be replaced with a more robust metric. As a proof-of-concept it is easy to implement, but the downside is an inherent dependency on the scene and size of the viewport. For example, imagine a situation with a pilot flying in a star field versus a fast-paced shooter in a close quarters combat scenario. Rendering the star field results in a lot empty space, where most of the pixels are not touched. On the other hand, the fast-paced shooter likely has an entire image of colored pixels. Thus requiring different thresholds for the layer reduction per scene. A more suitable threshold might be based on the location and view direction of the player.

3.5 View dependent properties

Illumination models are used to compute the light intensity reflected from a given point on the surface of an object. The light intensity is commonly computed as the sum of three components: ambient, diffuse and specular reflection, with the ambient and diffuse reflection being independent of the position of the viewer with respect to a given surface. Unfortunately, the specular reflection has a dependency, and needs special attention.

In previous sections, the topic has been the epipolar interpolation and how to improve the performance of the basic algorithm. The implicit working assumption was that surfaces only had an ambient and diffuse light response, because there is no dependency on the viewer position. But, this section will relax that assumption and discuss support for view dependent properties. More specifically, the implementation of the framework allows for support of the (Blinn-)Phong reflection model.

The Phong reflection model [36] is a local illumination model that is based on empirical correctness instead of physical correctness. Given a situation comparable to Figure 3.7, the specular intensity k_{spec} is calculated using the formula shown in Equation 3.3. Herein, R is the reflection vector of the incoming light on a surface, V is the viewpoint vector and n determines the shininess of the surface.

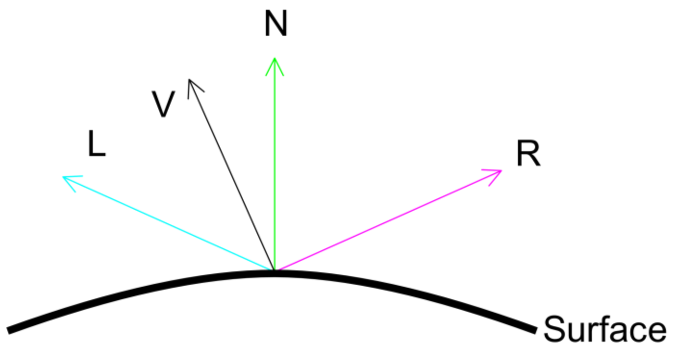

Figure 3.7. *Visual representation of a surface with the necessary vectors for computing the specularity according to the Phong reflection model.* ***L*** *is the incoming light;* ***V*** *is the eye position;* ***N*** *is the surface normal;* ***R*** *is the reflection direction vector. The vectors have the following constraint: the angle between* ***L*** *and* ***N*** *is identical to the angle between* ***N*** *and* ***R****.*

$$k_{spec} = \|R\|\|V\| \cos^n(\alpha) = (R \cdot V)^n \tag{3.3}$$

A disadvantage of the Phong reflection model is continual recalculation of $R \cdot V$. As such, a computationally more efficient model is used in practice. It is commonly known as the Blinn-Phong reflection model, and is formulated as Equation 3.4, where N is the surface normal, L is the light direction vector and V is the viewpoint vector.

$$k_{spec} = \left(N \cdot \left(\frac{L+V}{\|L+V\|}\right)\right) \tag{3.4}$$

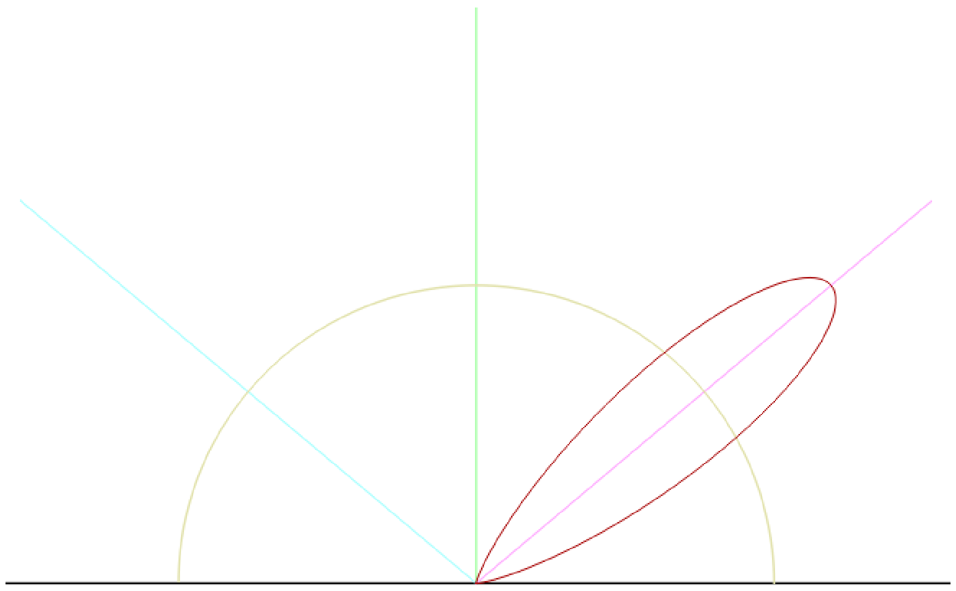

Figure 3.8. *Polar plot of the reflected light distribution by the Blinn-Phong reflection model. The green line is the surface normal. The cyan line represents the incident light and is configured at $50°$. The magenta line is the reflected light direction. The red lobe is the light distribution, whereby the distance from the center signifies the intensity. Source: BRDF Explorer by Disney Enterprises, Inc.*

The pseudocode for calculating the specularity given a surface position, including its normal, a light position, a camera position, and various color and light attributes is given in Code listing 3.14. The given algorithm computes the specularity in world space. For traditional rendering pipelines, using world space coordinates is slightly less efficient to implement than using view space coordinates, but the world space coordinates of the different attributes given a quad in an epipolar plane don't change. As such, using world space coordinates is the only viable option.

Note that the Blinn-Phong reflection model has been superseded by more physically correct BRDFs for most real-time, photorealistic rendering applications. The reason for opting to show support for view dependent properties using the Blinn-Phong reflection model is that it is widely known. Moreover, the model is relatively simple compared to state-of-the-art reflection models in terms of both understanding and implementing it. Furthermore, this is only a proof-of-concept that view dependent properties can be implemented in combination with the presented view interpolation algorithm. Additionally, the Blinn-Phong reflection model is symmetric around the reflection vector at non-grazing angles. An example of this property can be seen in Figure 3.8. This allows for piecewise approximation without introducing significant error as brute forcing specularity is too costly performance-wise.

```
ComputeSpecularity($Coords_w$, $Normal_w$, $Color_s$, $Color_i$, $Color_l$,
$CameraPosition_w$, $LightPosition_w$,
$LightAttenuation_c$, $LightAttenuation_l$, $LightAttenuation_q$):

$SurfaceLightVec$ := $LightPosition_w - Coords_w$
$SurfaceLightDist2$ := $SurfaceLightVec \cdot SurfaceLightVec$
$SurfaceLightDist$ := sqrt($SurfaceLightDist2$)
$Luminosity$ := $\frac{1}{LightAttenuation_c + SurfaceLightDist * LightAttenuation_l + SurfaceLightDist2 * LightAttenuation_q}$

$LightDir$ := $\frac{LightPosition_w - Coords_w}{||LightPosition_w - Coords_w||}$
$ViewDir$ := $\frac{CameraPosition_w - Coords_w}{||CameraPosition_w - Coords_w||}$
$HalfwayDir$ := $\frac{LightDir + ViewDir}{||LightDir + ViewDir||}$
$Specularity$ := $max(Normal_w \cdot HalfwayDir, 0)^{Color_i}$

return $Luminosity * Specularity * Color_l * Color_s$
```

Code listing 3.14. *Pseudocode for calculating the specularity of a surface point in world space, given the necessary parameters.* $Coords_w$ are the coordinates of the surface point, $Normal_w$ is the corresponding normal, $Color_s$ is the specular surface color, $Color_i$ is the specular intensity, $Color_l$ is the color of the light source, $CameraPosition_w$ is the position of the viewer, $LightPosition_w$ is the position of the light source, and respectively $LightAttenuation_c, LightAttenuation_l, LightAttenuation_q$ are the constant, linear and quadratic light attenuation.

A straightforward implementation to support view dependent properties would be to interpolate the necessary properties for the per-view computation and do the actual computation for each pixel in each view. Quality wise, the outcome would be on par with a baseline deferred pipeline implementation. However, the performance will probably suffer. Moreover, the Big O performance characteristic of the entire interpolation pipeline would be a function that is linear relative to the number of views. Thus, not leading to a theoretical performance improvement over the brute force deferred pipeline.

Alternatively, specular reflection is incorporated into the epipolar view interpolation algorithm by approximation. The idea is to do a piecewise approximation by splitting every quad in the epipolar plane into multiple connected parts. At every vertex, the specularity is computed. The computed specular contribution will be interpolated by the rasterizer and added to the diffuse component in the fragment shader. The number of splits is variable and left to be decided by a user. More splits will result in higher quality images, but also requires more computational and memory resources. See Figure 3.5 for a visual representation of the integration of split geometry in the interpolation algorithm. This method takes advantage of hardware acceleration to interpolate the specular highlights to the different views. A disadvantage is the added overhead to compute the locations of the geometry splits. Furthermore, more geometry must be processed by the rasterizer, and extra memory is needed to store the extra geometry.

The location where quads are split in epipolar space is important. They must be placed such that the key features of specular highlights are visible after interpolation. For the Blinn-Phong reflection model, this equates to the brightest spot. As an example, consider an even spread of the splits in the epipolar geometry. This will probably result in a reduced intensity or missing specular highlight because the location of the highlight doesn’t coincide with the location of a split vertex.

To sidestep this issue, at least one epipolar geometry split is placed on the scanline in the epipolar plane that coincides with the view containing a location for a specular highlight. This location can easily be computed for the Blinn-Phong reflection model since it coincides with the pixel locations for which the light reflection vector of a surface intersects with the line segment between the left- and rightmost camera. Finding the intersection is done with a line segment-to-line segment intersection in world-space. The computation is done in world space since the position of a source pixel remains identical in the entire epipolar plane. As a result, the light reflection vector is constant since the position of the light source is kept constant during the rendering of a single frame. The mathematical formulas for the intersection test can be found in Code listing 3.15.

Input: Two line segments in the form $y = a + xb$, where a and b are 3d vectors. The first line segment consists of a surface location S and a light reflection direction R. The second line segment consists of the position of the leftmost camera C_l and the direction vector $C_d = C_r - C_l$, where C_r is the position of the rightmost camera. Every vector is expected to be in world-space.
Output: If the line segments intersect, a value I between 0 and 1, such that $C_l + I * C_d$ corresponds with the location of the intersection. Otherwise, a value of -1 is returned.

```
LineIntersect($S$, $R$, $C_l$, $C_d$):

// Check if the given line segments are collinear
$W$ := $R \cdot C_d$
if $W \cdot W == 0$:
    return $-1$
end if

// Compute the closest distance between the given line segments
$I_{cam}$ := $(((S - C_l) \times R) \cdot W) * (\frac{1}{W \cdot W})$
$I_{reflect}$ := $(((S - C_l) \times C_d) \cdot W) * (\frac{1}{W \cdot W})$

// Check if the intersection is in range of the line segments
if $I_{cam} < 0$ or $I_{cam} > 1$:
    return $-1$
end if

// Check if closest distance between the given line segments is 0
$DistanceLine$ := $(S + I_{reflect} * R) - (C_l + I_{cam} * C_d)$
if $(DistanceLine \cdot DistanceLine) \neq 0$:
    return $-1$
else
    return $I_{cam}$
end if
```

Code listing 3.15. *Pseudocode for computing the intersection between two line segments in 3-dimensional space.*

A mathematical overview of integrating specular reflection into the basic epipolar view interpolation algorithm can be found in Code listing 3.17. Additionally, Code listing 3.16 contains the pseudocode for several functions that control the placement of the splits for the quads in epipolar space. Note that several lines of the original pseudocode have been omitted for brevity.

The described method comes with a remark regarding the general applicability. The piecewise approximation only works if only one light source is considered. In this case, the location of the specular reflection can easily be computed. As such, the current implementation is limited to a single light source.

A second remark about the support for specularity is its usage in conjunction with the primitive coalescing optimization. If the quads are only split along one direction, as is currently the case, then it could happen that the specular highlight may be missed because it falls in a pixel range that is coalesced together. For this reason, the activation of these options is mutually exclusive in the proof-of-concept implementation. It might be possible to split a quad in epipolar space along multiple axes, but this is left as future work.

```
ComputeIntersection($Coords_w$, $Normal_w$, $CameraPosition_w$, $CameraDirection_w$, $LightPosition_w$):

$LightDir$ := $LightPosition_w - Coords_w$
$LightReflectDir$ := $LightDir - 2 * (LightDir \cdot Normal_w) * Normal_w$
$LightReflectDir$ := $\frac{LightReflectDir}{||LightReflectDir||}$

$IntersectionPoint$ := LineIntersect($Coords_w$, $LightReflectDir$, $CameraPosition_w$, $CameraDirection_w$)
if $IntersectionPoint = -1$:
    return 0
else
    return $IntersectionPoint$
end if

GenerateEpipolarSplits($IntersectionPoint$, $SplitCount$):

$SectionLength$ := $\frac{1}{SplitCount - 1}$
$CenterSplitIndex$ := $max(0, min(round(\frac{IntersectionPoint}{SectionLength}), SplitCount - 2))$

$Splits$ := []
$Counter$ := 0
$SectionLength$ := $\frac{IntersectionPoint}{CenterSplitIndex}$
while $Counter < CenterSplitIndex$:
    $Splits[Counter]$ := $Counter * SectionLength$
    $Counter$ := $Counter + 1$
end while

$Splits[Counter]$ = $IntersectionPoint$
$Counter$ := $Counter + 1$

$SectionLength$ := $\frac{1 - IntersectionPoint}{SplitCount - Counter}$
while $Counter < SplitCount$:
    $Splits[Counter]$ := $IntersectionPoint + (Counter - CenterSplitIndex) * SectionLength$
    $Counter$ := $Counter + 1$
end while

return $Splits$
```

Code listing 3.16. *Several support functions to calculate the locations on a quad where the splits should be placed.*

```
modified DrawDepthLayer($ActivePixels$, Projection parameters $MVP_{v,i}$; $MVP_l$; $MVP_r$, $LightPosition_w$,
$SplitCount$):

.....

foreach fragment $(x, y, z, c_{l,d}, Stride) \in ActivePixels$:

    // Compute epipolar plane end points corresponding to current fragment
    $Coords_{l,v}$ := Unproject $(x, y, z)$ to view coordinates using $MVP_{v,i}$
    $Coords_{l,w}$ := Unproject $Coords_{l,v}$ to world coordinates using $MVP_{v,i}$
    .....

    Get color $c_{r,d}$ proper for $x + Stride$
    $Coords_{r,v}$ := Get coordinates proper for $x + Stride$ and
                    unproject to view coordinates using $MVP_{v,i}$
    $Coords_{r,w}$ := Unproject $Coords_{r,v}$ to world coordinates using $MVP_{v,i}$
    .....

    Get normal $Normal_{l,w}$ for the coordinates $(x, y, z)$ in world space
    Get specular color $Color_{l,s}$ for the coordinates $(x, y, z)$
    Get specular intensity $Color_{l,i}$ for the coordinates $(x, y, z)$

    Get normal $Normal_{r,w}$ for the coordinates $(x + Stride, y, z)$ in world space
    Get specular color $Color_{r,s}$ for the coordinates $(x + Stride, y, z)$
    Get specular intensity $Color_{r,i}$ for the coordinates $(x + Stride, y, z)$

    // Loop over epipolar splits and emit corresponding vertices
    $LightIntersect$ := ComputeIntersection($Coords_{l,w}$, $Normal_{l,w}$, $CameraPosition_l$, $CameraPosition_l -$
 $CameraPosition_r$, $LightPosition_w$)
    $EpipolarSplits_l$ := GenerateEpipolarSplits($LightIntersect$, $SplitCount$)
    $LightIntersect$ := ComputeIntersection($Coords_{l,w}$, $Normal_{r,w}$, $CameraPosition_l$, $CameraPosition_l -$
 $CameraPosition_r$, $LightPosition_w$)
    $EpipolarSplits_r$ := GenerateEpipolarSplits($LightIntersect$, $SplitCount$)

    $Vertices$ := []
    $VertexCount$ := 0
    for $Index$ := 0 to length($EpipolarSplits$):
        .....

        $c_{l,s}$ := ComputeSpecularity($Coords_{l,w}$, $Normal_{l,w}$, $Color_{l,s}$, $Color_{l,i}$)
        $c_{r,s}$ := ComputeSpecularity($Coords_{r,w}$, $Normal_{r,w}$, $Color_{r,s}$, $Color_{r,i}$)

        $Vertices[VertexCount]$ := EmitVertex($Coords_{l,v}$, $MVP$, $\alpha$, $n$, $c_{l,d}$, $c_{l,s}$)
        $VertexCount$ := $VertexCount + 1$
        $Vertices[VertexCount]$ := EmitVertex($Coords_{r,v}$, $MVP$, $\alpha$, $n$, $c_{r,d}$, $c_{r,s}$)
        $VertexCount$ := $VertexCount + 1$
    end for

    # Loop over vertices to assemble created triangles
    .....
end foreach

$RasterizedFragments$ := RasterizeGeometry($Identity$, $EpipolarTriangles$)

$Fragments$ := $\{\emptyset\}$
foreach fragment $(x, y, z, d, c_d, c_s) \in RasterizedFragments$:
    $Fragments$ := $Fragments \cup \{(x, y, z, d, c_d + c_s)\}$
end foreach

return $Fragments$
```

Code listing 3.17. *Modified function DrawDepthLayers with support for specularity. In addition to the original parameters, the function expects the position of the light source as $LightPosition_w$, and the number of splits to be placed in each quad as $SplitCount$. Note that several unchanged lines are omitted for brevity.*

4 Results

To evaluate whether the goal, as stated in Chapter 1, is met by the framework as described in the previous chapter, a means of quantifying the performance and image quality of the framework is needed. Furthermore, to guarantee the reproducibility of the presented research the test method needs to be documented.

This chapter documents the measurements to quantify the performance and image quality of the previously described framework. Since the performance and image quality are heavily dependent upon the hardware used to measure, it is considered part of the test method. As such, Section 4.1 starts with documenting the hardware used for measuring. Then, Section 4.2 explains which measures are used and which measurements are performed to determine the quality of the framework. Afterwards, Section 4.3 outlines the ground truth against which the rendering framework is compared. Lastly, Section 4.4 and Section 4.5 give the outcome of the measurements for, respectively, the performance and image quality.

4.1 Test platform

The performance of an algorithm is highly dependent on the used hardware platform. Furthermore, the performance also depends on the software layers connecting the hardware to the algorithm. Examples are the operating system and device drivers. There could even be a difference in performance within different versions of the same software. Additionally, the translation of source code to machine understandable instructions also has a significant influence. Newer compiler versions could transform source code to functionally be the same, but with increased utilization of the underlying hardware. Thus, to improve the reproducibility of the presented study, the used hardware and software is documented in Table 4.1. Additionally, this makes it easier for future researchers to compare the performance of improved versions or algorithms. Note, only the hardware, including driver versions, and software impacting the test results are disclosed.

Note that the tests were measured using two different GPUs. Most of the performance and all image quality tests were performed on the Nvidia Titan X. However, due to an unforeseen failure of the previously mentioned GPU, a subset of the performance testing was done on the Nvidia GeForce GTX 1080Ti. All performance tests using the center epipolar pipelines with the San Miguel and Hairball scenes were done with the faster GTX 1080Ti. The remaining tests with the center epipolar pipeline, and other pipelines, were ran on the Nvidia Titan X. All image quality tests have also been completed on the Nvidia Titan X.

Table 4.1. *The hardware and software used to build and test the framework as presented in Section 3. This includes version information as different versions could have a different impact on performance.*

	Model	**Remarks**
CPU	Intel Core i7-5820k	Clock rate: 3.3 GHz
RAM	32 GB DDR4	
GPU	Nvidia Titan X Nvidia GeForce GTX 1080Ti	Driver version: 387.92 Driver version: 388.59
Operating System	Windows 10	
Compiler	Microsoft Visual C++ 2017	Toolset: v141

4.2 Systematic testing

In Section 1.1, the goal of the presented study was to find a method of rendering faster than existing methods, while making sure that the image quality remained reasonable. To determine whether the three described epipolar rendering pipelines satisfy the stated goal, three elements are required. This section focusses on the systematical process of quantifying the performance and image quality.

Input specification

To have comparable results between the baseline and epipolar rendering pipelines, the input to both must be equal. As noted in Section 3, the input consists of a 3-dimensional description of a scene and a set of cameras.

It is undoable to test every possible input configuration. As such, the chosen 3d models represent common use cases. Starting with a simple cube. This model represents very low poly, extremely simple test scenes. A medium complexity scene, in terms of vertices and triangles, included in the test set is referred to as Sponza (xxx), where xxx refers to the amount of subdivision. It represents an architectural setting, which seems very common when looking to popular AAA-games such as GTA V ™, Mafia III ™ and Sleeping Dogs ™. In addition to the original Sponza scene, several modified versions have been included to observe the scaling of the algorithms. These have been subdivided between 0 and 3 times using Blender to artificially increase the number of vertices and triangles. A subdivision means that each edge is split 0, 1, 2 or 3 times. The reason for subdividing a scene opposed to using a different and more complex scene to show the algorithm scaling is that a different scene might lead to different behavior by the algorithm. An example is a different depth complexity, which might lead to a different number of depth layers. The San Miguel scene has been added to represent high complexity scenes. Like the Sponza scene, it went through the artificial complexity boosting process. Besides, the algorithms have also been tested with the Hairball model, since this model has a lot of detail in a small object. The exact number of vertices and triangles contained in each model can be found in Table 4.2.

Aside from the previously mentioned models. The view dependent properties are additionally tested with a separate set of models. The difference between the previously covered models are the specular material properties, such as specular color and specular intensity. Although the sponza scene has specularity in its materials, the chosen models are affected more by specularity, which should lead to a more pronounced effect. Thus, any errors introduced by the epipolar interpolation should be easier to spot and measure. The specularity test set consist of the Stanford Lucy model and the Mitsuba model.

Table 4.2. *Number of vertices and triangles for each of the models used during the testing process.*

Model name	Number of vertices	Number of triangles
Cube	8	12
San Miguel (Original)	4,488,339	7,838,629
San Miguel (High)	10,469,048	19,313,050
San Miguel (Extreme)	16,729,835	31,354,140
Sponza (Original)	145,185	262,267
Sponza (High)	551,211	1,049,026
Sponza (VeryHigh)	2,150,064	4,196,244
Sponza (Extreme)	8,494,946	16,784,976
Hairball	1,470,000	2,880,000
Lucy	249,771	499,530
Mitsuba	30,869	61,612

Note that the performance testing was only done using the San Miguel, Sponza and Hairball scenes. The quality testing was done using the Cube, Sponza (Original), Hairball, Lucy and Mitsuba scenes.

Besides the 3D models, the rendering framework specified in the previous chapter also expects a set of cameras. To be able to compare the baseline against the epipolar based algorithms, this set must be similar for all test cases using an identical configuration. To achieve this for the image quality measurements, a static camera position and view direction are chosen, saved to a file and reused for each tested configuration. The inter-camera distance controls the distance between the different cameras within a view set, and for each test configuration consists of a set of three values. The smallest distance represents a very small shift, where the content displaces by up to several tens of pixels. The middle distance is results in a moderate shift of content, where the content moves by up to several hundreds of pixels. The largest inter-camera distance is chosen such that the left- and rightmost views have a small overlap, but the content is still entirely visible.

Unfortunately, the inter-camera distance is dependent on the scene since the different models don't have the same scale. The different values belonging with each of the test scenes can be found in Table 4.3. To give an indication of the scale of the scene, the near and far plane of the cameras can be found in the same table.

To measure the performance, the same variation of inter-camera distance and the same number of views are used with regards to the image quality tests. However, the camera moves along a predefined path to simulate player movement through a scene. This will expose changes in the performance of the algorithm as the visible geometry changes. The path along which the cameras move is unique to each scene. Furthermore, it will be reset and replayed for each rendering configuration.

The distance between the leftmost and rightmost camera doesn't change for the performance tests, and equals the maximum numbers of views multiplied by the inter-camera distance. For the used test cases, the maximum is 500 views. For test configurations with less than the maximum numbers of views, the inter-camera distance is the distance between the leftmost and rightmost camera divided by the number of views. Lastly, the number of views is variable and can be one from the following set of numbers: $\{2, 10, 25, 50, 75, 100, 200, 300, 400, 500\}$.

Table 4.3. *Overview of the inter-camera distance settings for each test scene. The maximum distance is the distance between the left- and rightmost camera. It is computed by multiplying inter-camera distance with the maximum number of views, which is 500 for the reported values. The near and far plane give an indication of the scale of the scene.*

Model name	Inter-camera distance	Max camera distance	Near	Far
Cube	$\{0.001;\ 0.01;\ 0.03\}$	$\{0.5;\ 5;\ 15\}$	1	500
Sponza	$\{0.3;\ 0.6;\ 0.9\}$	$\{150;\ 300;\ 450\}$	50	5000
Hairball	$\{0.0025;\ 0.005;\ 0.01\}$	$\{1.25;\ 2.5;\ 5\}$	0.1	50
Lucy	$\{0.0025;\ 0.005;\ 0.01\}$	$\{1.25;\ 2.5;\ 5\}$	0.1	50
Mitsuba	$\{0.0025;\ 0.005;\ 0.01\}$	$\{1.25;\ 2.5;\ 5\}$	1	50

Performance metrics

The metric used to quantify the performance of the algorithm is the frame time, and is reported in milliseconds. It is measured as an average over 32 frames, where each frame is taken as the time needed to compute a single set of views. Herein, averaging mostly compensates for interfering irregularities, such as the operating system preempting the execution of the algorithm to let other programs run.

However, note that the runtime doesn't include the time needed to setup the rendering pipeline. Examples hereof are allocation of memory for resources such as textures or buffers, or loading GLSL shaders from disk.

Image quality metrics

Image quality of two different image sets can be compared based on several characteristics. The goal for the epipolar rendering pipelines is to acquire image sets that humans cannot reliably distinguish from the baseline image sets in terms of image quality. The ideal solution to judge the image quality between several image sets would be to have a large enough group of test subjects judge the difference between image sets. Unfortunately, due to limitations imposed on the study, as discussed in Section 5.2, it is unfeasible to setup and conduct such an experiment.

The next best option to quantify the image quality objectively, is the usage of metrics. To compare different image sets several full-reference metrics have been used. The image quality was measured using the following four metrics: Mean Squared Error (MSE), Peak Signal-to-Noise Ratio (PSNR), Structural Similarity (SSIM) [47] and lastly PSNR-HVS-M [37].

The MSE is defined as

$$MSE = \frac{1}{W * H} * \sum_{w=1}^{W} \sum_{h=1}^{H} \left(f(w,h) - \hat{f}(w,h) \right)^2 \tag{4.1}$$

where W is the width of an image, H is the height of an image, $f(x,y)$ gives the color at the coordinates (x,y) of the interpolated image and $\hat{f}(x,y)$ returns the color at the coordinates (x,y) of the baseline image. Lastly, the unit of the measurements is the square of the pixel values.

One limitation of using MSE as an objective measure for image quality is that its uninformative without additional information. For example, a MSE of 100.0 for an RGB image with 8-bit color channels is very noticeable. On the other hand, a MSE of 100.0 for an RGB image with 10-bit color channels is barely noticeable. Another issue related to the MSE is its independence of the sign of the error, which may lead to drastically different visual fidelity, but result in the same measured error.

PSNR slightly improves om the MSE by scaling according to the range of possible values. The mathematical definition is as follows:

$$PSNR = 10 * \log_{10}\left(\frac{M^2}{MSE}\right) \tag{4.2}$$

where M is the maximum possible value of any pixel in a given image and MSE is the same as Equation 4.1. The unit of PSNR is defined to be decibels (dB).

Unfortunately, both the MSE and PSNR suffer from the fact that they do not consider how the human visual system responds to artifacts. They only measure the strength of the error, and don't distinguish between distinct types of errors. An example where the MSE breaks down is when a reconstructed image is changed to have several pairs of pixels' swap locations. This will result in the same MSE value but the human visual system could very easily detect the errors [48]. See some examples of this problem in Figure 4.1.

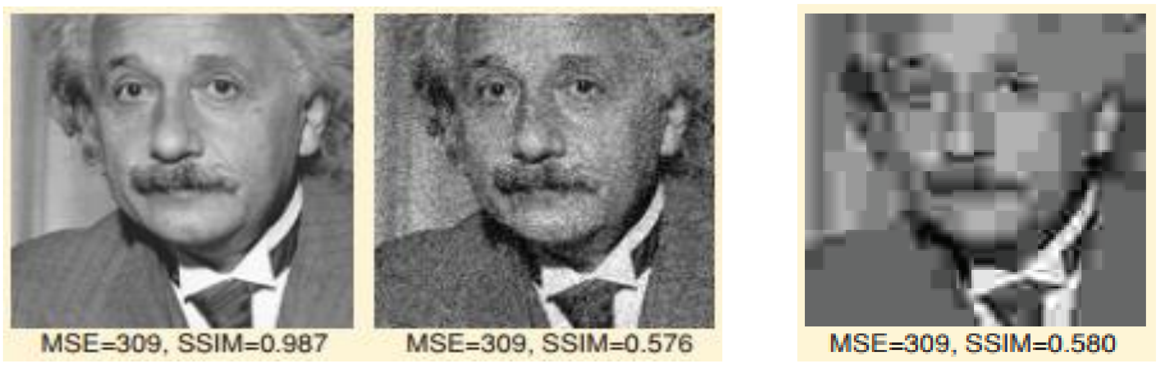

Figure 4.1. *Example of several images with the same MSE, but are perceived extremely different by the human visual system. Source: "Mean Squared Error: Love it or Leave It?" [46].*

Thus, leading to the inclusion of two measures that try to incorporate the human visual system response. Among other properties, the human visual system is sensitive to image contrast and luminance [13, 50]. The PSNR-HVS-M measure incorporates this insight into the $PSNR$ measure. It builds upon equation 4.2, but the MSE is replaced with equation 4.3.

$$MSE_{hvs-m} = \frac{1}{W * H} * \sum_{I=1}^{\frac{H}{8}} \sum_{J=1}^{\frac{W}{8}} \left(\sum_{i=1}^{8} \sum_{j=1}^{8} \left(\left(f(i,j) - \hat{f}(i,j) \right) * CM(i,j) * CSF(i,j) \right)^2 \right) \tag{4.3}$$

where W is the width of the image, H is the height of the image, (I, J) is the position of an $8x8$ block of pixels in an image, (i, j) is the position of a pixel in the $8x8$ block of pixels, $f(x, y)$ gives the color at the coordinates (x, y) of the interpolated image and $\hat{f}(x, y)$ returns the color at the coordinates (x, y) of the baseline image. Furthermore, $CSF(x, y)$ gives predetermined Discrete Cosine Transform (DCT) [1] coefficients that are determined based on the Contrast Sensitivity Function. This function models the sensitivity of the human visual system in response to visual stimuli frequencies. The values hereof are precomputed and can be found in [37]. Lastly, $CM(x, y)$ applies a contrast masking metric to the DCT coefficients. The unit of the PSNR-HVS-M measure is decibels.

