Studiengang:	Softwaretechnik
Prüfer:	Prof. Dr. rer. nat. H.-J. Bungartz
Betreuer:	Dipl. Inf. R.-P. Mundani
begonnen am:	15. August 2002
beendet am:	14. Februar 2003
CR-Klassifikation:	E.1, J.6, F.2.2, G.1.2

Diplomarbeit Nr. 2035

Erzeugung und Evaluierung von Oktalbaumstrukturen als Schnittstelle zu CAD-Programmen

Stefan Mahler

Institut für Parallele und
Verteilte Systeme
Universität Stuttgart
Breitwiesenstraße 20–22
D–70565 Stuttgart

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1

Einleitung

In den letzten Jahrzehnten hat sich das rechnergestützte Entwickeln neuer Produkte durchgesetzt. So ist heutzutage das CAD z.B. aus der Automobilindustrie nicht mehr wegzudenken. Neben dem Modellieren kommt auch der Simulation und Visualisierung der daraus resultierenden physikalischen Zusammenhänge eine große Bedeutung zu. Mit Hilfe der Simulation können Szenarien nachvollzogen, optimiert bzw. verstanden werden. Computersimulationen finden in unterschiedlichsten Bereichen eine breite Anwendung (vgl. [Bun02a]). Einen besonders hohen Stellenwert haben Simulationen in Bereichen, in denen die Simulationen auf physikalischen Modellen basieren. Beispiele hierfür sind der Automobilbau (Strömungssimulation, Crashtests) oder das Bauingeneurwesen (Häuser in Erdbebengebieten, Brückenstatiken). Simulationen sind häufig aus ethischen (Belastbarkeit in Extremsituationen) oder betriebswirtschaftlichen (Kostenreduktion, Verminderung des Zeitaufwands) Gesichtspunkten vorteilhaft. Unter Umständen ist die Alternative – ein mechanisches Modell – auch gar nicht möglich. So können Computersimulationen z.B. helfen, Schwächen in einem neuentworfenen Produkt schneller und kostengünstiger zu erkennen als das vielleicht an einem mechanischen Modell möglich wäre.

Die Vorteile der Nutzung des Computers zur Modellierung, Simulation und Visualisierung innerhalb verschiedenartigster Anwendungsgebiete liegen auf der Hand. Natürlich ist auch eine integrierte Lösung anzustreben. Es sollte also ein gemeinsames Modell für Simulation und Visualisierung genutzt werden können. Die Nutzung nur *einer* Datenbasis vermeidet Inkonsistenzen innerhalb der Daten. Mühsame und zeitaufwändige Neueingaben der Produktdaten für Simulation und Visualisierung werden unnötig. Doch genau hier liegt das Problem für einen integrierten Prozess, der sowohl Modellierung als auch Simulation und Visualisierung umfasst.

Zur Datenhaltung in zur Modellierung verwendeten CAD-Systemen kommen häufig oberflächenorientierte Datenmodelle oder das CSG-Modell zum Einsatz. Diese erscheinen auch hierfür am leistungsfähigsten, sind jedoch für die Simulation ungeeignet. Zur Simulation ist eine Datenbasis als diskretisiertes Volumenmodell notwenig. [Fra00]

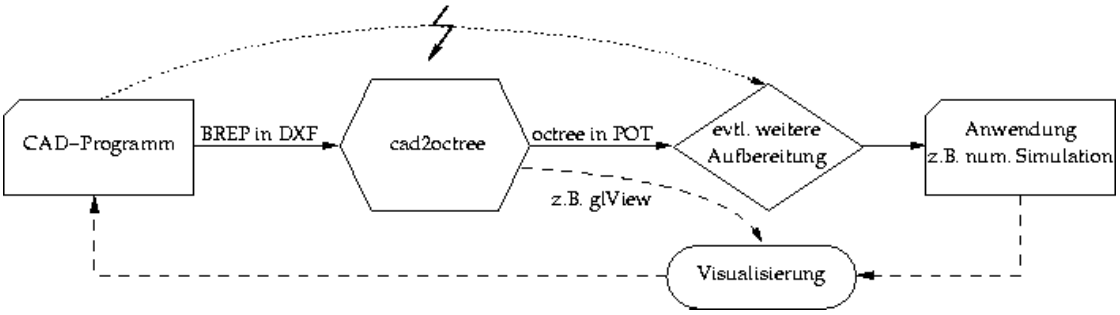


Abbildung 1.1: Octree als Schnittstellte zu CAD-Programmen

zeigt, dass die Oktalbaumstruktur als effiziente Lösung dieses Problems genutzt werden kann. Abbildung 1.1 stellt diese Zusammenhänge schematisch dar.

Thema dieser Arbeit ist, ein Verfahren zu entwickeln, welches aus gegebenen CAD-Daten, also einem Oberflächenmodell, die zugehörige Oktalbaumstruktur generiert.

Kapitel 2 gibt einen Überblick über die wichtigsten Formen zur Geometriebeschreibung. Besondere Aufmerksamkeit werden dabei den raumpartitionierenden Strukturen, insbesondere dem Octree, gegeben. Einen weiteren Schwerpunkt in diesem Kapitel bilden Splines, die als Beschreibungsmittel für Freiformflächen dienen.

Für den Import des CAD-Modells wird auf das DXF-Format zurückgegriffen. Weshalb die Entscheidung auf dieses Format fiel und welche Eigenschaften es besitzt, ist Inhalt von Kapitel 3.

Im Kapitel 4 werden grundlegende Verfahren zur Bearbeitung geometrischer Gebilde, wie Schnitt zweier Geraden im 3-Dimensionalen oder der Punkt-in-Ebene-Test erläutert. Es werden die für die Generierung der Oktalbaumstruktur aus dem CAD-Modell zugrundeliegenden Algorithmen beschrieben. Für manche Teilprobleme können unterschiedliche Algorithmen verwendet werden. Vor- und Nachteile der verschiedenen Lösungsalgorithmen werden gegenübergestellt.

Das im Rahmen dieser Arbeit erstellte Programm wird in unterschiedlichen Szenarien auf seine Effizienz untersucht. Die Ergebnisse sind neben implementierungtechnischen Details im Kapitel 5 dargestellt.

Abschließend fasst Kapitel 6 die erzielten Ergebnisse zusammen und zeigt für das entwickelte Programm Erweiterungsmöglichkeiten.

An dieser Stelle möchte ich mich bei Prof. Dr. Hans-Joachim Bungartz bedanken, der mir ermöglichte, dieses interessante Thema in dieser Diplomarbeit zu bearbeiten. Besonderen Dank geht an meinen Betreuer Dipl. Inf. Ralf-Peter Mundani für die vielen guten Ratschläge, die moralische Unterstützung, das Korrekturlesen, die Zeit, die er sich genommen hat, um mit mir offene Fragen und Probleme zu erörtern und dass er mir den Oktalbaumbetrachter `glView` zur Verfügung stellte.

Kapitel 2

Darstellungsarten eines Körpers

Die Beschreibung und Darstellung geometrischer Körper wird als geometrische Modellierung bezeichnet. Zur rechnerorientierten Beschreibung solcher Modelle sind entsprechende Datenstrukturen erforderlich.

Im Weiteren wird ausschließlich die Modellierung von Körpern mit zeitlich fester Geometrie (also keine dynamischen Geometrien) behandelt.

Es gibt unterschiedliche Repräsentationsmöglichkeiten der geometrischen Verhältnisse. Zur Bildung dreidimensionaler Modelle existieren kanten-, flächen- und volumenorientierte Datenstrukturen. Diese besitzen verschiedene Eigenschaften. Abschnitt 2.1 gibt einen Überblick über wichtige Oberflächen- und Volumenmodelle und deren Eigenschaften.

Volumenmodelle werden auch als direkte Schemata bezeichnet. Die Information über die Körpergeometrie und die Lokalisation der Körper kann direkt aus dem Datenmodell extrahiert werden. Im Gegensatz dazu sind Oberflächen- und Kantenmodelle indirekte Schemata. Um die Gestalt eines Körpers ermitteln zu können, müssen hier z.T. aufwändige zusätzliche Berechnungen erfolgen. Durch das Modell kann nicht sichergestellt werden, dass die Datenstruktur nur Rigid-Bodies (die Randgebiete sind geschlossen; es gibt also z.B. beim Oberflächenmodell immer eine Begrenzungsfläche zwischen Körperinnerem und -äußerem) repräsentiert. Entsprechnend aufwändig ist i.A. die Berechnung, ob sich ein Raumpunkt im Körperinneren befindet oder nicht. Je nach Fachgebiet werden bestimmte Repräsentationsformen bevorzugt, die für die jeweilige Anwendung am besten geeignet scheinen. Kantenorientierte Darstellungen sind für die Darstellung eindimensionaler Geometrieen gut geeignet. Volumen- und Oberflächenmodelle sind in der Lage, auch komplexe dreidimensionale Objekte zu beschreiben. Während bei der geometrischen Modellierung Oberflächenmodelle zumeist in CAD-Systemen verwendet werden, kommen bei Simulationen häufig Berechnungsgitter zum Einsatz, die durch räumliche Diskretisierung erzeugt wurden, also spezielle Volumenmodelle. Die Nutzung unterschiedlicher geometrischer Modelle ist eines der

größten Hindernisse zur Integration u.a. von CAD und Simulation. Abschnitt 2.2 erklärt die Besonderheiten der Oktalbaumstruktur, die es ermöglichen, sie als Schnittstelle zwischen CAD und Simulation zu verwenden.

2.1 Wichtige Oberflächen- und Volumenmodelle

Der folgende Abschnitt widmet sich den wichtigsten Oberflächen- und Volumenmodellen. Umfangreichere Beschreibungen liefern [Bun02b, Brü01, Aum93, Par90, Abr91].

2.1.1 Oberflächenmodelle

Von Oberflächendarstellungen spricht man, wenn ein Objekt nicht durch sein Volumen, sondern – indirekt – durch die das Objekt begrenzenden Flächen beschrieben wird. Sie werden deshalb auch boundary representations oder kurz B-Rep genannt.

Allerdings repräsentiert nicht jede Ansammlung von Flächen einen starren Körper im Sinne der geometrischen Modellierung. Hiermit ist gemeint, dass für jeden beliebigen Raumpunkt entschieden werden kann, ob er sich innerhalb, außerhalb oder auf der Oberfläche des Körpers befindet. Um dies zu gewährleisten, müssen weitere Bedingungen erfüllt sein:

1. **Geschlossenheit:** Jeder Kantenpunkt ist Randpunkt zweier Flächen. Eckpunkte werden von jeweils mindestens drei Flächen gebildet. Der Rand einer jeder Fläche besitzt genauso viele Kanten wie Eckpunkte. Es gibt also insbesondere keine Unterbrechungen in den Kanten oder Löcher in den Flächen, es sei denn, dass der Lochrand von weiteren Flächen, wie bei einem Hohlzylinder, ummandelt ist.
2. **Orientierheit der Oberflächen:** Eine Fläche ist dann orientierbar, wenn sie zwei wohlunterscheidbare Seiten besitzt. Bedingung 1 und Bedingung 3 garantieren zusammen die Orientierbarkeit der Oberflächen.
3. **Kein Eigenschnitt:** Es muss noch gewährleistet sein, dass die Flächen und Körper sich nicht selbst schneiden.

Sind alle drei Bedingungen erfüllt, kann durch die Orientierung der Flächen festgelegt werden, welche Punkte sich innerhalb bzw. außerhalb eines Körpers befinden. Besteht das Oberflächenmodell aus mehreren Zusammenhangskomponenten, ist dies auch notwendig, um entscheiden zu können, ob es sich z.B. hierbei um ein Hohlkörper oder mehrere Körper handelt, da sonst eventuell mehrere Zuordnungsmöglichkeiten von Körperäußerem und Körperinnerem bestehen.

Als Teilflächen sind im einfachsten Fall Dreiecke möglich. Diese Form der Modellierung eines Körpers wird auch als Triangulierung bezeichnet. Die Körperoberflächen werden also hierbei durch Dreiecksnetze approximiert. Der Vorteil des Einsatzes von

Dreiecksnetzen liegt in dem in Vergleich zu anderen Oberflächenmodellen relativ einfachen Handling der Geometrie:

- Zur Beschreibung eines Dreiecks sind immer genau 3 Eckpunkte erforderlich, woraus sich eine einfache Struktur für ein Oberflächenelement 'Dreieck' ergibt.
- Dreiecke spannen immer eine Ebene auf, die durch die 3 Punkte genau beschrieben werden. (Bei Vierecken beispielsweise könnte der vierte Eckpunkt außerhalb der Ebene, die durch die 3 anderen Eckpunkte aufgespannt wird, liegen.)
- Dreiecke sind stets konvexe Polygone. Für den Test, ob ein Punkt innerhalb oder außerhalb eines Dreiecks liegt, gibt es aufgrund dieser Eigenschaft effiziente Algorithmen.
- Welche Seite des Dreiecks Körperaußenfläche und welche Innenfläche ist, lässt sich leicht aus den drei Eckpunkten bestimmen, wenn folgende Festlegung bezüglich der Reihenfolge der Eckpunkte eingehalten wird:
  Der Normalenvektor der Ebene $\overrightarrow{n_E}$, der durch die Vektoren aufgespannt wird, die sich zwischen zweiten (B) bzw. drittem Eckpunkt (C) als Endpunkte und dem ersten Eckpunkt (A) als Anfangspunkt ergeben:

$$\overrightarrow{n_E} = \frac{\overrightarrow{AB} \times \overrightarrow{AC}}{|\overrightarrow{AB}||\overrightarrow{AC}|}, \tag{2.1}$$

  ist immer in Richtung des Körperäußeren gerichtet.

Besitzt ein Körper eine Oberfläche mit starken Krümmungen, so müssen die Dreiecke sehr klein gewählt werden, um den Körper überhaupt approximieren zu können. Das erzeugte Modell wirkt kantig, obwohl der zu modellierende Körper (z.B. bei einer Kugel) keine Kanten besitzt. Meist werden jedoch stetige Verläufe und nicht kantige Modelle benötigt. Zur Beschreibung von Körpern mit gekrümmter Oberfläche werden daher häufig Freiformkurven-/flächen zur Modellierung eingesetzt. Die Nutzung von Freiformflächen erlaubt die exakte Modellierung von Kugeln, Kegeln und Kugel-/Kegelteilkörpern. Freiformkurven-/flächen eigenen sich insbesondere für die geometrische Modellierung der häufig benötigten $\mathcal{C}^2$-stetigen[1] Verläufe.

Im Folgenden wird auf die Freiformtypen Bézier-Kurven, B-Splines und NURBS eingegangen. Anschließend wird die Freiformflächenerzeugung aus dem Freiformkurven-Schema hergeleitet. Für weiterführende Erläuterungen ist [Far94] zu empfehlen. [dB87, dB90] geben nähere Erläuterungen für das Verständnis von Splinefunktionen. Hier werden Algorithmen zum Einfügen und Löschen von Knoten ausführlich beschrieben. [Pie97] gibt eine ausführliche und dennoch übersichtliche Darstellung wesentlicher Zusammenhänge zwischen Bézier-Kurven/-Flächen, Splines und NURBS. Es werden für den Umgang wichtige Algorithmen vorgestellt. Das Buch enthält auch praktische Tipps zur Anwendung der Algorithmen und Implementierungen der Verfahren im C-Code.

[1] Die Kurve/Fläche ist zwei Mal (partiell) differenzierbar. Die zweite Ableitung ist stetig.

Um neben glatten Oberflächen auch gekrümmte Verläufe modellieren zu können, wird nun die lineare Darstellung zu einer polynomiellen erweitert. Polynome besitzen für die Modellierung gekrümmter Verläufe viele geeignete Eigenschaften:

- Sie sind über den ganzen Bereich der reellen Zahlen definiert und sind n-mal differenzierbar, wenn n der Grad des Polynoms ist. Sie haben also insbesondere keine Unstetigkeitsstellen oder 'Knickstellen'.
- Mit $n+1$ Punkten können beliebige Polynome n-ten Grades eindeutig dargestellt werden und umgekehrt.
- Polynome sind relativ leicht berechenbar. Es müssen also keine aufwändigen Berechnungsroutinen implementiert werden. Des Weiteren sind Polynome einfach analysierbar.
- Lineare Funktionen können als Polynome ersten Grades aufgefasst werden und sind somit nur ein spezielles Polynom.

Die polynomielle Interpolation erweist sich jedoch in vielen Fällen als ungünstig, da sie für Polynome höheren Grades im Allgemeinen unerwünschte numerische Eigenschaften besitzt: Die Funktion ist schlecht konditioniert. Kleine lokale Änderungen haben starke globale Veränderung. Diese Veränderungen sind nicht intuitiv. Schon deshalb ist dieser Ansatz für den konstruktiven Entwurf unbrauchbar.

Daraus ergeben sich zwei Kriterien:

Kontrollierbarkeit: Veränderungen der Kurven-/Flächen-Parameter führen zu voraussagbaren und kontrollierbaren Effekten bezüglich der Kurvengestalt.

Lokalitätsprinzip: Lokale Veränderungen bedingen maximal nur geringe globale Veränderungen. Verschiebt man z.B. einen Kontrollpunkt, so verändert sich der Verlauf entfernter Kurven-/Flächenstücke nur wenig oder – idealerweise – garnicht.

Die drei Approximationsverfahren, die Modellierung durch Bézier-Kurven, Splines und NURBS, erfüllen diese beiden Kriterien.

Bézier-Kurven

Eine Kurve, die durch

$$\mathbf{x}(t) = \sum_{i=0}^{n} B_i^n(t)\, \mathbf{b}_i, \quad t \in [0,1] \tag{2.2}$$

beschrieben wird, wird als Bézier-Kurve bezeichnet. Dabei sind die Bernstein-Polynome B_i^n definiert durch

$$B_i^n(t) = \binom{n}{i} (1-t)^{n-i}\, t^i, \quad t \in [0,1],\ i = 0, \ldots, n. \tag{2.3}$$

Damit gilt

$$\sum_{i=0}^{n} B_i^n(t) = 1, \quad \forall t \in [0,1] \tag{2.4}$$

und

$$B_0^n(0) = B_n^n(1) = 1 \tag{2.5}$$

bzw.

$$B_0^n(1) = B_n^n(0) = 0. \tag{2.6}$$

Bézier-Kurven sind die einfachste Form einer symmetrischen Darstellung durch polynomielle Approximation mit Kontrollpunkten. Die Kontrollpunkte $\mathbf{b}_i$ einer Bézier-Kurve werden Bézier-Punkte genannt. Sie bilden ein Kontrollpolygon. Bézier-Kurven besitzen folgende wichtige Eigenschaften:

1. **Konvexe-Hüllen-Eigenschaft:** Eine Bézier-Kurve liegt in der konvexen Hülle ihres Kontrollpolygons.
2. **Stetigkeit, Differenzierbarkeit:** Bézier-Kurven sind stetig. Besitzt die Bézier-Kurve Mehrfachkontrollpunkte oder besitzt der die Kontrollpunkte miteinander verbindende Kantenzug Schleifen (das Kontrollpolygon ist also nicht konvex), kann die Kurve jedoch an einigen Stellen eventuell nicht differenzierbar sein.
3. **Interpolation der Kurvenenden:** Kontrollpolygon und Bézier-Kurve stimmen in Anfangs- und Endpunkt überein. Die inneren Bézier-Punkte werden i.A. nur approximiert und liegen somit dann auch nicht auf der Bézier-Kurve.
4. **Kurvenanstieg an den Kurvenenden:** Die Strecken $\overline{\mathbf{b}_0\mathbf{b}_1}$ und $\overline{\mathbf{b}_{n-1}\mathbf{b}_n}$ des Kontrollpolygons verlaufen im Anfangs- und Endpunkt tangential zur Bézier-Kurve.
5. **Einfluss von Kontrollpunkten, Globale Modifikation:** $\mathbf{b}_i$ hat den größten Einfluß auf $\mathbf{x}(t)$ bei $t = i/n$. Bis auf Anfangs- und Endpunkt der Bezierkurve werden zur Berechung eines Kurvenpunkts *alle* Kontrollpunkte (wenn auch in unterschiedlichem Ausmaß) berücksichtigt. Die Modifikation eines Kontrollpunktes wirkt sich global auf die Bézier-Kurve aus.
6. **Formerhaltungs-Eigenschaft:** Ist das Kontrollpolygon konvex, so ist es auch die Bézier-Kurve.
7. **Beschränkte Schwankung:** (auch *variation diminishing* genannt) Keine Gerade schneidet die Bézier-Kurve öfter als das entsprechende Kontrollpolygon. Daraus ist auch direkt die Eigenschaft der *linearen Präzision* ableitbar: Liegen die Kontrollpunkte $\mathbf{b}_0, \ldots, \mathbf{b}_n$ einer Bézier-Kurve kollinear, dann liegt die Bézier-Kurve auf der Strecke $\overline{\mathbf{b}_0\mathbf{b}_n}$.
8. **Graderhöhung:** Bei fortgesetzter Graderhöhung durch Einfügen zusätzlicher Kontrollpunkte konvergiert das Kontrollpolygon gegen die Bézier-Kurve.

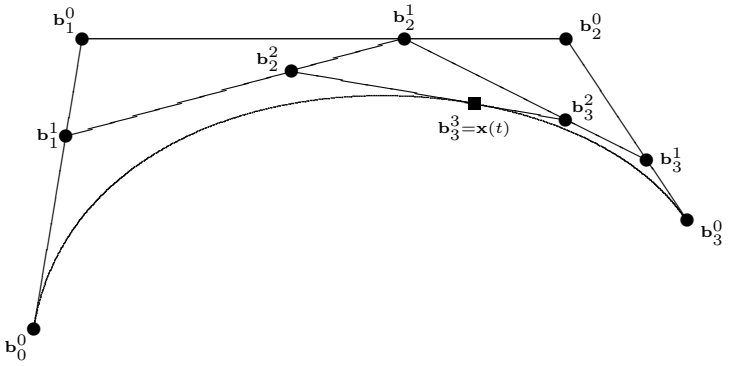


Abbildung 2.1: Algorithmus von de Casteljau (*geom.*)

9. **Affine Invarianz:** Bézier-Kurven sind bezüglich Rotation, Skalierung ($\neq 0$), Spiegelung und Translation invariant.

Eine effiziente und numerisch stabile Möglichkeit zur Berechnung der Kurvenpunkte $\mathbf{x}(t)$ ist der *Algorithmus von de Casteljau*. Abbildung 2.1 zeigt ein Schema zur geometrischen Konstruktion von $\mathbf{x}(t = \frac{2}{3})$ für $n = 3$ nach diesem Algorithmus.

Die Eigenschaften 2 und 5 können sich nachteilig auf die Modellierung von Bézier-Kurven auswirken. Diese Nachteile wie mangelnde Diffenrenzierbarkeit und unerwünschte Ausgleichseffekte von Kontrollpunkten infolge fehlender Lokalität der Kontrollpunktwirkung werden häufig bei höhergradigen Bézier-Kurven sichtbar. Zudem sind mathematische Beschreibungen der Bézier-Kurven vom Grad ≥ 10 unhandlich, Berechnungen sind rechenzeitintensiv. Andereseits erfordert das Nachbilden komplexer realer Kurven evtl. viele Kontrollpunkte.

Einen Ausweg bieten **zusammengesetzte Bézier-Kurven**. Die Gesamtkurve wird aus einzelnen Kurvensegmenten gleichen Grads zusammengesetzt. Das Segment i der Bézier-Kurve besteht somit aus den Kontrollpunkten $\mathbf{b}_{0,i}, \ldots, \mathbf{b}_{p,i}$, wenn die Segmente vom Grad p sind. Nun wird zweckmäßiger Weise ein Verfahren benötigt, was zu einem globalen Parameter u eindeutig den lokalen Parameter t des i-ten Segments bestimmt. O.B.d.A. sei angenommen, dass die Gesamtkurve normalisiert, also $u \in [0, 1]$, sei. Läuft u somit über $[0, 1]$ wird die Gesamtkurve überstrichen. Mit Hilfe der Zerlegung $0 = u_0 < u_1 < \ldots < u_n < u_{n+1} = 1$ des Intervalls $[0, 1]$ kann das i-te Segment über dem Intervall $I_i = [u_i, u_{i+1}]$ definiert werden. Der lokale Paramterwert t des Intervalls I_i kann dann mit

$$t = \frac{u - u_i}{u_{i+1} - u_i}, \quad u \in [u_i, u_{i+1}] \tag{2.7}$$

definiert werden.

Wegen Eigenschaft 3 lässt sich die Stetigkeit von zusammengesetzten Bézier-Kurven leicht bewerkstelligen: Hierfür müssen die Anfangspunkte des nachfolgenden Seg-

ments nur mit den jeweiligen Endunkten des Vorgängersegments übereinstimmen:

$$\mathbf{b}_{n,i} = \mathbf{b}_{0,i+1}. \tag{2.8}$$

Schwieriger lässt sich allerdings hinreichende Differenzierbarkeit an den Segmentübergängen gewährleisten. Um an den Segmentübergängen $\mathcal{C}^1$-Stetigkeit zu erhalten, muss

$$\frac{\mathbf{b}_{n,i} - \mathbf{b}_{n-1,i}}{u_{i+1} - u_i} = \frac{\mathbf{b}_{1,i+1} - \mathbf{b}_{n,i}}{u_{i+2} - u_{i+1}} \tag{2.9}$$

gelten. Dies bedeutet, dass die drei Punkte $\mathbf{b}_{n-1,i}$, $\mathbf{b}_{n,i} = \mathbf{b}_{0,i+1}$ und $\mathbf{b}_{1,i+1}$ kollinear sein müssen *und* dass $\mathbf{b}_{n,i}$ die Strecke $\overline{\mathbf{b}_{n-1,i}\mathbf{b}_{1,i+1}}$ im Verhältnis $(u_{i+1} - u_i) : (u_{i+2} - u_{i+1})$ teilt. Um die häufig geforderte $\mathcal{C}^2$-Stetigkeit zu erhalten, muss das Schema unter Einbeziehung der Punkte $\mathbf{b}_{n-2,i}$ und $\mathbf{b}_{2,i+1}$ entsprechend erweitert werden.

Nachteilig an zusammengesetzten Bézier-Kurven ist, dass die ersten und letzten Kontrollpunkte eines jeden Segments die Bedingungen (2.8) und (2.9) erfüllen müssen, um die Differenzierbarkeit der Gesamtkurve sicherzustellen. Dafür müssen eventuell zusätzliche Kontrollpunkte eingefügt werden. Dieses Problem kann durch die Nutzung von B-Spline-Kurven umgangen werden.

B-Splines

B-Spline-Kurven vom Grad p sind definiert durch

$$\mathbf{x}(u) = \sum_{i=0}^{n} N_i^p(u)\, \mathbf{c}_i, \tag{2.10}$$

wobei $\mathbf{c}_i$ die Kontrollpunkte und N_i^p die zur B-Spline-Kurve gehörigen Basisfunktionen über den Knotenvektor $U = \{u_0, \ldots, u_n\}$ darstellen. Analog zu den zusammengesetzten Bézier-Kurven ist $u_{\min} = u_0 < u_1 < \ldots < u_n = u_{\max}$ eine Zerlegung des Intervalls $[u_{\min}, u_{\max}]$. Wieder ergeben sich Teilkurven zu jedem Teilintervall. Die Glattheit der Gesamtkurve ist hier jedoch auch über die Stoßstellen der Teilkurven gesichert, da B-Spline-Basisfunktionen anstelle der Bernsteinpolynome verwendet werden.

Die Basisfunktionen p-ter Ordnung N_i^p können ausgehend von der Basisfunktion erster Ordnung N_i^1

$$N_i^1(u) = \begin{cases} 1 & u_i \le u < u_{i+1} \\ 0 & \text{sonst} \end{cases} \quad (0 \le i < n) \tag{2.11}$$

rekursiv mit

$$N_i^p(u) = \frac{u - u_i}{u_{i+p-1} - u_i}\, N_i^{p-1}(u) + \frac{u_{i+p} - u}{u_{i+p} - u_{i+1}} \tag{2.12}$$

berechnet werden. Alle Terme $(u_i - u)/(u_{i+p-1} - u_i)$ und $(u_{i+p} - u)/(u_{i+p} - u_{i+1})$ werden für $u_{i+p-1} = u_i$ bzw. $u_{i+p} = u_{i+1}$ mit 0 festgelegt. Somit gilt

$$\sum_{i=0}^{n-p} N_i^p(u) = 1, \ \forall u \in [u_{p-1}, u_{n-p+1}] \tag{2.13}$$

und

$$N_i^p(u) \begin{cases} > 0 & u \in (u_i, u_{i+p}) \\ = 0 & \text{sonst.} \end{cases} \tag{2.14}$$

Mit Hilfe des *de-Boor-Algorithmus* können B-Spline-Kurven beliebigen Grades ohne Kenntnis der Basisfunktionen erzeugt werden. Er ist das Äquivalent des Algorithmus von Casteljau für B-Spline-Kurven: Sei eine B-Spline-Kurve p-ten Grades durch die Kontrollpunkte $\mathbf{c}_0, \ldots, \mathbf{c}_n$ mit zugeordneten Parameterwerten $u_0, \ldots, u_n$, wobei $u_{\min} = u_0 < \ldots < u_n = u_{\max}$ ist, gegeben.

