Entwicklung und Evaluation eines Bresenham-Algorithmus

für die Bildrekonstruktion eines Ultraschall-Computertomographen

BACHELORARBEIT

für die Prüfung zum

Bachelor of Science

des Studiengangs Angewandte Informatik

an der Dualen Hochschule Baden-Württemberg Mannheim

von

Norbert Spieß

7. Juni 2010

Bearbeitungszeitraum	12 Wochen
Matrikelnummer, Kurs	259586, TAI07AIM
Ausbildungsfirma	Karlsruher Institut für Technologie, Karlsruhe
Betreuer der Ausbildungsfirma	M.Sc. Michael Zapf
Gutachter der Dualen Hochschule	Prof. Dr. Eckhard Kruse

Erklärung

gemäß § 5 (2) der „Studien- und Prüfungsordnung DHBW Technik“ vom 18. Mai 2009.

Ich habe die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet.

Karlsruhe, 8. Juni 2010 _____

Norbert Spieß

Zusammenfassung

Am Karlsruher Institut für Technologie (KIT) arbeitet eine Wissenschaftlergruppe an einem Ultraschall-Computertomographen (USCT) zur Früherkennung von Brustkrebs. Dieser USCT erzeugt Daten durch Aussenden und Empfangen von Schallwellen, wobei das abgebildete Objekt mit diesen Schallwellen interagiert und diese verändert. Aus den aufgenommenen Signalen werden Bilder rekonstruiert, welche eine Abbildung des Objekts und dessen Bestandteile liefert.

Die Bildrekonstruktion der Reflektivität basiert auf einem SAFT (synthetic aperture focusing technique) genannten Verfahren, welches eine ellipsoidale Rückprojektion von Reflexionspulsen verwendet. Dieses Vorgehen ist mit sehr viel Aufwand verbunden, da für jeden Voxel des Bildes der entsprechende Bereich aus einem Signal genommen wird, und das für Millionen Signale. Mit einer ellipsoiden Rasterung eines Signalwertes und selektiver Wahl der zu verwendenden Signaldaten ließe sich Aufwand und Zeit sparen.

Ziel dieser Arbeit war das Entwerfen und Implementieren eines Algorithmus zum Berechnen eines Ellipsoiden nach dem Prinzip des Bresenham-Algorithmus.

Die Bresenham-Konzepte für geometrische Formen wurde für Kreise und Kugeln in den reellen Zahlenbereich erweitert. Anschließend wurde eine Evaluierung der Qualität und Performance der beiden Algorithmen durchgeführt.

Die Evaluierung der Algorithmen zeigte eine korrekte Rasterung der Kreise und Kugeln. Der Rasterungsfehler bleibt unter der geforderten Größe eines halben diagonalen Voxel. Dabei erreicht die Kreisrasterung eine Leistung von 12 MVoxel/sec und die Kugelrasterung $3,5$ MVoxel/sec.

Mit Annahme der Komplexitätszunahme von Kugel zu Ellipsoid um Faktor Zwei und Reduktion der verwendeten Daten, ergibt sich eine benötigte SAFT-Performance von 200 MVoxel/sec für ein vergleichbares Ergebnis. Mit angedeuteten Optimierungen und Parallelisierung der bresenhamartigen Algorithmen wird allerdings eine deutlich schnellere Lösung entwickelt werden können.

Die Arbeit bietet Konzepte und Vorgehen, um eine Rasterung von weiteren Körpern umzusetzen. Die entwickelten Konzepte können dazu wiederverwendet und erweitert werden. Die Bildrekonstruktion des USCT-Projekts kann dadurch langfristig gesehen beschleunigt werden und sich dem Ziel der Echtzeitrekonstruktion annähern.

Abstract

At Karlsruher Institute of Technology (KIT) an ultrasound computer tomograph system (USCT) is under development for early breast cancer detection. The USCT emits sound waves, which interfere with an object within the USCT tank and modulate the sound waves. The modulated sound waves are received by sensors of the USCT and called A-Scans.

With the A-Scans a reflectivity imaging of the inner structure of the object is possible. The imaging is done using a SAFT (synthetic aperture focusing technique) algorithm. The SAFT is based on ellipsoidal backprojection which requires processing of millions of A-Scans for each reconstructed voxel. Instead of SAFT an ellipsoidal rasterization could be performed which allows to reconstruct with the relevant A-Scan subset to optimize the reconstruction performance.

The aim of this work was the designing and implementation of an bresenham algorithm to rasterize an ellipsoid.

In this work two rasterization algorithm for circles in 2D and spheres in 3D were designed and implemented. The bresenham concepts for circles and spheres were extended and implemented to support floating point parameter.

The evaluation of the algorithm showed a proper rasterization of the circles and spheres. The achieved peak performance of the implementations were 12 MVoxel/sec for the circle algorithm and $3,5$ MVoxel/sec for the sphere algorithm.

Even with assuming an increase in complexity by a factor two for porting the sphere algorithm to an ellipsoid algorithm and using an reduction of the used A-Scan samples the effective speed of the SAFT algorithm can be achieved. Certainly with optimization and parallelization the bresenham algorithm performance can outperform the SAFT algorithm performance.

The developed algorithm and implementation can be still expanded to implement rasterization of other shapes. For the USCT project this work introduces the Bresenham-style algorithm for image reconstruction and encourages extended further research.

Inhaltsverzeichnis

Abkürzungsverzeichnis

A-Scan	Amplituden-Scan
B	Byte
cm	Zentimeter
CPU	Central Processing Unit
GB	GigaByte ($1 \cdot 10^6$ Byte)
GHz	GigaHertz ($1 \cdot 10^6$ Hertz)
IPE	Institut für Prozessdatenverarbeitung und Elektronik
KIT	Karlsruher Institut für Technologie
KPixel	KiloPixel ($1 \cdot 10^3$ Pixel)
KVoxel	KiloVoxel ($1 \cdot 10^3$ Voxel)
MATLAB	Matrix Laboratory
min	Minute
mm	Millimeter
MPixel	MegaPixel ($1 \cdot 10^6$ Pixel)
MRT	Magnetresonanztomographie
MVoxel	MegaVoxel ($1 \cdot 10^6$ Voxel)
PB	PetaByte ($1 \cdot 10^9$ Byte)
USCT	Ultraschall-Computertomographie
Voxel	Volumetric Pixel
RAM	Random Access Memory
SAFT	Synthetic Aperture Focusing Technique
sec	Sekunde

Abbildungsverzeichnis

Tabellenverzeichnis

Gleichungsverzeichnis

Listings

1 Einführung

Das folgende Kapitel soll einen Einblick in das Projekt des Ultraschall-Computertomographen geben. Weiterhin wird auf die verwendete Bildgebungsmethodik eingegangen und die Motivation dieser Arbeit daraus abgeleitet.

1.1 Brustkrebs

Am Institut für Prozessdatenverarbeitung und Elektronik (IPE) des Karlsruher Institut für Technologie (KIT) wird zur Zeit ein auf Ultraschall basierendes, bildgebendes Verfahren zur Brustkrebsfrüherkennung entwickelt. Brustkrebs ist die häufigste Krebsart bei Frauen. Im Jahr 2004 erkrankten ungefähr 57.000 Frauen in Deutschland neu daran, was einem Anteil an Krebserkrankungen von 28 % entspricht. Durchschnittlich sterben jedes Jahr in Deutschland ungefähr 18.000 Frauen an dieser Krebsart.[1] Das Problem liegt hierbei am zu späten Erkennen des Tumors. Erkrankt eine Brust, so ist der Tumor erst ertastbar, wenn er sich stark im Gewebe ausgebreitet hat. Aus diesem Grund wurden verschiedene Verfahren entwickelt, um eine Erkrankung frühzeitig zu erkennen.

Mammographie

Das Verfahren der Mammographie ist eine Röntgenuntersuchung der Brust. Sie bildet die bisher beste Möglichkeit Brustkrebs frühzeitig zu erkennen. Um die Messung durchführen zu können wird die zu messende Brust zwischen zwei Plexiglasscheiben gepresst, um die Tiefe des zu durchstrahlenden Gewebes so gering wie möglich zu halten. Dieses Pressen der Brust kann schmerzhaft sein und führt zu einer nicht reproduzierbaren Verformung der Brust.

Mithilfe der Mammographie werden ungefähr 85 bis 95 % aller Tumore entdeckt. Ist das Brustgewebe besonders dicht, beispielsweise bei jungen Frauen, so sinkt allerdings die Aussagekraft des projektierten Bildes. Durch die Quetschung der Brust ist die Messprojektion oft nicht mit späteren Aufnahmen vergleichbar. Wei-

terhin entsteht durch die Verwendung von Röntgenstrahlen eine Belastung des Körpers.[2]

Magnetresonanztomographie

Die Magnetresonanztomographie (MRT) arbeitet mit Magnetfeldern und Radiowellen um Schichtbilder der Brust zu erzeugen. Dabei entstehen sehr viele einzelne zwei-Dimensionale Bilder, welche übereinandergelegt die Brust repräsentieren. Die Untersuchung durch die MRT bildet sehr gut weiches Gewebe ab. Begründet werden kann dieser Fakt dadurch, dass bei der MRT die magnetischen Eigenschaften von Wasserstoffatomen verwendet werden. In weichem Gewebe, beispielsweise dem Gehirn oder inneren Organen, sind viele Wasserstoffatome enthalten. Harte Gewebe, zum Beispiel Knochen, beinhalten weniger und können dementsprechend schlechter dargestellt werden. Tumore und Entzündungen weisen oft einen sehr hohen Gehalt an Wasserstoff auf. Aus diesem Grund sind sie auf den Schichtbildern oft sehr gut vom restlichen Gewebe zu unterscheiden. Ähnliches Gewebe wird ähnlich abgebildet, was eine Identifikation erschwert. Deshalb wird den Patienten meist vorher ein Kontrastmittel verabreicht, welches sich über die Blutbahn im Körper verbreitet. Dieses Kontrastmittel erscheint im Bild heller als normales Gewebe, was das Erkennen von Arterien, Venen und Tumoren erleichtert, da sich das Mittel in diesen anreichert.[3]

Sonographie

Unter Sonographie versteht man die Verwendung von Ultraschall, um Gewebe zu durchleuchten. Man nutzt hierbei die Eigenschaften von Gewebe Schall zu dämpfen und abzulenken. Unterschiedliches Gewebe reflektiert, dämpft und lenkt die Schallwellen unterschiedlich ab. Aus den zurückgeworfenen Schallwellen können Bilder berechnet werden, welche die verschiedenen Gewebearten in unterschiedlichen Graustufen darstellen. Hierbei lassen Gebiete mit Wasser die Wellen fast ungehindert durch, während Knochen die Wellen vollständig reflektieren. Wasserreiche Gebiete erscheinen daraufhin im Ergebnis dunkel, Knochen sehr hell. Je nach Dichte des durchdrungenen Gewebes verändert sich die Helligkeit. Die Sonographie wird deswegen häufig zur Untersuchung von weichen Organen und Gewebe verwendet.[4]

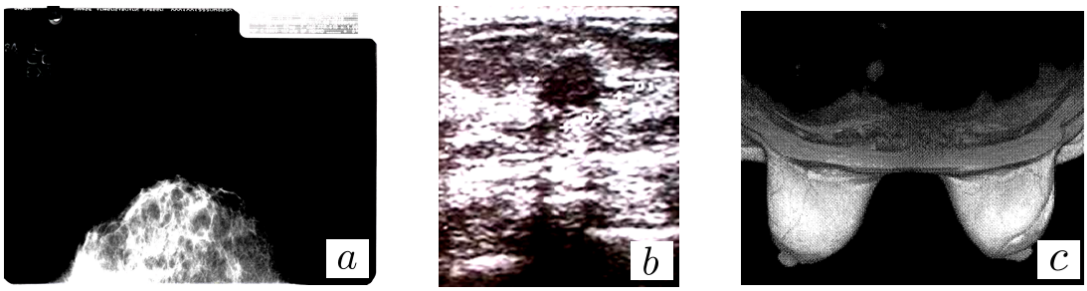

Abbildung 1.1: Herkömmliche Verfahren zur Brustkrebsfrüherkennung.
a) Mammographie b) MRT c) Sonographie.

1.2 Ultraschall-Computertomographie

Die Ultraschall-Computertomographie (USCT) ist ein neuartiges Verfahren zur Früherkennung von Brustkrebs. Sie bietet Schichtbilder sowie dreidimensionale Volumenbilder einer Brust. Diese weisen eine wesentlich höhere Bildqualität als herkömmliche bisher verwendete Verfahren auf, die Ultraschall verwenden. Diese Methode hat zudem keine negativen Auswirkungen auf den Körper der Patientin. Da mit Ultraschall gearbeitet wird, ist die Behandlung vollkommen ungefährlich für den menschlichen Körper beispielsweise im Gegensatz zur Mammographie. Außerdem wird die Brust nicht deformiert, sondern hängt frei in den Zylinder des USCT.

Der aktuell für Entwicklungszwecke verwendete 3D USCT I, siehe Abbildung 1.2, besteht aus einem mit Wasser als Koppelmedium gefülltem Edelstahlzylinder. Er ist mit circa 2000 Ultraschallwandlern, 384 Sendern und 1536 Empfängern, besetzt und kann um sechs Rotationsstufen gedreht werden. Bei einer Messung werden für jede Rotationsstellung alle Sender nacheinander angesprochen, eine Kugelwelle zu senden. Während ein Sender ein Signal aussendet, sind alle Empfänger innerhalb des Gerätes aktiv. Jeder Empfänger nimmt hierbei ein Signal, einen Implus über die Zeit, auf. Diese Signale werden als Amplituden-Scans (A-Scans) bezeichnet und entstehen durch einen direkten Signalweg und Ablenkung durch Objekte im Messbereich des USCT, siehe Abbildung 1.3. Dabei gibt es zwei Arten von Impulsen. Transmissionspulse, welche direkt von Sender zu Empfänger gelangen, und Reflexionspulse, welche durch Reflektion mit Streuern im Behälter und den Wänden entstehen. Eine komplette Messung umfasst ungefähr $3,5$ Millionen A-Scans, welche digital ungefähr 20 GigaByte (GB) entsprechen.

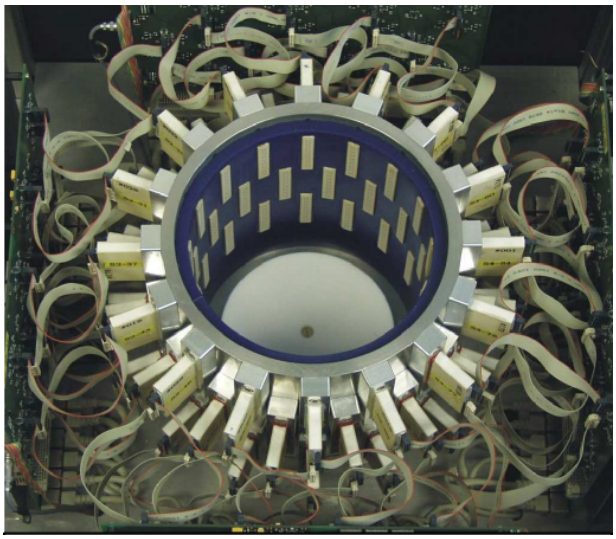

Abbildung 1.2: Blick in den 3D USCT I Messbehälter. *Der zylindrische Behälter ist mit Ultraschallwandlern bestückt. Diese sind in die weißen Bereichen innerhalb des Zylinders eingebettet. Die freien Stellen zwischen den Wandlern werden durch mögliche Rotationsschritte des gesamten Topfes ausgefüllt.*

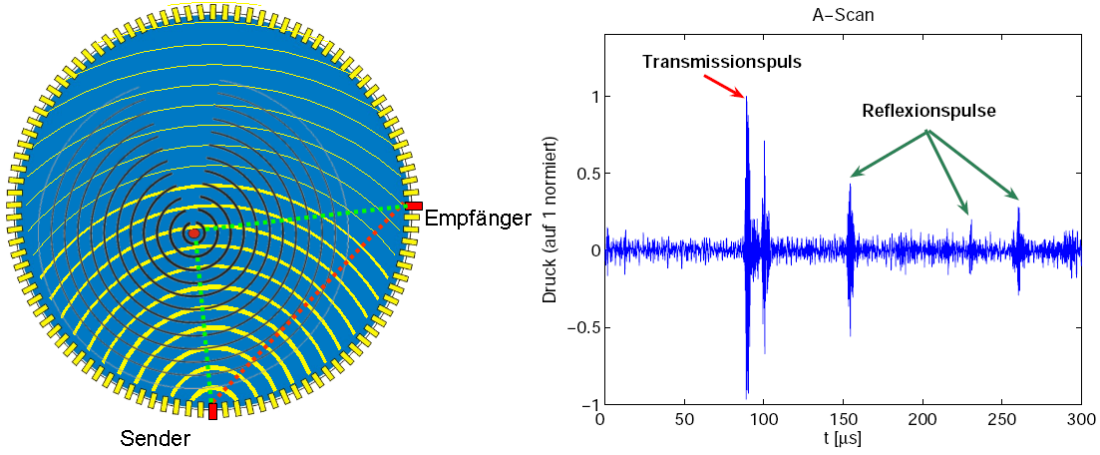

Abbildung 1.3: Methodik der Datenaufnahme beim USCT im zwei-Dimensionalen. *Im Messbehälter befindet sich ein Punktstreuer, hier als roter Punkt dargestellt. Er ist mit Wasser umgeben, hier blau dargestellt. Der Messbehälter ist mit Ultraschwallwandlern umgeben, welche als Sender und Empfänger arbeiten. Bei einer Messung senden iterativ einzelne Sender, während alle Empfänger das Signal pro Sender empfangen. Ein aufgenommenes Signal ist ein Amplituden-Scan über der Zeit mit 3000 Abtastwerten, genannt A-Scan. Er weist mehrere Pulse auf: Das direkte Signal, Transmissionspuls, sowie Reflexionspulse vom Punktstreuer und den Wänden des Messbehälters.*

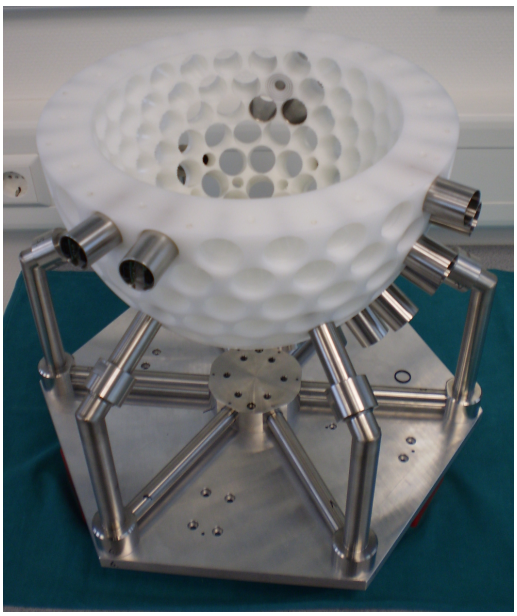

Abbildung 1.4: Der Messbehälter des 3D USCT II mit einzelnen eingesteckten Sensorhüllen. *Die neue ellipsoidale Form verspricht eine bessere Ausleuchtung des Messbehälterinhaltes durch die Sendesignale. Der 3D USCT II ist für die ersten klinischen Studien vorgesehen.*

Momentan wird an der Entwicklung des 3D USCT II gearbeitet, siehe Abbildung 1.4, welcher die bisherige Version ablösen soll. Die Besonderheit hier ist der ellipsoidale Aufbau des Messbehälters. Durch den Aufbau und dafür entwickelte Ultraschwallwandler verspricht man sich eine bessere Bildausleuchtung, und damit eine höhere Qualität der Signale. Mit diesen verbesserten A-Scans erhofft man bessere Bilder rekonstruieren zu können. Der 3D USCT II ist mit 628 Sendern und 1413 Empfängern ausgestattet. Weiterhin kann der gesamte Aufbau rotiert und gehoben werden.[5, 6]

1.3 Bildgebung beim USCT

Im Projekt des USCT wird eine SAFT (synthetic aperture focusing technique) genannte Methodik verwendet, um aus den A-Scans Bilder oder Volumen zu rekonstruieren. Dabei wird die Menge an, für die Rekonstruktion ausgewählten Signalen, iterativ behandelt. Die Verarbeitung eines A-Scan ist schematisch in Abbildung 1.5 dargestellt. Die Ausgangslage für das Verarbeiten sind die be-

kannten Positionen des Senders $\overrightarrow{P}_{Sender}(x,\, y,\, z)$, sowie die zugehörige Position des Empfängers $\overrightarrow{P}_{Empfänger}(x,\, y,\, z)$, die gewollte Auflösung, sowie die Start- und Endkoordinaten des Ergebnisvolumens. Weiterhin ist der vektorielle Abstand der Streuposition zu $\overrightarrow{P}_{Sender}$ gegeben. Mit Hilfe dieser Daten wird nun für jeden Ergebnisvoxel[1] der zuständige Bereich im A-Scan bestimmt. Dabei werden die beiden Punkte des Voxels ausgewählt, die den kleinsten und größten Abstand gegenüber des Mittelpunktes von Sender und Empfänger aufweisen und auf den Ellipsoiden der möglichen Streupositionen liegen, $\overrightarrow{P}_{min}$ und $\overrightarrow{P}_{max}$. Über diese Punkte werden die spezifischen vektoriellen Abstände D_{min} und D_{max} berechnet, welche anschließend auf den zeitlichen Aspekt des A-Scan projiziert werden. Der Wert des nun gewählten Bereiches wird in den Voxel hineingeschrieben.

Dieses Verfahren wird für alle zu verwendenden A-Scans durchgeführt und die Teilbilder über eine Maximaaddierung vereint. Bei einer hinreichend großen Anzahl an A-Scans ergeben sich durch die diversen Sender-Empfänger-Paare hohe Werte für den Bereich der stattgefundenen Reflektionen. Durch das Verwenden der kompletten A-Scans entstehen ungewollte Rauscheffekte.
Der Aufwand des Algorithmus in Speicherlesen und -schreiben beträgt für eine komplette Rekonstruktion $0,31$ PetaByte (PB), wie in Gleichung 1.1 aufgezeigt. Dieser Wert ergibt sich für ein beispielhaftes zwei-Dimensionales Schichtbild von $2048x2048$ Voxel. Dabei wird von A-Scans durch 384 Sendern, 1513 Empfänger und sechs Rotationspositionen ausgegangen. Die Datengröße liegt bei acht Byte (B). Im Algorithmus wird ein Abtastwert eines A-Scan und der aktuelle Voxelwert aus dem Speicher gelesen. Anschließend wird der neue Voxelwert zurückgeschrieben. Das entspricht zwei Lese- und einer Schreiboperation.[7]

$$(2\,\text{read} + 1\,\text{write}) \cdot 2048 \cdot 2048 \cdot 384 \cdot 1513 \cdot 6 \cdot 8\,\text{B} \;=\; 0,31\,\text{PB} \qquad (1.1)$$

Durch diese aufwändige Bildrekonstruktion kann eine Messung unfokussiert und damit schnell aufgenommen werden.

[1] Voxel - Ein Rasterpunkt im drei-dimensionalen Raum, entsprechend einem Pixel im zwei-dimensionalen Raum.

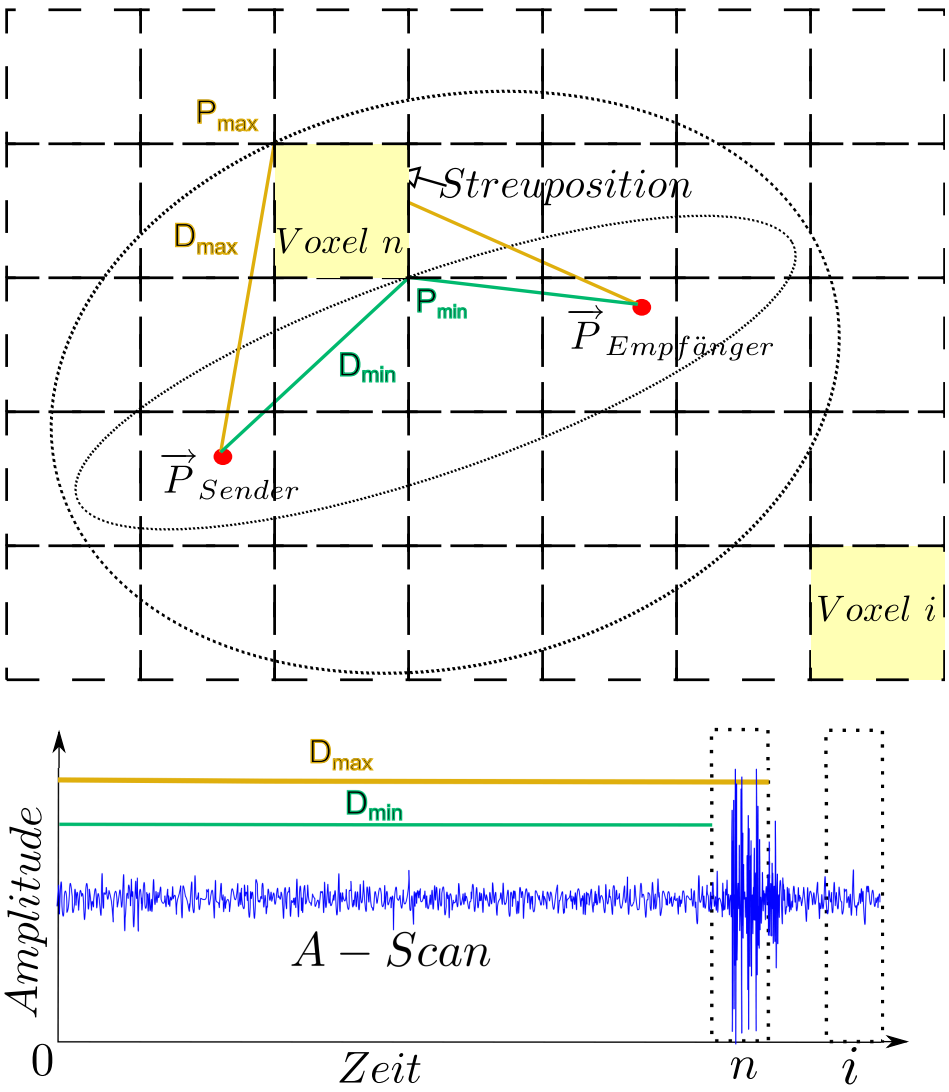

Abbildung 1.5: Die ellipsoidale Rückprojektion mit SAFT. *$\overrightarrow{P}_{Sender}$, $\overrightarrow{P}_{Empfänger}$ sind die Koordinaten der Sender-Empfänger-Kombination, D_{min}, D_{max} geben die Summe der vektoriellen Abstände von $\overrightarrow{P}_{Sender}$ bzw. $\overrightarrow{P}_{Empfänger}$ zu $\overrightarrow{P}_{min}$ bzw. $\overrightarrow{P}_{max}$ an. $\overrightarrow{P}_{min}$, $\overrightarrow{P}_{max}$ sind die geringsten beziehungsweise größten Abstände vom Mittelpunkt der Koordinaten $\overrightarrow{P}_{Sender}$, $\overrightarrow{P}_{Empfänger}$ und dem Voxel n. Voxel i ist ein weiterer Voxel im Raster.*
Bekannt sind die Position des Senders $\overrightarrow{P}_{Sender}$, des Empfängers $\overrightarrow{P}_{Empfänger}$, die gewollte Voxelauflösung, der vektorielle Abstand der Streuposition zu $\overrightarrow{P}_{Sender}$ sowie die Start- und Endkoordinaten des Ergebnisvolumens. Mit diesen Daten wird für jeden Voxel der zuständige Bereich im A-Scan bestimmt. Dazu werden die beiden Punkte $\overrightarrow{P}_{min}$ und $\overrightarrow{P}_{max}$ des Voxels mit zugehörigen Abständen D_{min} und D_{max} ausgewählt. Diese werden anschließend auf den zeitlichen Aspekt des A-Scan projiziert. Der Wert des nun gewählten Bereiches wird in den Voxel hineingeschrieben.

1.4 Motivation dieser Arbeit

Die Bildgebung des USCT basiert auf einfachen geometrischen Körpern, genauer rotationssymmetrischen Ellipsoiden. Weiterhin wird mit einem ganzzahligem Voxelraster gearbeitet. Untersuchungen im Laufe des Projekts des USCT haben gezeigt, dass nur ungefähr 1 % je A-Scan nötig sind, um eine Abbildung der gemessenen Objekte zu erzielen.[8]

Anstatt nun für jeden Voxel den Wert eines Bereiches im A-Scan zu berechnen, ist es möglich das Vorgehen umzukehren. Eine schematische Darstellung ist in Abbildung 1.6 aufgezeigt. Für einen Abtastwert D im A-Scan können die Voxel berechnet werden, die eine optimale Rasterung des Ellipsoids für die möglichen Streupositionen erreichen. Dies geschieht durch die bekannten Positionen des aktuellen Senders und Empfängers, $\overrightarrow{P}_{Sender}$ und $\overrightarrow{P}_{Empfänger}$ und Definition des zu verwendenden Rasters. Zum Berechnen der betroffenen Voxel ist ein Verfahren zu wählen, geometrische Körper auf einer Voxelrasterung möglichst optimal anzunähern.

Durch das Selektieren von spezifischen Abtastwerten eines A-Scan verspricht man sich eine stärkere Unterdrückung des Rauschverhaltens. Das bedeutet, dass die Aussagekraft eines Bildes durch bessere Abgrenzungen von Gewebegrenzen steigt. Weiterhin ist bei einem schnellen Rasterungsverfahren und dem Selektieren von konkreten A-Scan-Abtastwerten eine deutliche Verbesserung in der Laufzeit der Rekonstruktion zu erwarten.

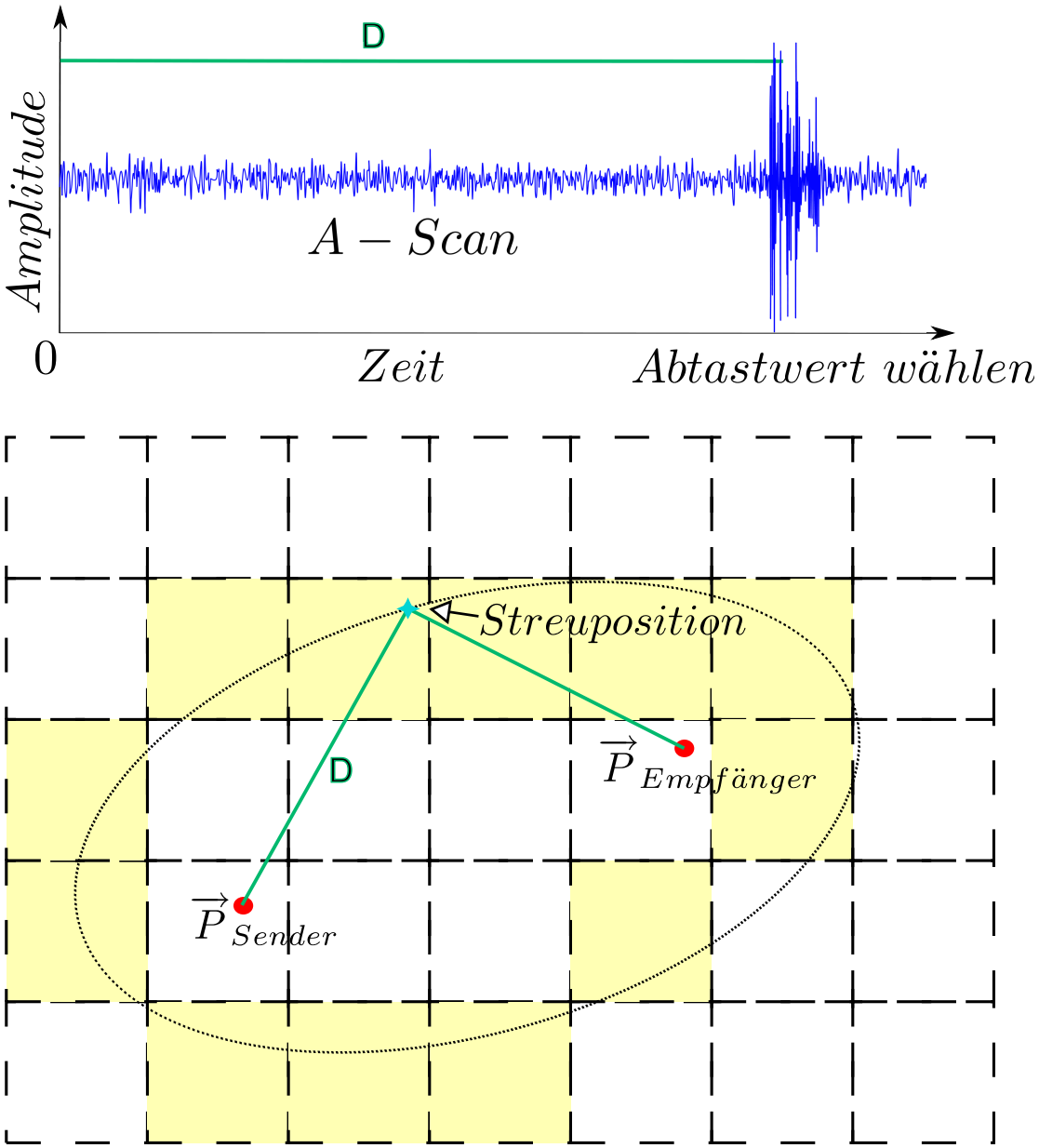

Abbildung 1.6: Rastern der Streupositionen aus einem A-Scan-Abtastwert. *$\overrightarrow{P}_{Sender}$, $\overrightarrow{P}_{Empfänger}$ stehen für die Koordinaten der Sender-Empfänger-Kombination, D steht für die Distanz für einen gewählten Abtastwert im A-Scan und ist die Summe der vektoriellen Abstände der beiden Koordinaten $\overrightarrow{P}_{Sender}$, $\overrightarrow{P}_{Empfänger}$ zu der Streuposition. Durch das Wählen eines Abtastwertes ist die Länge des Abstandes D gegeben, welcher die Summe aus den vektoriellen Abständen von $\overrightarrow{P}_{Sender}$ und $\overrightarrow{P}_{Empfänger}$ zur Streuposition ist. Über ihn kann eine Ellipse der möglichen Streupositionen berechnet werden, welche anschließend über die Voxel gerastert wird.*

2 Anforderungen und Projektmethodik

2.1 Anforderungen

Aus der Verwendung des geplanten Algorithmus ergeben sich Anforderungen, die beim Entwerfen zu beachten sind. Diese sollen im Folgenden aufgeführt und erläutert werden.

2.1.1 Anforderungen aus dem USCT-Projekt

Das Projekt des USCT stellt an den zu entwerfenden Algorithmus besondere Anforderungen. Diese sollen im Folgenden aufgelistet und erläutert werden.

Use Case der Bildrekonstruktion

Mit Abschluss des Projekts rund um den USCT werden Mediziner diesen und die zugehörige Bildrekonstruktion verwenden. Hier gibt es Vorstellungen und Grenzen für die Laufzeit der gesamten Bildrekonstruktion, in die der zu entwickelnde Algorithmus eingebettet werden wird. Die maximale gewünschte Dauer für eine Rekonstruktion liegt hierbei bei 30 min. Der zu entwerfende Algorithmus nimmt hierbei später einen Großteil der getätigten Berechnungen ein. Aktuelle Rekonstruktionen dauern ungefähr acht Stunden bei einem Volumen von $10\,cm^3$ mit einer Auflösung von $0,1\,\mathrm{mm}$. Das entspricht einem Ergebnisvolumen mit 1000^3 Voxel und damit einer Berechnungsgeschwindigkeit von 200 MVoxel/sec.[1] Gewünscht ist ein Algorithmus, der näher an das zeitliche Ziel heranreicht und langfristig echtzeitnah arbeiten kann.

[1] Messumgebung: Intel Core 2 Quad CPU 2, 4 GHz, 7 GB RAM, Microsoft Windows Server 2003 Standard x64 Edition Service Pack 2, MATLAB R2008a 64 Bit

Eingabewerte im Gleitkommazahlenbereich

Die Rekonstruktion des Messbehälterinhaltes basiert auf Sender- und Empfängerpositionen sowie dem Umfang des USCT-Messbehälters. Durch die verschiedenen Positionen und Kombinationen der Ultraschallwandler ist es notwendig, dass der hier existierende Parameterraum durch den zu entwerfenden Algorithmus abgedeckt wird. Das bedeutet für die Komponenten der zu berechnenden geometrischen Körper die Möglichkeit, eine Gleitkommazahl annehmen zu können. Dies betrifft die Start- und Endpunkte einer Geraden P_1 und P_2, den Mittelpunkt P_M sowie die Radien r, r_x, r_y, r_z bei Kreisen, Kugeln, Ellipsen und Ellipsoiden.

Keine doppelten Koordinaten

In der Bildrekonstruktion des USCT werden, wie im Abschnitt 1.3 erwähnt, eine große Anzahl an A-Scans verarbeitet und über eine ellipsoidale Backprojection vereint, siehe [7]. Bei duplizierten Koordinaten kommt es hier zu ungewollten Artefakten im Ergebnis. Aus diesem Grund ist darauf zu achten, dass der Algorithmus keine doppelten Koordinaten als Ergebnis liefert.

Rasterungsfehler

Durch die Rasterung der geometrischen Körper auf einem ganzzahligen Voxelraster wird eine Interpolation der diskreten Koordinaten vorgenommen. Hierbei soll der nearest neighbour Rasterungsfehler im zwei-dimensionalen den Wert $\frac{\sqrt{2}}{2} = 0,707$ Pixel nicht überschreiten. Im drei-dimensionalen ist der Grenzwert gleich dem Wert $\frac{\sqrt{3}}{2} = 0,866$ Voxel. Diese Werte ergeben sich aus der Diagonalen eines Pixel beziehungsweise Voxel mit der Kantenlänge Eins in allen Dimensionen, siehe Abbildung 2.1. Diese Anforderung ist notwendig, da die Ergebnisse für medizinische Prognosen und Operationen verwendet werden sollen. Schlechte Annäherungen können hier zu Fehldiagnosen führen.

Geschlossene Rasterung

Die gerasterten Körper dürfen keine Lücken im Verlauf der Rasterung aufweisen. Das bedeutet, dass die Geschlossenheit die untere Abgrenzung der Menge der Nachbarn pro Koordinate definiert. Zuviele Nachbarn sind nicht gewollt, was zu einer „optischen Dünne" führt.

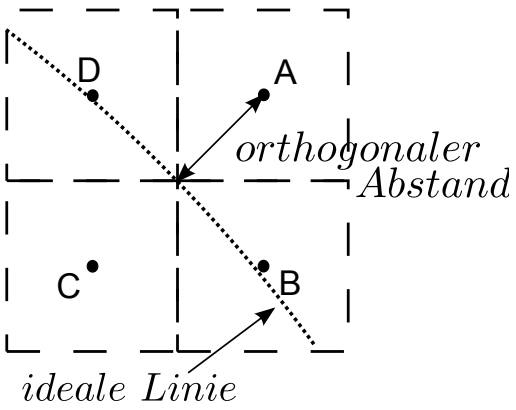

Abbildung 2.1: Illustration der Grenzwertherleitung der Rasterung. *Als Grenzfall der Rasterung kann der abgebildete Fall angenommen werden. Dabei verläuft die ideale Linie zwischen Pixel A und C. Wenn die Koordinate A oder C gewählt werden, entspricht der orthogonale Abstand der Diagonalen des Pixel. Die Länger der Diagonalen eines Quadrates mit Kantenlänge Eins ist* $\sqrt{2} \approx 1,414$*. Der maximale orthogonale Abstand für eine Rasterung entspricht der halben Diagonalen und damit* $\frac{\sqrt{2}}{2} = 0,707\, Pixel$ *im zwei-dimensionalen beziehungsweise* $\frac{\sqrt{3}}{2} = 0,866\, Voxel$ *im drei-dimensionalen.*

Robustheit

Der geforderte Algorithmus soll in seinem Entwurf robust sein. Das bedeutet, dass für alle Parametrisierung ein deterministisches konstantes Ergebnis geliefert wird.

2.1.2 Algorithmische Anforderungen

Aus den Projektanforderungen resultieren technische Anforderungen an den Entwurf. Diese sollen im Folgenden aufgelistet und erläutert werden.

Performance

Im Vordergrund des Algorithmusentwurfes steht die Korrektheit des Ergebnisses. Hierbei sollen Laufzeit- und Speicherbetrachtungen getätigt, jedoch keine optimale Rechenlösung für das Problem entwickelt werden. Optimierungen, beispielsweise durch Auftrennen der Operationen innerhalb des Algorithmus oder Parallelisierung, werden nicht gefordert.

Der Algorithmus soll in MATLAB implementiert werden, und nicht in hardwarenahen Sprachen, wie zum Beispiel einer Assemblersprache oder C. Das bedeutet

Einbußen in der Performance, da MATLAB nicht auf Geschwindigkeit optimiert ist.

Eingabe von spezifischen Parametern

Das in der Bildrekonstruktion bestehende Interface soll stabil bleiben. Das bedeutet für den zu entwickelnden Algorithmus eine spezifische Parametrisierung. Dabei werden die beiden Brennpunkte als Sender- und Empfängerpositionen $\overrightarrow{P}_{Sender}$ und $\overrightarrow{P}_{Empfänger}$ eingegeben. Weiterhin werden die beiden Abstände von Streuposition zu $\overrightarrow{P}_{Sender}$ und $\overrightarrow{P}_{Empfänger}$ summiert angegeben. Diese ist im A-Scan als Distanz zu einem Abtastwert zuordenbar. Die Bedeutung dieser Werte für die Ellipse ist in Abbildung 2.2 aufgezeigt. Weiterhin werden zusätzliche Parameter übergeben, um das zu verwendende Voxelraster zu definieren.

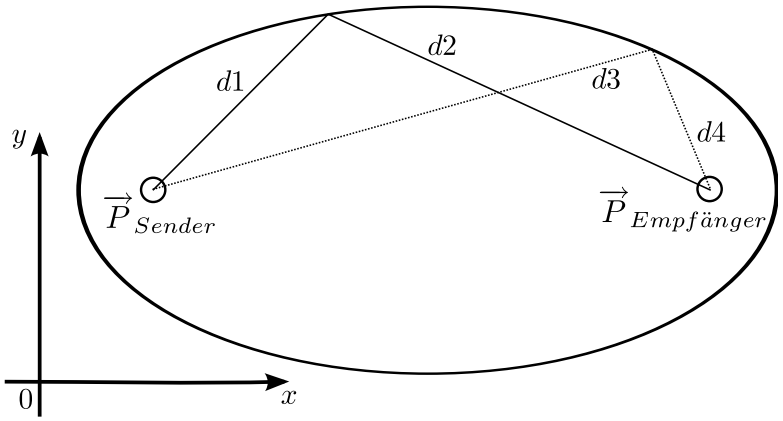

Abbildung 2.2: Geplante Eingabewerte für den Zielalgorithmus. *$\overrightarrow{P}_{Sender}$, $\overrightarrow{P}_{Empfänger}$ sind die Brennpunkte der Ellipse. d1 und d3 geben den Abstand eines Punktes zu $\overrightarrow{P}_{Sender}$ an, d2 und d4 den Abstand des selben Punktes zu $\overrightarrow{P}_{Empfänger}$. Eine Besonderheit hierbei ist, dass $d1 + d2 = d3 + d4$ für alle Punkte der Ellipse mit ihren spezifischen Abständen gilt. Die Brennpunkte und die Abstände sind als Eingabewerte für den Algorithmus geplant.*

2.1.3 Auflistung aller Anforderungen

Im Folgenden sind alle Anforderungen in der Tabelle 2.1 zusammengefasst. Dies soll der späteren Referenzierung dienen.

#	Anforderung
[Anf01]	Bildrekonstruktionsdauer $\leq$ 30 min
[Anf02]	$\{P_{start}, P_{end}, P_m, P_{Sender}, P_{Empfänger}, D, r, r_x, r_y, r_z\} \in \mathbb{Q}$
[Anf03]	Keine doppelten Koordinaten
[Anf04]	Orthogonaler Fehler $error_{\perp} \leq \frac{\sqrt{n}}{2}$ in n-ter Dimension
[Anf05]	Robuster Entwurf, Implementierung in MATLAB
[Anf06]	Auf Hardwarenahe Sprachen portierbare Lösung
[Anf07]	$\{P_{Sender}, P_{Empfänger}, D\}$ =Parameter für Ellipse
[Anf08]	Geschlossene Rasterungen

Tabelle 2.1: Anforderungen an das Projekt.

2.2 Projektmethodik

In dieser Arbeit wird eine umfangreiche Literaturrecherche durchgeführt werden. Die Abbildung der Vorgaben des Projektes und die Teilbereiche der Konzipierung, Implementierung und Evaluierung werden mit der Methodik der testgetriebenen Entwicklung umgesetzt werden.

2.2.1 Testgetriebene Entwicklung

Als methodisches Vorgehen für die Umsetzung der Teilbereiche Konzipierung, Implementierung und Evaluierung wurde die testgetriebene Entwicklung gewählt. Diese Wahl wurde getroffen, um zu garantieren, dass die entwickelten Konzepte und Implementierungen den gestellten Anforderungen entsprechen.

2.2.1.1 Metriken

Um die Performance und Effektivität der entworfenen bresenhamartigen Algorithmen zu testen, werden sie anhand mehrerer Metriken entgegen einer Brute-Force-Rasterung verglichen. Die Rasterung nach Brute-Force wird dabei als Optimum angenommen. Damit soll bewiesen werden, dass das algorithmische Ergebnis eine optimale Rasterung erreicht oder zumindest heranreicht. Die quantitative Evaluation der Qualität wird dabei über vier Metriken durchgeführt. Der maximale Fehler dient hierbei dazu, die Rasterannäherung der individuellen Rasterpunkte zu prüfen. Der mediane, arithmetische und quadratische Mittelwert sollen die Annäherung der Gesamtkurve beschreiben. Der Median gibt dabei die genaue Mitte der Werte an und bestimmt dabei den Durchschnitt der Werte. Der Median ist für viele Fälle robust entgegen Ausreißern. Der arithmetische Mittelwert gibt ebenfalls den Durchschnitt der Werte an. Dabei haben alle Werte die selbe Gewichtung und Ausreißer haben demnach einen identischen Einfluss wie die restlichen Werte. Der arithmetische Mittelwert sollte sich für hinreichend große Körper dem Wert Null annähern. Der quadratische Mittelwert maximiert, entgegengesetzt dem Median, die Auswirkungen von großen Abweichungen auf den Durchschnitt. Es wurde sich für mehrere Mittelwerte entschieden, um eine umfangreiche Metrikwahl zu ermöglichen.
Als Metriken der Performance wurde im zwei-dimensionalen Pixel/sec und im drei-dimensionalen Voxel/sec gewählt. Darüber lässt sich aussagen, wie sich die Laufzeit zur Komplexität der Rasterung verhält.

Quantitative Qualitätsmetriken:

Maximaler Fehler $error_{\perp max} = |\max(error_1, error_2, \ldots, error_n)|$

Median $\overline{error}_{med} = \begin{cases} \left|error_{\frac{n+1}{2}}\right| & n\,ungerade \\ \left|\frac{1}{2}\left(error_{\frac{n}{2}} + error_{\frac{n}{2}+1}\right)\right| & n\,gerade \end{cases}$

Arithmetischer Mittelwert $\overline{error}_{arithm} = \left|\frac{1}{n}\sum_{i=1}^{n} \mathrm{error}_i\right|$

Quadratischer Mittelwert $\overline{error}_{quadr} = \sqrt{\frac{1}{n}\sum_{i=1}^{n} error_i^2}$

Metriken der Performance:

Performance für
zwei-dimensionales Bild $Performance_{2D} = Pixel/sec$
drei-dimensionales Volumen $Performance_{3D} = Voxel/sec$

2.2.1.2 Unit-Tests

Um konzeptunabhängig die Anforderungserfüllung zu testen, wurden Unit-Tests formuliert. Diese gelten in der gesamten Entwicklungszeit und werden nach jeder Konzeptimplementierung abgeprüft. Sie resultieren aus den Anforderungen, welche in Abschnitt 2.1 erläutert und in Tabelle 2.1 zusammengefasst wurden.

Keine doppelten Punkte	Prüfung auf doppelt berechnete Voxelkoordinaten
Maximale Abweichung	n-dimensional: $error_{\perp max} \leq \left\lvert \pm \frac{\sqrt{n}}{2} \right\rvert$
Gleitkommaeingaben	Parameter $\in \mathbb{Q}$
Geschlossenheit	Keine Lücken in Koordinatenwahl

3 Stand der Forschung

Das folgende Kapitel befasst sich mit einer Literaturrecherche zum Rastern von geometrischen Körpern. Als Rasterungsverfahren wird der Bresenham-Algorithmus betrachtet und für das Projekt evaluiert. Andere Konzepte finden hier keine Betrachtung. Schlussfolgernd daraus werden die Ziele des Projektes ausformuliert.