Furthermore, the human visual system is sensitive to the spatial frequency. The SSIM measure incorporates this information by comparing the structural information between a pair of images. Wang et al. [47] note that images are highly structured. The pixels contained within an image exhibit strong dependencies, especially when they are spatially proximate. The information within these dependencies provide essential information about the structure of objects to the human visual system.

For the mathematical definition of SSIM refer to [47]. The SSIM reports the similarity between a pair of images as a percentage, where a value of one corresponds with an image that is identical to a given reference image.

4.3 **Ground truth**

To classify the performance of the epipolar rendering framework, a baseline is needed against which the performance and image quality can be compared. The intended usage is in the realm of real-time rendering algorithms. As such, it doesn't make sense to compare the epipolar rendering framework against the family of ray tracing algorithms, since these types of rendering algorithms focus more on image quality and usually don't generate images in real-time.

In real-time rendering environments, the deferred rendering technique is commonly used. Several graphics engines that implement a variation of this technique are Unreal Engine 4 [45], Unity [44], Frostbite 2 [28] and Rockstar Advanced Game Engine [33]. These graphics rendering frameworks are used in numerous AAA games such as Gears of War 4, Battlefield 4, FIFA 17, Grand Theft Auto V. Since the epipolar rendering framework targets similar real-time constraints as deferred rendering is used for, it will serve as the ground truth against which the presented epipolar rendering framework will be benchmarked.

The implementation of the deferred rendering technique used for benchmarking is custom, and geared towards the specific, presented research setting. Typical production-ready implementations would include optimizations such as frustum culling or accounting for level of detail to reduce the geometry processing done by the GPU. For some example algorithms, refer to respectively [2, 8] and [29]. Alongside the optimizations, the resulting images are often post-processed to increase the visual fidelity using methods such as color grading or antialiasing. Some example algorithms can be found in respectively [5] and [23]. However, these types of methods are outside the scope of the presented research and thus not included, otherwise the comparison between the deferred and epipolar rendering pipelines would be unfair and result in skewed results. But note that both a deferred rendering pipeline and an epipolar pipeline could benefit in the same manner, since the respective pipelines don't need to change internally to accommodate the mentioned additions.

4.4 Measured Performance

The described experiments have resulted in several traces characterizing the performance of the baseline algorithm and the epipolar based view interpolation algorithms. A comparison between the measured frame time for the different pipeline configurations can be seen in Figure 4.2 and Figure 4.3. The figures respectively correspond with measurements for the CPU timeline and GPU timeline. When considering only the CPU based measurements, the consensus is, that the deferred rendering pipeline outperforms all interpolation-based pipelines except for several test cases with the Sponza(Original), Sponza(High) and SanMiguel(Original) scenes. However, comparing the two timelines, the GPU is reporting higher frame times to process a single frame, and is thus leading in the frame time judgement. This is typical for algorithms involving the GPU since the only responsibility of the CPU is preparing and sending commands and data. The GPU performs the actual work.

An observation which is less typical is the constant amount of work performed by the CPU under the interpolation-based pipelines. For the deferred rendering pipeline, the frame time on the CPU timeline increases with an increasing number of views. On the contrary, the epipolar interpolation-based pipelines have a constant frame time.

The measurements on the GPU timeline show a slightly different story. The frame time for the deferred rendering pipeline still appears linear in the number of views. However, the frame time of the interpolation-based pipelines with source acquisition using depth peeling also appear linear in the number of views. But, with a much lower slope compared to the deferred rendering pipeline, which means that the interpolation-based pipelines scale better to larger number of views. Furthermore, the performance of the hierarchical epipolar pipelines appear to scale superlinearly. Though, the evidence is not very conclusive since the results at more than 100 views are missing. Most test cases related to the hierarchical epipolar pipeline could not complete their execution within reasonable time. Combined with the better performance of the center and dual epipolar pipelines in all test cases, it was decided not to wait for the probably uninteresting results. It seems the hierarchical epipolar pipelines could improve on the performance of the deferred pipeline, but only in cases with very high vertex and triangle counts. This can be seen in the test cases marked with San Miguel (High) and San Miguel (Extreme) at roughly 75 views.

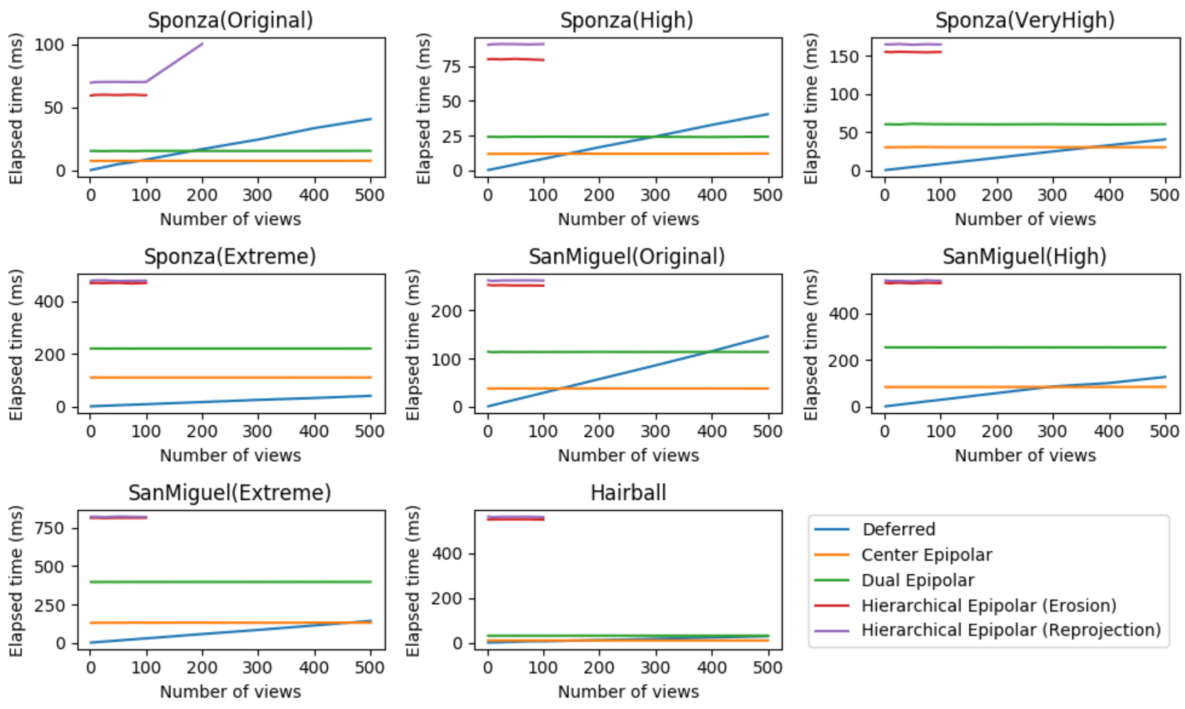

Figure 4.2. *For all test scenes, a comparison between the measured rendering pipelines. The shown times are measured on the CPU. No optimizations or extensions were active for the shown test cases. Note, the graphs are scaled differently.*

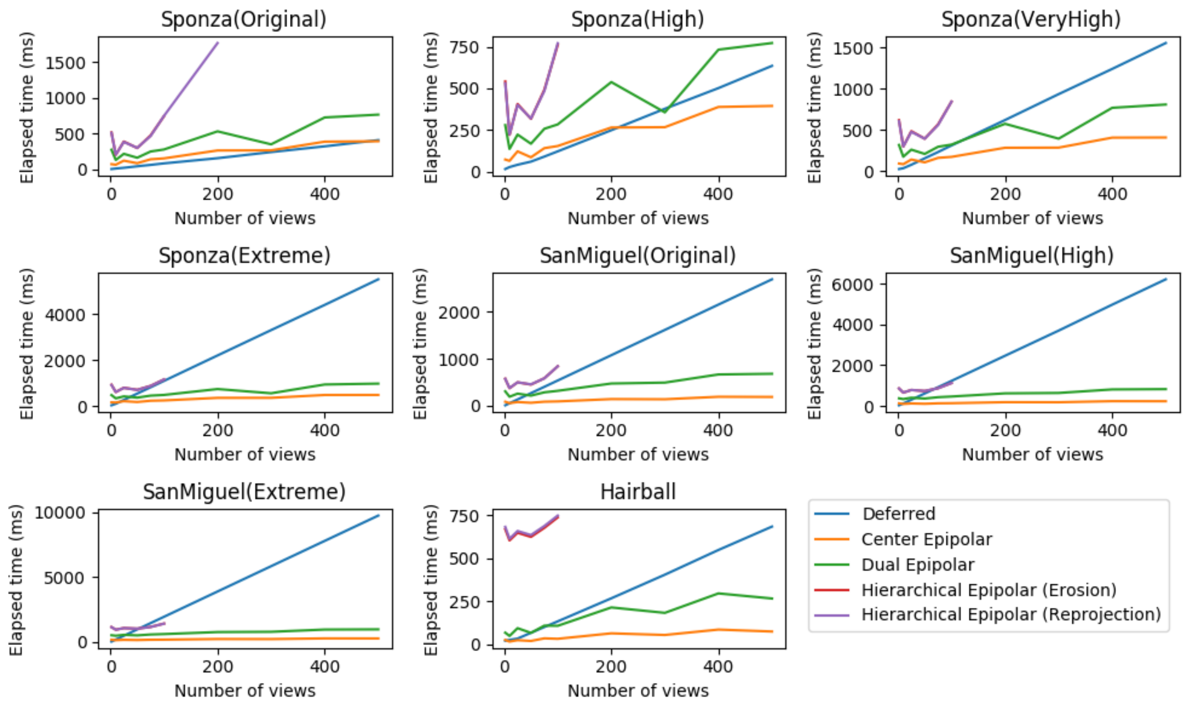

Figure 4.3. *For all test scenes, a comparison between the measured rendering pipelines. The shown times are measured on the GPU. No optimizations or extensions were active for the shown test cases. Note, the graphs are scaled differently.*

Deferred pipeline

Figure 4.4 shows the accumulated time to render all views in a single frame, for the different test scenes, using the deferred rendering pipeline. It shows a perfectly linear correlation between the number of views and the elapsed time. Furthermore, the slope of the line increases as the scene complexity increases, meaning the time spend per view increases. This holds for both the CPU and GPU timelines.

Figure 4.5 shows a breakdown for the time spend in the individual steps to render one view. The specific breakdown comes from a set of 500 views, which were generated without any additional options active, such as optimizations or specularity. Nonetheless, this behavior is typical for test cases with differing numbers of views. It illustrates the geometry buffer stage has the most notable impact on the processing time and becomes more dominating with higher fidelity scenes. This stage only consists of a simple transformation and rasterization of the scene geometry, and several texture fetches to associate the geometry data with color information. Though not shown in any figure, this also holds true with specularity activated.

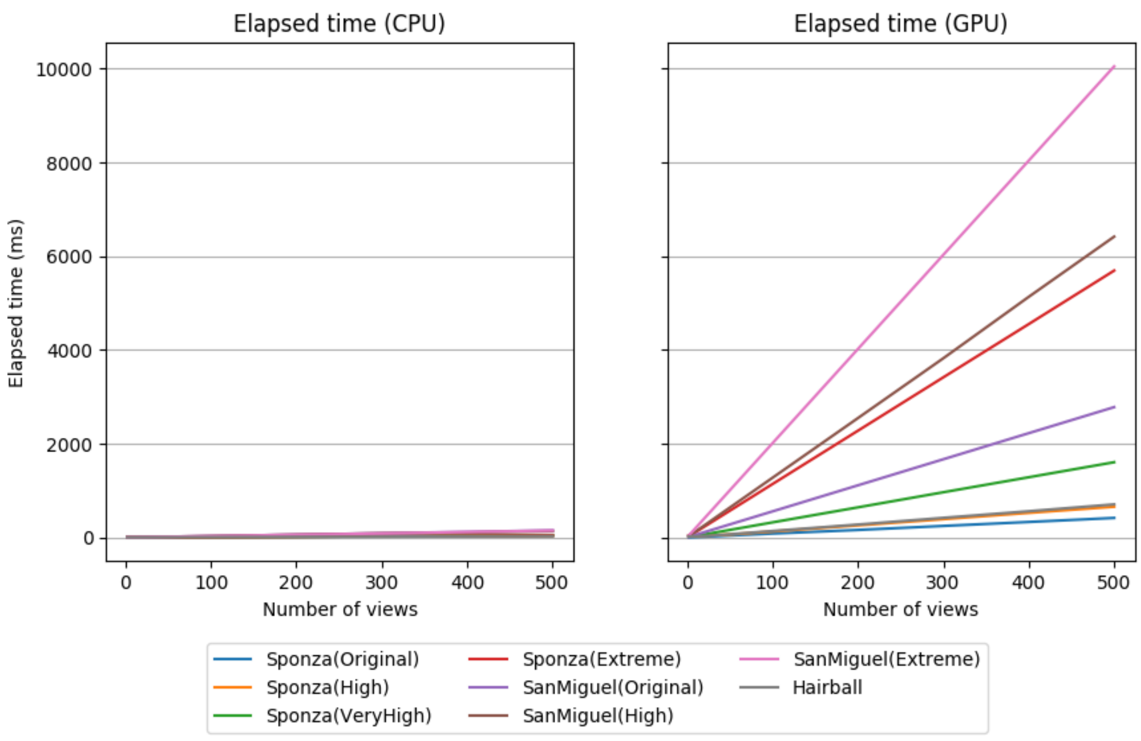

Figure 4.4. *Measured performance for the deferred rendering pipeline across the different scenes. This shows how the deferred rendering pipeline scales with the different scenes. For these measurements optimizations and additional options were turned off.*

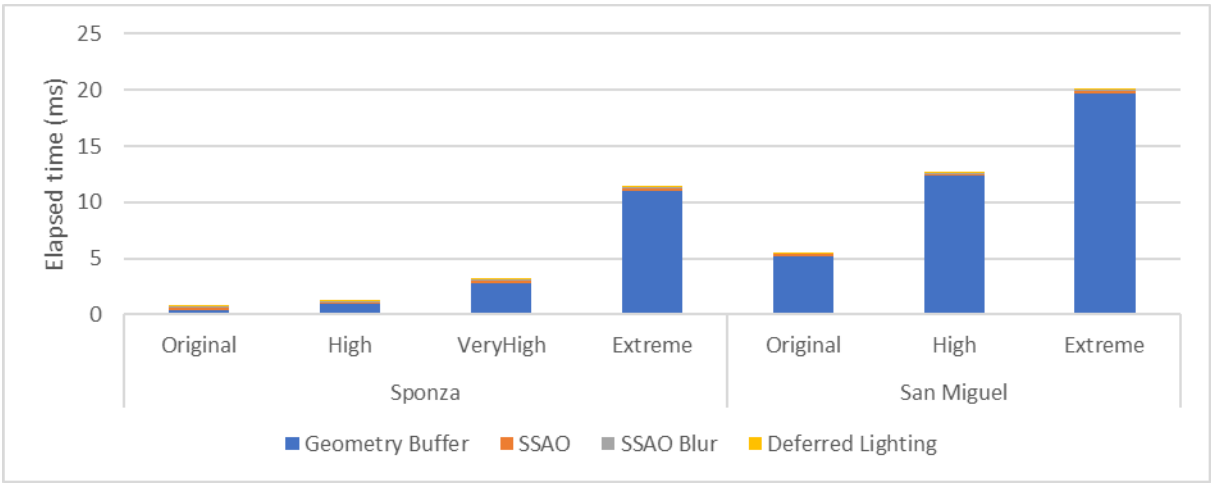

Figure 4.5. *Breakdown showing the elapsed time spend in the individual steps when rendering a single view using the deferred rendering pipeline. This was measured on the GPU timeline. Optimizations or specularity was not active for this test case.*

Center epipolar pipeline

Figure 4.6 shows how the center epipolar pipeline fares for the different scenes. One observation is that the measurements for the different Sponza scenes follow a very similar pattern. Equally, the performance measurements with the San Miguel scene also follow a similar pattern.

Additionally, it is also noticeable that the Sponza and San Miguel scenes, compared to each other, lead to different growth rates. The graphs corresponding with the Sponza scene have a steeper slope. If this is crosschecked against the amount of geometry, then it is possible to conclude that higher complexity scenes lead to better performance and better scalability since the San Miguel scenes consist of more vertices and triangles than the Sponza scenes.

Also, note the lower frame time for 10, 25, 50 or 75 views. Not only do most test cases with the center epipolar pipeline suffer from this peak in frame time, but this pattern is also visible with the dual epipolar and hierarchical epipolar pipelines. The results were checked for outliers in the measurements, but no evidence for this was found since usage of the median or discounting the n highest and lowest timings resulted in marginally different timings.

Figure 4.7 shows how the performance changes under the different optimizations and extensions. Only primitive coalescing significantly improves the performance and seems to reduce the slope of graph. As such, the variable time cost per view is slightly reduced instead of the constant overhead. Additionally, specularity does not seem to negatively impact the performance of the algorithm.

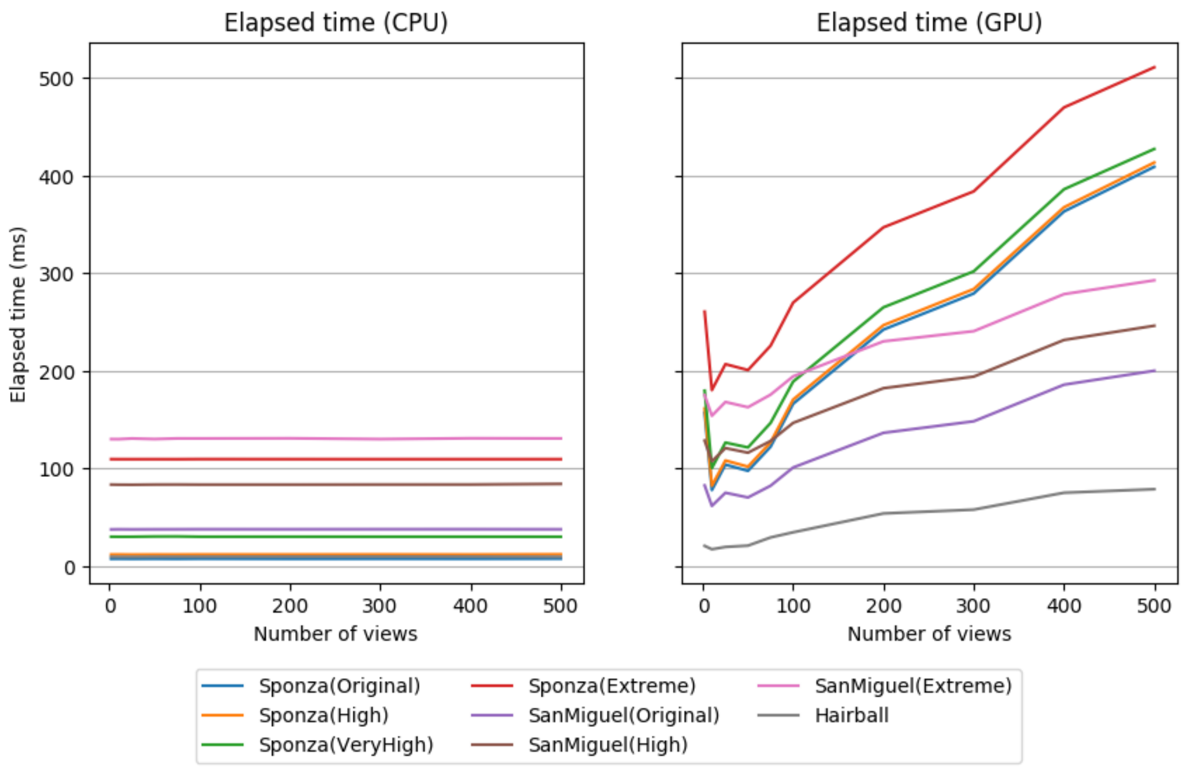

Figure 4.6. *Measured timings for the different scenes using the center epipolar pipeline. For these measurements optimizations and extensions were turned off.*

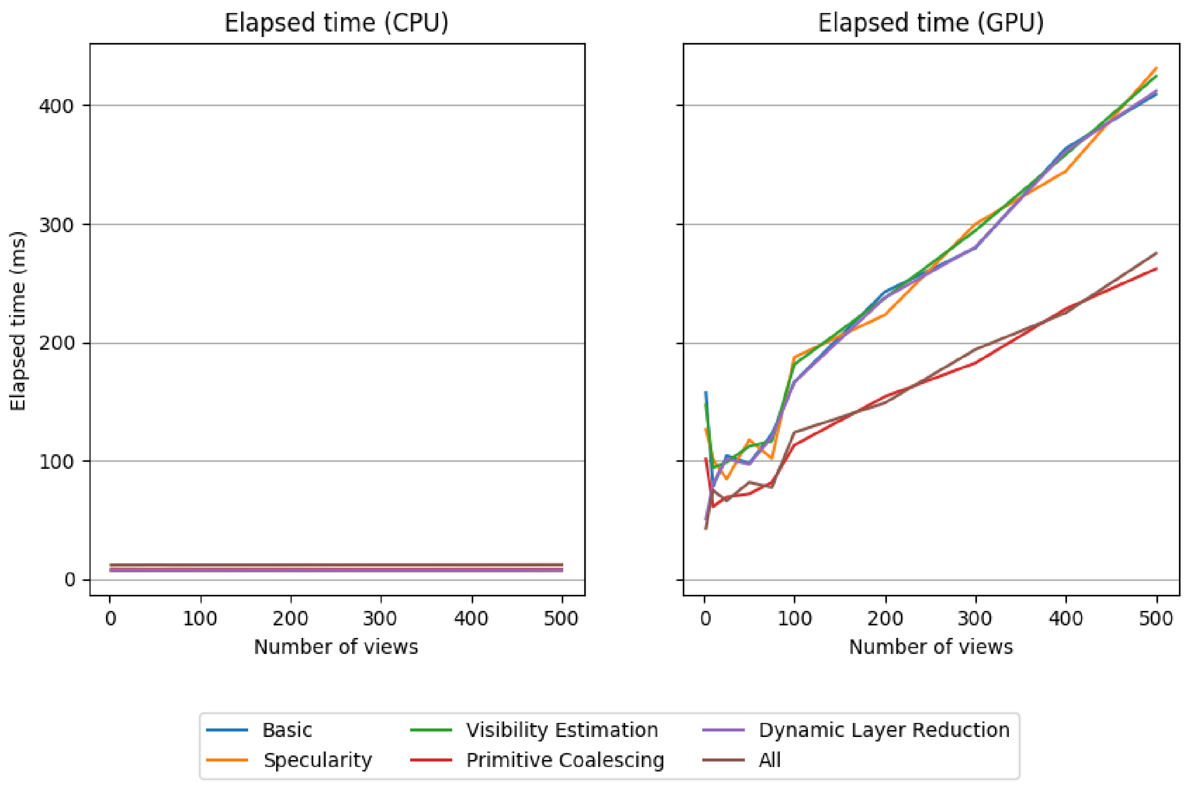

Figure 4.7. *Measured performance of the center epipolar pipeline under the optimizations and extensions. Each option has only one optimization active, except for "Basic", which corresponds with no active optimizations or extensions, and "All", which corresponds with all optimizations actived. The rendered scene was Sponza (Original).*

Dual epipolar pipeline

The dual epipolar pipeline is built on the same components as the center epipolar pipeline. Instead of performing depth peeling on the center view, the left- and rightmost views are used as source for view interpolation. The interpolation stage shares the same code and is thus similar. For the different test scenes, Figure 4.8 shows the accumulated times to render complete sets of views in a single frame for the dual epipolar pipeline. This is with no optimizations or extensions active. The frame times exhibit similar behavior, but are slightly higher, compared to the center epipolar pipeline, with exception of the reduced frame time at 300 views. The timing breakdown of the exceptional test case reveals that the interpolation for the views from the rightmost source view takes significantly less time, and this happens consistently over 32 rendered frames of which the graph shows the average.

A second difference between the center and dual epipolar pipelines is the difference in scaling behavior for the Sponza and San Miguel scenes. Figure 4.6 displays a clear distinction showing that the Sponza scene results in worse scaling behavior compared to the San Miguel scene, for the center epipolar pipeline. On the contrary, the dual epipolar pipeline contains a similar but far less pronounced effect as is shown in Figure 4.8.

Figure 4.9 shows the consequences of the different optimizations and extensions when rendering the Sponza (original) scene. Dynamic layer reduction performs similar as with the center epipolar pipeline and does not appear to reduce the frame time. Visibility estimation evidently does seem to reduce the frame time significantly, and primitive coalescing has the largest impact on the frame time.

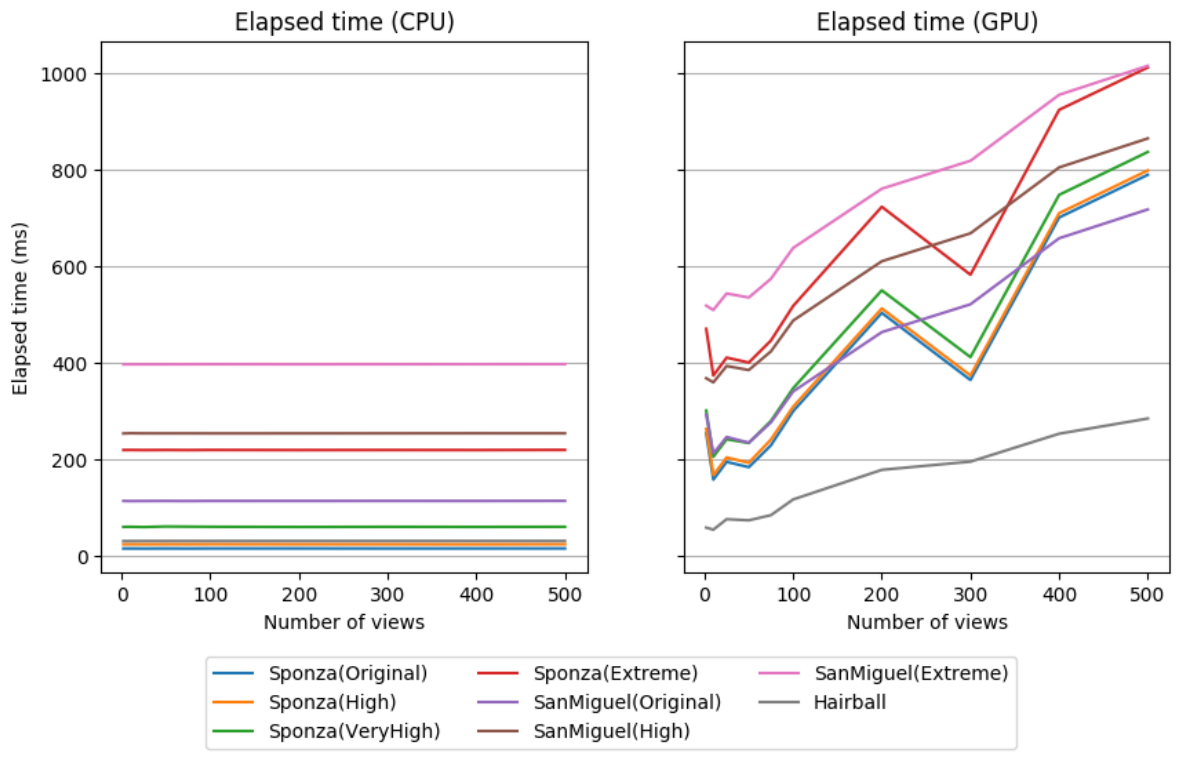

Figure 4.8. *Measured performance for the dual epipolar pipeline rendering the different scenes. For these measurements optimizations and additional options were turned off.*

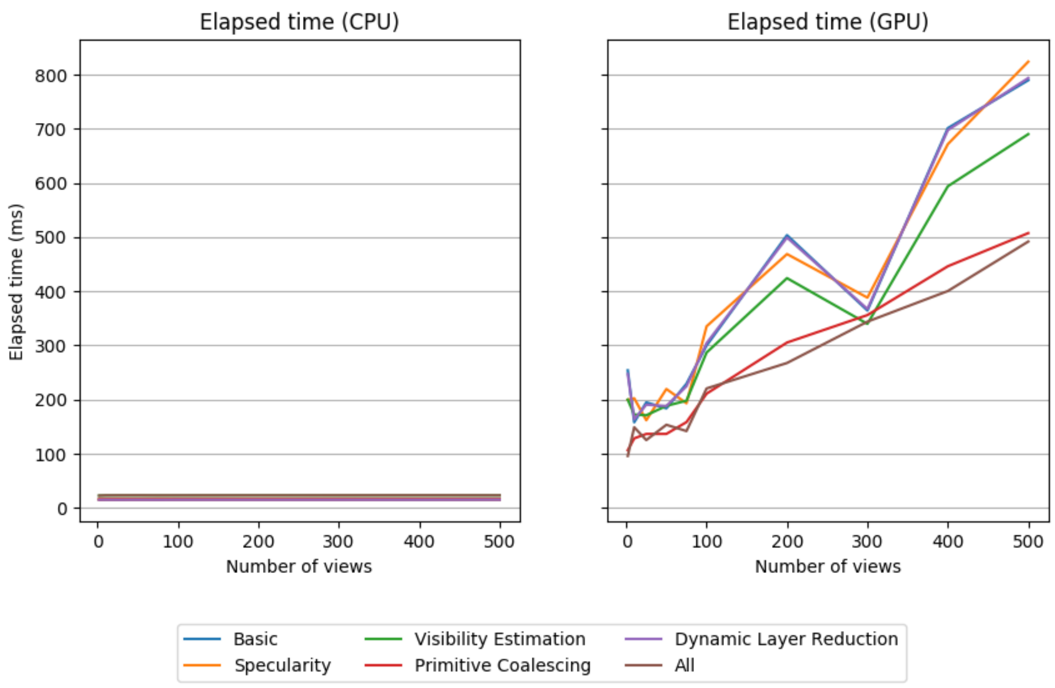

Figure 4.9. *Measured performance for the dual epipolar pipeline with the different optimizations and extensions active. Each option has only one optimization active, except for "Basic", which corresponds with no active optimizations or extensions, and "All", which corresponds with all optimizations actived. The rendered scene is Sponza (Original).*

Regarding the dual epipolar pipeline, Figure 4.10 displays a breakdown of the time cost for each step. It shows, the cost for epipolar rendering is significant compared to depth peeling, whereas the cost for depth peeling increases with increasing scene complexity. For unusually complex scenes, the depth peeling accounts for nearly half the frame time. Furthermore, the raw values for the graph show the combination of both primitive coalescing and visibility estimation is less effective than their respective reductions in frame time.

In Figure 4.11 a breakdown is given for an increasing number of views. For 2 views, the cost of epipolar rendering is 102 ms, and averages to 51 ms per view. At 500 views, the cost of epipolar rendering is roughly 599 ms, or 1.19 ms per view. This is significantly less than the deferred pipeline, which uses a constant 12 ms to render a single view for the same scene.

Before moving to the test results of the hierarchical epipolar pipeline, a note on additional tests performed with the dual epipolar pipeline rendering 2048 low resolution images (256 x 256 px). The underlying reason is that the described test configuration of Section 4.2 might be biased towards the deferred rendering pipeline since it is more suitable for generating a relatively small number of views at high resolution. However, additional testing was limited due to being an afterthought.

