FAKULTÄT FÜR !NFORMATIK

High-performance GPU based Rendering for Real-Time, rigid 2D/3D-Image Registration in Radiation Oncology

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering/Internet Computing

eingereicht von

Jakob Spörk
Matrikelnummer 0726567

an der
Fakultät für Informatik der Technischen Universität Wien und am
Zentrum für Medizinische Physik und Biomedizinische Technik
der Medizinischen Universität Wien

Betreuung:
Betreuer: Ao. Univ.-Prof. Dipl.-Ing. Dr.techn. Eduard Gröller
Ao. Univ.-Prof. Mag. Dr. Wolfgang Birkfellner
Mitwirkung: Univ.Ass. Dipl.-Ing. Dr.techn. Stefan Bruckner
Support: Austrian Science Foundation FWF, Projects P19931, L503

Wien, 22.01.2010

(Unterschrift Verfasser/in) (Unterschrift Betreuer/in)

Erklärung zur Verfassung der Arbeit

"Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit — einschließlich Tabellen, Karten und Abbildungen —, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe."

Ort	Datum	Unterschrift

Abstract

This thesis presents a comparison of high-speed rendering algorithms for the application in 2D/3D-image registration in radiation oncology. Image guided radiation therapy (IGRT) is a technique for improving the treatment of cancer with ionizing radiation by adapting the treatment plan to the current situation using 2D/3D-image registration. To accelerate this procedure, also rendering of Digitally Rendered Radiographs (DRR), as a part of it, has to be performed faster. In the context of this thesis, a GPU based wobbled splat rendering algorithm based on the work of Spoerk et al. (2007) is further improved and compared to a newly developed GPU based ray casting implementation. For comparison, rendering quality and performance as well as the influence on the quality and performance of the overall registration process are measured and analyzed in detail. These results show that both methods are competitive although ray casting has in its current implementation several advantages over wobbled splat rendering.

Kurzfassung

Diese Arbeit präsentiert einen Vergleich verschiedener Hochgeschwindigkeits-Rendering Verfahren für die Anwendung in der 2D/3D Bildregistration im Bereich der Strahlentherapie. Image Guided Radiation Therapy (IGRT) ist eine Technik, welche die Krebsbehandlung mit ionisierender Strahlung dahingehend verbessert, dass im Vorhinein erstellte Behandlungspläne auf die aktuelle Situation während der Behandlung abstimmt werden. Dazu wird 2D/3D Bildregistration eingesetzt. Um diese zu beschleunigen, hat auch das Rendering von Digitally Rendered Radiographs (DRR) schneller zu erfolgen. Diese Arbeit entwickelt das GPU basierte Wobbled Splat Rendering Verfahren von Spoerk et al. (2007) weiter und vergleicht dieses mit einem neu entwickelten GPU basierten Ray Casting Verfahren. Dazu wird die Qualität der DRRs und die Leistungsfähigkeit der Verfahren sowie der Einfluss der unterschiedlichen Renderingverfahren auf den gesamten Registrationsprozess gemessen und im Detail analysiert. Die Resultate dieser Messungen zeigen, dass beide Verfahren zwar konkurrenzfähig sind, Ray Casting jedoch in seiner aktuellen Implementierung einige Vorteile gegenüber Wobbled Splat Rendering aufweist.

Table of Contents

1 Introduction

Today, computer graphics and visualization have emerged from a high art of some experts to widespread disciplines. They are essential tools for nearly all scientific fields. Many techniques were developed and improved over the years and visualization gets more and more important for all kinds of application.

Medicine is one of these fields. Computer graphics and visualization allow to make information about the body visible, that would otherwise be unaccessible for doctors. The development of computed tomography (CT) and magnet resonance imaging (MRI) are just two examples of standard diagnostic devices that are used today and that would not have been possible without computer graphics. Visualization has emerged to be the most important aspect of modern diagnostic medicine.

Also for therapeutic medicine, computer graphics and visualization gets more and more important as a tool to support doctors or to allow higher precision and better results. Most techniques can be summed up as image guided surgery where a surgeon uses imaging devices to improve his navigation and to decrease the trauma associated to surgery.

Also in oncology, imaging gets more and more important. One major procedure for treating cancer is radiation therapy. Malignant tissue is destroyed by means of ionizing radiation. To be sure that the tumor is completely destroyed, also a high dose of radiation has to be applied to surrounding, healthy tissues. Uncertainties in the planning of such a treatment and movements of the patient as well as of the tumor inside of the body imply that more dose than required is delivered to healthy tissue.

To improve the dose delivery by applying more dose to malignant and less dose to healthy tissue, many techniques were developed. One of these techniques is image guided radiation therapy (IGRT). Imaging modalities are used to localize the tumor before every treatment session and radiation plans are adopted to the new situation. Therefore, an image of the patient has to be aligned with the planning data in order to transfer the coordinate system of the plan to the current situation. This procedure is called 2D/3D-image registration.

These techniques allow to react to day by day movements of the tumor, normally a result of the state of nearby organs or on changes of the tumor shape. One problem that cannot be solved today is the compensation of local movements caused by for example the heartbeat/pulse, breathing or muscles.

To improve the situation, a project at the center for medical physics and biomedical engineering (CBMTP) with support from the Austrian Science Foundation (FWF) was started in 2008 to optimize and enhance a 2D/3D-image registration software suite that shall allow to perform such registrations in real-time. This should enable doctors to detect movements online and

compensate them in order to optimize the dose delivery.

The goal of this thesis is to develop a high-performance rendering module utilizing the parallel computation power of modern graphics hardware. This rendering module shall be integrated into the existing software suite for rigid 2D/3D-image registration. As a basis, the GPU based wobbled splat rendering algorithm of Spoerk et al. [1] shall be further improved and compared to an implementation of GPU based ray casting with several optimization techniques to receive a wide and well analyzed portfolio of rendering procedures that can be used to achieve the medium-term goal of real-time 2D/3D-image registration.

1.1 Structure of this thesis

This thesis is divided into a theoretical and a practical part. The theoretical part gives an introduction to the domain and the technical aspects of this work. It starts by presenting an overview to radiation therapy (see Sect. 2.1). The process of 2D/3D-image registration is at first presented in general (see Sect. 2.2) and than the three parts of this procedure — DRR computation (see Sect. 2.3), merit function computation (see Sect. 2.4) and optimization (see Sect. 2.5) — are discussed in detail. This includes also a presentation of techniques that are used to optimize the rendering process (see Sect. 2.6 and 2.7). Finally, Sect. 2.8 gives information about graphics hardware to introduce the reader to this computation platform.

The second section of the theoretical part of this thesis (see Sect. 3) gives an introduction to the project this thesis is part of. Therefore, the status of July 2009 of the project is analyzed and the goals and current topics are discussed.

Sect. 4 presents methods for improving the DRR rendering and is the first section of the practical part of this thesis. Improvements for Wobbled Splat Rendering and the ray casting algorithm as well as optimization techniques for them are presented and described in detail.

Sect. 5 provides a lot of uninterpreted data about the quality and performance of the different developed rendering approaches as well as about the influences of these approaches on the quality and performance of the overall 2D/3D-image registration process.

Finally, Sect. 6 and 7 interpret the results from Sect. 5, discuss them in detail and conclude with an outlook on topics for future work in this field.

1.2 Acknowledgment

Special thanks go to Ao. Univ.-Prof. Dipl.-Ing. Dr.techn. Eduard Gröller and Ao. Univ.-Prof. Mag. Dr. Wolfgang Birkfellner for the supervision of this thesis.

The project team of the project presented in Sect. 3 was a big help while working on the practical part of this thesis. They helped to achieve a seamless integration of the developments into the

existing software suite. Therefore, special thanks go to Christelle Gendrin and Christoph Weber for the good team work and also to Christoph Bloch, Supriyanto Ardjo Pawiro, Daniella Fabri, Hugo Furtado, Michael Figl, Markus Stock and Dietmar Georg for many impulses and helpful suggestions.

The major part of the work has been done in the rooms and with the equipment of the Center for Biomedical Engineering and Physics of the Medical University of Vienna which was a big help for me and for what I want to thank. Additionally, I want to thank the Austrian Science Foundation FWF for funding the project this thesis is part of.

2 Theoretical Background

This section defines the theoretical background of the thesis and will concentrate on the following core aspects:

Radiation Therapy is the domain of this thesis — see Sect. 2.1. Radiation therapy is one possible treatment for cancer by using ionizing radiation to destroy malignant neoplasms [2]. The two other widespread cancer treatments are surgery and chemotherapy.

2D/3D-Image Registration is the topic of the project where this thesis is part of — see Sect. 2.2. 2D/3D-Image Registration describes the process of finding the correct affine transformation of a volume so that a perspective rendering of the volume in this position produces an image identical (or best fitting) to a 2D reference image [3, 4, 5].

Volume Rendering Techniques for Medical Data are the first step in the 2D/3D-image registration process and is the core topic of the practical part of this thesis — see Sect. 2.3. To compare the volume's position to a 2D image, a perspective rendering of the volume is required. In the domain of radiation therapy, so called Digitally Rendered Radiographs (DRR) are generated that are virtual radiographs computed of 3D volume data produced by a Computed Tomography (CT) scanner.

Merit Functions are the next important step in the 2D/3D-image registration process — see Sect. 2.4. These functions allow to calculate a value that describe how "similar" two images (or in general data and the fitting model) are [6].

Optimization Algorithms are the final step in the 2D/3D-image registration process — see Sect. 2.5. These algorithms iteratively change the rotation and translation parameters of the volume to optimize the value of the merit function in order to align the projection of the volume to the reference image.

Sparse Sampling of Image Data is a technique to improve the performance of 2D/3D-image registration — see Sect. 2.6. Zöllei et al. [7] and Birkfellner et al. [3] show that not all image data is required to perform precise image registration using intensity based merit functions. Therefore, sparse sampling the image data allows to increase the performance of 2D/3D-image registration.

Direct Volume to Surface Transformation is a process that can be used to further speedup ray casting as one rendering technique — see Sect. 2.7. A surface representation of the windowed volume data can be used to define a more exact bounding structure of a volume and therefore helps to skip empty space during ray casting. This optimization technique is described by Scharsach [8]. Because the transformation from volume data to surface data is complex, this is described in a separate section.

Graphics Hardware and General-Purpose Computation on Graphics Processing Units are the core technologies to implement DRR generation in a fast and efficient way — see Sect. 2.8. Using the GPU as additional computation source allows to boost overall performance for rendering [5, 1, 8] as well as for non rendering tasks [5, 9, 10] due to the high parallel design of such hardware.

2.1 Radiation Therapy

Radiation therapy (sometimes also called Radiotherapy or Radio-oncology) describes the treatment of cancer with the help of ionizing radiation. It was first used shortly after the discovery of x-ray radiation by Wilhelm Conrad Röntgen in 1895 [11]. The first documented application of radiation therapy was the treatment of a skin disease by Leopold Freund, an Austrian scientist who lived in Vienna. Beside cancer treatment, ionizing radiation is also used to treat several other diseases, especially chronic-inflammatory or degenerating diseases [12].

Many different radiation types have the ability of damaging cells, for example, infrared or ultraviolet light or microwave radiation. Radiation therapy only uses ionizing radiation for the treatment. Today, several types of such radiation are used because every type has different properties and therefore different fields of application [2]:

High-energy electro-magnetic or indirect ionizing radiation These radiation types (especially x-ray and gamma radiation) are the most used types. They are produced with linear accelerators (LINAC) and have good properties for the treatment of moderately deep located tumors. The effect that destroys the malignant cells is direct radiation as well as indirect effects like bremsstrahlung. Direct effects influence the molecules of the cells and destroy them directly. Indirect effects ionize the water and lead to the creation of free radicals that have a very high toxicity. These radicals lead to double-strand breaks in the DNA that cannot be repaired by the cell or even let the cell enter apoptosis [2]. High energetic electro-magnetic radiation has its dose maximum not at the skin, but 20–40 mm below it [13]. This helps to protect the skin of damage. If the skin itself shall be the target of the treatment, a bolus called object has to be placed above the tumor to "focus" at the skin.

Particle or direct ionizing Radiation This radiation (especially charged particles such as electrons, protons or heavy ions) has nearly the same effect as x-ray or gamma radiation, but a different depth-dose-profile. Electrons can only penetrate to a depth of half the acceleration voltage in millimeters. Therefore, electron radiation is good if the malignant structure is in front of a very sensible structure, because after a short distance, only a little dose is left. As a result of this, electron radiation therapy is often used intra-operatively, which has the additional advantages that the tumor area can be visualized and directly treated and radio-sensitive normal tissue can be shielded [14]. Larger charged particles are not

widespread in use, because they require large, complex and therefore expensive equipment to be produced. Their depth-profile is very well suited for deep structures, because they have their maximum effect shortly before they are stopped (the so called Bragg-effect — see Fig. 1) but then emit all their energy at once without injuring deeper tissue [13]. This type of radiation therefore spares healthy tissue in front and behind of the target volume.

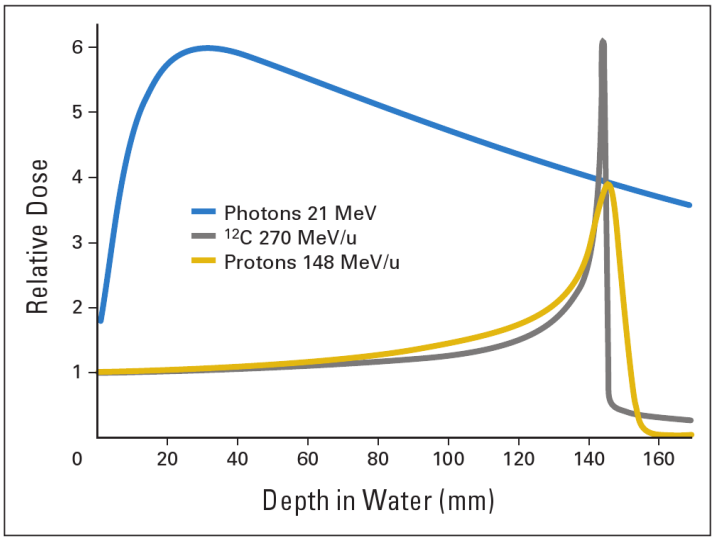

Fig. 1: Depth-dose-profile of photon, proton and heavy-ion radiation [13]. Photon radiation has its maximum dose after a short distance and decreases slowly over the whole distance while proton as well as heavy-ion radiation have their maximum dose at a deeper position and decrease then nearly immediately to zero. These radiations also have a low dose until shortly before the optimum. This effect is called Bragg-effect.

In general, radiation therapy is differentiated in teletherapy and brachytherapy. Teletherapy describes the application of radiation by an external linear accelerator through the skin and other healthy tissues. To protect these healthy parts of the patient, radiation is applied from different directions and the volume that has to be treated is carefully planned in a preprocessing step [15] — see Fig. 2. Therefore, the tumor volume is segmented in CT or MRI data and this information is used to calculate the dose that is applied to the target to prevent healthy tissue of being overdosed and malignant tissue of being underdosed. It is also possible to define the dose at different target areas and to inverse the planned setup for the radiation therapy.

To allow to define the best possible treatment plans, radiation therapists have to be able to adapt the radiation beam to the tumor shape and to modulate the intensities at different areas of this shape. This is called Intensity-Modulated Radiation Therapy (IMRT) [16]. For this, modern systems allow to bring lead apertures into the radiation path to change the shape of its beam. Multileaf collimators (MLC) allow to define very complex beam shapes. They consist of several "leafs" that can be positioned individually. Furthermore, modern systems allow to modulate the intensities inside the field by using objects of differing thickness that can be positioned in

the radiation path or through variation of the shape over time. Although IMRT allows a better definition of the target volume, Hall [16] shows that it may also increase the likelihood of serious spillovers. For this technique many control images are required for which more radiation dose is delivered to the whole body of the patient. Also the use of more radiation angles result in low dose being applied to more body regions.

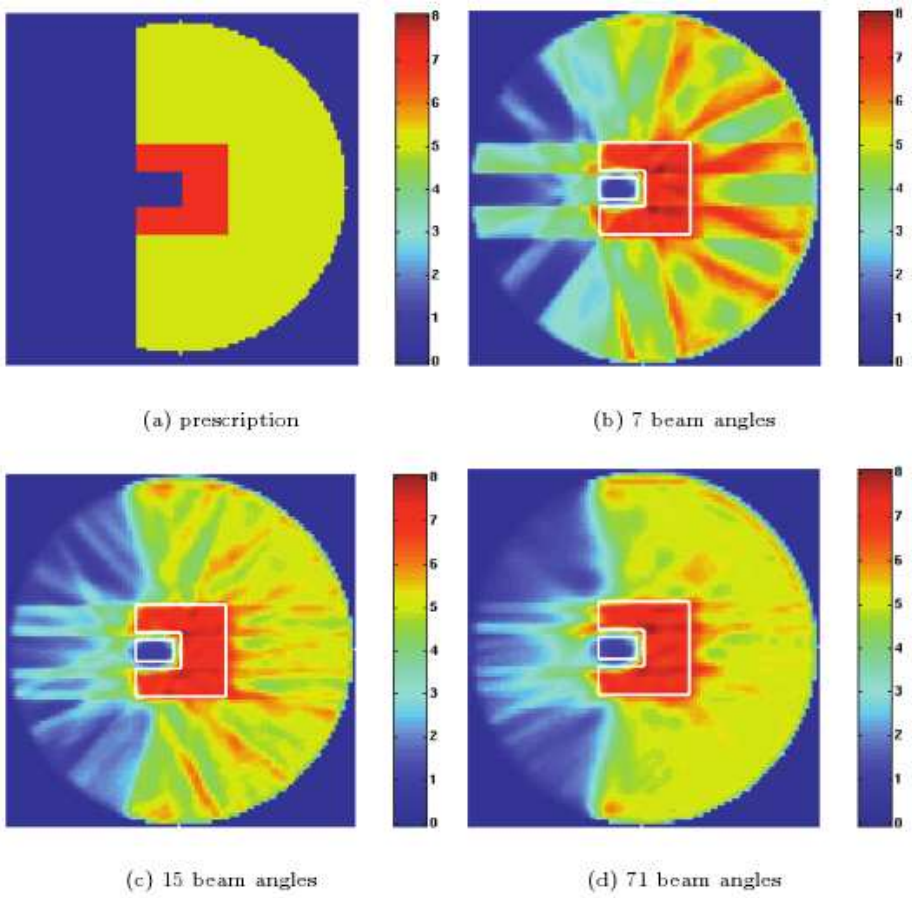

Fig. 2: Schematic illustration of the process of radiation therapy planning and different methods that help to fulfill complex treatment plans [15]. (a) shows the plan with blue being volume sensible to radiation, red the target volume and yellow normal volume. (b), (c) and (d) show how this plan can be realized using 7, 15 or 71 different angles during the treatment.

ps

Brachytherapy describes the application of radiation with small sources that are injected into a tumor or placed inside of cavities such as the uterus or the trachea [11]. There are two ways to do this. Using the afterloading technique, objects containing radiation emitting material are inserted into the tumor or to a cavity near to the tumor for a specified time (in the magnitude of 10–400 hours [17]) and are then removed afterwards. Using the implantation technique, the object with the radiation emitting material is implanted permanently but the radiation source loses its effect after some time. The duration of the material emitting radiation as well as of

seed remaining in the body depends on the type of disease and can be specified by using specific materials [17].

Another important technique in combination with radiation therapy is fractionation. The dose is not applied at once but in several treatments during a period of weeks [2]. This has the effect that a higher dose can be applied to the target volume. This is possible because healthy tissues regenerate faster from the irradiation by the radiation than malignant tissues [2]. Radiation therapy is also combinable with other treatments such as surgery (before the surgery to shrink the tumor or after the surgery to destroy remaining cells) or chemotherapy. In some situations, tumors can be very resistant to radiation. These tumors are often treated with hyperthermia. High temperatures have also the effect of destroying the malignant tissue.

Radiation therapy can lead to several spillovers. Immediately after the therapy, it is possible that damages of the epithelial surface, general inflammations with swellings or infertility occur. More than three months after the therapy, the patient may suffer from scars, hair loss, dryness or fatigue. It is of course also possible that cancer or the death of the patient is the result of the treatment. The late spillovers can accumulate when several radiation therapies are made and therefore careful planning is even more important. At last, stochastic spillovers are possible. These have no correlation to the dose or the duration of the treatment and can occur anytime. Stochastic spillovers are tumors or damages to the germline that may result in birth deformations or damages of the following generations [16].

2.1.1 Image Guided Radiation Therapy (IGRT) and other Improvement Techniques

Different extensions to radiation therapy exist that shall allow to, on the one hand, reduce the dose delivered to healthy tissue and on the other hand allow easier positioning of the patient under the radiation therapy device [16, 18, 19].

In general, therapy planning is done before the first treatment. This data is then used to perform the actual treatment. To align the treatment plan with the real situation during the treatment, the patient's pose has to be exactly the same as when recording the planning dataset. Several strategies exist for this purpose. One method is to have a CT and a radiation therapy device that allow identical positioning of the patient with the help of fixation of the patient [11]. Another widespread method is the use of surface ink markers [11]. With this method, the planned field shapes and other information is marked onto the skin of the patient. The radiation device then has a targeting system that allows to orient the radiation source on these ink marks in order to adjust the beam's shape. The drawback of these techniques is that they require to set the planned target volume (PTV) larger than the tumor to compensate geometric uncertainties, targeting uncertainties and small movements. This leads to a higher dose delivery to healthy tissue and/or to less dose delivered to the malignant tissue [19].

A more sophisticated method for patient pose verification is the so called image guided radiation therapy (IGRT) that adds the use of diagnostic image data [11]. Most modern linear accelerators offer besides the radiation device itself an Electronic Portal Imaging (EPI) detector for generating mega-voltage images, or a separated kilo-voltage imaging device to produce images of the patient while he/she is in the treatment device. These images can be used as reference images for 2D/3D-image registration to estimate the patient pose and align the treatment plan precisely to the current situation. The more exact this step is, the smaller the planned target volume can be and the less dose is delivered to healthy tissues. Another advantage of IGRT is that with this technique, it is also possible to adapt the treatment plan to changes of the tumor shape and size, often occurring during treatment because of changes in other internal structures (e.g. filling of bladder or intestine) or as result of the treatment itself. Due to radiation, the malignant tissue often shrinks by itself. The goal of IGRT is to react even on movements because of the heartbeat/pulse or breathing. This so called on-line monitoring of the tumor position is today only possible if implanted markers are used [11]. This is often done during the treatment of prostate tumors, but requires the surgical step of implanting the markers, which is time and money demanding and an additional risk for the patient.

Birkfellner et al. try with their current project to remove this drawback of on-line IGRT by improving the 2D/3D-image registration process to allow a full registration within 200 ms. More details about this project, which is the context of this thesis, can be found in Sect. 3.

Another technique for optimizing the dose delivery in radiation therapy is active breathing control (ABC) as described in [11, 18]. This technique can be applied for the treatment of thoracal or abdominal cancers. When treating these diseases, breathing produces many geometric uncertainties and therefore large margins around the planned target volume are required. To reduce these margins, the breath of the patient is actively held for short periods of time at specific moments in the breathing cycle to fix the position and optimize the dose delivery.

2.1.2 Sources and Apertures for Radiation Therapy

During the 20th century, so called radioisotope sources (cesium or cobalt cannons) were a common radiation source for radiation therapy [11]. Radioisotope sources are devices that produce in contrast to x-ray sources radiation all the time. They are based on cesium or cobalt as radiation source and their advantage is the relative reliability and the simple maintenance. Because of security problems and several accidents in the past, these devices are more and more replaced by so called linear accelerators.

The most widely-used (photon) radiation sources for radiation therapy are external beam linear accelerators (LINAC) [11]. They are typically similar looking to x-ray sources but produce radiation of much higher energy that is used for treatment instead of diagnostics. A LINAC system (the Elekta Synergy IGRT System at the Department for Radiation Therapy at the

Medical University Vienna) is shown in Fig. 3.

Fig. 3: An Elekta Synergy IGRT System in the Department of Radiation Therapy of the Medical University Vienna. (a) is the patient support and (b) shows the Linear Accelerator (LINAC) itself. (c) and (d) are the kV-imaging support for Image Guided Radiation Therapy and (e) is the Electronic Portal Imaging (EPI) detector for the acquisition of MV-images.

Such radiation therapy systems consist of LINACs positioned in front of patient supports, one or both of them movable. This, in combination with collimators allow to focus the radiation on the tumor. As shown in Fig. 3, an Electronic Portal Imaging (EPI) detector can be placed additionally into the beam direction to allow the acquisition of mega-voltage images but these only feature poor contrast [11]. If the system supports image guided radiation therapy (IGRT), an additional kV-imaging device is positioned orthogonal to the radiation direction. This allows to acquire kilo-voltage images that can be used for the patient pose estimation.

For brachytherapy, other sources for radiation are required. These are permanent radiation emitting sources that are formed for the specific application. For example, if superficial cancer shall be treated, the source is placed on the skin or if prostate cancer shall be treated, the source is formed to small "seeds" that are implanted in or around the tumor [11]. There are seeds that have to be removed after a certain time while others stay permanently and just stop to emit radiation. Beside implantation, the insertion of the source into pre-existing body cavities is an alternative for the brachytherapy.

2.2 2D/3D-Image Registration

2D/3D-image registration is the process of transforming a volume so that a perspective projection of this volume is best aligned to a two dimensional reference image. In most situations, transformation is performed with six degrees of freedom: rotation around the x-, y- and z-axis and translation along the direction of these axes. The process of registration itself consists of three main steps:

1. Rendering of a Digitally Rendered Radiograph (DRR) on the basis of the volume. Several volume rendering algorithms exist that can perform this task, for example ray casting, splat rendering, shear warp factorization or texture mapping — see Sect. 2.3.
2. Calculation of a merit function with the DRR and a reference image as input producing a single output value that describes the degree of similarity between the two images — see Sect. 2.4.
3. Using of heuristic optimization algorithms to optimize the similarity measure calculated by adjusting the three translation and three rotation parameters of the DRR rendering — see Sect. 2.5. These new parameters are then used for the next iteration.

These steps are repeated until the result of the merit function is minimized/maximized or after a fixed number of iterations has been reached.

Registration of medical data is in general sub-divided into two types of algorithms:

Rigid Registration sets the restriction that images differ only in translations and rotations, while the content of the image itself has nearly the same shape. This thesis only discusses rigid registration algorithms.

Non-Rigid Registration allows deformation of structures and therefore introduces additional degrees of freedom. These techniques exceed the scope of this thesis.

2D/3D-image registration is the target of much research. In the following, several of these studies shall be presented.

Birkfellner et al. investigate different aspects of rigid 2D/3D-image registration to improve its performance. The main attention lies on DRR rendering [1, 20] as well as on the development of new merit functions [3, 21]. At the moment, this group works on a project with the goal to perform 2D/3D-image registration at a rate of 5 Hz. More details about this project can be found in Sect. 3.

Chen et al. investigated the dual energy digital radiography (DEDR) technique for detecting cardiac calcifications by comparing it to CT scans using 2D/3D-image registration [4]. They developed a shear-warp based DRR rendering algorithm (see Sect. 2.3.3) and used Mutual In-

formation (see Sect. 2.4.1) as merit function for the registration.

Kubias et al. developed a GPU based 2D/3D-image registration algorithm that performs the DRR rendering as well as parts of the merit function calculation on the GPU [5]. This allows to additionally boost image registration. It was possible, because parts of the similarity measurement could be computed in parallel and so the CPU had less work without an increase of transfer time between GPU and CPU because of the use of multi pass shaders introduced with Shader Model 3.0.

Beside the already mentioned projects, many works exist that show the application of 2D/3D-image registration in different fields of application or using different rendering methods or merit functions, for example [22, 23, 24].

2.3 Volume Rendering Techniques for Medical Data

There are many different rendering techniques for the processing of medical volume data, generated by Magnetic Resonance Imaging (MRI) or X-ray Computed Tomography (CT) scanners. These rendering algorithms differ in performance, fields of application and in the quality of the resulting images. There are several distinguishing features for them.

In general, rendering techniques can be divided into **image space** based and **object space** based algorithms. Image space based rendering operates on every pixel of the target image while object space based means that an image is built through an operation on each voxel of the original volume [25, 26]. Therefore, image space based rendering is slower for larger image sizes and more or less robust to the complexity of the object to render while object space based rendering operates well for large image sizes but is sensitive to complex objects.

Another distinguishing feature is the representation of the volume data. The two main types are **Direct Volume** and **Surface or Indirect Volume rendering** algorithms [27]. Direct Volume Rendering is the process of directly transferring the volume data into the 2D image without the help of intermediate formats. Popular direct volume rendering algorithms are ray casting, splatting and shear warp rendering [28]. Surface or Indirect Volume Rendering is the technique of representing 3D objects as closed surfaces constructed of many polygons (dots, lines, triangles, ...). This technique is today very popular in most modern video games because it is the way common graphics hardware processes scene data. The polygonal surfaces are used to calculate the final 2D image. Popular surface rendering algorithms are ray casting and rasterization [28], which is the default technique used by modern consume graphics hardware. This thesis only discusses direct volume rendering techniques because they are more feasible for medical data.

For all these different types of rendering algorithms, different rendering styles exist, that create totally different types of images [26, 29]:

Summed Voxel Rendering (also called x-ray or average rendering) results in the computation of Digitally Rendered Radiographs (DRR). All voxels are composed equally to calculate the final image — see Fig. 4 (a).

Maximum Intensity Projection Rendering considers only the most intense voxels for composition. Using this technique, only the most dense structures are displayed which can for example be the bones or in case of angiography the blood vessels — see Fig. 4 (b).

Full Volume Rendering (also called translucent rendering [26]) renders an opaque volume with semitransparent tissue above. This is achieved by composing the voxel data from front to back until an threshold for the opacity is exceeded — see Fig. 4 (c).

Isosurface Rendering creates an image of an opaque volume. This is done by defining a so called ISO-value. Every voxel above this value is said to be "inside" the volume while every voxel below is "outside". When traversing the volume front to back, the first voxel above the ISO-value is rendered. This technique is also called First-Hit-Shading [30] — see Fig. 4 (d).

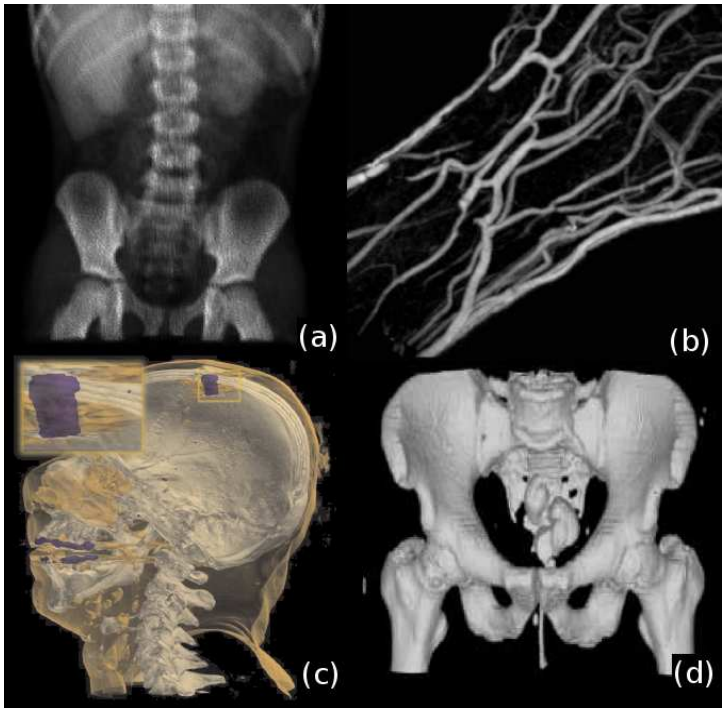

Fig. 4: Comparison of different rendering styles for medical volume data: (a) shows Summed Voxel Rendering [20], (b) shows Maximum Intensity Projection Rendering [31], (c) shows Full Volume Rendering [32] and (d) shows Isosurface Rendering [33].

The following sections present and describe the most popular rendering techniques for generating Digitally Rendered Radiographs (summed voxel rendering) of medical volume data.

2.3.1 Ray Casting

Ray Casting is the most popular direct volume rendering method for scientific fields of application. It is an image space based technique that produces high quality images and has good support on common consumer graphics hardware [8, 34].

The algorithm for ray casting is quite simple. For every pixel, a ray is created and all data along this ray is composed according to the desired rendering style — see Fig. 5. When using orthogonal projection, the rays are all parallel to each other and normal to the image plane while when using perspective projection, the rays have one common point, the so called eye or focal position.

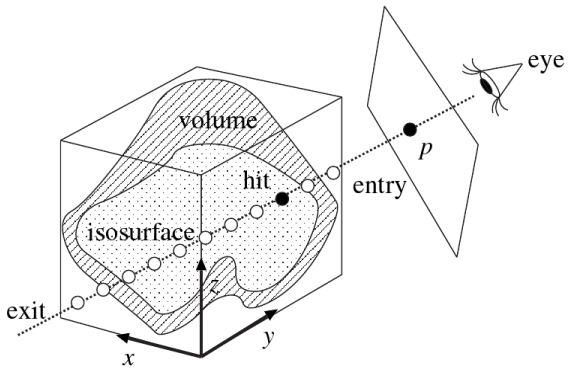

Fig. 5: Schematic illustration of a perspective ray casting algorithm [35]. A ray is sent from an eye point through the specific pixel and intersected with the volume to render. Inside the volume, samples are taken at certain intervals and composed later to compute the final pixel. In some implementations it is also possible that the volume is positioned between the eye and the image plane.

The volume can be positioned behind the image plane as well as between the eye position and the image plane. To compose the color of a pixel, samples of the volume data are taken at certain intervals on the ray and combined to compute the desired rendering style.

One advantage beside the good hardware support of ray casting is that there is nearly no difference between orthogonal and perspective projection [25]. The only difference lies in the way the direction of the ray is calculated: eye position to pixel for perspective projection and normal of image plane at the pixel's position on the image plane for orthogonal projection.

Because of the per pixel design of this image space based algorithm, where every pixel is independently calculated, ray casting is perfect for parallelization [36]. Therefore, the image plane is divided into several parts that are calculated independently on different nodes of a cluster. In a last step, these image patches are re-combined. Another advantage of ray casting is its simplicity in design. A simple ray caster can be developed fast and without much effort, but produces only images of moderate quality. To increase the image quality, several optimization

techniques have to be implemented and make the algorithm therefore more complex. This is at the same time also a big advantage of ray casting. Because of this technique being very popular, it is also very well investigated and there are many works about how to optimize ray casting algorithms in terms of quality of the resulting image as well as memory consumption or rendering performance [8, 36, 37, 38, 39].

Ray Casting for direct volume rendering should not be confused with ray tracing, although both terms can be used as synonyms. Ray tracing is a rendering technique in computer graphics that produces very high quality renderings and also includes more complex effects like reflection, refraction, shadows and many more [40]. This is done by using secondary rays. When a ray hits an object, secondary rays are cast to all light sources to compute shadows and into the reflected and/or refracted direction to render these effects. One example of a ray traced high quality image can be seen in Fig. 6.

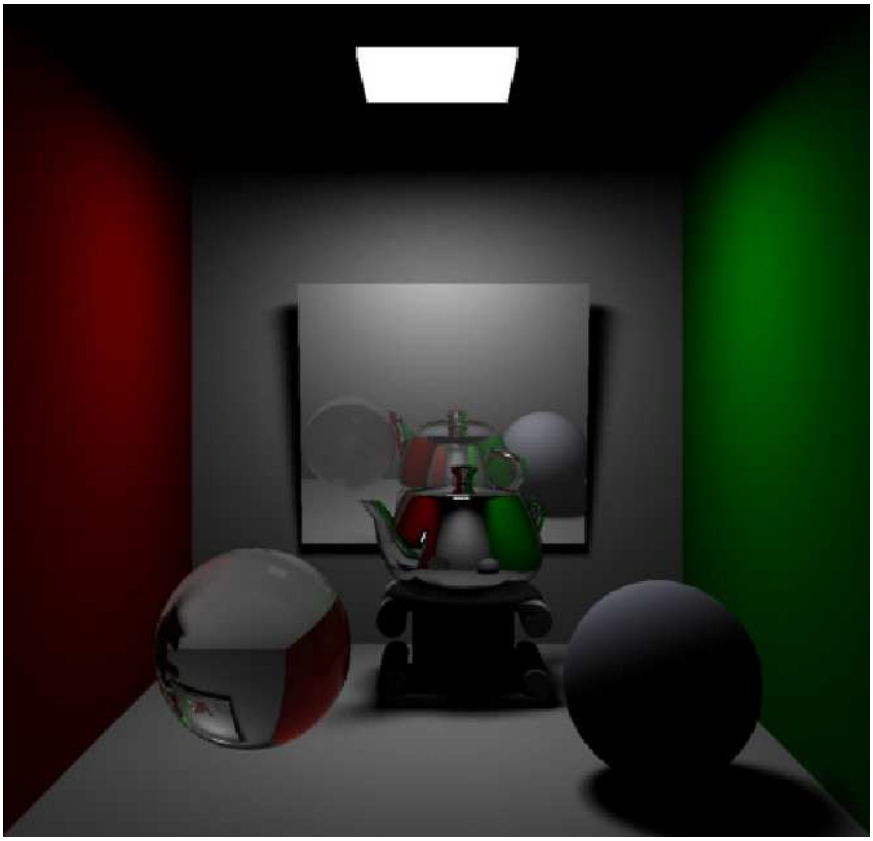

Fig. 6: Scene generated with ray tracing published by Guerrero [41]. Ray tracing allows to achieve far more sophisticated effects than when using ray casting for direct volume rendering.

There are also several problems when using ray casting as rendering algorithm. One problem — although its an advantage on the other hand — is the image space based design. The amount of work does not correlate with the amount of present data. Therefore, without special

optimization techniques, volumes that are nearly empty require the same rendering time as volumes completely populated with data. If data with holes in the middle shall be displayed, even optimization algorithms can hardly be applied to skip these sections.

Another problem is that very thin structures can easily be missed with ray casting, because not all data is used for composing the final image but only the data samples at certain intervals [8]. If a structure is thin enough to fit between two sampling positions, it will not occur in the image. A similar problem occurs when performing perspective ray casting. Small structures can then be easily missed because in larger distances, the rays are relatively far away from each other.

The next problem only occurs when rendering isosurfaces or full volumes. In most cases, the ray is traversed and samples are taken in certain, constant intervals to compose the final value of the pixel. When rendering isosurfaces, it hardly occurs that the exact surface is hit. This phenomena is called under sampling — because you would need a higher sampling rate to better hit surfaces — and can lead to artifacts that look similar to annual rings of trees — see Fig. 7 (a).

As mentioned earlier, there exist many optimization techniques and strategies for ray casting to improve its performance, memory consumption and image quality. One strategy to boost the performance is to utilize the Graphics Processing Units (GPUs) to do the rendering [34, 8, 26, 30, 32, 36]. GPUs are specialized processing units that process data in parallel. Because of the per pixel design, ray casting is perfect for this kind of design. For every pixel of the image, the shader casts a ray through the volume stored as a texture and saves the result of this ray as color information into the framebuffer. For a more in detail description of GPU ray casting, see Sect. 4.2.2.

Another very efficient and also popular acceleration technique is the use of bounding structures [8, 39]. This is the wrapping of complex objects into simpler structures to simplify the initial test if a ray hits this structure. If this is the case, the test can be applied to the more complex structure. This saves time because rays missing a structure are much faster detected. The simplest case is to test if a ray hits the volume at all. This allows to set all these rays to the background color by just testing against one object.

Another important simplification and therefore optimization of the ray casting algorithm is to start sampling rays as late as possible and to stop sampling as early as possible to reduce the number of samples per ray. This optimization technique is called early ray termination. Using one bounding structure for the whole volume allows to just sample the rays inside of this structure and therefore reduce the number of samples dramatically [8]. Also when sampling from front to back, stopping when the opacity of the ray exceeds a threshold can help to stop sampling as early as possible without losing too much information [39].

To reduce the number of tests if a voxel is traversed by the ray is crucial for increasing the performance. This can be achieved for example by using three dimensional line drawing algo-

rithms [42]. The volume is represented as a bounding box and the two intersections with it are calculated. With these intersections, it is possible to use for example the 3D DDA algorithm or the 3D Bresenham algorithm to calculate the voxels that are traversed by the ray. Using these algorithms produces the problem of aliasing that has to be further solved. If orthogonal projection is used, it is even possible to prepare one line template which can be applied to all pixels.

To avoid sampling in empty space, several authors propose octree data structures [39, 43, 38]. Voxels are summarized to larger blocks of data. This can be done at several levels. If a ray hits a higher level structure, it can be fast decided which of the sub volumes have to be further refined and which can be skipped.

One technique to avoid the missing of thin structures is to have a sampling distance smaller or equal the voxel size, which on the other hand increases the rendering time. A solution that does not decrease the performance is the use of irregular sampling patterns [8]. These change the sampling distance for every pixel (ray) which allows to hit the structure with some rays while missing it with other rays. Dizzling leads to more correct results but suffers from sampling artifacts.

