TECHNISCHE UNIVERSITÄT CHEMNITZ

Fakultät für Informatik

Professur Graphische Datenverarbeitung und Visualisierung

Studienarbeit

Vergleichende Studie zur Verwaltung großer volumetrischer 3D-Datensätze mit dem Schwerpunkt auf den hierarchischen und lauflängenkodierten Datenstrukturen

Holger Gerth

Chemnitz, den 10. April 2007

Prüfer: Prof. Dr. Guido Brunnett
Betreuer: Dipl.-Inf. David Brunner

Gerth, Holger
Vergleichende Studie zur Verwaltung großer volumetrischer 3D-Datensätze mit dem Schwerpunkt auf den hierarchischen und lauflängenkodierten Datenstrukturen
Studienarbeit, Fakultät für Informatik
Technische Universität Chemnitz, April 2007

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

1.1 Motivation

Modellierung und Animation von Geometrie sind das Herzstück vieler Anwendungen der Computergrafik. Es besteht daher eine besondere Herausforderung darin, effiziente und qualitativ hochwertige Darstellungen solcher verformbarer Oberflächen zu entwickeln.

In der Computergrafik unterscheidet man zwei grundlegende Methoden: Polygongrafiken und volumetrische Grafiken. Für die Darstellung eines 3D Objekts und die meißten Anwendungen reichen Polygongrafiken völlig aus. Auch benötigen sie relativ wenig Speicherplatz im Vergleich zu volumetrischen Darstellungen.
Die Frage stellt sich nun, wozu dann volumetrische Grafiken benötigt werden. Sie eignen sich besonders für Anwendungsfälle, in denen das tatsächliche Volumen eines Objekts benötigt wird, zum Beispiel wenn ein Objekt nach einer Verformung das gleiche Volumen haben soll wie vorher. Dagegen läßt sich Polygongrafiken das tatsächliche Volumen eines Objekts nur umständlich approximieren und eine Verformung unter Erhaltung des Volumens nur annähernd und unter recht hohem Rechenaufwand realisieren.
Ein gutes Beispiel für den Einsatz volumetrischer Grafiken ist die korrekte Simulation von Flüssigkeiten unter physikalisch gegebenen Bedingungen (z.B. Dichte, Druck, Fließgeschwindigkeit).

Abbildung 1.1: Das Bild zeigt die Verwendung eines volumetrischen 3D Objekts in der Filmindustrie: das Tar-Monster aus dem Film Scooby Doo 2 (Quelle: [HBN+06])

1.2 Ziel dieser Arbeit

Bei der Verwendung von volumetrischen Grafiken ist der im Vergleich zu Polygongrafiken sehr hohe Speicherbedarf ein Problem. Während es bei einer Polygongrafik genügt wenige Punkte auf der Oberfläche des Objekts und gegebenfalls noch eine Triangulierung dieser Punkte abzuspeichern, wird bei einer Voxelgrafik für jeden einzelnen Punkt, also auch die im Inneren des Objekts liegenden oder gar nicht dazu gehörenden Punktes, Speicherplatz benötigt.
Es wurden verschiedene Datenstrukturen zur Lösung dieses Problems entwickelt, welche den Anspruch erheben, speichereffizient zu sein, aber trotzdem einen schnellen Zugriff auf die Daten ermöglichen. Unter Speichereffizienz versteht man dabei, daß möglichst nur für Voxel mit relevanten Daten Speicher benötigt wird. Also zum Beispiel die Voxel, welche den Rand oder die Konturen eines Objekts beschreiben.

Um diese relevanten Daten möglichst effizient zu speichern, nutzen die Datenstrukturen prinzipiell zwei Methoden. Bei der ersten Methode wird der benötigte Speicherplatz mittels Lauflängenkodierung[1] komprimiert. Vorteil dieser Methode ist, daß auch Informationen über Gebiete erhalten bleiben, die nicht zur Objektgrenze[2] gehören.
Bei der zweiten Methode werden nur die Voxel mit relevanten Daten gespeichert. Zu den anderen Voxeln werden keinerlei Informationen gespeichert. Dadurch kann man zwar einen geringeren Speicherbedarf als bei der Lauflängenkodierung erreichen, allerdings verliert man sämtliche Informationen über die Bereiche außerhalb und innerhalb der Objektgrenze.

In dieser Studienarbeit werden folgende Datenstrukturen vorgestellt und miteinander verglichen:

- das vollständige (kartesische) Voxelgitter als Grundlage für den Vergleich,
- das *Dynamische Tubuläre Gitter* von Nielson und Museth [NM06],
- die *Lauflängenkodierte Spärliche Niveaumenge* von Housten u.a. [HWB04]
- das *Perfekte Räumliche Hashing* von Lefebvre und Hoppe [LH06] sowie
- die *Hierarchische Lauflängenkodierte Niveaumenge* von Housten u.a. [HBN+06].

Neben diversen Besonderheiten der einzelnen Datenstrukturen, stehen vor allem Speichereffizienz und die Zugriffsgeschwindigkeit auf die Daten der Datenstruktur im Mittelpunkt der Studienarbeit.
Der Schwerpunkt der Untersuchung liegt dabei auf der Hierarchischen Lauflängenkodierten

[1] Siehe Kapitel 2.2
[2] Die Objektgrenze bezeichnet die äußeren Grenzen eines grafischen Objekts. Im 2D bezeichnet sie demnach den Rand des Objekts, im 3D dessen Hülle.

Niveaumenge, welche das Problem der Speichereffizienz mittels Lauflängenkodierung zu lösen versucht.

2 Definitionen

2.1 3D-Bild

Ein 3D-Bild B ist definiert als eine Menge von Punkten b des $\mathbb{R}^3$, denen bestimmte Farbwerte zugeordnet wurden:

$$B = \{b_1, b_2, ..., b_n\} \text{ mit } b_i \in \mathbb{R}^3$$

Es gibt zwei wesentliche Möglichkeiten, Bilder in der Computergrafik zu erzeugen:

Polygongrafiken oder auch polygonale Darstellungen sind definiert als Tupel

$P = (V, E)$ mit
$V = \{v_1, v_2, ..., v_n\}, v_i \in \mathbb{R}^d$ als Menge der Vertizes und
$E = \{e_1, e_2, ..., e_m\}, e_k = v_i \bar{v}_j, i \neq j$ als Menge der Kanten.

Farbwerte werden bei polygonalen Darstellungen nur für die Vertices definiert. Für die Kanten und die davon eingeschlossenen Flächen (Polygone) werden die Farbwerte dann beim Rendering interpoliert.
Vorteil von Polygongrafiken sind ein relativ geringer Speicherbedarf und eine qualitätsverlustfreie Skalierbarkeit.

Bei den *Punktgrafiken*, im dreidimensionalen Raum als *Voxelgrafiken* bezeichnet, werden Farbinformationen zu jedem einzelnen Punkt eines Objekts gespeichert. Sie entsprechen also der allgemeinen Definition eines 3D-Bilds.

2.2 Lauf/Lauflängenkodierung

Unter einem Lauf versteht man die Zusammenfassung von Daten der selben Art, indem man nur Typ, Startpunkt und Anzahl (Länge des Laufs) speichert. Das Verfahren der Kompression bezeichnet man dann als *Lauflängenkodierung* (LLK).
Bei der Lauflängenkodierung eines Texts würde zum Beispiel aus *aaabbccccddd* die Kodierung *a3b2c4d3* entstehen.

2.3 Niveaumenge

Bei einer Niveaumenge (engl. Level Set) wird in einem n-dimensionalen Raum der $(n-1)$-dimensionale Rand Γ eines n-dimensionalen Objekts als Nullstellenmenge einer n-dimensionalen Hilfsfunktion φ beschrieben. Die Hilfsfunktion wird auf dem ganzen betrachteten Gebiet definiert, und zwar mit positiven Werten auf der einen und negativen Werten auf der anderen Seite von Γ.

Der Vorteil einer Niveaumenge liegt darin, daß man Kurven und Oberflächen auf einem räumlich festen (Eulerschen) Koordinatensystem berechnen kann, ohne Parametrisierungen dieser Objekte verwenden zu müssen. So muß die Topologie[1] des Objekts nicht unbedingt bekannt sein bzw. kann sich während der Berechnung ändern. Damit ist eine einfache Verfolgung der Ränder beweglicher Objekte möglich.

Ein Beispiel für eine Niveaumenge wäre ein Kreis in 2D. Der Rand Γ ist gegeben durch die Kreisfunktion $(x-x_M)^2+(y-y_M)^2=r^2$, wobei $p=(x,y)\in\Gamma$ ein Randpunkt ist. Der Mittelpunkt des Kreises ist gegeben durch (x_M,y_M) und sein Radius durch r.
Die Hilfsfunktion φ ist in diesem Beispiel eine vorzeichenbehaftete Abstandsfunktion, welche für Punkte $p\in\Gamma$ mit Null, für $(x-x_M)^2+(y-y_M)^2>r^2$ mit positiven Abstand und für $(x-x_M)^2+(y-y_M)^2<r^2$ mit negativen Abstand definiert ist.

2.4 Kartesisches Gitter

Ein kartesisches Gitter (Rechteckgitter) $\mathcal{L}$ ist durch die Punkte des $\mathbb{Z}^n$ definiert:

$$\mathcal{L}=\{p\in\mathcal{Z}^n\}$$

Den Punkt $p\in\mathcal{L}$ bezeichnet man als *Gitterpunkt* auf dem n-dimensionalen kartesischen Gitter $\mathcal{L}$. Die durch die halben Distanzen zwischen den Gitterpunkten verlaufenden Geraden bezeichnet man als *Abstandsteilungen*, die von ihnen eingegrenzten Bereiche als *Voronoizellen* der Gitterpunkte. Bei einem kartesischen Gitter sind die Abstandsteilungen orthogonal[2] und parallel zu den Koordinatenachsen (bzw. Koordinatenebenen im 3D)
In Abbildung 2.1 werden die Gitterpunkte blau umd die Abstandsteilungen als rote und schwarze Linien dargestellt.

[1] Die Topologie kennzeichnet Eigenschaften eines geometrischen Körpers, die sich durch Homöomorphismen (bijektive, stetige Abbildungen zwischen zwei Objekten wie Dehnen, Stauchen, Verbiegen oder Verzerren) nicht verändern lassen. So hat z.B. eine Kugel die selbe Topologie wie ein Quader.

[2] Die Abstandsteilungen des Gitters stehen senkrecht aufeinander.

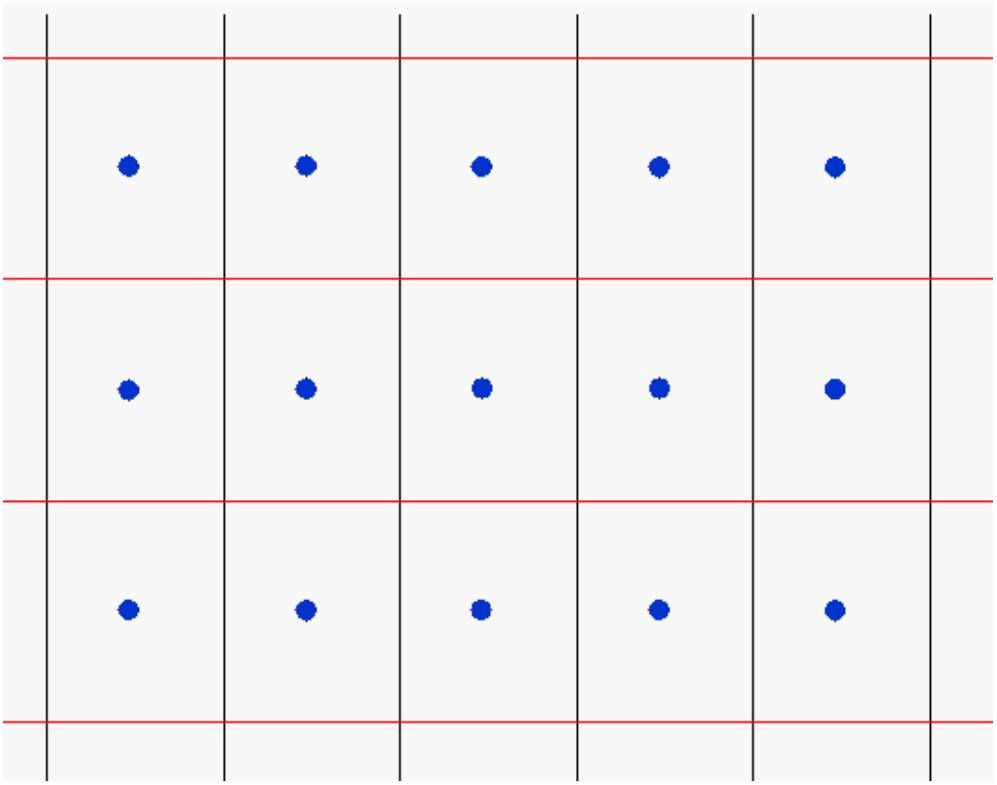

Abbildung 2.1: Beispiel für ein zweidimensionales kartesisches Gitter

3 Übersicht über verschiedene Datenstrukturen

3.1 Vollständiges Voxelgitter

Das vollständige Voxelgitter ist definiert als n-dimensionales kartesisches Gitter. Die Voxel definieren sich dabei über die Voronoizellen des Gitters.

Die Werte der Voxel werden in der Datenstruktur in einem dem Gitter entsprechenden n-dimensionalen Array gespeichert, unabhängig davon, ob sie definierte Daten enthalten oder nicht. Der Vorteil liegt darin, daß in konstanter Zeit auf jedes beliebige Element des Voxelgitters zugegriffen werden kann. Der Nachteil liegt im enormen Speicherbedarf für die Datenstruktur, da für jeden Gitterpunkt Speicher allokiert werden muß, auch wenn dieser als innerhalb oder außerhalb der Objektgrenze klassifiziert worden ist.

Weitere Vorteile des vollständigen Voxelgitters sind zum einem die Möglichkeit zur Darstellung nicht-geschlossener Objekte, dadurch das für jeden einzelnen Gitterpunkt unabhängig Daten gespeichert werden können, und zum anderen die Möglichkeit zur Festlegung einer expliziten Bounding Box über die Ausdehnung des Gitters.