Nonetheless, the measured results are shown in Figure 4.12. The shown results correspond with a test case whereby the Sponza (Original) scene is rendered and no optimizations or extensions are active. The original test cases reported times that were similar to or higher than the frame times for the baseline rendering pipeline. However, at high view counts with low resolution images, the dual epipolar pipeline reports significantly lower frame times compared to the baseline with 198.61 ms for the deferred rendering pipeline and 100.89 ms for the dual epipolar pipeline. Furthermore, the difference in frame times increases for more complex scenes. Although, not shown in this report, the dual epipolar pipeline recorded frame times 8.6x lower when rendering the Sponza (High) scene. The deferred rendering pipeline reported a frame time of 858.82 ms, whereas the dual epipolar pipeline had a frame time of 100.43 ms.

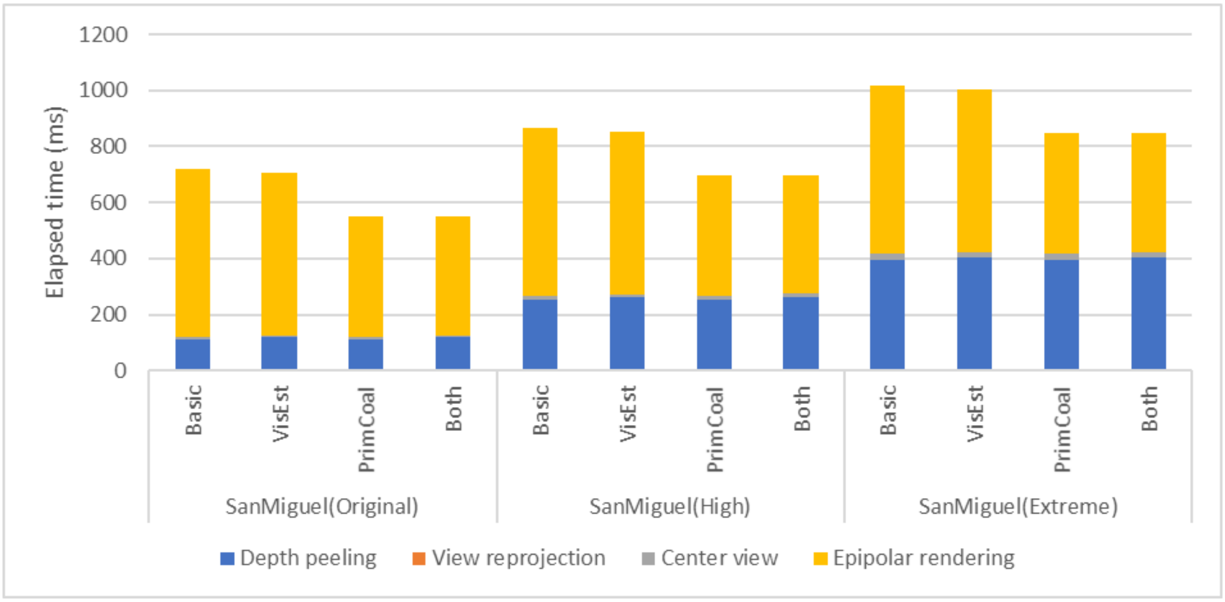

Figure 4.10. *Breakdown, of the time spend on the GPU, into the separate steps for the generation of a single set of views using the dual epipolar pipeline. 500 views are generated per set, and options are defined as follows: Basic = no optimizations, VisEst = visibility estimation, PrimCoal = primitive coalescing and Both = visibility estimation and primitive coalescing.*

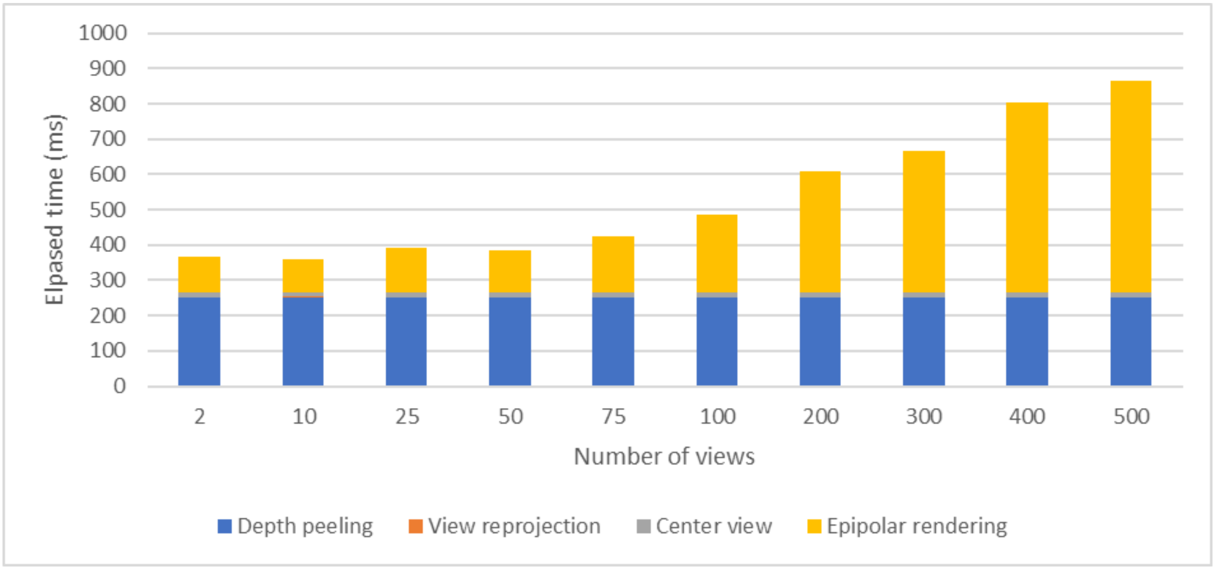

Figure 4.11. *Breakdown of the time spend on the GPU into the separate steps for the generation of the view sets using the dual epipolar pipeline. This figure shows how the spend time increases for higher view counts. The shown graph corresponds with the San Miguel (High) scene.*

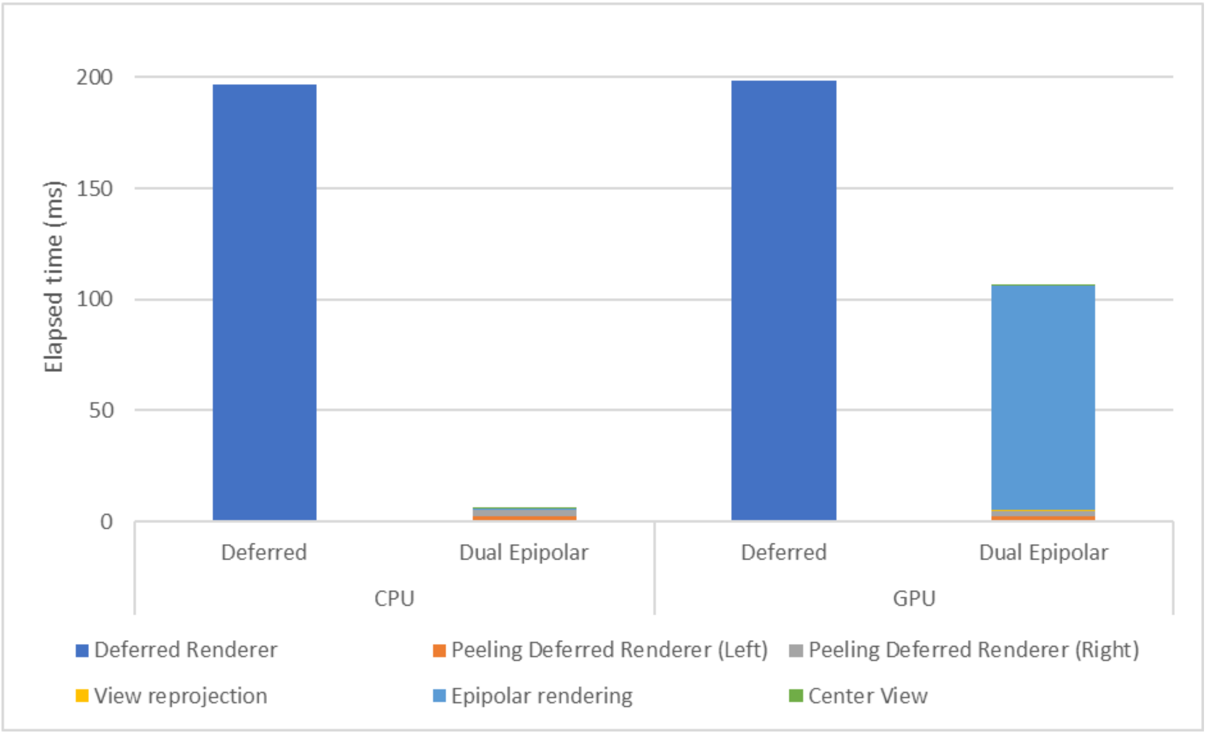

Figure 4.12. *Comparison of the baseline deferred rendering pipeline against the dual epipolar interpolation pipeline. For this test case, 2048 low resolution views (256 x 256 px) were rendered with no optimizations or extensions active. The rendered scene was Sponza (Original).*

Hierarchical epipolar pipeline

The source view acquisition stage for the hierarchical epipolar pipelines is fundamentally based on a different method compared to both pipelines using depth peeling for source view acquisition. Thus, it is expected to see a difference in the performance. Figure 4.13 shows the measured timings for the hierarchical epipolar pipeine with the erosion-based image quality metric, and no optimizations or extensions active. The performance is notably worse than the dual epipolar pipeline, with the worst-case difference being approximately 500 ms.

Figure 4.14 shows a breakdown of the time spent for the hierarchical epipolar pipeline (erosion). Compared to the dual epipolar pipeline, a lot more time is spent capturing the source views. For hierarchical epipolar pipelines this stage is denoted by *“Deferred Render PVS”*. A second observation is the correspondence between the CPU and GPU timelines for the source view acquisition stage. The time on both timelines is very similar.

The hierarchical epipolar pipeline with reprojection based image quality measure exhibits similar behavior to the pipeline with the erosion-based measure. Figure 4.15 displays the performance for the different test scenes without any optimizations and extensions active. A breakdown of a single frame is displayed in Figure 4.16, and shows that the different image quality measures result in similar algorithm behavior.

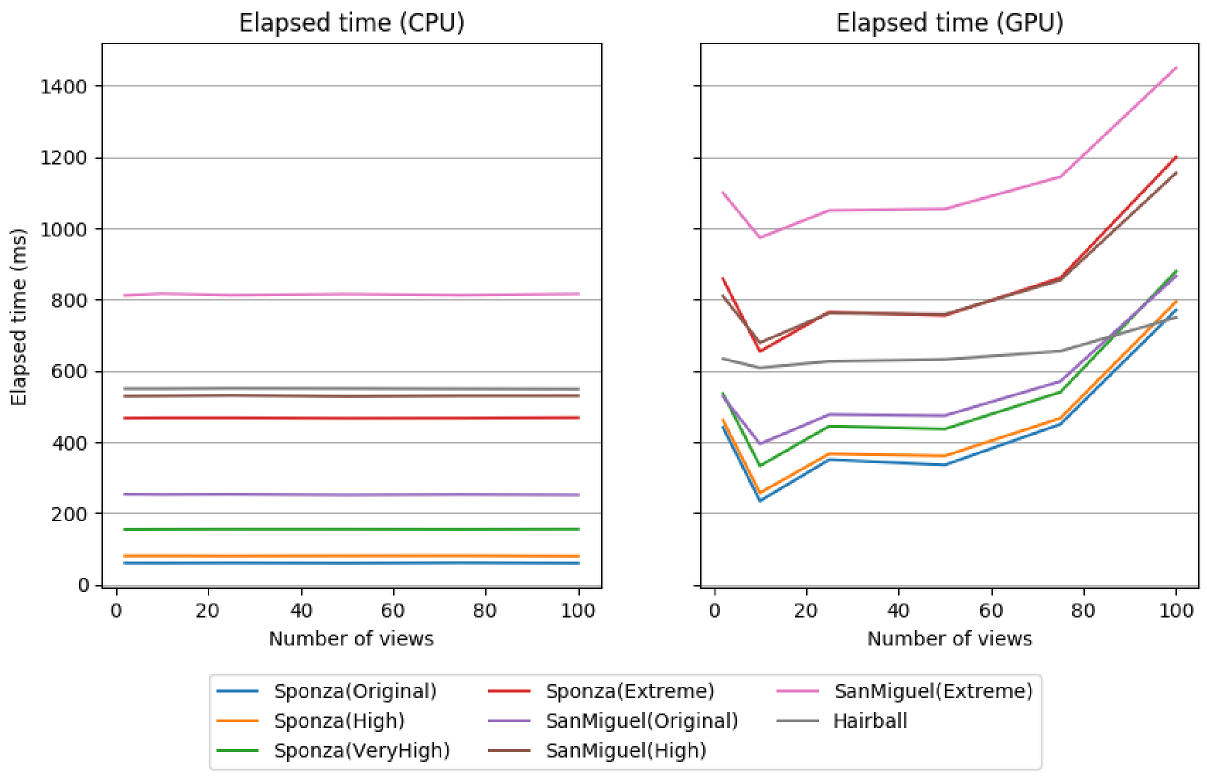

Figure 4.13. *Measured performance for the hierarchical epipolar pipeline (Erosion) across the different scenes. For these measurements optimizations and extensions were deactivated.*

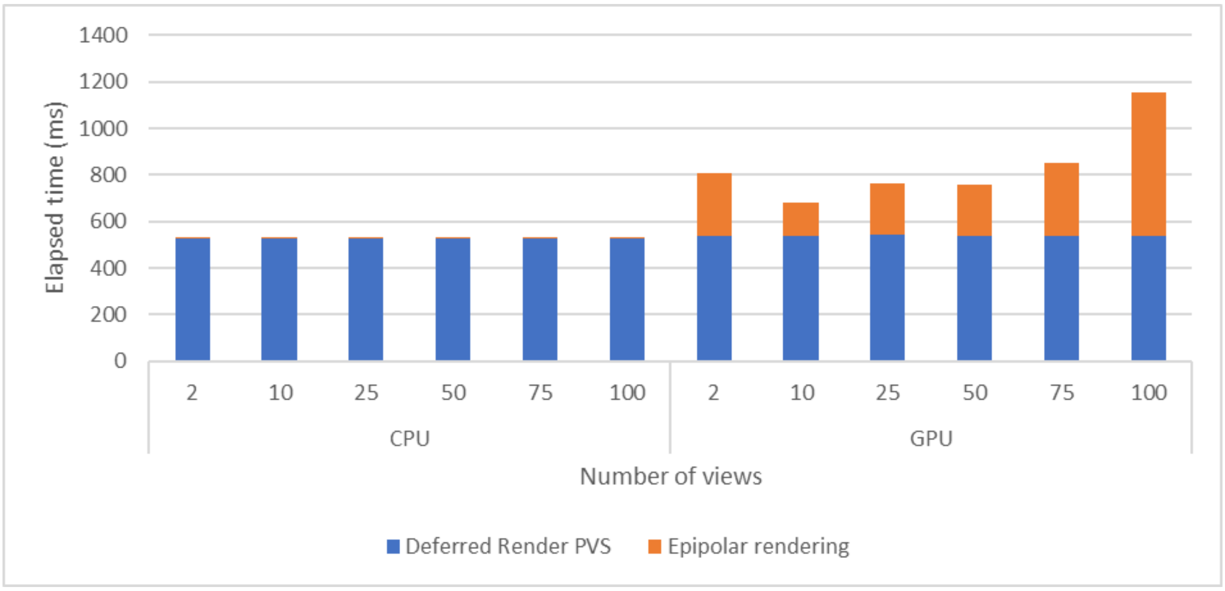

Figure 4.14. *Breakdown of timings in individual algorithm steps. The measurements were taken with the hierarchical epipolar pipeline (erosion). The shown test case used the San Miguel (High) scene. Any optimizations and extensions were turned off.*

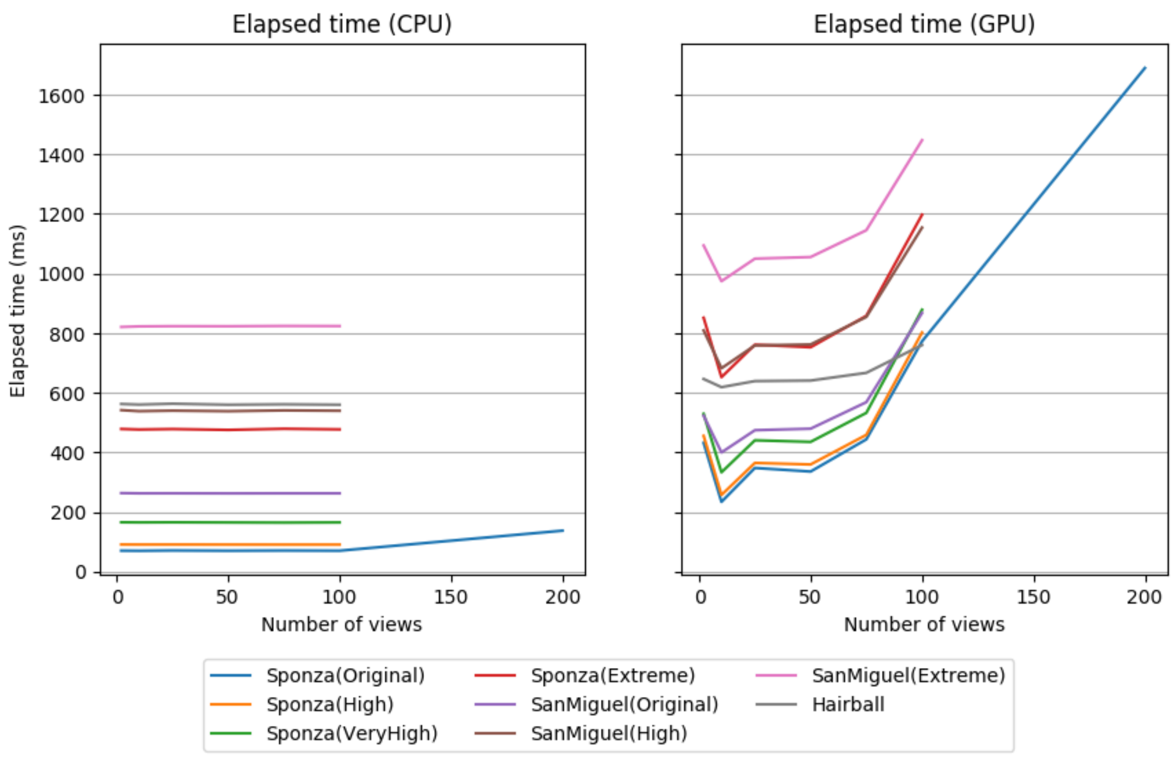

Figure 4.15. *Measured performance for the hierarchical epipolar pipeline (Reprojection) rendering the different scenes. For these measurements optimizations and additional options were turned off.*

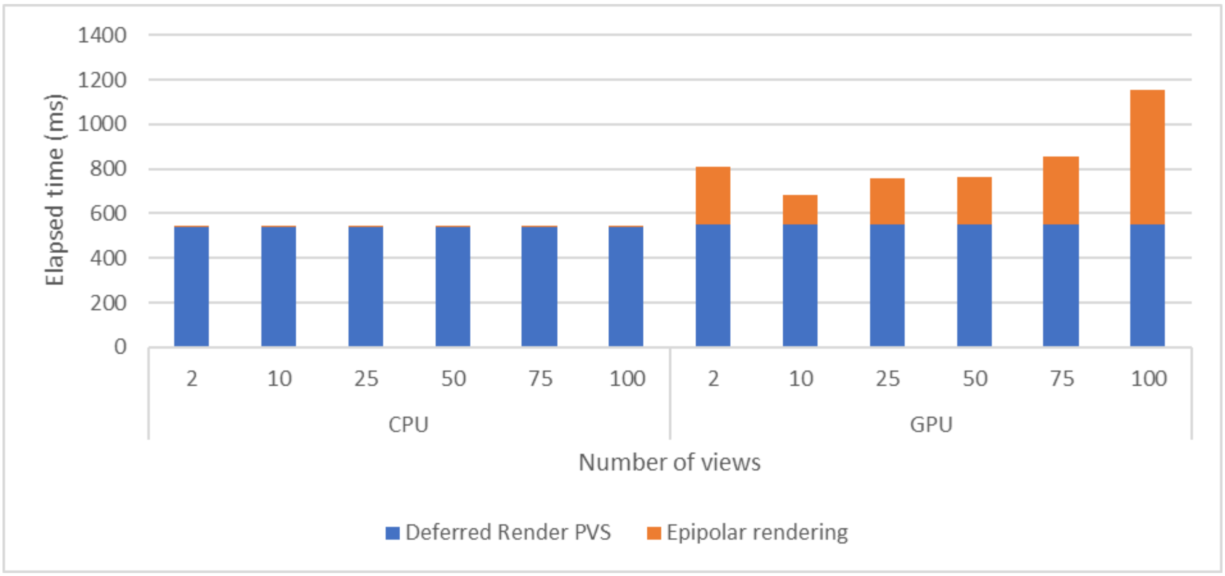

Figure 4.16. *Breakdown of the timings from a single frame into individual algorithm steps. The measurements were taken with the hierarchical epipolar pipeline (reprojection). The shown test case used the San Miguel (High) scene. Any optimizations and extensions were turned off.*

The previous graphs show the distinct performance deficit compared to the center and dual epipolar pipelines. Opposed to the optimizations implemented for the center and dual epipolar pipeline, both hierarchical epipolar pipelines had less optimizations implemented. Only primitive coalescing was implemented which previously led to a significant reduction in the measured frame times. For the hierarchical epipolar pipelines the reduction is less pronounced. Additionally, support for specularity was also implemented for the hierarchical epipolar pipelines. Figure 4.17 and Figure 4.18 show the impact of the two features.

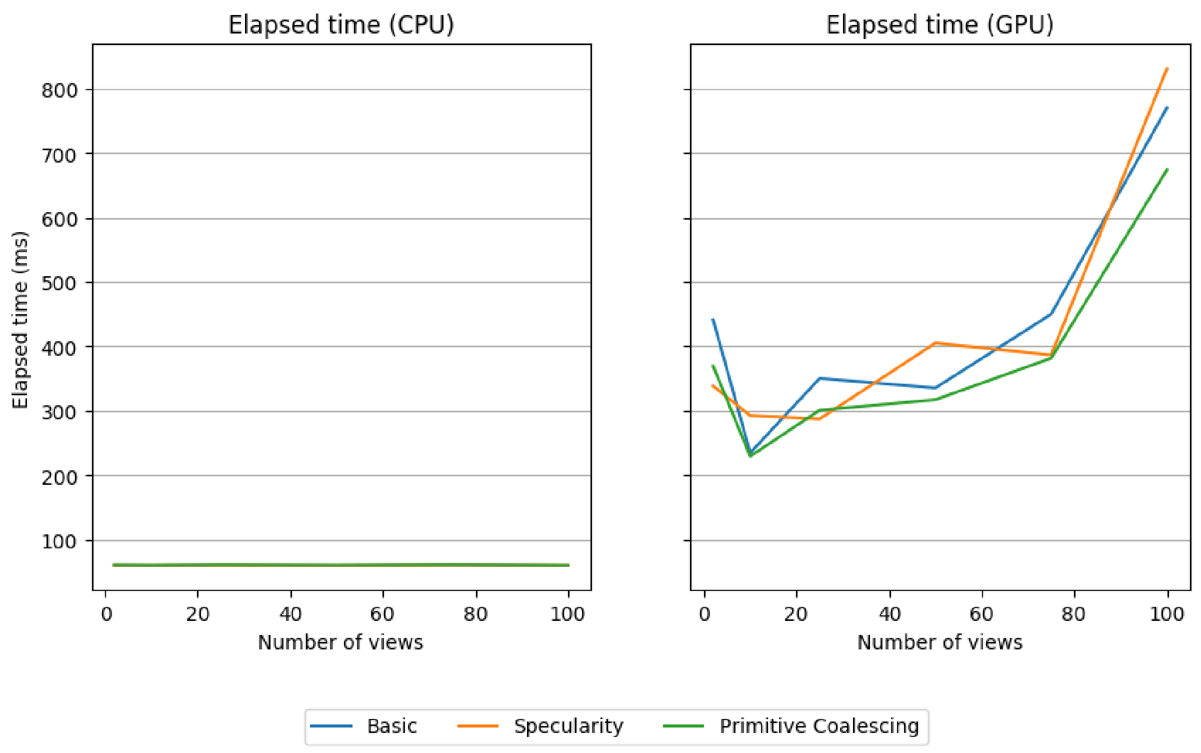

Figure 4.17. *Impact of the optimizations and extensions on the frame time for the hierarchical epipolar pipeline (erosion). The only optimization implemented was primitive coalescing. Additionally, specularity is also supported. The test scene was Sponza (Original).*

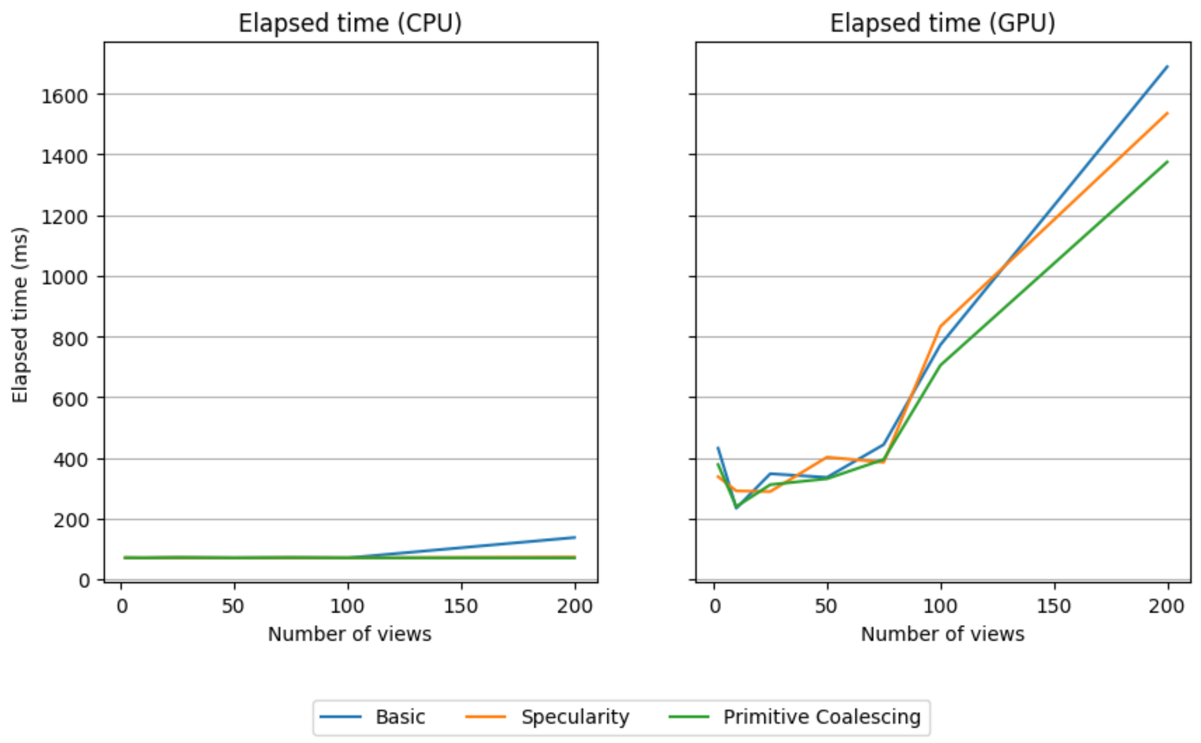

Figure 4.18. *Impact of the optimizations and extensions on the frame time for the hierarchical epipolar pipeline (reprojection). The only optimization implemented was primitive coalescing. Additionally, specularity is also supported. The test scene was Sponza (Original).*

4.5 Measured image quality

The image quality measurement experiments have resulted in several quality traces. Table 4.4, Table 4.5 and Table 4.6 show an overview of the measured image quality for the different test scenes. The tables respectively refer to the center view acquisition, dual view acquisition and hierarchical view acquisition render pipelines combined with epipolar view interpolation.

The tables give the average, minimum and maximum PSNR and SSIM values. The values for a given test scene are derived from all generated images for a fixed configuration. For example, the values for the Sponza test case are computed from the images in the combined view sets corresponding with 2, 10, 25, etc. views. Unless otherwise specified, the configuration was such that primitive coalescing was not active, and specularity was also not activated. If an option was activated, it's tagged onto the model name.

Center epipolar pipeline

Table 4.4 displays the accumulated PSNR and SSIM values for the center epipolar pipeline. For respectively the Lucy and Mitsuba scene, the PNSR fall in the mid-twenties and mid-to-high thirties. The SSIM values are upward of 98%. On the contrary, the different test configurations for the Sponza scene resulted in SSIM values in the mid 80% range.

The accumulated results also show the center epipolar pipeline can generate perfect images according to the SSIM measure. Figure 4.19 shows an exemplary, but typical, spread of the image quality within a set of images. The center view has the highest image quality, and the image quality degrades for views further away from the center. More so, according to SSIM, the center image is a perfect match with the baseline. The PSNR also reports a high image quality, although not perfect. Additionally, when specularity is added, the image quality of the center view drops.

Table 4.4. *Overview of quality measurement results for the rendering pipeline configured as center view acquisition with epipolar rendering. Unless otherwise stated, the results are given for the basic pipeline, and without specularity.*

Center Epipolar						
Model	**PSNR**			**SSIM**		
	Avg.	**Min.**	**Max.**	**Avg.**	**Min.**	**Max.**
Hairball	19.405	9.630	23.616	0.911	0.668	1.000
Lucy	37.321	32.312	41.281	0.991	0.984	1.000
Lucy (Specularity)	37.321	32.312	41.281	0.991	0.984	1.000
Mitsuba	25.174	17.998	36.341	0.983	0.950	1.000
Mitsuba (Specularity)	25.187	17.995	51.443	0.982	0.950	1.000
Sponza	22.598	15.466	31.227	0.869	0.716	1.000
Sponza (Primitive Coalescing)	22.634	15.467	53.929	0.868	0.716	0.998
Sponza (Specularity)	22.321	15.410	38.546	0.866	0.713	1.000

Note that it would be unfair to compare the image quality for the different scenes test scenes because the center epipolar rendering pipeline suffers from an inherent problem. It uses only images captured from the center camera to generate all views. As it stands, a part of the images that correspond to several left- and right-side cameras miss information which cannot be reconstructed. In turn, the image quality of the Sponza scene drops because it fills an entire image whereas the Lucy scene, and to a lesser degree, the Mitsuba scene only cover a part of the image. See Figure 4.20 for several example images showing why Lucy and Mitsuba are not suffering from the mentioned problem, while the Sponza scene reports a lower image quality.

When the results are compensated for the lacking information by leaving part of the image out of the image quality comparison, then the average PSNR increases to 29.740 for the basic pipeline. The averages for the pipelines configured with primitive coalescing and specularity respectively increase to 29.737 and 28.898. The average SSIM also increases to 0.915, 0.914 and 0.913 for respectively the basic pipeline, the pipeline with primitive coalescing activated and the pipeline with specularity activated.