Setzt man $\mathbf{b}_i^{(0)} = \begin{cases} \mathbf{c}_i & i \in [0, n] \\ 0 & \text{sonst,} \end{cases}$ dann enthält $\mathbf{b}_\phi^{(p-1)}(u)$ nach $p-1$-maliger Anwendung der Rekursionsformel

$$\mathbf{b}_i^{(r)}(u) = \frac{u_{i+p-r} - u}{u_{i+p-r} - u_i}\, \mathbf{b}_{i-1}^{(r-1)}(u) + \frac{u - u_i}{u_{i+p-r} - u_i}\, \mathbf{b}_i^{(r-1)}(u) \tag{2.15}$$

den Punkt der B-Spline-Kurve zu u mit $u \in [u_\phi, u_{\phi+1}]$ und $0 \leq \phi < n$. Substituiert man

$$\alpha_i^r = \frac{u - u_i}{u_{i+p-r} - u_i} \tag{2.16}$$

erhält man die zum Algorithmus von Casteljau analoge Darstellung

$$\mathbf{b}_i^{(r)} = (1 - \alpha_i^r)\, \mathbf{b}_{i-1}^{(r-1)} + \alpha_i^r\, \mathbf{b}_i^{(r-1)}. \tag{2.17}$$

Mit $p = n + 1$, dem Knotenvektor $U = \{0, \underbrace{1, \ldots, 1}_{n}\}$ und $u_{-n} = \ldots = u_{-1} = u_{\min}$ ergibt sich $\alpha_i^r = u$. Bézier-Kurven sind somit nur ein Spezialfall von B-Spline-Kurven.

B-Splines besitzten viele Eigenschaften von Bézier-Kurven (vgl. Seite 7):

- Konvexe-Hüllen-Eigenschaft (sogar in strengerer Form)
- Invarianz bezüglich affiner Transformationen
- Endpunkt-Interpolation, falls die ersten bzw. letzten $p + 1$ Knoten den gleichen Wert besitzen
- Kontrollpunkt-Symmetrie
- Formerhaltungs-Eigenschaft
- Beschränkte Schwankung (variation diminishing)

Hinzu kommt die lokale Trägereigenschaft (vgl. (2.14)): Modifikationen an einem Kontrollpunkt bleiben auf p Nachbarn (bei einer B-Spline-Kurve p-ten Grades) begrenzt.

Allerdings sind B-Splines wie Bézier-Kurven *nicht* invariant bezüglich projektiven Abbildungen. Diesen Nachteil kann man durch NURBS beseitigen.

NURBS

NURBS ist die Kurzform für *Non Uniform Rational B-Splines*. Wie schon aus den Namen hervorgeht, werden hier im Gegensatz zu Bézier-Kurven und Splines rationale Funktionen zur Beschreibung verwendet. NURBS sind definiert durch

$$\mathbf{x}(u) = \frac{\sum_{i=0}^{n} N_i^p(u)\ w_i\ \mathbf{c}_i}{\sum_{i=0}^{n} N_i^p(u)\ w_i}. \tag{2.18}$$

Den Kontrollpunkten $\mathbf{c}_i$ werden noch zusätzlich Gewichte w_i zugeordnet, die überlicherweise als Formparameter verwendet werden. Je größer das Gewicht eines Kontrollpunkts ist, desto stärker ist sein Einfluss auf den Kurvenverlauf, indem sich die NURBS-Kurve stärker diesem Kontrollpunkt annähert.

Werden allerdings alle Gewichte um den gleichen Faktor verändert, ändert das nichts am Verlauf der NURBS-Kurve. Besitzen alle Kontrollpunkte das gleiche Gewicht, so ergibt das wieder die 'gewöhnliche' B-Spline. B-Splines (und somit auch Bézier-Kurven) sind also Spezialfälle von NURBS.

Eine NURBS-Kurve kann man sich als eine in den 3-dimesionalen Raum projezierte 4-dimensionale B-Spline-Kurve vorstellen. Wie sich zeigen lässt, sind NURBS-Kurven bezüglich projektiven Abbildungen invariant.

Bézier-Kurven, B-Splines und NURBS besitzen somit i.A. unterschiedliche Eigenschaften. Andererseits sind ihre Datenstrukturen unterschiedlich komplex. Es existieren Algorithmen zum Konvertieren geometrischer Modelle von NURBS-Darstellung in Bézier-Kurven. Die drei Darstellungsformen Bézier-Kurven, B-Splines und NURBS sind somit gleichmächtig. Je nach Aufgabenstellung kann es evtl. günstig sein, vor der Bearbeitung des Modells die eine in eine andere Darstellungsform umzuwandeln.

Flächenerzeugung aus Kurven

Im folgenden Abschnitt soll es um die Gewinnung von Flächen aus Kurven gehen. Analog zum *Verschiebegeometrieschema* auf Seite 14, wo aus einem 2D-Objekt, welches entlang einer Kurve verschoben wird, ein 3D-Objekt erzeugt wird, könnte hier aus einer Kurve (1D) eine Fläche (2D) erzeugt werden.

Coons Patch

Im einfachsten Fall lassen sich Vierecke $P_1P_2P_4P_3$ durch Verschieben der Strecke $\overline{P_1P_2}$ auf geradem Weg nach $\overline{P_3P_4}$ generieren. Ein Viereckspunkt $x_{s,t}$ wird dann durch die Gleichung

$$\mathbf{x}_{s,t}(s,t) = (1-s)(1-t)\ \mathbf{P}_1 + s(1-t)\ \mathbf{P}_2 + (1-s)t\ \mathbf{P}_3 + st\ \mathbf{P}_4 \tag{2.19}$$

beschrieben, wobei $s, t \in [0, 1]$ sind. Es stellt den linearen Fall des Tensorprodukts dar. Allgemein lässt sich die Gleichung

$$\mathbf{x}(s,t) = \mathbf{x}_s(s,t) + \mathbf{x}_t(s,t) - \mathbf{x}_{s,t}(s,t) \tag{2.20}$$

ermitteln. Hierbei ist

$$\mathbf{x}_s(s,t) = g_0(t)\gamma_0(s) + g_1(t)\gamma_1(s) \tag{2.21}$$

und

$$\mathbf{x}_t(s,t) = f_0(s)\delta_0(t) + f_1(s)\delta_1(t), \tag{2.22}$$

wobei $f_0(s) + f_1(s) = 1$ ($s \in [0,1]$) und $f_0(0) = f_1(1) = 1$ bzw. $g_0(t) + g_1(t) = 1$ ($t \in [0,1]$) und $g_0(0) = g_1(1) = 1$ ist. γ_0, γ_1, δ_0 und δ_1 bilden die Randkurven, f_0, f_1, g_0 und g_1 sind allgemeine Interpolationskurven. Der Anteil $\mathbf{x}_{st}$ ist sowohl in $\mathbf{x}_s$ als auch in $\mathbf{x}_t$ enthalten und muss deshalb von der Summe wieder abgezogen werden, um nicht fälschlicher Weise doppelt aufgerechnet zu werden.

Bézier-Flächen

Unter Nutzung von Bernstein-Polymen $\mathbf{b}_{i,k}$ ergibt sich mit (2.2)

$$\mathbf{x}(s,t) = \sum_{i=0}^{n} \sum_{k=0}^{m} B_i^n(s)\, B_k^m(t)\, \mathbf{b}_{i,k}, \quad s,t \in [0,1], \tag{2.23}$$

wobei die Bézier-Punkte $\mathbf{b}_{i,k}$ das zur Bézier-Fläche zugehörige Kontrollnetz aufspannen.

B-Spline-Flächen

Analog ergibt sich für B-Spline-Flächen mit (2.10)

$$\mathbf{x}(s,t) = \sum_{i=0}^{n} \sum_{k=0}^{m} N_i^p(s)\, N_k^q(t)\, \mathbf{c}_{i,j}. \tag{2.24}$$

Dabei sind $\mathbf{c}_{i,k}$ die Kontrollpunkte der Splinefläche, die das zugehörige Kontrollpunktnetz aufspannen und $N_{i,p}(s)$ bzw. $N_{k,q}(t)$ die Basisfunktionen. Die zur Flächenpunkt-Berechnung der verwendeten B-Splines genutzten Algorithmen finden sich im Abschnitt 4.3.

2.1.2 Volumenmodelle

Mit Hilfe von Volumenmodellen werden direkt die zu den beschreibenden Körpern gehörenden Volumina definiert. Prinzipiell sind hierfür zwei Wege möglich:

1. Der Raum wird diskretisiert in Zellen aufgeteilt und die einzelnen Zellen als zum Körper zugehörig oder nicht definiert oder

2. der Körper wird konstruktiv aus Primitiven aufgebaut. Die Primitive können eventuell parametrisierbar sein.

Modelle des Typ 1 werden auch raumpartitionierende Modelle genannt. Beispiele hierfür sind das Normzellen-Aufzählungsschema oder das Oktalbaumschema. Hauptvorteil dieser Schemata ist, dass die Lokalisation eines Punktes (Befindet sich der Punkt innerhalb oder außerhalb des Körpers?) direkt aus dem Modell ermittelt werden kann und nicht hierfür aufwändige Berechnungen erfolgen müssen. Dies ist gerade im Bereich der Simulation wichtig. Ein Vertreter des Typ 2 ist das CSG-Schema, welches im Bereich des CAD Anwendung findet.

Normzellen-Aufzählungsschema (NAZ)

Das zu betrachtende dreidimensionale endliche Teilgebiet wird vollständig in gleich große dreidimensionale Zellen, meistens Würfel aufgeteilt. Jede Zelle wird nun nach festgelegten Regeln als zum Körper gehörend oder nicht gehörend definiert. Für den Körperrand findet häufig folgende Regel Anwendung: Wird die Zelle von der Körperoberfläche geschnitten, so wird dem Körper zugeordnet, wenn der Zellmittelpunkt sich innerhalb des Körpers befindet, ansonsten nicht. Abb. 2.2 zeigt ein Normzellen-Aufzählungsschema im 2-Dimensionalen.

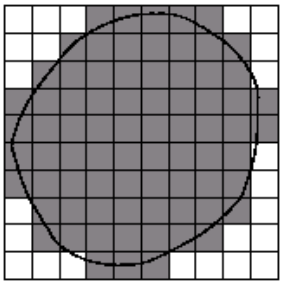


Abb. 2.2: NAZ

Für eine gegebene Zellgröße ist die Präsentation des Körpers durch Normzellen somit eindeutig. Je kleiner die Zellgröße desto genauer kann der Körper dargestellt werden. Als Datenstruktur wird im dreidimensionalen Fall am einfachsten ein dreidimensionales Feld verwendet. Eine Verdopplung der Auflösung in alle Raumrichtungen hat damit aber auch eine Verachtfachung des Speicheraufwands zur Folge. Da aus dem Schema nicht direkt ersichtlich ist, welche Zellen den Rand eines Objekts widerspiegeln und sich in einem solcen Fall die Größe aller Zellen sich ändert, müssen alle Zellen für eine verfeinerte Darstellung neu ermittelt werden. Modelle in Normzellen-Aufzählungsschemata werden häufig zur Aufbewahrung komprimiert.

Zellzerlegungsschema

Wird nicht wie beim Normzellen-Aufzählungsschema nur ein Primitiv verwendet, sondern können unterschiedliche Zellarten genutzt werden, spricht man allgemein vom Zellzerlegungsschema.

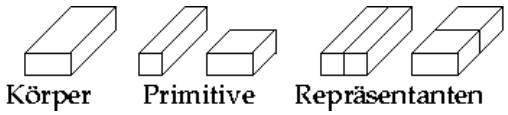


Abb. 2.3: Mehrdeutige Darstellung beim Zellzerlegungsschema

Neben Zellen mit glatter Oberfläche können auch Zellen mit gekrümmter Oberfläche

als Basisobjekte zum Einsatz kommen. Entsprechend dem Lego-Prinzip wird der Körper aus den Basisobjekten, den Primitiven, zusammengesetzt.

Wie Abb. 2.3 zeigt, muss die Präsentation des Körpers nicht eindeutig sein, da unterschiedliche Primitive und in unterschiedlicher Reihenfolge zur Konstruktion des Objekts verwendet werden können.

Constructive Solid Geometry

Bei der Constructive Solid Geometry - kurz CSG-Schema - wird der modellierte Körper als Konstruktion durch Nutzung von binären Mengenoperationen auf Basisobjekte dargestellt. Verwendbare Basisobjekte, auch Primitive genannt, sind z.B. Quader, Kugeln, Kegel und Zylinder, die zuvor definiert wurden.

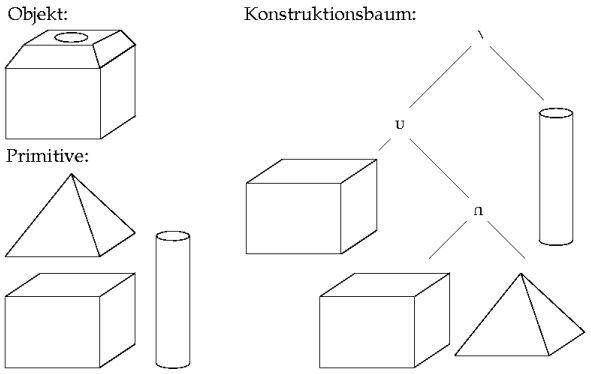


Abb. 2.4: CSG-Schema mit Konstruktionsbaum

Typische Mengenoperationen sind Schnitt, Vereinigung und Differenz. Als Erweiterung des Schemas können noch zusätzlich die Transformationen Rotation, Translation bzw. Skalierung als Operationen neben Mengenoperationen zugelassen werden.

Ein CSG-Schema kann durch einen Binärbaum dargestellt werden (vgl. Abb. 2.4). Dabei repräsentieren die Blätter die verwendeten Primitive, an den inneren Knoten werden die genutzten Operationen vermerkt. Von den Blättern ausgehend wird mit Hilfe der Operationen sukzessive der Körper konstruiert. Um eine eindeutige Darstellung des Körpers zu erhalten, muss der Konstruktionsbaum normalisiert werden. Als Normalform können die disjunktive oder konjunktive Normalform (DNF bzw. KNF) auftreten.

Verschiebegeometrieschema

Beim Verschiebegeometrieschema wird eine endliche Fläche (z.B. Polygon) entlang einer Verschiebekurve bewegt. Das überstrichene Gebiet wird als das Volumen des modellierten Körpers aufgefasst. Wird z.B. ein Rechteck einen Vektor entlang verschoben, dessen Richtung senkrecht zur Rechtecksfläche verläuft, entsteht ein Quader. Verschiebt man das Rechteck entlang einer Kurve, die eine Kreislinie darstellt, wird ein (Hohl)Zylinder erzeugt (vgl. Abb. 2.5).

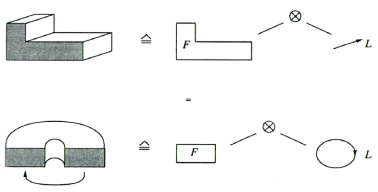


Abb. 2.5: Verschiebegeometrieschema

Primitiven-Instanzierung

Bei der Primitiven-Instanzierung werden alle Körper aus einem parametrisierbaren Grundobjekt erzeugt. Das Grundobjekt kann dabei durchaus komplex sein. Ist das Grundprimiv z.B. eine Metallschraube mit den Parametern Steigung, Länge und Kopfgröße, können verschiedene Metallschraubenarten beschrieben werden. In Abb. 2.6 ist ein 6-etagiges Gebäude mit 6 Fensterspalten, das aus dem Primitiv 'Blockhaus' erzeugt wurde.

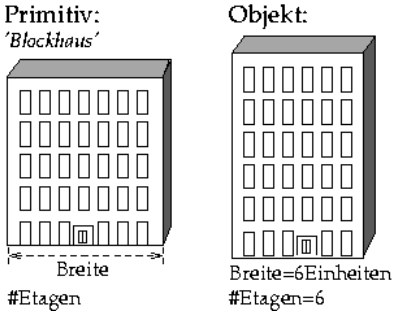


Abb. 2.6: Primitiven-Instanzierung

Oktalbäume

Beim Oktalbaumschema handelt es sich um ein hierachisch, rekursives Schema, bei dem der zu diskretisierende Raum in Zellen unterteilt wird. Es wird von einem, den gesamten Körper umschließenden Würfel (boundary cube) ausgegangen.
Beginnend mit diesem Würfel erfolgt eine sukzessive Zerlegung bis der betrachtete Würfel sich vollständig innerhalb oder außerhalb des Körpers befindet oder die zuvor festgelegte Auflösung den Würfelabmessungen entspricht (dies ergibt sich direkt aus der maximalen Anzahl der Zerlegungsschritte, welche auch der maximalen Baumtiefe entspricht). Bei jedem Zerlegungsschritt wird der Würfel in jede Achsrichtung halbiert, woraus sich 8 Unterwürfel (Oktanten) ergeben.

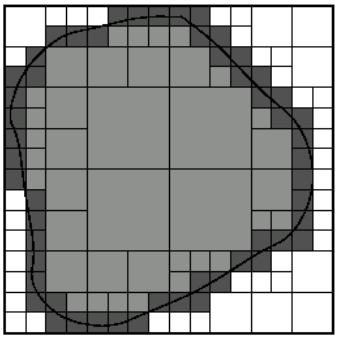


Abb. 2.7: Raumpartionierung durch Quadtree

Die Blätter werden als im Körper befindlich oder nicht zum Körper gehörend markiert. Für Blätter, die Zellen repräsentieren, die den Körper teilweise enthalten, kann eine spezielle Markierung definiert werden. Alternativ werden solche Zellen je nach Anwendungsszenario entsprechend dem Normzellen-Aufzählungsschema einfach zum Körper mitgerechnet oder als außerhalb des Körpers liegend aufgefasst. Abb. 2.7 zeigt einen Quadtree, das 2-dimensionale Pendant des Oktalbaums.

Jedes Normzellen-Aufzählungsschema mit Würfeln als Primitive lässt sich leicht in das entsprechende Oktalbaumschema überführen und umgekehrt. Da nur die Randbereiche des Körpers beim Oktalbaumschema maximal aufgelöst werden müssen, ergibt sich im Allgemeinen ein signifikant geringerer Speicheraufwand als beim Normzellen-Aufzählungsschema. Eine weitere Folge dieser Eigenschaft ist, dass bei jeder Verdopplung der Auflösungsgenauigkeit in jede Achsrichtung nur mit einer Vervierfachung

des Speicheraufwands gerechnet werden muss[2]. Zur persistenten Datenhaltung empfiehlt sich einer Linearisierung des Oktalbaums. Weitergehende Betrachtungen zu Eigenschaften von Oktalbäumen finden sich im Abschnitt 2.2. Auf Oktalbäumen arbeitende Algorithmen sind im Kapitel 4 dargestellt.

2.2 Der Oktalbaum als Schnittstelle zwischen CAD und Simulation

Bei Simulationen besitzt die Lokalisation innerhalb eines geometrischen Modells – also die Bestimmung, wo sich ein Punkt bzw. eine Zelle bezüglich eines Körpers befindet – einen hohen Stellenwert. Sie muss deshalb durch das Modell effizient erfolgen können. Aus diesem Grund sind für Simulationsaufgaben nur direkte Volumenmodelle vorteilhaft. Dabei sollte die bei Simulationen auftretenden Raumdiskretisierungen durch das Modell unterstützt werden. Dem Oktalbaumschema kommt dabei eine hohe Bedeutung zu, da es eine Reihe von Vorteilen gegenüber anderen Modellen hat. Das fällt besonders gegenüber dem ebenfalls in diesem Zusammenhang bedeutenden Normzellen-Aufzählungsschema auf. Ursache hierfür sind vor allem die im Oktalbaumschema vorkommenden Organisationsprinzipien.

Während beim Normzellen-Aufzählungsschema über das gesamte Gebiet die Zellen die gleiche (minimale) Größe besitzen, wird beim Oktalbaumschema nur der Rand des Körpers maximal aufgelöst. Verdoppelt man die Auflösung in jede Raumrichtung, ist nicht – wie beim Normzellen-Aufzählungsschema – mit einer achtfach, sondern nur mit einer vierfach höheren Gesamtzellzahl zu rechnen. Da Zeit- und Speicheraufwand von der Gesamtzellanzahl abhängen, gelten für sie equivalente Komplexitätsabschätzungen. Ist bereits ein Oktalbaum geringerer Auflösung vorhanden, müssen nur die Randknoten (ausgehend von der gröberen Auflösung) verfeinert und nicht der ganze Oktalbaum neu erzeugt werden, was weitere Geschwindigkeitsvorteile bringt. Auf Oktalbäumen lassen sich Normzellen-Aufzählungsschemata 'emulieren', womit der Oktalbaum prinzipiell auch überall dort eingesetzt werden kann, wo ein Normzellen-Aufzählungsschema verwendet wird. Ein weiteres Indiz für die Vorteile der Verwendung des Oktalbaums als Schnittstelle ist seine Verwendung in unterschiedlichsten Anwendundungsgebieten, da dies für seine flexible Nutzbarkeit spricht.

2.2.1 Anwendungsgebiete für Oktalbäume

Neben der Simulation kommen Oktalbäume in anderen wichtigen Gebieten zum Einsatz. Hierzu nennt [Fra00, S. 26]

[2] Diese Aussage ist nicht auf Körper mit fraktaler Struktur anwendbar. Solche Strukturen sind jedoch innerhalb des CAD eher untypisch.

- die Bildverarbeitung: zur Erzeugung eines 3D-Modells aus 2D-Daten (z.B. CT-Daten), zur effizienten Bilddatenspeicherung und -reduktion, zum Einfärben von Flächen oder zur Bild- oder Objekterkennung,
- die Computergrafik: zur Sichtbarkeitsentscheidung oder Rückseitenerkennung, sowie in den Bereichen Raytracing, Radiosity oder Animation,
- die Visualisierung: zur Isoflächengenerierung beim Volume Rendering und
- die Robotik: zur Interferenzdetektion oder Bahnplanung.

Wird die Oktalbaumstruktur zur Gittergenerierung für Simulationsaufgaben verwendet, dient sie meist als nützliches Hilfskonstrukt, die vor der eigentlichen Simulation nachbearbeitet werden muss.

2.2.2 Datenstrukturen für Oktalbäume

Aus der Definition von Oktalbäumen (vgl. Abschnitt *Oktalbäume* auf Seite 15) ergibt sich die Verwendung von hierachisch rekursiven Datenstrukturen für Oktalbäume. Folgende Beziehungen sind dabei wichtig (unter '▷' sind mögliche Operationen[3] dargestellt, mit denen der Status dieser Beziehungen ermittelt werden kann. Diese Operationen, sowie analoge Operationen zur Modifikation müssen die entsprechenden Datenstrukturen bereitstellen):

- Durch Knoten werden begrenzte Volumina wiedergegeben.
- Es gibt einen ausgezeichneten Knoten, die *Wurzel*. Er repräsentiert den *boundary cube* – also ein die gesamte modellierte Geometrie einhüllender Würfel. Die Wurzel ist Einstiegsknoten für den Oktalbaum, über sie können alle anderen Knoten erreicht werden. Jeder Knoten kann über genau einen Weg von der Wurzel aus erreicht werden.

  ▷ **`function`** getRoot**`()`** **`return`** `_octree` liefert die Wurzel.
- Über die Verfeinerungsaktion bei der Oktalbaumgenerierung entstehen Unteroktanten zu einem bestimmten Volumen, woraus sich eine Hierachie ergibt. Die Wurzel stellt die höchste Hierachieebene dar. Die Knoten, die die Untervolumina repräsentieren, die durch eine Verfeinerungsaktion hervorgegangen sind, werden als Sohnknoten zu einem Vaterknoten bezeichnet. Die Wurzel ist der einzigste Knoten ohne Vaterknoten, alle anderen Knoten besitzen genau einen Vaterknoten.
- Entsprechend der Verfeinerungstiefe ergibt sich somit eine Knotentiefe d für einem Knoten innerhalb des Baums. Der Wurzel wird die Tiefe 0 zugewiesen. Sohnknoten c haben eine um 1 größere Tiefe als ihr Vaterknoten p: $d_c = d_p + 1$.

[3] Dies sind die Methoden, die in dieser Arbeit hierfür werden.

Als Tiefe d_t des Baums t ergibt sich die maximale Tiefe eines Knoten n des Baums: $d_t = \max_{n \in t}\{d_n\}$.

Es kann eine maximale Verfeinerungstiefe definert definiert werden. Damit ergibt sich auch eine *maximale Baumtiefe* $d_{t_{\max}}$.

Manchmal ist es günstiger die *Knotenhöhe* h_n anstatt seiner Tiefe festzulegen. Ein Oktalbaumknoten auf der untersten Verfeinerungstiefe ($d_n = d_{t_{\max}}$) hat die Höhe $h_n = 0$. Er muss immer ein Blatt sein. Der Wurzelknoten besitzt die Höhe $d_{t_{\max}}$. Sohnknoten c haben eine um 1 niedrigere Höhe als ihr Vaterknoten p: $h_c = h_p - 1$. Zwischen Knotentiefe und Knotenhöhe besteht folgender Zusammenhang: $h_n = d_{t_{\max}} - d_n$.

▷ **function** getMaxTreeDepth() **return** Depth liefert die maximale Baumtiefe $d_{t_{\max}}$.

Des Weiteren besteht zwischen Knotentiefe d_n und der Länge des von ihm repräsentierten Würfels l_n folgender Zumsammenhang:

$$l_n = \frac{l_0}{2^{d_n}}. \tag{2.25}$$

l_0 ist die Länge der boundary cube bzw. des modellierten Gesamtvolumens, was durch den Oktalbaum repräsentiert wird. Dieser Zusammenhang wird noch später zur Indizierung von Oktalbaumknoten und zur Zuordnung von Punkten zu einer Zelle des Oktalbaums verwendet.

- Ist ein Knoten Vaterknoten, besitzt er also Sohnknoten, wird er als *innerer Knoten* bezeichnet. Aus der Definition des Oktalbaums ergibt sich, dass jeder innere Knoten genau acht Söhne besitzt. Dabei sind die Oktanten (und damit die acht Söhne) in einer bestimmten Reihenfolge festgelegt. Die restlichen Baumknoten sind *Blätter*, die demzufolge keine Söhne besitzen. Blätter von Oktalbäumen besitzen ein wichtiges Attribut, welches definiert, ob der dadurch repräsentierte Raumteil zu einem Körper gehörig (in) oder nicht (out) oder auf dem Rand des Körpers befindlich (on) zu zählen ist. In dieser Arbeit wurde jedoch dieses Attribut auf Grund der folgenden Anforderungen etwas abweichend definiert:
  - Der Rand soll immer zu dem Körper gehörig gezählt werden.
  - Mehrere Körper können durch einen Oktalbaum modelliert werden.

  Das Attribut speichert hier eine Farbe. Jedem Körper wird eine Farbe eineindeutig zugeordnet. Eine weitere Farbe ist als *zu keinem Körper gehörend* defniert. Alle Blätter die Raumteile repräsentieren, die zu keinem Körper gehören, werden mit dieser speziellen Farbe markiert.

  ▷ **function** isLeaf(*Node node*) **return** boolean liefert true falls *node* ein Blatt ist und für innere Knoten false.

▷ *{ Vorbedingung: isLeaf(*`node`*) = false,* `i` $\in [0, 7]$ *}*
`function` getChild(*`_octree parent, Parttype i`*)
`return` `_octree` liefert den *`i`*-ten Sohn zum Vaterknoten *`parent`*.

▷ *{ Vorbedingung: isLeaf(*`node`*) = true }*
`function` getColor(*`Node node`*) **`return`** `Color` liefert die Farbe des Blatts.

▷ *{ Vorbedingung: isLeaf(*`node`*) = true }*
`function` isNoObject(*`Node node`*) **`return`** boolean liefert true falls getColor(*`node`*) = *'zu keinem Körper gehörend'* und sonst false.

Hiermit sind wesentliche Beziehungen, die für Oktalbäume gelten, definiert. Bäume im Allgemeinen besitzen eine Reihe von weiteren interessanten Eigenschaften, welche u.a. in [Sed94] beschrieben sind.

Im Weiteren werden vier Datenstrukturen vorgestellt, die für Oktalbaumstrukturen eingesetzt werden. Besondere Bedeutung innerhalb dieser Arbeit haben das *baumartige Zeigergeflecht* und die *Präorder-Traversierung*. Das baumartige Zeigergeflecht ist Basis für die Oktalbaumstruktur, die zur Generierung des Oktalbaummodells benutzt wurde (vgl. Kapitel 4). Hierauf wurde ein Index analog zur *Positionscodierung* definert. Zum permanenten Datenhaltung wurde das Zeigergeflecht mit Hilfe der Präorder-Traversierung linearisiert (vgl. Abschnitt 3.2).