3.2 Dynamisches Tubuläres Gitter

Das *Dynamische Tubuläre Gitter* (DT Gitter [NM06]) ist eine effiziente Datenstrukter für n-dimensionale dynamische tubuläre Gitter. Unter einem tubulären Gitter T versteht man eine Untermenge von Gitterpunkten, welche auf einem unendlichen Gitter innerhalb einer festen euklidischen Entfernung *dist* zur Objektgrenze eines n-dimensionalen Objekts definiert sind. Sei $d_{border}(p)$ der euklidische Abstand eines Gitterpunkts p zur Objektgrenze:

$$T = \left\{ p \in \mathbb{Z}^3 : d_{border}(p_i) < dist \right\}$$

Als *Projektionsspalte* (p-Spalte) bezeichnet eine eindimensionale Menge von Gitterpunkten in einem tubulären Gitter der Dimension n, welche sich mittels orthogonaler Projektion in den durch die ersten $n-1$ Koordinatenrichtungen aufgespannten Unterraum projezieren lassen.

3.2.1 Aufbau der Datenstruktur

Ein DT Gitter ist eine hierarchische Datenstruktur und definiert sich rekursiv über DT Gitter hierarchisch untergeordneter Dimension. Somit läßt sich die Datenstruktur für jede beliebige Dimension verwenden.
Die nummerischen Werte der Gitterpunkte (z.B. Farbwerte) werden entsprechend ihrer Koordinaten lexikographisch in einem Werte-Array des DT Gitters der höchsten Dimension n gespeichert. Die Koordinaten werden in Form sogenannter Verbundskomponenten in Koordinatenrichtung der jeweiligen Dimension d in einem Koordinaten-Array (in Abbildung 3.1 mit yKoord bzw. xKoord bezeichnet) gespeichert. Unter einer Verbundkomponente versteht man die maximale Anzahl adjazenter Gitterpunkte entlang einer p-Spalte. Es werden dabei nur die Start- und Endkoordinaten der Verbundskomponenten abgespeichert.
Ein zusätzliches Array (in Abbildung 3.1 als acc bezeichnet) enthält außerdem die Indizes der ersten Gitterpunkte einer p-Spalte. Dieses Array ist für einen effizienten Direktzugriff notwendig.

Das DT Gitter der nächst niedrigeren Dimension $d-1$ erhält man nun mittels Projektion des Objekts in den Unterraum, der durch die ersten $d-1$ Koordinatenrichtungen aufgespannt wird. Die verbleibenden DT Gitter niedrigerer Dimension werden dann durch Projektion in den jeweiligen Unterraum ermittelt.
DT Gitter der Dimendion $d-1$ und niedriger enthalten ebenfalls ein Werte Array, allerdings werden darin keine nummerischen Werte gespeichert, sondern Paare von Indizes in das Werte Array und das Koordinaten Array des DT Gitters der nächst höheren Dimension.

Dazu ein Beispiel in 2D:

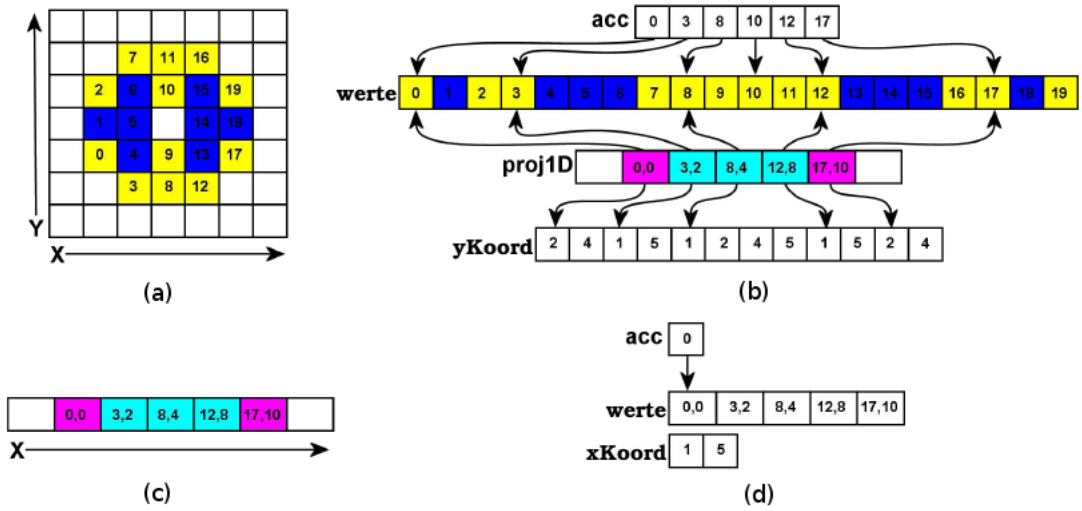

Abbildung 3.1: Beispiel für ein DT Gitter der Dimension 2

Abbildung 3.1 zeigt den Aufbau eines DT Gitters der Dimension 2.
Unter (a) ist ein 2D Objekt als Voxelgitter abgebildet. Zum besseren Verständnis wurden die Voxel farblich markiert. Beginn und Ende einer Verbundskomponente sind gelb, die innerhalb einer Verbundskomponente liegenden Voxel sind blau dargestellt. Die Werte der Voxel sind zur besseren Übersicht durchnummeriert.
Unter (b) ist das entsprechende tubuläre Gitter der Dimension 2 abgebildet. Das acc-Array enthät die Indizes der ersten Gitterpunkte einer p-Spalte, das Werte-Array die nummerischen Werte der Voxel in lexikographischer Reihenfolge, das yKoord-Array die Anfangs- und Endkoordinaten der Verbundskomponenten in y-Richtung und proj1D ist ein Zeiger auf ein tubuläres Gitter der nächst niedrigeren Dimension, in diesem Fall also auf das die x-Achse kodierende DT Gitter.
In (c) wird das zusammengefasste 1D Gitter gezeigt, welches durch Projektion der p-Spalten in y-Richtung auf die x-Achse erzeugt wurde. (d) zeigt das korrespondierende 1D DT Gitter. Das Werte-Array enthält in diesem Fall Paare von Indizes, welche in das Werte-Array des 2D DT Gitters und dessen yKoord-Array verweisen.

Da in der Datenstruktur lediglich die Verbundskomponenten gespeichert werden, ist keine Innen/Außen Klassifikation[1] möglich. Somit kann das DT Gitter keine nicht-geschlossenen Objekte darstellen, diese müssen vorher durch zusätzliche Voxel geschlossen werden.
Ein weiterer Nachteil des DT Gitters ist das Fehlen einer expliziten Bounding Box.

3.2.2 Direktzugriff

Der Direktzugriff auf den Gitterpunkt X_n auf einem n-dimensionalen DT Gitter ist rekursiv folgendermassen definiert:

Im ersten Schritt wird der Direktzugriff des $n-1$-dimensionalen DT Gitter aufgerufen (falls dieses exisitiert), um zu überprüfen, ob die p-Spalte X_{n-1} in der Projektion des tubulären Gitters enthalten ist.
Ist dies der Fall, wird ermittelt, ob die n-te Koordinate des Gitterpunktes, x_n, zwischen den minimalen und maximlane Koordinaten der p-Spalte X_{n-1} liegt. Liegt sie dazwischen, wird mittels Binärer Suche im nKoord-Array die naheste Startkoordinate einer Verbundskomponente entlang der n-ten Koordinatenrichtung ermittelt.
Zum Schluß wird ermittelt, ob sich der Gitterpunkt überhaupt innerhalb dieser p-Spalte befindet, also ob er in einer von deren Verbundskomponenten enthalten ist. Ist dies der Fall, wird der entsprechende nummerische Wert zurück geliefert.

Bei hierarchisch untergeordneten DT Gittern wird anstatt eines Wertes nur der Index in das Werte-Array dieses DT Gitters zurückgeliefert. Zur Berechnung dieses Index wird folgende Formel verwendet: sei k der Index auf die Startkoordinate der Verbundskomponente, dann ist

$$index = acc[k \gg 1] + x_n - nKoord[k].$$

Dabei ist $\gg$ der Rechtsshift-Operator, k wird für den Zugriff auf das acc-Array um eins nach rechts verschoben, da k ein Index in das nKoord-Array ist, welches doppelt soviele Elemente wie das acc-Array enthält. Mit Hilfe dieses Index kann der Algorithmus die korrekte p-Spalte ermitteln, in der nach dem Gitterpunkt gesucht werden soll.

Für den Direktzugriff folgendes Beispiel: in dem in Abbildung 3.1 dargestellten DT Gitter soll auf den Gitterpunkt mit dem Wert 5 und den Koordinaten (2,3) zugegriffen werden. Folgende Schritte sind notwendig:
Der Algorithmus geht zuerst zur tiefsten Dimension runter, also in das 1D DT Gitter der x-Achse. Diese enthält immer nur eine Projektionsspalte. Es wird nun überprüft, ob die x-Koordinate zwischen den Start- und Endkoordinaten dieser p-Spalte liegen. Da dies bei $x = 2$

[1] Bei der Innen/Außen-Klassifikation werden Bereiche die nicht zu den Objektgrenzen gehören, entsprechend ihrer Lage in innerhalb des Objekts bzw. außerhalb des Objekts klassifiziert.

der Fall ist, wird mittels der Binären Suche auf dieser p-Spalte die naheste Startkoordinate einer enthaltenen Verbundskomponente gesucht. Nun wird überprüft, ob sich die Koordinate innerhalb dieser Verbundskomponente befindet. Dies ist für $x = 2$ mit $1 \leq 2 \leq 5$ in dem Beispiel der Fall. Nun wird entsprechend obiger Formel der Index in das Werte-Array errechnet, welcher 1 ergibt und auf das Index-Paar (3,2) im Werte-Array des 1D DT Gitters zeigt. Dies wird nun an den Direktzugriff des 2D DT Gitters der y-Achse zurück gegeben.
Dieses Index-Paar verweist zum einen in das nKoord-Array auf den ersten Punkt der p-Spalte mit der x-Koordinate 2 und zum anderen auf den Wert dieses Punkts im Werte-Array. Innerhalb dieser p-Spalte wird nun nach der Startkoordinate der Verbundskomponente gesucht, die die Koordinate $y = 3$ enthält. Der ermittelte Index in das nKoord-Array ist 2. Da für $y = 3$ anscheinend $1 \leq 3 \leq 5$ gilt, befindet sich der Gitterpunkt innerhalb dieser Verbundskomponente. Es wird nun wieder der Index in das Werte-Array berechnet. Dieser ist 5 und da sich der Algorithmus wieder im DT Gitter der höchsten Dimension befindet, wird der numerische Wert an Position 5 des Werte-Array zurückgeliefert, welcher im Beispiel ebenfalls 5 ist.

3.2.3 Sequenzieller Zugriff

Der sequenzielle Zugriff des DT Gitters gestaltet sich als Iteration über die definierten Gitterpunkte entlang der p-Spalten und wird zum Beispiel von einigen Deformationsalgorithmen benötigt, welche sequenziell auf die Daten zugreifen. Auch benötigen einige Operatoren für Niveaumengen einen schnellen sequenziellen Zugriff.
Beim sequenziellen Zugriff wird eine Zeiger-Struktur (ein sogenannter Locator) mittels eines Inkrement-Algorithmus über die Datenstruktur iteriert. Dieser Locator ist folgendermassen definiert:

```
1 struct LocatorND {
2   Locator(N-1)D   loc;
3   unsigned int    iWerte;
4   unsigned int    iKoord;
5   Index           Xn;
6 };
```

Er enthält also zum einen Indizes in das Werte und das Koordinaten Array sowie die n-te Koordinate Xn. Der Zeiger loc verweißt außerdem auf einen Locator für die nächst niedrigere Dimension.

Um diesen Locator über die Datenstruktur zu iterieren, wird ein Iterator verwendet, welcher den Locator mittels seines Inkrement-Algorithmus auf den nächsten Gitterpunkt zeigen läßt.

```
1 struct IteratorND {
2   Iterator(N-1)D  it;
3   double          wert;
4   DTGrid          pDTGitter;
5 };
```

Der Iterator enthält einen Verweis auf das DT Gitter, über das iteriert werden soll, den numerischen Wert des aktuellen Gitterpunkts sowie einen Verweis auf einen Iterator für die nächst niedrigere Dimension.

Der Inkrement-Algorithmus des Iterators erhält als Eingabe den Locator loc, der über das DT Gitter iteriert werden soll. Dieser Locator zeigt zu Beginn auf den ersten Gitterpunkt der ersten Verbundskomponente in der Datenstruktur.
Im ersten Schritt wird der Zeiger iWerte des Locators auf den Index des nächsten Werts im Werte Array gesetzt und der Wert des Iterators entsprechend aktualisiert. Es wird nun überprüft, ob Xn bereits die letzte Koordinate der Verbundskomponente ist (Xn == iKoord + 1). Ist dies nicht der Fall, wird Xn auf die nächste Koordinate der Verbundskomponente gesetzt und die Ikrementierung endet.
Falls doch, wird iKoord auf den Startpunkt der nächsten Verbundskomponente gesetzt und Xn entsprechend angepaßt. Nun muß überprüft werden, ob eine Inkrementierung des Locators in der nächst hierarchisch untergeordneten Dimension notwendig ist. Dies ist der Fall, wenn die Koordinate des Index-Paars im DT Gitter der nächst niedrigeren Dimension, auf den der entsprechende Locator für diese Dimension mit iWerte zeigt, den gleichen Koordinatenindex wie der aktuelle Locator hat:

pDTGitter.proj(N-1)D.werte[loc.loc.iv + 1].iKoord == loc.iKoord

Ist dies der Fall, wird die Inkrementierung rekursiv für die nächste Dimension weitergeführt. Ansonsten endet sie an der Stelle.
Nach der Inkrementierung enthält der Locator die Koordinaten des aktuellen Gitterpunkts und der Iterator den dazugehörigen Wert.

