University of Stuttgart Visualisation Research Centre

University of Stuttgart
Allmandring 19
D–70569 Stuttgart

Bachelorarbeit Nr. 170

Adaptive Frameless Raycasting for Interactive Volume Visualization

Constantin Weißer

Course of Study:	Informatik
Examiner:	Prof. Dr.rer.nat. Dr.techn.hc. Dr.-Ing.E.h. Thomas Ertl
Supervisor:	Steffen Frey
Commenced:	September 15, 2014
Completed:	October 31, 2014
CR-Classification:	I.3.3

Abstract

There have been many successful attempts to improve ray casting and ray tracing performance in the last decades. Many of these improvements form important steps towards high-performance interactive visualisation. However, growing challenges keep pace with enhancements: display resolutions skyrocket with modern technology and applications become more and more sophisticated. With the limits of Moore's law moving into sight, there have been many considerations about speeding up well-known algorithms, including a plenitude of publications on frameless rendering.

In frameless renderers sampling is not synchronised with display refreshes. That allows for both spatially and temporally varying sample rates. One basic approach simply randomises samples entirely. This increases liveliness and reduces input delay, but also leads to distorted and blurred images during movements. Dayal et al. tackle this problem by focusing samples on complex regions and by applying approximating filters to reconstruct an image from incoherent buffer content. Their frameless ray tracer vastly reduces latency and yet produces outstanding image quality. In this thesis we transfer the concepts to volume ray casting. Volume data often poses different challenges due to its lack of plains and surfaces, and its fine granularity. We experiment with both Dayal's sampling and reconstruction techniques and examine their applicability on volume data. In particular, we examine whether their adaptive sampler performs as well on volume data and which adaptions might be necessary.

Further, we develop another reconstruction filter which is designed to remove artefacts that frequently occur in our frameless renderer. Instead of assuming certain properties due to local sampling rates and colour gradients, our filter detects artefacts by their age signature in the buffer. Our filter seems to be more targeted and yet requires only constant time per pixel.

Kurzfassung

In den letzten Jahrzehnten gab es zahlreiche Versuche, die Effizienz von Ray-Casting und Ray-Tracing zu verbessern. Viele dieser Verbesserungen bilden wichtige Schritte hin zu leistungsstarken, interaktiven Visualisierungen. Mit der Performanz steigen aber auch die Herausforderungen: die technisch möglichen Bildschirmauflösungen liegen um ein vieles höher und Anwendungen stellen immer größere Anforderungen an die Software. Da die Hardware langsam an die Grenzen von Moores Gesetz stößt, liegt der wissenschaftliche Fokus immer deutlicher auf der Verbesserung der Algorithmen, zum Beispiel durch *frameless Rendering*.

Beim frameless Rendering ist das Sampling nicht mit dem Anzeigeprozess synchronisiert. Das bietet zusätzliche Freiheiten für Algorithmen: räumliche und zeitliche Abtastraten können so variieren. Ein grundlegender Ansatz randomisiert Samples mit einer Gleichverteilung. Das führt zu kleineren Eingabeverzögerungen und erhöht die Lebhaftigkeit der Visualisierung. Gleichermaßen werden aber Bilder durch Bewegungen verzerrt. Dayal et al. bewältigen dieses Problem durch zielgerichtetes Sampling (guided sampling). Dabei werden hohe Abtastraten auf komplexe Bildregionen fokussiert und in einfachen Bildregionen Rechenzeit eingespart. Außerdem werden Bildraumfilter verwendet, um aus den inkohärenten Daten ein möglichst wahrheitsgetreues Bild zu approximieren. Der frameless Ray-Tracer von Dayal et al. bietet stark reduzierte Latenz bei hervorragender Bildqualität.

In dieser Arbeit übertragen wir die Konzepte auf Ray-Casting von Volumendaten. Volumendaten bieten oft andere Herausforderungen, da sie keinerlei Oberflächen aufweisen und oft sehr feingranulär sind. Wir experimentieren mit Dayals Sampling- und Rekonstruktionsmethoden und untersuchen deren Eignung für Volumendaten. Insbesondere untersuchen wir, ob deren adaptiver Sampler Volumendaten ebenso gut verarbeiten kann und welche Anpassungen eventuell nötig sind. Des Weiteren entwickeln wir einen eigenen Rekonstruktionsfilter, welcher speziell auf häufige Bildartefakte beim Rendern von Volumendaten ausgelegt ist. Anstatt, wie Dayal, den Filter an die lokale Abtastrate und Farbgradienten anzupassen, werden durch unseren Filter Artefakte anhand ihrer Alterssignatur erkannt. Dabei scheint unser Ansatz zielgerichteter und benötigt dennoch nur konstante Laufzeit pro Pixel.

Contents

1 Introduction 11
- 1.1 Motivation . . . 11
- 1.2 Related work . . . 14
- 1.3 Overview and contributions . . . 16

2 Technical background 19
- 2.1 Ray casting . . . 19
- 2.2 k-d trees . . . 23
- 2.3 Buffering . . . 25
- 2.4 Image space properties . . . 25
- 2.5 CUDA . . . 26

3 Adaptive sampler 29
- 3.1 Sampling . . . 30
- 3.2 Means for guided sampling . . . 32
- 3.3 Improvements . . . 38

4 Reconstruction 41
- 4.1 Reconstruction filters . . . 42
- 4.2 Age filter . . . 45
- 4.3 Subsequent filtering . . . 47

5 Evaluation 49
- 5.1 Timing . . . 49
- 5.2 Adaptive sampling . . . 56
- 5.3 Reconstruction . . . 60

6 Summary and future work 77

Bibliography 79

List of Figures

2.1 Left: object with object coordinates in global world space in front of an image plane with depicted FOV. Right: Schematic of ray casting with one ray per pixel. One exemplary ray hits the volume. 19
2.2 Example of a 2-d tree, dividing an area into tiles. The numbers indicate levels in the tree, the letters identify leaf nodes. 24

3.1 The rendering loop sketched in pseudo code. 30
3.2 Example for failed merge condition for leaf node a. While a appears in the leaf array, b is not. Its children prevent b from being split. 36

4.1 White pixels are up-to-date, coloured pixels are out-dated. Left: single spots, i.e. pixels with eight newer neighbours. Right: cluster spots with less than eight newer neighbours. Our less strict version of the age filter removes small clusters of outdated samples or weakens their intensity. 46
4.2 A vast amount of "litter pixels" as it appears when zooming out quickly. To the right: exact same situation with activated hit filter. The stale samples nearly disappear. 48

5.1 Distribution of time over the different rendering phases. For big volume data sets the phases for tile adaption and reconstruction become more and more insignificant. The leaf update phase and sorting phases appear to be weak spots of our implementation while parallel tree processing generally works well. . . . 50
5.2 Raw buffer at different sampling rates showing a shifting movement. Around 50% the sampler performs well and distortions are vastly reduced. Depending on the filters the ratio may even drop below 50%. Ratios over 75% seem superfluous. The difference to 100% is not significant. Furthermore, the sampler's performance suffers from ratios close to 100% and likely performs worse than a framed sampler. 65
5.3 Filtered with age filter. The difference between 75% and 100% is almost not notable, proving that sampling ratio should not be too high in adaptive samplers. Furthermore, by filtering, the quality of a 50% ratio is raised to the same magnitude as 75%. 25% however seems generally too low for volume data. Low sampling ratios increase both stale samples and delay for tiling adaptions. 66

5.4 Small tiles covering both dense and permeable parts of the volume. Even if warps sample spatially close samples, threads may still idle because of local differences. Guided sampling attracts many rays to these exact locations. 67
5.5 Undersampled fringe due to late k-d tree adaption. The effect can be mitigated by better tile assessment and our age filter. However, to eliminate this effect, the tiling must be adapted predictively. 68
5.6 The volume is shifted to the right. Many stale samples are visible between the bigger drops of the volume and as dark spots on the drops as well. Our age filter removes many of the spots in between or at least conceals (darkens) them. The dark spots on the drops are almost entirely removed. Only few remain on the fringe. 69
5.7 Rotation around the y-axis. The filter improves the image quality considerably. Again almost all dark spots are removed. The filter also performs well with bright spots in between. Due to its targeted functioning it only adds about 3ms overhead to the reconstruction phase. 70
5.8 A rotation around y-axis. The filter cleans the inner parts of the volume but fails in the outer regions. Image 5.8d shows the k-d tree in this very moment. The tiling indicates low sampling rates in the severely spotty areas. 71
5.9 From top to bottom: movement of the volume to the upper right corner. Left: Dayal's filter, right: age filter. Dayal's filter leaves considerable amounts of stale samples on the bottom left which the age filter successfully removes. Furthermore, the inner regions look much cleaner when age filtered. 72
5.10 Effect of different filters on a quickly moving scene. Dayal's and the dynamic Gaussian filter both rely on per-tile sampling rates. But the measure adapts too late in order to guide filtering reasonably. The static Gaussian eliminates artefacts but also blurs the image permanently. Our age filter removes many artefacts while maintaining a focused image. 73
5.11 Exact same portion of the image plane on a static scene at three different moments. Images 5.11d and 5.11e show the difference between the images above (visually enhanced, i.e. adjusted brightness and saturation). The changing subpixel locations cause considerable temporal differences. Image 5.11f shows the temporal gradient of this scene. Even though the scene is static, the gradient is very high at the edges. Due to weighting, this causes ever changing colours of each pixel. 74
5.12 Both images show the scene after suddenly zooming out. The right image is filtered with our age filter. It is clearly visible that the age filter is to the hit filter, as most stale samples in empty areas are quickly concealed. 75

List of Tables

5.1 Distribution of sampling time and standard deviation over the different phases tested on different data sets. First line shows the average values, the second standard deviation. All times are in ms. 50
5.2 Timing analysis for rotating scene on different data sets. All times in ms. Sampling times slightly differ and vary more due to dynamic scene. Reconstruction and tree adaptions are not affected by the rotation. 52
5.3 Timing analysis for shifting motion on different data sets. All times in ms. Sampling times are lower, due to delay of k-d tree adaptions. Reconstruction takes slightly longer because of an unbalanced k-d tree. Smaller leaf nodes lower time for leaf update. Remaining phases perform at constant time as usual. 52
5.4 Effect of amount of samples on sampling time. The correlation is about linear. Duration for a 100% ratio is higher than for framed rendering, because the sampler samples challenging parts more densely. 54
5.5 Effect of sampling distributions on sampling time. In brackets the standard deviation per round. All times in ms. Assigning spatially close samples to one warp increases the performance vastly. In addition, the durations vary less. . . . 54
5.6 Computation time for different filters on static scenes. All times in ms. The slight differences between data sets are due to properties of the k-d tree and occur identically with none filters applied (see previous chapter). Dayal's filter performs fast as on a static scene the filter box is chosen quire small. 55
5.7 Computation time for different filters on dynamic scenes (rotation). All times in ms. Static Gaussian filter and the age filter are not affected by the motion. The adaptive filter of Dayal et al. however skyrockets to a fourfold of the static scene. 56
5.8 Number of iterations until tree is adapted when performing a "sudden shift" and a "gradual shift". The results are generally not conclusive enough to claim an advantage over regular variance. There might be a slight mitigation. 58

List of Listings

List of Algorithms

1 Introduction

1.1 Motivation

1.1.1 Research in interactive volume visualisation

Volume visualisation has a plenitude of practical applications, many of which become more utilisable when performed in real-time. Common examples are medical imaging such as the visualisation of CT or MRI data, as well as data collected by particle accelerators, molecular structures or physical experiments. Simulations often create volume data sets as well.

While some of these applications do not require real-time performance (such as the rendering of movie scenes), some do or at least become more powerful due to responsiveness to changes. Especially for scientific use, interactive volume visualisation enables the user to examine data sets more easily. The basic idea of visualisations is, to display data in an alternative way. Visual input can be processed more easily by humans than a set of data shown as an array of numbers for example. Scientists may find patterns or new information, they were not even aware of before. To examine a data set, the user might change parameters of the visualisation frequently, until the generated image exhibits the wanted properties. These parameters may include point of view (i.e. position of the virtual camera, this may also include zooming), the field of view (FOV) and changes to the transfer function, which translates a data point of the volume to a colour. In the past, visualising huge volume data sets with acceptable frame rates has been a big challenge. Even today, using up-to-date graphics hardware, the performance of ray casters lag behind conventional polygonal rendering algorithms. There have been many efforts in research to speed up ray casting (and ray tracing as well), for example by massive parallelisation through distributed networks [IBH11]. The proposed improvements often make ray casting more usable, but for for example in [IBH11], this also implies enormous costs for running the software on big computer clusters. The ultimate goal is to increase performance without a vast increase in costs.

1.1.2 Ray casting

In conventional computer graphics, polygonal models are the most common type of data sets to be visualised. Thus, the hardware acceleration through GPUs mostly supports the rendering pipeline as needed for polygonal models. This technique is known as rasterisation. In the past

rasterisation rendering has proven to be powerful and still is the method of choice today. Very briefly summarised, rasterisation renderers work as follows: For the calculation of frame, the renderer iterates over all polygons, projects each to the image plane and subsequently colours the pixels of the image plane accordingly with help of a scanning algorithm. Scanning detects which pixels are "affected" by a polygon. Modern displays with 4K resolutions roughly have nine million pixels. Thus iterating over all polygons seems to be the better choice. A model with several million polygons is considered very big for common purposes. In volume data sets on the other hand there are no polygons nor similar structures within the data set. The model to be visualised is a (mostly uniform, three dimensional) scalar field. A data point in such a grid is called a *voxel*. To use the conventional rendering technique, the renderer would need to project each and every voxel to the image plane. Data sets of $1000^3 = 1,000,000,000$ voxels are common (and even bigger than that). Due to the vast amount of data points even in normal-sized data sets, the "conventional" technique performs poorly.

High-quality volume rendering is usually implemented using the ray casting technique, an image-order rendering algorithm. The renderer produces a ray for each pixel, originating from the virtual point of view through the pixel and subsequently steps through the space. At each step that is located within the volume, the information is sampled and included into the calculation of the pixel's colour. Being that simple and yet accurate makes the algorithm a very convincing approach. Furthermore, the algorithm's performance only weakly depends on the scene complexity. It only increases logarithmically with scene size [WPS+03]. Ray tracing, an extension to the ray casting algorithm, is often used to render photo-realistic scenes with complex lighting and reflection circumstances [WPS+03].

Unfortunately, ray tracing has long been considered a slow algorithm, applicable only for offline rendering. The lack of "natural" hardware support has made it difficult to create high-performance ray casting renderers. However, the vast increase in hardware performance plus the development of "General-purpose computing on graphics processing units (GPGPU)" in the last decade provide a powerful tool set to develop efficient real-time ray casters and ray tracers utilising parallelisation techniques on the GPU. However, rendering large data sets with common viewport resolutions remains a challenge and further research is necessary.

A detailed discussion of ray casting can be found in chapter 2.1.2.

1.1.3 Frameless rendering

Conventional rendering for both object-order and image-order techniques usually is "framed". Therefore the renderer keeps two framebuffers of equal size in memory, sometimes called "front" and "back buffer". While the *front buffer* is used to feed the graphics output with the information to display the image, the render algorithm fills the *back buffer* with data for the next frame to display. For each frame, the colour (in theoretical terms the "luminance") of every pixel is calculated. With modern GPU support there is a great variety of parallelisations

that can improve the performance. Once the entire frame is calculated, *front* and *back buffer* are swapped making the previous *back buffer* the new *front buffer*.

Object-order rendering projects each polygon to the image plane, independent of which pixels it might cover. In image-order techniques the renderer "chooses" explicitly to calculate every pixel for each frame. This seems to be the simplest approach that results in coherent frames. However, a lot of computation time might be wasted for several reasons. Examples include:

- the area might be "empty". That means no object (or part of a volume) covers this part of the field of vision.
- pixels might be calculated too thoroughly. This happens, when following a ray further does not change its colour anymore.
- the considered data is too detailed, e.g. distant parts of volume require a lesser resolution than nearby areas.

For the named examples (and others) there are improvements to the naïve algorithm which usually exploit additional knowledge about the data. Note that all of the examples are spatial properties. Yet, all of the enhancements keep the concept of fixed frames intact. The displayed image on screen is a coherent temporal slice.

The concept of *frameless* rendering loosens up this restriction. Samples are no longer part of temporal slices but located anywhere in space-time. This allows a variety of new improvements to the basic ray casting algorithm. The renderer has the ability to concentrate sampling on certain areas of the image and reducing the sample frequency in other areas. By analysing the image and detecting certain properties, the renderer must decide which areas need higher sampling rates. Such areas cover temporally changing objects, caused by movement, rotation or a zooming motion. Also areas containing "visual edges" (sharp change in colour) require more detailed sampling, in order to avoid glitches such as aliasing.

The new degree of freedom can be exploited to distribute the calculation time as needed rather than uniformly across the image. That way, if implemented correctly, the same amount of operations on the GPU can provide better results and higher liveliness compared to conventional sampling. While the advantages seem tempting, they come at a price. The goal of this thesis is to explore the possibilities and evaluate different approaches for the following issues:

- focused sampling, that is, locally increase sampling rates in some areas of the image while lowering it in others.
- providing data to the sampler enabling it to choose "wisely". That is, information gained from the current view and recent samples must be guided back into the sampler and ways to detect interesting regions must be explored.
- reconstruct an image as close to the "perfect image" as possible. Due to the degree of freedom in time, the latest samples do not constitute a coherent image. A "real" image has to be approximated.

- implement the points above entirely on the GPU to exploit GPU's superior parallel compute power.

In particular, our work is based on the ideas and techniques of Dayal et al. [DWWL05]. While their approach is based on ray casting and thus optimised for polygonal models, we try to explore the same and other ideas for volume rendering.

1.2 Related work

1.2.1 Entirely random sampling

In the article *Frameless Rendering: Double Buffering Considered Harmful* [BFMZ94], Bishop et al. describe the basic idea of loosening frame restrictions. Instead of using conventional double-buffering as described above, they randomise the sampling entirely. That is, the next position to sample is chosen uniformly at random from all possible sampling positions. No additional data is included into the process of choosing a position. Each pixel of the output shows the most recent sample taken at this position. While conventional rendering jumps discretely from frame to frame, this technique provides fluid motions and a continuously updated image [BFMZ94]. When the user interacts with the software, e.g. "moves around", the program's response seems to be immediate. However, the resulting images exhibit samples with different ages. The picture does not show a moment in time but rather a period of time. When the scene is static, the renderer iteratively steps towards a consistent image. After a scene change the first images are "crude images" [BFMZ94]. They are calculated quickly but only a fraction of the pixels are up-to-date, the rest show stale samples from the scene prior to changes. The pixels to update are selected randomly to avoid effects like tearing and to ensure, that all areas of the image become updated more or less evenly. The crude image obviously is not accurate, but the accurate parts (i.e. the updated pixels) are more recent compared with a conventional renderer. The calculation of a frame requires more time, as more pixels are to be sampled. If the scene remains unchanged, the renderer chooses more and more stale pixels to update over time, resulting in a consistent image eventually. While in theory there is a chance that some pixels remain stale "forever", in practice this usually does not happen. If the scene keeps moving, there will be a constant motion blur because older and newer pixels mix.

For scientific use this might not even be harmful. When examining data, a scientist might adapt the visualisation parameters and then subsequently render a full scene. The crude images might serve as a quick preview. However, consider how much computation time is wasted for samples which are randomly selected twice before another scene change occurs. There is no mechanism that guides sampling smartly to regions that appear to be important within some taxonomy.

1.2.2 Crude images and subsequent *adaptive* refinement

[Lev90] describes a way to deterministicly calculate crude images as a speedup for conventional ray-tracing. They first reduce the sampling rate to less than one sample per pixel. These initial samples are arranged in a uniform grid. The "lack of data" is compensated by interpolating the available samples. In subsequent steps the interpolated colours are discarded and the image is refined by adding more samples, i.e. casting more rays.

To guide the refinement, they use "recursive subdivision based on colour differences" [Lev90]. Using colour differences, i.e. the colour gradient, in distinct areas of the image seems to be a useful yet simple way to guide sampling, or more general, to detect areas of interest. High spatial gradients may indicate an edge, and thus a higher amount of samples is required to provide a sufficient resolution. Image space properties are of rather simple nature and often neatly calculated in parallel.