Baumartiges Zeigergeflecht

Das Zeigerflecht wird häufig zur Veranschaulichung von Bäumen verwendet und ist daher die naheliegendste Datenstruktur. Sie besitzt folgende wesentliche Eigenschaften:

- Auf jedes Blatt kann in $\mathcal{O}(d)$ Zeit zugegriffen werden. Dabei ist d die Blatttiefe. Oktalbäume besitzen in typischen Anwendungen eine Tiefe von 6 bis 15.
- Da es sich bei einem Zeigergeflecht um eine dynamische Struktur handelt, ist sie leicht anpassbar. Modifikationen können effizient durchgeführt werden. So wird z.B. für das Einfügen oder Löschen von Blättern $\mathcal{O}(d)$ Zeit benötigt. Es gibt keine strukturbedingten Größenbeschränkungen, wie z.B. bei statischen Feldern. Auch können Hilfszeiger eingefügt werden. So kann z.B. der Nachbar eines Blatts in $\mathcal{O}(1)$ gefunden werden, wenn entsprechende Referenzen zuvor im Oktalbaum gesetzt wurden. Durch das Einfügen von zusätzlichen Zeigern ergeben sich natürlich auch Nachteile, wie z.B. zusätzlicher Aufwand für die Konsistenzhaltung das Oktalbaums.
- Je nach Speicherarchitektur werden zusätzlich für jeden Zeiger 1 bis 2 Wörter zum Speichern dieses Zeigers benötigt. Das ist meist sogar mehr, als an Speicherplatz für die eigentliche Information (Blatt oder kein Blatt, Blattfarbe) verbraucht

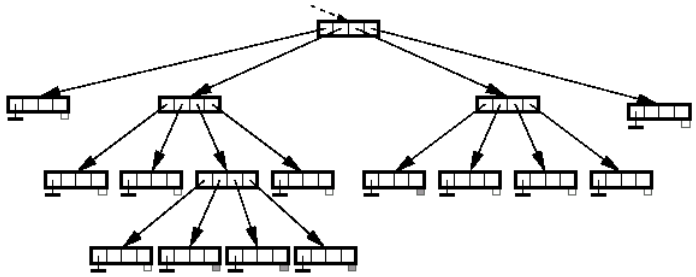


Abbildung 2.8: Zeigergeflecht eines Quadtree

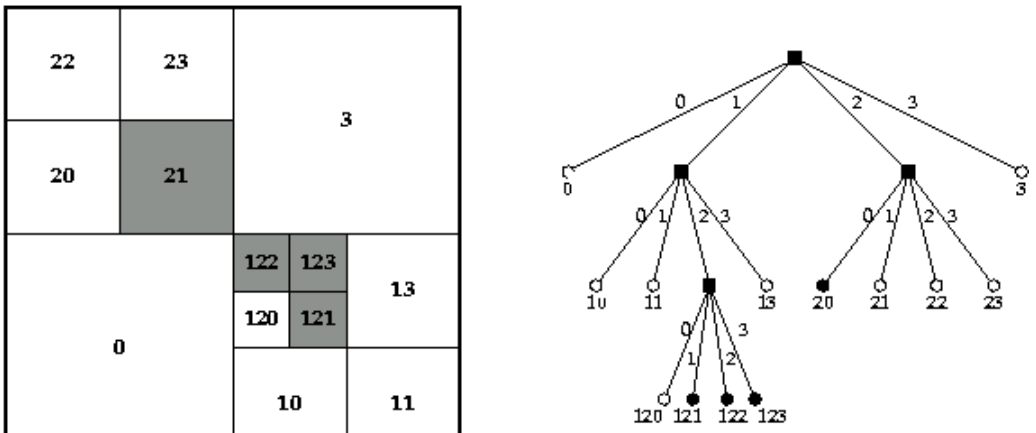
wird. Zeigergeflechte verbrauchen deshalb gerade bei größeren Strukturen mehr Speicher als kompakte Datenstrukturen.

- Allgemein bedingt die Benutzung von Zeigern insbesondere bei starken Modifikationen (ständiges Einfügen und Löschen von Knoten) einige technische Probleme. Zum Beispiel
  - können mehrere Zeiger gleiche Knoten referenzieren (im Oktalbaum: wenn er durch zusätzliche Referenzen beispielsweise auf die Nachbarknoten erweitert wird). Wird der entsprechende Speicherbereich über den einen Zeiger freigegeben, enthält der andere Zeiger eine falsche Referenz.
  - muss das Problem des *Carbage Collection* beachtet werden. Es muss eine Instanz geben, die sich um nicht mehr verwendete Speicherbereiche kümmert. Es können Segmentierungen des Haldenspeichers auftreten. Es ist zu beachten, dass Speicherbereiche, die nicht mehr verwendet und dennoch nicht freigegeben werden –, wie z.B. bei Ringlisten, auftreten können. Der Cabage Collector kann zeitliche Abläufe des Programms beeinflussen und die Laufzeiteigenschaften des Programms verschlechtern.

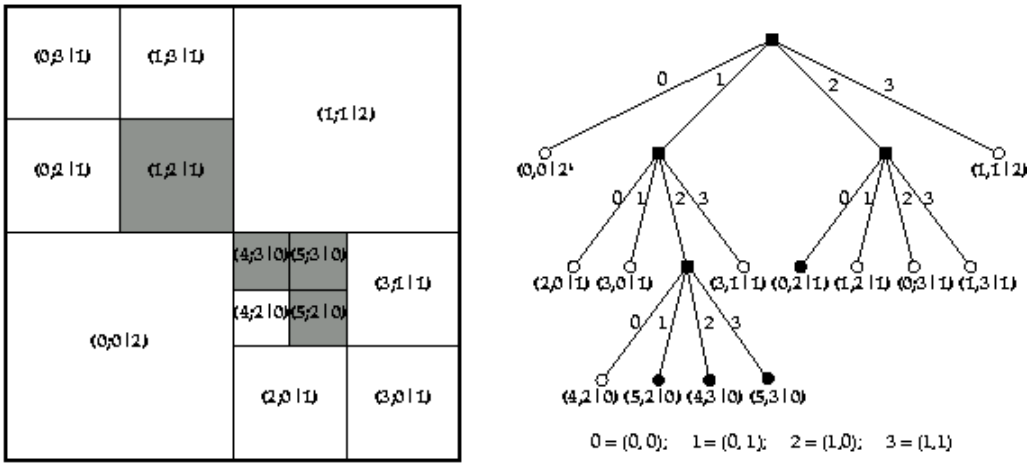
Abbildung 2.8 zeigt das Zeigergeflecht zum Quadtree in Abbildung 2.9.

Positionscodierung

Bei der Positionscodierung erhält jeder Knoten einen eindeutigen Index. Für die Sohnknoten muss eine bestimmte Reihenfolge definiert werden, woraus sich ein entsprechendes Nummerierungsschema ableitet. Häufig wird die Reihenfolge an Hand der Lebesgue-Kurve festgelegt. Der Index eines Knotens ist dann der Vektor, der sich aus Nummern der Knoten ergibt, die auf dem Pfad von der Wurzel zu ihm besucht wurden. Besonders anschaulich ist die Verwendung von Oktalzahlen für den Index. Jede Ziffer gibt die Nummer des entsprechenden Sohnknotens wieder. Die Anzahl der Ziffern liefert die Tiefe des indizierten Knotens. Hiermit kann auch die Position der Zelle des Oktalbaums bestimmt werden, die dem Knoten zugeordnet ist. Hierher rührt auch



a) üblich



b) in dieser Arbeit verwendet

Abbildung 2.9: Positionscodierung

der Name Positionscodierung für diese Repräsentationsform. Abbildung 2.9a) zeigt die Positionscodierung nach diesem Verfahren im 2-Dimensionalen zu einem Quadtree.

Gegenüber dem Zeigergeflecht ist die Identifikation von Knoten einfacher, woraus eine Vereinfachung verschiedener Algorithmen resultiert.

In dieser Arbeit wird eine leicht abgewandelte Form benutzt, die besonders intuitiv ist und auf der das Normzellenschema leicht simuliert werden kann. Es wird jede Raumrichtung gesondert betrachtet, wodurch ein 3-dimensionaler Vektor aus Binärzahlen entsteht. Somit wird für jede Raumrichtung bestimmt, ob der Sohn im jeweils vorderen oder hinteren Teilraum liegt. Zusätzlich wird noch die Knotenhöhe im Index vermerkt (vgl. Abbildung 2.9b).

Die Positionscodierung wird in der Literatur auch als *linearer Code* oder *leafcode* bezeichnet.

Codierung über Raum-Position und Größe

Statt über den Pfad durch den Baum lässt sich ein Knoten auch über seine Position P im Raum und seine Größe G identifizieren. Die Position kann dabei über die Koordinaten eines bestimmten Zellpunkts, über die Anzahl der Maschenweiten eines virtuellen Gitters feinster Auflösung oder durch eine Binäradresse ähnlich der Positionscodierung dargestellt werden. Die Größe wird entweder durch die reale Größe oder ein Vielfaches der Maschenweite angegeben. Nachteilig an dieser Repräsentationsform ist die fehlende Unterstützung des Organisationsprinzips Hierachie.

Der in Abbildung 2.9 dargestellte Quadtree könnte durch folgenden Code $C = (P; G)$ dargestellt werden:

Blätter:	(0, 0; 4), (0, 4; 2), (0, 6; 2), (2, 4; 2), (2, 6; 2), (4, 0; 2), (4, 2; 1), (4, 3; 1), (4, 4; 4), (5, 2; 1), (5, 3; 1), (6, 0; 2), (6, 2; 2)
inner Knoten:	(0, 0; 8), (0, 4; 4), (4, 0; 4), (4, 2; 2)

Präorder-Traversierung

Es gibt unterschiedliche Formen, um alle Knoten eines Baums abzulaufen, also zu traversieren. Durch Tiefensuche über den Baum entsteht, je nachdem ob dabei der Vaterknoten vor, in der Mitte oder nach den Söhnen notiert wird, *Präorder-*, *Inorder-* oder *Postorder-Traversierung*. Wird Breitensuche angewendet, also alle Knoten einer Tiefe besucht, bevor ihre Söhne besucht werden, entsteht eine *Levelorder-Traversierung*.

Für Oktalbäume wird dabei häufig die Präorder-Traversierung verwendet. Da innere Knoten stets acht Söhne besitzen, ist eine besonders kompakte Darstellung möglich: Beispielsweise können innere Knoten mit `1`, Blätter mit `0` markiert werden. Das zusätzliche Farbattribut wird einfach hinter die blattmarkierende `0` hinzugefügt. Damit liegt der Oktalbaum sehr kompakt in einer linearisierten Form vor und kann somit als binärer Strom (binary Stream) verarbeitet werden.

Mit dieser Codierung ergibt sich für den Quadtree aus Abbildung 2.9:

```
1001000 01000101 01001010 0000000
```

Die Präorder-Traversierung wird auch Depth-First-Strategie genannt. In der Literatur kommen auch die Namen *DF-expression* oder *treecode* für diese Form der Codierung vor.

2.2.3 Operationen auf Oktalbäumen

Neben den im Abschnitt 2.2.2 beschriebenen grundlegenden Operationen, gibt es weitere Grundoperationen auf einem Oktalbaum. Hierzu gehören das Erzeugen, Löschen oder die Aufspaltung eines Blatts, sowie die Vereinigung gleichgefärbter Blätter zu einem Blatt der darüberliegenden Ebene (Kompaktieren). Diese Operationen sind auf den hier verwendeten Zeigergeflecht als Basisstruktur leicht effizient zu realisieren.

Raumpartitionierende Strukturen – und somit der Oktalbaum – sind geradezu prädestiniert für die binären Mengenoperationen Schnitt, Vereinigung und Differenz: Die beiden Bäume werden synchron (z.B. durch Präorder-Traversierung) durchlaufen und an den Blättern entsprechende Vergleiche durchgeführt, um so den resultierenden Baum zu konstruieren. In der Literatur finden sich auch effiziente Algorithmen zur Realisierung der affinen Transformationen Translation, Rotation und Skalierung.

Des Weiteren ist die Bestimmung der Nachbarzelle zu einem Oktalbaumknoten von Bedeutung. Die Nachbarzelle ist jedoch nicht direkt aus der Oktalbaumstruktur ersichtlich, insbesondere wenn diese durch ein Zeigergeflecht repräsentiert wird. Hierfür wird die Positionscodierung verwendet. Mit dem in dieser Arbeit verwendeten Index (vgl. Abschnitt *Positionscodierung* auf Seite 20) kann leicht der entsprechende 'theoretische' Nachbar auf gleicher Ebene ermittelt werden. Der so ermittelte Nachbar kann jedoch unterhalb des 'realen' Nachbars liegen, wenn der entsprechende Baumast nicht so tief ist wie der Ausgangsknoten. Mit Hilfe von `function getExistNode(NodeIndex p) return NodeIndex` (Algorithmus 4.3) wird deshalb der Index des tiefsten existierenden Knotens ermittelt, der sich auf dem Pfad zu p befindet. Der so gefundene Knoten muss jedoch nicht ein Blatt sein. Diese können durch rekursiven Abstieg zu den entsprechenden Söhnen (Gegenrichtung als die Richtung zum Nachbarn) gefunden werden.

Es finden sich Algorithmen, die einen balancierten Oktalbaum voraussetzen. *Balanciert* bedeutet, dass sich die Maschenweite zweier benachbarten Zellen um maximal den Faktor 2 unterscheiden (also die Tiefe von Nachbarknoten maximal um ± 1 differiert). Da sich Aufspaltungen oder Vereinigungen innerhalb einer Zelle bei der Balancierung höchstens auf deren Nachbarzellen auswirken, haben Änderungen im Baum nur lokale Wirkung. Somit lässt sich die Balancierung eines Baums effizient umsetzen.

Zur Aufzählung räumlich angeordneter diskreter Daten – entsprechend der Lebesgue-Kurve – kann der *Morton-Index* verwendet werden. Als herausragende Eigenschaft ist zu nennen, dass Nachbarschaftsbeziehungen in einer so erzeugenden Sequenz bereits im ursprünglichen Baum vorhanden waren. Der Morton-Index kann so effizient zur Nachbarschaftsbestimmung von Knoten im Oktalbaum genutzt werden und eignet sich zur dimensionsunabhängigen Sequentialisierung von Quad-/Octree-Zeigerstrukturen. Ferner kann der Morton-Index zur Oktalbaumgenerierung aus einem Normzellen-Aufzählungsschema verwendet werden.

Kapitel 3

Unterstützte Formate

Wie bereits diskutiert, eignen sich Oktalbäume zur Integration von Modellierung und Simulation. Durch die im Rahmen dieser Arbeit entstandene Implementierung sollen Oktalbäume unter Zuhilfenahme eines Oberflächenmodells erzeugt werden können.

Es musste ein *geeignetes* Format für das Oberflächenmodell festgelegt werden, aus welchem der Oktalbaum generiert wird. Es wird im Folgenden als Importformat bezeichnet. Abschnitt 3.1 beschäftigt sich mit Auswahl und Eigenschaften des gewählten Importformats.

Der erzeugte Oktalbaum wird abschließend in eine Datei geschrieben. Das hierfür verwendeteFormat wird als Exportformat bezeichnet. Abschnitt 3.2 erläutertdas unterstützte Exportformat *POT*.

3.1 Import

Um ein geeignetes Importformat finden zu können, müssen zunächst Kriterien festgelegt werden, die dieses Format erfüllen muss. Die Auswahlkriterien sind Bestandteil des Abschnitts 3.1.1. Im zweiten Schritt erfolgt die Auswahl eines Formats nach diesen Kriterien.

Zu dem gewählten DXF-Format existieren frei zugängliche Bibliotheken, die den Umgang mit dem Format erleichtern. Abschnitt 3.1.2 bewertet drei DXF-Bibliotheken hinsichtlich ihrer Anwendbarkeit zur Umsetzung der Aufgabenstellung.

3.1.1 Anforderungen an Importformate

Die Anforderungen ergeben sich unmittelbar aus der Aufgabenstellung für die gesamte Arbeit und sind:

Unterstützung von Dreiecksnetzen und Freiformflächen
Das Format muss Strukturen bereitstellen, die es ermöglichen, sowohl Körper mit einfacher polygonaler Oberfläche (mindestens Drei- oder Vierecke) als auch Freiformflächen in Form von Spline-Flächen zu generieren. Es sollte auch eine gemischte Darstellung (eine Datei enthält z.B. zwei Körper, von denen der eine mit triangulierter Oberfläche und der andere durch Splineflächen modelliert ist) möglich sein.

Erzeugbarkeit durch in der Abteilung vorhandene Applikationen
Die in der Abteilung SgS vorhandenen regulären Flächenmodelle[1], die in einem üblichen Format vorliegen, sollen durch eine Exportfunktion, die sich bereits in einem der in der Abteilung genutzten Modellierungswerkzeuge befindet, einfach in das Importformat überführt werden können, falls das Oberflächenmodell nicht bereits in diesem Format vorliegt. Alternativ gibt es zu diesem Zweck ein frei verfügbares Konvertierungswerkzeug. Das so erzeugte Modell stellt (natürlich) den ursprünglich modellierten Körper in einer regulären Form da. Ein Modell soll auch mehrere Körper enthalten können.

Hohe Verbreitung
Das Format ist am Markt verbreitet. Es gibt eine Vielzahl von Werkzeugen, die das Format verwenden. Es finden sich Verweise auf das Format in der einschlägigen Literatur. Es gibt Anwendungen zum Erstellen und Bearbeiten von geometrischen Modellen, die in diesem Format vorliegen.

Freie Zugänglichkeit der Formatspezifikation
Die Formatspezifikation ist frei einsehbar. Es finden sich Formatbeschreibungen im Internet und in einschlägiger Literatur. Es handelt sich also um ein standardisiertes Format oder um einen offenen Pseudostandard.

Einfaches Handling
Idealerweise existieren frei nutzbare Bibliotheken für das Format. Alternativ ist das Format einfach und leicht verständlich strukturiert. Dies ist auch bei vorhandenen Bibliotheken für Verifikationen wünschenswert. Hierzu gehören auch frei verfügbare Betrachter für dieses Format. ASCII-Formate sind Binärformaten wegen der leichteren Verifizierbarkeit vorzuziehen.

Die Formate DXF, IGES und STEP erfüllen diese Kriterien. IGES und STEP werden von zahlreichen kommerziellen Systemen unterstützt. Für DXF finden sich hingegen auch viele frei verfügbare Quellen, weshalb die Wahl auf DXF als Importformat fiel. Tabelle 3.1 stellt die Formate DXF, IGES und STEP gegenüber.

DXF besitzt einen relativ einfachen Aufbau. DXF ist ein ASCII-Format. Es gibt auch eine Binärversion DXB. Die Geometrieinformationen werden in der Sektion *Entity* abgespeichert. Für unterschiedliche geometrische Formen gibt es unterschiedliche Entities. Für diese Arbeit sind die Entities 3DFACE für planare Oberflächen (Drei- oder Vierecke),

[1] Flächenmodelle von starren Körpern im Sinne der geometrischen Modellierung

Format	DXF	IGES	STEP
Version	2002	5.3	214
Quelle	[aut02]	[Mat00a]	[Mat00b]
Standard	offener Psyeudostandard	ANS US	AP

Tabelle 3.1: Formate DXF, IGES und STEP

POLYLINE (gekrümmte Kurve oder Fläche, in diesem Fall kubische B-Spline-Fläche) und VERTEX (Einzelner Punkt der Spline-Fläche) wichtig. Einzelne Informationen finden sich hinter dem entsprechenden *Group Code* wieder. So steht beispielsweise hinter dem Group Code 10 der 3DFACE-Entity die x-Koordinate des ersten Eckpunkts. Die Orientierung von Flächen ergibt sich aus der Reihenfolge der Eck-/Kontrollpunkte (vgl. mit (2.1) auf Seite 5). Die Entities werden einfach hintereinander gespeichert.

3.1.2 DXF-Bibliotheken

Im Folgenden wurden einige DXF-Bibliotheken evaluiert. Gesucht wurde eine Bibliothek zum Einlesen der im DXF-Format vorliegenden Geometriedaten.

Folgende Kriterien wurden zur Bewertung der Importbibliotheken angewendetet:

Lizenz
: Die Bibliothek muss frei verfügbar sein. Die hier betrachteten Bibliotheken stehen unter (L)GPL und sind somit frei nutzbar.

Installierbarkeit
: Mit welchem Aufwand kann die Bibliothek zum Laufen gebracht werden?

Funktionalität
: Hierbei wurde getestet, inwieweit die Bibliotheken mit Geometriemodellen von Körpern mit triangulierten bzw. Splineoberflächen zurechtkamen. Hierfür konnten zum Teil beiliegende Beispielprogramme verwendet werden. In anderen Fällen wurde die eingeschränkte Benutzbarkeit aus der Dokumentation ersichtlich. Ein weiteres Augenmerk lag auf dem Einarbeitungsaufwand für das Verständnis der relevanten Teile und der Komplexität der Einbindung. Die Bibliothek muss in C++-Programme integrierbar sein.

Dokumentation
: Mit welchen Mitteln wird die Bibliothek dokumentiert? Welchen Umfang hat die Dokumentation? Wie verständlich ist sie?

Aktivität
: Inwieweit wird die Bibliothek weiterentwickelt und gepflegt? Wann fand die letzte Aktualisierung statt?

Library	**dime**	**dxflib**	**libdxf**
Version	0.9.1	0.1.2	0.7
Quelle	[Coi02]	[Mus02]	[Bar97]
Lizenz	GPL	LGPL	GPL
Paket-Format	tar.gz	tar.gz	tar.gz
Installierbarkeit	◯	◯	◯
Funktionalität	+	−	◯
Dokumentation	◯	−	−
Aktivität	+	+	−
Verbreitung	+	◯	−
Eignung	+	−	−

Tabelle 3.2: DXF-Bibliotheken dime, dxflib und libdxf

Tabelle 3.2 zeigt die Bewertung für dime, dxflib und libdxf. Laut Dokumentation lassen sich alle 3 Bibliotheken sowohl auf Unix-Derivaten wie auf allen Windows-Versionen einsetzen: Zur Installation werden lediglich neben einem `tar.gz`-Entpacker, ein `gcc`-kompatibler Compiler und eine funktionsfähige `make`-Umgebung benötigt. Die Bibliotheken wurden ausschließlich auf Linux (SuSE Linux 8.0) getestet.

Ein Vielzahl von Links verweisen auf die **dime**-Bibliothek. Sie lässt sich u.a. als Debian-Paket auf den Debian-Seiten finden. Die hier eingesetzte Version stammt von Coin3d, dem Hersteller von dime. Zuerst muss das `tar.gz`-Archiv entpackt werden. Anschließend kann mit Hilfe des `configure`-Werkzeugs und `make` dime relativ einfach installiert werden. Neben dem Source-Code ist eine doxygen[2]-generierte Klassenreferenz und ein VRML-Konverter als Beispielprogramm enthalten. Die Klassen- und Methodenbeschreibungen sind knapp gehalten, manchmal zu knapp. So fehlt zu einigen Methoden die Beschreibung gänzlich, für andere ist sie wenig aussagekräftig. Dennoch ist die Dokumentation wie auch das Beispielprogramm sehr hilfreich. Manchmal kann auch ein Blick in den gut strukturierten Code Klarheit bringen. Man arbeitet sich relativ schnell in die Bibliothek ein. Alle notwendigen Geometriemodelle können mit dime extrahiert werden. Dime wird seit 1999 entwickelt und weiterhin gepflegt. Die aktuelle Version enthält den Stand vom Oktober 2002. Dime ist als Importbibliothek für die beschrieben Zwecke geeignet.

Als Alternative zu dime wird seit August 2000 **dxflib** entwickelt. Die dxflib-Projektseite findet sich auf SourceForge[3], eine populäre Seite für OpenSource-Projekte. Die getestete Version ist von April 2002, dxflib wird auch weiterhin gepflegt. Einige Vermerke zu *libdxf* scheinen eigentlich dxflib gewidmet. So beziehen sich die Einträge im Debian-Bugreport unter libdxf offensichtlich auf dxflib. Die Installation erfolgt analog zu der von dime. Es müssen jedoch noch zusätzliche Bibliotheken installiert werden, die nicht

[2] Mit Hilfe von doxygen ([vH02]) kann aus Beschreibungstexten u.a. im C++-Source-Code eine Klassenreferenz (ähnlich `javadoc` für Java) generiert werden.

[3] `http://www.sorceforge.net`

```
{ Vorbedingung:
   • filename enthält den Namen einer beschreibbaren Datei.
}
procedure writePot(Depth depth, String filename)
begin
   out.open(filename)
   out.writeLine(depth)
   wCount:= 0
   wBinary:= 0
   writeTree(getTree())
   if wCount > 0 then
      wBinary:= 2^8−wCount* wBinary
      out.writeChar(wBinary)
   end if
   out.close()
end procedure
```

Algorithmus 3.1: **procedure** writePot(*depth, filename*)

im Standardumfang der Distribution enthalten sind. Für dxflib existiert eine Klassenreferenz. Dxflib scheint sich noch in einem frühen Stadium zu befinden. Wesentliche Funktionen für den notwendigen Import sind noch nicht enthalten. Dxflib ist somit in der aktuellen Version als Importbibliothek unbrauchbar. Die laufende Pflege verspricht aber für zukünftige Zeiten Besserung.

Zu **libdxf** finden sich nur eine begrenzte Anzahl von Verweisen. Libdxf wurde seit 1997 entwickelt, wird aber anscheinend seit 1999 nicht mehr weitergepflegt. Der tar-Ball[4] lässt sich ähnlich zu dime installieren und enthält neben dem Source-Code ein Beispielprogramm. Mit einigen der Beispielgeometrien konnte das Beispielprogramm von libdxf nicht umgehen. Eine weitere Analyse zeigte, dass dies nicht am Beispielprogramm sondern an der mangelnden Unterstützung durch die Bibliothek liegt. Libdxf ist somit als Importbibliothek unbrauchbar.

Als Importbibliothek wurde deshalb dime verwendet wird.

3.2 Export

Zur permanenten Speicherung der Oktalbaumstruktur wird sie in einem präordertraversierten Binärstrom geschrieben, der der Ausgabedatei zugeordnet ist (vgl. Algorithmus 3.1). Das hier als POT-Format bezeichnete Format besitzt folgende Struktur:

[4]So werden in der Unix-Welt häufig `tar.gz`-Archive genannt.

```
{ Vorbedingung:
   • out ist der der Ausgabedatei zugeordnete geöffnete
     Ausgabestrom.
   • wCount enthält die Anzahl der 'gecachten' Bits - also beim
     externen Aufruf von writeTree() 0.
   • wBinary enthält die 'gecachten' Bits - also beim externen
     Aufruf von writeTree() 0.
}
procedure writeTree(_octree tree)
begin
   Node node:= getNode(tree)

   if isLeaf(node) then
      integer colBit:= 0
      if getColor(node) ≠ NO_OBJECT_COLOR then
         colBit:= 1
      end if
      wBinary:= 4*wBinary + colBit
      wCount:= wCount + 2
   else
      wBinary:= 2*Binary + 1
      wCount:= wCount + 1
   end if

   if wCount ≥ 8 then
      wCount:= wCount / 8
      integer bitsToWrite:= wBinary / 2^wCount
      out.writeChar(bitsToWrite)
      wBinary:= wBinray - bitsToWrite
   end if

   if ¬isLeaf(node) then
      for PartType i from 0 to 2^dim − 1 step 1 loop
         writeTree(getChild(tree, i))
      end for
   end if
end procedure
```

Algorithmus 3.2: **procedure** writeTree(*tree*)

Als Dateikopf (Header) dient eine Zeile im ASCII-Format, die die maximale Baumtiefe enthält. Es folgt der in Präorder-Traversierung (vgl. Abschnitt *Präorder-Traversierung* auf Seite 22) umgewandelte Oktalbaum, wofür Algorithmus 3.2 verwendet werden kann.

Für einen inneren Knoten wird dabei eine `1` geschrieben, ein Blattknoten der Farbe NO_OBJECT_COLOR wird durch `00` repräsentiert und alle anderen Knoten durch `01`.

Um eine möglichst kompakte Darstellung zu erhalten, werden jeweils acht Bits des Binärstroms zu einem Byte zusammengefasst. (Es wird also *nicht* für einen Knoten ein Byte verwendet.)

Kapitel 4

Algorithmen

Basierend auf den Erläuterungen zu Oktalbäumen und zu Oberflächenmodellen, insbesondere zu B-Splines, werden die in dieser Arbeit verwendeten Algorithmen zur Oktalbaumgenerierung aus Oberflächenmodellen mit glatten und gekrümmten Flächen näher beschrieben. Hierzu müssen als erstes grundlegende geometrische Algorithmen, wie die Lagebestimmung oder Schnitt von Objekten im Raum, erklärt werden (Abschnitt 4.1). Ausgehend von Punkten und Linien wird anschließend dargestellt, wie Drei- oder Vierecke (und somit Objekte mit glatter Oberfläche) in den Oktalbaum eingefügt werden können (Abschnitt 4.2). Abschließend widmet sich Abschnitt 4.3 der Generierung von Oktalbäumen für Körper mit gekrümmter Oberfläche.

4.1 Grundlegende geometrische Algorithmen

Bei vielen 'klassischen' Algorithmen werden – wie [Sed94, S.399] vermerkt – "Text und Zahlen verwendet, die in den meisten Programmiersystemen auf natürliche Art dargestellt und verarbeitet werden. Tatsächlich sind die benötigten elementaren Operationen in den meisten Computersystemen hardwaremäßig implementiert. Wir werden sehen, daß die Situation bei geometrischen Problemen anders ist: Selbst die einfachsten Operationen mit Punkten und Linien können mittels Computer schwer realisierbar sein.
Geometrische Probleme lassen sich leicht veranschaulichen, doch das kann störend sein. Viele Probleme, die jemand lösen kann, der ein Stück Papier betrachtet (Beispiel: Befindet sich ein gegebener Punkt innerhalb eines Polygons?), erfordern nichttriviale Programme. Für komplizierte Probleme kann sich (wie bei vielen anderen Anwendungen) das für eine Implementation geeignete Lösungsverfahren durchaus stark von der für einen Menschen geeigneten Lösungsmethode unterscheiden."

Es existieren viele Spezialfälle (z.B. Dreieck zu Strecke oder Punkt entartet), die einer gesonderten Behandlung bedürfen. Andererseits finden sich für geometrische Probleme viele Algorithmen in der Literatur, insbesondere im Internet. Das Beachten (bzw.