3.3 Lauflängenkodierte Spärliche Niveaumenge

Die *Lauflängenkodierte Spärliche Niveaumenge* war die erste Implementation eines Level Sets, bei der die Gitterpunkte effizient mittels Lauflängenkodierung gespeichert wurden [HWB04]. Außerdem wird einen Innen/Außen-Klassifikation der Bereiche, die nicht zum *Begrenzungsbereichs* gehören, vorgenommen, wodurch es möglich ist, auch nicht-geschlossene Flächen darzustellen. Als Begrenzungsbereich bezeichnet man den Bereich, der innerhalb eines bestimmten euklidischen Abstands zu den Objektgrenzen in einer Niveaumenge liegt. Der Zugriff auf die einzelnen Gitterpunkte ist relativ schnell, dank der Verwendung von zwei LookUp-Tabellen.

3.3.1 Aufbau der Datenstruktur

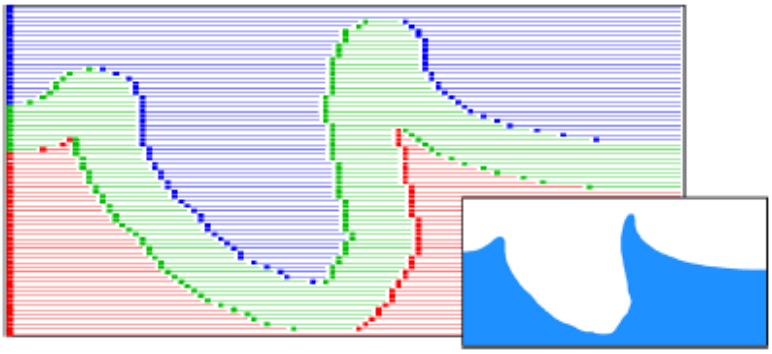

Abbildung 3.2: Kodierung eines 2D Objekts in Läufe. (Quelle: [HWB04])

Abbildung 3.2 zeigt die Zerlegung eines 2D Objekts in Läufe. Der Bereich außerhalb des Objekts wird blau dargestellt, der Bereich innerhalb rot und der Begrenzungsbereich grün. Die Läufe werden also in folgende drei Typen klassifiziert:

- *definiert*: definierter Lauf, bezeichnet einen Lauf, der entlang des Begrenzungsbereich verläuft.
- $-\infty$: negativer Lauf, bezeichnet einen Lauf der innerhalb der Objektgrenzen verläuft
- $+\infty$: positiver Lauf, bezeichnet einen Lauf der außerhalb der Objektgrenzen verläuft

Die Datenstruktur baut sich (in 3D) folgendermassen auf: Die Datenstruktur verwendet, um einen schnellen Zugriff zu gewährleisten, zwei LookUp-Tabellen. In einer Laufstart-Tabelle werden die Koordinaten aller Läufe entlang der Kodierungsachse gespeichert, an denen sie beginnen. Eine zweite Tabelle, die Segmentstart-Tabelle, enthält die Voxelkoordinaten der ersten Läufe einer lauflängenkodierten Reihe auf den anderen beiden Koordinatenachsen,

also den ersten Lauf eines lauflängenkodierten Segments.
Die Werte der Voxel auf dem Begrenzungsbereich werden in einem separaten Array gespeichert. Die als definiert gekennzeichneten Läufe in der Laufstart-Tabelle enthalten zur Ermittlung des Wertes Indizes in dieses Array.

Folgende Abbildung zeigt die Kodierung eines Voxelobjekts mittels der LLK Spärlichen Niveaumenge in 2D. Zur besseren Darstellung sind die verschiedenen Läufe wie in Abbildung 3.2 farblich gekennzeichnet.

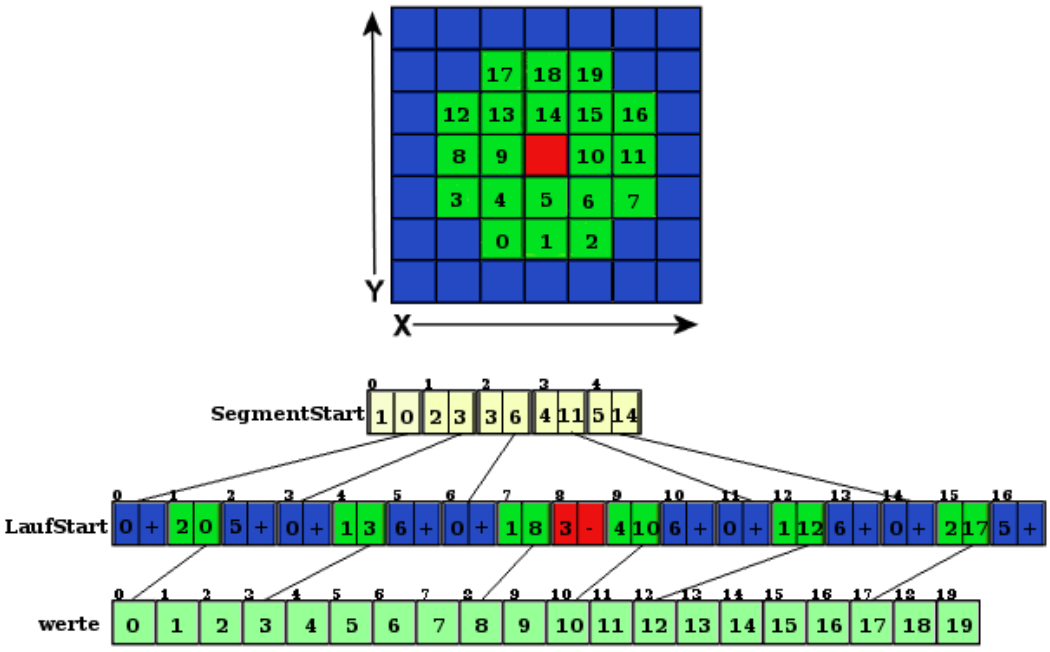

Abbildung 3.3: Lauflängenkodierte Spärliche Niveaumenge

Die SegmentStart-Tabelle enthält die Koordinaten des ersten Laufs einer lauflängenkodierten Reihe (Segment) auf der y-Achse (erster Index) und einen Verweis auf diesen Lauf in der LaufStart-Tabelle (zweiter Index). Die LaufStart-Tabelle enthält nun die x-Koordinaten aller Läufe sowie den jeweiligen Typ des Laufs. Nicht-definierte Läufe werden mit $-\infty$ bzw. $+\infty$ gekennzeichnet, während definierte Läufe einen Index in das Werte Array enthalten.

3.3.2 Direktzugriff

Den Direktzugriff besteht aus drei Schritten. Im ersten Schritt wird mit Hilfe der vorher beschriebenen Segmentstart-Tabelle das Segment ermittelt, in dem sich der gesuchte Punkt befindet. Als nächstes wird durch eine Binäre Suche in der Laufstart-Tabelle innerhalb dieses Segments der entsprechende Lauf ermittelt. Falls dieser Lauf definiert ist, kann mittels des Laufstart-Index als Offset der entsprechende nummerische Wert ermittelt werden: $index = defIndex + xKoord - xStartKoord$. Dabei ist $defIndex$ der Index auf den Wert des ersten Gitterpunkts in dem entsprechenden definierten Lauf, *xKoord* die x-Koordinate des gesuchten Punkts und *xStartKoord* die x-Koordinate vom ersten Gitterpunkt des Laufs.

Als Beispiel für den Direktzugriff soll auf den Punkt (2,3) mit dem Wert 9 in Abbildung 3.3 zugegriffen werden. Dazu wird in der SegmentStart Tabelle zuerst nach dem Eintrag mit der y-Koordinate 3 gesucht. Dies ist der Eintrag an Position 2, welcher den Index 6 enthält. An Position 6 in der Laufstart-Tabelle befindet sich also der erste Lauf des Segments, in dem sich der Punkt befindet. Nun wird mittels der Binären Suche in diesem Segment der entsprechende Lauf ermittelt, der die x-Koordinate 2 enthält. Da $1 \leq 2 < 3$ ist der Lauf an Position 7 der gesuchte. Da dieser definiert ist, kann mittels dessen Index der entsprechende Wert im Werte Array ermittelt werden: $index = 8 + 2 - 1 = 9$. An Position 9 im Werte Array befindet sich demnach der nummerische Wert des Voxels mit den Koordinaten (2,3), welcher im Beispiel ebenfalls 9 beträgt.

3.4 Perfektes Räumliches Hashing

Eine neue Datenstruktur für Voxelgrafiken ist das *Perfekte Räumliche Hashing* [LH06], bei dem versucht wird, die Daten mittels Hashing in einer kompakten n-dimensionalen Tabelle zu speichern, dabei aber trotzdem einen effizienten Direktzugriff zu ermöglichen.

3.4.1 Grundlagen

Hashing

Beim Hashing wird davon ausgegangen, daß es eine große Menge U von potentiellen Schlüsseln (das *Universum*) gibt, die tatsächlich zu verwaltende Menge $S \subset U$ an Schlüsseln aber viel kleiner ist (im allgemeinen ist S aber vorher nicht bekannt). Diese relativ kleine Menge relevanter Schlüssel wird nun mittels einer sogenannten Hash-Funktion h auf eine Hash-Tabelle H mit ausreichend vielen freien Einträgen abgebildet. Der über die Hash-Funktion berechnete Wert dient demnach als Adresse der entsprechenden Daten in der Hash-Tabelle.

Da der Wertebereich einer Hash-Funktion viel kleiner als der der Quellmenge ist, kann es dazu kommen, daß für zwei verschiedene Schlüssel die selbe Position in der Hash-Tabelle berechnet wird. Man spricht in diesem Fall von einer *Kollision*.
Um diese Kollisionen zu vermeiden, gibt es verschiedene Ansätze. Beim offenen Hashing wird mittels Sondieren ein freier Platz in der Hash-Tabelle gesucht, an den die Daten verschoben werden, oder es wird mittels einer zweiten Hash-Funktion ein bestimmtes Intervall in der Hash-Tabelle eingegrenzt. Eine andere Möglichkeit ist die Verwendung verketteter Listen an den Hash-Positionen, damit an einer Position mehrere Daten gespeichert werden können.

Hashing bietet den Vorteil, eine kleine Menge an relevanten Schlüsseln effizient zu speichern und einen schnellen Direktzugriff auf die Daten zu ermöglichen.
Ein Nachteil ist eine mögliche *Entartung*[2] der Hash-Tabelle, wenn ein bestimmter Füllgrad überschritten wurde. Dies kann nur durch eine Vergrößerung der Tabelle behoben werden. Außerdem ist ein sequenzieller Zugriff auf die Daten nur schwer möglich, da sie in ungeordneter Reihenfolge in der Hash-Tabelle gespeichert werden.

Hashing und Grafikhardware

Traditionelles Hashing bereitet auf aktueller Grafikhardware folgende Probleme:

- *Kollisionen*: Um Kollisionen zu vermeiden, führen Hash-Algorithmen normalerweise eine Reihe von Abfragen der Hash-Tabelle durch, welche in der Anzahl bei jeder

[2] Als Entartung einer Hash-Tabelle versteht man eine sehr ungleichmäßige Verteilung der Daten in der Tabelle.

Anfrage variieren können. Diese Vorgehensweise ist allerdings auf einer GPU[3] sehr ineffizient, da durch die SIMD (Single Instruction, Multiple Data) Parallelisierung alle Pixel auf die Worst Case Anzahl an Abfragen warten müssen.

- Kohärenz der Daten: Um übermäßige Kollisionen und Clusterbildung zu vermeiden, wird eine Hash-Funktion benötigt, die die Daten möglichst gleichmäßig auf der Hash-Tabelle verteilt. Daher weisen Hash-Tabellen oft eine schlechte Lokalität[4] auf, welche zu gelegentlichen Cache Misses und einem langsamen Speicherzugriff führt.

3.4.2 Aufbau der Datenstruktur

Perfektes Hashing

Um Hashing mit aktueller Grafikhardware kompatibler zu machen, benutzen die Autoren eine *perfekte* Hash-Funktion, welche für eine statische Menge an Elementen vorberechnet wird, damit keine Kollisionen auftreten. Außerdem soll die Funktion noch minimal sein, damit es in der Hash-Tabelle möglichst keine ungenutzten Einträge gibt.
Die Entwickler der Datenstruktur definieren eine solche minimale perfekte mehrdimensionale Hash-Funktion folgendermassen:

$$h(p) = h_0(p) + \Phi[h_1(p)]$$

Dabei werden zwei unperfekte Hash-Funktionen h_0, h_1 mit einer Offset-Tabelle Φ kombiniert. Die Offset-Tabelle dient dazu, aus der unperfekten Funktion h_0 eine perfekte Hash-Funktion h zu machen.

Räumliche Kohärenz

In der Computergrafik wird auf 2D und 3D Texturen oftmals kohärent von der parallelen GPU zugegriffen, daher sind sie besonders für diesen Zugriff optimiert. Eine Hash-Tabelle (Hash-Textur) sollte daher ähnlich aufgebaut sein, um diesen kohärenten Zugriff auszunutzen.
Die Funktionen h_0 und h_1 wurden daher räumlich kohärent entworfen, um einen effizienten Zugriff auf die Offset-Tabelle Φ und die Hash-Tabelle H zu ermöglichen. Bemerkenswerterweise handelt es sich bei den Funktionen um simple Modulo-Adressierungen über die Tabellen.
Weiterhin werden die Offset-Werte in Φ optimiert, um die Kohärenz von h selbst zu maximieren.

[3] Graphical Processing Unit - der Grafikprozessor
[4] Datensätze, die aufeinander folgen, werden in der Hash-Tabelle weit voneinander entfernt gespeichert.