3.1 Bresenham-Algorithmus

Der Bresenham-Algorithmus wurde von J. Bresenham 1965 veröffentlicht. Das Ziel dieses Algorithmus ist die beste Annäherung einer Geraden zwischen zwei Punkten in einem ganzzahligen Pixelraster. Dabei wird so wenig wie möglich Prozessorleistung benötigt. Um dies zu erreichen, arbeitet der Algorithmus nur mit ganzzahligen Integerwerten, Multiplikationen mit 2, Additionen und Subtraktionen. Die Qualität der Rasterung erfüllt dabei mehrere Eigenschaften. Die einzelnen Rasterpunkte nähern sich der zu rasternden Geraden optimal an. Weiterhin rastert der Algorithmus mit einer wohldefinierten Dicke. Jede gewählte Koordinate, ausgenommen Start- und Endpunkt, besitzt genau zwei Nachbarn.
Die Steigung einer Geraden variiert in einem zweidimensionalen kartesischen Koordinatensystem in acht Variationen, weshalb man die Quadranten des Systems noch einmal in jeweils zwei Bereiche unterteilen kann. Diese acht Bereiche werden als Oktanten bezeichnet und entgegen dem Uhrzeigersinn nummeriert. Die Nummerierung dieser Oktanten und die jeweils mögliche Steigung m einer Geraden ist in Abbildung 3.1a aufgezeigt. Eine Gerade liegt immer in einem dieser Oktanten und kann beliebig in einen der anderen transformiert werden. Die dazu benötigte Koordinatentransformation ist in Abbildung 3.1b aufgezeigt.

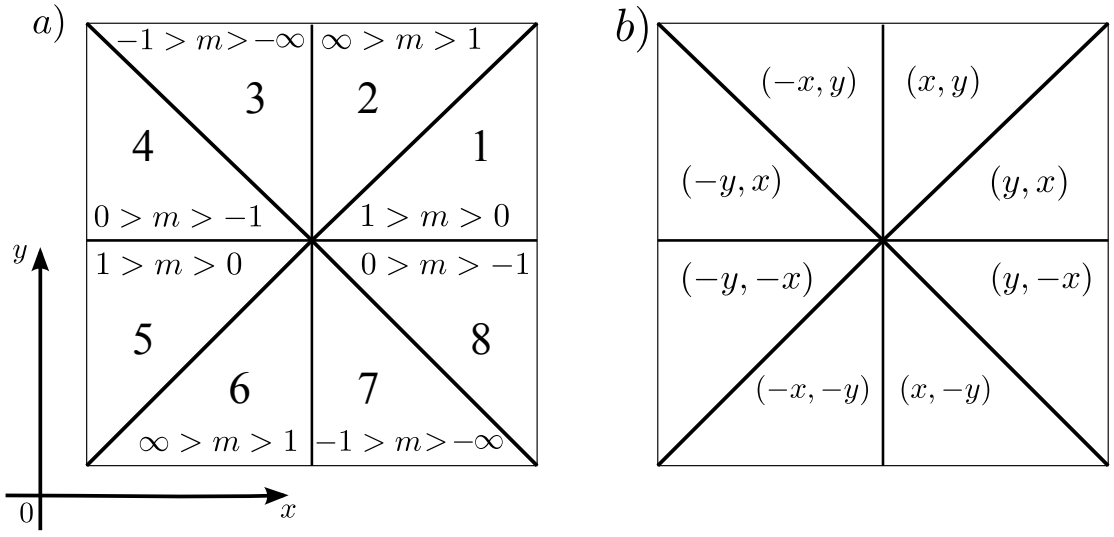

Abbildung 3.1: Oktantenaufteilung mit Geradensteigungseigenschaften und Symmetrie im zwei-dimensionalen. *Teilbild a zeigt die Aufteilung und Nummerierung des Koordinatensystems in Oktanten und den absoluten Verlauf der Steigung m einer Geraden. Diese wechselt in jedem Oktant des Koordinatensystems. Teilbild b zeigt das Symmetrieverhalten zwischen den Oktanten. Eine Gerade kann über die entsprechenden Koordinatentransformationen in jeden Oktanten gespiegelt werden.*

3.1.1 Prinzip des Bresenham-Algorithmus

Eine durch den originalen Bresenham-Algorithmus beschriebene Gerade liegt immer im ersten Oktant. Andere Oktanten beachtet der Algorithmus nicht, sondern erreicht diese durch Transformation der Ergebniskoordinaten über die Oktantensymmetrie. Im ersten Oktant gilt für die Steigung m einer Geraden $0 \leq m \leq 1$. Beim Rastern wird vom Bresenham-Algorithmus zuerst der Startpunkt gewählt, welcher optimal auf dem Pixelraster liegt. Danach wird eine Fehlervariable initialisiert, welche im Laufe des Algorithmus bei jedem Schritt in Richtung Endpunkt angepasst und als Entscheidungsvariable für die Auswahl der Schrittrichtung verwendet wird. Sie repräsentiert den Fehler, der durch die Interpolation auf dem Raster entsteht. Der Algorithmus läuft nun Schritt für Schritt auf dem Pixelraster entlang und wählt über die Fehlervariable inkrementell den zu tätigenden Schritt. Durch diese Fehlervariable wird dabei immer die optimale Pixelannäherung gewählt. Die möglichen Schrittrichtungen sind über die Eigenschaft $0 \leq y_{end} \leq x_{end}$ der Start- und Endpunktkoordinaten definiert. Für den Algorithmus hat das die Bedeutung, dass die x-Dimension bei jedem Schritt verändert wird. Die y-Dimension wird nur hin und wieder angepasst.[9] Bei einer Pixel-

größe mit dem Wert Eins sind die möglichen Schritte $(x + 1,\ y)$ und $(x + 1,\ y + 1)$. In Abbildung 3.2 ist eine mögliche Schrittwahl für eine Gerade mit den beiden Punkten $P_{start}\,(x_{start},\ y_{start})$ und $P_{end}\,(x_{end},\ y_{end})$ dargestellt.

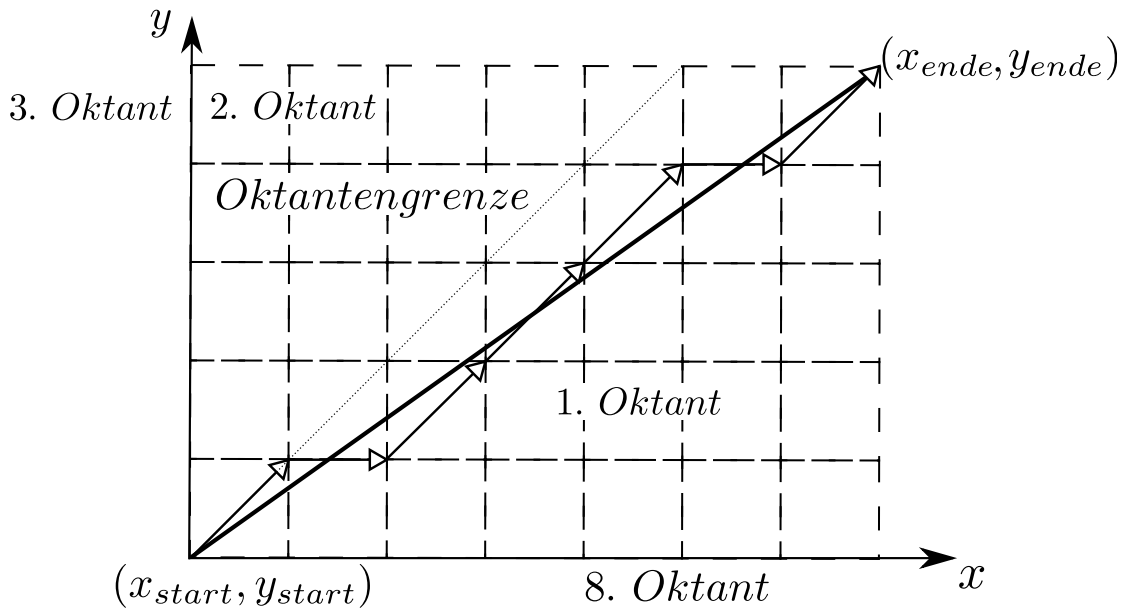

Abbildung 3.2: Schrittwahl im ersten Oktant. *Mögliche Schrittwahl für eine Gerade mit Startpunkt $(x_{start},\ y_{start})$ und Endpunkt $(x_{end},\ y_{end})$ nach dem Bresenham-Algorithmus. Die Pixelannäherung ist immer optimal, betrachtet man die Abstände der gewählten Koordinaten zur idealen Linie.*[1]

3.1.2 Algorithmischer Ablauf

In der Abbildung 3.3 ist ein Flussdiagramm zum Ablauf des Algorithmus dargestellt. Hierbei werden nach Start des Algorithmus die Initialwerte für die aktuellen Koordinaten x und y, die Fehleränderungen $error_x$ und $error_{xy}$ und die Fehlervariable *error* festgelegt. *error* ist durch die Eigenschaft $y_{end} \leq x_{end}$ negativ, außer wenn $x_{end} = y_{end}$. Nach einer Prüfung der Fehlervariable auf den Status $error < 0$ wird der nächste Schritt ausgewählt. Bei einem positiven Vergleich wird ein x-Schritt getätigt und *error* mit $error_x$ addiert. Der Wert von *error* nähert sich dabei der 0 an oder überschreitet sie. Sobald *error* einen positiven Wert annimmt, ist ein Schritt in den Dimensionen x und y notwendig. *error* wird anschließend durch

[1] Mit Änderungen entnommen aus: M. Pitteway (1967), S. 282

die Subtraktion mit $error_{xy}$ wieder negativ. Dieses Vorgehen wird durchgeführt, bis der Endpunkt erreicht ist.[10]

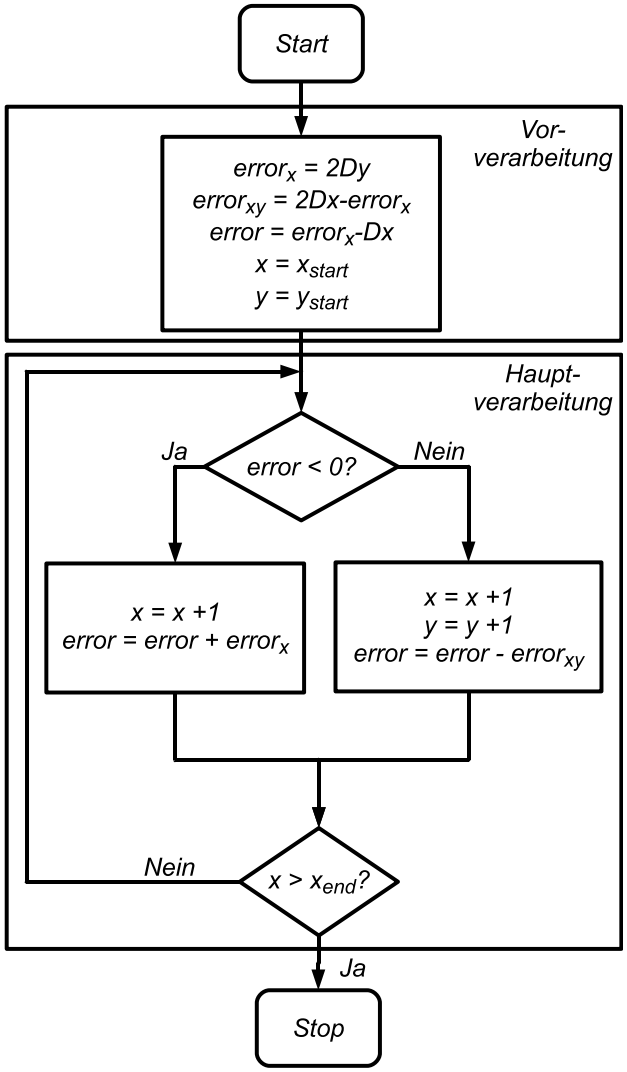

Abbildung 3.3: Flussdiagramm zum Ablauf des Bresenham-Algorithmus zur Geradenrasterung. *x, y geben die aktuellen Koordinaten an, die ausgewählt wurden. x_{end}, y_{end} sind die Endkoordinaten der Geraden. error ist die Fehlervariable, welche als Entscheidungsvariable für die Schrittwahl verwendet wird. $error_x$ ist die Fehlerveränderung bei einem x-Schritt, $error_{xy}$ bei einem xy-Schritt.*
Nach Initialisierung der ersten Koordinate und Fehlervariable in der Vorverarbeitung findet in einer Schleife die Schrittwahl als Hauptverarbeitung statt. Dies geschieht über den Status der Variable error. Ist der Endpunkt in x erreicht beziehungsweise überschritten ($x > x_{end}$), bricht der Algorithmus ab.[2]

[2]Mit Änderungen entnommen aus: M. Pitteway (1967), S. 282

3.2 Bresenham-Verfahren für den Kreis

J. Bresenham selber [11] und einige andere, z.b. [12, 13, 14], haben sich mit der Erweiterung des Bresenham-Verfahrens auf Kreise beschäftigt. M.D. McIlroy definiert in [15] die theoretischen mathematischen Eigenschaften einer Kurve nach dem Bresenham-Verfahren. Die sechs Eigenschaften sind in der Tabelle 3.1 aufgeführt. Sie stellen ein Optimum dar und werden in der Praxis oft nicht vollständig erreicht.

Eigenschaft	**Erläuterung**
Metrisch genau	Jeder Punkt der Rasterung sollte, unabhängig der Betrachtung, so nah wie möglich an der Kurve liegen.
Verbunden	Die Rasterkoordinaten sollten durch Schachzüge des Königs verbunden sein.
Topologisch korrekt	Der Verlauf der Schachzüge in der Rasterung sollte identisch der Laufrichtungen in der Originalkurve sein.
Dünn	Jede Rasterkoordinate sollte genau zwei, über Schachzüge des Königs erreichbare, Nachbarkoordinaten besitzen. Die Liniendicke ist die Folge aus der Eigenschaft der topologischen Korrektheit.
Symmetrisch	Die Rasterung sollte mit den symmetrischen Operationen des Rasters übereinstimmen: Parallelverschiebung, Rotation durch Multiplikationen mit $\pi/2$ und Reflektionen an horizontalen, vertikalen und diagonalen Achsen.
Beschreibbar	Die Rasterung sollte mathematisch beschreibbar sein, unabhängig vom zugehörigen Algorithmus.

Tabelle 3.1: Die theoretischen mathematischen Eigenschaften einer Kurve im Bresenham-Verfahren nach M.D. McIlroy. Diese Eigenschaften stellen ein Optimum dar und werden nicht immer erreicht.[3]

Mit Betrachtung der Kurvenrasterung wurde eine weitere Variante zur Wahl eines Schrittes entwickelt, der *Mittelpunktalgorithmus*. Dieser ist unterschiedlich zum Verfahren, welches von J. Bresenham bei der Geraden verwendet wurde und wird im Abschnitt 3.2.2 eingeführt. Das neue Konzept kann auch auf die Gerade angewendet werden.

Das von J. Bresenham verwendete Verfahren zum Wählen der Schritte basiert auf den Abständen zweier Punkte, weswegen diese Form als *Zweipunktalgorithmus* bezeichnet wird.

[3] Mit Änderungen entnommen aus: M.D. McIlroy (1992), S. 275–276

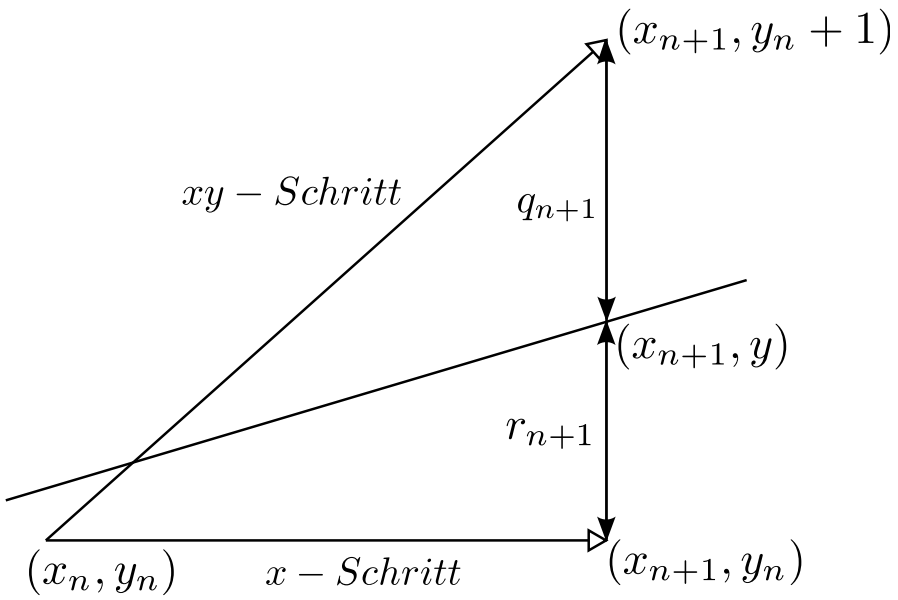

Abbildung 3.4: Die Schrittwahl beim Bresenham-Algorithmus nach dem Prinzip des Zweipunktalgorithmus. *Die Schrittwahl hängt von den beiden Abständen r_{n+1} und q_{n+1} ab. Der kleinere wird gewählt und führt zum nächsten Punkt (x_{n+1}, y_n) oder $(x_{n+1}, y_n + 1)$. (x_{n+1}, y) gibt den diskreten Wert der Geraden an.*[4]

3.2.1 Zweipunktalgorithmus

Der Zweipunktalgorithmus ist eine Methodik für die Auswahl eines Schrittes, siehe Abbildung 3.4.Es werden die beiden Abstände r_{n+1} und q_{n+1} der zwei möglichen Koordinaten $(x_{n+1}, y_n + 1)$ und (x_{n+1}, y_n) zur Wahrheit, dem diskreten Wert eines Körpers (x_{n+1}, y), verglichen und der kleinere gewählt. Dazu werden die beiden Abstände subtrahiert und das Vorzeichen betrachtet. Ist $q_{n+1} - r_{n+1} < 0$ wird (x_{n+1}, y_n) gewählt, bei $q_{n+1} - r_{n+1} \geq 0$ wird $(x_{n+1}, y_n + 1)$ ausgesucht.

3.2.2 Mittelpunktalgorithmus

1985 wurde von J. van Aken und M. Novak ein Algorithmus veröffentlicht, welcher eine Alternative zum Zweipunktalgorithmus darstellt, der Mittelpunktalgorithmus. Untersuchungen seitens der Autoren zeigten, dass dieser Algorithmus in Betrachtung des linearen Fehlers für gestreckte Ellipsen genauer arbeitet, ohne den Arbeitsaufwand zu erhöhen. Für Kreise sind die beiden Algorithmen in der Wahl der Koordinaten gleichwertig.[13]

[4]Mit Änderungen entnommen aus: M. Pitteway (1967), S. 283

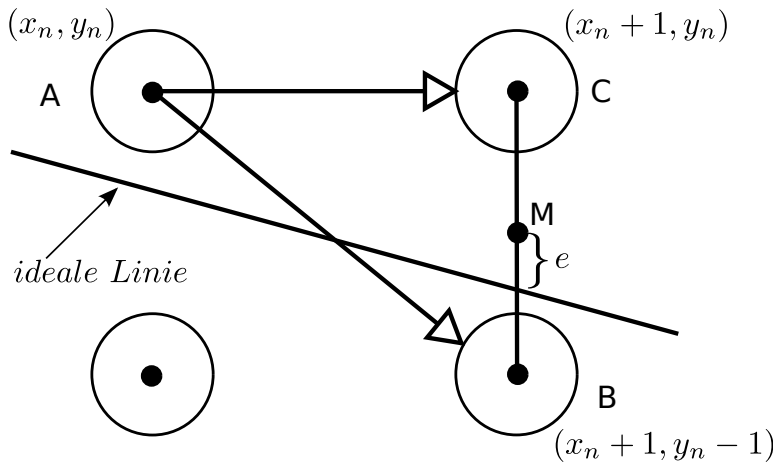

Abbildung 3.5: Auswahlverfahren beim Mittelpunktalgorithmus. *Von Pixel A* (x_n, y_n) *aus wird über M und e die Wahl zwischen den Koordinaten C* $(x_n + 1, y_n)$ *und B* $(x_n + 1, y_n - 1)$ *getroffen.*[5]

Das Prinzip dieses Algorithmus ist in Abbildung 3.5 dargestellt. Es wird mit dem Mittelpunkt zwischen den zwei Pixelkoordinaten C und B gearbeitet. Dabei sind die möglichen Schritte entweder $A \to C$ oder $A \to B$. Der Fehler e vom Mittelpunkt zur Wahrheit wird berechnet und als Entscheidungsvariable für die Schrittwahl verwendet. Dabei ist $e = 0$, wenn der Mittelpunkt auf dem Kreis, $e > 0$ wenn er außerhalb und $e < 0$ wenn er innerhalb des Kreises liegt.[14]

3.2.3 Algorithmischer Ablauf

Der Ablauf des Algorithmus verändert sich für einen Kreis nur wenig. Die Fehlervariablendefinition nimmt eine abgewandelte Form an und die Abbruchbedingung variiert entgegen dem Algorithmus der Geraden. In Abbildung 3.6 ist ein möglicher Ablauf aufgezeigt, welcher aus einem Codebeispiel bei [16] abgeleitet wurde. Berechnet wird der zweite Oktant, welcher anschließend transformiert wird. Dazu werden auch hier in der Vorverarbeitung die Startwerte *error*, x und y deklariert. x und y sind die Startwerte, die Schnittkoordinaten des Kreises mit der y-Achse. $error_0$ gibt den Startwert der Fehlervariable an, welche vom gewählten Vorgehen, Zweipunkt-/Mittelpunktalgorithmus, abhängt. Ebenso hängen die Variablen $error_x$ und $error_{xy}$ in der Hauptverarbeitung vom gewählten Vorgehen ab und stehen für die Fehlerveränderung pro jeweiligem Schritt. x_m und y_m stehen für die Koordinaten des Mittelpunktes P_m. Der Bereich *Koordinaten spiegeln* in

[5] Mit Änderungen entnommen aus: J. van Aken, M. Novak (1985), S. 153

der Nachverarbeitung gibt hierbei eine Menge an Anweisungen an, welche die berechneten Koordinaten des zweiten Oktant über Koordinatentransformationen zu einem kompletten Kreis spiegeln.

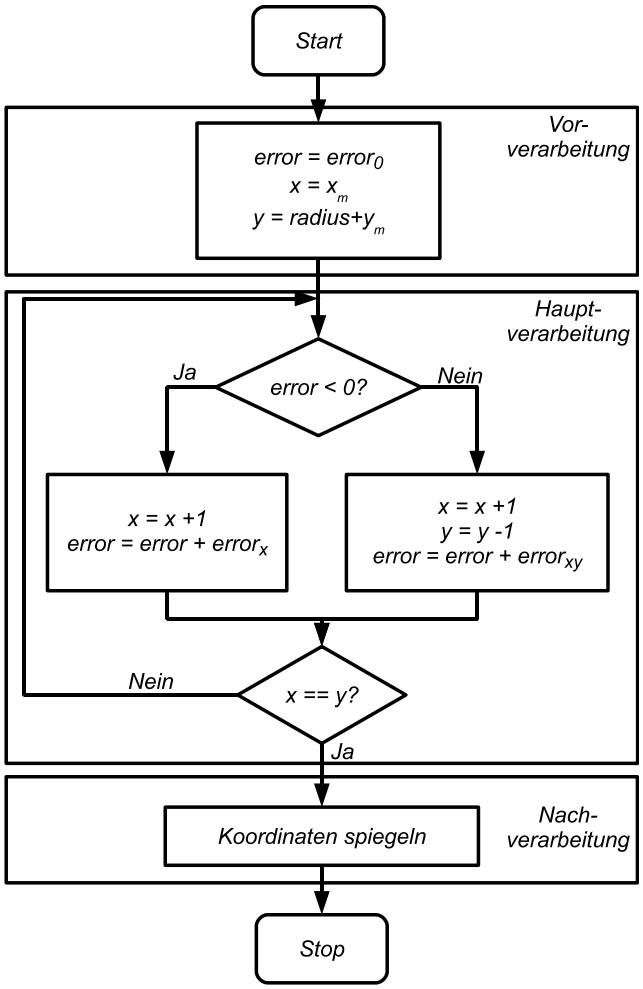

Abbildung 3.6: Flussdiagramm zum Ablauf des Bresenham-Kreisalgorithmus. *x, y geben die aktuellen Koordinaten an, die ausgewählt wurden. x_m, y_m geben die Koordinaten des Mittelpunktes des Kreises an. radius steht für den Radius des zu rasternden Kreises. error ist die Fehlervariable, welche als Entscheidungsvariable für die Schrittwahl verwendet wird. $error_0$, $error_x$ und $error_{xy}$ sind die Werte der Fehlervariable bei der Initialisierung bzw. die Fehlerveränderung bei einem x-Schritt oder xy-Schritt. Auch hier wird in der Vorverarbeitung initialisiert. Anschließend folgt eine Schleife zur Schrittwahl im zweiten Oktant als Hauptverarbeitung. In der Nachverarbeitung mit der Anweisung „Koordinaten spiegeln" findet die Transformation der Koordinaten in die anderen Oktanten statt. Dieser Schritt kann auch in die Hauptverarbeitung positioniert werden und beträfe in diesem Fall immer die aktuelle Koordinate.*

3.3 Bresenham-Verfahren für die Ellipse

Eine Ellipse ist einem Kreis sehr ähnlich. Auch sie weist einen Mittelpunkt $P_m\,(x_m,\, y_m)$ auf. Der Unterschied zum Kreis ist hier allerdings das Existieren von zwei Radien r_x und r_y, wobei einer in x, und einer in y wirksam ist. Eine Ellipse kann dadurch, wie ein Kreis, achsensymmetrisch sein. Die Achsen der Ellipse können allerdings gegenüber den Koordinatenachsen auch um einen bestimmten Winkel verschoben, also achsenasymmetrisch sein. Durch die Tatsache, dass ein Kreis ein Spezialfall einer Ellipse ist, bei dem r_x und r_y den gleichen Wert besitzen, kann der Kreisalgorithmus auf eine achsensymmetrische Ellipse portiert werden.

3.3.1 Achsensymmetrisch

Eine achsensymmetrische Ellipse zeichnet sich durch die Parallelität von r_x zur x-Achse und r_y zur y-Achse aus, siehe Abbildung 3.7a. Hier kann, ähnlich wie beim Kreis, die Symmetrieeigenschaft für den Algorithmus ausgenutzt werden, siehe Abbildung 3.7b. Durch die beiden Radien muss allerdings ein kompletter Quadrant berechnet werden, welcher anschließend gespiegelt werden kann. Das wird erreicht, indem zwei Oktanten getrennt berechnet werden, welche zusammen einen Quadrant bilden, beispielsweise erster und zweiter Oktant ergeben zusammen den ersten Quadrant. Die Oktantengrenze im Quadranten in Abbildung 3.7c ist gekennzeichnet durch den Wert für die Tangentensteigung $m_t = -1$.

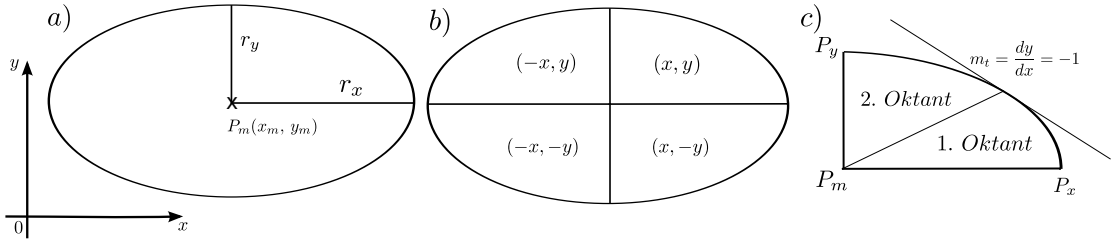

Abbildung 3.7: Eigenschaften einer achsensymmetrischen Ellipse mit Symmetrieverhalten. *P_m gibt den Mittelpunkt an, r_x und r_y die spezifischen Radien, P_y ist der positive Schnittpunkt mit der y-Achse, P_x der positive Schnittpunkt mit der x-Achse. m_t gibt die Steigung der angelegten Tangente an, dy und dx sind die Dynamiken der y- und x-Koordinaten.*[6]

[6] Mit Änderungen entnommen aus: J. Kennedy, S. 6; J. van Aken und M. Novak (1985), S. 157.

J. van Aken hat in [13] einen solchen Algorithmus veröffentlicht. Hier unterscheidet er, wie beim Kreis, zwischen Zweipunkt- und Mittelpunktalgorithmus. J. van Aken zeigt in seiner Veröffentlichung, dass der Mittelpunktalgorithmus hierbei für stark unterschiedliche Werte bei r_x und r_y (Faktor vier) den kleineren maximalen linearen Fehler generiert.

Es gibt noch weitere Algorithmen, um eine achsensymmetrische Ellipse berechnen zu können. J. Kennedy geht hierbei oktantenweise vor und berechnet die Oktanten eins und zwei jeweils von ihren Schnittpunkten mit den Achsen, um sie während der Berechnung auf die anderen Oktanten zu spiegeln. Dabei geht er komplett inkrementell für den Fehler und die Abbruchbedingung der Schleifen vor.[17] J. Kennedy gibt allerdings keine Grenzen oder getestete Szenarien für seinen Algorithmus an, weswegen dieser für diese Arbeit nicht verwendet wird.

Ähnlich sieht es bei einer Variante von D. Eberly aus. Er veröffentlichte in [18] algorithmische Vorgehensweisen für eine achsensymmetrische und -asymmetrische Ellipse, geht aber ebenfalls nicht auf Grenzfälle und Probleme ein. Weiterhin gibt er zwar Code an, allerdings nur für die symmetrische Variante. Seine Methode ist das Berechnen des ersten Quadranten, um diesen anschließend zu Spiegeln, beziehungsweise er berechnet die allgemeine Ellipse über Unterkreise.

3.3.2 Achsenasymmetrisch

Eine allgemeine Ellipse ist eine achsensymmetrische Ellipse mit Winkelverschiebungen entgegen den Achsen des Koordinatensystems. Durch die Verschiebungen bleibt die Symmetrieeigenschaft einer Ellipse erhalten, jedoch verändern sich die Achsen, an denen gespiegelt werden kann.

Ausgehend von der Kegelschnittgleichung 3.1 stellt M. Pitteway einen Algorithmus auf, welcher eine komplette Ellipse berechnet.

$$k = \alpha \cdot y^2 + \beta \cdot x^2 + 2\gamma \cdot x \cdot y + 2 \cdot dx \cdot y - 2 \cdot dy \cdot x \tag{3.1}$$

Hierfür nutzt er die Möglichkeit mit einem Kegelschnitt eine Gerade zu definieren. Aus dem Linienalgorithmus nach J. Bresenham wird damit ein Algorithmus zum Rastern einer allgemeinen Ellipse. Dabei beachtet er die Veränderungen in den Schrittdimensionen und berechnet auf diesem Weg eine komplette Ellipse.[10]

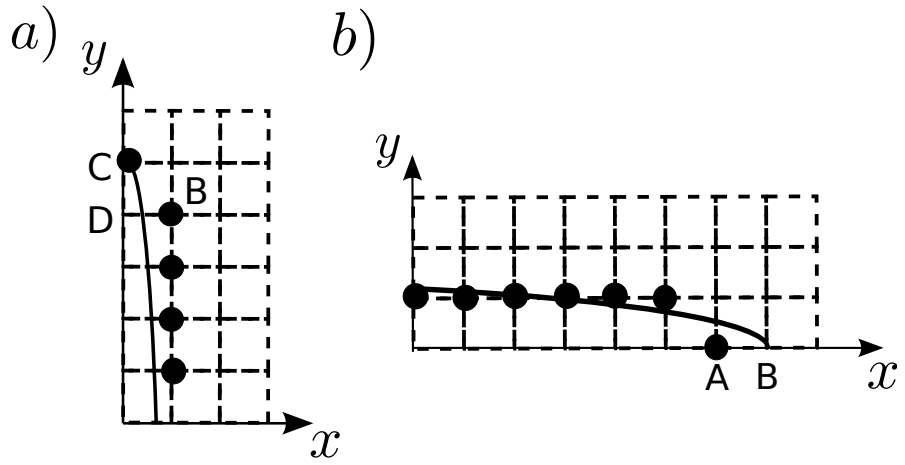

Abbildung 3.8: Probleme bei den Grenzfällen der Ellipse. *Teilbild a zeigt einen falschen Oktantenwechsel vom zweiten in den ersten Oktant. Hier sollte für die beste Annäherung nach Koordinate C die Koordinate D gewählt werden. Stattdessen findet ein Oktantenwechsel statt und es wird Koordinate B gewählt. Teilbild b zeigt eine falsche Rasterung mit Auslassen der Koordinate B im Endbereich der Ellipse.*[7]

3.4 Kritik an den Algorithmen

An den Algorithmen von J. van Aken [13, 14] und M. Pitteway [10] wurde an mehreren Stellen Kritik geübt.[12, 15, 19] M. Sarfraz zeigt in [19] zwei Fälle auf, in denen die beiden Algorithmen versagen, siehe Abbildung 3.8.

Abbildung 3.8a zeigt einen vorzeitigen Oktantenwechsel, obwohl ausgehend von Koordinate C die Koordinate D eine bessere Annäherung bietet. Fall b zeigt eine fehlerhafte Rasterung des Endbereiches einer Ellipse. Hier wird die Koordinate B ausgelassen, obwohl die Ellipse nicht vollständig ist.

A. Janser und W. Luther korrigieren die fehlenden Rasterpunkte aus Abbildung 3.8b mit zusätzlichen Linien bis zum Schnittpunkt mit der entsprechenden Achse.[12]

Unabhängig vom Typ der Ellipse wurden von M. McIlroy in [15] Probleme aufgezeigt, die beim Strecken einer Ellipse auftreten. Oft versagen Algorithmen in diesen Bereichen, indem sie den schmalen Bereich der Ellipse falsch berechnen, wie in Abbildung 3.9 angedeutet.

[7] Entnommen aus: M. Sarfraz, S. 3

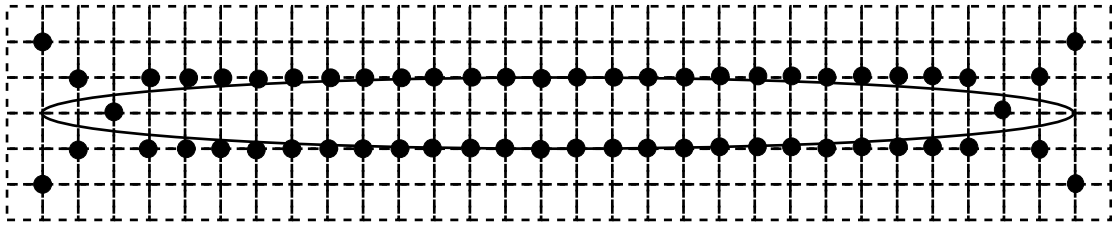

Abbildung 3.9: Falsch gerasterte gestreckte Ellipse. *Bei stark gestreckten Ellipsen, z.B. $r_y = 1$, $r_x = 15$, versagen oft Algorithmen und rastern die Ellipse in den Kanten falsch. Die Abbruchbedingungen der Oktanten greifen nicht und es entstehen die markanten Rasterungslinien, welche von der Ellipse wegführen. Weiterhin wird die Koordinate der Schwanzspitze nie erreicht.*[8]

Weiterhin bemängelt er die getrennte Betrachtung der Radien verschiedener Dimensionen bei der Schrittwahl. Er selber stellt einen Algorithmus über die Freeman-Approximation auf. Mit ihr erreicht er eine Beachtung der Verschiebungen bei der Schrittwahl in beiden Dimensionen und erzielt damit eine Behandlung der Grenzfälle durch Streckung.

D. Fellner und C. Helmberg kritisieren jedoch ebenfalls M. McIlroys Variante und veröffentlichten 1993 mit [20] einen neuen Ansatz, welcher zwei Kreise überlagert. Damit erreichen sie einen robusten Ansatz gegenüber Streckungen der Ellipse. Weiterhin betrachten sie Eingaben von Radien unterhalb einer Pixellänge, was in allen anderen Veröffentlichungen keine Rolle gespielt hat.

3.5 Zusammenfassung

In der Literatur sind vielfältige funktionierende und beschriebene Algorithmen vorhanden. Das Bresenham-Verfahren ist von einer Geraden im zweidimensionalen auf Kreise und Ellipsen im zwei-dimensionalen und Geraden im drei-dimensionalen erweiterbar. Weitere Rasterungen im drei-dimensionalen Raum wurden nicht gefunden. Allerdings gelten alle veröffentlichten Algorithmen für ganzzahlige Koordinaten in Start- und Endpunkten, sowie bei Mittelpunkten und Radien. Eine Gleitkomma-Lösung existiert nicht. Weiterhin gibt es verschiedene Ansätze die Schrittwahl zu berechnen, den Zweipunkt- und den Mittelpunktalgorithmus. Im Vergleich der beiden zeigt sich ein gleichwertiger Rechenaufwand. Bei der Pixelwahl allerdings wurden von J. van Aken Unterschiede festgestellt. Diese tau-

[8] Entnommen aus: M.D. McIlroy (1982), S. 267

chen bei stark gestreckten Ellipsen auf. Der Mittelpunktalgorithmus arbeitet hier genauer, wenn man den orthogonalen Abstand der gewählten Pixelkoordinate zur wahren Linie betrachtet. Beispielsweise liefert der Zweipunktalgorithmus bei $r_x = 4$ und $r_y = 1$ eine maximale Abweichung von $0,632$ Pixel. Der Mittelpunktalgorithmus liefert einen Fehler von $0,326$ Pixel.[13]

Mit einer Anpassung des Verfahren nach J. Bresenham an Gleitkomma-Parameter ist das Verfahren für das USCT Projekt nutzbar. Dazu müssen allerdings Rasterungen von Ellipsoiden im drei-dimensionalen Raum ermöglicht werden. Da die konzeptuelle Betrachtung in der Literatur nur bis zu einer drei-dimensionalen Geraden geht, muss hier ein neues Konzept entworfen werden.

4 Design

4.1 Zielsetzung

Nachdem das Bresenham-Verfahren nachweislich für einen Teil der vom USCT geforderten geometrischen Körper umsetzbar ist, ist es das Ziel dieser Arbeit das Verfahren für das Projekt anzupassen.
Es soll als Proof of Concept bewiesen werden, dass das Verfahren nach J. Bresenham auch auf Gleitkommaeingaben anwendbar ist. Weiterhin sollen geometrische Körper über eine drei-dimensionale Gerade hinaus erzeugt werden können. Im Laufe der Arbeit sollen Geraden im zwei-dimensionalen und drei-dimensionalen Raum, Kreise, Kugeln, Ellipsen und Ellipsoide berechenbar sein.
Die verschiedenen Algorithmen sollen auf ihre Performance in Voxel/sec und Rasterungsgenauigkeit evaluiert werden.
Ist das Rastern eines allgemeinen Ellipsoid gelungen, soll dieser in die Bildrekonstruktion integriert werden. Anschließend soll das neue Verfahren der Bildgenerierung mit dem aktuellen Verfahren SAFT bezogen auf die Laufzeit verglichen werden.

Im Folgenden werden Konzepte aufgezeigt, um die bestehenden Algorithmen an die Bedingungen des USCT anzupassen. Dabei wird ein schrittweises Vorgehen für die geometrischen Körper durchgeführt. Von einer Geraden im zwei-dimensionalen ausgehend wird das Verfahren auf drei-dimensionalität portiert. Es werden Kreise berechnet, welche anschließend zu Kugeln modifiziert werden. Anschließend wird auf eine Ellipse eingegangen. Dieses Vorgehen wurde gewählt, da die Prinzipien aufeinander aufbauen und die Komplexität mit jedem Schritt steigt. So kann ein guter Überblick gegeben werden.

4.2 Konzept für Gerade

Um den Bresenham-Algorithmus für zweidimensionale Geraden an Gleitkommaeingaben anzupassen, muss vorher die Herleitung der verwendeten Formeln und Ausdrücke geklärt werden.

4.2.1 Schrittwahl

In Abbildung 3.4 ist die Entscheidung über die Schrittwahl detailliert aufgeführt. Es werden immer die Längen r_{n+1} und q_{n+1} beachtet, die den Abstand von (x_{n+1}, y_n) beziehungsweise $(x_{n+1}, y_n + 1)$ zu (x_{n+1}, y) angeben. Diese Dreiecksbeziehungen sind in den Formeln des Algorithmus verarbeitet. Die folgenden Herleitungen sind an [21] angelehnt und verwenden den Zweipunktalgorithmus. Diese Wahl wurde getroffen, da in den Formeln des Mittelpunktalgorithmus Bruchrechnungen getätigt werden müssen, bedingt durch die Koordinaten des Mittelpunktes zwischen den beiden möglichen wählbaren Rasterkoordinaten.

Mit Einsetzen von $m = \frac{Dy}{Dx} = \frac{y_{end} - y_{start}}{x_{end} - x_{start}}$ in die Geradengleichung $y = m \cdot x + u$ ergeben sich mit $y = r_{n+1}$ für r_{n+1} und q_{n+1} die Gleichungen 4.1 und 4.2. Dabei wird $u = 0$ gesetzt, da die Verschiebung entlang der y-Achse später durch die Angabe des Startpunktes getätigt wird.

$$\begin{aligned} y &= \frac{Dy}{Dx} \cdot x \\ r_{n+1} &= \frac{Dy}{Dx} \cdot (x_{n+1} - x_n) \end{aligned} \tag{4.1}$$

$$\begin{aligned} q_{n+1} &= 1 - r_{n+1} \\ q_{n+1} &= 1 - \frac{Dy}{Dx} \cdot (x_{n+1} - x_n) \end{aligned} \tag{4.2}$$

Um nun entscheiden zu können, ob (x_{n+1}, y_n) oder (x_{n+1}, y_{n+1}) als nächstes angesprochen wird, subtrahiert man r_{n+1} mit q_{n+1} und betrachtet das Vorzeichen. Bei $r_{n+1} - q_{n+1} < 0$ vollzieht man einen x-Schritt, bei $r_{n+1} - q_{n+1} \geq 0$ einen xy-Schritt. Die Formel für solch eine Subtraktion ist in Gleichung 4.3 aufgeführt. Sie gibt die Formel für einen konkreten Fehlerwert $error_n$ an. Zu beachten ist hierbei, dass $x_{n+1} - x_n = 1$ gesetzt wird, da die Formel sich auf einen x-Schritt bezieht und hier immer $x_{n+1} = x_n + 1$ gilt.

$$
\begin{aligned}
error_n = r_n - q_n &= \frac{Dy}{Dx} \cdot (x_{n+1} - x_n) - \left(1 - \frac{Dy}{Dx} \cdot (x_{n+1} - x_n)\right) \\
error_n = r_n - q_n &= 2\frac{Dy}{Dx} \cdot 1 - 1 \\
error_n = r_n - q_n &= \frac{2Dy - Dx}{Dx} \\
error_n = Dx \cdot (r_n - q_n) &= 2Dy - Dx
\end{aligned} \tag{4.3}
$$

Für die Startbedingung bei $x_n = 0$ und $y_n = 0$ für einen Startpunkt bei $P_{start}\,(0, 0)$ ergibt sich hieraus die Formel für den Startfehlerwert, siehe Formel 4.4.

$$error_0 = 2Dy - Dx \tag{4.4}$$

Um nun die Fehlerveränderung für beide Schritte zu berechnen, muss die allgemeine Formel für $error_{n+1}$ aufgestellt und dann die betroffenen Werte für die beiden Schrittmöglichkeiten eingesetzt werden. Hierbei ist zu beachten, dass im ersten Oktant die x-Dimension bei jedem Schritt manipuliert wird und eine Pixelweite den Wert 1 besitzt. Für die Formel bedeutet dies, dass $x_{n+1} - x_n = 1$ gilt. Die aufgestellte Formel für einen allgemeinen Schritt ist in der Gleichung 4.5 aufgezeigt.

$$
\begin{aligned}
error_{n+1} - error_n &= 2Dy \cdot (x_{n+1} - x_n) - 2Dx \cdot (y_{n+1} - y_n) \\
error_{n+1} &= error_n + 2Dy \cdot 1 - 2Dx \cdot (y_{n+1} - y_n)
\end{aligned} \tag{4.5}
$$

Soll nun die Fehlerveränderung für einen x-Schritt berechnet werden, so gilt $y_n = y_{n+1}$ und damit $y_{n+1} - y_n = 0$.

$$
\begin{aligned}
error_{n+1} &= error_n + 2Dy - Dx \cdot (y_{n+1} - y_n) \\
error_{n+1} &= error_n + 2Dy \\
error_x &= 2Dy
\end{aligned} \tag{4.6}
$$

Für eine Fehlerveränderung bei einem xy-Schritt gilt $y_{n+1} - y_n = 1$, bedingt durch die Pixelgröße 1.

$$
\begin{aligned}
error_{n+1} &= error_n + 2Dy - 2Dx \cdot (y_{n+1} - y_n) \\
error_{n+1} &= error_n + 2\,(Dy - Dx) \\
error_{xy} &= 2\,(Dy - Dx)
\end{aligned} \tag{4.7}
$$

Mit diesen Formeln lässt sich der Bresenham-Algorithmus für ganzzahlige Start- und Endpunkte ausführen. Allerdings funktioniert es momentan nur für Geraden

im ersten Oktant. Um auch das restliche Koordinatensystem abzudecken, muss der Algorithmus weiter angepasst werden.

4.2.2 Ausweitung auf alle Oktanten

Um den Algorithmus für alle acht Oktanten tauglich zu machen, bedarf es einer Vorbetrachtung der Laufrichtungen, welche aus [16] entnommen und im Anhang A aufgezeigt ist. Zuerst wird Dx geprüft und eine Variable $incX$ angelegt, welche die Laufrichtung für x abspeichert. Das selbe Verfahren wird danach für Dy durchgeführt. Anschließend werden je nach Steigung der Geraden die Koordinatenveränderungen $parallelX$ und $parallelY$ angelegt, welche später für einen Schritt in nur einer Dimension genutzt werden. Ebenfalls wird die Fehlerveränderung für beide Schrittformen deklariert, $errorParallel$ und $errorDiagonal$. Die Auftrennung dieser beiden Fehlerwerte ist bedingt durch die Definition der Fehlerwerte. So gilt allgemein für alle Oktanten die Formel 4.8 als Startwert.

$$error_0 \quad = \quad 2 \cdot errorDiagonal - errorParallel \tag{4.8}$$

Ebenso gilt allgemein gesehen für einen parallelen Schritt die Formel 4.9 und für einen diagonalen die Formel 4.10.

$$error_x = error_{parallel} = 2 \cdot errorDiagonal \tag{4.9}$$

$$error_{xy} = error_{diagonal} = 2 \cdot (errorDiagonal - errorParallel) \tag{4.10}$$

Zuletzt werden die Koordinatenveränderungen für einen Schritt in beiden Dimensionen gesetzt, $diagonalX$ und $diagonalY$.

Durch Anpassung der verwendeten Variablen ist der Algorithmus nun fähig, alle Geraden in einem zweidimensionalen Koordinatensystem zu berechnen. Allerdings besteht weiterhin die Voraussetzung, dass die Koordinaten ganzzahlige Werte sind.

4.2.3 Abbruchbedingung

Ein Abbruch der Schrittberechung findet statt, wenn die Dynamik in der schnellen Dimension durch die Schrittzahl erreicht wurde. Dazu wird nur die aktuell getätigte Schrittzahl benötigt und mit der schnellen Dynamik verglichen werden. Dabei darf der Startpunkt nicht vergessen werden, welcher in die Schrittzahl mit hineinfließt.