1.2.3 Adaptive frameless rendering

In [DWWL05], Dayal et al. advocate their frameless renderer for polygonal models. Their renderer is divided into two major components: a sampler and a reconstructor.

Their sampler takes samples randomly across the image space. While displaying is synchronised, the samples are all located freely within space-time. Thus, their renderer allows for both variable spacial and temporal sampling rates. While the renderer in [BFMZ94] treats all sample locations equally, Dayal et al. improve this by adapting the probability distribution to the current scene. They advocate image tiling as a means to detect regions that require higher sampling densities. The colour variance across a tile is used for assessment. Tiles with high colour variation cover more complex parts of the image and thus likely need a higher sampling rate. Their tiling is updated continuously as new samples are taken and is able to adapt to ever changing scenes. Their renderer tries to maintain a tiling, such that each tile covers approximately the same amount of colour variation. Therefore, they utilise simple merge and split operations. That is, tiles with a high colour variance are split and neighbouring tiles with small variance are merged. The tiling is backed by a k-d tree and maintained within GPU memory.

With help of the tiling, their sampler is able to distribute samples adaptively across the image, making each tile equally probable. Hence, pixels in small tiles are more likely to be sampled than such in big tiles.

Sampling without frame restrictions leads to buffer content of older and newer samples being interleaved. Unlike in framed renderers, the buffer does not reflect a "real" snapshot of a certain moment in time. At every point in time, some pixels will be up-to-date, while others are obsolete. In practice, this leads to unpleasant effects such as blurring or spots (single obsolete pixels).

Dayal et al. tackle this problem by processing the samples in the buffer before displaying them. They call this process "image reconstruction". In [DWWL05] they claim to experiment with different filters and advocate a simple Gaussian for use in practice. Their filtering process is, however, more complex than simply applying a Gaussian blur. Their renderer keeps track of spatial and temporal colour gradients (or rather estimations of these) to make filtering adaptive. They also keep older samples in the buffer for each location, by making each pixel location a small (fix-sized) queue.

Their filter then is a dynamically resized three dimensional box, while the sizing depends on *local* sampling densities and gradient values. If temporal gradients are high, e.g. during bigger changes in the scene, the filter becomes spatially wide but temporally narrow [DWWL05]. If the scene is stable, the temporal gradients drop and the filter includes older samples as well. That way, they implement a locally dynamic anti-aliasing and achieve superior image quality while maintaining the advantages of a frameless renderer, in particular its responsiveness and low latency.

We build the work of this thesis upon their ideas. In particular, we adopt the idea of guided sampling and image reconstruction. We experiment if their techniques are feasible for volume data, too, as they develop and implement their renderer for polygonal models. We implement our own version of the guided sampler and describe its technical details in chapter 3. Furthermore, we implement their reconstruction algorithm as well as our own ideas (chapter 4) and evaluate the usability (chapter 5).

Walter, Drettakis, and Parker describe a renderer that produces partial frames with help of a cache and reprojection. They also use the cache to direct sampling to areas of interest so that the renderer improves image quality more quickly [WDP99]. Zagier describe the balance between responsiveness and coherence for frameless rendering. They specifically address artefacts that occur when frame restrictions are abandoned and displayed images are more or less coherent [Zag96].

1.3 Overview and contributions

In this thesis we cover three topics. First, we describe our implementation of guided sampling using a k-d tree. Dayal et al. describe a plenitude of methods in their paper [DWWL05]. However, they mostly state *what* they do and justify their ideas, but hardly describe *how* they achieve or implement it. Especially with massive parallel programming, one might face challenges in implementing complex algorithms. Furthermore, frameless sampling for volume data is rather unexplored and new challenges may arise from properties unique to volume data. We adopt the ideas from [DWWL05] and examine how those techniques may be implemented and what adaptions are necessary for volume data. We describe our GPU-based guided sampler using CUDA, including k-d tree management and collection of the necessary data to adapt the

tree. Our implementation is described in chapter 3. In this chapter, we also discuss sample distribution over threads, sampling for dynamic anti-aliasing and a feedback mechanism to the k-d tree.

Second, we cover image reconstruction for volume data. In [DWWL05] all techniques are developed for polygonal models. We experiment with reconstruction for volume data and describe our ideas in detail. Some of the techniques may perform well and improve image quality, while some might not be very helpful. Nonetheless we share our ideas hoping they may inspire someone to use and improve them. We also implement the adaptive reconstruction filter outlined in [DWWL05]. Once more, Dayal et al. developed their filter for polygonal data. We implement and test it on volume data and share our findings. Image reconstruction is covered by chapter 4.

Last, we evaluate our implementation of the guided sampler as well as the several reconstruction techniques we experiment with. We compare our approach to the one of Dayal et al.. We test especially how well reconstruction methods by Dayal et al. work on volume data. Our results are described in chapter 5. Further, within the last chapter, we share our ideas on how guided volume ray casting may be improved further in the future.

Our contributions to frameless rendering are:

- description of a frameless volume sampler with image space tiling for adaptive sampling, especially...
- ...implementation details for maximised parallelisation on GPUs. Our focus lies on parallel tree processing
- evaluation of the sampling technique's usability for volume data including considerations on tile assessment
- detailed analysis of our renderer's performance including identification of problematic phases that must be well-thought-out
- examination of the effect of sample amount and distribution on the runtime of guided samplers. We focus on particular issues due to adaptive guidance and volume data (instead of polygonal data)
- evaluation of Dayal's filtering and reconstruction algorithm on volume data
- further filter variants inspired by Dayal's idea
- our own new filter technique based on age signature of samples rather than local sampling rates. It targets specific artefacts that frequently cause image distortions. It works more precisely and can adapt to fine grained artefacts
- comparison to other filters and identification of specific weak spots that may be relevant for future research

2 Technical background

2.1 Ray casting

2.1.1 World space, object space, image plane

A renderer visualises an object by projecting it to an image plane. The image plane is then shown on screen. An object is usually modelled in its own coordinate system called *object coordinates* (which measures the *object space*). This is mostly relevant for polygonal models. Volumes on the other hand are usually uniform three-dimensional grids. The arrangement within the object space results naturally. Another space forms the set of possible locations for all objects. It is called *world space* (and accordingly *world coordinates*). The camera as well as all objects, in our case the volume, have absolute coordinates within the world space. The image plane is located (usually with a fixed distance) in front of the camera. The distance from the camera and the size of the plane also define the *field of view* (FOV). Figure 2.1 shows the image plane, an exemplary object, both coordinate systems, the virtual camera and the FOV

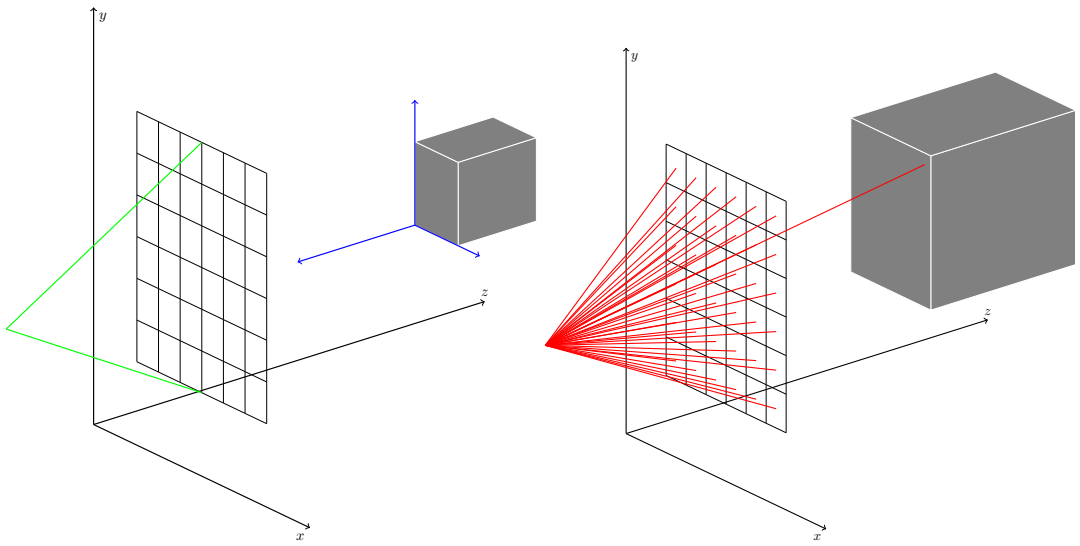


Figure 2.1: Left: object with object coordinates in global world space in front of an image plane with depicted FOV. Right: Schematic of ray casting with one ray per pixel. One exemplary ray hits the volume.

(in this case vertical FOV). In order to visualise the object from this perspective, a renderer has to calculate a projection of the object to the image plane, as seen from the virtual camera's point of view.

2.1.2 Basic algorithm

Ray casting is one way to calculate such a projection. The idea is somewhat natural and copied directly from physics. If we see an object, the light emitted or reflected by it hits our retina. Light travels along a straight line. To detect which objects are visible to the virtual camera, the renderer simply "walks" along such rays in the opposite way. A so-called *primary ray* originates from the virtual camera, passes through one of the cells of the pixel grid.

Let $v \in \mathbb{R}$ be the virtual camera's point in world space. Let $p \in \mathbb{R}$ be the sub-pixel location within the image plane, e.g. the pixel's centre. The ray's direction results in:

$d' = p - v$

In order to simplify following steps, the vector d is a normalised version of d':

$d = \frac{d'}{||d'||}$

In order to march along the ray, we have to start at a certain location. Usually, objects too close to the virtual camera are omitted, for example objects between the camera and the image plane are not shown. In general, any plane can be used to limit the viewing frustum. We call the plane limiting the viewing frustum close to the camera *near plane*. The plane limiting the visual range in the distance is called *far plane*. Hence, we can simply use the intersection of a ray with the near plane as starting point n for the march through space. Also see section 2.1.4 for a practical approach.

Whichever way the starting point n is determined, the first sample is taken at position:

$s(0) = n$

Subsequent steps through spaces are taken by marching along the ray, that is, marching in the direction of d. The step width w is an important parameter and has to be chosen carefully. The next step may be calculated recursively using the current sampling position $s(i)$ and step width w:

$s(i + 1) = s(i) + w \cdot d$

Figure 2.1 shows the "casting" of rays. Here, one ray through each of the pixels' centre is emitted. Higher numbers of rays per pixel at different subpixel-locations can be used for anti-aliasing. In Figure 2.1 one exemplary ray hits the volume, the others are not shown completely for clarity.

2.1.3 Sampling and transfer function

In general, volume data may exhibit arbitrary data formats. There are no restrictions as to what a volume represents. Consequently, the question arises how to calculate a colour from an arbitrary data type. This is achieved by a mapping called *transfer function*:

$$f_t : V \to L$$

V is the volume data's space, each data point is an element of V. L is the luminance space, which contains all colours[1]. In practice this means the transfer function takes in a single data point and returns a colour value associated with it.

When the primary ray passes through the volume, the renderer has to determine which voxels are hit by the ray. The hit voxels' values will be included into the calculation. Usually, a ray does not hit voxels "precisely" but passes by. To cope with this, either every voxel has to be associated with a surrounding volume, or the value at a given (arbitrary) position within the volume is an interpolation of the voxels nearby. For an arbitrary position $s \in \mathbb{R}^3$, let

$$v_i : \mathbb{R}^3 \to V$$

be the function, that maps this position to its interpolated value.
Adding the pieces together leads to the following mapping from step index i to the resulting colour $l(i)$ (do not confuse this with the rendering equation's luminance L):

$$l : \mathbb{N} \to \mathbb{R}^3 \to V \to L$$
$$l(i) = f_t(v_i(s(i)))$$

[1]The luminance in physical terms is simply a measure for the amount/intensity of light in a given direction. For visualisation matters and according to the usual technical representation of colour in computers, each value is a triple consisting of the intensity of red, green and blue

The taken samples are then accumulated step by step to one resulting colour. Obviously smaller steps lead to more accurate sampling but also require more computation time, because more steps are necessary to march through the same distance.

2.1.4 Common improvements

Frameless rendering as described in this thesis is meant to be a performance improvement to the basic ray casting algorithm. In this section we want to quickly describe other improvements that are already commonly used. Our ray tracer implements one of these, namely early ray termination.

Empty space skipping: In section 2.1.2 we described a possible way to determine the starting point for a march through the volume. However, if the volume is far from the near plane of the viewing frustum, the renderer would step with rather small step width through a vast amount of empty space. Fortunately, volumes are usually on a uniform grid. Hence, the whole volume can be contained within a cuboid. In general, we speak of a *bounding volume*. Because of its rather simple mathematical nature, the intersection between ray and bounding volume can be calculated quickly. The renderer first determines *if* ray and bounding volume intersect. If they do, the first intersection (i.e. the one closer to the camera) is used as a starting point for the march. Because the sampling only starts after jumping over all the empty space, this is called empty space skipping.

This method may be extended to improve performance further by skipping empty spaces *within* the bounding volume. Therefore, in an offline preprocessing, the volume is divided into smaller cubes. For each cube the "emptiness" is determined and stored in a special field within the data structure. During sampling the renderer first checks if a cube is empty and if so, it skips it directly, otherwise it proceeds to sample as usual [FSK13].

Early ray termination: In theory, every ray has to pass through the entire volume to calculate the accurate colour for a pixel. However, in practice this might not be necessary in most cases. The transfer function returns a colour with an additional field for opacity. When the samples are accumulated, the opacity increases. Once the opacity exceeds a certain threshold, subsequent samples would not influence the colour significantly anymore, and they can be omitted.

Dynamic level of detail: [FSK13] advocate dynamic transitions between levels of detail (LOD) while sampling. The volume is preprocessed offline to calculate several resolutions for partial volumes, i.e. in several iterations the resolution is reduced. The renderer then chooses the appropriate resolution due to the distance to the camera. Partial volumes far from the camera can be sampled with a lower resolution and thus sampling may be performed faster. [FSK13]

How much computation time these improvements may save strongly depends on the transfer function and other visualisation parameters including the viewing angle and distance and of course on the volume data itself. [FSK13] have shown that in many cases great amounts of sampling time is saved by one of the aforementioned improvements.

Empty space skipping as well as *early ray termination* are improvements specific to ray casting and ray tracing. There is a variety of more general improvements, which will not be covered by this thesis.

2.2 k-d trees

A k-d tree is a binary tree used to organise points in a k-dimensional space. Each node in the tree represents a partial space. The whole space is represented by the root node. A partitioning of the entire space into two so-called *half-spaces* is represented by the root node's children. Subsequently, the two children may be partitioned again by adding two children to them in turn and so forth. In general, the tree is not balanced. That is, of two siblings one may have children while the other has none. However, no node may have one child. Since a partial space can be *either divided or not* every node has exactly two or no children at all.

The space to be organised can be k-dimensional with $k \geq 1$. A *1-d tree* is a regular binary tree as commonly used in data structures. A *2-d tree* is often called *quadtree*, *3-d trees* are called *octrees*. For $k = 1, 2, 3$ the splitting geometry is a single point, a straight line and a plane respectively. For $k > 3$ there is no geometric representation of the splitting. More general, splittings in any dimension are performed along *hyperplanes*; that includes $k \leq 3$ (e.g. a straight line is a hyperplane in a two-dimensional space).

Since all k-d trees are binary trees, how to split several dimensions? Each level of the tree splits a certain dimension. For example assume a 2-d tree. The root node (level 0) may split the space by a straight line parallel to the x-axis. Its children (level 1) split their partial spaces at a line parallel to the y-axis, the next level again parallel to the x-axis and so forth.

In general, the partitioning of a space does not lead to partitions of equal size. Often k-d trees split spaces so that they are both of equal value according to some taxonomy. The exact position of the splitting hyperplane is stored within the node.

Figure 2.2 shows a 2-d tree and the resulting tiling in the plane. Note that the leaves have different sizes. The splitting position must be stored within the nodes.

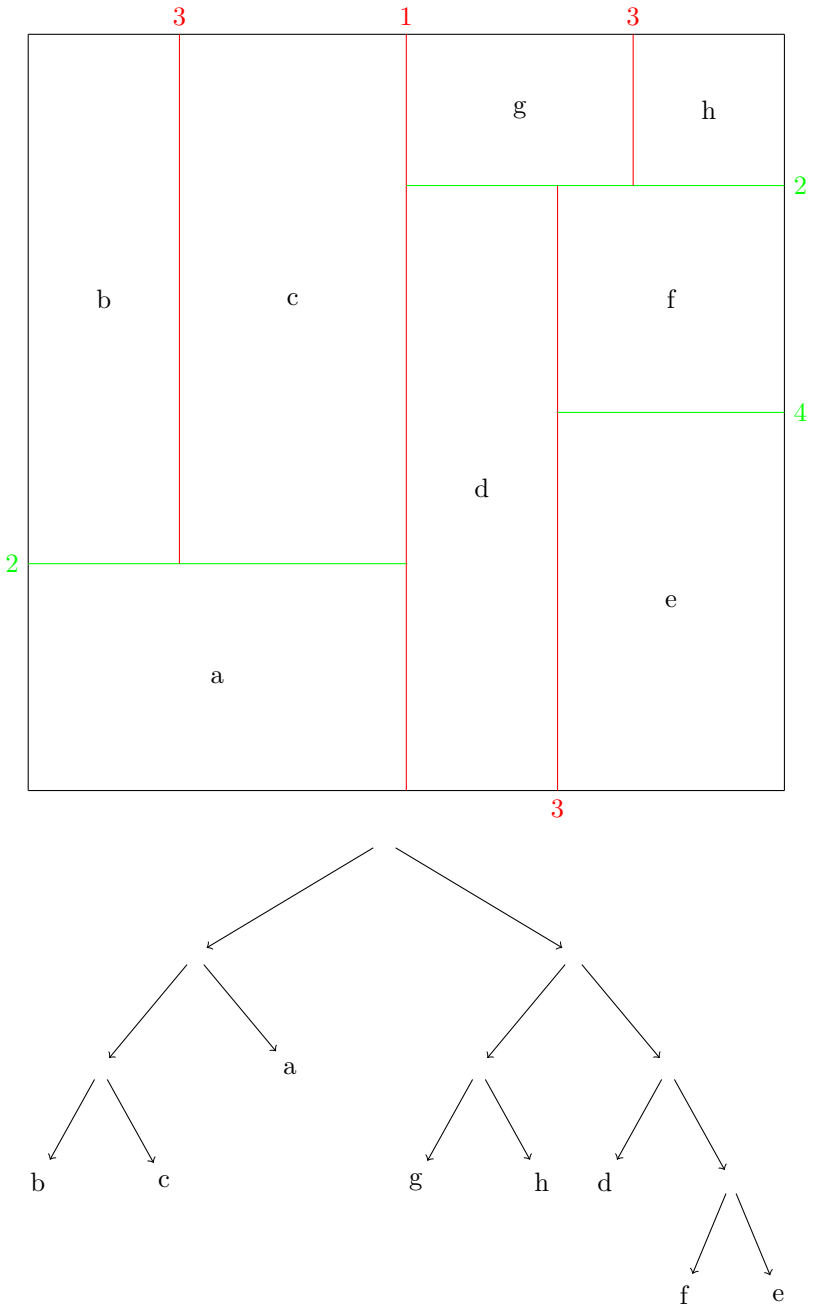


Figure 2.2: Example of a 2-d tree, dividing an area into tiles. The numbers indicate levels in the tree, the letters identify leaf nodes.

2.3 Buffering

Buffering provides an important means to real-time and interactive visualisations. Common renderers use *double buffering*. Monitor's refresh rates are often higher than frame rates achieved by renderers, even more so by ray tracers. In order to keep an image for the monitor available at all times, a frame buffer used. The frame buffer simply contains the colour for every pixel to be displayed on screen.

However, if the monitor displays an image while it is still being calculated, some parts of it may be old while others were already updated. To avoid this two framebuffers of equal size are kept in memory, called "front" and "back buffer". While the *front buffer* is used to feed the graphics output with the information to display the image, the render algorithm fills the *back buffer* with data for the next frame to display. Once the entire frame is calculated, *front* and *back buffer* are swapped making the previous *back buffer* the new *front buffer*.