Aufbau des perfekten Hash-Schemas

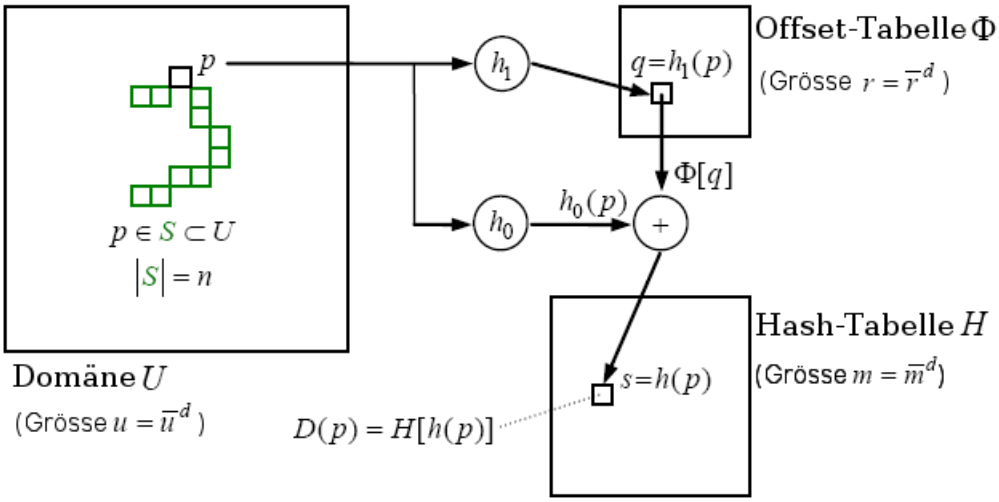

Abbildung 3.4: Aufbau des Perfekten Räumlichen Hash Schemas

Die Domäne U sei ein d-dimensionales Gitter mit $u = \bar{u}^d$ Positionen. Die Untermenge $S \subset U$ bezeichnet die relevanten Daten und umfaßt n Gitterpunkte. Jede Position $p \in S$ besitzt einen zugehörigen Datensatz $D(p)$. Das Ziel ist nun, $D(p)$ durch eine dicht gepackte Hash-Textur $H[h(p)]$ zu ersetzen. Es gilt:

- die Hash-Tabelle H ist ein quadratisches d-dimensionales Array der Größe $m = \bar{m}^d \geq n$ und enthält die Datensätze $D(p)$, $p \subset S$ und
- die perfekte Hash Funktion $h(p) : U \rightarrow H$ ist eine injektive Abbildung begrenzt auf S, jede Position $p \subset S$ wird eindeutig auf einen Platz $s = h(p)$ abgebildet.

Wie in Abbildung 3.4 erkennbar ist, besitzt die perfekte Hash-Funktion folgende Form:

$$h(p) = h_0(p) + \Phi[h_1(p)] \mod \bar{m}, \text{ wobei:}$$

- die Offset-Tabelle Φ ein quadratisches d-dimensionales Array der Größe $r = \bar{r}^d = \sigma n$ ist und d-dimensionale Vektoren enthält,
- die Abbildung $h_0 : p \rightarrow M_0 p \mod \bar{m}$ von der Domäne U in die Hash-Tabelle H eine einfache lineare Transformation mit einer $d \times d$ Matrix M_0, modulo der Tabellengröße, und
- die Abbildung $h_1 : p \rightarrow M_1 p \mod \bar{r}$ von der Domäne U in die Offset-Tabelle ϕ ähnlich aufgebaut ist.

Konstruktion der Hash-Tabellen

Das Erzeugen der Hash-Tabelle und der Offset-Tabelle gestaltet sich als sehr aufwendig und rechenintensiv, da im ungünstigsten Fall mehrere Anläufe benötigt werden, um alle Voxel ohne Kollisionen in der Tabelle zu verteilen. Folgende Schritte sind dazu nötig:

Zuerst müssen die Größen m und r der beiden Tabellen bestimmt werden. Die Größe der Hash-Tabelle m wählt man so, daß sie gerade genug Einträge besitzt, um alle definierten Punkte darin abzubilden. Bei n definierten Gitterpunkten beträgt die Größe der Hash-Tabelle m also mindestens n. Da $m = \bar{m}^d \geq n$ ist also $\bar{m} = \lceil \sqrt[d]{n} \rceil$.
Die Größe der Offset-Tabelle r wird zu Beginn so klein wie möglich gewählt. Für eine schnelle Konstruktion hat sich die Wahl von $r = \bar{r}^d \geq 1/(2d) * n$ als günstig erwiesen, $\bar{r}$ wird somit auf $\lceil \sqrt[d]{1/(2d) * n} \rceil$ gesetzt. Sollte sich die Größe $\bar{r}$ als zu klein erweisen, wird sie im Verlauf der Konstruktion je Durchlauf um eins vergrößert.

Es werden nun nach h_1 die Positionen der Punkte in der Offset-Tabelle berechnet. Punkte, die demnach auf die selbe Position in der Offset-Tabelle abgebildet würden, werden zu einer Gruppe zusammengefaßt, welche später den selben Verschiebungskoeffizienten erhalten.
Man betrachtet nun nacheinander alle diese Gruppen entsprechend ihrer Größe, angefangen bei der Gruppe mit den meisten Punkten. Können alle Punkte der Gruppe ohne Kollision in der Hash-Tabelle mittels h_0 abgebildet werden, wird als Verschiebungskoeffizient ein Null-Vektor in der Offset-Tabelle eingetragen. Ansonsten müssen die Punkte in der Hash-Tabelle so verschoben werden, daß es bei keinem der Punkte aus der jeweiligen Gruppe zu Kollisionen kommt. Der entsprechende Verschiebungsvektor wird dann in der Offset-Tabelle gespeichert.

Dies wird nun solange fortgeführt, bis alle Punkte in der Hash-Tabelle abgebildet wurden. Wird bei einer Kollision kein freier Platz in der Hash-Tabelle mehr gefunden, muß die Offset-Tabelle verGrößert werden. Die Berechnung der Hash-Tabellen startet dann mit dieser Größeren Offset-Tabelle neu.

3.4.3 Direktzugriff

Bei dem Direktzugriff auf die Daten einer Voxelmenge über die Hash-Tabellen ergeben sich zwei Probleme. Zum einen geht aus den Hash-Tabellen nicht hervor, ob ein Punkt definiert ist oder nicht, so daß ein nicht definierter Punkt über die Modulo-Funktionen auf den Wert eines anderen definierten Punktes abgebildet werden kann, und zum anderen ist die Ausdehnung der Voxelmenge ebenfalls unbekannt.

Diese Probleme läßt sich mit einer zusätzlichen d-dimensionalen Tabelle (Domäne-Bit Ta-

belle) umgehen, die die Ausdehnung der originalen Voxelmenge besitzt und für jeden Gitterpunkt ein Bit speichert, welches angibt, ob der Gitterpunkt definierte Daten enthält oder nicht.

Der eigentliche Direktzugriff benutzt die gleichen Modulo-Hashfunktionen, die bereits für die Erzeugung der Hash-Tabellen verwendet wurden. Nachdem über das Domäne-Bit überprüft wurde, ob sich an der Position definierte Daten befinden, wird mittels der Funktion h_1 zuerst der Verschiebungsvektor aus der Offset-Tabelle ermittelt. Dieser wird dann zum Ergebnis von h_0 addiert und über den damit berechneten Index läßt sich der Wert des Gitterpunkts ermitteln.

4 Hierarchische Lauflängenkodierte Niveaumenge

In der *Hierarchischen Lauflängenkodierten Niveaumenge* (H-LLK Niveaumenge, [HBN+06]) vereinen sich die Vorzüge des DT Gitters und der Lauflängenkodierten Spärlichen Niveaumenge. Die Daten werden effizient mittels Lauflängenkodierung gespeichert und durch die hierarchische Struktur sind Änderungen an der Niveaumenge mit relativ wenig Aufwand möglich. Die Datenstruktur bietet somit ein hohes Maß an Flexibilität und gute Performanz bezüglich Speicherbedarf und Zugriffszeiten.

Der grundlegende Lösungsvorschlag beruht auf der dimensionsweisen Anwendung der Lauflängenkodierung (LLK) auf das Objekt. Die Lauflängenkodierung wird also für jede Dimension unabhängig entlang einer bestimmten Kodierungsachse durchgeführt, wobei Bereiche außerhalb des Begrenzungsbereichs kompakt gespeichert werden. In Abschnitt 4.2 wird die Erzeugung der Datenstrukutur noch genauer beschrieben.

Kodierung der Läufe: Wie bereits erwähnt, nutzt die H-LLK Niveaumenge die Lauflängenkodierung, um das Objekt in einer Serie von Läufen zu kodieren. Es werden dabei folgende drei Kodierungen verwendet: *positive Läufe* beschreiben den Bereich außerhalb des Objekts, *negative Läufe* den Bereich im Inneren und *definierte Läufe* den Bereich innerhalb des Begrenzungsbereichs.

4.1 Aufbau der LLK Datenstruktur

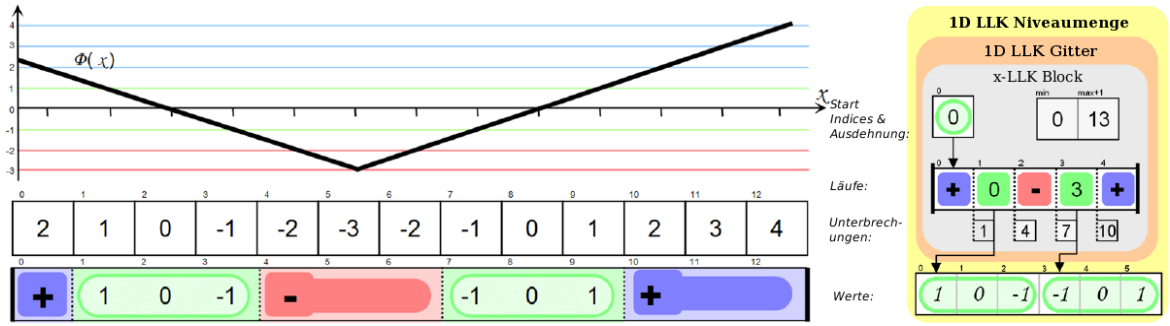

Abbildung 4.1: 1D LLK Niveaumenge

Die H-LLK Datenstruktur ist eine rekursive Datenstruktur. Sie besteht aus einem LLK Block, welcher die Läufe einer bestimmten Dimension kodiert, und einem Verweis auf einen LLK Block der nächst niedrigeren Dimension. Die niedrigste Dimension enthält außerdem noch ein Array für die Werte der innerhalb der definierten Läufe liegenden Gitterpunkte (Werte-Array). Diese Werte entsprechen dem euklidischen Abstand der Voxel zur Objektgrenze.

Ein LLK Block setzt sich aus folgenden Arrays zusammen:

- *Start Indizes:* beschreibt die Unterteilung des LLK Blocks in verschiedene Segmente. Das Array enthält Indizes, die als Zeiger auf den ersten Lauf des Segments im Läufe-Array dienen.
- *Ausdehnung:* die Werte *Min* und *Max* beschreiben die minimalen und maximalen (plus eins) Coordinaten entlang der Kodierungs-Achse. Durch diese Werte wird außerdem eine explizite Bounding Box des Objekts definiert.
- *Läufe:* beschreiben inwiefern jedes Segment in verschiedene Läufe unterteilt ist. Negative und positive Läufe bezeichnen komprimierte Läufe, während definierte Läufe einen Index enthalten, welcher in der niedrigsten Dimension als Zeiger in das Werte Array dient und bei höherer Dimensionen auf das entsprechende Segment in den Start Indizes des nächsten hierarchisch untergeordneten LLK Blocks zeigt.
- *Unterbrechungen:* enthalten in geordneter Reihenfolge entsprechend der Läufe die absoluten Koordinaten entlang der Kodierungs Achse, an denen jeder Lauf beginnt, mit Ausnahme des ersten Laufs eines Segments, dessen Startkoordinate durch *Min* gegeben ist.

4.2 Aufbau der hierarchischen Datenstruktur

Durch den hierarchischen Aufbau der H-LLK Datenstruktur läßt sie sich für beliebige Dimensionen benutzen. Für jede Dimension wird immer nur in eine Koordinatenrichtung kodiert. Bei einem Objekt der Dimension n werden die Voxel zuerst in Richtung der ersten Kodierungsachse (z.B. entlang der x-Achse) kodiert. Das Ergebnis wird dann auf den Unterraum projeziert, der durch die verbliebenen $n-1$ Koordinatenachsen aufgespannt wird. Aus den Kodierungsreihen, welche definierte Daten enthalten, werden bei dieser Projektion definierte Gitterpunkte erzeugt. Vollständig innerhalb oder außerhalb liegende Kodierungsreihen werden in negative oder positive Gitterpunkte projeziert. Anschließend wird dieses neue Voxelgitter der Dimension $n-1$ entlang der nächsten Kodierungsachse kodiert. Auf diese Art wird weiter fortgefahren, bis man in der niedrigsten Dimension angekommen ist.

Der hierarchische Aufbau der Datenstruktur soll an einem Beispiel in 2D (Abbildung 4.2) aufgezeigt werden. Die Datenstruktur besteht in diesem Beispiel aus zwei LLK Blöcken entsprechend der beiden Kodierungsachsen. Der obere Block kodiert das Objekt in Richtung der y-Achse und der untere Block in Richtung der x-Achse.