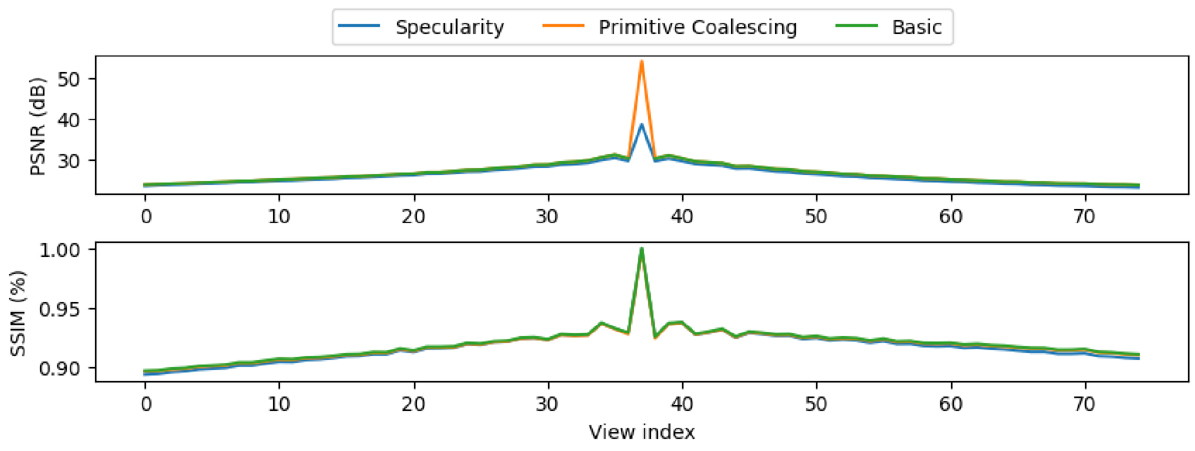

Figure 4.19. *Individual image quality trace. It shows the center epipolar pipeline can generate perfect images, and degrading image quality for views further from the center. Test scene: Sponza (original); render pipeline: center epipolar; view set size:75 views.*

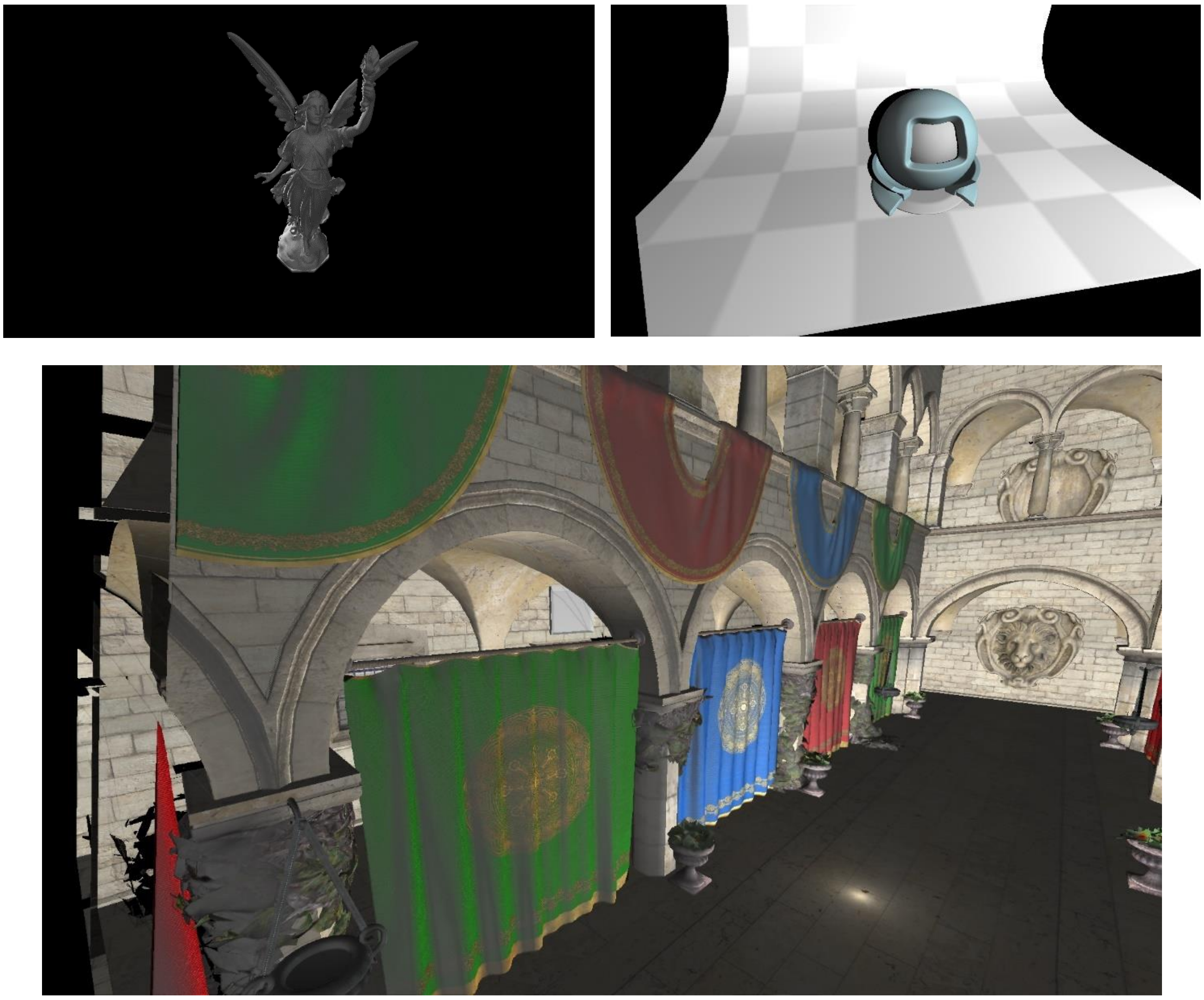

Figure 4.20. *Several example images of the leftmost views, captured using the center epipolar render pipeline. The shown test scenes are Lucy (top left), Mitsuba (top right) and Sponza (bottom). The sponza scene is missing part of the left side due to missing information in the interpolation source views. Lucy does not suffer from this problem since only part of the image is covered, and Mitsuba is only missing a small section.*

Note, the coverage of a scene within an image is a large contributor to the high PSNR and SSIM values for the Lucy and Mitsuba scenes. Large sections of the scenes are background and are considered equal by the image quality metrics. However, this does not measure the quality of the algorithms. As such, these scenes are not preferred when judging the image quality of the view interpolation algorithm.

That said, another noticeable result is the image quality of the Hairball scene. Like the test scenes Lucy and Mitsuba, the Hairball does not cover an entire image. However, the Hairball related PSNR and SSIM values are the lowest of all test scenes, especially the minimum values. See Figure 4.21 for example images of the Hairball test scene, rendered using the center epipolar interpolation pipeline with no primitive coalescing or specularity.

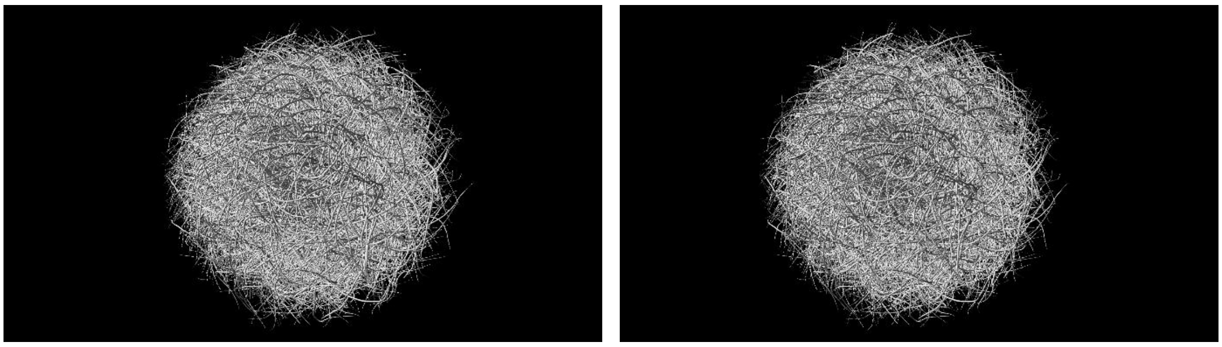

Figure 4.21. *Examples of left- and rightmost view of the Hairball scene. The set of views from which these images originated contains 100 views.*

Dual epipolar pipeline

The dual view interpolation pipeline corrects the missing information of the previous render pipeline. A corresponding increase in SSIM values can be seen with the Sponza scene, shown in Table 4.5. Furthermore, the minimums are higher, and the maximums are slightly lower. See Figure 4.22 for an example of how the measured image quality relates to an actual view of the Sponza scene. The Lucy and Mitsuba scenes show comparable results to the center epipolar pipeline for PSNR and SSIM. Hairball has similar average and maximum PSNR values, but the minimum PSNR has doubled. The corresponding SSIM outcome is slightly lower.

Table 4.5. *Overview of quality measurement results for the rendering pipeline configured as dual view acquisition with epipolar rendering. Unless otherwise stated, the results are given for the basic pipeline, and without specularity. The models annotated with "mod." are generated based on a modified test to circumvent an edge case.*

Dual Epipolar						
Model	**PSNR**			**SSIM**		
	Avg.	**Min.**	**Max.**	**Avg.**	**Min.**	**Max.**
Hairball	18.746	17.753	22.607	0.895	0.867	0.957
Lucy	37.677	35.973	45.635	0.991	0.988	0.998
Lucy (Specularity)	37.677	35.973	45.635	0.991	0.988	0.998
Mitsuba	35.165	34.040	38.804	0.994	0.992	0.998
Mitsuba (Specularity)	35.138	34.027	38.787	0.993	0.992	0.998
Sponza	28.663	24.055	36.476	0.916	0.892	0.991
Sponza (Primitive Coalescing)	28.662	24.054	36.408	0.915	0.892	0.989
Sponza (Specularity)	28.595	24.030	36.377	0.916	0.892	0.991
Sponza (mod.)	33.023	31.781	39.080	0.945	0.924	0.993
Sponza (mod.; Primitive Coalescing)	33.015	31.777	39.001	0.944	0.923	0.991
Sponza (mod.; Specularity)	32.979	31.733	39.021	0.945	0.924	0.993

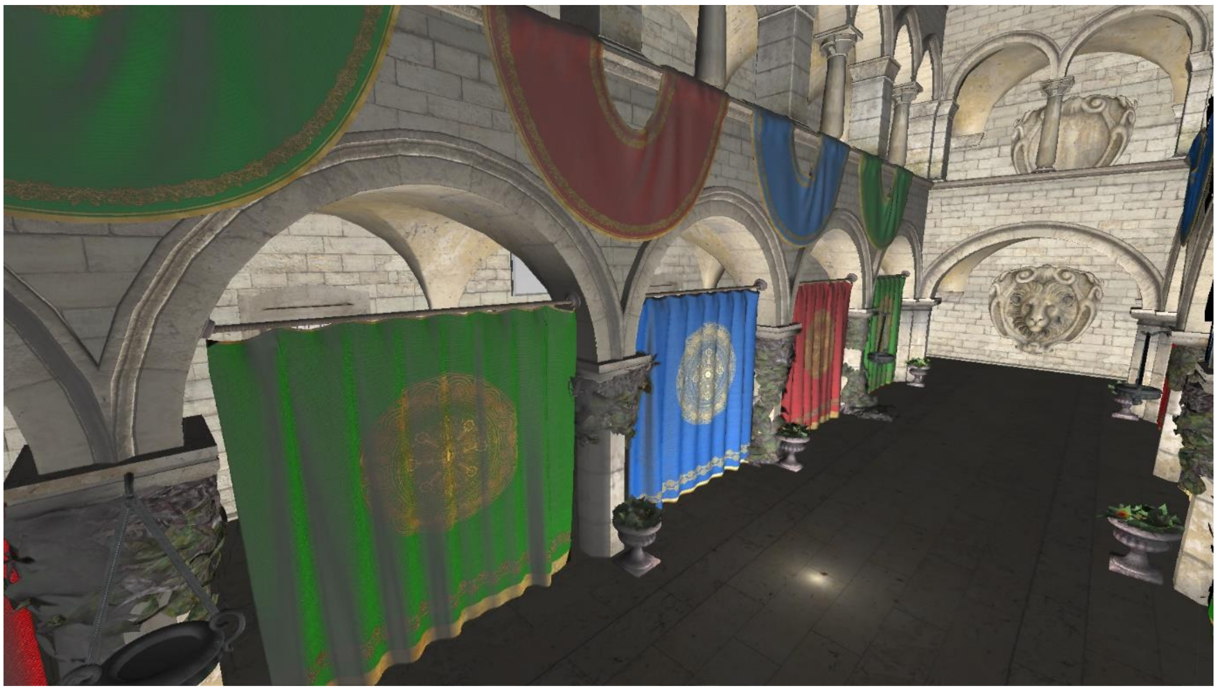

Figure 4.22. *Shows view 121 of 200 for a test configuration with the Sponza scene, rendered with the dual epipolar pipeline. View 0 corresponds with the leftmost view, and view 200 with the rightmost view.*

Another noticeable result is that the average values lie significantly closer to the minimum values than the maximums. This suggests that the bulk of the images have an image quality close to minimum and average quality, with some images having an image quality that can be considered outliers.

Looking at the measured quality of individual images, the quality drops for larger view sets. Figure 4.23 shows the image quality of individual images for a view set consisting of 100 views. The image quality is similar across the different views. Contrastingly, larger view sets have an observable decrease in image quality for the right half of the views, when considering test cases with the Sponza scene. Figure 4.24 and Figure 4.25 show the individual image quality for view sets containing 300 and 500 views. When the image sets containing these views are left out, then the values are as follows: the average, minimum and maximum PSNR are respectively 30.725, 29.517, 36.476. The resulting average, minimum and maximum SSIM values are respectively 0.929, 0.907, 0.991. These results are for the Sponza scene without any optional settings active. This suggests the test cases trigger an edge case with the dual epipolar pipeline, and this is also argued in Section 5.1. When the test cases are changed to account for these edge cases, then the results improve to respectively 0.945, 0.924 and 0.993 for the average, minimum and maximum SSIM values. Several example image quality traces are shown in Figure 4.26 and Figure 4.27.

Furthermore, the maximum SSIM values are not 1.000, or 100%. However, the views used as basis for the interpolation should have resulted in identity transformations, and thus identical views. When looking at the individual image quality within a set of views, as expected, the source views have the highest image quality, but are not perfect. For some examples, see the graphs displaying the individual image quality in Figure 4.23, Figure 4.24 and Figure 4.25.

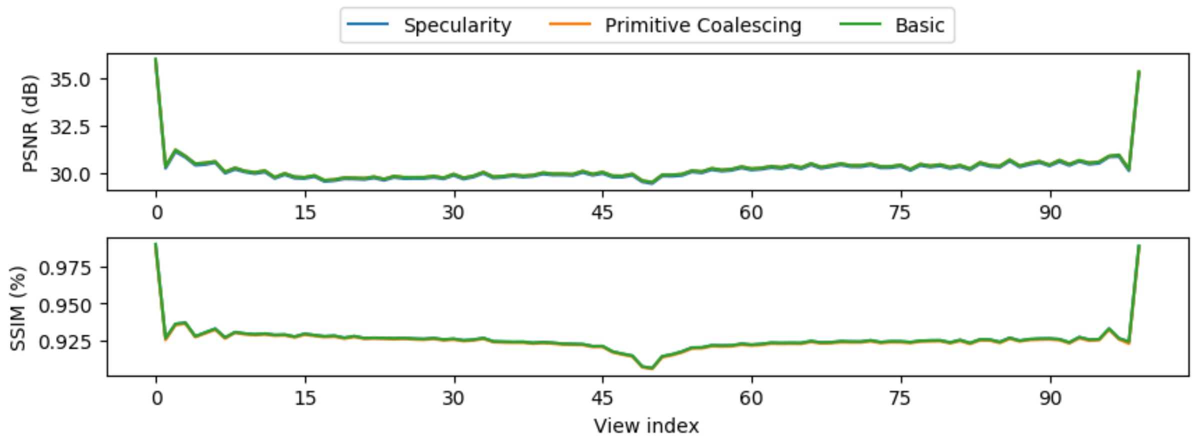

Figure 4.23. *Individual image quality of the Sponza test scene. The image set comprised 100 views, and the quality is relatively even over the complete image set. Note, the view index is a reference to a specific camera for which the views were generated. View index 0 corresponds with the leftmost camera, and index 100 represents the rightmost camera.*

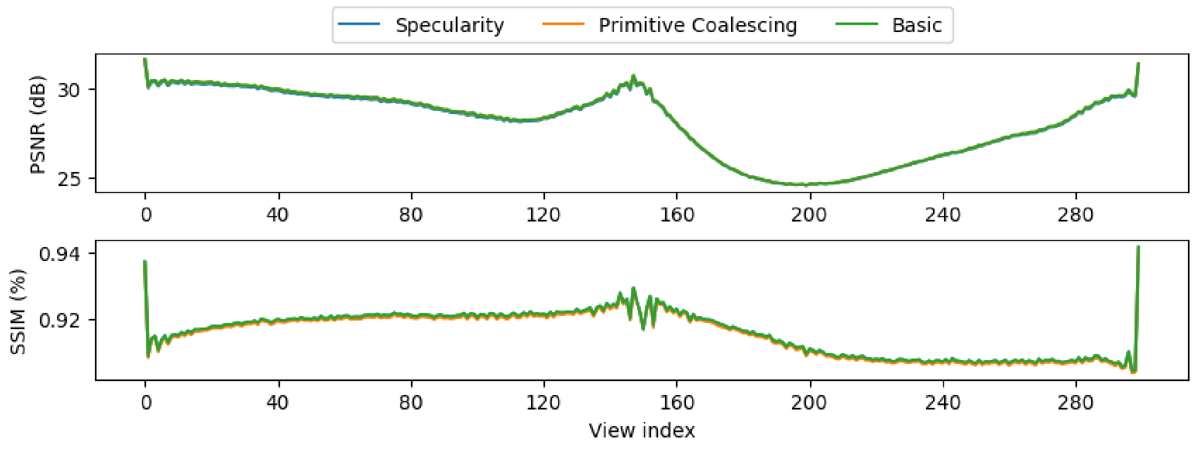

Figure 4.24. *Individual image quality of the Sponza test scene from an image set comprised of 300 views. The right half of the views shows a decline in image quality. Note, the view index is a reference to a specific camera for which the views were generated. View index 0 corresponds with the leftmost camera, and index 300 represents the rightmost camera.*

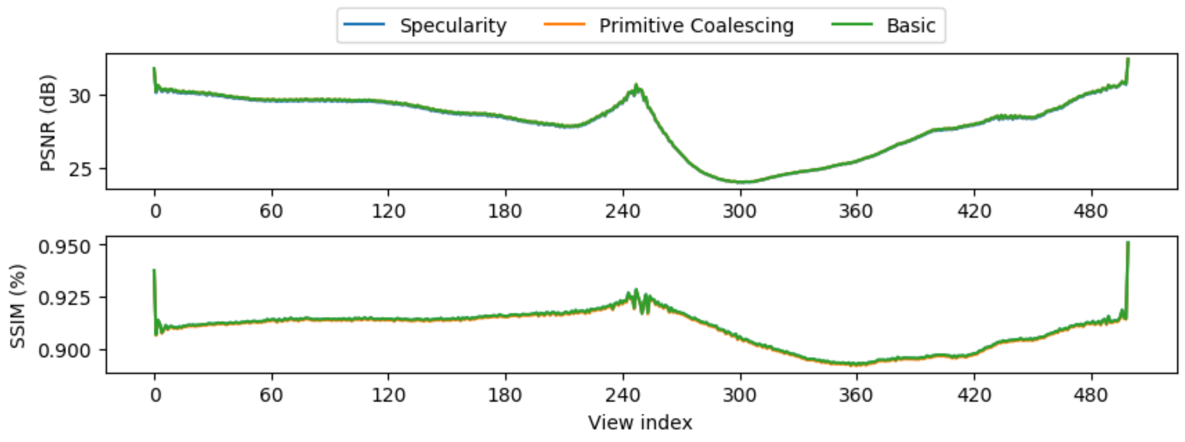

Figure 4.25. *Individual image quality of the Sponza test scene, whereby the image set comprised 500 views. Like the view set containing 300 views, the quality decreases for the right half of the images. Note, the view index is a reference to a specific camera for which the views were generated. View index 0 corresponds with the leftmost camera, and index 500 represents the rightmost camera.*

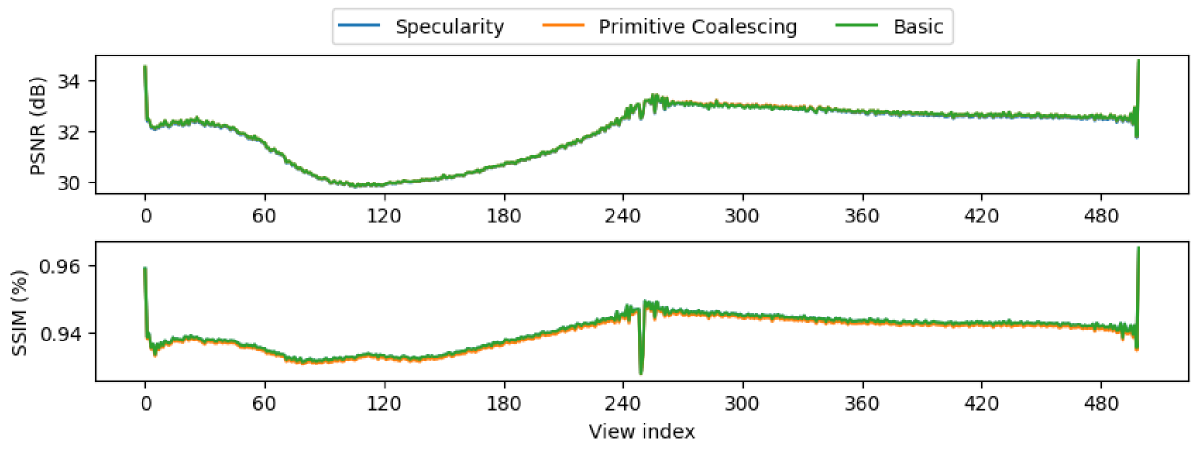

Figure 4.26. *Individual image quality trace with modified test case. The camera position and orientation were changed such that the view direction did not run parallel with a wall, which increased the image quality. However, from inspection of the images some holes were visible due to missing information. Test scene: Sponza; render pipeline: dual epipolar; view set size: 500 views.*

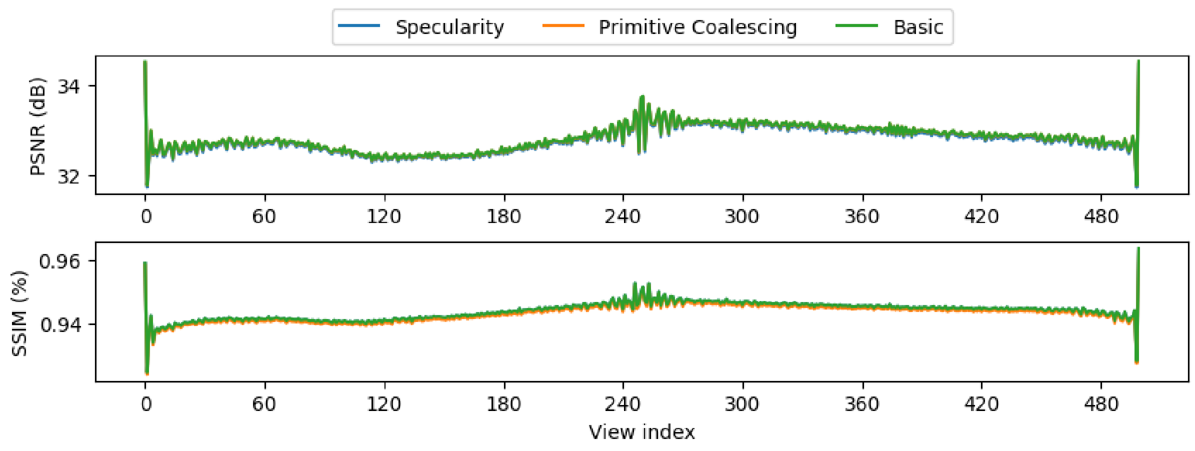

Figure 4.27*. Individual image quality trace with modified test case. Compared to the results from Figure 4.26, these results were generated using a smaller distance between cameras. This reduces hole forming and thus increases image quality. Test scene: Sponza; render pipeline: dual epipolar; view set size: 500 views.*

The image quality traces don't show how the interpolated images differ from the reference images. To this end, Figure 4.28 and Figure 4.29 show some examples of the absolute error in the generated images. The absolute error shows the interpolation introduces noise throughout the entire image, with most of the error located on the edges between different surfaces. For more examples, see Appendix A. Another anomaly in the image quality traces is the decreased image quality between the left- and rightmost views, and the views in between. The SSIM values for the images corresponding to the absolute error maps shown in Figure 4.28 and Figure 4.29 have a difference of roughly 2 percent when comparing the SSIM metric. However, the difference between the absolute error maps is minimal and mostly limited to the sides of the images.

Figure 4.28. *Absolute error of view 0 from the same viewset used to generate the graph shown in Figure 4.27. This view was generated using no optimizations or extensions. For more examples see Appendix A.*

Figure 4.29. *Absolute error of view 2 from the same viewset used to generate the graph shown in Figure 4.27. This view was generated using no optimizations or extensions. For more examples see Appendix A.*

Hierarchical epipolar pipeline

Table 4.6 gives an overview of the measured image quality of the hierarchical view acquisition-based interpolation pipeline. For the pipeline configured with the erosion-based image quality measure, the Sponza scene gives SSIM values around 0.85, with slightly lower values when primitive coalescing is active. However, when the test case is modified identical to the modification done for the previously discussed pipeline, then an increase is noticeable to SSIM values of roughly 0.90. A similar behavior can be observed for both image quality measure configurations of the hierarchical pipeline.

The most notable difference compared to the previously mentioned pipelines is the lower image quality according to SSIM. Whereas, the PSNR values are on par with the center and dual epipolar pipelines. However, a side-by-side comparison of several images (for the sponza scene) doesn't show any obvious or major errors. An exemplary side-by-side comparison of a single view for the different pipelines is given in Figure 4.33.

The quality of individual images within view sets show interesting patterns. Figure 4.30 has a noticeable drop in image quality for the images on the right side of the spectrum. Inspection of the images shows that some cameras clip into the geometry resulting in a different behavior compared to the baseline generated images. Additionally, the image quality peaks for a small set of views. However, based on results of the previous pipelines, the expectation would be to see peaks for the views used as source for the interpolation. But, it is known, the generation of this set of views used more than two views as source. An example of the expected pattern can be seen in Figure 4.31.

However, the expected image quality trace also contains a similar loss of image quality as with the dual epipolar render pipeline rendering the Sponza scene. This holds for both image quality prediction measures. Figure 4.32 shows an example of this behavior for the erosion-based measure.

Figure 4.31 does raise the suggestion that the pattern on the right half of the views should be similar on the left half of the views due to the slight bumps in quality visible around view 350 and view 440, which are likely to reference the used source views. If the test is adjusted to circumvent the edge case, then the result is as shown in Figure 4.34 and Figure 4.35. The changed tests show an improvement in image quality, and clearly show the view indices that were used as source view.