4.2.4 Erweiterung für drei-dimensionalen Raum

Im Internet kursieren diverse Funktionen, die die Bresenhamgerade auf den dreidimensionalen Raum portieren. Die Methodik dahinter ist folgende: Zuerst wird wie im zweidimensionalen Raum die Schrittveränderungen für x, y und z nach positiver oder negativer Dynamik gesetzt. Anschließend wird betrachtet, welche Dynamik die größte ist. Jede Dimension erhält hierbei einen eigenen Bereich. Über zwei getrennte Fehlervariablen werden die Schritte in den beiden langsameren Dimensionen abgefragt, sie Formel 4.11 und 4.12.

$$error1 = 2 \cdot DynamikLangsam1 - DynamikSchnell \tag{4.11}$$

$$error2 = 2 \cdot DynamikLangsam2 - DynamikSchnell \tag{4.12}$$

Die Fehleranpassung wird damit aufgetrennt. Muss ein langsamer Schritt durch $errorN > 0$ getätigt werden, wird der Wert eines langsamen Schrittes von der entsprechenden Fehlervariable nach Formel 4.13 abgezogen.

$$errorN = errorN - 2 \cdot DynamikSchnell \tag{4.13}$$

Anschließend wird auf beide Fehlervariablen die Veränderung für einen Schritt in der schnellen Dimension nach der Gleichung 4.14 aufaddiert.

$$errorN = errorN + 2 \cdot DynamikLangsamN \tag{4.14}$$

Hierbei wird der Schritt durch Anpassung der Koordinaten durchgeführt. Ein solcher Algorithmus für MATLAB ist im Anhang C aufgeführt.[22]

4.2.5 Gleitkommaparameter

An dieser Stelle wird der Bresenham-Algorithmus für Geraden nicht weiter ausgebaut. Die Gleitkommavariante wird in dieser Arbeit nicht entwickelt. Vielmehr wird zum Kreis übergegangen. Dieser Schritt wird über die Tatsache begründet, dass das Ziel dieser Arbeit ein Algorithmus für einen Ellipsoid ist. Die Komplexität des Rasterungsproblems ist bei einem Kreis größer, da die Steigung nicht konstant ist, sondern variiert. Dies ist demnach ein sinnvoller Schritt der Rasterung von Ellipsen näher zu kommen

4.3 Konzept für Kreis

Um das Bresenham-Verfahren von Geraden auf Kreise zu portieren, gilt es die Eigenschaften eines Kreises zu beachten und daraus das Verhalten des Bresenham-Verfahren abzuleiten.

4.3.1 Eigenschaften eines Kreises

Ein Kreis ist ein geometrisches Objekt im zweidimensionalen Raum. Er ist definiert über einen Mittelpunkt $P_m(x_m, y_m)$, von dem im Winkel von $0 - 360°$ der Abstand zu einer Koordinate auf dem Kreis immer genau dem Radius r entspricht, siehe Abbildung 4.1a. Ein Kreis ist vollkommen achsensymmetrisch und kann parallel der x- und y-Achse gespiegelt werden. Der Verlauf der Steigung m_t der Tangente t[1] ist in Abbildung 4.1b dargestellt. Nach ihr kann man einen Kreis in acht Teilbereiche, Oktanten, einteilen. Die Eigenschaft der Koordinatensymmetrie ist in der Abbildung 4.1c dargestellt. Alle Oktanten lassen sich über die entsprechenden Koordinatentransformationen in die anderen Oktanten spiegeln.

Soll ein kompletter Kreis berechnet werden, müssen die acht Teilbereiche des Kreises getrennt betrachtet werden. Das hängt mit den Eigenschaften des Bresenham-Verfahrens zusammen, da immer eine schnelle und langsame Laufrichtung benötigt wird. Im Verlauf eines Kreises wechseln sich diese Laufrichtungen in ihrem Status ab. Durch die Symmetrieeigenschaft des Kreises ist es möglich, einen Algorithmus zu entwerfen, der einen Oktanten des Kreises berechnet. Anschließend kann dieser

[1] Eine Tangente ist eine Gerade im Winkel von 90° zum Berührungspunkt mit dem Kreis und gleicher Richtung an diesem Punkt.

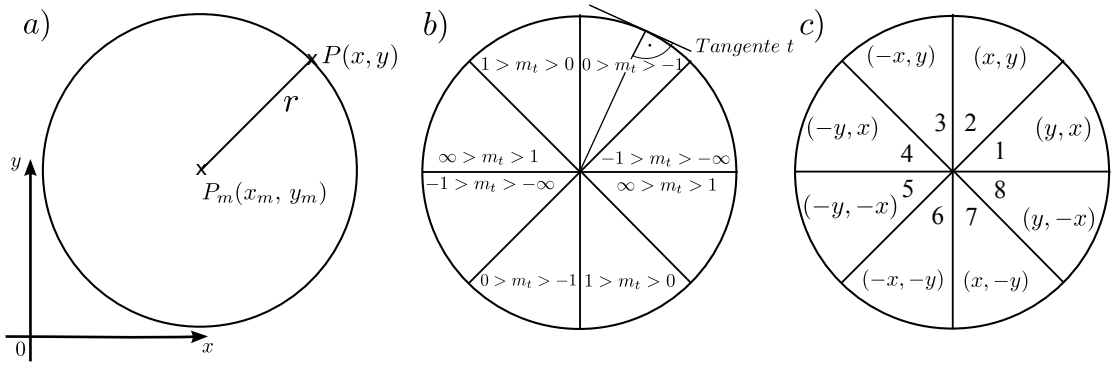

Abbildung 4.1: Eigenschaften eines Kreises und anlegbarer Tangente. *Teilbild a zeigt die Beziehung von Mittelpunkt P_m und Radius r um Punkt P zu erzeugen. Teilbild b zeigt den Verlauf der Steigung einer Tangente, welche am Kreis angelegt wird. Teilbild c zeigt die daraus resultierende Aufteilung des Kreises in Oktanten mit der Nummerierung und den entsprechenden Koordinatentransformationen zum Spiegeln.*

siebenfach gespiegelt werden, um die Koordinaten für den kompletten Kreis zu erhalten. Für den folgenden Entwurf wurde der zweite Oktant gewählt. Er beginnt am Schnittpunkt mit der y-Achse im Punkt $P_y\,(0,\, r)$. x gilt als schnelle Laufrichtung, y als langsame. Die mögliche Schrittwahl ist $(x_n + 1,\, y_n)$ für einen x-Schritt oder $(x_n + 1,\, y_n - 1)$ für einen xy-Schritt, siehe Abbildung 4.2.

4.3.2 Schrittwahl

Für die Kurvenrasterung existieren zwei mathematische Ansätze zur Schrittwahl. Der Zweipunkt- und der Mittelpunktalgorithmus. Die Herleitungen der jeweils zu verwendenden Formeln soll im Folgenden augezeigt werden. Es werden beide Algorithmen ausführlich hergeleitet, da die algorithmische Wahl in der späteren Umsetzung im Vorfeld nicht getroffen werden kann.

4.3.2.1 Mittelpunktalgorithmus

Der Mittelpunktalgorithmus erzeugt bei Kreisen identische Ergebnisse. Jedoch zeigt sich bei Ellipsen die Stärke des Algorithmus. Hier wird präziser gerastert, siehe [13]. Zur Erläuterung der mathematischen Herleitungen soll der Folgende Abschnitt dienen.

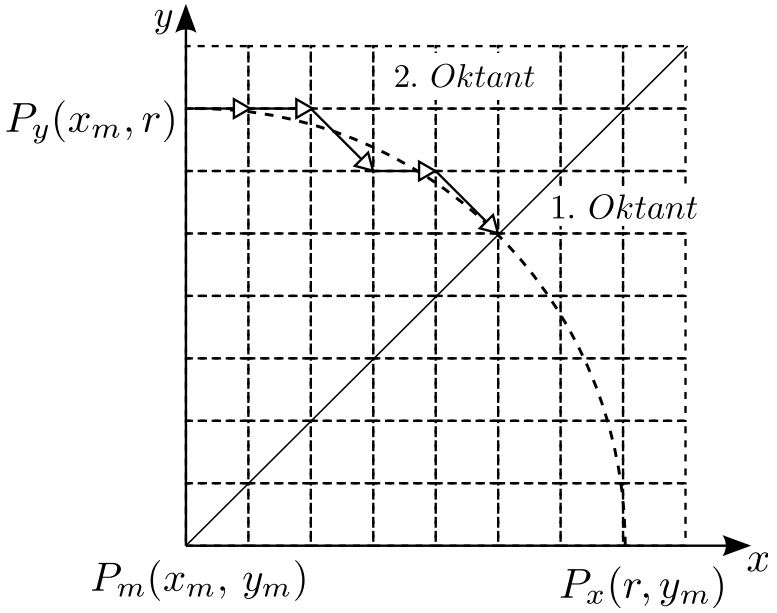

Abbildung 4.2: Mögliche Schrittwahl eines Bresenham-Algorithmus für einen Kreis im zwei-dimensionalen Raum. *Mit dem Start an P_y im zweiten Oktant läuft die Schrittwahl bis $x = y$ und damit bis zur Oktantengrenze. Die restlichen Koordinaten werden über Spiegelung erreicht.*

Bei mathematischer Betrachtung wird ebenfalls mit der umgestellten Kreisgleichung aus der Formel 4.19 gearbeitet. Als einzusetzende Koordinate gilt allerdings die des Mittelpunktes zwischen den möglichen Koordinaten.

$$error_n = (x_n + 1)^2 + \left(y_n - \frac{1}{2}\right)^2 - r^2 \tag{4.15}$$

Der Startwert der Fehlervariable für den zweiten Oktant berechnet sich durch einsetzen von $x_n = 0$ und $y_n = r$ und ist in Formel 4.16 aufgezeigt.

$$\begin{aligned} error_0 &= 1 + r^2 - r + \frac{1}{4} - r^2 \\ error_0 &= \frac{5}{4} - r \end{aligned} \tag{4.16}$$

Für die Fehlerveränderung pro Schritt verwendet man auch hier die Vorgehensweise $error_{n+1}$ mit $error_n$ zu subtrahieren. Für einen x-Schritt mit $x_{n+1} = x_n + 1$ und $y_{n+1} = y_n$ ergibt sich daraus Formel 4.17.

$$\begin{aligned} error_{n+1} - error_n &= (x_{n+1}+1)^2 + \left(y_n - \frac{1}{2}\right)^2 - r^2 \\ &\quad - \left((x_n+1)^2 + \left(y_n - \frac{1}{2}\right)^2 - r^2\right) \\ &= (x_n+2)^2 - (x_n+1)^2 \\ &= x_n^2 + 4x_n + 4 - x_n^2 - 2x_n - 1 \\ &= 2x_n + 3 \\ error_x &= 2x_n + 3 \end{aligned} \tag{4.17}$$

Für einen xy-Schritt ergibt sich mit $x_{n+1} = x_n + 1$ und $y_{n+1} = y_n - 1$ die Formel 4.18.

$$\begin{aligned} error_{n+1} - error_n &= (x_{n+1}+1)^2 + \left(y_{n+1} - \frac{1}{2}\right)^2 - r^2 \\ &\quad - \left((x_n+1)^2 + \left(y_n - \frac{1}{2}\right)^2 - r^2\right) \\ &= (x_n+2)^2 + \left(y_{n+1} - \frac{3}{2}\right)^2 - (x_n+1)^2 - \left(y_n - \frac{1}{2}\right)^2 \\ &= x_n^2 + 4x_n + 4 + y_n^2 - 3y_n + \frac{9}{4} - r^2 \\ &\quad - x_n^2 - 2x_n - 1 - y_n^2 + y_n - \frac{1}{4} + r^2 \\ &= 2x_n - 2y_n + 5 \\ error_{xy} &= 2(x_n - y_n) + 5 \end{aligned} \tag{4.18}$$

Berechnet man diese Formeln für alle Oktanten, so ergeben sich die Fehlerveränderungen aus Tabelle 4.1. Diese Formeln können verwendet werden, um jeden Oktanten einzeln zu berechnen oder einen beliebigen auszuwählen, um ihn anschließend zu spiegeln.

Oktant	Parallelschritt	Diagonalschritt
1	$2y_n + 3$	$2(-x_n + y_n) + 5$
2	$2x_n + 3$	$2(x_n - y_n) + 5$
3	$-2x_n + 3$	$2(-x_n - y_n) + 5$
4	$2y_n + 3$	$2(x_n + y_n) + 5$
5	$-2y_n + 3$	$2(x_n - y_n) + 5$
6	$-2x_n + 3$	$2(-x_n + y_n) + 5$
7	$2x_n + 3$	$2(x_n + y_n) + 5$
8	$-2y_n + 3$	$2(-x_n - y_n) + 5$

Tabelle 4.1: Fehlerveränderungen für alle Kreisoktanten im Mittelpunktalgorithmus.

4.3.2.2 Zweipunktalgorithmus

Die folgenden Herleitungen sind an [23] angelehnt.
Wie bei der Geraden auch, muss die Fehlervariable mit einem Startwert initialisiert werden. Hierzu muss zuerst die Fehlerformel für einen Fehler $error_n$ aufgestellt werden. Aus der Kreisgleichung kann durch Umstellen nach 0 die Fehlervariable für eine Dimension berechnet werden, siehe Formel 4.19.

$$\begin{aligned} r^2 &= x^2 + y^2 \\ error &= x^2 + y^2 - r^2 \end{aligned} \tag{4.19}$$

Für die Schrittvarianten x und xy ergeben sich mit der Gleichung 4.19 die beiden folgenden Formeln 4.20 und 4.21.

$$error_x = (x_n + 1)^2 + y_n^2 - r^2 \tag{4.20}$$

$$error_{xy} = (x_n + 1)^2 + (y_n - 1)^2 - r^2 \tag{4.21}$$

Addiert man diese beiden Schrittfehler aus den Formeln 4.20 und 4.21, so erhält man den allgemeinen Fehler $error_n$ mit der Formel 4.22.

$$\begin{aligned} error_n &= error_x + error_{xy} \\ &= (x_n + 1)^2 + y_n^2 - r^2 + (x_n + 1)^2 + (y_n - 1)^2 - r^2 \\ &= 2(x_n + 1)^2 + y_n^2 + (y_n - 1)^2 - 2r^2 \end{aligned} \tag{4.22}$$

Mit der Gleichung 4.22 kann nun der Startwert der Fehlervariable für $error_0$ berechnet werden. Die Startbedingungen sind im zweiten Oktant $x_n = 0$ und $y_n = r$, bei einer Mittelpunktposition von $P_m\,(0,\,0)$. Diese Werte variieren bei Verschiebung des Mittelpunktes auf $x_n = 0 + x_m$ und $y_n = r + y_m$. Eingesetzt ergibt sich die Formel 4.23.

$$\begin{aligned} error_0 &= 2\,(0+1)^2 + r^2 + (r-1)^2 - 2r^2 \\ &= 2 - r^2 + r^2 - 2r + 1 \\ &= 3 - 2r \end{aligned} \tag{4.23}$$

Um den allgemeinen Schrittfehler berechnen zu können muss $error_{n+1}$ mit $error_n$ subtrahiert werden, siehe Gleichung 4.24.

$$\begin{aligned} error_{n+1} - error_n &= \left(2\,(x_{n+1}+1)^2 + y_{n+1}^2 + (y_{n+1}-1)^2 - 2r^2\right) \\ &\quad - \left(2\,(x_n+1)^2 + y_n^2 + (y_n-1)^2 - 2r^2\right) \\ &= \left(2x_{n+1}^2 + 4x_{n+1} + 2 + y_{n+1}^2 + y_{n+1}^2 - 2y_{n+1} + 1 - 2r^2\right) \\ &\quad - \left(2x_n^2 + 4x_n + 2 + y_n^2 + y_n^2 - 2y_n + 1 - 2r^2\right) \\ &= 2x_{n+1}^2 + 4x_{n+1} + 2y_{n+1}^2 - 2y_{n+1} \\ &\quad -2x_n^2 - 4x_n - 2y_n^2 + 2y_n \end{aligned} \tag{4.24}$$

Setzt man nun in die Formel 4.24 die jeweiligen Schritte ein, ergeben sich die Fehlerveränderungen für die einzelnen Schritte. Für einen x-Schritt mit $x_{n+1} = x_n + 1$ und $y_{n+1} = y_n$ ergibt sich für $error_x$ die Formel 4.25.

$$\begin{aligned} error_{n+1} - error_n &= 2\,(x_n+1)^2 + 4\,(x_n+1) + 2y_n^2 \\ &\quad -2y_n - 2x_n^2 - 4x_n - 2y_n^2 + 2y_n \\ &= 2x_n^2 + 4x_n + 2 + 4x_n + 4 - 2x_n^2 - 4x_n \\ error_x &= 4x_n + 6 \end{aligned} \tag{4.25}$$

Für $error_{xy}$ ergibt sich mit $x_{n+1} = x_n + 1$ und $y_{n+1} = y_n - 1$ Formel 4.26.

$$
\begin{aligned}
error_{n+1} - error_n &= 2\left(x_n+1\right)^2 + 4\left(x_n+1\right) + 2\left(y_n-1\right)^2 \\
&\quad -2\left(y_n-1\right) - 2x_n^2 - 4x_n + 2y_n^2 - 2y_n \\
&= 2x_n^2 + 4x_n + 2 + 4x_n + 4 + 2y_n^2 - 4y_n + 2 \\
&\quad -2y_n + 2 - 2x_n^2 - 4x_n - 2y_n^2 + 2y_n \\
&= 4x_n - 4y_n + 10 \\
error_{xy} &= 4(x_n - y_n) + 10 \qquad (4.26)
\end{aligned}
$$

Berechnet man diese Formeln für alle Oktanten, so ergeben sich die Fehlerveränderungen aus Tabelle 4.2. Diese Formeln können verwendet werden, um jeden Oktanten einzeln zu berechnen oder einen beliebigen auszuwählen, um ihn anschließend zu spiegeln.

Oktant	**Parallelschritt**	**Diagonalschritt**
1	$4y_n + 6$	$4\left(-x_n + y_n\right) + 10$
2	$4x_n + 6$	$4\left(x_n - y_n\right) + 10$
3	$-4x_n + 6$	$4\left(-x_n - y_n\right) + 10$
4	$4y_n + 6$	$4\left(x_n + y_n\right) + 10$
5	$-4y_n + 6$	$4\left(x_n - y_n\right) + 10$
6	$-4x_n + 6$	$4\left(-x_n + y_n\right) + 10$
7	$4x_n + 6$	$4\left(x_n + y_n\right) + 10$
8	$-4y_n + 6$	$4\left(-x_n - y_n\right) + 10$

Tabelle 4.2: Fehlerveränderungen für alle Kreisoktanten im Zweipunktalgorithmus.

In der späteren Umsetzung wurde Zweipunktalgorithmus gewählt. Diese Wahl wird über die equivalenten Rasterergebnisse für Kreise, siehe [13], und das Abhandensein von rationalen Zahlen in den Formeln begründet.

4.3.3 Gleitkommaparameter

Mit Eingabe von Gleitkommawerten für die Parameter des Kreises ergeben sich Verschiebungen unterhalb einer Pixelgröße zusätzlich zu den ganzzahligen Verschiebungen. Das bedeutet eine Verschiebung des Mittelpunktes P_m in x und y. Die beiden Verschiebungen sind hier als $disloc_x$ und $disloc_y$ bezeichnet, siehe Abbildung 4.3. Der dadurch entstehende Abstand zum nächsten ganzzahligen Wert entspricht den Variablen $stepsize_x$ und $stepsize_y$. Ganzzahlige Verschiebungen werden im Nachhinein durch eine zusätzliche Addition durchgeführt.

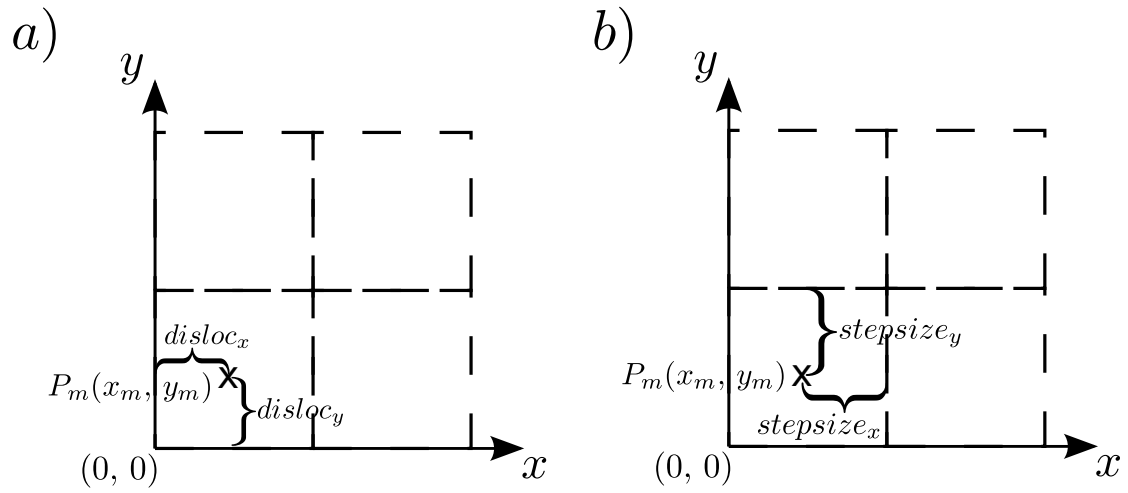

Abbildung 4.3: Verschiebung durch Gleitkommaeingaben mit Variablenbenennung im zwei-dimensionalen Raum. *Durch die Gleitkommaverschiebung ergeben sich neue Werte. So ist die Verschiebung an sich hier als $disloc_x$ und $disloc_y$ angegeben. $stepsize_x$ und $stepsize_y$ geben den Abstand zur jeweils nächsten ganzzahligen Dimension an.*

Damit bleiben die Koordinaten, mit denen gerechnet wird, so klein wie möglich, da sie um den Nullpunkt liegen. Mit der Verschiebung werden die Koordinaten des Startpunktes im zweiten Oktant durch $x_0 = 0 + disloc_x$ und $y_0 = r + disloc_y$ ausgedrückt.

Mit Parametern im Gleitkomma-Bereich ist das Bresenham-Verfahren in der Hauptverarbeitung weiterhin anwendbar. Allerdings muss die Betrachtung der Start- und Endpunkte angepasst werden.

4.3.4 Startpunktwahl

Im Folgenden werden verschiedene Ansätze zur Wahl des Startpunktes für die Berechnung der Schritte aufgeführt. Dabei wird auf die Vorgehensweise sowie Stärken und Schwächen eingegangen.

4.3.4.1 Runden auf Ganzzahl

Das Bresenham-Verfahren arbeitet mit einer Fehlervariable. Diese gibt beim Kreis den aufsummierten quadrierten Abstand zum Kreis für die aktuelle Rasterposition an. Sie gilt als Entscheidungsvariable für die möglichen Schritte. Mit einer Verschiebung im Gleitkomma-Bereich wird ein Fehler entgegen dem Pixel-

raster getätigt. Ein einfacher Ansatz diesen Fehler zu kompensieren ist es, die Fehlervariable bei der Initialisierung entsprechend der Verschiebung anzupassen. Dazu wird die Verschiebung quadriert auf den Startfehlerwert aufaddiert, $error_0 + disloc_x^2 + disloc_y^2$. Negative Verschiebungen müssen nach der Quadrierung wieder in den negativen Zahlenbereich transformiert werden. Als Koordinaten für den Startwert im zweiten Oktant können $round(x_0)$ und $round(y_0)$ verwendet werden. Um entscheiden zu können, mit welchem ganzzahligen Radius gerechnet werden soll, wird der Initialfehler mit auf- und abgerundetem Radius berechnet, und über den kleineren Fehlerwert der entsprechende Radius gewählt.

Dieser Ansatz führt zu diversen Problemen. Das gezwungene Umbrechen der Gleitkommawerte des Startpunktes und Radius auf Ganzzahlen erzeugt zusätzliche Fehler, welche nicht kompensiert werden. Diese verändern die Schrittwahl und behindern die optimale Annäherung. Weiterhin wird durch das Runden des Radius ein Kreis berechnet, welcher vom gewollten Kreis in den Ausmaßen abweicht.

4.3.4.2 Rechnen in x-Dimension

Ein neuer Ansatz verwendet die Gleitkommawerte, um die Berechnungen durchzuführen. Es wird wieder nur der zweite Oktant berechnet, anschließend gespiegelt und im ganzzahligen Bereich verschoben. Dabei wird für die Startkoordinate der y-Wert $y_0 = (disloc_y + r)$ generell aufgerundet. Anschließend wird über die allgemeine Fehlerformel des Kreises $error_0 = 2x_0^2 - 2r^2 + y_0^2 - (y_0 - 1)^2$ ein Fehlerwert für diese Koordinate berechnet. Für x_0 wird $disloc_x$ verwendet, da dies die wahre Position in x angibt und für den y-Wert passend ist. Wenn nun $error > 0$ annimmt, wird ein Schritt in der y-dimension über $y_0 = y_0 - 1$ getätigt. Die Fehlervariable wird hierbei so angepasst, als wäre ein Schritt in y getätigt worden. Um die Fehlerveränderung für den Zweipunktalgorithmus in diesem Fall zu berechnen, nutzt man die Formel 4.24 und setzt $x_{n+1} = x_n$ und $y_n = y_0$. y_{n+1} entspricht dementsprechend $y_0 - 1$. Daraus ergibt sich eine Fehlerveränderung für den y-Schritt mit dem Wert $-4y_n + 4$. Dieser Wert wird auf den Fehler $error_0$ aufaddiert. Wenn $error_0 = 0$ gilt, die y-Koordinate also einen optimalen Start ermöglicht, wird die Fehlervariable mit dem Standardwert $error_0 = 3 - 2r$ neu initialisiert. Der Startpunkt für die Berechnung ist dementsprechend ($disloc_x$, y_n). Hierbei ist die x-Koordinate weiterhin eine Gleitkommazahl, weswegen hier aufgerundet wird. Die Fehleranpassungen während der Schrittwahl werden ebenfalls mit dem Gleitkommawert $disloc_x$ für x berechnet. Allerdings muss $disloc_x$ in je-

dem Schleifendurchlauf inkrementiert werden, wie es das Bresenham-Verfahren vorsieht.

Dieser Ansatz erhöht die Genauigkeit der Rasterung des Kreises. Allerdings entsteht weiterhin ein Kreis unterschiedlich in den Ausmaßen zum gewollten Kreis. Das liegt daran, dass der gerasterte y-Wert nicht zum verwendeten x-Wert passt. Weiterhin wird die Verschiebung während der Schrittwahl nur in einer Dimension beachtet. Beim Spiegeln des Oktant kann es zu Lücken kommen, bedingt durch das anfängliche Aufrunden des x-Startwertes.

4.3.4.3 Beachten der Verschiebung in beiden Dimensionen

Betrachtet man die Verschiebung des Mittelpunktes genauer, so ergibt sich für den zu rasternden Kreis ein neues gültiges Unterkoordinatensystem, welches in Abbildung 4.4 durch x'-y' gekennzeichnet ist. Es entspricht dem ursprünglichen Koordinatensystem x-y mit einer Verschiebung des Nullpunktes um $disloc_x$ und $disloc_y$. Ganzzahlige Verschiebungen können nach der Berechnung durch eine zusätzliche Addition getätigt werden. Die Veränderung des Koordinatensystems durch die Verschiebung gilt es zu beachten, da sich nun zwei Koordinatentypen ergeben. Die Koordinaten des Rasters und die Koordinaten des wahren Kreises.

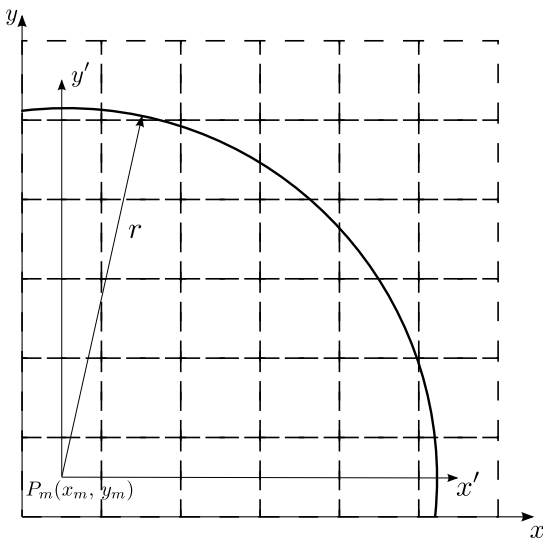

Abbildung 4.4: Entstehendes Unterkoordinatensystem durch Gleitkommaverschiebung bei Kurven. Durch die Gleitkommaverschiebung des Punktes $P_m(x_m, y_m)$ entsteht ein zusätzliches Untersystem x'-y' im bestehenden zweidimensionalen Koordinatensystem $x - y$.

Nutzt man den vorherigen Ansatz der x-Koordinatenwahl und erweitert die Berechnung der y-Koordinate, so ergibt sich der richtige Kreis. Hierzu wird $disloc_x$ wieder aufgerundet. Nun wird für den neuen ganzzahligen x-Wert der zugehörige wahre y-Wert berechnet. Das geometrische Vorgehen dafür ist in Abbildung 4.5 angedeutet.

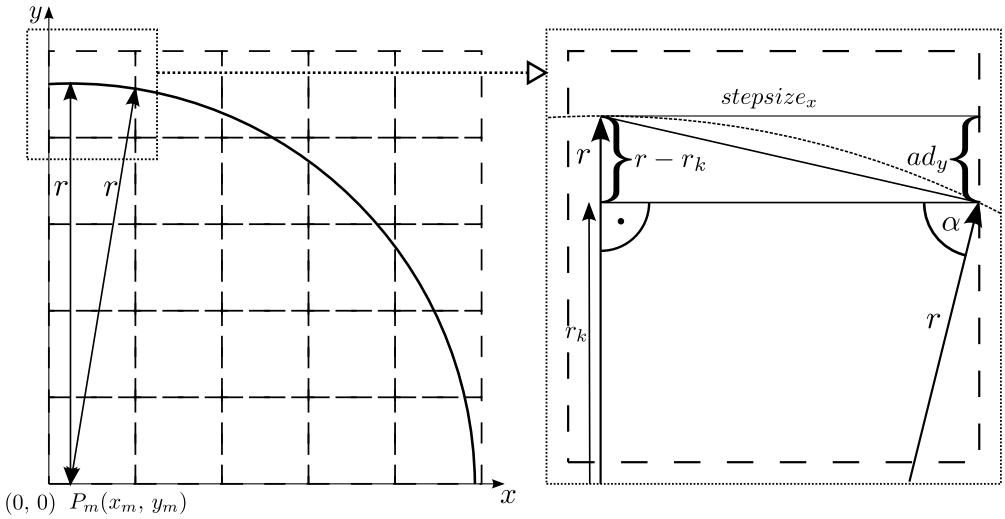

Abbildung 4.5: Startpunktwahl beim Kreis für Gleitkommaeingaben. *Mit dem Wählen der aufgerundeten x-Koordinate muss der y-Wert ebenfalls angepasst werden. Über Winkelberechnungen mit α lässt sich r_k berechnen. Über $r - r_k$ lässt sich die zusätzliche Verschiebung in y, ad_y, berechnen. Mit der neuen y-Koordinate $y - ad_y$ ist eine optimale Rasterpunktwahl in y möglich.*

Es wird der wahre y-Wert im Rastersystem x-y über einen zusätzlichen Abstand ad_y berechnet. Um ad_y berechnen zu können wird zuerst der Winkel α über die Kosinus-Funktion eines rechtwinkligen Dreiecks $\cos(\alpha) = \frac{Ankathete}{Hypotenuse}$ berechnet, siehe Formel 4.27. Die Ankathete ist hierbei der Abstand von $disloc_x$ zum nächsten ganzzahligen Wert, hier als $stepsize_x$ bezeichnet. Die Hypotenuse entspricht dem Radius r.

$$\begin{aligned} \cos(\alpha) &= \frac{stepsize_x}{r} \\ \alpha &= \arccos\left(\frac{stepsize_x}{r}\right) \end{aligned} \tag{4.27}$$

Anschließend wird über die Sinusfunktion $\sin(\alpha) = \frac{Gegenkathete}{Hypotenuse}$ die Länge des kurzen Radius r_k berechnet, wie in Formel 4.28.

$$\begin{aligned} \sin(\alpha) &= \frac{r_k}{r} \\ r_k &= \sin(\alpha) \cdot r \\ r_k &= \sin\left(\arccos\left(\frac{stepsize_x}{r}\right)\right) \cdot r \end{aligned} \tag{4.28}$$

Durch den Zusammenhang von $r - r_k = ad_y$ kann nun ad_y berechnet werden, siehe Gleichung 4.29.

$$\begin{aligned} ad_y &= r - r_k \\ ad_y &= r - \sin\left(\arccos\left(\frac{stepsize_x}{r}\right)\right) \cdot r \end{aligned} \tag{4.29}$$

Durch das nun berechnete ad_y kann im zweiten Oktant der wahre y-Wert im Rastersystem durch $y_0 = disloc_y + radius - ad_y$ berechnet werden. Dieser gilt für den aufgerundeten x-Wert. Mit diesem Wissen können nun die Abstände zu den beiden Rasterwerten in y berechnet werden, welche diesen Punkt umschließen. Für die obere Koordinate wird dies über ein Aufrunden von y_0 mit anschließender Subtraktion selbiger erreicht, $\lceil y_0 \rceil - y_0$[2]. Der Abstand der unteren Koordinate wird durch ein Dekrement mit 1 während des Aufrunden erreicht, $\lceil y_0 - 1 \rceil - y_0$. Die Koordinate mit dem kleineren Abstand wird als Startwert im Raster verwendet. Der Startpunkt für den zweiten Oktant ist damit durch die Werte $(\lceil stepsize_x \rceil,\ y_0)$ gewählt. Nun wird der Fehlerwert berechnet, welcher für die Schrittwahl von Bedeutung ist. Dieser wird über die allgemeine Fehlerformel 4.22 berechnet. Allerdings wird hier mit den Kreiswerten und nicht den Pixelkoordinaten gerechnet. Damit ergeben sich für x_0 und y_0 die Werte $x_0 = stepsize_x$ und $y_0 = y_0 - disloc_y$. Der Radius r bleibt unverändert, siehe Formel 4.30.

$$\begin{aligned} error_0 &= 2(x_0 + 1)^2 - 2r^2 + y_0^2 - (y_0 - 1)^2 \\ error_0 &= 2(stepsize_x + 1)^2 - 2r^2 + (y_0 - disloc_y)^2 - (y_0 - disloc_y - 1)^2 \end{aligned} \tag{4.30}$$

Während der Schrittwahl wird $stepsize_x$ über $stepsize_x + 1$ inkrementiert und y_0 mit $y_0 - 1$ dekrementiert, wie es das Bresenham-Verfahren für eine Pixelgröße von 1 vorschreibt. Die Fehlerveränderungen während der Schrittwahl werden, gleich dem Vorgehen für $error_0$ mit $x_n = stepsize_x$ und $y_n = y_n - disloc_y$ getätigt.

[2] $\lceil ... \rceil$ - Formelzeichen für Aufrunden (engl. ceiling).

Dieser Ansatz garantiert das Rechnen mit den richtigen Kreiskoordinaten im ganzzahligen Koordinatensystem. Der zweite Oktant wird hierbei optimal gerastert. Allerdings versagt er bei $r < 1$. Denn in diesem Fall wird durch Gleitkommaverschiebung die Winkelbeziehung beim Berechnen der y-Position verändert. Hier wechseln sich $stepsize_x$ und r in ihren Eigenschaften, da $stepsize_x$ die Hypotenuse bildet, anstatt r. Die verwendete Winkelgleichung erzeugt hierbei komplexe Ergebnisse, die nicht verwendet werden können. Eine Lösungsmöglichkeit ist, Oktanten auszuschließen, bei denen die Bedingung $|stepsize| \geq r$ erfüllt ist. Damit entstehen allerdings möglicherweise Lücken.

Mit diesem Konzept wurde lediglich die Startwahl algorithmisch verändert. Die Hauptverarbeitung bleibt algorithmisch das selbe Vorgehen. Allerdings müssen nun Gleitkomma- statt Ganzzahlberechnungen durchgeführt werden. Das erzeugt auf aktuellen Rechnern einen Geschwindigkeitsverlust durch größere Memory-Zugriffe.

4.3.5 Abbruchbedingung der Schrittwahl

Bei der Berechnung der Schritte in einem Kreis muss Oktantenweise vorgegangen werden. Hierzu wird eine Abbruchbedingung benötigt, welche garantiert, dass ein Oktant vollständig berechnet ist.

4.3.5.1 Betrachtung der Tangente

Ein Ansatz offenbart sich, wenn man den Grund der Einteilung des Kreises revidiert. Die Oktanten definieren sich durch die Steigung einer anlegbaren Tangente, wie in Abbildung 4.1b aufgezeigt wurde. J. Kennedy nutzt in [17] diese Eigenschaft bei der Ellipse. Da ein Kreis einen Spezialfall einer Ellipse darstellt, ist der Ansatz auch hier anwendbar. Dazu nimmt man die Formel für achsensymmetrische Ellipsen und führt ein implizites Differential durch. Durch $r_x = r_y$ erhält man damit die Steigung y' der angelegten Tangente eines Kreises, siehe Gleichung 4.31.

$$
\begin{aligned}
1 &= \frac{x_n^2}{r_x^2} + \frac{y_n^2}{r_y^2} \\
r_y^2 \cdot r_x^2 &= r_y^2 \cdot x_n^2 + r_x^2 \cdot y_n^2 \\
0 &= 2r_y^2 \cdot x_n + 2r_x^2 \cdot y \cdot y' \\
y' &= \frac{-2r_y^2 \cdot x_n}{2r_x^2 \cdot y_n} \\
y' &= \frac{-r_y^2 \cdot x_n}{r_x^2 \cdot y_n} \\
y' &= \frac{-x_n}{y_n}
\end{aligned} \tag{4.31}
$$

Die Steigung ist in jedem Oktant, im Vergleich mit den beiden anliegenden Oktanten, unterschiedlich. Hier kann dementsprechend die Schleife abgebrochen werden, wenn die Steigung einen bestimmten Wert erreicht. In Tabelle 4.3 sind die Bedingungen für alle Oktanten aufgezeigt.

Oktant	**Abbruchbedingung**
1	$-\infty \leq m_t \leq -1$
2	$0 \geq m_t \geq -1$
3	$0 \leq m_t \leq 1$
4	$\infty \geq m_t \geq 1$
5	$-\infty \leq m_t \leq -1$
6	$0 \geq m_t \geq -1$
7	$0 \leq m_t \leq 1$
8	$\infty \geq m_t \geq 1$

Tabelle 4.3: Abbruchbedingungen aller Oktanten beim Tangentenansatz für Kreis

Dieser Ansatz hat den Nachteil, dass in jedem Schleifendurchlauf der Schrittwahl die aktuelle Tangentensteigung berechnet werden muss.

4.3.5.2 Vorberechnen der benötigten Schritte

Um diese aufwändige Tangentenberechnung für jeden Schritt zu vermeiden, kann die mögliche maximale Anzahl der Schritte in der schnellen Dimension für jeden Oktant vorberechnet werden. Das ermöglicht ein präzises Preallokieren der Koordinatenmatrix. Weiterhin steht vorher fest, ob Oktanten nicht berechnet werden müssen, was zu Einsparungen im Aufwand führen und bei einem Radius $r \approx 1$ auftreten kann. Ausgangslage können die später eingeführten Bere-

chungsbereiche aus Abbildung 4.8 sein, siehe Abschnitt 4.3.6.2. Es wird die Eigenschaft des Kreises verwendet, dass die Oktantengrenze immer bei einem Winkel von 45° und einer Steigung $m = |1|$ auftritt. Über die Einheitsvektoren der Grenzen $\overrightarrow{grenze}_{1-2}(1,1)$, $\overrightarrow{grenze}_{7-8}(1,-1)$, $\overrightarrow{grenze}_{5-6}(-1,-1)$, $\overrightarrow{grenze}_{3-4}(-1,1)$, die Länge des Radius an der Grenze $\sin(45) \cdot r$ und dem Vektor der Gleitkommaverschiebung $\overrightarrow{disloc}(disloc_x, disloc_y)$ wird der Grenzpunkt P_{grenze} ohne ganzzahlige Verschiebung berechnet, siehe Gleichung 4.32.

$$P_{grenze} = \overrightarrow{grenze} \cdot \sin(45) \cdot r + \overrightarrow{disloc} \tag{4.32}$$

Anschließend wird der Grenzpunkt P_{grenze} für den ersten und zweiten Oktant aufgerundet, was die Pixelkoordinate C aus der Abbildung 4.6 ergibt, $\lceil P_{grenze} \rceil =$ C. Hierbei treten drei Fälle für die Lage der Oktantengrenze auf, welche in Abbildung 4.6a, b und c dargestellt sind. Welche der möglichen Pixelkoordinaten eines Oktant vom Algorithmus angesprochen werden, kann vorher nicht gesagt werden. Das ist begründbar über den aufsummierten Fehler während der Schrittwahl. Dieser kann nicht vorbestimmt werden.

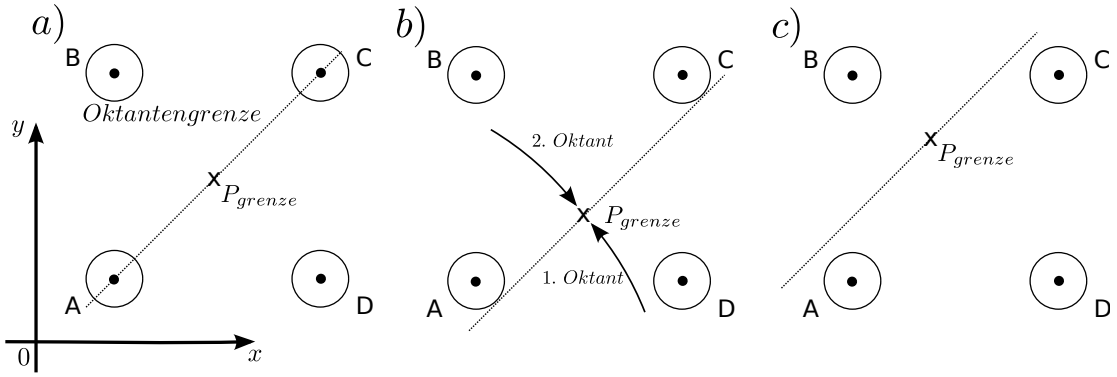

Abbildung 4.6: Varianten der Grenzpunktlage beim Kreis im Pixelraster im zwei-dimensionalen Raum. *Teilbild a zeigt den Fall, bei dem die Grenze direkt durch die Pixel* A *und* C *verläuft. Pixel* A *und* C *kommen für beide Oktanten als Endpunkt in Frage, je nach aufsummiertem Fehler. Pixel B und D können dem zweiten beziehungsweise ersten Oktant zugeordnet werden. Teilbild b zeigt den Fall, dass die Grenze unterhalb von Pixel* A *und* C *verläuft. Hier kommt für den ersten Oktant Pixel* D *in Frage, für den zweiten Oktant Pixel* A, B *und* C. *Teilbild c zeigt den Verlauf der Grenze oberhalb von Pixel* A *und* C. *Hier kommt für den zweiten Oktant Pixel* B *in Frage, für den ersten Oktant Pixel* A, C *und* D.

Bei der Auswahl der Pixelkoordinate C wird immer die, vom Mittelpunkt aus gesehen, äußerste Koordinate gewählt. Diese Koordinate wird verwendet, um die mögliche maximale Schrittanzahl jedes Oktanten zu berechnen. Die Tabelle 4.4 zeigt die Berechnung der Grenzpunkte des gesamten Kreises und die Behandlung dieser, um den zu betrachtenden Pixel zu ermitteln.

Oktant	**Punkt der Grenze**	**Prüfpixel aus P_{grenze}**
1,2	$P_{grenze12} = \overrightarrow{grenze}_{1-2} \cdot \sin(45) \cdot r + \overrightarrow{disloc}$	$(\lceil x \rceil, \lceil y \rceil)$
3,4	$P_{grenze34} = \overrightarrow{grenze}_{3-4} \cdot \sin(45) \cdot r + \overrightarrow{disloc}$	$(\lceil x \rceil, \lfloor y \rfloor)$
5,6	$P_{grenze56} = \overrightarrow{grenze}_{5-6} \cdot \sin(45) \cdot r + \overrightarrow{disloc}$	$(\lfloor x \rfloor, \lfloor y \rfloor)$
7,8	$P_{grenze78} = \overrightarrow{grenze}_{7-8} \cdot \sin(45) \cdot r + \overrightarrow{disloc}$	$(\lfloor x \rfloor, \lceil y \rceil)$

Tabelle 4.4: Mit den Formeln lassen sich die jeweiligen Grenzpunkte der acht Oktanten eines Kreises berechnen. Dabei wird der Einheitsvektor der jeweiligen Grenze $\overrightarrow{grenze}$ mit der Länge des Radius für die Grenze $\sin(45) \cdot r$ multipliziert und anschließend mit der Gleitkommaverschiebungsvektor $\overrightarrow{disloc}$ addiert. Der anschließend zu betrachtende Pixel wird durch auf- ($\lceil ... \rceil$) und abrunden ($\lfloor ... \rfloor$) ermittelt.

Die Koordinate wird anschließend über die Betrachtung der Tangentensteigung m_t abgeprüft, ob sie im aktuell behandelten Oktant enthalten ist. Dafür wird die Koordinate durch Subtraktion der Gleitkommaverschiebung in das Unterkoordinatensystem x'-y' des Kreises transformiert aus der Abbildung 4.4 und die Tangente nach der bekannten Formel 4.31 berechnet. An dieser Stelle können die Bedingungen aus der Tabelle 4.5 verwendet werden, um die Oktanten klar abzugrenzen. Schlägt die Prüfung der Tangentensteigung für einen Oktant fehl, wird die schnelle Dimension des Oktant bei der Pixelkoordinate verringert beziehungsweise der jeweiligen Achse um einen Schritt angenähert. Die Differenz aus dem ersten zu berechnenden Pixelwert der schnellen Dimension und dem zugehörigen Wert der Prüfkoordinate ergibt die maximale Schrittanzahl des Oktant. Dabei muss der Startpunkt mit eingerechnet werden, Differenz + 1. In den Oktanten 1, 4, 5, 8 wird die y-Dimension betrachtet, in den Oktanten 2, 3, 6, 7 die x-Dimension.

Dadurch, dass nur die maximal mögliche Schrittanzahl für einen Oktant berechnet werden kann, kommt es zu doppelten Koordinaten. Diese müssen abschließend herausgefiltert werden. Der entsprechende Teil wird im Abschnitt 4.3.6.3 erläutert.

Für Radien $r \approx 1$, wobei der Wert Eins für eine Pixel-/Voxellänge steht, kommt es durch die Gleitkommaverschiebung zu weiteren Problemen. Die Oktanten fallen teilweise in einzelne Pixel/Voxel zusammen, bedingt durch die Schrittgrößen.

4.3.6 Berechnung des gesamten Kreises

Beim Gleitkommaansatz wird durch eine Spiegelung ein Oktant zu einem kompletten Kreis ausgeweitet. Hierbei ist allerdings zu beachten, dass gewählte Koordinaten auch auf der Oktantengrenze liegen können. Hier kommt es dementsprechend zu Verdoppelungen, wenn man den kompletten Oktanten spiegelt. Eine Anforderung an den zu entwerfenden Algorithmus ist allerdings das Vermeiden von doppelten Punkten, wie in Anforderung [Anf03] angegeben wurde. Für die Spiegelung bedeutet das eine Betrachtung wie die Kreisrasterung aussieht.

4.3.6.1 Spiegelung der Koordinaten

Um entscheiden zu können, ob eine Rasterkoordinate auf dem Oktantenwechsel liegt, nutzt man wieder die Eigenschaft, dass der Oktantenwechsel bei 45° stattfindet. Über die Geradengleichung $y = m \cdot x + n$ mit $m = 1$ für den Fall von 45°, x als letzte berechnete x-Koordinate und $n = 0$ erhält man einen y-Wert. Stimmt dieser mit dem letzten berechneten y-Wert überein, liegt ein Rasterpunkt auf der Oktantengrenze. Hier muss anders gespiegelt werden, als wenn dieser Fall nicht eintritt. In Abbildung 4.7 ist das Vorgehen für die beiden Fälle dargestellt. Abbildung 4.7a zeigt dabei das Vorgehen bei einem Rasterpunkt auf der Oktantengrenze, Abbildung 4.7b die Koordinatenspiegelung, wenn kein Punkt auf der Grenze liegt. Die zugehörige Koordinatentransformation, um aus den Koordinaten des zweiten Oktant alle anderen zu transformieren, ist Abbildung 4.1 zu entnehmen.