Dayal et al. extend common frame buffers by a temporal dimension. Each pixel in the buffer now is a (limited) queue that keeps several samples from the past. New samples are add in the front and, if full, the oldest sample is dropped. Keeping a sample history enables them to calculate temporal colour gradients for each position. They also are able to include older samples into the image when the scene is rather static. They use this ability to implement a dynamic and local anti-aliasing.

2.4 Image space properties

In order to guide sampling and focus computation power on areas which require it, Dayal et al. include image space properties for analysis. Calculations in image space are mostly simple to implement and their algorithm's execution time can be estimated. Most importantly, they do not depend on scene complexity. Hence, computation time also remains unaffected by sudden scene changes. And yet they often reveal plenty of information about the scene.

2.4.1 Colour gradient

Colour gradient can easily be calculated for every sample in a buffer. There may be different definitions for what exactly the colour gradient is. Dayal et al. propose "average horizontal and vertical absolute luminance differences" [DWWL05]. Let $f_l(x, y)$ be a location's luminance, i.e. a vector with a component for every primary colour. For each sample at location (x, y) they calculate $g_x = \frac{|f_l(x-1,y)-f_l(x,y)|+|f_l(x+1,y)-f_l(x,y)|}{2}$. g_y for the vertical gradient is calculated accordingly. If g_x and g_y are small, the colour changes only gradually in the nearest surrounding. A high colour gradient indicates abrupt changes observable as *edges* in the image.

Dayal et al. extend the concept by temporal gradients. They keep several samples for the same location in a buffer. To calculate the temporal gradient they examine a new sample and the most recent one in the buffer. Let $f'_l(x, y, t)$ be the luminance at position (x, y) sampled at time t. The most recent sample in the buffer was taken at time t_0, the new sample at time t. This leads to $g_t = \frac{|f'_l(x,y,t) - f'_l(x,y,t_0)|}{t - t_0}$.

A sudden change in colour (e.g. caused by occlusion) leads to a high temporal gradient.

2.4.2 Colour variance

Colour variance is defined as the statistical variance of colour within a certain region. We analyse colour variance within leaf nodes of the k-d tree. Say an area consists of the n colour samples $S = s_1, s_2, \ldots, s_n$. Its colour variance may be defined as:
$Var(S) = \frac{1}{|S|} \cdot \sum_{s \in S} (s - \hat{s})^2$

Where $\hat{s}$ is the average luminance of S.

Colour variance measures the "width" of the examined area's colour spectrum. A high colour variance indicates a non-uniform area; a low one indicates a virtually uniform area. If the area is single-coloured, the variance equals zero (because $\hat{s}$ equals s for all samples).

A single-coloured area requires only one sample to determine the correct colour for the entire area. An area with high colour variance on the other hand requires many samples (worst case: at least one per location) to achieve sufficiently accurate sampling.

2.5 CUDA

Compute Unified Device Architecture (CUDA) is a parallel computing platform and programming model invented by NVIDIA[2]. It allows the programmer to access parallelisation mechanisms directly and utilise the GPU for general purpose computations (GPGPU). That way, it is also possible to implement alternative rendering algorithms that do not follow the common rendering pipeline implemented in GPUs' hardware. The programmer may write code which is executed on the GPU, directly access GPU memory and control synchronisation between GPU and CPU. CUDA provides extensions to the C, C++ and FORTRAN languages, one of which is a set of modifiers defining a function's type. Depending on the type, a function is executed on the CPU (called *host*) or the GPU (*device*).

[2] http://www.nvidia.com/object/cuda_home_new.html

2.5.1 Kernel and parallelisation

Kernels are executed on the host, i.e. on the GPU. The programmer can write a set of kernels; the operations within these functions are executed directly on the GPU. Parameter passing is handled automatically by CUDA and corresponds to regular function calls; there is no return value however.

To structure code executed on the GPU, there also are so called *device functions*. They are equivalent to functions as known from imperative languages, however, device functions may only be called from kernels. Unlike kernels, they do have a return value and, additionally, pointers to variables in shared memory as well as automatic memory (i.e. local variables in kernels) may be passed back and forth to exchange data.

Kernels are always executed in parallel. Therefore, a kernel call must be *configured*. The programmer must specify how many parallel threads should execute the kernel's code. Threads on the device are organised into blocks.

Each block and each thread within a block is assigned an ID, which is accessible from within the kernel code. That way, different threads can be assigned to different data for example. There are no global synchronisation mechanisms between threads. There are, however, means for synchronised threads within one block. Furthermore, there are some atomic operations like addition and increment that may be performed on shared memory.

2.5.2 Thrust

Thrust is a library providing a high-level programming toolkit that is interoperable with CUDA. The library contains a set of data structures and functions that facilitate using the GPU for common problems. It provides vectors and the means for sorting and other more specialised operations on vectors. For more information consult the CUDA documentation[3].

[3] http://docs.nvidia.com/cuda/thrust

3 Adaptive sampler

Rendering images using the ray casting technique allows for the choice of alternative sample distributions across the image. Basic and well established approaches cover the image plane with a grid of equidistant sample positions, e.g. one sample at each pixel's centre. For anti-aliasing the grid may also place several samples within one pixel. We also apply a regular grid, but we do not sample all pixels in a fixed temporal order (thus frameless). We exploit the additional degree of freedom and sample *selectively*. The selected sample locations are chosen by the sampler, which *adapts* to current image space properties. This way, some locations are sampled more frequently than others, resulting in samples, that are located freely in space-time and a great range of means for adaption.

This idea is not new. The basic method is described in the paper [BFMZ94] from 1994 written by Bishop et al.. They loosen up frame restrictions by sampling the entire image plane uniformly at random. Further improvements to this approach have been made in the past. We base our work mostly on the ideas of Dayal et al. as outlined in their paper [DWWL05]. There are, however, two crucial issues: their techniques are developed for and tested on polygonal scenes. The question arises, whether their approach can be applied as successfully to volume data. Second, while describing their ideas in detail, there hardly is information as to the technical issues; in particular, how to implement their guided sampler fully on the GPU and exploiting parallel programming. In this chapter we provide a working implementation of their ideas to guided sampling and extend it to our needs. In subsequent chapters we experiment with their reconstruction technique and evaluate how well the approaches work on volume data.

The sampler is implemented entirely on the GPU. Therefore, the volume data must be loaded into video memory. The adaptive renderer needs several phases in addition to the common rendering phase. Figure 3.1 shows an overview of the rendering loop by which our renderer operates.

```
while (running) {
      sample(buffer);
      reconstruct_image(buffer, out_buffer);
      assess_tiles(leaf_array);
      sort_by_variance(leaf_array);
      splitmerge_condition(leaf_array);
      splitmerge(leaf_array);
}
```

Figure 3.1: The rendering loop sketched in pseudo code.

3.1 Sampling

3.1.1 Sample ratio

The number of samples taken by the sampling kernel weakly depends on the degree of parallelisation provided by the GPU. We want to exploit parallelism to the maximum and keep the amount of sequential operations low. Consequently, the number of samples taken at once is also independent of the buffer resolution. Theoretically, the resolution, i.e. the number of pixels in the buffer, can be both higher or lower than the number of samples taken in one iteration. We experiment with a resolution of $768 \cdot 768 = 589,824$ pixels and different amount of samples. We measure how the amount of time spent with sampling depends on the amount of samples.

We call the ratio of amount of samples divided by pixels in the buffer *sample ratio*. The sampler needs feedback provided by an analysis algorithm in order to guide sampling efficiently. We experiment how well this mechanism works with different sample ratios. Higher sample ratios obviously lead to more up-to-date buffer content. But sampling volume data can be very time-consuming and may decrease interactivity. Hence, the more accurate information may not necessarily increase the performance. If the number of samples is too low, the post-processing is performed too often; the changes are too slight. Plus, low sampling rates let the renderer appear lazy and many stale samples remain in the image.. Through low sampling rates, the sampler may be misguided, i.e. the adaptive sampler's state represents an old situation while in reality the scene has already changed vastly. It might perform poorly then, further diminishing the renderer's usability.

The overall goal is to reduce sampling time without vastly reducing image quality. By using adaptive sampling, it may be possible to reduce the amount of samples by focusing these samples on the right areas. The sample ratio indicates vaguely how much time is spent with sampling compared to a conventional framed renderer. The overhead caused by sampler adaptions must be considered, since they are not necessary for a framed renderer. If the ratio is one, the renderer takes as many samples as there are pixels for a full non-anti-aliased image,

i.e. it requires as much sampling time as a regular frame. Hence, we experiment with rates less than one.

Since it samples less than all pixels, the renderer has to choose *where* to sample. In the following, we will describe the means to smartly choose regions for higher sample densities.

3.1.2 Buffering

As described before, conventional framed ray tracing uses double-buffering, but this technique is not suitable for frameless rendering (see in particular [BFMZ94]). Thus, an alternative approach is needed. A single image buffer is one possibility. The sampler updates arbitrary pixels and "snapshots" of this buffer are displayed. Because samples do not form coherent images, they are passed through filters in post-processing to reconstruct an approximated image. According to [DWWL05], for image reconstruction, it is useful to make older samples at the same location accessible. Dayal et al. propose a *deep buffer*, which keeps several samples for each location in a queue. New samples are added in front and push back all older samples by one position. If the maximum size is reached, the oldest sample is dropped. We adopt this buffering technique and experiment in our implementation and experiment with both deep buffer techniques and "shallow buffering", i.e. a deep buffer with depth 1.

The buffer is kept entirely in GPU memory and is quickly accessible by both the sampler and subsequent processing. For a buffer with depth n, in order to add a sample, $n-1$ older samples need to be moved in memory first. Thus, by increasing n not only the memory usage but also the computation time for sampling is increased. Again the balance between quality and complexity needs to be considered carefully.

We implement the deep buffer simply as an array on the GPU's shared memory. The deeper layers are accessed by simple pointer arithmetic. For example: in a buffer with resolution $768 \cdot 768$ samples, consider the position (x, y). In a linear (one dimensional) array of samples, this sample is located at index $y \cdot 768 + x$, while a sample older by n time steps can be found at $n \cdot 768^2 + y * 768 + x$.

For subsequent processing, the buffer stores additional information for each sample. In addition to the sample itself, the buffer also contains local gradient estimates in x-, y- and temporal (or z-) direction. This information will later be used to guide some of the reconstruction filters. In order to calculate values such as colour gradient, each buffer entry also contains a sample age. We use CUDA's means (**cudaEvents**) to measure the time.

3.1.3 Subpixel randomisation

The benefit of a deep buffer is, that older samples are accessible at each location. When the scene is quickly changing, older samples are not really of interest. Assuming however that the

scene is static, after some time, samples are taken repeatedly the same location. If samples are taken at the *exact* same location, they will always exhibit the same luminance value and thus they are redundant to older ones. Dayal et al. use older samples to increase the local sample density and produce an anti-aliased image. Obviously, this only makes sense, if the sample locations slightly differ for samples of different points in time. [DWWL05] does not mention how exactly they choose subpixel locations to sample. We implement this feature by by *randomising* the subpixel location. If switched on, the deep buffer contains various samples of rays that passed the image plane at different locations within one pixel. Thus, if the the renderer displays a static scene, it buffers samples of the same location with a slight jitter. We allow for different degrees of jitter, ranging from very small differences up to half a pixel width. It then may include such older samples to increase the pixel's resolution. With a depth of about 5, there likely are 5 subpixel locations sampled instead of just one. Offline ray casters often similarly use higher sampling resolutions, taking several samples for each pixel. With the efficient deep buffer, the renderer can dynamically switch anti-aliasing on and off by including older samples or not. With advanced image reconstruction, as Dayal et al. propose it, anti-aliasing may even be used partially; that is, for image plane areas that cover static parts of the scene, anti-aliasing is used and older samples are included. Other areas may show changing parts of the scene and exclude any older samples showing only the newest. In these areas the image obviously exhibits less quality in favour of currentness.

3.2 Means for guided sampling

Our goal is to detect regions that require high sample density in order to guide the sampler in its decision where to sample next. The scene and rendering parameters may constantly change, which is why any method for analysis must perform quickly. Furthermore, the implementation should be mostly parallel, utilising the strength of modern GPUs.

Dayal et al. propose dividing the image into tiles backed by a 2-d tree. k-d trees (section 2.2) are well-known data structures in renderers. They allow for efficient hierarchical partitioning of any space, are quickly adapted and searched. Further they are quite economical in memory usage.

A tile in image space is a rectangular area within the image. Using the k-d tree, the tile is represented by a leaf node of the tree. Section 2.2 explains how to back a tiling with help of a 2-d tree.

In the following, we describe how we implement image space tiling on the GPU. The challenges mostly lie within parallelisation. We want to adapt and traverse the tree in parallel. Adaptions must not cause dirty read or write operations and thus must be organised cleverly. Further, we try to limit time for tree traversal using a simple acceleration data structure.

3.2.1 Parallel tree processing

The tiling is stored as a 2-d tree. The tree is a simple pointer structure within GPU memory. Each node has two pointers to its possible two children and one to its parent node. Further, several variables are stored within a node, including the tile boundaries and image space properties like the latest value of the corresponding tile's luminance variance.

For both adaptions to the tree and consultation of the tree by the sampler, we mostly work with the leaf nodes. To save the time necessary to walk from the root node to the leaf, we store all leaves in an array of pointers. This is easily possible, because we keep the number of tiles constant. This also facilitates distributing the leaves over the threads for parallel analysis. Of course, leaf nodes do not have to be on the same level and thus cover different amount of pixels.

To adapt the tiling and thus the tree to scene changes, we need support for two basic operations, namely *merge* and *split*. Due to the nature of k-d trees, these operation are very simple: splitting a node n is done by adding two children n_1, n_2 to n. In the image, the area covered by n is now split into two tiles. The area covered by n is split into two halves and the new boundaries are assigned to n_1 and n_2. Meanwhile, n keeps its boundaries stored within the node. In case of a merge operation, it requires no update of the data. Because n_1 and n_2 together cover the same area as n, merging them simply requires to remove n_1 and n_2 from the tree. These operations are performed very quickly. To keep the amount of tiles/leaves constant, the renderer performs as many splits as merges in one iteration.

The question arises, how to perform this in parallel, without having two threads interfere with each other's operations. We suggest a processing performed in three phases.

First, for n leaf nodes, we use n threads to assess each leaf node. How assessment is accomplished is described in section 3.2.2. The evaluation of each leaf results in a luminance variance value by which we then sort the array. Thus in the following, leaves with low variance values are located on the left of the array, while such with high variance are on the right.

We split/merge at most x leaves in one iteration of our sampler. Thus, the second phase uses x threads to detect which nodes actually should be merged and which should be split. There are several restrictions that may prevent merging or splitting of single nodes. These restrictions include size constraints for example. Our conditions are outlined in detail below. The kernel of the second phase will check the x leftmost nodes for the merge condition and the x rightmost nodes for the split condition. Nodes that match the condition will be inserted into a respective list. The insertion into this list is atomic and can thus happen in parallel. We use CUDA's atomicAdd() operation to request an index of this list and then insert the element at the given position.

In the third phase, the merges and splits are actually carried out. Merging happens first, because it reduces the number of leaves by one. Thus, an array position will be freed, so that it subsequently will be filled with the one additional leaf from splitting. For this reason, we always perform as many splits as merges. The minimum of possible splits and merges will form an upper bound for the amount operations carried out.

The merge condition is slightly more complex and we explain it thoroughly below. For two siblings that can be merged, only the left one will be marked as mergeable (added to the merge list). The merging then works as follows: through the pointer in the left leaf l, we determine the parent node p and then the sibling s of l. Note that l and s are both in the leaf array. By merging s and l, they are removed from the tree and from the array and p becomes a leaf. p should then occupy one of the array positions of either l or s.

We are aware of l's array position, because we analysed l as one of the left-most leaves in the array. But how to find its sibling s? The array is sorted by luminance variance and does not reflect the tree's structure at all. We could perform a search, however that would be quite costly and require at least logarithmic time. Instead, our algorithm uses a small trick: the array only contains pointers. Thus, we do not need to find s's position in the array at all. We simply reuse the allocated memory for s and copy the node p to this memory block. Thus, p is "automatically" inserted into the leaf array, leaving only the pointer of p's parent invalid, but updating it is trivial.

If a node n is selected for splitting, two child nodes are added and the pointers are updated accordingly. We know n's position in the array because that is how we selected it in the first place. One of the new children may be stored at n's former location, while the other can be stored at l's location freed by merging. Now, merge and split operations are completed for this iteration.

3.2.2 Tile assessment

In order to adapt the tree, the renderer must assess each tile and determine whether it should be split or merged. We adopt *luminance variance* as a measure from [DWWL05]. Remember section 2.4.2: Luminance variance is a measure for how uniform a tile's colour distribution is. A high variance indicates a wide distribution in colour while zero indicates a plain-coloured tile. We want to split nodes with high and merge such with low variance in order to keep the variance in balance among all tiles. If the scene changed previously, the luminance variance of each leaf needs to be updated. In the tile assessment phase, each leaf is assigned one thread for evaluation. Each thread calculates the variance by iterating over the pixels of the tile. Because big tiles need plenty of computation time for evaluation, we estimate the variance by skipping pixels in a regular pattern. To ensure all samples are regularly included into analysis, we cycle the patterns after each iteration.

In some scenarios luminance variance seemed to "drag behind" with accurate assessment of tiles, for example when moving the camera and tiles with high variance suddenly become empty and prior empty tiles cover a part of the volume. This scenario usually does not occur in typical polygonal models. Hence, it likely does not cause any issues with the work of Dayal et al.. Our idea is to facilitate splitting big nodes so that newly covered areas are quickly sampled densely.

We experiment with a slight adaption of luminance variance and introduce weighted luminance variance. It is defined as follows:

Let $S = \{s_1, s_2, \ldots, s_n\}; s_1, \ldots, s_n \in L$ be the samples of a tile. Each is a luminance vector. The weighted luminance variance is defined as:

$$CVar_w(S) := |S| \cdot CVar(S) = \sum_{s \in S}(s - \hat{s})^2$$

This implies that bigger tiles are preferred over smaller tiles with the same variance. An empty region has maximum-sized tiles, because they are plain-coloured with the background colour. As the volume moves, the variance of prior empty tiles increases, while other tiles become empty and their variance eventually drops to zero. Note that the decrease in variance does not happen immediately. The sampler works in a randomised manner. Thus, even small tiles may take several iterations until all sample locations are updated.

Because pixels in large tiles are sampled relatively seldom, their variance also increases slowly. Weighting the variance with tile sizes, gives bigger tiles a better chance to be selected for splitting. We suppose this adaption is only feasible for volume date. As Dayal et al. render complex scenes, the described scenario does not occur. Volume data on the other hand is often viewed from afar leafing areas of the image plane empty. Thus, the differentiation between tiles which cover "nothing" and such that cover parts of the volume are accurate in this context.

We experiment with both assessments and compare the results in different scenarios. Since calculation of both methods is almost identical, a dynamic switchover can also easily be implemented.

3.2.3 Split and merge conditions

We enforce a simple split condition to keep our tree effective: a minimum node size must not be underrun by any split operation. Making nodes too small can have a negative effect on sampling: because pixels in small nodes are more likely to be sampled, very small nodes will be sampled "too densely". Of course, sampling a location densely may improve the image quality in this tile (due to subpixel randomisation). But sampling a single location more often than the buffer is deep, is useless. The additional samples will simply be dropped. Furthermore,

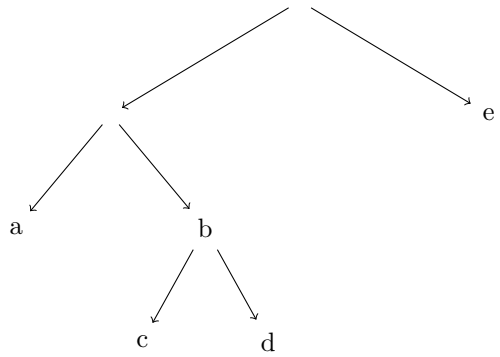


Figure 3.2: Example for failed merge condition for leaf node a. While a appears in the leaf array, b is not. Its children prevent b from being split.

increasing sample density above one per pixel should happen gradually over several iterations, when the scene is static. First, as much area of the image plane as possible should be covered with new samples.