Table 4.6. *Overview of quality measurement results for the hierarchical view rendering pipelines. The first section employs the erosion-based quality measure, and the second section is based on the reprojection based quality measure. Unless otherwise stated, the results are given for the basic pipeline, and without specularity. The models annotated with "mod." are generated based on a modified test to circumvent an edge case.*

Hierarchical Epipolar (Erosion)						
Model	**PSNR**			**SSIM**		
	Avg.	**Min.**	**Max.**	**Avg.**	**Min.**	**Max.**
Hairball	12.705	11.876	13.508	0.743	0.726	0.756
Lucy	38.039	36.603	39.807	0.991	0.989	0.994
Lucy (Specularity)	38.039	36.603	39.807	0.991	0.989	0.994
Mitsuba	34.732	32.730	36.917	0.990	0.971	0.996
Mitsuba (Specularity)	34.629	32.691	36.746	0.990	0.971	0.995
Sponza	27.547	25.106	28.053	0.851	0.842	0.869
Sponza (Primitive Coalescing)	26.563	23.465	28.102	0.826	0.738	0.859
Sponza (Specularity)	27.178	24.993	27.683	0.851	0.841	0.871
Sponza (mod.)	30.262	29.351	30.793	0.902	0.896	0.909
Sponza (mod.; Primitive Coalescing)	30.224	29.402	30.950	0.903	0.897	0.911
Sponza (mod.; Specularity)	29.410	27.647	30.975	0.908	0.901	0.919

Hierarchical Epipolar (Reprojection)						
Model	PSNR			SSIM		
	Avg.	Min.	Max.	Avg.	Min.	Max.
Hairball	12.428	11.740	13.281	0.734	0.720	0.747
Lucy	37.131	35.523	39.807	0.989	0.987	0.994
Lucy (Specularity)	37.131	35.523	39.807	0.989	0.987	0.994
Mitsuba	33.450	32.833	36.917	0.962	0.951	0.996
Mitsuba (Specularity)	34.350	32.762	36.746	0.980	0.951	0.995
Sponza	24.724	19.317	29.440	0.850	0.798	0.913
Sponza (Primitive Coalescing)	24.725	19.317	29.446	0.850	0.798	0.912
Sponza (Specularity)	24.459	19.256	28.899	0.850	0.797	0.912
Sponza (mod.)	28.535	26.840	30.987	0.901	0.893	0.918
Sponza (mod.; Primitive Coalescing)	28.537	26.829	30.995	0.901	0.893	0.917
Sponza (mod.; Specularity)	28.490	26.749	30.925	0.906	0.898	0.922

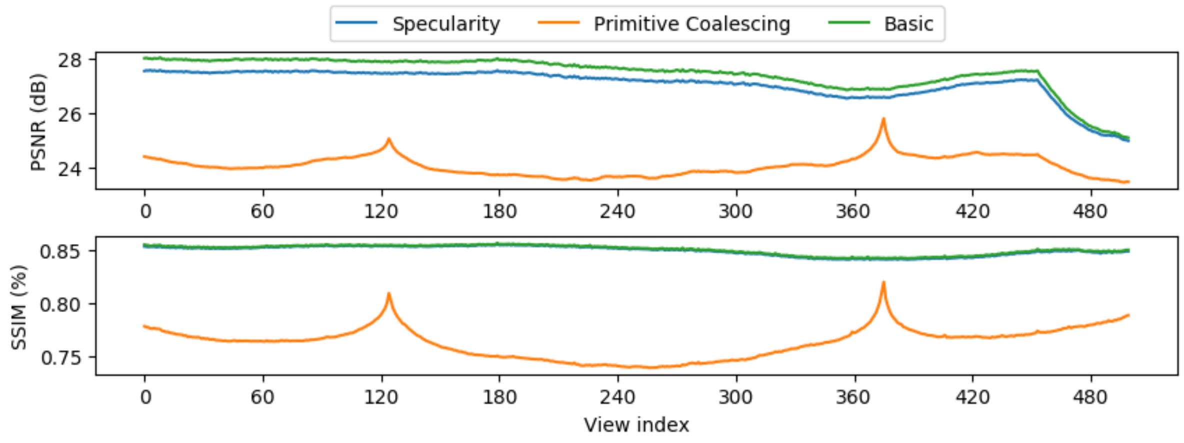

Figure 4.30. *Individual image quality trace. Notice the decreased image quality with respect to the PNSR metric for view indices in the range 450 to 500. Furthermore, only the image quality peaks when primitive coalescing is active. Test scene: Sponza (Original); render pipeline: hierarchical epipolar (Erosion); view set size:500 views.*

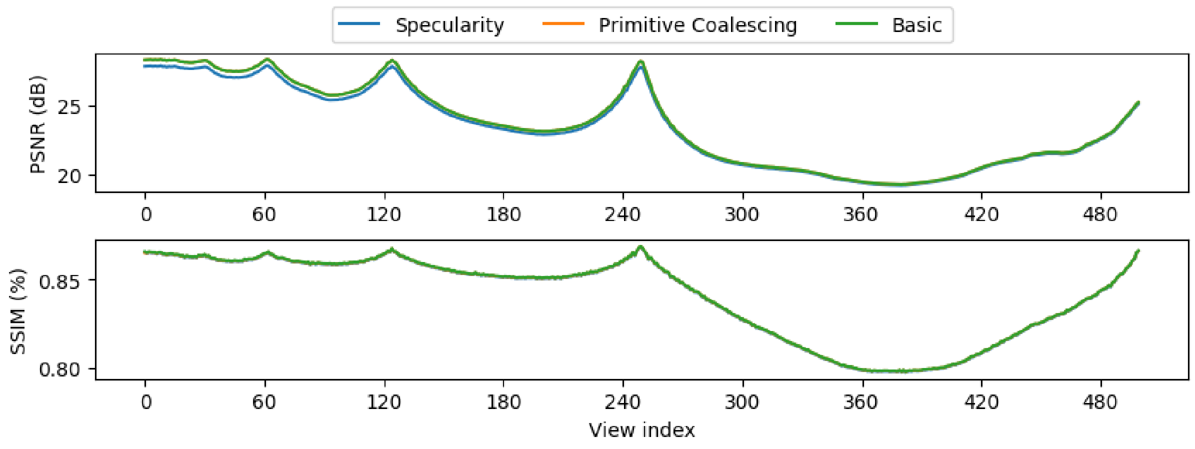

Figure 4.31. *Individual image quality trace. The image quality peaks for the views which are included in the source view set. Test scene: Sponza; render pipeline: hierarchical epipolar (Reprojection); view set size: 500 views.*

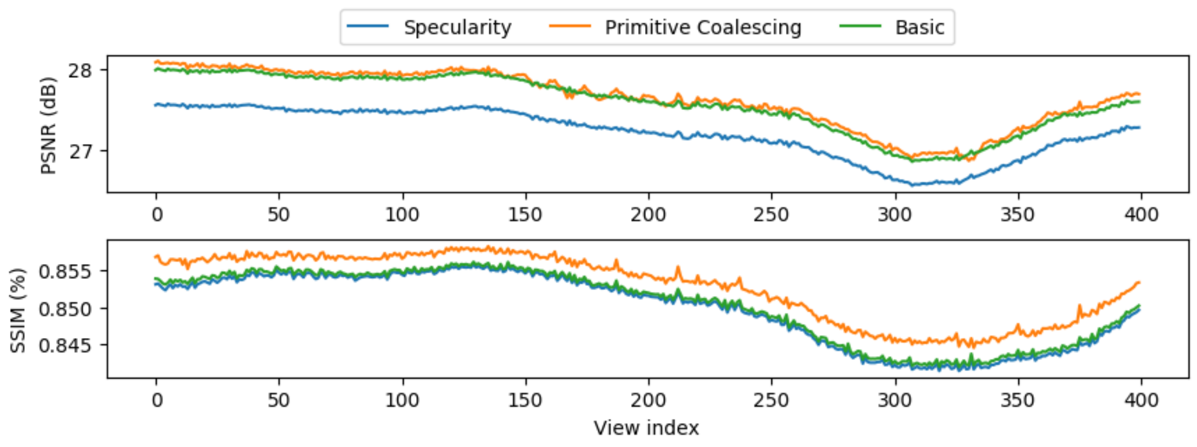

Figure 4.32. *Individual image quality trace. Like the dual epipolar pipeline, the image quality is lower to the right of the center view. Test scene: Sponza; render pipeline: hierarchical epipolar (Erosion); view set size: 400 views.*

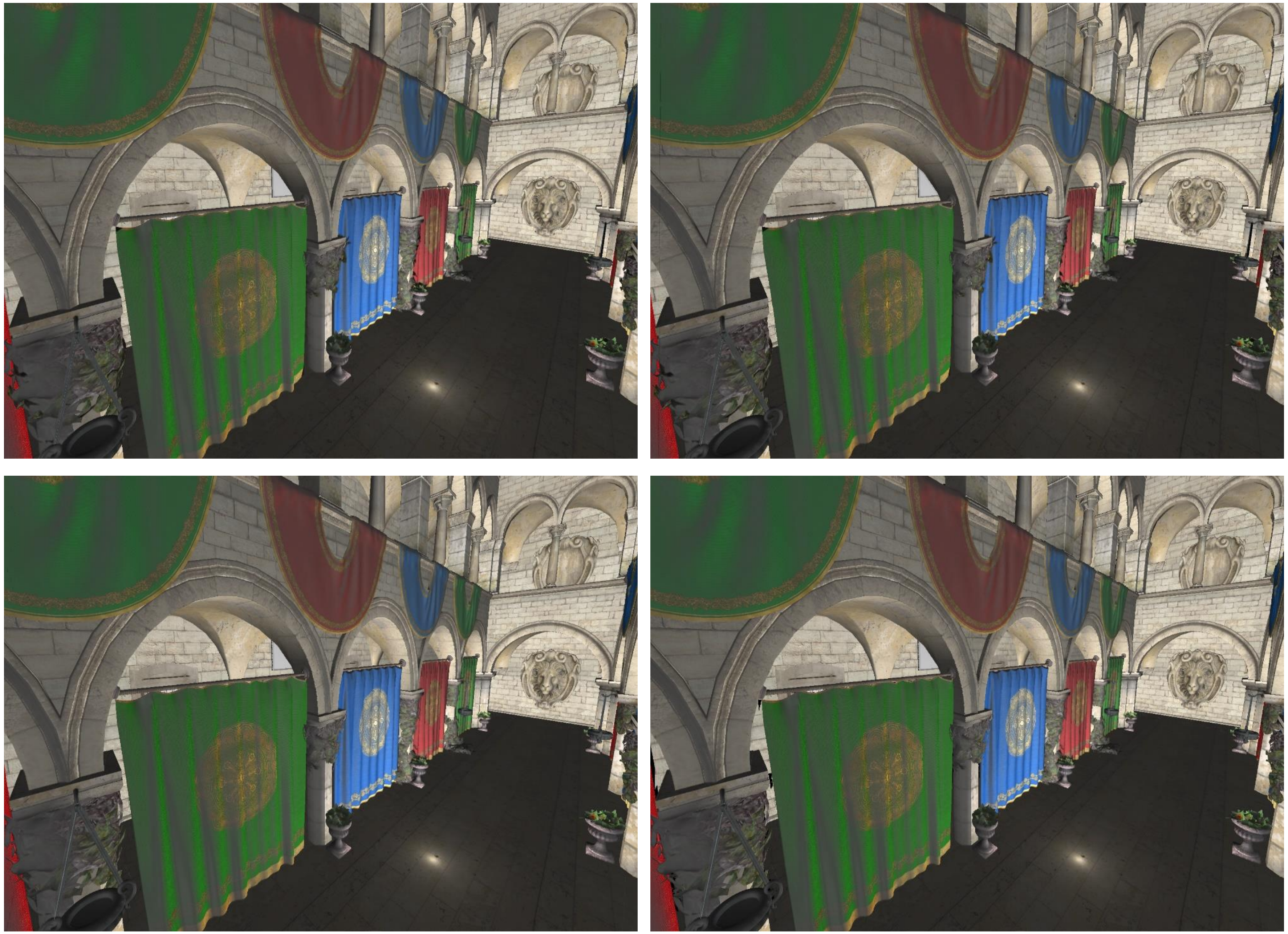

Figure 4.33. *Side-by-side comparison of a single view between the deferred rendering pipeline (top left), dual epipolar (top right), hierarchical epipolar (Erosion) (bottom left) and hierarchical epipolar (Reprojection) (bottom right) pipelines.*

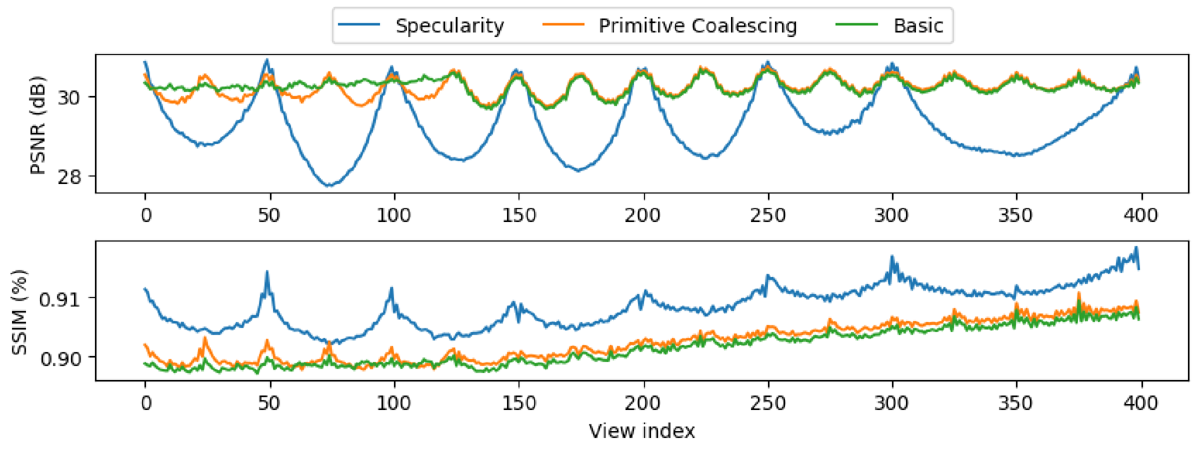

Figure 4.34. *Individual image quality trace of changed test case to circumvent edge cases in the original tests. The original trace is shown in Figure 4.32. The modifications lead to an improvement in quality. Test scene: Sponza; render pipeline: hierarchical epipolar (Erosion); view set size: 400 views.*

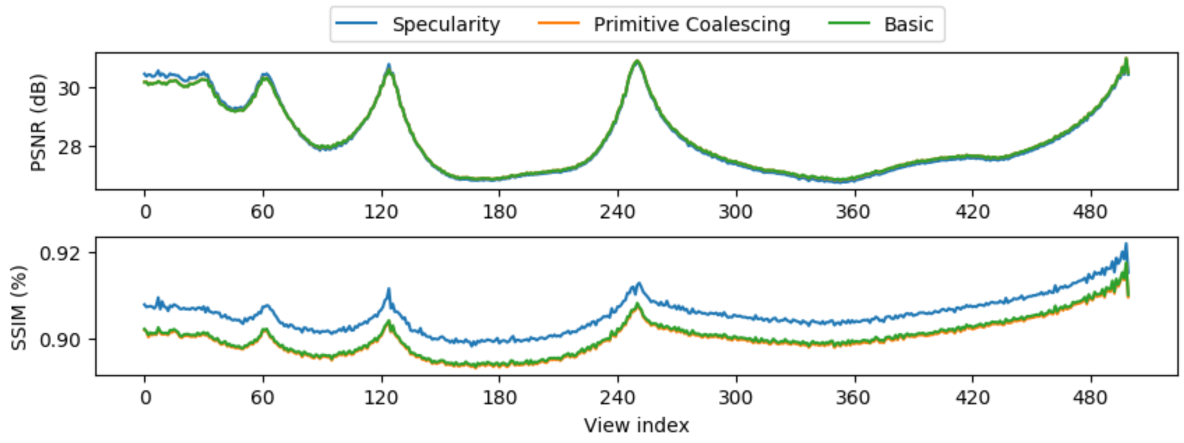

Figure 4.35. *Individual image quality trace of changed test case to circumvent the edge cases in the original tests. The original trace is shown in Figure 4.31, the modifications lead to an improvement in quality. Test scene: Sponza; render pipeline: hierarchical epipolar (Reprojection); view set size: 500 views.*

Regarding the support for specularity, it was added as an afterthought to the basic epipolar interpolation algorithm. However, the aggregated numbers in Table 4.4, Table 4.5 and Table 4.6 show that the support for specularity is adequate, and on par with the other test configurations. The quality traces for the individual images also show minimal differences in quality. See, Figure 4.36 for an example comparison of support for specularity using the Mitsuba test scene.

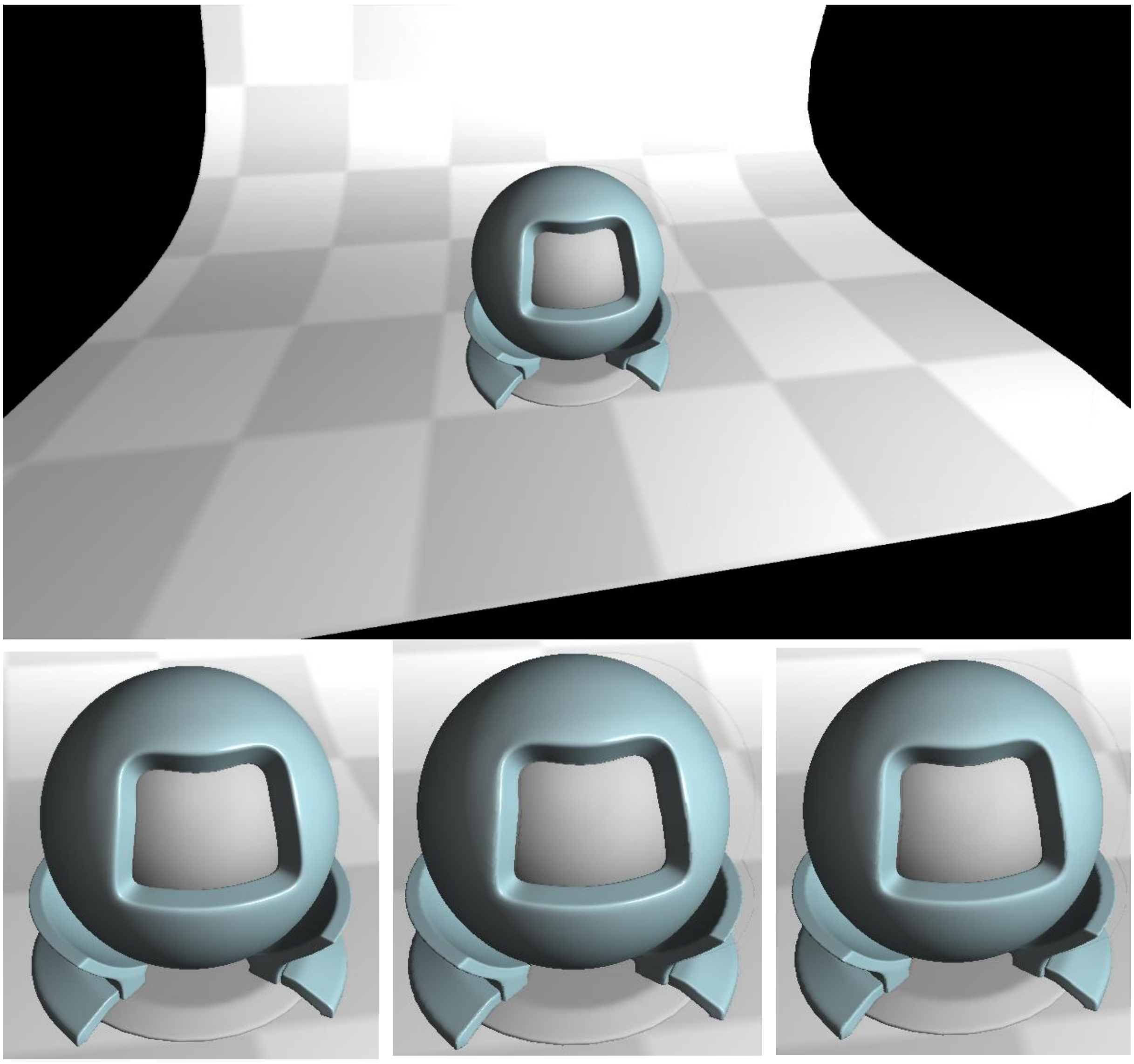

Figure 4.36. *Example of support for specularity by the epipolar interpolation pipeline. The shown top image was generated using the dual epipolar pipeline. The specularity is manifested as the white higlights around the rectangular hole. The top image gives an overview of the camera sight. The bottom row shows cropped versions of the same view. Hereby, the bottom left was generated using the baseline deferred rendering pipeline. The bottom middle is a cropped version of the top image. The bottom right image was also generated using the dual epipolar pipeline but has no specularity.*

5 Discussion

The previous chapter reported the measured numbers with respect to the performance of the different algorithms and the image quality of the resulting images. But how do these numbers relate to the research question posed in Chapter 1? Is the outcome of the conducted study an improvement in the field of computer graphics? And what were the limitations of the research? This chapter will answer these questions.

Section 5.1 starts with interpreting the results as reported in the previous chapter and tries to give insight into the behavior of the algorithms based on the measured results. Also, a few recommendations on algorithm usage and an idea of the impact this study has on computer graphics are given. Section 5.2 ends this chapter with choices that were made during the entire process and explaining the limitations inherent to the conducted research.

5.1 Implications of results

The developed algorithms are evaluated on two characteristics. The quality of the generated images, and the performance characteristics. The main expectation regarding performance would be to see the epipolar based pipelines outperform the baseline. As a starting point, and to guide expectations, a theoretical analysis of the computational complexity was given. In Appendix B.1, the runtime complexity of the deferred pipeline was reported to be $O\left(x * (v + p)\right)$. This is the baseline against which the interpolation-based pipelines are compared. The runtime complexity of the center view acquisition, dual view acquisition and hierarchical view acquisition were determined to respectively be $O(v + p)$, $O(v + p)$ and $2 * O(v + p) + O(1) + 9 + T/B * (O(p^2 + 2 * p) + O(v + p) + O(p) + 3 * O(1) + 11) + 1$ in Section 3.3.1. Furthermore, Section 3.3.2 established the runtime complexity of the epipolar based view interpolation algorithm to be $O(p)$. As such, considering only the theoretical result, the center and dual view interpolation based pipelines take less computational resources based on the analysis of the runtime complexity because the computational complexity for both pipelines is known to be $O(v + p) \ + O(p)) = O(v + p)$, and is less than the computational complexity of the deferred rendering pipeline given enough views. The computational complexity for the baseline was determined to be $O(x * (v + p))$. The computational complexity of the hierarchical epipolar pipeline is more nuanced and does not clearly state under which conditions it becomes better than the deferred rendering pipeline.

Performance

Unfortunately, a theoretical analysis doesn't paint the whole picture since a lot of the constant overhead is not considered with computational complexity, which is also the case for the presented method. Furthermore, a functioning implementation must incur extra overhead due to bookkeeping of used resources. Or, maybe an algorithm doesn't translate well to actual hardware because of hidden data or computational dependencies that lead to underutilization of the available parallel processing power. This leads to the question: how does the theoretical performance result carry over when evaluating a real-world implementation?

The computational complexity predicts that the center epipolar pipelines should have better performance with two or more views. However, it is expected this is optimistic due to a lot of constants hidden by the theoretical analysis, which is reflected in the measured timings. The lower complexity scenes favor the deferred rendering pipeline and shows the overhead for view interpolation resulting in higher frame times. However, this is mostly a constant cost. From the measurements it was apparent the deferred rendering pipeline was very sensitive for the volume of geometry. As shown in Figure 4.5, higher fidelity models had a highly negative impact on the frame times. It appears the deferred rendering pipeline is bottlenecked by the generation of the geometry buffer. More specifically, the data suggests it is bound by vertex processing.

The epipolar based pipelines are less affected by this since they are image-based algorithms. The overhead of brute force calculating each view becomes higher than the overhead of epipolar based interpolation at roughly 2 million vertices and 4 million triangles. The epipolar pipelines use deferred rendering when capturing the source views. As such, the volume of geometry must have some effect. But this is kept to a minimum because the number of rendered views is capped. Figure 4.3 shows that the basic center epipolar pipeline seems to have better performance starting from roughly 500 or more views with the original Sponza scene, and less as the fidelity of the scenes increase. For the original San Miguel scene, there is an improvement at roughly 50 or more views. In the most demanding test case, the speedup achieved by the dual epipolar pipeline is roughly 11.85. Figure 4.10, Figure 4.11, Figure 4.14 and Figure 4.16 are some exemplary breakdowns of how the time is spend rendering a frame. The bulk of the processing is spend doing the interpolation. More so, when the number of views increases.

A key observation on the computational complexity of the epipolar pipelines is the independence from the number of views. This result is not exactly corroborated by the measured performance. Figure 4.3 and Figure 4.6 hint at the existence of a linear correlation between the frame time and the number of views for the basic center and dual epipolar pipelines. Nonetheless, the time cost per view for the center and dual epipolar pipeline is much less than the cost per view for the deferred rendering pipeline. As such, the improvement over the deferred pipeline will continue to grow as the number of views increases. Arguably, a more important result is that the basic center and dual epipolar pipelines will always outperform the deferred rendering pipeline given enough views, as predicted by the computational complexity analysis. Coupled with optimizations, the case for the center epipolar pipeline improves because the frame times decrease by up to 150ms for the original Sponza scene, as shown in Figure 4.7. More importantly, the optimizations reduce the time to generate a single view. In other words, the growth rate of the frame time is reduced.

The dual epipolar pipeline utilizes a method like the center epipolar pipeline. As such, it’s to be expected that the performance follows a similar pattern. However, the dual epipolar pipeline performs roughly twice the amount of work. Thus, it stands to reason that the frame times double, or at least increase. Figure 4.2 and Figure 4.3 show this expectation appears to be true. However, the frame times between the mentioned pipelines can’t be compared based on the presented measurements because the results were gathered on different test platforms. As such, it is not possible to judge if the frame times have doubled. In return for the lower performance, the interpolated images are complete. The center epipolar pipeline is not capable of such a feat because the center view does not span the visible range of all other views in general. The increased frame times mean that the dual epipolar pipeline is still usable in common and workable situations but requires larger size view sets to be competitive with the baseline deferred rendering pipeline, compared to the center epipolar pipeline.

Based on the computational complexity, the performance characteristics of the hierarchical epipolar pipeline should be different. The expectation is it performs worse than the previous two mentioned pipelines. This result is reflected in the performance measurements reported in Section 4.4, which report a higher frame time compared to the baseline deferred rendering pipeline under equal circumstances. Additionally, Figure 4.2 and Figure 4.3 show higher frame times on both the CPU and GPU compared to the center and dual epipolar pipelines. The most likely reason is the higher volume of data that is processed each frame. Like the center and dual epipolar pipelines, the source view capture process is based on deferred rendering with hardware queries to transfer results from the GPU to the CPU. The difference being that no effort has been put in reducing overlapping source information. This makes it likely that the performance of the hierarchical epipolar pipeline can be improved significantly, but this requires a considerable time investment.

A positive property of the hierarchical epipolar pipeline is that it's also an image-space interpolation algorithm. As such, it is less affected by increasing model complexity than the deferred rendering pipeline. The results show the hierarchical epipolar pipeline can perform on par with or possibly outperform the deferred rendering pipeline for the highest fidelity models. If the measured results are extrapolated to higher view counts, then it is likely that the hierarchical epipolar pipeline has better performance than the deferred rendering pipeline. As such, it might be worthwhile to investigate viable solutions to improve the source view capturing process.

Unfortunately, a reduction of the captured source information alone is not enough. If the source acquisition stage were to be improved to filter out parts of the source images, a bottleneck is still present in the source view acquisition stage that will undermine the viability of this method. Figure 4.13 and Figure 4.15 show that both hierarchical epipolar pipeline variations spend a lot of time on the CPU. More specifically, Figure 4.14 shows the time is mostly allocated to *"Deferred Render PVS"*. As described, this process consists of deferred rendering and image quality feedback. The baseline deferred rendering pipeline shows it doesn't use much CPU time, and since this deferred pipeline is similar it also shouldn't. It is known, the communication between the CPU and GPU using OpenGL query objects leads to a lot of idle CPU time. However, relative to the other epipolar based pipelines, the spend time is also significantly more than expected. It is not clear why this method takes more time compared to the depth peeling methods. Two possibilities are, either, hierarchical source view acquisition is more impacted by the stall, or more views are captured. Based on the computational complexity, it is likely more views are captures.

It should be noted that depth peeling has a similar issue. Depth peeling stalls the GPU from processing rendering commands, and is a known drawback. Unfortunately, the feedback is necessary for deciding to continue. As such, the time spend on the CPU for source view acquisition is roughly equal to the time needed by the GPU to process all corresponding commands. An example hereof can be observed by combining the CPU measurements in Figure 4.8 with the depth peeling timings in Figure 4.10 or Figure 4.11.

A second disadvantage of depth peeling is its inherent need for multiple passes over the geometry. More so, Figure 4.10 shows an increasing percentage of frame time is spend on depth peeling with higher fidelity scenes. Both problems could possibly be solved with methods that capture all depth layers in a single pass. The tradeoff with such methods is the substantial increase in memory usage. Refer to Section 7.4 for more detailed information on a possible solution.

One unexpected commonality between the center and dual epipolar pipelines is how the frame time grows with an increasing number of views depending on the scene. Figure 4.6 displays this phenomenon for the center epipolar pipeline, and correspondingly Figure 4.8 shows it for the dual epipolar pipeline, although to a lesser extent. Both pipeline variations seem to scale better with the San Miguel scene compared to the Sponza scene. A possible explanation for this is the difference in the type of geometry that is rendered in the respective scenes. The Sponza scene consists of mostly large surfaces, whereas San Miguel contains smaller geometry with a decent amount of foliage. As explained, the extended depth offset optimization is considered part of the basic pipeline. This optimization filters out small geometry located closely behind each other. An example of this is the foliage in the San Miguel scene. It is speculated that due to the camera position, overlooking the tree in the courtyard from the balcony, more geometry is filtered out from the depth layers with the San Miguel scene. Thus, less information must be interpolated and, hence, leads to lower growth rates.

The aforementioned behavior can result in two conclusions. Firstly, the source view acquisition stage is optimized for small geometry that sits closely together, which seems logical since geometry occluded by other (very) nearby geometry does not allow a lot of sight lines and thus doesn't contribute much to the interpolated views. Secondly, the developed image-space interpolation algorithm is dependent on the area of connected geometry that gets captured by the source view acquisition stage. However, this behavior combined with the measured impact of the visibility estimation optimization and experience with the lesser performance of the hierarchical epipolar pipeline, it might be advantageous to adopt a more aggressive visibility estimation algorithm to filter out more information in the source views.

Besides the basic epipolar interpolation pipelines, several optimizations were described to reduce the frame time. After testing, it was found that primitive coalescing is the most impactful optimization. With the original Sponza scene and generating 500 views, an improvement was seen of roughly 150ms and 280ms for respectively the center and dual epipolar pipelines. The hierarchical epipolar pipelines showed improvements of roughly 100ms (erosion) and 90ms (reprojection) at 100 views for the same scene. However, arguably, a more important result is, this optimization reduces the frame time growth rate. Additionally, this optimization significantly reduces pressure on the rasterizer since the number of primitives is reduced, and could hint at a bottleneck. This means either the GPU is not built to handle a high volume of relatively small geometry, or the number of generated fragments is too high.

The two remaining optimizations, dynamic layer reduction and visibility estimation, are only applicable to the center and dual epipolar pipelines. Dynamic layer reduction had no measurable impact on the performance.

Visibility estimation had a minor impact. The center epipolar pipeline showed no improvement with visibility estimation. On the contrary, the dual epipolar pipeline showed a modest improvement of about 100ms at 500 views for the original Sponza scene. A possible explanation for the minor impact could be that it's fighting with the always active extended depth offset. This optimization merges different depth layers which leads to many small, unconnected areas at different depths. However, the visibility estimation optimization works best for large connected sections since it is a conservative estimate of the potentially visibile set of geometry. For small geometry sections, the algorithm cannot guarantee invisibility for occluded geometry.

In addition to the optimizations, an extension was described in Section 3.5 to prove that the developed pipelines are capable of interpolating view dependent properties. Specularity was added by means of the (Blinn-)Phong reflection model. Regarding the performance, the expectation was to see an increase in frame time. However, the addition was barely noticeable in the measured timings for both the deferred rendering pipeline and the newly developed algorithms. Most likely because this reflection model is not computationally expensive.

Besides the performance characteristics of the different pipelines, the measurements also contained several outliers compared to surrounding results. All interpolation-based pipelines, for view set sizes between 10 and 75 views, had a lower frame time than could be expected from other view set sizes. This result is not due to outliers in the measurements, since using the median or discounting n lowest and highest timings resulted in similar graphs. Another possibility could be an issue with the test cases. For example, clipping geometry could lead to less depth peeling layers because a couple layers are not located between the camera near and far plane, and thus not included in the captured depth layers. As a result, this should reduce the frame time. However, after double checking the framing, this was found not to be the case.

The most likely reason is a difference in the distance between cameras for the different number of views. The performance tests are setup such that the distance between the left- and rightmost cameras are fixed across the different view set sizes. Consequently, the distance between cameras must change. For less views, the distance between consecutive cameras increases. For the geometry in the epipolar plane, this means that the angle increases since the overall disparity remains identical.

In other words, the disparity per view is more. Likely, this leads to more overlapping fragments which are discarded due to depth testing and would lead to less fragment shader invocations. This reasoning is corroborated by less impact of this behavior with the San Miguel scenes, which were earlier inferred to have less captured pixels due to the foliage. Furthermore, this reasoning also holds for the other pipelines, which exhibit similar behavior.

Besides, if the algorithm is bound by processing of the substantial number of fragments, then it could also explain the lower frame times with the San Miguel scene compared to comparable, in number of vertices and / or triangles, Sponza scenes. All other parameters equal, it would be expected that a scene with more geometry would take longer to render, but this is not the case.

Taken at face value, the mentioned behavior does not explain the increased frame time for 2 views. However, for 2 views, it is likely that the processing time is dominated by the transformation of pixels to geometry in epipolar space. Furthermore, the distance between the leftmost and rightmost views is such that there is not much overlap between the two views. As such, nearly all captured pixels are needed to generate the set of interpolated views. Again, this would also hold for the dual and hierarchical epipolar pipelines.

Lastly, a note on the lower than expected frame time of the dual epipolar pipeline at 300 views. As shown in Figure 4.8 and Figure 4.9, a drop in the frame time is visible only for test cases with the Sponza scenes. No other pipeline had a similar behavior. Furthermore, the detailed timing information showed, this consistently happened over a series of frames. Also, the anomalous results were measured in succession. Unfortunately, the source of this behavior is unknown. It could be due to an external factor such as Windows installing updates, but an anomaly in the measurements or a bug in the implementation is equally likely.

Image quality

The second aspect on which the presented view interpolation algorithm is evaluated is the quality of the output images. The expectation before measuring is that the quality of the images is less than the baseline images. The reason being that the view interpolation algorithms are approximating algorithms. Nonetheless, for the epipolar interpolation algorithm to be usable, the quality should still be sufficient. This raises the question: what is the definition of reasonable or usable? And, how does this translate to the used measures?

In a perfect world, the best solution would be to ask people if sets of images are indistinguishable. However, such is infeasible for the presented research project due to the scope of the project and the time involved to do this properly. Fortunately, the presented algorithm is not the first to generate or use approximate images. Some examples are static image compression algorithms for still pictures, or, image stream compression algorithms for tv broadcasting. The goal of these types of algorithms is to trade image quality for storage space, but such that the image quality is still usable. Fortunately, these algorithms are mostly compared and calibrated using measures identical to the ones used for this project.