Nichtbeachten) von Spezialfällen sowie die Effizienz der vorgeschlagenen Lösungen ist aber meist nicht sofort ersichtlich und kann von Anwendunsszenario zu Anwendungsszenario variieren. Die Missachtung von Spezialfällen ist ein häufiger Programmierfehler.

Dennoch lassen sich einige Prinzipien für alle geometrischen Algorithmen formulieren: Fast immer wird auf Mittel der Vektorrechnung zurückgegriffen. Damit wird auch meist das Problem sehr anschaulich beschrieben. Die Lösung ist auch einfach auf höherdimensionale analoge Probleme anwendbar. (Allerdings gibt es manchmal Lösungen, die sich anderer geometrischer Beziehungen bedienen, die effizienter sind und direkt (ohne Sonderbehandlung) mehr Spezialfälle abdecken). Algorithmen mit Ganzzahlarithmetik sind aus Effizenzgründen ihren Fließkommapendants vorzuziehen. Laufvariablen sollten zur Vermeidung von Akkumulationsfehlern nie Fließkommazahlen sein. Besonderes Augenmerk verdient die diskrete Aufteilung des Raums in Zellen. Allgemein sind Rundungsfehler und weitere numerische Probleme (z.B. Auslöschung) zu beachten. Wichtig ist hier die Frage nach der notwendigen Genauigkeit der Lösung. Die Analyse unterschiedlicher Algorithmen zu einem geometrischen Problem kann deshalb wichtig sein.

Nachfolgend sollen einige verwendete geometrische Algorithmen vorgestellt werden.

4.1.1 Lagebestimmung geometrischer Objekte

Die Bestimmung der Lage eines Objekts bezüglich eines anderen hat zur Oktalbaumgenerierung eine herausragende Bedeutung. Schließlich wird die Geometrie des Oktalbaums über die Frage '*Befindet sich das Objekt ganz innerhalb oder außerhalb oder teilweise inner-/außerhalb einer bestimmten Zelle?*' definiert.

Punkt-in-Zellen-Test

Um einen Punkt einer Zelle zuordnen zu können, müssen die realen Koordinaten des Punkts im gegebenen CAD-Modell auf die Koordinaten des virtuellen Gitters abgebildet werden. Die Abbildung der CAD-Punkte auf das Gitter soll als `GeomPoint` bezeichnet werden.

Zuordnung eines realen Punktes zu einer Zelle unterster Baumebene

Zunächst wird von Zellen der tiefsten Ebene ausgegangen. Sie besitzen die Knotenhöhe $h_{\mathrm{idx}} = 0$ bzw. die Tiefe $d_{t_{\max}}$. In jeder Achsrichtung befinden sich $2^{d_{t_{\max}}}$ Zellen, die mit 0 beginnend durchnummeriert seien. Für einen beliebigen Zellindex **idx** dieser Ebene gilt deshalb

$$\mathbf{idx} \in \left[\left((0)^{\dim} \mid 0 \right) ; \left(\left(2^{d_{t_{\max}}} - 1 \right)^{\dim} \mid 0 \right) \right]. \tag{4.1}$$

Nachbarzellen besitzen in jede Achsrichtung den Index-Abstand 1. Für ein `GeomPoint` **g** gilt:

$$\mathbf{g} \in \mathbf{idx} : \mathbf{g}[i] \in [\mathbf{idx}[i]; \mathbf{idx}[i]+1) \ \wedge \ h_{\mathbf{g}} = h_{\mathbf{idx}} = 0 \tag{4.2}$$

mit $i = 0, \ldots, \dim - 1$. Dabei sind $\mathbf{g}[i]$ und $\mathbf{idx}[i]$ die i-te Koordinate des `GeomPoint` bzw. der Anteil in i-Richtung des Index. Alle Zellen besitzen die gleiche Breite. Unter Verwendung (2.25) ergibt sich eine Maschenweite λ von

$$\lambda = \max_{i \in [0;\dim)} \{\mathbf{p}_{\max}[i] - \mathbf{p}_{\min}[i]\} \ / \ 2^{d_{t_{\max}}}.$$

Dabei enthalten $\mathbf{p}_{\max}$ und $\mathbf{p}_{\min}$ die jeweils größten bzw. kleinsten Koordinaten der boundary cube. Somit kann jeder Punkt $\mathbf{p}$ des CAD-Modells mit $\mathbf{p} \in [\mathbf{p}_{\min}; \mathbf{p}_{\max})$ mit Hilfe der linearen Abbildung

$$\mathbf{g} = (\mathbf{p} - \mathbf{p}_{\min}) \ / \ \lambda \tag{4.3}$$

als entsprechender `GeomPoint` **g** dargestellt werden. Wird $\mathbf{p}_{\min}$ durch

$$\mathbf{p}_{\min} = \left(\min_{\mathbf{p}} \{\mathbf{p}[i]\} \right)^{i=0,\ldots,\dim-1} \tag{4.4}$$

bzw. $\mathbf{p}_{\max}$ durch

$$\mathbf{p}_{\max} = \left(\max_{\mathbf{p}} \{\mathbf{p}[i]\} \right)^{i=0,\ldots,\dim-1} \tag{4.5}$$

definert, können jedoch die Punkte nicht in den Oktalbaum abgebildet werden, die mindestens eine $\max$-Koordinate enthalten. Das sind alle Punkte der drei Randflächen der boundary cube $x = \mathbf{p}_{\max}[0]$, $y = \mathbf{p}_{\max}[1]$ bzw. $z = \mathbf{p}_{\max}[2]$. Solche Punkte würden einer Zelle zugeordnet, die wegen (4.9) einen Index-Anteil von $2^{d_{t_{\max}}}$ ergeben und mit (4.8) außerhalb des Oktalbaums liegen. Eine Lösung wäre, $\mathbf{p}_{\max}$ in jeweils jede Achsrichtung um $\epsilon : \epsilon > 0$ nach 'rechts' zu verschieben, also $\mathbf{p}_{\max}$ anstatt mit Hilfe von (4.5) durch

$$\mathbf{p}_{\max} = \left(\max_{\mathbf{p}} \{\mathbf{p}[i]\} + \epsilon \right)^{i=0,\ldots,\dim-1}$$

zu definieren. Der überschüssige relative Anteil wäre dann jedoch vom CAD-Modell abhängig. Besitzt die Geometrie nur kleine Abmessung, wäre dieser Anteil unnötig groß. Alternativ wird die Maschenweite auf

$$\lambda = \frac{\max_{i \in [0;\dim)} \{\mathbf{p}_{\max}[i] - \mathbf{p}_{\min}[i]\}}{2^{d_{t_{\max}}} - \epsilon} \tag{4.6}$$

vergrößert. Um eine besonders kleine Maschenweite zu erhalten, wird $\epsilon \ll 1$ gesetzt.

Der `GemPoint` **g** befindet (vgl. (4.2)) sich in der Zelle mit dem Index **idx**

$$\mathbf{idx} = \lfloor \mathbf{g} \rfloor. \tag{4.7}$$

Verallgemeinerung auf beliebige Zellebenen

Die im letzten Punkt gemachten Betrachtungen lassen sich auf beliebige Zellebenen übertragen. Es gelten dementsprechend für beliebige Höhen des die Zelle repräsentierenden Oktalbaumknotens h (bzw. Tiefen d) analog zu (4.1), (4.2) und (4.6):

$$\mathbf{idx} \in \left[\left((0)^{\dim} \mid h \right) ; \left(\left(2^d - 1 \right)^{\dim} \mid h \right) \right], \tag{4.8}$$

$$\mathbf{g} \in \mathbf{idx} : \mathbf{g}[i] \in [\mathbf{idx}[i]; \mathbf{idx}[i] + 1) \ \wedge \ h_{\mathbf{g}} = h_{\mathbf{idx}} \tag{4.9}$$

mit $i = 0, \ldots, \dim -1$ bzw.

$$\lambda = \frac{\max_{i \in [0;\dim)} \{\mathbf{p}_{\max}[i] - \mathbf{p}_{\min}[i]\}}{2^d - \epsilon} \tag{4.10}$$

und somit

$$\mathbf{g} = \left((\mathbf{p} - \mathbf{p}_{\min}) \ * \ \frac{2^d - \epsilon}{\max_{i \in [0,\ldots,\dim -1)} \{\mathbf{p}_{\max}[i] - \mathbf{p}_{\min}[i]\}} \mid h \right). \tag{4.11}$$

Zuordnung des Zellindex zu einem `GeomPoint`

$g_{\mathbf{idx}_{\downarrow}} = \mathbf{idx}$ liefert den 'rechten unteren vorderen' `GeomPoint` der Zelle **idx**. Als `GeomPoint` $g_{\mathbf{idx}}$ einer Zelle wird häufig ihr Mittelpunkt definiert. Wir setzen

$$g_{\,\mathbf{idx}} = \ \mathbf{idx} + (½)^{\dim}. \tag{4.12}$$

Punktgleichheit

Wann sind zwei Punkte $\mathbf{g}_1$ und $\mathbf{g}_2$ im Oktalbaum als identisch anzusehen? Zum einem könnte man zwei Punkte als gleich betrachten, wenn sie zur gleichen Zelle gehören, also $\lfloor \mathbf{g}_1 \rfloor = \lfloor \mathbf{g}_2 \rfloor$. Unter anderem für den auf der nächsten Seite behandelten *Punkt-in-Ebene-Test* eignet sich eine Abstandsbeziehung besser. Die Eckpunkte einer Zelle $\mathbf{g}_{\,\mathbf{idx}^*}$ haben vom Mittelpunkt $\mathbf{g}_{\mathbf{idx}}$ einen Abstand $a = \sqrt{3 * ½^2}$. Wir definieren

$$\mathbf{g}_1 = \mathbf{g}_2 \iff |\mathbf{g}_1 - \mathbf{g}_2|^2 < ¾. \tag{4.13}$$

Punkt-auf-Strecke-Test

Ein Punkt $\mathbf{Q}$ befindet sich auf der Strecke $\overline{\mathbf{AB}}$ falls

- $\mathbf{Q}$ ein Endpunkt der Strecke ist, also $\mathbf{Q} = \mathbf{A}$ oder $\mathbf{Q} = \mathbf{B}$, oder
- $\mathbf{Q}$ sich auf der Geraden, die durch $\mathbf{A}$ und $\mathbf{B}$ verläuft, befindet und zwischen $\mathbf{A}$ und $\mathbf{B}$ liegt.

$\mathbf{Q}$ liegt auf der Geraden $\mathbf{AB}$, wenn der Fußpunkt $\mathbf{F_Q}$ zu $\mathbf{Q}$ auf der Geraden $\mathbf{AB}$ mit $\mathbf{Q}$ zusammenfällt, also gilt

$$\mathbf{Q} = \mathbf{F_Q}. \tag{4.14}$$

Der Fußpunkt kann mit Hilfe des Skalarprodukts berechnet werden:

$$\overrightarrow{OF_Q} = \frac{<\overrightarrow{AQ}; \overrightarrow{AB}>}{|\overrightarrow{AB}|^2} \overrightarrow{AB} + \overrightarrow{OA}. \tag{4.15}$$

Ein Punkt $\mathbf{Q}$, der sich auf der Geraden $\mathbf{AB}$ befindet, liegt zwischen $\mathbf{A}$ und $\mathbf{B}$, wenn gilt:

$$|\overline{AQ}| < |\overline{AB}| \ \wedge \ |\overline{QB}| < |\overline{AB}|. \tag{4.16}$$

Punkt-in-Ebene-Test

Im Folgenden werden zwei Methoden erläutert, um zu bestimmen, ob sich ein Punkt $\mathbf{Q}$ in der Ebene E, die durch die Punkte $\mathbf{A}$, $\mathbf{B}$ und $\mathbf{C}$ gegeben ist, befindet. Dabei ist zu beachten

1. **Sonderfälle:** Sind $\mathbf{A}$, $\mathbf{B}$ und $\mathbf{C}$ kollinear (oder gar identisch), so ist undefiniert, ob $\mathbf{Q}$ in E liegt.
2. ***'Abstandsregel'*:** Der Punkt-in-Ebene-Test wird insbesondere dazu benutzt, um zu bestimmen, ob E durch eine Zelle mit dem Index $\mathbf{i}$ verläuft. Der Zelle wird der Punkt $\mathbf{g}_i$ zugeordnet, der den Mittelpunkt der Zelle darstellt. Verläuft die Ebene nicht durch den Zellmittelpunkt sondern lediglich durch die Zellrandfläche, muss dennoch die Zelle als zur Ebene gehörig angesehen werden. Deshalb werden Punkte, die in einen Abstand von $\leq$ ½ zu E besitzen, mit zur Ebene E gerechnet.

Über den Fußpunkt

Gilt für den Fußpunkt $\mathbf{F_Q}$ des Punktes $\mathbf{Q}$ auf der Ebene E

$$\mathbf{F_Q} = \mathbf{Q}, \tag{4.17}$$

dann liegt $\mathbf{Q}$ in E. $\mathbf{F_Q}$ kann über

$$\overrightarrow{OF_Q} = \overrightarrow{OQ} - \left(\left\langle \overrightarrow{n_E}; \overrightarrow{OQ} \right\rangle - d \right) \overrightarrow{n_E} \tag{4.18}$$

berechnet werden. Dabei ist $\overrightarrow{n_E}$ nach (2.1) der Normalenvektor der Ebene.

Über das Determinantenverfahren

Das Volumen V des durch die Vektoren $\overrightarrow{a}$, $\overrightarrow{b}$ und $\overrightarrow{c}$ gegebenen Spans kann durch

$$V = \det\left(\overrightarrow{a}; \overrightarrow{b}; \overrightarrow{c} \right) \tag{4.19}$$

ermittelt werden. Damit ergibt sich folgende Beziehung für den Abstand a des Punktes $\mathbf{Q}$ zur Ebene E

$$\det{}^2\left(\overrightarrow{AB};\overrightarrow{AC};\overrightarrow{AP}\right) < a^2 \left|\overrightarrow{AB} \times \overrightarrow{AC}\right|^2. \tag{4.20}$$

Mit einem maximal zulässigen Abstandsquadrat von $a^2 = ½$ ergibt sich

$$\mathbf{Q} \in E \iff \det{}^2\left(\overrightarrow{AB};\overrightarrow{AC};\overrightarrow{AP}\right) \leq ½ \left|\overrightarrow{AB} \times \overrightarrow{AC}\right|^2. \tag{4.21}$$

Punkt-in-Polygon-Test

Für den Test, ob sich ein Punkt $\mathbf{Q}$ in einem *beliebigen* Polygon befindet, bietet sich die **Strahlmethode** an:

Sei ein Polygon H durch den geschlossenen Streckenzug $\overline{\mathbf{p}_0\mathbf{p}_1} \ldots \overline{\mathbf{p}_{n-2}\mathbf{p}_{n-1}}\ \overline{\mathbf{p}_{n-1}\mathbf{p}_n}$ mit $\mathbf{p}_n = \mathbf{p}_0$ gegeben. Dann kann mit Hilfe des Teststrahls $\overrightarrow{\mathbf{QW}}$ überprüft werden, ob $\mathbf{Q}$ auf dem Rand, innerhalb oder außerhalb von H liegt. Dabei muss garantiert sein, dass sich $\mathbf{W}$ außerhalb von H befindet. Zu diesem Zweck wird $\mathbf{W}$ am besten auf eine Position außerhalb der boundary cube festgelegt, z.B. : $\mathbf{W} = \left(\mathbf{Q}[0];\mathbf{Q}[1];2^{d_\mathbf{Q}}|h_\mathbf{Q}\right)$. Nun wird für alle Streckenzüge $\overline{\mathbf{p}_i\mathbf{p}_{i+1}}$ mit $i = 0,\ldots,n-1$ überprüft, ob sie den Teststrahl schneiden. Liegt $\mathbf{Q}$ auf einem Streckenzug, liegt $\mathbf{Q}$ auf dem Polygonrand und der Test kann abgebrochen werden. Ist ansonsten die Anzahl der Schnitte gerade, dann liegt $\mathbf{Q}$ innerhalb, ist die Anzahl der Schnitte ungerade, befindet sich $\mathbf{Q}$ außerhalb des Polygons, also gilt:

$$\mathbf{Q}\begin{cases} \text{on } H, & \mathbf{Q} \in \overline{\mathbf{p}_i\mathbf{p}_{i+1}} \\ \text{in } H, & \nu \bmod 2 = 0 \\ \notin H, & \nu \bmod 2 = 1 \end{cases}. \tag{4.22}$$

Dabei ist $i = 0,\ldots,n-1$ und ν die Anzahl der Schnitte. Folgende Sonderfälle sind zu beachten: Ein Streckenzug wird in einem Eckpunkt vom Teststrahl geschnitten oder fällt mit dem Teststrahl zusammen. Diese Fälle werden genau dann als Schnitt gezählt, wenn es im Folgenden zu einem Seitenwechsel bezüglich des Teststrahls kommt.

Das Verfahren lässt sich auch auf beliebige **Polyeder** (dann mit Test auf Schnitt der Teiloberflächen) übertragen.

Alternativ lässt sich das Verfahren leicht abwandeln, wenn man die *Orientierung* der Polygon-/Polyeder-Ränder ausnutzt (**Teststrahlmethode**). Jetzt muss nur der zu $\mathbf{Q}$ am nächsten gelegene Streckenzug bzw. die zu $\mathbf{Q}$ am nächstengelegene Teiloberfläche[1] ermittelt werden. Dann kann mit Hilfe des unten beschriebenen Tests der *Lage eines Punktes bezüglich einer Ebene* (als Basispunkt der Ebene den Schnittpunkt $\mathbf{S}$ einsetzen)

[1] wieder entlang eines Teststrahls oder alternativ der/die allgemein am nächsten gelegene Streckenzug bzw. Teiloberfläche

bestimmt werden, ob sich der Punkt innerhalb oder außerhalb des Polyeders befindet. Liegt $\mathbf{Q}$ in der Ebene der Teiloberfläche, so wird diese Fläche einfach unberücksichtigt gelassen. Der Sonderfall $\mathbf{Q}$ liegt in der Teilfläche wurde zuvor durch $\mathbf{Q} = \mathbf{S}$ überprüft. Befindet sich $\mathbf{Q}$ in der Ebene, aber nicht auf der Teilfläche, muss es mindestens eine weitere Fläche geben, die genauso weit von $\mathbf{Q}$ entfernt ist, deren Ebene jedoch nicht mit dem Teststrahl $\overarrow{\mathbf{QS}}$ zusammenfällt. Kann keine nächstliegende Fläche gefunden werden, befindet sich $\mathbf{Q}$ außerhalb des Polyeders. Prinzipiell lässt sich das Verfahren auch auf Objekte mit gekrümmten Rand übertragen. Dabei ist jedoch – neben der schwereren Schnittpunktermittlung – zu beachten, dass mehrere Schnittpunkte (nicht nur im Sonderfall des Zusammenfallens von Streckenzug/Flächenstück mit dem Teststrahl) existieren können. Des Weiteren ist die Orientierung gekrümmter Flächen i.A. nicht global gleich und muss dehalb am entsprechenden Schnittpunkt ermittelt werden.

Werden nicht beliebige, sondern nur *konvexe Polygone* eingesetzt, können effizientere oder einfachere Verfahren gefunden werden:

Besonders effizient ist dabei das **Umlaufsinnverfahren**. Sei ein Dreieck $H = \triangle\mathbf{ABC}$ im *Zweidimensionalen* gegeben. Der Punkt $\mathbf{Q}$ befindet sich genau dann in H, wenn der Umlaufsinn der Dreiecke die durch Ersetzen von jeweils einem der Punkte $\mathbf{A}$, $\mathbf{B}$ bzw. $\mathbf{C}$ durch $\mathbf{Q}$ entstehen, mit dem Umlaufsinn von H identisch ist, also

$$\mathrm{ccw}(H) = \mathrm{ccw}(\triangle\mathbf{QBC}) = \mathrm{ccw}(\triangle\mathbf{AQC}) = \mathrm{ccw}(\triangle\mathbf{ABQ}). \tag{4.23}$$

Dabei ist ccw( ) ein Funktion, die zu einem Dreieck seinen Umlaufsinn bestimmt. Die Umlaufsinnbestimmung kann im zweidimensionalen Fall mit Hilfe des Skalarprodukts sehr effizient geschehen (vgl. [Sed94]). Die Übertragung ins Dreidimensionale erweist sich als schwieriger. Hier gibt es keine natürliche Definition des Uhrzeigersinns. Als Lösung könnten im Falle, dass $\mathbf{Q}$ in der Ebene des Dreiecks H liegt, die betreffenden Punkte in einer (zweidimensionalen) Basis zweier Ebenenvektoren dargestellt werden. Alternativ könnten die Richtungen der Normalvektoren der Dreiecke verglichen werden.

[Sha01] stellt das in dieser Arbeit verwendete **Winkelverfahren** vor. Ein Punkt $\mathbf{Q}$, der in der Ebene E des konvexen Polygons H liegt, befindet sich genau dann innerhalb von H, wenn gilt:

$$\sum_{i=0}^{n-1} \arccos \left\langle \frac{\overrightarrow{Qp_i}}{|\overrightarrow{Qp_i}|}; \frac{\overrightarrow{Qp_{i+1}}}{|\overrightarrow{Qp_{i+1}}|} \right\rangle = 2\pi. \tag{4.24}$$

Dabei ist wieder $\overline{\mathbf{p}_0\mathbf{p}_1}, \ldots, \overline{\mathbf{p}_{n-1}\mathbf{p}_n}$ (mit $\mathbf{p}_n = \mathbf{p}_0$ und $\mathbf{Q} \notin \overline{\mathbf{p}_i\mathbf{p}_{i+1}}$) der den Rand von H definierende geschlossen Streckenzug.

Für konvexe Polygone kann auch das **Eckpunkt-Schnittverfahren** verwendet werden. Ein Punkt $\mathbf{Q}$ der sich nicht auf dem Polygonrand befindet, liegt genau dann im Polygon H, welches durch den Streckenzug $\overline{\mathbf{p}_0\mathbf{p}_1}$,…,$\overline{\mathbf{p}_{n-1}\mathbf{p}_n}$ mit $\mathbf{p}_n = \mathbf{p}_0$ gegeben ist, wenn gilt

$$\forall i \in [0;n) \forall k \in [0;n) \setminus \{i; i+1\} : \overline{\mathbf{Qp}_i} \wedge \overline{\mathbf{p}_k\mathbf{p}_{k+1}} = \emptyset, \tag{4.25}$$

also keine Strecke $\overline{\mathbf{Qp}_i}$ durch einen Streckenzug des Polygons $\overline{\mathbf{p}_k\mathbf{p}_{k+1}}$ geschnitten wird.

Lage eines Punktes bezüglich einer Fläche

Aus der Orientierung der Flächen nach (2.1) ergibt sich für alle Flächen eine Innen- und Außenseite. Für einen Punkt $\mathbf{Q}$, der sich nicht auf der Ebene befindet, kann nun durch

$$\mathbf{Q} \text{ is } \begin{cases} \text{inside}, & \varphi > 0 \\ \text{outside}, & \varphi < 0 \end{cases} \tag{4.26}$$

sein Lage berechnet werden. φ ist ein dem Winkel zwischen dem Normalenvektor $\overrightarrow{n_E}$ und dem Vektor zwischen $\mathbf{Q}$ und einem Flächenpunkt $\mathbf{F}$ entsprechender Wert. Es gilt

$$\varphi = \left\langle \overrightarrow{n_E}; \overrightarrow{QF} \right\rangle . \tag{4.27}$$

4.1.2 Schnitt geometrischer Objekte

Im Folgenden werden einige Schnittverfahren für einfache Objekte vorgestellt. Wie im oberen Abschnitt zu sehen ist, wird die Schnittbestimmung insbesondere für einige Verfahren zur Lagebestimmung verwendet.

Schnitt zweier Strecken

[Sed94] beschreibt einen besonders effizienten Algorithmus, der überprüft, ob sich zwei Strecken $\overline{\mathbf{AB}}$ und $\overline{\mathbf{CD}}$, die sich in einer Ebene befinden, schneiden. Der Einfachheit halber soll verlangt sein, dass die Punkte $\mathbf{A}$, $\mathbf{B}$, $\mathbf{C}$, $\mathbf{D}$ paarweise verschieden sind und nicht kollinear liegen. (Der Originalalgorithmus macht diese Einschränkung nicht, da hier für diese Sonderfälle die $\text{ccw}(\mathbf{p}_1,\ \mathbf{p}_2,\ \mathbf{p}_3)$ speziell definierte Werte liefert.)

$$\begin{gathered} \overline{\mathbf{AB}} \text{ intersect } \overline{\mathbf{CD}} \iff \\ (\text{ccw}(\mathbf{A, B, C}) = \text{ccw}(\mathbf{B, A, D})) \wedge (\text{ccw}(\mathbf{C, D, A}) = \text{ccw}(\mathbf{D, C, B})) \end{gathered} \tag{4.28}$$

Schnitt einer achsparallelen Geraden mit einem Polygon

Es soll der Schnittpunkt zwischen einer achsparallelen Geraden mit einem Polygon H bestimmt werden, wenn ein solcher existiert. Hier soll auf die Mittel der Vektorrechnung zurückgegriffen werden. Die Verwendung einer achsparallelen Gerade für den Schnitt hat den Vorteil, dass bereits zwei Koordinaten des Schnittpunkts bekannt sind. Es sind die jeweiligen Koordinaten der Geraden. O.B.d.A. soll eine Gerade parallel zur z-Achse betrachtet werden, die durch den Punkt $\mathbf{Q} = (x_0; y_0; z_0|h)$ verläuft:

$$g = \{(x_0; y_0; z|h) : z \in \mathcal{R}\} .$$

Dann muss für den Schnittpunkt $\mathbf{S} = (x_0; y_0; z_\mathbf{S})$ gelten:

$$(x_0; y_0; z_\mathbf{S}) \in H. \tag{4.29}$$

Die Gerade liegt in der Polygonebene oder verläuft parallel zu ihr, falls

$$\overrightarrow{n_H}\,[2] = 0. \tag{4.30}$$

Solche Fälle werden im Folgenden nicht weiter betrachtet. Für den Schnittpunkt gilt nun

$$z_\mathbf{S} = z_0 + \frac{\left\langle \overrightarrow{Qp_0}; \overrightarrow{n_H} \right\rangle}{\overrightarrow{n_H}\,[2]}. \tag{4.31}$$

Zur Bestimmung, ob der Schnittpunkt mit der Polygonebene auch im Polygon selbst liegt, kann die Projektion in die xy-Ebene betrachtet und die Umlaufsinnmethode angewendet werden.

Für die Strahlmethode (im Abschnitt *Punkt-in-Polygon-Test* auf Seite 38 beschrieben) muss nun noch $z_\mathbf{S} > \mathbf{Q}[2]$ sein.

4.2 Erzeugung von Körpern mit glatter Oberfläche

Zur Oktalbaumgenerierung wird häufig ein Verfahren ähnlich dem bei [Jak01] verwendet: Suksessiv wird der Oktalbaum solange verfeinert, bis eine durch den Oktalbaum-Knoten repräsentierte Zelle sich vollständig innerhalb oder außerhalb des Körpers befinden oder die maximale Baumtiefe erreicht wurde. Die Blätter werden entsprechend gefärbt.

Gerade bei der Verwendung von Körpern mit gekrümmten Flächen ist aber die Lokalisation von Punkten und damit der zu betrachtenden Zellen aufwändig. Deshalb wurde innerhalb dieser Arbeit ein alternatives Verfahren entwickelt und analysiert. Um den Zeitaufwand zur Oktalbaumgenerierung zu minimieren, sollte die Anzahl der notwendigen Lagebestimmungen von Zellen (bezüglich der modellierten Körper) minimiert werden. Stattdessen sollte die Struktur der modellierten Körper durch effiziente Verfahren in den Oktalbaum übertragen werden. Die Grundidee ist dabei, zuerst die Oberfläche der Körper in die Oktalbaumstruktur zu generieren und anschließend die Körper zu 'füllen'. Die Anzahl der Lagebestimmungen η beträgt dann idealerweise nur noch

$$\eta = 1 + \kappa, \tag{4.32}$$

wenn κ die Anzahl zusammenhängender vollständig abgetrennter Körpersegmente ist. Aufbauend auf dem Verfahren zur Punktegenerierung in Oktalbäumen wird dargestellt, wie glatte Oberflächen in einer Oktalbaumstruktur erzeugt werden können. Danach wird auf Füllmethoden eingegangen. Abschließend werden die unterschiedlichen Algorithmen zur Übertragung von Körpermodellen in eine Oktalbaumstruktur verglichen.

4.2.1 Punkte

Das einfachste Objekt, welches in einem Oktalbaum dargestellt werden kann, ist ein Punkt. Dabei ist jedoch zu beachten, dass dem Punkt im Oktalbaum ein Volumen zugeordnet wird (nämlich das Volumen der Zelle, in der er sich befindet). Da Punkte jedoch eigentlich keine Abmessungen besitzen, sind sie zweckmäßiger Weise in der untersten Baumebene (also mit der Knotenhöhe $h = 0$) einzufügen. Im Abschnitt 4.1.1 wurde bereits erläutert, wie ein Punkt des CAD-Modells in einen `GeomPoint` **g**, also in die Koordinaten des Oktalbaums, transformiert werden kann. Der Index $\mathbf{g}_{\text{idx}}$ von **g** entsprach dabei dem Zellindex **idx**.