Die hier dargestellte Koordinatenwahl in der Spiegelung ist nicht fest vorgeschrieben, hier können Abweichungen gemacht werden. So kann zum Beispiel der erste Oktant nicht ab der Oktantengrenze gespiegelt werden, dafür muss der achte Oktant diesen Bereich mit einschließen. Die Spiegelung funktioniert allerdings nur für Sonderfälle mit ganzzahliger Verschiebung. Ansonsten werden hier weitere Rundungsfehler erzeugt. Das ist durch eine asymmetrischen Koordinatenwahl bei Gleitkommaverschiebung begründbar. Dementsprechend können nicht mehr alle

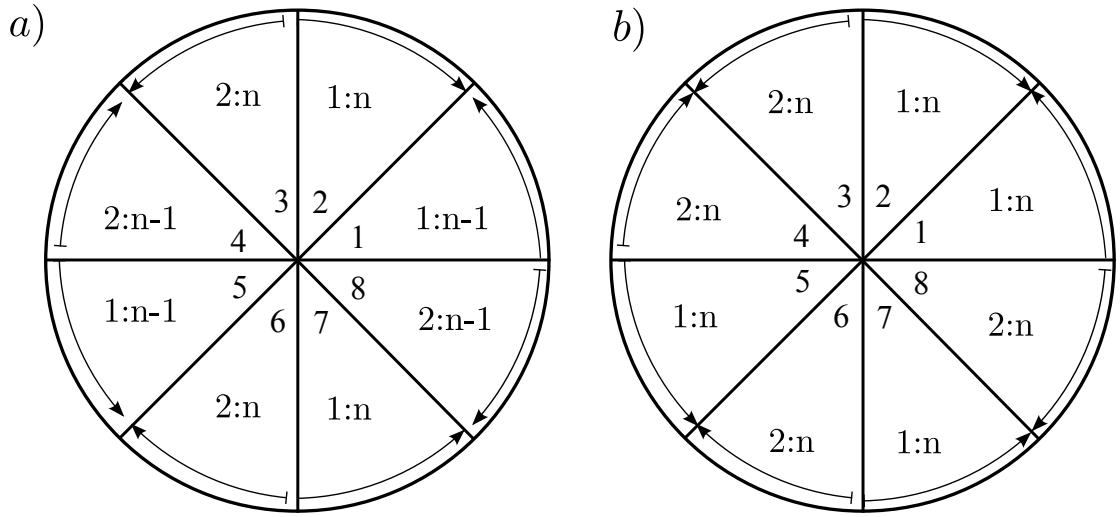

Abbildung 4.7: Verschiedene Koordinatenspiegelungsbereiche für den Kreis zur Vermeidung von doppelten Koordinaten. *Fall a zeigt das Vorgehen, bei komplett berechnetem zweitem Oktant und Rasterpunkt auf der Oktantengrenze. Fall b zeigt die Spiegelungsbereiche, wenn kein Rasterpunkt auf der Oktantengrenze liegt. Ein nicht anschließender oder nicht auf einer Oktantengrenze startender Pfeil symbolisiert das Auslassen der betroffenen Koordinate. Zur Definition wurde angegeben, welche der 1:n-Koordinaten des zweiten Oktant gespiegelt werden müssen. Dabei steht der Doppelpunkt für ein „bis". 1:n heißt dementsprechend alle Koordinaten von der ersten bis zur letzten müssen gespiegelt werden.*

Oktanten in alle anderen gespiegelt werden. Es muss jeder Oktant einzeln berechnet werden. Das ist allerdings nicht von Nachteil, da Berechnungen in der CPU als schneller angenommen werden können, als Lesen und Schreiben aus dem Speicher.

4.3.6.2 Tangentenbetrachtung zur Berechnung aller Oktanten

Die Einzelberechnung ist möglich, wenn man den zuvor besprochenen Tangentenansatz zum Abbrechen der Schrittwahl verwendet, siehe Abschnitt 4.3.5.1. In Abbildung 4.8 ist ein Vorgehen zur Berechnung der Oktanten schematisch aufgezeigt. Es wird im zweiten Oktant nicht auf der y-Achse angefangen zu rechnen, sondern auf der nächsten ganzzahligen x-Koordinate. Die Koordinatenberechnung läuft bis vor die Oktantengrenze und bricht dort ab. Der erste Oktant wird vom Schnittpunkt mit der x-Achse an berechnet und läuft bis und auf die Oktantengrenze. Der achte Oktant startet wiederum eine Koordinate unter dem Schnittpunkt mit der x-Achse und so weiter.

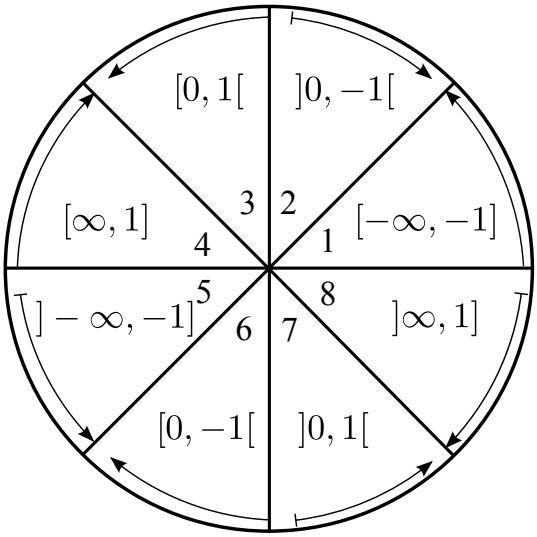

Abbildung 4.8: Berechnungsbereiche der Oktanten ohne doppelten Punkte beim Kreis. *Wenn im zweiten Oktant angefangen wird, so wird hier nach der y-Achse angefangen und bis vor die Oktantengrenze gerechnet. Der erste Oktant wird danach von der x-Achse aus bis auf die Oktantengrenze berechnet. Der achte Oktant beginnt unter der x-Achse und läuft bis auf die Oktantengrenze und so weiter. Die jeweiligen Tangentensteigungen sind als Intervall angegeben.*

Dieses Vorgehen erzeugt acht Berechnungsschleifen, mit einzelnen Abbruchbedingungen, siehe Tabelle 4.5. Die verwendeten Grenzbetrachtungen können hier-

bei individuell angepasst werden. Die Oktantengrenze zwischen erstem und achten Oktant kann beispielsweise beim achten Oktant berechnet werden. Hier muss der erste Oktant jedoch so angepasst werden, dass er über diesem Wert beginnt.

Oktant	Abbruchbedingung
1	$-\infty < m_t \leq -1$
2	$0 > m_t > -1$
3	$0 \leq m_t < 1$
4	$\infty > m_t \geq 1$
5	$-\infty < m_t \leq -1$
6	$0 \geq m_t > -1$
7	$0 < m_t < 1$
8	$\infty > m_t \geq 1$

Tabelle 4.5: Abbruchbedingungen der Oktanten für den Tangenten-Ansatz beim Kreis zur Vermeidung von doppelten Punkten. *Die Bedingungen mit $\geq$, $<$, $>$ und $\leq$ sichern, dass nur der betroffene Oktant berechnet wird und keine Schrittwahl über den gewollten Bereich hinausläuft.*

Die Steigung der Tangente muss in jedem Durchlauf für die aktuellen relativen Rasterkoordinaten mit der in Formel 4.31 aufgezeigten Gleichung berechnet werden. Die relativen Rasterkoordinaten sind das inkrementierte $stepsize_x$ und $y_n - disloc_y$. Der Grund hierfür ist wieder das Unterkoordinatensystem, was in Abbildung 4.4 aufgezeigt wurde.

4.3.6.3 Verwenden der vorberechneten Schrittanzahl

Wird der Ansatz zum Berechnen der maximal möglichen Schrittanzahl pro Oktant gewählt, benötigt man keine umfangreiche Abbruchbedingung mehr. Hier können vom verwendeten Startpunkt aus die Schritte gezählt werden. Ist die berechnete Schrittanzahl erreicht, ist der Oktant abgeschlossen. Werden keine Schritte für den gesamten Kreis berechnet, wird angenommen, dass alle Oktanten auf einen Pixel zusammenfallen. In diesem Fall werden die Abstände vom Mittelpunkt $P_m\,(x_m,\, y_m)$ zu den umgebenden Rasterkoordinaten A, B, C und D berechnet und der minimale gewählt, siehe Abbildung 4.9

Durch das Berechnen der maximal möglichen Schrittwahl kommt es jedoch zu maximal zwei doppelten Punkten pro Oktantengrenze, siehe Abbildung 4.10. Die doppelten Koordinaten können durch die berechnete Schrittanzahl jedes Oktanten leicht indiziert und abgeprüft werden. Geprüft werden müssen die jeweils letzten

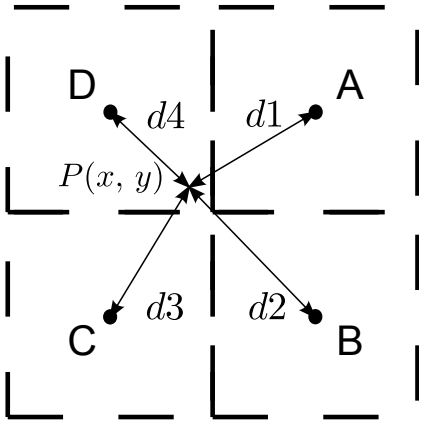

Abbildung 4.9: Ein Pixelwahlverfahren zur Annäherung eines Punktes P(x, y). *Über den Punkt $P(x, y)$ werden die Abstände $d1$, $d2$, $d3$ und $d4$ zu den Koordinaten A, B, C und D berechnet. A, B, C und D sind dabei die vier umgebenden Rasterpunkte. Die Koordinate mit minimalem Abstand wird gewählt.*

beiden Koordinaten der Oktanten, die einen Quadrant bilden. Mit einem abschließenden Kopieren der Koordinatenmatrix werden die doppelten Koordinaten ausgelassen.

Nähert sich der Radius der Größe eines Pixel an, so überlappen sich die Oktanten möglicherweise in ihren Laufweiten. Nicht jeder Oktant hat in diesem Bereich eine Schrittzahl von mindestens Zwei. Hier kann ein Brute-Force-Vergleich aller Koordinaten untereinander durchgeführt werden. Da der Grenzwert des Radius bedingt durch die Gleitkommaverschiebung für solch einen Fall nicht genau bestimmt werden kann, wurde der Bereich $r =]0, 3[$ ermpirisch gewählt. Die Pixelanzahl liegt in diesem Fall maximal um den Wert 25. Das bedeutet, dass der algorithmische Aufwand minimal ist, im Vergleich der zu berechnenden Anzahl an Pixel bei großen Radien.

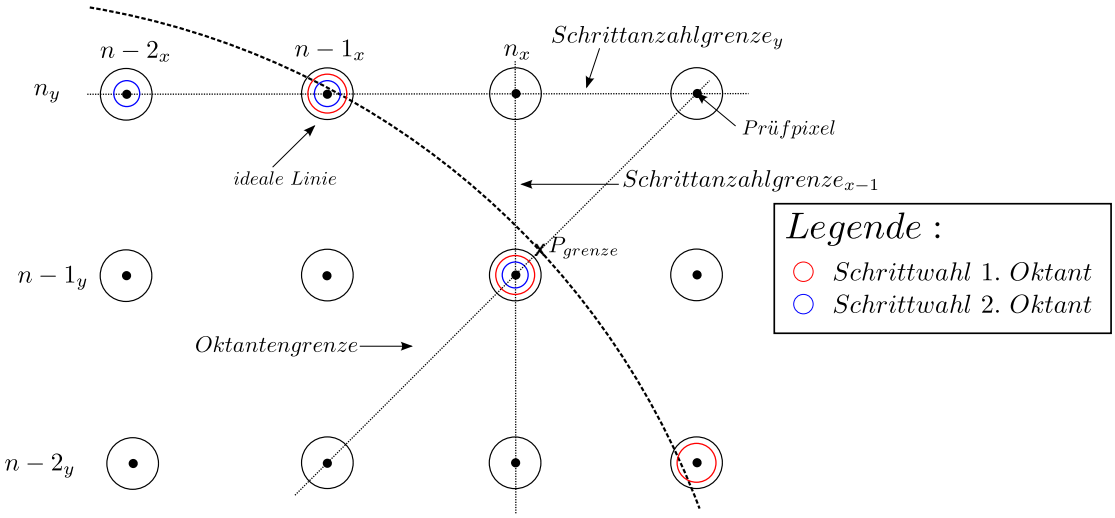

Abbildung 4.10: Doppelte Punkte durch Schrittzahlberechnung beim Kreis. *Mit den Grenzen der Schrittanzahl $Schrittanzahlgrenze_{x-1}$ für den zweiten und $Schrittanzahlgrenze_y$ für den ersten Oktant aus dem Prüfpixel $\lceil P_{grenze} \rceil$ wurde eine Schrittanzahl n_x bzw. n_y berechnet. Diese ist die maximal mögliche. Die Oktanten laufen über die Grenze hinaus, sichtbar an den roten und blauen Kreisen, welche für die Schrittwahl des ersten bzw. zweiten Oktant stehen.*

4.3.6.4 Spezialfall des Radius

In Anforderung [Anf02] wurde gefordert, dass der gesamte Parameterraum des USCT-Projektes durch den Zielalgorithmus abgedeckt sein wird. Dementsprechend kann der Radius den Wert $r \leq 0$ annehmen. Diese Variante ist bisher nicht durch den Algorithmus abgedeckt.
Um $r < 0$ zu ermöglichen, bedarf es lediglich einer anfänglichen Behandlung. Es wird nicht mehr mit dem eingegebenen Radius gerechnet, sondern mit dem absoluten Betrag des Radius $|r|$.

Für den Fall $r = 0$ wird nach Abbildung 4.9 vorgegangen. Hier werden die Abstände vom Mittelpunkt $P_m\,(x_m,\, y_m)$ zu den umgebenden Rasterkoordinaten A, B, C und D berechnet und der minimale gewählt.

Mit Betrachtung dieses Spezialfalls ist das Kreiskonzept fähig, Gleitkomma-Parameter korrekt zu verarbeiten.

4.4 Konzept für Kugel

Die Kugel ist eine Repräsentation mehrerer winkelgedrehter Kreise mit gemeinsamen Mittelpunkt im dreidimensionalen Koordinatensystem x-y-z. Durch diesen Zusammenhang können der Kreisalgorithmus und seine Prinzipien für die Kugel verwendet werden.

4.4.1 Berechnung von Unterkreisen

Eine Kugel in einem Raster besteht aus Kreisschichten, die in einer Dimension an den entsprechenden ganzzahligen Stellen positioniert sind. Jeder Teilkreis weist hierbei einen speziellen Radius auf. Diese einzelnen Radien können nach dem Winkelkonzept aus Abschnitt 4.3.3 *Beachten der Verschiebung in beiden Dimensionen* berechnet werden. Dabei wird für die ganzzahligen Rasterpunkte in der z-Dimension der jeweilige Radius r_n für den Unterkreis berechnet, siehe Abbildung 4.11. Die Distanz $stepsize_z$ gibt dabei die z-Position für den in x-y zu berechnenden Kreis an und . Da hier wieder mit den Winkelverhältnissen zwischen Radius r und Schrittweite $stepsize_z$ gearbeitet wird, treten auch hier Fälle auf, in denen $stepsize_z \geq r$ gilt. Hier kann kein realer Radius berechnet werden, weswegen diese Fälle gesondert behandelt werden müssen.

Die speziellen Radien müssen für jede Schicht berechnet werden, da der genaue Radius des Kreises benötigt wird. Wird hier ein Radius aus einer Annäherung in z-y verwendet, wird nicht die korrekte Kreisschicht gerastert. Diese Tatsache ist durch die Gleitkommaverschiebung bedingt.

In z werden keine Koordinaten berechnet, sondern die x-y-Koordinaten an die entsprechenden Stellen positioniert. Damit kann die Schrittabschätzung aus dem Abschnitt 4.3.5.2 für die Unterkreise verwendet werden. Dementsprechend steht vor dem Berechnen fest, wieviele Koordinaten maximal berechnet werden und die Koordinatenmatrix kann ordnungsgemäß preallokiert werden.

Dieser Ansatz erzeugt eine Kugel durch Verwendung des Bresenham-Verfahrens in zwei Dimensionen. Diese können frei gewählt werden. Da der Kreis schon in x-y konzipiert wurde, wurde sich entschieden die Konzepte weiterhin zu nutzen. Allerdings kommt es zu Problemen an den Grenzpunkten der Kugel in der z-Dimension. Erreicht die Kugel hier keinen Rasterpunkt, wird die entsprechende Stelle ausge-

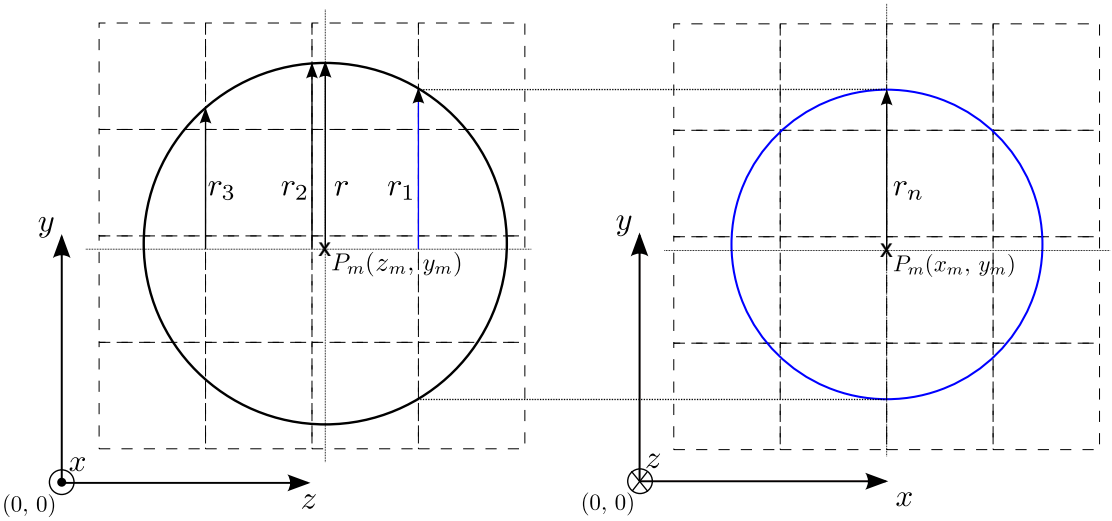

Abbildung 4.11: Kugelerzeugung durch das Verwenden des Kreis-Algorithmus. *Durch ein Bestimmen der verschiedenen Radien r_n für Rasterpunkte in z-y, können über den Kreis-Algorithmus die zugehörigen x-y-Koordinaten in den entsprechenden Unterkreisen berechnet werden. Diese Kreise werden anschließend an die zugehörige z-Position gesetzt.*

lassen. Aus dieser Problemstellung heraus wurde eine Lösung entworfen, um diese Grenzfälle zu behandeln.

4.4.2 Betrachtung der Grenzpunkte in z-Dimension

Mit Gleitkommaeingaben beim Radius r und dem Mittelpunkt $P_m\,(x,\, y\,, z)$ liegen die Grenzen der Kugel in z-Dimension nicht in allen Fällen auf einer Rasterdimension. Durch die bisherige Betrachtung des jeweiligen Radius r_n für die Rasterpositionen in z werden diese Fälle nur beachtet, wenn die Rasterposition innerhalb des Kreises liegt und $stepsize_z < r$ gilt. Für die anderen Fälle kann angenommen werden, dass der Unterkreis möglicherweise auf einen Voxel zusammenfällt. Um entscheiden zu können, ob diese Annahme zutrifft müssen die Distanzen d_z1 und d_z2 der Kugel zum nächstgelegenen Rasterpunkt betrachtet werden, siehe Abbildung 4.12.

Mit den Grenzpunkten $(z_m - radius,\, y_m)$ und $(z_m + radius,\, y_m)$ und den Schrittweiten $stepsize_{z1}$ und $stepsize_{z2}$ können die Abstände d_z1 und d_z2 berechnet und betrachtet werden. Bei einer Voxellänge von Eins ist eine optimale Rasterung bei einem Abstand von $0,5$ Voxel anzunehmen. Dementsprechend kann

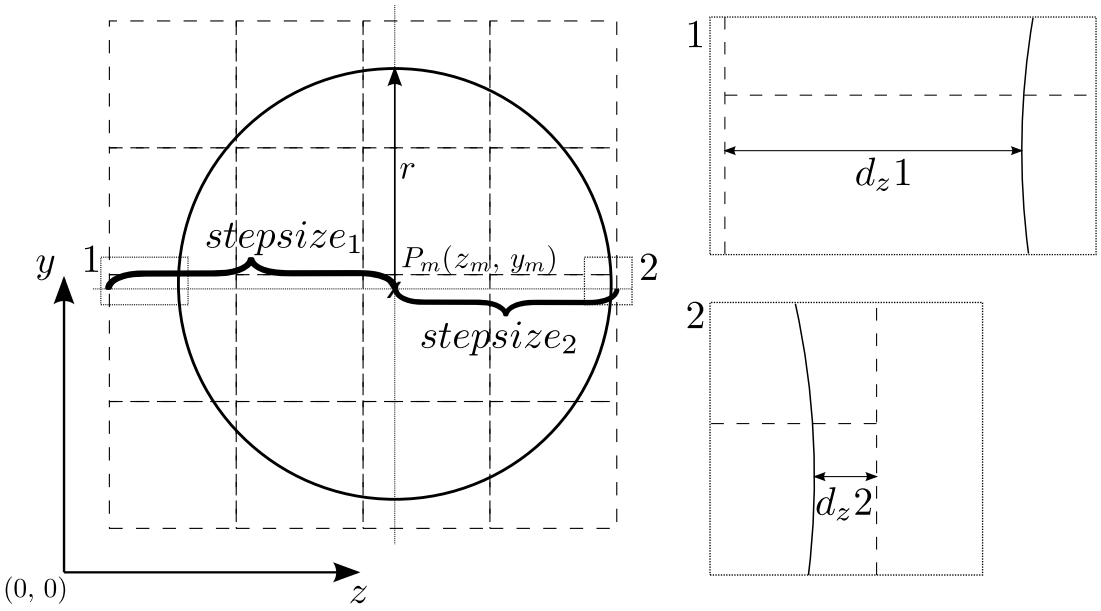

Abbildung 4.12: Die Grenzbetrachtung an der Kugel in z-Dimension. Über die Grenzpunkte $(z_m - radius,\, y_m)$ und $(z_m + radius,\, y_m)$ und der jeweiligen Schrittweite auf eine ganzzahlige Position $stepsize_1$ und $stepsize_2$ können die Abstände d_z1 und d_z2 berechnet werden.

bei $d_z1 \leq 0,5$ oder $d_z2 \leq 0,5$ die bestmögliche Annäherung über einen Voxel nach Abbildung 4.13 im drei-dimensionalen getätigt werden.

Mit dieser Betrachtung ist das Kugelkonzept fähig, Kugeln mit Gleitkommma-Parametern zu rastern. Dabei wird das Kreiskonzept komplett wiederverwendet und eine halbe Lösung nach Bresenham-Verfahren verwendet.

4.5 Achsensymmetrische Ellipse

Der folgende Abschnitt befasst sich mit einem Rasteransatz für achsensymmetrische Ellipsen. Dieser ist von J. van Aken in [13] vorgestellt worden. Er wurde gewählt, da er der beste beschriebene und dokumentierte Ansatz in der vorhandenen Literatur war. Im Folgenden wird dabei nur auf den Zweipunktalgorithmus als Schrittwahl eingegangen. Für die Erläuterung und Herleitung des Mittelpunktalgorithmus wird auf [13] verwiesen.

Die Schrittwahlmöglichkeiten für die Ellipse mit dem Zweipunktalgorithmus sind in Abbildung 4.14 aufgezeigt. Dabei ist die Schrittwahl a im ersten Oktant und b im zweiten Oktant zutreffend.

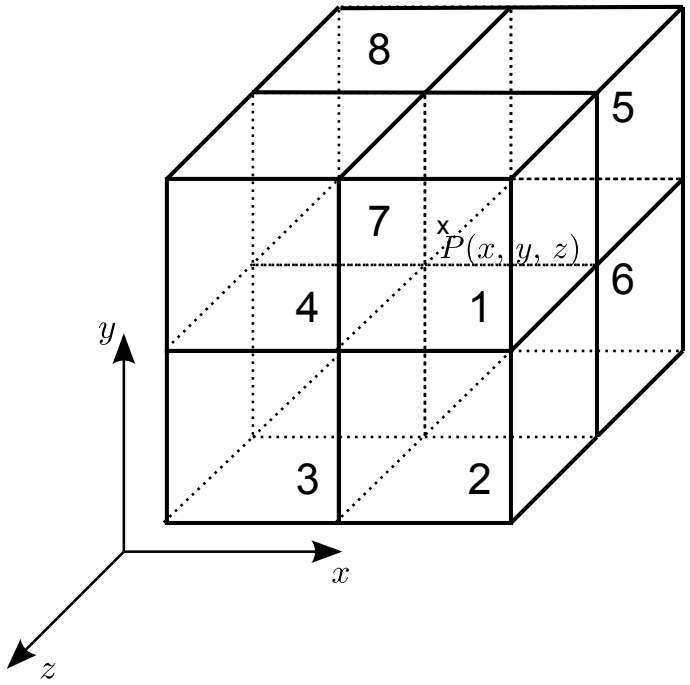

Abbildung 4.13: Ein Voxelwahlverfahren zur Annäherung eines Punktes P (x, y, z). *Wie im zwei-dimensionalen müssen die vektoriellen Abstände zu den umliegenden Voxeln bestimmt werden. Hier sind allerdings acht mögliche Voxel zu prüfen, statt vier Pixel. Der Voxel mit minimalem Abstand wird gewählt. Im Bild liegen die Voxel 1-4 auf der vorderen z-Ebene und 5-8 auf der hinteren.*

Vom Startpunkt A aus wird betrachtet, ob die Koordinate C oder D beziehungsweise die Koordinate B oder C als nächstes angesprochen werden soll. Dies geschieht über die Subtraktion der spezifischen Abstände $e1$, $e2$ und $e3$. Sie geben den Abstand der Koordinate von der Ellipse an. Die Formelherleitung geht von der allgemeinen Ellipsengleichung aus, welche über Umstellen eine Form zur Berechnung des Fehlers erreicht, siehe Gleichung 4.33.

$$\begin{aligned} 1 &= \frac{x_n^2}{r_x^2} + \frac{y_n^2}{r_y^2} \\ \text{error} &= r_y^2 \cdot x_n^2 + r_x^2 \cdot y_n^2 - r_x^2 \cdot r_y^2 \end{aligned} \tag{4.33}$$

[3]Mit Änderungen entnommen aus: J. van Aken (1984), S. 31

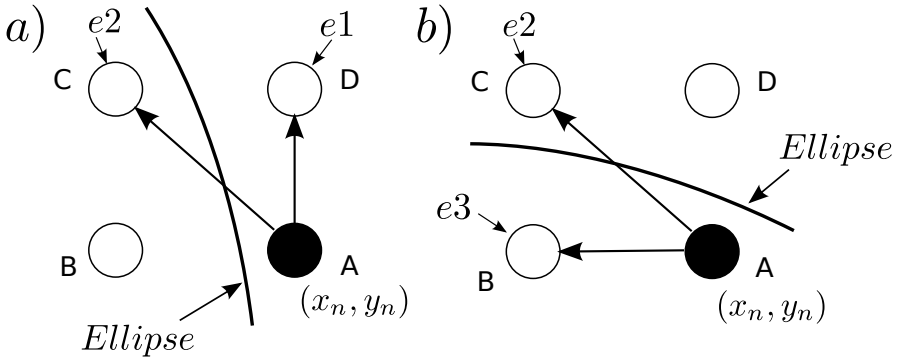

Abbildung 4.14: Die Schrittwahlmöglichkeiten bei einer Ellipse für den Zweipunktalgorithmus. *Im ersten Oktant gilt die Schrittwahl wie sie in a aufgezeigt ist, im zweiten Oktant die Variante b. e1, e2 und e3 geben den Abstand der Koordinate zur Ellipse an. A ist immer der Ausgangspunkt, B,C und D sind mögliche Positionen für den nächsten Schritt.*[3]

Im ersten Oktant liegt die Wahl zwischen den Koordinaten C und D. Der Wert von *error* gibt an, wo die Ellipse im Verhältnis zu diesen beiden Punkten liegt. Ist $error < 0$ liegt die Koordinate innerhalb der Ellipse. Bei $error > 0$ liegt sie außerhalb. Für D wird $e1 = \text{error}\,(x_n,\, y_n + 1)$ verwendet, $e2 = \text{error}\,(x_n - 1,\, y_n + 1)$ für Koordinate C. Damit werden die Fehlerwerte der Koordinaten erhalten.

Im Algorithmus werden zwei Entscheidungsvariablen $d1$ und $d2$ verwendet. Im ersten Oktant entscheidet das Vorzeichen von $d1$ über die Schrittwahl. Wenn $d1 < 0$ ist, wird Koordinate D angesprochen, bei $d1 \geq 0$ Koordinate C, siehe Formel 4.34.

$$\begin{aligned} d1 &= e1 - (-e2) \\ &= \text{error}\,(x_n,\, y_n + 1) + \text{error}\,(x_n - 1,\, y_n + 1) \\ &= r_y^2 \cdot x_n^2 + r_x^2 \cdot (y_n + 1)^2 - r_x^2 \cdot r_y^2 \\ &\quad + r_y^2 \cdot (x_n - 1)^2 + r_x^2 \cdot (y_n + 1)^2 - r_x^2 \cdot r_y^2 \\ &= r_y^2 \cdot \left(2x_n^2 - 2x_n + 1\right) \\ &\quad + r_x^2 \cdot \left(2y_n^2 + 4y_n + 2\right) - 2r_x^2 \cdot r_y^2 \end{aligned} \tag{4.34}$$

Die Entscheidungsvariable $d2$ ist zuständig für die Schrittwahl im zweiten Oktant zwischen den Koordinaten B und C, siehe Formel 4.35.

$$\begin{aligned} d2 &= e2 - (-e3) \\ &= r_y^2 \cdot (x_n - 1)^2 + r_x^2 \cdot (y_n + 1)^2 - r_x^2 \cdot r_y^2 \\ &\quad + r_y^2 \cdot (x_n - 1)^2 + r_x^2 \cdot y_n^2 - r_x^2 \cdot r_y^2 \\ &= r_y^2 \cdot \left(2x_n^2 - 4x_n + 2\right) \\ &\quad + r_x^2 \cdot \left(2y_n^2 + 2y_n + 1\right) - 2r_x^2 \cdot r_y^2 \end{aligned} \tag{4.35}$$

Da der Algorithmus im ersten Oktant startet, wird $d2$ auch als Grenzbetrachtung für die Oktanten verwendet, denn $d2$ wird positiv, sobald der zweite Oktant erreicht ist. Die Startwerte von $d1$ und $d2$ mit $x_0 = r_x$ und $y_0 = 0$ für den ersten Oktant sind in den Formeln 4.36 und 4.37 aufgezeigt.

$$\begin{aligned} d1_0 &= \text{error}\,(x_0, y_0 + 1) + \text{error}\,(x_0 - 1, y_0 + 1) \\ d1_0 &= 2r_x^2 - 2r_x \cdot r_y^2 + r_y^2 \end{aligned} \tag{4.36}$$

$$\begin{aligned} d2_0 &= \text{error}\,(x_0 - 1, y_0 + 1) + \text{error}\,(x_0 - 1, y_0) \\ d2_0 &= r_x^2 - 4r_x \cdot r_y^2 + 2r_y^2 \end{aligned} \tag{4.37}$$

Wenn D mit $x_{n+1} = x_n$ und $y_{n+1} = y_n + 1$ gewählt wird ergeben sich für $d1_{n+1}$ und $d2_{n+1}$ die folgenden Formeln 4.38 und 4.39.

$$\begin{aligned} d1_{n+1} &= r_y^2 \cdot \left(2x_{n+1}^2 - 2x_{n+1} + 1\right) \\ &\quad + r_x^2 \cdot \left(2y_{n+1}^2 + 4y_{n+1} + 2\right) - 2r_x^2 \cdot r_y^2 \\ &= r_y^2 \cdot \left(2x_n^2 - 2x_n + 1\right) \\ &\quad + r_x^2 \cdot \left(2\,(y_n + 1)^2 + 4\,(y_n + 1) + 2\right) - 2r_x^2 \cdot r_y^2 \\ &= \text{error}\,(x_n,\, y_n + 1) + \text{error}\,(x_n - 1,\, y_n + 1) \\ &\quad + r_x^2 \cdot (4y_n + 6) \\ &= d1_n + 4r_x^2 \cdot y_{n+1} + 2r_x^2 \end{aligned} \tag{4.38}$$

$$d2_{n+1} = d2_n + 4r_x^2 \cdot y_{n+1} \tag{4.39}$$

Wenn C mit $x_{n+1} = x_n - 1$ und $y_n = y_n + 1$ gewählt wird, ergeben sich die Formeln 4.40 und 4.41.

$$d1_{n+1} = d1_n - 4r_y^2 \cdot x_{n+1} + 4r_x^2 \cdot y_{n+1} + 2r_x^2 \tag{4.40}$$

$$d2_{n+1} = d2_n - 4r_y^2 \cdot x_{n+1} + 4r_x^2 \cdot y_{n+1} + 2r_y^2 \tag{4.41}$$

Im Anhang B ist der Algorithmus nach J. van Aken aus [13] aufgezeigt.

5 Implementierung

Das folgende Kapitel befasst sich mit der Umsetzung der entwickelten Konzepte. Dabei werden hauptsächlich Besonderheiten in den Implementierungen berücksichtigt. Die Implementierung wurde aus den Anforderungen [Anf05] und [Anf06] motiviert. Alle in diesem Kapitel aufgeführten Codeausschnitte sind dementsprechend in MATLAB verfasst. Anforderung [Anf07] fordert keine dynamische Funktionalität, was bei der Implementierung beachtet wurde.

5.1 Gerade

Der Geraden-Algorithmus wurde nach den Konzepten und Vorarbeiten aus dem Abschnitt 4.2 entworfen.

5.1.1 Preallokierung

Um die Rasterkoordinaten zu speichern wird ein Array *coords* definiert. Es ist dazu bestimmt die berechneten x- und y-Koordinaten aufzunehmen. Eine naive Implementierung würde dynamische Arrays vorziehen. Hier zeigt sich mit MATLAB allerdings eine schlechte Performance, da der Interpreter wachsende Arrays mit Kopierbefehlen umsetzt. Hier schlägt der MATLAB-interne Lint („m-lint“) Alarm, wenn nicht preallokiert wird. Für eine Preallokierung muss allerdings feststehen, wie groß das Array werden soll. Für die Gerade nach Bresenham im n-dimensionalen Raum ist dies berechenbar. Dafür kann die Eigenschaft verwendet werden, dass während der Schrittwahl in jedem Schleifendurchlauf eine Dimension immer verändert wird, die schnelle Laufrichtung. Die größte Dynamik der Geraden gibt dementsprechend an, wieviele Koordinaten berechnet werden. Hinzu kommt die Startkoordinate. Im Listing 5.1 ist die Berechnung für die Gerade in 3D aufgeführt. Es wird über die $max()$-Funktion der größte absolute Wert ($abs()$) zwischen der Dynamik in x ($diffX$), der Dynamik in y ($diffY$), und der Dynamik

in z ($diffZ$), gewählt. Anschließend wird das Array *coords* als Integer-Array preallokiert, siehe Listing 5.1 Zeile 2. Dabei ist die Größe über die Variable *linesize* gegeben, welche die größte Dynamik in einer Dimension, wie im Abschnitt 4.2.3 erläutert, repräsentiert, siehe Listing 5.1 Zeile 1. Die Addition mit dem Wert Eins findet statt, um den Startpunkt mit zu erfassen. bei der Preallokierung kommt durhc Es entsteht ein zweidimensionales Array mit $n \times 3$ Einträgen, welches drei Dimensionen in den Koordinaten abbildet. Es mit dem Wert Null gefüllt sind. Über $coords(n, N)$ wird später auf die einzelnen Stellen zugegriffen und die Positionen abgespeichert. Zu beachten ist die Dimension N, welche die Werte 1, 2 oder 3 annehmen kann und für x, y und z steht. n steht für die Zeile der gewollten Koordinate (x, y, z).

```
1 linesize = max(abs(diffX),max(abs(diffY),abs(diffZ))+1;
2 coords = zeros(linesize,3,'int32');
```

Listing 5.1: Preallokieren der Koordinatenmatrix im Geradenalgorithmus im drei-Dimensionalen.

5.1.2 Hauptverarbeitung

Die Hauptverarbeitung findet in *for*-Schleifen statt. Das spart die Vergleiche in jedem Schleifendurchlauf bei einer *while*-Schleife und definiert genau die Grenzen der Schleifen. Die Grenzen der Schleifen sind die jeweils größeren Dynamiken der beiden langsamen Richtungen.

5.2 Kreis

Der Kreisalgorithmus wurde nach den Konzepten der Vorberechnung der nötigen Schritte (Abschnitt 4.3.5.2) und Schrittwahl mit Beachtung beider Dimensionen (Abschnitt 4.3.4.3) implementiert. Weiterhin wurde das Konzept für die Behandlung des Spezialfall des Radius umgesetzt, in denen der Radius $r \leq 0$ ist (Abschnitt 4.3.6.4). Der Code weist wenig dynamische Interpreter-Funktionen auf und ist damit leicht auf andere Sprachen portierbar.

5.2.1 Grundaufbau

In Abbildung 5.1 ist der grobe Ablaufplan des implementierten Bresenham-Kreis-Algorithmus aufgezeigt. In den folgenden Abschnitten werden Aspekte aus dem Algorithmus näher erläutert.

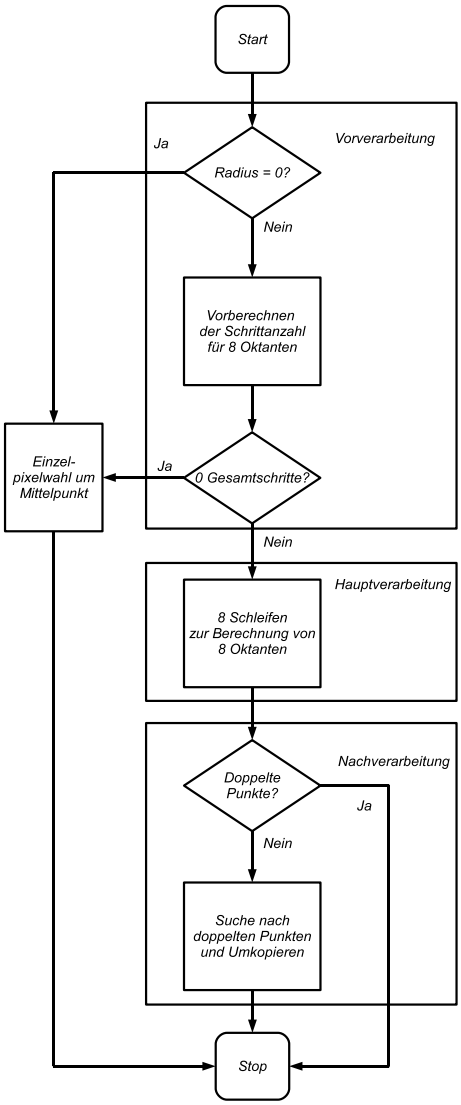

Abbildung 5.1: Abstrakter Programmablaufplan des implementierten Bresenham-Kreis-Algorithmus. *Ist der Radius $r = 0$ wird ein Pixel um den Mittelpunkt gewählt. Ansonsten werden die Startwerte und Schrittzahlen pro Oktant berechnet. Hat der Kreis dabei eine Gesamtschrittzahl von Null, wird ebenfalls ein Pixel am Mittelpunkt gewählt. Anschließend folgt die Hauptverarbeitung mit acht for-Schleifen, in denen die Oktantkoordinaten berechnet werden. Zum Schluss folgt in der Nachverarbeitung die Entfernung der doppelten Koordinaten.*

5.2.2 Einzelpixelwahl um Mittelpunkt

Der Spezialfall des Radius, siehe Abschnitt 4.3.6.4, wird zu Beginn durch eine Transformation in den absoluten Wert und eine if-Anweisung zur Prüfung auf $r = 0$ abgehandelt, siehe Abbildung 5.1. Das ermöglicht eine klare Trennung entgegen der restlichen Verarbeitung. Die Suche nach dem bestmöglichen Nachbarn wurde in eine Unterfunktion *chooseOnePoint* ausgelagert, siehe Listing 5.2. Dieser Schritt wurde getätigt, da das Verfahren der Pixelwahl um den Mittelpunkt bei einer berechneten Gesamtschrittzahl von Null ebenfalls angewendet wird.

```
function coords = chooseOnePoint(centre)                                    1
x1 = ceil(centre(1)); x2 = x1 - 1;                                          2
y1 = ceil(centre(2)); y2 = y1 - 1;                                          3
sumerror1 = norm([x1 y1]-centre);                                           4
sumerror2 = norm([x1 y2]-centre);                                           5
sumerror3 = norm([x2 y1]-centre);                                           6
sumerror4 = norm([x2 y2]-centre);                                           7
[sumerror,i] = min(sumerror1,min(sumerror2,min(sumerror3,sumerror4          8
   )));
switch i                                                                    9
  case 1                                                                    10
    coords = [x1 y1];                                                       11
  case 2                                                                    12
    coords = [x1 y2];                                                       13
  case 3                                                                    14
    coords = [x2 y1];                                                       15
  case 4                                                                    16
    coords = [x2 y2];                                                       17
end                                                                         18
end                                                                         19
```

Listing 5.2: Die Unterfunktion *chooseOnePoint* im Bresenham-Kreis-Algorithmus. Hier wird die Koordinate um den Mittelpunkt *centre* gewählt, die den minimalen Abstand aufweist.

In dieser Funktion werden die vier möglichen Koordinaten um den Mittelpunkt *centre* erstellt, siehe Listing 5.2 Zeilen 2-3. Anschließend wird in den Zeilen 4-7 über norm $([xN\ yN] - centre)$ die Länge des Abstandsvektor ermittelt. Zum Schluss wird in den restlichen Zeilen die minimale Abweichung gewählt und die entsprechende Koordinate als gewählt gesetzt.

5.2.3 Vorverarbeitung

Die Vorverarbeitung aus dem Ablaufplan in Abbildung 5.1 wurde als eine Unterfunktion *calculateStartvaluesPerOctant* implementiert, siehe Listing 5.3. In ihr werden Schrittanzahl, Startposition in der langsamen Dimension und Startfehlerwert für einen ausgewählten Oktant über eine switch-Anweisung berechnet. Dies dient der Übersichtlichkeit und soll als mögliche Optimierungsstelle für ein späteres Refactoring gelten. Eine Generalisierung wurde hier nicht umgesetzt, da Unterschiede in den Berechnungen des Startfehlers und der Startposition bestehen.

Anfangs findet eine Abfrage statt, um die Verletzung des Winkelverhältnisses bei der Startpixelwahl durch $|stepsize| \geq radius$ zu verhindern. Die Funktion beendet sich und gibt für den aktuellen Oktant eine Schrittzahl von 0 zurück, siehe Listing 5.3 Zeilen 3-8.

Anschließend findet die Berechnung der Schrittzahl statt, siehe Zeilen 14-24. Mit dem Grenzpunkt des Oktant *borderpoint* werden, beispielsweise für den ersten Oktant, die Koordinaten auf die nächste Koordinate aufgerundet (ceil (*borderpoint*)). Ist anschließend die entsprechende Tangentenbedingung erfüllt, liegt die Koordinate nicht im zu berechnenden Oktant. Die jeweils schnelle Dimension (im ersten Oktant y - pixel(2)) wird um Eins verringert. Durch eine Subtraktion mit der Startkoordinate des Oktant wird die Schrittzahl berechnet. Hinzu kommt die Startkoordinate selber durch +1.

```
function [nSteps,choosenPixel,starterror] =                    1
    calculateStartvaluesPerOctant(octand,stepsize,realStart,
    pixelstart,borderpoint)
% wenn Winkelbeziehung verletzt, überspringe Oktant            2
if abs(stepsize)>=radius                                       3
  nSteps = int32(0);                                           4
  choosenPixel = 0;                                            5
  starterror = 0;                                              6
  return                                                       7
end                                                            8
additionalDelta = radius - sin(acos(stepsize/radius))*radius;  9
% switch über die 8 Oktanten                                   10
switch octand                                                  11
  case 1                                                       12
    % Berechnen der Schrittanzahl                              13
```

```
    pixel = ceil(borderpoint);                                          14
    if -((pixel(1)-dislocX)/(pixel(2)-dislocY)) > -1                    15
      pixel(2) = pixel(2)-1;                                            16
    end                                                                 17
    nSteps = int32(abs(pixel(2)-pixelstart)+1);                         18
    % Startpixelwahl in langsamer Dimension                            19
    realValue = realStart-additionalDelta;                              20
    choosenPixel = chooseInSlowDirection(realStart,                     21
       additionalDelta);
    % Berechnung des Startfehlers                                       22
    starterror = 2*(stepsize+1)^2 - 2*radius^2 + (choosenPixel-         23
       dislocX)^2 + (choosenPixel-dislocX-1)^2;
    return                                                              24
  case 2                                                                25
    ...                                                                 26
  case 3                                                                27
    ...                                                                 28
```

Listing 5.3: Unterfunktion *calculateStartValuesPerOctant* im Bresenham-Kreis-Algorithmus zur Berechnung der Schrittanzahl, Startpixelwahl in langsamer Schrittdimension und Startfehler.

Die Wahl der Startdimension in langsamer Laufrichtung wurde in eine weitere Unterfunktion *chooseInSlowDirection* ausgelagert, siehe Listing 5.4. Dieser Schritt wurde durchgeführt, da der Code für die Wahl bei jedem Oktant einheitlich ist und damit Codeduplizierung vermieden wird. In dieser Funktion werden die Abstände zu den beiden umliegenden Koordinaten *dislocHighPoint* und *dislocLowPoint* bestimmt und der minimale gewählt. Damit ist die Wahl der Startkoordinate des Oktant in beiden Dimensionen abgeschlossen.

```
function pixeldimension = chooseInSlowDirection(realStart,
    additionalDelta)
dislocHighPoint = (ceil(realValue)-realValue);
dislocLowPoint = ((ceil(realValue)-1)-realValue);
if abs(dislocLowPoint) <= abs(dislocHighPoint)
  pixeldimension = realValue + dislocLowPoint;
else
  pixeldimension = realValue + dislocHighPoint;
end
end
```

Listing 5.4: Die Unterfunktion *chooseInSlowDirection* im Kreis-Algorithmus zum Wählen des Startpixels in langsamer Laufrichtung.

5.2.4 Preallokierung der Koordinatenmatrix

Mit durchgeführter Berechnung steht die mögliche Schrittzahl für jeden Oktant zur Verfügung. Nun kann, wie beim Geraden-Algorithmus auch, die Koordinatenmatrix preallokiert werden, siehe Listing 5.5. Es entsteht ein $n \times 2$ großes Integerarray, in die die Koordinaten eingespeichert werden. Die Variablen $n1$ bis $n8$ stehen für die Anzahl der Schritte des ersten bis achten Oktant.

```
% addieren der Schrittanzahl pro Oktant
n=n1+n2+n3+n4+n5+n6+n7+n8;
% Preallokation
coords = zeros(n,2,'int32');
```

Listing 5.5: Preallokieren der Koordinatenmatrix im Bresenham-Kreis-Algorithmus.

5.2.5 Hauptverarbeitung

Die Hauptverarbeitung aus Abbildung 5.1 in Form der Koordinatenberechnung für die Rasterung des Kreises findet in acht for-Schleifen statt, welche jeweils pro abgearbeitet werden. Die Grenzen sind durch die jeweils berechnete Schrittzahl gegeben. Die Reihenfolge der Schleifen beginnt beim zweiten Oktant und läuft im Uhrzeigersinn bis zum dritten Oktant (Reihenfolge: 2.-1.-8.-7.-6.-5.-4.-3. Oktant). Die Reihenfolge der Schleifen ist beliebig, da die Oktantberechnung unabhängig abläuft. Die Aufteilung in diese acht Schleifen ist notwendig, da jeder Oktant

eine andere Laufrichtung aufweist und die Formeln der Fehlerveränderungen daraufhin ebenfalls variieren. Im Listing 5.6 ist exemplarisch die Schleife für den zweiten Oktant aufgeführt. Darin wird zuerst die aktuelle Koordinate abgespeichert (Zeilen 2-3), welche zu Beginn den Startpunkt darstellt. *stepsize*2 ist dabei der aktuelle Schritt in x-Dimension bis zum nächsten ganzzahligen Rasterwert. *dislocX* gibt die statische Verschiebung des Mittelpunktes im Gleitkommabereich dar. $stepsizeN + disloc$ ergibt in allen Fällen einen ganzzahligen Wert, da die beiden Variablen sich ergänzen. *choosenPixel*2 gibt den aktuellen Wert in der y-Dimension an. Anschließend wird die Fehlervariable *sumerror* auf ihren Status abgeprüft. Bei $sumerror < 0$ ist ein Schritt in der x-Dimension nötig, ansonsten in x und y. Die entsprechenden Fehleranpassungen werden durchgeführt und *stepsize*2 und *choosenPixel*2 entsprechend angepasst oder unverändert gelassen, siehe Zeilen 4-14.

```
for i=1:n2                                                          1
  coords(i,1) = stepsize2+dislocX;                                  2
  coords(i,2) = choosenPixel2;                                      3
  if sumerror<0                                                     4
    % addiere Fehlerveränderung für x-Schritt                       5
    sumerror = sumerror + 4*stepsize2 + 6;                          6
  else                                                              7
    % addiere Fehlerveränderung für xy-Schritt                      8
    sumerror = sumerror + 4*(stepsize2-(choosenPixel2-dislocY)) +   9
       10;
    % vollziehe y-Schritt                                           10
    choosenPixel2 = choosenPixel2 - 1;                              11
  end                                                               12
  % vollziehe  x-Schritt                                           13
  stepsize2 = stepsize2+1;                                          14
end                                                                 15
```

Listing 5.6: Die Verarbeitungsschleife des zweiten Oktant. Das Abspeichern der Koordinaten ist in den Dimensionen aufgetrennt und die Veränderung in x wird in jedem Durchlauf durchgeführt.