Merging is not as trivial as splitting nodes. Before performing a merge, the sampler has to determine several **merging properties** and decide if a node can be merged or not. The renderer first checks, whether merging two nodes violates the size restrictions we defined. Similar to lower limits, we also allow to define upper size limits. Nodes that cover a huge area will cause these pixels to be sampled very scarcely. With both limits, we keep the size in a reasonable range.

Once the size condition is matched, the renderer check whether the examined node is a left child of its parent node. In the leaf array there are both left and right children of parent nodes. Because the merge function must only be called once, we choose either of them and ignore the other.

At last, the algorithm determines whether two siblings are possible to be merged at all. Consider figure 3.2. It shows part of a 2-d tree with the leaf nodes a, c, d, e. Say that node a matches the first two merging criteria, i.e. a's parent is not too big and obviously a is a left child. Still, a and b are not mergeable in one step, because b is not a leaf node. c on the other hand would be marked as mergeable with d. If c and d are merged in this iteration, a and b can be merged in the next. If all three conditions are matched, the left sibling node is added to the atomic merging list.

3.2.4 Tile sizes

By restricting the tile sizes for the above-mentioned reasons, the question arises what sizes to use as a limit. While we experiment with different values for the upper limit, we calculate the lower limit as follows. Remember that the smallest leaves cover areas, which are considered most important. By making all leaves equally probable for sampling, these areas will be sampled with the highest density. By adjusting the smallest tile size, we can fix the sampling rate to one per pixel and iteration.

Let s be the total number of samples taken in one iteration and let l be the number of tiles/leaves in the 2-d tree. Since all leaves are equally probable, the sampler spends $\frac{s}{l}$ samples on each tile. Thus, the smallest tiles should cover at most $t := \frac{s}{l}$ pixels in order to achieve a sufficiently high sampling rate. We use this value as a reference for our lower size limit.

3.2.5 Sample distribution

The goal of tiling is, to find an appropriate distribution of samples across the image. *Sample locations are chosen randomly.* Making all sample location equally probable is one naïve way to do this. Our renderer tries to guess "more important" locations using the tiling by the k-d tree, as proposed in [DWWL05]. We experiment with different ways of selecting a random tile and subsequently a random pixel within this tile. While the two strategies described below seem to be very similar, they differ vastly in performance. Currently, the following strategies are implemented:

- **Random**: This is the basic sampling mode as proposed in [BFMZ94]. It considers all sample location equally probable and chooses one randomly. We implement this method merely for comparison to other methods.
- **Equally probable tiles**: In this mode, all tiles are equally probable to be sampled. The sampler first chooses a random tile and afterwards a random pixel within this tile. Obviously, any tile may be selected multiple times and there is no guarantee that *every* tile is selected at least once. However, due to the great amount of samples taken, this is irrelevant in practice.
- **Fixed per tile**: This mode statically selects every tile at least once. CUDA organises kernel threads in blocks (see chapter 2.5). We use a multiple of the amount of tiles as block count. Then, the sampler statically assigns blocks to tiles, such that each tile is assigned the same amount of blocks. Hence, every tile is always selected. All thread blocks are of equal size, thus the same amount of samples are taken within each tile, independent of its size. Note that the expected probability distribution does not change in comparison to the *Equally probable tiles* strategy. The position within each tile is chosen randomly.

While the second and the third strategy seem to be very much alike, they behave differently in practice. In ray casting, the duration to sample a pixel may differ vastly at different locations. If a ray misses the volume, the sampling time is minimal: after a negative hit check with the bounding volume, the thread completes its workload. If a ray hits dense areas, it needs some steps through the volume but the ray will be saturated quickly. Thus, such a ray needs moderate time for traversal. Rays that pass through permeable regions of the volume may take plenty of time for traversal. In the worst case they pass through the entire volume.

Due to CUDA's organisation of threads into blocks, the runtime behaviour of *Equally probable tiles* is much worse. A thread block requires as much time as the slowest of its threads. Because our sampler makes rays through the volume highly probable, a majority of thread blocks calculate at least one ray that passes through the volume. Thus, almost all threads that traverse missing rays are slowed down due to a thread that slowly traverses the volume.

By reorganising the block distribution, we group threads that hit the volume and such that miss the volume. Now, entire blocks complete their workload faster, because all of their threads miss the volume and subsequent blocks are executed earlier. This simple rearrangement decreases the sampling time vastly and makes the renderer perform much better. We provide a detailed analysis in our evaluation chapter 5.1.3.

As mentioned before, "emptiness" is rarely a part of polygonal visualisations. That is, if the camera is mostly located within the polygonal model, all rays hit some polygon. There are, however, similar effects regarding traversal durations: depending on the complexity of the scene (and renderer) the traversal time may vary considerably. We are unaware of the complexity in case of [DWWL05]. For volume data this is an inherent issue.

3.3 Improvements

3.3.1 Tree-aware sampler

Originally, our sampling kernel consulted the tree only for information about important sampling regions. That is, from the sampling kernel a function call **get_samplepos()** is performed. It returns the pixel/sample location that the randomised algorithm chooses. In our first implementation, the sampler is unaware of the 2-d tree. The function call does not reveal the internals of the algorithm. Sample modes can easily be switched by implementing different alternatives within **get_samplepos()**.

We extended this interface. **get_samplepos()** now returns a sample location plus the leaf node in which the location is contained. The additional effort is minimal. The function determines this leaf node in any case (as long as the 2-d tree is used for sampling) and the sampler may still ignore the additional information if it is not necessary. We still provide random sampling without 2-d tree. In that case the leaf pointer is returned with value NULL.

By providing this information, the sampler is able to feed back and provide information that can be stored in the tree. We use the mechanism to report useful information which the sampling kernel calculates anyways.

3.3.2 Bounding volume hit tests

As described in section 2.1.4 our renderer performs a simple collision check of the bounding volume with the ray to be traversed. This check usually results in a starting position for the ray traversal; or, if there is no collision, the sampler discards this ray.

Usually the collision check is only relevant for the sampling kernel. However, with the newly implemented feedback mechanism, we are able to feed the tree with this information at virtually no additional cost. We update a variable in each leaf indicating whether the corresponding tile covers actual parts of the volume or simply shows the background.

We implement three states that a node can be in:

- **Undefined**: Leaves are in an undefined state, if they just became leaf nodes by merge or split operations. The state indicates that it is *unclear* whether rays through this tile will hit the volume or not.
- **Hit**: This state indicates, that at least one ray hits the bounding volume. Once a leaf entered this state, it does not change its state further.
- **Miss**: A leaf may change from the *undefined* state to the *miss* state. This indicates, that (up to now) not a single ray hit the volume. This state is unstable, as a single hit will change the leaf into the *hit* state.

We use the information for both reconstruction filters and 2-d tree adapting. With help of "miss leaves" we are able to filter out a vast amount of obsolete samples which usually pollute the image after the volume has been moved. Further, we are able to "artificially" push down luminance variance of empty nodes and speed up merges.

3.3.3 Speed up merges

Moving the volume around is a very common scenario. The renderer needs to adapt quickly and focus its samples on the new position of the volume in order to show a focused and up-to-date image. Due to random sampling, suddenly empty areas are only updated gradually, leading to delayed decrease of luminance variance. Thus, the tiles covering such empty areas only merge with a delay as well.

Note that we already increase the chance of big-sized nodes to be split quickly by weighting luminance variance with node size. But this also prevents average sized nodes from being

merged. Consider the following: After moving the object, big nodes are split because the tiles now cover the volume. Small nodes are merged because the luminance gradually decreases. At some intermediate state, the new volume position might be covered by *smaller* (but not minimal) tiles than the old position. The small tiles should quickly be split further to increase the sample frequency. However, the now bigger old nodes prevent this from happening, because weighting makes them more important than the smaller nodes. Thus, the renderer has to "wait" until the luminance variance within the old nodes decreases further. This may cause a delay in 2-d tree adaption.

With our node classification, we can push down *miss* nodes' luminance variance. We accomplish this, by dividing the value by node size. Hence, we *remove* the weight for *miss* nodes and remove their errant advantage over the smaller nodes covering the volume. Remember, this only concerns nodes, which do not contain a single ray that hits the volume. We simply reduce obsolete samples' influence on the guided sampling. If the algorithm initially is mistaken and at a later point a ray hits the volume, the normal *hit* state is restored. Because the renderer only changes the weight, the initial wrong assumption does not discard any data.

4 Reconstruction

Due to selective sampling and locally different sampling frequencies, resulting slices of the deep buffer are not coherent images. Different samples have different ages and thus a slice generally shows a scene/situation that never occurred this way. However, if the scene remains static for some time, the buffer content converges into a real image. The renderer must be aware of this fact and try to approximate coherent images until the buffer content is up-to-date. This step is commonly called *image reconstruction*, because it is a *reassembly* of partial snapshots of a scene. We adopt this terminology. Basically, reconstruction works by applying a set of filters to the sample buffer. Those filters can be *inclusive*, i.e. let samples influence a resulting pixel's colour or *exclusive*, that is, discarding samples according to some criterion. Usually, filters will consider neighbouring samples as well.

There are several points to consider: first, the renderer should accelerate this convergence to the real image if possible. We already accelerate the process by guiding sampling to regions of interest. However, we may also adapt reconstruction so that scene changes are approximated before all data in the buffer is updated. That is, regions that are not up-to-date yet should be approximated with the data available until enough up-to-date samples are available. Second, once the scene is static, the reconstructor ideally should not lower the image quality by unnecessary filtering. Filters need some mechanism that "switches them off" on static scenes or at least minimises their effect on up-to-date samples.

During scene changes, one issue is that old samples remain in the buffer. Especially regions whose luminance variance drops rapidly will be covered by big tiles and thus sampled scarcely. This is not a defect but rather the consequence of guided sampling. A good reconstruction algorithm should detect areas with old samples and hide as many as possible *without* affecting up-to-date samples. In volume rendering, this is even more important, because the "empty" background can often be seen and old samples stand out greatly in front of a dark background.

We adopt ideas from [DWWL05] and exploit image-space properties such as colour gradient and sample age, which can easily be calculated and stored within the deep buffer. We experiment with filters which include these measurements into calculation. Additionally we advocate object-space properties that can easily be detected by our 2-d tree. We use collision checks with the bounding volume to filter out obsolete samples.

We also implement the adaptive Filter as described in [DWWL05]. We follow the description closely and experiment whether their reconstruction technique developed for polygonal models works for volume data in equal measure.

4.1 Reconstruction filters

The task of reconstruction filters is to approximate an image with help of the incoherent content of the sample buffer. Filter algorithms calculate a resulting pixel at a given location by processing samples from the buffer at the respective position and within its close vicinity. Reconstruction filters try to hide artefacts caused by out-of-date samples, which mainly stand out when the scene is changing rapidly.

4.1.1 Adaptive filtering by Dayal et al.

In [DWWL05], Dayal et al. propose an adaptive filter that spreads both spatial and temporal extents of the deep buffer. They use the local sampling rate to derive the size of their three dimensional filter. That is, the inverse of the sampling rate defines the space-time volume which the filter box should cover [DWWL05]. They use local estimates of spatial and temporal gradients to determine the extents in the different directions. That is to say, while the sampling rate indicates the total size of the box, the gradients specify the exact shape. Where the temporal gradients are high, their filter box becomes shallow, i.e. it does not include many older samples but spreads more neighbouring samples in x- and y-directions. This way, quickly changing areas are approximated using the most recent samples, keeping the latency low. If the colour gradients are low, the filter box is shaped the opposite way: It covers little or none neighbours in x- and y-direction but several older samples. Due to different subpixel locations of the older samples, they achieve an anti-aliasing effect on static scenes. As these mechanisms only define the filter size and shape, the question arises which filter to use. [DWWL05] advocates a Gaussian filter as a working solution.

We add an implementation of this filter method to our reconstructor. We try to follow the description in [DWWL05] as closely as possible. Our implementation might not perform as fast as theirs, but it is sufficiently fast to evaluate its functioning on volume data.

Furthermore, we *additionally* implement several minor modifications. That is, we manipulate several parameters and evaluate the effects. We notice that dynamic anti-aliasing does not provide the expected results. By adapting the parameters, we try to determine the exact cause. The modifications include:

- We experiment with different ranges of jitter in randomised subpixel sampling (see chapter 3.1.3). The jitter influences how far off the pixel's centre samples may be

taken. As we notice that different subpixel locations often change the samples' colour considerably, we experiment with different distances from the centre.

- We experiment with a different filter shape: it consists of a (two dimensional) Gaussian filter on the topmost layer of the deep buffer and a "tube" including older samples, but only these at the calculated pixel's location.
- For comparison, we experiment with permanently switched on anti-aliasing. We use a regular pattern of subpixel locations and equal weights for all samples independent of their age.
- We implement different weightings for older samples:
  - Weighting as advocated by Dayal et al.: $e^{-3,47a}$, where a is sample sage [DWWL05].
  - 2^{-a}.
  - 2^{-d}, d is the number of steps into temporal direction of the deep buffer.
  - Fixed weight of 1 for the most recent sample (even if it is an older one, for older samples weighting as described by in [DWWL05].
  - equal weights for older samples

4.1.2 Raw mode

This mode is not an actual filter but merely shows the latest pixel available at each position. Therefore, it simply evaluates the transfer function with the most recent sample taken at this position. We implement this simplest form mainly for comparison purposes. We test and compare the results of more advanced filtering techniques by examining their effect on the raw buffer content. The raw filter will always display the most recent data unaltered and as soon as it is available. It is comparable with the rendering mode implemented by Bishop et al. [BFMZ94]. However, it will show outdated samples in the very same way as new ones. Hence, when the scene is changing the image will look indistinct until enough obsolete samples disappear. Because we guide sampling towards regions that show detailed parts of the volume, other regions are sampled scarcely. Regions that *become* empty through a scene change may contain stale samples that only disappear slowly due to the low local sampling rate. We call these samples *litter pixels*.

They seem to be an inherent problem for frameless volume rendering. If the renderer guides sampling away from those areas, single stale samples will always occur due to the randomised choice of samples. And a guided sampler will always focus on important regions and will reduce sampling rate for unimportant areas.

4.1.3 Static Gaussian filter

Gaussian filters are commonly used in image processing to smooth pictures and reduce image noise. They are based on a two dimensional Gaussian function and a filter kernel of arbitrary size. The Gaussian function is defined as follows:

$$G(x,y) = \frac{1}{2\pi\sigma^2} \cdot e^{-\frac{x^2+y^2}{2\sigma^2}}$$

The function value is used as a weighting at the respective position relative to the pixel being reconstructed. We experiment with statically sized Gaussian filters and reduce image noise, i.e. obsolete samples which stand out, with great success. The Gaussian filter is able to hide most artefacts within the volume. It can even hide litter pixels at the fringe or outside the volume, e.g. after a shifting movement. However, the blurring effect of this filter is visible to the naked eye. It does not only smooth areas that lack up-to-date samples, but also such that are densely sampled already. Thus, the static filters constantly keep the image blurry, even if there is no movement or change of scene. Static blurring filters may be inappropriate in some cases.

4.1.4 Adaptive Gaussian filter

Due to the positive effects of a Gaussian filter on the image, we try to improve the approach of a static filter by resizing the filter box dynamically according to local image properties. We adopt the idea of Dayal et al. to use the local sampling rate as a measure for filter size. The adaptive Gaussian filter is merely a simplification of Dayal's filter, as it uses the same measure for filter size but ignores older samples from the deep buffer. There seem to occur several issues in our implementation when older samples are included. We try to mitigate the effects with a simplified version of the filter.

Theoretically, our k-d tree is able to detect an exact amount of samples per iteration. However, this requires a great amount of atomic operations, which should be avoided if possible. Consequently, we use a local estimate of the sampling rate which can be derived with help of the probability distribution.

Where the sampling rate is high, many pixels are likely to be sampled and thus up-to-date. Consequently, the filter kernel can be small in these areas. If the sampling rate equals one, the filter extent should drop to zero as a filtering on entirely up-to-date samples decreases the image quality. If the rate is low, there may be many old samples and the filter should spread several pixels. Unlike Dayal et al., we do not include older samples from the deep buffer into reconstruction at this point.

As filter kernel size (for both x- and y-direction) we propose $\sqrt{\frac{1}{R}} - 1$ where R is the local sampling rate within the respective leaf of the k-d tree during the last iteration of the rendering

loop. Consequently $\frac{1}{R}$ is the area that a single sample "should cover". A sample rate of 100% implies one sample must cover exactly one pixel. By subtracting one, our filter then has size zero, hence it only includes the pixel itself.

In addition, we limit the maximum kernel size to $7x7$. Greater values have a drastic influence on our renderer's performance. A $7x7$ filter exhibits good properties when used as a static Gaussian filter. The calculation of the entire filter box takes too much time. For high viewport resolutions, not all filter threads may be executed in parallel but must be processed in sequence. Consequently, the high computation times carry even more weight. Besides, bigger filter boxes only increase blurring effects but do not necessarily raise the image quality. We see the purpose of image reconstruction mainly in reducing artefacts caused by incoherent buffer contents, as it is our sampler's task to produce coherent buffer content eventually. While we try to reduce negative effects caused by old samples, we still focus on showing new samples unaltered, if possible. In practice it is more acceptable to have some temporary artefacts than a constantly altered image which never converges to the "correct" image.

4.2 Age filter

We experiment with an age-based box filter to remove obsolete spots of the image. Due to random sampling, tiles with outdated samples are not uniformly outdated. That is, they do not appear as tiles of stale data. They rather contain spots, i.e. small clusters of pixels that are not updated yet, surrounded by up-to-date samples. As mentioned before, this problem is more severe with volume data because the usually strong contrast to the background makes spots more visible.

Our findings of experimenting with filter guidance through *sample rate* show, that it does not perform as well as expected. However, Dayal et al. have shown, that it works outstandingly well on polygonal data. We assume the difference in performance is caused by the properties unique to volume data: the lack of planes, its transparency, the constant visibility of the fringe etc.

We further assume, that a more targeted filter is required to reconstruct partially up-to-date images from such fine grained data. We thus advocate a filter that specifically targets old spots within the buffer and leaves up-to-date areas untouched. Spots appear in both densely and scarcely sampled areas. Such in rather up-to-date regions disappear faster than such in undersampled regions. But nonetheless they occur temporarily. A filtering guided by sample rate does not take this into account. Thus, depending on the exact location, the filter tends to either blur too much and conceal up-to-date samples, or blur too little and fails to hide artefacts. We advocate a filter that ignores the local sampling rate and targets artefacts by a more local property: sample age.

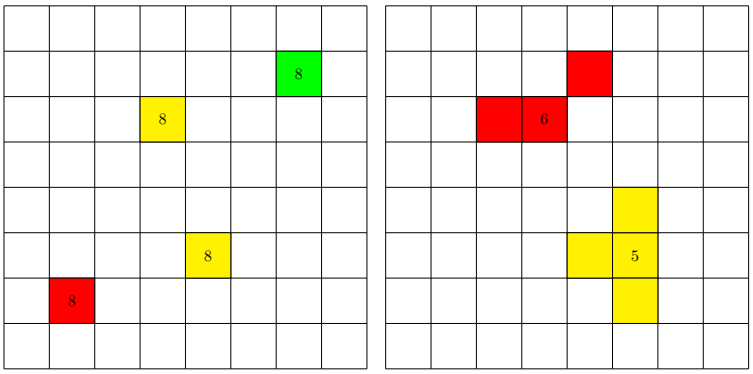


Figure 4.1: White pixels are up-to-date, coloured pixels are out-dated. Left: single spots, i.e. pixels with eight newer neighbours. Right: cluster spots with less than eight newer neighbours. Our less strict version of the age filter removes small clusters of outdated samples or weakens their intensity.

Those spots have a characteristic age signature within the buffer. As our renderer records ages for each sample, we can exploit this information for filtering. *Single spots* appear if the direct environment of a pixel becomes updated, hence the neighbouring pixels are selected by the randomised sampler. The spot itself, however, is a pixel that is not updated yet. Consequently, when comparing the pixel's age with the ages of all its neighbouring samples, it is older than all of them. *Cluster spots* are pixels whose neighbours are mostly newer than the pixel itself. Thus, for a cluster spot an old pixel may be neighbour of an even older pixel. Figure 4.1 shows both types of spots.