Unfortunately, there is no authoritative source which gives concrete values for reasonable or usable images. Furthermore, papers using identical image quality measures don't give specific thresholds or value ranges for images that are considered good. This makes sense because the judgement depends on the use case. As such, for the purposes of this report, reasonable or usable is defined as commonly used and reported values. For PSNR, a range of 25dB to 30db is commonly used for acceptable quality, and 30db to 35db is typically considered good quality. SSIM is a newer measure, and thus less commonly used. But, SSIM values of roughly 0.90 and higher typically correspond with a relatively good image quality.

From all test scenes, only Sponza covers the entire image. Furthermore, it is most representative for common usage scenarios since most (or all) used scenes cover the entire image. As such, the judgement for usable image quality is mostly based on it. The image quality for Lucy and Mitsuba are better but would unfairly boost the results.

From the original image quality tests, as displayed in Table 4.4, Table 4.5 and Table 4.6, it is known that the average PSNR values for the center epipolar pipeline rendering the Sponza scene lie within the range $[15.410;\ 53.929]$ with an average around 22.51. This is uncorrected for the missing information on the sides of images. The range after correction is $[28.898; 29.740]$. The dual view epipolar pipeline gives PSNR values in the range $[24.030;\ 36.476]$ with an average around 28.64. The PSNR results for the hierarchical epipolar pipeline lie in the range $[23.465;\ 28.102]$ with the average around 27.10 or $[19.317;\ 29.446]$ with an average around 24.64 for respectively the pipeline configured with erosion or reprojection based quality measures. However, as mentioned in Section 4.2, the PSNR only measures the strength of the error and does not consider the human visual system, which may not notice or correct small, specific types of errors. As such, the image quality was also measured using SSIM. The average values for the center epipolar pipeline are within the range $[0.713; 1.000]$ with an average of 0.878, when the measurements are uncorrected for the missing image parts. If the image quality is corrected for this, then the average SSIM value is 0.914. The dual view epipolar pipeline reports SSIM values in the range $[0.892; 0.991]$ with an average of 0.916. Furthermore, the hierarchical view interpolation pipeline with erosion-based quality measure gives SSIM results in the range $[0.738; 0.871]$ with an average of 0.843. For the same pipeline with reprojection based quality measure, the range is $[0.797; 0.913]$ with an average of 0.85.

Unfortunately, those results were obtained with a test case that triggered an edge case inherent to the algorithm and is a compound of two problems. The test cases were configured with too large of a view offset. As such, the distance between the left- and rightmost views were such that the parts of the scene that are visible in the leftmost view, are (mostly) not visible in the rightmost. This leads to a situation where the information captured in source views don't cover enough visible geometry, and holes form in the interpolated views. This problem is exacerbated by undersampling due to storing the source views in textures. The source views captured geometry that ran almost parallel to the viewing direction, and on the eye direction vector. As such, the source views (almost) didn't capture geometry which should be visible in other views but were missing in the interpolated views. Unfortunately, this problem is inherent to the used algorithm, but its solution is left as future work.

To get representative results and circumvent the edge case, the image quality tests using the Sponza scene were redone with a modified camera position and orientation to not trigger the edge case. This was only done for the dual and hierarchical epipolar pipelines since they were only affected by this issue.

The resulting measurements are slightly better. The dual epipolar pipeline has PNSR results in the range $[31.733;\ 39.080]$ with an average of 33.006. The corresponding SSIM values fall in the range $[0.923;\ 0993]$ with an average of 0.945. The hierarchical epipolar pipeline with erosion-based image quality measure results in PNSR values in the range [27.647; 30.975] with an average of 29.965, and SSIM values in the range [0.896; 0.919] with an average of 0.904. When based on the reprojection image quality measure, the PSNR values are within the range [26.749; 30.995] with an average of 28.521. Its corresponding SSIM values are within the range [0.893; 0.922] with an average of 0.903.

For the unmodified Sponza test cases, the minimums show an image quality which could be considered low image quality. However, the average values are above the lower bound for an image quality which can be considered quite good. Furthermore, manually inspecting the images reveals that most artifacts are hardly noticeable. An analogous conclusion can be made based on the SSIM values. When looking at the modified test case, then the minimum image quality for both the PSNR and SSIM metric are close to the lower bound for good image quality, with averages and maximums well above the threshold. As such, it can be concluded that the images produced by the interpolation algorithm have sufficient quality to be usable in the general case. However, this is highly dependent on the specific use case and needs to be determined on a case-by-case basis.

At the beginning of this section, a remark was made about the higher image quality of non-image filling scenes. However, the Hairball scene falls in this category. Contrastingly, it led to images with the lowest overall image quality. This was expected since it shows a pathological case where the algorithm breaks down. The discretization of the 3d description of the hairball model to the image space domain resulted in precision loss. Furthermore, some amount of information is lost with the interpolation.

However, one unexpected observation was the lower image quality of the dual epipolar pipeline compared to the center epipolar pipeline. Based on the results from the center epipolar pipeline, the conclusion can be drawn that the interpolation-based generation is able to generate perfect images with respect to the baseline. Most notably, this holds true for the center views on the center epipolar pipeline. In general, the views used as source views should have perfect image quality since they are copied from the original. However, technically, these views are regenerated based on the interpolation. But, the calculations for these views should lead to the identity transformation. As such, the upper bound on the SSIM quality results should be 1.0. The dual view and hierarchical interpolation pipelines use the images corresponding to two or more cameras as the basis for view interpolation. But, the test results show these images are less than perfect.

This could stem from several causes. The interpolation could introduce a small amount of error in the computed transformations, possibly due to rounding errors. In turn, this could lead to small rasterization errors that lead to one-pixel offset differences. However, a more probable cause is imperfect mixing of the different source views during interpolation. In other words, depth testing does not select the correct mix of colors in every pixel. This was improved by introducing an additional depth bias, and led to an increase in image quality. But, it is apparent that this solution is not ideal. The absolute error maps in Figure 4.28 and Figure 4.29 show that the error should be applied selectively since certain image areas are only covered by one source view. These image sections don't suffer from view mixing errors, but the depth bias is applied regardless. As such, the geometry is slightly moved which could lead to a slightly higher error.

Memory cost

The performance and image quality of an image reconstruction algorithm are important aspects for the adoption in production. However, if better performance comes at the cost of unreasonable amounts of memory, then an algorithm still cannot be used outside academia.

Unfortunately, the OpenGL specification doesn't specify an API to measure the memory usage of individual resource. Some GPU manufacturers released vendor-specific OpenGL extensions that only provide information on the total or available GPU memory. For example, Nvidia provides GL_NVX_gpu_memory_info [42], and AMD exposes the WGL_AMD_gpu_association [14] and GL_ATI_meminfo [4] extensions. Moreover, Nvidia is working on an extension that provides object level memory usage statistics.

However, even object level memory usage would give an answer near the truth because of GPU drivers or internal hardware requirements. As such, OpenGL might not exactly allocate what was asked for. For example, OpenGL might allocate more than one copy of a resource to promote pipelining and avoid stalls. A more concrete example could be that a GPU doesn't support a 32-bit color framebuffer with 16-bit depth buffer. So internally, OpenGL might create a combination that is supported but uses more memory. Thus, resulting in hardware dependent memory usage information from a very small GPU sample size because of time and hardware availability constraints.

The next best option is a theoretical comparison where the memory footprint of a resource is counted as requested by the algorithm implementation, and not counting any duplicates, and is the chosen route for the presented analysis. An example memory usage breakdown of a deferred rendering pipeline is shown in Table 5.1. This example assumes an output of 500 distinct views, with each view having a resolution of 1920 by 1080 pixels. The breakdown is based on the implementation used for performance and image quality testing. The total memory usage sums up to roughly 4.21 GB.

Table 5.1. *Example memory usage breakdown of the deferred rendering pipeline. The resolution of the images in this example is 1920x1080 pixels, and the number of views expected as output is chosen to be 500.*

Categorization	Description	Dimensions (Width x Height x Depth)	Required memory (MB)
	Composed views	1920 x 1080 x 500	4,147.2
Geometry-Buffer	Albedo	1920 x 1080 x 1	6.22
	Depth	1920 x 1080 x 1	8.29
	Normals	1920 x 1080 x 1	24.88
	Specular Color	1920 x 1080 x 1	6.22
	Specular Intensity	1920 x 1080 x 1	8.29
Screen Space Ambient Occlusion	SSAO	1920 x 1080 x 1	8.29
	SSAO Blur	1920 x 1080 x 1	8.29
	SSAO Noise	4 x 4 x 1	$9.6 * 10^{-5}$

Table 5.2 shows the memory usage breakdown for the center view acquisition pipeline combined with the epipolar view interpolation pipeline. Also note that the shown example has included the extended depth offset optimization to limit the number of depth layers. Identical to the deferred rendering pipeline example, the assumed output is 500 distinct views with a resolution of 1920 by 1080 pixels for each view. The memory usage of the interpolation-based pipeline is 7.84 GB. As such, the required memory is a little less than twice the amount needed for the baseline for this example.

On top of this, two of the performance optimizations increase the memory usage. Table 5.3 gives a breakdown of the memory usage for the visibility estimation and primitive coalescing optimizations under the same conditions as mentioned for the non-optimized pipeline. The additional memory usage is roughly equivalent to 0.75 GB.

Table 5.2. *Example memory usage of the center view acquisition pipeline combined with the epipolar view interpolation pipeline. The resolution of images in this example is 1920x1080 pixels, and the number of views expected as output is chosen to be 500.*

Categorization	Description	Dimensions (Width x Height x Depth)	Required memory (MB)
	Depth layers (Composed)	1920 x 1080 x 10	82.94
Geometry-Buffer	Albedo	1920 x 1080 x 1	6.22
	Depth layers (Depth)	1920 x 1080 x 10	82.94
	Depth layers (Normals)	1920 x 1080 x 10	248.83
	Depth layers (Specular Color)	1920 x 1080 x 10	62.21
	Depth layers (Specular intensity)	1920 x 1080 x 10	82.94
Screen Space Ambient Occlusion	SSAO	1920 x 1080 x 1	8.29
	SSAO Blur	1920 x 1080 x 1	8.29
	SSAO Noise	4 x 4 x 1	$9.6 * 10^{-5}$
Epipolar View Interpolation	Epipolar planes (Color)	1920 x 1080 x 500	4,147.2
	Epipolar planes (Depth)	1920 x 1080 x 500	3,110.4

Table 5.3. *Example memory usage breakdown of the two performance optimizations for the dual view acquisition and epipolar view interpolation pipelines that increase memory usage. The example shown uses the exact same conditions as used in Table 5.1 and Table 5.2.*

Categorization	Description	Dimensions (Width x Height x Depth)	Required memory (MB)
Visibility Estimation	Discontinuity map	1920 x 1080 x 1	8.29
	Occlusion map	1920 x 1080 x 1	2.07
Primitve Coalescing	Primitive mipmap	1920 x 2127 x 10	326.71
	Primitive buffer	1920 x 1080 x 10	414.72

But, what about less or more views? Or different image resolutions? Respectively equation 5.1.1, 5.1.2 and 5.1.3 are the functions to calculate the bytes per pixel for the deferred rendering pipeline, center view acquisition with epipolar view interpolation pipeline and dual view acquisition with view interpolation pipeline. The input to the functions is the number of views. To see which data types are used for the different resources and a more detailed derivation of these functions, see Appendix C. As expected, these functions show that the presented view interpolation-based methods have a higher memory consumption than the deferred rendering pipeline, independent of the image resolution and number of views. The deferred rendering pipeline has a baseline of 4 bytes per pixel, whereas the interpolation-based methods have a baseline of 7 bytes per pixel. Furthermore, all the pipelines have an additional per pixel memory overhead that decreases as the number of views increases. So, for large numbers of views, the epipolar based method should still be usable in practical use cases with a little less than double the number of bytes per pixel compared to the baseline in the best-case scenario. For smaller number of views, it is more beneficial to use a deferred rendering pipeline.

$$BytesPerPixel_{Deferred}(v) = 4 + \frac{34}{v} \tag{5.1.1}$$

$$BytesPerPixel_{Center}(v) = 7 + \frac{481}{v} \tag{5.1.2}$$

$$BytesPerPixel_{Dual}(v) = 7 + \frac{951}{v} \tag{5.1.3}$$

The hierarchical view acquisition algorithm was left out from the memory usage analysis because the exact number of views captured for use by the epipolar view interpolation stage is unknown. However, during development, the hierarchical view acquisition algorithm has shown that the number of captured images is similar or slightly higher than the number of depth layers captured by the dual view acquisition pipeline. As such, it requires at least a similar amount of memory as the dual view acquisition algorithm. But, it could also need more memory.

However, note that the memory cost analysis is done based on the latest proof-of-concept implementation. As such, both the baseline deferred rendering pipeline and epipolar based pipelines could benefit from several common memory reduction techniques. For example, the geometry buffers could be packed more tightly by combining the several textures into one. This is done by assigning different image channels to hold data from different textures in the original configuration. Compared to the deferred pipeline, the epipolar based pipelines should benefit more from this optimization since it uses one geometry buffer per depth layer.

Furthermore, the memory usage could be reduced by changing the order of operations within the epipolar based pipelines. Currently, all depth layers are captured before any interpolation is done. However, that means at least ten geometry buffers must be stored in memory before they can all be processed by the interpolation stage. It should be possible to capture and interpolate one depth layer at a time. Unfortunately, a likely downside is a performance hit due to extra state changes configuring the graphics pipelines.

Lastly, small benefits may be gained from a memory reduction technique known as memory aliasing. This allows distinct resources to occupy the same memory region at different moments in time. However, this requires the usage of more low-level graphics APIs, such as DirectX 12 or Vulkan, since OpenGL doesn't expose explicit memory placement capabilities in its current API.

Recommendations for usage

Based on observations of the performance and image quality traces, and the resulting characteristics of the algorithm, several recommendations can be given for the use of the different pipeline configurations.

Performance-wise, the current iteration of the center view acquisition combined with epipolar interpolation is very usable. However, the general advise for its usage would be negative, not considering any modifications. The view space spanned by the center view, and thus correctly interpolated are of high quality. But, as shown in the previous section, the sides of images not corresponding with the center view are incorrect for scenes which cover the entire image. As such, it might be worth it to fix the missing information. A possibility would be to account for this by widening the horizontal view angle, but this might result in image distortions due to perspective projection. A second alternative might be to patch the images individually, however, this could get expensive timewise. Thirdly, the interpolated views from two different invocations of the pipeline could be combined such that only the missing image parts are stitched together.

The dual view acquisition pipeline with epipolar view interpolation does not suffer from the mentioned problems and should be preferred over the center view acquisition and epipolar interpolation pipeline. Although, it has worse performance, the performance is such that the baseline deferred rendering pipeline is still outperformed. Besides, the offset in image quality likely results in better performance than the previous pipeline combined with an additional patch up step. However, the image quality tests showed that the dual epipolar view interpolation pipeline suffers from too much distance between the left- and rightmost cameras. This might lead to holes due to undersampling of the geometry. The solution for this would be to clamp the distance to a maximum. Unfortunately, specific maximum distances are dependent on the used scene and need to be determined on a project basis. Support for larger distances is possible by splitting the total range into several smaller distinct subranges that are separately interpolated.

The hierarchical view acquisition and interpolation pipeline is built on the concept of dividing the cameras in distinct sets which are separately interpolated. Just like the other pipelines, the image quality tests show that it suffers from (large) geometry running parallel to the view direction, but to a smaller extent. When this is accounted for, large distances between cameras have little to no impact on the quality of the resulting images. Regrettably, the image quality is currently somewhat lacking compared to the dual view acquisition and interpolation pipeline. As mentioned, this is likely due to the missing depth bias since, compared to the other epipolar based pipelines, a similar implementation is used for the interpolation, and the image quality should thus be comparable. Unfortunately, the hierarchical epipolar pipeline was developed late in the project. More specifically, this pipeline variation is less optimized and, thus, lacks in terms of performance since the interpolation stage must process more data. As such, it's not beneficial to run the hierarchical epipolar pipeline to render views unless the used scenes contain a very high number of vertices and triangles.

Lastly, it should be noted that this algorithm is targeted at (very) high end graphics cards at the time of conducting this study. This is mostly due to the memory requirements. At full HD resolution, the views already consume 4,1 GB. On top of this, the interpolation algorithm has a significant overhead. Furthermore, currently there is no need to have mobile graphics chips generate such a high amount of views since there is not any mobile application (yet). Additionally, the typical trajectory from research to usage in products typically spans several years. At that time, it is likely that current generation high end graphics cards are in common use.

5.2 Limitations of conducted research

As with any conducted study, there are certain conditions and design choices that lead to imperfections in the result. In the end, they may impact the reported results, and ultimately may lessen the impact on the field of computer graphics. Unfortunately, the work presented in this report also suffers from this.

Firstly, not all possible directions of research have been explored. Several more months or even years could be spent tweaking, expanding and/or optimizing the various pipelines. Unfortunately, only a finite amount of time and funding is available. As such, several recommendations for potential future research directions are outlined in Chapter 7.

Furthermore, there is a discrepancy between the theoretical performance of an algorithm and an actual implementation of the algorithm on real hardware. Meaning, the implementation used to measure the results as reported in Chapter 4 is probably not optimal. However, overcoming this requires a deeper understanding about the inner workings of the used graphics API and knowledge about the specifics of the hardware used to conduct the performance and image quality measurements, which is not feasible within the confounds of the performed study.

Additionally, a deeper understanding of the hardware utilization may result in new directions for future research. For example, if it turns out that the current transition from image-space data to epipolar space geometry via intermediate buffers hurts performance since the post-vertex shader cache is underutilized. Or, the usage of geometry shader could be replaced with compute shaders to increase performance, since geometry shaders make heavy use of atomics. Or, the massive amounts of micro geometry in epipolar space lead to performance degradations. Answers to these types of question could point to troublesome parts in the developed algorithm that increase the performance most relative to the time spend fixing. Besides, they may increase the confidence of the conclusions made in the section on implications of results, or, debunk false implications.

Also, the reported timing results in Chapter 4 are highly dependent on the hardware that was used to perform the measurements. Moreover, if the hardware is the same in all use cases, timings are dependent on the implementation of the graphics API. Hardware vendors provide these implementations in the form of device drivers. The problem is that different versions of the device drivers may contain changes and/or optimizations as insight is gained into bottlenecks in previous iterations of the same device drivers, which lead to different timings results. Thus, to aid the reproducibility of presented study the exact test platform is reported in Chapter 4. All things considered, differences in hardware and graphics API implementations will probably lead to the same implications, but the extent of the impact may be different. As such, the developed algorithms must be tested on more hardware and driver configurations to have more confident in the results.

Not only those concerns may impact the applicability of the results. A variety of models were used as input for the performance and image quality measurements. Scenes ranging from very a simple cube with a low triangle count to architectural models with a moderate number of triangles, and even unusually complex scenes with a high triangle count. Furthermore, the test models include the Mitsuba and Lucy statue models which specifically target specularity. However, for practicality reasons the set of models is finite and doesn't cover all imaginable situations. Instead, the test set is chosen such that the most common use cases are covered.

Additionally, the set of test models could benefit from a scene specific to the dual and hierarchical epipolar pipeline. Both use images from multiple cameras as source; possibly with heuristics to guide the source view capturing process. With the current test set, this most likely leads to different captures for the different views. To better understand how the different views contribute to the final images a symmetric scene should be added. With the center of all cameras placed on the symmetry axes, this should lead to predictable source view captures and eliminates parameters from the source view mixing except for the actual merging of the different views.

Another impacting decision is the baseline against which the developed pipelines were tested. Although, the performance of the different algorithms was tested in a variety of settings. The performance of the algorithm was tested against a barebone baseline implementation. This is a result of the features implemented in the interpolation framework, and the scope of research. As such, the vertex and pixel processing workloads are not representative of real-world usage patterns. Production rendering pipelines are likely to have more computations in the vertex and fragment shaders. Since the presented interpolation pipelines reduce these types of computations, the results are likely to become more favorable for the interpolation framework. However, this needs to be tested to have a certain answer.

Moreover, the current iteration of the framework is only tested in situations which are vertex shader bound. In other words, the rendering pipeline is flooded with vertices to check if the processing time is dominated by vertex processing, and the interpolation-based algorithm already improves over the baseline deferred rendering pipeline. However, current rendering pipelines are more likely to be fragment shader bound with the adoption of complex lighting models. In other words, processing time is dominated by the computations in the fragment shader. Since the interpolation utilizes very simple fragment shaders, the situation will likely favor the interpolation-based view generation.

The last notable limitation of the current test setting is the possible focus on the strong points of the baseline deferred rendering pipeline. The current test configuration is exclusively targeted at a relatively small number of high resolution views. As shown by the limited, additional test with the dual epipolar pipeline rendering many small resolution views, the current test configuration appears to favor the deferred rendering pipeline. As such, the shown performance of the epipolar-based pipelines is likely not worse than the performance in typical use cases. On the other hand, the chosen test cases do not properly show how much performance can be gotten from the developed algorithms, and what use cases are best for the epipolar-based pipelines. However, due to time constraints, the evidence for this is limited and more testing must be done to be certain.

Besides the human and practicality factor influencing the research outcome, the implications of the performance and image quality is also limited by certain inherent properties. Starting with the performance of the implemented algorithms. To make generalized statements recommending the presented framework to be used by the masses requires knowledge about the specific use case, since it is not only dependent upon the used scene but also on the requirements set by the end user, and the software alongside which the proposed algorithm needs to operate and cooperate with. A couple examples are the hardware platform on which the final product needs to run or the maximum available memory.

Likewise, making generalized statements about the image quality not affecting people is limited since any measure trying to quantify image quality focuses on a single aspect of the image. For example, they do not consider that less image quality is required when a human is looking at high speed content since a lot of the fidelity is filtered out by humans. In the end, the implications of the results do provide a strong indication, and lead to probable predictions. But, ultimately users should decide for themselves whether the proposed solution is good enough.

6 Conclusion

In the last several years, the tendency for current gaming markets has shifted towards the development and integration of virtual reality. The corresponding display devices require more than one view as input. The specific focus of this report has been devices that require massive numbers of views as input, such as light field displays. The presented study questioned whether it was possible to build a framework around the concept of epipolar space, allowing the exploitation of coherence between images to speed up rendering.

The development of the proof-of-concept implementation resulted in several rendering pipelines that can be used to generate many views more efficiently than a deferred rendering pipeline. From a high-level viewpoint, each pipeline consists of two stages. The first stage is named source view acquisition, and gathers the input required for the second stage. Three different methods were developed for this stage, of which two are based on depth peeling and one is a novel approach that samples different cameras which are used as source view. The second stage consists of the actual generation of all distinct images. This process is based on interpolation and is done in epipolar space. The first pipeline captures the depth layers from the center views, and is referred to as center epipolar pipeline. The second combination is named dual epipolar pipeline since it uses the depth layers of the leftmost and rightmost view as source view. The combination named hierarchical epipolar pipeline, is the third and last combination, and is named as such because it recursively splits the range of views in half and uses the center view of each split as source view. The recursion is stopped by means of a heuristic.

The various components are self-contained and use conventions to pass data between stages. As such, this allows substitution of different methods and integration of different optimizations and extensions within the various stages with relative ease. Included are several optimizations to reduce the amount of information to be processed by the pipelines. Furthermore, it is shown that support for specularity is possible since the basic interpolation stage is not able to use standard specular reflection models without modification.

The formed pipelines were reviewed on their performance. Specifically, the research subquestion was formulated as follows: "*How much is the performance increase over a baseline rendering pipeline? In other words, how much is the reduction in frame time when the newly developed framework is compared to a baseline rendering pipeline?*"

The answer to this question is more complicated than a single x-times improvement, or y-times reduction of frame time, over the baseline deferred rendering pipeline. For the basic center epipolar pipeline with Sponza (Original) scene, an improvement is noticeable starting from 500 views. With the Sponza (Extreme) scene, the improvement is visible starting from roughly 50 views, and the difference at 500 views is one order of magnitude. If optimizations are activated, the frame times are further reduced and the center epipolar pipeline is even better. The dual epipolar pipelines shows similar behavior except it takes more time to generate complete images. It does twice the amount of work, but in return the images have acceptable quality. Whether the extra work translates in twice the frame time cannot be determined based on the gathered information since the test platforms were different. To summarize, the improvement for both epipolar pipelines with depth peeling based source view acquisition becomes better with increasing number of views or scenes with higher vertex and triangle count. On the contrary, the hierarchical epipolar pipeline has differing results. When comparing the frame times measured on the CPU, it is slower than the baseline deferred rendering pipeline for the measured test cases. The frame times from the perspective of the GPU show that in most cases the hierarchical epipolar pipelines is also slower, except for the two highest detail San Miguel scenes. Using those two scenes, the hierarchical epipolar pipeline improves over the baseline pipeline starting from roughly 75 views. This holds true for both image quality measures.

The last dimension on which the developed pipelines were evaluated was the quality of the generated images. The subquestion regarding image quality was stated as follows: "*Are the resulting images of reasonable quality? More precisely, have the generated images using interpolation an image quality such that they are usable in consumer products?*"

Based on observations of the measured image quality, it can be stated that interpolation in epipolar space is capable of producing images that are of sufficient quality. However, the different pipelines have their own nuances and peculiarities. The center epipolar pipeline produces images with SSIM values in the range $[0.713; 1.000]$ with an average of 0.878. This comes with the side note that the images contain sections that are incorrect. The center view cannot span the visible area of all views, and as such, the interpolation misses information. When this is corrected for, the average SSIM increases to 0.914. The dual epipolar pipeline generates images with SSIM values in the range $[0.923;\ 0993]$ with 0.945 as average. But, only if the inter-camera distance is within a safe operating envelope. For large inter-camera distances, the interpolation must rely on too little information, and as such, holes can form. The SSIM values fall in the range $[0.893; 0.922]$ with 0.904 being the average when considering the hierarchical epipolar pipeline. For the purposes of this document, an SSIM value of 0.90 was the threshold for sufficient image quality.

7 Recommendations for future research

The results in Section 4 and the explanation in Section 5 show that the presented framework is already an improvement over the baseline method. However, the current iteration of the algorithm still has several shortcomings and more work needs to be done to solve them.

This chapter discusses several problems with the current algorithm and gives some directions that might be explored to overcome them. Section 7.1 starts with possible solutions for the edge case found during image quality testing. Then, Section 7.2 discusses the possibility to relax the camera offset to allow vertical view offset. Section 7.3 discusses support for multiple light sources and the problems associated with them. Afterwards, Section 7.4 mentions a possible area to improve the runtime of the framework by replacing depth peeling with per-pixel linked lists. Section 7.5 goes over a possible performance improvement for the hierarchical epipolar renderer. A possibly viable method to artificially increase the number of generated views is discussed in Section 7.6. Finally, support for a multitude of lighting models is discussed in Section 7.7.

7.1 Geometry parallel to view direction

During image quality testing, an edge case of the interpolation algorithm was found. The problem was attributed to geometry going parallel to the view direction and coinciding with the view direction vector of the views. As a result, this geometry was not getting captured since geometry is infinitesimally thick. However, other views may see this area as a large object. Thus, to increase the usefulness of the algorithms to more situations, additional processing is required. Note, this is more likely to happen for the center view acquisition and dual view acquisition methods. The hierarchical view acquisition methods suffer less since more distinct camera positions are sampled.

Fortunately, it is possible to detect this type of geometry. The subset of geometry can then be processed for additional information. A naïve method would be to capture the offending geometry using a virtual camera that is orthogonal to the original set of cameras and use the existing epipolar interpolation to fill the holes. A more involved option would be to manually sample the offending geometry and perform epipolar interpolation on the samples. The advantage over the naïve method is reduced overhead since less empty pixels need to be processed.

7.2 Vertical view offset

The introduction chapter posed several restrictions on the placement of a set of viewpoints. One restriction stated that the viewpoints could only have an offset in the horizontal direction. This will lead to a configuration where all the viewpoints lie on a single line. The reasoning behind this restriction is due to the current graphics rasterization pipelines implemented on graphics cards. The current pipelines are currently only capable of efficiently rasterizing 3d geometry to 2d images. As explained in Section 3.1, when the viewpoints are restricted to lie on single line, then the problem of capturing 3d epipolar geometry can be reduced to 2 dimensions. This allows the previously presented algorithm to reap the benefits of the highly-optimized rasterization hardware.

However, with a 2d grid of evenly spaced viewpoints, a naïve implementation would be to treat each row as a separate set of views. A more experimental, but possibly more performant implementation would use the implementation as described to generate all views for the top and bottom row. Then, the resulting images are used as input for the second pass. Each pair of images, from the top and bottom row, in a column is interpolated with only a vertical offset.

However, the method as described in Section 3 needs to slightly adapted. Currently, the epipolar renderer discards any epipolar geometry that cannot be seen in any image by means of depth testing. However, this information might be needed when a rendering with a horizontal offset. As such, this information needs to be captured and interpolated during the horizontal interpolation pass. Secondly, the interpolation method as discussed in Chapter 3 is an approximation. Thus, the input of the horizontal pass might lead to unacceptable results that need to be fixed by an additional pass.

7.3 Multiple light sources

To reduce the computational resources spend on the specular contribution to the final pixel colors, the framework outlined in Chapter 3, approximates the light contribution. This is done by calculating the light contribution in several views, and the computed value is approximated by linear interpolation for the other viewpoints.

A major downside of the current method is, the current support is limited to a single light source. However, current AAA games use up to several hundred lights in a single scene. Thus, to make the current implementation viable for consumption by the masses, support needs to be added for multiple light sources. One issue herein lies in the need to capture the most noticeable characteristics of a specular reflection. More specifically, the brightest spot of a highlight. For a single light source this can be computed, and coincides with the spots where the reflection direction vector intersects with a camera location.

Unfortunately, this no longer holds for multiple light sources. A straightforward way to circumvent this problem is to approximate the specular contribution separately for the different light sources and add together the specular intensities. However, this will lead to an algorithmic complexity which is dependent on the number of views and light sources. This can be reduced to only be dependent on the number of light sources, if the contribution of all the light sources can be described by a single equation.

7.4 Per-pixel linked lists

As described in Section 3, the first stage used by the center and dual epipolar renderers uses depth peeling to uncover occluded geometry. Unfortunately, depth peeling is considered inefficient by current standards. One of the reasons for this is the synchronization needed between the CPU and GPU to read back how many primitives have been rendered. This is worsened by the fact that a synchronization happens after each depth layer and leads to two consequences. Firstly, the CPU must wait on the GPU to finish previously queued rendering commands before it can generate the next set of rendering commands. Secondly, the read command leads to a pipeline flush on the GPU since it requires all commands sent by the CPU to have finished before the readback command. A second reason contributing to the slowness of depth peeling is the need to rasterize the geometry multiple times, once for each depth layer.

Depth peeling has been superseded by several methods. A possible replacement for depth peeling, that also fits the current use case, are per-pixel linked lists [51]. The idea is to build linked lists per pixel where fragments that coincide with a pixel are linked together by a singly linked list. This allows the scene geometry to be captured in a single pass, instead of the multiple passes required when using depth peeling.

However, there are certain practicalities that need to be improved before per-pixel linked lists are a viable option to replace depth peeling. Starting with a method to remove samples that are hidden behind geometry and cannot be seen from any of the views in the final view set. As described in Section 3, depth peeling adds an extra offset to the depth layers to skip any samples that cannot be seen; which leads to a reduction in the number of depth layers. An unfortunate problem with per-pixel linked lists is the order in which geometry is rasterized. Consequently, it is not known during the rasterization pass which fragments can be discarded based on the extra offset. Thus, basic per-pixel linked lists would lead to an inordinate number of pixels that need to be interpolated by the epipolar renderers.