To solve the earlier mentioned problem of under sampling when rendering isosurfaces, Scharsach [8] proposes a method called hitpoint refinement. This method allows to render surfaces with noticible fewer artifacts although lower sampling rates can be used which has the side effect of additionally increasing the performance. This technique works as follows:

The volume is traversed front to back with a sampling rate higher than the voxel size. If the first sample above the iso-value chosen for surface extraction is detected, the refinement procedure starts. Instead of continuing with the forward sampling, a step of half the actual sampling size is made backwards and a sample is taken. If this sample lies again outside of the volume, the step size is again halved and a step is made forward, otherwise a half step is made backwards again. This half-stepping forward or backward depending on the last sample is repeated several times until a step size smaller than the voxel size is reached. Through this algorithm, the position of the surface is found exactly. This algorithm allows to traverse the volume outside the target surface very fast while finding the surface position with high precision [8]. The result of hitpoint refinement can be seen in Fig. 7.

One optimization technique for the memory consumption is the division of the data into memory blocks/bricks [8, 39, 38, 44, 43]. This technique cannot only be applied to ray casting but is useful for all possible rendering algorithms. To avoid the storing and rendering of empty space, the data is represented as an octree structure that only stores non-empty blocks of data. All blocks are loaded into the volume separately and are connected by some superstructure [45, 44]. This allows to trade memory savings for one additional indirection when accessing the data in

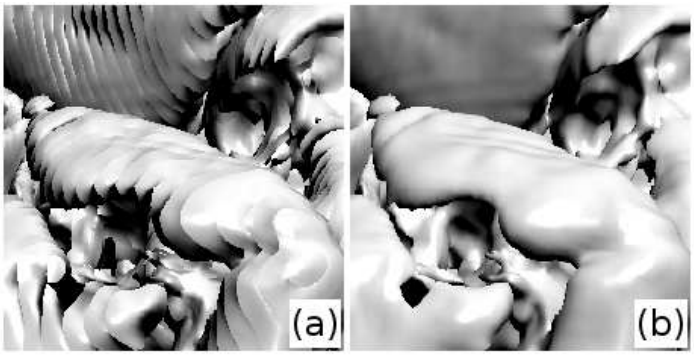

Fig. 7: The problem of under sampling when rendering isosurfaces and its solution. (a) Artifacts similar to annual rings of trees as result of under sampling. (b) The same image after applying Hitpoint Refinement [8].

the memory.

Sect. 4.2 presents a CPU and a GPU implementation of ray casting using some of the already described optimization techniques. Additionally, this GPU implementation is analyzed in terms of quality and performance in Sect. 5 and Sect. 6.

2.3.2 Splat Rendering

Splat rendering is an object space based technique that — similar to ray casting — is used for the computation of high quality images [46] and is, according to [25], the result of optimizations in early volume rendering approaches.

Splat Rendering projects every voxel onto the image plane (geometric projection) where one voxel has an effect on several pixels (it is splatted). The projected position is used as the center of a footprint that describes how the surrounding pixels are affected [25, 46] as shown in Fig. 8. In perspective rendering, it is also necessary to adjust this footprint according to the angle between the projection direction and the normal of the image plane to get the best results [25] as illustrated in Fig. 9.

The image itself is constructed by composing all these "splats" together. Depending on the function used for this step, the different rendering styles can be achieved. For example, if all voxel footprints are summarized, summed voxel shading is achieved. If all voxel footprints but the most intense one are discarded, a maximum intensity projected image is created. Depending on the rendering style, it may be necessary to order the volume data (e.g. using full volume or isosurface rendering style) [26] which decreases the performance of the algorithm.

Hansen and Johnson [26] as well as Ruijters et al. [48] distinguish between three different sub-types of splat rendering:

Composite-Only Splatting This method is the above mentioned: every voxel is individually

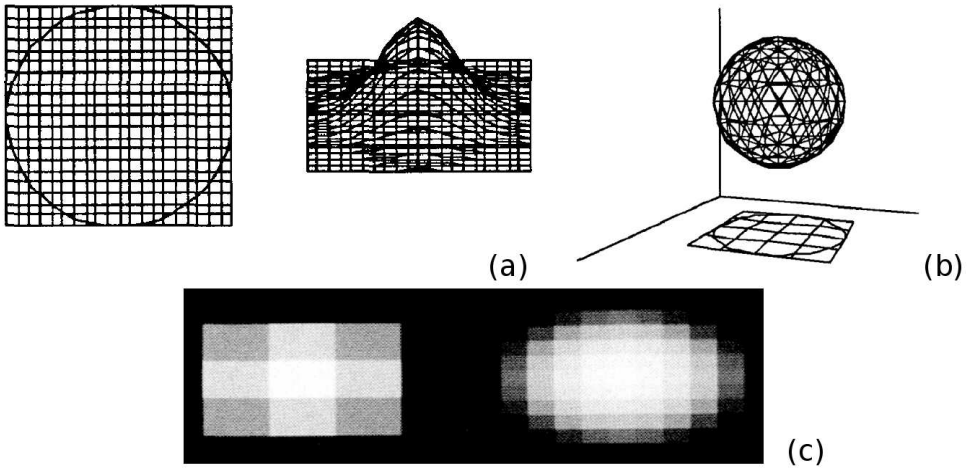

Fig. 8: Footprints of splatted volume data as illustrated in [47]. (a) shows a top down and a perspective illustration of such a footprint if the volume data is interpreted as a sphere as shown in (b). For the image itself, a singe splat — depending on the used footprint kernel size — results in a splat that is bright in the middle and gets darker for pixels farther away (c).

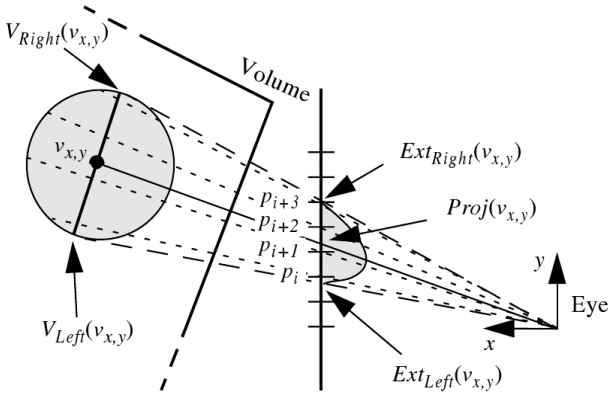

Fig. 9: Modification of the footprint of splat rendering when using perspective projection as described by [25]. As a result of the angular "hit" of the splatted voxel, the footprint distribution changes from Gaussian to asymmetrical.

composed to all already splatted voxels to create the final image. This individual composition may lead to so called color bleeding artifacts but is the most simple and therefore fastest technique.

Axis-Aligned Sheet-Buffered Splatting This technique tries to address artifacts possible by composing all voxels individually by creating sheets of data normal to the axis most aligned to the viewing direction. These data sheets are then composed to create the final image. Because of the axis alignment of the sheets, changing the viewing direction leads to so called popping artifacts. These are sudden changes of the rendering when the viewing direction gets better aligned to another axis during rotation. Because of the combination of voxels to sheets, this method requires to order the volume data.

Image-Aligned Sheet-Buffered Splatting To address the popping artifacts, image-aligned sheet-buffered splatting works similar to axis-aligned sheet-buffered splatting, but voxels are combined to sheets always orthogonal to the viewing direction. This is a more complex operation but does not lead to popping artifacts and creates therefore images of much higher quality.

The use of Gaussian footprints for the rendering allows to construct smooth images based on the discrete volume data. Furthermore, since all voxels are projected, no features can be missed. Splat rendering shows good performance for sparsely populated volumes and is nearly independent of the size of the final image. Only the size of the footprint influences the performance [46].

Splat rendering has also several disadvantages. If using isosurface or full volume rendering style, it is not possible to just process the data that is later visible. It is necessary to project every voxel before it can be decided which "splats" are visible and which do not contribute to the final image.

Another problem — specific for the topic of this thesis — is the missing possibility to exclude parts of the image from the rendering process and just compute a subset of all pixels. This limitation originates in the object space based design of splat rendering [26].

The algorithm is, in comparison to ray casting, more complex although not producing images of significantly better quality. Therefore, ray casting and splat rendering can be seen as alternative methods depending on the scene that has to be rendered.

Mueller and Yagel describe a ray driven splat rendering technique [25]. In this approach, instead of splatting all voxels in parallel, rays are emerged from an eye position through the image plane and all voxels are determined that are "hit" by this ray. In a next step, only these voxels are splatted to determine the pixels' color and intensity. This technique combines advantages of splat rendering and ray casting. For example, it allows to process only the voxels that contribute

to the final image and therefore allows to get better performance by just rendering parts of an image, while constructing the pixel itself with the smooth splat rendering technique.

An often used technique to increase splat rendering performance is Early Splat (or Point) Elimination. This is a technique similar to early ray termination in ray casting [26]. It can only be applied if the voxels are splatted in front to back order, which is often the case when using image or axis aligned splatting techniques. The alpha value is accumulated and when a certain threshold is exceeded, no further splats are composed for that position [49].

A very fast summed voxel splat rendering approach is described in [20]. This approach reduces the footprint of a splatted voxel to just the nearest neighbor of the projection which allows to render an image of poor quality with just one matrix multiplication per voxel. The most serious problem with quality using this technique is that the regular grid of the original data is also projected as shown in Fig. 10.

Fig. 10: Sampling artifacts produced when computing a DRR with splat rendering without using Gaussian footprints for every splat as described by [20]. The artifacts are a result of the regular grid in which the volume data is positioned.

A fast and efficient solution to this problem is the so called "wobbling" of either the focal (projection) spot's position or the volume data position (see Fig. 11) [20]. This allows to reduce the regular line artifacts at the cost of increased image noise (see Fig. 12) which can be weakened by using classical low pass filtering techniques.

This algorithm has proven to be a very fast technique that is applicable for the field of 2D/3D-image registration. It was further accelerated by implementing it on the GPU [1]. More details about this algorithm can be found in Sect. 3 and Sect. 4.1.

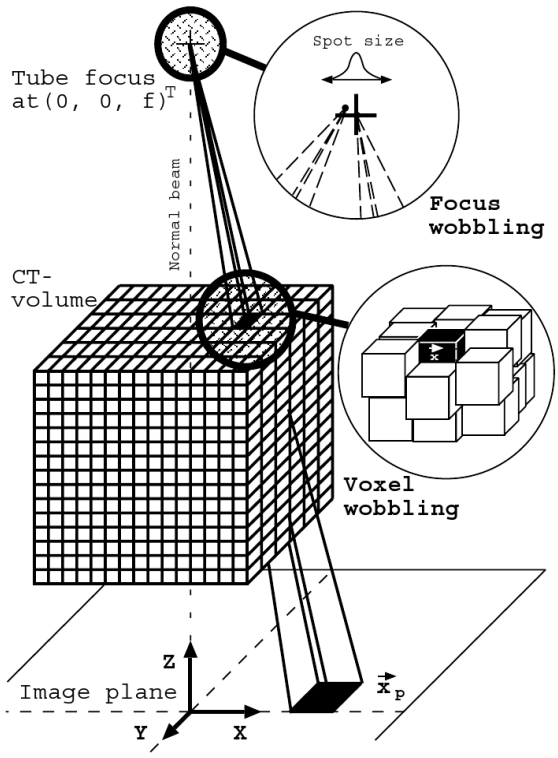

Fig. 11: Illustration of wobbled splat rendering as proposed by [20]. Either the position of the focal (projection) spot or of each individual voxel is modulated to avoid artifacts because of the regular grid. The voxels are then projected onto the image plane and blended to produce an image with summed voxel style (DRR).

Fig. 12: DRR generated with simplified splat rendering and voxel wobbling [20]. (a) shows the unfiltered image while (b) shows the same image after low pass filtering. Although the splat rendering algorithm is extremely simplified, the resulting image quality remains good enough for many applications, including 2D/3D-image registration.

2.3.3 Shear Warp Factorization

Shear Warp Factorization is an object space based volume rendering technique that uses an intermediate coordinate system to simplify the blending of parallel slices of volume data [50]. The algorithm consists in general of three steps:

1. The object slices are transformed into the so called sheared object space. This can be achieved by shearing the slices normal to the main viewing axis. If perspective rendering shall be achieved, the slices have additionally to be scaled. The process of orthogonal shear object space transformation is shown in Fig. 13 (a) while Fig. 13 (b) shows the same for perspective projection. This operation can be mathematically formalized as

$$M_{shear} = P \cdot S \tag{1}$$

with P being a permutation matrix to align the volume with the z-axis (if z is the viewing direction) and S being either

$$S_{ortho} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ s_x & s_y & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{2}$$

for orthogonal projection or

$$S_{persp} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ s'_x & s'_y & 1 & s'_w \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{3}$$

for perspective projection.

2. The volume slices in sheared object space can be easily blended to form an intermediate 2D image. All different rendering styles can be applied.

3. The intermediate 2D image is at last transformed to the final image in a so called warping step. This warping step is the inverse operation to the sheared object space transformation and can mathematically be expressed as

$$M_{warp} = P^{-1} \cdot S^{-1}. \tag{4}$$

Lacroute [50] sums up the shear warp rendering algorithm using the pseudo code shown in listing 1.

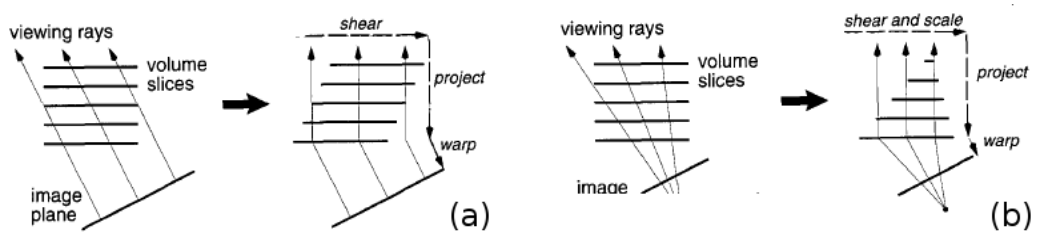

Fig. 13: Illustrations of shear warp factorization by [24]. (a) shows orthogonal shearing where the slices of the volume are rearranged so that the composition is simplified. (b) shows perspective shearing which additionally requires a scaling of the slices.

Listing 1: Pseudocode implementation of shear warp factorization [50].

```
ComputeShadingLookupTable();
Clear(tmp_image);
Foreach (voxel_slice from front to back)
  tmp_image = tmp_image over Resample(Shade(voxel_slice));
end
image = Warp(tmp_image);
```

Shear warp factorization is an algorithm that performs very fast when implemented in software [22]. On the other hand, the quality of the produced renderings is worse than that of ray casted or splatted images.

As with all rendering methods, parallelization can be used to further improve the performance of shear warp factorization [50, 22]. This parallel algorithm concentrates on the shearing and composing of the image but also improves the warping step. The volume, i.e. every slice of the data, is separated into several parts and these are sheared, composed and warped independently. At last, the different parts are reassembled to get the final image.

2.3.4 Texture Mapping

Texture Mapping is a method based on the use of the texturing capabilities of GPUs [23] and was first described by Cabral et al. [51] Although it is often presented as an independent rendering method, it is just another name for slice based splat rendering as already described in Sect. 2.3.2. There, this technique can be found under the terms **Axis-Aligned Sheet-Buffered Splatting** or **Image-Aligned Sheet-Buffered Splatting**.

2.4 Merit Functions

To compare different images, merit functions are required that produce a single value that correlates with the "similarity" of the two images. In general, two types of merit functions exist

for the comparison of images [52]:

Feature based merit functions identify image features in the images and try to align them. Examples of features are high intensity spots (markers), edges, dense structures or homogeneous regions.

Intensity based merit functions correlate intensity patterns (for example the distribution of the histogram) of the images without analyzing the image content itself.

There are several other distinguishing features for merit functions, for example if the two images must have similar intensities or not or if they have to be from the same modality or if they can come from different ones (for example a DRR from a CT in comparison to the mega-voltage image from a LINAC).

For the project described in Sect. 3, the following constraints have to be considered.

- The merit function must be able to compare multimodal images.
- It shall be fast to compute.
- The possibility of parallelization would be good for future performance improvements.
- Sparse sampling as an optimization technique (see Sect. 2.6) is only applicable for intensity based merit functions.

In the following sections, several merit functions are presented that are applicable for 2D/3D-image registration in general or that were implemented for this project (see Sect. 3).

2.4.1 Mutual Information

Mutual Information is a merit function that measures the similarity of two images or volumes by determining the statistical dependence between them. This dependence can be calculated on the basis of the Shannon entropy that can be expressed as

$$H = \sum_i p_i \, log \, \frac{1}{p_i} = - \sum_i p_i \, log \, p_i \tag{5}$$

with p_i being the variables under consideration. In the case of registration, this can be the intensities of the pixels or voxels [53]. The Shannon entropy is a measure for how easy it is to guess the value of any p_i. For example, if an image only contains one intensity, its Shannon entropy is very low because there is only one possible value p_i can have.

Joint distribution is also very important for mutual information. It defines the probability that a pixel has intensity r in the first image and intensity s in the second image. This is equivalent to

the correlation of the intensities of the two images. The better the correlation of them is, the more information one image reveals about the content of the other image.

The equations

$$I(A,B) = H(B) - H(B|A) \tag{6}$$

$$I(A,B) = H(A) + H(B) - H(A,B) \tag{7}$$

and

$$I(A,B) = \sum_{a,b} p(a,b)\, log\, \frac{p(a,b)}{p(a)p(b)} \tag{8}$$

with $I(A,B)$ being the mutual information, H being the Shannon entropy, A and B being the two images, $p(a,b)$ being the joint distribution and $p(a)p(b)$ being the joint distribution in case of independence of the images [53, 54] show how to compute the mutual information. All of these equations are equivalent. Eq. 6 expresses that the mutual information is "*the amount of uncertainty about image B minus the uncertainty about B when A is known*" [53]. This is in some way similar to the amount of certainty that can be achieved for one image by knowing the other. Eq. 7 states that mutual information is the sum of the entropies of both images reduced by the entropy of the joint distribution of the images. Eq. 8 is just another way of expressing Eq. 7. They describe the difference between the entropy of both images if they are independent from each other and the entropy of the images if they are combined. To sum up, mutual information is a measure of the dependence of the information present in both images.

Mutual information is a popular similarity measurement because there are many different types of methods that are summarized under the term mutual information methods. A summation of them by Pluim et al. [53] is shown in Tab. 1.

An important advantage of mutual information is that it is applicable for multimodal images [55]. There is no need for a linear dependency between the intensities, because it relies on the joint distribution instead of the histogram, which is used by many other merit functions.

A problem of mutual information is that it only compares local intensities and does not consider the surrounding of the pixels, although this is no mutual information exclusive problem. It can easily be demonstrated with the gray stripe experiment [56]. This experiment consists of one black image with a gray stripe and another image where every vertical line has another intensity but also showing some kind of lighter stripe in the middle (see Fig. 14).

When using mutual information to register these two images, the algorithm shows an optimum for any position although this is obviously not correct (see Fig. 15). The problem is that mutual

Tab. 1: Different types of mutual information methods summed up by Pluim et al. [53] Bold attributes show the attributes important for the project described in Sect. 3.

Method	**Application**
Preprocessing	Modalities
Measure	monomodality
entropy	**multimodality**
normalization	modality to model
spatial information	modality to physical space
Transformation	Subject
rigid	intrasubject
affine	intersubject
perspective	model
curved	
Implementation	Object
interpolation	
probability distribution function estimation	
optimization	
acceleration	
Image dimensionality	Number of images
2D/2D	**2**
3D/3D	$>$ 2, with known inter-image geometry
2D/3D	$>$ 2, with unknown inter-image geometry

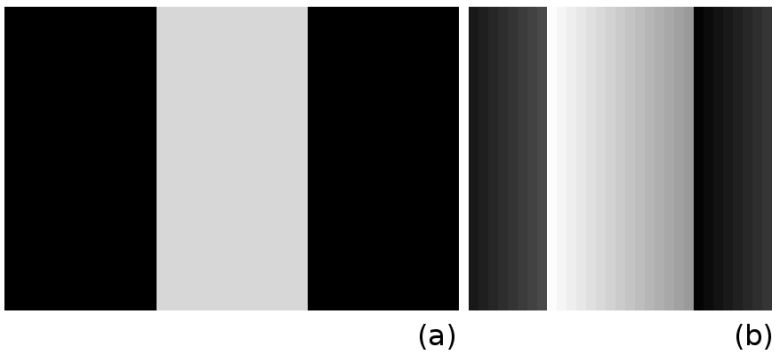

Fig. 14: Images representing the gray stripe experiment as described by Roche et al. [56] (a) shows a black image with a gray stripe while (b) shows an image where every vertical line has another intensity. Mutual information fails registering these images.

information does not consider special information about the whole image which is maybe present but only the pure, local intensities.

Beside 2D/3D- or 3D/3D-registration, mutual information can also be used in other fields such as for example detecting features on images [57].

2.4.2 Pattern Intensity

Pattern intensity is a merit function that considers also the structure of the images for comparison [58]. It was proposed as a better merit function for 2D/3D-image registration during surgeries for aligning preoperative CT-data to the current patient pose.

This merit function requires a different workflow than the others presented in this thesis. It uses only a subset of the 3D-data for the image registration. This subset shall contain a rigid structure like a bone or a vertebra, which has to be visible on the 3D-data as well as on the later 2D-images and that is used for the registration. This structure has to be segmented in a preprocessing step. During the registration, a DRR is computed from the segmented sub-volume and subtracted from the reference image [59]. Penney et al. [60] show that it is also possible to register the complete images and that the segmentation step can be skipped. The resulting difference image is then processed using the following equation

$$P = \sum_{i,j} \sum_{(i-k)^2+(j-l)^2 \leq r^2} \frac{\sigma^2}{\sigma^2 + (I_{diff}(i,j) - I_{diff}(k,l))^2} \tag{9}$$

with P being the pattern intensity similarity measure, i, j being the pixel's coordinates, r defining an area which shall be searched for structures, k, l being the coordinates in the area that is searched, σ defining the sensitivity of the measure for structures and I_{diff} being the fluoroscopy image minus the DRR of the segmented or unsegmented rigid structure [58, 59].

Because only the rigid structure is of interest, the difference image has only to be evaluated around this structure which additionally accelerates the process. Eq. 9 gives a measure of how smooth the image is. The subtraction of aligned images results in a smoother image than the subtraction of non-aligned images, because parts of the rigid structure from both images will remain. The amount of remaining features can then be used as the similarity value.

2.4.3 Cross Correlation

Cross correlation, which is also known as correlation coefficient [23] or Pearson moment correlation coefficient [3] is a measure of the similarity of two images that compares the pixels at every position following the equation

$$M_{CC} = \frac{\sum_{ij}(I_{ijBase} - \overline{I}_{Base})(I_{ijMatch} - \overline{I}_{Match})}{\sqrt{\sum_{ij}(I_{ijBase} - \overline{I}_{Base})^2 \sum_{ij}(I_{ijMatch} - \overline{I}_{Match})^2}} \quad (10)$$

with M_{CC} being the cross correlation, I being the intensity of a pixel, i, j being the x- and y-coordinates of the pixels and $\overline{I}$ being the mean intensity value of an image [3, 61].

One limitation of cross correlation according to [23] is that it requires a linear relationship between the intensities of the images. If this is not the case, for example when comparing a PET image with a CT image, Cross Correlation does not provide useful solutions. This is an important limitation for 2D/3D-image registration, because the data is always multimodal and a linear relationship can not be guaranteed.

The algorithm is in general designed for affine and rigid transformations. Studies show that extended algorithms are also able to be applied to images that differ in perspective or other small deformations [55].

The quality of registration with cross correlation can be improved by preprocessing the image, for example by low pass filtering the image or just using the edges. This preprocessing to get better results works for nearly all merit functions, especially for all correlation based merit functions [55]. Using edge detection also removes the drawback of the required linear intensity relationship between the two images [55].

According to Birkfellner et al. [3], high intensity pixels (e.g. high intensity artifacts) are a problem for cross correlation which can be removed by using careful radiometric calibrations, for instance by histogram equalization.

2.4.4 (Stochastic) Rank Correlation

Rank correlation was proposed for the first time by Birkfellner et al. [3] It is a merit function based on Spearman's rank correlation which is a statistical measure for ordinal (not metric) data.

Rank correlation for two images can be calculated in several serial steps [3]:

1. Generate a list per image containing the intensities and the associated coordinates.
2. Sort these lists by their intensities.
3. Calculate an average rank index for each intensity group (the index of a rank would be the mean index of all members of the rank, but in practice, the first index of the rank is also applicable).
4. Replace the intensities by the average rank index.

5. For each pixel coordinates in the base image, search for the corresponding coordinates in the match image, calculate the squared rank difference $\Delta\rho_n^2 = (\rho_{Base_n} - \rho_{Match_n})^2$ and finally calculate the rank correlation coefficient M_{RC}^* for the two images using the following equation

$$M_{RC}^* = 1 - \frac{6\sum_{n=1}^{N}\Delta\rho_n^2}{N(N^2-1)} \tag{11}$$

with N being the number of pixels.

This merit function has the advantage over cross correlation (see Sect. 2.4.3) that it does not require a linear relationship between the intensities of the two images. Another advantage of rank correlation is that it is far more robust to high intensity artifacts in comparison to cross correlation because it does not use the mean intensity for the calculation.

A drawback of rank correlation in its original version is the high runtime requirement because of the sorting operation.

The first optimization to improve the runtime is the so called stochastic rank correlation. Based on the work of Zöllei et al. [7], Birkfellner et al. [3] propose to only use parts of the image for the merit function calculation to accelerate the whole process. This subset has to be randomly chosen from the whole image so that the distribution and therefore the order of the intensities is not modified significantly.

Later research on rank correlation showed that it can also be calculated without the sorting step and that it can be calculated in a time competitive to cross correlation. This is possible because the rank index of each pixel only depends on the number of pixels in its rank as well as the number of pixels in all ranks before. These values can also be calculated on the basis of the histogram which allows to skip the ordering procedure under the restriction that there are fewer different intensity values than pixels in the image.

2.4.5 Correlation Ratio

The correlation ratio is a measure for the functional relationship between two images [62]. It is based on the fact that if two images are registered correctly, a function can be defined that maps every intensity of one image to an intensity of the other image [55]. This functional dependence between the two images can then be used as a similarity measure.

After computing the joint probability density functions of both images, the correlation ratio can be computed using the following equations

$$M_{CR} = 1 - \frac{1}{\sigma_I^2} \sum_j p_j \sigma_{I|j}^2 \tag{12}$$

where M_{CR} is the correlation ratio and

$$m_I = \sum_i i p_i \quad (= \text{mean}), \tag{13}$$

$$\sigma_i^2 = \sum_i (i - m_I)^2 p_i \quad (= \text{variance}), \tag{14}$$

$$m_{I|j} = \frac{1}{p_j} \sum_i i p_{ij} \quad (\text{conditional mean}), \tag{15}$$

$$\sigma_{I|j}^2 = \frac{1}{p_j} \sum_i (i - m_{I|j})^2 p_{ij} \quad (= \text{conditional variance}) \text{ [63]}. \tag{16}$$

The correlation ratio is a very robust merit function that can be used for images with any relationship between their intensities. This is indicated by the results of the gray stripe registration experiment (see Sect. 2.4.1) that shows much better performance with correlation ratio than for example mutual information — see Fig. 15.

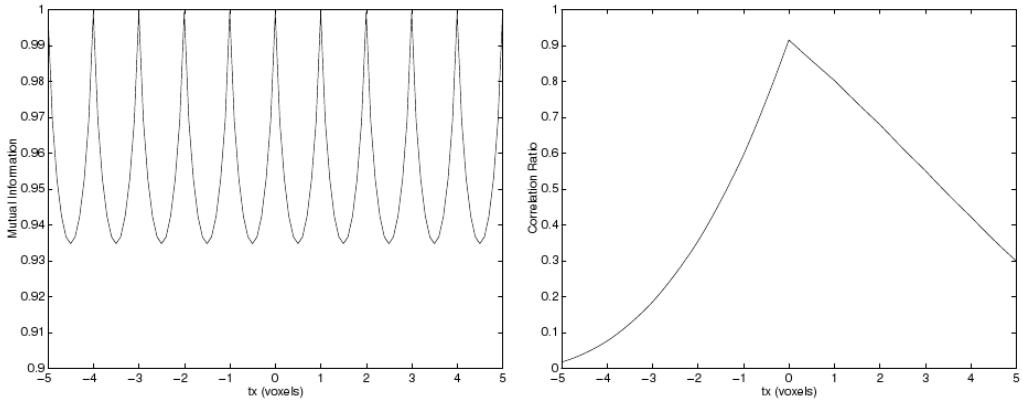

Fig. 15: Results of the gray stripes registration experiment for mutual information and correlation ratio [64]. Mutual information fails to detect the (for humans) right position while correlation ratio performs well for this problem.

2.4.6 Other Merit Functions for 2D/3D-Image Registration

Methods like cross correlation, rank correlation or correlation ratio can be summarized under the term correlation-like methods. Other methods of this class are:

Sequential Similarity Detection Algorithm (SSDA) This algorithm measures the difference between all pixels and uses the sum as a measure of the similarity [55, 65].

Sum of Squared Differences (SSD) Instead of optimizing the difference as in SSDA, SSD used the square of the difference as a measure of similarity [55, 66]. This method as well as SSDA are very sensitive to intensity differences between the images.

Hausdorff Distance (HD) After transforming images to its edge representation, the Hausdorff distance of the two images can be used as a merit function [55].

Beside correlation-like methods, there are other methods like for example Fourier methods [55]. These methods do not compare intensities of the images but the Fourier representation of them in the frequency domain. They are faster to compute and more robust to varying conditions and frequency-dependent noise.

Another approach uses the decomposition of the images into Zernike moments [21]. It shows more robustness to histogram differences than cross correlation and can be computed in competitive time, if the construction of the Zernike moments is reduced to just a few significant moments.

2.5 Optimization Algorithms

To find the right parameters for transforming a volume so that its projection fits the reference image at best, optimization of an at least six-dimensional problem is required. Several algorithms exist to perform this task. This section describes some of these optimization strategies that are implemented in the project presented in Sect. 3 or that are used in publications about 2D/3D-image registration.

In general, all presented algorithms are heuristic optimization strategies. A heuristic strategy is a search algorithm that cannot provide perfect solutions in terms of quality, time and/or space, but which has a high probability that it will find an acceptable solution. This is acceptable if no algorithm for finding the best solution exists. If two non-identical images shall be compared, there cannot be an ideal solution and it is also possible that several solutions of nearly similar quality can be found. Because of that, heuristics are the best way to optimize the image similarity.

There are three different types of optimization algorithms:

Geometric Algorithms like the downhill simplex algorithm (see Sect. 2.5.1) operate on the merit function itself.

Gradient-based Algorithms like the conjugate gradient method (see Sect. 2.5.2) operate on the gradient of the merit function to find the optimum. This requires the calculation of derivations of the merit function and therefore several renderings and merit function evaluations per optimization iteration.

Probabilistic Algorithms like simulated annealing (see Sect. 2.5.3) work similar to simple search strategies but use random steps to avoid local optima.

In the following, several heuristic optimization algorithms applicable for the problem of 2D/3D-image registration are presented.

2.5.1 Downhill Simplex Algorithm

The downhill simplex algorithm (also called Nelder-Mead or amoeba method) is a simple algorithm that achieves good results for the optimization of 2D/3D-image registration problems. A specialty of this algorithm is that it does not require derivations to be calculated [67]. Calculations of derivations are slow because they require two, three, five or even more function evaluations (depending on the quality that shall be achieved). When talking about 2D/3D-image registration, one function evaluation is a full rendering step followed by one merit function calculation. This shows that it is important to sample as few positions as possible.

The downhill simplex algorithm is based on the structure of a simplex. A simplex is a polytope of $N + 1$ vertices with N being the number of dimensions of the simplex. This means for example that a one-dimensional simplex has two vertices (equates to a line segment) or a three-dimensional simplex has four vertices (equates to a tetrahedron).

This simplex structure is used to describe the correlation between sample points in the problem space and to choose the next sample position during the search. The initial positions are $N + 1$ points. In the concrete application of 2D/3D-image registration, one position consists of the six degrees of freedom (three translations and three rotations). The initial positions can for example be calculated based on an initial guess. The user tries to align the two images by hand. Based on this position, seven points are calculated: the initial guess position is modified for each parameter separately so that one position per parameter modification and the original initial guess position can be used as starting simplex.

The downhill simplex method is an iterative algorithm. Every iteration consists of replacing the worst simplex vertex (sample point) by another, better position. In a first step, all vertices of the simplex have to be sorted (let the best be called P_1, the second best P_2, ..., the second worst P_N and the worst P_{N+1}). To find a better position, four different operations exist [68, 67] as shown in Fig. 16.

Reflection The worst sample is reflected at the center of gravity of the remaining samples as described by

$$P_R = (1 + \alpha) \cdot P_g - \alpha \cdot P_{N+1} \tag{17}$$

with P_R being the reflected point, α being a factor that describes the extent of the

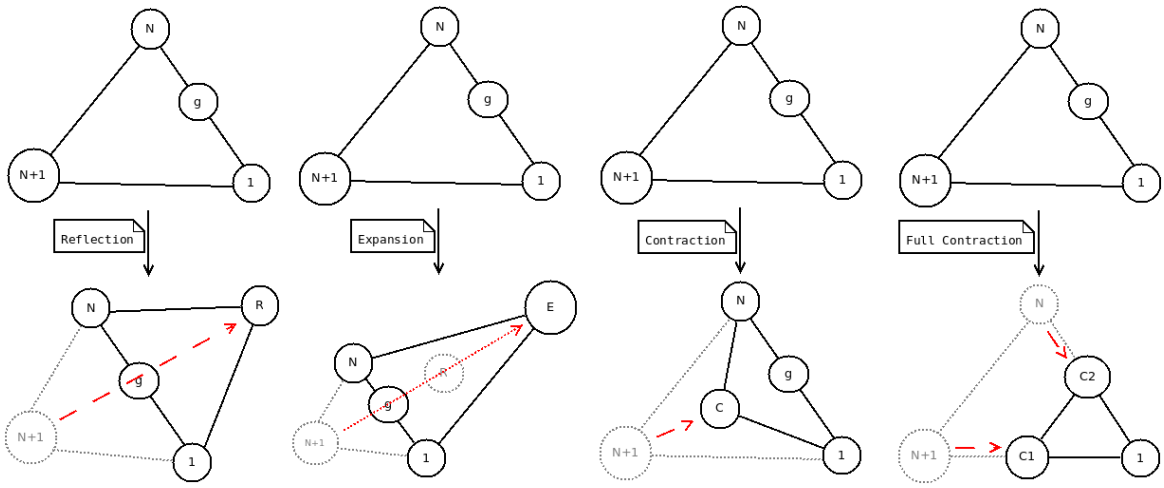

Fig. 16: Illustration of the four simplex operations. $N+1$ represents the worst point, 1 the point with the best value and N one point in-between. g is the center of gravity of all points devoid of the worst. R is the reflected point, E the expanded and C, $C1$ and $C2$ are the contracted points.

reflection (this is normally set to 1 as described by [68]), P_g being the center of gravity of the remaining simplex points and P_{N+1} being the worst simplex point. This reflected Point P_R replaces the worst point P_{N+1} if the value of P_1 is better than the reflected point's value but if its value is still better than the value of P_N. The idea behind this is that if the function gets worse in one direction, it is likely that it gets better in the opposite direction.

Expansion The reflection can be extended to the expansion, which means that the segment between the center of gravity and the new point is enlarged so that the new point is farther away as described by

$$P_E = (1-\gamma) \cdot P_g + \gamma \cdot P_R \tag{18}$$

with P_E being the expanded point, γ being the factor that describes the extent of the expansion (this is normally set to 2 as described by [68]). This expanded point P_E replaces the worst point P_{N+1} if its value is better than the value of the so far best point P_1; is this not the case, the reflected point P_R is taken. This allows to more quickly overcome homogeneous falling/rising sections of a function to save iteration cycles.

Contraction If the reflected point P_R can not improve the situation (its value is worse than the value of the so far second worst point P_N), a contraction is performed. This is done using the equation

$$P_C = (1 - \beta_1) \cdot P_g + \beta_1 \cdot P_0 \tag{19}$$

with P_C being the contracted point, β_1 being the factor that describes the extent of the contraction (this is normally set to 0.5 as described by [68]) and P_0 being the worst point (either the original worst point P_{N+1} or the reflected point P_R, depending on which has the worse value). This operation reduces the size of the simplex either from the direction of the reflected or the original worst point, if the value of the contracted point P_C is better than the value of the worst point P_0. This allows the simplex to change direction and "slow down" if it is in danger of "stepping over" an optimum.

Full (Multiple) Contraction If the contraction is also not feasible (because P_C is also worse than so far worst point P_0), the simplex performs a full contraction. This operation is performed following the equation

$$P_i = (1 - \beta_2) \cdot P_1 + \beta_2 \cdot P_i \mid i > 1 \, and \, i \leq N + 1 \tag{20}$$

with P_i being all points but the best, β_2 being the factor that describes the extent of the full contraction (this is normally set to 0.5 as described by [68]) and P_1 being the point with the best value. This operation of the simplex allows to follow curved valleys into the direction of the optimum [68].

To sum up these operations, Nelder and Mead [68] provide a flowchart of the operations and the conditions when to choose which of the operations. It is shown in Fig. 17.

One iteration of the downhill simplex algorithm consists of sorting, performing one of the four operations and replacing the former worst point by the new one. After this, end conditions can be checked. If they are met, the optimum is found, otherwise, the next iteration is performed. Possible end conditions are a certain number of iterations [69], a merit value (value of the best position of the simplex) below/above a certain threshold [70], the standard deviation of the function values of the simplex [68] or the simplex gets smaller than a certain size [67].

One application example of the downhill simplex algorithm is the 2D/3D-image registration as will be shown in Sect. 3 and as described in [54, 71]. Other fields of application are for example automated texture registration [72], detection thresholding [57], the computation of motion vectors in 3D space for video analysis [73] or the pricing of convertible bonds [69].

It also exists a modified implementation of the downhill simplex algorithm that is more optimized for the use in distributed environments [70]. This implementation reduces the number of required iterations for the optimization (even when used in a single core environment) but to achieve this, a more complex calculation for each of the iterations has to be computed. It does not

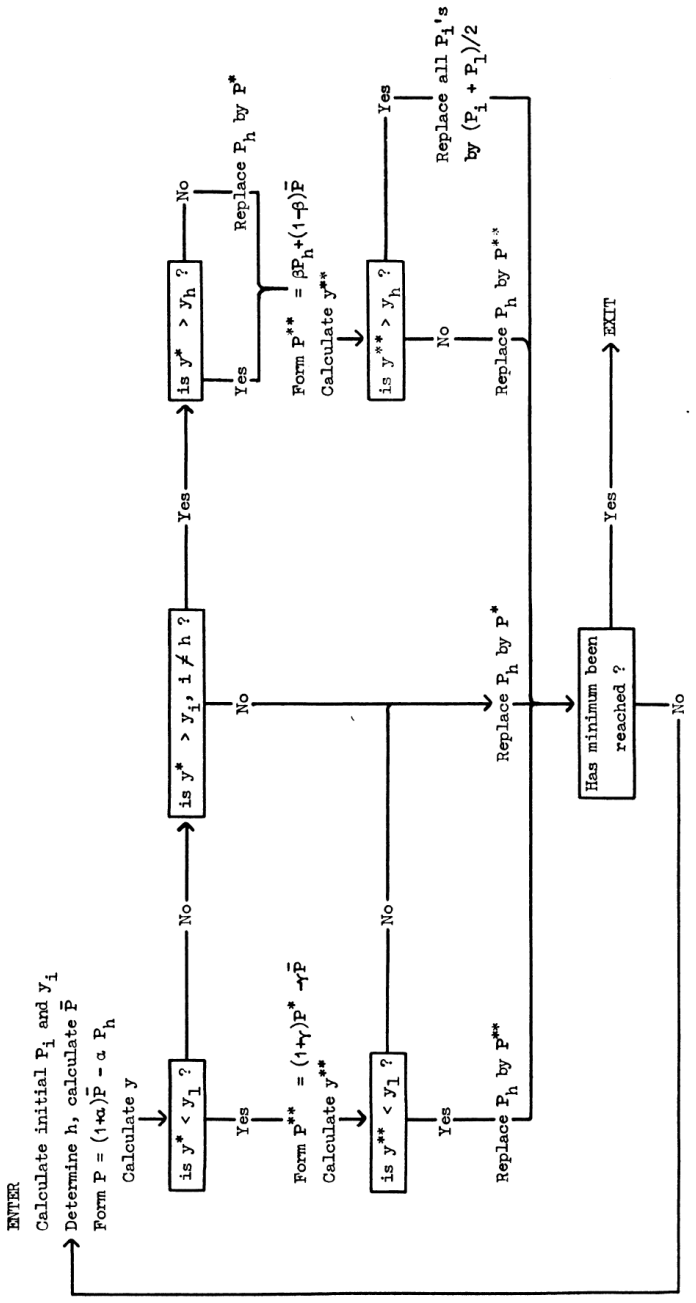

Fig. 17: Flowchart showing the different decisions and operations that are performed during the downhill simplex algorithm [68]. P_h referrs in this diagram to the point with the worst value. At first, the reflected point is calculated and it is checked if its value is better than the value of the so far best point. Is this the case, the expansion is calculated and set as new point if its value is even better than the reflection. Otherwise the reflected point is set as the new point. Was the reflected point's value worse than the value of the best but better than the value of the second worst, it is set as the new one. Is the reflected point's value worse than the so far worst point's value, the contraction from the worst or the reflected point is calculated and set as the new point if its value is better than the so far worst point's value. If also the contracted point's value is worse than the so far worst value, a full contraction is performed.

distribute the iteration steps but assigns different vectors of parameters to the different nodes of a distributed environment. First investigations have shown that it is not applicable for use in the project described in 3, because the distribution requires too much time relative to the optimization itself.