5.2.6 Nachverarbeitung

Da das Konzept der Vorberechnung der Schrittzahl zu doppelten Punkten an den Oktantengrenzen führt, ist die Entfernung dieser nach den Schleifen abgetrennt.

Die Verarbeitung ist über eine if-Anweisung erreichbar, da sie vom dritten Parameter des Algorithmus abhängt und damit eine optionale Verarbeitung bietet. Es wird mit Schleifen über die jeweils letzten beiden Koordinaten der Oktanten nach doppelten Punkten gesucht. Dabei werden die Grenzen zwischen dem ersten und zweiten, dritten und vierten, fünften und sechsten sowie siebten und achten Oktant abgeprüft. Sind doppelte Punkte gefunden, wird die Anzahl mitgezählt und in die Variablen *nDouble12*, *nDouble34*, *nDouble56*, *nDouble78* abgespeichert. Anschließend wird die Koordinatenmatrix umkopiert und die doppelten Koordinaten ausgelassen, siehe Listing 5.7. Dabei wird nach Oktantengrenzen und Dimensionen getrennt vorgegangen. Zuerst wird die Grenze zwischen ersten und zweiten Oktant behandelt Dabei wird in den Zeilen 5-6 über die ursprüngliche Schrittzahl n und die Anzahl der doppelten Punkte *nDouble12* die zu kopierende Stelle im Array *coords* abgegriffen. Anschließend muss weiter mitgeführt werden, wieviele Koordinaten ausgelassen wurden, um weiterhin mit den ursprünglichen Schrittzahlen ein lückenloses Füllen der temporären Matrix zu gewährleisten, siehe Zeile 8. Die weitere Verarbeitung folgt dem selben Prinzip. Es wird der nächste Bereich in tmpcoords angesprochen und die Koordinaten umgespeichert, siehe Zeilen 10-21.

Durch das Preallokieren der temporären Matrix und dem Auftrennen der Koordinaten ist der Code frei von jeglicher spezifischen MATLAB-Funktionalität.

Würden die jeweils letzten beiden Punkte eines Oktant an das Ende der Koordinatenmatrix gespeichert werden, wäre die Entfernung in Sprachen wie C durch ein realloc einfacher zu lösen.

```
tmpcoords = zeros(nCoords-nDouble12-nDouble34-nDouble56-nDouble78,2,'int32');
% Schrittanzahl 1.+2. Oktant
n = n1 + n2;
% Umspeichern 1.+2. Oktant
tmpcoords(1:n-nDouble12,1) = coords(1:n-nDouble12,1);
tmpcoords(1:n-nDouble12,2) = coords(1:n-nDouble12,2);
% Anzahl doppelter Punkte
nDouble = nDouble12;
% Umspeichern 7.+8. Oktant
tmpcoords(n-nDouble+1:n+n8+n7-nDouble-nDouble78,1) = coords(n+1:n+n8+n7-nDouble78,1);
tmpcoords(n-nDouble+1:n+n8+n7-nDouble-nDouble78,2) = coords(n+1:n+n8+n7-nDouble78,2);
```

```
n = n + n8 + n7;
nDouble = nDouble + nDouble78;
% Umspeichern 5.+6. Oktant
tmpcoords(n-nDouble+1:n+n6+n5-nDouble-nDouble56,1) = coords(n+1:n+
    n6+n5-nDouble56,1);
tmpcoords(n-nDouble+1:n+n6+n5-nDouble-nDouble56,2) = coords(n+1:n+
    n6+n5-nDouble56,2);
nDouble = nDouble + nDouble56;
n = n + n5 + n6;
% Umspeichern 3.+4. Oktant
tmpcoords(n-nDouble+1:n+n4+n3-nDouble-nDouble34,1) = coords(n+1:n+
    n4+n3-nDouble34,1);
tmpcoords(n-nDouble+1:n+n4+n3-nDouble-nDouble34,2) = coords(n+1:n+
    n4+n3-nDouble34,2);
% Überschreiben der alten Koordinaten
coords = tmpcoords;
```

Listing 5.7: Vorgehen zum Umkopieren der Pixelkoordinaten beim Kreis, um doppelte Punkte auszulassen.

5.2.7 Datentypen

MATLAB arbeitet im numerischen Bereich standardmäßig mit dem Datentyp Double. Im Code sind mehrere Stellen, an denen Ergebnisse über die MATLAB-Funktion *int32()* in Integer umgewandelt werden. Weiterhin werden die Koordinatenmatritzen als Integer preallokiert, siehe Listing 5.5. MATLAB-Verarbeitungen sind mit Double-Variablen schneller, was allerdings eine Besonderheit von MATLAB ist. Andere Sprachen, beispielsweise C, sind bei Integerverarbeitungen schneller. Da der Code langfristig gesehen auf C portiert werden soll, wurde diese Verarbeitung implementiert. Sie soll aufzeigen, welche Variablen als Integer gerechnet werden können und welche als Float bzw. Double. Weiterhin verbraucht ein Integer (32 Bit) nur die Hälfte an Speicher eines Double (64 Bit). Es sind also größere Koordinatenmatritzen durch Integer möglich, ohne den Speicher komplett zu füllen oder zu überlaufen.

Diese Festlegung ist nicht final. Zukünftige Betrachtungen können zeigen, dass weitere Zuweisungen oder Abschnitte ebenfalls in Integer gerechnet werden können.

5.3 Kugel

Der Kugelalgorithmus wurde nach den Konzepten aus den Abschnitten 4.11 und 4.12 entwickelt. Weiterhin wurde das Konzept für die Behandlung des Spezialfall des Radius umgesetzt, in denen der Radius $r = 0$ ist (Abschnitt 4.13). Der Code weist wenig MATLAB-spezifischen Code auf und ist damit leicht auf andere Sprachen portierbar.

5.3.1 Grundaufbau

In Abbildung 5.2 ist der grobe Ablaufplan des implementierten Bresenham-Kreis-Algorithmus aufgezeigt. In den folgenden Abschnitten werden Aspekte aus dem Algorithmus näher erläutert.

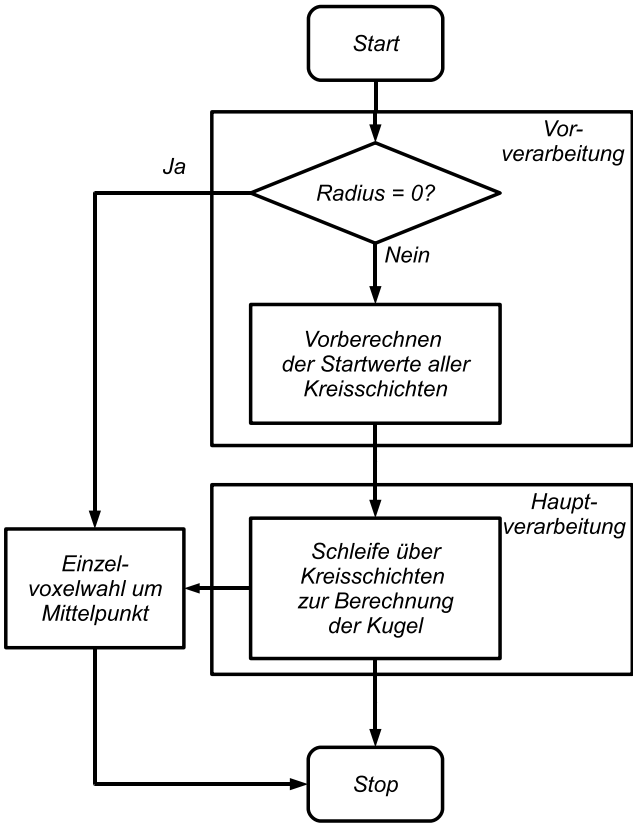

Abbildung 5.2: Abstrakter Programmablaufplan des implementierten Bresenham-Kugel-Algorithmus. *Ist der Radius $r = 0$ wird ein Voxel um den Mittelpunkt gewählt. Ansonsten werden die Startwerte und Schrittzahlen Kreisschicht berechnet. Anschließend folgt eine Schleif über die Schichten, in denen die spezifischen Kreise berechnet werden.*

5.3.2 Einzelvoxelwahl um Mittelpunkt

Auch bei der Kugel wird in der Vorverarbeitung nach Abbildung 5.2 eine Prüfung auf $r = 0$ abgehandelt. Die Wahl des besten Voxel um den Mittelpunkt wird nach Abbildung 4.13 getätigt. Die Funktionalität wurde hier, ähnlich dem Kreis, in eine Unterfunktion *chooseOnePoint* ausgelagert, siehe Listing 5.8. Dieser Schritt wurde getätigt, da das Verfahren der Voxelwahl um einen Punkt ebenfalls bei einer berechneten Gesamtschrittzahl von Null für eine Kreisschicht angewendet werden muss.

```
function coords = chooseOnePoint(centre)
x1 = ceil(centre(1)); x2 = x1 - 1;
y1 = ceil(centre(2)); y2 = y1 - 1;
z1 = ceil(centre(3)); z2 = z1 - 1;
sumerror1 = norm([x1 y1 z1]-centre);
sumerror2 = norm([x1 y2 z1]-centre);
sumerror3 = norm([x2 y2 z1]-centre);
sumerror4 = norm([x2 y1 z1]-centre);
sumerror5 = norm([x1 y1 z2]-centre);
sumerror6 = norm([x1 y2 z2]-centre);
sumerror7 = norm([x2 y2 z2]-centre);
sumerror8 = norm([x2 y1 z2]-centre);
[sumerror i] = min(sumerror1,min(sumerror2,min(sumerror3,min(
   sumerror4,min(sumerror5,min(sumerror6,min(sumerror7,sumerror8)
   )))));
switch i
  case 1
    coords = [x1 y1 z1];
  case 2
    coords = [x1 y2 z1];
  case 3
    coords = [x2 y2 z1];
  case 4
    coords = [x2 y1 z1];
  case 5
    coords = [x1 y1 z2];
  case 6
    coords = [x1 y2 z2];
  case 7
    coords = [x2 y2 z2];
  case 8
```

```
    coords = [x2 y1 z2];                                    30
end                                                         31
end                                                         32
```

Listing 5.8: Die Unterfunktion *chooseOnePoint* im Bresenham-Kugel-Algorithmus. Hier wird die Koordinate um den Mittelpunkt *centre* gewählt, die den minimalen Abstand aufweist.

Die Funktionalität ist nach dem selben Prinzip wie in Abschnitt 5.2.2 umgesetzt. Es werden die acht möglichen Koordinaten um den Mittelpunkt *centre* erstellt. Anschließend wird über norm $([xN\, yN\, zN] - centre)$ die Länge des Abstandsvektor ermittelt. Final wird der minimale Abstand mit entsprechender Koordinate gewählt.

5.3.3 Vorberechnung

Im Listing 5.9 ist ein Teil der Vorberechnung aufgezeigt. Dabei werden zuerst die Grenzpunkte $border1$ und $border2$ in z-Dimension berechnet, siehe Zeilen 7 und 8. Daraus wird die maximale Schrittzahl $nZPos$ bzw. $nZNeg$ für den jeweiligen Bereich der z-Dimension ermittelt. Anschließend wird ein Array *sliceData* mit $(nZPos + nZNeg)$ x 34 Arraystellen preallokiert, siehe Zeile 11. Damit repräsentiert es die möglichen Kreisschichten mit jeweils 34 Werten. Die 34 Werte sind das jeweilige $stepsizeZ$, der spezifische Radius $radiusN$ und $4 \cdot 8$ Werte pro Kreisoktant (bsp.: n1, choosenVoxel1, stepsize1, error1 für ersten Oktant). Es folgt eine Schleife über die Schrittzahl in positiver Richtung $nZPos$. Dabei wird die Winkelverletzung durch $|stepsizeZ| \geq radius$ in Zeile 14 abgeprüft und die Abstandsbetrachtung aus dem Abschnitt 4.12 durchgeführt (Zeile 15). Ist diese Bedingung erfüllt, wird der jeweilige Abstand zur nächsten Rasterkoordinate abgeprüft und bei einem Abstand $\leq 0,5$ die aktuelle Schrittweite als Schicht abgespeichert (sliceData(i,1) = stepsizeZ;). Der Radius $r = 0$ muss an dieser Stelle nicht abgespeichert werden, da das Array mit dem Wert Null gefüllt ist und dementsprechend schon eine Null als Wert vorhanden ist.

Ist der Abstand $> 0,5$ wird ein Dekrementierungswert erhöht und die Schleife abgebrochen, da das Ende der Kugel erreicht ist, siehe Zeile 23 und 24.

Ist die Winkelverletzung nicht gegeben wird der spezifische Radius $radiusN$ berechnet und die Vorberechnung der zugehörigen Kreisschicht über die Funktion *initStartCalc* gestartet, siehe Zeilen 28-31. Die Funktion *initStartCalc* arbei-

tet nach bekanntem Prinzip aus dem Kreisalgorithmus, indem oktantenspezifisch die Funktion *calculateStartvaluesPerOctant* aus dem Abschnitt 5.2.3 aufgerufen wird.

```
% Gleitkommaverschiebung
disloc(3) = mod(abs(centre(3)),floor(abs(centre(3))))*sign(centre(3));
disloc(2) = mod(abs(centre(2)),floor(abs(centre(2))))*sign(centre(2));
disloc(1) = mod(abs(centre(1)),floor(abs(centre(1))))*sign(centre(1));
% floor()+1 entspricht ausschließendem ceil (aufrunden)
stepsizeZ = (floor(disloc(3))+1) - disloc(3);
% Grenzpunkte rechts und links des Mittelpunkt mit Schrittanzahl
border2 = radius+disloc(3); nZPos = int32(ceil(border2)+1);
border1 = radius-disloc(3); nZNeg = int32(ceil(border1)+1);
minus = int32(0);
% Array für Werte der Kreisschichten
sliceData = zeros(nZPos+nZNeg,34);
for i=1:nZPos
% Schleife über Schritte ins positive
  if stepsizeZ >= radius
    if abs(stepsizeZ+disloc(3) - border2) <= 0.5
      % Winkelverletzung und Abstand zum nächsten Rasterpunkt <= 0,5
      % aktuelle Schrittweite abspeichern
      sliceData(i,1) = stepsizeZ;
      % radius = 0 muss nicht gesetzt werden durch zeros
      % sliceData(i,2) = 0;
      % Abbruch
      break
    else
      % wenn Abstandsbedingung nicht stimmt, erhöhe Dekrementierung
      minus = minus + 1;
      break;
    end
  else
    % bei gültigem Winkelverhältnis berechne speziellen Radius
    additionalDelta = radius - sin(acos(stepsizeZ/radius))*radius;
    radiusN=radius-additionalDelta;
    % berechne Startwerte des Kreis
```

```
    sliceData(i,:)= initStartCalc(radiusN,stepsizeZ,disloc(1:2),
        sliceData(i,:));
  end
  stepsizeZ = stepsizeZ + 1;
end
stepsizeZ = floor(disloc(3))-disloc(3);
for j=1:nZNeg
% Schleife über Schritte ins negative
  ...
end
```

Listing 5.9: Ausschnitt aus der Vorberechnung des Bresenham-Kugel-Algorithmus. Es werden die Schritte rechts der y-Achse abgehandelt und anschließend die Schritte links der y-Achse.

5.3.4 Kreisberechnungen

Die Berechnung der Kreisschichten ist im Listing 5.10 aufgezeigt. Dazu wird zunächst die Preallokierung der Koordinatenmatrix nach bekanntem Prinzip durchgeführt (`coords = zeros(sum(steps),3,'int32');`). `sum(steps)` gibt dabei die Summe aller Oktantenschritte und damit die Anzahl der zu berechnenden Koordinaten an. Über die Funktion *callCircle* wird der erste Kreis berechnet. Parameter sind die 34 Werte aus dem *sliceData*-Array. Sie wurden zwecks der Übersichtlichkeit ausgelassen. Da die Kreisfunktion durch die Entfernung doppelter Punkte weniger Koordinaten zurückgeben kann, als vorher angenommen, werden die Koordinaten zwischengespeichert und der Unterschied in *nUnused* abgespeichert. Diese Variable wird später dazu verwendet, die Indizierung der Koordinaten ohne Lücken zu erzeugen durchführen zu können. Anschließend folgt eine Schleife über das Array *steps*, in dem die spezifische Anzahl der Kreiskoordinaten pro Schicht abgespeichert sind. Wenn die Schrittzahl Eins beträgt ist der Radius Null. Hier wird die Funktion *chooseOnePoint* aus Abschnitt 5.3.2 aufgerufen. Ansonsten wird wieder über *callCircle* der entsprechende Kreis berechnet. Die Kreiskoordinaten bestehen weiterhin aus x- und y-Werten. Die zugehörige z-Position wird über `tmpCoords(:,3)= int32(sliceData(i,1)+disloc(3));` in die dritte Dimension eingefügt. sliceData(i,1)+disloc(3) gibt dabei die Rasterposition in z an.
Sind alle Kreise berechnet, werden in Zeile 23 die ungenutzten Array-Stellen entfernt.

```
% Preallokieren der Koordinatenmatrix
coords = zeros(sum(steps),3,'int32');
% Aufruf des ersten Kreis
tmpCoords = callCircle(...);
tmpCoords(:,3) = int32(sliceData(1,1)+disloc(3));
nCoords = size(tmpCoords,1); nUnused = steps(1)-nCoords;
coords(1:nCoords,:) = tmpCoords;
for i=2:size(steps,1)
  if steps(i) == 1
    % wenn Schrittzahl=1 und damit Radius=0
    coords(nCoords+1,:) = chooseOnePoint(disloc);
    coords(nCoords+1,3) = sliceData(i,1)+disloc(3);
    nCoords = nCoords + 1;
  else
    % normaler Kreisaufruf
    tmpCoords = callCircle(...);
    tmpCoords(:,3) = int32(sliceData(i,1)+disloc(3));
    n = int32(size(tmpCoords,1));
    coords(nCoords+1:nCoords+n,:) = tmpCoords;
    nCoords = nCoords + n; nUnused = nUnused + steps(i)-nCoords;
  end
end
coords(nCoords+1:end,:) = [];
```

Listing 5.10: Die Aufrufe der Kreisberechnungen im Bresenham-Kugel-Algorithmus.

5.3.5 Datentypen

Im Code sind auch hier mehrere Stellen, an denen Ergebnisse über die MATLAB-Funktion *int32()* in Integer umgewandelt werden. Die Koordinaten sind dabei, wie beim Kreis, als Integer preallokiert.

Diese Festlegung ist auch in diesem Algorithmus nicht final. Zukünftige Betrachtungen können zeigen, dass weitere Zuweisungen oder Abschnitte ebenfalls in Integer gerechnet werden können.

6 Evaluation

Das folgende Kapitel befasst sich mit dem Aufführen und Interpretieren von Messergebnissen der implementierten Algorithmen. Um eine quantitativ aussagekräftige Aussage führen zu können, wurde ein Brute-Force-Algorithmus implementiert, welcher als Vergleichsgrundlage gilt. Weiterhin wurde getestet, ob die Algorithmen die für das Projekt gestellten Anforderungen innerhalb der Unit-Tests erfüllen. Zur Unterstützung der Unit-Tests wurden White-Box-Tests entworfen, welche aus den Konzepten erstellt wurden und Problemstellen und Konzeptpfade abprüfen. Anschließend findet eine Evaluation bezüglich der Performance statt.

6.1 Testumgebung

Alle in diesem Kapitel aufgeführten Messergebnisse wurden in folgender Messumgebung erstellt:

- Intel Pentium 4 - 3,00 GHz
- Single-Core-System mit Hyperthreading
- 4 GB RAM
- Windows XP Professional 32 Bit Service Pack 3
- MATLAB R2007b 32 Bit

6.2 Brute-Force-Rasterung als Vergleichsreferenz

Um einen vergleichbaren, als Optimum angenommenen, Algorithmus zu haben, wurde ein Rasterung nach Brute-Force-Ansatz umgesetzt. Die Rasterung arbeitet konzeptionell vollständig unabhängig vom Bresenham-Verfahren. Es wird eine Koordinatenmatrix mit Rasterkoordinaten um den zu rasternden Körper generiert.

Die Umgebung ist hierbei quadratisch im zweidimensionalen Bereich und würfelförmig im dreidimensionalen. Für jeden Rasterpunkt in diesem Raum wird der spezifische Abstand zum Rasterkörper ermittelt.

Die Auswahl der Rasterkoordinaten erfolgt nach Wählen des minimalen Abstands zum Körper. Um zu gewährleisten, dass eine Rasterung ähnlich dem Bresenham-Verfahren entsteht, wird die Anzahl der Nachbarn pro Rasterpunkt betrachtet. Ein Nachbar ist eine weitere Rasterkoordinate in den acht umliegenden möglichen Positionen. Die Nachbarn werden außerdem dazu verwendet, die Rasterung abzubrechen.

Im Folgenden werden die Bedingungen der Nachbarn für die getesteten Körper aufgeführt.

Kreis Um einen geschlossenen Kreis mit dem Bresenham-Verfahren zu erzeugen, muss jede Koordinate zwei Nachbarn haben. Dies entspricht dem optimalen Fall. Es kann hier zu einer Abweichung kommen, dass einzelne Koordinaten drei Nachbarn aufweisen. Für den Rasterung nach Brute-Force bedeutet das die Prüfung auf mindestens zwei Nachbarn für jede Koordinate und ein Gewähren von drei Nachbarn, wenn sich die Nachbarverhältnisse der Nachbarn dadurch verbessert, also dem Wert Zwei annähert.

Kugel Eine geschlossene Kugel ist nach dem Bresenham-Verfahren gegeben, wenn jede Koordinate mindestens acht von 26 möglichen besitzt. Auch hier kann es zu Abweichungen in der Bedingung kommen, so dass manche Koordinaten bis zu 11 Nachbarn aufweisen, bedingt durch die zusätzliche z-Dimension. Mehr als acht Nachbarn werden dementsprechend wieder zugelassen, wenn sich die Nachbarverhältnisse der Nachbarn dem Wert Acht annähern.

6.3 Qualität der Rasterung

Für das Projekt wurden anfangs Unit-Tests aufgestellt, welche für alle entwickelten Algorithmen gelten. Diese wurden im Kapitel 2.2.1.2 aus den Projektanforderungen abgeleitet und erläutert. Im Folgenden werden die Unit-Tests noch einmal wiederholt und Testfälle aus den spezifischen Konzepten als White-Box-Tests aufgeführt. Für die Algorithmen werden die zugehörigen Testergebnisse aufgeführt.

Anschließend findet eine Gegenüberstellung des Brute-Force-Ansatz mit den Algorithmen statt, bezogen auf die quantitativen Metriken aus Abschnitt 2.2.1.1.

6.3.1 Unit-Tests

Für das Projekt wurden die folgenden Unit-Tests aufgestellt:

Keine doppelten Punkte Im gerasterten Körper dürfen keine doppelt berechneten Pixel-/Voxelkoordinaten auftreten.

Maximale Abweichung Der maximale orthogonale Abstand in der n-ten Dimension darf die Bedingung $error_{\perp} \leq \left|\pm\frac{\sqrt{n}}{2}\right|$ nicht verletzen.

Gleitkommaeingaben Alle Parameter müssen als reelle Zahl möglich sein und korrekt verarbeitet werden. ($\in \mathbb{Q}$)

Geschlossenheit Der gerasterte Körper darf in der Koordinatenwahl keine Lücken aufweisen. Kreis-Bedingung: Mindestens zwei Nachbarn pro Koordinate. Kugel-Bedingung: Mindestens vier Nachbarn pro Koordinate.

6.3.2 White-Box-Tests

Aus den jeweiligen Konzepten ergeben sich spezifische Testfälle. Diese werden in der Evaluation als Unterstützung der Unit-Tests gehandhabt. Die sogenannten White-Box-Tests prüfen das spezifische Verhalten in möglichen Problemfällen oder alle möglichen Ablaufpfade innerhalb der Algorithmen ab. Bei jedem White-Box-Test werden die Unit-Tests abgeprüft. Nachfolgend werden die Tests pro Algorithmus aufgeführt.

Kreisalgorithmus

Die White-Box-Tests des Kreis-Algorithmus gruppiert, kurz erläutert und numeriert sind:

Pixelwahl am Mittelpunkt:

TestKr01 $r \in \mathbb{Z}$, $P_m \in \mathbb{Z}$; $r = 0$, $P_m\,(-5; 5)$: Koordinatenwahl bei P_m auf Pixel

TestKr02 $r \in \mathbb{Z}$, $P_m \in \mathbb{Q}$; $r = 0$, $P_m\,(-5,1; 5,9)$: Koordinatenwahl 1

TestKr03 $r \in \mathbb{Z}$, $P_m \in \mathbb{Q}$; $r = 0$, $P_m\,(-5,1; 5,1)$: Koordinatenwahl 2

TestKr04 $r \in \mathbb{Z}$, $P_m \in \mathbb{Q}$; $r = 0$, $P_m\,(-5,9; 5,1)$: Koordinatenwahl 3

TestKr05 $r \in \mathbb{Z}$, $P_m \in \mathbb{Q}$; $r = 0$, $P_m\,(-5,9;5,9)$: Koordinatenwahl 4

TestKr06 $r \in \mathbb{Z}$, $P_m \in \mathbb{Q}$; $r = 0$, $P_m\,(-5,5;5,5)$: Koordinatenwahl bei gleichem Abstand aller Rasternachbarn

Negativer Radius

TestKr07 $r \in \mathbb{Z}$, $P_m \in \mathbb{Z}$; $r = -15$, $P_m\,(-5;5)$: Negativer Radius

TestKr08 $r \in \mathbb{Z}$, $P_m \in \mathbb{Q}$; $r = -15,5$, $P_m\,(-5,5;5,5)$: Negativer Radius

Startpixelwahl in langsamer Dimension

TestKr09 $r \in \mathbb{Z}$, $P_m \in \mathbb{Q}$; $r = 15$, $P_m\,(-4,3;5,9)$: Obere Startkoordinate in langsamer Dimension in allen Oktanten

TestKr10 $r \in \mathbb{Z}$, $P_m \in \mathbb{Q}$; $r = 15$, $P_m\,(-4,9;5,1)$: Untere Startkoordinate in langsamer Dimension in allen Oktanten

TestKr11 $r \in \mathbb{Q}$, $P_m \in \mathbb{Z}$; $r = 15,9$, $P_m\,(-5;5)$: Obere Startkoordinate im 1., 2., 3., 8. Oktant, untere Startkoordinate im 4., 5., 6., 7. Oktant

TestKr12 $r \in \mathbb{Q}$, $P_m \in \mathbb{Z}$; $r = 15,4$, $P_m\,(-5;5)$: Obere Startkoordinate im 4., 5., 6., 7. Oktant, untere Startkoordinate im 1., 2., 3., 8. Oktant

TestKr13 $r \in \mathbb{Q}$, $P_m \in \mathbb{Q}$; $r = 15,5$, $P_m\,(5,1;5,1)$: Obere Startkoordinate in allen Oktanten

TestKr14 $r \in \mathbb{Q}$, $P_m \in \mathbb{Q}$; $r = 15,5$, $P_m\,(4,9;4,9)$: Untere Startkoordinate in allen Oktanten

Radius kleiner Eins (Auslassen von Oktanten)

TestKr15 $r \in \mathbb{Q}$, $P_m \in \mathbb{Z}$; $r = 0,8$, $P_m\,(5;5)$: 2., 8., 7.,5. Oktant ausgelassen

TestKr16 $r \in \mathbb{Q}$, $P_m \in \mathbb{Z}$, $r = 0,5$, $P_m\,(5;5)$: 2., 8., 7.,5. Oktant ausgelassen

TestKr17 $r \in \mathbb{Q}$, $P_m \in \mathbb{Z}$; $r = 0,2$, $P_m\,(5;5)$: 2., 8., 7.,5. Oktant ausgelassen

TestKr18 $r \in \mathbb{Q}$, $P_m \in \mathbb{Q}$; $r = 0,8$, $P_m\,(5,1;4,9)$: 2., 8., 7.,5. Oktant ausgelassen

TestKr19 $r \in \mathbb{Q}$, $P_m \in \mathbb{Q}$; $r = 0,5$, $P_m\,(4,9;5,1)$: 1., 3., 4., 6. Oktant ausgelassen

TestKr20 $r \in \mathbb{Q}$, $P_m \in \mathbb{Q}$; $r = 0,2$, $P_m\,(5,3;5,3)$: alle Oktanten ausgelassen

Kugelalgorithmus

Im Kugelalgorithmus wurden die Kreiskonzepte mit ihrem entsprechenden Code unverändert wiederverwendet. Weiterhin werden mit dem Konzept Kreise in die z-Position eingesetzt. Mit diesem Vorgehen ergibt sich ein unverändertes Verwenden des Kreisalgorithmus. Da dieser bereits umfangreich evaluiert wurde, wird im Folgenden nur auf die kugelspezifischen White-Box-Tests eingegangen.

Die White-Box-Tests der Kugel sind:

Voxelwahl am Mittelpunkt:

TestKu01 $r \in \mathbb{Z}$, $P_m \in \mathbb{Z}$: $r = 0$, $P_m\,(5;5;5)$: Koordinatenwahl bei P_m auf Voxel

TestKu02 $r \in \mathbb{Z}$, $P_m \in \mathbb{Q}$; $r = 0$, $P_m\,(5,9;5,9;5,9)$: Koordinatenwahl 1

TestKu03 $r \in \mathbb{Z}$, $P_m \in \mathbb{Q}$; $r = 0$, $P_m\,(5,9;5,1;5,9)$: Koordinatenwahl 2

TestKu04 $r \in \mathbb{Z}$, $P_m \in \mathbb{Q}$; $r = 0$, $P_m\,(5,1;5,1;5,9)$: Koordinatenwahl 3

TestKu05 $r \in \mathbb{Z}$, $P_m \in \mathbb{Q}$; $r = 0$, $P_m\,(5,1;5,9;5,9)$: Koordinatenwahl 4

TestKu06 $r \in \mathbb{Z}$, $P_m \in \mathbb{Q}$; $r = 0$, $P_m\,(5,9;5,9;5,1)$: Koordinatenwahl 5

TestKu07 $r \in \mathbb{Z}$, $P_m \in \mathbb{Q}$; $r = 0$, $P_m\,(5,9;5,1;5,1)$: Koordinatenwahl 6

TestKu08 $r \in \mathbb{Z}$, $P_m \in \mathbb{Q}$; $r = 0$, $P_m\,(5,1;5,1;5,1)$: Koordinatenwahl 7

TestKu09 $r \in \mathbb{Z}$, $P_m \in \mathbb{Q}$; $r = 0$, $P_m\,(5,1;5,9;5,1)$: Koordinatenwahl 8

TestKu10 $r \in \mathbb{Z}$, $P_m \in \mathbb{Q}$; $r = 0$, $P_m\,(5,5;5,5;5,5)$: Koordinatenwahl bei gleichem Abstand aller Rasternachbarn

Grenzfallbetrachtung:

TestKu11 $r \in \mathbb{Z}$, $P_m \in \mathbb{Z}$; $r = 4$, $P_m\,(-5;5;5)$: $d_z1 = d_z2 = 0$

TestKu12 $r \in \mathbb{Q}$, $P_m \in \mathbb{Q}$; $r = 4,2$, $P_m\,(-5,5;5,5;5,5)$: $|d_z1| < 0,5$, $|d_z2| < 0,5$

TestKu13 $r \in \mathbb{Q}$, $P_m \in \mathbb{Q}$; $r = 3,8$, $P_m\,(-5,5;5,5;5,5)$: $|d_z1| > 0,5$, $|d_z2| > 0,5$

6.3.3 Ergebnisse aus Unit- und White-Box-Tests

Im Folgenden werden die Ergebnisse aus den White-Box-Tests aufgezeigt. Dabei wird auch Bezug zu den Messungen der Unit-Tests genommen.

6.3.3.1 Kreis

Die Unit-Test-Ergebnisse aus den White-Box-Tests des Kreises sind in Tabelle 6.1 aufgeführt. Der Unit-Test *Keine doppelten Punkte* ist in allen Testfällen erfüllt. Bei den orthogonalen Fehlern $error_\perp$ konnte die Vorgabe nach $error_\perp \leq \left|\pm\frac{\sqrt{2}}{2}\right| \approx 0,707$ Pixel aus Abschnitt 2.1 in zwei Fällen nicht erfüllt werden. Bei beiden ist $r < 1$. Bezüglich der Geschlossenheit konnten in allen Fällen außer TestKr17 positive Ergebnisse erzielt werden. Der Fall von keinen Nachbarn in den Tests TestKr01-06 und 20 gilt als geschlossen, da in diesen Fällen jeweils eine Koordinate gewählt wurde, welche als geschlossene Rasterung gilt.

Testnummer	**maximale Abweichung in Pixel**	**Anzahl N doppelte Punkte**	**Anzahl N eliminierte Punkte**	**Geschlossen durch Nachbarn**
TestKr01	0,707	0	0	Ja
TestKr02	0,141	0	0	Ja
TestKr03	0,141	0	0	Ja
TestKr04	0,141	0	0	Ja
TestKr05	0,141	0	0	Ja
TestKr06	0,707	0	0	Ja
TestKr07	0,440	4	4	Ja
TestKr08	0,491	8	8	Ja
TestKr09	0,456	3	3	Ja
TestKr10	0,454	4	4	Ja
TestKr11	0,379	8	8	Ja
TestKr12	0,411	0	0	Ja
TestKr13	0,479	4	4	Ja
TestKr14	0,479	4	4	Ja
TestKr15	0,200	2	2	Ja
TestKr16	**0,914**	2	2	Ja
TestKr17	**1,214**	3	3	**Nein**
TestKr18	0,304	1	1	Ja
TestKr19	0,304	1	1	Ja
TestKr20	0,424	0	0	Ja

Tabelle 6.1: Ergebnisse der Unit-Tests des Kreis-Algorithmus.

Einen Überblick über den Erfüllungsgrad bezüglich den Erwartungen in den White-Box- und Unit-Tests zeigt die Tabelle 6.2. Die Erwartungen konnten in allen Fällen erfüllt werden.

Testnummer	White-Box- ...Tests erfüllt?	Unit-	Quelle des Fehlers
TestKr01	Ja	Ja	-
TestKr02	Ja	Ja	-
TestKr03	Ja	Ja	-
TestKr04	Ja	Ja	-
TestKr05	Ja	Ja	-
TestKr06	Ja	Ja	-
TestKr07	Ja	Ja	-
TestKr08	Ja	Ja	-
TestKr09	Ja	Ja	-
TestKr10	Ja	Ja	-
TestKr11	Ja	Ja	-
TestKr12	Ja	Ja	-
TestKr13	Ja	Ja	-
TestKr14	Ja	Ja	-
TestKr15	Ja	Ja	-
TestKr16	Ja	**Nein**	Maximale Abweichung
TestKr17	Ja	**Nein**	Maximale Abweichung Geschlossenheit
TestKr18	Ja	Ja	-
TestKr19	Ja	Ja	-
TestKr20	Ja	Ja	-

Tabelle 6.2: Übersicht über den Erfüllungsgrad des Kreis-Algorithmus bei den Tests.

6.3.3.2 Kugel

Die Unit-Test-Ergebnisse aus den White-Box-Tests der Kugel sind in Tabelle 6.3 aufgeführt. Die maximale Abweichung liegt in allen Testfällen unter der geforderten Grenze von $\frac{\sqrt{3}}{2} \approx 0,866$ Voxel, siehe Abschnitt2.1. Die Prüfung auf doppelte Punkte ergab ein erfolgreiches Entfernen aller dieser Punkte in allen Testfällen.

Testnummer	maximale Abweichung in Pixel	Anzahl N doppelte Punkte	Anzahl N eliminierte Punkte	Geschlossen durch Nachbarn
TestKu01	0,000	0	0	Ja
TestKu02	0,173	0	0	Ja
TestKu03	0,173	0	0	Ja
TestKu04	0,173	0	0	Ja
TestKu05	0,173	0	0	Ja
TestKu06	0,173	0	0	Ja
TestKu07	0,173	0	0	Ja
TestKu08	0,173	0	0	Ja
TestKu09	0,173	0	0	Ja
TestKu10	0,866	0	0	Ja
TestKu11	0,394	105	105	Ja
TestKu12	0,359	136	136	Ja
TestKu13	0,755	116	116	Ja

Tabelle 6.3: Ergebnisse der Unit-Tests des Kugel-Algorithmus.

Einen Überblick über den Erfüllungsgrad des Algorithmus bezüglich den Erwartungen in den White-Box-Tests und den Unit-Tests zeigt die Tabelle 6.4. Die Erwartungen der White-Box-Tests konnten in allen Fällen erfüllt werden, die Abweichungen in den Unit-Tests sind auch hier noch einmal aufgeführt.

Testnummer	White-Box-...Tests erfüllt?	Unit-...Tests erfüllt?	Quelle des Fehlers
TestKr01	Ja	Ja	-
TestKr02	Ja	Ja	-
TestKr03	Ja	Ja	-
TestKr04	Ja	Ja	-
TestKr05	Ja	Ja	-
TestKr06	Ja	Ja	-
TestKr07	Ja	Ja	-
TestKr08	Ja	Ja	-
TestKr09	Ja	Ja	-
TestKr10	Ja	Ja	-
TestKr11	Ja	Ja	-
TestKr12	Ja	Ja	-
TestKr13	Ja	Ja	-

Tabelle 6.4: Übersicht über den Erfüllungsgrad des Kugel-Algorithmus bei den White-Box- und Unit-Tests.

6.3.4 Vergleich entgegen Rasterung mit Brute-Force

Zur Qualitätsüberprüfung der Algorithmen werden sie mit Hilfe der quantitativen Metriken entgegen dem Brute-Force-Ansatz verglichen. Dabei werden vier Metriken verwendet, welche in Abschnitt 2.2.1.1 eingeführt worden sind.

6.3.4.1 Verwendete Metriken

Die zu vergleichenden Metriken werden im Folgenden noch einmal aufgelistet. Der maximale Fehler dient hierbei dazu, die Rasterannäherung der Rasterpunkte zu prüfen. Die verschiedenen Mittelwerte sollen die Gesamtannäherung der Kurve beschreiben. Der Median gibt die Mitte der Werte an und bestimmt dabei einen gegen Ausreißer robusten Durchschnitt der Werte. Der arithmetische Mittelwert gibt ebenfalls den Durchschnitt der Werte an. Dabei haben alle Werte die selbe Gewichtung und Ausreißer haben demnach den gleichen Einfluss wie alle anderen Koordinaten. Der quadratische Mittelwert maximiert, entgegengesetzt dem Median, die Auswirkungen von großen Abweichungen auf den Durchschnitt. Es wurde sich für diese drei Mittelwerte entschieden, um einen umfangreichen Vergleich durchführen zu können.
Als Metriken der Performance wurde im zwei-Dimensionalen Pixel/sec und im drei-Dimensionalen Voxel/sec gewählt. Darüber lässt sich aussagen, wie sich die Laufzeit zur Komplexität der Rasterung verhält.
Die Metrik-Ergebnisse des Bresenham-Verfahrens sollten sich den Ergebnissen der Brute-Force-Rasterung annähern. Dabei sollten die Werte der Brute-Force-Rasterung kleiner sein, da dieser als Optimum angenommen wird. Diese Annahme ist dadurch begründbar, dass das Bresenham-Verfahren sich nicht nach der Lage einzelner Positionen orientiert, sondern über einen aufsummierten Fehler die Rasterung der Kurve beachtet. Damit erreicht das Bresenham-Verfahren eine lockerere an den Kreis anliegende Rasterung als der Ansatz nach Brute-Force. Allerdings ist eine resultierende Eigenschaft des Bresenham-Verfahrens eine einheitlich minimale Dicke der Linie. Diese Eigenschaft erreicht die Rasterung nach Brute-Force-Prinzip nicht in allen Fällen.

Quantitative Qualitätsmetriken:

Maximaler Fehler $error_{\perp max} = |\max(error_1, error_2, \ldots, error_n)|$

Median $\overline{error}_{med} = \begin{cases} \left|error_{\frac{n+1}{2}}\right| & n\,ungerade \\ \left|\frac{1}{2}\left(error_{\frac{n}{2}} + error_{\frac{n}{2}+1}\right)\right| & n\,gerade \end{cases}$

Arithmetischer Mittelwert $\overline{error}_{arithm} = \left|\frac{1}{n}\sum_{i=1}^{n} \mathrm{error}_i\right|$

Quadratischer Mittelwert $\overline{error}_{quadr} = \sqrt{\frac{1}{n}\sum_{i=1}^{n} error_i^2}$

Metriken der Performance:

Performance für

zwei-dimensionales Bild $Performance_{2D} = Pixel/sec$

drei-dimensionales Volumen $Performance_{3D} = Voxel/sec$

6.3.4.2 Ergebnisse

Im Folgenden werden der Kreis- und Kugel-Algorithmus dem Brute-Force-Ansatz anhand der Metriken gegenübergestellt.

Kreis

Im Folgenden wird zuerst der maximale, anschließend der mediane Mittelwert der Abweichungen betrachtet. Es folgen der arithmetische und quadratische Mittelwert. Die numerischen Werte sind im Anhang F und G als Tabellen aufgelistet.

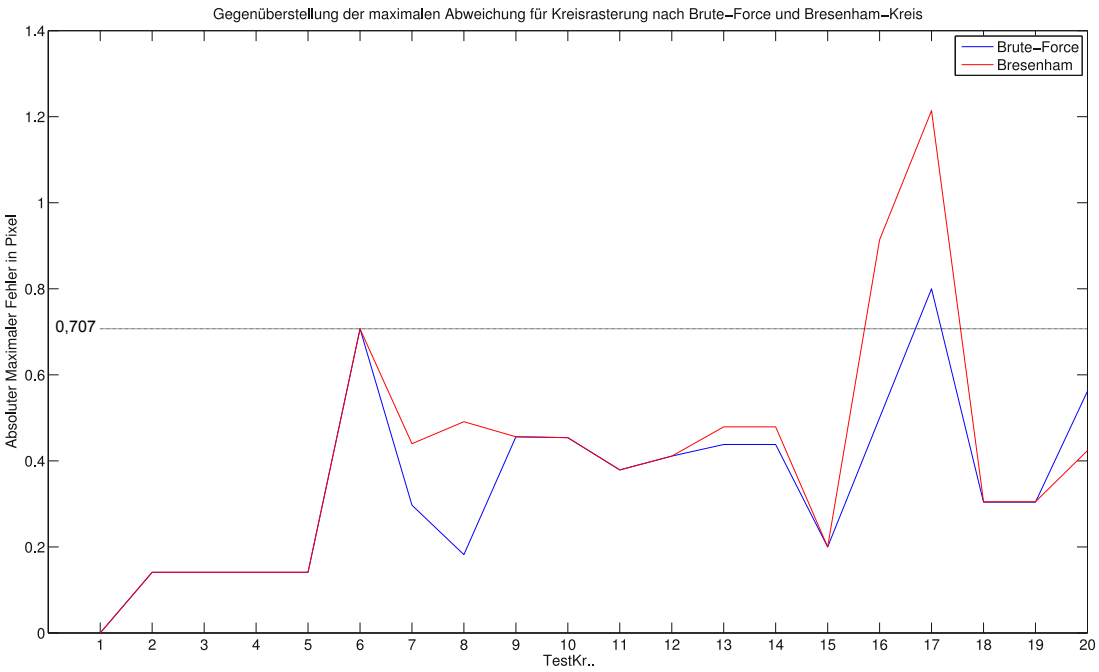

Abbildung 6.1: Gegenüberstellung des Brute-Force-Kreisrasterung und des Bresenham-Kreis anhand des maximalen Fehlers. *Der Brute-Force-Ansatz (blau) verstößt im Testfall TestKr17 gegen den Unit-Test* $error_{\perp max} \leq \frac{\sqrt{2}}{2} = 0,707\,Pixel$. *Der Bresenham-Kreis (rot) im Testfall TestKr16 und TestKr17.*

Die maximalen Fehler der beiden Ansätze sind in der Abbildung 6.1 gegenübergestellt. Dabei ist die Grenze von $\frac{\sqrt{2}}{2} = 0,707$ Pixel ebenfalls eingezeichnet. Beide Algorithmen erreichen in fast allen Fällen eine maximale Abweichung $error \leq \frac{\sqrt{2}}{2}$ Pixel. Das zeugt von einer guten Rasterung in beiden Algorithmen. Die Ausreißer in den Tests TestKr16 und TestKr17 sind damit begründbar, dass in diesen Testfällen der Radius $r < 1$ ist. Wie in Abschnitt 6.3.3.1 erläutert, erzeugen die Algorithmen hier Fehler. Weiterhin zeigt sich, dass der Bresenham-Kreis in fast allen Fällen über oder auf der maximalen Abweichung der Kreisrasterung nach Brute-Force liegt. Das bestätigt die Erwartung, dass die Brute-Force-Kreisrasterung ein Optimum für die meisten Fälle darstellt. Der Testfall TestKr20 ist eine Ausnahme, da hier alle Oktanten bei einem Radius $r > 0$ auf einem Pixel zusammenfallen. Der Bresenham-Kreis wählt einen einzelnen Punkt mit der besten Annäherung zum Mittelpunkt. Die Rasterung nach Brute-Force-Verfahren ist allerdings nicht in der Lage, bei einem Radius $r > 0$ einen einzelnen Punkt zu wählen. Die Nachbar-Prüfung funktioniert nur für mehrere Koordinaten.

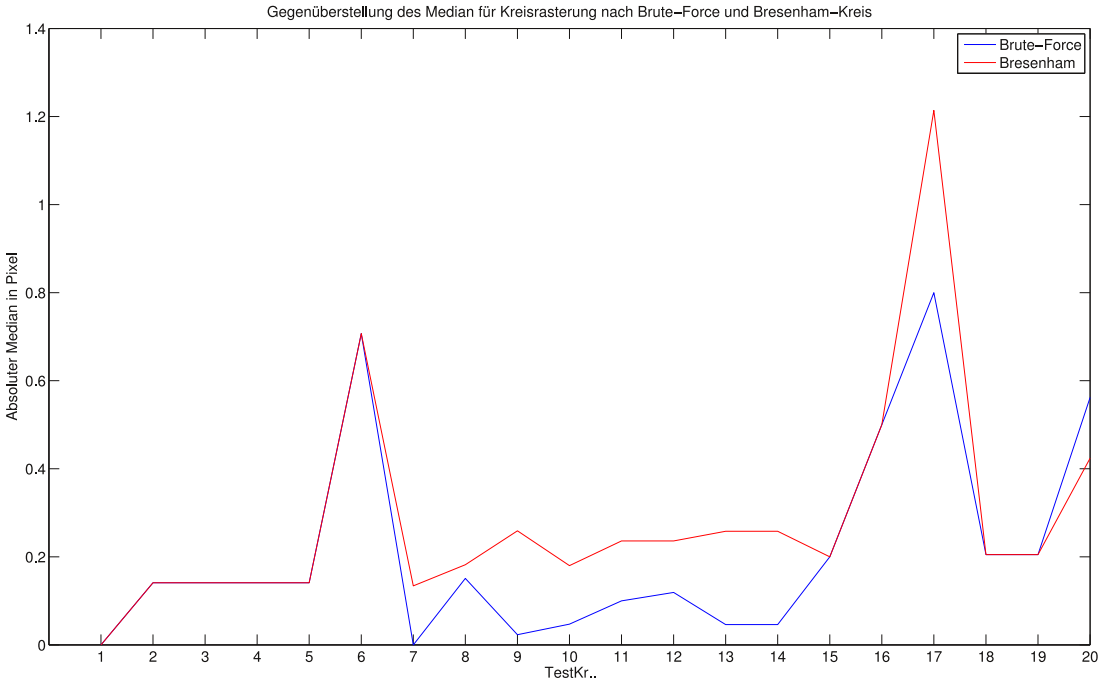

Abbildung 6.2: Gegenüberstellung des Brute-Force-Kreisrasterung und des Bresenham-Kreis anhand des Median der Fehler. *Bei Betrachtung des Median der Abweichungen zeigt sich ein ähnliches Bild, wie beim maximalen Fehler. Der Brute-Force-Ansatz liegt in allen Testfällen außer TestKr20 bei einem geringeren Fehler.*

Der Median $\overline{error}_{med}$ zeigt ein ähnliches Ergebnis, siehe Abbildung 6.2. Auch hier liegt der Bresenham-Kreis auf oder kurz vor dem Optimum der Brute-Force-Rasterung. Diese Eigenschaft wird im Testfall TestKr20 aus den oben erwähnten Gründen gebrochen.