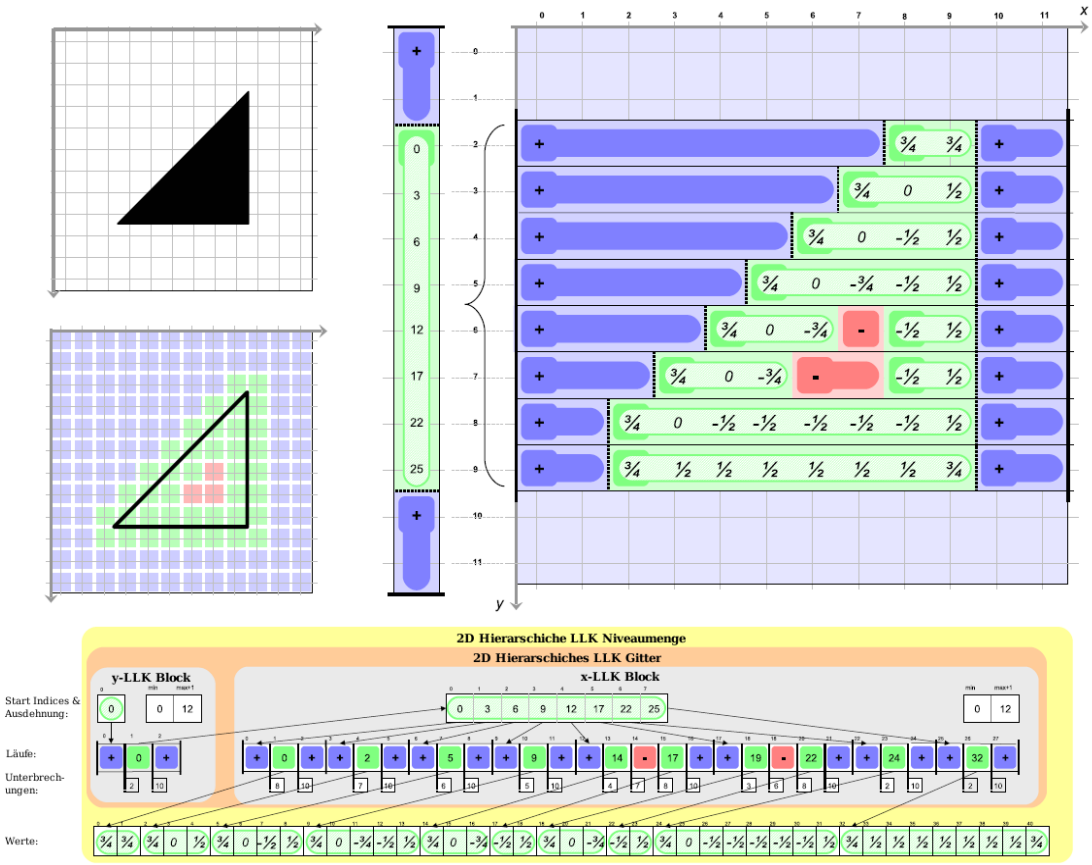

Abbildung 4.2: Kodierung eines 2D Objekts in eine 2D H-LLK Niveaumenge

Auf der linken Seite ist ein Dreieck auf einem Voxelgitter abgebildet, direkt darunter die entsprechenden Voxel. Außerhalb liegende Voxel werden blau dargestellt, innerhalb liegende rot. Die Voxel auf dem Begrenzungsbereich sind grün dargestellt.

Rechts ist eine schematische Darstellung der Kodierung dieses Objekts in eine H-LLK Niveaumenge aufgezeigt. Die Kodierung entlang der x-Achse ist auf dem 12×12 Gitter und die Kodierung entlang der y-Achse auf dem 1×12 Gitter dargestellt. Jeder lineare Durchlauf der Lauflängenkodierung entlang der x-Achse wird im x-Block gespeichert. Ausgenommen sind davon jedoch Durchläufe, die keine definierten Läufe enthalten. Im Beispiel sind das die Bereiche 0-1 und 10-11 bei der Kodierung der y-Achse. Diese wurden als komplett außerhalb klassifiziert und werden daher bei der Kodierung entlang der x-Achse nicht mit betrachtet. Somit werden Segmente die entlang einer bestimmten Kodierungsachse keinen definierten Lauf enthalten, also entweder vollständig innerhalb oder vollständig außerhalb

sind, an die darüber liegende Dimension weitergegeben. Der höhere LLK Block wird entlang der y-Achse kodiert und ergibt den dargestellten y-LLK Block.
Die definierten Werte der y-Achsen Kodierung fungieren als Offsets für die entsprechenden Segmente der x-Achsen Kodierung. Die definierten Werte der x-Achsen Kodierung sind die eigentlichen Werte der Voxel und entsprechen dem euklidischen Abstand der Voxel zur Objektgrenze.

Das untere Bild zeigt nun die aus der schematischen Darstellung resultierende 2D H-LLK Niveaumenge. Die definierten Werte der y-Achsen Kodierung werden dabei im Start Indizes Array des x-Blocks gepeichert. Die definierten Werte der x-Achsen Kodierung, also die eigentlichen Werte der Voxel, werden in einem speziellen Werte Array gespeichert.

Wie man sieht, speichern die LLK Blöcke die Läufe sequentiell. Die definierten Werte des x-Blocks sind Indizes in das Werte-Array und verweisen auf den Wert des Voxels eines definierten Laufs im Werte-Array. Die definierten Werte im y-Block hingegen verweisen auf die Start Indizes des x-Blocks. Somit ist erkennbar, daß der niedrigste LLK Block die Werte der Voxel kodiert, während alle höheren Blöcke das Ergebnis der vorangegangenen Lauflängenkodierung kodieren. Dadurch läßt sich die Datenstruktur sehr einfach für höhere Dimensionen verallgemeinern.

4.3 Grundlegende Algorithmen auf der H-LLK Niveaumenge

In diesem Abschnitt werden die wichtigsten Algorithmen für die H-LLK Niveaumenge beschrieben, welche in erster Linie den Zugriff auf die Elemente der Datenstruktur sowie die Neuordnung der Niveaumenge beschreiben. Im Anhang befindet sich eine genauere Beschreibung dieser Algorithmen in Pseudocode.

4.3.1 Direktzugriff

Ein schneller Zugriff auf einzelne Elemente ist für viele Applikationen der Computergrafik wichtig, wie zum Beispiel bei der Kollisionsdetektion oder beim Ray Tracing. Der Direktzugriff auf die H-LLK Datenstruktur arbeitet rekursiv:

(1) Der Direktzugriff auf dem obersten LLK Block wird mit zwei Parametern gestartet: einem Vektor mit den Koordinaten des gefragten Gitterpunkts $Q = (q_1, ..., q_n)$ und einem Segment Index *si*, welcher zu Beginn mit Null initialisiert wird. Von dem Koordinatenvektor wird nun die oberste Koordinate q_n abgespalten.

(2) Über den gegebenen Segment Index läßt sich nun der Index des ersten Laufs in dem Segment, die erste Unterbrechung sowie die Anzahl der Läufe in dem Segment (die Segmentlänge) bestimmen. Der Lauf, welcher die gesuchte Koordinate q_i enthält, läßt sich nun mittels Binärer Suche innerhalb der Grenzen des Segments bestimmen. Falls der Lauf negativ oder positiv ist, wird dessen Lauf-Kodierung zurückgeliefert. Ansonsten berechnet man den Index für die definierten Daten. Dabei dient der von dem Lauf gelieferte Index als Offset und man erhält den gesuchten Index, indem man dieses Offset mit q_i addiert und die Koordinate der letzten Unterbrechung vor dem Lauf subtrahiert.

(3) Falls noch ein LLK Block in einer hierarchisch untergeordneten Dimension existiert, wird der Direktzugriff für diesen mit dem restlichen Koordinatenvektor und dem vorher berechneten Index als Segment-Index aufgerufen. Ansonsten wird der Index für die definierten Daten, der dann als Zeiger in das Werte-Array dient, zurückgegeben.

4.3.2 Sequenzieller Zugriff

Da viele Schemata zur Deformation von Oberflächen sequenziell auf eine Niveaumenge zugreifen, wurde die H-LLK Niveaumenge ähnlich wie das DT Gitter besonders für diese Anwendungen optimiert. Der sequenzielle Zugriff arbeitet ebenfalls rekursiv auf der Datenstruktur und iteriert entsprechend der Kodierungsachsen über die Daten:

Der sequenzielle Zugriff wird ähnlich wie der Direktzugriff als Prozeduraufruf im obersten LLK Block gestartet. Dabei werden zwei Parameter übergeben: ein Segment Index *si* initialisiert mit Null und ein leerer Vektor für Eltern-Koordinaten.

Die Prozedur beginnt nun, die Läufe des gegebenen Segments zu traversieren. Nicht-definierte Läufe werden ignoriert. Wird ein definierter Lauf gefunden, werden seine Länge und seine Startkoorinate entlang der aktuellen Kodierungsachse bestimmt. Anschließend wird über alle Koordinaten des Laufs iteriert:

Zu jeder Koordinate wird der Index für die definierten Daten berechnet (siehe Abschnitt 4.3.1 Direktzugriff). Desweiteren wird ein Vektor der aktuellen Koordinaten erzeugt, welcher aus den jeweiligen Koordinatenvektoren besteht. Diesen Koordinatenvektor erhält man, indem der Vektor der Eltern-Koordinaten mit der jeweils aktuellen Koordinate vereint wird. Falls der aktuelle LLK Block der unterste ist, wird zum Ergebnis der aktuelle Koordinatenvektor und dessen zugehöriger Index der definierten Daten hinzu gefügt.
Ansonsten wird die Prozedur rekursiv für die nächst niedrigere Dimension unter Verwendung des Index für die definierten Daten als Segment-Index und dem aktuellen Koordinatenvektor als Eltern-Koordinatenvektor aufgerufen und der dadurch erhaltene Rückgabewert zum Ergebnis hinzu gefügt.

Der sequenzielle Zugriff liefert schließlich eine Liste aller definierten Gitterpunkte sowie ihre zugehörigen Indizes in das Werte-Array zurück.

4.3.3 Zugriff auf Nachbarn

Für viele Anwendungen ist ein effizienter Zugriff auf benachbarte Gitterpunkte von großer Bedeutung, wie zum Beispiel bei der *Fast Marching Methode* [Set96].

Die rekursive Struktur der H-LLK Niveaumenge läßt sich für eine effiziente Suche nach Nachbarn ausnutzen. In der niedrigsten Dimension können Nachbarn in konstanter Zeit ermittelt werden, da dazu nur das entsprechende Segment, in dem sich der gegebene Gitterpunkt befindet, durchsucht werden muss.
Bei höheren Dimensionen gestaltet sich die Suche ähnlich einfach. Die Nachbarn werden mittels Binärer Suche nach der Koordinate der niedrigsten Dimension in den durch die anderen Koordinaten angegebenen entsprechenden benachbarten Segmenten ermittelelt.

Als Beispiel wird gezeigt, wie die Achternachbarschaft in 2D anhand des Punktes (4,7) in Abbildung 4.2 ermittelt wird. Zunächst wird mittels Binärer Suche im Segment mit der y-Koordinate 7 nach dem Lauf gesucht, der die x-Koordinate 4 enthält. Auf die zwei Nachbarn in diesem Segment kann nun in konstanter Zeit mittels Inkrementieren bzw. Dekrementieren des Index in das Werte-Array auf deren definierte Werte zugegriffen werden, da sie ebenfalls

im definierten Lauf von (4,7) liegen.
Für die benachbarten Segmente mit den y-koordinaten 6 und 8 wird ähnlich verfahren. Mittels Binärer Suche nach der x-Koordinate 4 in den zwei Segmenten werden die Punkte (4,6) und (4,8) als direkte Nachbarn ermittelt. Für die Achternachbarschaft werden nun noch die Nachbarn dieser zwei Punkte entlang der x-Achse benötigt. Auf diese kann wiederum in konstanter Zeit ausgehend von den zwei vorher ermittelten Nachbarn zugegriffen werden.

4.3.4 Neuordnung der H-LLK Niveaumenge

Bei der Neuordnung einer Niveaumenge (engl. Rebuilding) soll gewährleistet werden, daß zu jedem Zeitpunkt alle Gitterpunkte der Nullstellenmenge (Ausgangskontur, siehe Kapitel 2.3) sowie ein ausreichend großes umgebendes Band enthalten sind, damit die Finiten-Differenzen-Operatoren, wie sie in den Berechnungen für Niveaumengen verwendet werden, unterstützt werden. Die Neuordnung der Niveaumenge wird also benötigt, wenn der Begrenzungsbereich der Niveaumenge zu klein wird. Im Anschluß an die Neuordnung wird außerdem noch eine Reinitialisierung der Niveaumenge vorgenommen, um die Werte der definierten Daten korrekt entsprechend einer vorzeichenbehafteten Abstandsfunktion zu setzen.

Die Neuordnung setzt sich im Prinzip aus zwei Schritten zusammen. Im ersten Schritt werden alle Gitterpunkte, die zu weit von der Objektgrenze entfernt sind (nummerischer Wert des Voxels ist größer einem gegebenen Wert $dist$ oder kleiner $-dist$), aus der Datenstruktur gelöscht. Im zweiten Schritt wird die Untermenge der verbliebenen Gitterpunte dilatiert. Der dafür verwendete Dilationsalgorithmus wird im folgenden Unterpunkt beschrieben.

Dilationsalgorithmus

Da sich eine direkte Dilation als ineffizient und langsam erwiesen hat, haben die Autoren der H-LLK Niveaumenge einen neuen Algorithmus entworfen, welcher den rekursiven Aufbau der Datenstruktur besser ausnutzt. Dieser Algorithmus arbeitet ebenfalls rekursiv und läßt sich folgendermaßen beschreiben:

Zu Beginn wird auf dem obersten LLK Block eine 1D Dilation ausgeführt. Um nun den nächst tieferen LLK Block zu dilatieren, werden die Koordinaten der definierten Läufe des dilatierten LLK Blocks iterativ durchgegangen:

Die aktuelle Koordinate sei y_m. Das korrespondierende LLK Segment der hierarchisch untergeordneten Dimension wird nun erzeugt, indem man zuerst alle Segmente im Bereich $y_m - n$ bis $y_m + n$ (wobei n die Anzahl der Dilations-Punkte ist) im originalen LLK Block dieser Dimension dilatiert.
Im zweiten Schritt wird nun aus diesen individuell erzeugten LLK Segmenten durch eine Vereinigung (Union) ein einzelnes dilatiertes Segment erzeugt, welches dann zum neuen di-

latierten LLK Block der hierarchisch untergeordneten Dimension hinzu gefügt wird.

Dieses Verfahren wird nun rekursiv auf alle LLK Blöcke in der hierarchisch untergeordneten Dimension angewandt und endet, wenn alle definierten Elemente des obersten LLK Blocks abgearbeitet sind.

Zum Abschluss der Dilation wird nun ein neues Werte-Array erzeugt. Bereits existierende Gitterpunkte erhalten ihren ursprünglichen Wert. Neue Gitterpunkte werden initialisiert.

Zum besseren Verständnis der Funktionsweise des Dilationsalgorithmus folgt nun eine genauere Beschreibung der 1D Dilation sowie des verwendeten Union-Algorithmus zum Vereinigen zweier H-LLK Datenstrukturen gleicher Dimension.