Im Folgenden wird gezeigt, wie sich hieraus der Pfad durch den Oktalbaum zum entsprechenden Knoten bestimmen lässt. Bei der Verfeinerung einer Zelle wird diese in jede Achsrichtung halbiert. Liegt die Sohnzelle im jeweiligen 'oberen' Teilraum, so erhält die Pfadkomponente dieser Achsrichtung η_{ax} den Wert 1 zugewiesen und 0 entsprechend für den 'unteren' Teilraum. Ist $\mathbf{g}_m$ der Mittelpunkt der Zelle gilt also:

$$\eta_{\text{ax}} = \begin{cases} 0, & \mathbf{g} < \mathbf{g}_m \\ 1, & \mathbf{g} \geq \mathbf{g}_m \end{cases} . \tag{4.33}$$

Über die Pfadkomponenten für jede Achsrichtung kann die Pfadnummer η definiert werden:

$$\eta = \sum_{i=0}^{\dim -1} \eta_i \, 2^i. \tag{4.34}$$

Die Reihenfolge der Sohnknoten soll entsprechend der Pfadnummerierung erfolgen. Für den virtuellen Pfad zur Wurzel wird die Pfadnummer 0 definiert. Somit hat die Wurzel auch die 'Sohnnummer' 0.

Für den Index-Teil $\mathbf{idx}^h_{\text{ax}}$ einer Zelle für die Höhe h in ax-Richtung ergibt sich:

$$\mathbf{idx}^h_{\text{ax}} = (\ \mathbf{idx}[\text{ax}]/2^h\ ) \mod 2 = \begin{cases} 0, & \text{Zelle im 'unteren' Teilraum} \\ 1, & \text{Zelle im 'oberen' Teilraum} \end{cases} . \tag{4.35}$$

Damit gibt $\mathbf{idx}^h_{\text{ax}}$ die Sohnnummer des Knotens wieder, der sich auf dem Pfad zum Knoten, der die Zelle **idx** repräsentiert, auf der Höhe h befindet (vgl. Algorithmus 4.1). Der Index **idx** ist somit eine reguläre *Positionscodierung* (vgl. S. 20). Er wird deshalb auch als Index des entsprechenden Knotens (im Sinne der Positionscodierung) angesehen.

Mit Hilfe des Algorithmus 4.2 kann zu einem Index der zugehörige Oktalbaumknoten in $\mathcal{O}(d_{\mathbf{idx}})$ und somit insbesondere in $\mathcal{O}(d_{t_{\max}})$ gefunden werden.

Durch `function isIn(NodeIndex idx) return bool` kann ermittelt werden, ob durch `idx` ein Volumen des Oktalbaums referenziert wird. Dazu wird (4.8) verwendet.

$$\text{isIn}(\texttt{idx}) = \begin{cases} \text{true}, & \mathbf{idx} \in \left[\left((0)^{\dim}|h_{\mathbf{idx}}\right); \left((2^{d_{t_{\max}}-h}\mathbf{idx} - 1)^{\dim}|h_{\mathbf{idx}}\right)\right] \\ \text{false}, & \text{sonst} \end{cases} \tag{4.36}$$

```
function getPart(NodeIndex idx) return PartType
begin
   PartType part:= 0
   for Axis ax from 0 to dim −1 step 1 loop
      part:= part + (idx[ax]/2^(h−1)) mod 2
   end for
   return part
end function
```

Algorithmus 4.1: **function** getPart(*idx*) **return** PartType

```
{ Vorbedingung: isIn(idx) ∧ getExistNode(idx) = idx }
function getNode(NodeIndex idx) return Node
begin
   _octree nodeAddr:= getTree()
   for Hight h from d_tmax to h_idx step −1 loop
      nodeAddr:= getChild(nodeAddr, getPart(idx))
   end for
   return nodeAddr
end function
```

Algorithmus 4.2: **function** getNode(*idx*) **return** _octree

Im Algorithmus 4.3 liefert getExistNode(*idx*) *idx*, falls der entsprechende Knoten bereits im Zeigergeflecht des Oktalbaums als Knoten eingetragen ist und ansonsten den Knoten mit kleinster Höhe auf dem Pfad zu ihm, der existiert.

Beim Hinzufügen von Punkten zum Oktalbaum ist zu beachten, dass i.A. der Pfad zum hinzuzufügenden Knoten teilweise existiert, der untere Teil des Pfads muss aber erst im Oktalbaum erzeugt werden. Algorithmus 4.4 zeigt ein Verfahren, das zum Hinzufügen von Punkten mit dem Index *idx* und der Farbe *color* in den Oktalbaum verwendet werden kann.

function createLeafs(*_octree nodeAddr*) **return** _octree wird zum Erzeugen neuer Söhne zu einem Blatt im Oktalbaum verwendet. Der Rückgabewert ist eine Referenz auf die neuerzeugten Sohnknoten. Der Vaterknoten ist damit – natürlich – jetzt innerer Knoten. Es soll zunächst davon ausgegangen werden, dass die Sohnknoten mit der Farbe NO_OBJECT_COLOR (also als 'zu keinem Körper gehörend') initialisiert werden. Für den Füllalgorithmus im Abschnitt 4.2.4 wird allerdings eine andere Initialisierungsfarbe verwendet.

Aus einem Index lässt sich also in $\mathcal{O}(d_{t_{\max}})$ der zugehörige Knoten ermitteln oder in den Baum einfügen (und damit seine Farbe ermitteln oder setzen).

```
{ Vorbedingung: isIn(idx) }
function getExistNode(NodeIndex idx) return NodeIndex
begin
   Hight h:= d_tmax
   _octree nodeAddr:= getTree()

   while ¬isLeaf(getNode(nodeAddr)) ∧ h > h_idx loop
      nodeAddr:= getChild(nodeAddr, getPart(idx))
      h:= h - 1
   end for

   NodeIndex retVal:= idx
   for Axis ax from 0 to dim −1 step 1 loop
      retVal[ax]:= retVal[ax] /2^(h−h_idx)
   end for
   h_retVal:= h
   return retVal
end function
```

Algorithmus 4.3: **function** getExistNode(*idx*) **return** NodeIndex

```
procedure add(NodeIndex idx, Color color)
begin
   NodeIndex leafAtPath:= getExistNode(idx)
   _octree nodeAddr:= getNode(leafAtPath)
   while h_leafAtPath > 0 loop
      PartType part:= getPart(leafAtPath)
      nodeAddr:= createLeafs(nodeAddr)
      leafAtPath:= getChild(leafAtPath, part)
   end for
   setColor(getNode(nodeAddr), color)
end procedure
```

Algorithmus 4.4: **procedure** add(*idx, color*)

Algorithmus 4.4 lässt sich auch zum Erzeugen eines Oktalbaums aus einem Normzellen-Aufzählungsschema verwenden. Für die maximale Baumtiefe $d_{t_{\max}}$ muss gelten:

$$\nu = 2^{d_{t_{\max}}}, \tag{4.37}$$

wenn ν die Anzahl der als Würfel gegebenen Normzellen in jeder Achsrichtung ist. Für die Konvertierung des gesamten Normzellen-Aufzählungsschemas ist somit $\mathcal{O}(d_{t_{\max}} * 2^{\dim * d_{t_{\max}}}) = \mathcal{O}(\log(\nu) * \nu^3)$ Zeit notwendig. Der gleiche Zeit-

```
function octree2nas() return array (0..ν − 1)^dim of Color
begin
   array (0..ν − 1)^dim of Color colors
   for AxIndex z from 0 to ν − 1 step 1 loop
      for AxIndex y from 0 to ν − 1) step 1 loop
         for AxIndex x from 0 to ν − 1 step 1 loop
           NodeIndex idx:= (x; y; z|0)
           idx:= getExistNode(idx)
           colors[x][y][z]:= getColor(idx)
         end for
      end for
   end for
   return colors
end function
```

Algorithmus 4.5: **function** octree2nas() **return array** [][][] **of** Color

aufwand ist auch für die andere Richtung – die Konvertierung des Normzellen-Aufzählungsschemas aus einem Oktalbaum – notwendig. Hierzu müssen lediglich die Funktionen getExistNode() (vgl. Algorithmus 4.5) und getColor() verwendet werden.

4.2.2 Linien

Zum Erzeugen von Linien im Oktalbaum kann auf das Generieren von Punkten aufgebaut werden.

Um einer Strecke $\overline{AB}$ im Oktalbaum darzustellen, muss in alle Zellen, durch die die Strecke führt, ein Punkt eingefügt werden.

Aus dem Bereich der Rastergrafik ist der Bresenham-Algorithmus bekannt. Eine Linie von einem Anfangspunkt **A** zu einem Endpunkt **B** wird über das Raster auf der xy-Ebene approximiert, indem sukzessive zu dem direkten oder diagonalen Nachbar gegangen wird, der in Richtung **B** der Ideallinie am nächsten liegt. Die ausschließliche Nutzung ganzer Zahlen ist eine weitere Eigenschaft des Bresenham-Algorithmus (vgl. Abb. 4.6).

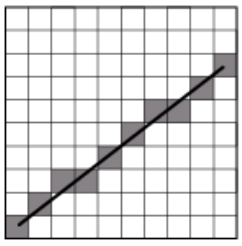


Abb. 4.6: Bresenham-Algorithmus

In der digitalen Regelungstechnik zum Verfahren von Strecken kommt ein ähnliches Verfahren zum Einsatz. Hierbei wird jedoch das zu regelnde Teil nur entlang einer Koordinatenachse gleichzeitig bewegt. Während der Bresenham-Algorithmus 16 Fallunterscheidungen (für jede Koordinatenachse, jede Winkelhalbierende und jede Fläche zwischen ihnen für beide Richtungen) macht, werden in dieser Alternative alle Fälle gleichartig behandelt. Des Weiteren ist die Übertragung ins 3-Dimensionale hier einfach. Das benutzte Verfahren lehnt sich stark an das Alternativverfahren an (vgl. Abb. 4.7).

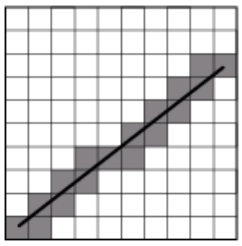


Abb. 4.7: Scan-Line-Basisverfahren dieser Arbeit

Unabhängig vom konkret benutzten Verfahren gelten folgende Beziehungen:

- Aus dem Differenzvektor $\Delta = \mathbf{B} - \mathbf{A}$ lässt sich die Steigung bzw. das Steigungsverhältnis der Strecke ermitteln.
- Es wird immer eine Nachbarzelle/ein Nachbarpunkt gesucht. Der Abstand zwischen zwei Nachbarpunkten auf einer Koordinatenachse ist nie größer als 1. Daraus ergibt sich der zusammenhängende Verlauf der entstehenden Kurve.
- Da Δ die Richtung von $\overrightarrow{AB}$ festlegt, ist somit auch die Richtung gegeben, in welcher auf der jeweiligen Koordinatenachse der Nachbar gesucht werden muss.
- Im Allgemeinen wird die $\overrightarrow{AB}$ durch eine treppenartige Kurve angenähert, die von $\mathbf{A}$ ausgeht und in $\mathbf{B}$ endet. Für jeden Punkt der Kurve ergibt sich zur Ideallinie: $\overrightarrow{AB} \equiv \{\mathbf{p} : \mathbf{p} = \mathbf{A} + \tau\ \Delta\}$ mit $\tau \in [0, 1]$. Damit kann ein Fehler η zu jedem Kurvenpunkt ermittelt werden. Es gilt:

  $$\eta_{\mathbf{A}} = \eta_{\mathbf{B}} = 0 \tag{4.38}$$

  und für einen beliebigen Punkt

  $$\eta \geq 0. \tag{4.39}$$

  Als jeweils nächsten Punkt wird immer der mit $\eta = \eta_{\min}$ (also minimalen Fehler) aus einer Menge ausgesucht. (Falls es mehrere solche Punkte gibt, wird der erstbeste von ihnen genommen.)

Methoden, die durch dieses Verfahren eine Linie in einem Raster zwischen zwei Punkten annähern, werden als **Scan-Line-Algorithmen** bezeichnet.

Im Folgenden werden zwei Scan-Line-Algorithmen vorgestellt. Das erste Verfahren (Algorithmus 4.8) benutzt auch reellwertige Zahlen zur Bestimmung des Nachfolgepunktes. Das zweite (Algorithmus 4.9) verwendet ausschließlich ganzzahlige Werte.

Zur Ermittlung des Nachfolgepunktes wird ein Vektor **dir** benötigt, der die möglichen Richtungen zum Nachbarpunkt beinhaltet:

$$\mathbf{dir}[\mathrm{ax}] = \mathrm{sign}(\boldsymbol{\Delta}[\mathrm{ax}]),\ \ \forall \mathrm{ax} \in [0; \dim -1]. \tag{4.40}$$

```
{ Vorbedingung: h_idx = h_start = h_end ∧ idx ≠ end }
function bestAxis(NodeIndex idx, NodeIndex start, NodeIndex end)
   return AxIndex
begin
   Axis keptAxis:= 0
   while dir[keptAxis] = 0 ∧ keptAxis < dim loop
      keptAxis:= keptAxis + 1
   end for

   NodeIndex p:= getNext(keptAxis)
   minError:= |g_p g_end→ × Δ/|Δ||
   for Axis ax from keptAxis + 1 to dim −1 step 1 loop
      if dir[ax] ≠ 0 then
         p:= getNext(ax)
         error:= |g_p g_end→ × Δ/|Δ||
         if error < minError then
            keptAxis:= ax
            minError:= error
         end if
      end if
   end for
   return keptAxis
end function
```

Algorithmus 4.8: **function** bestAxis(*idx, start, end*) **return** AxIndex

Dabei soll $\mathrm{sign}(x)$ das Vorzeichen von x liefern:

$$\mathrm{sign}(x) = \begin{cases} 1, & x > 0 \\ 0, & x = 0 \\ -1, & x < 0 \end{cases} . \tag{4.41}$$

Damit steht die Menge P der Punkte fest, aus denen der Nachfolgepunkt von **idx** ausgewählt wird. Hierfür gilt:

$$P = \left\{ \mathrm{ax} \in [0; \dim -1], \mathrm{dir}[\mathrm{ax}] \neq 0 \, \mathrm{getNext}(ax) \right\} \tag{4.42}$$

mit **function** getNext(*AxIndex ax*) **return** NodeIndex

$$h_{\mathrm{getNext}(ax)} = h_{\mathbf{idx}} \wedge \mathrm{getNext}(ax)[i] = \begin{cases} \mathbf{idx}[i] + \mathbf{dir}[i], & i = \mathrm{ax} \\ \mathbf{idx}[i], & i \neq \mathrm{ax} \end{cases} \quad (\forall i \in [0; \dim -1]). \tag{4.43}$$

Algorithmus 4.8 zeigt eine Methode, die den (reellwertigen) Abstand error aller Punkte $\in P$ zur Ideallinie berechnet und die Achse ax liefert, über die ein

```
function getNewErrorVec(Axis testAx) return NodeIndex
begin
   NodeIndex newErrorVec:= errorVec
   AxIndex addErr:= 1
   for Axis ax from 0 to dim −1 step 1 loop
      if ax ≠ testAx ∧ dir[ax] ≠ 0 then
         addErr:= addErr * |dir[ax]|
      end if
   end for
   newErrorVec[testAx]:= newErrorVec[testAx] + addErr
   return newErrorVec
end function
```

Algorithmus 4.9: **function** getNewErrorVec(*testAx*) **return** NodeIndex

Punkt $g_{\min}$ ($g_{\min} \in P \mid \text{error}_{g_{\min}} = \min_{g \in P}\{\text{error}_g\}$) erreicht wird. Der Nachfolgepunkt getNext(*idx*) von **idx** (**idx** $\neq$ **end**) der Scan-Line kann somit über getNext(*bestAxis*(idx, start, end)) berechnet werden.

Um für die Scan-Line-Erzeugung ausschließlich ganzzahlige Werte nutzen zu können, ist das Zwischenspeichern des alten Fehlers – des Punktes **idx** – notwendig. Im dreidimensionalen Fall muss hierfür ein Vektor $(\text{error})^{\dim}$ (in Algorithmus 4.9 *errorVec*) verwendet werden.

$\boldsymbol{\Delta}$ enthält, wie oft in die jeweilige Achsrichtung gegangen werden muss. Ein Punkt $\mathbf{p}$ befindet sich genau dann auf der Ideallinie, wenn für jeweils zwei Achsen i, k ($\boldsymbol{\Delta}[i] \neq 0$, $\boldsymbol{\Delta}[k] \neq 0$) mit $\overrightarrow{v} = \overrightarrow{\mathbf{pg_{end}}}$ gilt:

$$\overrightarrow{v}\,[i] : \boldsymbol{\Delta}[i] = \overrightarrow{v}\,[k] : \boldsymbol{\Delta}[k].$$

Allgemeiner kann somit geschrieben werden:

$$\overrightarrow{v}\,[i] * \prod_{j \in [0;\dim -1] \setminus (\{i\} \cup \{m \mid \boldsymbol{\Delta}[m]=0\})} \boldsymbol{\Delta}[j] = \overrightarrow{v}\,[k] * \prod_{l \in [0;\dim -1] \setminus (\{k\} \cup \{m \mid \boldsymbol{\Delta}[m]=0\})} \boldsymbol{\Delta}[l] \tag{4.44}$$

Ein Schritt entlang der i-Achse hat demnach ein Gewicht von $\prod_{j \in [0;\dim -1] \setminus (\{i\} \cup \{m \mid \boldsymbol{\Delta}[m]=0\})}$ und erfordert entsprechend viele Schritte auf den anderen Achsen, um wieder auf die Ideallinie zu gelangen. **errorVec** wird für den Anfangspunkt für jede Achse mit 0 initialisiert. Demnach ergibt $\text{error} = \max_{i \in [0;\dim -1]}\{\textbf{errorVec}[i]\} - \min_{i \in [0;\dim -1]}\{\textbf{errorVec}[i]\}$ einen Fehler, der zum Abstand des Punktes von der Ideallinie äquivalent ist. Hiermit kann eine getNext()-Methode analog zum reelwertigen Verfahren entwickelt werden.

Sollen auch diagonale Schritte (und nicht nur achsparallele) Schritte möglich sein, müssen Kombinationen aus P mit Punkten, zudem man über unterschiedliche Achsen gelangt, möglich sein. Hierfür kann eine Laufzahl axComb $\in [1; 2^{\dim} - 1]$ verwendet werden, die als Binärvektor $\{0;1\}^{\dim}$ aufgefasst, die gewählten Achsen widerspiegelt. Als

```
procedure addLine(NodeIndex start, NodeIndex end, Color color)
begin
   NodeIndex idx:= start
   add(idx, color)
   while hasNext(idx) loop
      idx:= getNext(idx)
      add(idx, color)
   end for
end procedure
```

Algorithmus 4.10: **procedure** addLine(*start, end, color*)

Ausgangsmenge zur Nachfolgepunkt-Bestimmung erhält man somit $2^P \setminus \emptyset$. Mit Hilfe einer Fehlerbestimmungsmethode analog zur oben erläuterten erhält man das gewünschte Verfahren zur Ermittlung des nächsten Punktes der Scan-Line.

Unter Verwendung der oben beschriebenen Methoden entsteht Algorithmus 4.10 zum Hinzufügen einer Linie in den Oktalbaum. **function** hasNext(*NodeIndex idx*) **return** boolean soll dabei definiert sein über

$$\text{hasNext}(idx) \iff \textbf{idx} \neq \textbf{end}. \tag{4.45}$$

4.2.3 Polygone

Im Folgenden sollen nur Dreiecke und konvexe Vierecke betrachtet werden, da nur sie für das Hinzufügen zu Oktalbäumen in dieser Arbeit relevant sind. Allgemein kann das Hinzufügen konvexer Polygone durch Triangulierung und anschließendes Hinzufügen der erhaltenen Dreiecke durchgeführt werden. (Dieses Verfahren ist natürlich auch auf Vierecke anwendbar.)

Es soll zunächst das Hinzufügen von Dreiecken $\triangle\mathbf{ABC}$ betrachtet werden: Schraffiert man die Dreiecksfläche mit Linien, die jeweils entlang von $\overline{AB}$ beginnen und $\overline{AC}$ enden und deren Abstand nicht 1 pro Achse übersteigt, erhält man das gewünschte Dreieck. Wie Algorithmus 4.11 zeigt, kann das genau mit Hilfe zweier ineinander geschachtelte Scan-Lines erreicht werden.

Dafür sollen hasNext() und next() (setzt den aktuellen Punkt mittels getNext() auf den Nachfolgepunkt) in der Klasse **class** ScanLine gekapselt sein. Mit Hilfe von **function** getCurrent() **return** NodeIndex soll auf den aktuellen Punkt **idx** der Scan-Line zugegriffen werden können.

Algorithmus 4.12 zeigt ein analoges Verfahren für planare konvexe Vierecke.

Alternativ ist das Einfügen der Polygonflächen über den rekursiven Abstieg möglich. Dabei können die Verfahren aus Abschnitt 4.1.1 verwendet werden.

```
procedure addTriangle( NodeIndex pA, NodeIndex pB, NodeIndex pC,
                       Color color)
begin
   ScanLine lineAB:= ScanLine(pA, pB)
   ScanLine lineAC:= ScanLine(pA, pC)
   add(pA, color)
   while lineAB.hasNext() ∨ lineAC.getNext() loop
      if lineAB.hasNext() then
         lineAB.next()
      end if
      if lineAC.hasNext() then
         lineAC.next()
      end if
      addLine(lineAB.getCurrent(), lineAC.getCurrent(), color)
   end for
end procedure
```

Algorithmus 4.11: **procedure** addTriangle(*pA, pB, pC, color*)

4.2.4 Polyeder

In den letzten Abschnitten wurden Verfahren hergeleitet, mit denen die Oberflächen von Körpern aus einem Oberflächenmodell in den Oktalbaum übertragen werden können. Der so erzeugte Oktalbaum enthält jedoch nur den Rand und nicht – bei entsprechend hoher Auflösung – das Innere von Körpern. Die Körper wären 'Hüllen ohne Inhalt'. Der Mittelpunkt einer Kugel würde zum Beispiel (eine ausreichend hohe Auflösung vorausgesetzt) fälschlicher Weise als außerhalb des Körpers befindlich definiert werden. Abschnitt *Füllalgorithmen* auf dieser Seite stellt ein Verfahren vor, dass in einem Oktalbaummodell, welches den Rand eines Körpers enthält, den inneren Zellen eines Körpers (und denen die sich außerhalb des Körpers befinden) die richtige Farbe zuordnet. Im Abschnitt 4.2.4 wird das Verfahren aufgegriffen, bei dem mit Hilfe des rekursiven Abstiegs und der Bestimmung, ob sich eine Zelle innerhalb, außerhalb oder auf dem Körperrand befindet, der Oktalbaum erzeugt wird. Abschnitt 4.2.5 vergleicht die vorgestellten Algorithmen und stellt Vor- und Nachteile gegenüber. Die praktischen Ergebnisse (Zeit- und Speicheraufwand), bei dem der Oktalbaum mit Hilfe dieser Algorithmen in unterschiedlichen Maximaltiefen und mit unterschiedlichen Modellen erzeugt wurde, sind jedoch im Abschnitt 5.2 zu finden.

Füllalgorithmen

Aus der Abgeschlossenheit der Körperoberflächen ergibt sich, dass Nachbarzellen von Körperinnenzellen immer Körperinnenzellen oder Körperrandzellen, aber niemals

```
procedure addQuadrilateral( NodeIndex pA, NodeIndex pB,
                            NodeIndex pC, NodeIndex pD,
                            Color color)
begin
   ScanLine lineAB:= ScanLine(pA, pB)
   ScanLine lineDC:= ScanLine(pD, pC)
   add(pA, color)
   while lineAB.hasNext() ∨ lineDC.getNext() loop
      if lineAB.hasNext() then
         lineAB.next()
      end if
      if lineDC.hasNext() then
         lineDC.next()
      end if
      addLine(lineAB.getCurrent(), lineDC.getCurrent(), color)
   end for
end procedure
```

Algorithmus 4.12: **procedure** addQuadrilateral(*pA, pB, pC, pD, color*)

Körperaußenzellen sind. Sucht man aus jedem abgeschlossenen Körperinnenbereich (bezüglich des Oktalbaums) einen Knoten aus und markiert die Nachbarzellen in jede Achsrichtung suksessiv solange, bis der Rand des Körpersegments erreicht ist, erhält man ein Körpermodell, in welchem nicht nur der Rand sondern auch das Körperinnere als zum Körper gehörend repräsentiert wird (natürlich sind äußere Zellen immer noch mit NO_OBJECT_COLOR markiert). Das Verfahren funktioniert also wie ein klassisches Füllverfahren für Objekte in der Rastergeometrie z.B. in der Bildbearbeitung. Dabei sind jedoch zwei Punkte zu beachten:

1. Je nach Körpergeometrie und gewählter Auflösung kann der Körper aus unterschiedlich vielen zu füllenden Körpersegmenten bestehen.
2. Im Gegensatz zum klassischen Füllverfahren sind die Zellen i.A. unterschiedlich groß, da sie auf unterschiedlichen Höhen liegen können. Das bedeutet insbesondere, dass eine Zelle mehrere Nachbarzellen in einer Achsrichtung haben kann.

Zu 1: Die Anzahl der zu füllenden Körpersegmente ist im Voraus nur sehr schwer zu bestimmen. Mit **procedure** createLeafs() neu erzeugte Blätter werden deshalb bei der Anwendung des Füllalgorithmus nicht mit NO_OBJECT_COLOR (vgl. Abschnitt 4.2.1) sondern mit UNDEF_OBJ_COLOR initialisiert. Körperränder werden bei der Oberflächenerzeugung im Oktalbaum wieder mit der Körperfarbe definiert, so dass vor der Anwendung des Füllalgorithmus bis auf alle Körperrandzellen alle Blätter die Farbe UNDEF_OBJ_COLOR besitzen. Wird bei der Traversierung über den Oktalbaum ein UNDEF_OBJ_COLOR-Blatt ge-

```
procedure fillParts(NodeIndex idx, Axis ax, AxDirection dir)
begin
   idx:= getExistNode()
   if isLeaf(idx) then
      fill(idx)
      return
   end if
   for PartType i from 0 to 2^dim − 1 step 1 loop
      if (i/2^ax) mod 2 = { 0, dir = FORWARD
                          { 1, dir = BACKWARD   then
         fillParts(getChild(idx, i), ax, dir)
      end if
   end for
end procedure
```

Algorithmus 4.13: **procedure** fillParts(*idx, ax, dir*)

funden, wird es bezüglich des Körpers lokalisiert (*inside*, *outside*[2]) und entsprechend mit der Körperfarbe bzw. NO_OBJECT_COLOR markiert. Die gleiche Farbe wird nun über die Nachbarblätter auf das gesamte Körpersegment übertragen, bevor die Traversierung mit dem nächsten Blattknoten fortgesetzt wird. Wird auf eine Zelle des gleichen Körpersegments gestoßen, so besitzt sie bereits die Körperfarbe, und muss nicht lokalisiert werden.

Zu 2: Befindet sich ein Blatt **idx** und sein Nachbarblatt auf unterschiedlicher Höhe, gibt es hierfür zwei Möglichkeiten:

Nachbarblatt höher Hier bietet sich die Nutzung von Algorithmus 4.3 an.

Nachbarblatt niedriger Ausgehend vom inneren Nachbarknoten gleicher Höhe werden alle Nachbarblätter durch rekursiven Abstieg ermittelt. Dabei werden jeweils die 4 Söhne in Richtung des **idx**-Knoten – also *entgegengesetzt* der Richtung der Nachbarsuche – betrachtet.

Algorithmus 4.13 sucht die Nachbarzellen zu der Zelle **idx** und ruft fill( ) auf, um den Füllalgorithmus auf ihnen fortzusetzen. Man beachte, dass der Oktalbaum *nicht balanciert* sein muss, damit der Algorithmus korrekt arbeitet.

Algorithmus 4.14 zeigt ein Verfahren, dass das Körpersegment von **idx** vom Blattknoten **idx** aus füllt. Die Traversierung über alle Blätter garantiert, dass nach der Traversierung kein Blatt die Markierung UNDEF_OBJ_COLOR besitzt. Jeder Zelle wurde also die richtige Farbe zugewiesen. Der Oktalbaum bleibt in seiner Struktur erhalten. Lediglich Blattmarkierungen werden verändert.