We can tackle spots by considering the age of samples and of their direct neighbours with a box filter. The algorithm counts, how many of the neighbours are older. A strict filtering is only applied, if all 8 neighbours are older. Such a filter removes *single spots*. We also experiment with a less strict version and apply filtering if more than k pixels in the environment are older, while k is some value around 5. This also targets *cluster spots*. *If* the age condition matches, the filtering is performed by a fixed-sized Gaussian filter with the spot at its centre. The box size is deliberately chosen small, because the above-mentioned age signature also occurs in areas, where all samples are up-to-date and do not require filtering. Applying small age filters to a static scene usually does not cause any visual difference.

The filter is computationally intensive, as all box filters are. Switching the filter on slightly increases the time required for image reconstruction, but unlike other filters, its overhead is constant. Our less strict version does not only catch single spots but also conceals slightly bigger clusters of outdated samples. Furthermore, its minimal interference with up-to-date samples is advantageous, because it does not diminish the image quality as much as other Gaussian filter approaches do. The filter also does not depend on the k-d tree for information. Thus, it is not vulnerable to inaccuracies of tile assessment.

4.3 Subsequent filtering

Our renderer allows for additional filtering after one of the major reconstruction filters is applied. This way, filters may be combined. We currently implement one additional filter which uses information fed back by the sampler into the tree. Our findings show later, that this filter's effect is redundant with our age filter. Nonetheless, we describe it briefly as it may be useful in combination with other filters.

4.3.1 Hit filter

In section 3.3.1 we describe a feedback mechanism of our sampler. During sampling, a *hit test* is performed. The check is performed on the bounding volume of the volume data. The *tree-aware sampler* stores the result of the check in the according leaf node.

Volume visualisation differs from rendering polygonal scenes. In polygonal scenes, the camera usually is *within* the scene. Volumes are more often viewed from afar. Further, volume data is contained within one bounding volume, in which it usually fits closely, and the rest of the scene is empty. Empty parts are displayed with an arbitrary background colour. Thus, the renderer often shows partially empty images, even when zoomed in closely. When the volume is moved or the camera zooms out, it leaves image regions empty that previously showed parts of the volume. Because with their decreasing luminance variance the tiles are merged and become bigger, the sampling rate decreases rapidly. This often causes "litter pixels", that is, old samples which are only updated after several iterations. In polygonal scenes such artefacts usually do not occur; a "background colour" is rarely visible in such scenes. In volume rendering however, they occur frequently. Figure 4.2 shows a typical appearance of litter pixels. With help of the stored hit test results, we are able to detect many of the litter pixels and hide them. Because filters do not change the buffer content but rather the way it is displayed, the hit filter does not influence the sampler any further.

We make our hit filter a *conservative* filtering technique. This implies that the reconstructor only hides pixels which are highly likely to be litter. In chapter 3.3.2 we introduce three leaf states that classify all leaf nodes in the tree. If a node belongs to the *hit* or *undefined* class, the filter leaves them untouched. Solely nodes classified as *miss* nodes are hidden. If the scene changes, the tree adapts to the change and produces fresh leaf nodes. Subsequent samples report their hit tests to the tree. Only if *all* rays miss, the node will be reported as a miss. Any hit in a following iteration will rehabilitate the node. Remember that there is a transition from *miss* to *hit* but not the other way round. Consequently, if a node contains up-to-date pixels it will always become classified as *hit* eventually and keep this property in the following.

The only nodes that are classified as miss *by mistake* are such that cover the fringe of the volume. Only then, a majority of the rays misses the volume. Because our 2-d tree converges into a stable state when the scene is static, all nodes are classified correctly eventually. Hence,

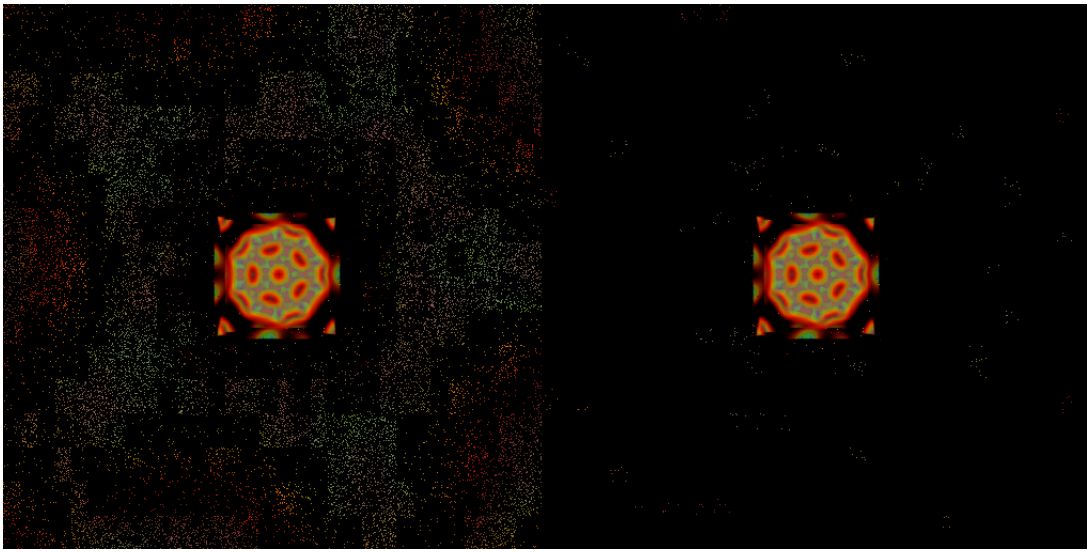


Figure 4.2: A vast amount of "litter pixels" as it appears when zooming out quickly. To the right: exact same situation with activated hit filter. The stale samples nearly disappear.

the reconstructor eventually displays a correct image with no wrongly hidden pixels. In the process, it may lead to "blinking" pixels until the tree becomes stable. However, in practice this rarely occurs.

The implementation is simple: first the appropriate leaf is determined with help of the pixel's location. This is done by walking from the tree's root node to the leaf node by choosing the correct half-space in each inner node (see chapter 2.2 for details). If the leaf is classified as *miss* the pixel is set to the background colour, independent of the buffer content. Otherwise no action is performed.

5 Evaluation

In the following chapters we evaluate our work and provide findings of experiments with previous techniques. In particular, we examine the performance of our sampler and locate possible weak spots. We also examine filter runtimes and the effects of scene changes on these. We thoroughly analyse the impact of sample amounts and distributions on the sampling duration. Finally, we compare different reconstruction filters and their effectivness. Our analysis focuses on Dayal's filter, their dynamic anti-aliasing and our age filter.

5.1 Timing

Interactive renderers must perform quickly in order to provide a good user experience. We analyse how fast our rendering algorithm is. In particular, we analyse each phase of the rendering process and measure the execution time.

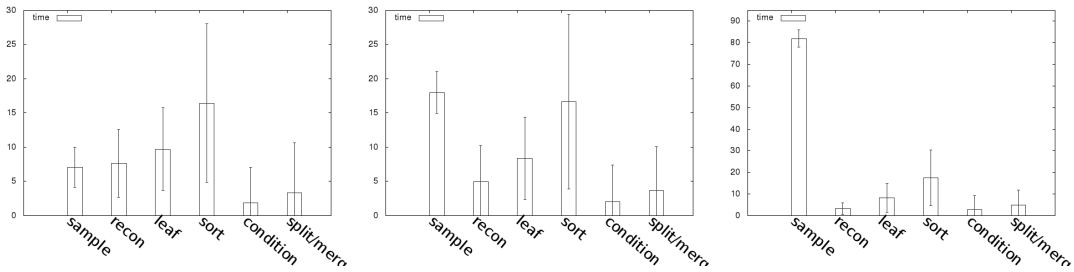
The entire algorithm is executed on the GPU and only controlled from the CPU. Benchmarks are performed on a NVIDIA GTX 680 with 4GB memory. We use *cudaEvents* to measure the execution time of each kernel call. The function call **cudaEventElapsedTime()** returns the passed time in milliseconds with a resolution of around 0.5 microseconds. We try to keep the execution environment as stable as possible: all tests are performed on one and the same machine, no other programs are running concurrently (except for necessary system processes). We perform 1000 rendering iterations to clear the results from local peaks and single delays. After measurement the average times as well as standard deviations are calculated. To make our algorithms comparable, we use several scenarios and renderer configurations and perform the profiling on each of them.

We experiment with three volume data sets:

- Bucky, resolution: 32^3
- Jet, resolution: $720x320x320$
- Chameleon, resolution: $1024x1024x1080$

		sampling	recon	leaf	sort	condition	split/merge	**total**
Bucky	∅	7.08	7.66	9.70	16.46	1.90	3.36	46.16
	σ	2.91	4.94	6.06	11.61	5.18	7.25	
Jet	∅	18.00	4.94	8.38	16.67	2.05	3.67	53.71
	σ	3.08	5.31	6.04	12.76	5.35	6.45	
Chameleon	∅	81.99	3.30	8.18	17.49	2.92	4.89	118.77
	σ	3.98	2.72	6.63	12.92	6.22	6.91	

Table 5.1: Distribution of sampling time and standard deviation over the different phases tested on different data sets. First line shows the average values, the second standard deviation. All times are in ms.



(a) Bucky **(b)** Jet **(c)** Chameleon

Figure 5.1: Distribution of time over the different rendering phases. For big volume data sets the phases for tile adaption and reconstruction become more and more insignificant. The leaf update phase and sorting phases appear to be weak spots of our implementation while parallel tree processing generally works well.

5.1.1 Overview

The first scenario simply runs the renderer in an idling cycle. No user input is received but the k-d tree is maintained and sampling is also guided by the tree. The reconstructor calculates pixel values with the raw filter and no other filters are applied. We render with a resolution of $1024x768$ pixels and take $589,824$ samples per per iteration (75% of the image plane). The volume is rather close to the camera and covers most of the image plane. The results of the time analysis are shown in table 5.1. A visualisation can be seen in figure 5.1.

As expected, time spent on sampling strongly depends on the data set. The smaller data sets Bucky and Jet are sampled quickly while the Chameleon data set requires the majority of rendering time for sampling alone. Our work targets visualisation of volumes by which conventional renderers reach their performance limits. Consequently, tiny data sets such as Bucky are usually not relevant. We analyse our renderers behaviour on data sets of different size to detect both strengths and weak spots. As one can see, for a small data set such as

Bucky, a framed renderer is probably more appropriate. For big data sets there may be great benefits from adaptive rendering, as the additional time required for adaption constitutes only a fraction of the rendering time.

The reconstruction phase performs within a few milliseconds and loosely depends on the scene and data set. As all implemented reconstruction modes require data from the k-d tree, we determine the respective leaf once per kernel execution. The required time is influenced by the tiling. If the tree is deeper, the leaf retrieval requires slightly more time. No complex filters are applied at this time.

The *leaf update phase* is used to reevaluate each leaf's luminance variance. The variance is estimated in order to lower execution time. Yet, this phase is computationally intensive. It requires sequential operations on each tile. Higher resolutions imply more or bigger tiles and thus worsen the delay. There are parallel algorithms for variance calculation. However, we already are close to the maximal number of parallel threads. A parallel algorithm might also add a slight overhead to the calculation time, because partial results must be merged afterwards. This is the second heaviest phase of the adaptive processing (we ignore sampling for now, as it is inevitable and researched much better in other publications). If colour variance is to be used in the future, the parallel calculation must be accelerated. It might help to distribute threads more fine grained over the tiles, i.e. subdivide each leaf in a set of blocks and simply detect an estimate value over the blocks.

Sorting requires a considerable amount of time for each iteration and is the worse spot of our implementation. Depending on the data set the phase takes between 16 and 17ms. We require the sorting phase to detect tiles with high and low luminance variance in order to split and merge them respectively. Using the k-d tree for sampling guidance, we currently do not see another possibility identify such tiles more quickly. The sorting is already executed on the GPU entirely. There is no data transfer between host and device. We use CUDA's Thrust library and assume it is a solid implementation for parallel sorting. However, there might be a faster way if additional knowledge is included.

The last two phases are used for tree adaption perform quickly. The *condition phase* determines which of the leaves match the merge and split conditions. The last phase executes merges and splits on the respective leaves. Compared to the sampling time these phases do not carry weight for big data sets. We implement these two phases so that they can be executed with maximal parallelisation. All operations on leaf nodes are accelerated by the leaves array. The renderer saves time to walk from the root to the respective leaf. The *condition phase* merely relies on atomic add operations on the GPU. They are used for parallel list insertions. Note that the conditions can be chosen arbitrarily and thus this technique might be useful in other applications as well. Merges and splits can then be executed completely independent of each other. Due to our small "hack", an array index of one leaf node suffices to perform a merge with its sibling. There is no need for a look-up. This saves vast amount of time and makes our tree adaptions perform quickly.

		sampling	recon	leaf	sort	condition	split/merge	**total**
Bucky	∅	6.96	6.86	10.38	17.73	2.39	4.72	49.04
	σ	2.83	4.89	5.78	12.077	5.72	6.77	
Jet	∅	24.87	4.01	8.94	17.3	2.40	4.78	62.3
	σ	5.86	3.83	6.31	12.55	5.74	6.70	
Chameleon	∅	66.15	3.93	7.60	17.04	2.82	4.15	101.69
	σ	17.72	3.86	6.43	12.70	6.13	6.18	

Table 5.2: Timing analysis for rotating scene on different data sets. All times in ms. Sampling times slightly differ and vary more due to dynamic scene. Reconstruction and tree adaptions are not affected by the rotation.

		sampling	recon	leaf	sort	condition	split/merge	**total**
Bucky	∅	7.10	11.38	8.16	18.68	2.08	4.64	52.04
	σ	3.5	5.10	3.91	11.37	5.39	6.57	
Jet	∅	15.42	7.96	6.57	16.50	2.09	4.34	52.88
	σ	2.66	4.15	4.73	12.50	5.39	6.40	
Chameleon	∅	51.79	7.16	5.92	15.77	3.89	3.86	88.39
	σ	14.79	2.79	5.29	12.16	6.93	6.07	

Table 5.3: Timing analysis for shifting motion on different data sets. All times in ms. Sampling times are lower, due to delay of k-d tree adaptions. Reconstruction takes slightly longer because of an unbalanced k-d tree. Smaller leaf nodes lower time for leaf update. Remaining phases perform at constant time as usual.

One whole iteration takes about 46ms, 54ms or 119ms for different volumes respectively. Hence, for the tested data sets our renderer can perform between 8 and 22 iterations per second. For productive use, this might be further increased by handling the weak spots of our implementation.

5.1.2 Dynamic scenes

We apply the same configuration as above to dynamic scenes. One shows a constant rotation while the other shows a continuous shifting motion. Results can be seen in table 5.2 and 5.3.

Rotation causes a higher variance in sampling time for Jet and Chameleon data sets. They both are cuboid shaped and by rotating, their projection covers an area of varying size. Thus, the number of samples that hit the volume periodically goes up and down. Bucky is cube shaped and exhibits similar sampling time as in the static scene.

Shifting causes lower sampling times than the static scene. There are two reasons for this behaviour: first, the volume is farther away from the camera and thus exhibits a smaller footprint. Second, the volume keeps shifting into areas where the tiling is coarse. Thus, it is sampled scarcely until the tiling adapts to the new scene. Shifting the Chameleon data set also increases the variance. This data set is rather complex to sample and thus the difference between a hitting and a missing ray carry weight. This is not the case for the simple Bucky data set.

The reconstruction phase's performance slightly drops in the shifting scene. As the volume covers a smaller area of the image, the k-d tree is less balanced and leaf retrieval takes slightly more time. Rotation does not unbalance the tree as much. Leaf assessment on the other hand becomes quicker as in many situations there are no nodes of maximal size. The remaining tree adaption phases perform similarly on dynamic and static scenes. The operations of the sort, condition and split/merge phases have constant complexity. They all operate on the fix sized leaf array.

5.1.3 Sampling

We examine two parameters that may influence sampling time in particular: the amount of samples taken and the distribution pattern of samples across the image. There has been research about sampling collocation and distribution patterns for parallel ray tracers before. However, for guided sampling this is of high significance. Guided samplers focus computational power on the most complex parts of a volume and thus guidance may even damage the performance. The exact parameters must be chosen wisely to gain an advantage. In this section we examine runtimes for different sampling parameters. In chapter 5.2.3 we share high-level considerations that are important for any guided sampler.

Table 5.4 shows different amount of samples on different data sets. Obviously, sampling time increases with more samples, but it does so less than linearly within some ranges. This might be due to the lower influence of overhead when executing more threads in one call. However, overall the relation is close to linear. The times for 100% sampling ratio are likely higher than the sampling duration for a framed renderer. This is due to the fact, that guidance directs more rays to complex regions. Thus, complex areas are sampled more than once while rather uniform or empty areas are sampled scarcely. Consequently, it poses problems if the sampling ratio is chosen too high.

Table 5.5 shows the effect of sample distribution over warps. When each thread chooses any sample in the image, the performance suffers considerably. Assigning spatially close samples to one warp drastically lowers the sampling time. This is due to the fact that spatially close samples *mostly* exhibit similar properties. If sampling takes about the same time in each thread of one warp, none of the threads must idle. Warps that terminate quickly can free the resources earlier. This is a well-known effect in volume rendering (and also ray casting/tracing in

	25%	50%	75%	100%
	196,608	393,216	589,824	786,432
Bucky	2.87	4.90	7.08	9.20
Jet	6.22	12.24	17.24	23.86
Chameleon	27.55	54.29	81.99	110.27

Table 5.4: Effect of amount of samples on sampling time. The correlation is about linear. Duration for a 100% ratio is higher than for framed rendering, because the sampler samples challenging parts more densely.

data set	random	spatially close
Bucky	7.73 (σ=2.41)	4.78 (σ=2.84)
Jet	69.70 (σ=5.04)	12.34 (σ=3.47)
Chameleon	80.83 (σ=5.73)	53.15 (σ=4.20)

Table 5.5: Effect of sampling distributions on sampling time. In brackets the standard deviation per round. All times in ms. Assigning spatially close samples to one warp increases the performance vastly. In addition, the durations vary less.

general). Also the variance over sampling iterations decreases using an organised distribution of samples. Sampling collocation is discussed further in chapter 5.2.3.

We also experiment with different warp sizes. For the same reasons as above, rather small warps perform better. Less samples per thread imply that the duration's variance is usually smaller. If the warps sample spatially close regions the performance gain is maximised. However, if warps are too small, overhead may suddenly carry weight. We experiment with different sizes. In some ranges the results are not conclusive and vary from execution to execution. Eventually, we use 48 threads per block. Resulting times seem to be quite stable and much lower than with big thread blocks. Only by resizing warps we reduced sampling time by up to 50%. The effect can be measured with both distribution patterns. We assume, that challenging micro structures of volume data slow down warps, even if they sample small connected areas.

5.1.4 Filters

We compare the efficiency of different reconstruction filters on both dynamic and static scenes. Note, that the efficiency refers to *our* specific implementation and is not representative for the general complexity of the filtering method. This is of particular importance regarding Dayal's filter. The performance of our implementation is likely lower. We primarily implement the filter for comparison of its output. The absolute duration of execution is important for practical use and may be improved if the filter technique is feasible for practical use. The difference in

data set		Static Gauss	Age filter	Dayal's filter
Bucky	∅	17.28	11.10	6.49
	σ	4.73	5.19	5.14
Jet	∅	16.87	8.03	6.52
	σ	4.84	5.40	5.57
Chameleon	∅	15.33	6.63	6.58
	σ	4.63	2.46	4.99

Table 5.6: Computation time for different filters on static scenes. All times in ms. The slight differences between data sets are due to properties of the k-d tree and occur identically with none filters applied (see previous chapter). Dayal's filter performs fast as on a static scene the filter box is chosen quire small.

performance between static and dynamic scenes however may be more meaningful (as long as it can be explained).

Table 5.6 and 5.7 show the timing analysis for the reconstruction phase with different filters. There are slight differences between the data sets for all filters alike. This is due to the exact tree properties. We discussed this effect in the previous chapter and will ignore it in the following analysis.