2.5.2 Conjugate Gradient

Conjugate gradient is an improvement of the gradient descent [55] or steepest (gradient) descent algorithm [74]. Steepest descent can be described in a simple way by the following steps:

1. Calculate the gradient at the actual position. For the first iteration, this position is chosen by the user as initial guess.
2. Solve a one-dimensional minimization problem along the direction of the steepest descent — this means to "follow" the direction of the gradient until the minimum along this line is found.
3. Use this minimum as position for the next iteration to search in a new direction for a further descent.
4. If no direction can be found that improves the solution or a threshold is exceeded, the algorithm terminates.

This method seems, apart from calculating the gradient of the merit function, very simple, but has several problems. An important problem is that the initial position has to be on a slope that leads to the global minimum, otherwise the minimum cannot be found. Another problem is that with every iteration, the algorithm only turns into a direction orthogonal to the direction of the former iteration. This leads often to a high number of iterations until the minimum is found.

Conjugate gradient optimizes the steepest descent method by choosing the new direction not locally as the steepest descent algorithm does, but "*as being conjugate to the previous one with respect to the function to minimize*" [74]. This means if g is the vector in the direction of the steepest descent and d is the direction to search the next minimum position for, instead of setting

$$d_{i+1} = g_{i+1} \tag{21}$$

for every iteration, conjugate gradient uses the equation

$$d_{i+1} = g_{i+1} + \gamma_i \cdot d_i \tag{22}$$

$$\gamma_i = \frac{g_{i+1} \cdot g_{i+1}}{g_i \cdot g_i} \quad \text{or} \quad \gamma_i = \frac{(g_{i+1} - g_i) \cdot g_{i+1}}{g_i \cdot g_i} \tag{23}$$

for calculating the next direction to search the minimum for [74]. The first equation for γ_i was proposed by Fletcher and Reeves while the second equation was proposed by Polak and Ribiere and shows faster convergence for non-quadratic functions. These equations mean that, instead of using directly the new steepest descent, the gradient is conjugated with the direction of the step before which leads to directions better pointing to the minimum of the function because the turns are no longer orthogonal but can have an arbitrary angle.

Both, steepest descent as well as conjugate gradient can find the exact optimum after a fixed number of iterations, but this can be very time consuming. The advantage of the conjugate gradient method is that after very few iterations, it produces a value very close to the exact optimum.

A drawback is, that both methods require many function evaluations (DRR rendering and merit function evaluation) for the calculation of the gradient and directional minimum determination. Therefore, the bottleneck of a registration solution is shifted from the optimization routine to the DRR rendering and similarity measurement computation.

Another problem of conjugate gradient is that there are only few publications that describe this technique for solving minimization problems. Most sources use conjugate gradient to heuristically solve large equation systems and it is hard, especially for computer scientists, to map these descriptions to concrete problems.

2.5.3 Simulated Annealing

Simulated annealing is a very well known, widely used and well investigated heuristic procedure that is applicable for many different problems like for example registration, scheduling or searching [75]. Its main difference to normal, local search is that it also allows steps that lead to a worse solution. This enables the algorithm to leave local optima.

Simulated annealing is based on the Monte Carlo algorithm. It defines a probability with which a solution is accepted for the next step although it is worse than the current solution. The difference to the Monte Carlo algorithm is that this probability decreases over time which allows the algorithm to converge to an optimum. The equation

$$e^{\frac{f'-f}{T}} \tag{24}$$

describes this annealing of the temperature that controls the algorithm [76]. f' in equation 24 is the value of the new solution, f the value of the old solution and T is the temperature that anneales over time.

The simulated annealing algorithm can be described with the pseudo code illustrated in listing 2. It consists of two main loops. The inner loop performs the simple steps of the heuristic search and

Listing 2: Pseudocode implementation of the simulated annealing algorithm. After setting the initial values, the outer loop is executed with a certain temperature that is modified at the end of the loop. At the beginning of the outer loop, the inner loop chooses a random neighbor and sets it as the new position if its value is better. If its value is worse, a random number is compared with the solution of equation 24 to see if it is taken as the new position although its value is worse.

```
begin
  t = 0
  T = initial T
  x = initial guessed solution
 repeat
    repeat
      choose a random x' from the neighborhood of x
      if x' is better than x
        x = x'
      else
        if a random number Z < pow(e, -abs(f(x'), f(x)) / T)
      t = t + 1
    until temperature-update criterion met
    T = g(T,t);
 until termination criterion method
end
```

is left if the temperature-update criterion is met. This can be for example after a fixed number of steps or if the solution has not changed for several iterations. The outer loop terminates if the temperature fulfills the final termination criterion which can be for example the reaching of a specific temperature value.

Several extensions exist to improve this algorithm. Some of them are for example a non-monotone annealing plan, dynamic annealing plans, better choosing of the neighbor, parallelization or the combination with other procedures [77].

The simulated annealing algorithm can be applied to many different problems like image registration, traveling salesman problems or shortest circuit path problems [78]. It is seldom the fastest and best algorithm for a problem but nearly always provides "good enough solutions" [77].

2.5.4 Other Optimization Algorithms for 2D/3D-Image Registration

Beside the already mentioned optimization procedures, several other algorithms and heuristics exist to find the correct registration parameters.

Beside steepest descent and conjugate gradient, that were presented in Sect. 2.5.2, several other

gradient based methods exist that are all searching for the next local minimum into the direction of the local gradient, but all differ in the way the next search direction is chosen. Examples of such algorithms are Powell's direction set method, Quasi-Newton methods or the Levenberg and Marquardt method (least-squares methods) [55, 74]. A comparison of these algorithms and the downhill simplex method for the registration of a CT and an MRI image illustrated for a two-dimensional subspace is shown in Fig. 18 [74]. This figure shows that all different gradient based algorithms find the optimum very well but differ in the number of iterations they need to reach it.

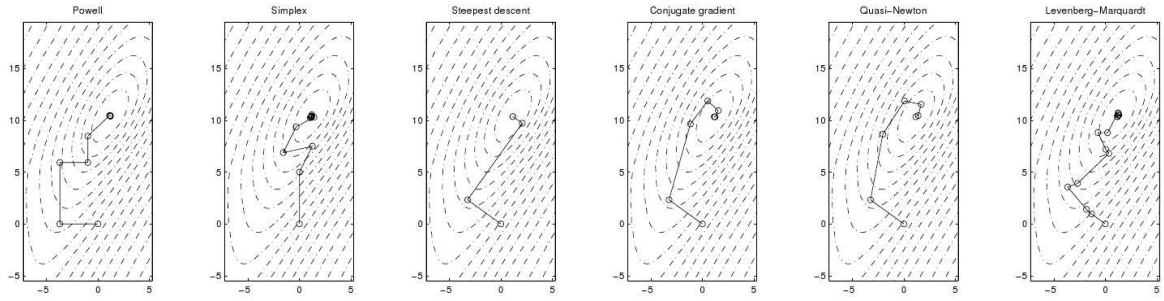

Fig. 18: Comparison of different optimization methods for a two-dimensional subspace (the x-axis shows the rotation angle around the x-axis and the y-axis shows the translation in y-direction) of the registration problem of a CT and an MRI image [74]. The shown methods from the left to the right are: Powell's direction set, downhill simplex, steepest descent, conjugate gradient, Quasi-Newton and Levenberg-Marquardt.

Beside these methods, many other search algorithm can be applied to the registration problem [79, 58], like for example local search or variable neighborhood search, but most of them are very inefficient for the concrete problem of 2D/3D-image registration and are therefore very rarely used.

2.6 Sparse Sampling of Image Data

The size of the images used for 2D/3D-image registration has a strong influence on the performance because larger images

1. require the generation of larger DRRs which is especially more expensive for image space based rendering methods such as ray casting.
2. require more pixel values to be compared in the merit function and therefore decrease performance of this step.

To overcome these performance problems, Zöllei et al. [7] proposed the use of sparsely sampled histogram estimators. This means that not the whole image is used for the calculation of the merit function, but only a stochastically chosen subset of the pixels.

The idea behind this "trick" is that the histogram of an image is similar to the histogram of a subset of the image as long as the pixels used for the subset are chosen randomly. One restriction is that this is only applicable for intensity based merit functions. Feature based merit functions do not work with sparsely sampled data because features may not be visible if not all pixels are generated.

Birkfellner et al. [3] present stochastic rank correlation as a new merit function which was in its first implementation only competitive to other merit functions because of sparsely sampling of the image data. By only using 3–5% of image data for the registration, this implementation, which is normally slow because it depends on sorting, can be performed with about the same time as cross correlation.

2.7 Direct Volume to Surface Transformation

As described in Sect. 2.3.1, one optimization for GPU based ray casting is the use of more complex bounding structures [8]. The smaller this bounding structure is, the more empty space can be skipped during the ray casting itself. This allows a tremendous speedup, because the vertex shader, that has to handle the complex bounding structure, is not used for the ray casting itself and so the introduced complexity can be handled without slowing down the whole process too much. To generate a bounding structure that best represents the content of a thresholded volume, this volume has to be transformed into a surface representation.

The most basic algorithm for such a transformation is the Cuberille algorithm. This algorithm iterates over all voxels and tests if they are part of the volume to be extracted. For every voxel inside of the volume, all sides to not included voxels are modeled as a quad or as two triangles. All voxels are therefore directly represented as small cubes, which gives this algorithm also its name. Cuberille produces very "blocky" looking structures with many corners but is very easy to implement [80].

The most famous algorithm that performs a volume to surface transformation is the marching cubes algorithm [81] which is described in this section and which is used in Sect. 4.2.4 to speedup the GPU based ray casting implementation of the project described in Sect. 3.

Marching cubes processes a volume represented by voxels and creates a list of triangles as well as the normals for these triangles. Depending on an iso-value, it is possible to distinguish which voxels lie inside of the volume and which lie outside. For the transformation, marching cubes splits the volume into cubes of eight voxels. For each of these cubes, the eight voxels can either be "in-" or "outside" of the volume, which results in 256 different cases that can be summarized to 15 different cases (as shown in Fig. 19 (a)) because many cases are symmetric [81].

The algorithm now marches through the volume and produces the list of triangles and normals by performing the following steps [81]:

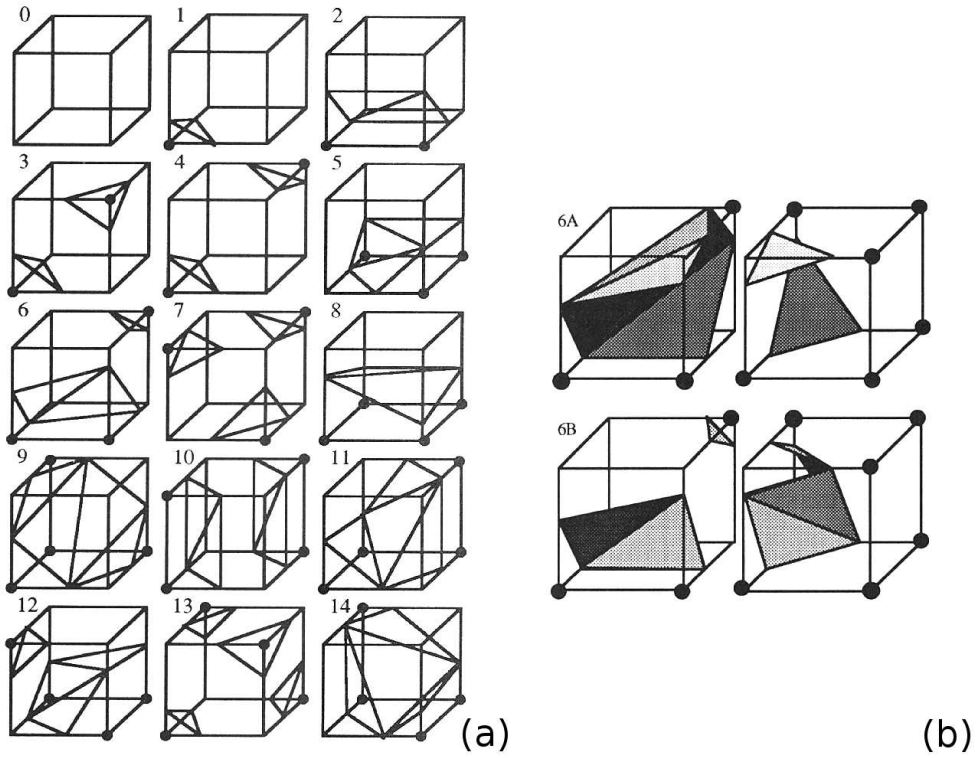

Fig. 19: Marching cubes cases illustrated by Nielson and Hamann [82]. (a) shows the 15 remaining cases after symmetric cases were removed and (b) shows two examples for the ambiguity problem of the marching cubes algorithm.

1. Choose the next cube consisting of eight voxels.
2. Calculate for each voxel if it is inside or outside and code this as a byte where each bit represents one voxel, 0 represents an outside voxel and 1 represents an inside voxel.
3. Use this byte to decide which of 256 possible cases is present and load the prepared vertex data for this case.
4. Interpolate between the voxels to find the exact positions of the vertices for the new triangles.
5. Calculate the normals of the triangles.
6. Append the new triangles and their normals to the output lists.

One problem of the marching cube algorithm as described is that for some cases, ambiguous solutions exist how triangles can be used to represent the cube (see Fig. 19 (b)) [82]. This can lead to holes in the resulting surface which will invalidate the whole surface model. Nielson and Hamann [82] therefore propose an asymptotic decider that allows to determine the correct triangle setup for a concrete situation and so to always compute valid surface models.

This algorithm can be further improved by storing already processed voxels, reduction of resolution [81], data-structures like octrees for the volume representation (allows to skip parts of the volume that lie completely in- or outside the volume by preprocessing using a surface tracker) [83], simplification by skipping interpolation steps (e.g. always use the middle between two voxels) and therefore precompute and just lookup normal vectors [84] or using the GPU for the computation [85].

2.8 Graphics Hardware and General-Purpose Computation on Graphics Processing Units

Graphic cards are today relatively cheap hardware components for common consumer PCs that can be used as high performance co-processors, especially for calculation intensive operations. The main purpose is the processing of surface data to generate high quality renderings for video games as well as the processing (encoding/decoding) of video data.

General-Purpose Computation on Graphics Processing Units (GPGPU) is the field of using common consumer graphics hardware for other purposes than classical rendering by rasterization of polygons. Literature provides many examples of such (in the past) unintended use of graphics hardware, for example to perform complex data base operations [86], belief propagation for cloud tracking and reconstruction [9], analysis of financial markets [87] or detection and tracking of white blood cells in video microscopy [10].

The intention of this section is to present and discuss the main aspects of Graphics Processing

Unit (GPU) programming including graphics hardware itself, Application Programming Interfaces (API) for graphics hardware as well as libraries that especially address GPGPU and try to allow programmers without knowledge of graphics processing to use graphics hardware for parallel data processing. If several technologies are available (as for programming APIs), this section only concentrates on the technologies used in the later described project (see Sect. 3).

2.8.1 Consumer Graphics Hardware

During the last 30 years, there was a fast evolution in computer graphics hardware. In the past, space consuming and expensive hardware was available for scientific purposes which was very limited in flexibility and performance. Over the years, these devices developed into small and general applicable co-processors for consumer PCs, easily available for everyone.

In the 90th of the 20th century, the usage of graphic cards in consumer PCs got more and more common. This increased popularity was mostly driven by the PC entertainment industry, especially by the video game manufacturers. The first consumer graphics hardware was very limited in its functions but every new generation had larger memory, faster bus connections to the CPU and more (fixed) functions available for processing the data.

At the beginning of the 21st century, the manufactures of computer hardware started to make the data processing more and more flexible and included programming capabilities into their hardware. High Level Shading languages (HLSL) allowed the user (mostly game developers) to adjust the processing of geometric data to achieve more sophisticated effects like for example reflection or refraction. In the middle of this decade, scientists started to exploit these programmable graphics cards to perform general purpose computations.

The design of earlier graphic processing units (GPU) was that of a fixed function pipeline as shown in Fig. 20. Input data was always processed in the same way. This could only be influenced by setting parameters. To standardize these configuration capabilities of graphics hardware, application programming interfaces (API) like OpenGL (see Sect. 2.8.2) or Direct3D (see Sect. 2.8.5) were introduced.

With the introduction of programmable shading processors to the graphics pipeline as shown in Fig. 21 (a), the graphics pipeline evolved to the shape shown in Fig. 21 (b) with the vertex and fragment processor as programmable steps in the pipeline. These are accessed through the former configuration APIs (mostly OpenGL or Direct3D).

The vertex processor is positioned at the beginning of the graphics hardware. Its main purpose is the modification of vertex data according to the model, world and viewing transformations specified. A program written for this processor is executed once per vertex without the possibility to access information about other vertices or the context of the vertex inside of its primitive. It supports many different vector operations implemented in hardware as well as branching and

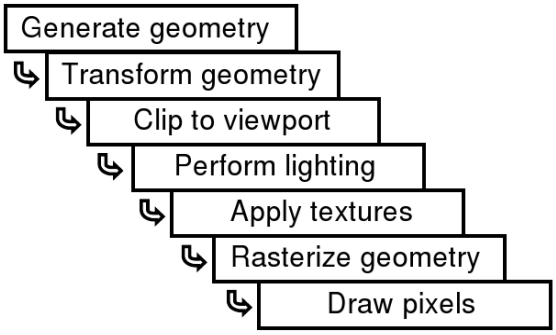

Fig. 20: Simplified design of a traditional fixed function graphics pipeline as used by older graphics hardware [88]. These pipelines were not programmable at all. The only flexibility was through configuration of the different steps in the pipeline.

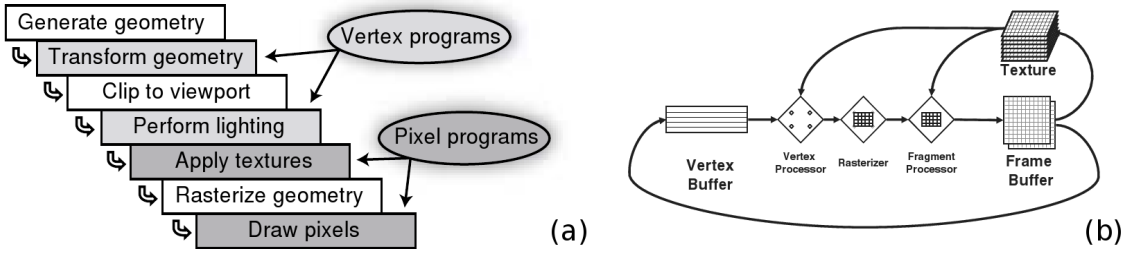

Fig. 21: (a) shows the evolution from fixed function graphic pipelines to programmable pipelines [88] that is presented in (b) [89]. Data is processed in two programmable shaders connected by fixed functions. Input is taken from the vertex buffer, processed in the vertex processor, rasterized, processed in the fragment processor and finally written to the framebuffer. Both, vertex and fragment processor can access the texture memory.

texture lookups [90].

The fragment or pixel processor is called after the rasterization step. Rasterization is the process of transforming the primitives into so called fragments. These fragments are similar to pixels, but in contrast to pixels, several fragments may exist at the same position, because primitives can overlay each other (the process of rasterization is illustrated in Fig. 22 and Fig. 24). The fragment shader is called for every of these fragments and processes them. The main purpose of it is the calculation of lightning effects, but it is also responsible for texturing and effects like fog, refraction or reflection.

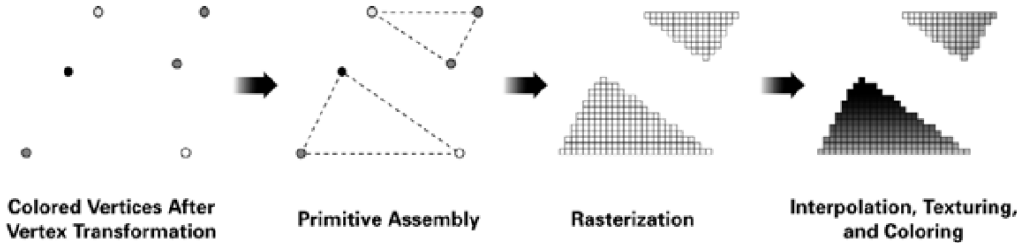

Fig. 22: Evolution of geometric data when passing the graphics pipeline as illustrated by [90]. Isolated vertex data is first connected to form primitives, which are rasterized to produce fragments which are then filled with the interpolated data of the former vertices.

After leaving the fragment shader, the data has to pass several tests and operations before it is finally displayed (or rendered to a texture) as shown in Fig. 23.

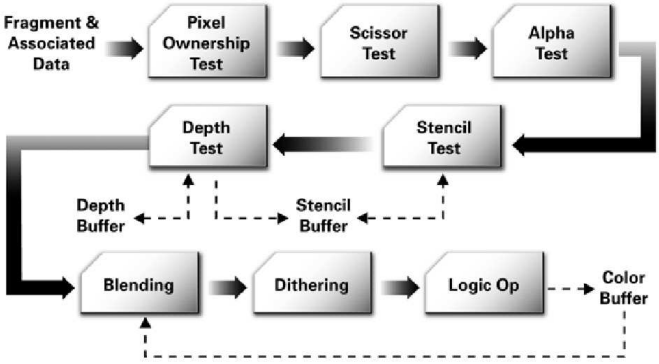

Fig. 23: Final steps of the graphics pipeline [90]. The fragment data undergoes several tests and operations before it is definitively rendered.

Pixel Ownership Test is performed to allow the operating system to have some control over the rendering process. Every fragment is evaluated if the pixel at the same position is owned by the context responsible for rendering. If not, the fragment may be discarded. This allows a rendering window to be partially occluded by other windows [91].

Scissor Test simply discards all fragments that lie outside a specified area [91].

Alpha Test compares the alpha value of a fragment to one reference value to decide if a fragment shall be discarded or not [86, 91].

Stencil Test is performed to allow the developer to reduce the rendering to parts of the display (or framebuffer). The stencil buffer is some kind of mask that encodes which fragments/pixel are drawn and which are discarded [86] based on conditions that compare the value of the stencil buffer with a reference value. After this comparison, the stencil buffer may be modified [91]. Beside these functions similar to the scissor test, the stencil test can for example be used to tag pixel positions for later rendering passes to achieve better reflection and shadow effects [92].

Depth Test is performed to determine which of all fragments at one pixel position is visible and which are occluded. The fragments depth value is compared with the current closest depth value that is stored in the so called z-Buffer (or Depth-Buffer). If the fragment is closer to the eye position, its depth value is written to the z-Buffer [86, 90].

Blending is the process of composing all remaining fragments at the same pixel position to one color that is stored in the framebuffer (or a texture) [90, 91].

Dithering modifies the color of fragments slightly depending on their position [91].

Logical Operations are performed between the fragment and the value at the corresponding position of the framebuffer [91].

Today, consumer graphics hardware is a nearly fully programmable co-processor for streamed data which has a very good performance because it is fully designed for parallel data processing. The graphics pipeline was further evolved and today contains a third programmable step called primitive processor [93] — see Fig. 24.

The primitive or geometry processor is called per primitive and can produce none, one or several primitives as output. Therefore it is the only programmable step of the pipeline that can enhance the data stream [93]. In the primitive processor, the information about all vertices of the primitive as well as the information of all adjacent vertices is available [94]. This is also special because it is the only processor that has access to more than the current unit of data.

In the future, CPU and GPU design will more and more converge. This can already be observed with multicore CPUs on the one hand and more general GPU programming models on the other hand. Parallelizations, as provided by the GPU, will be the major point for future developments in CPU as well as GPU design as can be already seen with the announcement of Direct3D 11 [95].

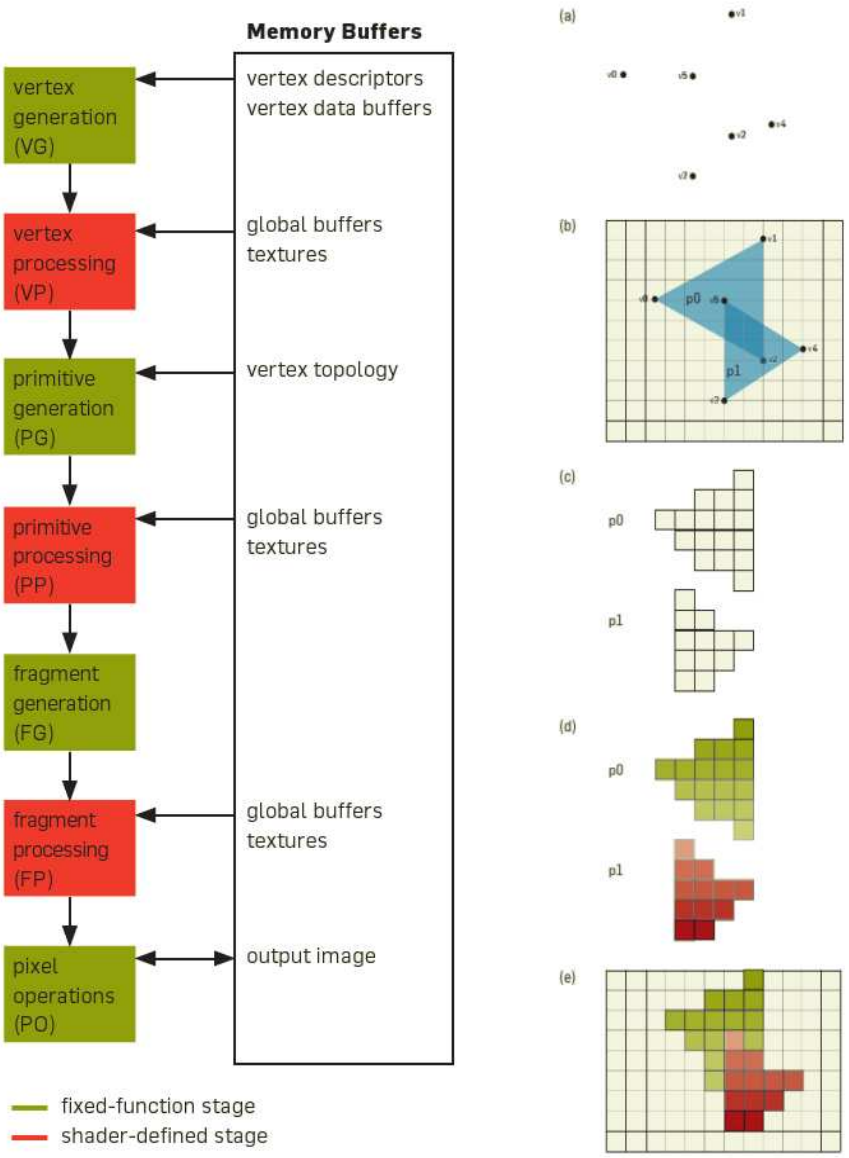

Fig. 24: Simplified representation of modern graphics hardware consisting of three programmable steps (red) connected by fixed functions (green) [93]. On the right, the transformation of geometric data while passing the pipeline is illustrated.

2.8.2 OpenGL

OpenGL (Open Graphics Library) is one of the major application programming interfaces (API) for the configuration and programming of graphics hardware. It is the successor of IrisGL, which was a proprietary API developed by Silicon Graphics [96]. Its first version was published in 1992 and it is at the moment at version 3.2, published in August 3rd, 2009 [97]. As Kilgard and Akeley [96] state, OpenGL is more than just an API, it is an architecture.

OpenGL has the following design philosophy [96]:

- high-performance
- defined by a specification and not an implementation
- it is a rendering state machine
- extendability
- data type rich
- cross-platform
- multi-language bindings

OpenGL supports all major features of today's graphics hardware including the Shader Model 4.0, texture arrays and primitive processing. The most important advantages of OpenGL are its cross platform capabilities.

There are several fields for future developments in OpenGL including, tessellation and geometry amplification, reducing communication to less but more powerful commands to improve the ratio of GPU to single-core CPU performance, improve compatibility to other systems (i.e. Direct3D), improve the computation support (OpenCL, Nvidia CUDA, AMD Stream processing) and extending OpenGL for unconventional graphics devices (handhelds, smart-phones, embedded systems, ...) [96].

2.8.3 Cg - Language

Cg is a language for directly programming graphics hardware and is developed and supported by NVIDIA corporation [90]. Its first version (1.1) was published in December 2002 and it is currently at version 2.2 (since October 2009) [98]. Its syntax is derived from C/C++ with some extensions required to address the specific features of graphics hardware [93]. It is compatible with OpenGL and Direct3D as well as with hardware of all major vendors (although only with NVIDIAs own hardware, all features of *Cg* can be used).

Cg is built around the concept of profiles. A profile describes which features a GPU supports and which cannot be used. The other concept is that of programs, which are high level functions that

Listing 3: Several vector and matrix specific operations provided by Cg. It is possible to initialize vectors and matrices directly (1., 3. and 6. line) and it is possible to access single elements as well as subsets of the elements of vectors or matrices in any order (2., 4., 5., 7. and 8. line).

```
float4 v = float4(1, 2, 3, 4);          // v = (1, 2, 3, 4)
v.x = 2;                                // v = (2, 2, 3, 4)
float2 u = float2(5, 6);                // u = (5, 6)
u = v.yz;                               // u = (2, 3)
v = u.xyxy;                             // v = (2, 3, 2, 3)
float4x4 m = {  0.0, 1.0, 2.0, 3.0,     // m = (0.0, 1.0, 2.0, 3.0,
                0.1, 1.1, 2.1, 3.1,     //      0.1, 1.1, 2.1, 3.1,
                0.2, 1.2, 2.2, 3.2,     //      0.2, 1.2, 2.2, 3.2,
                0.3, 1.3, 2.3, 3.3};    //      0.3, 1.3, 2.3, 3.3}
u = m._m11_m23;                         // u = (1.1, 2.3)
u.yx = u;                               // u = (2.3, 1.1)
```

are executed once for each data entity (vertex, primitive or fragment). If a program is bound to a profile that uses functionality not supported by the profile, an error occurs. The Cg compiler only compiles the code that can be executed from a certain starting point. This feature allows to write several shaders into the same file that can be called depending on the supported profiles [99]. Furthermore, it is possible to tag functions with profiles so that it is possible that the right function is called through function overloading depending on the used profile [99].

The general structure of a *Cg* program is that of a serial procedure although it is executed simultaneously for several data entities. Input and output are modeled as data structures or parameters that are modified inside the function. It is also possible to call other functions, to use branching and to use control loops, although the latter can be limited by the used profile (for example, simpler profiles prohibit loops that cannot be enrolled at compile time) [90].

Other specifics of the *Cg* syntax in contrast to CPU-languages are binding semantics. These are extensions for types that "bind" them to hardware features. For example, this allows to define that some input parameter shall contain the geometric data of the input vertices or that an output vector shall be treated as a color. The most common binding semantics are position, color and texture coordinates. Another *Cg*-specific keyword is *uniform* which marks a parameter as rarely changing (constant) so that it can be placed optimally in the memory [99]. Parameters in *Cg* are further marked as IN or OUT. Input is always read from the in-parameters and processed data is written to the out-parameters. Due to the widespread use of vectors and matrices in graphics processing, *Cg* supports additional operations for these types as illustrated in listing 3.

Cg provides many functions to process data, organized in a standard library. Many of these functions are specific for the processing of graphics data like vectors and matrices. These

include for example dot-product, vector-product, normalization of vectors, calculating the length of a vector or matrix-multiplication. Other examples of standard *Cg*-functions are *lerp* which blends two values depending on a weighting factor, functions for texture lookups, mathematical functions like sine or cosine or functions to calculate reflected or refracted vectors in one step [90].

Examples for shader programs written in *Cg* can be found in the appendix: the vertex shader for focus wobbled splat rendering (see listing 4), the vertex shader for voxel wobbled splat rendering (see listing 5), the vertex shader for focus wobbled splat rendering using a noise texture for the wobbling (see listing 6) and the vertex and fragment shader for ray casting (see listing 7).

2.8.4 Nvidia CUDA

Compute Unified Device Architecture (CUDA) is an extension of C/C++ developed by NVIDIA with the intention to allow developers to "*focus on parallel algorithms*" and "*not on the mechanics of a parallel programming language*" [100]. It is a framework that allows to develop homogeneous applications that are executed on the CPU where flexibility and serial processing is required and on the GPU where parallel processing of much data is possible. The intention behind this is to allow scientists to use graphics hardware as high-performance co-processors without having special knowledge about this hardware or the required graphics APIs. NVIDIA, as the provider, extended through this the target audience of its hardware from gamers and multimedia designers to scientists of all possible fields. CUDA is therefore the result of the GPGPU trend, already widespread at the time of CUDAs introduction.

CUDA is not just another library or API, it consists of its own compiler driver [100] that understands C++ and the newly introduced CUDA extensions and compiles everything with its respective compiler (cudacc, g++, cl, ...) [101]. The vendor describes this compilation architecture as illustrated in Fig. 25. The nvcc compiler driver separates CPU and GPU code and invokes the necessary compilers so that traditional C/C++ code as well as PTX object code for the GPU is generated.

CUDA introduces new function modifiers for the creation of such distributed applications that allow developers to tag functions as [101]:

__global__ This tags a function as being top level on the GPU. Global functions can be called by the CPU and must return void. These are the starting points (main methods) of the so called kernels (see later).

__device__ This tags a function as being a subroutine on the GPU. Device functions can only be called by other device or global functions. They are used to build structured and reusable GPU code and avoid duplication of code, unreadable algorithms and routines with too many lines.

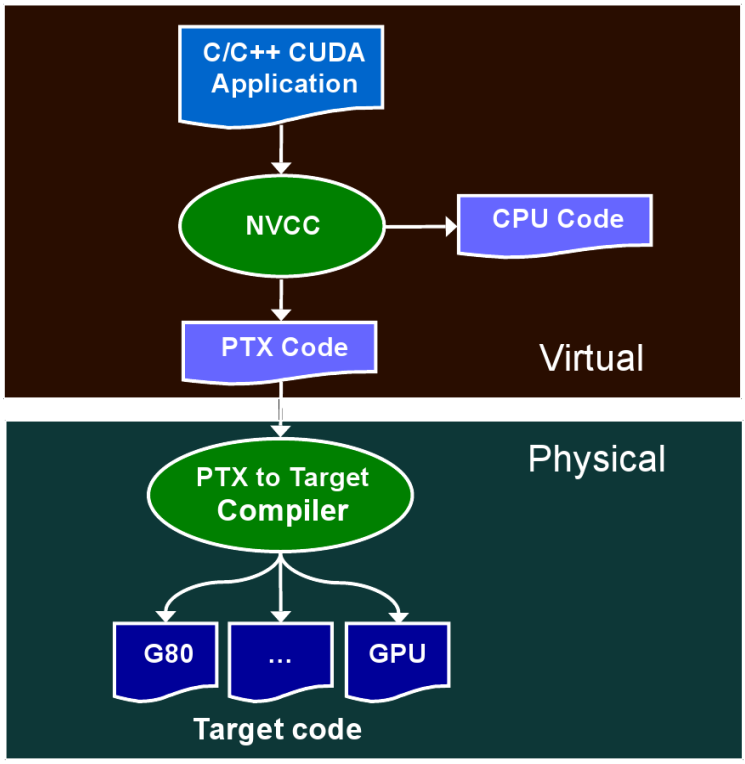

Fig. 25: NVIDIA's nvcc compiler driver that allows to compile CUDA code. CPU and GPU code is separated and both are compiled with their respective compiler. GPU assembly (PTX) is then transferred to the GPU [100].

__host__ This (or no modifier at all) tags a function as CPU only. Host functions behave like normal C/C++ functions and can call global functions (as kernels — see later) to start calculations on the GPU.

__host__ and __device__ This tag causes the compiler to generate GPU and CPU code. This allows to execute algorithms if possible on the GPU (if required features are supported) or otherwise on the CPU.

CUDA uses the concept of kernels to execute functions (algorithms) on the GPU. A kernel is an enclosed piece of code that is intended to be executed on a huge amount of data while each unit of data can be processed independently. Kernels (that execute __global__ tagged functions) cannot access the normal memory of a PC. Therefore, it is required to transfer all necessary data into the video memory before executing them. After the execution has finished, the data can stay in the video memory for further processing (like when using multi pass shaders) or has to be retransferred back to the "normal" memory.

CUDA kernels are executed on blocks consisting of threads. The term thread in the context of CUDA is similar to a thread on the CPU: it is the execution of one path in the code. In CUDA, threads execute concurrently on the same code in contrast to CPU threads, that normally execute different tasks. The only difference between threads is, that every thread has its own index that can be accessed using the *threadIdx* keyword [102]. Threads are again grouped in blocks. Also the index of the block is transparent to the thread (using the *blockIdx* keyword). Therefore, if data is stored in an array, every thread can access its own unit of data to apply parallel processing to the data by calculating the unique index consisting of *thread-index* plus *number of threads per block* times *block-index* [102]. Every block consists of the same number of threads which allows them to be synchronized and to share data.

The memory of the GPU in CUDA is divided into three different types [102]:

Local Memory Every thread has its own local memory to read and write. This memory cannot be shared among threads.

Shared Memory Threads of the same block have access to a shared memory. This can be used to communicate and it allows threads of the same block to be synchronized.

Global Memory Textures and global constants are stored in a global memory that can be accessed from all blocks of a kernel and also from other kernels. It allows to pass data from one algorithm to another without transferring data between GPU and CPU.

CUDA is today already very popular in different scientific fields because it is easy to use and allows to increase the performance of many different algorithms. Many papers describe this performance gain of which some shall be presented in the following.

Greef et al. [103] show in their work that ray casting can be accelerated in contrast to a

multithreaded CPU implementation by a factor of 2.1 – 10.1. They also show a speedup in dose calculation for radiation therapy planning when using CUDA.

Muyan-Ozcelik et al. [104] show in their work a CUDA implementation of the Demons deformable registration technique. They achieved with their implementation a speedup factor of 55 in contrast to CPU implementations of the same algorithm.

Stone et al. [105] show in their work the acceleration of MRI reconstructions using CUDA. This allowed a performance gain of about 20% while reducing the error rate form 42% to 12% by using more advanced reconstruction techniques.

Other exemplary fields where CUDA was used to increase performance are for example belief propagation for cloud tracking and reconstruction [9], pathfinding [106], molecular orbit computations [107], analysis of financial markets [87] or detection and tracking of white blood cells in video microscopy [10].

2.8.5 Other Shading or GPGPU-Programming Languages

Beside Nvidia's *Cg* (see Sect. 2.8.3), there exist other languages for programming GPUs. The most prominent representatives of them are *OpenGL's GLSL* and *Microsoft's HLSL*. OpenCL in contrast is provided by the same company as OpenGL and is a competitor to Nvidia's CUDA as GPGPU language.

OpenGL Shading Language (GLSL), at the moment in version 1.5 [97] is a language for GPU programming and in principle similar to NVIDIA's *Cg*. It is part of the OpenGL product family, has its own specification [108] and is maintained by the Khronos Group.

Beside OpenGL, Direct3D is the other major 3D programming API, which is part of the DirectX API provided by Microsoft. It is at the moment in version 11. It features normal fixed function pipelines, programmable vertex, geometry and fragment (pixel) shader and the newest features (that are not supported by OpenGL) such as tessellation and multithreaded rendering and resource creation [95]. The shading language for the Direct3D API is called HLSL and is provided by Microsoft. HLSL is in its syntax and usage similar to Cg or GLSL.

OpenCL, maintained by the Khronos Group like OpenGL and GLSL, is a language for "*general purpose parallel programming across CPUs, GPUs and other processors, giving software developer portable and efficient access to the power of these heterogeneous processing platforms*" [109]. It is in syntax and usage similar to NVIDIAs CUDA.

3 The "Real-time 2D/3D Image Registration in Radiation Oncology" Project - Status of July 2009

This section presents the current project of Birkfellner et al. at the Center for Medical Physics and Biomedical Engineering (CBMTP) of the Medical University of Vienna with the status of July 2009. The topic of this project is "**Real-time 2D/3D image registration in radiation oncology**". It was started in 2008 and is planned to have a duration of about three years. It offers work for master as well as PhD students and is a cooperation of the Medical University of Vienna and the university of applied sciences FH Technikum Vienna. The project is funded by the Austrian state through the Austrian Science Foundation (FWF), the official institution for funding scientific research in Austria (`http://www.zbmtp.meduniwien.ac.at/index.php?id=101`).

The ambition of this project is to create a software suite that allows to perform 2D/3D-image registration on common consumer hardware (including graphics hardware but also clusters of several consumer PCs) with a frequency of about 5 Hz. This shall allow online tumor motion detection in image guided radiation therapy (IGRT — see Sect. 2.1.1). The reason why 5 Hz was chosen as the goal frequency for this project is that this is the frequency of the kilo-voltage imaging device of the linear accelerator at the general hospital in Vienna.