[2] Das Blatt kann sich nicht auf dem Körperrand befinden. Sonst wäre es ja schon entsprechend markiert.

```
{ Vorbedingung: isLeaf(idx) ∧ color = Körperfarbe }
procedure fill(NodeIndex idx)
begin
   if getColor(idx) ≠ UNDEF_OBJ_COLOR then
      return
   end if
   setColor(idx, color)
   AxIndex max:= 2^(d_tmax − h_idx) − 1
   for Axis ax from 0 to dim−1 step 1 loop
      AxIndex pos:= idx[ax]
      NodeIndex neighbor:= idx
      if pos < max then
         neighbor[ax]:= pos + 1
         fillParts(neighbor, ax, FORWARD)
      end if
      if pos > 0 then
         neighbor[ax]:= pos − 1
         fillParts(neighbor, ax, BACKWARD)
      end if
   end for
end procedure
```

Algorithmus 4.14: **procedure** fill(*idx*)

Dadurch, dass zur Körperranderzeugung stets Knoten auf der untersten Ebene in den Oktalbaum eingefügt werden und das Körperinnere in der gleichen Farbe wie der Körperrand markiert wird, kann es dazu kommen, dass alle Söhne eines Knotens die gleiche Farbe besitzen. Ein Extrembeispiel ist ein achsparalleler Würfel. Um ihn als Oktalbaum zu präsentieren, muss nur die Wurzel als Blattkonten mit der Farbe des Würfels definiert werden. Erzeugt man den Oktalbaum des Würfels, indem man die Würfeloberfläche in den Oktalbaum überträgt und anschließend den Füllalgorithmus startet, erhält man einen Oktalbaum der Tiefe $d_{t_{\max}}$, indem alle Blätter die Würfelfarbe besitzen. Um den Oktalbaum von unnötigen Verfeinerungen zu befreien, kann Algorithmus 4.15 verwendet werden. Dieses Verfahren soll *Kompaktieren* genannt werden. Ausgehend von den untersten Knoten des Baums werden die Söhne eines Vaterknotens entfernt, wenn sie alle gleichfarbige Blätter sind. Der Vaterknoten ist jetzt somit Blatt und erhält die Farbe seiner einstigen Söhne. Das Verfahren wird sukzessiv bis zur Wurzel fortgesetzt.

```
procedure compact(_octree nodeAddr:= getTree())
begin
   if isLeaf(getNode(nodeAddr)) then
      return
   end if
   for PartType i from 0 to 2^dim − 1 step 1 loop
      compact(getChild(nodeAddr, i))
   end for
   Node child:= getNode(isChild(nodeAddr, 0))
   if ¬isLeaf(child) then
      return
   end if
   Color color:= getColor(child)
   for PartType i from 1 to 2^dim − 1 step 1 loop
      child:= getNode(getChild(nodeAddr, i))
      if ¬isLeaf(child) or else getColor(child) ≠ color then
         return
      end if
   end for
   removeChilds(nodeAddr)
   Node node:= getNode(nodeAddr)
   setLeaf(node)
   setColor(node, color)
end procedure
```

Algorithmus 4.15: **procedure** compact(*nodeAddr*)

Klassisches Verfahren

Statt einer der im letzten Abschnitt vorgestellten Alternativalgorithmen zur Oktalbaumgenerierung zu verwenden, kann natürlich auch das klassische Verfahren des rekursiven Abstiegs verwendet werden.

Um eine Zelle bezüglich des Körpers zu lokalisieren, eignet sich wieder die Verwendung der Teststrahlmethode. Für das klassische Verfahren kommt jedoch im Gegensatz zum im Abschnitt *Punkt-in-Polygon-Test* auf Seite 38 erläuterten Vorgehen der Frage, ob sich die Zelle auf der Polyederoberfläche (und somit in einem Polygon) befindet, eine herausragende Bedeutung zu. In locate() (Algorithmus 4.16) wird *atBorder* genau dann auf true gesetzt, falls dies der Fall ist. Der Rest des Algorithmus verhält sich entsprechend dem Teststrahlverfahren.

Der Körper ist als Flächenmodell in **class** CadModel gegeben. Mit Hilfe von first() und next() kann über **class** CadModel iteriert werden.

```
{ Vorbedingung:
   • cadModel enthält das Oberflächenmodell des Körpers.
}
procedure locate( NodeIndex idx,
                  out boolean atBorder, out Color color)
begin
   color:= NO_OBJ_COLOR
   GeomPoint p:= GeomPoint(idx)
   AxIndex dist:= +∞

   cadModel.first()
   while cadModel.hasNext() loop
      Polygon polygon:= cadModel.getObject()
      polygon.setHight(h_p)

      if polygon.isIn(p) then
         atBorder:= true
         color:= cadModel.getObjColor()
         return
      end if

      if n_polygon[2] ≠ 0 then
         GeomPoint pA:= polygon.getPoint()
         Coordinate t:= <pp_A; n_polygon> / n_polygon[2]
         boolean inside:= t * n_polygon[2]
         Geompoint q:= p
         q[2]:= p[2] + t
         if polygon.isIn(q) ∧ t ∈ (0; dist) then
            dist:= t
            color:= { cadModel.getObjColor(),  inside
                    { NO_OBJECT_COLOR,         ¬inside
         end if
      end if
      cadModel.next()
   end for
end procedure
```

Algorithmus 4.16: **procedure** locate(*idx, **out** atBorder, **out** color*)

hasNext() liefert genau dann true, falls noch mindestens ein weiteres Polygon in **class** CadModel vorhanden ist.

Mit Hilfe von getObject() und getObjColor() kann das aktuelle Oberflächenpolygon

```
{ Vorbedingung:
   • octree ist als Oktalbaum, in dem das Körpermodell erzeugt
     werden soll, vorhanden.
}
procedure genClassic(NodeIndex idx:= NodeIndex(0, 0, 0, d_tmax))
begin
   boolean atBorder
   Color color

   locate(idx, atBorder, color)
   if ¬atBorder ∨ h_idx = 0 then
      octree.add(idx, color)
      return
   end if

   for PartType part from 0 to 2^dim − 1 step 1 loop
      genClassic(octree.getChild(idx, part))
   end for
end procedure
```

Algorithmus 4.17: **procedure** genClassic(*idx*)

bzw. die Körperfarbe ausgelesen werden.

Algorithmus 4.17 erzeugt mit Hilfe von Algorithmus 4.16 zur Lokalisation der Zellen durch rekursiven Abstieg den Oktalbaum.

4.2.5 Vergleich der Algorithmen

Neben der klassischen Oktalbaumgenerierung wurde ein alternatives Verfahren erarbeitet. Es unterteilt sich in drei Schritte:

1. Erzeugung der Körperoberfläche (vgl. Abschnitt 4.2.3)
2. Erzeugung des eigentlichen Körpers durch Füllen des Körpers (vgl. Abschnitt *Füllalgorithmen* auf Seite 50)
3. Kompaktieren des Modells (vgl. Algorithmus 4.15)

Über die Scan-Line-Methode werden sehr effizient Drei- und Vierecke erzeugt, welche die Körperoberfläche repräsentieren. Alternativ kann jedes Oberflächen-Polygon über den hierachischen Ansatz mit Hilfe einer isIn()-Methode (vgl. Algorithmus 4.18 und Abschnitt *Winkelverfahren* auf Seite 39) in den Oktalbaum eingefügt werden. Dieser hybride Ansatz ist nicht ganz so effizient, wie die Scan-Line-Methode.

```
function Polygon::isIn(NodeIndex idx) return boolean
begin
   setHight(h_idx)
   if isCorner(idx) then
      return true
   end if
   if ¬isInPlane(idx) then
      return false
   end if
   if isAtBorder(idx) then
      return true
   end if
   GeomPoint q:= getFootpoint(idx)
   Coordinate α:= 0
   for integer i from 0 to getCount() − 1 step 1 loop
      α:= α + arccos ⟨ Qp_i / |Qp_i| ; Qp_{i+1} / |Qp_{i+1}| ⟩
   end for
   return α = 2π
end function
```

Algorithmus 4.18: **function** Polygon::isIn(*idx*) **return** boolean

Allerdings kann der visuelle Eindruck des erzeugten Oktalbaummodells besser sein. So zeigten Schnitte entlang achsparalleler Ebenen der durch Polygone (Drei-/Vierecke) angenäherten Kugel (vgl. Modell `kugel_poly` Abschnitt 5.2.1) mit Verwendung des Scan-Line-Algorithmus nach Triangulierung der Oberfläche ein zahnradähnliches Muster (vgl. Abb. 4.19), während die Schnittbilder bei der Oktalbaumgenerierung durch das hybride Verfahren keine 'Zacken' zeigen. Das durch das Scan-Line-Verfahren erzeugte Oktalbaummodell ist dennoch korrekt.

Abb. 4.19: Zahnradmuster im Schnittbild

Durch unterschiedliche 'Verfahrwege' (Wege entlang der Scan-Line), ob also die Scan-Line von **A** nach **B** oder von **B** nach **A** erzeugt wird, liegen einmal die Punkte direkt unterhalb oder direkt oberhalb der Ideallinie auf der Scan-Line.

Es kann allgemein zu Unterschieden zwischen Oktalbäumen, die aus dem gleichen Oberflächenmodell durch unterschiedliche Verfahren erzeugt wurden, innerhalb der gewählten Toleranz (eine Zelle der Auflösung der untersten Ebene) kommen.

Um das Modell des eigentlichen Körpers (und nicht nur seine Oberfläche) zu ehrhalten, muss nochmal über den gesamten Baum traversiert und der Füll-Algorithmus angewendet werden. Die hierfür benötigte Zeit liegt in der Größenordnung der Oberflächenübertragung. (Hier muss zwar die aufwändige Punktlokalisation durchgeführt

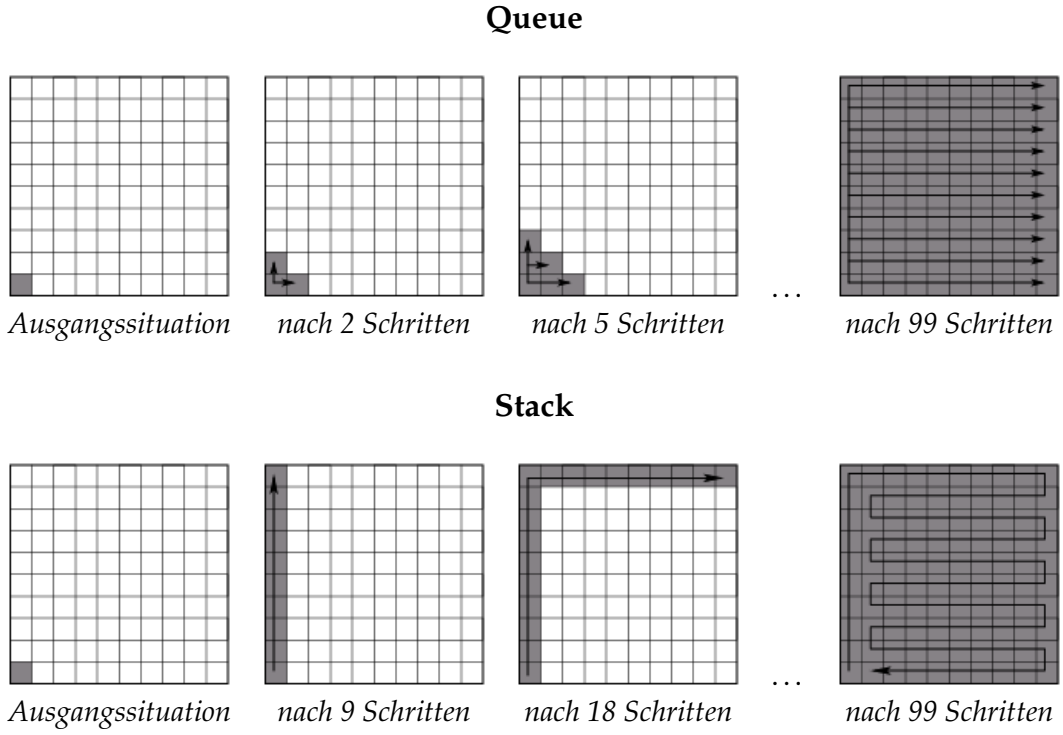


Abbildung 4.20: Gegenüberstellung: Queue- und stackbasiertes Füllverfahren

werden, dies aber nur in vergleichsweise geringer Anzahl (gegenüber dem klassischen Verfahren). Algorithmus 4.14 (mit Algorithmus 4.13) führt jedoch bei maximalen Baumtiefen $d_{t_{\max}} > 6$ schnell zu großen Rekursionstiefen von `fill()`. Deshalb stieg unter SuSE Linux 8.0 (dem Testsystem, vgl. Abschnitt 5.2.2) der vom Programm verwendete Speicher in diesem Schritt erheblich an, obwohl keine neue Knoten in den Oktalbaum eingefügt wurden. Auf anderen Systemen kam es sogar darauf hin zum `Segmentation fault`. Abhilfe schafft eine Begrenzung der maximalen Rekursionstiefe von `fill()` oder das Umwandeln in einen iterativen Algorithmus unter Verwendung einer Daten-Queue. Bei der Stackbegrenzung kann sich ein erreichter Punkt gemerkt werden, an dem das Füllen neu angesetzt wird (**`LIMITED_STACK`**). Es können die Füllgebiet-Randpunkte (Punkte, die genau dann erreicht werden, wenn der Stack bis zur Begrenzung gefüllt ist) auch eine spezielle Markierung erhalten (**`MARK_BORDER`**). Wird bei der Oktalbaumtraversierung ein Punkt mit dieser Markierung erreicht, wird daraus die eigentliche Körperfarbe extrahiert und der Füllalgorithmus neu aufgesetzt. Ursache für die hohe Rekursionstiefe für den Algorithmus ist die durch die Tiefensuche entstehende Wegwahl zum Füllen. Wird stattdessen eine Daten-Queue verwendet , unterliegt der Füllalgorithmus einer Breitensuche (vgl. Abbildung 4.20).

Als letzter Schritt wird der Baum kompaktiert. Die hierfür benötigte Zeit ist wesentlich geringer als die für Schritt 1 und 2 benötigte.

Wird das Körpermodell mit Hilfe des klassischen Verfahrens des rekursiven Abstiegs erzeugt, muss für jeden Knoten die Lage bezüglich des Körpers bestimmt werden. Hierzu wird überprüft, welche Flächen des CAD-Modells (also welche Oberflächenstücke des Körpers) einen Teststrahl schneiden. Um nicht für jeden Knoten das gesamte CAD-Modell durchlaufen zu müssen, kann folgende Optimierung vorgenommen werden: Zur Lagebestimmung eines Sohnknotens werden nur die Flächen herangezogen, die den Teststrahl ausgehend von der Vaterzelle schneiden. Da die im klassischen Verfahren verwendeten Organisationsstrukturen bei den Alternativverfahren teilweise verloren gehen, kann es ratsam sein, in folgenden Szenarien das klassische Verfahren den hier vorgestellten Alternativalgorithmen vorzuziehen:

- falls Parallelisierung angestrebt wird,
- bei Verfeinerung der Oktalbaumstruktur durch nachträgliche Erhöhung der Auflösung (entspricht Vergrößerung der maximalen Baumtiefe).

Zusammenfassend kann festgestellt werden, dass alle Alternativen zur Oktalbaumerzeugung in ihrem Zeit- und Speicheraufwand von der Oberfläche des zu modellierenden Körpers abhängig sind und somit die Komplexität $\mathcal{O}(n^2)$ besitzen. Die höchste Effizenz ist dabei vom Scan-Line-Verfahren zu erwarten, insbesondere bei komplexeren Geometrien (also mit vielen Oberflächenstücken). Der benötigte höhere Speicheraufwand infolge der konsequenten Körperrandauflösung bis zur untersten Ebene hält sich in Grenzen. Es sollte aber dabei auf eine Umsetzung des Füllalgorithmus geachtet werden, bei der die Rekursionstiefe beschränkt ist (oder die Rekursion durch eine entsprechende Datensruktur ersetzt wird, die eine iterative Umsetzung möglich macht).

4.3 Spline-Flächen-Generierung

In den letzten Abschnitten wurden Möglichkeiten der Oktalbaumgenerierung für glatte Körperoberflächen ausführlich beschrieben. Zur Oktalbaumerzeugung mit Spline-Oberflächen wird ein Trick benutzt:

Die Genauigkeit des Oktalbaummodells ist durch das durch die Zellen der untersten Ebene entstehende Gitter begrenzt. Durch (4.13) ist gegeben, wann zwei Punkte als gleich angesehen werden können. Eine Oberfläche, die durch entsprechend kleine Polygone approximiert wurde, wird demnach durch den selben Oktalbaum repräsentiert, wie die ursprüngliche Spline-Oberfläche. Zur Oktalbaumgenerierung wird die Spline-Oberfläche demnach durch ein Polygonnetz ersetzt, welches die Spline-Fläche hinreichend genau annähert.

Um für ein Parameterwertpaar $(u; v)$ den entsprechenden Oberflächenpunkt $\mathbf{x}(u; v)$ zu bestimmen, werden die sehr effizienten Algorithmen 4.21, 4.22 und 4.23 verwendet, die auf den in [Pie97] beschriebenen Methoden *SurfacePoint()*, *FindSpan()* und *BasisFuns()* basieren.

```
{ Vorbedingung: u ∈ [0; max{u_i^* | u_i^* ∈ U}) ∧ v ∈ [0; max{v_i^* | v_i^* ∈ V}) }
function getFacePoint(Coordinate u, Coordinate v)
   return GeomPoint
begin
   array 0..3 of Coordinate N_u
   array 0..3 of Coordinate N_v

   integer u_span:= findSpan(U_DIR, u)
   basisFuns(U_DIR, u_span, u, N_u)
   integer v_span:= findSpan(V_DIR, v)
   basisFuns(V_DIR, v_span, v, N_v)

   GeomPoint x:= (0; 0; 0|0)
   integer u_ind:= u_span - 3
   for integer k from 0 to 3 step 1 loop
      GeomPoint temp:= (0; 0; 0|0)
      integer v_ind:= v_span - 3
      for integer l from 0 to 3 step 1 loop
         temp:= temp + N_u[l] * c_{u_ind+l; v_ind+k}
      end for
      x:= x + N_v[k] * temp
   end for
   return x
end function
```

Algorithmus 4.21: **function** getFacePoint(*u, v*) **return** GeomPoint

extent gibt an, welche der beiden Richtungen der Splinefläche betrachtet werden soll. Als Knotenvektoren dienen U und V. Die zugehörigen (von 0 verschiedenen) Basisfunktionen werden in N_u und N_v gespeichert. $\mathbf{c}_{l;k}$ bezeichnet den $(l;k)$-ten Kontrollpunkt der Spline.

Man beachte, dass die in [Pie97] beschriebenen und in den Algorithmen 4.21, 4.22 und 4.23 verwendeten Knoten u_i^* offener B-Splines aus implementierungstechnischen Gründen einen Indexwert $i \in [0; n+p+1)$ haben. Im Gegensatz dazu hatten die im Abschnitt 2.1.1 eingeführten Spline-Knoten u_{i° Indexwerte $i^\circ \in [-p-1; n)$. Dabei ist n die Anzahl der Kontrollpunkte und p der Grad der Spline-Kurve.

Da die in der Abteilung erstellten DXF-Modelle ausschließlich kubische B-Splines als Spline-Flächen enthalten, wurde das Verfahren auf diese Gruppe von Splines eingeschränkt. (Eine Erweiterung auf beliebige Splines ist jedoch sehr einfach möglich.) Der Knotenvektor $U = \{u_i^*\}$ einer Ausdehnungsrichtung über n Kontrollpunkte ist für eine

```
function findSpan(Extent extent, Coordinate u) return integer
begin
   if isClosed(extent) then
      return ⌊u⌋
   end if

   n:= getControllpointCount(extent)
   if u ≥ n then
      return n
   end if

   return ⌊u⌋ + 3
end function
```

Algorithmus 4.22: **function** findSpan(*extent, u*) **return** integer

offene Spline mit

$$u_i^* = \begin{cases} 0, & i \leq 3 \\ i-3, & i \leq n \\ n-3, & i > n \end{cases} \tag{4.46}$$

($i \in [0; n+p+1)$) und im geschlossenen Fall mit

$$U = \{0, \ldots, n\} \tag{4.47}$$

gegeben. Im geschlossen Fall seien zusätzlich die Kontrollpunkte $c_n = c_0$, $c_{n+1} = c_1$,... und $c_{-1} = c_{n-1}$.

Durch die Polygonnetz-Approximation der B-Spline-Oberfläche entsteht eine große Menge an Objekten, die auf etwaigen Schnitt mit dem Teststrahl zur Lagebestimmung von Zellen (vgl. Abschnitt *Punkt-in-Polygon-Test* auf Seite 38 und Algorithmus 4.16) durchlaufen werden müssen. Für die Lokalisation von Zellen innerhalb des Füllalgorithmus (Abschnitt 4.2.4) ist aber unter Umständen die Verwendung des Kontrollpunktnetzes anstatt des feinen Oberflächenpolygonnetzes ausreichend:

Vor Anwendung des Füllalgorithmus ist bereits der Körperrand in den Oktalbaum übertragen worden. Damit der Füllalgorithmus korrekt arbeitet, müssen die zur Bestimmung der Füllfarbe aufgewählten Punkte korrekt lokalisiert werden. Wir nehmen an, dass das Kontrollpunktnetz nicht selbstschneidend ist (für den modellierten Körper muss dies ohnehin gelten (vgl. Abschnitt 2.1.1). Des Weiteren sollen, falls ein Körper aus mehreren nicht zusammenhängenden Teilen besteht oder mehrere Körper sich innerhalb eines Modells befinden, die dazugehörigen Kontrollpunktnetze einander schnittfrei sein. Dann werden aufgrund der Eigenschaft, dass sich Spline-Oberflächen immer innerhalb ihres Kontrollpunktnetzes befinden, Körperinnenpunkte immer richtig lokalisiert.

```
{ Vorbedingung: u ∈ [getKnot(extent, i); getKnot(extent, i+1)) }
procedure basisFuns(Extent extent, integer i, Coordinate u,
                    out array 0..3 of Coordinate N)
begin
   array 0..3 of Coordinate left
   array 0..3 of Coordinate right

   N[0]:= 1
   for integer k from 1 to 3 step 1 loop
      left[k]:= u - getKnot(extent, i + 1 - k)
      right[k]:= getKnot(extent, i + k) - u
      Coordinate saved:= 0
      for integer r from 0 to k − 1 step 1 loop
         Coordinate temp:= N[r] / (left[k-r] + right[r+1])
         N[r]:= saved + temp*right[r+1]
         saved:= left[k-r] * temp
      end for
      N[k]:= saved
   end for
end procedure
```

Algorithmus 4.23: **procedure** basisFuns(*extent, i, u,* ***out*** *N[]*)

Wurden keine Hohlkörper modelliert (es gibt kein Gebiet, dass vollständig vom Körperrand umschlossen ist und nicht zum Körper gehört), muss jetzt nur noch der Außenbereich richtig bestimmt werden: Beim Füllalgorithmus wird immer der erstmögliche Punkt zur Ermittlung der Füllfarbe verwendet. Befindet sich die Zelle $\lfloor \mathbf{g}_{\mathbf{p}_{\min}} \rfloor$ nicht auf den Körperrand, wird sie zur Füllfarbenermittlung verwendet. Liegt in ihr auch kein Kontrollpunkt, wird der gesamte Außenbereich korrekt mit der Farbe NO_OBJECT_COLOR markiert. Dass die Zelle $\lfloor \mathbf{g}_{\mathbf{p}_{\min}} \rfloor$ stets 'freie' Zelle ist (sie enthält kein Modellierungspunkte, wie z.B. Kontrollpunkte), kann allgemein durch eine entsptechende Verschiebung des Punkts $\mathbf{p}_{\min}$ nach weiter 'außen' (in Richtung $(-\infty; -\infty; -\infty)$) um eine Zelle erreicht werden (vgl. zur Anpassung des Oktalbaumgebiets Abschnitt 4.1.1). Bei höheren Auflösungen müssen somit wesentlich weniger Polygone zur Lokalisation beachtet werden. Die Zahl der zu durchlaufenden Polygone ist – wie bei der Modellierung glatter Körperoberflächen – unabhängig von der Oktalbaumtiefe.

Beim klassischen Verfahren kann die Auflösung des Approxiamtion-Netzes für die Spline-Fläche entsprechend der Knotentiefe adaptiert werden. Es muss also nicht schon bei der Wurzel eine Spline-Extrahierung mit Polygonen der Abmessungen entsprechend der Zellen der untersten Ebene erfolgen.

Kapitel 5

Implementierung

Dieses Kapitel beschäftigt sich mit der innerhalb dieser Arbeit erzeugten Implementierung. Programmaufbau und Programmoptionen werden im Abschnitt 5.1 beschrieben. Ergebnisse bezüglich Zeit- und Speicheraufwand bei der Oktalbaumgenerierung mit dieser Implementierung sind im Abschnitt 5.2 zu lesen.

5.1 Umsetzung

Zunächst soll auf einige implementierungstechnischen Besonderheiten eingegangen werden. Anschließend erfolgt im Abschnitt 5.1.2 eine Beschreibung der erstellten Implementierung.

5.1.1 Einige implementierungstechnische Details

Zur Implementierung wurde die Programmiersprache C++ verwendet. Einschlägige C++-Compiler sind für alle gängigen Betriebsysteme, insbesondere für alle Unix-Derivate wie Linux und alle Windows-Varianten verfügbar. Der Quellcode kann somit auf unterschiedlichen Plattformen übersetzt werden.

In C++ können Programme im objekt-orientierten Programmierstil erstellt werden. In der hier erzeugten Implementierung wurde unter anderem von den Mitteln des Exception Handling, des Polymorphismus und des Overloading und der Unterstützung von Assertions Gebrauch gemacht. C++ lässt die Erzeugung von hochperformantem Code zu. (In dieser Arbeit wurde der `g++`-Compiler mit der Option `-O3` verwendet.)

Wegen ihrer hohen Popularität existieren eine Vielzahl von Veröffentlichungen zu C++. Neben Werken, die eine Übersicht über die Programmiersprache geben (z.B. [Str95]),

```
struct Node;
typedef Node* _octree;

struct Node {
  struct _octree parts;
  Color flag;
};
```

Abbildung 5.1: Implementierte Zeigerstruktur des Oktalbaums

gibt es Abhandlungen zu speziellen Merkmalen von C++, wie die STL[1] (z.B. [Bre99]) oder algorithmische Umsetzungen in C++([Sed94]), die auch in dieser Arbeit verwendet wurden. Zusätzlich finden sich Bibliotheken für konkrete Aufgabenstellungen, die sich einfach verwenden lassen. Ein Beispiel hierfür ist *dime*, eine Bibliothek, die zum Einlesen der Oberflächenmodelle im DXF-Format (vgl. Abschnitt 3.1.2) benutzt wird.

Eine Besonderheit von C++ ist die Verwendbarkeit von sogenannten Präprozessormakros. Sie wurden z.B. für *Optionsschalter* eingesetzt. Soll das Programm mit eingeschalteten Option 'XYZ' compiliert werden, so muss in `global.h` die Zeile

```
#define XYZ
```

stehen. Zum Ausschalten der Option 'XYZ' muss diese vor erneuter Compilierung durch `//` auskommentiert werden. In der Datei `global.h` steht also

```
//#define XYZ
```

anstatt obiger Zeile. Zum Wiedereinschalten wird `//` wieder entfernt. Im ausführbaren Programm sind somit immer nur Methoden zu einer Optionsschalterstellung (ein *oder* aus) enthalten. Optionen, die sich von Programmlauf zu Programmlauf ändern können, werden als Programmparameter beim Programmstart übergeben. Über alle Optionsschalter und Programmparameter gibt Abschnitt 5.1.2 Auskunft.

Zum besseren Verständnis des Quellcodes wurde eine Klassenreferenz mit Hilfe von *doxygen* ([vH02]) generiert.

Wegen ihrer herausragenden Bedeutung soll hier die implementierte Zeigerstruktur des Oktalbaums vorgestellt werden (vgl. Abbildung 5.1):

Ein Knoten besteht aus einer Referenz auf seine Söhne `parts` und einem Status-Attribut `flag`. Für Blätter ist `parts` = `NULL` und `flag` enthält die Knotenfarbe. Für innere Knoten referenziert `parts` die acht Sohnknoten, `flag` ist undefiniert. `parts` wird für innere Knoten dynamisch als Feld erzeugt. Durch das gemeinsame Abspeichern aller Söhne hintereinander und die Nutzung nur einer Referenz

[1] Standard Template Library: Bibliothek, die grundlegende Datenstukturen wie Listen, beliebig lange Felder und Zeichenketten (Strings) oder Ströme (Streams) enthält.

- ist gewährleistet, dass ein Knoten entweder genau acht oder gar kein Sohn hat. Der erzeugte Baum ist somit immer ein Oktalbaum im Sinne der Definition.
- wird weniger Speicherplatz benötigt, da nur einer anstatt acht Zeiger für die acht Söhne eines inneren Knotens benötigt wird.

Diese Datenstruktur wurde bereits in anderen Implementierungen als Oktalbaumstruktur von der Abteilung Simulation großer Systeme eingesetzt und hat sich dort bewährt. Des Weiteren ist gewährleistet, dass die Wurzel eines Oktalbaums immer referenzierbar ist, auch wenn der Oktalbaum kein Geometrie-Modell enthält.

Für jeden Knoten des Oktalbaums werden in den verwendeten Testumgebungen 64 Bytes Speicher alloziert.

5.1.2 Programmbeschreibung

Mit **cad2octree** lässt sich zu einem gegebenen Oberflächen-Modell ein Oktalbaum generieren, der das gleiche Modell repräsentiert.

Anforderungen an das gegebene Oberflächenmodell

Das Oberflächenmodell stellt einen Rigid Body dar. Als Format wird DXF verwendet. Zur Modellierung von glatten Oberflächen wurde die ENTITY `3DFACE`, für Spline-Oberflächen die ENTITY `POLYLINE` benutzt. (Es werden nur `POLYLINE`-Entities vom Oberflächentyp *Cubic B-spline surface* (Group code 75), dem Typ *polygon mesh* und dem Flag *Spline-fit, 3D polygon mesh* (Groupcode 70) ausgewertet, vgl. Abschnitt 3.1 und [aut02, S. 62, 97].)

Erzeugter Oktalbaum

Der generierte Oktalbaum wird in Präorder-Traversierung als Binärstrom geschrieben, vgl. Abschnitt 3.2.