1D Dilation

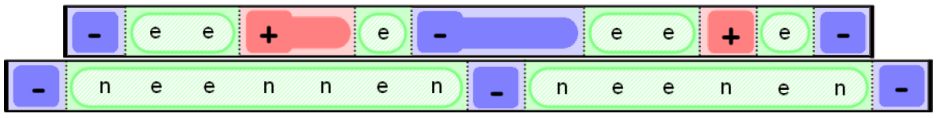

Abbildung 4.3: Die 1D Dilation veranschaulicht als Grafik

Der 1D Dilationsalgorithmus ist recht einfach aufgebaut und besteht aus zwei Schritten:

Im ersten Schritt werden alle definierten Läufe des LLK Segments unabhängig dilatiert. Es wird also die Unterbrechung vor dem Lauf und die Unterbrechung nach dem Lauf (als dessen Grenzen) um die Anzahl der Dilationspunkte n verkleinert bzw. vergrößert.

Anschließend wird ein neues LLK Segment geformt, indem die dilatierten Läufe vereint werden. Die verbleibenden negativen oder positiven Läufe werden dabei konsistent mit dem originalen Segment gehalten. Wurde die minimale oder maximale Koordinate durch einen dilatierten definierten Lauf überschritten, gibt es zwei Möglichkeiten. Falls der erste bzw. letzte Lauf definiert war, wird Min bzw. Max einfach angepaßt. War der Lauf nicht definiert, wird ein entsprechender Lauf der Länge n an die entsprechende Stelle eingefügt.

Union-Algorithmus

Die von der Dilation benötigte Vereinigung zweier H-LLK Datenstrukturen unter Berücksichtigung der korrespondierenden Segmente tieferer Dimensionen hat sich als recht komplex erwiesen und wird daher zum besseren Verständnis hier beschrieben. Vom Prinzip her funktioniert eine Vereinigung von zwei H-LLK Datenstrukturen folgendermaßen:

Als Eingabe erhält der Algorithmus eine Liste von H-LLK Datenstrukturen der selben Dimension, die vereint werden sollen. Falls die Liste nur ein Element enthält, endet der Algorithmus und dieses eine Element wird unverändert zurückgegeben.

Es werden nun die ersten zwei Elemente aus der Liste betrachtet. Zu Beginn werden Min und Max dieser beiden LLK Segmente verglichen und das kleinere Min bzw. größere Max als Ausdehnung für das neue Segment gewählt. Nun werden die einzelnen Läufe der beiden Segmente iterativ miteinander verglichen. Dabei wird ein neues Segment folgendermaßen gebildet:

Folgende drei Fälle müssen unterschieden werden:

1. Sind beide ausgewählte Läufe definiert, wird ein definierter Lauf und die größte der beiden Unterbrechungen zum Rückgabe-Segment hinzu gefügt. Falls ein LLK Block hierarchisch untergeordneter Dimension existiert, werden die einzelnen Koordinaten zwischen der letzten und der aktuellen Unterbrechung des Rückgabe-Segments betrachtet. Dabei kann die Unterbrechung des kürzeren Laufs überschritten werden, so daß bei dem betroffenen Eingabe-Segment zum nächsten Lauf gewechselt werden muß.
   Falls die jeweiligen Läufe der Eingabe-Segmente beide definiert sind, werden die zu der aktuellen Koordinate korrespondierenden hierarchisch untergeordneten LLK Segmente ermittelt und mittels der Union Operation vereint. Das Ergebnis wird dann zum LLK Block des Rückgabe-Segments hinzu gefügt. Ist einer der beiden Läufe nicht definiert, werden die korrespondierenden hierarchisch untergeordneten LLK Segmente des definierten Laufs unverändert übernommen.

2. Ist der Lauf des einen Segments definiert, der des anderen aber nicht, wird zum Rückgabe-Segment ein definierter Lauf sowie die Unterbrechung des LLK Segments, in dem sich dieser definierte Lauf befindet, hinzugefügt. Falls ein LLK Block in einer hierarchisch untergeordneten Dimension existiert, wird ähnlich verfahren wie oben.

3. Ist keiner der beiden Läufe definiert, wird die kleinste Unterbrechung zum Rückgabe-Segment hinzu gefügt. Ist ein Lauf davon negativ (innerhalb des Objekts), wird zum Rückgabe-Segment ein negativer Lauf hinzu gefügt, ansonsten ein positiver Lauf.

Es wird nun gegebenfalls in den beiden Segmenten jeweils zum nächsten Lauf gesprungen, falls dessen Unterbrechung kleiner oder gleich der letzten Unterbrechung im Rückgabe-Segment ist.
Ist die letzte Unterbrechung des Rückgabe-Segments größer als der Max-Wert eines der Eingabe-Segmente aber kleiner als der Max-Wert des anderen, so werden die verbleibenden Läufe von diesem grösseren Segment zum Rückgabe-Segment hinzu gefügt.

Existieren in der Liste noch weitere Elemente, wird nun das nächste LLK Segment aus der Liste mit dem durch den Algorithmus neu erzeugtem LLK Segment vereint. Dies wird solange fortgesetzt, bis die Liste leer ist.

5 Vergleich von Zugriffszeiten und Speicherbedarf der Datenstrukturen

In diesem Abschnitt werden Speicherbedarf und Zugriffszeiten des Direktzugriffs auf einzelnen Elemente sowie des sequenziellen Zugriffs auf alle definierten Daten der Datenstrukturen miteinander verglichen.

Die Implementation des Perfekten Räumlichen Hashing läuft komplett in Software und unter Verwendung von dreidimensionalen Arrays für die Hash-Tabellen anstatt von Texturen. Eine Implementierung auf der Grafikhardware, wie von den Autoren der Datenstruktur vorgeschlagen, würde noch bessere Zugriffszeiten ermöglichen.

Getestet wurde auf einem Intel Pentium M 725 1.6 GHz, 1280 MB RAM, ATI Radeon 9700 mobile mit 64 MB Videospeicher.
Betriebssystem war ein Gentoo Linux mit Kernel 2.6.17-gentoo-r4, glibc: 2.4-r4, Qt: 4.1.4-r2 und GCC: 3.4.6.

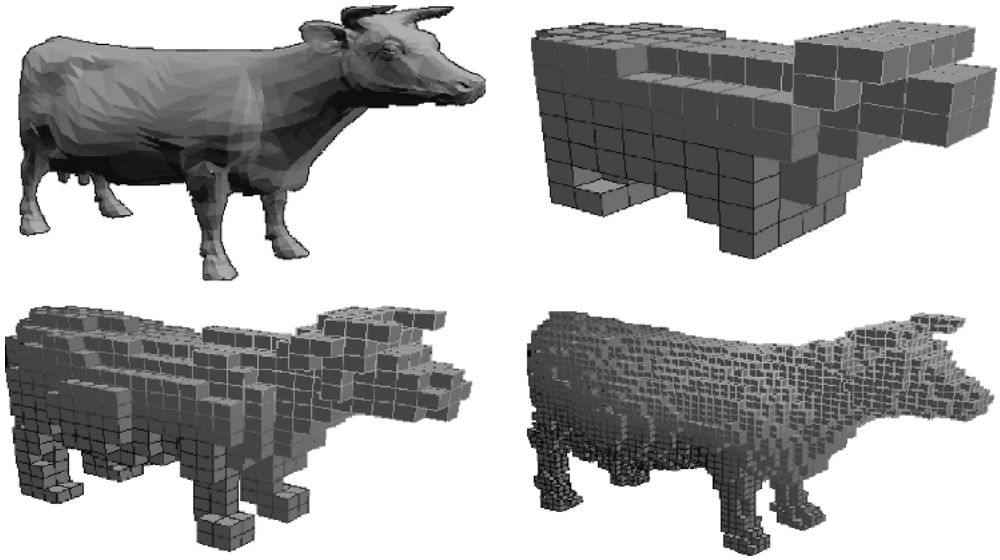

Abbildung 5.1: Das Kuh-Modell als Original und in verschiedenen Voxel-Auflösungen

5.1 Speicherbedarf

Speicherbedarf des Kuhmodells in verschiedenen Auflösungen:

Bounding Box	# def. V.	Voxelgitter	H-LLK	DT Gitter	PR Hashing	S.LLK
69x47x30	3750	1,48 MB	63,6 KB	63,4 KB	135,6 KB	89,6 KB
118x80x51	10630	7,35 MB	178,8 KB	180,3 KB	579,1 KB	256,2 KB
171x116x74	22484	22,4 MB	376,8 KB	377,9 KB	1,65 MB	542,4 KB
203x137x88	31511	37,34 MB	529,4 KB	529,7 KB	2,65 MB	762,7 KB
228x154x99	39752	53,04 MB	670,3 KB	667,6 KB	3,76 MB	966,3 KB
247x167x107	46580	67,35 MB	785 KB	783,4 KB	4,72 MB	1,11 MB
265x179x115	53751	83,24 MB	905,5 KB	901,5 KB	5,8 MB	1,28 MB
287x194x124	63034	105,35 MB	1,04 MB	1,03 MB	7,3 MB	1,49 MB
300x203x130	68920	120,8 MB	1,13 MB	1,13 MB	8,3 MB	1,64 MB
312x211x135	74527	135,61 MB	1,23 MB	1,22 MB	9,3 MB	1,77 MB
323x218x140	79946	150,42 MB	1,32 MB	1,31 MB	10,33 MB	1,9 MB

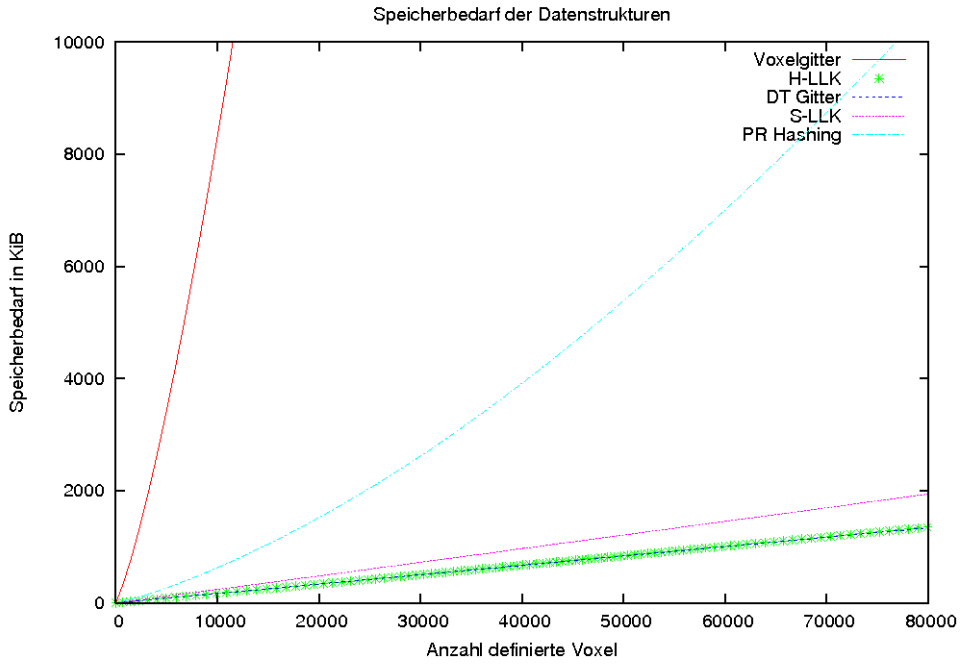

Abbildung 5.2: Speicherbedarf der verschiedenen Datenstrukturen in Bezug zu den relevanten Daten

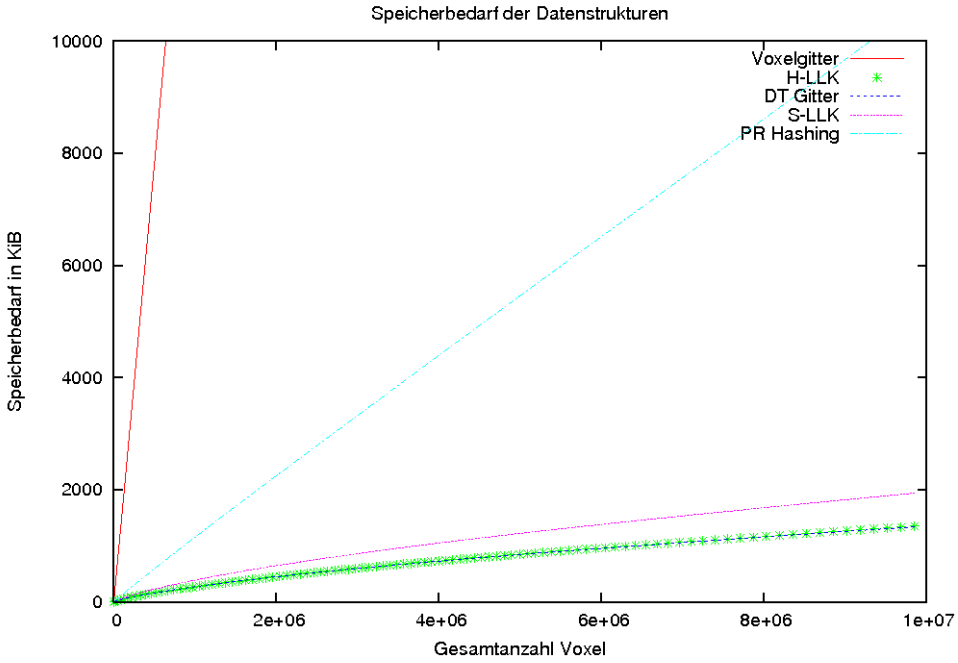

Abbildung 5.3: Speicherbedarf der verschiedenen Datenstrukturen

Wie in den Abbildungen 5.2 und 5.3 ersichtlich, liegen DT Gitter und H-LLK Niveaumenge nahezu gleichauf. Etwas mehr Speicher benötigt die Spärliche LLK Niveaumenge. Im Vergleich mit dem vollständigen Voxelgitter und dem Perfekten Räumlichen Hashing benötigen die drei Datenstrukturen wenig Speicher und bei größeren Datenmengen steigt der Speicherbedarf annähernd linear an. Dagegen steigt der Speicherbedarf für das vollständige Voxelgitter und für die Hash-Tabellen beim Perfekten Räumlichen Hashing in Bezug auf die Anzahl Voxel mit relevanten Daten fast quadratisch an, da auch für Voxel ohne relevante Daten Speicherplatz benötigt wird. Beim PR Hashing wird zwar deutlich weniger Speicher benötigt als beim vollständigen Voxelgitter, aber immer noch weitaus mehr als bei den anderen.