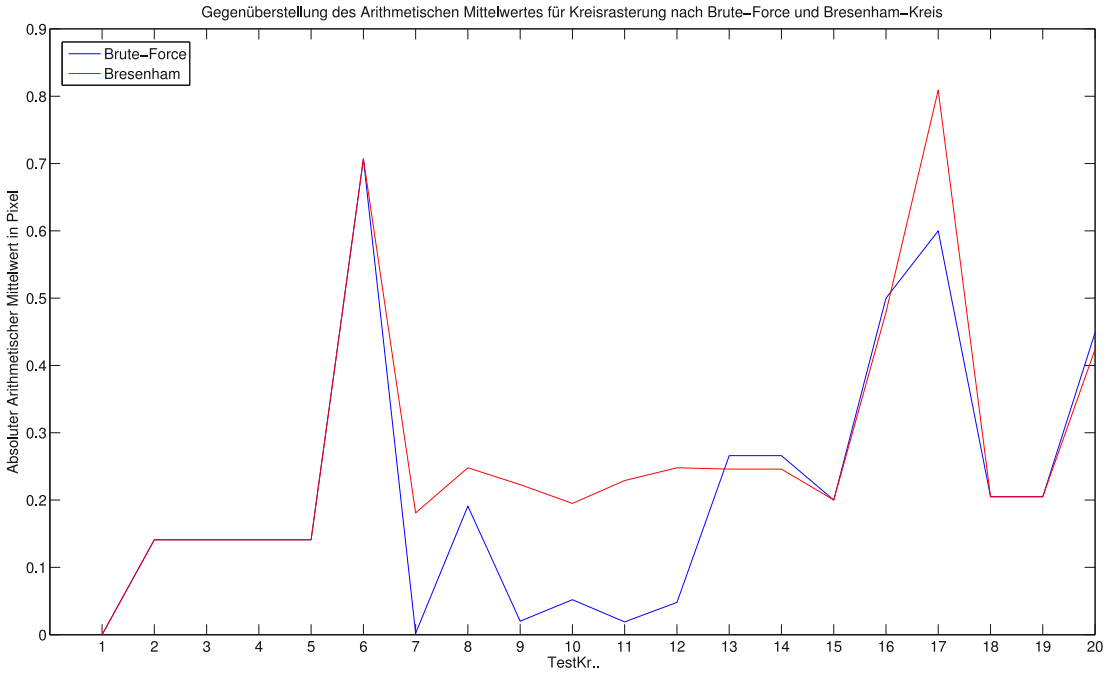

Abbildung 6.3: Gegenüberstellung des Brute-Force-Kreisrasterung und des Bresenham-Kreis anhand des arithmetischen Mittelwert der Fehler. *Der arithmetische Mittelwert der Fehler liegt für den Bresenham-Kreis in fast allen Fällen über dem des Brute-Force-Ansatzes. Ausnahmen sind TestKr13, TestKr14, TestKr16 und TestKr20.*

Ein Vergleich der arithmetischen Mittelwerte ist in Abbildung 6.3 abgebildet. Der arithmetische Mittelwert $\overline{error}_{arithm}$ des Kreis-Algorithmus erreicht in vier Fällen (TestKr13, TestKr14, TestKr16, TestKr20) einen niedrigeren Wert als die Kreisrasterung nach Brute-Force. Das ist sehr wahrscheinlich dadurch bedingt, das eine unterschiedliche Anzahl an Pixel bei beiden Algorithmen berechnet wurden. Der Wert für den TestKr20 ist wie im vorherigen Abschnitt begründbar. Ansonsten nähert er sich dem Optimum an und erreicht es teilweise. Bei hinreichend großen Pixelzahlen läuft der arithmetische Mittelwert gegen den Wert Null.

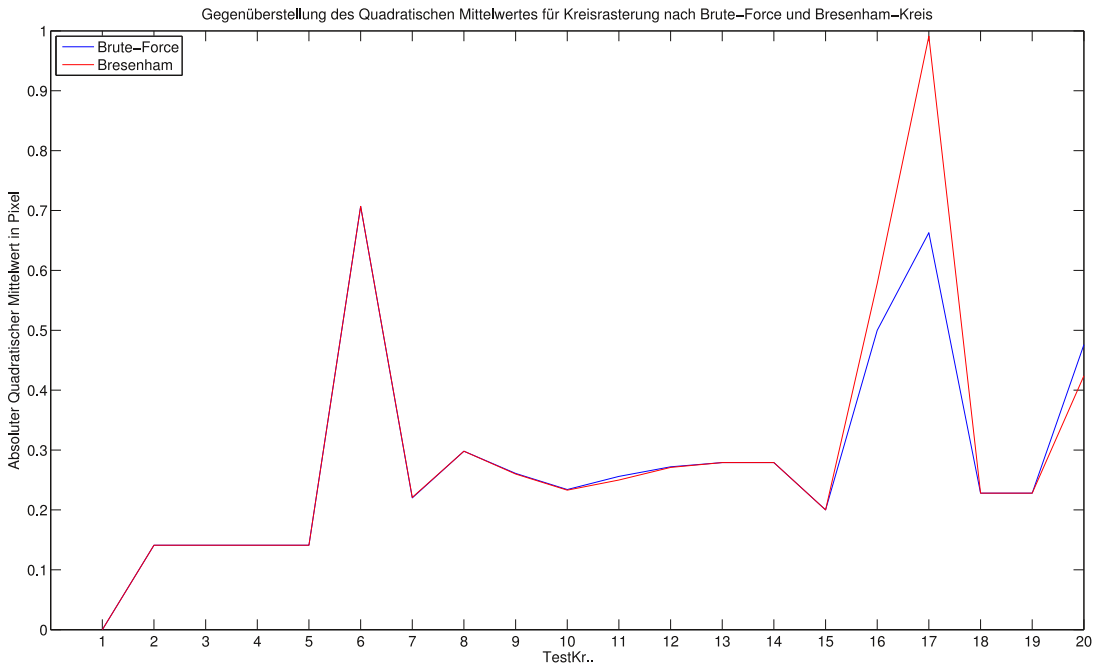

Abbildung 6.4: Gegenüberstellung des Brute-Force-Kreisrasterung und des Bresenham-Kreis anhand des quadratischen Mittelwert der Fehler. *Der quadratische Mittelwert der Fehler liegt für beide Algorithmen ungefähr gleich, außer bei TestKr16, TestKr17 und TestKr20.*

Der quadratische Mittelwert $\overline{error}_{quadr}$ liegt in der ersten Kommastelle bei beiden Algorithmen gleich, siehe Abbildung 6.4. Hier bestehen nur minimale Unterschiede. Ausnahme ist hier der Testfall TestKr17, bei dem der Bresenham-Algorithmus einen zu großen maximalen Abstand erreicht.

Kugel

Auf den nachfolgenden Seiten sind die Ergebnisse der Metriken der Kugelrasterung nach Brute-Force-Prinzip und Bresenham-Kugel-Ansatz in Graphen dargestellt und erläutert. Dabei wird wieder zuerst der maximale, anschließend der mediane Mittelwert der Abweichungen betrachtet. Es folgen auch hier der arithmetische und quadratische Mittelwert. Die konkreten Werte sind auch hier im Anhang H und I als Tabellen aufgelistet.

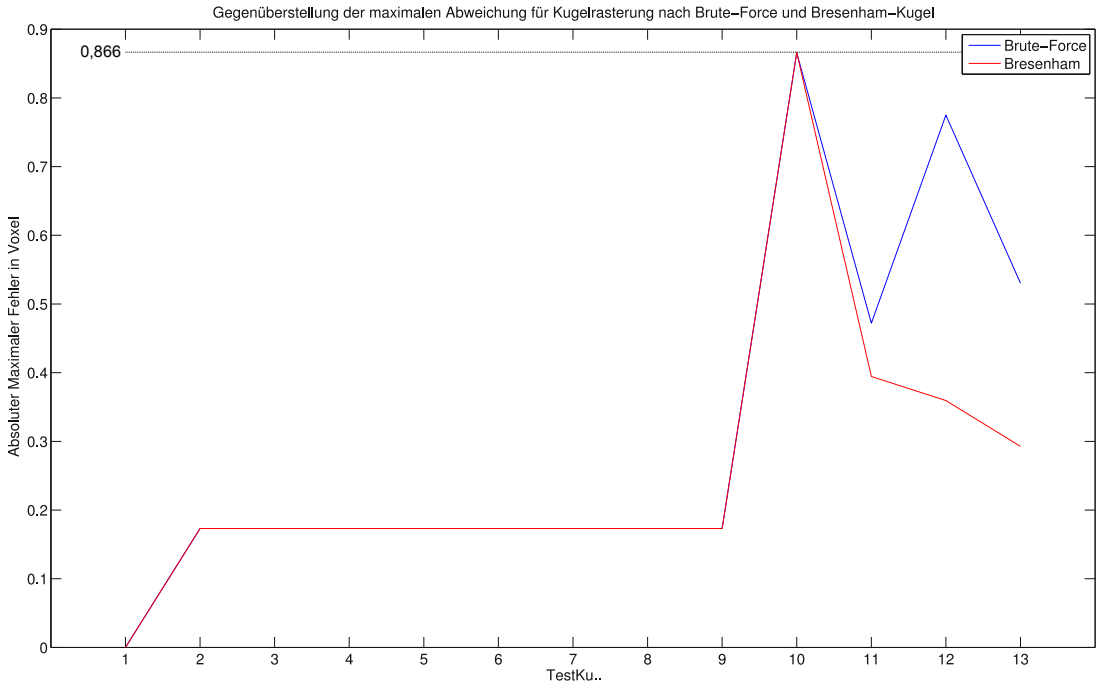

Abbildung 6.5: Gegenüberstellung des Brute-Force-Kugelrasterung und der Bresenham-Kugel anhand des maximalen Fehlers. *Beide Algorithmen verstoßen nicht gegen den Unit-Test* $error_{\perp max} \leq \frac{\sqrt{3}}{2} = 0,866\,Voxel$. *Der Brute-Force-Ansatz (blau) weist in allen Tests eine höhere maximale Abweichung entgegen der Bresenham-Kugel (rot) auf.*

Bei Betrachtung des maximalen Fehlers in Abbildung 6.5 zeigt sich, dass die Brute-Force-Kugelrasterung im Test TestKu11 eine höhere maximale Abweichung aufweist als die Bresenham-Kugel. Beide Algorithmen erreichen einmalig die Unit-Test-Grenze $\frac{\sqrt{3}}{2} = 0,866$ Voxel für die Abweichung. Sie rastern damit nachweislich den Körper mit einer Auswahl der bestmöglichen Voxel. Allerdings zeigen sich Unterschiede in den Tests TestKu11, TestKu12 und TestKu13. Diese Abweichungen unter den Algorithmen kommen wahrscheinlich durch die unterschiedlichen Rasterkonzepte zustande. Der Brute-Force-Ansatz wählt nach direkten Abständen und über Nachbarprüfung. Die Bresenham-Kugel wird über Kreisschichten in z-Dimension erzeugt. Es kann, nach der Abbildung, gesagt werden, dass das Verfahren für die Bresenham-Kugel bessere Rasterungsergebnisse liefert.

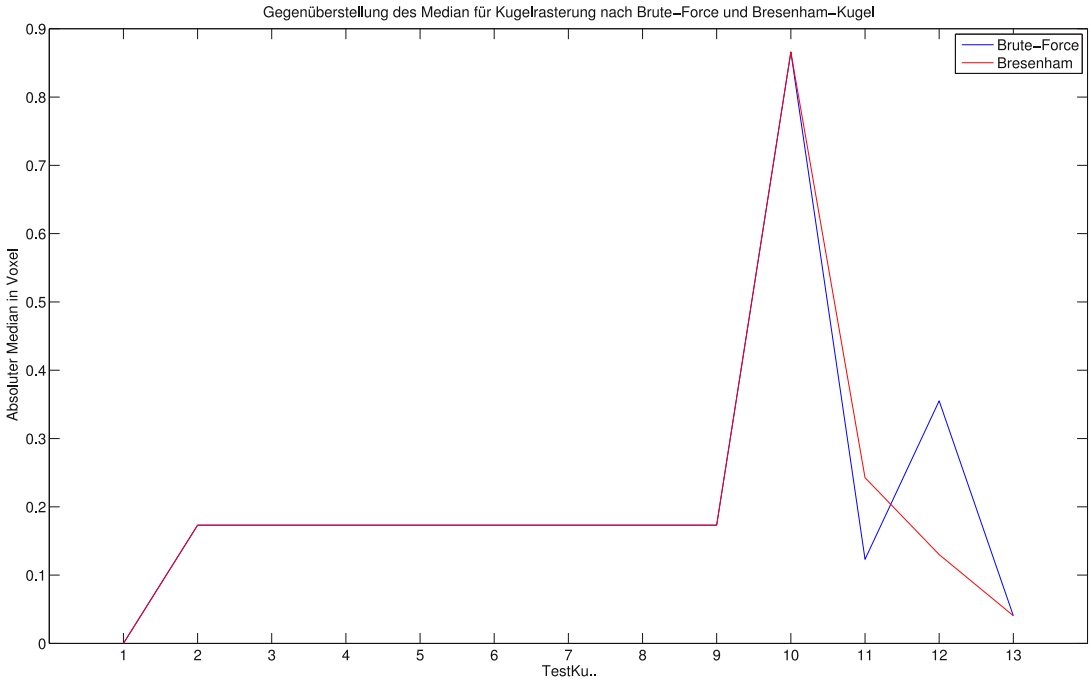

Abbildung 6.6: Gegenüberstellung des Brute-Force-Kugelrasterung und der Bresenham-Kugel anhand des medianen Fehlers. *Die mediane Abweichung der Brute-Force-Rasterung liegt in allen Fällen, außer bei TestKu12 bei einem niedrigeren oder gleichen Wert entgegen der Bresenham-Kugel.*

In Abbildung 6.6 sind die medianen Abweichungen der beiden Algorithmen gegenübergestellt. Dabei zeigt sich, dass der Median der Rasterung nach Brute-Force insgesamt niedrigere Werte aufweist. Der Ausreißer in TestKu12 ist bedingt durch den großen Unterschied in der maximalen Abweichung in diesem Testfall.

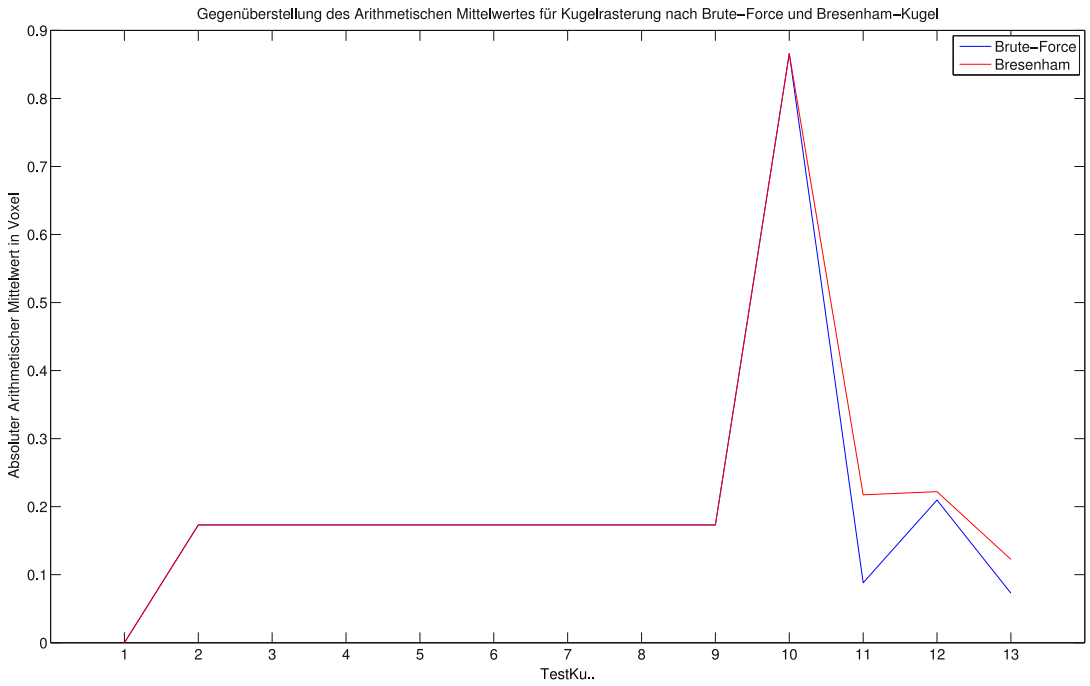

Abbildung 6.7: Gegenüberstellung des Brute-Force-Kugelrasterung und der Bresenham-Kugel anhand des arithmetischen Mittelwertes der Abweichungen. *Es zeigen sich niedrigere Abweichungen durch den Brute-Force-Ansatz. Die Bresenham-Kugel nähert sich den Werten an.*

Bei Betrachtung des arithmetischen Mittelwertes in Abbildung 6.7 zeigt als erste Metrik einen erwarteten Verlauf. Die Brute-Force-Kugelrasterung liegt in allen Fällen unter oder auf den Mittelwerten der Bresenham-Kugel und deutet auf eine bessere Rasterung als die Kugel nach Bresenham-Verfahren hin.

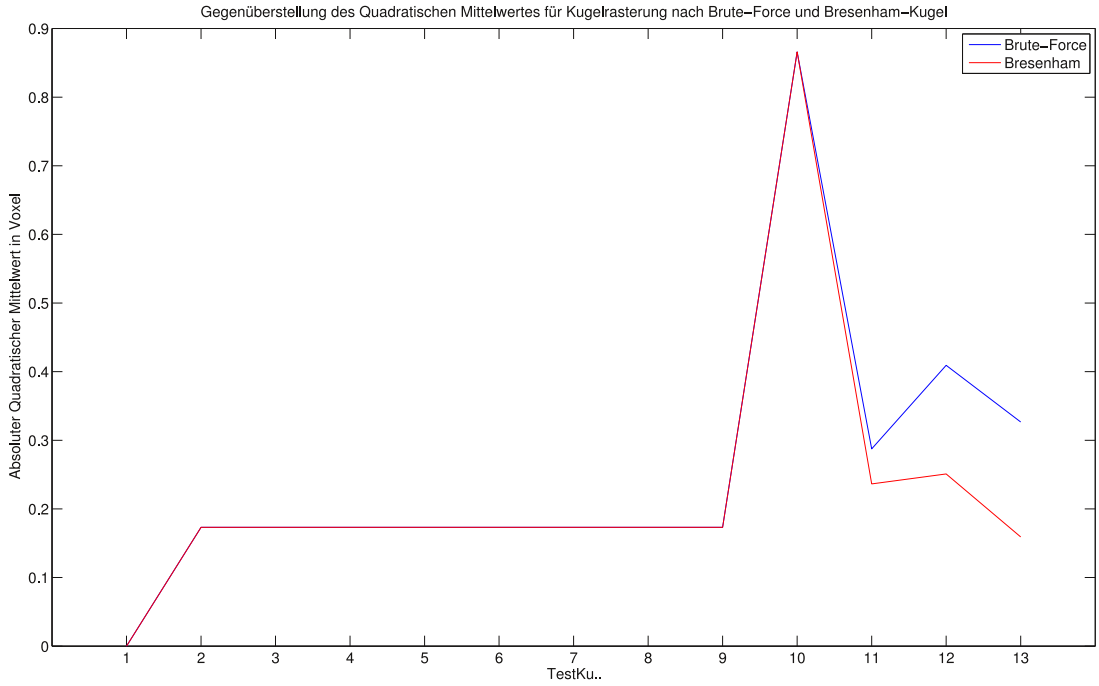

Abbildung 6.8: Gegenüberstellung des Brute-Force-Kugelrasterung und der Bresenham-Kugel anhand des quadratischen Mittelwertes der Abweichungen. *Im Vergleich zu den arithmetischen Mittelwerten dreht sich das Verhältnis für den quadratischen Mittelwert um. Die Bresenham-Kugel erreicht niedrigere Werte als der Brute-Force-Ansatz.*

Bei Betrachtung des quadratischen Mittelwertes der Abweichungen in Abbildung 6.8 zeigt sich ein invertiertes Ergebnis der arithmetischen Mittelwerte. Die Brute-Force-Rasterung weist größere Werte auf als die Rasterung nach dem Bresenham-Verfahren.

6.4 Performance

Der folgende Abschnitt befasst sich mit der Performance der Algorithmen in Pixel/sec beziehungsweise Voxel/sec. Dabei werden die Bresenham-Algorithmen ebenfalls mit dem Brute-Force-Ansatz verglichen. Für den Kreis wird außerdem ein Vergleich zur Ganzzahlvariante nach dem Bresenham-Verfahren, siehe Abbildung 3.6, durchgeführt.

6.4.1 Algorithmusspezifisch

Dieser Abschnitt befasst sich mit den spezifischen Laufzeitverhältnissen der implementierten Bresenham-Kurven-Algorithmen. Dabei wurde die Auslöschung der doppelten Punkte nicht aktiviert.

6.4.1.1 Kreis

Abbildung 6.9 zeigt das Leistungsverhalten des entwickelten Bresenham-Kreis-Algorithmus. Es wird darin deutlich, dass der Algorithmus sich für hinreichend große Werte der Leistung von 12 MPixel/sec annähert. Eine Abschätzung des algorithmischen Aufwands nach der Landau-Notation ergibt für die Koordinatenberechnung einen Aufwand von O(n). Diese Abschätzung kommt durch die Schleifen zur Koordinatenberechnung zustande und wird durch den Test für große Werte bestätigt. Die Abweichungen zu Beginn sind durch Einflüsse der Vorverarbeitung zu erklären. Diese fallen später nicht mehr ins Gewicht.

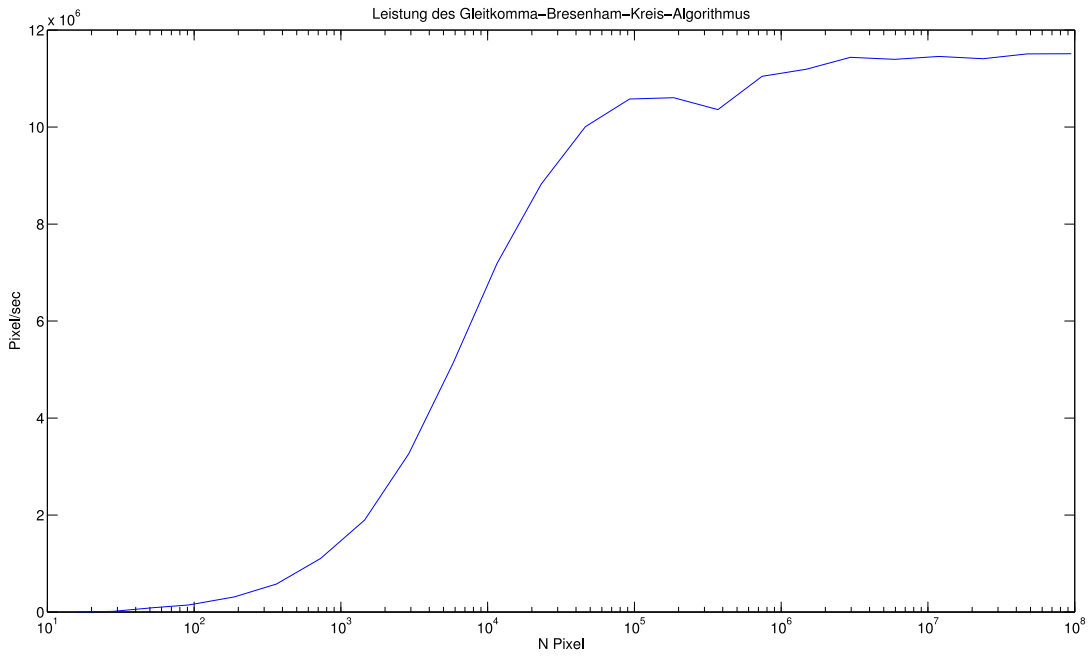

Abbildung 6.9: Leistung des Bresenham-Kreis-Algorithmus in Pixel/sec. *Die x-Achse listet die entsprechende Anzahl berechneter Pixel auf. Auf der y-Achse ist über die Laufzeit des Algorithmus die Leistung in* Pixel/sec *aufgetragen. Der Bresenham-Kreis-Algorithmus nähert sich mit hinreichend großen Werten einer Leistung von* $12\,MPixel/sec$ *an.*

6.4.1.2 Kugel

In der Abbildung 6.10 ist ein Leistungsprofil des Bresenham-Kugel-Algorithmus abgebildet. Dabei zeigt sich, dass der Algorithmus bei $N > 10^7$ Voxel bis zu $3,5$ MVoxel/sec erreicht. Bei einer Abschätzung nach der Landau-Notation wird auch hier ein Aufwand von O(n) geschätzt. Dieser lineare Aufwand wird für Größen von $N > 1,5 \cdot 10^7$ Voxel wahrscheinlich erreicht. Dabei verhindert die aufwändige Vorverarbeitung in Form von Radienberechnungen die frühzeitige Sättigung. Eine weitergehende Laufzeitbetrachtung war aus speichertechnischen Gründen nicht möglich.

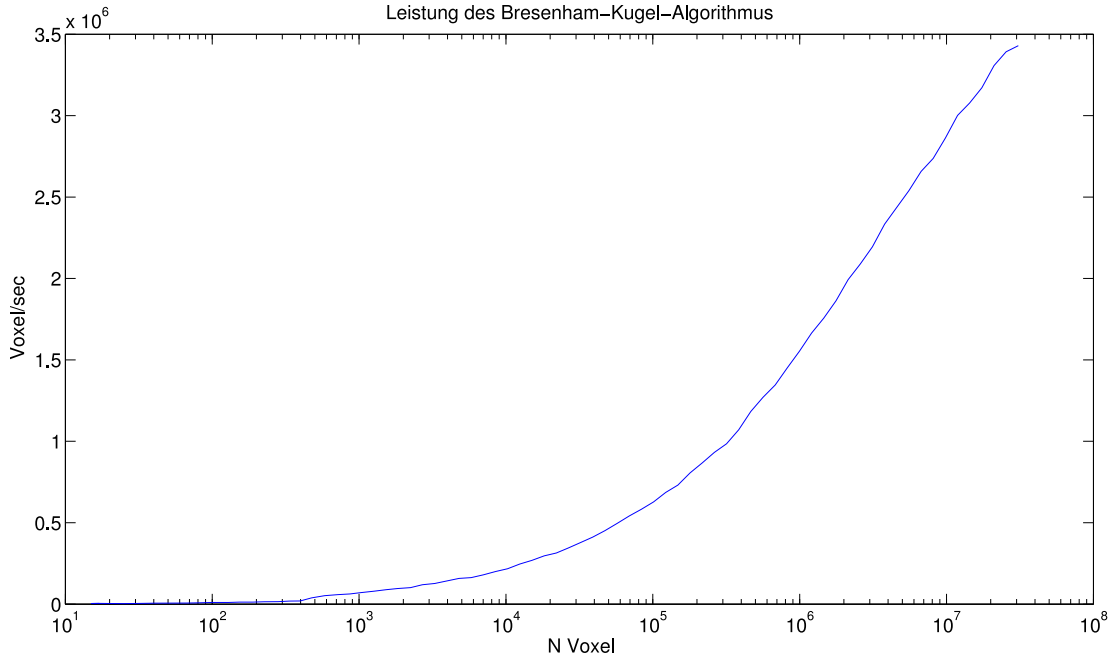

Abbildung 6.10: Leistung des Bresenham-Kugel-Algorithmus in Voxel/sec. *Die x-Achse listet die entsprechende Anzahl berechneter Voxel auf. Auf der y-Achse ist über die Laufzeit des Algorithmus die Leistung in* Voxel/sec *aufgetragen. Der Kugel-Algorithmus nähert sich mit hinreichend großen Werten* $3,5\,MVoxel/sec$ *an.*

6.4.2 Entgegen Brute-Force-Rasterung

Im Folgenden werden die Laufzeiten der Brute-Force-Ansätze aufgezeigt, um einen Unterschied zu den Bresenham-Algorithmen ziehen zu können. Der Vergleich mit der Rasterung nach Brute-Force-Verfahren zeigt dabei den zeitlichen Gewinn gegenüber einer angenommen optimalen Lösung.

6.4.2.1 Kreis

Die Kreisrasterung nach Brute-Force-Ansatz erreicht im Leistungstest ein Maximum von fast 550 Pixel/sec, siehe Abbildung 6.11. Mit Erhöhung der Komplexität sinkt die Leistung und nähert sich schließlich den 0 Pixel/sec an. Die zugehörige Aufwandsabschätzung von $O(n^2)$ ist bedingt durch eine Berechnungsschleife über alle Koordinaten und die Vergleiche aller Koordinaten untereinander bei den Nachbarprüfungen. Der geschätzte Aufwand ist als rote Kurve eingezeichnet. Der Verlauf der Leistung der Brute-Force-Kreisrasterung nähert sich der Funktion an und bestätigt damit die Abschätzung. Der Unterschied ist über die Vorverarbeitung zu erklären, welche in die Abschätzung nicht mit einfließt und den Kurvenverlauf dadurch nicht beeinflusst.

Im Vergleich mit dem Bresenham-Kreis zeigt das einen drastischen Unterschied. Bei einer Berechnung von 500 Pixel ist der Brute-Force-Ansatz bereits fast bei 0 Pixel/sec angelangt. Der Bresenham-Kreis erreicht hier eine Geschwindigkeit von 30 KPixel/sec bei einem vergleichbaren Rasterungsergebnis.

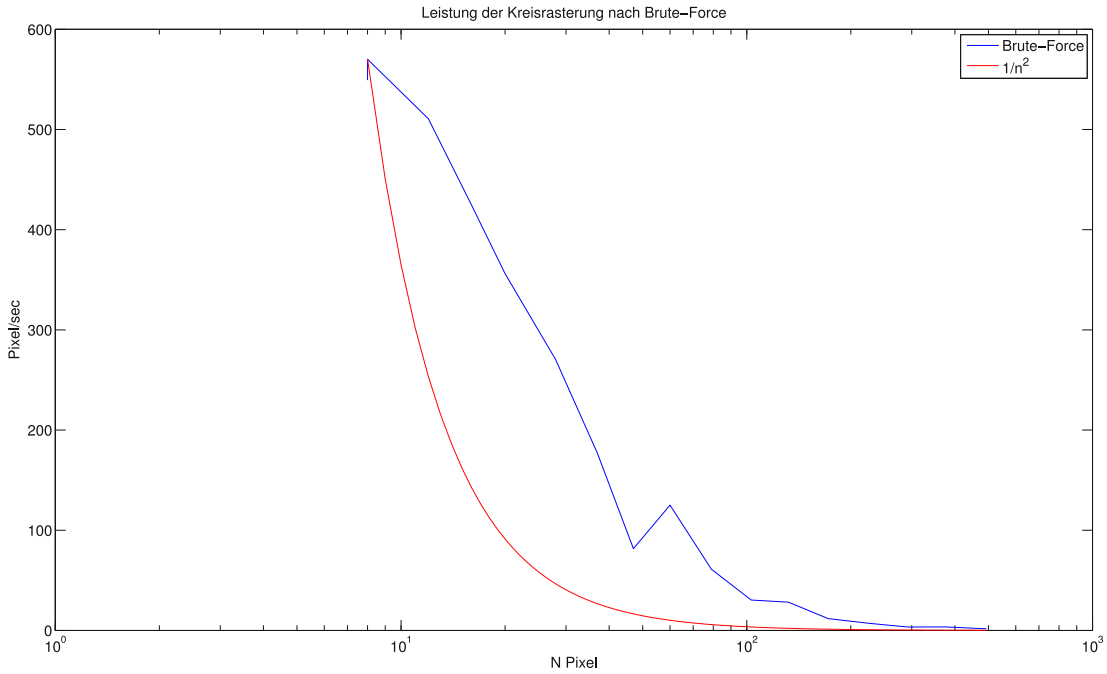

Abbildung 6.11: Leistung der Brute-Force-Kreisrasterung in Pixel/sec. *Die x-Achse listet die entsprechende Anzahl berechneter Pixel auf. Auf der y-Achse ist über die Laufzeit des Algorithmus die Leistung in Pixel/sec aufgetragen. Die Rasterung nach Brute-Force (blau) erreicht für den Kreis nur bei sehr kleinen Werten sein Maximum mit ungefähr* 500 *Pixel/sec. Mit größer werdender Pixelanzahl sinkt die Leistung rapide und nähert sich den* 0 *Pixel/sec an. Die Aufwandsabschätzung ist als rote Kurve eingezeichnet. Dies dient dem Beweis, dass die Schätzung* $O(\mathrm{n}^2)$ *in der Praxis erreicht wird. Dazu musste, bedingt durch die logarithmische x-Achse, das Reziproke* $\frac{1}{n^2}$ *verwendet werden. Die Abweichungen zwischen den beiden Kurven sind durch die Vorverarbeitung zu erklären, welche nicht in die Abschätzung eingeflossen ist.*
Im Vergleich zum Bresenham-Kreis-Algorithmus mit 2 *MPixel/sec bei* 10^3 Pixel *hängt die Brute-Force-Kreisrasterung deutlich hinterher.*

6.4.2.2 Kugel

Die Aufwandsabschätzung im Falle der Kugel variiert nicht zum Kreisfall. Auch hier findet eine Schleife über die Koordinaten mit zugehörigen Nachbarprüfungen statt. Der Aufwand nach der Landau-Notation ist demnach ebenfalls $\mathrm{O}(\mathrm{n}^2)$. Bei Betrachtung der Tests ist allerdings ein erhöhter Aufwand im Gegensatz zur Kreisvariante erkennbar. Abbildung 6.12 zeigt ein deutliches Leistungsprofil. Mit einem Start an 280 Voxel/sec für 18 berechnete Voxel liegt die Rasterung mit

Brute-Force-Prinzip deutlich unter dem Bresenham-Verfahren. Mit steigender Voxelanzahl sinkt auch hier der Brute-Force-Ansatz deutlich ab. Mit 10^2 zu berechnenden Voxeln nähert sich der Ansatz ebenfalls den 0 Voxel/sec an. Der Ansatz nach Bresenham-Prinzip erreicht für diese Größenordnung eine Leistung von 1 KVoxel/sec und steigert sich im weiteren Verlauf auf bis zu $3,5$ MVoxel/sec.

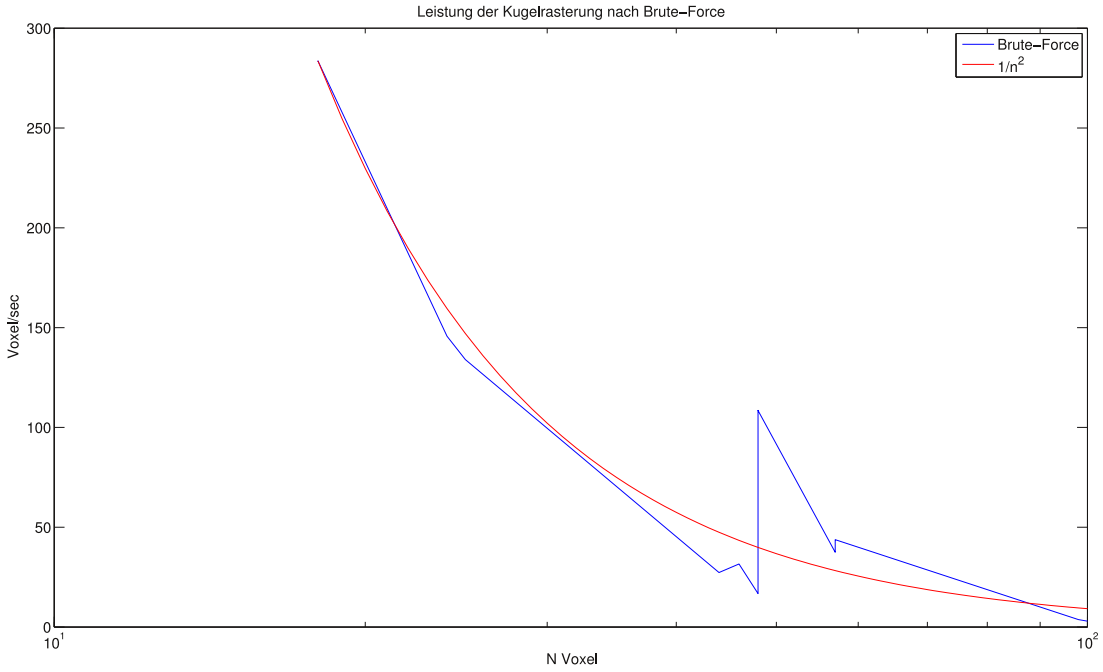

Abbildung 6.12: Leistung der Brute-Force-Kugelrasterung in Voxel/sec. *Die x-Achse listet die entsprechende Anzahl berechneter Voxel auf. Auf der y-Achse ist über die Laufzeit des Algorithmus die Leistung in* Voxel/sec *aufgetragen. Der Brute-Force-Ansatz (blau) erreicht für Kugeln nur bei sehr kleinen Werten sein Maximum mit ungefähr* $280\,Voxel/sec$*. Mit größer werdender Voxelanzahl sinkt die Leistung rapide und nähert sich den* $0\,Voxel/sec$ *bereits bei* $10^2\,Voxel$ *an. Die Kurve nähert sich dem abgeschätzten Aufwand (rot) an. Hier musste durch die logarithmische x-Achse ebenfalls das Reziproke* $\frac{1}{n^2}$ *verwendet werden. Für die Artefakte im Leistungsverlauf gibt es die mögliche Erklärung, dass die Nachbarprüfung der Kugel mit Parametern im Gleitkommabereich starken Schwankungen im Aufwand unterliegt. Ein weiterer möglicher Einfluss ist das Betriebssystem oder installierte Software, welche während der Messung aktiv war.*

6.4.3 Entgegen Ganzzahl

Der entwickelte Kreis-Algorithmus wird im Folgenden entgegen der Ganzzahl-Variante aus der Literatur [16] nach dem Ablaufplan 3.6 aus dem Abschnitt 3.2 verglichen. Dabei wurden typische Performance-Aspekte, wie zum Beispiel Preallokierung, angepasst. Ebenfalls wurde die Koordinatenspiegelung nicht in der Berechnung getätigt, sondern einmalig am Schluss. Mit dem Vergleich soll ermittelt werden, welche Leistungsveränderung durch die Erhöhung der Fähigkeiten des Verfahrens entstehen.

Der Kugel-Algorithmus kann nicht gegen eine Ganzzahl-Variante verglichen werden, da kein bestehender Algorithmus in der Literatur gefunden wurde.

Eine Evaluation entgegen dem Verfahren nach SAFT wird ebenfalls nicht durchgeführt. Diese Entscheidung ist begründbar mit der Tatsache, dass die implementierten Algorithmen einen geringeren Fähigkeitenbereich aufweisen. Ein Vergleich ist erst sinnvoll, wenn Ellipsen und Ellipsoide nach dem Bresenham-Verfahren gerastert werden können.

Abbildung 6.13 zeigt einen Vergleich der beiden Bresenham-Kreis-Varianten bezogen auf die Leistung in Pixel/sec. Dabei zeigt sich, dass die Ganzzahl-Variante bis zu einer Pixelmenge von ungefähr $N = 10^6$ Pixel deutlich über der Gleitkomma-Variante liegt. Sie erreicht in diesem Bereich bis zu 20 MPixel/sec. Allerdings sind Unregelmäßigkeiten im Verlauf erkennbar. Begründbar sind diese möglicherweise durch Hardware-Eigenschaften des Testrechners. Die Ganzzahl-Variante arbeitet nach dem Prinzip einen Kreisoktant zu berechnen und diesen anschließend auf einen vollständigen Kreis zu spiegeln. Das erzeugt viele lesende und schreibende Speicherzugriffe. Durch die komplexe Speicherverwaltung in modernen PCs bricht die Performance hier drastisch auf 4 MPixel/sec ein und bleibt dort konstant. Die Erklärung der Artefakte im Verlauf würde einer tiefergehenden Betrachtung benötigen. Der Ganzzahl-Algorithmus ist jedoch speichertechnisch nicht optimal konzipiert. Ein Eliminieren der Spiegelung würde sehr wahrscheinlich die starken Ausschläge und den Abfall eliminieren.

Der Grund für den stabilen Verlauf der Gleitkommavariante liegt wahrscheinlich darin, dass die Verarbeitung sequenziell in acht Schleifen getätigt wird. Der benötigte Speicher steht vor allen Schleifen fest und kann allokiert werden. Anschließend finden nur noch Zugriffe auf die bestimmten Speicherstellen statt. Der

Gleitkomma-Algorithmus erreicht damit auch für große Werte eine Leistung von fast 12 MPixel/sec und liegt damit deutlich über der Ganzzahl-Variante.

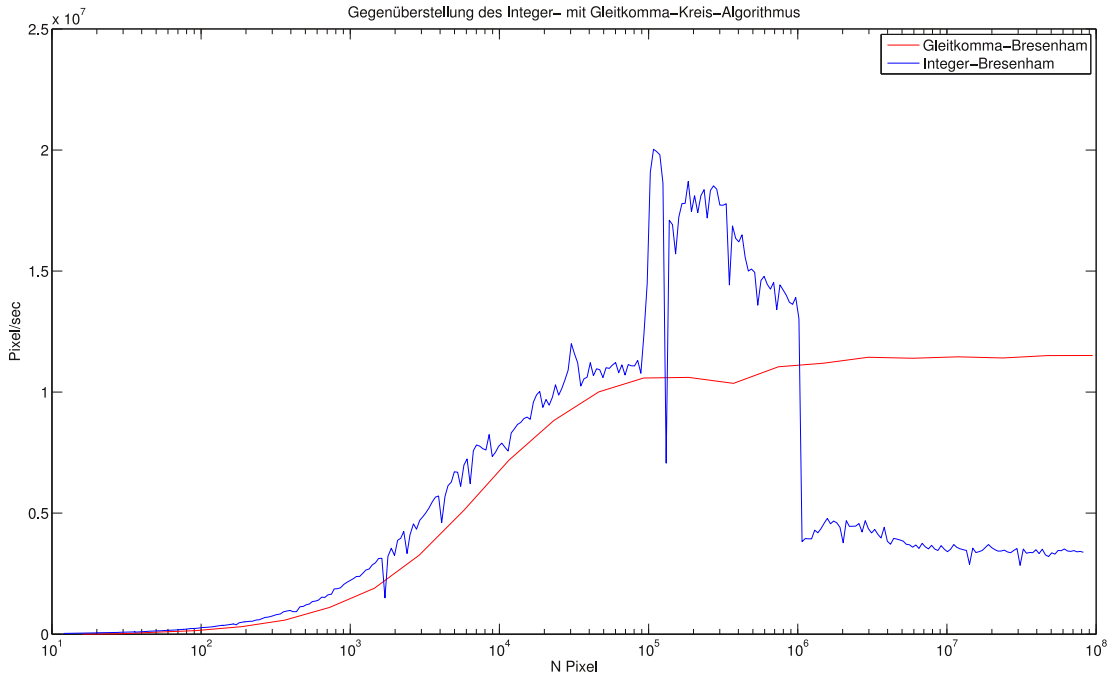

Abbildung 6.13: Vergleich des Ganzzahl- mit dem Gleitkomma-Bresenham-Kreis-Algorithmus bezüglich der Leistung in Pixel/sec. *Die x-Achse listet die entsprechende Anzahl berechneter Pixel auf. Auf der y-Achse ist über die Laufzeit des Algorithmus die Leistung in* Pixel/sec *aufgetragen. Die Ganzzahl-Variante erreicht, bis ca.* 10^5 *zu berechnenden Pixel, einen höheren Pixeldurchsatz von* $20\,MPixel/sec$*, im Vergleich zum Gleitkomma mit* $11\,MPixel/sec$*. Allerdings findet anschließend ein Einbruch auf ca.* $4\,MPixel/sec$ *statt. Der Gleitkomma-Algorithmus bleibt stabil und nähert sich den* $12\,MPixel/sec$ *an. Da der Ganzzahl-Algorithmus ein komplexeren Kurvenverlauf als der Gleitkomma-Algorithmus aufwies, wurde dieser in x-Dimension höher aufgelöst. Die beiden Algorithmen wurden ebenfalls unterschiedlich gemittelt, der Ganzzahl-Algorithmus aus Zeitgründen weniger als der Gleitkomma-Algorithmus.*

7 Ergebnisse und Diskussion

Das folgende Kapitel behandelt kritisch die Vorgehen, Konzepte, Implementierungen und Ergebnisse des Projekts

7.1 Entwickelte Konzepte und Vorgehen

Innerhalb dieser Arbeit wurde eine umfangreiche Literaturrecherche und Konzeption für den Kreis und die daraus resultierende Kugel durchgeführt.

7.1.1 Kreis

Bisherige Algorithmen verwendeten zur Erzeugung eines Kreises die Symmetrieeigenschaften dieses Körpers. Mit der Einteilung in Oktanten bedeutet das algorithmisch die Berechnung eines Oktant mit anschließender Spiegelung in die anderen sieben Oktanten. Mit einer Verschiebung im Gleitkomma-Bereich geht die Symmetrieeigenschaft des Rasterergebnisses verloren. Eine Spiegelung ist ohne zusätzliche Fehler in der Rasterung nicht möglich. Es wurde entschieden, die Oktanten einzeln in separaten Schleifen zu berechnen. Das ist auch bei Betrachtung der algorithmischen Leistung eine bessere Lösung, da Speicherzugriffe in aktuellen Computern generell als langsamer als Berechnungen innerhalb der CPU angenommen werden können .

Um die Koordinatenberechnungen der einzelnen Oktanten gezielt zu kontrollieren, wurden die Berechnungsgrenzen für jeden Oktant so definiert, dass sie sich an den Oktantengrenzen gegenseitig ausschließen (Abbildung 4.8). Die Wahl der aus- und eingeschlossenen Oktantengrenzen wurde dabei für den globalen Kreis nicht symmetrisch gewählt. Zur Vereinheitlichung ist eine Optimierung an dieser Stelle vorstellbar. Der Aufwand zur Anpassung der Grenzbedingungen und die Veränderungen am Code für diesen Schritt sind minimal.

Die Oktantenberechnung findet in separaten Schleifen statt. Dieser Schritt wurde getätigt, da jeder Oktant seine eigene Laufrichtungskombination der beiden Dimensionen x und y aufweist. In der Vorberechnung wurde ebenfalls nach Oktanten getrennt. Dieser Schritt wurde an dieser Stelle getan, da die Radienanpassung für den ersten, zweiten, dritten und achten Oktant unterschiedlich zu den anderen vier Oktanten ist. Weiterhin ist die Berechnung des Startfehlers abhängig von den spezifischen Laufrichtungen.

Eine generalisierte Lösung der Oktantenvor- und Koordinatenberechnung, beispielsweise über MATLAB-spezifische „function handles“, würde die Zeilenzahl im Code und möglicherweise den Aufwand des Algorithmus verringern.

Mit dem Berechnen der einzelnen Oktanten mussten Abbruchbedingungen entworfen werden, die die Bestimmungen zur Berechnung der Oktantengrenzen widerspiegeln. Dazu wurden zwei Ansätze entwickelt beziehungsweise aus der Literatur verwendet:

Tangentensteigung

Die Betrachtung der Tangente als Abbruchbedingung ermöglicht Präzision beim Wählen des letzten Berechnungsschrittes anhand der Oktantengrenzen. Allerdings ist ein Algorithmus nach diesem Konzept laufzeittechnisch problembehaftet, da in jedem Berechnungsschritt die aktuelle Tangentensteigung berechnet werden muss. Das erzeugt einen zusätzlichen Rechenaufwand und verlangsamt den Algorithmus.