Programmparameter

cad2octree wird durch:

```
cad2octree [-q] [-d <depth>] <input> [-o <output>]
```

aufgerufen. Dabei ist:

-q	die Option, die das Aufpalten von Vierecken in 2 Dreiecke erzwingt.
<depth>	die maximale Baumtiefe
<input>	der Name der Eingabedatei. Das ist die DXF-Datei, die das Oberflächenmodell entält. Sie muss die Erweiterung `dxf` oder `dxb` besitzen.

<output> der Name der Ausgabedatei für den Oktalbaum als Binärstrom in Präordertraversierung. Der Dateiname muss die Endung `pot` haben. Wird nur die Eingabedatei, aber keine Ausgabedatei angegeben, wird der Oktalbaum zur Eingabedatei *xyz*`.dxf` in die Ausgabedatei *xyz*`.pot` geschrieben.

Optionsschalter

Mit *cad2octree* ist es prinzipiell möglich, aus unterschiedlichen Verfahren einen Algorithmus zur Oktalbaumgenerierung auszuwählen. Die Optionsschalter müssen *vor* dem Compilieren von cad2octree festgelegt werden (vgl. Abschnitt 5.1.1). Einige Optionsschalter sind nur in Verbindung mit anderen wirksam. So ist z.B. `ALGORITHM_CHECK_DET` nur wirksam, falls `ALGORITHM_ISIN` eingeschaltet ist und `ALGORITHM_ISIN` hat nur eine Wirkung, wenn `CLASSIC_MODE` ausgeschaltet ist. Diese Abhängigkeiten werden aus der Zuordnung der Optionsschalter in der folgenden Beschreibung ersichtlich. ***ein*** steht für einen eingeschalteten, ***aus*** für einen ausgeschalteten Optionsschalter. Fehlt die Angabe ***ein*** bzw. ***aus***, ist immer der eingeschaltete Optionsschalter gemeint.

`CLASSIC_MODE`

- ***aus*:** Nutze ein Alternativverfahren.

  `FILL_SOLIDS`

  - ***aus*:** Es wird ausschließlich die Körperoberfläche erzeugt.
  - ***ein*:** Nach der Generierung der Körperoberfläche wird ein Füllalgorithmus gestartet und danach kompaktiert.

    `MARK_BORDER:` Der Rand des Füllgebiets wird beim Erreichen der Rekursionstiefe MAX_RECURSIVE_DEEP innerhalb der Funktion `fill()` mit der entsprechenden Randfarbe markiert und der Füllalgorithmus abgebrochen.

    `LIMITED_STACK:` Beim Erreichen der Rekursionstiefe MAX_RECURSIVE_DEEP innerhalb der Funktion `fill()` wird der Füllalgorithmus abgebrochen und auf einem zuvor gemerkten Punkt wieder aufgesetzt.

  `ALGORITHM_ISIN`

  - ***aus*:** Das Scan-Line-Verfahren wird verwendet.

    `COMB_SCAN_LINE`

    * ***aus*:** Als Nachbarpunkte werden nur die Nachbarn entlang der Achsrichtungen angesehen.
    * ***ein*:** Als Nachbarpunkte können auch die Nachbarn entlang der Diagonalrichtungen angesehen werden.

  - ***ein*:** Es wird ein hybrides Verfahren eingesetzt, d.h. der Körperrand wird über den rekursiven Abstieg mit Hilfe des Teststrahlverfahrens auf den Oberflächenpolygonen generiert.

    `ALGORITHM_CHECK_DET`

    * ***aus*:** Ob ein Punkt in der Polygonebene liegt, wird über seinen Abstand zum Fußpunkt bestimmt.
    * ***ein*:** Ob ein Punkt in der Polygonebene liegt, wird mit Hilfe des Determinantenverfahrens bestimmt.

- ***ein*:** Nutze das klassisches Verfahren. Der Oktalbaum wird mit Hilfe des rekursiven Abstiegs erzeugt.

`NDEBUG`

- ***aus*:** Die Überprüfungen durch Assertions sind eingeschaltet. Dies ist während dem Entwicklungsstadium des Programms hilfreich, da dadurch mögliche Fehler in der Implementierung besser eingegrenzt werden können. So wird z.B. vor der Bestimmung der Farbe eines Knotens durch `getColor()` überprüft, ob der Knoten ein Blatt innerhalb des Oktalbaums ist.
- ***ein*:** Es werden keine zusätzlichen Überprüfungen durch *Assertions* durchgeführt, was die Ausführgeschwindigkeit des Programms erhöht.

Unterstützte Plattformen

Das Programmpaket ist in C++ nach dem ANSI C++ - Standard entwickelt worden. Es sollte somit auch mit einem beliebigen ANSI C++ - konformen Compiler übersetzbar und ausführbar sein. Zum Übersetzen bzw. zum Ausführen von cad2octree wird dime benötigt. Wird die Ausführung abgebrochen, da dime nicht gefunden werden kann, obwohl es installiert ist, kann evtl. das Hinzuügen des Verzeichnisses, indem sich die compilierte dime-Bibliothek `libdime.la` befindet, zur Umgebungsvariable `LD_LIBRARY_PATH` helfen.

Compilieren der Sourcen

Als erstes muss *dime* installiert werden (eine Installationsanleitung ist im dime-Programmpaket enthalten). Zum Compilieren können die `Makefile`s verwendet werden, falls *gmake Version 3.79* oder kombatibel installiert ist. Vor dem Aufruf von `make` im `src`-Verzeichnis zum Compilieren der Sourcen müssen die Pfadangaben in `Makefile.incl` (im Projekthauptverzeichnis) angepasst werden:

- `INCLUDES` muss das dime-Include-Verzeichnis enthalten.
- `LDFLAGS` muss das Verzeichnis enthalten, indem die compilierte dime-Bibliothek steht.
- Eventuell müssen noch die Angaben bei `CC`, `LD` und `AR` (entsprechend auch bei `CCFLAGS`, `LDFLAGS` und `ARFLAGS`) angepasst werden, falls ein anderer

Compiler als `g++` verwendet wird, bzw. `ld` nicht als Linker oder `ar` zur Codeerzeugung nicht zur Verfügung stehen.

Alternativ zur Nutzung von `make` kann cad2octree auch direkt unter Angabe aller Programmbibliotheken (einschließlich dime) übersetzt werden.

Verwendete Bibliotheken Dritter

cad2octree verwendet die DXF-Bibliothek **dime** [Coi02]. Es ist die GPL Version 2 zu beachten[2].

5.2 Leistungstest

Wie bereits im letzten Abschnitt erläutert wurde, sind in *cad2octree* verschiedene Verfahren zur Oktalbaumgenerierung implementiert worden. Diese Verfahren wurden mit Hilfe von unterschiedlichen Modellen (vgl. Abschnitt 5.2.1) auf den im Abschnitt 5.2.2 beschriebenen Systemen ausgeführt. Die Ergebnisse (insbesondere Laufzeit- und Speicherbedarf-Vergleiche) sind in Abschnitt 5.2.3 aufgelistet.

5.2.1 Verwendete Modelle

Die verwendeten Oberflächenmodelle lassen sich in zwei Gruppen unterteilen. Während die Modelle der ersten Gruppe nur glatte Oberflächen besitzen, wurden bei den Modellen der zweiten Gruppe auch (oder ausschließlich) kubische B-Spline-Flächen eingesetzt.

Zur Erzeugung der Oberflächenmodelle im DXF-Format wurde ausschließlich Maya Unlimited 4.0 von Alias | Wavefront eingesetzt.

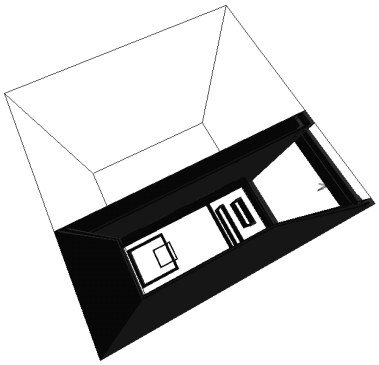
Die (Datei-)Namen der Modelle werden ohne die Erweiterung *.dxf* angegeben. Abkürzend wird für *2_kugeln_nurbs_u_poly.dxf* nur *2_kugeln* geschrieben.

Modelle mit polygonaler Oberfläche

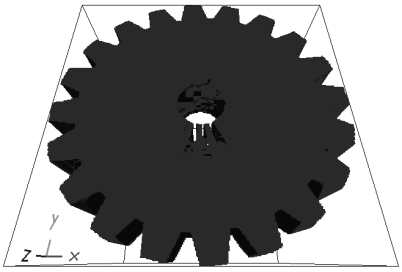
In Tabelle 5.1 sind die verwendeten Modelle mit ausschließlich glatten Oberflächen aufgelistet.

bungalow Modell eines kleinen Gartenshauses. Die einzelnen Bauteile wie Decke, Boden, Wände, Stützpfeiler und Fensterrahmen wurden als separate Teile modelliert.

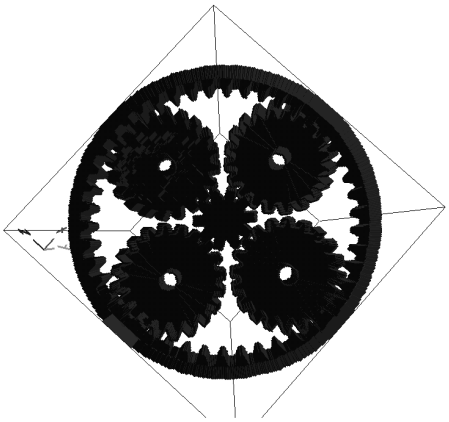
[2] Zu finden unter: `http://www.gnu.org/licenses/gpl.txt`,
Deutsche Übersetzung: `http://www.gnu.de/gpl-ger.html`



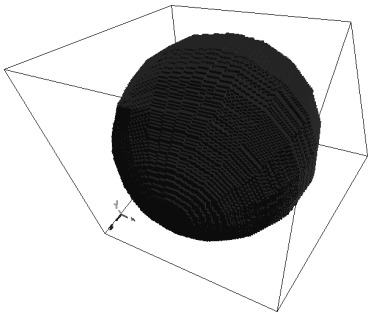
a) *bungalow*



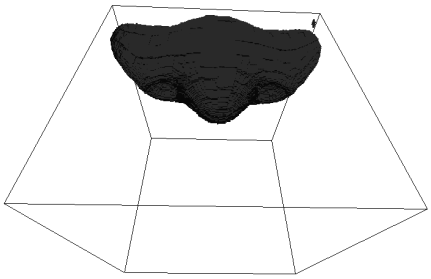
b) *gear*



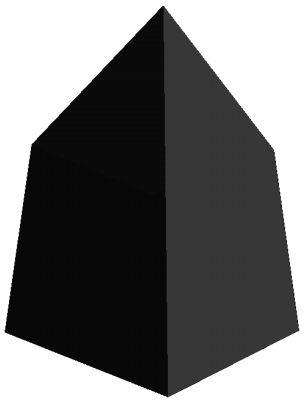
c) *gear2*



d) *kugel_poly*



e) *ship*



f) *wuerfel*

Abbildung 5.2: Modelle mit polygonaler Oberfläche

Modell	**bungalow**	**gear**	**gear2**	**kugel_poly**	**ship**	**wuerfel**
# Vertices	1 200	1 440	9 940	1 560	3 712	24
# Dreiecke	200	0	100	40	0	0
# Vierecke	150	360	2 410	360	928	6
Dateigröße [kB]	104	114	738	104	270	2
Abbildung	5.2 a)	5.2 b)	5.2 c)	5.2 d)	5.2 e)	5.2 f)

Tabelle 5.1: Modelle mit polygonaler Oberfläche

Modell	**2_kugeln**	**gelenk**	**kugel_nurbs**	**sgs_logo**
# Splineflächen	1	13	1	114
# Kontrollpunkte	176	2 273	176	5 168
# Vertices	1 736	2 273	176	5 168
# Dreiecke	40	-	-	-
# Vierecke	360	-	-	-
Dateigröße [kB]	131	445	27	826
Abbildung	5.3 a)	5.3 b)	5.3 c)	5.3 d)

Tabelle 5.2: Modelle mit Spline-Oberfläche

gear Modell eines einzelnen Zahnrads. Das Zahnrad enthält in der Mitte eine Bohrung.

gear2 In der Mitte des Modells befindet sich ein Sonnenritzel (ohne Bohrung), um welches sich innerhalb des Planetenträgers 4 Planeten um das Sonnenritzel drehen. Unter den getesteten Modellen mit rein polygonaler Oberfläche ist es das Komplexeste.

kugel_poly Modell einer durch polygonale Oberflächenstücke approximierten Kugel.

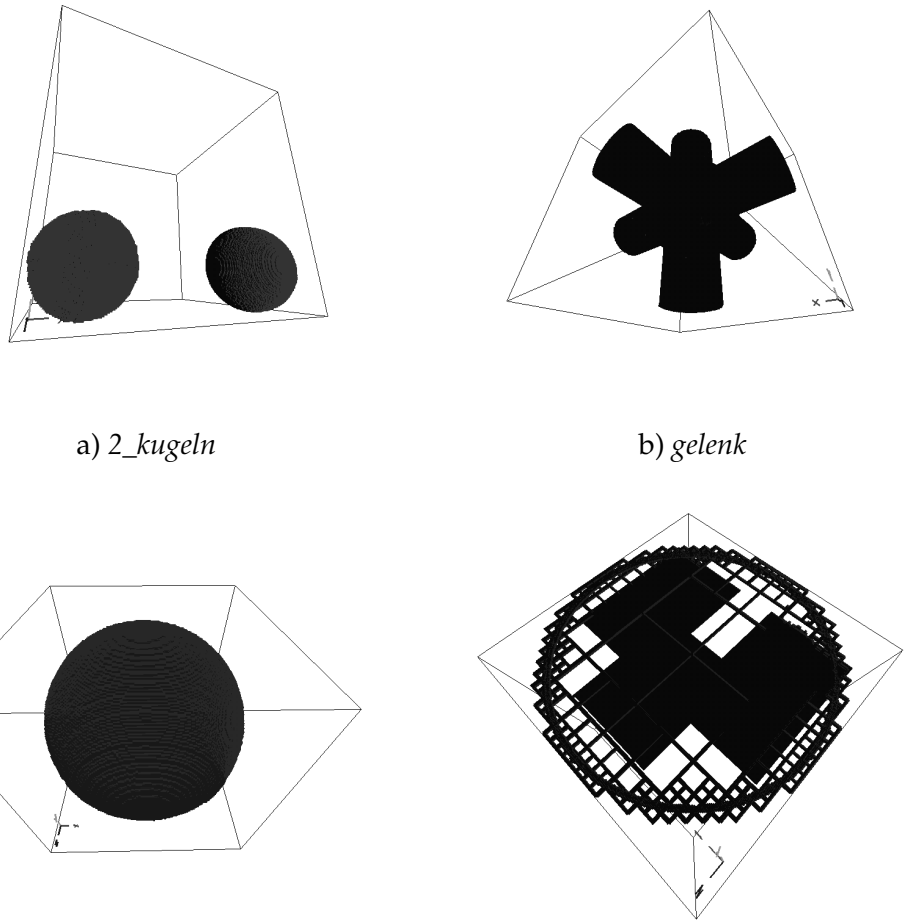
ship Modell eines Raumschiffs. Es enthält Einbuchtungen, ist also nicht vollständig konvex.

wuerfel Achsparalleler Würfel. Das einfachste getestete Modell.

Modelle mit Spline-Oberfläche

Tabelle 5.2 gibt einen Überblick über die verwendeten Modelle, die B-Spline-Oberflächen enthalten.

2_kugeln Modell zweier nebeneinanderliedenen Kugeln. Die eine Kugeloberfläche ist durch Polygone approximiert, die andere Kugel durch eine Spline-Fläche modelliert.



a) *2_kugeln* b) *gelenk*

c) *kugel_nurbs* d) *sgs_logo*

Abbildung 5.3: Modelle mit Spline-Oberfläche

gelenk Das Modell besteht aus sieben Segmenten. Das Mittelsegment ist eine Kugel. In jeder Achsrichtung schließen sich beidseitig Zylinder (Tuben) an. Alle Segmentoberflächen sind abgeschlossen und durch Spline-Flächen modelliert.

kugel_nurbs Eine Kugel, die durch eine Spline-Fläche modelliert ist.

sgs_logo Das Modell zeigt einen dünnen Torus, der sich in einem Oktalbaumgitter (als Rohre) befindet. In der Mitte sind die drei Buchstaben 'SgS' auf dünnen Quadern dargestellt. Alle Element sind über (bis auf die Buchstaben) Spline-Flächen modelliert. Die Buchstaben werden bei der Oktalbaumgenerierung ignoriert (und sind in der Abbildung nicht darbestellt).

5.2.2 Testumgebung

Als Basistestsystem kam SuSE Linux 8.0 auf einem Rechner mit Mobile Pentium (III) 1000MHz CPU, 256 MB RAM und einer 124 MB großen swap-Partition zum Einsatz. Als Compiler wurde g++ Version 2.95.3 verwendet, der als Zielprozessor `i486-suse-linux` gebrauchte.

Als alternatives Testsystem wurde ein Rechner mit Mandrake Linux 8.2, Pentium (II) 350 MHz CPU, 192 MB RAM und 93MB großer swap-Partition genutzt. Als Compiler kam g++ Version 2.96 zum Einsatz, der Code für `i586-mandrake-linux-gnu` erzeugte.

Zur Überprüfung der Korrektheit der erzeugten Oktalbaumstrukturen wurden unterschiedliche Werkzeuge verwendet. Eine zentrale Rolle spielte dabei der Betrachter *glView*. glView ist eine abteilungseigene Entwicklung, die mit Hilfe von OpenGL Oktalbäume im POT-Format (vgl. Abschnitt 3.2) dreidimensional darstellen kann. Zum Zweck der Validierung der erzeugten Geometrie, wurde zusätzlich ein weiterer Export-Filter in cad2octree integriert. Durch diesen Export-Filter können XPM-Dateien[3] erzeugt werden, die achsparallele Schnitte – ähnlich dem bekannten Verfahren bei der Computer-Tomographie – durch den generierten Oktalbaum zeigen.

Zum Visualisieren der DXF-Modelle wurde neben Maya Unlimited 4.0 von Alias|Wavefront, in dem die DXF-Modelle erzeugt wurden, [Ram02], AC3D Version 3.5 von Inivis[4] und Volvo View Express Version 2000-129 für Windows von Autodesk[5] eingesetzt.

5.2.3 Testergebnisse

Die Tests wurden in den im letzten Abschnitt geschilderten Umgebungen durchgeführt. Des Weiteren kann auf die Testdaten von [Jak01, S.53] zurückgegriffen werden, die in Tabelle 5.3 aufgelistet sind. Die Angaben sind in der Form Laufzeit (in Sekunden)/Speicherbedarf (in Tausend kB) dargestellt. Laufzeiten *xx:yy* sind als xx Minuten yy Sekunden zu lesen. Man beachte, dass diese in einer anderen Umgebung mit anderen Modellen erhoben wurden.

Im Folgenden werden die Resultate der eigenen Analysen vorgestellt. Die Laufzeit wurde mit Hilfe der C-Funktionen `setititmer()` mit Option `ITIMERRAL` und `getittimer()` ermittelt, die die abgelaufene Systemzeit messen. Entsprechende Marken, die diese Befehle enthalten, wurden in das Programm eingefügt. Zeitangaben im Format *xx.y* stehen für xx,y Sekunden, *xx:yy* für xx Minuten und yy Sekunden. Zur Messung des Speicherbedarfs von cad2octree wurde `top` verwendet.

[3] X Windows system pixmap file
[4] `http://www.ac3d.org`
[5] `http://usa.autodesk.com/adsk/section/0,,837637-123112,00.html`

Modell	# Dreiecke	Tiefe			
		5	6	7	8
Kugel	528	3/1MB	12/ 1MB	45/ 4MB	3:00/ 17MB
Würfel	24	1/1MB	1/ 2MB	4/ 8MB	16/ 32MB
Tor	498	14/2MB	55/ 8MB	3:35/34MB	15:18/137MB
Kühlturm	1920	1:09/2MB	4:15/ 7MB	20:35/32MB	87:46/141MB

Tabelle 5.3: Laufzeiten [s] / Speicherbedarf zur Oktalbaumgenerierung von [Jak01]

Modell		Tiefe				
		5	6	7	8	9
gear2	s)	3.4/1MB	5.4/2MB	7.0/4MB	8.4/ 7MB	22.1/16MB
	m)	4.0/2MB	11.5/4MB	11.2/8MB	24.9/11MB	1:09/22MB
kugel_poly	s)	5.8/1MB	5.7/2MB	6.1/5MB	13.3/10MB	49.6/32MB
	m)	3.5/1MB	6.5/3MB	10.6/8MB	32.4/12MB	2:24/34MB
ship	s)	3.9/1MB	3.8/2MB	5.1/3MB	7.5/ 7MB	20.2/22MB
	m)	5.5/1MB	5.1/2MB	6.9/6MB	21.6/10MB	1:05/26MB
wuerfel	s)	5.9/1MB	5.6/2MB	6.4/5MB	12.6/12MB	33.6/39MB
	m)	4.0/1MB	5.2/4MB	8.3/7MB	19.2/13MB	1:18/41MB

Tabelle 5.4: Laufzeiten [s] / Speicherbedarf auf s) (= P1000 mit SuSE) und m) (= P350 mit Mandrake) (polygonale Modelle/Scan-Line-Verfahren)

Modelle mit polygonaler Oberfläche

Um die Plattformabhängigkeit der Laufzeit zu analysieren, wurden Testreihen in zwei unterschiedlichen Umgebungen durchgeführt. Beim Speicherbedarf sind in diesem Fall keine Unterschiede zu erwarten, da beide Systeme auf der gleichen Architektur basieren. Die Tabellen 5.4, 5.5 und 5.6 stellen die Ergebnisse vergleichend dar. Mit *s)* wurde die Basis-Umgebung mit SuSE-Linux auf P-1000 bezeichnet, *m)* steht für die Alternativ-Plattform mit Mandrake auf P-350.

Für die Tests in Tabelle 5.4 kam das Scan-Line-Verfahren zum Einsatz, die Ergebnisse in den Tabellen 5.5 und 5.6 basieren auf dem hybriden Anstatz bzw. auf dem klassischen Verfahren. Als Füllmodus wurde `MARK_BORDER` mit einer `MAX_RECURSIVE_DEEP` von 8000 verwendet.

Alle weiteren Analysen beziehen sich auf das P-1000 / SuSE-Linux System.

In Abbildung 5.4 sind die Laufzeiten des Scan-Line-Verfahrens sowie der Speicherverbrauch des Scan-Line und des hybriden Verfahrens dargestellt.

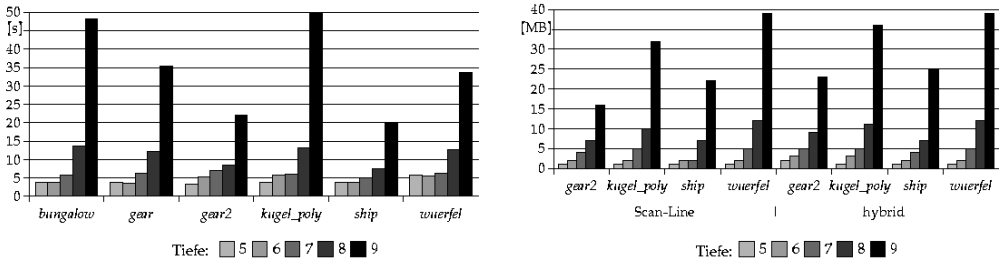
Tabelle 5.7 zeigt wieviele Blätter und Knoten insgesamt ein durch das Scan-Line-Verfahren erzeugter Oktalbaum eines bestimmten Modells enthält. Die Werte in Klammern geben die jeweilige Knotenanzahl nach der Kompaktierung an, die Werte da-

Modell		Tiefe				
		5	**6**	**7**	**8**	**9**
gear2	s)	14.6/2MB	25.2/3MB	41.6/5MB	1:17/ 9MB	3:04/23MB
	m)	29.4/2MB	53.0/4MB	1:22/6MB	2:47/10MB	6:42/25MB
kugel_poly	s)	5.0/1MB	9.9/3MB	18.4/5MB	52.2/ 9MB	2:46/36MB
	m)	11.7/1MB	16.8/2MB	37.8/7MB	1:46/13MB	5:56/38MB
ship	s)	7.2/1MB	10.4/2MB	18.1/4MB	33.7/ 7MB	1:26/25MB
	m)	13.8/1MB	21.6/2MB	32.1/5MB	1:07/ 9MB	3:04/27MB
wuerfel	s)	3.3/1MB	3.8/2MB	13.5/5MB	39.5/12MB	2:22/39MB
	m)	4.9/1MB	7.4/2MB	32.1/5MB	1:16/14MB	5:08/41MB

Tabelle 5.5: Laufzeiten [s] / Speicherbedarf auf s) (= P1000 mit SuSE) und m) (= P350 mit Mandrake) (polygonale Modelle/Hybrid-Verfahren)

Modell		Tiefe				
		5	**6**	**7**	**8**	**9**
wuerfel	s)	2.0/1MB	9.3/2MB	36.3/3MB	2:30/10MB	9:58/38MB
	m)	5.8/1MB	20.3/2MB	1:20/3MB	5:27/10MB	22:05/38MB

Tabelle 5.6: Laufzeiten [s] / Speicherbedarf auf s) (= P1000 mit SuSE) und m) (= P350 mit Mandrake) (*wuerfel*/Klassisches Verfahren)



Laufzeiten Scan-Line *Speicherverbrauch*

Abbildung 5.4: Laufzeiten, Speicherverbrauch (polygon.)

vor entsprechend vor der Kompaktierung. In der Spalte #*Randknoten* steht die Anzahl der Knoten, die die Körperoberfläche wiedergeben. *d* gibt die Maximaltiefe des Oktalbaums an. Die Spalte #*Normzellen* enthält die Anzahl der Zellen (auf Knotenhöhe 0), die das Modell enthalten. Hieraus kann abgeleitet werden, wieviele Zellen in einem entsprechenden Normzellen-Aufzählungsschema die Geometrie tragen (#*Normzellen* korreliert somit mit der Generierungszeit eines solchen Normzellenaufzählungsschemas).

Die Zusammenhänge zwischen maximaler Baumtiefe (und damit der Auflösung) und Knoten- bzw. Zallanzahl sind in Abbildung 5.5 für das *kugel_poly*-Modell und für das *ship*-Modell dargestellt. Abbildung 5.6 zeigt mögliche Abhängigkeiten zwischen Lauf-

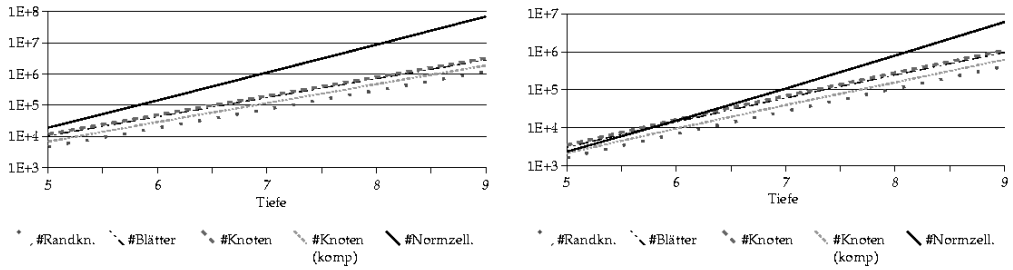
Modell	d	# Blätter	(kompaktiert)	# Knoten	(kompaktiert)	# Rand-knoten	# Norm-zellen
bungalow	5	5 797	(5 713)	6 625	(6 529)	2 440	2 440
	6	22 877	(8 401)	26 145	(9 601)	17 694	17 814
	7	146 735	(113 534)	167 697	(129 753)	71 377	90 552
	8	646 374	(471 962)	738 713	(539 385)	285 670	518 278
	9	2 646 064	(1 906 556)	3 024 073	(2 178 921)	1 141 322	3 323 782
gear	5	4 831	(1 786)	5 521	(2 041)	2 464	4 744
	6	22 002	(21 862)	25 145	(24 985)	9 830	9 830
	7	90 322	(89 706)	103 225	(102 521)	39 753	39 753
	8	366 052	(365 170)	418 345	(417 337)	157 501	157 501
	9	1 465 682	(1 464 660)	1 675 065	(1 673 897)	631 725	631 725
gear2	5	3 452	(2 248)	3 945	(2 569)	2 032	2 260
	6	17 676	(10 760)	20 201	(12 297)	9 368	13 526
	7	83 056	(49 162)	94 921	(56 185)	38 823	87 452
	8	351 527	(284 341)	401 745	(324 961)	155 097	285 807
	9	1 427 056	(1 163 422)	1 630 921	(1 329 625)	611 128	1 787 403
kugel_poly	5	10 438	(5 972)	11 929	(6 825)	4 664	19 108
	6	42 995	(25 334)	49 137	(28 953)	18 647	142 010
	7	175 547	(102 607)	200 625	(117 265)	74 851	1 114 576
	8	707 757	(413 946)	808 625	(473 081)	299 128	8 763 306
	9	2 833 034	(1 651 728)	3 237 753	(1 887 689)	1 199 874	69 407 440
ship	5	3 088	(1 898)	3 529	(2 169)	1 635	2 350
	6	14 337	(8 359)	16 385	(9 553)	6 961	15 316
	7	60 873	(34 994)	69 569	(39 993)	27 683	107 257
	8	246 639	(137 460)	281 873	(157 097)	107 817	800 866
	9	971 328	(552 133)	1 110 089	(631 009)	418 355	6 178 095
wuerfel	5	11 992	(1)	13 705	(1)	5 768	32 768
	6	52 368	(1)	58 849	(1)	23 816	262 144
	7	219 880	(1)	250 377	(1)	96 776	2 097 152
	8	896 512	(1)	1 024 585	(1)	390 152	16 777 216
	9	3 627 576	(1)	4 145 801	(1)	1 566 728	134 217 728

Tabelle 5.7: Knoten- und Normzellenanzahl für polygonale Modelle

zeit und Randknoten bzw. Polygonanzahl.