As expected a static Gaussian filter is neither affected by the data set nor by movement. It is applied equally frequent and requires the same time for every pixel.

Dayal's filter has a small footprint on static scenes but increases in complexity when the scene is changing. The filter becomes bigger where the sampling rate is low. As we move the volume into bigger tiles, the sampling rate locally drops repeatedly until the tree adapts. Consequently, on average the filter is bigger in moving scenes. Our implementation may be suboptimal and could be improved. [DWWL05] contains thoughts about quicker implementations of the filter. However, the increasing complexity on changing scenes seems to be inherent as the filter includes more data and thus requires more time for processing. Hence, the filter trades quality for responsiveness.

Our age filter seems to loosely depend on the exact scene property and the volume data set. Jet and Chameleon exhibit more uniform areas (also our transfer functions produces less colours for these volumes). We assume that samples are more likely to be scattered and the specific age signature of spots does not occur as often. Remember that spots are single old pixels surrounded by newer ones. However, the correlation does not seem to be significant enough to determine its cause with certainty. On the other hand, our filter seems fully motion-resistant. The rotation does not cause any increase in computation time. For interactive applications, this is an invaluable property as it makes the software's performance stable and immune to worst-case scenarios. Especially for high-performance renderers where real-time aspects play

data set		Static Gauss	Age Filter	Dayal's filter
Bucky	∅	17.98	10.68	22.69
	σ	4.92	4.95	5.27
Jet	∅	15.32	7.43	22.44
	σ	4.28	4.59	5.04
Chameleon	∅	15.48	7.85	25.29
	σ	3.78	4.54	5.32

Table 5.7: Computation time for different filters on dynamic scenes (rotation). All times in ms. Static Gaussian filter and the age filter are not affected by the motion. The adaptive filter of Dayal et al. however skyrockets to a fourfold of the static scene.

an important role, constant complexity is a much-needed feature. We confirm these properties with a shifting motion as well.

5.2 Adaptive sampling

We test our adaptive sampler by applying it to several scenarios with different configurations. We vary parameters including the visualised volume, tile assessment and the scenario itself. Sampling poses different challenges when performed in the context of frameless and adaptive rendering. We do not examine well-known effects which are described much better in other publications. We rather provide findings that are closely related to the adaptive sampling. Within this chapter we discuss the effects of sampling ratios, tile assessment and general difficulties posed by adaptive sampling.

5.2.1 Sample ratio

We introduced sample ratio in chapter 3.1.1. The ratio indicates what portion of the image plane is sampled. Because sampling volume data is computationally intensive, we try to reduce the amount of samples without diminishing the renderer's performance. The visualisation must still be updated quickly after changes to the scene, camera or transfer function. Besides, guided samplers automatically focus samples on areas of interest. Consequently the total amount of samples should be reduced. Otherwise many areas will be sampled more than once causing redundant samples. The amount may not be too low either because both image currentness and tree adaption will suffer from an excess of outdated samples. We experiment with different sample ratios and compare the effects.

Figure 5.2 shows three different sampling ratios applied while visualising a shifting motion. The images are not filtered and show the raw buffer content. There is an obvious improvement

from 25% to 50%. Sampling only a quarter of the image plane seems to be insufficient. Many stale samples remain after the change and even densely sampled areas (inner regions) are grainy. Obsolete data vastly decreases the usability, even more than slight delays of user input. Our experiments with different ratios show that the lower bound is located somewhere around 50%. Due to guided sampling, the image is quickly updated for the most part. Many of the remaining flaws may be mitigated by filtering. Depending on the effectiveness of applied filters it may be lowered even further. Figure 5.2 shows that the volume hardly loses its shape at 50%. There are simply more stale samples than at 75%. A sampling ratio of 25% however visually distorts the shape. A reasonable upper bound might be around 75%. The differences between 75% and 100% are hard to spot. The undersampled regions on the right remain even with a 100% sampling ratio. They can only be tackled by improving the guidance. The other parts of the image hardly differ at all. The impression is confirmed by filtered images. Figure 5.3 shows the (age) filtered equivalents to the pictures of figure 5.2. The difference between 75% and 100% is hardly recognisable. We confirm similar results with other volume data sets.

Keep in mind that too high sampling ratios harm the sampler rather than improving performance. Guided sampling requires smaller amounts of samples due to its more advanced sampling strategy and thus should not be run at a ratio of 100%.

5.2.2 Luminance variance

Dayal et al. propose luminance variance as assessment for the k-d tree. The effectivity of this measure is crucial to the sampler. Without a working assessment for a tile's importance the sampler cannot adapt the k-d tree appropriately and thus cannot guide sampling efficiently. We evaluate whether our adaption, *weighted luminance variance*, performs better on volume data. We use both measures in the same scenarios and count the steps needed for k-d tree adaptions to the changes. We keep the sample ratio constant and thus provide similar conditions to both assessments.

In a "sudden shift", the volume is moved from one location to another without any intermediate states. We freeze the renderer, relocate the volume and continue rendering stepwise afterwards. A gradual shift is a stepwise movement from one position to the other while including all the overlapping intermediate states. Table 5.8 shows the results.

While the results for a sudden shift seem to be inconclusive, weighted luminance variance seems to perform better for gradual shifts. The additional hit-check (chapter 3.3.2) removes the weight from nodes which are likely covering empty parts. In some cases this seems to improve the results further. While these results may not be conclusive enough to manifest a distinct advantage of our adaption, it shows at least that tile assessment may influence the performance of a guided sampler. We aim to raise the tiling's responsiveness to common changes. While our sampler generally works reliably, there are still difficulties that may not be

data set	mode	sudden shift	gradual shift
Bucky	normal variance	12	10
Bucky	weighted variance	15	7
Bucky	weighted, hit-check	10	7
Jet	normal variance	10	13
Jet	weighted variance	12	9
Jet	weighted, hit-check	7	7
Chameleon	normal variance	10	15
Chameleon	weighted variance	8	8
Chameleon	weighted, hit-check	9	7

Table 5.8: Number of iterations until tree is adapted when performing a "sudden shift" and a "gradual shift". The results are generally not conclusive enough to claim an advantage over regular variance. There might be a slight mitigation.

handled by better assessments. The tiling always "loses focus" around the fringe of the volume. Our adaption mitigates the effect but fails to eliminate it. Also see chapter 5.2.4.

5.2.3 Problems with volume data

We experiment with several techniques developed for polygonal data and transfer them to volume data. The ideas include those mentioned by Dayal et al. [DWWL05]. We recognise the following issues that occur with volume data:

when sampling volume data, the variance across sampling times of different pixels is quite high. Rays that do not hit the volume at all are traversed within minimal time, while rays through permeable parts may take several orders of magnitude more. We do not have any knowledge as to the severity of this problem with the work of Dayal et al.. Basic ray tracers do not struggle with this issue as much, because rays usually exhibit more similar properties. They all can be traversed in roughly the same period of time. However, advanced ray tracers support many additional calculations for rays. [DWWL05] does not mention any details about their ray tracer's properties. For volume data in any case this is an inherent problem. The sampling of volume data always exhibits varying complexities depending on the data, transfer function, sampling accuracy etc. Unfortunately, guided sampling can worsen this effect.

Threads on the GPU are executed in blocks called *warps*. All threads within one warp are executed in a synchronised manner. That is, if one thread takes more time, all other threads must wait until it finishes. The guided sampler deliberately makes rays through computationally intensive parts of the volume *more* probable. If warps are organised naïvely, virtually all blocks will calculate at least one computationally intensive ray. We reorganise our warps to sample spatially close parts of the volume while maintaining the probability distribution. The new

distribution drastically lowers computation time of our sampling phase. In some cases it reduces the sample time to a fifth (see chapter 5.1.3 for results). Warps now sample spatially close pixels. Empty regions are thus sampled quickly and the entire warp terminates early.

However, one problem remains: transitions from dense to permeable parts of the volume may exhibit visual edges. Tiles across such edges in turn exhibit high luminance variance, which makes them likely to be split. However, even the smallest tiles often cover both dense and permeable parts. Figure 5.4 depicts the situation. Permeable parts (dark blue) are assessed with rather small values and tiling is coarse. In the left half, where density changes are steep, tiling is fine grained. Despite the smallness, many tiles cover different densities. The same effect as described above occurs and threads in one warp must await the slowest one to terminate, even though they sample spatially close regions. Again guided sampling intensifies this effect by guiding rays to these very locations.

Considering these effects closely, we recognise another problem. By zooming in and out, the volume covers different portions of the image plane. Zoomed in, the volume may cover the entire image; viewed from afar it may be a small fraction, while the rest shows the background colour. The sampler directs most of the rays through the volume and samples the empty areas scarcely. This is the purpose of guided sampling and a wanted effect. However, if the portion of the covered image plane is small, the renderer may sample this area more densely than necessary. Note that oversampling will vastly diminish the renderer's performance: the sampler picks the most complex areas and may sample them redundantly, multiple times.

Currently, we prevent this by implementing a fixed limit for tile sizes. The tiles' size may not underrun a certain value. That way, we ensure that the sampling rate does not exceed one (or any constant). However, this is only a compromise. Our solution implies, that unnecessary samples are simply forced to empty areas, so that they require minimal time. A "real" solution to the problem makes a dynamic adaption of the sample amount per iteration inevitable. If only a fraction of the image plane is non-empty, the sampler must reduce the total number of samples such that all empty areas are sampled scarcely. However, detecting this fraction poses new difficulties and may increase the time required for analysis. We can estimate emptiness of tiles and direct this information back into the k-d tree. But determining the non-empty area requires a big amount of either sequential or atomic operations across all tiles. Both are not optimal for parallel computing. Alternatively, the sampler could estimate the covered fraction of the image plane using the bounding volume and camera distance.

5.2.4 Fringe

While generally sampling volume data with an adaptive sampler works as it does for polygonal data, the fringe of connected components and the volume itself always poses a problem. This is due to the measure by which the tiling is assessed. When the volume is moved, it moves

into previously empty tiles; in general those exhibit maximal size. In order to quickly sample these regions densely, the big tiles must be split. Yet, splitting is only performed once the colour variation is sufficiently high (compared to other tiles). When moving the volume quickly, undersampled regions cause "holes" in the buffer. The newly occluded areas are not sampled densely and the background colour appears between the randomly distributed samples. Figure 5.5 shows the effect. The picture shows a movement to the bottom right causing visible holes. The not sufficiently small tiles are also recognisable.

For polygonal data, there have been considerations about "hole-filling" as a reconstruction mode [WDP99]. Walter, Drettakis, and Parker successfully apply this method in combination with interpolation to reconstruct the solid figures. However, this reconstruction is based on the assumption that the visualised object exhibits an opaque surface. Volume data does not have this property. Hole-filling might effectively restore some undersampled connected components. But it will also hide fine grained holes that are part of the data.

Figure 5.5 also shows that our age filter mitigates the problem at least in some cases. We propose however to tackle the problem at its root. We already try to adapt our sampler to increase sampling rates at the fringe more quickly. But our adaptions only take effect *after* the volume has moved. Even "optimal" tile assessment cannot prevent temporary undersampled areas following a quick movement. Thus, we propose a predicting adaption of the k-d tree as a possible improvement to frameless rendering. If the k-d tree encircles the volume with a mesh of small tiles (in combination with hit checks), it may adapt to movement as it takes place. Hence, the fringe of the volume is always sampled densely and the sampler's focus does not drag behind. Alternatively, the user input may be forwarded to the k-d tree to react early.

5.3 Reconstruction

Our renderer implements several reconstruction filters that may be applied to the raw buffer. We adopted the one of Dayal et al. [DWWL05], developed a variant that is also based on local (per tile) sample rates but does not include deep buffer content. Furthermore, we implement our age-based filter which targets outdated spots in the buffer and particularly focuses on artefacts occurring frequently in volume rendering.

In the following, we evaluate the different filters. We first provide results of our age filter. We compare the filtered output with the raw buffer and the final converged image in order to see how many artefacts the filter can remove. In the subsequent chapter we analyse how different filters perform compared to each other. We focus in particular on Dayal's and our technique. We also provide data about the filters' performance on different data sets in section 5.1, showing that our filter's performance is superior due to constant (and rather low) runtimes and resistance to motion.

5.3.1 Age filter

We apply the age filter described in chapter 4.2 and compare its output to the raw buffer and to the "real image" once the buffer content converges into the static scene. We allow for the filter to remove cluster spots if at least 5 of the 8 direct neighbours are older than the pixel itself. In the figures 5.6 through 5.8 the first picture shows the raw buffer, the second one the output of the age filter and the third the optimal image.

Figure 5.6 shows a shifting movement and figure 5.7 shows a rotation. Both times the raw buffer exhibits vast amount of litter pixels. There are dark spots on the volume and bright spots in between the parts of the volume. They both represent an old scene in which the volume was located elsewhere. The filter succeeds to remove or at least conceal many of the bright spots. Bigger spots that are not removed at least become darker and stand out less. The dark spots are almost completely removed. The filter "restores" areas that still lack up-to-date samples by including the direct surrounding. Consider figure 5.6. The yellow areas appear almost completely clean due to the age filter. Only some spots on the fringe remain.

In figure 5.7 the smaller drops on the bottom are only partially restored. It shows that the filter struggles with restoring small components of the volume.

In general, the filter is able to approximate the image well in areas where the buffer content is at least close to a coherent image. In heavily out-dated areas, however, the filter merely mitigates the lack of up-to-date samples. Note that the images are taken *during* the respective motion. Even the most recent samples become outdated quickly this way. Consequently, all the images will contain some sort of distortions and litter pixels. Most of the unpleasant effects are located around the fringe of the volume. They usually occur when parts of the volume move into image areas with big tiles. Only after a split the sampling rate increases sufficiently. See also chapter 5.2.3 for more thoughts on this issue.

In figure 5.8 the filter reaches its limits. While spots in the inner regions are removed, the undersampled outer regions still exhibit stale samples. Again, this image shows the situation *during* the rotation, thus many out-dated samples occur. Image 5.8d shows, that all spotty areas are in undersampled regions, because they are covered by big tiles. We consider a combination of a filter similar to the one of Dayal et al. with an age-based filter like ours. It may be advisable to heavily blur regions where the sampling rate is very low. However, simply connecting these two filters does not improve quality sufficiently. In many cases the spot filter reverses positive effects of the blurring and the other way round. Also, additional blurring increases the reconstruction time further.

The filter also exhibits a small weak spot when very small areas are displayed. If components of the volume only cover a handful of pixels on the image space, the filter might cause a slight flickering. For more than a few pixels this does not happen.

5.3.2 Comparison of reconstruction filters

We compare different filter methods described in chapter 4.1. We freeze our renderer showing a particular scene and switch between different filters. That way we can easily compare how filters perform in different situations. We focus on Dayal's filter in comparison with our age filter, as the two approaches seem the most promising ones.

Figure 5.9 shows three steps of a shifting movement. The left images are processed by our implementation of Dayal's filter. The right ones are filtered with our age filter. We identify the following effects: Dayal's filter misses plenty of old samples around the chameleon's snout. At least after the third step, there are many spots visible. Due to the volume's previous position, the sample rate is still high in this area and thus filter extents are kept small. The spots are barely (or not at all) blurred. For this very reason there are also dark spots within the chameleon's head. The guided sampler already concentrates rays on these regions but single spots are missed due to the randomisation. Our age filter, on the other hand, removes practically all spots around the snout. We designed our filter to target this kind of artefacts and it performs well. In equal measure, the dark spots in the inner regions are removed and the chameleon's head appears as one cleanly sampled area. The rather coarse (once per tile) choice by Dayal's algorithm cannot remove these spots.

Also consider the right side of the volume. As previously mentioned, the areas at the fringe are undersampled until the tiling is adapted. As our sampler currently does not support predictive tiling, one can clearly see rectangular dark areas around the fringe. With Dayal's filter these artefacts seem to be slightly worse. With same amount of samples our filter produces a cleaner image. Yet it requires less time than our implementation of Dayal's filter. Additionally our filter exhibits constant calculation times even during motions (chapter 5.1). However, neither of the filters succeeds to to remove bigger artefacts such as the out-dated areas around the upper part of the chameleon's head. Once more, the fringe of the volume poses additional problems.

Figure 5.10 shows different filters applied to a rapidly moving volume. In the right quarter of the image, there are clearly undersampled areas. Also, there are stale samples to the bottom left of the volume. The static Gaussian filter mitigates the negative effects. There are less stale samples visible and the undersampled areas seem to be more uniform in colour. Our implementation of Dayal's filter and the dynamic Gaussian filter (chapter 4.1.4) do not achieve the same mitigation. Note that the dynamic Gaussian filter is a simplified version of Dayal's filter. They both rely on the local sample rate to calculate filter size and apparently the adaptions happen too late. Furthermore, our implementation of Dayal's filter seems to make "holes" in undersampled areas more coarse. We assume that the calculation of local filter extents is problematic:

the filter always tries to balance spatial and temporal gradients. An up-to-date pixel at the border of a hole has roughly equal spatial and temporal gradients. An older pixel at this

location is probably black. Due to its location at the border it may be neighbouring with both black and blue samples. Hence, the filter extents lie spatially and temporally in the same magnitude. Due to the local sampling rate (within the tile), the filter is rather big. Consequently, the surrounding area but also older pixels are included into the calculation. Because the older samples are mostly black, the wholes appear bigger as the border of holes is darkened by the filter. This effect can obviously be mitigated by weighting older samples much less. This, on the other hand, causes artefacts with dynamic anti-aliasing (see chapter 5.3.3). For the pictures in figure 5.10 we used the weighting advocated in [DWWL05].

Our age filter suffers from neither constant blurring nor vast amount of outdated spots. Like all discussed filters, it fails on heavily undersampled areas, but generally succeeds to reconstruct images of volume data better than the other candidates.

5.3.3 Dynamic anti-aliasing by Dayal et al.

Anti-aliasing is commonly used to improve image quality. Dayal et al. propose deep buffering and a reconstruction filter to dynamically and locally enable anti-aliasing when possible. However, as we experiment with our implementation of their technique, it does not seem to perform as well. Consequently, we try to determine the problem by experimenting with modifications of the filter. In section 4.1.1 we describe the parameters which we manipulate. In the following we explain our findings.

Our renderer performs subpixel randomisation in order to fill the deep buffer with samples for anti-aliasing. However, when switched on in combination with Dayal's filter, the visual edges (transitions from permeable to dense areas) in our volume data sets exhibit temporal patterns. The colour seems to change permanently. Figure 5.11 shows the same image section thrice at different moments. The scene is static, yet areas around the edge differ slightly. In the renderer this effect is visible to the naked eye. The two images 5.11d and 5.11e show the differences between the images above. In uniform areas, the discrepancy are slight and not notable. Around the edges, however, there are strong differences over time. The difference images are visually enhanced and do not represent the actual difference in colour.