5.2 Direktzugriff

In der nachfolgenden Tabelle sind die durchschnittlichen Zugriffszeiten auf ein Element in der jeweiligen Datenstruktur aufgeführt:

Bounding Box	# def. V.	Voxelgitter	H-LLK	DT Gitter	PR Hashing	S.LLK
69x47x30	3750	1,06 μs	11,28 μs	12,41 μs	1,19 μs	12,64 μs
118x80x51	10630	1,09 μs	11,17 μs	12,4 μs	1,17 μs	12,64 μs
171x116x74	22484	1,08 μs	11,11 μs	12,4 μs	1,16 μs	12,83 μs
203x137x88	31511	1,11 μs	11,3 μs	12,37 μs	1,16 μs	12,81 μs
228x154x99	39752	1,12 μs	10,08 μs	12,23 μs	1,15 μs	12,99 μs
247x167x107	46580	1,13 μs	10,99 μs	12,25 μs	1,15 μs	13,11 μs
265x179x115	53751	1,14 μs	10,99 μs	12,26 μs	1,15 μs	13,11 μs
287x194x124	63034	1,14 μs	11,01 μs	12,47 μs	1,15 μs	13,1 μs
300x203x130	68920	1,14 μs	10,88 μs	12,32 μs	1,15 μs	13,16 μs
312x211x135	74527	1,14 μs	11,06 μs	12,55 μs	1,14 μs	13,19 μs
323x218x140	79946	1,14 μs	11,1 μs	12,22 μs	1,14 μs	13,2 μs

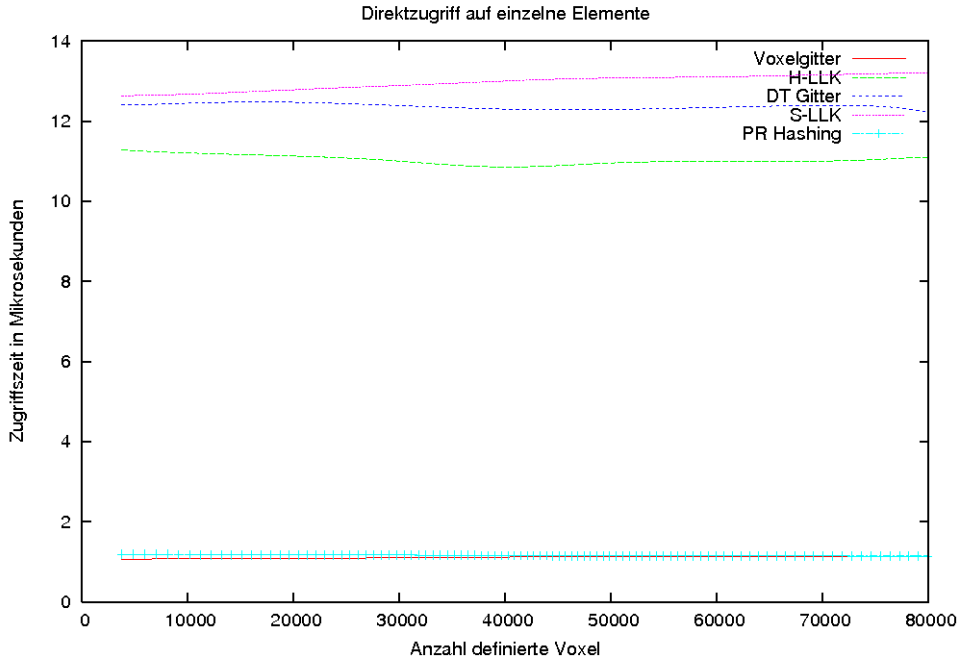

Abbildung 5.4: Direktzugriff auf die Datenstrukturen

Wie aus der Abbildung 5.4 zu erkennen ist, wird der Direktzugriff bei allen fünf Datenstrukturen in fast konstanter Zeit unabhängig von der Menge der Daten realisiert. Lediglich bei

der Spärlichen LLK Niveaumenge ist ein geringer Anstieg zu verzeichnen, bedingt durch die relativ aufwendige Suche nach dem richtigen Segment. Bei der H-LLK Niveaumenge, der Spärlichen LLK Niveaumenge sowie dem DT Gitter beträgt der Direktzugriff etwa das zehnfache der Zeit, die beim vollständigen Voxelgitter benötigt wird. Beim Perfekten Räumlichen Hashing läßt sich eine fast so gute Zugriffszeit wie beim vollständigen Voxelgitter erreichen.

5.3 Sequenzieller Zugriff

Der sequenzielle Zugriff auf die Daten ist für diverse Operationen auf Niveaumengen notwendig. Dabei wird entsprechend der Koordinatenrichtungen sequenziell auf die Daten zugegriffen. Die H-LLK Niveaumenge und das DT Gitter wurden speziell für diesen Anwendungsfall optimiert. Somit ist es möglich, ausschließlich auf die definierten Daten zuzugreifen, während bei den anderen Datenstrukturen über das komplette Gitter iteriert werden muß.

Im Test wurde die Zeit gemessen, die benötigt wird, bis der Zugriff sämtliche definierte Voxel inklusive ihrer nummerischen Werte ermittelt hat. In der nachfolgenden Tabelle sind die Zugriffszeiten in der jeweiligen Datenstruktur aufgeführt.

Bounding Box	# def. V.	Voxelgitter	H-LLK	DT Gitter	PR Hashing	S.LLK
69x47x30	3750	103,44 ms	39,16 ms	41,62 ms	116,74 ms	71,8 ms
118x80x51	10630	528,33 ms	114,1 ms	119,33 ms	564,71 ms	238,29 ms
171x116x74	22484	1,59 s	246,12 ms	253,81 ms	1,7 s	482,11 ms
203x137x88	31511	2,73 s	328,83 ms	343,46 ms	2,83 s	672,3 ms
228x154x99	39752	3,89 s	432,81 ms	451,2 ms	3,99 s	844,98 ms
247x167x107	46580	4,99 s	493,35 ms	522,95 ms	5,07 s	986,41 ms
265x179x115	53751	6,2 s	561,52 ms	594,61 ms	6,25 s	1,14 s
287x194x124	63034	7,86 s	658,3 ms	700,27 ms	7,91 s	1,37 s
300x203x130	68920	9,02 s	777,43 ms	815,73 ms	9,07 s	1,5 s
312x211x135	74527	10,12 s	839,64 ms	899,19 ms	10,17 s	1,58 s
323x218x140	79946	11,22 s	879,1 ms	914,18 ms	11,27 s	1,71 s

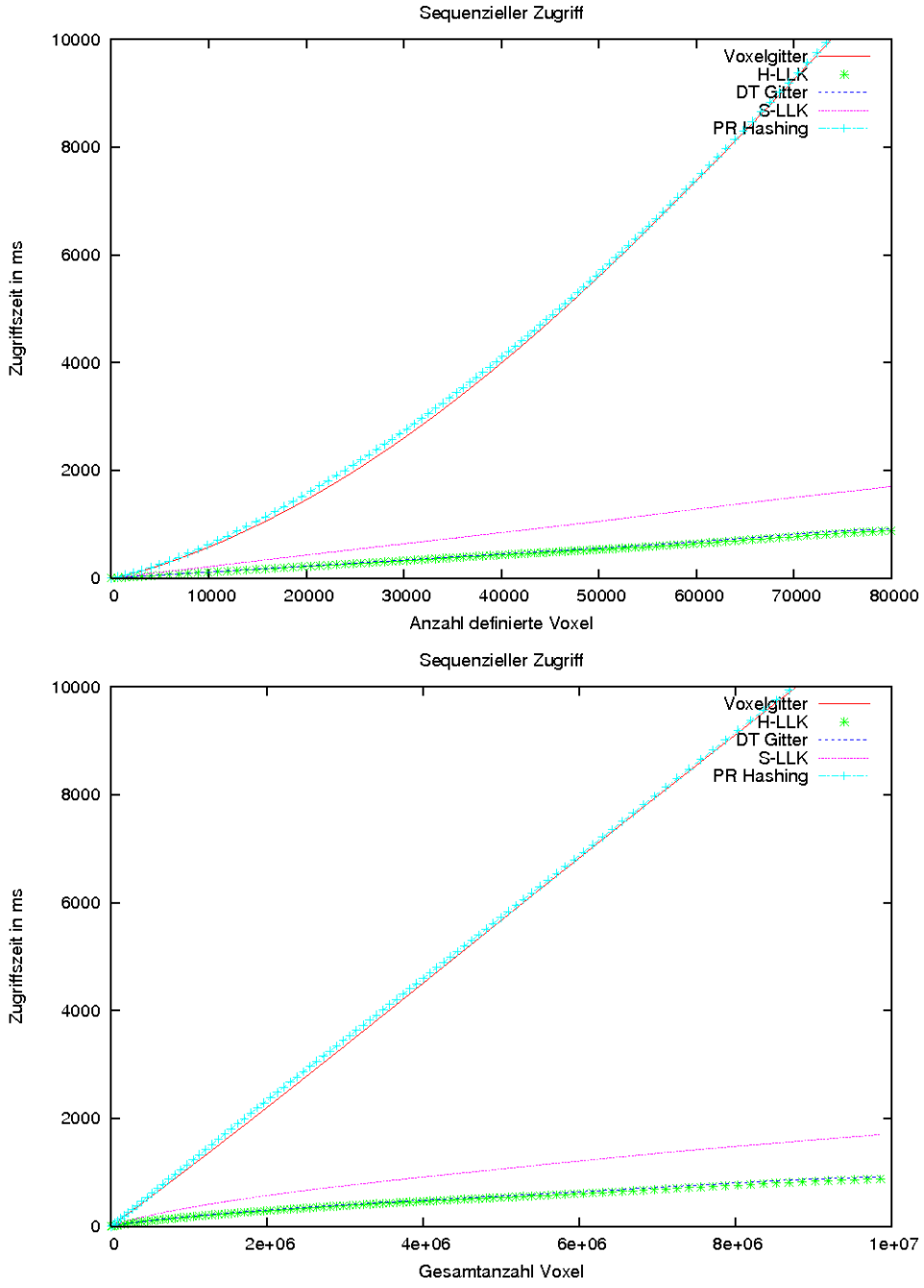

Abbildung 5.5: Sequenzieller Zugriff auf die Datenstrukturen

Durch die Optimierung der H-LLK Datenstruktur und des DT Gitters für den sequenziellen Zugriff sind diese zwei Datenstrukturen deutlich schneller als das vollständige Voxelgitter, das PR Hashing oder die Spärliche LLK Niveaumenge. Da der Zugriff bei der H-LLK Niveaumenge und dem DT Gitter ausschließlich direkt auf den definierten Daten erfolgt, steigt der Zeitaufwand linear entsprechend der Anzahl der definierten Voxel.
Bei der vollständigen Voxelmenge und den Hash-Tabellen muß hingegen über sämtliche Gitterpunkte iteriert werden. Ihre Zugriffszeiten steigen somit linear entsprechend der Gesamtzahl der Voxel.
Für die Spärliche LLK Niveaumenge wurde auch ein optimierter sequenzieller Zugriff verwendet. Bei dieser Datenstruktur ist es allerdings immer noch notwendig, über alle Läufe zu iterieren, da die definierten Läufe nicht direkt adressiert werden können. Da nicht über jeden Gitterpunkt iteriert werden muß sondern nur über die Läufe, läßt sich zwar ein schnellerer sequenzieller Zugriff erreichen als bei der vollständigen Voxelmenge, trotzdem ist die Datenstruktur für diese Aufgabe deutlich langsamer als die H-LLK Niveaumenge oder das DT Gitter.

6 Zusammenfassung

In dieser Studienarbeit wurden fünf Datenstrukturen für volumetrische Objekte vorgestellt und miteinander verglichen. Dabei zeigte sich, daß jede der Datenstrukturen ihre Vor- und Nachteile hat und sich somit für spezifische Anwendungen mehr oder weniger eignen kann.

Ausgangspunkt für die Vergleiche war die vollständige Voxelmenge, bei der Daten zu jedem Voxel in einem d-dimensionalen Array gespeichert werden. Sie bietet die größte Flexibilität bei Änderungen der Daten und der Zugriff auf die Daten ist immer in konstanter Zeit möglich, da der jeweilige Gitterpunkt direkt adressiert werden kann. Außerdem kann man über die Ausdehnung des Arrays eine explizite Bounding Box festlegen, was aber nicht unbedingt sein muß. Eine Innen/Außen-Klassifikation der Voxel außerhalb des Begrenzungsbereichs ist ebenfalls möglich. Diese Vorteile erkauft man sich mittels eines enormen Speicherbedarf, da für jeden Gitterpunkt Speicherplatz benöigt wird, unabhängig davon, ob dieser relevante Daten enthält.

Der Schwerpunkt bei dem Vergleich lag auf der Hierarchischen LLK Niveaumenge, welche sich als sehr flexibel bei Änderungen an den Daten gezeigt hat, die Daten aber trotzdem speichereffizient verwaltet und einen schnellen Zugriff auf sie gewährleistet. Außerdem nimmt sie als Niveaumenge eine Innen/Außen-Klassifikation der Bereiche der Voxelmenge vor, die nicht auf dem Begrenzungsbereich liegen. Somit ist eine Darstellung nicht-geschlossener und geschlossener Objekte möglich. Dank der Max und Min Ausdehnungen wird desweiteren eine explizite Bounding Box festgelegt, welche eine zusätzliche Berechnung zum Beispiel bei der Kollisionsdetektion unnötig macht.
Das DT Gitter, auf dem die H-LLK Niveaumenge zum größten Teil basiert, hat zwar ähnlich gute Zugriffszeiten und geringen Speicherbedarf, weist aber einige grobe Nachteile auf. So ist weder eine explizite Bounding Box gegeben, noch geht aus den Verbundskomponenten hervor, ob die angrenzenden Bereiche innerhalb oder außerhalb des Objekts liegen. Eine Darstellung nicht-geschlossener Objekte ist somit nicht möglich, diese müssen erst geschlossen werden (siehe Abbildung 6.1).