However, it might be possible to apply the extra depth offset by updating the linked lists in an additional pass. This pass would need to sort the samples for each pixel by depth and remove any samples that are within the depth offset as computed by the current method.

The next practicality issue would be to prune the per-pixel linked lists. The current depth peeling implementation uses an erosion-based, conservative visibility estimation method to find any pixels in a depth layer that are always hidden behind geometry from an earlier depth layer and this could be adapted to work per-pixel linked lists. However, this might lead to an imbalance of workload due to linked lists having different lengths. In turn, the divergence in data and branch execution may lead to a significant performance decrease [34, 35, 38].

7.5 Visibility estimation with clustering

The current hierarchical epipolar renderer implementation, as outlined in Section 3, generates the source images by starting with generation of the view corresponding with the left- and rightmost viewpoints. Next, the algorithm decides whether it has enough source images by counting the holes with help of an erosion-based visibility estimation. If more source images are needed, the center location between the two surrounding viewpoints is taken and image is therefore generated. This recursive process is continued until no more source images are needed or until the hierarchical epipolar renderer has run out of time.

The problem with always taking the center between two viewpoints is that it does not necessarily lead to more holes being filled. One example of this not being the case is if the center view is mostly covered by a flat wall.

A solution might be to append an extra step, which computes a new view location that is better. A byproduct of the visibility estimation is an estimation of the location of the holes. It might lead to better and more stable results if the new view location is based on the pixel location around which the most holes are clustered. This pixel location can be projected to view or world space because the camera parameters are known. If the location of the new view is computed such that the largest concentration of holes is roughly centered on the center of the new view, then it might have a better contribution to filling the holes opposed to always taking the center view. One class of clustering algorithms that could be used is k-means clustering. Several GPU based implementations can be found in [26, 40].

7.6 Epipolar view volume interpolation

In its current form, the output of the epipolar rendering pipelines consists of the 3-dimensional epipolar volume. As explained in Chapter 3, each vertical 2-dimensional slice of the volume corresponds with one of the predetermined views. Additionally, each consecutive pair of cameras has the same horizontal offset.

Since the views are ordered in the 3-dimensional volume, it might be possible to do a pixelwise interpolation between two consecutive vertical slices to generate extra views between the original cameras. However, this can only work if the original camera array is sufficiently close together. The idea is that the information belonging in the intermediate views doesn't change dramatically and can thus be inferred from the surrounding views.

However, some questions remain: What kind of interpolation? For example, a linear, pixelwise interpolation or is something more sophisticated necessary? Besides this, may interpolation lead to undesirable image quality? Or is it better to half the inter-camera distance in the original camera array and incur the overhead of the presented epipolar rendering pipeline?

7.7 A multitude of lighting models

Current lighting models separate the light response of a surface into four components. These are the ambient, diffuse, specular and emissive responses. Hereof, the ambient, diffuse and emissive light contributions do not depend on the position of the camera. As such, the color that emits from such surfaces can be interpolated to the different views without needing to be recomputed.

However, the specular light contribution of a surface does depend on the position of the viewer. Thus, the interpolation of this type of light response needs to be explicitly incorporated into the epipolar rendering algorithm.

As described in Section 3.5, the current implementation of the epipolar rendering framework only supports the basic phong reflection model [36]. But it has been superseded by more complex, physically based models such as the Ward anisotropic distribution [49] or the Cook-Torrance model [7]. The epipolar framework uses several assumptions about the used reflection model, but does this also translate to other reflection models? Can more complex reflection models also be supported but with their own set of assumptions? And lastly, does the epipolar rendering framework not lose any of the key details of other reflection models due to the interpolation?

References

[1] Ahmed, N. et al. 1974. Discrete cosine transform. *IEEE transactions on*. (1974).

[2] Assarsson, U. and Moller, T. 2000. Optimized view frustum culling algorithms for bounding boxes. *Journal of graphics tools*. (2000).

[3] Bavoil, L. and Myers, K. 2008. Order independent transparency with dual depth peeling. *NVIDIA OpenGL SDK*. (2008).

[4] Blackmer, R. et al. 2009. GL_ATI_meminfo. ATI Technologies; Khronos.

[5] Bonneel, N. et al. 2013. Example-based video color grading. *ACM Trans. Graph.* (2013).

[6] Brown, P. and Zolnowski, E. 2012. ARB_transform_feedback2. AMD; NVIDIA; Khronos Group Inc.

[7] Cook, R.L. and Torrance, K.E. 1982. A Reflectance Model for Computer Graphics. *ACM Transactions on Graphics*. 1, 1 (Jan. 1982), 7–24.

[8] Coorg, S. and Teller, S. 1997. Real-time occlusion culling for models with large occluders. *Proceedings of the 1997 symposium on Interactive*. (1997).

[9] Décoret, X. et al. 2003. Erosion Based Visibility Preprocessing. *Proceedings of the Eurographics Symposium on Rendering*. (2003).

[10] Didyk, P. et al. 2010. Perceptually-motivated real-time temporal upsampling of 3D content for high-refresh-rate displays. *Computer Graphics Forum*. 29, 2 (2010), 713–722.

[11] Engel, W.F. 2009. *ShaderX7: Advanced Rendering Techniques*. Charles River Media.

[12] Everitt, C. 2001. Interactive order-independent transparency. *White paper, nVIDIA*. (2001).

[13] Foley, J. 1994. Human luminance pattern-vision mechanisms: masking experiments require a new model. *JOSA A*. (1994).

[14] Haemel, N. 2009. AMD_gpu_association. AMD; Khronos.

[15] Halle, M. Multiple Viewpoint Rendering.

[16] Hartley, R. and Zisserman, A. 2004. *Multiple view geometry in computer vision*. Cambridge University Press.

[17] Hasselgren, J. and Akenine-Möller, T. 2006. An Efficient Multi-View Rasterization Architecture. *Eurographics Symposium on Rendering*. (2006), 61–72.

[18] Havran, V. et al. 2003. Exploiting temporal coherence in ray casted walkthroughs. *Proceedings of the 19th spring conference on Computer graphics - SCCG '03* (New York, New York, USA, 2003), 149–155.

[19] He, Y. et al. 2014. Extending the graphics pipeline with adaptive, multi-rate shading. *ACM Transactions on Graphics*. 33, 4 (Jul. 2014), 1–12.

[20] Herzog, R. et al. 2010. Spatio-temporal upsampling on the GPU. *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games - I3D 10* (New York, New York, USA, 2010), 91–98.

[21] Jensen, H.W. 2001. *Realistic image synthesis using photon mapping*. A K Peters.

[22] Jevans, D.A. 1992. Object Space Temporal Coherence for Ray Tracing. *Proceedings of the conference on Graphics interface '92* (Vancouver, 1992), 176–183.

[23] Jimenez, J. et al. 2011. Filtering approaches for real-time anti-aliasing. *ACM SIGGRAPH*. (2011).

[24] Kim, Y. et al. 2016. Adaptive undersampling for efficient mobile ray tracing. *The Visual Computer*. 32, 6–8 (Jun. 2016), 801–811.

[25] Lee, S. et al. 2010. Real-time lens blur effects and focus control. *ACM Transactions on Graphics (TOG)*. (2010).

[26] Li, Y. et al. 2010. Speeding up K-Means Algorithm by GPUs. *2010 10th IEEE International Conference on Computer and Information Technology* (Jun. 2010), 115–122.

[27] Lichtenbelt, B. et al. 2010. ARB_transform_feedback3. AMD; ARM; NVIDIA; Khronos Group Inc.

[28] Lighting you up in Battlefield 3: 2011. *http://www.frostbite.com/wp-content/uploads/2013/05/GDC11_LightingYouUpInBattlefield3.pdf*. Accessed: 2017-05-03.

[29] Luebke, D. et al. 2003. *Level of detail for 3D graphics*. Morgan Kaufmann.

[30] Martin, W. et al. 2002. Temporally Coherent Interactive Ray Tracing. *Journal of Graphics Tools*. 7, 2 (Jan. 2002), 41–48.

[31] Maule, M. et al. 2012. Memory-Efficient Order-Independent Transparency with Dynamic Fragment Buffer. *2012 25th SIBGRAPI Conference on Graphics, Patterns and Images* (Aug. 2012), 134–141.

[32] Nehab, D. et al. 2007. Accelerating Real-time Shading with Reverse Reprojection Caching. *Proceedings of the 22Nd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware*. (2007), 25–35.

[33] No longer just a programming niche, deferred rendering is becoming an increasingly popular technique on consoles too? 2009. *http://www.develop-online.net/tools-and-tech/build-deferred-rendering/0116368*. Accessed: 2017-05-03.

[34] Nvidia Corporation 2012. OpenCL Programming Guide for the CUDA Architecture. *Nvidia*. 41, (2012), 371–379.

[35] Nvidia Corporation 2009. PTX : Parallel Thread Execution ISA Version 1.4. (2009), 133.

[36] Phong, B.T. and Tuong, B. 1975. Illumination for computer generated pictures. *Communications of the ACM*. 18, 6 (Jun. 1975), 311–317.

[37] Ponomarenko, N. et al. 2007. On between-coefficient contrast masking of DCT basis functions. *Proceedings of the*. (2007).

[38] Sartori, J. and Kumar, R. 2013. Branch and data herding: Reducing control and memory divergence for error-tolerant GPU applications. *IEEE Transactions on Multimedia*. (2013).

[39] Sellers, G. 2010. ARB_transform_feedback_instanced. AMD; Khronos Group Inc;

[40] Shalom, S. et al. 2008. Efficient k-means clustering using accelerated graphics processors. *International Conference on Data*. (2008).

[41] Sitthi-amorn, P. et al. 2008. Automated reprojection-based pixel shader optimization. *ACM Transactions on Graphics*. 27, 5 (2008), 1.

[42] Stroyan, H. et al. 2003. NVX_gpu_memory_info. NVIDIA Corporation; Khronos.

[43] Tawara, T. et al. 2004. Exploiting temporal coherence in global illumination. *Proceedings of the 20th spring conference on Computer graphics - SCCG '04* (New York, New York, USA, 2004), 23.

[44] Unity 3 Feature Preview – Deferred Rendering: 2010. *https://blogs.unity3d.com/2010/09/09/unity-3-feature-preview-deferred-rendering/*. Accessed: 2017-05-03.

[45] Unreal Engine 4 - Rendering Overview: *https://docs.unrealengine.com/latest/INT/Engine/Rendering/Overview/index.html*. Accessed: 2017-05-03.

[46] Wang, R. et al. 2009. An efficient GPU-based approach for interactive global illumination. *ACM Transactions on Graphics*. 28, 3 (Jul. 2009), 1.

[47] Wang, Z. et al. 2004. Image quality assessment: from error visibility to structural similarity. *IEEE transactions on*. (2004).

[48] Wang, Z. and Bovik, A. 2009. Mean squared error: Love it or leave it? A new look at signal fidelity measures. *IEEE signal processing magazine*. (2009).

[49] Ward, G.J. et al. 1992. Measuring and modeling anisotropic reflection. *ACM SIGGRAPH Computer Graphics*. 26, 2 (Jul. 1992), 265–272.

[50] Watson, A. and Solomon, J. 1997. Model of visual contrast gain control and pattern masking. *JOSA A*. (1997).

[51] Yang, J.C. et al. 2010. Real-Time Concurrent Linked List Construction on the GPU. *Computer Graphics Forum*. 29, 4 (Aug. 2010), 1297–1304.

[52] Yang, L. et al. 2008. Geometry-aware framebuffer level of detail. *Computer Graphics Forum*. 27, 4 (2008), 1183–1188.

[53] Yang, L. et al. 2011. Image-based bidirectional scene reprojection. *ACM Transactions on Graphics*. 30, 6 (2011), 1.

Appendix A Additional image quality data

The developed algorithms have been evaluated on the quality of the generated views. The most interesting and relevant findings were discussed in Section 4.5. However, several additional, complementary artifacts were produced during the analyses of the image quality.

To locate the troublesome areas in images, Section 4.5 showed several error maps by means of the absolute error between a pair of interpolated and reference images. Figure A.1 and Figure A.2 show additional error maps. Like the previous error maps, the most notable errors are located on pixels which transition between different surfaces.

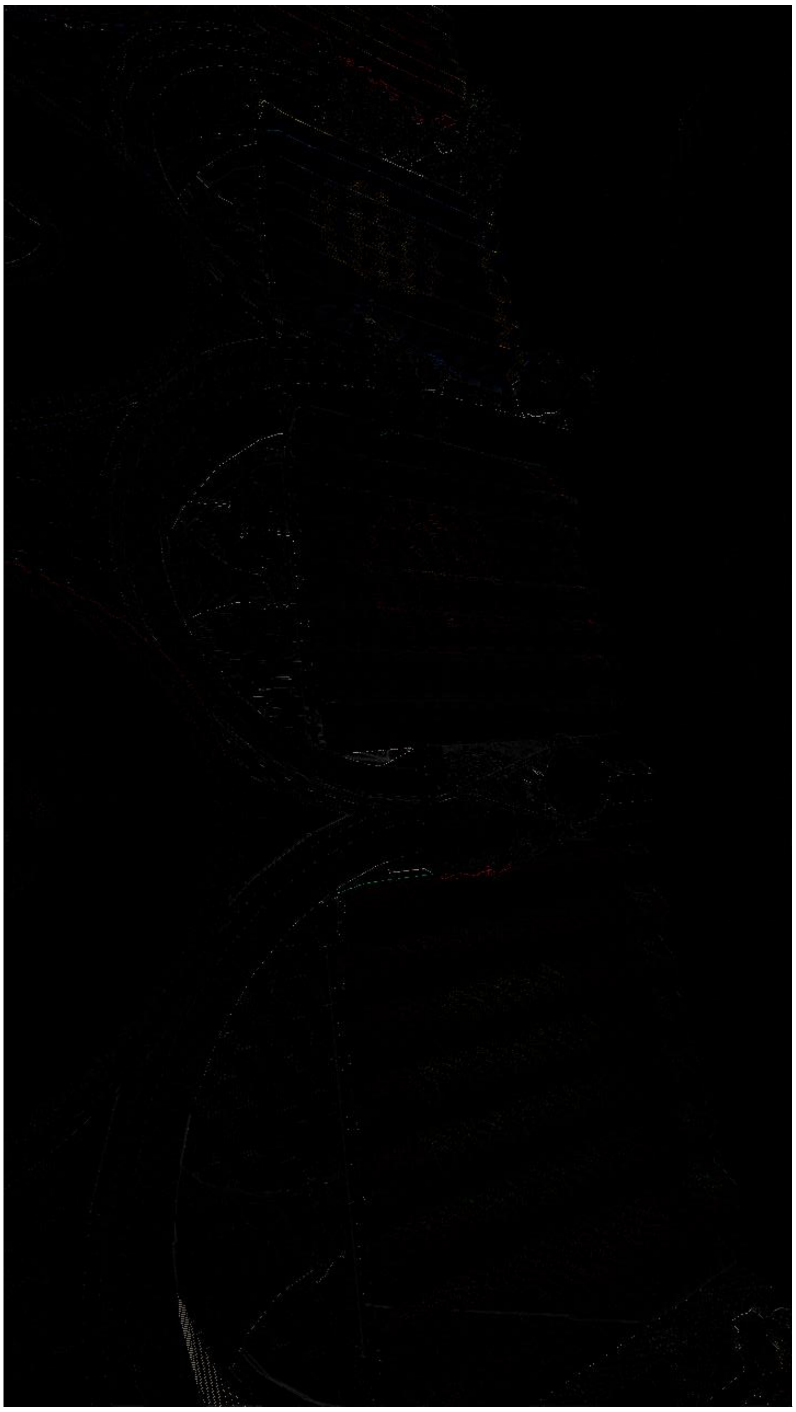

Figure A.1. *Absolute error of view 120. This view was generated using no optimizations or extensions. Test scene: Sponza; render pipeline: dual epipolar; view set size: 500 views.*

Figure A.2. *Absolute error of view 252. This view was generated using no optimizations or extensions. Test scene: Sponza; render pipeline: dual epipolar; view set size: 500 views.*

Appendix B Derivation of time complexity

Part of the mission statement of the presented research was to produce an algorithm that generates a set of views more efficiently than brute force computation. To understand the efficiency of an algorithm a valuable property is the time complexity, as it links the size of the input to the number of basic operations to produce a result. Furthermore, it allows the efficiency of two algorithms to be compared independently of a specific hardware platform. Besides, it is a precursor for the measured runtime when compared to a baseline.

One more benefit of a time complexity analysis is the ability to show, with some certainty, what an algorithm does for unreasonably large input sizes. In other words, data sets which are too large current generation hardware. An example that applies to the presented algorithm is the limited amount of available memory to compute more than 1000 views.

For the comparison of the time complexity between two algorithms it is necessary that the analysis assigns the same cost to basic operations. To this end, the time complexity analyses in this document operate under the uniform-cost measurement model. Regardless of the bit sizes of the numbers involved, this model assigns a constant cost to every basic operation. Some exemplary basic operations are variable assignment, multiplication or comparison.

The use of the chosen cost model is justified because the computations used by the presented algorithms don't have a strict need for more precision than is provided by most commonly used hardware. On top of this, basic operations on actual hardware have similar runtimes. Furthermore, more complex cost models would only clutter the analysis whilst not providing more information about the performance characteristic of the algorithm.

The remainder of this appendix shows detailed analyses of respectively a deferred rendering pipeline, center view acquisition, hierarchical view acquisition and epipolar view interpolation in Appendix B.1, Appendix B.2, Appendix B.3 and Appendix B.4.

Appendix B.1 Deferred rendering

The determination of the time complexity for the baseline deferred rendering pipeline is based upon the pseudocode shown in Code listing B.1.1. For completeness of the time complexity analysis, several support functions that cover the standard graphics pipeline operations are included. The pseudocode is shown as Code listing B.1.2.

Starting with the time complexity of *RasterizeGeometry*. Several algorithms exist to rasterize geometry into fragments. Examples are Bresenham's line rasterization algorithm or triangle rasterization based on half-spaces. Unfortunately, the exact implementation by the various GPU manufacturers is a closely guarded secret. However, it is a reasonable assumption that the time complexity is always a function of the amount of geometry because each piece of geometry can be distinct from all others and thus must be rasterized separately. So, for the purposes in this report, the amount of geometry is summarized by v and the time complexity of *RasterizeGeometry* is aggregated into and approximated by $O(v)$, where $O(x)$ is the order of function x. Furthermore, the different rendering pipelines discussed in this report are not inherently bound to a rasterization algorithm. As such, the rasterization is assumed to be identical for the different pipelines.

The time complexity of *DetermineVisibleFragments* is determined by summation of the time complexity of the return statement on line 15 of Code listing B.1.2, the comparison, return and assignment statements from lines 1 through 5, and the foreach loop spanning lines 6 through 13. A return statement is assumed identical to any basic operation, and as such takes a constant amount of time. Lines 1 through 4 and 15 have a total time complexity of 3. The assignment on line 5 is the combined time complexities of an assignment and set creation. The set creation takes $O(p)$ operations because F contains $O(p)$ distinct tuples. As such, the time complexity of line 5 is $1 + O(p)$. The foreach loop consists of two comparisons, one set complement, one set union and an assignment. The set operations can be done in constant time if the set is based on an hashmap with key (x, y) and tuples with similar key are sorted based on their z-coordinate. The loop is executed $O(p)$ times, where p is the number of pixels in an image. Thus, the time complexity of *DetermineVisibleFragments* is $O(p)$. See equation B.1.1 for the derivation.

$$3 + 1 + O(p) + O(p) * 5 + 1 \leq O(p * 6 + 5) = O(p) \tag{B.1.1}$$

Lastly, the time complexity of *CalculateLighting* is the sum of one assignment, one return, and a foreach-loop. The loop runs over each pixel, and thus loops $O(p)$ times. The inner body of the loop consists of 3 assignments and one disjoint union. The time complexity hereof is 4. Furthermore, the loop body references a function to compute the color of a fragment. The time complexity is dependent on the used lighting model, and is outside the scope of the presented research. So, for this report, it is represented by the variable l, and is assumed to be constant for each fragment. This leads to a complete time complexity of $O(p)$ for *CalculateLighting*. See equation B.1.2 for the derivation.

$$2 + O(p) * (4 + l) \leq O(2 + 4 * p + l * p) = O(p) \tag{B.1.2}$$

Input: Set of camera projection parameters MVP. One for each camera, such that MVP_i are the projection parameters for camera i. Furthermore, scene geometry G.
Output: An ordered set of views V, such that V_i contains the projection of G as described by the camera projection parameters MVP_i.

```
AcquireDeferredViews($MVP$, $G$):

V := {$\emptyset$}
for $i$ = 0 to |$V$|:
    $F_i$ := RasterizeGeometry($MVP_i$, $G$)
    $F_{v,i}$ := DetermineVisibleFragments($F$, {$\emptyset$})
    if $F_{v,i} = \emptyset$:
        continue
    end if

    $V_i$ := CalculateLighting($F_{v,i}$)
    $V$ := $V \cup \{V_i\}$
end for

return $V$
```

Code listing B.1.1. *Pseudocode for a basic deferred rendering pipeline. The functions shown refer to several helper functions. These are shown in Code listing B.1.2. The time complexity analysis for the deferred rendering pipeline is done based on the shown mathematical description.*

With all the support functions analyzed, the time complexity of the deferred rendering pipeline can be determined. The for-loop body runs x times, once for each view. The loop body uses 4 assignments, one comparison and one disjoint set union. Furthermore, *RasterizeGeometry*, *DetermineVisibleFragments* and *CalculateLighting* are referenced once. Thus, the entire loop has a time complexity of $x * (O(v + p)$. Refer to equation B.1.3 for the derivation.

$$x * (O(v) + O(p) + O(p) + 4 + 1 + 1) =$$

$$x * (O(v + 2p) + 6) \leq$$

$$x * O(v + 2p + 6) =$$
$$x * O(v + p) \tag{B.1.3}$$

Additionally, *AcquireDeferredViews* uses one assignment and one return statement. As such, the time complexity for the deferred rendering pipeline is $O(x * (v + p))$. The derivation is shown in equation B.1.4.

$$2 + x * O(v + p) \leq O\big(2 + x * (v + p)\big) = O\big(x * (v + p)\big) \tag{B.1.4}$$

```
RasterizeGeometry(Projection parameters $MVP$, Geometry $G$):

returns a set of tuples $(x,y,z)$ such that the geometry $G$ projected in accordance
with the projection matrix MVP intersects with $(x,y,z)$, i.e. the geometry is
discretized into fragments.
```

```
DetermineVisibleFragments(Fragments $F$, Occluding Fragments $F_o$):

if $F_o = \{\emptyset\}$:
      return $F$
end if

$F_v$ := $\{(x,y,1) \mid (x,y,\_) \in F\}$
foreach fragment $(x,y,z) \in F$ do:
    if $F_v$ contains $(x,y,z_d)$ such that $z_d < z$ or
       $F_o$ contains $(x,y,z_o)$ such that $z_o > z$:
        continue
    end if

    $F_v$ := $(F_v \setminus \{(x,y,\_)\}) \cup \{(x,y,z)\}$
end foreach

return $F_v$
```

```
CalculateLighting(Fragments $F$):

$P$ := $\{\emptyset\}$
foreach fragment $(x,y,z) \in F$ do:
      $c$ := color of $(x,y,z)$, computed according to lighting model
      $p$ := $(x,y,z,c)$
      $P$ := $P \cup \{p\}$
end foreach

return $P$
```

Code listing B.1.2. *Pseudocode of several helper functions referenced by the different rendering pipelines. The functions shown in this code listing give a high-level, mathematical overview of core graphics pipeline parts and are included in the time complexity analyses of the different pipelines.*

Appendix B.2 Center view acquisition

The time complexity analysis for the center view acquisition algorithm is based on the pseudocode shown in Code listing B.2.1. This algorithm refers to the same support functions as the *AcquireDeferredViews* function from the previous section. For brevity, the same derivation won't be repeated in this section. See Appendix B.1 for the details on the derivation of the time complexity for the referenced support functions.

Input: Center camera projection parameters MVP. Furthermore, scene geometry G.
Output: A partially ordered set of textures V, such that V_i corresponds with depth layer i and where $0 \leq i \leq n. i = 0$ is the first depth layer and $i = n$ coincides with the farthest depth layer as seen from the center camera.

```
    AcquireCenterSourceViews($MVP$, $G$):

 1  $V$ := {$\emptyset$}
 2
 3  $F$ := RasterizeGeometry($MVP$, $G$)
 4  $F_{v,0}$ := DetermineVisibleFragments($F$, {$\emptyset$})
 5  $V_0$ := CalculateLighting($F_{v,0}$)
 6  $V$ := $V \cup \{V_0\}$
 7
 8  $i$ := 1
 9  while true:
10          $F_{v,i}$ := DetermineVisibleFragments($F$, $F_{v,i-1}$)
11          if $F_{v,i} = \{\emptyset\}$:
12              break
13          end if
14
15          $V_i$ := CalculateLighting($F_{v,i}$)
16          $V$ := $V \cup \{V_i\}$
17
18          $i$ := $i$ + 1
19  end while
20
21  return $V$
```

Code listing B.2.1. *Pseudocode for center source view acquisition pipeline. AcquireCenterSourceViews refers to several helper functions. These are shown in Code listing B.1.2. The time complexity analysis for the center source view acquisition pipeline is done based on this mathematical description.*

The time complexity of lines 1 through 8 of Code listing B.2.1 is $6 + 1 + O(v) + O(p) + O(p)$, which is the sum for respectively 6 assignments, one union between an empty and non-empty set, the reference to *RasterizeGeometry*, the reference to *DetermineVisibleFragments* and, the reference to *CalculateLighting*.

The operations within the depth peeling loop body consists of four assignments, one comparison, one addition, one disjoint set addition, the reference to *DetermineVisibleFragments* and the reference to *CalculateLighting*. The combined time complexity is $4 + 1 + 1 + 1 + O(p) + O(p)$ for the loop body. In theory, the number of loop iterations is unbounded. Fortunately, the number of layers captured using depth peeling can be bounded to ten layers, if the optimization discussed in Section 3.4.1 is applied. As shown by equation B.2.1, the depth peeling loop complexity is $O(p)$.

$$\begin{gathered} 10 * \left(4 + 1 + 1 + 1 + O(p) + O(p)\right) = \\ 10 * \left(7 + 2 * O(p)\right) = \\ 70 + 20 * O(p) \leq \\ O(70 + 20 * p) = \\ O(p) \end{gathered} \tag{B.2.1}$$

To complete the time complexity analysis, the return statement for line 21 needs to be added to the time complexity of *AcquireCenterSourceViews*. The total time complexity is $O(v + p)$. The derivation is shown in equation B.2.2.

$$6 + 1 + O(v) + O(p) + O(p) + O(p) + 1 =$$
$$O(v) + 2 * O(p) + 8 \leq$$
$$O(v + 2 * p + 8) =$$
$$O(v + p) \qquad (B.2.2)$$

Appendix B.3 Hierarchical view acquisition

The time complexity analysis for the hierarchical view acquisition algorithm is based on the pseudocode shown in Code listing B.3.1. This algorithm refers to the same support functions as discussed in Appendix B.1. For brevity, the derivation of the support functions won't be repeated in this section. See Appendix B.1 for a detailed derivation of the time complexity for the referenced support functions. However, additional functions are also referenced. The corresponding pseudocode is listed in Code listings B.3.2 and B.3.3.