The main focus for improving 2D/3D-image registration according to the official project description lies on the following tasks:

- Wobbled Splatting as described by [20, 1] shall be further improved or replaced by more efficient DRR rendering methods. This thesis is the main attempt to fulfill this task.
- Existing merit functions shall be further investigated and the newly defined merit function of (stochastic) rank correlation [3] shall be evaluated to find efficient and robust ways to compare images within short calculation times and that have smooth characteristics so that the optimization process converges as fast as possible.
- Reduction of number of degrees of freedom from six (three translations and three rotations) to just five as described in [110] to allow faster convergence.
- Simple to implement parallelization of the software suite on standard hardware by using Linux kernel extensions called MOSIX (`www.mosix.org`). This shall allow to calculate certain steps of the registration process in parallel with the help of a transparent layer (only few changes in the code). The final goal would be to use cluster consisting of eight consumer-PCs for the whole software suite.
- Extending the use of GPGPU in the process. As started by Spoerk et al. [1] to achieve faster DRR rendering, this technology shall be further exploited for the rendering process and maybe also be used to accelerate additional parts of the registration procedure. CUDA,

a GPGPU library provided by NVIDIA shall be investigated to fulfill this task.

- Extension of the software suite to also allow non-rigid registration (this is not a task for the real-time aspect of the project, but shall improve the software suite by extending the application domain).
- Evaluation and optimization of the whole software suite for clinical problems.

The following sections shall describe the architecture of the software suite with the status of July 2009, it shall in detail explain the at that time used DRR rendering method of GPU based wobbled splat rendering as described by Spoerk et al. [1], present the quality and speed of this solution and discuss the tasks at that time to further improve the solution.

3.1 Design and Features of the Software Suite

This section presents the design and features of the project's software suite at the status of July 2009. At this time, it was possible for the first time to perform under certain conditions a registration within a few seconds [111].

The whole suite is designed as an object oriented system where the three steps of the registration process were encapsulated in separated classes. The DRR renderer computes images that are used by the merit function to calculate a similarity measure that is used by the optimizer to calculate new parameters for the DRR renderer. See Fig. 26 for the structural connections and Fig. 27 for the functional connections between these components.

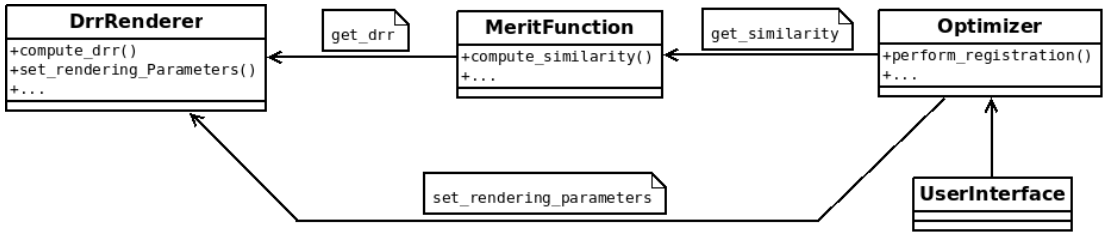

Fig. 26: General overview of the different components in the software suite. The user interface accesses the optimizer that sets the parameters to the renderer and calls the merit function. The merit function drives the computation of the DRR and calculates the similarity measure against the reference image. This value is used by the optimizer to adapt the six degrees of freedom for the rendering until an optimum of the similarity is reached.

All of the components from Fig. 26 have their own inheritance trees representing different types and implementations of DRR renderer, merit functions and optimizers. This allows the free combination of the different techniques to have the flexibility to choose the best implementation for a concrete registration problem instance.

Until July 2009, only wobbled splatting was implemented (the version developed by Spoerk et al. [1] that is described in Sect. 3.2) for the CPU as well as for the GPU. The software suite already

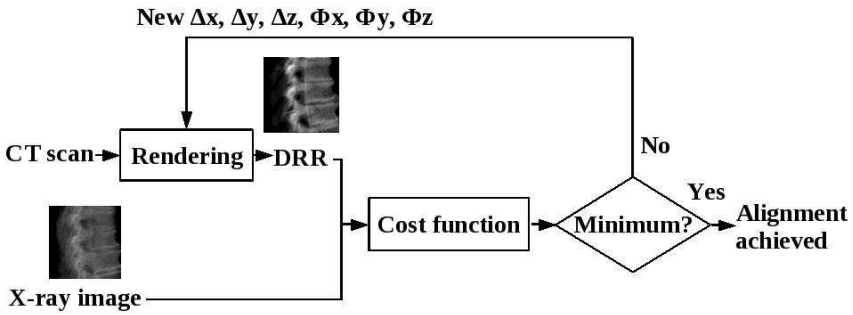

Fig. 27: The registration framework of the software suite and its workflow as described by Gendrin et al. [111] A DRR is rendered with iteratively adapted parameters and compared to the reference image until an optimum in similarity is reached.

provided cross correlation (Sect. 2.4.3), pattern intensity (Sect. 2.4.2) and rank correlation (Sect. 2.4.4) as merit functions and allowed to optimize using the downhill simplex (Sect. 2.5.1), the conjugate gradient (Sect. 2.5.2) or the simulated annealing (Sect. 2.5.3) algorithm.

3.2 GPU based Wobbled Splat Rendering until July 2009

To have a good idea of the original DRR rendering method used in the project, this section shall describe in depth this algorithm, its implementation and performance with the status of July 2009. All information given here has originally been published by Spoerk et al. [1].

The algorithm is based on the work of Birkfellner et al. [20] and was developed as a case study on how easy and in which ways the GPU can be utilized for 2D/3D-image registration. In general, it is a splat rendering approach with many simplifications to achieve short rendering times at the cost of image quality. It is based on geometric projection. This step is mathematically defined as

$$\vec{x}_p = P \cdot V \cdot \vec{x} \tag{25}$$

with

$$V = \begin{pmatrix} r_{00} & r_{01} & r_{02} & t_x \\ r_{10} & r_{11} & r_{12} & t_y \\ r_{20} & r_{21} & r_{22} & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad \text{and} \quad P = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{-f} & 1 \end{pmatrix}$$

where $\vec{x}$ is the original voxel and $\vec{x}_p$ is the projected voxel that can be interpreted as a fragment (see Sect. 2.8.1) that only has to be blended with other fragments projected onto the same

position to generate the DRR. The indexed rs are the coefficients for the rotation while the ts represent the coefficients for the translation of the volume. The parameter f is the focal width that influences the perspective.

This algorithm was implemented using C++, OpenGL and NVIDIA's Cg. The high level algorithm design is illustrated in Fig. 28. Pre- and post-processing of the data is done on the CPU. After loading all data to the GPU, a vertex shader is executed on each voxel (represented as a vertex with the primitive type point) and is projected according to Eq. 25. This vertex is rasterized and blended using OpenGL. After the shader has finished, data is extracted and post-processed (low-pass filtered and intensity scaled). The shader code for this algorithm is shown in the appendix of this thesis.

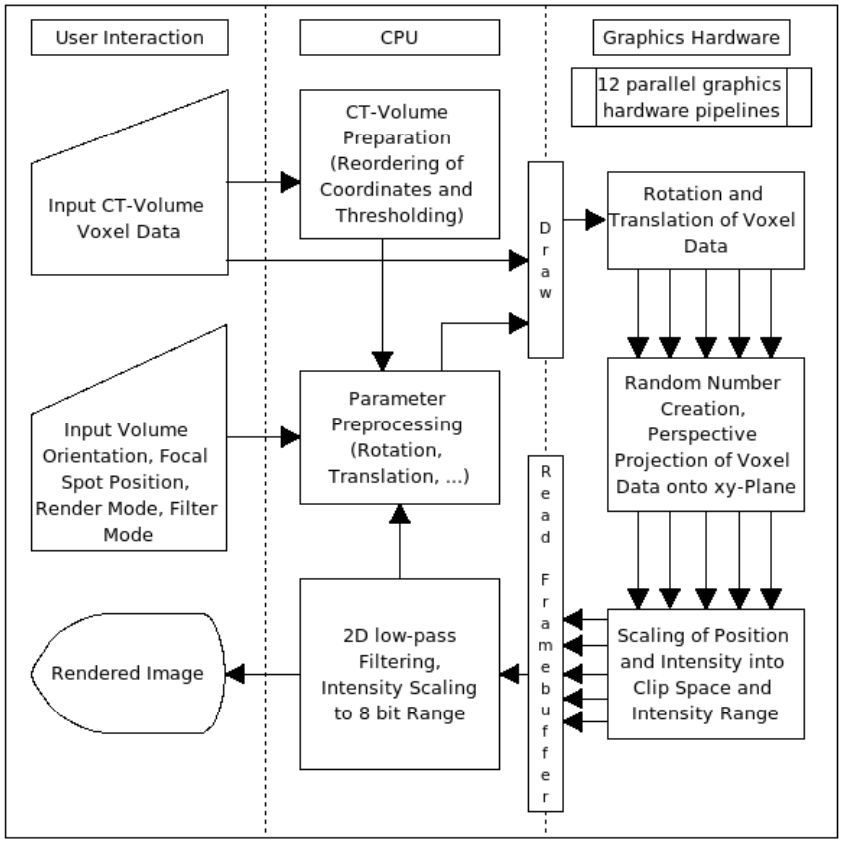

Fig. 28: Flowchart illustrating the workflow of the GPU based wobbled splat rendering algorithm shown by Spoerk et al. [1]. After a preprocessing step on the CPU, the data as well as the parameters are transferred to the GPU where they are processed in parallel in a vertex shader. Results are stored in the framebuffer and composed by OpenGL's alpha blending. The composed image is extracted, post-processed (e.g. low pass filtered) and displayed.

This algorithm allows to render DRRs very fast, but these DRRs suffer from eye-catching high-frequency artifacts, because also the regular grid of the volume is projected (see Fig. 10). To

avoid these artifacts, wobbling, i.e. stochastic noise, is introduced to the focal position (focal wobbling) or the voxel positions (voxel wobbling) to blur the image and therefore eliminate the artifacts [20]. Because generating Gaussian distributed random numbers as used by the original publication [20] is a very time consuming task, Spoerk et al. [1] proposed a technique based on sine computations as "random" number generator (although they are not even pseudo random numbers). The result of the equation

$$r = \quad D \cdot \sin(p_i \pm p_j) \cdot \sin(p_k \pm p_l) \tag{26}$$

with D representing the maximum displacement and $p_{i,j,k,l}$ representing voxel coordinates and intensities in varying combinations, produces values more or less Gaussian distributed as can be seen in Fig. 29. This simplification allows to further accelerate the algorithm. This task has a huge influence on the performance because it is required that every voxel computes its own random number (due to the parallel design of shader programs). This problem can also be solved by precomputing the random numbers as described in Sect. 4.1.2.

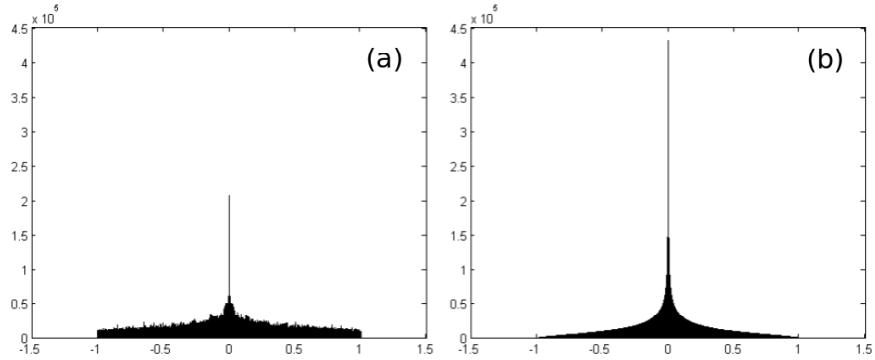

Fig. 29: "Random" number generation with approximated Gaussian distribution as shown in Spoerk et al. [1]. The result of calculating the product of two sines of different combinations of voxel positions and intensities is shown in (a) while (b) shows approximated Gaussian distributed random numbers produced using the Box-Muller method [112]. There are differences, but they can be ignored for the application in the wobbled splat rendering algorithm.

This wobbled splat rendering implementation produces images of moderate quality, which is tolerable because high quality with many features can even be counterproductive for 2D/3D-image registration. Some examples of the image quality are shown in Fig. 30.

At the time the algorithm and its implementation for the GPU were published (2007), it showed a reduction in rendering time in comparison to the CPU implementation of Birkfellner et al. [20] of 30–90 % on an Intel CoreDuo 2 x 2,4 GHz with 1 GB Ram and an ASUS Graphics Card using an NVIDIA GeForce 7600 GS GPU. The graphics card had 512 MB memory and 12 hardware

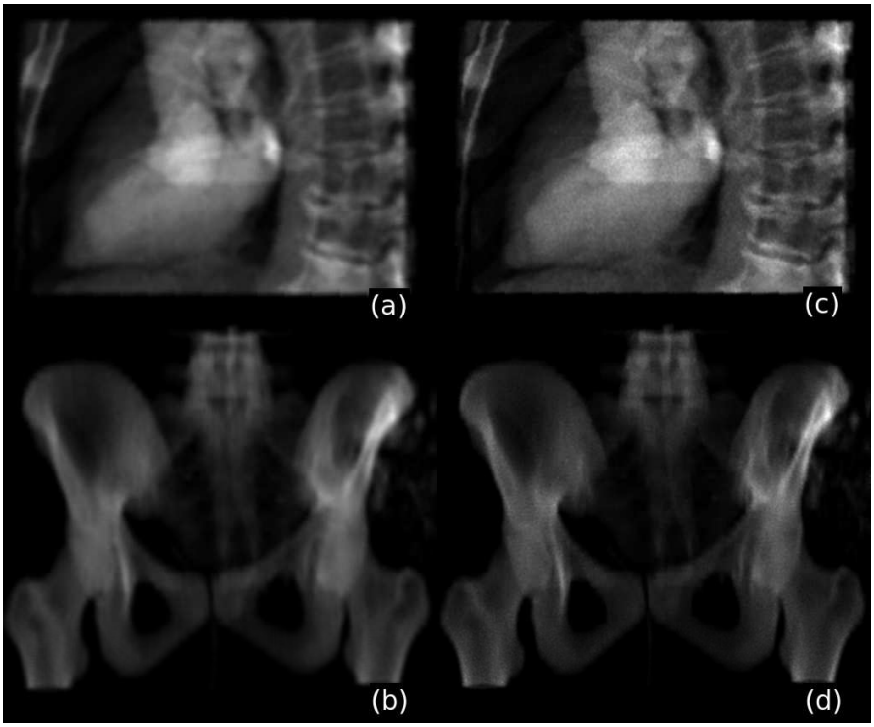

Fig. 30: Comparison of image quality of CPU based wobbled splatting (a,b) and GPU based wobbled splatting (c,d) [1]. The GPU implementation produces more noisy but sharper renderings. All four images were post-processed with a 3x3 Gaussian kernel filter for antialiasing.

graphics pipelines with 5 vertex- and 12 fragment processors. The results in comparison to CPU based wobbled splatting on the same machine are illustrated in Fig. 31.

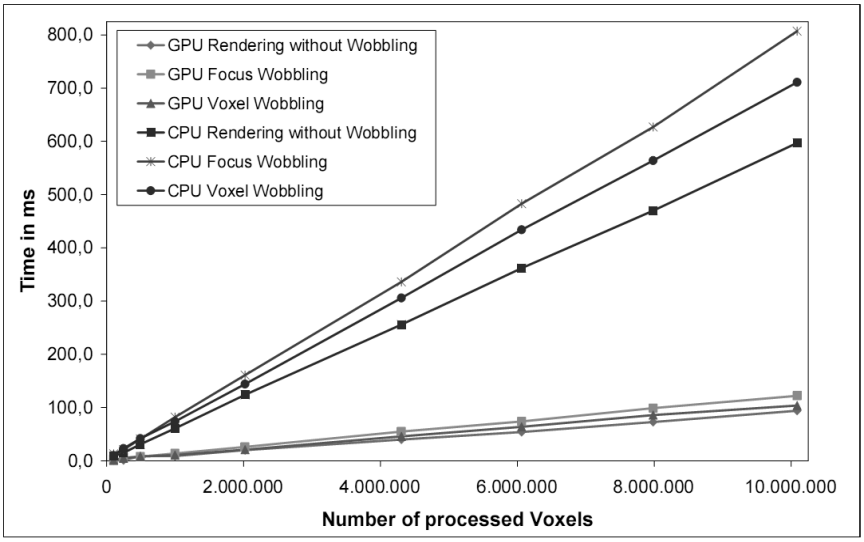

Fig. 31: Comparison of runtimes between CPU and GPU based wobbled splat rendering [1]. It shows the rendering times for the pelvis shown in Fig. 30. GPU-based rendering is remarkably faster then CPU-based rendering independent of the number of processed voxels.

3.3 Registration Performance and Quality

The above discussed DRR rendering approach was in use for achieving the first results of the project that were published by Gendrin et al. [111]. For evaluating the quality of the DRRs created using GPU based wobbled splat rendering, they were compared to renderings of the same data using a implementation of wobbled splat rendering utilizing the CPU. It is shown that there are differences in the contrast (see Fig. 32) and it was concluded that this is a result of the different scaling approaches. In the CPU algorithm, values are summarized and scaled in a post-processing step while in the GPU algorithm, scaling has to be performed before the rendering to avoid saturation of the framebuffer and therefore results in incorrect intensities.

In terms of registration quality and registration time, Gendrin et al. [111] shows following results:

- GPU based DRR rendering in comparison to CPU based DRR rendering divides the registration time by a factor of 3–8.
- The computation of the merit function on the CPU, the number of iterations of the optimizer and the transfer of data between CPU and GPU are bottlenecks of high speed registration.
- In terms of registration quality, stochastic rank correlation (SRC) performs better than

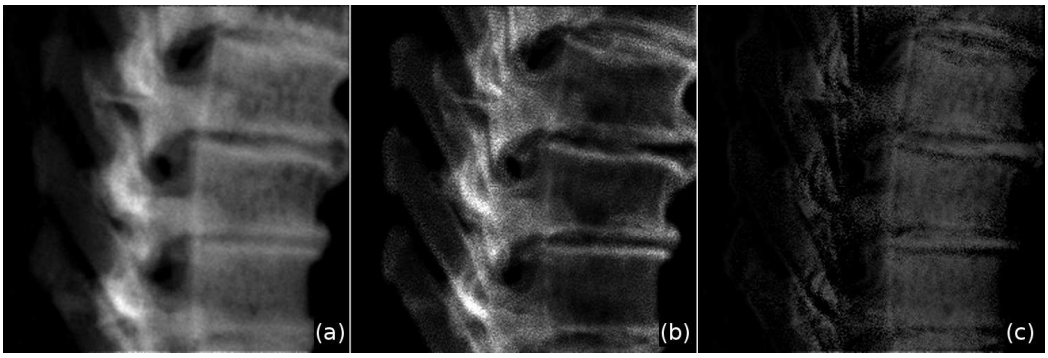

Fig. 32: Comparison of the quality of wobbled splat rendering [111]. (a) shows the result of the CPU implementation of this algorithm, (b) shows the result of the GPU implementation and (c) shows the difference of the two DRRs. The big difference is a result of the different scaling approaches of these algorithms.

cross correlation (CC).

- Stochastic rank correlation (SRC) is more robust to non-linearities between base image and matching image in the similarity measurement. Therefore, SRC is better with GPU based wobbled splat rendering than CC.
- On-line tumor monitoring based on 2D/3D-image registration shall be possible (with further software development and better hardware and clustering). At the time of publication, the whole procedure required about three seconds.

3.4 Topics for Improvement in July 2009

At the moment, several people work in this project on different aspects. One general goal is to further debug, refactor and improve the software suite.

This thesis presents the work done to further improve the rendering step of the software suite. Details about this can be found in the following chapters (Chap. 4, 5 and 6). In general, implementing a GPU based ray caster that supports sparse sampling and further improves the GPU based wobbled splat rendering algorithm are the two main goals in the field of rendering.

In the field of merit functions, implementing new methods like mutual information (Sect. 2.4.1) or correlation ratio (Sect. 2.4.5) are the main focus. Another topic is to further evaluate all merit functions to improve there robustness and efficiency (that they provide optima that can be easily found).

In the field of optimization, the main goals are improving the existing methods. Furthermore, possibilities to parallelize them have to be evaluated so that the computation can be distributed in a cluster environment.

In addition to this project, there are also attempts to produce a new phantom data set for

image registration including CT scans, different MRI scans and fluoroscopy images of a porcine head [113]. This sub-project was created because there are at the moment only a few standard datasets for evaluating registration methods publicly available and most of them only contain boney structures and no soft tissue.

Because of the first results of the project being available [3, 111, 113, 114], the next step is to create the connection between the available LINAC hardware and the software suite to test the online evaluation of the patient's pose.

4 Methods

The following sections present the work done to optimize the rendering process of 2D/3D-image registration. The main tasks were on the one hand to optimize the existing wobbled splat rendering approach of Spoerk et al. [1] and on the other hand to implement a ray casting solution for the existing software suite that is in addition capable of sparsely sampling as described in Sect. 2.6.

The main purpose of this thesis is to find a faster method for rendering Digitally Rendered Radiographs (DRR). Beside this performance criterion, existing and new solutions shall be evaluated in terms of image quality and fitness as input for the registration process, because the rendered images can have a huge influence on the overall optimization performance and quality.

This chapter presents the improvements and developments that were implemented within the scope of this thesis. Chap. 5 presents the results of all the work which is then discussed and analyzed in Chap. 6.

4.1 Further improvements in GPU based Wobbled Splat Rendering

The original version of the algorithm for GPU based wobbled splat rendering as implemented by Spoerk et al. [1] was already presented in detail in Sect. 2.3.2 and Sect. 3.2. Beside larger changes to this algorithm introduced in the following sections, many small optimizations were carried out. These include code cleanups, restructuring of the class hierarchy to include the new rendering methods, shader simplifications and corrections in the transformation operations so that these transformations perfectly match the real world coordinate systems.

4.1.1 Artifacts

During the improvement of the algorithm implementation, an error still present in the design of focus wobbling was discovered which is illustrated in Fig. 33. Focus wobbling varies the focal position to remove high-energy artifacts that are the result of projections of the regular voxel grid. Caused by the order of the matrix multiplications that apply transformation and projection to each voxel position, the focal position is varied before the transformation which results in no wobbling of the volume in its z-direction. Voxel wobbled splat rendering applies the wobbling to all three dimensions and therefore does not suffer from this problem. To switch therefore to voxel wobbling is no solution, because focus wobbling is the physically more correct version of the algorithm. A solution to this z-orthogonal wobbling error is to vary the focal position in focus wobbled splat rendering additionally in the z-direction. This removes the artifacts but is only possible by generating an additional random number. This either increases the runtime or the memory consumption of the implementation.

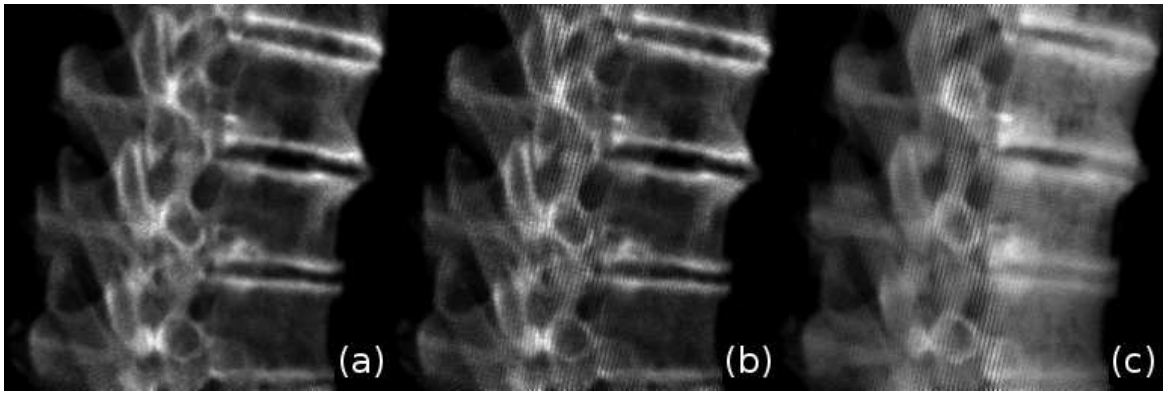

Fig. 33: Example for the z-orthogonal wobbling error of the focus wobbled splat rendering algorithm. All three images show the spine dataset rendered from a view orthogonal to the z-axis. While wobbled splat rendering using voxel wobbling (a) shows no high-energy artifacts, GPU (b) as well as CPU (c) based wobbled splat rendering with focus wobbling suffer from significant artifacts.

4.1.2 Faster Pseudo Random Number Generation for Wobbling

One main feature, but also an issue, in the GPU based wobbled splat rendering algorithm is the pseudo random number generation which is required for wobbling the computed DRR. To avoid high-energy artifacts, the focal or voxel positions are modified stochastically. As described in Sect. 3.2 and illustrated in Eq. 26 as well as Fig. 29, the required random numbers for this displacement are not even pseudo random numbers but just the product of two sines with different parameters. This way to compute the displacement is deterministic and depends on the voxels' positions and intensities, but it allows to simulate Gaussian distributed random numbers while preserving the performance. In fact, it always results in the same displacement for one volume.

The problem with this approach is on the one hand that the displacement depends on the voxels' position and intensity and that it was never evaluated how strong it influences the performance of the rendering. It can be suspected that the two sine calculations for each of the three directions could eventually be a bottleneck of the algorithm.

One approach to avoid the sine calculations is the use of a noise texture. Instead of calculating the random numbers on the fly in the vertex shader, they are calculated once in the preprocessing phase and stored in a three-dimensional texture. This allows every voxel to access its own texture position while the three color channels of the texture encode the three different random values for the wobbling. In order to access the correct value, the texture is interpolated with the nearest neighbor method. Instead of sine calculations of combinations of different parameters, a single texture fetch is used to produce the random numbers. An advantage is that because the texture is produced in the preprocessing phase, the random numbers can be generated with

a approximated Gaussian distribution using the Box-Muller method as described in Press et al. [112]. In general, this method exchanges memory consumption for possible speed.

Experiments show that one large texture that encodes random numbers for the whole volume is not applicable in practice. When assuming a volume of 512^3 voxels and storing three random numbers as floats (4 bytes per number) for each voxel, this would result in a texture with a memory requirement of 1536 Megabytes. A solution to this problem is to fix the size of the noise texture and reuse it for different parts of the volume. Wobbling is required to destroy the regular voxel grid of neighboring voxels but has not to be totally different on a global scale. Fixing the texture to 64^3 for example results in a much smaller memory consumption and, as a side effect, also increases the performance, because the video memory is used more efficiently.

The second approach to replace the current random number generation is not to avoid the sine calculations but to reduce their complexity in order to achieve a speedup. If angles are small, the sine of these angles can be approximated with Taylor series [115]. Because the quality of the random numbers is not really relevant for the wobbling, it is even possible to only evaluate the first or the first few steps of these Taylor series. This can also be an alternative to avoid cost intensive sine calculations.

4.1.3 Vertex Buffer Objects for Data Transfer Reduction

The biggest drawback of the GPU based wobbled splat rendering algorithm as implemented by Spoerk et al. [1] was always the way, the data is transferred to the video memory of the graphics hardware. Because of the object-space based design of wobbled splat rendering, it is required that a shader is called exactly once per voxel. This prohibits the use of the fragment shader. The projection has to be performed in a vertex shader, which implies that all voxel data has to be rendered as points that are processed by this shader. The result of this is that for each rendering, the whole volume data set has to be transferred to the video memory, with at least four floating point numbers (three coordinates and one intensity) each requiring four bytes per vertex.

Volumes consist in general of millions of voxels, which results in a data transfer rate of megabytes per rendering. When assuming a transfer bandwidth of *8 gigabyte per second* as with a *PCI-e x16* bus and a volume with *10 million voxels*, this leads to a hypothetical overhead of about *5 milliseconds per rendering*.

The solution to this performance bottleneck is the use of so called vertex buffer objects. Vertex buffers are OpenGL objects that can be used to store vertex information. When specifying the buffer as *STREAM_DRAW*, the specified vertex data is immediately transferred to the video memory and stored there persistently. This data remains there and can be drawn as often as required without having to be retransferred. Because the vertex data for the wobbled splat

rendering shader do not change over time, this is the perfect solution to overcome the data transfer bottleneck. In order to include this into the algorithm shown in Fig. 28 the workflow has to be adapted as shown in Fig. 34. For each rendering, only the projection and transformation information has to be retransferred to the graphics card, which is intended, because this data can differ from rendering to rendering.

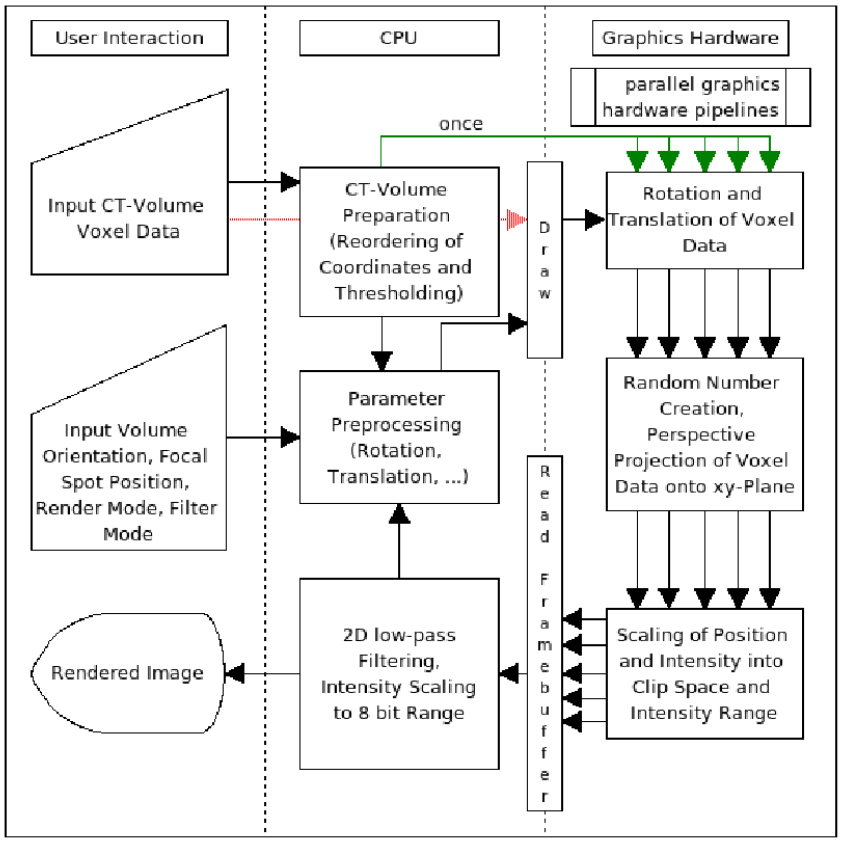

Fig. 34: Adaption of the GPU based wobbled splat rendering algorithm shown by Spoerk et al. [1]. Instead of transferring the whole voxel data repeatedly to the graphics hardware to initiate the data processing, data is stored as a vertex buffer object, which is used to transfer the data once to the graphics card and to store it there. It can then be rendered as often as required. Only the varying projection and transformation information has to be retransferred for each rendering. The red connection was therefore removed and replaced by the green one.

4.2 (Sparsely Sampled) Ray Casting

An alternative to the already well evaluated wobbled splat rendering algorithm [20, 1, 3] for 2D/3D-image registration is ray casting as described in Sect. 2.3.1. Much research into this field was already undertaken. In fact, ray casting is the default method for DRR generation in the literature, but there are virtually no studies that compare this method to other rendering algorithms and that compare their different influences on the overall registration process.

Therefore, a high performance ray casting algorithm is implemented and integrated into the existing software suite to be compared with the already existing, and within the scope of this thesis further optimized, wobbled splat rendering approach. The main goal is to evaluate the competitiveness of ray casting for high-performance DRR rendering in the field of 2D/3D-image registration.

4.2.1 CPU Implementation

As a base of operation, a CPU implementation of ray casting was developed. The main principle for this implementation was to use an octree structure to represent the volume data and a hierarchical refinement process to fastly evaluate which voxels are hit by a ray and which are not. The whole volume is, at the highest hierarchical level, represented as a single cube. This cube is divided into eight cubes of half the size for the second level from which every cube is again divided into eight smaller cubes for the next level. This is continued until the granularity of voxels is reached. All of these cubes have a reference to their children or hold the information that they have none. This allows to test hierarchically if a cube is hit and if not, to not further go down in the hierarchy and therefore reduce the number of hit-computations.

The test if a cube is hit by a ray can be computed using angle comparisons. This technique is shown in Fig. 35.

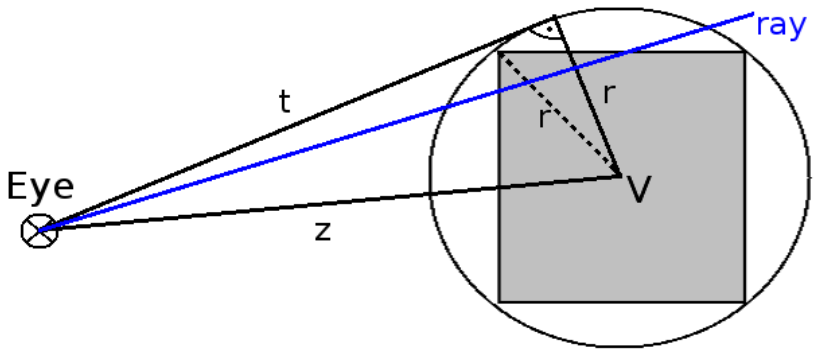

Fig. 35: Hit test by angle comparison. As a reference ray, the connection between the eye position and the voxel center is calculated. Then, the angle between the ray and the reference ray is compared to the angle between the tangent and the reference. Because the comparison of angles is identical to the comparison of cosines of angles and many values can be precomputed, this hit test can be computed efficiently.

The main principle is to compare the angle under which a voxel is seen from the eye position. As a reference ray, the connection between the eye position and the center of the voxel is calculated, which has only to be done once per rendering. Using the Pythagorean theorem, the square of the cosine of the angle between the tangent and this reference ray can be computed. The same measure can be computed between the ray and the reference ray using the scalar product.

Following these calculations, the "hit" criterion can be formulated as

$$(\vec{ray}_{norm} \cdot \vec{z}_{norm})^2 \geq \frac{\|\vec{z}\|^2 - \|\vec{r}\|^2}{\|\vec{z}\|^2} \tag{27}$$

with $\vec{ray}_{norm}$ being the normalized ray, $\vec{z}_{norm}$ being the normalized reference ray, $\|\vec{z}\|$ being the distance between the eye position and the voxel center and $\|\vec{r}\|$ being the radius of the voxel's circumcircle. Efficiency can be achieved because the right-hand side of Eq. 27 and the normalized reference vector from the eye to the voxel position has only to be computed once for each rendering. The normalized ray can be computed once for each pixel. As a result, only the comparison itself has to be done once per voxel and pixel.

The above formulated hit criterion is not an exact solution. If the angles between the reference ray and the ray and between the reference ray and the tangent are very similar, this method can detects a hit although it is a miss. Every voxel that passes this test has therefore to be further investigated in a second step. The ray is projected to the three coordinate planes and there, the hitpoint with the projection of the voxel is calculated in two dimensions. These values can then be used to calculate the intersections of the ray with the voxel in three dimensions. It is a simple technique that allows to calculate the exact positions where the ray enters and leaves a voxel and to determine the distance between these two points. This value can be used to improve the composing of the final image by adapting the opacity based on the length on which the ray goes through the voxel.

The ray casting itself is performed by using the recursive octree structure to test every voxel if it is hit and to compose the intensities of these voxels for each ray. A resulting DRR of this method is shown in Fig. 36 (a).

The main problem of the CPU based approach for ray casting is the very long rendering time. Rendering of a standard data set (see the spine in Sect. 4.4) with about 20 million voxels into a DRR with a resolution of 256^2 pixels requires for example about *6 seconds* on the test system described in Sect. 4.5. Even if a speedup of factor 100 would be possible by further improvements of the algorithm, the rendering time would for example still be more than twice of the rendering time achievable with GPU based wobbled splatting — see Sect. 5.2.

The second problem with this implementation is the interpolation. As illustrated in Fig. 36 (b), nearest neighbor interpolation is used that leads to aliasing artifacts and "pixelization" of the image. This problem can be overcome with implementing a trilinear interpolation, but this would further increase the rendering time.

Because of the very slow runtime, CPU based ray casting was excluded from all further considerations and investigations. It is neither competitive to a GPU based approach for DRR computation nor to CPU based wobbled splat rendering [20].

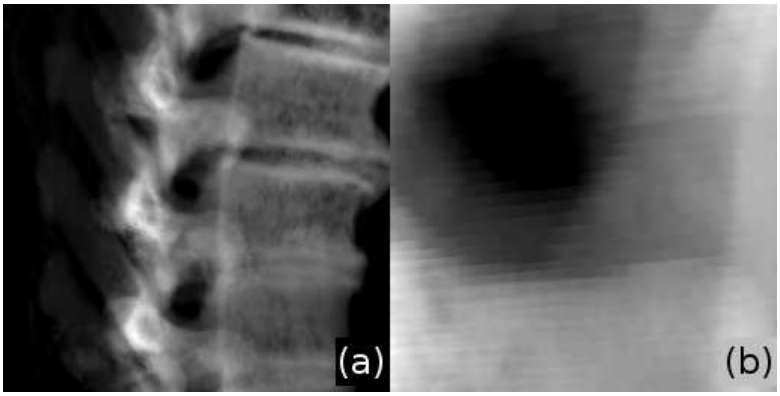

Fig. 36: DRR computed with the CPU based ray casting implementation. (a) shows that the rendering has fairly good quality when rendered with default parameters. (b) shows the pixelization effect that is the result of the nearest neighbor interpolation used for this method. Trilinear interpolation would remove the pixelization but increases the runtime.

4.2.2 GPU Implementation

Implementations of ray casting exploiting the high parallelism of graphics hardware have already been well investigated [8, 32, 34]. The main principles and several optimization techniques were already explained in Sect. 2.3.1.

The implementation for this thesis follows the descriptions provided by the above cited sources and is based on the online ray casting tutorial published by Peter Trier [116]. The ray casting is performed in two passes. The first pass renders the back faces of the bounding cube to a texture. The second pass renders the front faces, calculates the ray vector for each pixel by subtracting the front face and back face color and then performs the sampling along this ray between the front and the back faces. An illustration of the front and back faces for ray determination can be seen in Fig. 37.

This procedure for generating the ray vectors works because the bounding cube ranges from $(0,0,0)^T$ to $(1,1,1)^T$ and every vertex stores its coordinates also as its color. Therefore, when subtracting the colors of the front face and the back face at the same position, the ray vector for this position can be computed. These faces are shown in Fig. 37 (a) and (b). With the computed ray vector, the volume can in a next step be sampled and composed. This step can be directly included into the fragment shader of the front face.

The volume data itself has to be stored as a three-dimensional texture. This has the advantages that it has to be transferred to the video memory only once and that it can be accessed very efficiently. This texture can then be used to sample the volume data for composing the final pixel. No further coordinate conversions have to be done because the range of the colors between 0 and 1 in all dimensions is equivalent to the coordinate system of a volume texture.

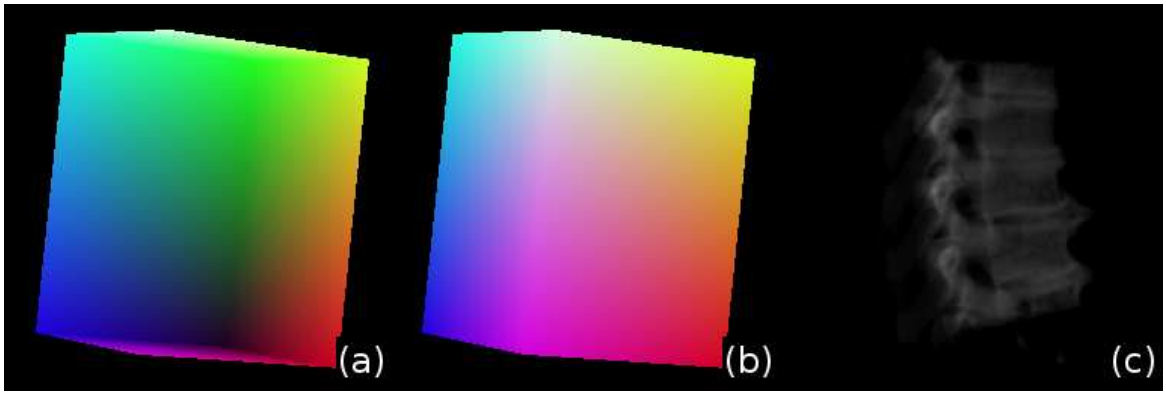

Fig. 37: Illustration of intermediate results of GPU based ray casting. (a) shows the back faces of a bounding cube for the volume while (b) shows the corresponding front faces. (c) shows the final image for the faces from (a) and (b) using the spine volume data specified in Sect. 4.4.

For each pixel, the ray is traversed by starting at the front face at the pixel's position and stepping in the direction of the ray. For each step, the texture is fetched and the intensity is added to the final intensity. When the back face is reached, the composed intensity is returned as the pixel's color. The step size of this sampling can be varied and it influences the rendering speed as well as the quality of the DRR. An exemplary result of this process is shown in Fig. 37 (c). A final overview over the composing procedure is given in Fig. 38. The code of the vertex and fragment shader for this ray casting algorithm can be found in the appendix.