Schrittzahlberechnung

Ein zweites Konzept beschleunigt die Schrittwahl innerhalb des Bresenham, indem die Schrittzahl für jeden Oktant im Vornherein berechnet wird. Damit vereinfachen sich die Schleifenbedingungen. Allerdings ist das in dieser Arbeit verwendete Konzept zur Berechnung nur eine Näherung, da die maximale theoretisch mögliche Schrittzahl bestimmt wird. Dadurch werden Lücken zwischen den Oktanten vermieden. Es entstehen an anderer Stelle allerdings Überlagerungen an den Oktantengrenzen. Hier werden doppelte Punkte berechnet, welche abhängig von der Eingabe des Benutzers entfernt werden können. Das Entfernen der doppelten Punkte ist ein zusätzlicher Rechenaufwand und wurde deshalb optional eingebaut.

7.1.2 Kugel

Als Konzept einer Kugelrasterung wurde der Kreisalgorithmus vollständig weiterverwendet, allerdings in der Struktur verändert. Es werden die korrekten x-y-Schichten in die entsprechende z-Dimension positioniert. Dieses Vorgehen ist möglich, da mit einem Raster gearbeitet wird. Für jede z-Position wird dafür der spezielle Radius berechnet, um die Kreisschicht korrekt zu rastern. Damit ist die Kugelrasterung eine hybride Lösung aus Bresenham-Verfahren und Brute-Force. In z-Dimension werden die Positionen über einen Brute-Force-Ansatz bestimmt, die Koordinaten in x und y werden über den Bresenham bestimmt.

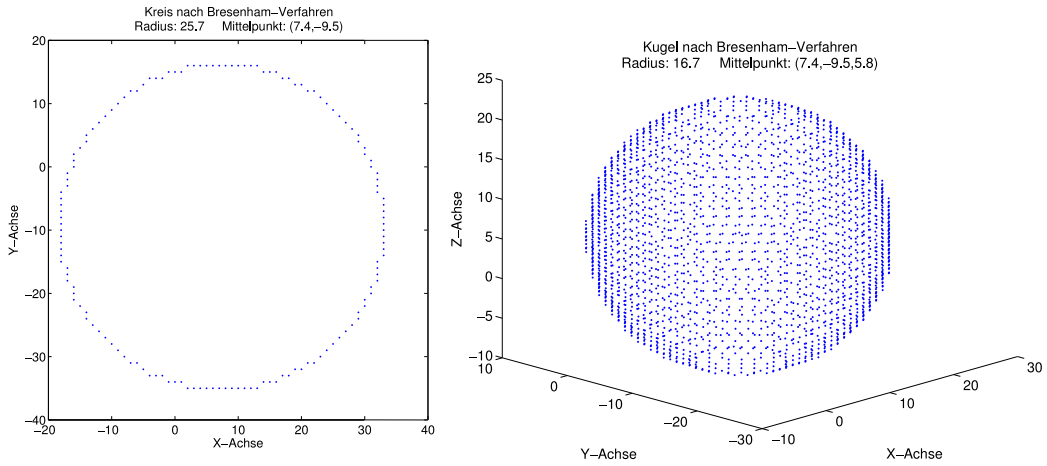

Abbildung 7.1: Beispielrasterungen eines Kreises und einer Kugel nach dem Bresenham-Verfahren. *Das Teilbild a zeigt einen Kreis mit Gleitkomma-Parametern mit dem implementierten Kreis-Algorithmus gerastert. Teilbild b zeigt eine Kugel mit Gleitkomma-Parametern mit dem implementierten Kugel-Algorithmus gerastert.*

7.1.3 Eigenschaften

Die entwickelten Algorithmen sollen im Folgenden auf die mathematischen theoretischen Eigenschaften einer Kurve im Bresenham-Verfahren aus der Tabelle 3.1 abgeprüft werden. Dazu werden die Eigenschaften noch einmal kurz wiederholt:

(1) Metrisch genau Jeder Rasterpunkt sollte so nah wie möglich an der Originalkurve liegen.

(2) Verbunden Die Rasterkoordinaten sollten durch König-Schachzüge verbunden sein.

(3) Topologisch korrekt Der Verlauf der Schachzüge zwischen den Koordinaten sollte den Laufrichtungen der Originalkurve folgen.

(4) Dünn Jede Rasterkoordinate sollte genau zwei Nachbarn besitzen.

(5) Symmetrisch Die Rasterung sollte mit den symmetrischen Operationen des Rasters übereinstimmen.

(6) Beschreibbar Die Rasterung sollte mathematisch beschreibbar sein.

Die metrische Genauigkeit aus Eigenschaft (1) wurde in diesem Kapitel für Kreis und Kugel beim Großteil der Fälle bestätigt. Ebenfalls ist Eigenschaft (2) erfüllt, bedingt durch die durchgeführten Schrittberechnungen der Oktanten. Die Eigenschaft (3) ist nur für die jeweiligen Teilbereiche (zum Beispiel Oktanten) der Körper gegeben. Für den kompletten Körper ist diese Eigenschaft nicht erfüllt. Das war allerdings auch keine Anforderung und ist damit nicht negativ zu bewerten. Durch diese Abweichung ist der Algorithmus nicht für Plotter[1] geeignet. Eigenschaft (4) gilt für eine ideale Kurve. In der Praxis kommt es hier bereits beim Kreis mit ganzzahligen Komponenten zu Abweichungen an den Oktantengrenzen. Für den Großteil der Rasterungsfälle ist diese Eigenschaft für den Kreis erfüllt. Bei der Kugel lautet diese Eigenschaft „Jede Rasterkoordinate sollte genau acht Nachbarn besitzen.". Sie trifft für die Kugel ebenfalls in den meisten Fällen zu. Die Eigenschaft (5) konnte durch die Betrachtung des Verfahrens auf Gleitkomma-Basis nicht eingehalten werden. Spiegelungen, Rotationen und Transformationen sind nicht mehr möglich. Die mathematische Beschreibbarkeit aus der Eigenschaft (6) ist ebenfalls nicht erfüllt, da keine linearen Ablaufpfade bestehen. Es wurde keine entsprechende Projektanforderung gestellt, deswegen ist diese Abweichung für das Ergebnis nicht von Bedeutung.

Der Nachweis der Rasterungsqualität wurde in Abschnitt 6.3 durchgeführt.

[1] Plotter - Ein Ausgabegerät zur Zeichnung von Funktionsgraphen.

7.2 Rasterung um Voxelgröße

Bei Rasterungen von Kurven mit einem Radius $r \approx 1$ Voxel kommt es zu Inkompatibilität mit dem Bresenham-Verfahren. Durch die gezwungenen Schritte in mindestens einer Dimension werden Rasterungen mit ungewollt großen Abweichungen berechnet, siehe Abbildung 7.2. Einige Oktanten fallen in diesen Fällen teilweise in Voxel zusammen. Eine definitive obere Grenze des Radius für solch ein Verhalten konnte nicht bestimmt werden. Durch die Verschiebung des Mittelpunktes im reellen Zahlenbereich besteht hier ein großer Raum der Varianz. Es wird vermutet, dass die obere Grenze bei einem Radius von $r \approx 1,8$ Voxel liegt. Beispiele sind auch in der Literatur zu finden, siehe Abbildung 3.8 und Abbildung 3.9. Im Ausblick wird ein Lösungansatz für dieses Problem aufgezeigt werden.

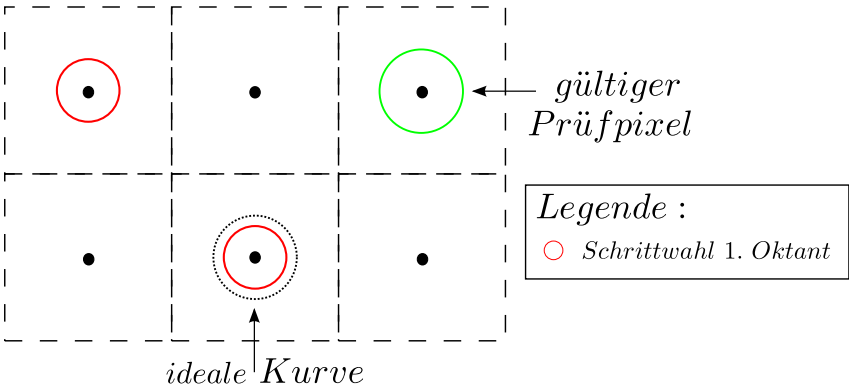

Abbildung 7.2: Zu große Schrittweiten der Oktanten bei einem Radius r < 1 Voxel. *Durch die Gültigkeit des Prüfpixel, da er auf der verlängerten Oktantengrenze liegt, erhält der erste Oktant eine Schrittanzahl von Zwei. Der Start ist auf dem Mittelpunkt des zu rasternden Kreises und der eine zusätzliche Schritt wird auf einen Pixel weit außerhalb des Kreises getätigt.*

7.3 Evaluation

Die implementierten Algorithmen wurden umfangreich gemessen und evaluiert. Es wurden Messungen über die Rasterungsqualität und Performance durchgeführt. Um eine quantitative Aussage treffen zu können, wurde eine Vergleichsrasterung nach dem Brute-Force-Prinzip entwickelt, welcher als Optimum angenommen wurde. Der Kreis-Algorithmus wurde zusätzlich entgegen der bestehenden Ganzzahl-Version aus der Literatur verglichen.

7.3.1 Kreis

Beim Vergleich zeigte sich, dass der Bresenham-Kreis-Algorithmus sich der Brute-Force-Kreisrasterung annähert und dabei die Bedingungen für eine optimale Rasterung fast in allen Fällen erfüllt. Abweichungen gibt es bei einem Radius $r \approx 1\,\text{Pixel}$. Das ist bedingt durch die Problematik mit dem Bresenham-Verfahren in diesem Bereich, siehe Abschnitt 7.2
Der Kreis-Algorithmus ist ebenfalls der ganzzahligen Variante gegenübergestellt worden. Dabei zeigte sich für entsprechend große Anzahlen an zu berechnenden Rasterpunkten ein deutlicher Unterschied in den Leistungen. Anfangs liegt die Ganzzahl-Variante in der Performance über dem Gleitkomma-Ansatz. Mit steigender Pixelanzahl zeigt sich jedoch die Schwäche der Spiegelung im Algorithmus. Dadurch findet ein starker Einbruch auf $4\,\text{MVoxel/sec}$ statt, wohingegen der Gleitkomma-Algorithmus sich weiter steigert. Damit wird ein Pixel-Durchsatz des Integer-Algorithmus multipliziert mit Faktor Drei erreicht.

7.3.2 Kugel

Der Kugel-Algorithmus erzeugt im Vergleich zum Brute-Force-Ansatz für manche Metriken, zum Beispiel die maximale Abweichung, bessere Ergebnisse. Die Grenze der anfangs definierten maximalen Abweichung von $\frac{\sqrt{3}}{2} = 0,866\,\text{Voxel}$ wurde in den Tests nicht überschritten. Da die Kreiskonzepte wiederverwendet wurden, ist allerdings anzunehmen, dass bei einem Radius $r \approx 1\,\text{Voxel}$ ebenfalls Verletzungen dieser Bedingung entstehen.

Die Performance des Kugelalgorithmus weist für den Testrechner eine Sättigung an $3,5\,\text{MVoxel/sec}$ auf. Damit liegt die Kugel deutlich unter dem Kreis, die Komplexität des Problems ist jedoch größer. Weiterhin ist das Kugelkonzept eine Verbindung aus Bresenham-Verfahren und Brute-Force.

7.4 Projektfazit

Im Rahmen dieser Arbeit wurde das Bresenham-Verfahren zur Rasterung von Kurven von ganzzahligen Parametern auf Parameter im Gleitkommabereich erweitert. Es ist damit möglich, die Rasterung von Kreisen im zwei-Dimensionalen Raum und Kugeln im drei-Dimensionalen Raum mit beliebigen Parametern zu

durchzuführen. Dementsprechend wurde eine Verallgemeinerung des Bresenham-Verfahrens in zwei Aspekten erreicht. Die Verschiebung des Körpers kann durch diese Arbeit einen beliebigen Wert im reellen Zahlenbereich annehmen. Dies war vorher nur für ganzzahlige Parameter möglich. Ebenfalls können die Radien der zu rasternden Kurven in den reellen Zahlenbereich positioniert werden. Auch hier war die bestehende Vorgabe der Ganzzahl-Parameter für die Algorithmen, sollte ein korrektes Ergebnis berechnet werden.

Die Komplexität des Gleitkomma-Problems am Bresenham-Verfahren ist bei der Planung des Projekts unterschätzt worden. Das Ziel des Ellipsoid konnte in der zur Verfügung stehenden Zeit nicht erreicht werden. Die entwickelten Konzepte bilden allerdings eine umfangreiche Wissensbasis, über die eine Rasterung von Ellipsen und Ellipsoiden ermöglicht wird. In diesem Rahmen zeigte sich, dass das Bresenham-Verfahren ungenügend für Rasterungen um die Voxelgröße ist.

7.4.1 Projektmethodik

Die testgetriebenen Entwicklung hat sich für das Projekt bewährt. Dabei wurden anfangs Unit-Tests für die zu entwickelnden Algorithmen aufgestellt, welche bei jedem Implementierungsschritt abgeprüft wurden. Weiterhin wurden Metriken definiert, anhand derer die Qualität der algorithmischen Ergebnisse gemessen werden konnten. Mithilfe der Unit-Tests und der Metriken konnten Fehler und Schwachstellen in den verschiedenen Konzepten frühzeitig erkannt werden. Daraufhin konnten diese spezifischer und mit einem erweiterten Betrachtungswinkel optimiert werden. Die abschließende Evaluation verwendete die Unit-Tests in algorithmusspezifischen White-Box-Tests und verglich anhand der Metriken zu einem angenommenen Optimum.

Das frühzeitige Auftreten von Fehlern beim Kreiskonzept führte dazu, dass weitere Anpassungen getätigt werden mussten. Die umfangreichen Anpassungen waren jedoch sehr zeitaufwändig und verhinderte dadurch das Erreichen der angestrebten Ziele.

7.4.2 Erfüllung der Anforderungen

Die an das Projekt gestellten Anforderungen aus der Abschnitt 2.1 konnten fast nachweislich in Abschnitt 6.3 vollständig erfüllt werden.

Bildrekonstruktionszeit

Der entwickelte Kreis-Algorithmus erreicht eine Performance von bis zu 12 MPixel/sec. Damit liegt er für sehr große Mengen an zu rasternden Pixeln sogar über der Ganzzahl-Variante aus der Literatur. Der Algorithmus zur Kugelrasterung erreicht fast $3,5$ MVoxel/sec. Damit besteht hier Potenzial, über weitere Optimierungen und Parallelisierung der Algorithmen die Bildrekonstruktionsdauer weiter zu verringern. Es kann angenommen werden, dass die Lösung für eine Ellipse beziehungsweise Ellipsoid nach dem Bresenham-Verfahren eine Steigerung der algorithmischen Komplexität um Faktor Zwei ($\Delta_{complex}$) aufweisen wird. Diese Annahme ist durch einen Vergleich der Formeln aus dem Abschnitt 4.5 mit denen des Kreis-Algorithmus abschätzbar. M. Weber stellte in [8] fest, dass nur ungefähr 1 % eines A-Scan für eine Bildrekonstruktion von Bedeutung sind. Mit dem Bresenham-Verfahren kann diese Einschränkung verwendet werden. Da das Verfahren nach SAFT alle 100 % eines A-Scan verwendet, muss dementsprechend die Leistung des Kugel-Algorithmus von $3,5$ MVoxel/sec (P_{Bres}) multipliziert mit Faktor 100 (x) erreicht werden. Damit ergibt sich für einen Ellipsoid in vergleichbarer Zeit eine benötigte Performance von $P_{SAFT} \approx 200$ MVoxel/sec, siehe Gleichung 7.1.

$$\begin{aligned} \frac{P_{Bres}}{\Delta_{complex}} \cdot x &= P_{SAFT} \\ \frac{3,5}{2} \cdot 100 &\approx 200\,\text{MVoxel/sec} \end{aligned} \tag{7.1}$$

Diese Performance wird momentan zwar erreicht, SAFT kann allerdings nicht weiter optimiert werden.[7] Die bresenhamartigen Algorithmen besitzen in diesem Bereich jedoch einiges an Potential. Durch weitere Optimierung, Portierung nach C und Parallelisierung, werden höhere Leistungen als mit dem aktuell verwendeten Verfahren SAFT erzielt werden können. Damit könnte, langfristig gesehen, eine Bildrekonstruktionszeit $t \leq 30$ min aus der Anforderung [Anf01] erreicht werden.

Parameter im reellen Zahlenbereich

Anforderung [Anf02] fordert Parameter im Zahlenbereich der reellen Zahlen $\mathbb{Q}$. Diese Anforderung konnte für die entwickelten Algorithmen vollständig erfüllt werden.

Keine doppelten oder unzuordenbare Koordinaten

In der Anforderung [Anf03] wurde gefordert, keine doppelten oder unzuordenbare Koordinaten zu berechnen. Über die White-Box-Tests konnte diese Eigenschaft der Algorithmen bewiesen werden.

Geometrisch enge Rasterung

Die Rasterung orientiert sich an einer optimalen Koordinatenwahl, bewiesen durch den Nachweis, dass $error_{\perp max} \leq \frac{\sqrt{n}}{2}$ Voxel gilt. Es gibt nur eine kleine Menge an Fällen, in denen diese Bedingung verletzt wird. Für einen Radius $r \approx 1$ fallen die Teilbereiche der Kreise und Kugeln in die selben Rasterbereiche. In diesen Fällen ist das Bresenham-Verfahren nicht zum Rastern geeignet. Durch das gezwungene Laufen eines Schrittes in mindestens einer Dimension werden Rasterpunkte mit zu großem Abstand gewählt. Die Anforderung [Anf04] ist für einen Großteil der möglicherweise auftretenden Fälle vollständig erfüllt.

Entwurf und Implementierung

Nach den Anforderungen [Anf05] und [Anf06] wurde ein robuster algorithmischer Entwurf in Hinblick auf Determinismus mit Implementierung in MATLAB gefordert. Diese Anforderung konnte für die entwickelten Konzepte erfüllt werden, siehe Kapitel 5.

Parameter der Ellipse

Anforderung [Anf07] bleibt in dieser Arbeit unerfüllt, da das Ziel der Ellipsen-Rasterung nicht erreicht wurde.

Geschlossene Rasterung

Die Erfüllung der Anforderung [Anf08] konnte nachweislich in Abschnitt 6.3 durch die Unit-Tests bewiesen werden.

7.5 Grundlagen für weitergehende Schritte

Mit der Kreis- und Kugelbetrachtung ist ein Großteil der Grundlagen und Konzepte für eine Ellipse beziehungsweise einen Ellipsoid erarbeitet. Das Konzept für Ellipsen oder Ellipsoide ist umfangreich vorgezeichnet. Es fehlt die detaillierte Anpassung und Umsetzung. Weiterhin wurde in dieser Arbeit auf ein Konzept

für eine Ganzzahl-Variante einer Ellipse eingegangen. Im Kapitel 8 wird gezeigt, welche Schritte und Probleme hier für zukünftige Arbeiten in Betracht kommen, um das Ziel zu erreichen.

8 Ausblick

Die vorliegende Arbeit ermöglicht vielfältige weitergehende Betrachtungen und Erweiterungen. Das folgende Kapitel soll hierzu einen Überblick geben.

8.1 Weitere Schritte

In dieser Arbeit wurden umfangreiche Grundlagen über das Bresenham-Verfahren auf Gleitkommabasis entwickelt. Mit dieser Basis lassen sich die entwickelten Konzepte auf weitere Körper ausweiten. Nach einer erfolgreichen Portierung können die Algorithmen dazu verwendet werden die Bildrekonstruktion deutlich zu beschleunigen.

8.1.1 Rasterungen um Voxelgröße

Es ist festgestellt worden, dass das Bresenham-Verfahren auf Rasterungen mit Parametern um die Pixel-/Voxelgröße nicht anwendbar ist. Das ist bedingt durch die notwendigen Schritte in mindestens einer Dimension. Hier kann eine hybride Lösung aus einem Rasterungsalgorithmus und einer Brute-Force-Suche ab einem bestimmten Radius durchgeführt werden. Dazu kann das Verfahren nach Abschnitt 4.3.6.4 beziehungsweise 4.4.2 auf mehrere Punkte angewendet werden. Das ist akzeptabel, da die Anzahl der zu berechnenden Koordinaten im Falle der Brute-Force-Suche gering ist. Nachteil ist ein Performance-Verlust, welcher allerdings insignifikant gering wäre. Alternativ kann ein Rasterungsverfahren gewählt werden, was in diesen Fälle korrekt rastert.

8.1.2 Rasterung weiterer Körper

Die Portierung auf Ellipsen und Ellipsoide ist, aus Überlegungen heraus, größtenteils schlussfolgernd aus den Kreis- und Kugelkonzepten.

8.1.2.1 Ellipse

Um die Konzepte auf generelle Ellipsen zu erweitern gibt es verschiedene empfohlene Ansätze. Diese sollen im Folgenden angedeutet werden. Die Auswahl stellt nur einen Teil der möglichen Verfahren dar.

Überlagerung zweier Kreise

Um eine generelle Ellipse zu rastern, wird der Ansatz nach D.W. Fellner und C. Helmberg aus [20] empfohlen. Der Ansatz des dort aufgezeigten Ellipsen-Algorithmus folgt dem Vorgehen, zwei Kreise, mit den beiden Radien der Ellipse berechnet, zu überlagern. Damit wird, nach Aussage der Autoren, eine robuste Ellipsenrasterung ermöglicht.

Scherung

Eine andere mögliche Rasterung genereller Ellipsen ist das Überlagern einer achsensymmetrischen Ellipse mit einer Geraden. Dabei muss eine Scherung durchgeführt werden. Hierzu wurde keine Literatur gefunden, sondern nur ein Verweis in [16]. Deswegen benötigt dieser Ansatz die meiste selbstständige mathematische Vorbetrachtung.

8.1.2.2 Ellipsoid

Ein Ellipsoid lässt sich nach dem Kugelprinzip umsetzen. Zu beachten ist hierbei, dass die berechneten Schichten Ellipsen darstellen und dementsprechend behandelt werden müssen. Weiterhin muss der korrekte Radius r_y für die jeweiligen Schichten berechnet werden.

8.1.2.3 Weiterverwertbare Konzepte

Aus der vorliegenden Arbeit können zur Unterstützung die Konzepte aus Abschnitt 4.3.5.2 und Abschnitt 4.3.4.3 verwendet werden. Dazu wird eine Möglichkeit benötigt, den konkreten Wert der jeweiligen Oktantengrenze auf der Ellipse zu berechnen. Anschließend wird als Schrittzahl die Distanz in der jeweilig schnellen Dimension verwendet. Das Konzept aus Abschnitt 4.3.6.4 kann ebenfalls verwendet werden, wenn ein einzelner Punkt berechnet werden muss. Mindestens der Fall der Radien $r_x = 0$ und $r_y = 0$ muss nach dem Prinzip behandelt werden.

8.1.3 Fixpunkt-Ansatz mit Integer

In dieser Arbeit wurde die Dynamik in den Nachkommstellenbereich der Floatingpoint Datentypen verwendet. Alternativ wäre ein Fixpunkt-Integer-Ansatz denkbar. Dieser nutzt die natürliche Dynamik des Integerdatentyps und mappt den abzubildenden Dynamikbereich darauf. Der Anwendungsfall muss dabei vorgeben, welche Auflösung und damit Genauigkeit benötigt wird. Damit können die Gleitkomma-Werte bis zu einem gewissen Grad auf dem gesamten Zahlenbereich des Integer verteilt werden. Der Vorteil dieser Methodik liegt darin, dass keine Float- oder Double-Variablen benötigt werden.

8.1.4 Integration in Bildrekonstruktion

Ist das Ziel des generellen Ellipsoid erreicht, kann dieser in die Bildrekonstruktion des USCT integriert werden. Nach diesem Schritt kann ein Vergleich des Algorithmus entgegen dem momentan verwendeten Verfahren SAFT gezogen werden. Durch eine weitere Optimierung der bresenhamartigen Algorithmen für verschiedene geometrische Körper kann der aufwändige SAFT-Algorithmus langfristig gesehen sehr wahrscheinlich verdrängt werden.

8.1.5 Portierung nach C

Als direkter nachfolgender Schritt ist eine Portierung der entwickelten Algorithmen in die Programmiersprache C denkbar. Der implementierte Code verwendet fast ausschliesslich nur auch in der C-Syntax existierende Konstrukte. Über eine Portierung ist eine Performance-Verbesserung zu ewarten. Während einer Portierung kann die Mathematik in den Algorithmus ebenfalls auf Geschwindigkeit optimiert werden. So können Multiplikationen teilweise durch $shift$-Befehle auf Bit-Ebene ersetzt werden. Weiterhin können Divisionen durch Multiplikationen mit dem Reziproke ersetzt werden. Auch Generalisierungen in den Verarbeitungsschleifen und Vorberechnungen sowie eine effizientere Speicherverwaltung sind Punkte eines möglichen Refactoring.

8.1.6 Parallelisierung

Eine Parallelisierung der Algorithmen ist denkbar, um auf modernen Computern mehrere CPUs für die Berechnungen zu verwenden. Parallelisiert werden können die Berechnungsschleifen und Vorberechnungen pro Oktant. Dabei kann jeder Oktant als Einheit gerechnet werden, unabhängig ob Schleife oder Vorberechnung. Bei der Kugel kann zusätzlich jede Schichtberechnung als parallele Aufgabe durchgeführt werden. Dieser Schritt wäre eine sinnvolle Erweiterung, da die Geschwindigkeit dadurch deutlich gesteigert werden kann.

8.2 Evaluation weiterer Rasteralgorithmen

Das Bresenham-Verfahren ist nicht das einzige Rasterungsverfahren. Im Laufe der Zeit wurden mehrere Methoden entwickelt, geometrische Körper zu rastern. Die im Folgenden aufgelisteten Ansätze sind nur exemplarisch zur Rasterung von Kreisen. Welche weiteren Verfahren für die bestehenden Anforderungen in Betracht kommen könnten, muss vorher überprüft werden. Dabei können sich Verfahren mit geringerem algorithmischen Aufwand oder besserem Verhalten bei Rasterungen um die Voxelgröße aufzeigen.

8.2.1 Horn-Algorithmus

Der Horn-Algorithmus wird unter anderem in [24] und [25] erwähnt und erläutert. Dieser Ansatz verwendet ausschließlich Additionen und Subtraktionen um geometrische Kreise zu rastern. Dabei wird ein Bereich von der Größe eines Pixel um die ideale Kurve gelegt und die nächsten beiden möglichen Rasterpositionen in ihrer Lage zu diesem Bereich abgeprüft.

8.2.2 Methode nach Metzger

Die Methode nach Metzger, siehe [25], geht nach den orthogonalen Abständen der Rasterpositionen durch jeweilige Radien. Die Koordinate mit dem geringeren Abstand zum idealen Radius wird gewählt.

Literaturverzeichnis

[1] Bertz, J. ; Dahm, S. ; Haberland, J. ; Kraywinkel, K. ; Kurth, B.-M. ; Wolf, U.: *Verbreitung von Krebserkrankungen in Deutschland. Entwicklung der Prävalenzen zwischen 1990 und 2010.* Berlin : http://www.rki.de/cln_151/nn_206802/DE/Content/GBE/Gesundheitsberichterstattung/GBEDownloadsB/Krebspraevalenz,templateId=raw,property=publicationFile.pdf/Krebspraevalenz.pdf, 11.04.2010

[2] *Die Mammografie.* http://www.krebsgesellschaft.de/mammografie,26189.html, 19.03.2010

[3] *Magnetresonanztomographie: Welche Technik steht dahinter? Welchen Stellenwert hat die Untersuchung in der Krebsmedizin?* http://www.krebsinformationsdienst.de/themen/untersuchung/kernspintomographie-technik.php, 19.03.2010

[4] *Ultraschalluntersuchung: Schallwellen zur Krebsdiagnose.* http://www.krebsinformationsdienst.de/themen/untersuchung/ultraschall.php, 19.03.2010

[5] Ruiter, N.V. ; Schwarzenberg, G.F. ; Zapf, M. ; Gemmeke, H.: Conclusions from an Experimental 3D Ultrasound Computer Tomograph. In: *IEEE NSS MIC*, 2008

[6] Stotzka, R. ; Ruiter, N.V. ; Müller, T.O. ; Liu, R. ; Gemmeke, H.: High resolution image reconstruction in ultrasound computer tomography using deconvolution. In: *SPIE's Internl. Symposium Medical Imaging 2005*, 2005, S. 315–325

[7] Zapf, M. ; Schwarzenberg, G.F. ; Ruiter, N.V.: High throughput SAFT for an experimental USCT system as MATLAB implementation with use of SIMD CPU instructions. In: *Medical Imaging 2008: Ultrasonic Imaging and Signal Processing* 6920 (2008), Nr. 1, S. 10–21

[8] Weber, M.: *Robuste Pulsdetektion für die Ultraschall–Computertomographie*. Masterarbeit, Hochschule Mannheim, 2006

[9] Bresenham, J.E.: Algorithm for computer control of a digital plotter. In: *IBM Systems Journal* 4 (1965), Nr. 1, S. 25–30

[10] Pitteway, M.L.V.: Algorithm for drawing ellipses or hyperbolae with a digital plotter. In: *The Computer Journal* 10 (Nov. 1967), Nr. 3, S. 282–289

[11] Bresenham, J.E.: A Linear Algorithm for Incremental Digital Display of Circular Arcs. In: *Communications of the ACM* 20 (1977), Nr. 2, S. 100–106

[12] Janser, A. ; Luther, W.: *Der Bresenham-Algorithmus*. Gerhard-Mercator-Universität-GH Duisburg, 1994

[13] van Aken, J.: An Efficient Ellipse-Drawing Algorithm. In: *Computer Graphics and Applications, IEEE* 4 (Sept. 1984), Nr. 9, S. 24–35

[14] van Aken, J. ; Novak, M.: Curve-Drawing Algorithms for Raster Displays. In: *ACM Transactions on Graphics (TOG)* 4 (1985), Nr. 2, S. 147–169

[15] McIlroy, M.D.: Getting Raster Ellipses Right. In: *ACM Transactions on Graphics (TOG)* 11 (1992), Nr. 3, S. 259–275

[16] Wikipedia: *Bresenham-Algorithmus*. `http://de.wikipedia.org/w/index.php?title=Bresenham-Algorithmus&oldid=70262651`, 27.03.2010

[17] Kennedy, J.: *A Fast Bresenham Type Algorithm For Drawing Ellipses*. `http://homepage.smc.edu/kennedy_john/BELIPSE.PDF`, 17.04.2010

[18] Eberly, D.: *Integer-based Algorithm for Drawing Ellipses*. `http://www.geometrictools.com/Documentation/IntegerBasedEllipseDrawing.pdf`, 2008

[19] Sarfraz, M.: *Performance And Accuracy Improvementes In Ellipse Drawing Algorithms*. `http://eprints.kfupm.edu.sa/43/1/Performance_And_Accuracy_Improvements_In_Ellipse_Drawing_Algorithms.pdf`, 19.04.2010

[20] Fellner, D.W. ; Helmberg, C.: Robust Rendering of General Ellipses and Elliptical Arcs. In: *ACM Transactions on Graphics (TOG)* 12 (1993), Nr. 3, S. 251–276

[21] *Bresenham: Linien–Algorithmus*. `http://www.phyta.net/brsnham.htm`, 15.04.2010

[22] Chaoui, J.: *Bresenham's 3D Line algorithm*. http://www.mathworks.com/matlabcentral/fileexchange/17658-bresenhams-3d-line-algorithm, 17.04.2010

[23] *Bresenham: Kreis–Algorithmus*. http://www.phyta.net/circalgo.htm, 15.04.2010

[24] McIlroy, M.D.: Best Approximate Circles on Integer Grids. In: *ACM Transactions on Graphics (TOG)* 2 (1983), Nr. 4, S. 237–263

[25] Wikipedia: *Rasterung von Kreisen*. http://de.wikipedia.org/w/index.php?title=Rasterung_von_Kreisen&oldid=57319119, 01.06.2010

Anhang

A. Vorbereitung der Bresenhamgerade für acht Oktanten

```
if (Dx >= 0)
  inkX = 1;
else
  inkX = -1;
  Dx = -Dx;
end
if (Dy >= 0)
  inkY = 1;
else
  inkY = -1;
  Dy = -Dy;
end
%  feststellen, welche Entfernung größer ist
if (Dx > Dy)             % x ist schnelle Richtung
  parallelX = inkX;
  parallelY = 0;
  errorParallel = 2*Dy; % Fehlerschritte schnell
  errorDiagonal = 2*Dx; % und langsam
else                     % y ist schnelle Richtung
  parallelX = 0;
  parallelY = inkY;
  errorParallel = 2*Dx;
  errorDiagonal = 2*Dy;
end
diagonalX = inkX;
diagonalY = inkY;
```

Listing 1: Vorbereitung der Variablen für die Bresenhamgerade in allen acht Oktanten.[1]

[1] Mit Änderungen entnommen aus: Wikipedia, Bresenham-Algorithmus

B. Erster Ellipsenquadrant nach Bresenham-Verfahren

```
x:=trunc(rx+1/2); y:=0;                                          1
t1:=rx*rx; t2:=2*t1; t3:=2*t2;                                   2
t4:=ry*ry; t5:=2*t4; t6:=2*t5;                                   3
t7:=rx*t5; t8:=2*t7; t9:=0;                                      4
d1:=t2-t7+t4; d2:=t1-t8+t5;                                      5
while d2<0 do          (* 1. Oktant der Ellipse *)                6
  setpixel(x,y);       (* setze Pixel an Stelle (x,y) *)            7
  y:=y+1;              (* inkrementiere y bei jedem Schritt *)      8
  t9:=t9+t3;                                                     9
  if d1 < 0 then       (* Fehleranpassung für y-Schritt *)          10
    d1:=d1+t9+t2;                                                11
    d2:=d2+t9;                                                   12
  else                 (* Schritt in langsame x-Richtung *)         13
    x:=x-1;                                                      14
    t8:=t8-t6;                                                   15
    d1:=d1-t8+t9+t2;                                             16
    d2:=d2-t8+t5+t9;                                             17
end                                                              18
repeat                 (* 2. Oktant der Ellipse *)                19
  setpixel(x,y);       (* setze Pixel an Stelle (x,y) *)            20
  x:=x-1;              (* Dekrementiere x bei jedem Schritt *)      21
  t8:=t8-t6;                                                     22
  if d2<0 then         (* Schritt in langsame y-Richtung *)         23
    y:=y+1;                                                      24
    t9:=t9+t3;                                                   25
    d2:=d2-t8+t5+t9;                                             26
  else                 (* Fehleranpassung des x-Schritt *)          27
    d2:=d2-t8+t5;                                                28
until x<0                                                        29
```

Listing 2: Erster Quadrant mit Zweipunktalgorithmus nach Bresenham-Verfahren. Durch die Variablen t1-t9 werden Multiplikationen und Exponentialfunktionen vermieden. In den Schleifen finden nur noch Vergleiche gegen 0, Additionen und Subtraktionen statt.[2]

[2]Mit Änderungen entnommen aus: J. van Aken (1984), S. 32

C. Bresenham-Algorithmus in MATLAB (3D)

```
function [Coords] = bres3Dm(x1, y1, z1, x2, y2, z2)
pixel = zeros(1,3,'int32');
pixel(1) = x1; pixel(2) = y1; pixel(3) = z1;
dx = x2 - x1; dy = y2 - y1; dz = z2 - z1;
lC = 3 + ceil(abs(dx)+abs(dy)+abs(dz));
Coords = zeros(lC,3,'int32');
iC = 1;
if dx < 0
  x_inc = -1;
else
  x_inc = 1;
end
le = abs(dx);
if dy < 0
  y_inc = -1;
else
  y_inc = 1;
end
m = abs(dy);
if dz < 0
  z_inc = -1;
else
  z_inc = 1;
end
n = abs(dz);
dx2 = 2*le; dy2 = 2*m; dz2 = 2*n;
if ((le >= m) & (le >= n))
  err_1 = dy2 - le; err_2 = dz2 - le;
  for i = 1:le
    Coords(iC,:) = pixel; iC = iC + 1;
    if (err_1 > 0)
      pixel(2) = pixel(2) + y_inc; err_1 = err_1 - dx2;
    end
    if (err_2 > 0)
      pixel(3) = pixel(3) + z_inc; err_2 = err_2 - dx2;
    end
    err_1 = err_1 + dy2; err_2 = err_2 + dz2;
    pixel(1) = pixel(1) + x_inc;
  end
elseif ((m >= le) & (m >= n))
```

```
  err_1 = dx2 - m; err_2 = dz2 - m;
  for i = 1:m
    Coords(iC,:) = pixel; iC = iC + 1;
    if (err_1 > 0)
      pixel(1) = pixel(1) + x_inc; err_1 = err_1 - dy2;
    end
    if (err_2 > 0)
      pixel(3) = pixel(3) + z_inc; err_2 = err_2 - dy2;
    end
    err_1 = err_1 + dx2; err_2 = err_2 + dz2;
    pixel(2) = pixel(2) + y_inc;
  end
else
  err_1 = dy2 - n; err_2 = dx2 - n;
  for i = 1:n
    Coords(iC,:) = pixel;
    iC = iC + 1;
    if (err_1 > 0)
      pixel(2) = pixel(2) + y_inc; err_1 = err_1 - dz2;
    end
    if (err_2 > 0)
      pixel(1) = pixel(1) + x_inc; err_2 = err_2 - dz2;
    end
    err_1 = err_1 + dy2; err_2 = err_2 + dx2;
    pixel(3) = pixel(3) + z_inc;
  end
end
Coords(iC,:) = pixel;
iC = iC + 1;
Coords(iC:lC,:) = [];
```

Listing 3: Der Bresenham-Algorithmus in 3D in MATLAB.[3]

[3]Mit Änderungen entnommen aus: J. Chaoui, Bresenham's 3D Line algorithm

D. Bresenham-Kreis-Algorithmus in MATLAB

```
function coords = bresenham_circle(centre,radius,doubledPoints)
%BRESENHAM_CIRCLE return points of the circle using brenham's alogrithm.
% Syntax: [py px] = bresenham_circle(radius,centre,doubledPoints);
%
% Inputs:
%   radius          radius of the circle
%   centre          centre of circle
%   doubledPoints let doubled Points happen (0-yes,1-no)
%
% Outputs:
%   px      x points
%   py      y points
%
%
% Example:
%   [y x] = bresenham_circle([5,5],100,1);
%   figure; plot(x,y,'*');axis square;
%
% Method: sum differences of possible pixel to real circlepoint
%           Two-Point-Algorithm
%
% Octants:
%       y
%    \ 3|2 /
%  4  \ | /  1
%  -----|-----x
%  5  / | \  8
%    / 6|7 \
%
% Variabletypes for possible portation in C:
% function bresenham_circle
% Integer: coords, pixelstart, n1, n2, ..., n7, n8, choosenPixel1, ..., choosenPixel8,
%       nCoords, n, nDouble, nDouble12, nDouble34, nDouble56, nDouble78,
%       tmpcoords
% Boolean: doubledPoints
% Float: all others
%
```

```
% function calculateStartvaluesPerOctant
% Integer: octant, pixelstart, pixel, numberOfSteps, choosenPixel
% Float: all others
%
% function chooseInSlowDirection
% Integer: pixelDimension
%
% function choosenOnePoint
% Integer: x1, x2, y1, y2
% Float: all others
%
% Editor: Norbert Spiess 21.05.2010

if nargin<2
    error('Enter correct parameter.');
end
if nargin<3
    warning('No choice taken on doubled Points. Doubled Points
       allowed.');
    doubledPoints=0;
end

% get absolute radius to prevent of negative radius
radius = abs(radius);

% dislocation in both dimensions through centre
dislocX = mod(abs(centre(1)),floor(abs(centre(1))))*sign(centre(1)
   );
dislocY = mod(abs(centre(2)),floor(abs(centre(2))))*sign(centre(2)
   );

% if radius == 0 check on best pixel approximation around centre
if radius == 0
    coords = choosenOnePoint(centre);
    return
end

%% startcalculation
octantbordernorm = sind(45)*radius;
% borderpoints
% 1. + 2. octant
borderpoint12 = [1 1].*octantbordernorm+[dislocX dislocY];
```

```
% 3. + 4. octant
borderpoint34 = [-1 1].*octantbordernorm+[dislocX dislocY];
% 5. + 6. octant
borderpoint56 = [-1 -1].*octantbordernorm+[dislocX dislocY];
% 7. + 8. octant
borderpoint78 = [1 -1].*octantbordernorm+[dislocX dislocY];

% 2. octant
pixelstart = floor(dislocX) + 1;  %ceil without =
stepsizeN = pixelstart-dislocX;
realStart = dislocY+radius; % true startvalue for y in pixelgrid
[n2,choosenPixel2,error2] = calculateStartvaluesPerOctant(2,
    stepsizeN,realStart,pixelstart,borderpoint12,radius,dislocX,
    dislocY);
stepsize2=stepsizeN;

% 1. Octant
pixelstart = ceil(dislocY);
stepsizeN = pixelstart-dislocY;
realStart = dislocX+radius; % real value for x
[n1,choosenPixel1,error1] = calculateStartvaluesPerOctant(1,
    stepsizeN,realStart,pixelstart,borderpoint12,radius,dislocX,
    dislocY);
stepsize1=stepsizeN;

% 8. octant
pixelstart = ceil(dislocY) - 1;
stepsizeN = pixelstart - dislocY;
% realStart = dislocX+radius; % same value for y as on 1. octant
[n8,choosenPixel8,error8] = calculateStartvaluesPerOctant(8,
    stepsizeN,realStart,pixelstart,borderpoint78,radius,dislocX,
    dislocY);
stepsize8=stepsizeN;

% 7. Octant
pixelstart = floor(dislocX) + 1;
stepsizeN = pixelstart -dislocX;
realStart = dislocY-radius; % same value for y as on 6. octant
[n7,choosenPixel7,error7] = calculateStartvaluesPerOctant(7,
    stepsizeN,realStart,pixelstart,borderpoint78,radius,dislocX,
    dislocY);
stepsize7=stepsizeN;
```

```
                                                                    111
% 6. Octant                                                         112
pixelstart = floor(dislocX);                                        113
stepsizeN = pixelstart - dislocX;                                   114
% realStart = dislocY-radius; % real value for y                   115
[n6,choosenPixel6,error6] = calculateStartvaluesPerOctant(6,        116
    stepsizeN,realStart,pixelstart,borderpoint56,radius,dislocX,
    dislocY);
stepsize6=stepsizeN;                                                117
                                                                    118
% 5. octant                                                         119
pixelstart = ceil(dislocY) - 1;                                     120
stepsizeN = pixelstart - dislocY;                                   121
realStart = dislocX-radius; % same value for x as 4. octant        122
[n5,choosenPixel5,error5] = calculateStartvaluesPerOctant(5,        123
    stepsizeN,realStart,pixelstart,borderpoint56,radius,dislocX,
    dislocY);
stepsize5=stepsizeN;                                                124
                                                                    125
% 4. Octant                                                         126
pixelstart = ceil(dislocY);                                         127
stepsizeN = pixelstart - dislocY;                                   128
% realStart = dislocX-radius; % real value for x                   129
[n4,choosenPixel4,error4] = calculateStartvaluesPerOctant(4,        130
    stepsizeN,realStart,pixelstart,borderpoint34,radius,dislocX,
    dislocY);
stepsize4=stepsizeN;                                                131
                                                                    132
% 3. Octant                                                         133
pixelstart = floor(dislocX);                                        134
stepsizeN = pixelstart - dislocX;                                   135
realStart = dislocY+radius; % same value for y as 2. octant        136
[n3,choosenPixel3,error3] = calculateStartvaluesPerOctant(3,        137
    stepsizeN,realStart,pixelstart,borderpoint34,radius,dislocX,
    dislocY);
stepsize3=stepsizeN;                                                138
                                                                    139
%% prealloc                                                         140
nCoords=n1+n2+n3+n4+n5+n6+n7+n8;                                    141
coords = zeros(nCoords,2,'int32');                                  142
                                                                    143
%% additional handling if no coords are to calculate                144
```

```
% if n==0 check on best pixel approximation around centre
if nCoords==0
    coords  = choosenOnePoint(centre);
    return
end

%% 2. Octant
sumerror = error2;
for i=1:n2

    coords(i,1) = stepsize2+dislocX;
    coords(i,2) = choosenPixel2;
    if sumerror<0
        % add sumerror for only x-step
        sumerror = sumerror + 4*stepsize2 + 6;
    else
        % add sumerror for y-step
        sumerror = sumerror + 4*(stepsize2-(choosenPixel2-dislocY)
           ) + 10;
        % do y-step (step in slow direction)
        choosenPixel2 = choosenPixel2 - 1;
    end
  % do x-step (step in fast direction), inc stepsize
    stepsize2 = stepsize2+1;
end
n = n2;

%% 1. Octant
sumerror = error1;
for i=n+1:n+n1
  coords(i,1) = choosenPixel1;
    coords(i,2) = stepsize1+dislocY;
    if sumerror<0
        % add sumerror for only y-step
        sumerror = sumerror + 4*stepsize1 + 6;
    else
        % add sumerror for xy-step
        sumerror = sumerror + 4*(-(choosenPixel1-dislocX)+
           stepsize1) + 10;
        % do x-step (step in slow direction)
        choosenPixel1 = choosenPixel1 - 1;
    end
```

```
    % do y-step (step in fast direction), inc stepsize
    stepsize1 = stepsize1+1;
end
n=n+n1;

%% 8. Octant
sumerror = error8;
for i=n+1:n+n8
  coords(i,1) = choosenPixel8;
    coords(i,2) = stepsize8+dislocY;
    if sumerror<0
        % add sumerror for only y-step
        sumerror = sumerror - 4*stepsize8 + 6;
    else
        % add sumerror for xy-step
        sumerror = sumerror - 4*((choosenPixel8-dislocX)+stepsize8
           ) + 10;
        % do x-step (step in slow direction)
        choosenPixel8 = choosenPixel8 - 1;
    end
    % do y-step (step in fast direction), inc stepsize
    stepsize8 = stepsize8-1;
end
n=n+n8;

%% 7. Octant
sumerror = error7;
for i=n+1:n+n7
  coords(i,1) = stepsize7+dislocX;
    coords(i,2) = choosenPixel7;
    if sumerror<0
        % add sumerror for only x-step
        sumerror = sumerror + 4*stepsize7 + 6;
    else
        % add sumerror for y-step
        sumerror = sumerror + 4*(stepsize7+(choosenPixel7-dislocY)
           ) + 10;
        % do y-step (step in slow direction)
        choosenPixel7 = choosenPixel7 + 1;
    end
    % do x-step (step in fast direction), inc stepsize
    stepsize7 = stepsize7+1;
```

```
end
n = n+n7;

%% 6. Octant
sumerror = error6;
for i=n+1:n+n6
  coords(i,1) = stepsize6+dislocX;
    coords(i,2) = choosenPixel6;
    if sumerror<0
        % add sumerror for only x-step
        sumerror = sumerror - 4*stepsize6 + 6;
    else
        % add sumerror for y-step
        sumerror = sumerror - 4*(stepsize6-(choosenPixel6-dislocY)
           ) + 10;
        % do y-step (step in slow direction)
        choosenPixel6 = choosenPixel6 + 1;
    end
    % do x-step (step in fast direction), inc stepsize
    stepsize6 = stepsize6-1;
end
n = n + n6;

%% 5. Octant
sumerror = error5;
for i=n+1:n+n5
    coords(i,1) = choosenPixel5;
    coords(i,2) = stepsize5+dislocY;

    if sumerror<0
        % add sumerror for only y-step
        sumerror = sumerror - 4*stepsize5 + 6;
    else
        % add sumerror for xy-step
        sumerror = sumerror + 4*((choosenPixel5-dislocX)-stepsize5
           ) + 10;
        % do x-step (step in slow direction)
        choosenPixel5 = choosenPixel5 + 1;
    end
    % do y-step (step in fast direction), inc stepsize
    stepsize5 = stepsize5-1;
end
```

```
n = n+n5;

%% 4. Octant
sumerror = error4;
for i=n+1:n+n4
  coords(i,1) = choosenPixel4;
    coords(i,2) = stepsize4+dislocY;
    if sumerror<0
        % add error for only y-step
        sumerror = sumerror + 4*stepsize4 + 6;
    else
        % add error for xy-step
        sumerror = sumerror + 4*((choosenPixel4-dislocX)+stepsize4
           ) + 10;
        % do x-step (step in slow direction)
        choosenPixel4 = choosenPixel4 + 1;
    end
    % do y-step (step in fast direction), inc stepsize
    stepsize4 = stepsize4+1;
end
n=n+n4;

%% 3. Octant
sumerror = error3;
for i=n+1:n+n3
  coords(i,1) = stepsize3+dislocX;
    coords(i,2) = choosenPixel3;

    if sumerror<0
        % add sumerror for only x-step
        sumerror = sumerror - 4*stepsize3 + 6;
    else
        % add sumerror for y-step
        sumerror = sumerror - 4*(stepsize3+(choosenPixel3-dislocY)
           ) + 10;
        % do y-step (step in slow direction)
        choosenPixel3 = choosenPixel3 - 1;
    end
    % do x-step (step in fast direction), inc stepsize
    stepsize3 = stepsize3-1;
end
```

```
%% tail
if doubledPoints
    % Check and Copy to eliminate doubled Points.
    if radius < 3
        % check every coord on every other
        eliminate = zeros(ceil(size(coords,1)/2),1,'int32');
        n = 0;
        for i=1:size(coords,1)-1
            for j=i+1:size(coords,1)
                if coords(i,1)==coords(j,1) && coords(i,2)==coords
                    (j,2)
                    n = n + 1;
                    eliminate(n) = j;
                    break
                end
            end
        end
        eliminate(n+1:end) = [];
        if ~isempty(eliminate)
            eliminate = sort(eliminate);
            tmpCoords = zeros(size(coords,1)-n,2,'int32');
            n = eliminate(1)-1;
            tmpCoords(1:n,:) = coords(1:eliminate(1)-1,:);
            for i=2:size(eliminate,1)
                tmpCoords(n+1:eliminate(i)-i,:) = coords(eliminate
                    (i-1)+1:eliminate(i)-1,:);
                n=eliminate(i)-i;
            end
            if size(eliminate,1)>1
                tmpCoords(n+1:end,:) = coords(eliminate(i)+1:end
                    ,:);
            else
                tmpCoords(n+1:end,:) = coords(eliminate+1:end,:);
            end
            coords = tmpCoords;
        end
    else
        % check between 1.+2. octant
        nDouble12 = int32(0);
        for i=1:2
            x1 = coords(n2-i+1,1);
            y1 = coords(n2-i+1,2);
```

```
        for j=1:2
            x2 = coords(n2+n1-j+1,1);
            y2 = coords(n2+n1-j+1,2);
            if (x1==x2 && y1==y2)
                nDouble12 = nDouble12 + 1;
            end
        end
    end
    n=n1+n2;
    % check between 7.+8. octant
    nDouble78 = int32(0);
    for i=1:2
        x1 = coords(n+n8-i+1,1);
        y1 = coords(n+n8-i+1,2);
        for j=1:2
            x2 = coords(n+n8+n7-j+1,1);
            y2 = coords(n+n8+n7-j+1,2);
            if (x1==x2 && y1==y2)
                nDouble78 = nDouble78 + 1;
            end
        end
    end
    n=n+n8+n7;
    % check between 5.+6. octant
    nDouble56 = int32(0);
    for i=1:2
        x1 = coords(n+n6-i+1,1);
        y1 = coords(n+n6-i+1,2);
        for j=1:2
            x2 = coords(n+n6+n5-j+1,1);
            y2 = coords(n+n6+n5-j+1,2);
            if (x1==x2 && y1==y2)
                nDouble56 = nDouble56 + 1;
            end
        end
    end
    n=n+n6+n5;
    % check between 3.+4. octant
    nDouble34 = int32(0);
    for i=1:2
        x1 = coords(n+n4-i+1,1);
        y1 = coords(n+n4-i+1,2);
```

```
            for j=1:2
                x2 = coords(nCoords-j+1,1);
                y2 = coords(nCoords-j+1,2);
                if (x1==x2 && y1==y2)
                    nDouble34 = nDouble34 + 1;
                end
            end
        end
        % copy to eliminate doubled points
        tmpcoords = zeros(nCoords-nDouble12-nDouble34-nDouble56-
            nDouble78,2,'int32');
        n = n1 + n2;
        nDouble = nDouble12;
        tmpcoords(1:n-nDouble,1) = coords(1:n-nDouble,1);
        tmpcoords(1:n-nDouble,2) = coords(1:n-nDouble,2);

        tmpcoords(n-nDouble+1:n+n8+n7-nDouble-nDouble78,1) =
            coords(n+1:n+n8+n7-nDouble78,1);
        tmpcoords(n-nDouble+1:n+n8+n7-nDouble-nDouble78,2) =
            coords(n+1:n+n8+n7-nDouble78,2);
        n = n + n8 + n7;
        nDouble = nDouble + nDouble78;

        tmpcoords(n-nDouble+1:n+n6+n5-nDouble-nDouble56,1) =
            coords(n+1:n+n6+n5-nDouble56,1);
        tmpcoords(n-nDouble+1:n+n6+n5-nDouble-nDouble56,2) =
            coords(n+1:n+n6+n5-nDouble56,2);
        n = n + n5 + n6;
        nDouble = nDouble + nDouble56;

        tmpcoords(n-nDouble+1:n+n4+n3-nDouble-nDouble34,1) =
            coords(n+1:n+n4+n3-nDouble34,1);
        tmpcoords(n-nDouble+1:n+n4+n3-nDouble-nDouble34,2) =
            coords(n+1:n+n4+n3-nDouble34,2);

        coords = tmpcoords;
    end
end
% integer coordtransformation
coords(:,1) = coords(:,1) + int32(floor(abs(centre(1)))*sign(
    centre(1)));
```

```
coords(:,2) = coords(:,2) + int32(floor(abs(centre(2)))*sign(centre(2)));
% coords = int32(coords);
end

%% startcalculation
function [numberOfSteps,choosenPixel,starterror] = calculateStartvaluesPerOctant(octant, stepsize, realStart, pixelstart, borderpoint, radius, dislocX, dislocY)

if abs(stepsize)>=radius
  numberOfSteps = int32(0);
    choosenPixel = int32(0);
    starterror = 0;
    return
end

additionalDelta = radius - sin(acos(stepsize/radius))*radius; % additional distance in slow dimension for integer-position in fast Dimension

switch octant
    case 1
        % calculate number of possible steps
        pixel = ceil(borderpoint);
        if -((pixel(1)-dislocX)/(pixel(2)-dislocY)) > -1
            pixel(2) = pixel(2)-1;
        end
        numberOfSteps = int32(abs(pixel(2)-pixelstart)+1);
        % take choice on pixeldimension in slow direction
        realValue = realStart-additionalDelta;
        choosenPixel = chooseInSlowDirection(realValue);
        % calculate starterror
        starterror = 2*(stepsize+1)^2 - 2*radius^2 + (choosenPixel-dislocX)^2 + (choosenPixel-dislocX-1)^2;
    case 2
        % calculate number of possible steps
        pixel = ceil(borderpoint);
        if -((pixel(1)-dislocX)/(pixel(2)-dislocY)) <= -1
            pixel(1) = pixel(1)-1;
        end
        numberOfSteps = int32(abs(pixel(1)-pixelstart)+1);
```

```
        realValue = realStart-additionalDelta;                          455
        % take choice on pixeldimension in slow direction              456
        choosenPixel = chooseInSlowDirection(realValue);                457
        % calculate starterror                                          458
        starterror = 2*(stepsize+1)^2 - 2*radius^2 + (choosenPixel      459
           -dislocY)^2 + (choosenPixel-dislocY-1)^2;
    case 3                                                              460
        % calculate number of possible steps                            461
        pixel(2) = ceil(borderpoint(2));                                462
        pixel(1) = floor(borderpoint(1));                               463
        if -((pixel(1)-dislocX)/(pixel(2)-dislocY)) >= 1                464
            pixel(1) = pixel(1)+1;                                      465
        end                                                             466
        numberOfSteps = int32(abs(pixel(1)-pixelstart)+1);              467
        realValue = realStart-additionalDelta;                          468
        % take choice on pixeldimension in slow direction              469
        choosenPixel = chooseInSlowDirection(realValue);                470
        % calculate starterror                                          471
        starterror = 2*(stepsize-1)^2 - 2*radius^2 + (choosenPixel      472
           -dislocY)^2 + (choosenPixel-dislocY-1)^2;
    case 4                                                              473
        % calculate number of possible steps                            474
        pixel(2) = ceil(borderpoint(2));                                475
        pixel(1) = floor(borderpoint(1));                               476
        if -((pixel(1)-dislocX)/(pixel(2)-dislocY)) < 1                 477
            pixel(2) = pixel(2) - 1;                                    478
        end                                                             479
        numberOfSteps = int32(abs(pixel(2)-pixelstart)+1);              480
        realValue = realStart+additionalDelta;                          481
        % take choice on pixeldimension in slow direction              482
        choosenPixel = chooseInSlowDirection(realValue);                483
        % calculate starterror                                          484
        starterror = 2*(stepsize+1)^2 - 2*radius^2 + (choosenPixel      485
           -dislocX)^2 + (choosenPixel-dislocX+1)^2;
    case 5                                                              486
        % calculate number of possible steps                            487
        pixel = floor(borderpoint);                                     488
        if -((pixel(1)-dislocX)/(pixel(2)-dislocY)) > -1                489
            pixel(2) = pixel(2)+1;                                      490
        end                                                             491
        numberOfSteps = int32(abs(pixel(2)-pixelstart)+1);              492
        realValue = realStart+additionalDelta;                          493
```

```
        % take choice on pixeldimension in slow direction
        choosenPixel = chooseInSlowDirection(realValue);
        % calculate starterror
        starterror = 2*(stepsize-1)^2 - 2*radius^2 + (choosenPixel
            -dislocX)^2 + (choosenPixel-dislocX+1)^2;
    case 6
        % calculate number of possible steps
        pixel = floor(borderpoint);
        if -((pixel(1)-dislocX)/(pixel(2)-dislocY)) <= -1
            pixel(1) = pixel(1)+1;
        end
        numberOfSteps = int32(abs(pixel(1)-pixelstart)+1);
        realValue = realStart+additionalDelta;
        % take choice on pixeldimension in slow direction
        choosenPixel = chooseInSlowDirection(realValue);
        % calculate starterror
        starterror = 2*(stepsize-1)^2 - 2*radius^2 + (choosenPixel
            -dislocY)^2 + (choosenPixel-dislocY+1)^2;
    case 7
        % calculate number of possible steps
        pixel(2) = floor(borderpoint(2));
        pixel(1) = ceil(borderpoint(1));
        if -((pixel(1)-dislocX)/(pixel(2)-dislocY)) >= 1
            pixel(1) = pixel(1)-1;
        end
        numberOfSteps = int32(abs(pixel(1)-pixelstart)+1);
        realValue = realStart+additionalDelta;
        % take choice on pixeldimension in slow direction
        choosenPixel = chooseInSlowDirection(realValue);
        % calculate starterror
        starterror = 2*(stepsize+1)^2 - 2*radius^2 + (choosenPixel
            -dislocY)^2 + (choosenPixel-dislocY+1)^2;
    case 8
        % calculate number of possible steps
        pixel(2) = floor(borderpoint(2));
        pixel(1) = ceil(borderpoint(1));
        if -((pixel(1)-dislocX)/(pixel(2)-dislocY)) < 1
            pixel(2) = pixel(2) + 1;
        end
        numberOfSteps = int32(abs(pixel(2)-pixelstart)+1);
        realValue = realStart-additionalDelta;
        % take choice on pixeldimension in slow direction
```

```
        choosenPixel = chooseInSlowDirection(realValue);
        % calculate starterror
        starterror = 2*(stepsize-1)^2 - 2*radius^2 + (choosenPixel
            -dislocX)^2 + (choosenPixel-dislocX-1)^2;
end
end

function pixeldimension = chooseInSlowDirection(realValue)
% % % % % % % % % % %
% choose best pixeldimension in slow direction
% % % % % % % % % % %
dislocHighPoint = (ceil(realValue)-realValue);
dislocLowPoint = ((ceil(realValue)-1)-realValue);
if abs(dislocLowPoint) <= abs(dislocHighPoint)
    pixeldimension = realValue + dislocLowPoint;
else
    pixeldimension = realValue + dislocHighPoint;
end
end

function coords = choosenOnePoint(centre)
% % % % % % % % % % %
% choose Pixel with lowest distance
% cases:
% ^y
% |
% | 4     |    1
% |       |
% |  __Pixel____
% |       |
% | 3     |    2
% --------------> x
% % % % % % % % % % %
x1 = ceil(centre(1)); x2 = x1 - 1;
y1 = ceil(centre(2)); y2 = y1 - 1;
sumerror1 = norm([x1 y1]-centre);
sumerror2 = norm([x1 y2]-centre);
sumerror3 = norm([x2 y2]-centre);
sumerror4 = norm([x2 y1]-centre);
[sumerror,i] = min(sumerror1,min(sumerror2,min(sumerror3,sumerror4
    )));
switch i
```

```
    case 1                                                    573
        coords = [x1 y1];                                     574
    case 2                                                    575
        coords = [x1 y2];                                     576
    case 3                                                    577
        coords = [x2 y2];                                     578
    case 4                                                    579
        coords = [x2 y1];                                     580
end                                                           581
end                                                           582
```

Listing 4: Der Bresenham-Kreis-Algorithmus in MATLAB.[3]

E. Bresenham-Kugel-Algorithmus in MATLAB

```
function coords = bresenhamBall(centre,radius,doubledPoints)

if nargin<3
  warning('No choice taken on doubled Points. Doubled Points
     allowed.');
    doubledPoints=0;
end

radius = abs(radius);

if radius == 0
    coords = chooseOnePoint(centre);
    return
end
disloc(3) = mod(abs(centre(3)),floor(abs(centre(3))))*sign(centre
   (3));
disloc(2) = mod(abs(centre(2)),floor(abs(centre(2))))*sign(centre
   (2));
disloc(1) = mod(abs(centre(1)),floor(abs(centre(1))))*sign(centre
   (1));

stepsizeZ = (floor(disloc(3))+1) - disloc(3);

border2 = radius+disloc(3);
nZPos = int32(ceil(border2)+1);

border1 = radius-disloc(3);
nZNeg = int32(ceil(border1)+1);

minus = int32(0);
sliceData = zeros(nZPos+nZNeg,34);

for i=1:nZPos
  if stepsizeZ >= radius
       if abs(stepsizeZ+disloc(3) - border2) <= 0.5
           sliceData(i,1) = stepsizeZ;
     % no saving of radius required (cause of zeros)
           % sliceData(i,2) = 0;
           break
       else
```

```
            minus = minus + 1;
            break
        end
    else
        additionalDelta = radius - sin(acos(stepsizeZ/radius))*
            radius;
        radiusN=radius-additionalDelta;
        sliceData(i,:)= initStartCalc(radiusN,stepsizeZ,disloc
            (1:2),sliceData(i,:));
  end
  stepsizeZ = stepsizeZ + 1;
end

stepsizeZ = floor(disloc(3))-disloc(3);

for j=1:nZNeg
  if abs(stepsizeZ) >= radius
        if abs(abs(stepsizeZ+disloc(3)) - border1) <= 0.5
            sliceData(j+i-minus,1) = stepsizeZ;% sliceData(i,2) =
                0;
            break;
        else
            minus = minus + 1;
            break
        end
    else
        additionalDelta = radius - sin(acos(stepsizeZ/radius))*
            radius;
        radiusN=radius-additionalDelta;
        sliceData(j+i-minus,:)= initStartCalc(radiusN,stepsizeZ,
            disloc(1:2),sliceData(j+i-minus,:));
  end
    stepsizeZ = stepsizeZ-1;
end

steps = zeros(i+j-minus,1,'int32');
for i=1:size(steps,1)
    if sliceData(i,2) == 0
        steps(i) = 1;
    else
```

```
        steps(i) = sliceData(i,3)+sliceData(i,7)+sliceData(i,11)+
           sliceData(i,15)+sliceData(i,19)+sliceData(i,23)+
           sliceData(i,27)+sliceData(i,31);
    end
end

coords = zeros(sum(steps),3,'int32');

tmpCoords = callCircle(disloc,doubledPoints,sliceData(1,2),...
    sliceData(1,3),sliceData(1,4),sliceData(1,5),sliceData(1,6),
       sliceData(1,7),...
    sliceData(1,8),sliceData(1,9),sliceData(1,10),sliceData(1,11),
       sliceData(1,12),...
    sliceData(1,13),sliceData(1,14),sliceData(1,15),sliceData
       (1,16),sliceData(1,17),...
    sliceData(1,18),sliceData(1,19),sliceData(1,20),sliceData
       (1,21),...
    sliceData(1,22),sliceData(1,23),sliceData(1,24),sliceData
       (1,25),...
    sliceData(1,26),sliceData(1,27),sliceData(1,28),sliceData
       (1,29),...
    sliceData(1,30),sliceData(1,31),sliceData(1,32),sliceData
       (1,33),sliceData(1,34));
tmpCoords(:,3) = int32(sliceData(1,1)+disloc(3));
nCoords = size(tmpCoords,1);
nUnused = steps(1)-nCoords;
coords(1:nCoords,:) = tmpCoords;

% figure; plot3(coords(1:nCoords,1),coords(1:nCoords,2),coords(1:
   nCoords,3),'.'); drawnow;

for i=2:size(steps,1)
    if steps(i) == 1
        coords(nCoords+1,:) = chooseOnePoint(disloc);
        coords(nCoords+1,3) = sliceData(i,1)+disloc(3);
% figure; plot3(coords(nCoords+1,1),coords(nCoords+1,2),coords(
   nCoords+1,3),'.'); drawnow;
        nCoords = nCoords + 1;
    else
        tmpCoords = callCircle(disloc,doubledPoints,sliceData(i,2)
           ,...
```

```
            sliceData(i,3),sliceData(i,4),sliceData(i,5),sliceData(i,6),sliceData(i,7),...
            sliceData(i,8),sliceData(i,9),sliceData(i,10),sliceData(i,11),sliceData(i,12),...
            sliceData(i,13),sliceData(i,14),sliceData(i,15),sliceData(i,16),sliceData(i,17),...
            sliceData(i,18),sliceData(i,19),sliceData(i,20),sliceData(i,21),...
            sliceData(i,22),sliceData(i,23),sliceData(i,24),sliceData(i,25),...
            sliceData(i,26),sliceData(i,27),sliceData(i,28),sliceData(i,29),...
            sliceData(i,30),sliceData(i,31),sliceData(i,32),sliceData(i,33),sliceData(i,34));
        tmpCoords(:,3) = int32(sliceData(i,1)+disloc(3));
        n = int32(size(tmpCoords,1));

        coords(nCoords+1:nCoords+n,:) = tmpCoords;
    % figure; plot3(coords(nCoords+1:nCoords+n,1),coords(nCoords+1:nCoords+n,2),coords(nCoords+1:nCoords+n,3),'.');
       drawnow;
        nCoords = nCoords + n;
        nUnused = nUnused + steps(i)-nCoords;
    end
end
clear sliceData;
coords = coords(1:nCoords,:);


% errorVector = zeros(size(coords,1),1);
% for i=1:size(coords,1)
%     actualVoxelVect = double(coords(i,:));
%     errorVector(i,1) = norm(actualVoxelVect-(actualVoxelVect/norm(actualVoxelVect))*radius);
% end

coords(:,1) = coords(:,1) + int32(floor(abs(centre(1)))*sign(centre(1)));
coords(:,2) = coords(:,2) + int32(floor(abs(centre(2)))*sign(centre(2)));
coords(:,3) = coords(:,3) + int32(floor(abs(centre(3)))*sign(centre(3)));
```

```
% figure; plot3(coords(:,1),coords(:,2),coords(:,3),'.');
% zlabel('Z-Achse'); ylabel('Y-Achse'); xlabel('X-Achse');

end

%%
function coords = callCircle(disloc,doubledPoints,radius,...
    n1,choosenVoxel1,stepsize1,error1,n2,choosenVoxel2,stepsize2,
        error2,...
    n3,choosenVoxel3,stepsize3,error3,n4,choosenVoxel4,stepsize4,
        error4,...
    n5,choosenVoxel5,stepsize5,error5,n6,choosenVoxel6,stepsize6,
        error6,...
    n7,choosenVoxel7,stepsize7,error7,n8,choosenVoxel8,stepsize8,
        error8)
    if radius == 0
        coords = chooseOnePoint(disloc);
    else
        nCoords = n1+n2+n3+n4+n5+n6+n7+n8;
        coords = zeros(nCoords,2,'int32');
        % 2. octand
        sumerror = error2;
        for i=1:n2
            coords(i,1) = stepsize2+disloc(1);
            coords(i,2) = choosenVoxel2;
            if sumerror<0
                % add sumerror for only x-step
                sumerror = sumerror + 4*stepsize2 + 6;
            else
                % add sumerror for y-step
                sumerror = sumerror + 4*(stepsize2-(choosenVoxel2-
                    disloc(2))) + 10;
                % do y-step (step in slow direction)
                choosenVoxel2 = choosenVoxel2 - 1;
            end
            % do x-step (step in fast direction), inc stepsize
            stepsize2 = stepsize2+1;
        end
        n = n2;

        % 1. octand
        sumerror = error1;
```

```
for i=n+1:n+n1
    coords(i,1) = choosenVoxel1;
    coords(i,2) = stepsize1+disloc(2);
    if sumerror<0
        % add sumerror for only y-step
        sumerror = sumerror + 4*stepsize1 + 6;
    else
        % add sumerror for xy-step
        sumerror = sumerror + 4*(-(choosenVoxel1-disloc(1)
           )+stepsize1) + 10;
        % do x-step (step in slow direction)
        choosenVoxel1 = choosenVoxel1 - 1;
    end
    % do y-step (step in fast direction), inc stepsize
    stepsize1 = stepsize1+1;
end
n=n+n1;

% 8. octand
sumerror = error8;
for i=n+1:n+n8
    coords(i,1) = choosenVoxel8;
    coords(i,2) = stepsize8+disloc(2);
    if sumerror<0
        % add sumerror for only y-step
        sumerror = sumerror - 4*stepsize8 + 6;
    else
        % add sumerror for xy-step
        sumerror = sumerror - 4*((choosenVoxel8-disloc(1))
           +stepsize8) + 10;
        % do x-step (step in slow direction)
        choosenVoxel8 = choosenVoxel8 - 1;
    end
    % do y-step (step in fast direction), inc stepsize
    stepsize8 = stepsize8-1;
end
n=n+n8;

% 7. octand
sumerror = error7;
for i=n+1:n+n7
    coords(i,1) = stepsize7+disloc(1);
```

```
        coords(i,2) = choosenVoxel7;
        if sumerror<0
            % add sumerror for only x-step
            sumerror = sumerror + 4*stepsize7 + 6;
        else
            % add sumerror for y-step
            sumerror = sumerror + 4*(stepsize7+(choosenVoxel7-
                disloc(2))) + 10;
            % do y-step (step in slow direction)
            choosenVoxel7 = choosenVoxel7 + 1;
        end
        % do x-step (step in fast direction), inc stepsize
        stepsize7 = stepsize7+1;
    end
    n = n+n7;

    % 6. octand
    sumerror = error6;
    for i=n+1:n+n6
        coords(i,1) = stepsize6+disloc(1);
        coords(i,2) = choosenVoxel6;
        if sumerror<0
            % add sumerror for only x-step
            sumerror = sumerror - 4*stepsize6 + 6;
        else
            % add sumerror for y-step
            sumerror = sumerror - 4*(stepsize6-(choosenVoxel6-
                disloc(2))) + 10;
            % do y-step (step in slow direction)
            choosenVoxel6 = choosenVoxel6 + 1;
        end
        % do x-step (step in fast direction), inc stepsize
        stepsize6 = stepsize6-1;
    end
    n = n + n6;

    % 5. octand
    sumerror = error5;
    for i=n+1:n+n5
        coords(i,1) = choosenVoxel5;
        coords(i,2) = stepsize5+disloc(2);
```

```
        if sumerror<0
            % add sumerror for only y-step
            sumerror = sumerror - 4*stepsize5 + 6;
        else
            % add sumerror for xy-step
            sumerror = sumerror + 4*((choosenVoxel5-disloc(1))
                -stepsize5) + 10;
            % do x-step (step in slow direction)
            choosenVoxel5 = choosenVoxel5 + 1;
        end
        % do y-step (step in fast direction), inc stepsize
        stepsize5 = stepsize5-1;
    end
    n = n+n5;

    % 4. octand
    sumerror = error4;
    for i=n+1:n+n4
        coords(i,1) = choosenVoxel4;
        coords(i,2) = stepsize4+disloc(2);
        if sumerror<0
            % add error for only y-step
            sumerror = sumerror + 4*stepsize4 + 6;
        else
            % add error for xy-step
            sumerror = sumerror + 4*((choosenVoxel4-disloc(1))
                +stepsize4) + 10;
            % do x-step (step in slow direction)
            choosenVoxel4 = choosenVoxel4 + 1;
        end
        % do y-step (step in fast direction), inc stepsize
        stepsize4 = stepsize4+1;
    end
    n=n+n4;

    % 3. octand
    sumerror = error3;
    for i=n+1:n+n3
        coords(i,1) = stepsize3+disloc(1);
        coords(i,2) = choosenVoxel3;

        if sumerror<0
```

```
            % add sumerror for only x-step
            sumerror = sumerror - 4*stepsize3 + 6;
        else
            % add sumerror for y-step
            sumerror = sumerror - 4*(stepsize3+(choosenVoxel3-
                disloc(2))) + 10;
            % do y-step (step in slow direction)
            choosenVoxel3 = choosenVoxel3 - 1;
        end
        % do x-step (step in fast direction), inc stepsize
        stepsize3 = stepsize3-1;
    end

    % tail
    if doubledPoints
        % Check and Copy to eliminate doubled Points.
        if radius < 3
            % check every coord on every other
            eliminate = zeros(ceil(size(coords,1)/2),1,'int32'
                );
            n = 0;
            for i=1:size(coords,1)-1
                for j=i+1:size(coords,1)
                    if coords(i,1)==coords(j,1) && coords(i,2)
                        ==coords(j,2)
                        n = n + 1;
                        eliminate(n) = j;
                        break
                    end
                end
            end
            eliminate(n+1:end) = [];
            if ~isempty(eliminate)
                eliminate = sort(eliminate);
                tmpCoords = zeros(size(coords,1)-n,2,'int32');
                n = eliminate(1)-1;
                tmpCoords(1:n,:) = coords(1:eliminate(1)-1,:);
                for i=2:size(eliminate,1)
                    tmpCoords(n+1:eliminate(i)-i,:) = coords(
                        eliminate(i-1)+1:eliminate(i)-1,:);
                    n=eliminate(i)-i;
                end
```

```
                if size(eliminate,1)>1                                  325
                    tmpCoords(n+1:end,:) = coords(eliminate(i)          326
                        +1:end,:);
                else                                                    327
                    tmpCoords(n+1:end,:) = coords(eliminate+1:          328
                        end,:);
                end                                                     329
                coords = tmpCoords;                                     330
            end                                                         331
        else                                                            332
            % check between 1.+2. octand                               333
            nDouble12 = int32(0);                                       334
            for i=1:2                                                   335
                x1 = coords(n2-i+1,1);                                  336
                y1 = coords(n2-i+1,2);                                  337
                for j=1:2                                               338
                    x2 = coords(n2+n1-j+1,1);                           339
                    y2 = coords(n2+n1-j+1,2);                           340
                    if (x1==x2 && y1==y2)                               341
                        nDouble12 = nDouble12 + 1;                      342
                    end                                                 343
                end                                                     344
            end                                                         345
            n=n1+n2;                                                    346
            % check between 7.+8. octand                               347
            nDouble78 = int32(0);                                       348
            for i=1:2                                                   349
                x1 = coords(n+n8-i+1,1);                                350
                y1 = coords(n+n8-i+1,2);                                351
                for j=1:2                                               352
                    x2 = coords(n+n8+n7-j+1,1);                         353
                    y2 = coords(n+n8+n7-j+1,2);                         354
                    if (x1==x2 && y1==y2)                               355
                        nDouble78 = nDouble78 + 1;                      356
                    end                                                 357
                end                                                     358
            end                                                         359
            n=n+n8+n7;                                                  360
            % check between 5.+6. octand                               361
            nDouble56 = int32(0);                                       362
            for i=1:2                                                   363
                x1 = coords(n+n6-i+1,1);                                364
```

```
    y1 = coords(n+n6-i+1,2);
    for j=1:2
        x2 = coords(n+n6+n5-j+1,1);
        y2 = coords(n+n6+n5-j+1,2);
        if (x1==x2 && y1==y2)
            nDouble56 = nDouble56 + 1;
        end
    end
end
n=n+n6+n5;
% check between 3.+4. octand
nDouble34 = int32(0);
for i=1:2
    x1 = coords(n+n4-i+1,1);
    y1 = coords(n+n4-i+1,2);
    for j=1:2
        x2 = coords(nCoords-j+1,1);
        y2 = coords(nCoords-j+1,2);
        if (x1==x2 && y1==y2)
            nDouble34 = nDouble34 + 1;
        end
    end
end
% copy to eliminate doubled points
tmpcoords = zeros(nCoords-nDouble12-nDouble34-
   nDouble56-nDouble78,2,'int32');
n = n1 + n2;
nDouble = nDouble12;
tmpcoords(1:n-nDouble,1) = coords(1:n-nDouble,1);
tmpcoords(1:n-nDouble,2) = coords(1:n-nDouble,2);

tmpcoords(n-nDouble+1:n+n8+n7-nDouble-nDouble78,1)
    = coords(n+1:n+n8+n7-nDouble78,1);
tmpcoords(n-nDouble+1:n+n8+n7-nDouble-nDouble78,2)
    = coords(n+1:n+n8+n7-nDouble78,2);
n = n + n8 + n7;
nDouble = nDouble + nDouble78;

tmpcoords(n-nDouble+1:n+n6+n5-nDouble-nDouble56,1)
    = coords(n+1:n+n6+n5-nDouble56,1);
tmpcoords(n-nDouble+1:n+n6+n5-nDouble-nDouble56,2)
    = coords(n+1:n+n6+n5-nDouble56,2);
```

```
                n = n + n5 + n6;                                        402
                nDouble = nDouble + nDouble56;                          403
                                                                        404
                tmpcoords(n-nDouble+1:n+n4+n3-nDouble-nDouble34,1)      405
                    = coords(n+1:n+n4+n3-nDouble34,1);
                tmpcoords(n-nDouble+1:n+n4+n3-nDouble-nDouble34,2)      406
                    = coords(n+1:n+n4+n3-nDouble34,2);
                                                                        407
                coords = tmpcoords;                                     408
            end                                                         409
        end                                                             410
    end                                                                 411
end                                                                     412
                                                                        413
%%                                                                      414
function sliceData = initStartCalc(radius,stepsize,dislocXY,           415
    sliceData,borderpoints)
octandbordernorm = sind(45)*radius;                                     416
% borderpoints                                                          417
% 7. + 8. octand                                                        418
borderpoints(4,:) = [1 -1].*octandbordernorm+dislocXY;                  419
% 5. + 6. octand                                                        420
borderpoints(3,:) = [-1 -1].*octandbordernorm+dislocXY;                 421
% 3. + 4. octand                                                        422
borderpoints(2,:) = [-1 1].*octandbordernorm+dislocXY;                  423
% 1. + 2. octand                                                        424
borderpoints(1,:) = [1 1].*octandbordernorm+dislocXY;                   425
                                                                        426
sliceData(1,1) = stepsize;                                              427
sliceData(1,2) = radius;                                                428
% 2. octand                                                             429
pixelstart = floor(dislocXY(1)) + 1;  %ceil OHNE =                      430
stepsize = pixelstart-dislocXY(1);                                      431
realStart = dislocXY(2)+radius; %wahres natuerliches y am start        432
[n2,choosenVoxel2,error2] = calculateStartvaluesPeroctand(2,           433
    dislocXY,radius,stepsize,realStart,pixelstart,borderpoints
    (1,:));
stepsize2=stepsize;                                                     434
sliceData(1,7) = n2; sliceData(1,8) = choosenVoxel2; sliceData          435
    (1,9) = stepsize2; sliceData(1,10) = error2;
% 1. octand                                                             436
pixelstart = ceil(dislocXY(2));                                         437
```

```
stepsize = pixelstart-dislocXY(2);
realStart = dislocXY(1)+radius; % real value for x
[n1,choosenVoxel1,error1] = calculateStartvaluesPeroctand(1,
    dislocXY,radius,stepsize,realStart,pixelstart,borderpoints
    (1,:));
stepsize1=stepsize;
sliceData(1,3) = n1; sliceData(1,4) = choosenVoxel1; sliceData
    (1,5) = stepsize1; sliceData(1,6) = error1;
% 8. octand
pixelstart = ceil(dislocXY(2)) - 1;
stepsize = pixelstart - dislocXY(2);
% realStart = dislocXY(1)+radius; % same value for y as on 1.
    octand
[n8,choosenVoxel8,error8] = calculateStartvaluesPeroctand(8,
    dislocXY,radius,stepsize,realStart,pixelstart,borderpoints
    (4,:));
stepsize8=stepsize;
sliceData(1,31) = n8; sliceData(1,32) = choosenVoxel8; sliceData
    (1,33) = stepsize8; sliceData(1,34) = error8;
% 7. octand
pixelstart = floor(dislocXY(1)) + 1;
stepsize = pixelstart -dislocXY(1);
realStart = dislocXY(2)-radius; % same value for y as on 6. octand
[n7,choosenVoxel7,error7] = calculateStartvaluesPeroctand(7,
    dislocXY,radius,stepsize,realStart,pixelstart,borderpoints
    (4,:));
stepsize7=stepsize;
sliceData(1,27) = n7; sliceData(1,28) = choosenVoxel7; sliceData
    (1,29) = stepsize7; sliceData(1,30) = error7;
% 6. octand
pixelstart = floor(dislocXY(1));
stepsize = pixelstart - dislocXY(1);
% realStart = dislocXY(2)-radius; % real value for y
[n6,choosenVoxel6,error6] = calculateStartvaluesPeroctand(6,
    dislocXY(1:2),radius,stepsize,realStart,pixelstart,
    borderpoints(3,:));
stepsize6=stepsize;
sliceData(1,23) = n6; sliceData(1,24) = choosenVoxel6; sliceData
    (1,25) = stepsize6; sliceData(1,26) = error6;
% 5. octand
pixelstart = ceil(dislocXY(2)) - 1;
stepsize = pixelstart - dislocXY(2);
```

```
realStart = dislocXY(1)-radius; % same value for x as 4. octand
[n5,choosenVoxel5,error5] = calculateStartvaluesPeroctand(5,
    dislocXY,radius,stepsize,realStart,pixelstart,borderpoints
    (3,:));
stepsize5=stepsize;
sliceData(1,19) = n5; sliceData(1,20) = choosenVoxel5; sliceData
    (1,21) = stepsize5; sliceData(1,22) = error5;
% 4. octand
pixelstart = ceil(dislocXY(2));
stepsize = pixelstart - dislocXY(2);
% realStart = dislocXY(1)-radius; % real value for x
[n4,choosenVoxel4,error4] = calculateStartvaluesPeroctand(4,
    dislocXY,radius,stepsize,realStart,pixelstart,borderpoints
    (2,:));
stepsize4=stepsize;
sliceData(1,15) = n4; sliceData(1,16) = choosenVoxel4; sliceData
    (1,17) = stepsize4; sliceData(1,18) = error4;
% 3. octand
pixelstart = floor(dislocXY(1));
stepsize = pixelstart - dislocXY(1);
realStart = dislocXY(2)+radius; % same value for y as 2. octand
[n3,choosenVoxel3,error3] = calculateStartvaluesPeroctand(3,
    dislocXY,radius,stepsize,realStart,pixelstart,borderpoints
    (2,:));
stepsize3=stepsize;
sliceData(1,11) = n3; sliceData(1,12) = choosenVoxel3; sliceData
    (1,13) = stepsize3; sliceData(1,14) = error3;
end

%% startcalculation
function [numberOfSteps,choosenVoxel,starterror] =
    calculateStartvaluesPeroctand(octand,disloc,radius,stepsize,
    realStart,pixelstart,borderpoint)
% % % % % % % % % % %
% calculation of stepnumber, starterror and choosenVoxel per
    octand
% % % % % % % % % % %
if abs(stepsize)>= radius
  numberOfSteps = int32(0);
    choosenVoxel = int32(0);
    starterror = 0;
    return
```

```
end

additionalDelta = radius - sin(acos(stepsize/radius))*radius; % additional distance in slow dimension for integer-position in fast Dimension

switch octand
    case 1
        % calculate number of possible steps
        pixel = ceil(borderpoint);
        if -((pixel(1)-disloc(1))/(pixel(2)-disloc(2))) > -1
            pixel(2) = pixel(2)-1;
        end
        numberOfSteps = int32(abs(pixel(2)-pixelstart)+1);
        % take choice on pixeldimension in slow direction
        realValue = realStart-additionalDelta;
        choosenVoxel = chooseInSlowDirection(realValue);
        % calculate starterror
        starterror = 2*(stepsize+1)^2 - 2*radius^2 + (choosenVoxel-disloc(1))^2 + (choosenVoxel-disloc(1)-1)^2;
    case 2
        % calculate number of possible steps
        pixel = ceil(borderpoint);
        if -((pixel(1)-disloc(1))/(pixel(2)-disloc(2))) <= -1
            pixel(1) = pixel(1)-1;
        end
        numberOfSteps = int32(abs(pixel(1)-pixelstart)+1);
        realValue = realStart-additionalDelta;
        % take choice on pixeldimension in slow direction
        choosenVoxel = chooseInSlowDirection(realValue);
        % calculate starterror
        starterror = 2*(stepsize+1)^2 - 2*radius^2 + (choosenVoxel-disloc(2))^2 + (choosenVoxel-disloc(2)-1)^2;
    case 3
        % calculate number of possible steps
        pixel(2) = ceil(borderpoint(2));
        pixel(1) = floor(borderpoint(1));
        if -((pixel(1)-disloc(1))/(pixel(2)-disloc(2))) >= 1
            pixel(1) = pixel(1)+1;
        end
        numberOfSteps = int32(abs(pixel(1)-pixelstart)+1);
        realValue = realStart-additionalDelta;
```

```
        % take choice on pixeldimension in slow direction
        choosenVoxel = chooseInSlowDirection(realValue);
        % calculate starterror
        starterror = 2*(stepsize-1)^2 - 2*radius^2 + (choosenVoxel
            -disloc(2))^2 + (choosenVoxel-disloc(2)-1)^2;
    case 4
        % calculate number of possible steps
        pixel(2) = ceil(borderpoint(2));
        pixel(1) = floor(borderpoint(1));
        if -((pixel(1)-disloc(1))/(pixel(2)-disloc(2))) < 1
            pixel(2) = pixel(2) - 1;
        end
        numberOfSteps = int32(abs(pixel(2)-pixelstart)+1);
        realValue = realStart+additionalDelta;
        % take choice on pixeldimension in slow direction
        choosenVoxel = chooseInSlowDirection(realValue);
        % calculate starterror
        starterror = 2*(stepsize+1)^2 - 2*radius^2 + (choosenVoxel
            -disloc(1))^2 + (choosenVoxel-disloc(1)+1)^2;
    case 5
        % calculate number of possible steps
        pixel = floor(borderpoint);
        if -((pixel(1)-disloc(1))/(pixel(2)-disloc(2))) > -1
            pixel(2) = pixel(2)+1;
        end
        numberOfSteps = int32(abs(pixel(2)-pixelstart)+1);
        realValue = realStart+additionalDelta;
        % take choice on pixeldimension in slow direction
        choosenVoxel = chooseInSlowDirection(realValue);
        % calculate starterror
        starterror = 2*(stepsize-1)^2 - 2*radius^2 + (choosenVoxel
            -disloc(1))^2 + (choosenVoxel-disloc(1)+1)^2;
    case 6
        % calculate number of possible steps
        pixel = floor(borderpoint);
        if -((pixel(1)-disloc(1))/(pixel(2)-disloc(2))) <= -1
            pixel(1) = pixel(1)+1;
        end
        numberOfSteps = int32(abs(pixel(1)-pixelstart)+1);
        realValue = realStart+additionalDelta;
        % take choice on pixeldimension in slow direction
        choosenVoxel = chooseInSlowDirection(realValue);
```

```
        % calculate starterror                                                  574
        starterror = 2*(stepsize-1)^2 - 2*radius^2 + (choosenVoxel             575
           -disloc(2))^2 + (choosenVoxel-disloc(2)+1)^2;
    case 7                                                                      576
        % calculate number of possible steps                                   577
        pixel(2) = floor(borderpoint(2));                                       578
        pixel(1) = ceil(borderpoint(1));                                        579
        if -((pixel(1)-disloc(1))/(pixel(2)-disloc(2))) >= 1                    580
            pixel(1) = pixel(1)-1;                                              581
        end                                                                     582
        numberOfSteps = int32(abs(pixel(1)-pixelstart)+1);                      583
        realValue = realStart+additionalDelta;                                  584
        % take choice on pixeldimension in slow direction                       585
        choosenVoxel = chooseInSlowDirection(realValue);                        586
        % calculate starterror                                                  587
        starterror = 2*(stepsize+1)^2 - 2*radius^2 + (choosenVoxel              588
           -disloc(2))^2 + (choosenVoxel-disloc(2)+1)^2;
    case 8                                                                      589
        % calculate number of possible steps                                   590
        pixel(2) = floor(borderpoint(2));                                       591
        pixel(1) = ceil(borderpoint(1));                                        592
        if -((pixel(1)-disloc(1))/(pixel(2)-disloc(2))) < 1                     593
            pixel(2) = pixel(2) + 1;                                            594
        end                                                                     595
        numberOfSteps = int32(abs(pixel(2)-pixelstart)+1);                      596
        realValue = realStart-additionalDelta;                                  597
        % take choice on pixeldimension in slow direction                       598
        choosenVoxel = chooseInSlowDirection(realValue);                        599
        % calculate starterror                                                  600
        starterror = 2*(stepsize-1)^2 - 2*radius^2 + (choosenVoxel              601
           -disloc(1))^2 + (choosenVoxel-disloc(1)-1)^2;
end                                                                             602
end                                                                             603
                                                                                604
function voxeldimension = chooseInSlowDirection(realValue)                     605
% % % % % % % % % % %                                                          606
% choose best pixeldimension in slow direction                                  607
% % % % % % % % % % %                                                          608
dislocHighPoint = (ceil(realValue)-realValue);                                  609
dislocLowPoint = ((ceil(realValue)-1)-realValue);                               610
if abs(dislocLowPoint) <= abs(dislocHighPoint)                                  611
    voxeldimension = realValue + dislocLowPoint;                                612
```

```
else
    voxeldimension = realValue + dislocHighPoint;
end
end

%%
function coords = chooseOnePoint(centre)
  x1 = ceil(centre(1)); x2 = x1 - 1;
    y1 = ceil(centre(2)); y2 = y1 - 1;
    z1 = ceil(centre(3)); z2 = z1 - 1;

    sumerror1 = norm([x1 y1 z1]-centre);
    sumerror2 = norm([x1 y2 z1]-centre);
    sumerror3 = norm([x2 y2 z1]-centre);
    sumerror4 = norm([x2 y1 z1]-centre);
    sumerror5 = norm([x1 y1 z2]-centre);
    sumerror6 = norm([x1 y2 z2]-centre);
    sumerror7 = norm([x2 y2 z2]-centre);
    sumerror8 = norm([x2 y1 z2]-centre);

    [sumerror i] = min(sumerror1,min(sumerror2,min(sumerror3,min(
       sumerror4,min(sumerror5,min(sumerror6,min(sumerror7,
       sumerror8)))))));
    switch i
        case 1
          coords = [x1 y1 z1];
        case 2
      coords = [x1 y2 z1];
        case 3
            coords = [x2 y2 z1];
        case 4
      coords = [x2 y1 z1];
        case 5
          coords = [x1 y1 z2];
        case 6
      coords = [x1 y2 z2];
        case 7
            coords = [x2 y2 z2];
        case 8
      coords = [x2 y1 z2];
    end
end
```

Listing 5: Der Bresenham-Kugel-Algorithmus in MATLAB.[3]

F. Metrikwerte der Kreisrasterung nach Brute-Force

Testnummer	**Metrik in der Einheit Pixel**			
	$error_{\perp max}$	$\overline{error}_{med}$	$\overline{error}_{arithm}$	$\overline{error}_{quadr}$
TestKr01	0,000	0,000	0,000	0,000
TestKr02	0,141	0,141	0,141	0,141
TestKr03	0,141	0,141	0,141	0,141
TestKr04	0,141	0,141	0,141	0,141
TestKr05	0,141	0,141	0,141	0,141
TestKr06	0,707	0,707	0,707	0,707
TestKr07	0,297	0,000	0,002	0,220
TestKr08	0,182	0,151	0,191	0,298
TestKr09	0,456	0,023	0,020	0,261
TestKr10	0,454	0,047	0,052	0,234
TestKr11	0,379	0,100	0,019	0,256
TestKr12	0,411	0,119	0,048	0,272
TestKr13	0,438	0,046	0,266	0,279
TestKr14	0,438	0,046	0,266	0,279
TestKr15	0,200	0,200	0,200	0,200
TestKr16	0,500	0,500	0,500	0,500
TestKr17	0,800	0,800	0,600	0,663
TestKr18	0,304	0,205	0,205	0,228
TestKr19	0,304	0,205	0,205	0,228
TestKr20	0,562	0,562	0,449	0,476

Tabelle 1: Metrikwerte des Brute-Force-Kreis-Algorithmus

G. Metrikwerte der Kreisrasterung nach Brute-Force

Testnummer	**Metrik in der Einheit Pixel**			
	$error_{\perp max}$	$\overline{error}_{med}$	$\overline{error}_{arithm}$	$\overline{error}_{quadr}$
TestKr01	0,000	0,000	0,000	0,000
TestKr02	0,141	0,141	0,141	0,141
TestKr03	0,141	0,141	0,141	0,141
TestKr04	0,141	0,141	0,141	0,141
TestKr05	0,141	0,141	0,141	0,141
TestKr06	0,707	0,707	0,707	0,707
TestKr07	0,440	0,134	0,181	0,221
TestKr08	0,491	0,182	0,248	0,298
TestKr09	0,456	0,259	0,223	0,260
TestKr10	0,454	0,180	0,195	0,233
TestKr11	0,379	0,236	0,229	0,250
TestKr12	0,411	0,236	0,248	0,271
TestKr13	0,479	0,258	0,246	0,279
TestKr14	0,479	0,258	0,246	0,279
TestKr15	0,200	0,200	0,200	0,200
TestKr16	0,914	0,500	0,479	0,578
TestKr17	1,214	1,214	0,809	0,991
TestKr18	0,305	0,205	0,205	0,228
TestKr19	0,305	0,205	0,205	0,228
TestKr20	0,424	0,424	0,424	0,424

Tabelle 2: Metrikwerte der Kreisrasterung nach Brute-Force

H. Metrikwerte der Kugelrasterung nach Brute-Force

Testnummer	**Metrik in der Einheit Pixel**			
	$error_{\perp max}$	$\overline{error}_{med}$	$\overline{error}_{arithm}$	$\overline{error}_{quadr}$
TestKr01	0,000	0,000	0,000	0,000
TestKr02	0,173	0,173	0,173	0,173
TestKr03	0,173	0,173	0,173	0,173
TestKr04	0,173	0,173	0,173	0,173
TestKr05	0,173	0,173	0,173	0,173
TestKr06	0,173	0,173	0,173	0,173
TestKr07	0,173	0,173	0,173	0,173
TestKr08	0,173	0,173	0,173	0,173
TestKr09	0,173	0,173	0,173	0,173
TestKr10	0,866	0,866	0,866	0,866
TestKr11	0,472	0,123	0,088	0,287
TestKr12	0,775	0,355	0,210	0,409
TestKr13	0,530	0,041	0,073	0,327

Tabelle 3: Metrikwerte der Kugelrasterung nach Brute-Force

I. Metrikwerte des Bresenham-Kugel-Algorithmus

Testnummer	**Metrik in der Einheit Pixel**			
	$error_{\perp max}$	$\overline{error}_{med}$	$\overline{error}_{arithm}$	$\overline{error}_{quadr}$
TestKr01	0,000	0,000	0,000	0,000
TestKr02	0,173	0,173	0,173	0,173
TestKr03	0,173	0,173	0,173	0,173
TestKr04	0,173	0,173	0,173	0,173
TestKr05	0,173	0,173	0,173	0,173
TestKr06	0,173	0,173	0,173	0,173
TestKr07	0,173	0,173	0,173	0,173
TestKr08	0,173	0,173	0,173	0,173
TestKr09	0,173	0,173	0,173	0,173
TestKr10	0,866	0,866	0,866	0,866
TestKr11	0,394	0,243	0,243	0,236
TestKr12	0,359	0,130	0,130	0,251
TestKr13	0,293	0,041	0,041	0,159

Tabelle 4: Metrikwerte des Bresenham-Kugel-Algorithmus