RenderView is composed of five assignments, one return statement, one call to the *RasterizeGeometry* function, one call to the *DetermineVisibleFragments* function and one call to the *CalculateLighting* function. The time complexity for these operations is $5 + 1 + O(v) + O(p) + O(p)$. Furthermore, two statements to get the current time are made. The purpose of the timepoints is to deduce the elapsed time during the execution of *RenderView* by means of a subtraction. A common hardware implementation is to provide special registers that contain the timestamp as an integer. As such, getting a timestamp shouldn't be more time consuming than assigning a value to a variable. So, fetching a timestamp is assumed to take a single, constant unit of time. As such, 3 units of time need to be considered for the time management. Two units for getting the timestamps, and one unit of time to compute the elapsed time by subtracting the two timestamps. As such, the time complexity for *RenderView* is $O(v + p)$. See equation B.3.1 for the derivation.

$$5 + 1 + O(v) + O(p) + O(p) + 3 =$$
$$O(v) + 2 * O(p) + 9 \leq$$
$$O(v + 2 * p + 9) =$$
$$O(v + p) \qquad (\mathrm{B}.3.1)$$

The time complexity of *ComputeVisibleSet* is mostly composed of the computation time of *ErodeView* since it consists of 2 assignments, one return statement and 2 calls to *ErodeView*. Additionally, it uses a set union where the sizes of the sets are dependent on the number of pixels in the given image. So, the set union can be done in $O(p)$ operations. As such, equation B.3.2 derives a time complexity of $O(EV + p)$ for the *ComputeVisibleSet* function, where EV stands in for the time complexity of *ErodeView*.

$$2 + 1 + 2 * EV + O(p) = 3 + 2 * EV + O(p) \leq O(EV + p) \qquad (\mathrm{B}.3.2)$$

As shown in Code listing B.3.2, the *ErodeView* function starts off with an assignment on line 1. This is followed by a loop spanning lines 2 through 8. The loop body is executed once for every fragment in V_t, and thus runs $O(p)$ times. The condition of the if-statement corresponds to an edge detection algorithm. In Section 3.3.1, it was explained that a mean-based local threshold with a 5-pixel wide 1-dimensional window was used in the test implementation and this will be the basis for the time complexity analysis. The computation of the mean takes four additions and 1 divide. The threshold operation equates to one comparison. The remaining statements in the if-branch contains an assignment and a union of a disjoint set. The else-branch consists of identical operations. In the end, this leads to a time complexity of $1 + O(p) * (6 + 1 + 1)$ for lines 1 through 8. Line 10 consists of an assignment on line 10, which counts as a basic operation, and is followed by a loop. This loop runs over each tuple in EdgeMap, and the loop body executes $O(p)$ times. The reason being the number of contained elements is equal to the tuple count of V_t because one tuple is added to EdgeMap for each tuple in V_t, and V_t contains $O(p)$ items. The loop body consists of two assignments, 1 disjoint set union and a sum over p tuples in the worst-case scenario**.** As such, the time complexity of lines 10 through 14 sums to $1 + O(p) * (2 + 1 + p)$. Furthermore, the variable ErodedView gets assigned an empty set on line 16 which counts as a basic operation. Before ending *ErodeView* with the return statement on line 28, there is a foreach-loop spanning lines 17 through 26. The projections on lines 18 and 19 in the loop body are implemented as matrix projections, where the matrices have a 4x4 dimension. As such, the time complexity analysis counts the reprojections as several matrix vector multiplications. Each matrix vector multiplication takes sixteen multiplications and twelve additions. So, including the two assignments, the unprojection and reprojection are done in 58 basic operations. The condition of the if-else branch is one look-up in PrefixSumMap that can be done in $O(1)$ operations if the tuples are stored in order based on the x and y coordinates. Inside the if-else statement an assignment and disjoint set union of which the combined time complexity is 2. The foreach-loop is executed $O(p)$ times because PrefixSumMap contains a single tuple for each element in EdgeMap which was determined to hold $O(p)$ tuples, bringing the time complexity of the foreach-loop to $O(p) * (58 + 1 + 1 + 1 + 1) = 62 * O(p)$. As shown by the derivation in equation B.3.3, the time complexity of the *ErodeView* function equates to $O(p^2 + p)$.

$$\begin{gathered} 1 + O(p) * (6 + 1 + 1) + 1 + O(p) * (2 + 1 + p) + 1 + O(p) * (58 + 1 + 1 + 1 + 1) = \\ p * O(p) + 8 * O(p) + 3 * O(p) + 62 * O(p) + 3 = \\ p * O(p) + 73 * O(p) + 3 \leq \\ O(p * p + p) = \\ O(p^2 + p) \end{gathered} \tag{$B.3.3$}$$

CountHoles is composed of one assignment, one return statement and one foreach-loop, executing once for each pixel given as input. The loop body performs a constant number of operations. One comparison, one assignment and one addition. As such, the time complexity of *CountHoles* is derived to be $O(p)$ in equation B.3.4.

$$1 + 1 + O(p) * (1 + 1 + 1) = 3 * O(p) + 2 \leq O(3 * O(p) + 2) = O(p) \tag{B.3.4}$$

Input: An ordered set of camera projection parameters MVP_i, where $0 \leq i \leq n$ and i is the index of the camera. This set is ordered such that $i = 0$ corresponds with the leftmost camera parameters and $i = n$ corresponds with the rightmost camera parameters. Furthermore, quality threshold H_t, max renderbudget B and scene geometry G are given as input.
Output: A set of textures V, such that V_i corresponds with the i^{th} source view.

```
AcquireRangedSourceViews($MVP$, $G$, $H_t$, $B$):

$T$ := 0
RenderQueue := empty queue

$(V_0, T_0)$ := RenderView(0, $MVP_0$, $G$)
$(V_1, T_1)$ := RenderView($n$, $MVP_n$, $G$)
$V$ := $\{V_0\} \cup \{V_1\}$

RenderQueue.Push($(V_0, V_1)$)
$T$ := $T + T_0 + T_1$

while RenderQueue not empty and $T < B$:
        $(V_l, V_r)$ := RenderQueue.Pop()
        $Index$ := $(l + r) / 2$

        $V_{vis}$ := ComputeVisibleSet($T_l$, $MVP_l$, $T_r$, $MVP_r$, $MVP_{Index}$)
        $H$ := CountHoles($V_{vis}$)
        if $H < H_t$:
                $(V_i, T_i)$ := RenderView($Index$, $MVP_{Index}$, $G$)
                $V$ := $V \cup \{V_i\}$

                RenderQueue.Push($(V_l, V_i)$)
                RenderQueue.Push($(V_i, V_r)$)
                $T$ := $T + T_i$
        end if
end while

return V
```

```
RenderView(ViewIndex $I$, Projection parameters $MVP$, Geometry $G$):

$Time_s$ := start time

$F$ := RasterizeGeometry($MVP$, $G$)
$F_v$ := DetermineVisibleFragments($F$, $\{\emptyset\}$)
$V$ := CalculateLighting($F_v$)

$Time_e$ := end time

return $(V, Time_e - Time_s)$
```

Code listing B.3.1. *Pseudocode for the hierarchical view acquisition pipeline. The shown functions refer to several helper functions which are shown in Code listing B.1.2. Furthermore, the algorithmic description of the erosion based image quality metric is given in Code listing B.3.2 and B.3.3. The time complexity analysis for the hierarchical view acquisition pipeline is done based on this mathematical description.*

AcquireRangedSourceViews, shown in Code listing B.3.1, begins with two assignments. This is followed by three assignments, two calls to *RenderView* and a union of two 1-element sets. As determined previously, the time complexity of *RenderView* is $O(v + p)$. The queue push operation takes a constant number of operations. So, its time complexity is $O(1)$, and is followed by two additions and one assignment. In total, the time complexity of lines 1 through 9 is $2 + 3 + 2 * O(v + p) + 1 + O(1) + 2 + 1 = 2 * O(v + p) + O(1) + 9$. The following while-loop body starts with a queue pop operation, which takes $O(1)$ operations. The calculation of Index takes one addition, one division and an assignment. Line 15 has a time complexity of $1 + O(p^2 + p + p)$ for the assignment and call to *ComputeVisibleSet*. Line 16 has a time complexity of $1 + O(p)$ because of the assignment and call to *CountHoles*. The if-branch condition is a comparison between two numbers, and counts as one basic operation. The body of the if-branch consists of three assignments, one disjoint set union, one addition, two queue push operations, and one call to *RenderView*. The time complexity for the while-loop body is $O(1) + 1 + 1 + 1 + 1 + O(p^2 + p + p) + 1 + O(p) + 1 + 3 + 1 + 1 + 2 * O(1) + O(v + p) = O(p^2 + 2 * p) + O(v + p) + O(p) + 3 * O(1) + 11$. Lastly, the return statement accounts for one more basic operation.

The observant reader might have noticed that the number of executions of the loop body was not determined. Unfortunately, a general, platform independent solution for the time complexity of *AcquireRangedSourceViews* cannot be determined because the while-loop condition is dependent on the outcome of the image quality heuristic, which cannot be theoretically bound. Or, it is dependent on the ratio between the time to render a single view and the render budget, which makes it platform dependent.

For the remainder of this time complexity analysis it is assumed that *RenderView* completes in the same time as it takes to do T operations. Furthermore, a renderbudget of B is assumed. Then, the worst-case time complexity of *AcquireRangedSourceViews* is $2 * O(v + p) + O(1) + 9 + T/B * (O(p^2 + 2 * p) + O(v + p) + O(p) + 3 * O(1) + 11) + 1$.

ComputeVisibleSet(Left view V_l, MVP_l, Right view V_r, MVP_r, MVP_m):

```
$V_{l,e}$ := ErodeView($V_l$, $MVP_r$, $MVP_l$)
$V_{r,e}$ := ErodeView($V_r$, $MVP_l$, $MVP_r$)

return $V_{l,e} \cup V_{r,e}$
```

ErodeView(Target view V_t, Projection parameters MVP, Projection parameters MVP_u):

```
$EdgeMap$ := $\{\emptyset\}$
foreach fragment $(x,y,z) \in V_t$ do:
      if $p$ corresponds with an edge of object in scene geometry:
            $EdgeMap$ := $EdgeMap \cup \{(x,y,z,1)\}$
      else
            $EdgeMap$ := $EdgeMap \cup \{(x,y,z,0)\}$
      end if
end foreach

$PrefixSumMap$ := $\{\emptyset\}$
foreach tuple $(x,y,z,mark) \in EdgeMap$ do:
      $PrefixSum$ := $\sum_{i=0}^{x} mark$, where $(i,y,z,mark) \in EdgeMap$
      $PrefixSumMap$ := $PrefixSumMap \cup \{(x,y,z,PrefixSum)\}$
end foreach

$ErodedView$ := $\{\emptyset\}$
foreach tuple $(x,y,z,prefix\_sum) \in PrefixSumMap$ do:
      $x_w$ := unproject $x$ to world space with parameters $MVP_u$
      $x_r$ := reproject $x$ to image space with parameters $MVP$

      if $(x_r,y,z,eroded\_sum) \in PrefixSumMap$ and $prefix\_sum \neq eroded\_sum$:
            $ErodedView$ := $ErodedView \cup \{(x,y,1)\}$
      else
            $ErodedView := ErodedView \cup \{(x,y,0)\}$
      end if
end foreach

return $ErodedView$
```

Code listing B.3.2. *Pseudocode for the image-based erosion of geometry to compute the potentially visible geometry when looking from other positions than the provided MVP_l and MVP_r. The time complexity analysis of ComputeVisibleSet and ErodeView is based on the pseudocode shown in this Code listing.*

CountHoles(Eroded view V_e):

```
Count := 0
foreach pixel $(x,y,mark) \in V_e$ do:
      if $mark \neq 1$:
            Count := Count + 1
      end if
end foreach

return Count
```

Code listing B.3.3. *Pseudocode for counting the number of holes in an eroded image. The count returned by this function is compared against a user-provided threshold, and is the image quality decision-rule. This Code listing is the basis for the time complexity analysis of the function CountHoles.*

Appendix B.4 Epipolar view interpolation

The time complexity analysis of the epipolar view interpolation algorithm is done based on the pseudocode shown in Code listing B.4.1, Code listing B.4.2 and Code listing B.4.3. Before commencing the time complexity analysis of the epipolar view interpolation algorithm, shown as the function *AcquireInterpolatedViews*, the time complexities of *CaptureActivePixels* and *DrawDepthLayer* need to be known.

CaptureActivePixels(Source view v):

```
$ActivePixels$ := {∅}
foreach fragment $(x, y, z, c) \in v$:
    if $z$ = 0:
        continue
    end if

    $Stride$ := 0
    $ActivePixels$ := $ActivePixels \cup \{(x, y, z, c, Stride)\}$
end foreach

return $ActivePixels$
```

EmitVertex(*Coords*, Projection parameters *MVP*, *α*, *n*, Color *c*):

```
$x_{epi}$ := $x$-coordinate from $Coords$ after projection to clip space using $MVP$
$y_{epi}$ := $\frac{((1-\alpha)*0.5+\alpha*(n-0.5))}{n} * 2 - 1$
$z_{epi}$ := $z$-coordinate from $Coords$

return $(x_{epi}, y_{epi}, z_{epi}, c)$
```

Code listing B.4.1. *Mathematical description of several support functions used by the view interpolation algorithm.*

The for-loop body in the *CaptureActivePixels* function uses one comparison, two assignments and a disjoint-set union, with a time complexity of 4. The loop is executed $O(p)$ times because it was determined in Appendix B.2 and Appendix B.3 that a view contains $O(p)$ tuples. Additionally, *CaptureActivePixels* has an assignment and return statement. As such, the time complexity for *CaptureActivePixels* is $O(p)$. See equation B.4.1 for the complete time complexity derivation.

$$1 + O(p) * 4 + 1 = 4 * O(p) + 2 \leq O(4 * O(p) + 2) = O(p) \qquad \text{(B.4.1)}$$

The function *EmitVertex* has a time complexity of $O(1)$. The calculation of x_{epi} is counted as a matrix-vector multiplication of a 4-by-4 matrix and 4-dimensional vector. Identical to the reprojection in *ErodeView*, as analyzed in Appendix B.3. The time complexity of this is 29, including the assignment. The calculation for y_{epi} consists of four additions or subtractions, four multiplications or divisions and one assignment. These operations have a time complexity of 9. Computing z_{epi} consists of a read and assignment, and has a time complexity of 2. Furthermore, an additional return statement must be counted. All aspects considered, this leads to the time complexity derivation as shown in equation B.4.2.

$$29 + 9 + 2 + 1 = 41 \leq O(41) = O(1) \qquad \text{(B.4.2)}$$

```
DrawDepthLayer($ActivePixels$, Projection parameters $MVP_{v,i}$; $MVP_l$; $MVP_r$, $n$):

$CameraPosition_l$ := Get position from $MVP_l$
$CameraPosition_r$ := Get position from $MVP_r$
$CameraPosition_{v,i}$ := Get position from $MVP_{v,i}$

$MaxViewOffset$ := $|| CameraPosition_r - CameraPosition_l ||$
$ViewDistance$ := $\frac{||CameraPosition_{v,i} - CameraPosition_l||}{MaxViewOffset}$

$Disparity_l$ := $ViewDistance * MaxViewOffset$
$Disparity_r$ := $(1 - ViewDistance) * MaxViewOffset$

$EpipolarTriangles$ := $\{ \emptyset \}$
foreach fragment $(x, y, z, c_{l,d}, Stride) \in ActivePixels$:

    // Compute epipolar plane end points corresponding to current fragment
    $Coords_{l,v}$ := Unproject $(x, y, z)$ to view space using $MVP_{v,i}$
    $StartCoords_{l,v}$ := $Coords_{l,v} + (Disparity_l, 0, 0)$
    $EndCoords_{l,v}$ := $Coords_{l,v} - (Disparity_r, 0, 0)$

    Get color $c_{r,d}$ proper for $x + Stride$
    $Coords_{r,v}$ := Get coordinates proper for $x$ + $Stride$ and
                 unproject to view space using $MVP_{v,i}$
    $StartCoords_{r,v}$ := $Coords_{r,v} + (Disparity_l, 0, 0)$
    $EndCoords_{r,v}$ := $Coords_{r,v} - (Disparity_r, 0, 0)$

    // Loop over geometry splits and emit corresponding vertices
    $GeometrySplits$ := $[0, 1]$
    $Vertices$ := $[\,]$
    $VertexCount$ := 0
    for $Index$ := 0 to length($GeometrySplits$):
        $\alpha$ := $GeometrySplits[Index]$
        $Coords_{l,v}$ := $(1 - \alpha) * StartCoords_{l,v} + \alpha * EndCoords_{l,v}$
        $Coords_{r,v}$ := $(1 - \alpha) * StartCoords_{r,v} + \alpha * EndCoords_{r,v}$

        $ModelMatrix$ := $I$
        $ViewMatrix$ := Get view matrix from $MVP_l$ and modify such that position
                        corresponds with
                        $CameraPosition_l + \alpha * (CameraPosition_r - CameraPosition_l)$
        $ProjectionMatrix$ := Get projection matrix from $MVP_l$
        $MVP$ := $ProjectionMatrix * ViewMatrix * ModelMatrix$

        $Vertices[VertexCount]$ := EmitVertex($Coords_{l,v}$, $MVP$, $\alpha$, $n$, $c_{l,d}$)
        $VertexCount$ := $VertexCount + 1$
        $Vertices[VertexCount]$ := EmitVertex($Coords_{r,v}$, $MVP$, $\alpha$, $n$, $c_{r,d}$)
        $VertexCount$ := $VertexCount + 1$
    end for

    // Loop over vertices to assemble triangles
    for $TriWindowStart$ := 0 to $VertexCount - 3$ step 2:
        $EpipolarTriangles$ := $EpipolarTriangles \cup$
 $\{ (Vertices[TriWindowStart], Vertices[TriWindowStart + 1], Vertices[TriWindowStart + 2]) \}$
        $EpipolarTriangles$ := $EpipolarTriangles \cup \{ (Vertices[TriWindowStart +$
 $1], Vertices[TriWindowStart + 3], Vertices[TriWindowStart + 2] \}$
    end for
end foreach

$Fragments$ := RasterizeGeometry($I$, $EpipolarTriangles$)
return $Fragments$
```

Code listing B.4.2. *DrawDepthLayer transforms the geometry, gathered by the previously described view acquisition stages, into epipolar geometry.*

DrawDepthLayer, shown in Code listing B.4.2, starts with the computation of three positions from the projection parameters given as input to the function. The actual implementation of the algorithm does this by computing the inverse of the view matrix, followed by reading the three elements from the inverse matrix that make up the x, y and z-coordinate of the position. Computing the inverse of a view matrix takes $O(n^3)$ operations using Gauss-Jordan elimination[4], where $n = 4$ in this specific case since it represents the dimensions of the matrix. All aspects considered, computing the three positions, including reading the elements from the matrix and assigning them to the variables, has a time complexity of $3 * (O(4^3) + 3 + 3) = 3 * (O(64) + 6) \leq O(3 * O(64) + 18) = O(1)$. Calculating MaxViewOffset is done by a 3-dimensional vector subtraction, dot product and square root. The vector subtraction takes 3 basic operations; the dot product consists of 3 multiplications and 2 additions; the sqrt function takes $O(1)$ operations[5]. As such, calculating MaxViewOffset and ViewDistance is done using $2 * \left(3 + 3 + 2 + O(1)\right) + 1 + 2 = 2 * O(1) + 19 \leq O(2 * O(1) + 19) = O(1)$. Lines 8 through 11 consist of three assignments, two multiplications and one subtraction. The time complexity for this is 6. The foreach-loop executes $O(p)$ times since it executes once for tuple of ActivePixels, which was previously determined to hold $O(p)$ elements. The loop body starts with calculating two pairs of StartCoords and EndCoords using six assignments, six additions and six subtractions. The lefthand-side of the additions and subtractions are dependent on a pair of Coords, these are calculated using a unprojection. Identical to previous unprojections, this takes 28 basic operations. Furthermore, $Coords_{r,v}$ needs to read the values corresponding with $x + Stride$. This is implemented as textureFetch, and thus equates to reading a value from memory. Similarly, for fetching the color for $c_{r,d}$. As such, the time complexity for lines 15 through 21 is $6 + 6 + 6 + 28 + 28 + 1 + 1 = 76$. Furthermore, lines 24 through 26 consists of four assignments. The first inner for-loop, from lines 28 until 42 in the Code listing, starts with reading and assigning a value. Then, the computation of the pair of Coords. Until this point, the time complexity is $2 + 2 + 4 + 4 = 12$, for two assignments, four additions or subtractions and four multiplications. For generating the ModelMatrix and ProjectionMatrix, 32 assignments of scalar values are used. Computing the ViewMatrix means copying 16 scalar values and calculating the new position, which is computed using three additions, three multiplications and three subtractions. As such, ViewMatrix is generated using $16 + 3 + 3 + 3 = 25$ basic operations. Composing ModelMatrix, ViewMatrix and ProjectionMatrix into MVP is done using one assignment and two matrix-matrix multiplications. The time complexity for a single matrix-matrix multiplication is $O(n^3)$, where n can be substituted by 4 for this specific use case since n is the dimension of a square matrix. Lastly, adding the vertices to the array uses four assignments, two additions and two calls to *EmitVertex*. The time complexity hereof totals to $4 + 2 + 2 * O(1) = 2 * O(1) + 8 \leq O(2 * O(1) + 8) = O(1)$. The final detail for the time complexity of the for-loop spanning lines 28 through 42 is that its body is executed twice; once for each element in the array EpipolarSplits. As such, the total time complexity for emitting the vertices that describe the epipolar geometry corresponding to a single pixel is $2 * \left(12 + 32 + 25 + 1 + 2 * O(4^3) + O(1)\right) = 2 * O(64) + 2 * O(1) + 140 \leq O(2 * O(64) + 2 * O(1) + 14) = O(1)$. The loop-body for assembling the triangles in epipolar space uses two assignments, two disjoint-set union and reads six values from the Vertices array. The loop-body, spanning lines 45 through 48, is executed once because the array named Vertices only contains four vertices after it is filled. The combined time complexity of this for-loop is $2 + 2 + 6 = 10$.

[4] For more information on matrix inversion using Gauss-Jordan elimination, see https://en.wikipedia.org/wiki/Gaussian_elimination#Finding_the_inverse_of_a_matrix

[5] For completeness, computing the square root using Newton-Rhapson takes $\log_2(N)$ operations, where N is the number of bits. Since N is a constant with value 32, the time complexity of the function sqrt is $O(\log_2(32)) = O(5) = O(1)$. For more information on Newton-Rhapson, see https://en.wikipedia.org/wiki/Newton%27s_method

With each aspect of the foreach-loop analyzed, its total time complexity amounts to $O(1) + 6 + 76 + 4 + O(1) + 10 = 2 * O(1) + 96 \leq O(2 * O(1) + 96) = O(1)$. The function call to *RasterizeGeometry*, including the assignment to Fragments, has a time complexity of $O(v)$, as determined in Appendix B.1. In this case, v refers to the number of triangles generated by the foreach-loop. It was determined earlier that the loop runs $O(p)$ times. Furthermore, it outputs two triangles during each run. As such, the call to *RasterizeGeometry* has a time complexity of $2 * O(p)$. Lastly, the return statement on line 52 needs to be accounted for. The time complexity for the complete *DrawDepthLayer* function is analyzed to be $O(p)$. See equation B.4.3 for the final derivation.

$$O(1) + O(1) + 6 + O(1) + 2 * O(p) + 1 =$$
$$2 * O(p) + 3 * O(1) + 7 \leq$$
$$O(2 * O(p) + 3 * O(1) + 7) =$$
$$O(p) \tag{B.4.3}$$

Input: A set of n source views V and corresponding projection parameters MVP_v for each of the source views, indexed such that V_i is the i^{th} source view, and $MVP_{v,i}$ are its corresponding projection parameters. A single source view is either a depth layer, in case of center view acquisition and dual view acquisition, or an actual view, in case of hierarchical view acquisition. Furthermore, a pair of camera projection parameters MVP_l, corresponding with the leftmost camera, and MVP_r corresponding with the rightmost camera, are given as input.
Output: A partially ordered set $EpipolarVolume$ containing tuples of the form (x, y, z, d, c). This gives the color c at location (x, y, z) with depth value d. The pixels of a single view view can be retrieved by choosing a fixed z-coordinate and varying the x and y. $z = 0$ gives the view corresponding with the leftmost view and $z = k$ gives the rightmost view, where k is at most the number of views.

AcquireInterpolatedViews(Source views V, Projection parameters MVP_v; MVP_l; MVP_r, n):

```
$EpipolarVolume$ := { ∅ }
for $i = 1$ to $n$:
    $ActivePixels$ := CaptureActivePixels($v_i$)
    $ViewFragments$ := DrawDepthLayer($ActivePixels$, $MVP_{v,i}$, $MVP_l$, $MVP_r$, $n$)

    foreach tuple $(x, y, z, d, c) \in ViewFragments$:
        if $EpipolarVolume$ contains $(x, y, z, d_{ev}, \_)$ such that $d < d_{ev}$:
            $EpipolarVolume$ := $EpipolarVolume \setminus \{ (x, y, z, d_{ev}, \_) \}$
            $EpipolarVolume$ := $EpipolarVolume \cup \{ (x, y, z, d, c) \}$
        end if
    end foreach
end for

return $EpipolarVolume$
```

Code listing B.4.3. *Mathematical description of the entry point to the view interpolation algorithm.*

AcquireInterpolatedViews begins with an assignment to EpipolarVolume. Then, it contains a for-loop that runs once for each source view. For the depth peeling based source view acquisition pipelines the loop-body is executed at most twenty times. The number of executions with a hierarchical view acquisition is undeterminable in general. The loop-body is made up of two assignments, one function call to *CaptureActivePixels* and one function call to *DrawDepthLayer*. The time complexity of this section is $2 + O(p) + O(p)$. Furthermore, the for-loop encloses a foreach-loop. The body of the foreach-loop has a time complexity of $2 + 1 + 1 + O(1) = O(1) + 4$, since it is made up of two assignments, one disjoint-set complement, one disjoint-set union and one check that can be done in $O(1)$ operations. The foreach-loop is executed $O(\mathrm{n} * \mathrm{p})$ times because the triangles generated by *DrawDepthLayer* lie in the epipolar plane. The first dimension of each epipolar plane is the width of a view, in other words $O(p)$ pixels; the second dimension of each epipolar plane is the number of views, or n. Lastly, *AcquireInterpolatedViews* makes use of a return statement. As such, the aggregated time complexity is $O(n^2p)$ in the general case. See equation B.4.4 for the derivation of the time complexity from the individual parts. However, for the depth peeling based view acquisition pipelines, the time complexity can be simplified to $O(p)$ because the number of source views is a constant, as explained in Appendix B.2.

$$\begin{gathered}
1 + \mathrm{n} * \big(2 + \mathrm{O}(\mathrm{p}) + \mathrm{O}(\mathrm{p}) + \mathrm{O}(\mathrm{n} * \mathrm{p}) * (\mathrm{O}(1) + 4)\big) = \\
1 + 2 * n + O(2 * n * p) + 5 * O(n^2 * p) \leq \\
O\big(1 + 2 * n + O(2 * n * p) + 5 * O(n^2 * p)\big) = \\
O(n^2p) \qquad (B.4.4)
\end{gathered}$$

Appendix C Derivation of memory usage

Besides performance, memory usage is an important aspect for the adoption of an algorithm in practical use cases. The memory usage cost for the different pipelines is reported as a function to compute the bytes per pixel. The bytes per pixel is the ratio between the total memory usage and the dimensions of a view multiplied by the number of views. This provides a method that is image resolution independent, and only depends on the number of views that are expected as output of the different pipelines.

To get the memory usage of the different algorithms, the best option would be measure the memory cost in actual use cases. However, this is complicated due to OpenGL not providing APIs to query the memory usage of individual resources. Furthermore, the hardware and accompanying drivers may skew measurements because resources may be duplicated to promote pipelining and avoiding stalls. Besides, resources may have a different memory representation than was requested with OpenGL because of hardware compatibility. This would make the memory usage specific to a combination of hardware and drivers. To circumvent these problems, this appendix derives the theoretical memory usage.

The memory usage cost functions are derived from the OpenGL data types that the resources have in the implementation which was used to measure the reported performance in Section 4.4. The numerator of the derived memory cost functions is the sum of the used memory by the individual resources. So, for completeness, each section also provides a list of the used resources and its corresponding data type.

Appendix C.1 Deferred rendering

The deferred rendering pipeline, as used for testing purposes, makes use of the resources as listed in Table C.1. Each of the resources is a texture, with *composed views* being a 3d texture. The remaining resources are a 2d texture. All the textures, except for *"Composed views"* and *"SSAO Noise"* have the same dimension as the viewport. *"Composed views"* has 2 dimensions equal to the viewport size and the third dimension is the number of views. Lastly, *"SSAO Noise"* is a 4 by 4 texture.

Table C.1. *List of resources and corresponding OpenGL data types used in the baseline deferred rendering pipeline implementation.*

Categorization	Description	OpenGL data type
	Composed views	GL_RGBA8
Geometry-Buffer	Albedo	GL_RGB8
	Depth	GL_DEPTH_COMPONENT32F
	Normals	GL_RGB32F
	Specular Color	GL_RGB8
	Specular Intensity	GL_R32F
Screen Space Ambient Occlusion	SSAO	GL_R32F
	SSAO Blur	GL_R32F
	SSAO Noise	GL_RGB16F

The list of used resources weighted by their respective dimensions leads to a memory cost function as shown in equation C.1, where w and h are, respectively, the width and height of the viewport used for rendering, and v is the number of views expected as output.

$$f(v) = \frac{total\ memory\ of\ resources}{dimenions\ of\ view * number\ of\ views}$$

$$= \frac{w * h * (v * 4 + 3 + 4 + 12 + 3 + 4 + 4 + 4) + 4 * 4 * 6}{w * h * v}$$

$$= \frac{w * h * (4 * v + 34) + 96}{w * h * v}$$

$$= \frac{w * h * (4 * v + 34)}{w * h * v} + \frac{96}{w * h * v}$$

$$\approx_{w*h*v \gg 96} \frac{4 * v + 34}{v}$$

$$= 4 + \frac{34}{v} \qquad (C.1)$$

Appendix C.2 Center view acquisition and epipolar view interpolation

The combined memory usage of the center view acquisition and epipolar view interpolation pipelines is taken up by the resources listed in Table C.2. Each resource with "*Depth layers*" as part of their description is an array of 10 Texture 2ds. Herein, each depth layer has dimensions equal to the viewport. "*Albedo*", "*SSAO*" and "*SSAO blur*" are viewport sized texture 2ds. "*SSAO Noise*" is a 4-by-4 Texture 2d. Furthermore, respectively "*Epipolar planes (Color)*" and "*Epipolar planes (Depth)*" are a Texture 3d and a Texture 2d Array with 2 dimensions equal to the dimensions of the viewport and the last dimension equal to the number of views expected as output. Lastly, "*Primitive buffer*" contains one instance of a custom struct for each pixel in each depth layer. In other words, the number of instances it contains is ten times the width multiplied by the height of the viewport. The custom struct is composed of 2 32-bit floats and 3 32-bit integers, which totals to 20 bytes.

Table C.2. *List of resources and corresponding OpenGL data types used in the center view acquisition pipeline combined with the epipolar view interpolation pipeline implementation.*

Categorization	Description	OpenGL data type
	Depth layers (Composed)	GL_RGBA8
Geometry-Buffer	Albedo	GL_RGB8
	Depth layers (Depth)	GL_DEPTH_COMPONENT32F
	Depth layers (Normals)	GL_RGB32F
	Depth layers (Specular Color)	GL_RGB8
	Depth layers (Specular intensity)	GL_R32F
Screen Space Ambient Occlusion	SSAO	GL_R32F
	SSAO Blur	GL_R32F
	SSAO Noise	GL_RGB16F
Epipolar view interpolation	Epipolar planes (Color)	GL_RGBA8
	Epipolar planes (Depth)	GL_DEPTH_COMPONENT24
	Primitive buffer	Custom struct (20 bytes)

The list of used resources weighted by their respective dimensions leads to a memory cost function as shown in equation C.2, where w and h are, respectively, the width and height of the viewport used for rendering, and v is the number of views expected as output.

$$g(v) = \frac{total\ memory\ of\ resources}{dimenions\ of\ view * number\ of\ views}$$

$$= \frac{w * h * (10 * 4 + 3 + 10 * 4 + 10 * 12 + 10 * 3 + 10 * 4 + 4 + 4 + 10 * 20) + 4 * 4 * 6}{w * h * v} + \frac{w * h * (v * 4 + v * 3)}{w * h * v}$$

$$= \frac{w * h * (481 + 7 * v) + 96}{w * h * v}$$

$$= \frac{w * h * (7 * v + 481)}{w * h * v} + \frac{96}{w * h * v}$$

$$\approx_{w*h*v \gg 96} \frac{7 * v + 481}{v}$$

$$= 7 + \frac{481}{v} \qquad (C.2)$$

Appendix C.3 Dual view acquisition and epipolar view interpolation

Table C.3 has listed the used resources by the dual view acquisition and epipolar view interpolation pipelines. The resources in these pipelines are of the same type and have the same data types. The difference being that each resource description with "*Depth layers*" in their name has doubled. In other words, ten depth layers are used for rendering the leftmost view and the remaining ten depth layers are used for rendering the rightmost view. This has been accounted for in equation C.3.

Table C.3. *List of resources and corresponding OpenGL data types used in the dual view acquisition pipeline combined with the epipolar view interpolation pipeline implementation.*

Categorization	Description	OpenGL data type
	Depth layers (Composed)	GL_RGBA8
Geometry-Buffer	Albedo	GL_RGB8
	Depth layers (Depth)	GL_DEPTH_COMPONENT32F
	Depth layers (Normals)	GL_RGB32F
	Depth layers (Specular Color)	GL_RGB8
	Depth layers (Specular intensity)	GL_R32F
Screen Space Ambient Occlusion	SSAO	GL_R32F
	SSAO Blur	GL_R32F
	SSAO Noise	GL_RGB16F
Epipolar view interpolation	Epipolar planes (Color)	GL_RGBA8
	Epipolar planes (Depth)	GL_DEPTH_COMPONENT24
	Primitive buffer	Custom struct (20 bytes)

The list of used resources weighted by their respective dimensions leads to a memory cost function as shown in equation C.3, where w and h are, respectively, the width and height of the viewport used for rendering, and v is the number of views expected as output.

$$h(v) = \frac{total\ memory\ of\ resources}{dimenions\ of\ view * number\ of\ views}$$

$$= \frac{w * h * (10 * 2 * 4 + 3 + 10 * 2 * 4 + 10 * 2 * 12 + 10 * 2 * 3 + 10 * 2 * 4 + 4 + 4 + 2 * 10 * 20) + 4 * 4 * 6}{w * h * v} + \frac{w * h * (v * 4 + v * 3)}{w * h * v}$$

$$= \frac{w * h * (951 + 7 * v) + 96}{w * h * v}$$

$$= \frac{w * h * (7 * v + 951)}{w * h * v} + \frac{96}{w * h * v}$$

$$\approx^{w*h*v \gg 96} \frac{7 * v + 951}{v}$$

$$= 7 + \frac{951}{v} \tag{C.3}$$