We experiment with various different weights for older samples and downsizing the range for possible subpixel locations. That is, the jitter of the random locations is smaller. We also apply regular sampling pattern rather than random locations, for example three distinct subpixel locations which are sampled one after another. However, sampling does not seem to be the issue here, the problem remains. The way samples are included into the reconstruction filter on the other hand does influence the pixel's colour temporally. We assume that samples at a slightly different pixel locations differ considerably more for volume data than for polygonal models. Figure 5.11f visualises the temporal gradient on a *static* scene, when sampled with subpixel randomisation. The transitions from low to high densities are clearly visible, as the temporal gradient is very high. Volume data is very fine-grained and additionally exhibits

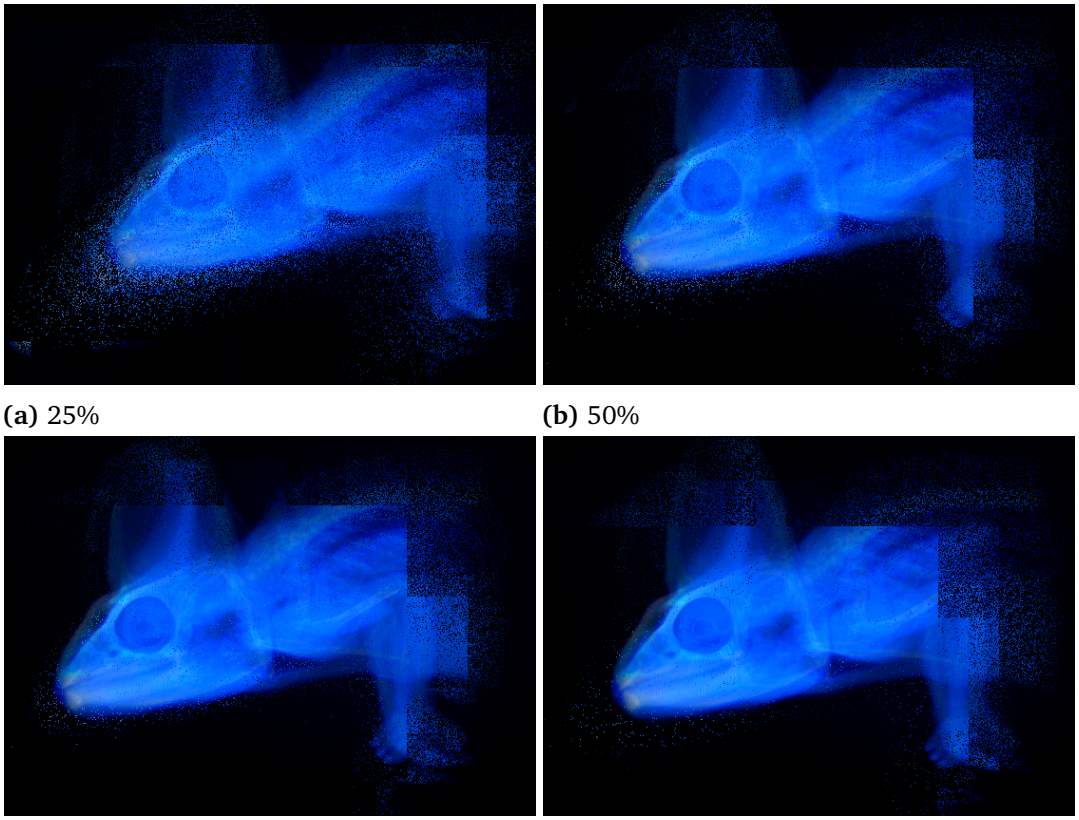
different properties when rays enter the volume at different angles. The filter described in [DWWL05] includes samples weighted negatively with their age. The most recent sample will thus influence the pixel's colour the most. Because new samples are continuously inserted into the buffer, the filter continuously receives different input. In each iteration a different subpixel location is weighted the most. This effect is then visible as continuously wafting edges. This even occurs when the sampling pattern is regular. However, we can limit the effect by applying regular sampling patterns and weighting all pixels alike.

In that case only the local filter depth may change over time and one can see only minimal billowing. However, weighting older samples as much as up-to-date ones decreases the filter's ability to adapt to changing scenes. It causes artefacts in tiles where the sampling rate is too low. As older artefacts influence the filter output more, it requires more time to converge into an up-to-date image after a change.

We identify that the weighting of samples is problematic for dynamic anti-aliasing. If the technique is to be applied, older samples must be weighted equally, or at least the difference to older samples must be low.

5.3.4 Hit filter

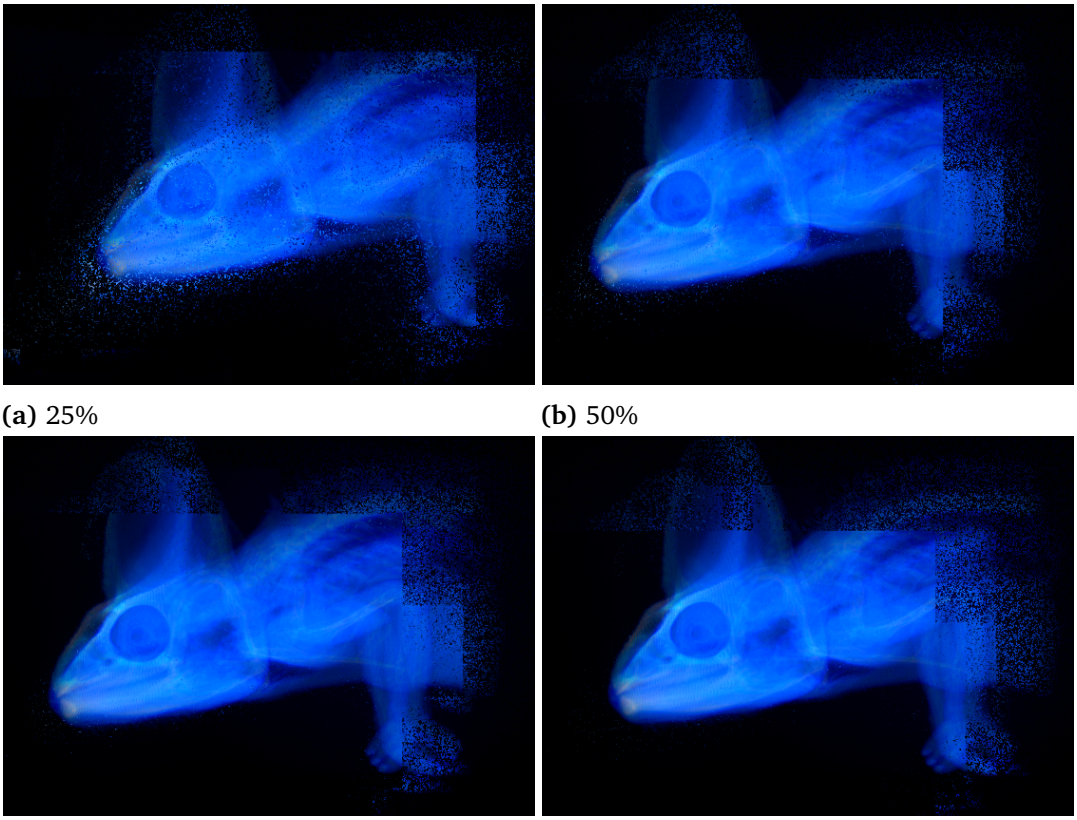
In chapter 4.3.1 we advocate a filter that hides many stale samples using hit checks and the k-d tree as feedback mechanism. If a tile is likely to cover only empty regions but still contains stale samples, they can be removed from the image plane by checking the according property in the k-d tree. The filter requires only very little computational time and can cheaply be applied. However, our experiments show, that using our age filter, the hit filter becomes almost redundant. Many of the stale samples are removed sufficiently fast by the age filter. It may however be of interest for other filtering techniques that do not respect age signatures. Figure 5.12 illustrates the redundancy.



(a) 25% **(b)** 50%

(c) 75% **(d)** 100%

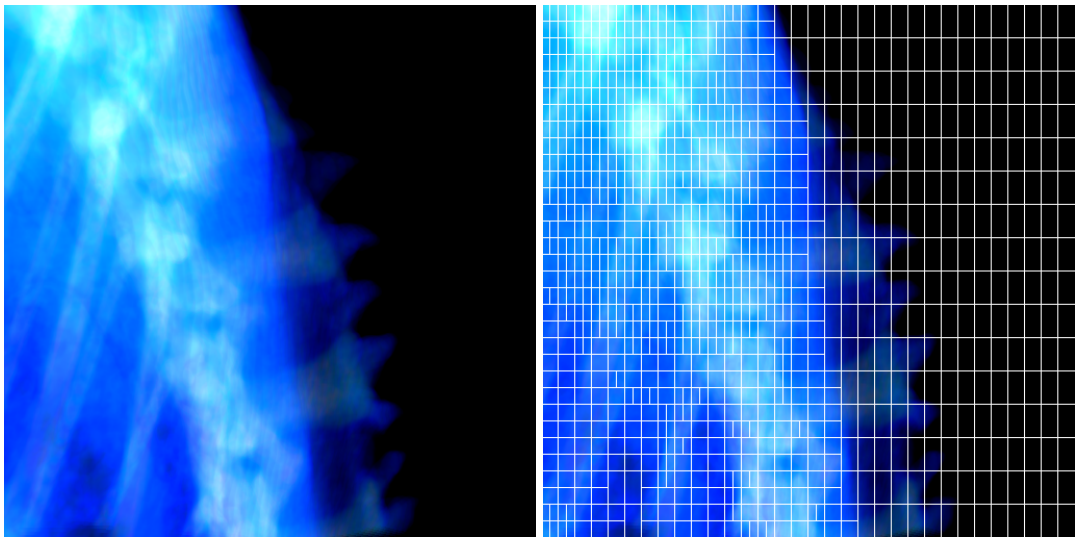
Figure 5.2: Raw buffer at different sampling rates showing a shifting movement. Around 50% the sampler performs well and distortions are vastly reduced. Depending on the filters the ratio may even drop below 50%. Ratios over 75% seem superfluous. The difference to 100% is not significant. Furthermore, the sampler's performance suffers from ratios close to 100% and likely performs worse than a framed sampler.



(a) 25% **(b)** 50%

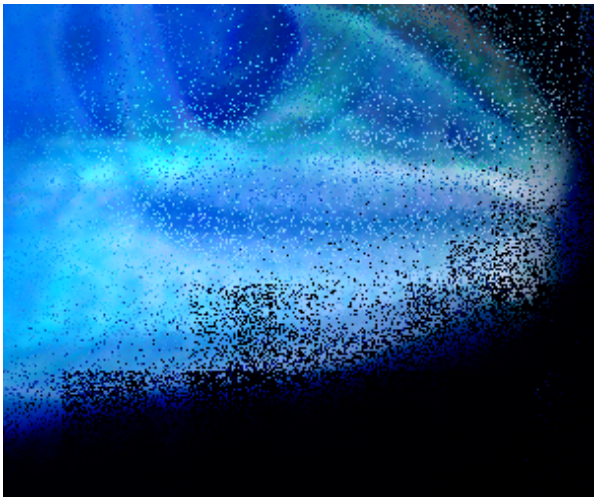
(c) 75% **(d)** 100%

Figure 5.3: Filtered with age filter. The difference between 75% and 100% is almost not notable, proving that sampling ratio should not be too high in adaptive samplers. Furthermore, by filtering, the quality of a 50% ratio is raised to the same magnitude as 75%. 25% however seems generally too low for volume data. Low sampling ratios increase both stale samples and delay for tiling adaptions.

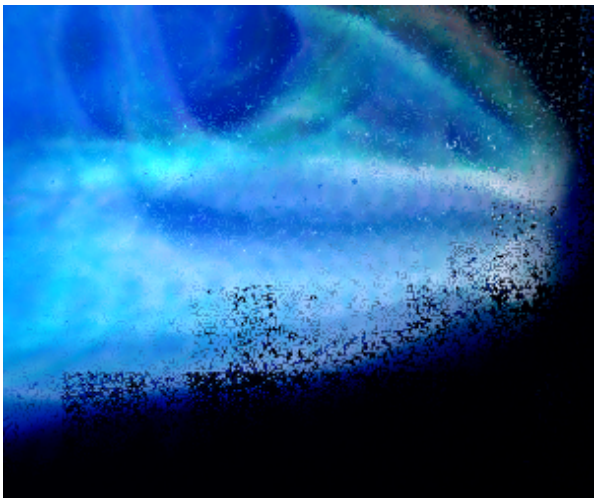


(a) Volume with differing density **(b)** Tiling guided by luminance variance

Figure 5.4: Small tiles covering both dense and permeable parts of the volume. Even if warps sample spatially close samples, threads may still idle because of local differences. Guided sampling attracts many rays to these exact locations.

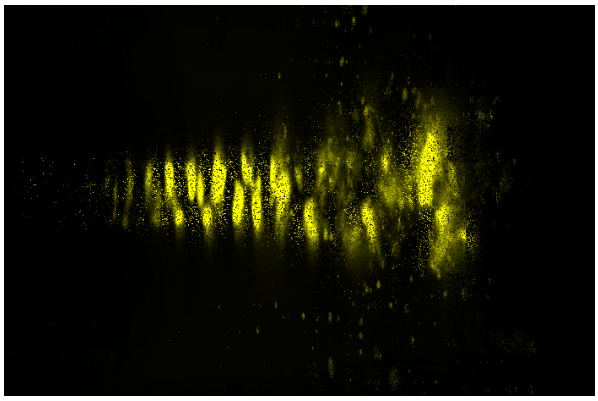


(a) undersampled tiles at the fringe

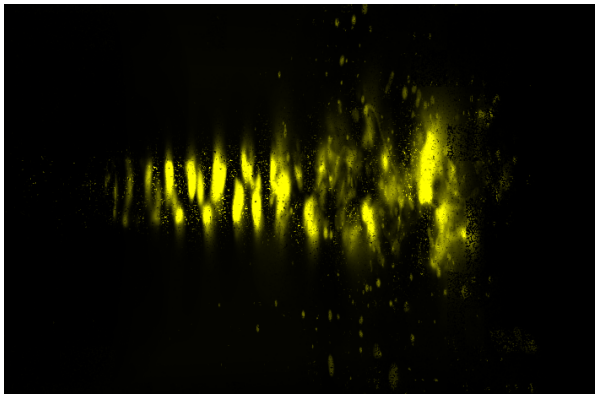


(b) Mitigation by age filtering

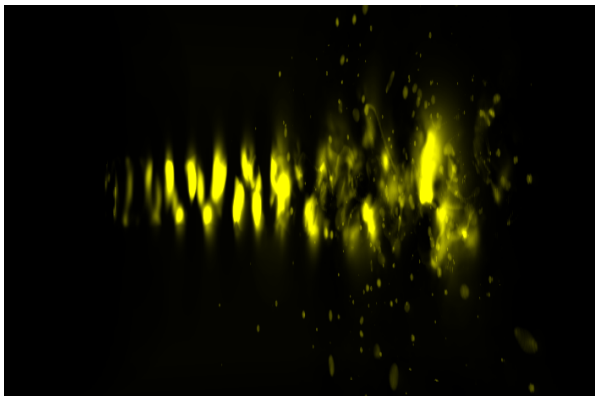
Figure 5.5: Undersampled fringe due to late k-d tree adaption. The effect can be mitigated by better tile assessment and our age filter. However, to eliminate this effect, the tiling must be adapted predictively.



(a) Raw buffer

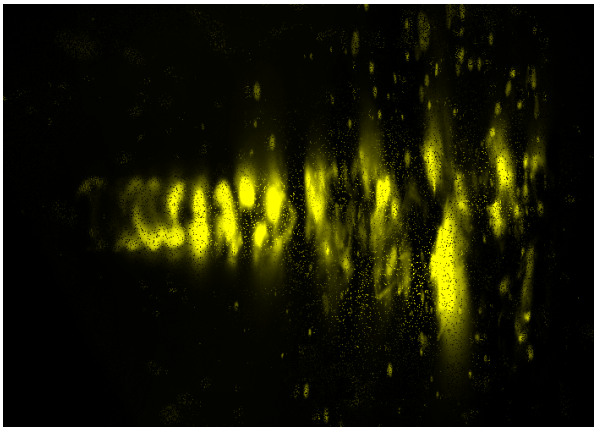


(b) Age filtered

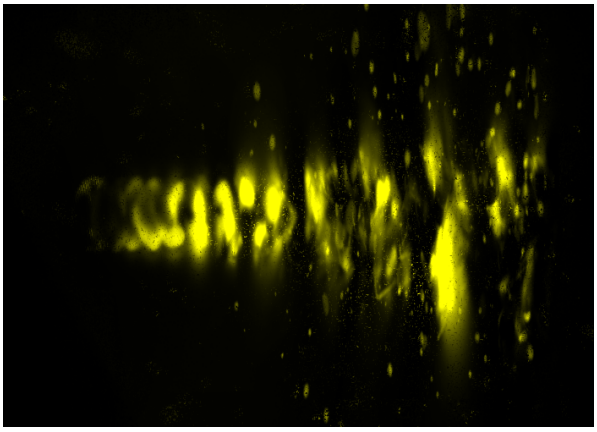


(c) Perfect image

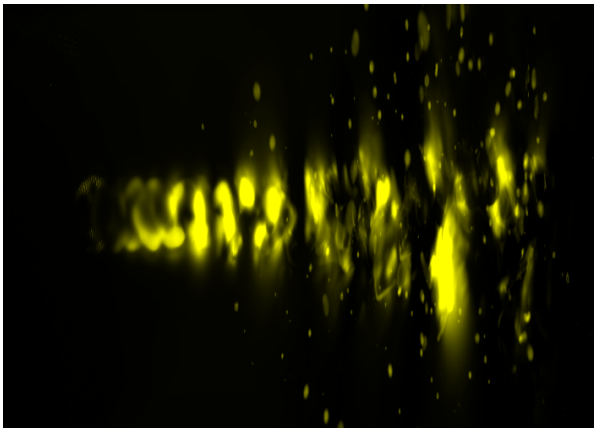
Figure 5.6: The volume is shifted to the right. Many stale samples are visible between the bigger drops of the volume and as dark spots on the drops as well. Our age filter removes many of the spots in between or at least conceals (darkens) them. The dark spots on the drops are almost entirely removed. Only few remain on the fringe.



(a) Raw buffer

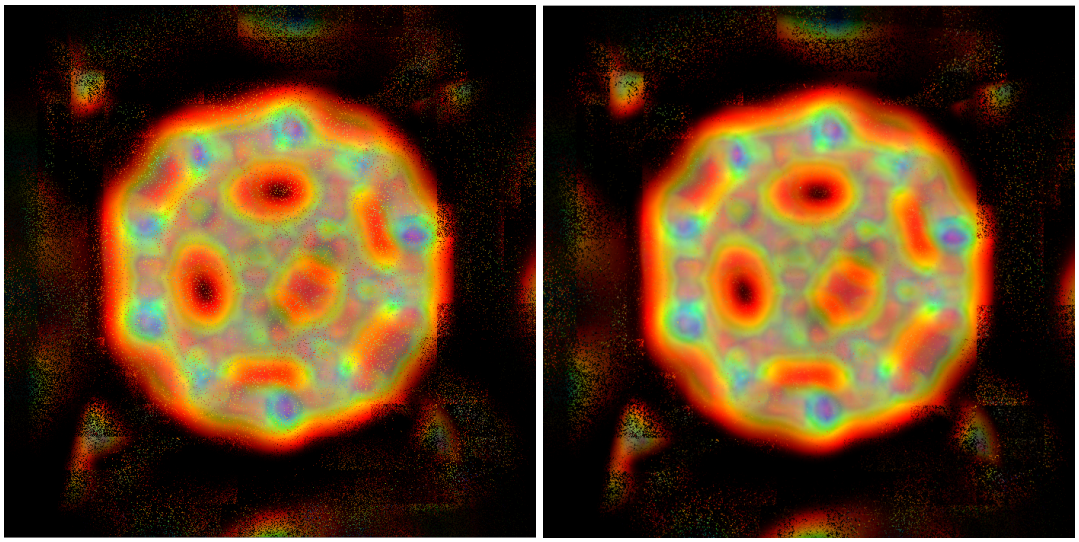


(b) Age filtered



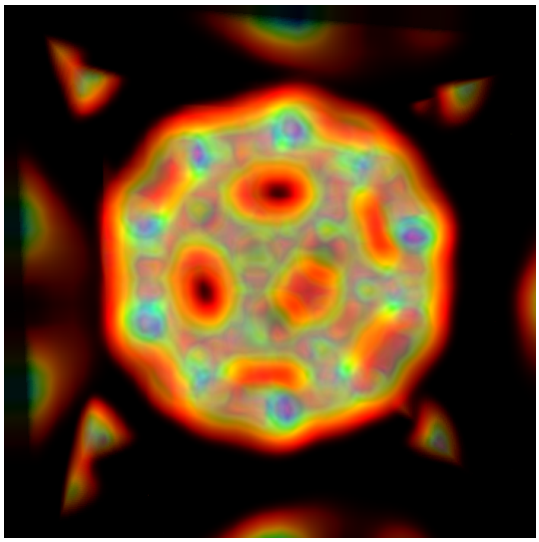
(c) Perfect image

Figure 5.7: Rotation around the y-axis. The filter improves the image quality considerably. Again almost all dark spots are removed. The filter also performs well with bright spots in between. Due to its targeted functioning it only adds about 3ms overhead to the reconstruction phase.