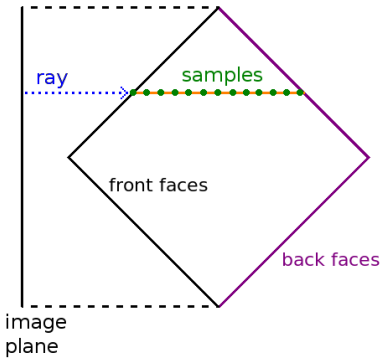

Fig. 38: Schematic illustration of the GPU based ray casting process. The ray for each pixel is calculated using the front and the back faces. This ray is then used to sample the volume texture and compose the final intensity.

This ray casting algorithm has many advantages. The volume has not to be ordered in a special way, it has to be transferred only once to the graphics memory, the intermediate back face texture can stay in the video memory between the two passes and the volume has only to be sampled between the front and the back faces.

A problem of this algorithm is that always the whole volume is sampled, even if it contains only

a few valid voxels. Another disadvantage is that the vertex shader is not used very much and therefore, computation power is wasted.

4.2.3 Sparse Sampling for Ray Casting

To further improve the performance of the ray casting implementation, sparse sampling was implemented. Many intensity based merit function for the similarity measurement can be performed on only a stochastically chosen subset of the pixels as already described in Sect. 2.6. Because of the image-space based design of ray casting, it is possible to only render the pixels that are required for the merit function evaluation by using a mask texture. This texture encodes the stochastically chosen positions that are used for the decision which pixels are rendered and which are skipped.

The exact implementation is presented in the appendix as an alternative fragment shader for the ray casting program. After computing the position which is later used to look up the color of the back face texture, this position is used to fetch the mask texture and to skip the rendering if the value at this position is zero.

Examples of sparsely sampled DRRs using 50% and 25% of the image content are shown in Fig. 39. The pixels are only chosen in the inner circle of the image because only this section is used for later merit function evaluation in order to avoid non overlapping regions.

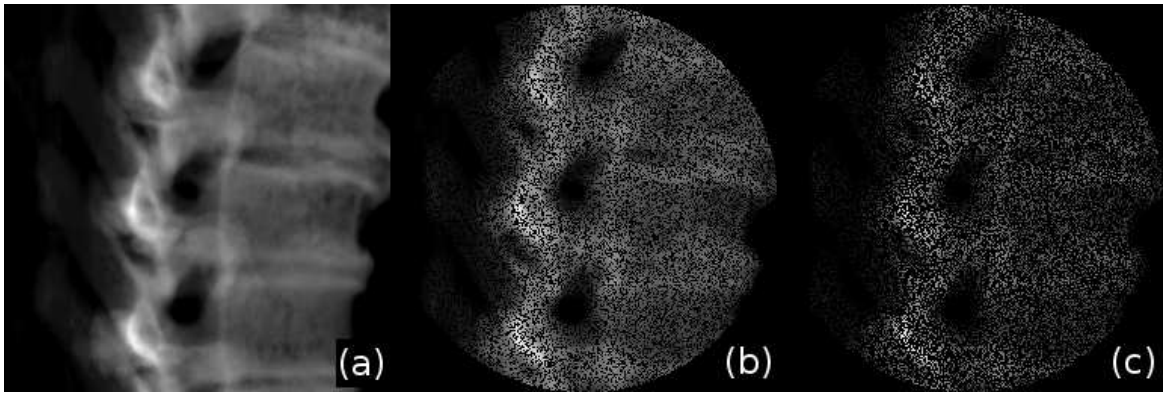

Fig. 39: Exemplary images of sparsely sampled DRRs as described in Sect. 2.6. (a) shows a normal ray casted DRR while (b) shows the same DRR with only 50% image content and (c) shows it with 25% image content. The pixels are only chosen in the inner circle of the image because only this section is used for the later merit function evaluation in order to avoid non overlapping regions.

4.2.4 Refined Bounding Structures

An important optimization technique for GPU based ray casting is the use of refined bounding structures as already described in Sect. 2.3.1. The motivation for this is to reduce the volume that has to be sampled, which results in a shorter rendering time. Although a more complex

structure has to be processed, this leads to a speedup because the bounding structure is handled by the vertex shader that is not used much yet [8]. The resources are better balanced between vertex and fragment shader and the pipeline design of the graphics hardware can be used more efficiently. Two methods of bounding structure refinement were implemented for comparison, both already presented in Sect. 2.7.

The cuberille approach (as already described in Sect. 2.7) was implemented in a simple and straightforward manner. The volume is traversed with a certain step size that represents how many voxels shall be summarized to form one cuberille. For each of these steps, it is evaluated if the cuberille contains valid data or it is skipped otherwise. If it contains data, the six neighboring cuberilles are checked for containing valid data. Between the current cuberille and all neighbors that do not contain any data, a quad is calculated representing the border and stored in a vertex list. After the whole volume has been processed in this way, the resulting list stores one or several closed surfaces of the valid data that can be used as a refined bounding structure for GPU based ray casting. Fig. 40 shows examples of such refined bounding structures that were produced using the cuberille algorithm with varying cuberille sizes.

Marching cubes (as already described in Sect. 2.7) is a more sophisticated method for generating surface models from volume data. The here presented implementation is based on the work of Cory Bloyd [117] that is available under the creative commons public domain license. The volume data is iterated and for every eight voxels, one of 256 possible cases is chosen depending on which voxels are valid and which lie outside of the threshold. Depending on this, the coordinates of corresponding triangles are calculated and added to a vertex list. In contrast to the base implementation of Cory Bloyd, all additional information like color or surface normals is not generated. The resulting surface list can then be used as the bounding structure of the valid data of the volume for GPU based ray casting. Fig. 41 shows the intermediate representations and the result of a ray casted DRR with a refined bounding structure produced using the marching cubes algorithm. Because of performance reasons, the refined bounding structure is stored using vertex buffer objects as already described for the wobbled splat rendering method — see Sect. 4.1.3.

4.3 Improvements independent from the Rendering Method

Beside the specific rendering methods, many other optimizations were included into the software suite of the project described in Sect. 3. Most of these optimizations were code refactorings and restructurings. These had less impact on the performance but increased readability, maintainability and reusability without decreasing the performance.

A change that had an impact on the performance was the migration to the *Linux kernel 2.6.31* in combination with the *ext4* file system. This change in the base infrastructure lead to an improvement in performance of about 10 %.

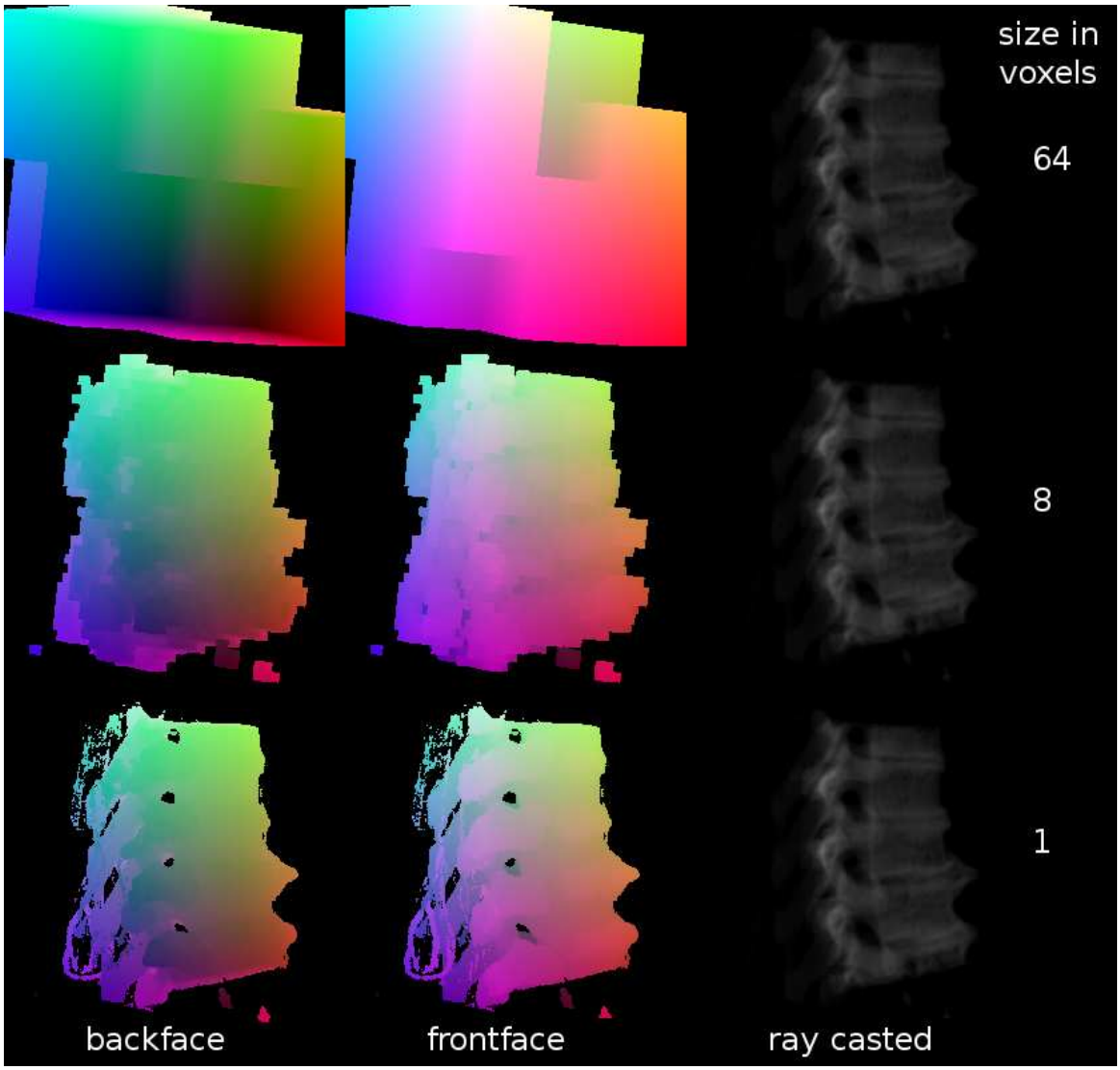

Fig. 40: Illustration of refined bounding structures that were generated utilizing the cuberille method. The first column shows the back faces of the bounding structures, the second column the front faces and the third column the final DRRs. The first row shows the rendering using a cuberille size of 64 voxels, the second row with a size of eight voxels and the third row with a cuberille size of one voxel.

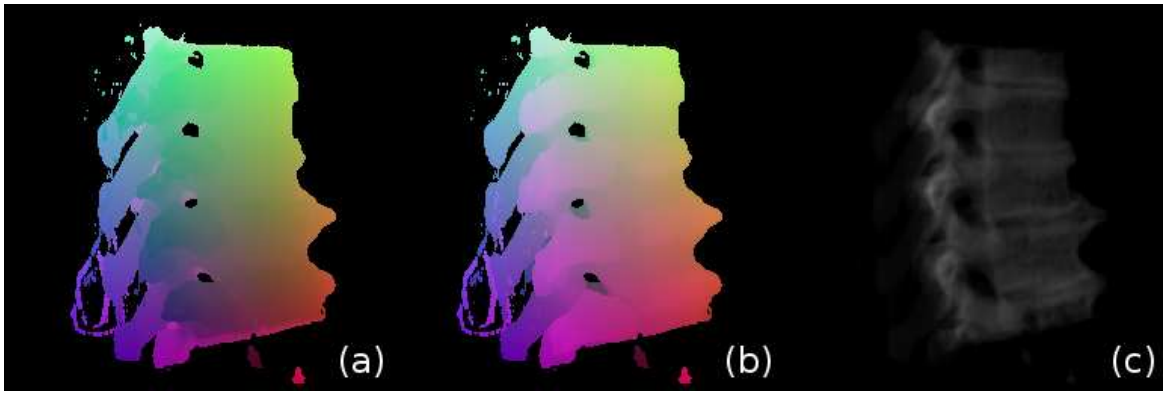

Fig. 41: Bounding structure generated using the marching cubes algorithm and the corresponding DRR. (a) shows the back faces of this bounding structure, (b) the front faces and (c) shows the final ray casted DRR.

4.4 Test setup and data

The different rendering approaches are tested in the following Sect. 5 with several setups to analyze them according to different questions.

- The first question is how the rendering methods differ in image quality. Therefore, DRRs are generated and compared with a gold standard image. For comparison, beside visual analysis, mutual information is used to quantify the similarity between the different images. Also exemplary images are shown that present some artifact specifics of the algorithms.
- The second question is about the rendering speed of the methods. Therefore, DRRs with differing transformations around a gold standard position of the volumes are computed using the different rendering algorithms and their optimization techniques and the performance is measured. These measurements are performed on two different datasets using different thresholds of the data to extract additional information.

  Every measurement is represented as the mean and standard deviation of 27000 renderings of the gold standard position whose rotation parameters are modified in the interval $[-180; 168]$ with a step size of 12. The translation parameters are fixed in order to not increase standard deviation through variations in the rendering times due to the relative position of the volume (an explanation for that can be found in Sect. 5.2, especially Fig. 49 and Fig. 50, and is discussed in Sect. 6.1 and 6.2). To show additional properties of the renderers, measurements of the time with varying image content, image sizes, cuberille sizes and percentage of rendered pixels have to be analyzed.
- The third question is about the influence of the rendering on the quality of the 2D/3D-image registration process. Rendering cannot be seen isolated but must be evaluated in the context of the registration. One dimensional merit function analysis is performed

for the different rendering methods that show the shape of the problem space around an optimum. Therefore, five of the six transformation parameters are fixed to the known gold standard values while the sixth parameter is varied around its gold standard position. This process is repeated for the other five parameters. The merit function shape can then be analyzed visually.

- The fourth and last question looks at the resulting overall registration performance and quality. Not only the rendering duration itself but also, as a result of the rendering quality, the performance of the whole registration process is possibly influenced by the rendering method. To measure this, registrations are performed and the time and the number of iterations are measured. In addition the mean projection distances (mPD) and the mean target registration errors (mTRE) of the results are calculated as a measure for the quality of the registrations. The mPD is the distance one point has between a projection with the found solution parameters and a projection with the known gold standard parameters. The mTRE represents the same measure but between the same point in the two volumes. These errors can be used to compare the influence of the rendering methods on the quality of the registration process and follow the terminology of Kraats et al. [118].

Following test-datasets are used for the evaluations of the rendering algorithms:

Spine The spine is a phantom consisting of vertebrae positioned in a water filled box. It was developed for evaluating 2D/3D-image registration methods [119]. The gold standard image is shown in Fig. 42 (a). The volume dataset has a width of 196 voxels, a height of 256 voxels and a depth of 196 voxels with an intensity range from 0 to 254. All voxels have a side length of 0.5 mm. The gold standard image has the resolution of 256^2 pixels and an intensity depth of 8 bit.

Pawiro Pig The Pawiro pig was especially developed as an alternative phantom dataset for 2D/3D-image registration. The specific characteristic of it is that it does contain a lot of soft tissue and not only bones. Soft tissue is far more complicated to register but is required if a tumor shall be the target of the registration. This dataset exists from many modalities such as CT, MRI (T1, T2 and Photon weighted) and has several gold standard images (kV and MV images) with relatively exact known transformation parameters. Gold standard kV images in anterior-posterior (AP) and lateral pose are shown in Fig. 42 (b) and (c). For all measurements of Sect. 5, the CT dataset and as gold standard the anterior-posterior (AP) pose were used, both reformatted and thresholded. The CT dataset has a width of 326 voxels, a height of 326 voxels and a depth of 330 voxels with an intensity range from 0 to 210. All voxels are reformatted to cubes with a side length of 1 mm. The gold standard image was also reformatted to pixels with a side length of 1 mm and has therefore a resolution of 410^2 pixels. Its intensity depth is 8 bit.

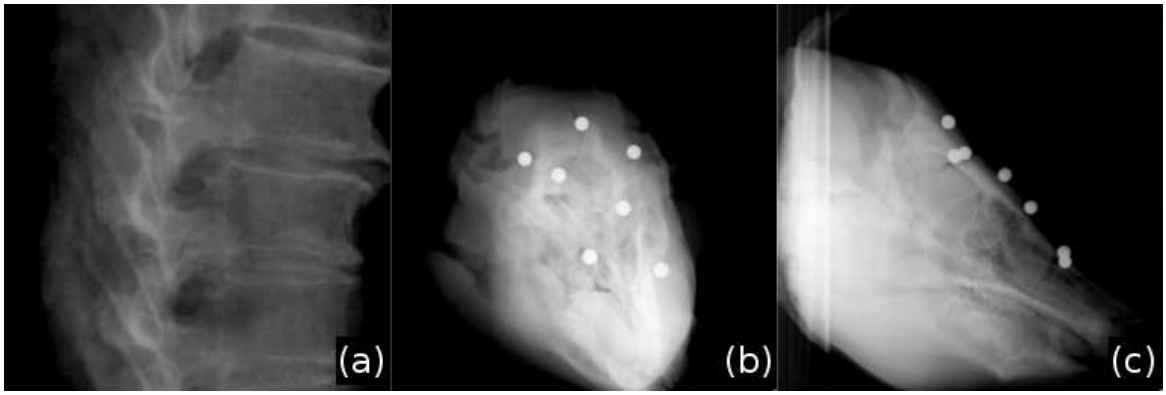

Fig. 42: Gold standard images of the used datasets. (a) shows the gold standard image of the spine, (b) the kilo-voltage gold standard of the Pawiro pig in anterior-posterior (AP) pose and (c) the kilo-voltage gold standard of the Pawiro pig in lateral pose.

4.5 Test system

All tests and benchmarks were performed on an *Intel(R) Core(TM)2 Duo T7700 CPU @ 2.40 GHz* with *4 GB Ram* running *Kubuntu 9.10 x64*, which uses the *2.6.31 Linux kernel* and the *ext4* file system. A *NVIDIA Quadro FX 570M* graphics card with *512 MB video memory* was used with the *NVIDIA 190.18 driver* for all rendering procedures. For more information about the used graphics card, see the CUDA Device Query information in the appendix.

The software suite itself was developed in the *Qt Creator 1.2.1* integrated development environment (IDE) using *g++ (Ubuntu 4.4.1-4ubuntu8) 4.4.1*, *Qt 4.5.2 (64 bit)*, *OpenGL 3.1.0 NVIDIA 190.18*, *NVIDIA Cg 2.3*, *NVIDIA Cuda Toolkit 2.3* and *GNU Scientific Library (GSL) 1.12*.

5 Results

This section presents all results of the different measurements described in Sect. 4.4. They are presented without any interpretation. The discussion and analysis of the results is presented in Sect. 6.

5.1 DRR quality

Comparing image quality in an objective way is not possible. Fig. 43 shows therefore some exemplary DRRs for visual analysis. In addition, Fig. 44 presents DRRs showing negative properties of the rendering methods and their optimization techniques.

Fig. 44 (a), (b) and (c) show that the refined bounding structures are sometimes visible in the final rendering. Fig. 44 (a) is a difference image between ray casting with and without cuberille refined bounding structure that was intensity scaled. In most situations, the cuberilles are not visible. Fig. 44 (c) is a ray casting without refined bounding structure. At the right side of the rendering, the corner of the bounding cube can be seen as a shadow. Fig. 44 (d) shows that splat rendering has a problem with objects being far away from the eye position. Many voxels are projected onto only a few pixels which saturate and information about the volume is lost. Fig. 44 (e) shows that the intensity of the renderings sometimes varies. Fig. 44 (f) shows that the high-energy artifacts of splat rendering are still not completely removed for some perspectives when not using noise textures.

To present some quantitative value for the "quality" of the images, the chart in Fig. 45 shows the results of mutual information with a gold standard image for the Pawiro pig dataset. Normally, mutual information lies in the interval between 0 and 1 and is at a maximum for best similarity. For better comparison with other merit functions, this is inversed so that best similarity is indicated by a minimum and the values are scaled between 0 and 1000.

In addition, the difference between wobbled splat rendering with and without noise texture could be quantified to a mutual information of about 203.0 (158.0 – 223.6) while the mutual information between ray casting using a cuberille refined bounding structure with a size of eight voxels or not is 207.1 (37.9 – 453.4).

5.2 Rendering time

Tab. 2 shows mean rendering time measurements for GPU based wobbled splat rendering and GPU based ray casting. In addition, also the rendering times for optional optimization techniques and some promising combinations of them are shown to evaluate their effect on the rendering performance. For each of these methods, the mean rendering time and its standard deviation (SD) are shown for both volume datasets and two different thresholds for each dataset. The

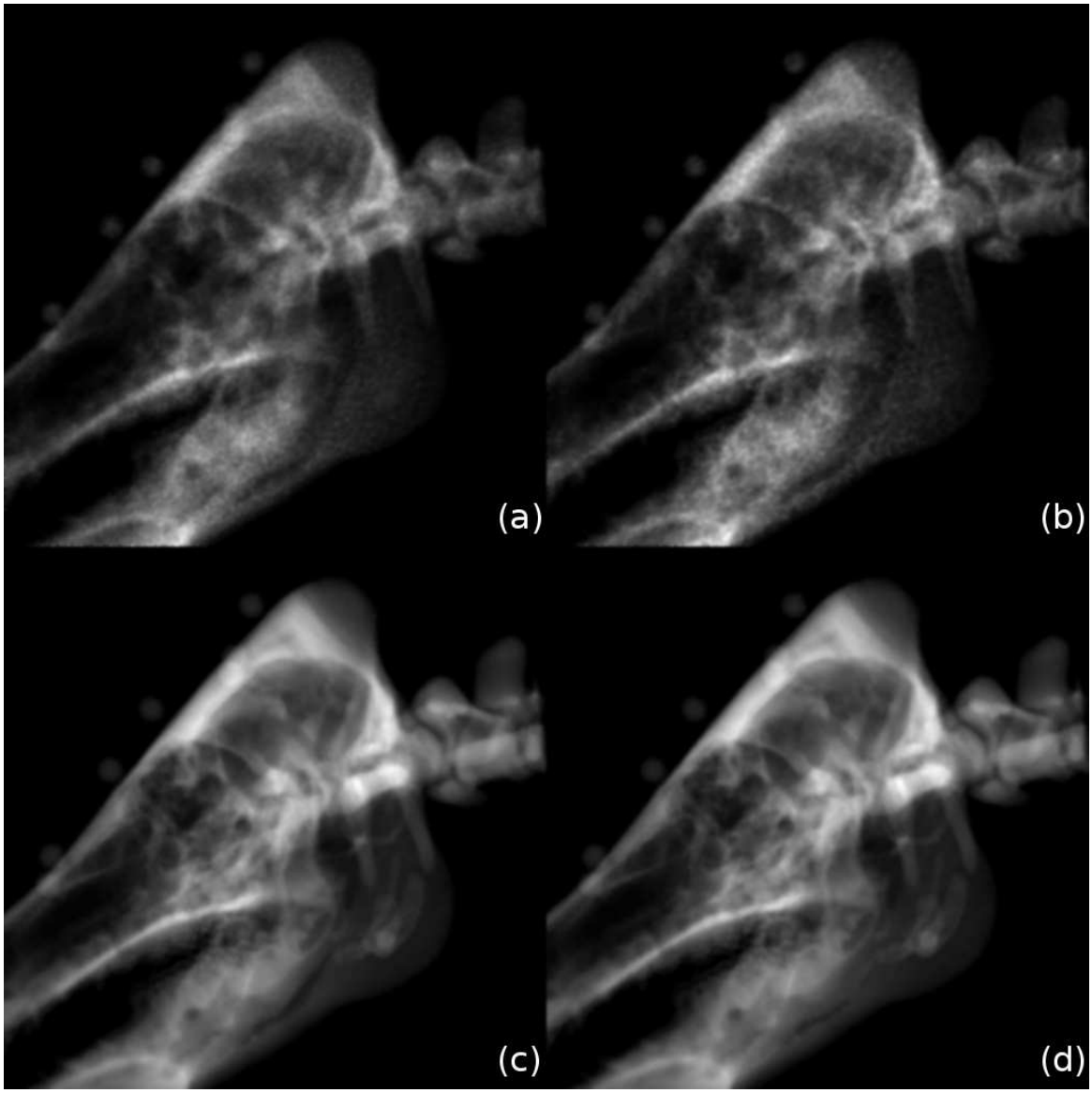

Fig. 43: The same perspective rendered with (a) splat rendering, (b) splat rendering using a noise texture, (c) ray casting and (d) using ray casting with a bounding structure refined by the cuberille method.

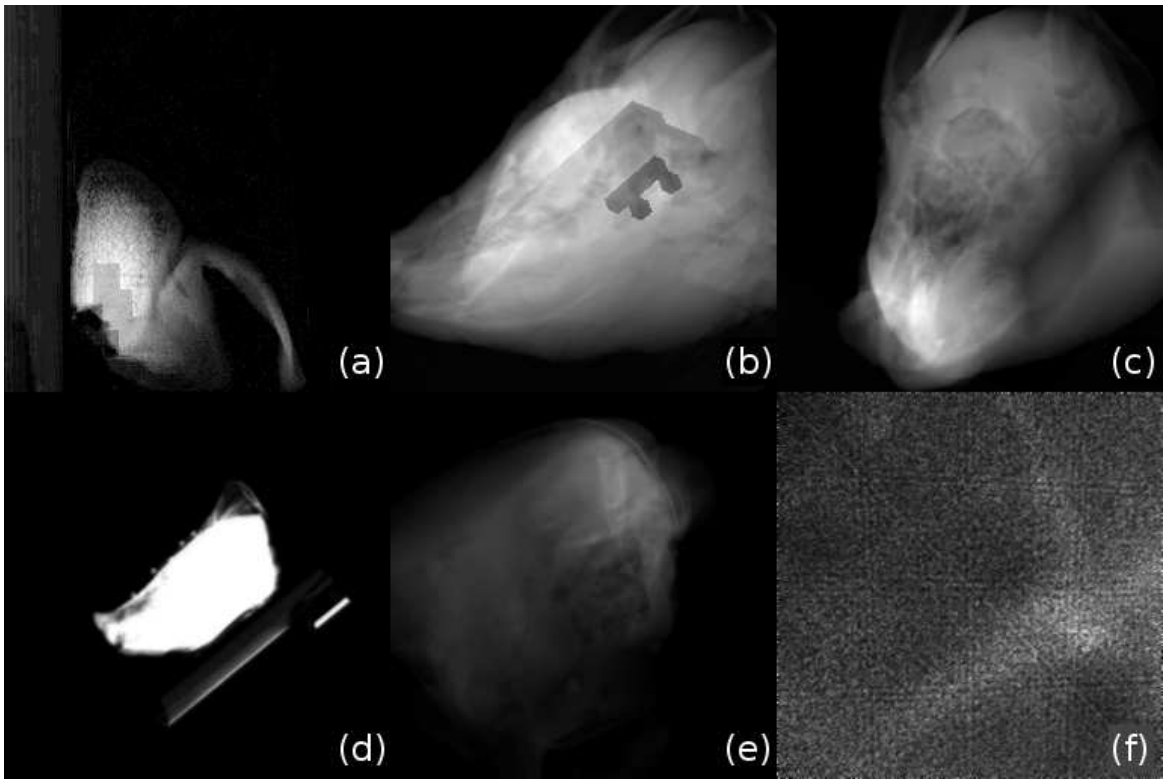

Fig. 44: Illustrations showing some problematic properties of the different rendering approaches and their optimization techniques.

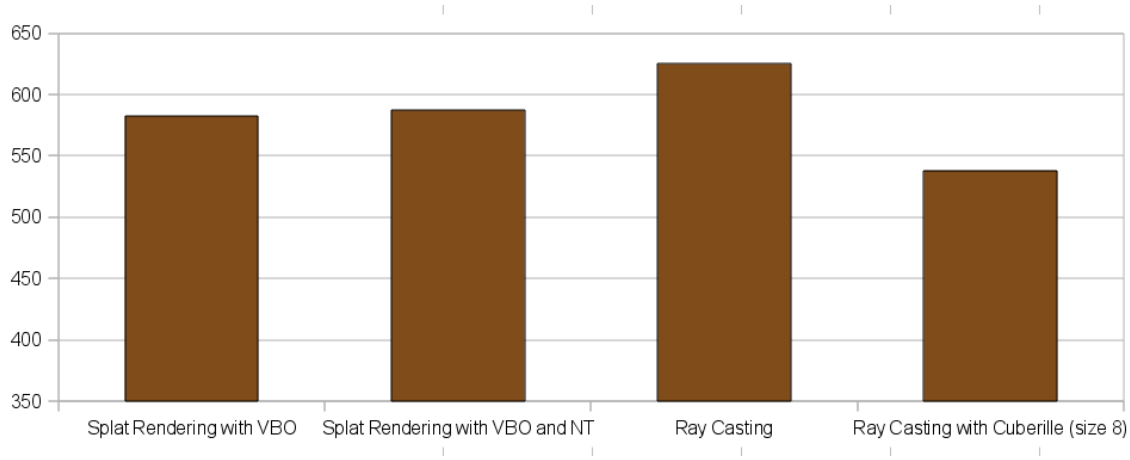

Fig. 45: Quantification of rendering quality using the Mutual Information of the DRRs acquired using different algorithms with gold standard x-ray images. The shown algorithms are wobbled splat rendering with vertex buffer objects (VBOs) and with and without noise texture (NT) and ray casting with and without cuberille refined bounding structure. The size of the cuberilles for the bounding structure refinement was eight voxels.

first line shows the rendering time of the wobbled splatting algorithm with the status of July 2009 as a base value.

To illustrate the influence of the cuberille size for bounding structure refinement on the rendering speed, different volumes with different thresholds are rendered using different cuberille sizes and the results are compared in Fig. 46.

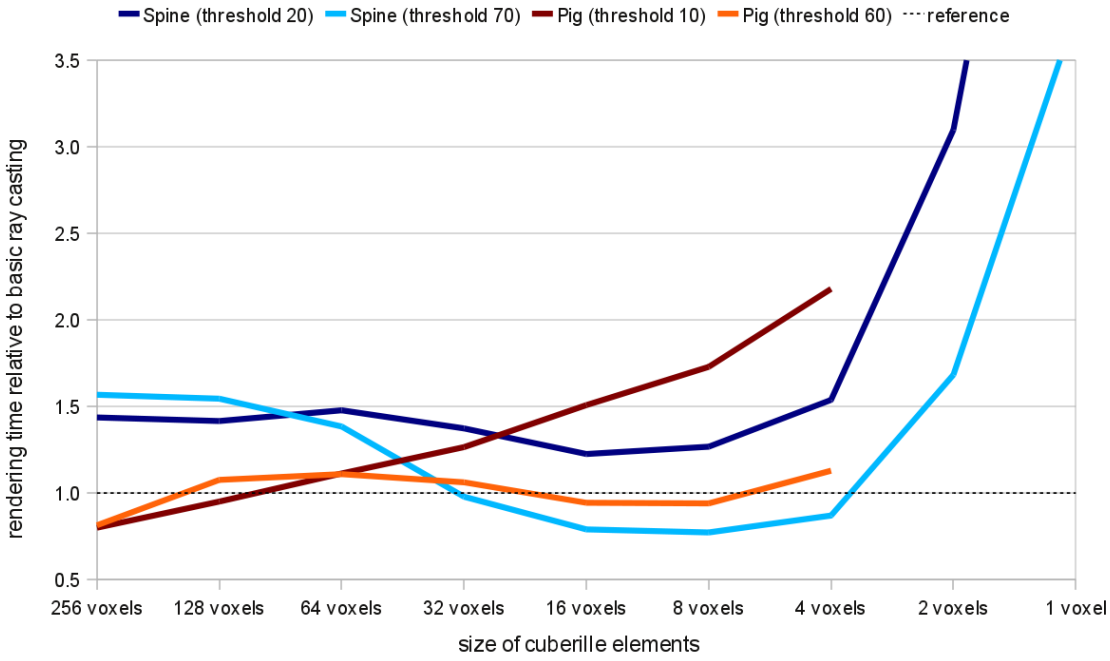

Fig. 46: Chart showing the rendering times for the different volumes using the GPU based ray casting with cuberille based bounding structure refinement with different cuberille sizes. Reference refers to a value of 1 which represents for all graphs the rendering time without cuberille refined bounding structure for the same dataset and threshold.

Also sparse sampling has a significant effect on the rendering performance. This is shown in the chart of Fig. 47 for the different volume datasets with different thresholds.

To quantify the impact of modifying the step size of ray casting, Fig. 48 shows the rendering times of a DRR in gold standard position with varying step sizes normalized with the rendering time at a step size of one voxel.

Fig. 49 shows the influence of the amount of content on the rendering performance. Amount of content means how much of the volume is visible on the image and how much is rendered off screen because it is out of the field of view. To demonstrate the effect of the zoom level, Fig. 50 shows rendering times for the same object but with differing z-coordinates for the wobbled splat renderer using vertex buffer objects and ray casting in its basic version and using a cuberille refined bounding structure (size eight voxels).

Tab. 2: Mean rendering times and their standard deviations (SD) for 27000 renderings using GPU based wobbled splat rendering and GPU based ray casting including several optimization techniques for them. The first row shows the results for the original version of July 2009 as a base value. All times were measured for both datasets and two different thresholds for each of them.

	Rendering Times and Standard Deviation in ms for			
Rendering Method (all GPU based)	Spine Dataset with Threshold 20	Spine Dataset with Threshold 70	Pig Dataset with Threshold 10	Pig Dataset with Threshold 60
Wobbled Splatting				
status July 2009	57.420 ± 2.010	16.017 ± 0.045	69.008 ± 0.207	46.076 ± 0.366
Wobbled Splatting				
basic	58.141 ± 1.447	15.044 ± 0.208	68.873 ± 1.559	46.103 ± 0.869
without wobbling	56.306 ± 1.910	15.446 ± 0.593	69.122 ± 1.495	45.905 ± 1.194
noise texture	57.000 ± 0.620	15.259 ± 0.445	68.027 ± 0.452	45.027 ± 0.296
vertex buffer objects	29.537 ± 2.674	8.374 ± 0.684	34.482 ± 2.625	22.973 ± 1.530
vertex buffer objects and noise texture	30.290 ± 4.175	9.077 ± 0.855	35.396 ± 4.081	23.457 ± 2.561
vertex buffer objects and w/o filtering	27.376 ± 2.654	6.265 ± 0.650	32.444 ± 2.640	20.823 ± 1.557
Ray Casting				
basic	14.136 ± 2.050	12.955 ± 1.866	22.182 ± 4.329	21.807 ± 4.353
line integral	14.567 ± 1.716	13.341 ± 1.568	24.319 ± 3.681	23.267 ± 3.507
Cuberille (256 voxels)	20.305 ± 3.235	20.309 ± 3.233	17.731 ± 4.360	17.739 ± 4.361
Cuberille (128 voxels)	20.016 ± 3.206	20.012 ± 3.205	21.090 ± 5.701	23.457 ± 5.500
Cuberille (64 voxels)	20.891 ± 3.443	17.933 ± 2.815	24.683 ± 6.184	24.187 ± 4.775
Cuberille (32 voxels)	19.413 ± 3.036	12.678 ± 2.146	28.062 ± 9.306	23.154 ± 5.660
Cuberille (16 voxels)	17.321 ± 2.772	10.231 ± 2.263	33.449 ± 12.93	20.575 ± 5.968
Cuberille (8 voxels)	17.919 ± 3.396	10.003 ± 2.982	38.349 ± 16.50	20.496 ± 6.447
Cuberille (4 voxels)	21.742 ± 5.291	11.270 ± 3.848	48.334 ± 20.82	24.617 ± 8.372
Cuberille (2 voxels)	43.798 ± 16.39	21.802 ± 9.231		
Cuberille (1 voxels)	95.978 ± 38.22	48.839 ± 22.66		
Marching Cubes	139.203 ± 25.39	81.132 ± 14.14		
Sparse Sampling (50%)	11.953 ± 1.787	11.064 ± 1.625	15.293 ± 2.924	15.334 ± 3.022
Sparse Sampling (45%)	11.633 ± 1.729	10.773 ± 1.575	15.040 ± 2.815	15.150 ± 2.853
Sparse Sampling (40%)	11.289 ± 1.661	10.448 ± 1.509	14.758 ± 2.648	14.790 ± 2.680
Sparse Sampling (35%)	10.878 ± 1.577	10.059 ± 1.438	14.422 ± 2.499	14.436 ± 2.480
Sparse Sampling (30%)	10.374 ± 1.488	9.610 ± 1.344	13.994 ± 2.299	13.898 ± 2.273
Sparse Sampling (25%)	9.759 ± 1.353	9.044 ± 1.233	13.370 ± 2.078	13.273 ± 2.025
Sparse Sampling (20%)	8.967 ± 1.190	8.279 ± 1.063	12.458 ± 1.805	12.307 ± 1.738
Sparse Sampling (15%)	7.909 ± 0.951	7.325 ± 0.845	11.190 ± 1.455	11.036 ± 1.391
Sparse Sampling (10%)	6.399 ± 0.613	5.823 ± 0.505	8.993 ± 0.921	8.784 ± 0.907
Sparse Sampling (5%)	4.784 ± 0.411	4.323 ± 0.467	6.766 ± 0.424	6.552 ± 0.497
Sparse Sampling (1%)	3.756 ± 0.430	3.512 ± 0.500	5.146 ± 0.353	5.003 ± 0.270
Cuberille (8 voxels) and Sparse Sampling (25%)	11.907 ± 1.966	7.271 ± 2.112	22.123 ± 8.842	12.810 ± 4.066

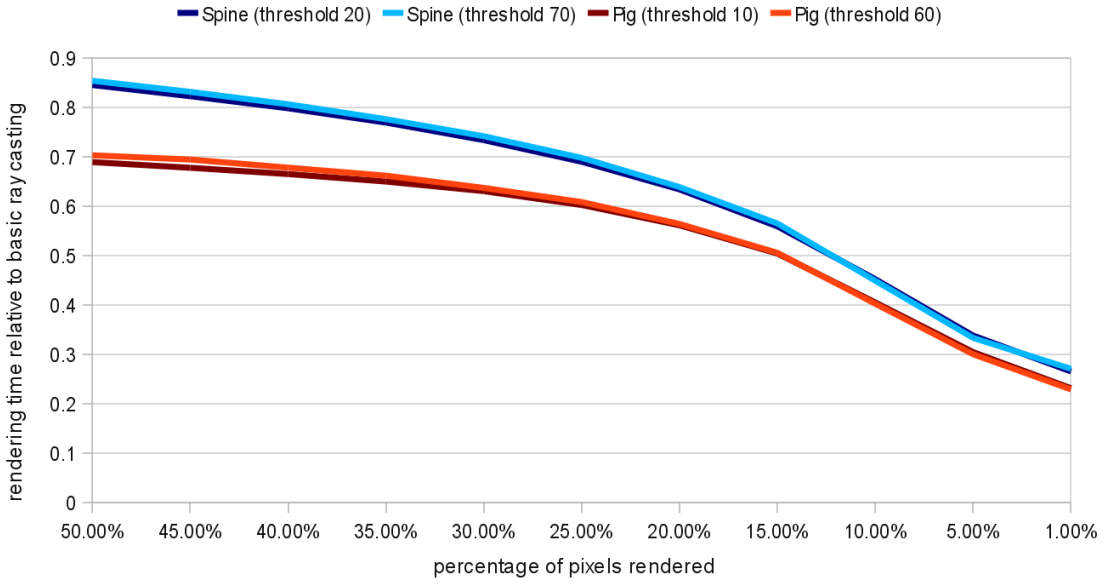

Fig. 47: Chart showing the rendering times for the different volumes using the GPU based ray casting with sparse sampling with different percentages of pixels actually rendered.

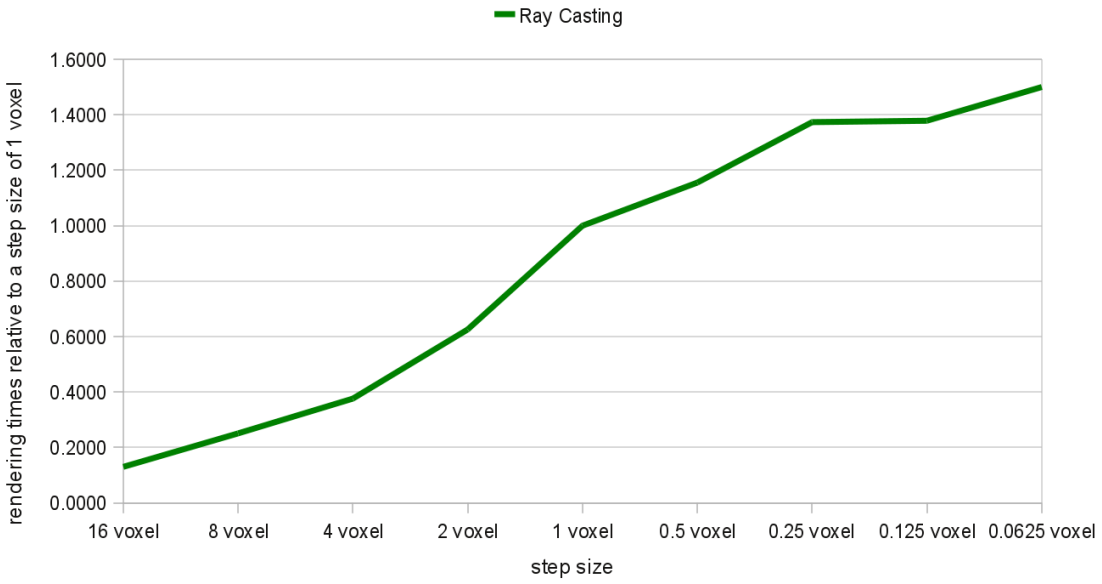

Fig. 48: Chart showing rendering performance of ray casting for different step sizes normalized with the rendering time at a step size of one voxel.

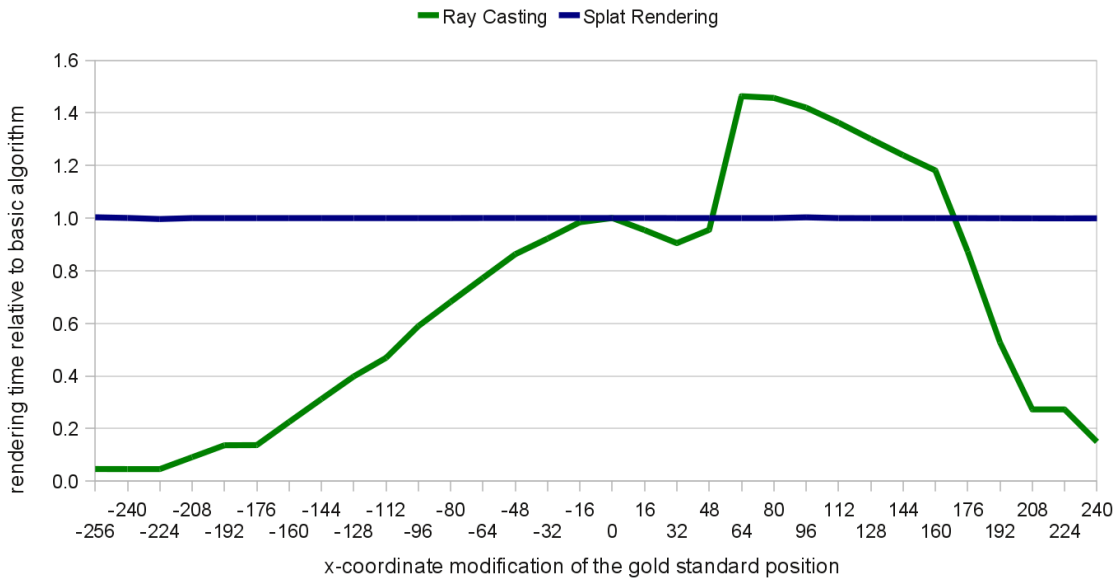

Fig. 49: Chart showing rendering performance for different amounts of image content. Image content means how much content is rendered onto the image and how much is rendered off screen. The volume in gold standard position is translated in x-direction from one -256 to +256. The values are normalized with the rendering times at the gold standard position.

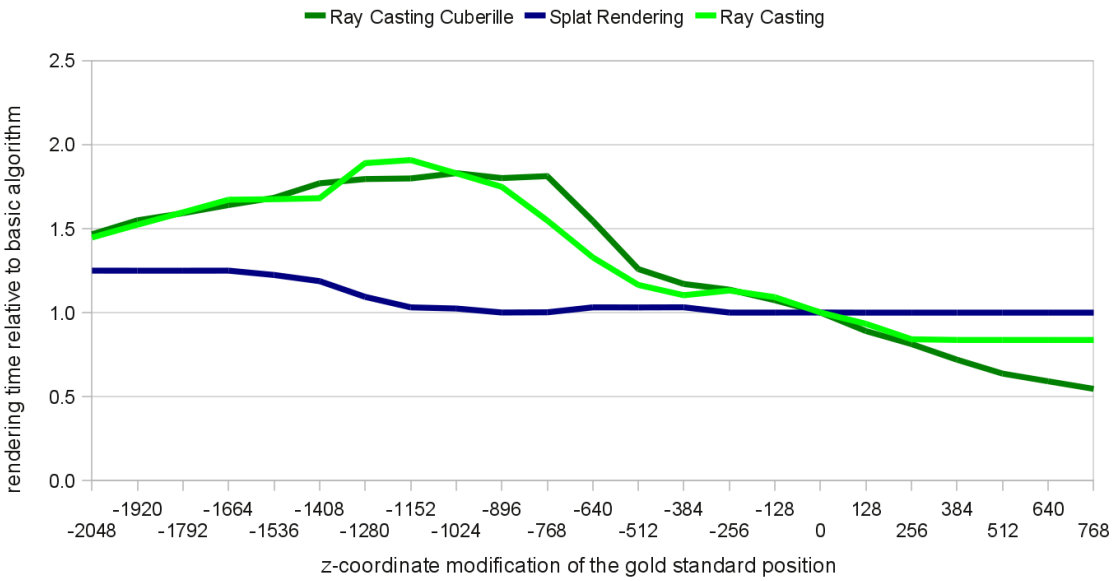

Fig. 50: Chart showing rendering performance for different positions along the z-axis of the same object in gold standard position. This is similar to the zoom level of the DRR. The values are normalized with the rendering times at the gold standard position.

Finally, the influence of the image size on the rendering performance is shown in Fig. 51. To normalize the relative amount of content in the images, the z-translation is varied so that the object's size follows the image size.

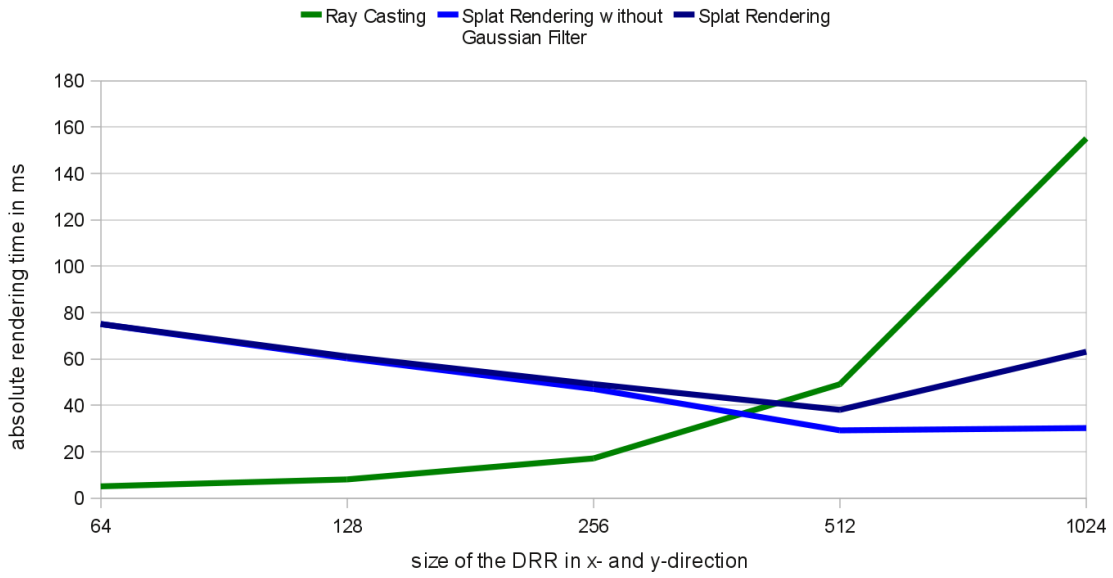

Fig. 51: Chart showing the influence of the image size on the rendering performance. To normalize the amount of image content, the object to render was moved along the z-axis so that the relative size was kept constant.

5.3 Influence of chosen registration merit function

To illustrate the influences of the rendering methods on the registration quality, the merit function is investigated around a known gold standard position. While five transformation parameters are fixed, one after the other is varied around the gold standard position and the function is evaluated and recorded.

The figures in this section all have an identical setup. They contain six sub-diagrams. The sub-diagrams of the first row shows the isolated modification of the three translation parameters (from left to right, x-, y- and z-translation). The sub-diagrams of the second row shows the isolated modification of the three rotation parameters (from left to right, x-, y- and z-rotation). The vertical blue line represents the position of the known gold standard and is at the same position for all graphs of a figure. The vertical red line for each graph shows the found minimum. On some figures, only one red line is visible. This is the result if in all shown graphs the same position is found as optimum.

Fig. 52 shows the merit function for wobbled splat rendering using vertex buffer objects and ray casting with cuberille refined bounding structure (size of eight voxels) for mutual information. As data, the Pawiro pig with a threshold of 40 was used.

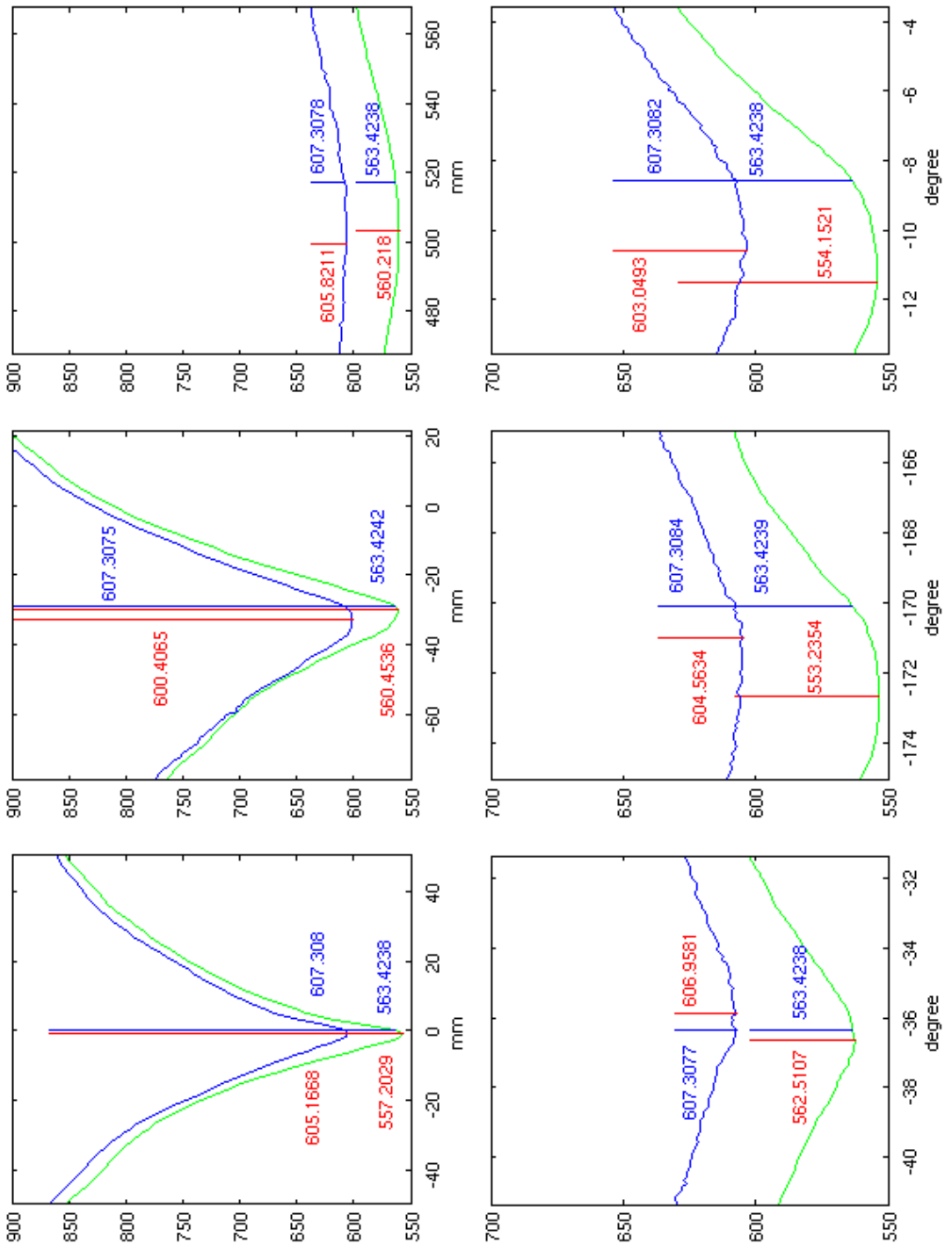

Fig. 52: Mutual information around the gold standard position independent for each parameter for the Pawiro pig with a threshold of 40. The blue graph shows wobbled splat rendering with vertex buffer objects while the green graph shows ray casting with cuberille refined bounding structure (size of eight voxels).

In contrast to Fig. 52, Fig. 53 shows cross correlation instead of mutual information. Fig. 54 shows again wobbled splat rendering using vertex buffer objects and ray casting with cuberille refined bounding structure (size of eight voxels) for the Pawiro pig with threshold 40 but with rank correlation as merit function. Fig. 55 finally shows wobbled splat rendering using vertex buffer objects and ray casting with cuberille refined bounding structure (size of eight voxels) for the Pawiro pig with threshold 40 with correlation ratio as merit function.

To investigate the influence of the wobbling on the merit function shape, Fig. 56 shows the one-dimensional analysis of mutual information for wobbled splat rendering with vertex buffer objects with and without applying wobbling for the Pawiro pig in gold standard position with a threshold of 40. To have additional information about the influence of the wobbling on the merit function shape, Fig. 57 shows the one-dimensional analysis of mutual information for wobbled splat rendering with vertex buffer objects with and without applying wobbling for the Pawiro pig in the position with all transformation parameters set to 0 with a threshold of 40.

To evaluate if the cuberille bounding structure refinement has an influence on the merit function, Fig. 58 shows mutual information for ray casting with and without cuberille refined bounding structure (size of eight voxels) and a threshold of 40 for the Pawiro pig dataset. Fig. 59 shows the same setup with a threshold of 60 for the Pawiro pig dataset

Fig. 60 shows the merit function evaluations for ray casting with cuberille refined bounding structure (size of eight voxels) and the same method using sparse sampling with 50% and 25% of the image content. The used merit function is mutual information and the used dataset is the Pawiro pig with a threshold of 40.

5.4 Overall registration speed and quality

Tab. 3 shows the results of the overall registration processes. These results are based on 150 registrations with different start positions. The starting positions are equally distributed in an interval of 0 to 15 mm mTRE with 10 starting position for each mm. The main data of the results are the mean registration time including its standard deviation (SD), the number of iterations and the mean mPD and mTRE errors and their standard deviations. The mean mPD error is the distance one point has between a projection with the found solution parameters and a projection with the known gold standard parameters. The mean mTRE error represents the same measure but between the same point in the two volumes.

Fig. 61 shows the distribution of the mPD errors from Tab. 3 for the four tested rendering methods. The x-axis shows the mPD error before and the y-axis the mPD error after registration. All dots below the blue line represent an improvement.

Fig. 62 shows the distribution of the mTRE errors from Tab. 3 for the four tested rendering methods. The x-axis shows the mTRE error before and the y-axis the mTRE error after registration.

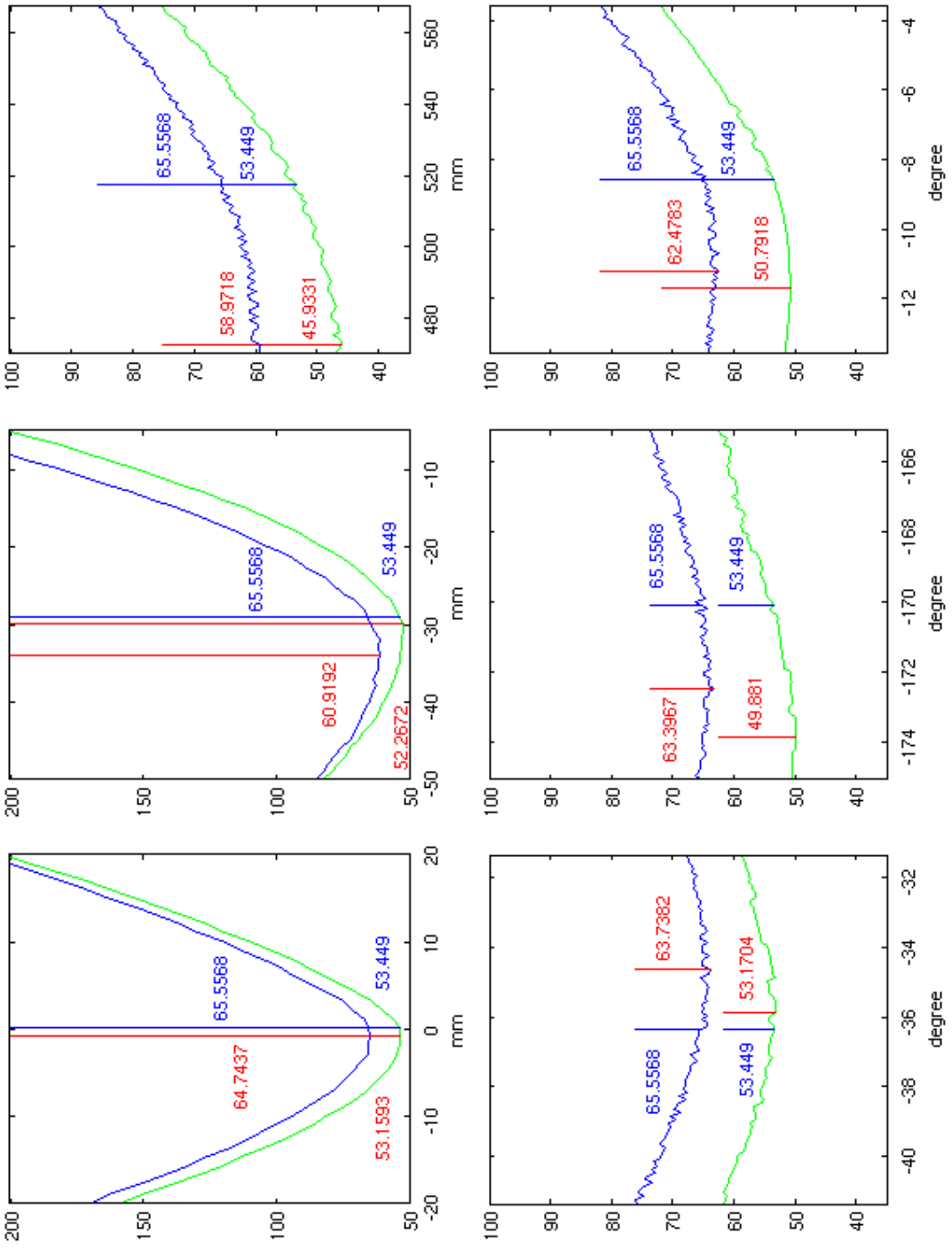

Fig. 53: Cross correlation around the gold standard position independent for each parameter for the Pawiro pig with a threshold of 40. The blue graph shows wobbled splat rendering with vertex buffer objects while the green graph shows ray casting with cuberille refined bounding structure (size of eight voxels).

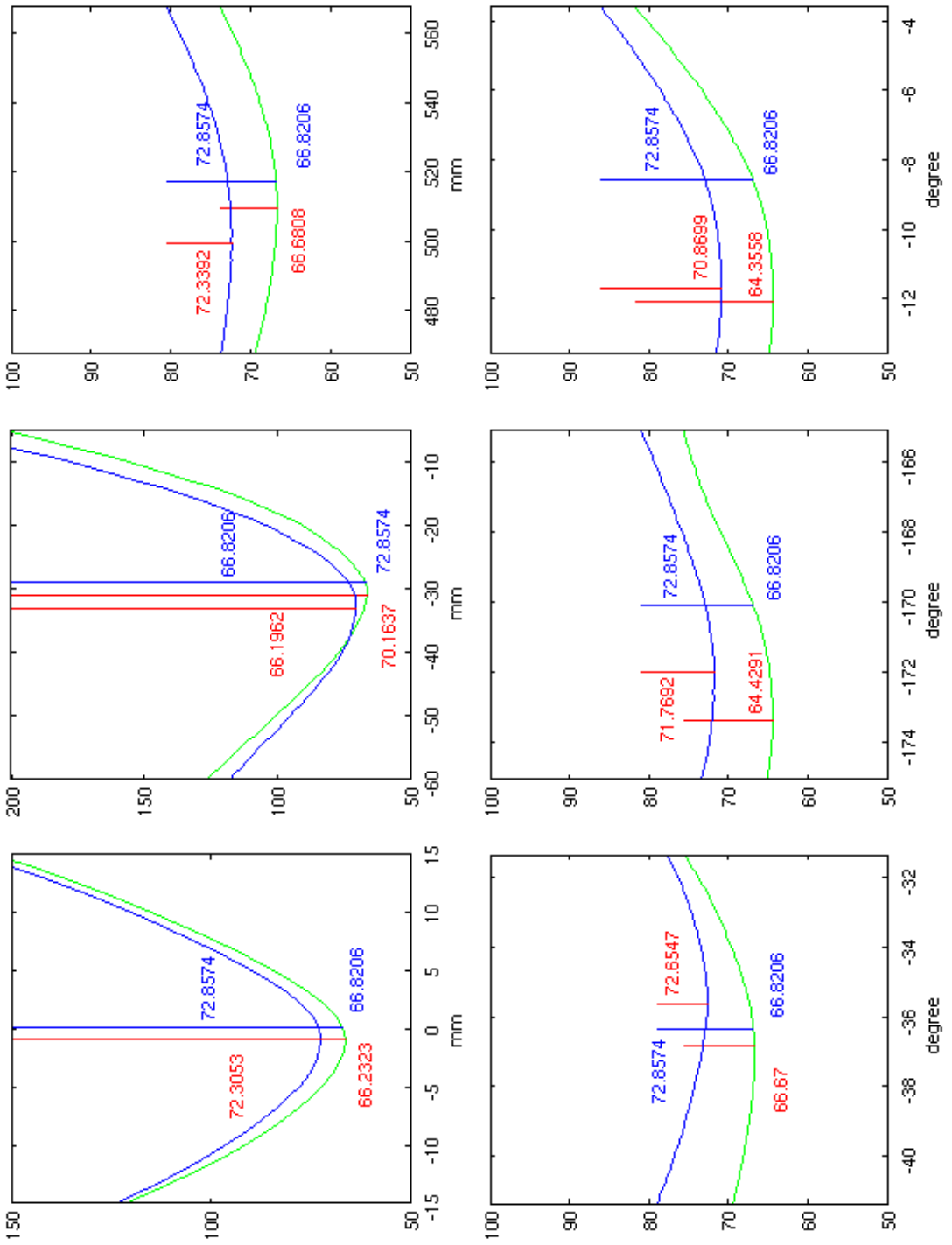

Fig. 54: Rank correlation around the gold standard position independent for each parameter for the Pawiro pig with a threshold of 40. The blue graph shows wobbled splat rendering with vertex buffer objects while the green graph shows ray casting with cuberille refined bounding structure (size of eight voxels).

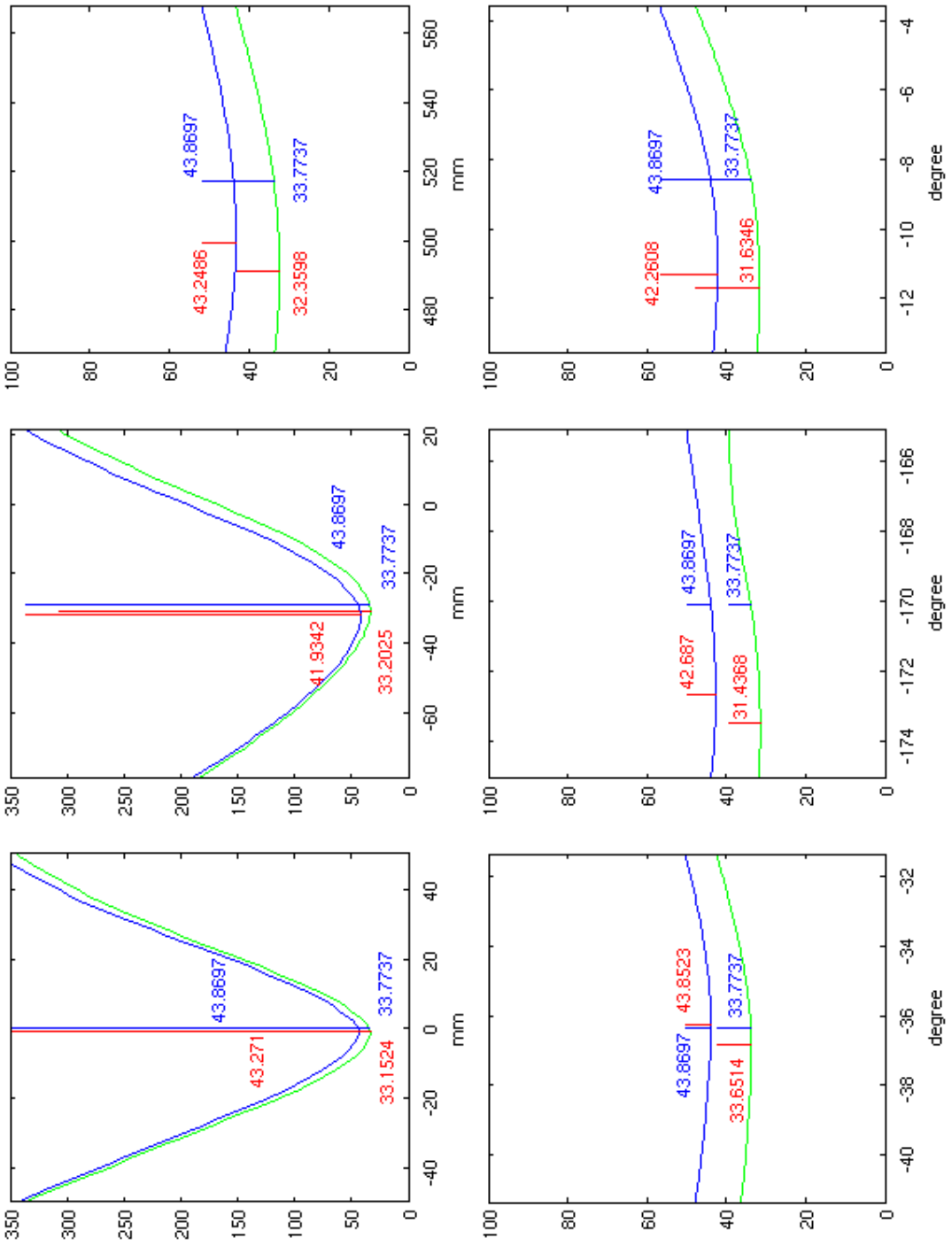

Fig. 55: Correlation ratio around the gold standard position independent for each parameter for the Pawiro pig with a threshold of 40. The blue graph shows wobbled splat rendering with vertex buffer objects while the green graph shows ray casting with cuberille refined bounding structure (size of eight voxels).

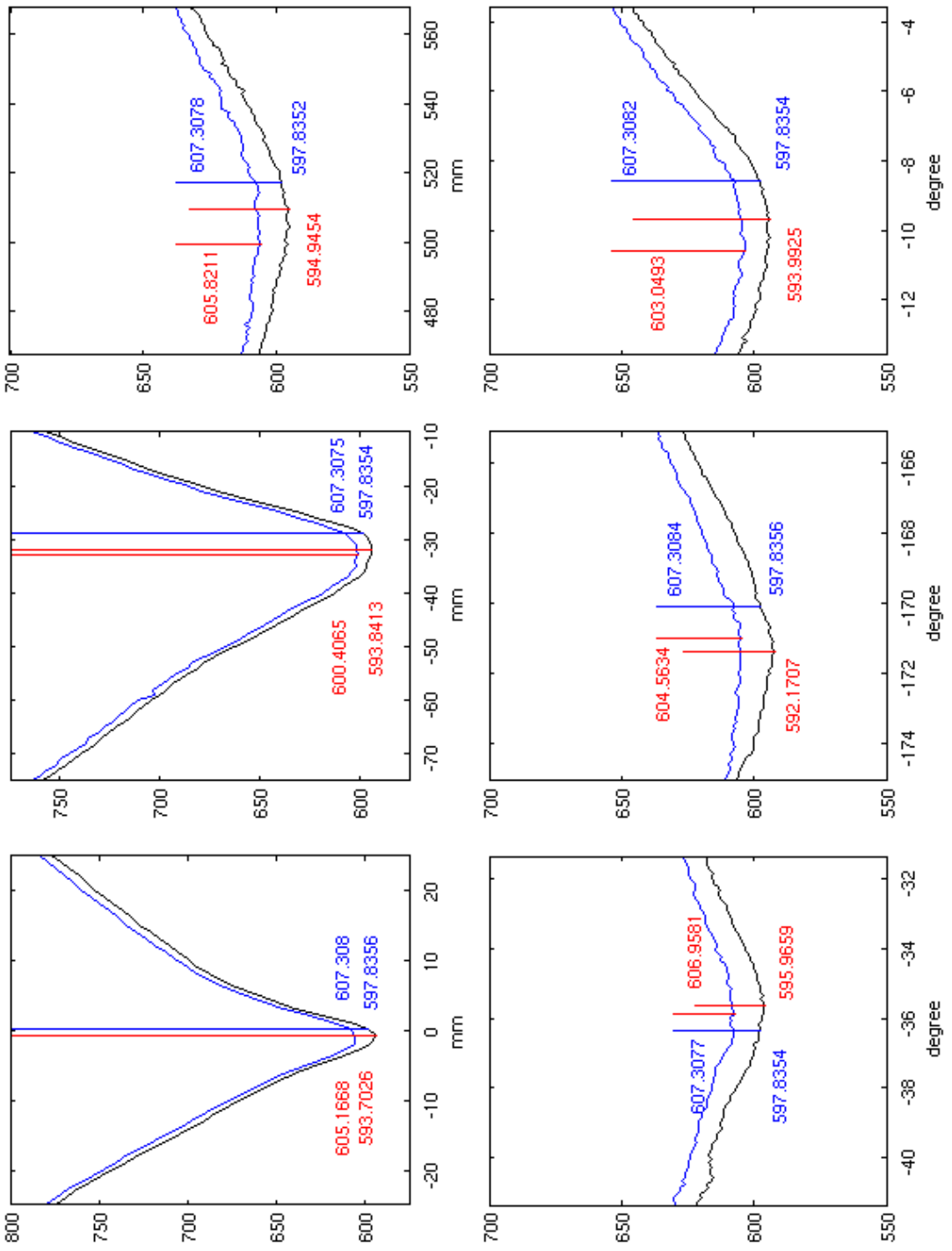

Fig. 56: Mutual information around the gold standard position independent for each parameter for the Pawiro pig with a threshold of 40. The blue graph shows wobbled splat rendering with vertex buffer objects while the black graph shows the same rendering method but without wobbling.

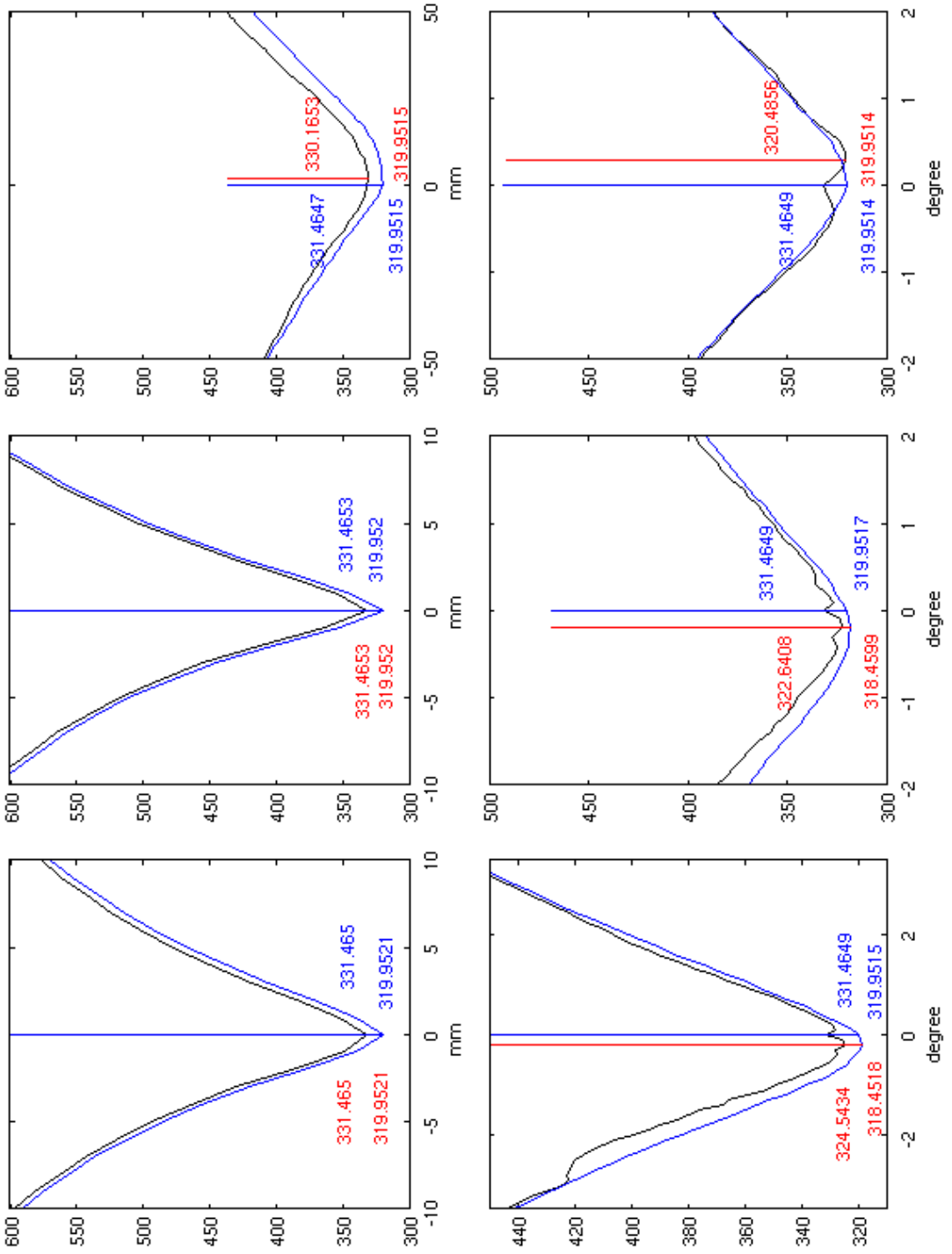

Fig. 57: Mutual information around the position $(0,\ 0,\ 0,\ 0,\ 0,\ 0)^T$ independent for each parameter for the Pawiro pig with a threshold of 40. The blue graph shows wobbled splat rendering with vertex buffer objects while the black graph shows the same rendering method but without wobbling.

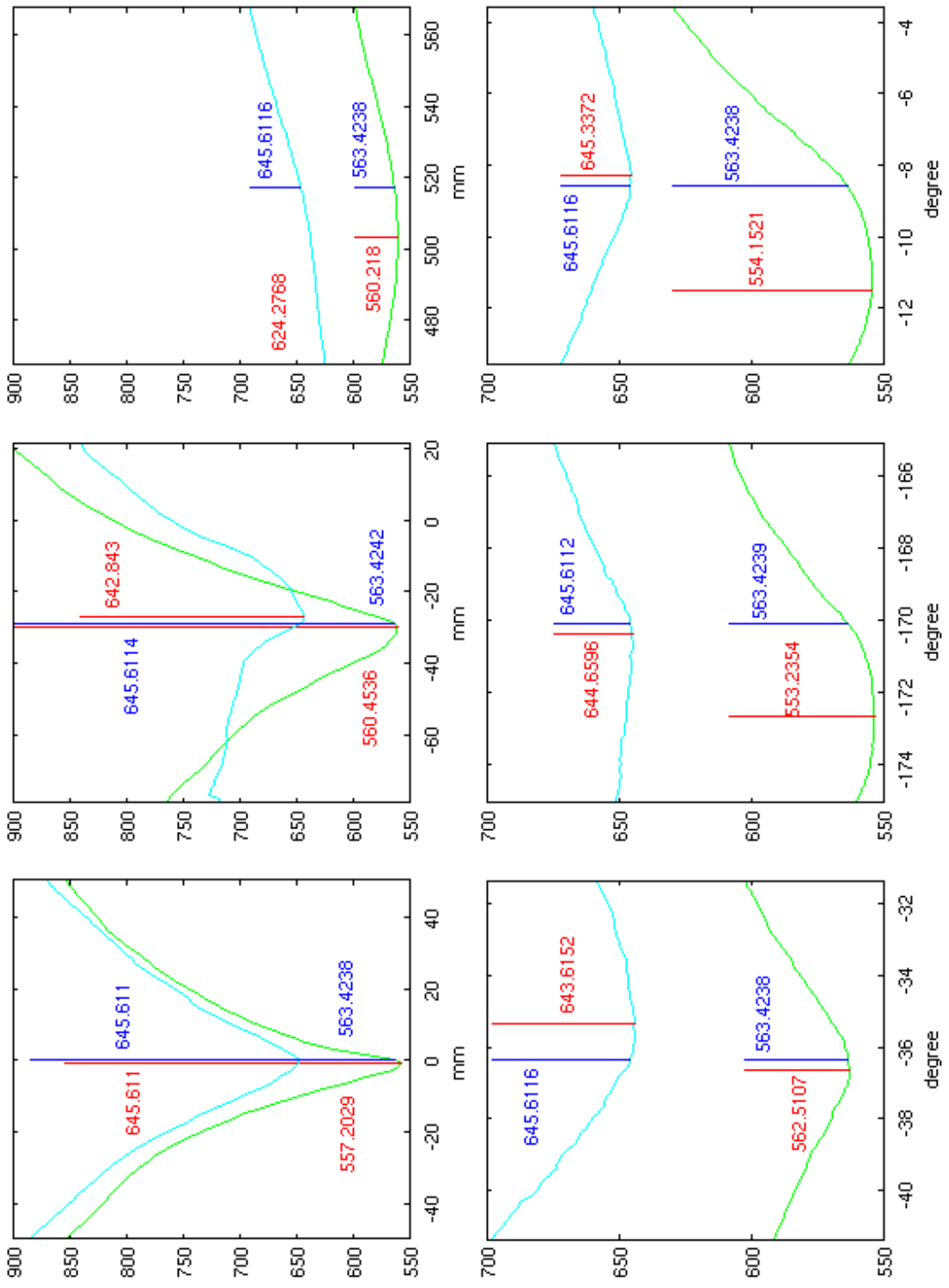

Fig. 58: Mutual information around the gold standard position independent for each parameter for the Pawiro pig with a threshold of 40. The green graph shows ray casting with a cuberille refined bounding structure (size of eight voxels) while the cyan graph shows the merit function for the same rendering algorithm without a refined bounding structure.

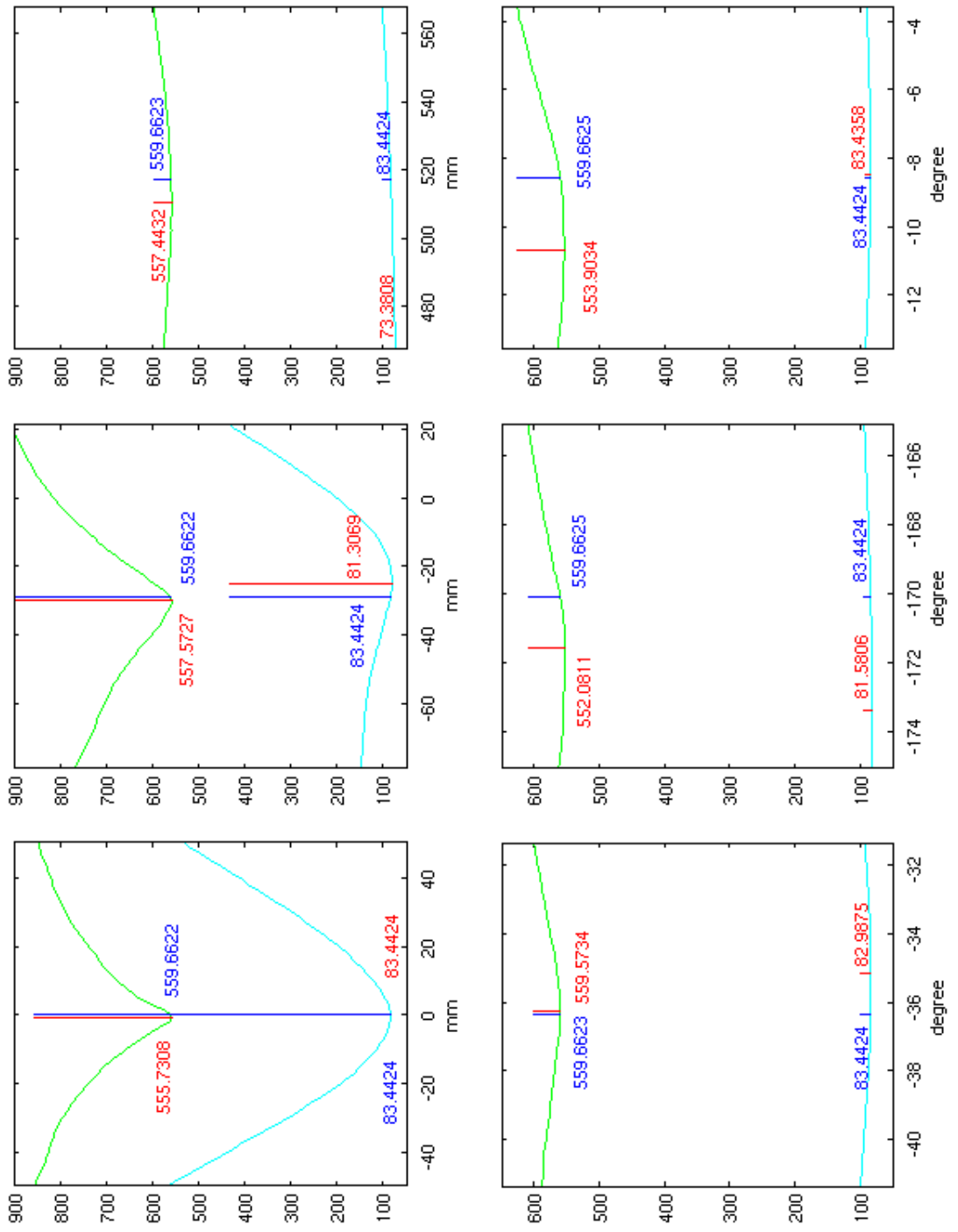

Fig. 59: Mutual information around the gold standard position independent for each parameter for the Pawiro pig with a threshold of 60. The green graph shows ray casting with a cuberille refined bounding structure (size of eight voxels) while the cyan graph shows the merit function for the same rendering algorithm without a refined bounding structure.

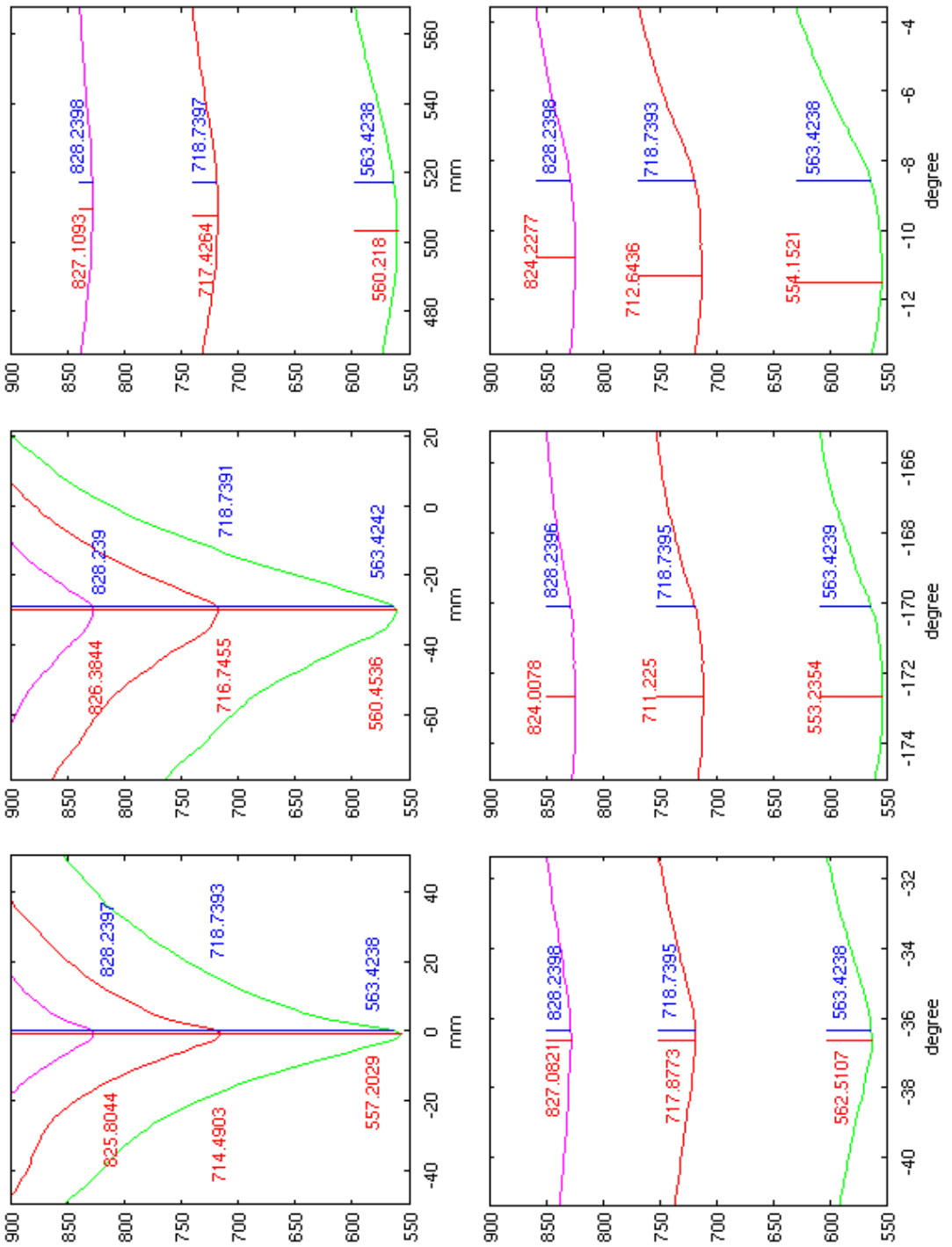

Fig. 60: Mutual information around the gold standard position independent for each parameter for the Pawiro pig with a threshold of 40. The green graph shows ray casting with cuberille refined bounding structure (size of eight voxels). The red graph shows the same method with sparse sampling of 50% and the magenta graph shows the same method with sparse sampling of 25% of the image content.

Tab. 3: Measured times including standard deviations for the overall registration process. In addition, the number of required iterations until the optima were found as well as the mean mPD and mTRE errors are shown as a measure of overall registration quality.

Rendering Method (all GPU based)	Reg. Time and SD in s	Number of Iterations	mPD Error ± SD in mm	mTRE Error ± SD in mm
Wobbled Splatting with Vertex Buffer Objects	11.722 ± 3.202	102.03 ± 22.4	10.85 ± 1.88	9.71 ± 2.00
Ray Casting	8.195 ± 1.955	134.05 ± 34.1	10.19 ± 2.81	9.66 ± 2.86
Ray Casting Cuberille (8 voxel)	9.922 ± 2.934	112.27 ± 28.5	6.86 ± 1.36	6.53 ± 1.92
Ray Casting Cuberille (8 voxel) and Sparse Sampling (25%)	6.230 ± 1.250	99.49 ± 25.8	5.72 ± 1.60	5.69 ± 2.03

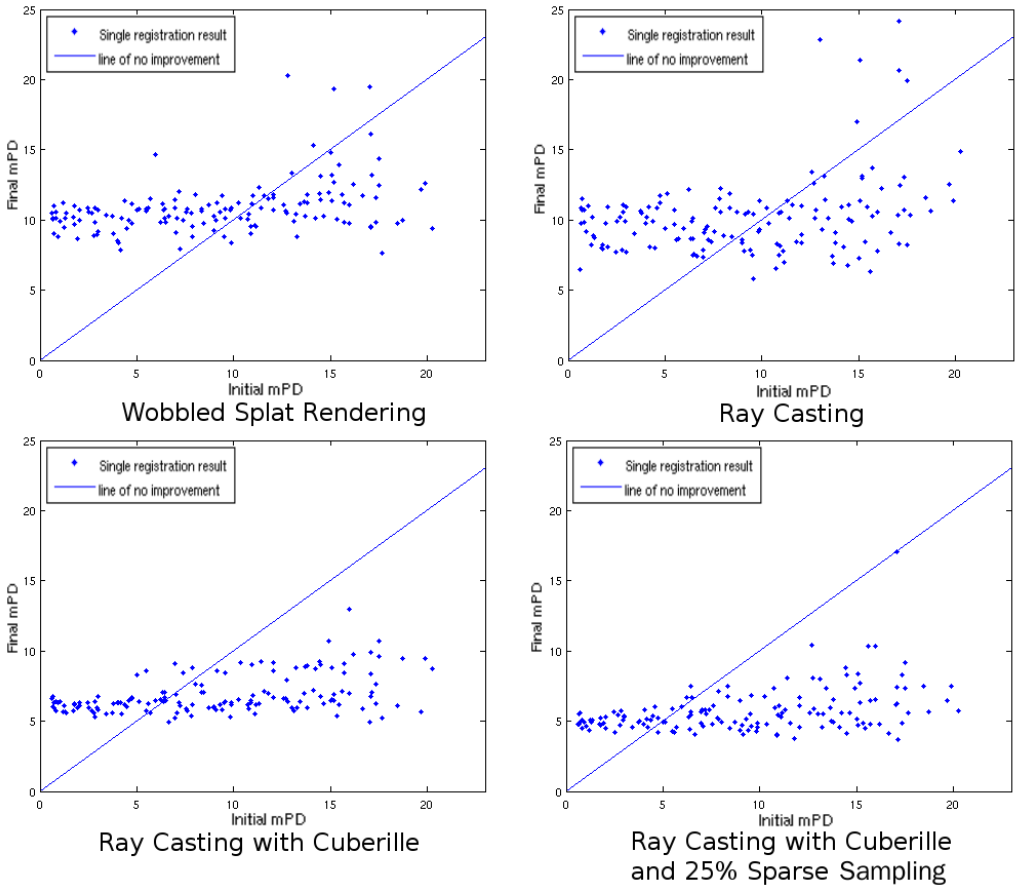

Fig. 61: Distribution of mPD errors from Tab. 3. The x-axis shows the mPD errors before and the y-axis the mPD errors after the registration. The blue line represents the line of improvement, which means that all dots below correspond to an improved error during registration.

All dots below the blue line represent an improvement.

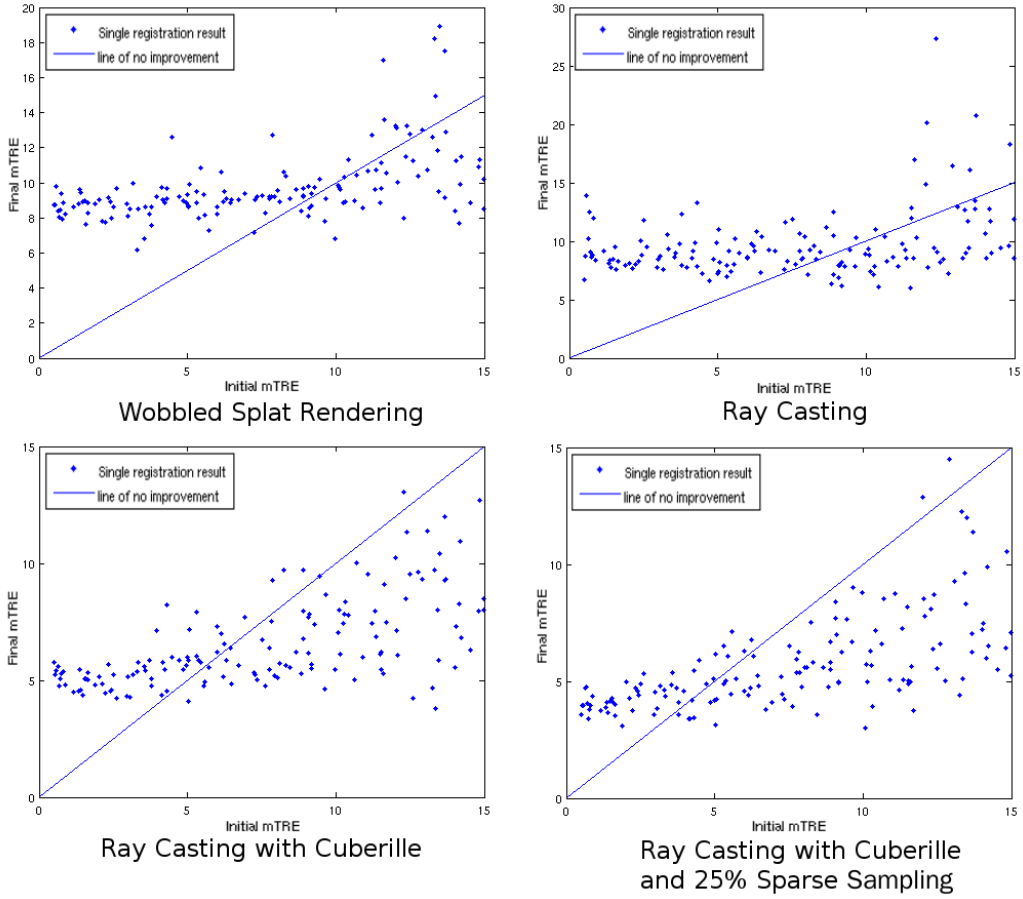

Fig. 62: Distribution of mTRE errors from Tab. 3. The x-axis shows the mTRE errors before and the y-axis the mTRE errors after the registration. The blue line represents the line of improvement, which means that all dots below correspond to an improved error during registration.

6 Discussion

This section gives an analysis and a discussion of the results presented in Sect. 5. This discussion is ordered by the developed rendering methods and extracts specifics, advantages and problems of them and their optimization techniques. Furthermore, some general aspects and additional technological issues are discussed. Also the general influence of the rendering method on the overall registration process is analyzed at the end of this section.

6.1 Wobbled Splat Rendering

Wobbled splat rendering was the originally implemented rendering method of the software suite presented in Sect. 3. Several extensions and optimizations were implemented and presented in Sect. 4 to further improve this DRR rendering algorithm.

The main advantage of wobbled splat rendering is its computational efficiency. As discussed by Birkfellner et al. for the CPU [20], this method is one of the fastest methods for DRR computations on this platform. Spoerk et al. [1] showed further improvements in performance by using the GPU as computation platform. The results of these works have been further improved in the context of this thesis. Tab. 2 clearly shows that retransferring the whole data to the GPU for every rendering was a huge bottleneck of the implementation by Spoerk et al. [1]. By using vertex buffer objects, the rendering time was reduced by 44–50 % in comparison to the same algorithm but without the use of vertex buffer objects. This speedup is not surprising but its amount is higher than expected. It shows the importance of efficient memory management when designing GPGPU applications.

Tab. 2 further shows that the Gaussian blur filtering in the post processing phase of the wobbled splat rendering algorithm is a new bottleneck. For example for the spine dataset with a threshold of 70, the filter is responsible for roughly one third of the overall rendering time. It is important to find better implementations for this operation. On the one hand, NVIDIA CUDA could be a technology to implement a filter with better performance. Many studies show that CUDA is perfectly suitable for filter implementations and allows a significant speedup [120, 121]. On the other hand, using a Median filter instead of the currently implemented Gaussian blur filter could probably lead to better rendering results. This type of filter does not modify the intensities but only exchanges pixel in order to remove noise. This would preserve the edges of the DRR while smoothening the rest. The drawback of this filter is that it requires sorting which is hard to be implemented on the GPU.

A drawback of the original wobbled splat rendering algorithm were small high-energy artifacts. In spite of wobbling, small artifacts were still visible, especially at perspectives with angles that are dividable by 45 degree. By using a third dimension for focus wobbling or changing the wobbling procedure from sine calculations to the fetching of a precalculated noise texture, these artifacts

could further be weakened and removed.

As Tab. 2 shows, changing the wobbling method from sine calculations to the fetching of a noise texture or to no wobbling at all, does not affect the rendering times significantly. The data even shows that it is possible that the rendering is slower without any wobbling method than with sine calculations. The reason for the good performance of the sine calculations is probably that the sine functions of NVIDIAs Cg are implemented in hardware. This behavior was also the reason why it was not possible to implement short Taylor series evaluations to speedup the wobbling as conceived in Sect. 4.1.2. If no wobbling is not significantly faster than when using sine functions, a hand made Taylor series evaluation, even if it only calculates the first step of the series, can not be faster.

When comparing the performance of wobbled splat rendering for different target image sizes as shown in Fig. 51, wobbled splat rendering shows a decrease of absolute rendering time with rising image size although the data that has to be processed is constant. This behavior is against the expectation that wobbled splat rendering is inert to changes of the image size. An explanation is that exhaustive alpha blending slows down the rendering. The amount of alpha blending increases if many voxels are projected onto a small area, which is equal to using a small image size.

For sizes of 512^2 pixels and larger, the influence of the post processing low pass filter can be observed in comparison to the graph showing the rendering times without filtering. Larger image sizes imply longer filtering which leads in total to longer absolute rendering times than for smaller image sizes.

An advantage of wobbled splat rendering is that it is not sensitive to changes in the amount of image content. As shown in Fig. 50, the rendering times for varying z-coordinates are constant. Only for very small z-coordinates, there is a higher rendering time that is probably again the result of more alpha blending. For changing x-coordinates as shown in Fig. 49, wobbled splat rendering shows no changes in the rendering performance at all, although for small or large x-coordinates, not the whole content is rendered onto the image plane. The explanation for this is that the algorithm does not know in advance which voxels are projected onto the image plane and which not and therefore is not able to skip voxels whose projected pixels are not visible.

Wobbled splat rendering still suffers from some problems. The biggest problem is the poor quality of the generated DRRs as can be observed in Fig. 43 (a) and (b). These images suffer from significant noise and blur effects in comparison to DRRs generated using ray casting (see Fig. 43 (c) and (d)). Especially when using noise textures, this noise is further increased because otherwise, high-energy artifacts would still be visible.

Fig. 44 (d) shows another quality problem of this rendering method. When the object is far away from the eye position, many voxels are projected onto a small area and the intensity at this

area saturates. The same effect can be observed when the image size is very small. As a result of this, information is lost and the DRR gets useless for 2D/3D-image registration.

Fig. 44 (f) shows that when the object is very close to the eye position, problems of the DRR quality occur. Very close renderings are extremely noisy and if rendered without noise textures and if the rotation parameters are dividable by 45 degrees, small high-energy artifacts are still visible. Similar to renderings of objects far away from the eye position, DRRs with this perspective are useless because the renderings look more like point clouds than a DRR of the volume.

To sum up, although wobbled splat rendering is not sensitive to rendering parameters in terms of rendering performance, these parameters have a tremendous influence on the quality of the generated DRRs.

Because of some yet unknown reason, the presented wobbled splat rendering implementation is very sensitive to the current system state. Changes in the amount of already used main memory and/or video memory and in the number of simultaneously running processes can increase the rendering time by about 25 %. It is yet not known how to make the algorithm more robust or which of the possible sources is the main reason for this performance fluctuations.

6.2 (Sparsely Sampled) Ray Casting

As an alternative to wobbled splat rendering, a GPU based ray casting algorithm was developed in Sect. 4.2. The rendering time measurements from Tab. 2 show that ray casting is competitive to wobbled splat rendering, even without additional optimization techniques. Only for images containing very few voxels (e.g. the spine dataset with a threshold of 70), wobbled splat rendering is significantly faster than ray casting without optimization techniques (approximately by one third). The use of linear summation instead of line integral calculation for composing the final ray intensity allows a small, additional speedup without reducing the image quality. The two main optimization techniques for this ray casting implementation are the refined bounding structure as already described in Sect. 4.2.4 and sparse sampling as described in Sect. 4.2.3.

The refinement of the bounding structure can lead to tremendous performance gains but can also have the inverse effect. It depends on the dataset and the kernel size that was used for the bounding structure refinement. Fig. 46 shows a comparison of relative rendering times for different datasets with different thresholds and different cuberille sizes. As a reference value to normalize these results, the rendering time without refined bounding structure was used. For most graphs, it can be observed that the rendering times have an optimum when using a cuberille size of 16 or eight voxels.

For smaller and larger cuberille sizes, the rendering takes more time than without any refined bounding structure. A possible reason for this when using very small cuberille sizes is that, because of the high amount of geometric information, the vertex shader gets slower than the

fragment shader which leads to a slow down of the whole rendering process. When using marching cubes instead of cuberille to refine the bounding structure, this effect gets even more important. Using the same size of refinement elements (marching cubes or cuberille size), marching cubes produces far more primitives that have to be processed in the vertex shader. This can be seen when comparing the rendering times from Tab. 2 for ray casting with cuberille refined bounding structure with a size of one voxel to the rendering times of ray casting with marching cubes refined bounding structure. Although both refinement methods use the same element size, renderings with marching cubes refined bounding structure require roughly 145–166 % of the time of renderings with cuberille refined bounding structure. While cuberille with a size of one voxel produces — for example with the spine dataset and a threshold of 70 — 541528 quads, marching cubes leads to a bounding structure consisting of 3374868 triangles.

The reason why a cuberille refined bounding structure with large cuberille sizes is slower than the basic algorithm is more difficult to explain. One explanation is that with such large sizes, no parts of the volume can be skipped while when using the basic algorithm, only the cube circumscribing the valid volume is rendered.

There are still three remaining points in Fig. 46 that cannot be clearly explained:

- Why have the rendering times of the Pawiro pig dataset their optimum at a cuberille size of 256 voxels?

  The only explanation for this behavior is the effect that can also be observed in Fig. 50 for ray casting performance with varying z-coordinate or zoom level. This effect and its implications on the cuberille performance are explained later.

- Why are the rendering times of the spine dataset with a threshold of 20 always above its reference value?

  As shown in Fig. 46, the rendering times of the spine dataset for low thresholds show the same shape as the high threshold graphs, with its optimum also at a size of 16 to eight voxels, but they are always above the reference value. Visual analysis has shown that the refined bounding structure with this threshold does not skip large portions of the data, but its surface gets more and more complex. Therefore, because no space is skipped, the performance is always worse than without any refinement. The shape is the same because this effect does not affect the balance between vertex and fragment shader.

- Why are the rendering times of the Pawiro pig dataset with a threshold of 10 strongly monotone rising and do not have the shape of all other graphs?

  This effect has probably the same explanation that was given for the spine with threshold of 20. The bounding structure gets more and more complex without much space being skipped. For this dataset, the effect is more pronounced because also the shape of the

graph is changed and rendering performance get worse with decreasing cuberille sizes.

The rendering times for different amounts of sparse sampling fulfill the expectations as illustrated in Fig. 47. Less image content has a linear correlation to shorter rendering times for all datasets and all thresholds. This reduction is independent of the data and only depends on the amount of content itself. First evaluations have shown that registration with less than 25 % of the image content is not suitable. With this amount of image content as a lower bound, the rendering time can be reduced using sparse sampling by approximately 30 % for the spine dataset and 40 % for the Pawiro pig dataset.

The major advantage of ray casting in comparison to wobbled splat rendering is the superior quality of the generated DRRs as illustrated in Fig. 43 (c) and (d). These DRRs show much less noise, sharper contours and more details. In addition, they are also useful when rendered with a position far away or very close to the eye position. Although these results are already very good, there are additional optimizations techniques for further improving the rendering quality that were described by Scharsach [8] and mentioned in Sect. 2.3.1 but not yet implemented. These include for example dizzling to be able to hit thin structures or hitpoint refinement to place the starting position of the ray sampling exactly on the begin of the valid data.

GPU based ray casting is a very flexible DRR rendering method. With the available optimization techniques and parameters, it is possible to achieve excellent image quality in very good rendering times. On a medium-class system like the test system described in Sect. 4.5, it is possible to render conventional datasets with rendering times down to 20 ms. The only problem is that the optimal parameters strongly depend on the used dataset and its threshold as well as on the pose in which the volume shall be rendered. Comparing the basic versions of wobbled splat rendering and ray casting from Tab. 2, it is clearly visible that the measurements of ray casting have a higher standard deviation. This is interesting, because all measurements were done with the volume centered on the DRR and without translations. It indicates that for different viewing angles, different rendering times were achieved. When also considering the results from Fig. 49 and Fig. 50, it is clear that the rendering time of ray casting varies when the volume changes its position. If the volume is not totally visible on the image, all data outside this section is skipped automatically and only the rest of the data is rendered. In such cases, the rendering time is reduced tremendously.

The zoom level, represented by the z-position of the object, shows some strange effect. The expectation was that for small z-values, fast rendering is achieved because only few pixels contain data while for large z-values, the rendering time increases. Fig. 50 shows a different behavior. From small to large z-values, the rendering time rises with a maximum at about -600 to -700 from the gold standard position and then monotonously falls. This behavior cannot be explained. The position where the maximum occurs is approximately the distance at which the bounding structure totally fills the whole image for the first time. Although the screen is totally filled by

the bounding structure from this position on, the rendering time decreases. Maybe this is the result of parts of the bounding geometry being clipped early, but this cannot be told for sure. The hypothesis is that for parts of the geometry that show several front faces of the bounding structure, all of them produce a fragment and therefore for all of them the sampling is started, although only the data of the nearest fragment is finally rendered. If this border sections of the bounding structure are skipped, it has a large influence on the overall rendering performance. This happens if the object is closer to the eye position. This could also be an explanation why ray casting with unrefined bounding structure has its maximum at lower z-coordinates, because such a bounding structure is normally larger.

This effect is maybe similar to the problem that was observed when analyzing the performance for varying cuberille sizes for bounding structure refinement. For large cuberille sizes, the bounding volume is much larger than without any refinement. This leads to the fact that the unrefined bounding structure is "farther away" than the refined bounding structure. Similar to Fig. 50, the performance of the "farther away" unrefined bounding structure is worse than of the refined bounding structure, which would be a possible explanation for the observed effect.

When modifying the step size of the ray sampling, the rendering time can also be affected as illustrated in Fig. 48. Larger step sizes allow to traverse the volume faster but it is possible to miss important structures in the volume. Visual analysis has shown that the step size can be set to a value slightly larger than one voxel, but to guarantee that no feature is missed, this value should not be smaller.

An extreme influence — positive and negative — on the rendering time can be observed for different cuberille sizes for the bounding structure refinement. As already mentioned, a refined bounding structure can reduce the mean rendering time. Because the bounding structure loses its symmetry (a cube has nearly the same thickness in all directions), the rendering time gets sensitive to the rotation parameter of the rendering. This is observable in the standard deviations for the different cuberille sizes. These rise from around 1.8–4.3 ms up to 10–40 ms for smaller cuberille sizes. This shows that for some angles, short rendering times can be achieved while for other angles, the rendering times can be very slow. All these influences lead to the problem that the time of ray casting is far more unpredictable than when using wobbled splat rendering.

Bounding structure refinement in its current implementation has also the problem of creating cavities for certain datasets. Considering the dataset of a head, if only the bones shall be rendered, the cavity of the brain is also modeled in the bounding structure although the ray is sampled from the first front to the last back face. This implies that far more geometry has to be processed in the vertex shader than required. To remove this drawback, either the bounding structure creation has to be modified in order to skip cavities or the sampling routine has to be able to detect holes and skip them [8].

The use of refined bounding structures has also another drawback. To store these structures, much video memory is required. The test system (see Sect. 4.5) was for example not able to store the bounding structures with cuberilles of size two or one voxels or for marching cubes for the Pawiro pig dataset. These algorithms produce so many primitives that the memory is exceeded. Therefore, memory is an additional limitation for the bounding structure refinement.

Also the volume itself can lead to memory problems. The used test datasets are in conventional sizes for medical data. In the future, these sizes will rise and memory will become an important factor. Many publications show possibilities to avoid these problems by dividing the volume into smaller blocks that are stored and connected using a lookup texture [8, 39, 38, 44, 43].

Sometimes, DRRs generated with the current implementation of GPU based ray casting show inconsistencies of their intensities as illustrated in Fig. 44 (e). Although no parameters were changed, some DRRs or even only parts of the DRRs are darker than the rest. At the moment, there is no explanation for this problem but it seems to be some racing condition in the OpenGL API.

Other artifacts can occur because of the bounding structures. Fig. 44 (c) shows that in specific poses, the bounding cube can be visible as a shadow. When using cuberille refined bounding structures, these artifacts occur more often and can have a larger impact on the image quality as shown in Fig. 44 (a) and (b). Fig. 44 (a) shows the difference image between two DRRs rendered with and without a cuberille refined bounding structure. It was afterwards intensity scaled in order to prepare this information for visual analysis. It can be observed that some cuberilles are visible, although only very weak. Fig. 44 (b) shows a far more disruptive artifact. When using the Pawiro pig dataset at certain thresholds, some cuberilles are visible as dark cubes. It is not yet known why these artifacts occur and if they are the result of a problem in the ray casting algorithm, in the cuberille algorithm or in the Pawiro pig dataset. Additional tests with special artificial test datasets are required should allow to clarify this.

Ray casting is very sensitive to changes in the image size. As shown in Fig. 51, increasing the DRR size exponentially increases the required rendering time. Because ray casting is an image-space based method, this behavior was expected.

All in all, based on the data concerning the rendering methods without the context of 2D/3D-image registration, ray casting seems to be superior to wobbled splat rendering. For nearly all tested situations, ray casting is faster than wobbled splat rendering, even without additional optimization strategies. Also the quality of ray casted DRRs is significantly better. The refinement of the bounding structure has turned out to be not automatically improving the performance and should be further analyzed and optimized.

6.3 General aspects

To round up the discussion about the presented rendering algorithms in terms of quality and rendering time, some general and technical aspects have to be discussed in addition.

One conclusion from the measurements, which was already a result of Spoerk et al. [1], is, that CPU implementations of DRR rendering algorithms are not competitive to GPU implementations. This is definitely true for wobbled splat rendering and ray casting and probably also for shear warp factorization. Because of this, no evaluations of CPU based algorithms in comparison to GPU based algorithms were done in this thesis . Such evaluations can be found in many other works [1, 5, 122].

Evaluations of the merit function for different thresholds and visual analysis of the DRRs showed an inconsistency between the two rendering approaches as illustrated in Fig. 63. While wobbled

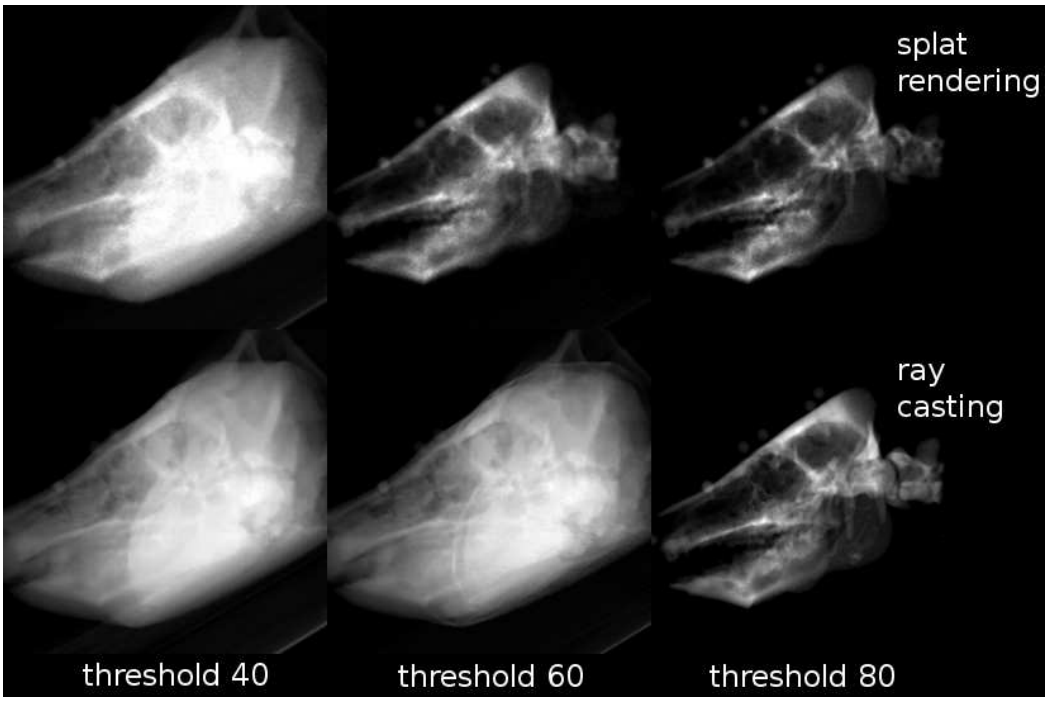

Fig. 63: DRRs rendered with wobbled splat rendering and ray casting with different thresholds. These DRRs show an inconsistency between the two rendering approaches.

splat rendering and ray casting produce nearly identical DRRs for a threshold of 40 and 80 for the Pawiro pig dataset, the DRRs for a threshold of 60 look totally different. The wobbled splat rendered DRR looks similar to the DRR with a threshold of 80 while the ray casted DRR looks similar to the DRR with a threshold of 40. It is unknown where this inconsistency is introduced and how it can be removed. Because of this problem, all quality measurements of the rendering algorithms and for the influence on the merit function were performed with a threshold of 40

instead of 60 to avoid data that shows big differences because of the DRRs looking different.

The comparison of absolute mutual information values between the different rendering methods from Fig. 45 reveals some additional facts. All methods have similar mutual information values, especially wobbled splat rendering with and without noise texture. One interesting fact is that, based on the mutual information values, ray casting seems to produce DRRs that are less similar to a real x-ray image, although for the human eye, they seem to have a better quality. The second interesting aspect is that a refined bounding structure improves the similarity, measured with mutual information, significantly. This is again in line with other data that shows that the cuberille refined bounding structure has a positive effect on the quality of the DRR. The comparison between ray casted DRRs with and without a refined bounding structure show that they are as different as wobbled splat rendered DRRs with and without noise texture. For the second case, the difference lies in the noise that is produced by the different random numbers.

Although presented in the theory part of this thesis (see Sect. 2.8.4), there are no implementations that use NVIDIA CUDA for accessing the GPU. This has several reasons. There were attempts to port the wobbled splat rendering algorithm to NVIDIA CUDA to evaluate if this allows to increase the rendering performance. One reason for stopping these attempts was that vertex buffer objects allowed to remove the bottleneck of retransferring the whole volume for every rendering. This bottleneck was the main motivation for a CUDA implementation. A second reason were the first results of a prototype implementation for wobbled splat rendering using NVIDIA CUDA. This prototype was quite simple. The preprocessed data was stored in an array at the video memory. Per voxel, a thread was started that calculated a unique index for the voxel data array, projected this voxel and stored the result in another array that was afterwards transferred back to the CPU. This straightforward approach suffered from several significant problems:

- The first problem is that one thread per voxel has to be used to process the data. As shown in the appendix, the test system for example has only 512 threads per block and also the number of blocks is limited. Therefore, especially for larger datasets, the number of blocks can be an upper bound which can only be solved by dividing the volume and therefore increasing the rendering time.
- The second problem is the high amount of memory required for this approach. All data has to be stored in the memory which seems to be less efficient with CUDA than with plain OpenGL/Cg.
- The third and biggest problem was the runtime of the algorithm, which was far behind the OpenGL/Cg implementation. In a non-CUDA shader, the final tests like alpha blending and z-buffering are hardwired and therefore very efficient. The problem with the CUDA prototype for wobbled splat rendering is that all threads have to write into a common

array which represents the output image. These writing operations are concurrent because the individual splats have to be summed. To perform this, CUDA has to synchronize the threads and the output array has to be stored in the global memory. Both, the synchronization and the access of global memory are very slow, which results in rendering times that were 10 to 15 times slower than with the plain OpenGL/Cg implementation.

There are maybe tweaks and tricks that allow to overcome these problems but it is likely that a CUDA implementation of wobbled splat rendering is not faster than the plain OpenGL/Cg implementation. CUDA uses some abstractions and generalizations that slow down the rendering but do not provide any advantages for the wobbled splat rendering algorithm itself, because it can be mapped very efficient to the graphics hardware pipeline.

Similar considerations lead to the decision not to implement ray casting using NVIDIA CUDA. Ray casting can be mapped very well to the graphics hardware pipeline and no generalizations or abstractions are required or are expected to bring advantages [8]. CUDA is a very good tool for implementing more general procedures like image filters or volume to surface transformations but cannot compete in the case of direct implementations of rendering algorithms.

6.4 Influences on the overall registration process

This section shall analyze the influence of the rendering method on the merit function and therefore on the quality of the overall registration process. The first impression from the diagrams showing the influence of ray casting and wobbled splat rendering on the merit function (see Fig. 52, 53, 54 and 55) is that ray casting is for all merit functions better than wobbled splat rendering. This impression is not true. The absolute value has no influence on the registration process, only the shape of the curves and the position of the optimum of the merit function has. It must also be stated that the shown diagrams are very exemplary and do not represent general attributes. They only give a hint on the general properties on the basis of one concrete example. When concentrating on the red, vertical lines in the diagrams, which represent the optima, in comparison to the blue, vertical lines, which represent the theoretical gold standard position, wobbled splat rendering sometimes, especially for y- and z-rotations, produces better results than ray casting. It is also never the case that the found optima of wobbled splat rendering is far worse than that of ray casting. All in all, wobbled splat rendering and ray casting can be seen to produce competitive results for the use in a 2D/3D-image registration procedure. More information and analysis about the different merit functions and the influence of the threshold on the quality of the registration, also in the context of the here presented rendering methods can be found in Gendrin et al. [114].

Fig. 56 and Fig. 57 show comparisons of the merit function shape for wobbled splat rendering with and without wobbling. Fig. 56 compares both methods for the Pawiro pig at its gold standard position. Because this position has no angles dividable by 45 and is translated from the origin,

no significant differences can be found in the shape of the functions. The algorithm without wobbling even seems to produce better results and has lower absolute values. This is maybe because wobbling introduces additional blurring and noise to the DRR. This blurring dilates features and therefore the margin in which the DRR "fits" to the gold standard image increases. If the reference image is in a pose that has rotations dividable by 90 degree and no (or only small) translations as shown in Fig. 57, the situation changes. This diagram shows the merit function with and without wobbling around an artificial gold standard with the pose $(0, 0, 0, 0, 0, 0)^T$ that was created using the ray casting algorithm. In this configuration, wobbled splat rendering shows better absolute values than splat rendering without wobbling. The local maxima, that can be observed exactly at the gold standard position for all three rotation parameters, have the biggest influence on the registration quality. For splat rendering without wobbling, these local maxima prevent a registration algorithm from finding the correct global optima and therefore lead to bad registration results. This diagram easily shows why wobbling of simplified splat rendering is important in the context of the 2D/3D-image registration process.

The use of a cuberille refined bounding structure has an enormous influence on the shape of the merit function as illustrated in Fig. 58 and Fig. 59 for thresholds of 40 and 60. The first diagram shows that the DRR rendered with a refined bounding structure has lower absolute values and better results. The second diagram shows that when only changing the threshold and keeping all other parameters, the DRR with unrefined bounding structure has lower absolute values although its results are still worse. This again shows how large the impact of the cuberille refined bounding structure is on the whole process. In general, the use of bounding structure refinement shows a positive influence on the overall registration process although for example with a threshold of 40 for the y- and z-rotation, it can lead to worse results. Again, similar to the results of the time measurements, the usefulness of a cuberille refined bounding structure depends on the dataset itself, the used threshold and the pose of the reference image and is therefore hard to estimate in advance.

A clear result is shown in Fig. 60 for the influence of sparse sampling on the shape of the merit function. An increasing amount of sparse sampling does not modify the optima significantly but only flattens the merit function around the optima and increases the absolute merit function values. This is also an explanation why there is an upper bound for the amount of sparse sampling. If the image content is too low, the merit function becomes so flat that the optima cannot be found reliably anymore.

Tab. 3 finally shows some data about the influence of the rendering method on the overall registration quality and performance. As expected from the results of Tab. 2, the use of wobbled splat rendering leads to longer total registration times but in contrast to the expectations does not produce worse results in term of registration quality. Ray casting performs better, but requires more iterations to find the optimum. An interesting effect is that a cuberille refined

bounding structure with a size of eight voxels is counterintuitive in this case. It reduces the number of iterations but increases the overall registration time significantly. Also an interesting effect is that the use of a cuberille refined bounding structure seems to improve the quality of the registration. This is in line with the results from Fig. 58 and Fig. 59 that also show hints for cuberille refined bounding structures improving the registration quality. As expected, sparse sampling improves the registration performance, but Tab. 3 shows that it additionally reduces the number of iterations and improves the quality.

The diagrams from Fig. 61 and 62 show the quality improvement for each registration. They clearly visualize that (for this concrete example), ray casting does not improve the quality significantly in comparison to wobbled splat rendering, but increases the deviation of the results. This means that more local optima exist. When using a cuberille refined bounding structure, the quality of the registrations is increased. The high absolute error is a result of the absence of further optimization techniques for the registration process. These measurements were only done to show differences between the rendering methods and use badly chosen thresholds. As Gendrin et al. [114] shows, using only a region of interest can further improve the total registration quality and errors are reduced from 10 to 1–2 mm mPD and mTRE. Also the choice of the appropriate threshold for a dataset is very important and can improve the quality. Therefore, for further analysis of the overall registration process and possible optimization techniques of it, refer to Gendrin et al. [114].

7 Conclusion and Outlook

This section summarizes the results of this thesis and gives a final conclusion about the question of how to achieve high performance DRR rendering for 2D/3D-image registration in radiation oncology. Finally, some topics for future work are presented.

7.1 Conclusion

Wobbled splat rendering is a very fast DRR rendering method that produces images of poor quality. Although ray casting is faster in most situations, wobbled splat rendering is still competitive and provides similar registration results. The utilization of vertex buffer objects allowed to remove one of the last big drawbacks of this method.

Ray casting allows to generate DRRs very fast and with a high quality. It is superior to wobbled splat rendering in most situations but does not necessarily produce better registration results. Surprisingly, a refined bounding structure using for example the cuberille algorithm improves the registration results but tends to slow down the rendering process. Sparse sampling has turned out to be a simple method that increases the performance without influencing the quality of the registration results significantly and should therefore be used where applicable.

As a conclusion, this thesis has shown ways to perform high performance DRR rendering in the context of 2D/3D-image registration without decreasing its quality. In addition to wobbled splat rendering [1, 111], ray casting has proven to be a well suited alternative for the rendering step of this process that is very flexible and good to optimize for a certain dataset. At the current state of development, it should be used as the default rendering approach.

To achieve real-time registration, the whole process of 2D/3D-image registration has to be further optimized. For DRR rendering, this thesis hits the algorithmic performance borders that can only be exceeded by using more powerful hardware. Therefore, for future development, the focus has to lie on the merit function calculation and the optimization algorithm. First results of these efforts can be seen in [113, 114].

7.2 Future Work

Several topics still remain for further work. This section gives a list of aspects that can be further improved or new topics that have appeared during the work on this thesis.

One topic is to further improve the implementations presented in this thesis. This includes two main points: refactoring and further optimizations of the software suite and searching for and removing remaining bugs and problems. Some of the already known bugs/problems are:

- There are still some artifacts when using wobbled splat rendering, even when using a noise texture. It is necessary to evaluate the way the parameters are used in the sine calculations

or for fetching the noise texture to remove these artifacts. This would maybe allow to decrease the overall amount of wobbling and therefore increase the image quality.

- One major problem is that wobbled splat rendering is sensitive to the current system state. It should be evaluated which parts cause these effects and how to make the algorithm more robust.
- The Gaussian filter for post processing the wobbled splat rendered DRR has to be implemented more efficiently or should be replaced by a better and faster filter.
- Because of some unknown reason, there are differences between ray casted DRRs with and without a cuberille refined bounding structure. These differences should be further analyzed and removed in order to utilize the performance gains of a refined bounding structure without the danger of decreasing rendering quality.
- For some poses, it is possible that the bounding structure could be visible. It has to be evaluated why this happens and ways have to be found to avoid this because it can have a large negative effect on the registration result.
- An automatic way has to be found to determine the optimal size of the bounding structure refinement kernels because a fixed size can have a negative effect on the registration performance.

Another topic for future work is to again evaluate if NVIDIA CUDA can be used to create DRRs faster than with the methods presented in this thesis. One advantage of such an implementation might be that the produced DRR could remain in the video memory for further post processing using CUDA. One example for such a post processing filter would be a Median filter for the wobbled splat rendering algorithm. This filter preserves edges while reducing the noise of the image. Because it is complex to calculate, CUDA would maybe allow to implement it while improving the overall performance of the rendering.

Also the calculation of parts or the merit function on the GPU using for example CUDA can be a topic for future developments. Kubias et al. [5] show for example how to implement this. A prototype implementation by the project group of this project showed that such an implementation is competitive to a CPU implementation and has again the advantage that no data has to be transferred between CPU and GPU during the rendering and the merit function calculation step.

A last topic for future developments is to implement a GPU based shear warp factorization algorithm and compare it to wobbled splat rendering and ray casting in terms of quality and performance. Because this methods is tailored to CPU implementations, it is not yet known if a GPU implementation is possible at all, but it is reasonable to try implementing this algorithm for the GPU.

List of Figures

Listings

List of Tables

Bibliography

[1] J. Spoerk, H. Bergmann, F. Wanschitz, S. Dong, and W. Birkfellner. Fast drr splat rendering using common consumer graphics hardware. *Medical Physics*, 34(11):4302–4308, 2007.

[2] E. B. Podgorsak and International Atomic Energy Agency. *Radiation oncology physics : a handbook for teachers and students*. International Atomic Energy Agency, Vienna, 2005.

[3] W. Birkfellner, M. Stock, M. Figl, C. Gendrin, J. Hummel, S. Dong, J. Kettenbach, D. Georg, and H. Bergmann. Stochastic rank correlation: A robust merit function for 2d/3d registration of image data obtained at different energies. *Medical Physics*, 36(8):3420–3428, 2009.

[4] X. Chen, R. C. Gilkeson, and B. Fei. Automatic 3d-to-2d registration for ct and dual-energy digital radiography for calcification detection. *Medical Physics*, 34(12):4934–4943, 2007.

[5] A. Kubias, F. Deinzer, T. Feldmann, D. Paulus, B. Schreiber, and Th Brunner. 2d/3d image registration on the gpu. *Pattern Recognition and Image Analysis*, 18(3):381–389, 2008.

[6] W. Birkfellner, M. Figl, J. Kettenbach, J. Hummel, P. Homolka, R. Schernthaner, T. Nau, and H. Bergmann. Rigid 2d/3d slice-to-volume registration and its application on fluoroscopic ct images. *Medical Physics*, 34(1):246–255, 2007.

[7] L. Zöllei, E. Grimson, A. Norbash, and W. Wells. 2d-3d rigid registration of x-ray fluoroscopy and ct images using mutual information and sparsely sampled histogram estimators. *Computer Vision and Pattern Recognition, IEEE Computer Society Conference on*, 2:696, 2001.

[8] H. Scharsach. Advanced gpu raycasting. In *Proceedings of the 9th Central European Seminar on Computer Graphics*, pages 69–76, May 2005.

[9] S. Grauer-Gray, C. Kambhamettu, and Palaniappan K. Gpu implementation of belief propagation using cuda for cloud tracking and reconstruction. In *Proceedings of the 5th IAPR Workshop on Pattern Recognition in Remote Sensing (PRRS)*, pages 1–4, Tampa, Fla, USA, 2008.

[10] M. Boyer, D. Tarjan, S. Acton, and K. Skadron. Accelerating leukocyte tracking using cuda: A case study in leveraging manycore coprocessors. In *23rd IEEE International Parallel and Distributed Processing Symposium*, pages 1–12, May 2009.

[11] L. Chin and W. Regine. *Principles and Practice of Stereotactic Radiosurgery*. Springer, New York, USA, 2008.

[12] M. H. Seegenschmiedt, A. Katalinic, H.-B. Makoski, W. Haase, G. Gademann, and E. Hassenstein. Radiation therapy for benign diseases: patterns of care study in germany. *International Journal of Radiation Oncology*Biology*Physics*, 47(1):195–202, 2000.

[13] D. Schulz-Ertner and H. Tsujii. Particle Radiation Therapy Using Proton and Heavier Ion Beams. *J Clin Oncol*, 25(8):953–964, 2007.

[14] G. M. Haase, D. P. Meagher Jr, L. K. McNeely, W. E. Daniel, M. A. Poole, M. Blake, and L. F. Odom. Electron beam intraoperative radiation therapy for pediatric neoplasms. *Cancer*, 74(2):740–747, 1994.

[15] D. M. Shepard, M. C. Ferris, G. H. Olivera, and T. R. Mackie. Optimizing the delivery of radiation therapy to cancer patients. *SIAM Rev.*, 41(4):721–744, 1999.

[16] E. J. Hall. Intensity-modulated radiation therapy, protons, and the risk of second cancers. *International journal of radiation oncology, biology, physics*, 65:1–7, 2006.

[17] C. G. Orton. Time-dose factors (TDFs) in brachytherapy. *Br J Radiol*, 47(561):603–607, 1974.

[18] J. W. Wong, M. B. Sharpe, D. A. Jaffray, J. M. Robertson, J. S. Stromberg, and A. A. Martinez. The use of active breathing control (ABC) to reduce margin for breathing motion. *International Journal of Radiation Oncology * Biology * Physics*, 44:911–919, 1999.

[19] D. A. Jaffray, D. Yan, and J. W. Wong. Managing geometric uncertainty in conformal intensity-modulated radation therapy. *Seminars in Radiation Oncology*, 9:4–19, January 1999.

[20] W. Birkfellner, R. Seemann, M. Figl, J. Hummel, C. Ede, P. Homolka, X. Yang, P. Niederer, and H. Bergmann. Wobbled splatting – a fast perspective volume rendering method for simulation of x-ray images from ct. *Physics in Medicine and Biology*, 50:73–+, May 2005.

[21] S. Dong, J. Kettenbach, I. Hinterleitner, H. Bergmann, and W. Birkfellner. The zernike expansion — an example of a merit function for 2d/3d registration based on orthogonal functions. In *MICCAI '08: Proceedings of the 11th International Conference on Medical Image Computing and Computer-Assisted Intervention, Part II*, pages 964–971, Berlin, Heidelberg, 2008. Springer-Verlag.

[22] M. B. Amin, A. Grama, and V. Singh. Fast volume rendering using an efficient, scalable parallel formulation of the shear-warp algorithm. In *PRS '95: Proceedings of the IEEE symposium on Parallel rendering*, pages 7–14, New York, NY, USA, 1995. ACM.

[23] A. Khamene, P. Bloch, W. Wein, M. Svatos, and F. Sauer. Automatic registration of portal images and volumetric ct for patient positioning in radiation therapy. *Medical Image Analysis*, 10(1):96 – 112, 2006.

[24] P. Lacroute and M. Levoy. Fast volume rendering using a shear-warp factorization of the viewing transformation. In *SIGGRAPH '94: Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, pages 451–458, New York, NY, USA, 1994. ACM.

[25] K. Mueller and R. Yagel. Fast perspective volume rendering with splatting by utilizing a ray-driven approach. In *VIS '96: Proceedings of the 7th conference on Visualization '96*, pages 65–ff., Los Alamitos, CA, USA, 1996. IEEE Computer Society Press.

[26] C. D. Hansen and C. R. Johnson. *The visualization handbook*. Elsevier - Academic Press, Amsterdam, 2005.

[27] M. Nelson. Optical models for direct volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 1(2):99–108, 1995.

[28] C. Rezk-Salama. *Volume rendering techniques for general purpose graphics hardware*. PhD thesis, University of Erlangen-Nürnberg, Erlangen-Nürnberg, 2004. `http://www.opus.ub.uni-erlangen.de/opus/volltexte/2004/15/`.

[29] D. Xue. *Volume Visualization using advanced Graphics Hardware Shaders*. PhD thesis, The Ohio State University, Department of Computer Science and Engineering, 2008.

[30] O. Kutter, A. Aichert, J. Traub, S. M. Heining, E. Euler, and N. Navab. Realtime volume rendering for high quality visualization in augmented reality. In *AMIARCS 2008*. MICCAI Society, September 2008.

[31] L. Mroz, H. Hauser, and E. Gröller. Interactive high-quality maximum intensity projection. *Computer Graphics Forum*, 19:341–350, 2000.

[32] P. Ljung, C. Winskog, A. Persson, C. Lundstrm, and A. Ynnerman. Full body virtual autopsies using a state-of-the-art volume rendering pipeline. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):869–876, 2006.

[33] J. W. Durkin and J. F. Hughes. Nonpolygonal isosurface rendering for large volume datasets. In *VIS '94: Proceedings of the conference on Visualization '94*, pages 293–300, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[34] J. Kruger and R. Westermann. Acceleration techniques for gpu-based volume rendering. In *VIS '03: Proceedings of the 14th IEEE Visualization 2003 (VIS'03)*, number 38, Washington, DC, USA, 2003. IEEE Computer Society.

[35] B. Domonkos, A. Egir, T. Foris, T. Juhasz, and L. Szirmay-Kalos. Isosurface ray-casting for autostereoscopic displays. In *Proceedings of WSCG*, pages 31 – 38, 2007.

[36] S. Frank and A. Kaufman. Out-of-core and dynamic programming for data distribution on a volume visualization cluster. *Comput. Graph. Forum*, 28(1):141–153, 2009.

[37] M. Wan, A. Kaufman, and S. Bryson. High performance presence-accelerated ray casting. In *VIS '99: Proceedings of the conference on Visualization '99*, pages 379–386, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.

[38] S. Grimm, S. Bruckner, A. Kanitsar, and E. Gröller. Memory efficient acceleration structures and techniques for cpu-based volume raycasting of large data. In C. Silva D. Silver, T. Ertl, editor, *Proceedings IEEE/SIGGRAPH Symposium on Volume Visualization and Graphics*, pages 1–8, October 2004.

[39] D. Ruijters and A. Vilanova. Optimizing gpu volume rendering. *Journal of WSCG*, 14(1–3):9–16, january 2006.

[40] M. Levoy. Efficient ray tracing of volume data. *ACM Trans. Graph.*, 9(3):245–261, 1990.

[41] P. Guerrero. Approximative real-time soft shadows and diffuse reflections in dynamic scenes. Master's thesis, Institute of Computer Graphics and Algorithms, Vienna University of Technology, Favoritenstrasse 9-11/186, A-1040 Vienna, Austria, 10 2007.

[42] R. Yagel and A. Kaufman. Template-based volume viewing. *Computer Graphics Forum*, 11(3):143–167, 1992.

[43] M. Hadwiger, C. Sigg, H. Scharsach, K. Bühler, and M. Gross. Real-time ray-casting and advanced shading of discrete isosurfaces. *Computer Graphics Forum*, 24(3):303–312, 2005.

[44] J. J. Choi, B.-D. Shin, Y. G. Shin, and K. Cleary. Efficient volumetric ray casting for isosurface rendering. *Computer Graphics Forum*, 24(5):661–670, october 2000.

[45] B. Smits. Efficiency issues for ray tracing. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Courses*, page 6, New York, NY, USA, 2005. ACM.

[46] M. Meißner, J. Huang, D. Bartz, K. Mueller, and R. Crawfis. A practical evaluation of popular volume rendering algorithms. In *VVS '00: Proceedings of the 2000 IEEE symposium on Volume visualization*, pages 81–90, New York, NY, USA, 2000. ACM.

[47] L. Westover. Footprint evaluation for volume rendering. *SIGGRAPH Comput. Graph.*, 24(4):367–376, 1990.

[48] D. Ruijters, B. M. ter Haar Romeny, and P. Suetens. Gpu-accelerated digitally reconstructed radiographs. In *Sixth IASTED international conference on biomedical engineering (BioMed)*, pages 431–435, 2008.

[49] N. Neophytou and K. Mueller. Gpu accelerated image aligned splatting. In *Fourth International Workshop pn Volume Graphics*, pages 197–242, june 2005.

[50] P. Lacroute. Analysis of a parallel volume rendering system based on the shear-warp factorization. *IEEE Transactions on Visualization and Computer Graphics*, 2(3):218–231, 1996.

[51] B. Cabral, N. Cam, and J. Foran. Accelerated volume rendering and tomographic reconstruction using texture mapping hardware. In *VVS '94: Proceedings of the 1994 symposium on Volume visualization*, pages 91–98, New York, NY, USA, 1994. ACM.

[52] S. Boda. Feature-based image registration. Master's thesis, National Institute of Technology, Rourkela, India, 2009.

[53] J.P.W. Pluim, J.B.A. Maintz, and M.A. Viergever. Mutual-information-based registration of medical images: a survey. In *IEEE Transactions on Medical Imaging*, Utrecht, Netherlands, 2003.

[54] M. M. Coselmon, J. M. Balter, D. L. McShan, and M. L. Kessler. Mutual information based ct registration of the lung at exhale and inhale breathing states using thin-plate splines. *Medical Physics*, 31(11):2942–2948, 2004.

[55] B. Zitova and J. Flusser. Image registration methods: A survey. *Image and Vision Computing*, 21(11):977–1000, 2003.

[56] A. Roche, G. Malandain, X. Pennec, and N. Ayache. The correlation ratio as a new similarity measure for multimodal image registration. In *MICCAI '98: Proceedings of the First International Conference on Medical Image Computing and Computer-Assisted Intervention*, pages 1115–1124, London, UK, 1998. Springer-Verlag.

[57] C. O Conaire, N. E. O'Connor, E. Cooke, and A. F. Smeaton. Detection thresholding using mutual information. In *VISAPP 2006 - International Conference on Computer Vision Theory and Applications*, pages 25–28, 2006.

[58] J. Weese, T. M. Buzug, C. Lorenz, and C. Fassnacht. An approach to 2d/3d registration of a vertebra in 2d x-ray fluoroscopies with 3d ct images. In *CVRMed-MRCAS '97: Proceedings of the First Joint Conference on Computer Vision, Virtual Reality and Robotics in Medicine and Medial Robotics and Computer-Assisted Surgery*, pages 119–128, London, UK, 1997. Springer-Verlag.

[59] R. Goecke, J. Weese, and H. Schumann. Fast volume rendering methods for voxel-based 2d/3d registration – a comparative study. In *Proceedings of International Workshop on Biomedical Image Registration '99*, pages 89–102, 1999.

[60] G. P. Penney, J. Weese, J. A. Little, P. Desmedt, D. L. G. Hill, and D. J. Hawkes. A comparison of simularity measures for use in 2d-3d medical image registration. In *MICCAI '98: Proceedings of the First International Conference on Medical Image Computing and Computer-Assisted Intervention*, pages 1153–1161, London, UK, 1998. Springer-Verlag.

[61] B. D. Lucas and T. Kanade. An iterative image registration technique with an application to stereo vision (darpa). In *Proceedings of the 1981 DARPA Image Understanding Workshop*, pages 121–130, April 1981.

[62] A. Roche, G. Malandain, N. Ayache, and S. Prima. Towards a better comprehension of similarity measures used in medical image registration. In *MICCAI '99: Proceedings of the Second International Conference on Medical Image Computing and Computer-Assisted Intervention*, pages 555–566, London, UK, 1999. Springer-Verlag.

[63] S. Clippe, D. Sarrut, C. Malet, S. Miguet, C. Ginestet, and C. Carrie. Patient setup error measurement using 3d intensity-based image registration techniques. *International Journal of Radiation OncologyBiologyPhysics*, 56:259–265, 2003.

[64] A. Roche, G. Malandain, N. Ayache, and X. Pennec. Multimodal image registration by maximization of the correlation ratio. Technical report, INRIA, 1998.

[65] L. M. G. Fonseca and B. S. Manjunath. Registration techniques for multisensor remotely sensed imagery. *Photogrammetric Engineering and Remote Sensing*, 62(9):1049–1056, september 1996.

[66] M. Gleicher. Projective registration with difference decomposition. *Computer Vision and Pattern Recognition, IEEE Computer Society Conference on*, 0:331–337, 1997.

[67] R. J. Koshel. Enhancement of the downhill simplex method of optimization. volume 4832, pages 270–282. SPIE, 2002.

[68] J. A. Nelder and R. Mead. A simplex method for function minimization. *The Computer Journal*, 7(4):308–313, January 1965.

[69] K. Mishchenko, V. Mishchenko, and A. Malyarenko. Adapted downhill simplex method for pricing convertible bonds. Quantitative Finance Papers 0710.0241, arXiv.org, October 2007.

[70] D. Lee and M. Wiswall. A parallel implementation of the simplex function minimization routine. *Comput. Econ.*, 30(2):171–187, 2007.

[71] G.-A. Turgeon, G. Lehmann, G. Guiraudon, M. Drangova, D. Holdsworth, and T. Peters. 2d-3d registration of coronary angiograms for cardiac procedure planning and guidance. *Medical Physics*, 32(12):3737–3749, 2005.

[72] H. P. A. Lensch, W. Heidrich, and H.-P. Seidel. Automated texture registration and stitching for real world models. In *PG '00: Proceedings of the 8th Pacific Conference on Computer Graphics and Applications*, page 317, Washington, DC, USA, 2000. IEEE Computer Society.

[73] W.-H. Pan, C.-K. Chiang, and S.-H. Lai. Adaptive multi-reference downhill simplex search based on spatial-temporal motion smoothness criterion. In *IEEE International Conference on Acoustics, Speech and Signal Processing, 2007. ICASSP 2007*, Honolulu, HI, USA, 2007.

[74] F. Maes, D. Vandermeulen, and P. Suetens. Comparative evaluation of multiresolution optimization strategies for multimodality image registration by maximization of mutual information. *Medical Image Analysis*, 3(4):373–386(14), december 1999.

[75] S. Kirkpatrick. Optimization by simulated annealing: Quantitative studies. *Journal of Statistical Physics*, 34:975–986, March 1984.

[76] W. L. Goffe, G. D. Ferrier, and J. Rogers. Global optimization of statistical functions with simulated annealing. *Journal of Econometrics*, 60:65–99, 1994.

[77] R. A. Rutenbar. Simulated annealing algorithms: an overview. *IEEE Circuits and Devices Magazine*, 5(1):19–26, 1989.

[78] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by Simulated Annealing. *Science*, 220:671–680, May 1983.

[79] M. Jenkinson, P. Bannister, M. Brady, and Smith S. Improved optimization for the robust and accurate linear registration and motion correction of brain images. *Neuroimage*, 17(2):825–841), october 2002.

[80] H. Carr, T. Theußl, and T. Möller. Isosurfaces on optimal regular samples. In *VISSYM '03: Proceedings of the symposium on Data visualisation 2003*, pages 39–48, Aire-la-Ville, Switzerland, Switzerland, 2003. Eurographics Association.

[81] W. E. Lorensen and H. E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. *SIGGRAPH Comput. Graph.*, 21(4):163–169, 1987.

[82] G. M. Nielson and B. Hamann. The asymptotic decider: resolving the ambiguity in marching cubes. In *VIS '91: Proceedings of the 2nd conference on Visualization '91*, pages 83–91, Los Alamitos, CA, USA, 1991. IEEE Computer Society Press.

[83] R. Shekhar, E. Fayyad, R. Yagel, and J. F. Cornhill. Octree-based decimation of marching cubes surfaces. In *VIS '96: Proceedings of the 7th conference on Visualization '96*, pages 335–342, Los Alamitos, CA, USA, 1996. IEEE Computer Society Press.

[84] C. Montani, R. Scateni, and R. Scopigno. Discretized marching cubes. In *VIS '94: Proceedings of the conference on Visualization '94*, pages 281–287, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[85] C. Dyken and G. Ziegler. High-speed marching cubes using histogram pyramids. In *Computer Graphics Forum*, volume 27, pages 2028–2039, 2008.

[86] N. K. Govindaraju, B. Lloyd, W. Wang, M. Lin, and D. Manocha. Fast computation of database operations using graphics processors. In *SIGMOD '04: Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 215–226, New York, NY, USA, 2004. ACM.

[87] T. Preis, P. Virnau, W. Paul, and J. J. Schneider. Accelerated fluctuation analysis by graphic cards and complex pattern formation in financial markets. *New Journal of Physics*, 11(9):093024 (21pp), 2009.

[88] C. J. Thompson, S. Hahn, and M. Oskin. Using modern graphics architectures for general-purpose computing: a framework and analysis. In *MICRO 35: Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*, pages 306–317, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[89] J. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Kruger, A. Lefohn, and T. Purcell. A Survey of General–Purpose Computations on Graphics Hardware. *Computer Graphics*, Volume 26, 2007.

[90] R. Fernando and M. J. Kilgard. *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[91] CORPORATE OpenGL Architecture Review Board. *OpenGL reference manual (2nd ed.): the official reference document to OpenGL, Version 1.1*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[92] M. J. Kilgard. Improving shadows and reflections via the stencil buffer. *nVidia Corp.*, 1999. http://developer.nvidia.com/attach/6641.

[93] K. Fatahalian and M.s Houston. A closer look at gpus. *Commun. ACM*, 51(10):50–57, 2008.

[94] D. Blythe. The direct3d 10 system. *ACM Trans. Graph.*, 25(3):724–734, 2006.

[95] C. Walbourn. Direct3D 11 Deployment for Game Developers, 2009. http://msdn.microsoft.com/de-at/library/ee416644%28en-us,VS.85%29.aspx [last visted on November 1st, 2009].

[96] M. J. Kilgard and K. Akeley. Modern opengl: its design and evolution. In *SIGGRAPH Asia '08: ACM SIGGRAPH ASIA 2008 courses*, pages 1–31, New York, NY, USA, 2008. ACM.

[97] M. Segal and Akeley K. *The OpenGL Graphics System: A Specification (Version 3.2 (Core Profile) - July 24, 2009)*. The Khronos Group Inc., July 2009.

[98] NVIDIA. Cg toolkit - gpu shader authoring language, version of April 2009. http://developer.nvidia.com/object/cg_toolkit.html [last visted on September 10th, 2009].

[99] NVIDIA. *Cg Language Specification - Cg Toolkit 2.2*. NVIDIA, 2009.

[100] G. Ruetsch and B. Oster. Getting started with cuda, 2008. http://www.nvidia.com/content/cudazone/download/Getting_Started_w_CUDA_Training_NVISION08.pdf [last visted on January 14th, 2010].

[101] NVIDIA. *CUDA Technical Training - Volume I: Introduction to CUDA Programming*. NVIDIA, 2008.

[102] NVIDIA. *NVIDIA CUDA Programming Guide Version 2.3.1*. NVIDIA, 2009.

[103] M. de Greef, J. Crezee, J. C. van Eijk, R. Pool, and A. Bel. Accelerated ray tracing for radiotherapy dose calculations on a gpu. *Medical Physics*, 36(9):4095–4102, 2009.

[104] P. Muyan-Ozcelik, J. D. Owens, J. Xia, and S. S. Samant. Fast deformable registration on the gpu: A cuda implementation of demons. In M. Gavrilova, O. Gervasi, A. Lagan, Y. Mun, and A. Iglesias, editors, *2008 International Conference on Computational Science and Its Applications*, pages 223–233, Los Alamitos, California, 2008. ICCSA 2008, IEEE Computer Society.

[105] S. Stone, J. Haldar, S. C. Tsao, W. Hwu, B. P. Sutton, and Z. Liang. Accelerating advanced mri reconstructions on gpus. *Journal of Parallel and Distributed Computing*, 68(10):1307–1318, October 2008.

[106] A. Bleiweiss. Gpu accelerated pathfinding. In *GH '08: Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, pages 65–74, Aire-la-Ville, Switzerland, Switzerland, 2008. Eurographics Association.

[107] J. E. Stone, J. Saam, D. J. Hardy, K. L. Vandivort, W. W. Hwu, and K. Schulten. High performance computation and interactive display of molecular orbitals on gpus and multi-

core cpus. In *GPGPU*, volume 383 of *ACM International Conference Proceeding Series*, pages 9–18. ACM, 2009.

[108] J. Kessenich. *The OpenGL Shading Language 1.50*. The Khronos Group Inc., 2009.

[109] Khronos OpenCL Working Group. *The OpenCL Specification 1.0.43*. The Khronos Group Inc., 2009.

[110] W. Birkfellner, J. Wirth, W. Burgstaller, B. Baumann, H. Staedele, B. Hammer, N. C. Gellrich, A. L. Jacob, P. Regazzoni, and P. Messmer. A faster method for 3d/2d medical image registration — a simulation study. *Physics in Medicine and Biology*, 48(16):2665–2679, 2003.

[111] C. Gendrin, J. Spoerk, C. Weber, M. Figl, D. Georg, H. Bergmann, and W. Birkfellner. 2d/3d registration at 1hz using gpu splat rendering. In *MICCAI 2009: Proceedings of the MICCAI-Grid 2009 Workshop*, pages 1–9, London, UK, 2009.

[112] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in C*. Press Syndicate of the university of Cambridge, Cambridge, 1994.

[113] S. A. Pawiro, C. Gendrin, M. Figl, M. Stock, C. Bloch, C. Weber, E. Unger, I. Nöbauer Huhmann, F. Kainberger, H. Bergmeister, D. Georg, H. Bergmann, and W. Birkfellner. A gold standard dataset for 2d/3d image registration evaluation. In *SPIE Medical Imaging*, San Diego, USA, 2010. accepted, conference will take place from 13.–18. February 2010.

[114] C. Gendrin, J. Spoerk, C. Bloch, S. A. Pawiro, C. Weber, M. Figl, P. Markelj, F. Pernus, D. Georg, H. Bergmann, and W. Birkfellner. Towards real-time 2d/3d registration for organ motion monitoring in image-guided radiation therapy. In *SPIE Medical Imaging*, San Diego, USA, 2010. accepted, conference will take place from 13.–18. February 2010.

[115] R. E. Bradley, L. A. D'Antonio, and C. E. Sandifer. *Euler at 300: an appreciation*. The Mathematical Association of America (MAA), 2007.

[116] P. Trier. Peter triers blog - gpu raycasting tutorial. `http://www.daimi.au.dk/~trier/?page_id=98` [last visted on January 14th, 2010].

[117] C. Bloyd. An implementation of marching cubes and marching tetrahedrons - siafoo. `http://www.siafoo.net/snippet/100` [last visted on January 20th, 2010].

[118] E. B. van de Kraats, G. P. Penney, D. Tomazevic, T. van Walsum, and W. J. Niessen. Standardized evaluation methodology for 2-D-3-D registration. In *IEEE Transactions on Medical Imaging*, volume 24, pages 1177–1189, 2005.

[119] D. Tomazevic, B. Likar, and F. Pernus. "gold standard" data for evaluation and comparison of 3d/2d registration methods. *3D/2D Registration of Medical Images*, pages 17–28, 2008.

[120] V. Podlozhnyuk. *Image Convolution with CUDA*. NVIDIA, June 2007.

[121] A. Kharlamov and V. Podlozhnyuk. *Image Denoising*. NVIDIA, June 2007.

[122] F. Ino, J. Gomita, Y. Kawasaki, and K. Hagihara. A gpgpu approach for accelerating 2-d/3-d rigid registration of medical images. In *Parallel and Distributed Processing and Applications (ISPA)*, volume 4330, pages 939–950, 2006.

Appendix

Shader for GPU based Focus Wobbled Splat Rendering

Listing 4: Optimized Cg vertex shader for GPU based focus wobbled splat rendering based on the implementation of Spoerk et al. [1]. At first, the displacement for the x- and y-direction is calculated. Afterwards, splatting is performed as geometric projection and resulting data is scaled according to some input parameters. Composition of the different splatted voxels is done using alpha blending.

```
void vs_main (  in                      float3          vs_iPosition    : POSITION,
                in                      float3          vs_iColor       : COLOR,

                in      uniform         float4x4        parameterMatrix[1],
                in      uniform         float           extremValues[5],

                out                     float4          vs_oPosition    : POSITION,
                out                     float4          vs_oColor       : COLOR)
{
        // for later focal wobbling
        float4x4 wobbleMat;
        float v1 = sin(vs_iPosition.y - vs_iColor.x) * sin(vs_iPosition.z + vs_iPosition.y);
        float v2 = sin(vs_iPosition.z + vs_iColor.x) * sin(vs_iPosition.x - vs_iPosition.z);
        float v3 = sin(vs_iPosition.x + vs_iColor.x) * sin(vs_iPosition.y + vs_iPosition.x);
        float w0 = parameterMatrix[0]._m00 * v1 + parameterMatrix[0]._m01 * v2 + parameterMatrix[0]._m02 * v3;
        float w1 = parameterMatrix[0]._m10 * v1 + parameterMatrix[0]._m11 * v2 + parameterMatrix[0]._m12 * v3;
        float invF = parameterMatrix[0]._m32;

        wobbleMat._m00_m10_m20_m30 = parameterMatrix[0]._m00_m10_m20_m30;
        wobbleMat._m01_m11_m21_m31 = parameterMatrix[0]._m01_m11_m21_m31;
        wobbleMat._m02_m12_m22_m32 = float4(    parameterMatrix[0]._m02 + invF * w0, parameterMatrix[0]._m12
                                              + invF*w1, 0.0f, parameterMatrix[0]._m32);
        wobbleMat._m03_m13_m23_m33 = float4(    parameterMatrix[0]._m03 + w0, w1 + parameterMatrix[0]._m13,
                                                0.0f , parameterMatrix[0]._m33);

        float4 vs_pos = float4(vs_iPosition.x, vs_iPosition.y, vs_iPosition.z, 1.0f);

        float4 tmpVek = mul(wobbleMat, vs_pos);
        tmpVek = tmpVek / tmpVek.w;
        vs_oPosition = float4(  tmpVek.x * extremValues[3], -tmpVek.y * extremValues[3],
                                0.0f, 1.0f);
        vs_oColor = float4(     0.0f, (vs_iColor.x - extremValues[0]) * extremValues[1],
                                0.0f, extremValues[2]);

        return;
}
```

Shader for GPU based Voxel Wobbled Splat Rendering

Listing 5: Optimized Cg vertex shader for GPU based voxel wobbled splat rendering based on the implementation of Spoerk et al. [1]. After the calculation of the individual voxel displacements in x-, y- and z-direction, splatting is performed as geometric projection and the resulting data is scaled according to some input parameters. Composition of the different splatted voxels is done using alpha blending.

```
void vs_main (  in                      float3          vs_iPosition    : POSITION,
                in                      float           vs_iColor,

                in      uniform         float4x4        parameterMatrix[1],
                in      uniform         float           extremValues[5],

                out                     float4          vs_oPosition    : POSITION,
                out                     float4          vs_oColor       : COLOR)
{
        float4x4 tmp = parameterMatrix[0];
        float u = 0.6f * sin(vs_iPosition.y - vs_iColor) * sin(vs_iPosition.z + vs_iPosition.y);
        float v = 0.6f * sin(vs_iPosition.z + vs_iColor) * sin(vs_iPosition.x - vs_iPosition.z);
        float w = 0.6f * sin(vs_iPosition.x + vs_iColor) * sin(vs_iPosition.y + vs_iPosition.x);
```

```
        vs_iPosition.x = vs_iPosition.x + u;
        vs_iPosition.y = vs_iPosition.y + v;
        vs_iPosition.z = vs_iPosition.z + w;
        float4 vs_pos = float4(vs_iPosition.x,vs_iPosition.y,vs_iPosition.z,1.0f);

        float4 tmpVek = mul(tmp, vs_pos);
        tmpVek = tmpVek / tmpVek.w;
        vs_oPosition = float4(tmpVek.x * extremValues[3], -tmpVek.y * extremValues[3], 0.0f, 1.0f);
        vs_oColor = float4(     (vs_iColor - extremValues[0]) * extremValues[1],
                                (vs_iColor - extremValues[0]) * extremValues[1],
                                (vs_iColor - extremValues[0]) * extremValues[1],
                                extremValues[2]);
        return;
}
```

Shader for GPU based Focus Wobbled Splat Rendering using a Noise Texture for Random Number Generation

Listing 6: Adaption of the Cg vertex shader for GPU based focus wobbled splat rendering as implemented by Spoerk et al. [1]. Instead of generating the "pseudo" random number using sine calculations, the numbers are fetched from a precomputed random number texture.

```
void vs_main (  in                      float3          vs_iPosition    : POSITION,
                in                      float3          vs_iColor       : COLOR0,

                in      uniform         float4x4        parameterMatrix[1],
                in      uniform         float           extremValues[5],

                        uniform         sampler3D       tex,

                out                     float4          vs_oPosition    : POSITION,
                out                     float4          vs_oColor       : COLOR)
{
        // for later focal wobbling
        float4x4 wobbleMat;
        float3 texc = (vs_iPosition.zxy * (0.015625f)) + extremValues[4];
        float4 v  = 1.5f * (tex3D(tex, texc) - 0.5f) * 2.0f;
        float w0 = parameterMatrix[0]._m00 * v.r + parameterMatrix[0]._m01 * v.g
                    + parameterMatrix[0]._m02 * v.b;
        float w1 = parameterMatrix[0]._m10 * v.r + parameterMatrix[0]._m11 * v.g
                    + parameterMatrix[0]._m12 * v.b;
        float invF = parameterMatrix[0]._m32;

        wobbleMat._m00_m10_m20_m30 = parameterMatrix[0]._m00_m10_m20_m30;
        wobbleMat._m01_m11_m21_m31 = parameterMatrix[0]._m01_m11_m21_m31;
        wobbleMat._m02_m12_m22_m32 = float4(parameterMatrix[0]._m02 + invF * w0,
                                    parameterMatrix[0]._m12 + invF * w1, 0.0f, parameterMatrix[0]._m32);
        wobbleMat._m03_m13_m23_m33 = float4(parameterMatrix[0]._m03 + w0,
                                   w1 + parameterMatrix[0]._m13, 0.0f, parameterMatrix[0]._m33);

        float4 vs_pos = float4(vs_iPosition.x,vs_iPosition.y,vs_iPosition.z,1.0f);

        float4 tmpVek = mul(wobbleMat, vs_pos);
        tmpVek = tmpVek / tmpVek.w;
        vs_oPosition = float4(  tmpVek.x * extremValues[3], -tmpVek.y * extremValues[3],
                                0.0f, 1.0f);
        vs_oColor = float4(     0.0f, (vs_iColor.x - extremValues[0]) * extremValues[1],
                                0.0f, extremValues[2]);

        return;
}
```

Shader for GPU based Sparsely Sampled Ray Casting

Listing 7: Cg program for fast GPU based ray casting of DRRs including a vertex shader for rotation/translation as well as two fragment shaders for the ray casting itself: with or without mask for sparsely sampling of the data. Ray casting is performed by calculating the direction vector between the currently rendered position (front face) and the already in another rendering pass prepared back face texture. Using this directional vector, the volume, accessible in the form of a 3D texture, is traversed in a loop and the values are composed. The final color is at the end written to the framebuffer.

```
// Define interface between the application and the vertex program
struct app_vertex
 {
   float4 Position     : POSITION;

};

// Define the interface between the vertex- and the fragment programs
struct vertex_fragment
 {
   float4 Position        : POSITION; // For the rasterizer
   float4 TexCoord        : TEXCOORD0;
   float4 Color           : TEXCOORD1;
   float4 Pos             : TEXCOORD2;
};

struct fragment_out
{
  float4 Color      : COLOR0;
};

// Raycasting vertex program implementation
vertex_fragment vertex_main( app_vertex IN )
{
  vertex_fragment OUT;

  // Get OpenGL state matrices
  float4x4 ModelView = glstate.matrix.modelview[0];
  float4x4 ModelViewProj = glstate.matrix.mvp;

  // Transform vertex
  OUT.Position = mul( ModelViewProj, IN.Position );
  OUT.Pos =  mul( ModelViewProj, IN.Position );
  OUT.TexCoord = IN.Position;
  OUT.Color = IN.Position;
  return OUT;
}

// Raycasting fragment program implementation
fragment_out fragment_main( vertex_fragment IN,
                            uniform float stepsize,
                            uniform sampler2D tex,
                            uniform sampler3D volume_tex
                           )

{
  fragment_out OUT;
  float2 texc = ((IN.Pos.xy / IN.Pos.w) + 1) / 2; // find the right place to lookup in the backside buffer
  float4 start = IN.TexCoord; // the start position of the ray is stored in the texturecoordinate
  float4 back_position  = tex2D(tex, texc);
  float3 dir = float3(0,0,0);
  dir = back_position.xyz - start.xyz;
  float len = length(dir.xyz); // the length from front to back is calculated and used to terminate the ray
  float3 norm_dir = normalize(dir);
  float3 delta_dir = norm_dir * stepsize;
  float delta_dir_len = length(delta_dir);
  float3 vec = start.xyz;
  float4 col_acc = float4(0,0,0,0);
  float alpha_acc = 0;
  float length_acc = 0;
```

```
  float4 color_sample;
  float alpha_sample;
  float end = len / delta_dir_len;

  for(int i = 0; i < end; i++)
    {
      color_sample = tex3D(volume_tex,vec);
      alpha_sample = color_sample.a * stepsize;
      col_acc   += color_sample / 128.0f;//(1.0 - alpha_acc) * color_sample * alpha_sample * 3;
      //alpha_acc += alpha_sample;
      vec += delta_dir;
      length_acc += delta_dir_len;
    }

  OUT.Color =  col_acc;
  return OUT;
}

// Raycasting fragment program implementation using texture based sparse sampling mask
fragment_out fragment_mask_main(  vertex_fragment IN,
                                  uniform float stepsize,
                                  uniform sampler2D tex,
                                  uniform sampler3D volume_tex,
                                  uniform sampler2D mask_tex
                                )

{
  fragment_out OUT;
  float2 texc = ((IN.Pos.xy / IN.Pos.w) + 1) / 2; // find the right place to lookup in the backside buffer
  if(tex2D(mask_tex,texc).x == 0) {
    OUT.Color = float4(0,0,0,1);
    return OUT;
  }
  float4 start = IN.TexCoord; // the start position of the ray is stored in the texturecoordinate
  float4 back_position  = tex2D(tex, texc);
  float3 dir = float3(0,0,0);
  dir = back_position.xyz - start.xyz;
  float len = length(dir.xyz); // the length from front to back is calculated and used to terminate the ray
  float3 norm_dir = normalize(dir);
  float3 delta_dir = norm_dir * stepsize;
  float delta_dir_len = length(delta_dir);
  float3 vec = start.xyz;
  float4 col_acc = float4(0,0,0,0);
  float alpha_acc = 0;
  float length_acc = 0;
  float4 color_sample;
  float alpha_sample;
  float end = len / delta_dir_len;

  for(int i = 0; i < end; i++)
    {
      color_sample = tex3D(volume_tex,vec);
      alpha_sample = color_sample.a * stepsize;
      col_acc   += color_sample / 128.0f;//(1.0 - alpha_acc) * color_sample * alpha_sample * 3;
      //alpha_acc += alpha_sample;
      vec += delta_dir;
      length_acc += delta_dir_len;
    }

  OUT.Color =  col_acc;
  return OUT;
}
```

Device Query Output of the Test Graphics Card

Listing 8: Output of the deviceQueryDrv program that is part of the CUDA SDK provided by NVIDIA. It shows many statistics of all present graphic cards on a system. The shown system is the test system used for this thesis.

```
CUDA Device Query (Driver API) statically linked version
There is 1 device supporting CUDA

Device 0: "Quadro FX 570M"
  CUDA Driver Version:                           2.30
  CUDA Capability Major revision number:         1
  CUDA Capability Minor revision number:         1
  Total amount of global memory:                 267714560 bytes
  Number of multiprocessors:                     4
  Number of cores:                               32
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       16384 bytes
  Total number of registers available per block: 8192
  Warp size:                                     32
  Maximum number of threads per block:           512
  Maximum sizes of each dimension of a block:    512 x 512 x 64
  Maximum sizes of each dimension of a grid:     65535 x 65535 x 1
  Maximum memory pitch:                          262144 bytes
  Texture alignment:                             256 bytes
  Clock rate:                                    0.95 GHz
  Concurrent copy and execution:                 Yes
  Run time limit on kernels:                     Yes
  Integrated:                                    No
  Support host page-locked memory mapping:       No
  Compute mode:                                  Default (multiple host threads can use this
                                                 device simultaneously)

Test PASSED
```