Die Tabellen 5.8 und 5.9 zeigen die unterschiedlichen Laufzeiten zur Generierung von *wuerfel* bzw. *ship* bei den Baumtiefen 6 und 8 bei der Nutzung verschiedener Algorithmen.

In Abbildung 5.7 sind die Laufzeitunterschiede zwischen den verschiedenen Generierungsverfahren grafisch dargestellt.



kugel_poly *ship*

Abbildung 5.5: Knoten- und Normzellenanzahl (polygon.)

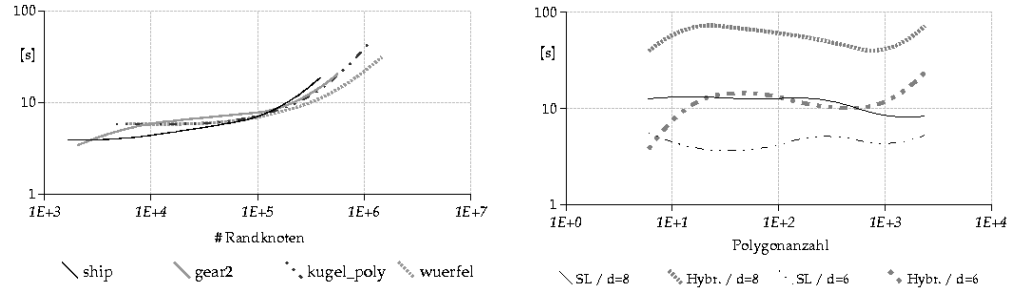


Abbildung 5.6: Laufzeit - Randknoten-/Polygonanzahl - Diagramm (polygon.)

Tiefe	**6**				**8**			
Std. Scan-Line	5.4	(1.9 /	1.7 /	2.0)	12.6	(2.8 /	8.0 /	1.8)
Limited Stack	5.4	(1.9 /	1.7 /	2.0)	10.1	(2.9 /	5.4 /	1.8)
Comb. Scan-Line	5.4	(1.9 /	1.7 /	2.0)	11.7	(3.8 /	6.1 /	1.8)
Hybrid	3.8	(2.1 /	1.7 /	0.0)	39.5	(29.6 /	8.0 /	1.9)
Check-Det	6.0	(2.3 /	1.7 /	2.0)	37.9	(29.9 /	6.1 /	1.9)
Klassisch	9.3				2:30			
Debug	3.6	(2.0 /	1.6 /	0.0)	14.0	(2.7 /	9.5 /	1.8)

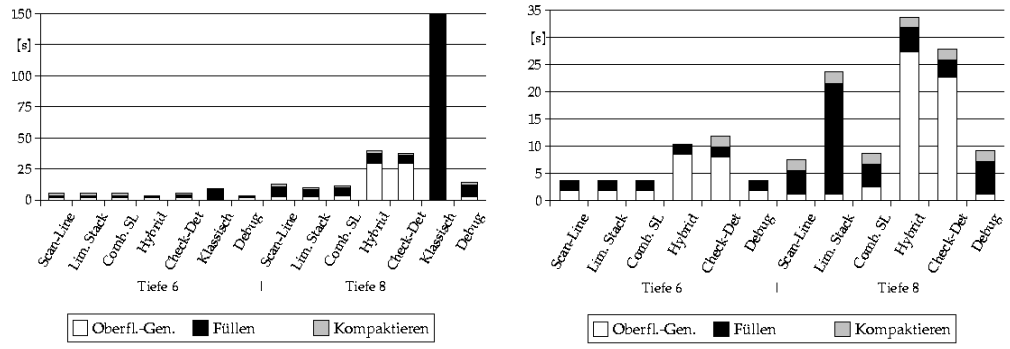
Tabelle 5.8: Laufzeiten [s] unterschiedlicher Algorithmen (*wuerfel*)

Modelle mit Spline-Oberfläche

Zur Oktalbaumgenerierung aus Spline-Flächen-Modellen, werden die Spline-Flächen durch Polygone, die nicht größer als die Randflächen der Zellen (auf Knotenhöhe 0) sind, approximiert. Diese Polygone werden zum Spline-Modell hinzugefügt. Dieser Vorgang wird als *Spline-Extrahierung* bezeichnet. Im weiteren Verlauf der Oktalbaumgenerierung werden die extrahierten Polygone an Stelle der Spline genutzt. Tabelle 5.10 listet auf, wieviele Polygone ein Spline-Modell nach der Extrahierung für eine bestimmte Maximalbaumtiefe besitzt.

Tiefe	6				8			
Std. Scan-Line	3.8	(1.9 /	1.9 /	0.0)	7.5	(1.3 /	4.2 /	2.0)
Limited Stack	3.8	(1.9 /	1.9 /	0.0)	23.6	(1.3 /	20.3 /	2.0)
Comb. Scan-Line	3.8	(1.9 /	1.9 /	0.0)	8.8	(2.6 /	4.2 /	2.0)
Hybrid	10.4	(8.5 /	1.9 /	0.0)	33.7	(27.3 /	4.5 /	1.9)
Check-Det	11.9	(8.0 /	1.9 /	2.0)	27.8	(22.6 /	3.2 /	2.0)
Debug	3.7	(1.9 /	1.8 /	0.0)	9.3	(1.3 /	6.0 /	2.0)

Tabelle 5.9: Laufzeiten [s] unterschiedlicher Algorithmen (*ship*)



wuerfel *ship*

Abbildung 5.7: Laufzeitabhängigkeit von der Verfahrenswahl (polygon.)

Modell	**Tiefe**				
	5	**6**	**7**	**8**	**9**
2_kugeln	642	1 394	4 434	16 658	65 682
gelenk	1 322	5 562	22 874	92 826	374 042
kugel_nurbs	1 986	8 066	32 514	130 562	523 266
sgs_logo	998	2 390	6 710	27 080	109 484

Tabelle 5.10: Anzahl der Polygone eines Spline-Modells nach Extrahierung

Die Messungen wurden analog zu den Modellen mit polygonaler Oberfläche durchgeführt. Die entsprechenden Ergebnisse sind in den Tabellen 5.11, 5.12, 5.13 und 5.14 aufgeführt und in den Abbildungen 5.8, 5.9 und 5.10 grafisch dargestellt.

Auswertung

Bevor aus den erhobenen Werten Schlussfolgerungen abgeleitet werden können, müssen mögliche Wertverfälschungen betrachtet werden:

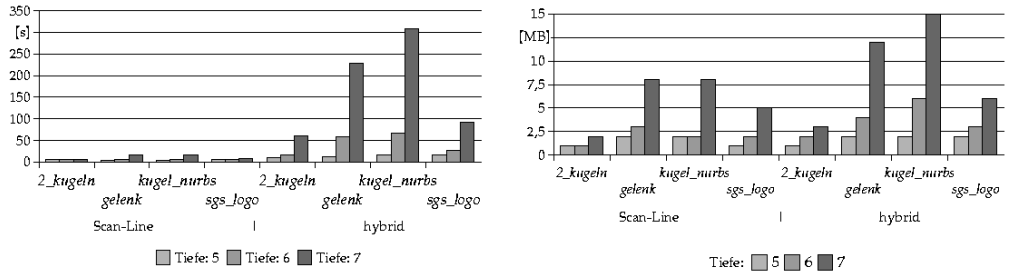
Speicherbedarf Hier ist nicht der reine Speicherbedarf für die Geometrie, sondern für das gesamte Programm cad2octree dargestellt worden. Der zugeteilte Halden-

Modell		Tiefe 5	Tiefe 6	Tiefe 7
gelenk	s)	5.1 / 2MB	5.9 / 3MB	15.8 / 8MB
	m)	4.1 / 1MB	12.6 / 3MB	59.2 / 10MB
kugel_nurbs	s)	4.7 / 2MB	5.5 / 2MB	17.8 / 8MB
	m)	4.2 / 1MB	13.6 / 5MB	4:14 / 11MB

Tabelle 5.11: Laufzeiten [s] / Speicherbedarf auf s) (= P1000 mit SuSE) und m) (= P350 mit Mandrake) (Spline-Modelle/Scan-Line-Verfahren)

Modell		Tiefe 5	Tiefe 6	Tiefe 7
gelenk	s)	12.5 / 2MB	58.6 / 4MB	3:49 / 12MB
	m)	25.3 / 2MB	1:47 / 6MB	8:18 / 14MB
kugel_nurbs	s)	17.3 / 2MB	1:08 / 6MB	5:09 / 15MB
	m)	32.4 / 2MB	2:21 / 8MB	11:36 / 17MB

Tabelle 5.12: Laufzeiten [s] / Speicherbedarf auf s) (= P1000 mit SuSE) und m) (= P350 mit Mandrake) (Spline-Modelle/Hybrid-Verfahren)



Laufzeiten Scan-Line *Speicherverbrauch*

Abbildung 5.8: Laufzeiten, Speicherverbrauch (Spline)

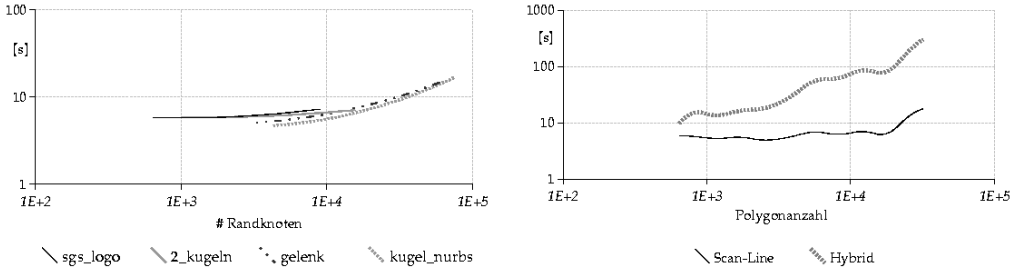


Abbildung 5.9: Laufzeit - Randknoten-/Polygonanzahl - Diagramm (Spline)

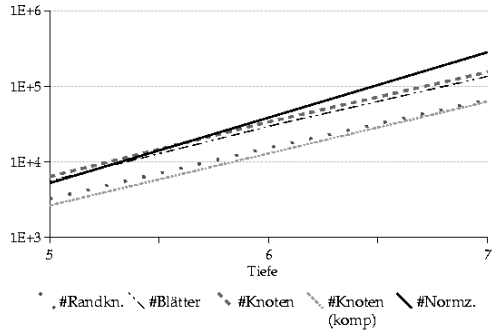
speicher durch das Betriebssystem kann größer sein, als es das Programm benö-

Modell	d	# Blätter	(kompaktiert)	# Knoten	(kompaktiert)	# Rand-knoten	# Norm-zellen
2_kugeln	5	2 255 (	1 520)	2 577 (	1 737)	994	1 634
	6	9 199 (	5 860)	10 513 (	6 697)	3 963	11 007
	7	38 039 (	22 268)	43 473 (	25 449)	16 802	80 945
gelenk	5	5 608 (	2 332)	6 409 (	2 665)	3 269	5 275
	6	29 632 (	11 376)	33 865 (	13 001)	15 387	38 493
	7	134 996 (	55 049)	154 281 (	62 913)	66 531	281 132
kugel_nurbs	5	9 479 (	5 307)	10 833 (	6 065)	4 290	14 530
	6	39 922 (	20 931)	45 625 (	23 921)	19 089	109 735
	7	165 222 (	86 045)	188 825 (	98 337)	80 756	841 690
sgs_logo	5	2 031 (	2 031)	2 321 (	2 321)	642	642
	6	6 840 (	6 840)	7 817 (	7 817)	2 417	2 417
	7	25 376 (	24 956)	29 001 (	28 521)	9 628	9 628

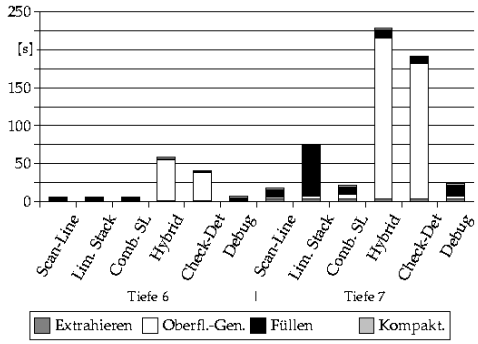
Tabelle 5.13: Knoten- und Normzellenanzahl für Spline-Modelle

Tiefe	**6**		**7**	
Std. Scan-Line	5.9	(1.4 / 1.7 / 2.8 / 0.0)	17.8	(3.8 / 2.2 / 9.8 / 2.0)
Limited Stack	5.9	(1.5 / 1.6 / 2.8 / 0.0)	1:15	(3.8 / 3.8 / 1:05 / 2.0)
Comb. Scan-Line	5.7	(1.5 / 1.3 / 2.9 / 0.0)	21.5	(3.8 / 5.8 / 9.9 / 2.0)
Hybrid	56.6	(1.5 / 54.0 / 1.1 / 2.0)	3:49	(3.8 / 3:31 / 11.7 / 2.0)
Check-Det	41.2	(1.5 / 37.0 / 2.7 / 0.0)	3:12	(3.8 / 2:58 / 8.1 / 2.0)
Debug	7.0	(1.4 / 1.5 / 2.1 / 2.0)	23.8	(3.6 / 3.6 / 14.6 / 2.0)

Tabelle 5.14: Laufzeiten [s] unterschiedlicher Algorithmen (*gelenk*)



#Knoten/Normzellen



Laufzeiten untersch. Verfahren

Abbildung 5.10: Knoten-/Zellenanzahl und Laufzeiten (*gelenk*)

tigt. Vom Programm nicht mehr verwendeter Speicher kann erst später wieder deallociert werden. Der Speicherbedarf ist zyklisch in Abstand einer Sekunde er-

mittelt worden. Als Speicherbedarf wird dabei der Maximalwert verwendet. Für die Meßwerte sind deshalb Abweichungen bis zu 1MB bzw. 10% einzurechnen.

Laufzeit Einerseits werden durch das Betriebssystem und die Hardware Optimierungen wie Caching eingesetzt. Andererseits können durch unterschiedliche Resourcen-Bereitstellungen des Betriebsystems Abweichungen auftreten, die mit 2 Sekunden bzw. 10% abzuschätzen sind.

Die scheinbar linearen Zusammenhänge zwischen maximaler Baumtiefe und Speicherbedarf und Laufzeit bei 'kleinen' Werten sind deshalb irreführend. Aussagekräftiger sind Speicherwerte von > 10MB und Zeiten von >10s. Ohnehin sind für Aufwandsabschätzungen 'große' Werte relevanter.

Allgemein ergibt sich eine Korrelation zwischen Speicherbedarf und Laufzeit (falls das gleiche Modell bei gleicher Maximalbaumtiefe auf gleicher Plattform und das gleiche Generierungsverfahren betrachtet wird). Die Unterschiede zwischen verbrauchtem Speicher bei unterschiedlichen Verfahren sind nur gering. Im Allgemeinen liefert das Scan-Line-Verfahren die besten Laufzeitergebnisse (Die meiste Zeit wird zum Füllen, beim Hybridverfahren hingegen zur Oberflächengenerierung benötigt).

Betrachtet man die Anzahl der benötigten Baum-Knoten, soll lassen sich für die hier verwendeten Modelle folgende Erkenntnisse ableiten: Die Anzahl der Baumknoten steigt um das maximal Vierfache, falls die maximale Baumtiefe um 1 erhöht wird. Hingegen verachtfacht sich die Anzahl der notwendigen Zellen in einem entsprechenden Normzellenaufzählungsschema. Die Zahl der inneren Knoten ist auf maximal ca. 20% der Gesamtknotenanzahl des Oktalbaums beschränkt. Im Allgemeinen kann durch Kompaktierung die Knotenzahl im Oktalbaum um 10% — 40% reduziert werden. Knotenanzahl und Speicherbedarf sind linear voneinander abhängig (nämlich mit der Konstante Speicherbedarf pro Knoten).

Für polygonale Modelle ergibt sich mit der Erhöhung der Maximalbaumtiefe um 1 maximal eine Vervierfachung der Laufzeit. Bei Spline-Modellen ist hingegen auch mehr als eine Vervierfachung möglich, da jetzt die Polygonzahl von der Baumtiefe abhängig ist (sie vervierfacht sich).

Kapitel 6

Fazit

Wie [Fra00] zeigt können Oktalbaumstrukturen als integrierendes Element zwischen CAD-Programmen zur Geometriemodellierung und Simulation genutzt werden. Ziel dieser Arbeit war die Erzeugung und Evaluierung von Oktalbaumstrukturen als Schnittstelle zu CAD-Programmen. Im Mittelpunkt stand dabei die Generierung des Oktalbaums aus einem geeignenten weitverbreiteten frei zugänglichen Oberflächenformat, in welches in der Abteilung befindliche Oberflächenmodelle leicht exportiert werden können. Neben der Unterstützung von triangulierten Modellen sollten auch Spline-Flächen verarbeitet werden können. Die Integrität des Oberflächenmodells soll durch den Erzeuger des Modells gewährleistet sein, eine Überprüfung innerhalb der in dieser Arbeit erstellten Implementierung ist somit nicht notwendig.

Es existieren bereits einige Lösungen, die in diese Richtung gehen und beachtenswerte Ergebnisse erzielen, wie [Jak01] oder [Mun02]. Diese Lösungen zeigen jedoch jeweils einige der folgenden Unzulänglichkeiten:

- Die Lösung ist nicht frei zugänglich. Neben dem Code oder der Architektur betrifft das z.T. die Lösungsidee oder die verwendeten Algorithmen.
- Es kann der Oktalbaum nur aus einem Teil der geforderten möglichen Oberflächenmodelle generiert werden, z.B. nur aus Polygonflächenmodellen nicht aber für Körper mit Freiformflächen als Oberfläche.
- Als Importformat für die Oberflächenmodelle wird ein 'Spezialformat' verwendet, also keine weitverbreiteten Formate, deren Struktur frei zugänglich dokumentiert ist. Oberflächenmodelle in üblichen Formaten lassen sich dadurch nicht einfach in dieses Format exportieren, wodurch nicht die einschlägigen Werkzeuge (CAD-Programme) zur Erzeugung der Geometriemodelle verwendet werden können.
- Für die Generierung der Oktalbaumstruktur aus dem Oberflächenmodell werden Methoden und Makros eines Programms verwendet, dass zur Erstellung und

Bearbeitung von Oberflächenmodellen dient. Somit wird zur Oktalbaumgenerierung stets das (nicht frei verfügbare) Modellierungswerkzeug benötigt. Der Umfang von einsetzbaren Datenstrukturen und verwendbaren Methoden ist stark durch den Umfang der durch das Modellierungswerkzeug bereitgestellter Bibliotheken korelliert, woraus sich auch weitere Erschwernisse bezüglich Erweiterbarkeit (insbesondere für andere Formate) und Optimierbarkeit ergeben.

Durch die im Rahmen dieser Diplomarbeit erstellte Implementierung werden die Anforderungen erfüllt. Sie stellt somit eine erfolgreiche Umsetzung der Ziele dar und konnte damit die Nutzbarkeit von Oktalbaumstrukturen als Schnittstelle zu CAD-Programmen bestätigen. Dabei wurde u.a. bewusst ein anderer Weg gegenüber dem klassischen Ansatz zur Oktalbaumerzeugung beschritten. Üblicher Weise wird sukzessive die Lage des durch den aktuellen Oktalbaumknoten repräsentierten Würfels bezüglich dem zu erzeugenden Körper analysiert (in/out/on) und entsprechend unter Ausnutzung hierachischen Struktur des Oktalbaums verfeinert. Bei der hier erarbeiten Implementierung wird ein Index für jeden Oktalbaumknoten definiert. Das Schema erlaubt mit Hilfe der Indizes einen zum Normzellenschema äquivalenten Zugriff. Nun können wie bei einem Normzellenschema die Oberflächenwürfel des Körpers in den Oktalbaum eingefügt werden, anschließend werden die Innen- und Außengebiete durch ein spezielles Verfahren gefüllt. Nach einer letztlichen Kompaktierung liegt der gewünschte Oktalbaum vor. Die durchgeführten Messungen bis zu einer Baumtiefe von 9 zeigen: Die durch dieses Vorgehen erzielten Generierungsgeschwindigkeiten zeigen sehr gute Resultate. Durch die verwendete kompakte Datenstruktur ist nur ein relativ geringer Speicherplatz zur Generierung nötig.

Wird die Baumtiefe um 1 erhöht, steigen Laufzeit und Speicherbedarf um den Faktor 4. Die gewählte Programmarchitektur ermöglicht das einfache Erweitern des Programms zur Integration weiterer Formate und das Hinzufügen alternativer bzw. das Erweitern vorhandener Algorithmen. Hiermit lassen sich Werte auch für größere maximale Baumtiefen abschätzen. Das Programm bietet jedoch keine Möglichkeit bereits aus einem CAD-Modell erzeugte Oktalbäume unter Nutzung des CAD-Modells weiter zu verfeinern.

Daraus lassen sich die Verbesserungsmöglichkeiten ableiten:

- Der erarbeitete Oktalbaumgenerator ist für den Ein-Prozessor-Betrieb ausgelegt. Der 'klassische Algorithmus' zur Oktalbaumgenerierung nutzt die Organisationsprinzipien *Hierachie*, *Rekursion* und *Adaptivität* aus und lässt sich leicht in einen effizienten Algorithmus für Mehrprozessorbetrieb umwandeln. Das hiervorgestellte Verfahren umgeht hingegen an einigen relevanten Stellen diese Organisationsprinzipien. So wird das Füll-Verfahren auf dem unterliegenden virtuellen Gitter durchgeführt, was eine effiziente Parallelisierung erschwert.
- Eine nützliche Erweiterung wäre die Oktalbaumgenerierung unter Nutzung mehrerer Quellen, also der Import von Teilgeometrien aus unterschiedlichen Dateien. Ein mögliches Vorgehen hierfür wäre jeweils gesondert einen neuen Ok-

talbaum für jede Datenquelle zu generieren und diesen mit den vorhandenen zu vereinigen. Es lassen sich einfach effiziente Algorithmen für Mengenoperationen implementieren. Besondere Aufmerksamkeit verdient dabei jedoch die Kollisionsauflösung. Die zentrale Frage dabei ist, wie reagiert werden soll, wenn sich in der gleichen Zelle unterschiedliche Körper befinden. Dabei müssen sich die realen Körper im Gegensatz zum diskretisierten Oktalbaummodell gar nicht überschneiden.

- Neben Optimierungen der Ausführungseffizenz des Programmcodes wäre die Unterstützung weiterer Formate eine sinnvolle Erweiterung. Die Architektur des Programms unterstützt das Hinzufügen weiterer Import-/Exportformate. Es könnte über den Einbau einer Importfunktion für ein CSG-Schema nachgedacht werden: Mengenoperationen auf Oktalbäumen sowie die Oktalbaumgenerierung für Grundprimitive lassen sich sehr effizient realisieren. Da die Primitivenanzahl begrenzt und ihre Gestalt im Voraus bekannt sind, können zur Effizienzsteigerung relevante Daten zur Oktalbaumgenerierung sogar in einer Tabelle abgelegt werden.[1] Für die Transformationsoperationen sind effiziente Verfahren bekannt [2] .
- Da die Diskretisierung, die bei der Oktalbaumgenerierung aus stetigen Modellen erfolgt, mit einem Informationsverlust einhergeht, ist eine Rückgewinnung der Ausgangsgeometrie aus der Oktalbaumstruktur hingegen schwierig. Die Oktalbaumstruktur müsste hierfür um zusätzliche Attribute erweitert werden. Hieraus würde sich direkt ein Verlust der Strukturiertheit ergeben. Bei jeder Modifikation des Oktalbaums müssten die Attribute mit überarbeitet werden, um die Konsistenz der Oktalbaumstruktur zu erhalten. Prämissen bei Effizenz, Modifikationsmöglichkeiten und Genauigkeit der rückgewonnen Geometrie wären unvermeidbar.

Dennoch zeigt sich die Eignung der Oktalbaumstruktur als Schnittstelle zur geometrischen Modellierung.

[1] Ähnliche Verfahren werden zur Quadratwurzelberechnung von Fließkommazahlen verwendet.
[2] [Fra00] verweist in diesem Zusammenhang auf einen Übersichtsartikel von Chen und Huang.

Literaturverzeichnis

[Abr91] ABRAMOWSKI, S.; MÜLLER, H.: *Geometrisches Modellieren*. BI, 1991.

[Aum93] AUMANN, G.; SPITZMÜLLER, K.: *Computerorientierte Geometrie*. BI, 1993.

[aut02] AUTODESK: *DXF Reference Guide*, AutoCAD 2002 Auflage, 2002. `http://www.autodesk.com/techpubs/autocad/dxf/`.

[Bar97] BARRERO, D.: *libdxf v0.7*, Juli 1997. `http://members.tripod.com/dbarrero/software/libdxf.tar.gz`.

[Bor97] BORN, G.: *Referenzhandbuch Dateiformate*. Addison-Wesley-Longman, 1997.

[Brü01] BRÜDERLIN, B.; MEIER, A.: *Computergrafik und geometrisches Modellieren*. Teubner, 2001.

[Bre99] BREYMANN, U.: *Komponenten entwerfen mit der C++ STL*. Addison-Wesley, 1999.

[Bun02a] BUNGARTZ, H.-J.: *Vorlesungsunterlagen zu 'Grundlagen der Modellbildung und Simulation*, 2002. `http://www.informatik.uni-stuttgart.de/ipvr/sgs/lehre/vorlesungen/modell_und_simulation_ws02.html`.

[Bun02b] BUNGARTZ, H.-J.; GRIEBEL, M.; ZENGER, CHR.: *Einführung in die Computergraphik*. Vieweg, 2002.

[Coi02] COIN3D: *Dime - DXF Import, Manipulation, and Export library v0.9.1*, Oktober 2002. `http://www.coin3d.org/Dime/about.php`.

[dB87] BOOR, C. DE: *A Practical Guide to Splines*. Springer, 4. Auflage, 1987.

[dB90] BOOR, C. DE: *Splinefunktionen*. Birkhäuser, 1990.

[Far94] FARIN, G.: *Kurven und Flächen im Computer Aided Geometric Design*. Vieweg, 1994.

[Fra00] FRANK, A. CHR.: *Organisationsprinzipien zur Integration von geometrischer Modellierung, numerischer Simulation und Visualisierung*. Doktorarbeit, TU München, Institut f. Informatik, 2000. Utz-Verlag.

[Jak01] JAKSCH, ST.: *Facettierung dreidimensionaler Gebiete und Gittergenerierung unter Verwendung von Octree-Datenstrukturen*. Diplomarbeit, TU München, Fakultät für Bauingenieur- und Vermessungswesen, Dezember 2001. `http://www.inf.bauwesen.tu-muenchen.de/personen/crouse/dokumente/diploma_thesis_jaksch.pdf`.

[Mat00a] MATRA DATAVISION: *OpenCASCADE – Data Exchange User's Guide – IGES FORMAT v3.0*, März 2000. `http://www.opencascade.org/frame.php?page=version/doc.html` : `dataexchange/IGESSTD.PDF`, Teil von `htmlhelp.zip`.

[Mat00b] MATRA DATAVISION: *OpenCASCADE – Data Exchange User's Guide – STEP AP 214 - AP 203 FORMATS v3.0*, März 2000. `http://www.opencascade.org/frame.php?page=version/doc.html` : `dataexchange/STEPSTD.PDF`, Teil von `htmlhelp.zip`.

[Mun02] MUNDANI, R.-P.: *Effiziente Erzeugung und Bearbeitung von Oktalbaummodellen*. In: *2. Workshop Arbeitsgruppe Produktmodell*, Universität Weimar, Februar 2002. `http://www.iib.bauing.tu-darmstadt.de/dfg-spp1103/de/downloads/modelle/protokoll_02_02_20_weimar_modelle.pdf`.

[Mus02] MUSTUN, A.: *dxflib v0.1.2*, April 2002. `http://dxflib.sourceforge.net`.

[Par90] PAREIGIS, B.: *Analytische und projektive Geometrie für die Computer-Graphik*. Teubner, 1990.

[Pie97] PIEGL, L.; TILLER, W.: *The NURBS book*. Springer, 2. Auflage, 1997.

[Ram02] RAMMELT, A. M.: *DXF Viewer Applet/Application [Java] v1.21.00beta*, Mai 2002. `http://www.escape.de/users/quincunx/dxfviewer/`.

[Sed94] SEDGEWICK, R.: *Algorithmen in C++*. Addison-Wesley, 1994.

[Sha01] SHAVIT, A.: *Collision Detection Tutorial*, 2000-2001. `http://www.cfxweb.net/~cfxamir/cd3.html`.

[Str95] STROUSTRUP, B.: *Die C++-Programmiersprache*. Addison-Wesley, 1995.

[vH02] HEESCH, D. VAN: *Doxygen v. 1.2.16*, April 2002. `http://www.stack.nl:80/~dimitri/doxygen/`.

Erklärung

Hiermit versichere ich, diese Arbeit
selbständig verfaßt und nur die
angegebenen Quellen benutzt zu haben.

_____

(Stefan Mahler)