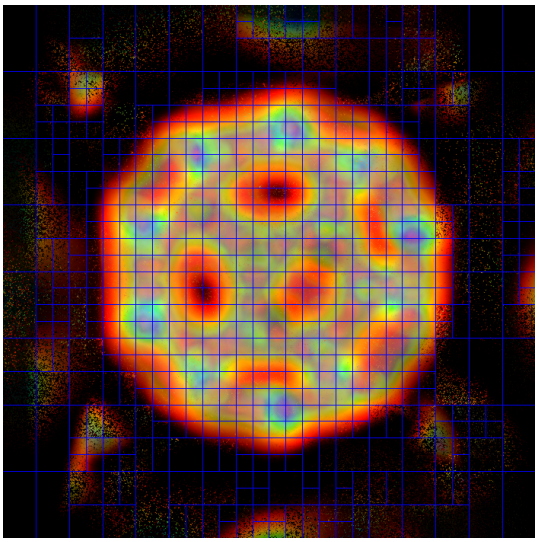


(a) Raw buffer

(b) Age filtered



(c) Perfect image



(d) k-d tree

Figure 5.8: A rotation around y-axis. The filter cleans the inner parts of the volume but fails in the outer regions. Image 5.8d shows the k-d tree in this very moment. The tiling indicates low sampling rates in the severely spotty areas.

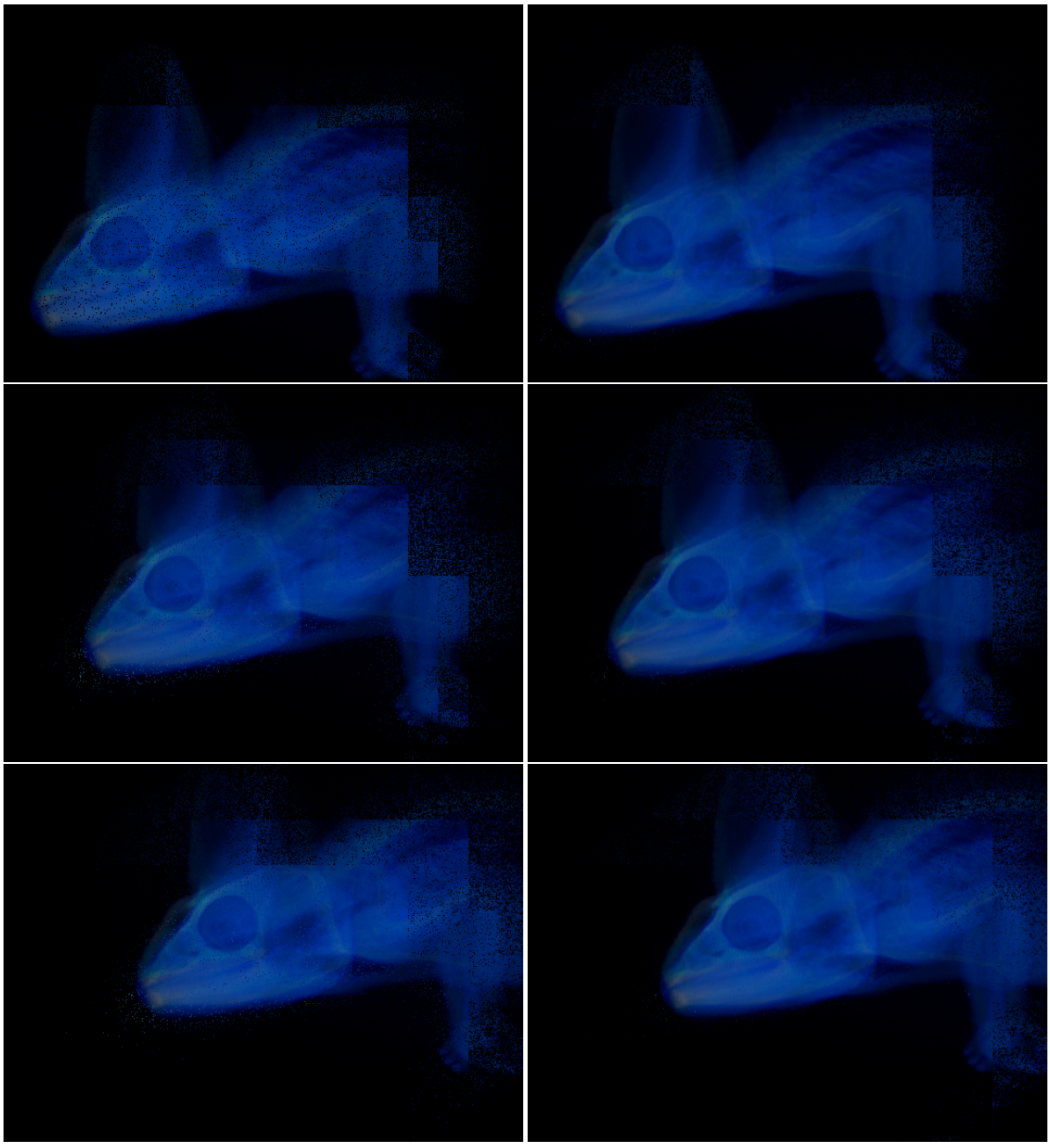
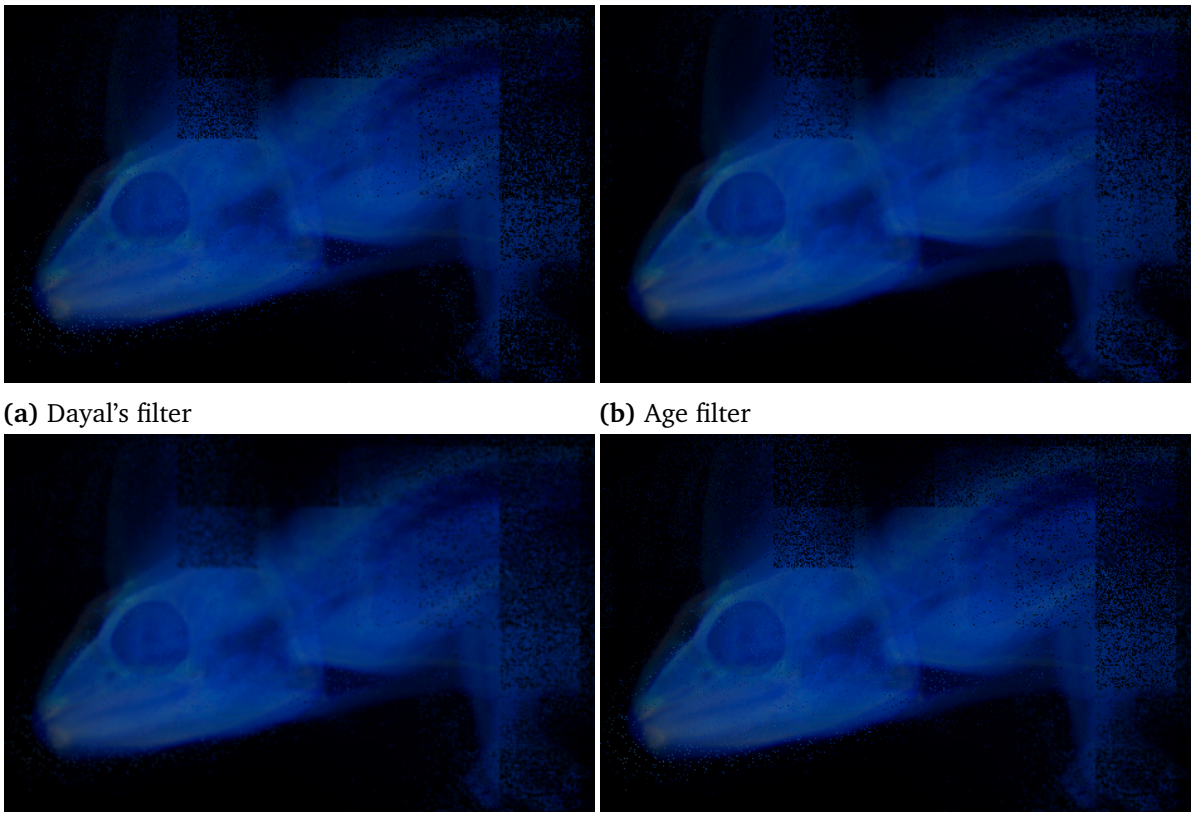


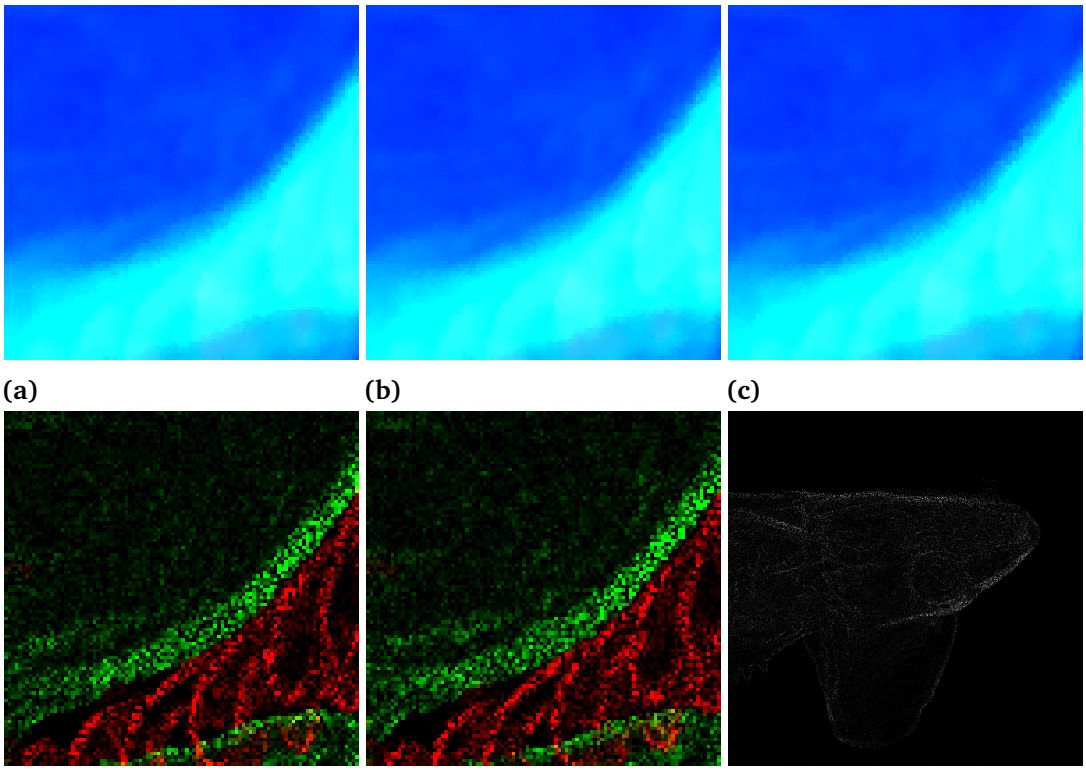
Figure 5.9: From top to bottom: movement of the volume to the upper right corner. Left: Dayal's filter, right: age filter. Dayal's filter leaves considerable amounts of stale samples on the bottom left which the age filter successfully removes. Furthermore, the inner regions look much cleaner when age filtered.



(a) Dayal's filter **(b)** Age filter

(c) Static Gaussian filter **(d)** Dynamic Gaussian

Figure 5.10: Effect of different filters on a quickly moving scene. Dayal's and the dynamic Gaussian filter both rely on per-tile sampling rates. But the measure adapts too late in order to guide filtering reasonably. The static Gaussian eliminates artefacts but also blurs the image permanently. Our age filter removes many artefacts while maintaining a focused image.



(a) **(b)** **(c)**

(d) diff 1/2 **(e)** diff 2/3 **(f)** temp. gradient

Figure 5.11: Exact same portion of the image plane on a static scene at three different moments. Images 5.11d and 5.11e show the difference between the images above (visually enhanced, i.e. adjusted brightness and saturation). The changing subpixel locations cause considerable temporal differences. Image 5.11f shows the temporal gradient of this scene. Even though the scene is static, the gradient is very high at the edges. Due to weighting, this causes ever changing colours of each pixel.

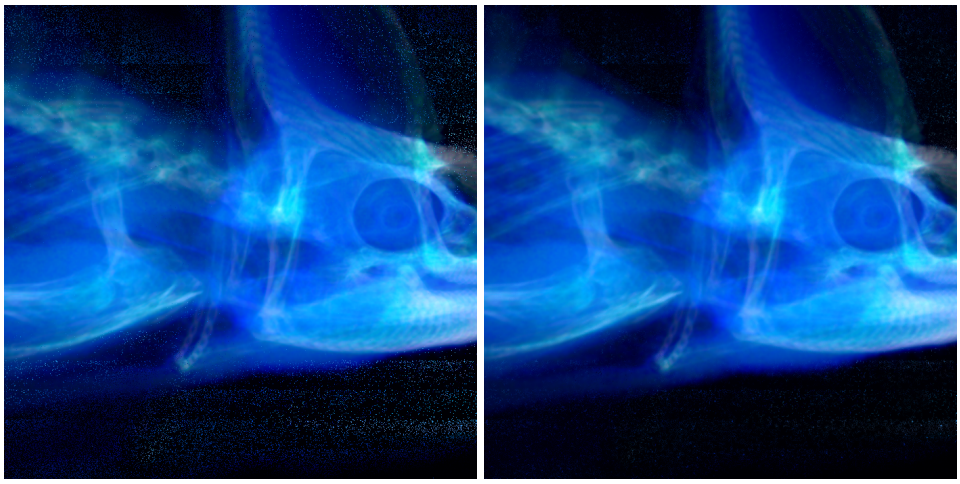


Figure 5.12: Both images show the scene after suddenly zooming out. The right image is filtered with our age filter. It is clearly visible that the age filter is to the hit filter, as most stale samples in empty areas are quickly concealed.

6 Summary and future work

In this thesis, we transfer the concepts of [DWWL05] to a volume renderer and examine their usability in this context. This includes the idea of guided sampling controlled with help of a k-d tree and tiling assessment based on colour variance. Furthermore, we adopt the concepts of filtering the incoherent buffer in order to reconstruct an image. We contribute detailed analysis of previous techniques within the context of volume rendering as well as our own new technique.

We provide a technical description of a frameless sampler with exploitation of parallel algorithms implemented on the GPU. Our implementation generally performs well. The leaf comparison (sorting) phase and tile assessment are weak spots that need to be addressed for productive use. As to sampler guidance, our examination shows that Dayal's techniques seems to perform well for volume data. The techniques do require fine adjustment however. The results vary with sample amounts and parameters for the k-d tree such as size limits for leaves. Furthermore, for big resolutions tile assessment becomes expensive. We propose estimations in order to keep the performance high. Further, we find that guided sampling often struggles with the volume's fringe. Tile assessment can only include samples from the past and thus an inevitable delay is caused. This delay then produces heavily undersampled areas at the fringe whenever a quick motion is performed. We propose predictive adjustments to the tiling based on user input.

Our implementation of previously described reconstruction filters does not perform as well as expected. We suspect slightly different properties of volume data as opposed to polygonal models to be the reason for that. The reconstruction filters may have been developed plains and surfaces in mind. The lack of such in volume data may worsen the results. Also, we assume that our implementation may differ from Dayal's, despite all efforts to close the gap. This shows how precisely concerted the techniques must be in order to perform well. Generally, blurring undersampled areas seems to be useful for temporary approximation. We show this by applying static blurring filters which mitigate artefacts vastly. But for practical use one must condone the negative effects, such as lack of detail. The filtering proposed by Dayal et al. tries to dynamically size filters with help of per-tile sampling rates. That way, the reconstruction may limit artefacts during motions and produce sharp, focused images on static scenes. But in our experiments, this measure seems to be too coarse. Also, it is not adapted immediately but produces a delay and thus misses many single outdated spots that occur directly after or during a motion.

To tackle these issues, we contribute a filter which targets these specific artefacts. It can both remove stale samples outside the volume and "fill in" samples where data is still missing. It targets outdated spots precisely by examining the age signature of samples and their surrounding. As all approximations it is only a compromise which works well in some scenarios and fails to provide the wanted quality in others. Thus, in heavily undersampled areas, the filter reaches its limits. However, if the sampler is able to prevent massive lack of samples, the filter performs well and produces clean and undistorted images in cases where Dayal's filter fails to do so. Our filter requires a constant and rather low overhead and thus is applicable for interactive renderers. Its runtime is superior to Dayal's filter, which inherently exhibits varying runtimes depending on the scene properties. Our filter is unaffected by motion and other scene changes.

Dynamic anti-aliasing with Dayal's filter seems to struggle with weights for older samples and the very fine grained structures of volume data. We provide detailed description of our experiments with this feature. We find that only regular subpixel patterns in combination with virtually equal weights for older samples can mitigate the problems. We assume that dynamic anti-aliasing for volume data requires a new approach different from Dayal's filter.

Future work may include improvements to guided sampling. Our work shows that the principle works. However, it exhibits weak spots at the fringe of the volume as well as with quick movements. Sorting tiles in order to determine such with lowest and highest colour variance requires too much time. The phase may either be improved so that it performs faster, or replaced by an algorithm to find the respective tiles. We explore reconstruction techniques that have successfully been used with polygonal data. Our modifications to the filters do not provide the wanted improvement. Different adaptions and more thorough exploration of the possibilities may be necessary in the future. Our age filter performs well in some cases but fails in others. Combination of the filter with other techniques may improve the quality. However, the interaction between both filters must be sophisticated in order not to produce additional artefacts. Furthermore, the dynamic anti-aliasing might require reevaluation for volume data. Taking the properties into account, it may be implemented successfully but might require a different approach for inclusion of older samples. In addition, future work may include research for omnipresent problems, such as ever growing display resolutions. Complexity for ray casters increases with higher amounts of pixels. New algorithms and techniques like frameless rendering and adaptive sampling may face new challenges when operated on a large scale.

Bibliography

[BFMZ94] G. Bishop, H. Fuchs, L. McMillan, E. J. S. Zagier. Frameless Rendering: Double Buffering Considered Harmful, 1994. (Cited on pages 14, 15, 29, 31, 37 and 43)

[DWWL05] A. Dayal, C. Woolley, B. Watson, D. Luebke. Adaptive frameless rendering. In *IN RENDERING TECHNIQUES*, pp. 265–275. Springer-Verlag, 2005. (Cited on pages 14, 15, 16, 17, 25, 29, 31, 32, 34, 37, 38, 41, 42, 43, 55, 58, 60, 63, 64 and 77)

[FSK13] T. Fogal, A. Schiewe, J. Kruger. An analysis of scalable GPU-based ray-guided volume rendering. In *Large-Scale Data Analysis and Visualization (LDAV), 2013 IEEE Symposium on*, pp. 43–51. 2013. doi:10.1109/LDAV.2013.6675157. (Cited on pages 22 and 23)

[IBH11] T. Ize, C. Brownlee, C. D. Hansen. Real-time Ray Tracer for Visualizing Massive Models on a Cluster. In *Proceedings of the 11th Eurographics Conference on Parallel Graphics and Visualization*, EGPGV '11, pp. 61–69. Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, 2011. doi:10.2312/EGPGV/EGPGV11/061-069. URL http://dx.doi.org/10.2312/EGPGV/EGPGV11/061-069. (Cited on page 11)

[Lev90] M. Levoy. Volume rendering by adaptive refinement. *The Visual Computer*, 6(1):2–7, 1990. doi:10.1007/BF01902624. URL http://dx.doi.org/10.1007/BF01902624. (Cited on page 15)

[WDP99] B. Walter, G. Drettakis, S. Parker. Interactive Rendering using the Render Cache. In D. Lischinski, G. Larson, editors, *Rendering Techniques' 99*, Eurographics, pp. 19–30. Springer Vienna, 1999. doi:10.1007/978-3-7091-6809-7_3. URL http://dx.doi.org/10.1007/978-3-7091-6809-7_3. (Cited on pages 16 and 60)

[WPS+03] I. Wald, T. J. Purcell, J. Schmittler, C. Benthin, P. Slusallek. Realtime ray tracing and its use for interactive global illumination. *Eurographics State of the Art Reports*, 1(3):5, 2003. (Cited on page 12)

[Zag96] E. J. S. Zagier. Defining and refining frameless rendering. Technical report, 1996. (Cited on page 16)

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature