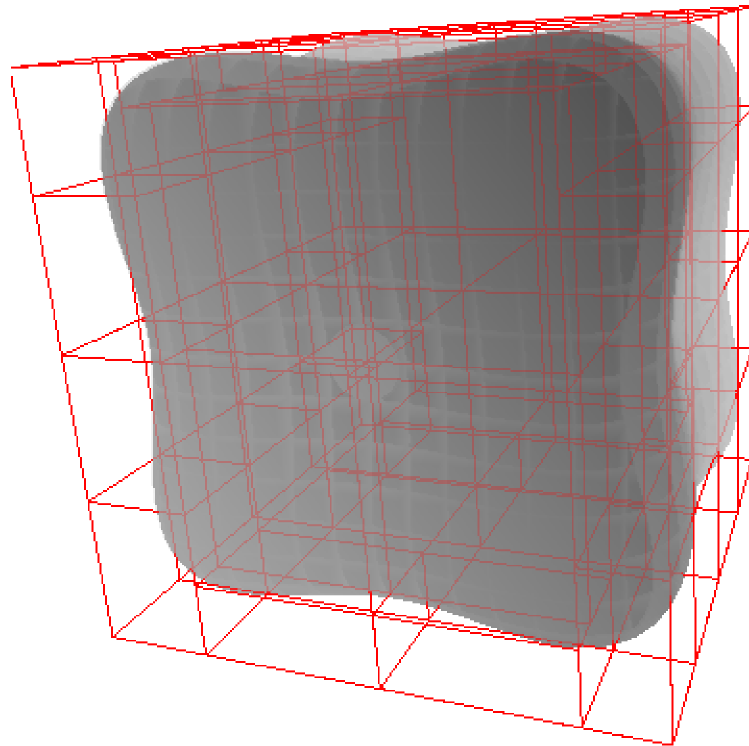
Raycasting
von adaptiven Isoflächen



Philip Frey

Semesterarbeit
Sommersemester 2005

Computer Graphics Laboratory
Departement Informatik
ETH Zürich

Prof. Dr. Markus Gross
Christian Sigg

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Zusammenfassung

Diese Semesterarbeit hat zum Ziel, einen Software basierten Raycaster zu implementieren. Als Szene soll eine adaptive Isofläche verwendet werden. Konkret wird die Oberflächeninformation in Form eines Gitters, das Datenwerte enthält, repräsentiert. Gespeichert werden nur die Datenwerte, die etwas zur Isoflächendefinition beitragen um den Speicherbedarf gering zu halten. Als Beschleunigungsstruktur, um innerhalb des Gitters das Sampling vorzunehmen, dient ein KD-Baum. Dieser ist insbesondere wichtig, weil nicht auf eine Hardwarebeschleunigung zurückgegriffen wird. Um komplexe Oberflächen einfacher darstellen zu können, wird die Constructive Solid Geometry unterstützt, deren boolsche Bedingungen sich mit einfachen min/max Operationen auswerten lassen. Die Beleuchtung wird mittels Phong Shading implementiert um ein möglichst realistisch wirkendes Bild zu erzeugen.

Abstract

The goal of this semester thesis is the implementation of a software based raycaster. The scene is represented using an adaptive isosurface. This means that the data to be sampled is stored in a regular data grid. In order to minimize the necessary storage space only data that is really needed to define the isosurface is kept in there. Since we do not use any kind of hardware acceleration a special software acceleration structure called a kd-tree is applied. In addition to that we support the concept of constructive solid geometry to allow an easier and more accurate modelling of rather complex surfaces. With isosurfaces and raycasting this is straightforward since the boolean conditions are mere min/max analyses. The shading is implemented using the common and quite realistic approach proposed by Phong.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Philip Frey
Semesterarbeit Sommersemester 2005

Raycasting von adaptiven Isoflächen

Einleitung

Implizite Flächen bieten für viele Aufgaben in der Modellierung eine geeignete Alternative zu Deiecksnetzen. Der immense Speicherbedarf von impliziten Repräsentationen macht ein adaptives Sampling bei komplexeren Modellen unumgänglich. Um scharfe Kanten auch mit reduzierter Samplingdichte eindeutig definieren zu können, wird das Objekt als boolscher Ausdruck von mehreren Teilobjekten verstanden.

Aufgabenstellung

Ziel dieser Arbeit ist es, einen Renderer für implizite Flächen zu implementieren, welcher adaptives Sampling und fliegende Auswertung von boolschen Operationen unterstützt. Der Renderer soll dabei auf einem Software Raycasting Ansatz basieren. Dazu wird für jedes Pixel ein Sichstrahl in das Volumen verfolgt und die skalare Funktion in variablen Intervallen ausgewertet. Die boolschen Besingungen lassen sich mit einfachen min/max Operationen auswerten. Folgende Funktionen müssen für den Algorithmus implementiert werden:

- Kd-Baum zur schnellen Suche von Strahl/Zellen-Intersektion.
- Sampling des Strahlensegments innerhalb einer Zelle.
- Auswertung des boolschen Ausdrucks zur Detektion einer Isoflächen-Intersektion.
- Beleuchtung der Isoflächen mit dem Phong Modell.

Ein Framework für implizite Flächen wird zur Verfügung gestellt.

Bemerkungen

Ein schriftlicher Bericht und eine mündliche Präsentation schliessen die Arbeit ab. Die Diplomarbeit steht unter der Obhut von Prof. Markus Gross.

Daten

Start der Semesterarbeit: 1.4.2005; Ende der Semesterarbeit: 1.7.2004

Kontakt

Christian Sigg / IFW D28.8 / Tel. 632 74 76 / siggc@inf.ethz.ch

1 Zusammenfassung . . . iii

2 Abstract . . . v

1 Einleitung . . . 1
1.1 Motivation und Hintergrund . . . 1
1.2 Gliederung . . . 2

2 Beschleunigungsstruktur . . . 3
2.1 Implizite Isoflächen . . . 3
2.1.1 Einführung . . . 3
2.1.2 Motivation . . . 4
2.1.3 Gitterdarstellung der Isofläche . . . 4
2.2 KD-Baum . . . 4
2.2.1 Motivation . . . 4
2.2.2 Die Beschleunigungsstruktur im Detail . . . 5
2.2.3 Aufbau . . . 6
2.2.3.1 Traversierung . . . 7
2.2.3.2 Alternative Beschleunigungsstruktur: Der Octree . . . 8

3 Visualisierung . . . 9
3.1 Software Raycaster . . . 9
3.1.1 Einführung: Das Prinzip des Raycasting . . . 9
3.1.2 Sichtstrahlen . . . 10
3.1.3 Intersektion . . . 11
3.1.4 Traversieren des KD-Baumes . . . 11
3.1.5 Sampling innerhalb einer CSG Zelle . . . 13
3.2 Constructive Solid Geometry . . . 14
3.2.1 Einführung: Kombination mehrerer Objekte . . . 14
3.2.2 De Morgansche Gesetze . . . 14
3.2.3 Beispiele . . . 15
3.2.4 Repräsentation und Auswertung in unserem Framework . . . 15
3.3 Phong Shading . . . 16
3.3.1 Die Phong Gleichung . . . 16
3.3.2 Veranschaulichung durch Beispiele . . . 17
3.4 Interpolation . . . 17
3.4.1 Interpolation fehlender Gitterdaten . . . 17
3.4.2 Trilineare Interpolation . . . 18
3.4.3 Probleme in Randbereichen . . . 19
3.5 Das Framework . . . 19

4 Implementation . . . 21
4.1 Zentrale Klassen . . . 21
4.1.1 Zellen . . . 21
4.1.2 KD-Baum . . . 21
4.2 Sichtstrahlen bestimmen . . . 22
4.3 Programmatischer Ablauf . . . 23
4.3.1 Call Hierarchy: Vom Zahlengitter zum gerenderten Bild . . . 23
4.3.1.1 Daten einlesen . . . 23
4.3.1.2 Rendering . . . 23
4.4 Verwendete Technologien . . . 23

5 Resultate . . . 25

5.1 KD-Baum 25
5.2 Gerenderte Isoflächen 26
5.2.1 Grayscale depth image 26
5.2.2 Phong Shading 27
5.3 Benchmarking 27
5.4 Anregungen für weiterführende Arbeiten 28
5.4.1 Rendering 28
5.4.2 Beschleunigungsstruktur 28

A Referenzen 29

1

Einleitung

1.1 Motivation und Hintergrund

Die rasante Weiterentwicklung der Computer ist nirgends so gut sichtbar wie in der Welt der Graphik. Die Objekte werden immer detaillierter und komplexer.

Ein gebräuchlicher Ansatz um dreidimensionale Objekte zu repräsentieren besteht darin, deren Oberflächen zu triangulieren. Sie wird durch ein Netz von Dreiecken dargestellt. Mit zunehmender Detailtreue werden diese Dreiecke immer kleiner und die Triangulation somit immer schwieriger und aufwändiger.

Eine Alternative zu den Dreiecksnetzen bildet die Repräsentation der Oberfläche durch eine implizite Darstellung. Dies ist der Ansatz, der in dieser Arbeit verfolgt wird. Implizite Flächen werden durch skalare Funktionen definiert. Diese Art der Darstellung lässt sich sehr gut und direkt für verschiedene Bereiche der Computergraphik nutzen (z.B. 3D Scanner). Eine ausführliche Erklärung folgt in Kapitel 2. Doch auch die Methode der impliziten Flächen kommt leider nicht ganz problemlos daher. Für komplexe Modelle wird auch hier sehr viel Speicher benötigt. Dies macht ein adaptives Sampling unumgänglich. In dieser Arbeit wird die implizite Fläche durch ein Gitter von Samplingwerten repräsentiert und darauf ein Interpolationsschema definiert.

Ein zentraler Teil der Arbeit stellt die Implementation einer Beschleunigungsstruktur in Form eines KD-Baumes dar. Es geht dabei, wie der Name schon sagt, darum die vorhandene implizite Beschreibung des Objektes so schnell wie möglich zu interpretieren und darzustellen. Der KD-Baum sorgt dafür, dass möglichst wenig nicht relevante Information verarbeitet wird.

Ein weiterer Schwerpunkt der Arbeit liegt in der Anwendung der Idee der Constructive Solid Geometry (CSG). Damit ist es möglich, beliebig komplexe Objekte durch Kombinationen aus einfacheren Objekten darzustellen. Das heisst, es ist möglich einen Körper mit einem zweiten zu schneiden und auf dem überlappenden Teil eine Mengenoperation auszuführen. So kann zum Beispiel ein Zylinder aus einer Kugel ausgeschnitten werden etc. Ein entscheidender Vorteil dieser Methode ist einerseits ihre logische und verhältnismässig einfache Darstellung von Objekten, die sich beliebig komplex schneiden, wie auch eine saubere Darstellung von scharfen Kanten trotz der verringerten Samplingdichte. Dieses Konzept wird in Kapitel 3 genauer erläutert.

Die Szene wird zum Schluss mit einer Implementation eines Raytracers gerendert, das heisst für den Betrachter in Form eines Bildes dargestellt.

1.2 Gliederung

Das Kapitel 2 enthält zunächst eine Einführung in das Konzept der impliziten Isoflächen und stellt anschliessend den KD-Baum als hier verwendete Beschleunigungsstruktur dar. Es geht dabei darum die Idee des KD-Baumes zu erläutern, wie auch eine detaillierte Betrachtung seiner wichtigsten Komponenten zu präsentieren. Dies beinhaltet Aspekte wie den Aufbau, die anschliessende Traversierung (Auswertung) und Ergänzung eines bereits bestehenden Baumes.

In Kapitel 3 wird dann auf die Visualisierung der Isofläche eingegangen. Dabei werden die verwendeten Techniken und Konzepte dargestellt. Dies sind der Software Raycaster, die Constructive Solid Geometry, die Beleuchtung mittels Phong Shading sowie eine kurze Abhandlung der verwendeten Interpolation. Auch wird eine Übersicht gegeben über das von Christian Sigg zur Verfügung gestellte Framework, in das die Implementation eingebettet ist.

Kapitel 4 ist ganz der Implementation gewidmet. Es geht darum, Einsicht in interessante Einzelheiten wie auch einen groben Überblick über das gesamte Verfahren zu geben. Dies ist insbesondere für eventuelle weitere Arbeiten an diesem Framework relevant. Es werden zudem die verwendeten Technologien erwähnt.

Zum Schluss werden die Resultate in Kapitel 5 zusammengefasst und Kapitel 6 enthält eine Liste von Referenzen und Copyright Beschränkungen.

2

Beschleunigungsstruktur

Im Folgenden wird das Konzept der Isoflächen eingeführt. Begonnen mit einer Motivation und gefolgt von einer mathematischen Beschreibung. Bei Isoflächen (implizite Flächen) haben alle Punkte auf der Oberfläche des Objektes den gleichen so genannten Isowert. Der Isowert ist der Wert der skalaren Funktion an dieser Stelle, respektive der Wert an dieser Stelle im Datengitter. Das ganze Samplingvolumen wird in einzelne Zellen aufgeteilt, wobei nur diejenigen Zellen, die zur Definition der Fläche beitragen, gespeichert werden. Um diese dünn besetzte Struktur (sparse structure) effizient abtasten (sampling) zu können verwenden wir eine Beschleunigungsstruktur: Den KD-Baum.

Im Anschluss an die Isofläche beschäftigt sich das Kapitel mit der Erläuterung dieses KD-Baumes, da er eine zentrale Rolle im Rendering der Isofläche spielt. Zuerst geht es darum, wie ein KD-Baum strukturiert ist und was seine Funktion ist. Danach wird aufgezeigt, wie so ein Baum aus einem gegebenen Datensatz konstruiert wird (generell und am Beispiel der hier verwendeten Informationen). Zum Schluss wird erklärt, wie der Baum verwendet wird. Dazu wird ein Algorithmus diskutiert, der eine korrekte Traversierung des Baumes ermöglicht und gleichzeitig sehr effizient arbeitet sowohl in Bezug auf Speicherplatz wie auf Geschwindigkeit. Auch das Problem einer Rebalancierung und Ergänzung einer bereits vorhandenen Struktur wird behandelt.

Das Ziel dieser Arbeit ist es, komplexe Objekte mit möglichst wenig Speicherbedarf und in möglichst kurzer Zeit detailliert, korrekt und sauber zu rendern wobei hier ein reiner Software Ansatz verfolgt wird.

2.1 Implizite Isoflächen

2.1.1 Einführung

Eine Isofläche ist durch eine skalare Funktion beschrieben und dadurch charakterisiert, dass der Funktionswert überall auf der Oberfläche den gleichen Wert hat - den so genannten Isowert. Isoflächen sind vergleichbar mit Höhenkurven auf einer Landkarte. Diese beschreiben die Menge aller Punkte mit dem gleichen Abstand zum Meeresniveau. In unserem Fall ist die Isofläche jedoch in drei Dimensionen definiert:

$$S = \{x \in \Re^3 | f(x) = Isovalue\}$$

2.1.2 Motivation

3D Scans können mittels Isoflächen visualisiert werden. Eine beliebte Anwendung dieser Technik findet sich zum Beispiel in der Computer-Tomographie (CT) oder im Magnetic Resonance Imaging (MRI). Die Repräsentation solcher relativ komplexer Oberflächen wird dadurch stark vereinfacht. Die alternative Darstellung durch Triangulation würde eine enorme Menge an sehr kleinen Dreiecken erfordern um die selbe Detailtreue zu erhalten. Dies bringt einen sehr grossen Speicherbedarf und eine dementsprechend lange Verarbeitungszeit mit sich. Des Weiteren ist die Triangulation nicht geeignet, wenn der Isowert interaktiv geändert werden soll, um durch das Volumen zu browsen. Dazu müssten laufend viel zu viele Dreiecke generiert werden.

2.1.3 Gitterdarstellung der Isofläche

Wir verwenden in dieser Arbeit an Stelle der skalaren Funktion ein Gitter mit gesampelten Werten. Um den Speicherverbrauch zu optimieren beschränken wir uns darauf, nur Gitterzellen zu speichern, die zur Definition der Oberfläche beitragen (rot markiert in Abbildung unten). Um am Schluss aus der gespeicherten Information ein Bild zu erzeugen wird das Gitter abgetastet (sampling). Das Konzept des KD-Baumes erlaubt uns ein sehr effizientes Auffinden der relevanten Zellen und somit ein schnelles Rendering.

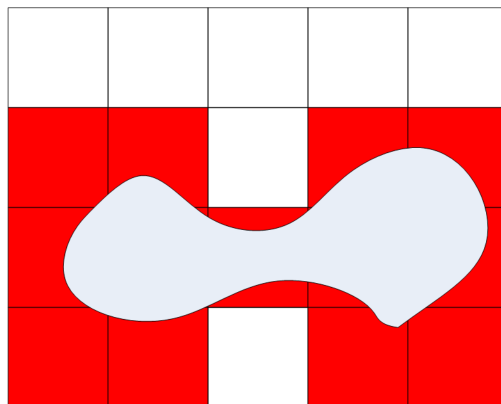


Abbildung 2.1: Gitterdarstellung der Isofläche

2.2 KD-Baum

2.2.1 Motivation

Im Folgenden soll das Konzept des KD-Baumes erläutert werden. Er spielt in dieser Arbeit eine zentrale Rolle, da wir das Rendering komplett in Software durchführen und nicht auf eine Hardwarebeschleunigung zurückgreifen. Doch auch bei der Hardware beschleunigten Implementation kommt der KD-Baum zum Einsatz [6].

Beim Raycasting muss der Schnittpunkt von jedem Sichtstrahl (Ray) mit der Isofläche gefunden werden, falls er existiert. Ohne eine geeignete Beschleunigungsstruktur müsste jedes Mal jede einzelne Zelle in unserem Gitter abgetastet werden um diesen Punkt zu finden. Da dieses naïve vorgehen einen linearen Zeitaufwand erfordert, verwenden wir den KD-Baum, der uns die Suche in O(log n) ermöglicht. Der Geschwindigkeitsvorteil wird umso markanter je mehr Zellen unser Gitter enthält.

Der Baum stellt grundsätzlich eine k-dimensionale Suchstruktur dar (daher der Name KD-Baum). In unserem Fall haben wir einen 3D-Baum implementiert, der die Suche in x-, y- und z-Richtung erlaubt.

2.2.2 Die Beschleunigungsstruktur im Detail

Der Baum wird grundsätzlich in zwei Phasen verarbeitet. Als erstes muss aus dem Gitter der Baum erstellt werden und anschliessend wird er während dem Rendering traversiert. Es ist zudem je nach Implementation auch möglich in einen bereits vorhandenen Baum weitere Zellen einzufügen oder vorhandene Zellen zu entfernen. Unter Umständen kann eine anschliessende Rebalancierung erforderlich werden.

Wie jeder Baum besteht auch der KD-Baum aus inneren Knoten und Blättern. Innere Knoten repräsentieren die aktuellen Splitplanes (Position und Richtung). Die Blätter sind Pointer auf Listen aller Zellen im entsprechenden Unterraum.

Der KD-Baum ist eine binäre Struktur, d.h. jeder innere Knoten hat zwei Kinder. Jedes Level von der Wurzel bis zu den Blättern unterteilt den Raum durch eine zweidimensionale Splitplane in der x-, y- oder z-Ebene in zwei möglichst gleich grosse Unterräume. In unserem Fall unterteilt die Wurzel des Baumes den Raum in x-Richtung. Deren Kinder unterteilen dann die jeweiligen Unterräume in y-Richtung und deren Kinder wiederum die neuen Unterräume in z-Richtung. Anschliessend wird die Unterteilung wieder in x-Richtung durchgeführt und so weiter. Dies wird so lange gemacht bis ein gewisses Abbruchkriterium erreicht ist. Dieses kann zum Beispiel definiert werden als minimale Grösse eines Unterraumes, minimale Anzahl Zellen im Unterraum etc.

Es ist auch denkbar, anstatt der fest vorgegebenen Reihenfolge der Unterteilung die jeweils optimale aktuelle Richtung zu wählen. Man stelle sich zum Beispiel vor, dass alle Zellen auf einer Geraden in x-Richtung liegen. In dem Fall macht es keinen Sinn, diesen Raum in y- oder z-Richtung zu unterteilen, da alle Zellen auf der einen oder anderen Seite landen und wir keine eigentliche Unterteilung vorgenommen haben. Beim traversieren des so erstellten Baumes müssen wir also einen Schritt machen, der uns auf der Suche nach dem aktuellen Schnittpunkt nicht weiter bringt und somit den Zeitaufwand unnötig vergrössert.

Die Wahl der Position der Splitplane ist ein sehr komplexes Problem. Im Optimalfall unterteilt die Splitplane den Raum immer in zwei gleich grosse Unterräume. Dadurch wird ein Baum erstellt, der eine Tiefe in $O(\log n)$ hat und somit in logarithmischer Zeit traversiert werden kann.

In dieser Arbeit wurde der Ansatz der fixen Reihenfolge gewählt, da dieser im Schnitt relativ gute Ergebnisse liefert. In einzelnen Spezialfällen wäre ein analytischer Ansatz jedoch besser, der die optimale Richtung für die Splitplane zuerst bestimmt.

Einen nicht unerheblichen Anteil der fürs Rendering benötigten Zeit macht das Erstellen des KD-Baumes aus. Wir haben darauf geachtet, nicht den best möglichen KD-Baum zu erstellen, sondern versucht, einen Kompromiss zwischen schnellem Aufbau und effizienter Struktur zu erzielen. Im worst case ist unser KD-Baum drei mal tiefer als der optimale Baum: Man stelle sich wieder das oben erwähnte Szenario vor, in dem alle relevanten Gitterzellen auf einer Geraden liegen. Jetzt wird nur in jedem dritten Unterteilungsschritt der Raum halbiert. In den anderen beiden Schritten, wo die Splitplane parallel zur Geraden verläuft, nützt die Aufteilung nichts. Wird zu viel Zeit in die Erzeugung der Baumstruktur investiert ist der Gesamtprozess des Rendering jedoch nicht mehr optimal.

In Kapitel 3 wird genauer auf die Visualisierung eingegangen. Dabei werden die Verwendung des KD-Baumes aus der Sicht des Raycasters nochmals aufgegriffen und weitere Probleme diskutiert.

2.2.3 Aufbau

Im folgenden Schema wird an Hand eines Beispiels die Konstruktion des KD-Baumes illustriert.

Zur einfacheren Darstellung wird das Vorgehen in 2D erläutert.

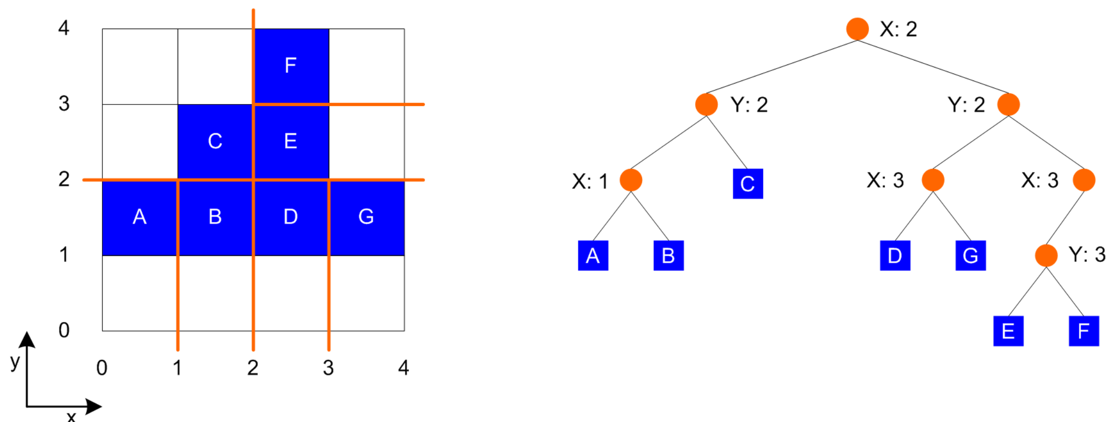


Abbildung 2.2: KD-Baum erstellen

Links ist das Gitter mit 4 auf 4 Zellen dargestellt und rechts der daraus resultierende KD-Baum. Als erstes wird in x-Richtung die optimale Position ermittelt. Optimal heisst in diesem Fall dass möglichst gleich viele Zellen auf der linken wie auf der rechten Seite der Splitplane liegen. Im Beispiel ist das an der x-Position 2. Wenn diese gefunden ist wird im Baum die Wurzel mit dieser Information erzeugt. Für jeden Knoten im Baum wird gespeichert ob es sich um einen inneren Knoten oder um ein Blatt handelt und falls es sich um einen inneren Knoten handelt wird zusätzlich noch die Position der Splitplane, die durch diesen Knoten repräsentiert wird, gespeichert sowie deren Richtung. Anschliessend wird der ganze Raum in einen linken und einen rechten Unterraum aufgeteilt. Diese Unterräume werden nun rekursiv weiter aufgeteilt. Im nächsten Schritt wird somit in y-Richtung halbiert, dann wieder in x-Richtung etc. Als Abbruchkriterium im obigen Beispiel gilt, dass jeder Unterraum höchstens eine Zelle enthalten soll. Dies führt zu einem Baum der Tiefe 3. Mit diesem binären Suchbaum kann somit eine entsprechende Zelle in maximal 3 Schritten gefunden werden. Mit linearer Suche wären dafür im worst case 7 Schritte notwendig gewesen.

Die inneren Knoten repräsentieren Splitplanes. Die eigentlichen Zellen, die wir beim Rendering suchen, werden in den Blättern des Baumes gespeichert. Als Abbruchkriterium der Implementation wird eine maximale Anzahl Zellen pro Unterraum verwendet. Ist diese Schranke grösser als 1, so werden Zellen im selben Unterraum als lineare Liste organisiert, auf dessen Head das entsprechende Blatt des Baumes zeigt.

Das Erstellen dieser Suchstruktur lohnt sich vor allem dann, wenn sie mehr als nur einmal verwendet werden kann. Mit dem in dieser Arbeit implementierten Ansatz ist es möglich, in einen bereits erstellten Suchbaum weitere Zellen einzufügen. Dafür werden sie zunächst in die Liste des entsprechenden Blattes eingefügt. Bei Bedarf kann dann der Suchbaum rebalanciert werden. Dies lohnt sich jedoch nicht in jedem Fall!

2.2.3.1 Traversierung

Das Traversieren des Baumes entspricht der Suche in einem Binärbaum. Gesucht wird in diesem Fall die Zelle, die an einer ganz bestimmten Position im Objekt Raum liegt.

Der in dieser Arbeit implementierte Ansatz zur Visualisierung basiert auf dem Raycasting Verfahren das im nächsten Kapitel diskutiert wird. Kurz gesagt wird für jeden Pixel auf dem zu erstellenden Bild ein Sichtstrahl von der Kamera in den Objekt Raum verfolgt und untersucht, was von diesem Strahl getroffen wird.

Um diese Stelle zu finden, die unser Sichtstrahl trifft, wird der KD-Baum traversiert. Vom Strahl wissen wir, welche Richtung er hat und wo er beginnt. Als erstes wird ein Intersektionstest mit der Axis Aligned Bounding Box (AABB) gemacht. Die AABB umfasst die ganze Szene. Das heisst für uns, dass wenn der Strahl an ihr vorbei schiesst, er unsere Objekte sicher auch nicht trifft. Falls er jedoch die AABB schneidet, so gibt es eine potentielle Intersektion zwischen Sichtstrahl und Objekt, in unserem Fall der Isofläche.

Als erstes wird jetzt der Eintrittspunkt in die AABB berechnet und auf Grund dessen der linke oder der rechte Teilbaum weiterverfolgt. Dies wird so lange gemacht, bis wir entweder die AABB verlassen oder in einem Blatt angelangt sind.

Wenn wir den Unterraum gefunden haben, der unseren potentiellen Schnittpunkt von Isofläche und Sichtstrahl enthält, so muss die Liste der in diesem Unterraum enthaltenen Zellen linear durchsucht werden. Die maximale Länge dieser einfach verketteten Liste ist direkt abhängig von der Wahl des Abbruchkriteriums beim Erstellen des KD-Baumes.

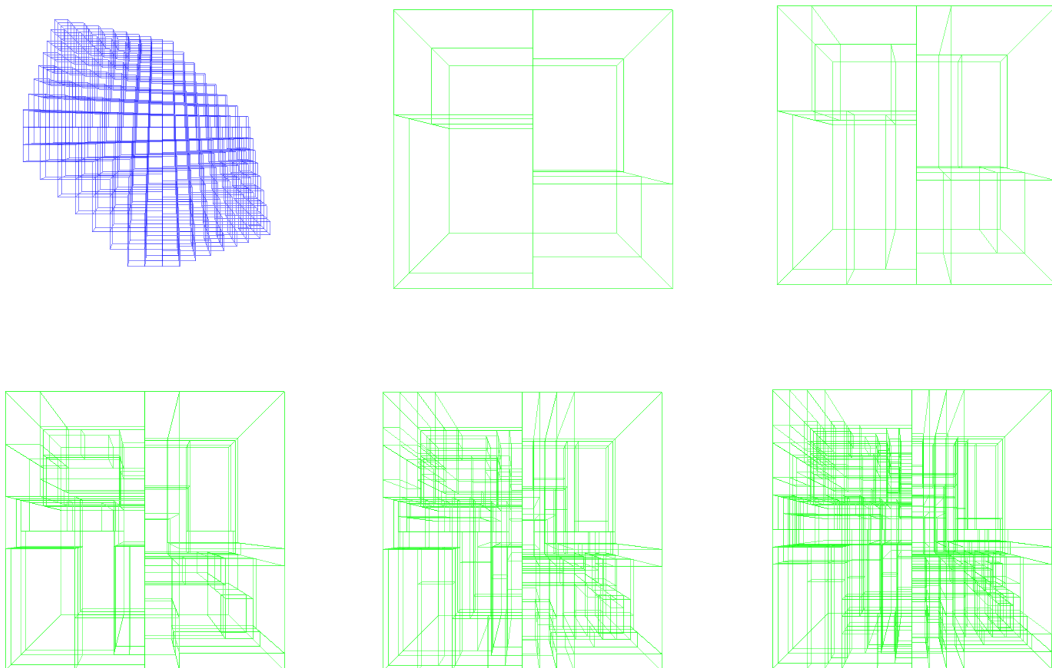


Abbildung 2.3: Verschieden Abbruchkriterien

Von oben links nach unten rechts: Visualisierung des Datengitters; Abbruchkriterium: höchstens 100 Zellen pro Unterraum, höchstens 50, höchstens 10, höchstens 5 und höchstens 2.

Wenn wir für die maximale Anzahl Zellen pro Unterraum einen grossen Wert wählen, so werden die Listen in den Blättern im Durchschnitt länger, der Baum jedoch weniger hoch. Auch der Aufwand um den Baum zu erstellen wird kleiner, da nicht so viele Rekursionsschritte gemacht werden müssen. Im Kapitel 5 werden verschiedene Werte für das Abbruchkriterium untersucht und verglichen. Abbildung 2.3 gibt einen visuellen Eindruck der unterschiedlichen Baumgranularitäten.

2.2.3.2 Alternative Beschleunigungsstruktur: Der Octree

Eine etwas weniger effiziente, dafür einfachere Beschleunigungsstruktur ist der Octree. Beide Strukturen gehören in die Kategorie der so genannten BSP Bäume. BSP steht für *binary space partitioning*.

Beim Octree wird der Raum jeweils in acht gleich grosse Unterräume aufgeteilt (daher der Name). Da die Unterräume gleich gross sind (im räumlichen Sinn und nicht in der Anzahl enthaltenen Zellen), kommt es häufig vor, dass gewisse Unterräume sehr viele Zellen enthalten und andere fast gar keine.

Auch diese Struktur ist rekursiv definiert und erlaubt eine beschleunigte Suche im Objekt Raum. Grundsätzlich gibt es zwei Varianten vom Octree: den adaptiven und den regulären. Der adaptive Octree unterteilt nur diejenigen Unterräum weiter, die viele Zellen beinhalten. Der reguläre unterteilt alle Unterräume, bis in jedem die Anzahl Zellen unter der vorgegebenen Schranke liegt. Das zweite Verfahren ist sehr einfach zu implementieren, bringt jedoch unter Umständen einen recht grossen Overhead mit sich.

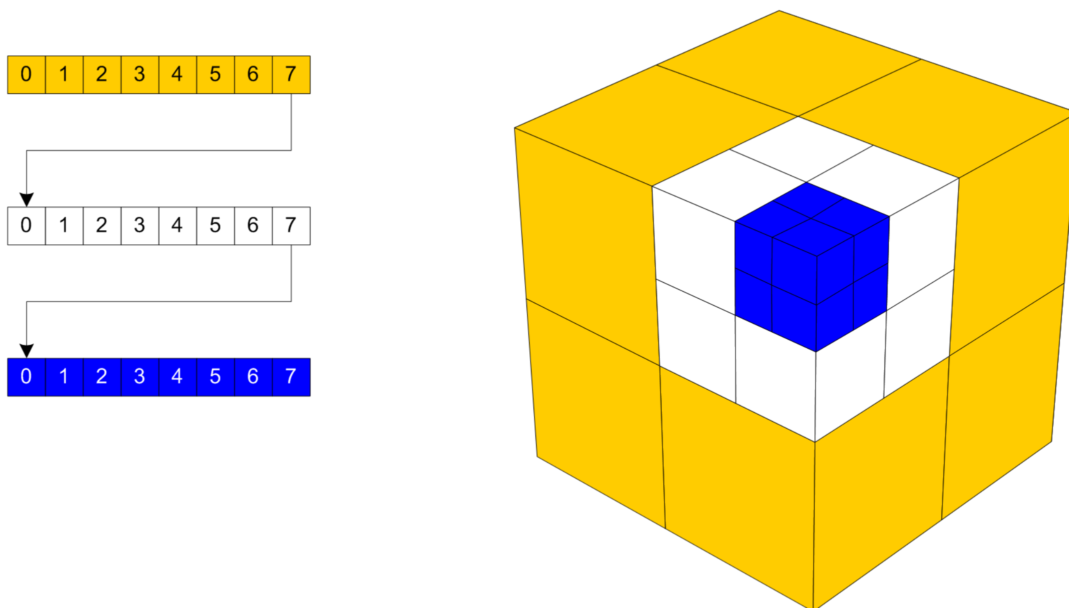


Abbildung 2.4: Beispiel eines adaptiven Octrees

Obige Abbildung zeigt ein Beispiel eines adaptiven Octree. Es werden nur diejenigen Unterräume unterteilt, die viele Zellen enthalten (adaptive space partitioning). Es ist also anzunehmen, dass in der oberen vorderen Ecke viele Details enthalten sind und der Rest verhältnismässig wenig Information enthält.

3

Visualisierung

Dieses Kapitel befasst sich mit dem gewählten Ansatz vom Software Raycaster zum rendern der Isoflächen. Eine Hardware gestützte Implementation ist in der Semesterarbeit von Olivier Chassot zu finden, die auch von Christian Sigg betreut wurde [6].

Zunächst wird die Idee des Rendering mittels Raycaster vorgestellt. Im Anschluss daran sollen die Constructive Solid Geometry (CSG) erklärt und deren entscheidende Vorteile aufgezeigt werden. Um das Objekt zu beleuchten wird Phong Shading verwendet, das auch kurz erläutert wird. Zum Schluss sollen noch die Problematik der Interpolation im dreidimensionalen Raum dargestellt, sowie das verwendete Framework kurz erwähnt werden.

3.1 Software Raycaster

3.1.1 Einführung: Das Prinzip des Raycasting

Das Konzept des Raycasting, das Verfolgen von Lichtstrahlen, ist schon relativ alt. Bereits Albrecht Dürer (1471 - 1528) hat zum Studium der "projektivischen Darstellung" ein Verfahren ähnlich dem heutigen Raycasting angewendet. Er hatte damals ein Gitterraster verwendet und mit einem Faden und einem daran befestigten Lot die direkte Verbindung (Gerade) von Punkten auf den Objekten zum Blickpunkt (Wand) geschaffen. Der entsprechende Rasterpunkt wurde dann auf dem Papier eingezeichnet (siehe Abbildung).

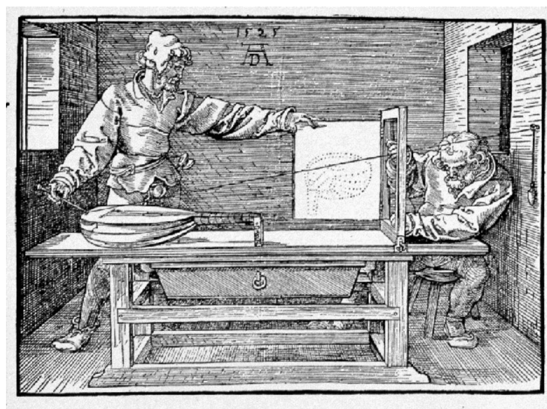


Abbildung 3.1: Albrecht Dürer (1525)

Die Idee des Raycasting besteht, wie der Name schon sagt, darin Lichtstrahlen zu verfolgen. Ausgangspunkt ist eine Lichtquelle, deren Strahlen an Objekten der Szene (eventuell mehrfach) reflektiert werden und auf der Kamera des Beobachters eintreffen. Diese Sichtweise ist jedoch nicht direkt praktisch implementierbar, da sehr viele Strahlen, die von der Lichtquelle ausgesandt werden, die Kamera (den Beobachter) nicht treffen und somit nichts zum Bild beitragen.

Als Lösung für dieses Problem verfolgt man die Lichtstrahlen beim Raycasting von der Kamera aus rückwärts zur Lichtquelle. So werden nur Strahlen berücksichtigt, die auch etwas zum Bild beitragen. Dieses Vorgehen ist in nachfolgender Darstellung veranschaulicht.

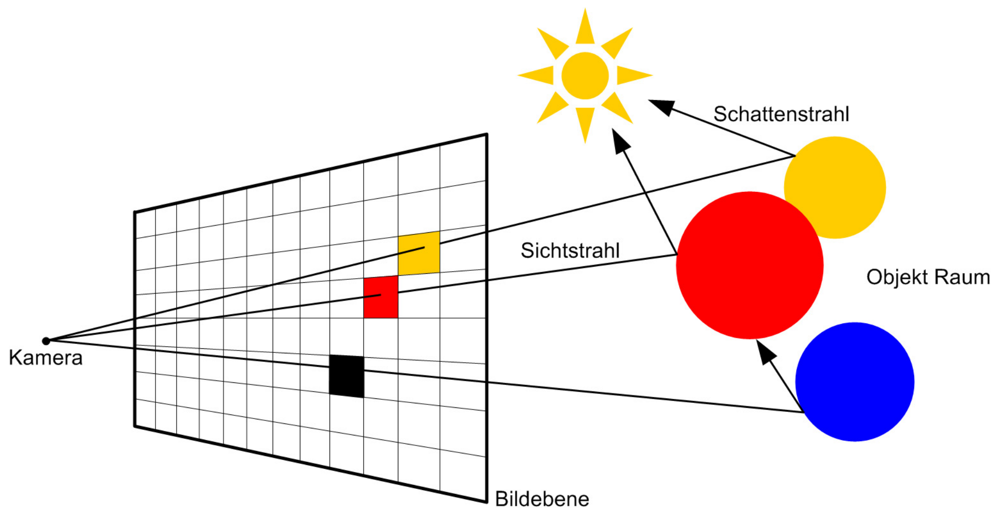


Abbildung 3.2: Prinzip des Raycasting

Raycasting ist ein so genanntes Bildraum Verfahren, da die Hauptschleife des Programms über die Pixel der Bildebene läuft und pro Pixel die Isofläche im Objekt Raum abgetastet wird.

Der Vorteil von Raycasting gegenüber der Polygon Pipeline besteht darin, dass die Rechenzeit weitgehend unabhängig von der Szenenkomplexität ist und die Behandlung von algebraisch definierten Flächen, wie der Isofläche, verhältnismässig einfach zu implementieren ist. Des weiteren ist es möglich, Mehrfachreflexionen, semitransparente Objekte und Schlagschatten realistisch darzustellen.

Der Unterschied zwischen Raycasting und Raytracing besteht darin, dass bei ersterem weder Reflexion noch Brechung berücksichtigt werden. Beim Raytracing hingegen werden Strahlen verzweigt, abgelenkt und anschliessend weiterverfolgt bis ein gewisses Abbruchkriterium erreicht ist.

Als Beleuchtungsmodell wird meistens Phong Shading verwendet, auf das in Kapitel 3.3 genauer eingegangen wird.

3.1.2 Sichtstrahlen

Die Sichtstrahlen werden durch Vektoren repräsentiert, die ihren Ursprung bei der Kamera haben. Die Richtung wird durch den Pixel der Bildebene, der gerendert werden soll, und dem Ursprung des Strahles bestimmt. Dieser wird dann in den Objekt Raum hinein verfolgt. Der erste Schnittpunkt zwischen Strahl und einem Objekt - in unserem Fall der Isofläche - bestimmt die Farbe des Pixels auf der Bildebene. Vom Schnittpunkt aus werden so genannte Schattenstrahlen in Richtung Lichtquelle verfolgt. Wenn ein anderes Objekt die Sicht zur Lichtquelle verdeckt, so liegt der Intersektionspunkt im Schatten. Wie dieser Fall behandelt

wird hängt von der Implementation der Beleuchtung ab. Falls mehr als eine Lichtquelle vorhanden sind, so muss ein Schattenstrahl zu jeder Lichtquelle verfolgt und die einzelnen Beiträge entsprechend aufsummiert werden.

3.1.3 Intersektion

Für jeden generierten Sichtstrahl muss untersucht werden, ob und wo er die Isofläche schneidet.

Wenn er an der Isofläche vorbei schiesst, wird die Farbe des Pixels im Bildraum auf die Hintergrundfarbe der Szene gesetzt. Im Falle eines Treffers wird das Phong Modell lokal ausgewertet und die Pixelfarbe auf den entsprechenden Wert gesetzt. Dieser hängt ab vom Einfallswinkel des Sichtstrahles, von der Wahl des Umgebungslichtes, von der Position der Lichtquelle(n) etc. In Kapitel 3.3 wird die Beleuchtung mittels Phong Shading genauer betrachtet.

Beim Traversieren des KD-Baumes wird der Unterraum des Objekt-Raumes identifiziert, der die Kandidaten einer möglichen Intersektion enthält. Anschliessend werden diese Zellen (geordnet nach der Distanz zur Kamera auf dem Sichtstrahl) auf eine mögliche Intersektion mit der Isofläche untersucht. Dies geschieht mit linearem Zeitaufwand, da die einzelnen Zellen in linearen Listen organisiert sind. Folgendes Beispiel veranschaulicht diesen Sachverhalt.

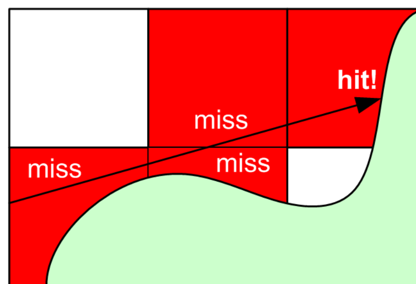


Abbildung 3.3: Isoflächen-Intersektions Test

Jede Zelle wird auf eine Intersektion mit dem Sichtstrahl hin getestet. Wenn dieser Test positiv ausfällt, so wird untersucht ob der Sichtstrahl die Isofläche, die in der Zelle definiert ist, schneidet. Trifft auch dies zu, wird lokal ein Phong Shading ausgewertet und die Farbe des Pixels im Bildraum entsprechend gesetzt. Ist dies nicht der Fall, so wird die nächste Zelle, die auf dem Sichtstrahl liegt, untersucht. Das wird so lange gemacht bis entweder die Isofläche getroffen wird oder aber der Strahl hinten aus der Axis Aligned Bounding Box (AABB) austritt. In dem Fall wird die Pixelfarbe wieder auf die Hintergrundfarbe gesetzt.

3.1.4 Traversieren des KD-Baumes

Ein effizientes Traversieren des Baumes ist entscheidend für ein schnelles Rendering. Ein generelles Problem von Suchbäumen besteht darin, dass sie in der Regel keine Möglichkeit bieten, direkte Nachbarn zu finden. Wenn wir also in einem Unterraum angelangt sind und feststellen, dass unser Strahl hier zwar CSG Zellen trifft, doch die Isofläche nicht durchstösst, so müssen wir im nächsten Unterraum weitersuchen. Eine naïve Implementation würde in diesem Fall die Suche nach dem benachbarten Unterraum wieder bei der Wurzel beginnen.

Um dies zu verhindern verwenden wir bei unserer Implementation einen Stack zur Effizienzsteigerung. Als erstes werden der Eintritts- und Austrittspunkt des Strahles in der AABB bestimmt. Anschliessend beginnt die eigentliche Traversierung. Bei jedem Knoten im Baum wird der Raum halbiert (definiert durch die Splitplane). Wir sind an der vordersten (am

nächsten bei der Kamera liegenden) Intersektion des Strahles mit der Isofläche interessiert, da diese alle dahinter liegenden verdeckt. Transparenz wurde in dieser Arbeit nicht implementiert und könnte Teil eines weiterführenden Projektes sein.

Wir untersuchen beim Traversieren also zuerst den vorderen der beiden Unterräume und legen den hinteren falls nötig auf den Stack. Wenn Eintritts- und Austrittspunkt auf der selben Seite der Splitplane liegen, so wird der Raum hinter der Splitplane nie von unserem Sichtstrahl getroffen und muss daher nicht auf dem Stack gespeichert werden.

Es werden grundsätzlich 9 verschiedene Fälle unterschieden. Die Unterscheidungskriterien sind die Eintritts- und Austrittsposition des Sichtstrahles im Unterraum in Bezug auf die Splitplane. Wird die Splitplane vom Sichtstrahl nicht durchstossen, so muss der Raum hinter der Splitplane nicht abgesucht werden.

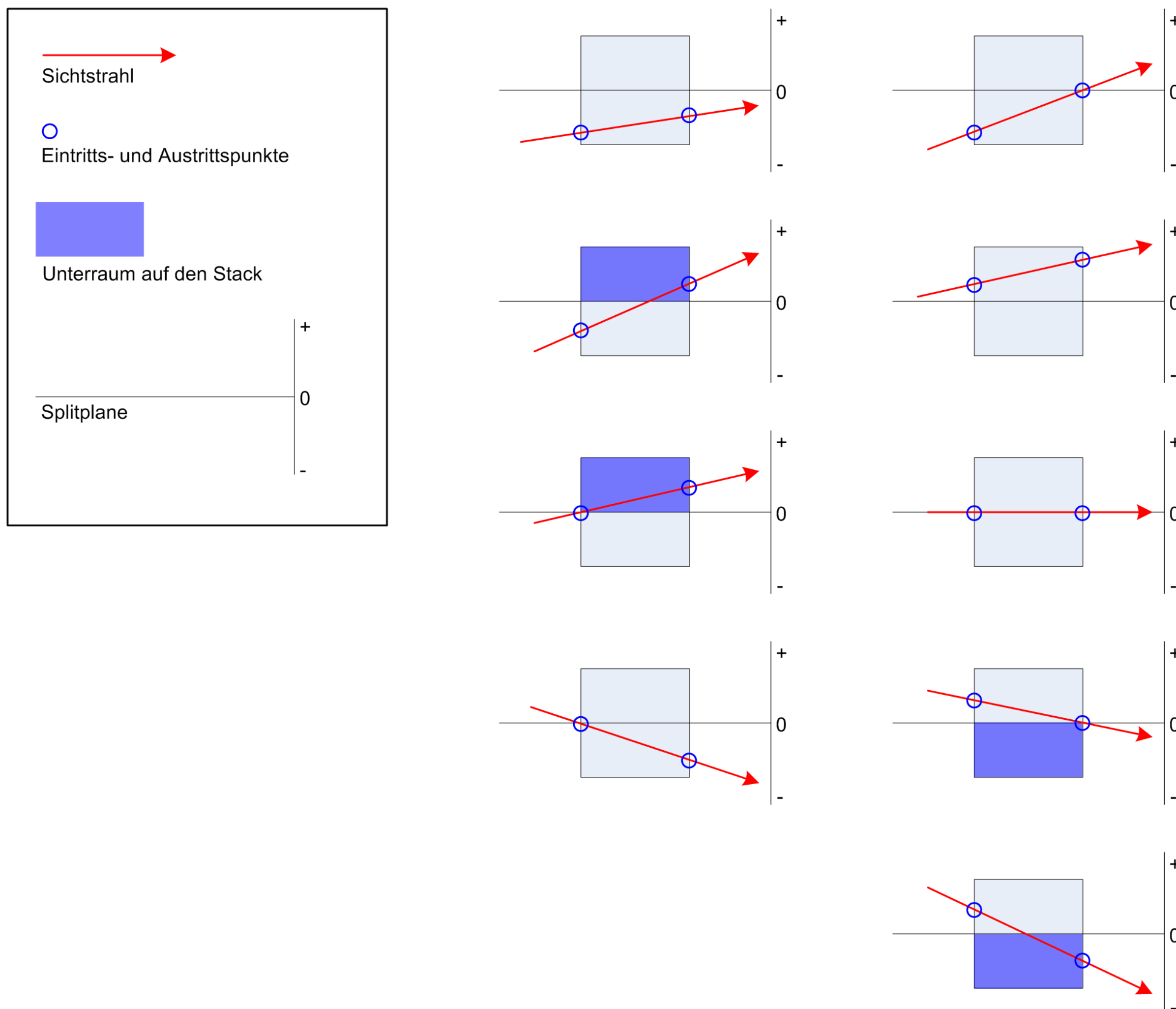


Abbildung 3.4: 9 Fälle, die beim Traversieren des KD-Baumes auftreten können

Unser Verfahren läuft wie folgt: Sobald wir bei einem Blatt des Baumes angelangt sind, untersuchen wir die CSG Zellen in der linearen Liste wie oben beschrieben auf eine Intersektion hin. Falls wir nun die Isofläche in diesem Unterraum verfehlen, so holen wir den nächsten (hinteren; far subspace) Unterraum, der zuoberst auf dem Stack liegt und fahren dort mit der Suche weiter. Dies erspart uns, eine neue Suche bei der Wurzel zu starten und macht den ganzen Prozess dadurch wesentlich effizienter. Abbildung 3.5 veranschaulicht das Vorgehen.

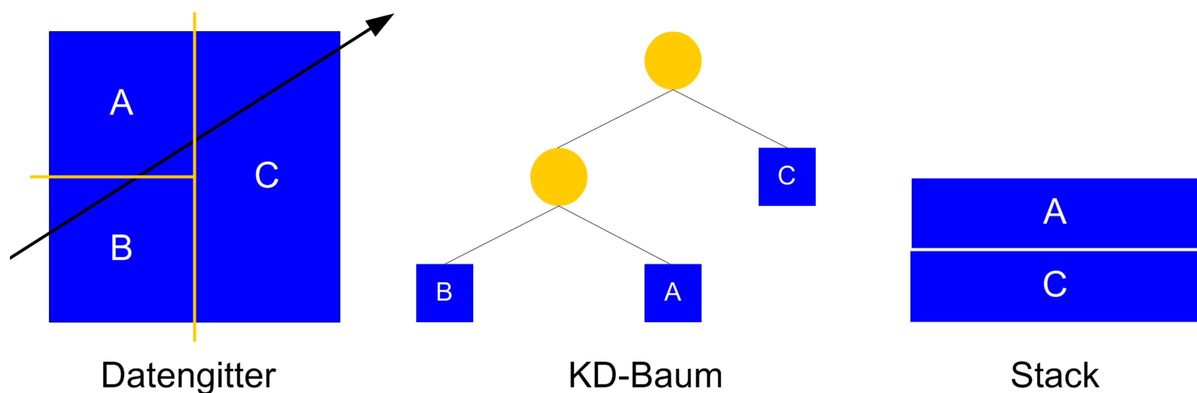


Abbildung 3.5: Effiziente Suche mit Hilfe eines Stacks

In diesem Beispiel wird zuerst der Unterraum B abgesucht. Wird keine Intersektion gefunden, so wird in Unterraum A weitergesucht. Um diesen zu erreichen wird das oberste Element vom Stack genommen. Finden wir auch in diesem Unterraum keine Intersektion, so bleibt noch Unterraum C. Ist der Stack leer und wir haben noch immer keine Intersektion, dann trifft unser Sichtstrahl die Isofläche nicht und dem Pixel im Bildraum wird die Hintergrundfarbe zugewiesen.

3.1.5 Sampling innerhalb einer CSG Zelle

Wenn wir eine Zelle gefunden haben, die vom Sichtstrahl durchstossen wird, so müssen wir untersuchen ob die Isofläche darin getroffen wird. Unter einer Zelle müssen wir uns hier ein dreidimensionales Gitter aus numerischen Werten vorstellen. Wir sind nun an allen Werten interessiert, die 0 sind. Diese beschreiben die Isofläche. Alles was grösser als 0 ist liegt ausserhalb, alles was kleiner als 0 ist liegt innerhalb unseres Objektes.

Wir müssen also den Strahl so lange verfolgen, bis wir auf einen Wert stossen, der kleiner oder gleich 0 ist. An diesem Punkt treffen wir die Isofläche und wir können das Phong Modell dort auswerten. Wir sind interessiert an der genauen Position innerhalb der Zelle wo der Strahl unsere Isofläche durchstösst.

Ein wichtiger Parameter unseres Verfahrens ist die Anzahl Schritte, die wir beim Sampling innerhalb der Zelle machen. Viele Schritte ergeben eine genauere Position der Isoflächenintersektion, sind dafür recheninternsiver. Unser Framework erlaubt uns, diese Werte individuell zu definieren. Sie sind jedoch anschliessend für alle Zellen gleich, unabhängig von der Krümmung oder Richtung der Isofläche.

Als erstes wird aus der definierten Anzahl Schritte die entsprechende Schrittweite bestimmt und anschliessend an jeder Samplingposition (falls nötig mit Interpolation) der Wert innerhalb des dreidimensionalen Datengitters evaluiert.

Die Samplingposition innerhalb der Zelle berechnet sich wie folgt:

*Eintrittspkt in CSG Zelle + (Richtung vom Sichtstrahl * Sampling Schritt * Sampling Distanz)*

In Abbildung 3.6 trifft keiner der Samplingpunkte die Isofläche exakt. Dazu wäre ein feineres Sampling mit mehr Schritten notwendig. Eine oft benutzte Technik verwendet eine grobe Abtastung für die ganze Zelle und eine feinere zwischen dem letzten Samplingpunkt ausserhalb und dem ersten innerhalb des Objektes um die Isofläche schnell und exakt zu bestimmen. Die simplere Abtastung kann zu Unebenheiten der Oberfläche, Löchern und

unscharfen Kanten führen. Daher ist im Allgemeinen eine komplexere Samplingmethode dieser vor zu ziehen.

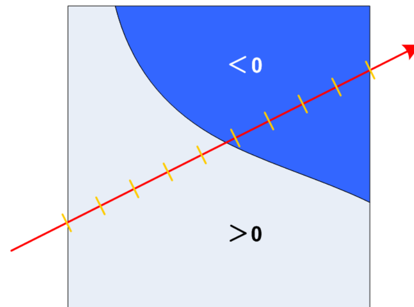


Abbildung 3.6: Sampling innerhalb einer CSG Zelle

3.2 Constructive Solid Geometry

3.2.1 Einführung: Kombination mehrerer Objekte

Die Idee des Constructive Solid Geometry Ansatzes besteht darin, ein komplexes Objekt zu modellieren indem verschiedene einfachere Objekte entsprechend kombiniert werden.

Dies ist besonders dann hilfreich, wenn man Objekte mit Löchern hat. Zum Beispiel Objekte bei denen etwas herausgeschnitten wurde. Anstatt die komplexe Oberfläche des resultierenden Objektes darzustellen wird eine Subtraktion zweier simplerer Körper gemacht. Die Komplementäroperation dazu ist die Addition. Man stelle sich beispielsweise zwei Kugeln vor, die sich zur Hälfte überlappen. Dies als einen Körper zu modellieren wäre relativ schwierig. Ein sehr gutes Resultat erzielt man hingegen, wenn man die zwei Kugeln einzeln darstellt und für das Rendering die CSG Technik zu Hilfe nimmt. Man hat damit auch eine gewisse Garantie, dass das Resultat realistisch aussieht.

Ein weiterer Vorteil dieses Verfahrens liegt darin, dass die Schnittkanten sauber dargestellt werden können.

3.2.2 De Morgansche Gesetze

Dank den Gesetzen von De Morgan lassen sich alle benötigten Operationen aus einer Kombination aus *Union*, *Intersection* und *Minus* darstellen. Im Zusammenhang mit Isoflächen lassen sich diese drei Grundoperationen sehr einfach wie folgt definieren:

Wir definieren zunächst zwei Isoflächen V und W so, wie wir sie im Projekt verwendet haben und darauf die drei Grundoperationen:

$$V = \{x | f(x) \leq 0\} \qquad W = \{x | g(x) \leq 0\}$$

Union $$V \cup W = \{x | min(f(x), g(x)) \leq 0\}$$

Intersection $$V \cap W = \{x | max(f(x), g(x)) \leq 0\}$$

Minus $$V - W = \{x | max(f(x), -g(x)) \leq 0\}$$

Die zwei Gesetze von De Morgan, die es uns erlauben, beliebige Terme durch die Grundoperationen auszudrücken, lauten wie folgt:

$$V \cap W = \overline{\overline{V} \cup \overline{W}}$$

$$V \cup W = \overline{\overline{V} \cap \overline{W}}$$

3.2.3 Beispiele

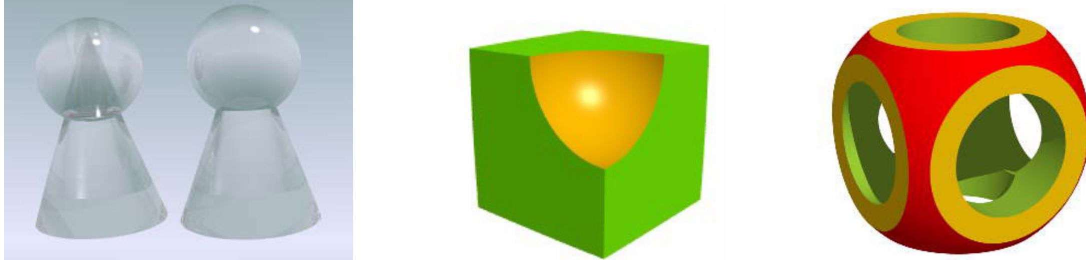


Abbildung 3.7: CSG Beispiele

Links: Kegel + Kugel (*links*: Addition, *rechts*: Union)

Mitte: Quader - Kugel

Rechts: Kugel - 3 Zylinder - 6 Quader

3.2.4 Repräsentation und Auswertung in unserem Framework

Die kombinierten CSG Operationen lassen sich am einfachsten und effizientesten als binärer Baum darstellen. Dieser wird bei der Wurzel beginnend rekursiv ausgewertet. Die inneren Knoten stellen die einzelnen CSG Operationen dar und in den Blättern sind die Primitivobjekte gespeichert. Folgendes einfaches Beispiel veranschaulicht dieses Vorgehen:

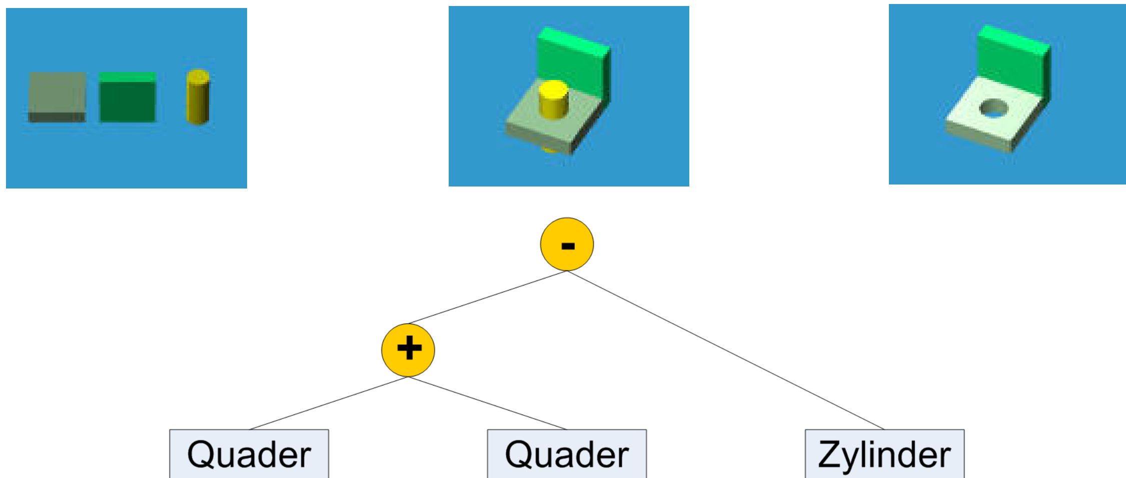


Abbildung 3.8: CSG Operationen als Baum

Ausgehend von zwei Quadern und einem Zylinder wird die Figur ganz rechts modelliert. Dazu werden zuerst die Quader miteinander verknüpft und von der daraus resultierenden Figur wird der Zylinder subtrahiert. Dies erweckt den Eindruck, als wäre ein Loch in das Objekt gebohrt worden.

3.3 Phong Shading

3.3.1 Die Phong Gleichung

Als Beleuchtungsmodell wird beim Raycasting meistens Phong Shading verwendet. Dieses Modell ergibt ein sehr realistisches Bild, da es zahlreiche physikalische Faktoren mitberücksichtigt. So spielen zum Beispiel der Winkel zwischen Oberfläche und Sichtstrahl, wie auch der Winkel zwischen Oberfläche und Lichtquelle eine entscheidende Rolle.

Mathematisch ausgedrückt lässt sich die Gleichung für die Beleuchtung wie folgt beschreiben:

$$I = I_a k_a O_d + f_{att} I_p [k_d O_d (\vec{N} \bullet \vec{L}) + k_s O_s (\vec{R} \bullet \vec{V})^n]$$

I = Lichtintensität

f_{att} = Abschwächungsfaktor (abhängig von der Distanz vom Objekt zur Lichtquelle)

$\vec{N}$ = Oberflächennormale

$\vec{L}$ = Vektor in Richtung Lichtquelle

$\vec{V}$ = Vektor in Richtung Beobachter (Kamera)

$\vec{R}$ = Reflektierter Lichtvektor

Es ist jedoch zu beachten, dass die hier präsentierte Version der Phong Gleichung bereits einige Umformungen beinhaltet, die das Berechnen der resultierenden Intensität erleichtern. Die Herleitung sowie die ursprüngliche Phong Gleichung findet sich in [4]. Das resultierende Bild ist abhängig von einer Vielzahl von Parametern, die es uns erlauben verschiedenste Oberflächeneffekte zu erzeugen.

Anschaulich lässt sich die Idee am besten durch folgende Abbildung darstellen:

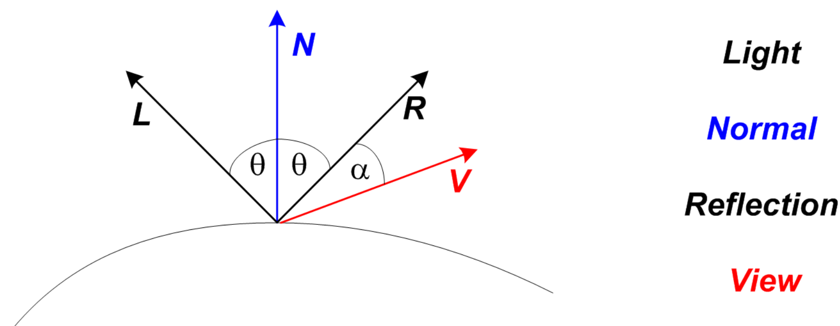


Abbildung 3.9: Phong Gleichungsparameter

Das Problem bei implizit definierten Objekten ist, dass wir keine eigentliche Oberfläche haben, auf der wir eine Normale direkt berechnen können. Es bleibt uns also nur eine Approximation dieses Vektors mit Hilfe einer geeigneten Methode.

Normalerweise berechnet man die Normale einer Isofläche aus dem normierten Gradienten der die Isofläche beschreibenden Funktion w im Punkt p:

$$\nabla w(p) = \left[\frac{\partial}{\partial x} w(\vec{p}), \frac{\partial}{\partial y} w(\vec{p}), \frac{\partial}{\partial z} w(\vec{p})\right]$$

$$\vec{n}_p = \frac{\nabla w(\vec{p})}{|\nabla w(\vec{p})|}$$

Wir haben folgende zwei Ansätze in dieser Arbeit untersucht:

Ansatz 1: Gradient der Funktion für die trilineare Interpolation.
Als Funktion w wählen wir die trilineare Interpolation (siehe Kapitel 3.4.2) und berechnen davon die Ableitung in alle Richtungen. Zum Schluss muss dieser Vektor noch normiert werden.

Ansatz 2: Unmittelbare Umgebung untersuchen
Diese Methode verwendet finite Differenzen um eine Approximation der Richtungsableitungen zu erhalten. In diesem Fall entspricht w(x) dem Funktionswert (=Gitterwert) an Position x.

$$\frac{\partial w}{\partial x} = \frac{w(x-h)-w(x+h)}{2h}, \quad \frac{\partial w}{\partial y} = \frac{w(y-h)-w(y+h)}{2h}, \quad \frac{\partial w}{\partial z} = \frac{w(z-h)-w(z+h)}{2h}$$

Anschliessend wird genau gleich wie im ersten Ansatz noch normiert. Auch dieses Resultat entspricht einer Annäherung der Normalen im Punkt p.

3.3.2 Veranschaulichung durch Beispiele

Folgendes Schema demonstriert den Einfluss dreier Parameter: n (der Exponent in obiger Gleichung) beeinflusst den Radius des Glanzpunktes. Anschaulich heisst das, dass er bestimmt wie glänzig oder matt die Oberfläche ist. Die Faktoren k_d und k_s legen die spekularen respektive diffusen Anteile der Lichtquelle fest.

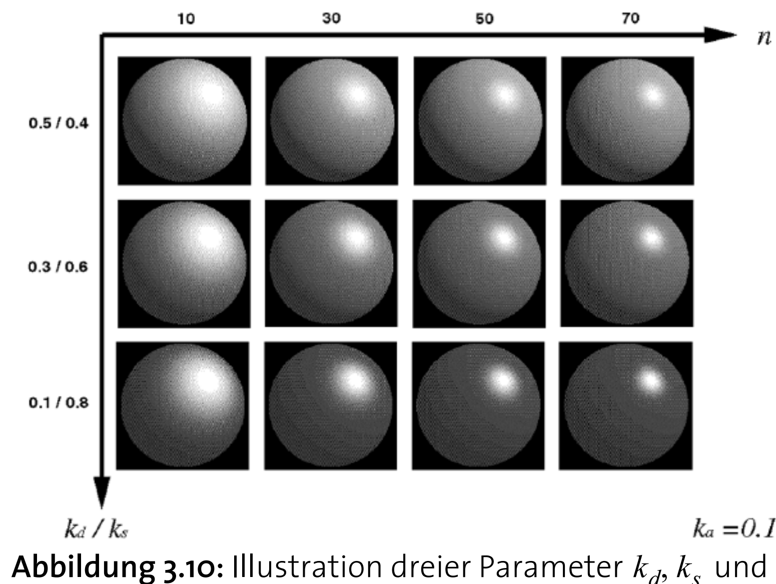


Abbildung 3.10: Illustration dreier Parameter k_d, k_s und n

3.4 Interpolation

3.4.1 Interpolation fehlender Gitterdaten

Das Datengitter, welches die Isofläche beschreibt, ist seiner Natur nach diskret. Es ist somit sehr wahrscheinlich, dass unsere Samplingpunkte nicht genau auf vorhandene Datenpunkte im Gitter fallen sondern irgendwo dazwischen liegen. In diesen Fällen müssen wir also die fehlende Information aus der vorhandenen interpolieren.

Es wird grundsätzlich unterschieden, wie viele Dimensionen interpoliert werden müssen. Falls keine der drei Dimensionen mit einer Fläche oder Kante übereinstimmt, so muss *trilinear* interpoliert werden. Dies ist der häufigste Fall. Ansonsten genügt eine *bilineare* oder sogar

lineare Interpolation. Es könnten auch Punkte, die auf einer Fläche oder Kante liegen trilinear interpoliert werden. Dies macht jedoch wenig Sinn und bringt Komplikationen mit sich. So sind zum Beispiel die acht Nachbarn nicht eindeutig definiert. Zudem benötigen die lineare oder bilineare Interpolation weniger Rechenoperationen als die trilineare Interpolation was einen Geschwindigkeitsvorteil mit sich bringt. Der Nachteil ist jedoch, dass zuerst untersucht werden muss, welche Interpolationsmethode nötig ist.

Folgende Darstellung gibt eine Übersicht über die verschiedenen Methoden:

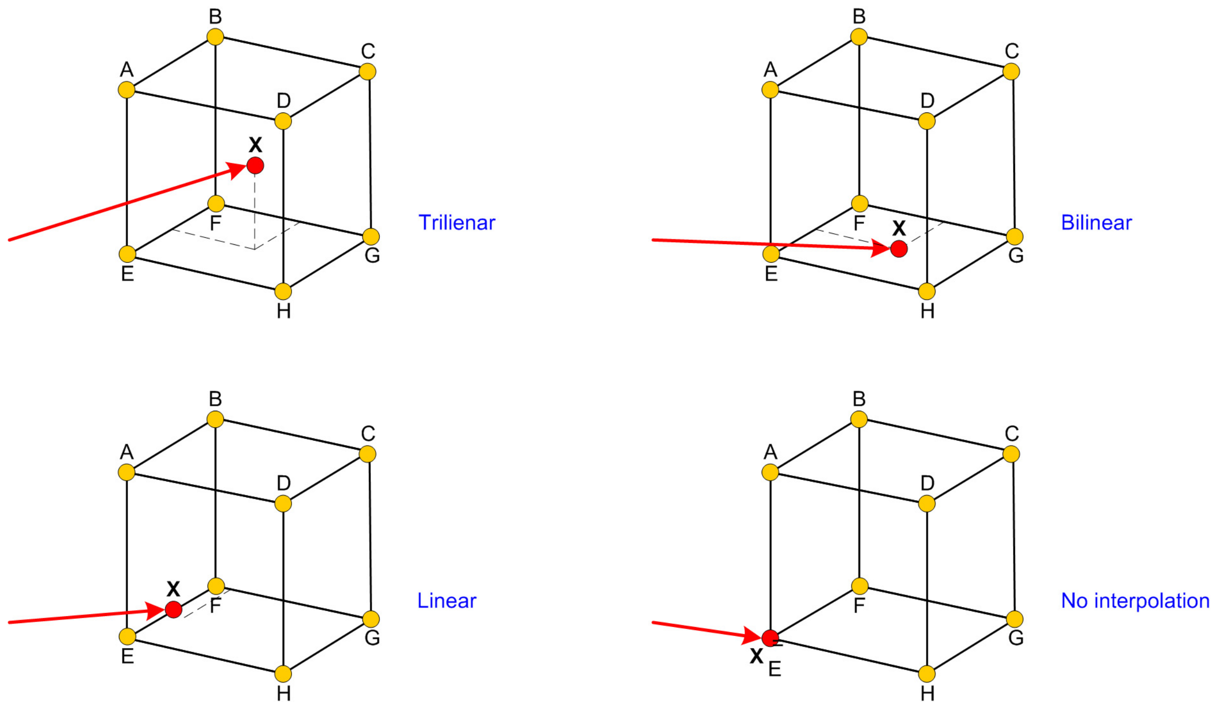
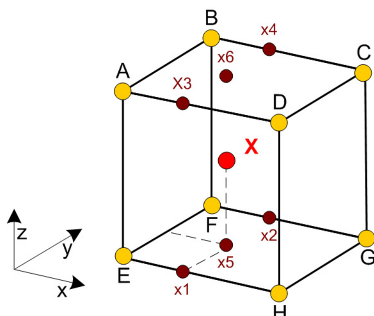


Abbildung 3.11: Verschiedene Interpolationen

3.4.2 Trilineare Interpolation

Die trilineare Interpolation ist im Prinzip nichts anderes als die Mittelwertbildung der acht Nachbarwerte in Abhängigkeit von deren Distanz zum gesuchten Punkt.

In unserem Projekt liegen die Daten (Gitterknoten) immer an ganzzahligen Positionen. Wenn wir also zum Beispiel den Wert X an Position [4.5, 4.8, 9.3] brauchen, so müssen wir gemäss nachfolgender Illustration trilinear interpolieren. Punkt E entspricht [4,4,9] und C ist an Position [5,5,10].



w1 := Distanz zwischen X und Gerade EF in x-Richtung
w2 und w3 analog in y und z-Richtung

x1 = (1-w1) * val(E) + w1 * val(H)
x2 = (1-w1) * val(F) + w1 * val(G)
x3 = (1-w1) * val(A) + w1 * val(D)
x4 = (1-w1) * val(B) + w1 * val(C)

x5 = (1-w2) * x1 + w2 * x2
x6 = (1-w2) * x3 + w2 * x4

X = (1-w3) * x5 + w3 * x6

Abbildung 3.12: Trilineare Interpolation im Detail

3.4.3 Probleme in Randbereichen

Es muss grundsätzlich für jedes Auslesen eines Datenwertes ermittelt werden, was für eine Interpolation nötig ist. Falls unser X jedoch auf einer der Seitenflächen, auf einer Kante oder sogar auf einem Knoten liegt, so müssen wir nicht trilinear interpolieren.

Als Referenzpunkt dient immer die Ecke mit dem kleinsten x-, y- und z-Index. In Abbildung 3.12 entspricht dies dem Knoten E. Alle Knoten haben eine ganzzahlige Position. Somit ist es ein leichtes, den entsprechenden Referenzpunkt zu finden. Es muss lediglich die Position von X auf die nächste Ganzzahl abgerundet werden.

Kritisch ist der Fall, wo unser X am rechten Rand des Datengitters liegt, da wir hier keine Nachbarn auf der rechten Seite mehr haben. Um nicht jedes Auslesen eines Datenwertes auf so einen Spezialfall hin überprüfen zu müssen, verschieben wir die Position des zu sampelnden Punktes um den kleinst möglichen Schritt nach links. So haben wir sichergestellt, dass wir immer Nachbarn auf der rechten Seite haben. Dies bringt natürlich auch eine gewisse Ungenauigkeit mit sich, die jedoch im gerenderten Bild nicht erkennbar und somit vernachlässigbar ist.

3.5 Das Framework

Als Ausgangspunkt für diese Arbeit wurde ein Framework für implizite Isoflächen verwendet, das von Christian Sigg an der ETH Zürich entwickelt wurde.

Dieses Framework ist unter anderem zuständig für das Einlesen von Datenwerten aus Volume Files. Es stellt diese dann als Datengitter für den Raycaster zur Verfügung. Um mit Pointern innerhalb des Speichers des Gitters zu navigieren muss folgendes beachtet werden: In x-Richtung erreicht man die nächste Zelle mit ($ptr + 1$), in y-Richtung (nach hinten) mit ($ptr + 512$) und in z-Richtung (nach oben) mit $(ptr + 512^2)$. Dieser Sachverhalt ist aus Abbildung 3.13 ersichtlich. Alle Daten werden eingelesen und in solchen Slices im Memory gespeichert.

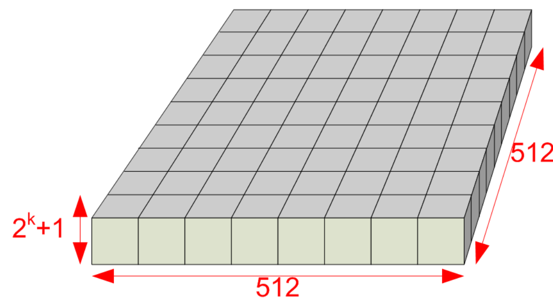


Abbildung 3.13: Slice des Datengitters im Memory

Des Weiteren stellt es ein GUI bereit, das das gerenderte Bild darstellt und Möglichkeiten zur einfachen Konfiguration des Raycasters ermöglicht.

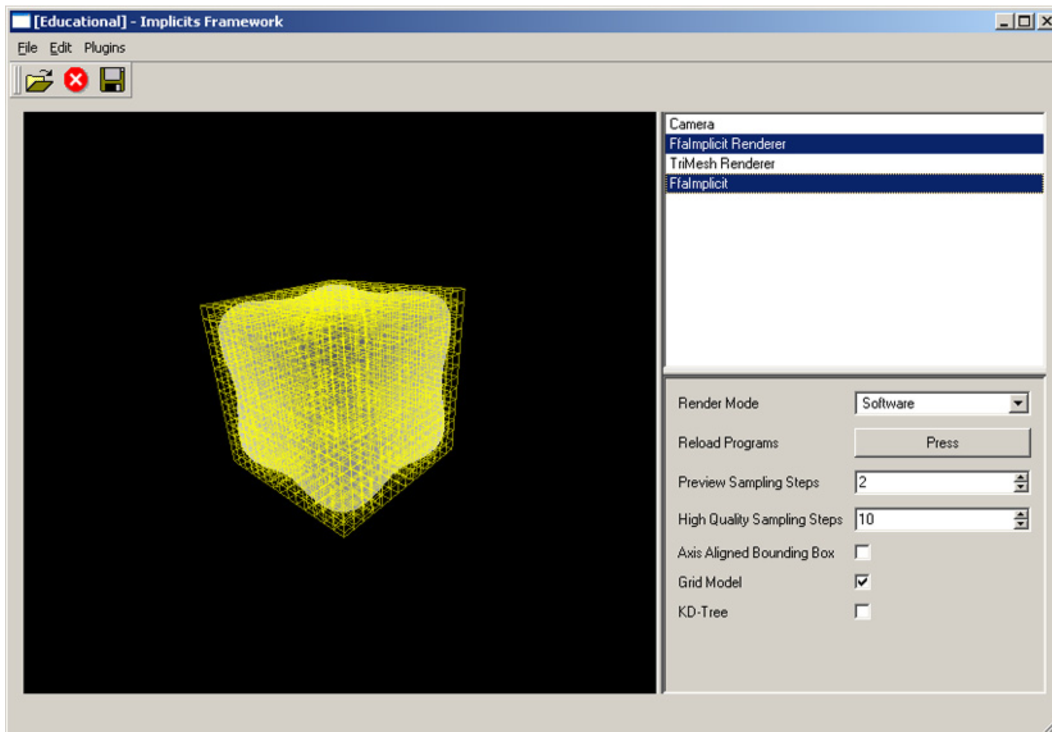


Abbildung 3.14: GUI des Frameworks

4
Implementation

In diesem Kapitel geht es um die Struktur des Programms an sich sowie um seine Einbettung in das zur Verfügung gestellte Framework. Hier soll weiterführenden Arbeiten einen einfacheren Einstieg in den vorhandenen Code ermöglicht werden. Es wird auf die Klassenstruktur eingegangen mit Schwerpunkt auf dem KD-Baum und dem Software Renderer. Des weiteren sollen auch einige der verwendeten Tricks hervorgehoben werden, die es ermöglichen, dass der Speicherplatz im Rahmen gehalten wird und möglichst wenig unnötige Operationen durchgeführt werden. Dazu leistet vor allem der in Kapitel 3 diskutierte KD-Baum einen entscheidenden Beitrag. Auch werden kurz die verwendeten Technologien erwähnt.

4.1 Zentrale Klassen

4.1.1 Zellen

Das Datengitter, das mit dem Raycaster gesampelt wird besteht aus Zellen. Diese sind im Code in den Klassen *CsgCell* und *Cell* implementiert. Jede *CsgCell* beinhaltet ihre Position im Objekt Raum, einen CSG Baum, der die CSG Operationen auf den einzelnen *Cells* definiert (siehe Kapitel 3.2), sowie Pointer auf die *Cells* an dieser Position. Anders ausgedrückt heisst das, dass an jeder Position im Objekt Raum, wo es einen Beitrag zur Isofläche gibt, sich eine *CsgCell* befindet und diese wiederum mehrere *Cells* (die alle an der gleichen Position wie die *CsgCell* sind) über den in ihr gespeicherten CSG Baum zum resultierenden Oberflächenstück verknüpft.

Als wichtigste Member Variable enthält jede *Cell* einen Pointer auf die eigentlichen Daten, die in Slices im Memory sehr kompakt gespeichert sind (siehe Abbildung 3.13). Dies sind die Daten, die wir mit dem Raycaster sampeln und anschliessend rendern.

Am Schluss dieses Kapitels ist eine Übersicht der Datenstruktur von Christian Sigg zu finden.

4.1.2 KD-Baum

Die *CsgCells* sind über zwei verschiedene Datenstrukturen erreichbar. Zum einen über das *CellSet* und zum andern über den *CellTree*, die beide im *CellGrid* gespeichert sind. Das *CellSet* ist ein Hashset, das einen direkten Zugriff auf die *CsgCells* ermöglicht. Dies ist zum Beispiel nützlich, wenn wir den Gradienten (Normale fürs Phong Shading) als zentrale Differenz auch

über Zellgrenzen hinaus berechnen möchten. Der *CellTree* ist die Implementation des in Kapitel 2 beschriebenen KD-Baumes.

Der KD-Baum besteht aus inneren Knoten, die die aktuelle Splitplaneposition und Achse beinhalten, sowie Blättern, die Pointer auf lineare *CsgCell* Listen darstellen.

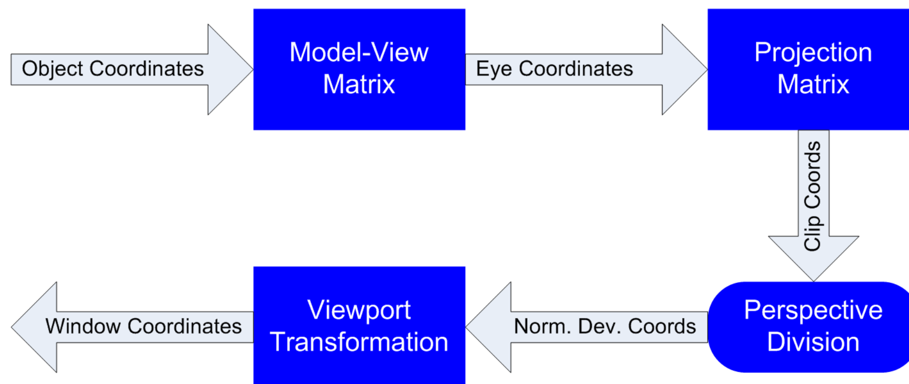
Alle Elemente des Baumes (innere Knoten und Blätter) sind als 32bit Datentypen gespeichert. Da alle *CsgCells* 4-aligniert sind, sind die letzten zwei Bits immer Null. Dies nutzen wir nun aus, um die Achse der Splitplane zu speichern sowie ein Flag zu setzen ob es sich um einen inneren Knoten oder ein Blatt handelt.

Der Baum wird intern als Array gespeichert. Um das Traversieren des Baumes zu vereinfachen wurde ein *Navigator* implementiert, der die Arraystruktur verbirgt.

4.2 Sichtstrahlen bestimmen

Um den Strahl, der von der Kamera aus in den Objektraum hinein verfolgt wird, zu definieren, können wir die OpenGL Koordinatentransformation wie sie in [1] Kapitel 2.11 beschrieben ist, rückgängig machen.

Der Prozess, den wir umkehren wollen ist in folgender Abbildung illustriert.



Mathematisch ausgedrückt bedeutet das, dass die Objektkoordinaten $[x_o, y_o, z_o, w_o]$ zuerst in Eye-Koordinaten umgewandelt werden müssen:

$$[x_e, y_e, z_e, w_e] = M \cdot [x_o, y_o, z_o, w_o]$$

Anschliessend werden diese in Clip-Koordinaten umgerechnet:

$$[x_c, y_c, z_c, w_c] = P \cdot [x_e, y_e, z_e, w_e]$$

Nun folgt die perspektivische Division:

$$[x_d, y_d, z_d] = [x_c/w_c, y_c/w_c, z_c/w_c]$$

Den Schluss bildet die Viewport Transformation. Diese ist bestimmt durch die Breite und Höhe des Viewports (p_x, p_y), sowie dessen Mittelpunkt (o_x, o_y):

$$\begin{bmatrix} x_w \\ y_w \\ z_w \end{bmatrix} = \begin{bmatrix} (p_x/2)x_d + o_x \\ (p_y/2)y_d + o_y \\ [(f-n)/2]z_d + (n+f)/2 \end{bmatrix}$$

Um nun von den Window Koordinaten der Pixel auf dem Bildschirm zu den Objekt Koordinaten zu kommen müssen all die oben erwähnten Transformationen in umgekehrter Richtung angewendet werden.

4.3 Programmatischer Ablauf

4.3.1 Call Hierarchy: Vom Zahlengitter zum gerenderten Bild

In diesem Unterkapitel soll der Ablauf, vom Öffnen eines Volume Files bis zur Darstellung auf dem Bildschirm, kurz erläutert werden.

4.3.1.1 Daten einlesen

Als erstes wird eine Datei mit der Endung .vol geöffnet und eingelesen. Diese beinhaltet die Definition der Daten sowie den Namen der Datei, die die Isoflächendefinition enthält. Nach dem Einlesen wird die vorhandene Information im Memory als Slices gespeichert. Im nächsten Schritt werden die *CsgCells* und die *Cells* entsprechend instanziert und in die beiden Datenstrukturen (*CellTree* und *CellSet*) eingefügt. Dies ist Aufgabe des *FfaImplicitReaders*.

4.3.1.2 Rendering

Nun sollen die Daten gerendert, d.h. als Bild sichtbar gemacht werden. Dieser Task wird vom *FfaImplicitRenderer* ausgeführt.

Er berechnet für jeden Pixel des Viewports einen Ray (Sichtstrahl) und verfolgt diesen in die Szene hinein. Konkret heisst das, dass für jeden Ray der KD-Baum traversiert und der Unterraum identifiziert wird, der mögliche Isoflächen-Intersektionen enthält. In diesem Unterraum wird nun zuerst die Bounding Box jeder *CsgCell* auf einen Schnitt mit dem Ray hin überprüft. Fällt dieser Test positiv aus, so wird innerhalb der *CsgCell* der *CsgTree* ausgewertet und die resultierende Isofläche auf einen Schnitt mit dem Ray hin untersucht. Existiert kein Schnitt, so wird mit der nächsten *CsgCell* weitergefahren. Sobald ein Schnitt gefunden wird, kann die Normale approximiert und das Phong Shading berechnet werden. Das Resultat dieser Operationen liefert die Farbe des Pixels im Viewport.

4.4 Verwendete Technologien

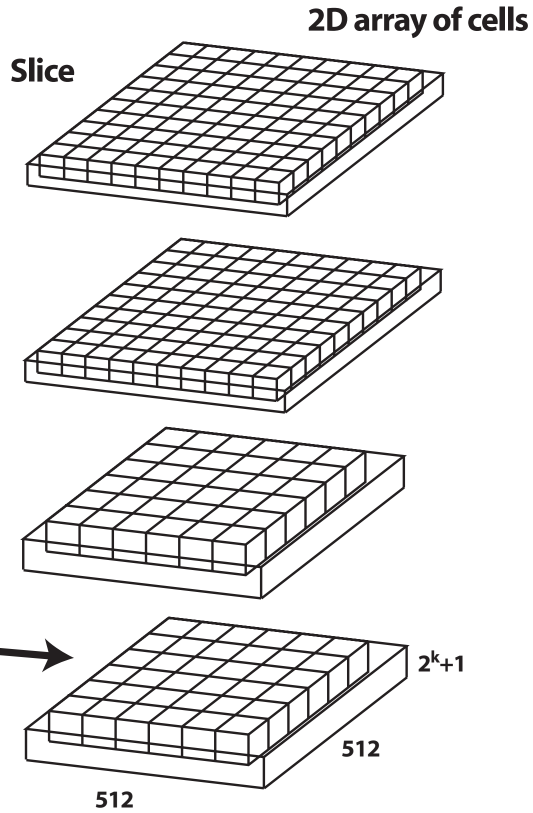
Die Arbeit wurde mit Microsoft Visual C++ implementiert. Als IDE verwendeten wir Visual Studio .NET. Die graphischen Elemente basieren auf OpenGL und QT.

Einige Elemente des Codes stammen aus der boost Library, die unter http://www.boost.org zu finden ist.

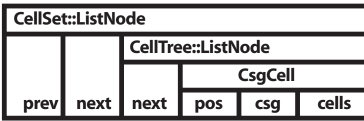
FfaImplicit

Datastructure Overview

Celldata memory managed in slices, stack them in GPU memory

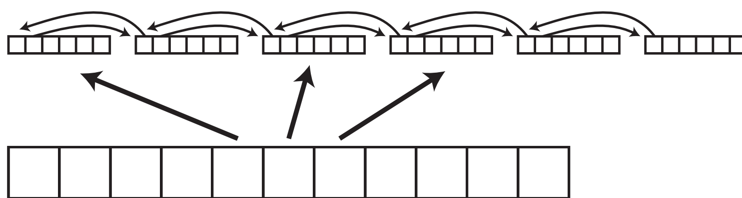


CsgCell inside a slist node and a list node, managed by small object allocator



Cell referencing one block of data in texture and in main memory

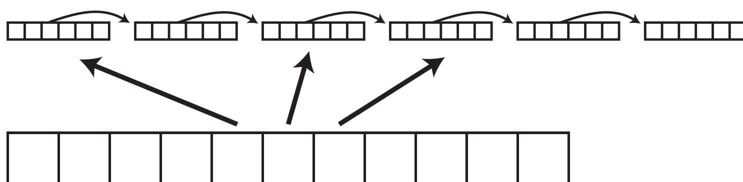
Hashset for direct cell access



doubly linked list containing slist nodes

vector of hash buckets

KdTree for spatial cell search



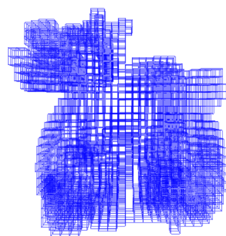
singly linked list containing cell references

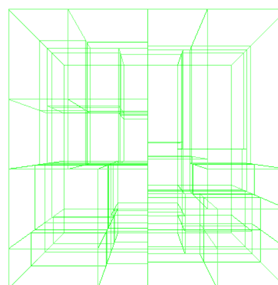
left balanced tree stored as array

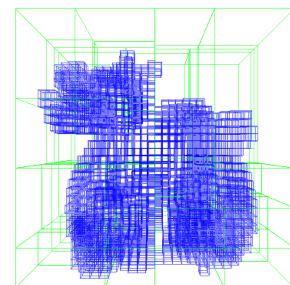
5 Resultate

Thema dieses Kapitels ist die Zusammenstellung der erzielten Resultate. Auch sollen Anregungen und Verbesserungen für weiterführende Arbeiten, die den Rahmen dieser Semesterarbeit gesprengt hätten, gegeben werden. Es werden sowohl visuelle Ergebnisse wie auch Performance Messungen aufgeführt. Dies ist besonders im Vergleich zu Verfahren ohne Beschleunigungsstrukturen oder der von Olivier Chassot unter dem gleichen Framework implementierten Arbeit mit Hardwarebeschleunigung interessant.

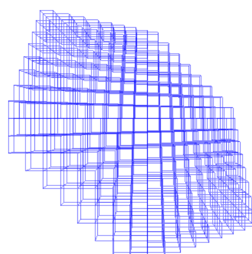
5.1 KD-Baum

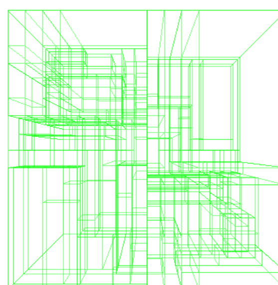


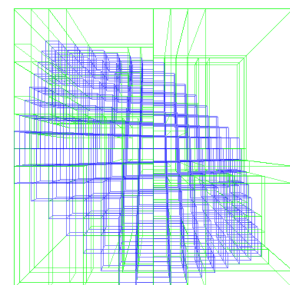




Dragon128s





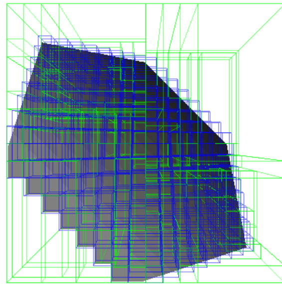
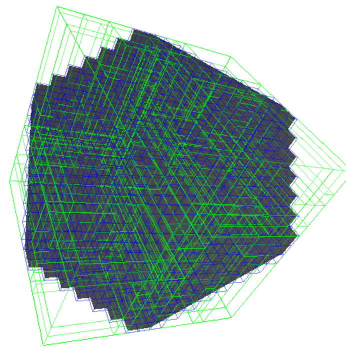
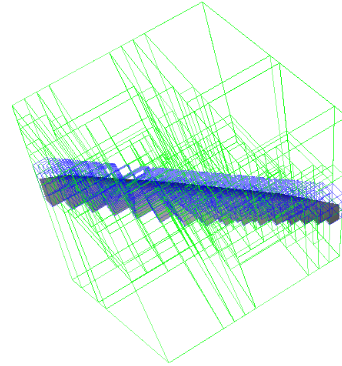


Diagonal64s

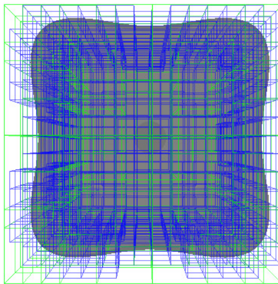
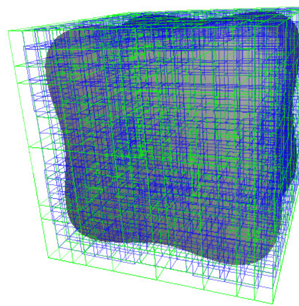
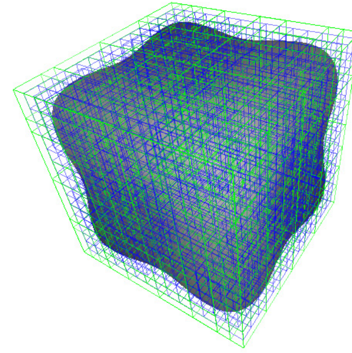
Links sind die im Objekt Raum gespeicherten *CsgCells* zu sehen. Es werden nur Zellen gespeichert, die etwas zur Oberflächendefinition beitragen. Deshalb erkennt man bereits hier grob die Form des Objektes. In der Mitte ist der daraus berechnete KD-Baum zu sehen und rechts eine Überblendung von beiden.

5.2 Gerenderte Isoflächen

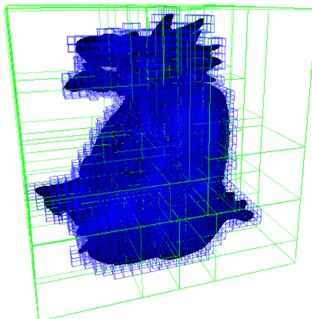
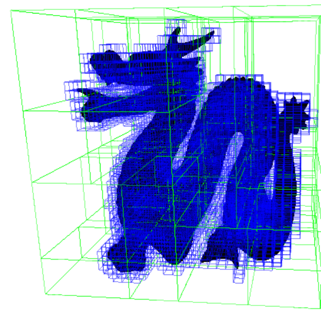
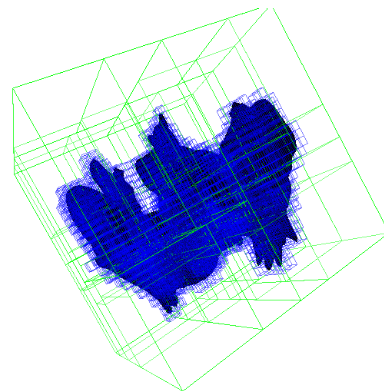
5.2.1 Grayscale depth image

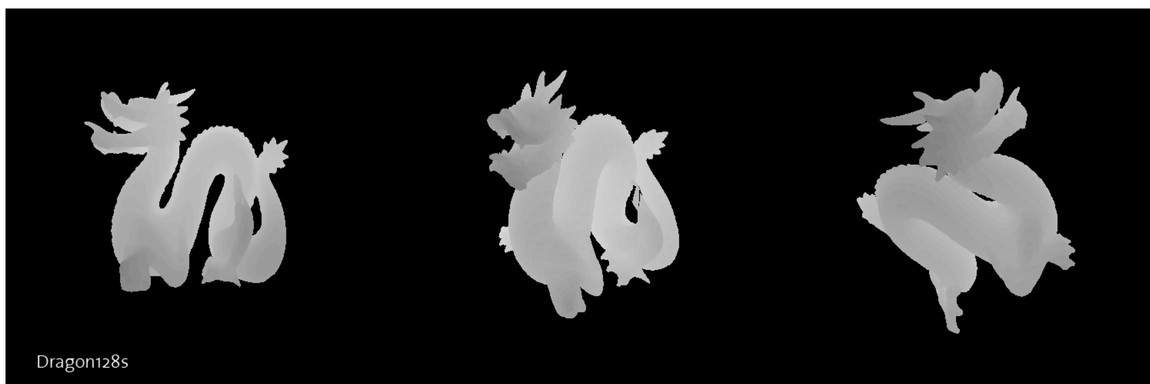
Diagonal64s

Gordon64s

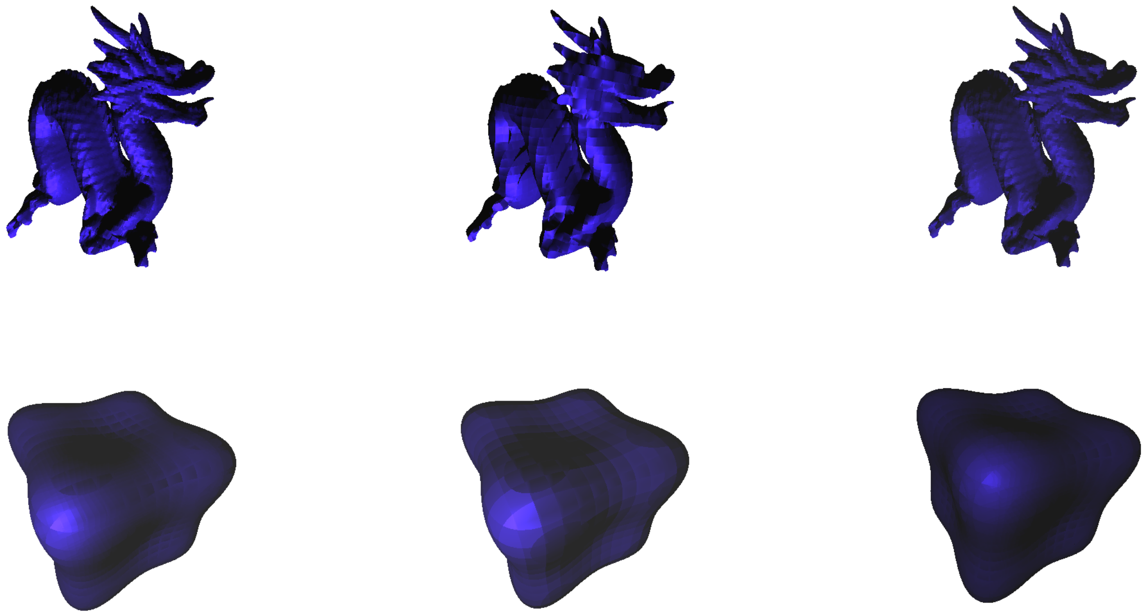




Dragon128s



Dragon128s

5.2.2 Phong Shading



Die Spalte ganz links zeigt den *Drachen128s* und den *Gordon64s,* wobei die Oberflächennormale fürs Shading als Ableitung der trilinearen Interpolation approximiert wurde. In der Mitte sind die Objekte mit den gleichen Beleuchtungsparametern aber anderer Normalenapproximation dargestellt. Hier wurde die Methode der finiten Differenzen verwendet. Das Resultat liesse sich noch optimieren, wenn die finiten Differenzen über Zellgrenzen hinweg berechnet würden. Ganz rechts sind die Modelle nochmals mit der ersten Methode beleuchtet, jedoch mit etwas abgeänderten Beleuchtungsparametern.

5.3 Benchmarking

Im Folgenden werden die Resultate der Performance Messungen zusammengefasst. Gerendert wurden die drei Isoflächen *Diagonal64s, Gordon64s* sowie *Dragon128s*, die auf der vorhergehenden Seite zu sehen sind. Jede dieser Isoflächen wurde zwanzigmal mit zehn Samples pro Zelle gerendert und die dafür benötigte Zeit ermittelt. Daraus wurde anschliessend die Anzahl Frames, die pro Sekunde gerendert werden können, berechnet [fps]. Diagonal64s und Gordon64s sind mit einer Resolution von 64x64x64 gespeichert und der Dragon128s mit 128x128x128.

Die Viewportgrösse betrug bei allen Messungen 512x471 Pixel wobei die AABB im Viewport, um das zu rendernde Objekt, jeweils 201x201 Pixel gross war. Fürs Rendering wurde ein Notebook mit einem Intel Pentium 1GHz Prozessor verwendet.

max. # *CsgCells* per Subspace	*Diagonal64s (574 cells)*	*Gordon64s (1190 cells)*	*Dragon128s (4441 cells)*
10000	0.5	0.2	0.07
100	2.1	1.9	1.3
50	3	2.6	1.75
20	3.5	3.5	1.9
10	3.5	3.5	1.9
4	4.2	1.9	1.3

Tabelle 5.1: Renderingzeiten der drei Objekte mit KD-Baum [fps]

Da keine der Isoflächen aus 10000 Zellen oder mehr besteht, entspricht die erste Messung dem Rendering ohne Beschleunigungsstruktur. Die Zellen sind in diesem Fall alle in einer linearen Liste gespeichert und jede muss einzeln untersucht werden.

Bei den anderen Messungen wurde die maximale Anzahl Zellen, die pro Unterraum erlaubt sind, schrittweise verkleinert. Man erkennt schon bei 100 Zellen pro Unterraum einen deutlichen Geschwindigkeitsvorteil gegenüber der linearen Liste. Am markantesten ist diese Beschleunigung beim Modell Dragon128s, bei dem eine Beschleunigung um den Faktor 27 auszumachen ist. Beim Gordon64s beträgt die Beschleunigung 17 und bei beim Diagonal64s noch rund 8. Dies zeigt, dass die Beschleunigung umso bedeutender ist, je komplexer das Objekt wird.

Interessant ist auch die Tatsache, dass es sich offenbar nicht in jedem Fall lohnt den Baum sehr tief und die Listen in den Blättern dadurch sehr kurz zu machen. Der Gordon64s und der Dragon128s sind mit maximal 10 bis 20 Zellen pro Unterraum performanter als mit maximal 4 Zellen pro Unterraum.

5.4 Anregungen für weiterführende Arbeiten

5.4.1 Rendering

Adaptives Sampling innerhalb der *CsgCell*

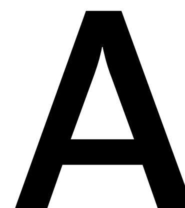
Transparenz

Reflexionen (mehrfach)

5.4.2 Beschleunigungsstruktur

Varianten für die Wahl der Splitplanes im KD-Baum (bezüglich Orientierung und Position)

Rebalancing des Baumes durch Rotationen



Referenzen

[1] SEGAL, M. AND AKELEY, K. The opengl graphics system: A specification, version 2.0, 2004.

[2] Standard Template Library Programmer's Guide
http://www.sgi.com/tech/stl/

[3] GROSS, M. Introduction to Computer Graphics (WS 2003/04)
http://graphics.ethz.ch/

[4] PEIKERT, R. Graphische Datenverarbeitung II (SS 2004)
http://graphics.ethz.ch/

[5] HADWIGER, M., SIGG, C., SCHARSACH, H., BÜHLER, K. AND GROSS, M. Real-Time Ray-Casting and Advanced Shading of Discrete Isosurfaces
http://graphics.ethz.ch/~siggc/publications/eg05.pdf

[6] CHASSOT, O. AND SIGG, C. Direct Rendering of Implicit Surfaces

[7] BIKKER, J. Raytracing Topics & Techniques (1-7)
http://www.flipcode.com/articles/article_raytrace01.shtml

[8] HAVRAN, V. Heuristic Ray Shooting Algorithm Chapter 5: Ray Traversal Algorithms for Kd-Trees

[9] PHONG, B. T. Illumination for computer generated images. *Communications of the ACM 18:311-317* (1975).

[-] Abbildung 3.1
http://www.cs.fh-aargau.ch/~gdv/script/html/node84.html

[-] Abbildung 3.7
http://www.f-lohmueller.de/pov_tut/csg/povcsg2e.htm

[-] Abbildung 3.8
http://www.cs.mtu.edu/~shene/COURSES/cs3621/NOTES/model/csg.html

[-] Abbildung 3.10
http://graphics.ethz.ch (GDVI Slides)