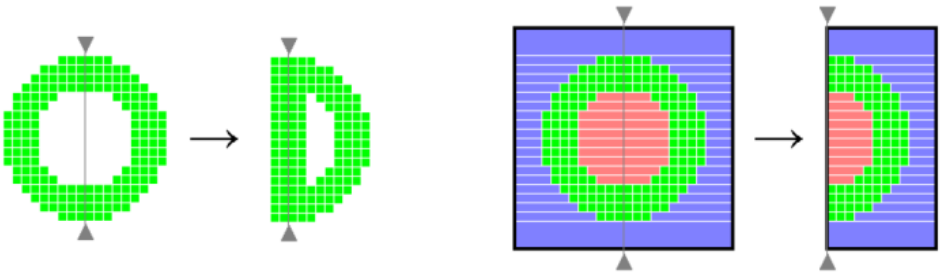

Abbildung 6.1: Beispiel für die Speicherung einer nicht-geschlossenen Fläche: ein Objekt wird an einer Ebene geschnitten - links als DT Gitter, rechts als H-LLK Niveaumenge.

Diese Innen/Außen-Klassifikation weist hingegen die Lauflängenkodierte Spärliche Niveaumenge auf, auf dem die H-LLK Niveaumenge zum anderen Teil basiert. Größter Nachteil der Datenstruktur liegt in der geringen Flexibilität gegenüber Änderungen an den Daten, da in dem Fall die LookUp-Tabellen komplett neu berechnet werden müssen. Auch fehlt eine explizite Bounding Box.

Nahezu konstante Zugriffszeiten und damit fast auf dem Niveau einer vollständigen Voxelmenge bietet das Perfekte Räumliche Hashing, welches sich durch seine besondere Struktur besonders gut für eine Implementation auf der Grafikhardware eignet. In dem Fall sollte sich der Zugriff auf die Daten gegenüber der hier verwendeten Implementation noch effektiver durchführen lassen. Die Nachteile der Datenstruktur liegen zum einen in einem im Vergleich zur H-LLK Niveaumenge oder dem DT Gitter sehr hohen Speicherbedarf, der zwar deutlich niedriger ist als bei einer vollständigen Voxelmenge aber immer noch erheblich höher als bei der H-LLK Niveaumenge, dem DT Gitter und der Spärlichen LLK Niveaumenge. Verantwortlich dafür sind die Domäne-Bit-Tabelle und die Offset-Tabelle. Die Domäne-Bit-Tabelle besitzt die gleichen Ausdehnungen wie das originale Voxelgitter, wird aber unbedingt benötigt um festzulegen, ob ein Gitterpunkt definiert ist oder nicht. Außerdem kann die Offset-Tabelle unter Umständen sehr groß werden, um eine kollsionsfreie Verteilung der Werte in der Hash-Tabelle zu ermöglichen. Im schlechtesten Fall hat die Offset-Tabelle die gleiche Größe wie die Hash-Tabelle (sprich $r = m$).
Der zweite grosse Nachteil liegt in der statischen Vorberechnung der Hash-Tabellen, die für einen perfekten Hash notwendig ist, aber die Datenstruktur gegenüber Änderungen der Daten sehr unflexibel macht. Zwar kann der Wert eines bestimmten definierten Voxels verändert werden, bei Verformungen, also Änderungen der Koordinaten der Voxel, ist allerdings eine komplette Neuberechnung der Hash-Tabellen notwendig, welche sehr aufwendig und sehr langsam ist, vor allem durch eventuelle Neustarts der Berechnung der Hash-Tabellen, falls die Größe der Offset-Tabelle nicht ausreicht, um Kollisionen zu vermeiden.

Literaturverzeichnis

[HBN+06] Ben Housten, Christopher Batty, Ola Nilsson, Michael Nielsen und Ken Museth: *Hierarchical RLE Level Set: A Compact and Versatile Deformable Surface Representation*, *ACM Transactions on Graphics*, Bd. 25, No. 1:S. 1–24, Jan. 2006.

[HWB04] Ben Houston, Mark Wiebe und Chris Batty: *RLE sparse level sets*, in *SIGGRAPH '04: ACM SIGGRAPH 2004 Sketches*, S. 137, ACM Press, New York, NY, USA, 2004, ISBN 1-59593-896-2.

[LH06] Sylvain Lefebvre und Hugues Hoppe: *Perfect spatial hashing*, in *SIGGRAPH '06: ACM SIGGRAPH 2006 Papers*, S. 579–588, ACM Press, New York, NY, USA, 2006, ISBN 1-59593-364-6.

[NM06] Michael B. Nielsen und Ken Museth: *Dynamic Tubular Grid: An Efficient Data Structure and Algorithms for High Resolution Level Sets*, *J. Sci. Comput.*, Bd. 26(3):S. 261–299, 2006, ISSN 0885-7474.

[Set96] J.A. Sethian: *A fast marching level set method for monotonically advancing fronts*, *Proc. of the National Academy of Sciences of the USA*, Bd. 93:S. 1591–1595, Febr. 1996.

Selbstständigkeitserklärung

Hiermit erkläre ich, daß ich die vorliegende Arbeit selbstständig angefertigt, nicht anderweitig zu Prüfungszwecken vorgelegt und keine anderen als die angegebenen Hilfsmittel verwendet habe. Sämtliche wissentlich verwendete Textausschnitte, Zitate oder Inhalte anderer Verfasser wurden ausdrücklich als solche gekennzeichnet.

Chemnitz, den 10. April 2007

Gerth Holger

Anhang

A Bedienung des Testprogramms

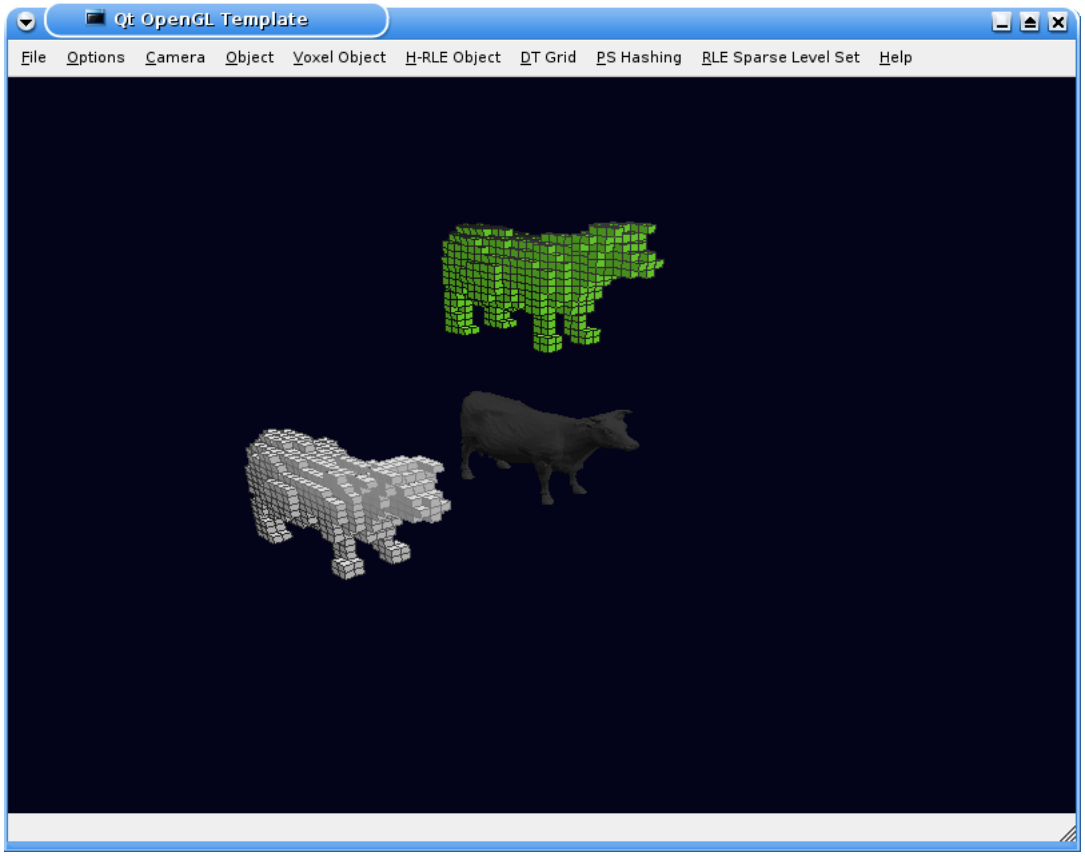

Abbildung A.1: Screenshot des Testprogramms

Die Bedienung des Programms erfolgt komplett über die verschiedenen Menüs des GUI. Über das Menü *File* kann ein Objekt im Simple Model Format (SMF) geladen und das Programm beendet werden. Dieses Modell kann dann über das Menü *Object* gerendert werden. Unter *Voxel Object* kann man aus dem Polygonmodell dann ein Voxelobjekt erzeugen. Dies kann dann in die verschiedenen Datenstrukturen unter den Menüs *H-RLE Object*, *DT Grid*, *PS Hashing* und *RLE Sparse Level Set* konvertiert werden. Unter all diesen Menüs finden sich Punkte für die Darstellung der Objekte sowie Informationen zu Zugriffszeiten und Speicherbedarf.

Die Menüs *Options* und *Camera* enthalten verschiedene Einstellungen für das Rendering und die zu verwendende Kamera.

B Algorithmen

In diesem Abschnitt werden alle wichtigen Algorithmen für die H-RLE Datemstruktur in einem C++-ähnlichen Pseudo-code beschrieben.

B.1 Allgemeine Datentypen

```
typedef struct struct_run_code {
  RUN_TYPE      type;   // RT_NEGATIVE, RT_POSITIVE oder RT_DEFINED
  unsigned int  index;  // Defined Data Index falls RT_DEFINED
} run_code;
```

```
typedef struct struct_coordinates_value {
  vector<int>   coordinates;  // Koordinatenvektor
  unsigned int  index;        // Defined Data Index falls RT_DEFINED
} coordinates_value;
```

```
typedef struct struct_neighbors {
    int                 dimension;      // auf welcher Achsenrichtung befinden sich
                                           die Nachbarn (1 = X, 2 = Y, 3 = Z usw.)
    bool                leftNeighbor;   // linker Nachbar existiert
    bool                rightNeighbor;  // rechter Nachbar existiert
    double              leftValue;      // Wert des linken Nachbarn
    double              rightValue;     // Wert des rechten Nachbarn
} neighbors;
```

B.2 Direktzugriff

Der Methode für den Direktzugriff wird ein Vektor *qv* mit Koordinaten des gewünschten Gitterpunkts übergeben sowie ein Segment Index *si*, initialisiert mit Null. Die Methode gibt den entsprechenden Lauf des Gitterpunkts zurück und, falls dieser definiert ist, den entsprechenden Index in das Werte-Array.

```
run_code randomAccess( vector<int> qv, int si = 0 ) {

  qi = qv.popBack();

  Ermittle First Run, First Break und Segment Länge des gegebenen Segments

  run_code run = BinarySearch( qi, firstRun, firstBreak );

  if ( run.type == RT_DEFINED ) {

    definedDataIndex = run.index + qi - letzter Break vor dem Run;

    if ( lower_hrle != NULL )
      return lower_hrle->randomAccess( qv, definedDataIndex )
    else run.index = definedDataIndex;
  }

  return run;
}
```

B.3 Sequenzieller Zugriff

Dem sequenziellen Zugriff wird ein zu Beginn leerer Vektor von Eltern-Koordinaten übergeben sowie ein Segment Index *si*, welcher wieder mit Null initialisiert wird. Es wird eine Liste aller definierten Gitterpunkte der Niveaumenge zusammen mit ihrem jeweiligen Index in das Werte-Array zurückgeliefert.

```
1  vector<coordinates_value> sequentialAccess( vector<int> parentCoord, int si = 0 ) {
2
3    vector<coordinate_value>  returnValue;
4    vector<int>               currentCoordinate;
5
6    Ermittle First Run und Segment Länge des gegebenen Segments
7
8    for ( i = alle Runs des Segments ) {
9
10     run_code run = runCodeArray.at(i);
11
12     Ermittle Start- und Endkoordinate des aktuellen Runs
13
14     for ( qi = alle Koordinaten des Runs ) {
15
16       currentCoordinate = parentCoord;
17       currentCoordinate.push_back( qi );
18       definedDataIndex = run.index + qi - letzter Break vor dem Run;
19
20       if ( lower_hrle == NULL ) {
21         coordinate_value  newCoordValue;
22         newCoordValue.coordinates = currentCoordinate;
23         newCoordValue.index = definedDataIndex;
24         returnValue.push_back( newCoordValue );
25       }
26       else {
27         vector<coordinate_value> tmp = lower_hrel->sequentialAccess( currentCoordinate, definedDataIndex );
28         returnValue.union( tmp );
29       }
30
31     }
32
33   }
34
35   return returnValue;
36
37 }
```

B.4 Zugriff auf Nachbarn

Bei der Methode für den Zugriff auf die Nachbarn können alle definierten direkten Nachbarn des Gitterpunktes zurückgeliefert werden oder nur die definierten Nachbarn in einer bestimmten Koordinatenrichtung. Dabei wurden folgende Festlegungen getroffen: 0 = alle Dimensionen, 1 = X-Achse, 2 = Y-Achse, 3 = Z-Achse usw. Die Methode bekommt also als Eingabe die Koordinaten des Gitterpunkts, dessen Nachbarn ermittelt werden sollen, die Koordinatenrichtung, auf der gesucht werden soll, und einen Segment Index, initialisiert mit Null.

```
1  vector<neighbors> neighborAccess( vector<int> qv, int axis, int segmentIndex = 0) {
2
3    int    qi = qv.pop_back();
4    int    qi_left = qi-1;
5    int    qi_right = qi+1;
6
7    neighbors         currentNeighbors;
8    vector<neighbors> returnValue;
9
10   Ermittle First Run, First Break und Segment Länge des gegebenen Segments
11
12   run_code run = BinarySearch( qi, firstRun, firstBreak );
13
14   if ( (axis == dimension) || (axis == 0) ) {
15
16     Ermittle die Runs (hier: nRun), in denen sich qi_left und qi_right befinden.
17
18     if ( nRun.type == RT_DEFINED ) {
19
20       if ( lower_hrle != NULL ) lower_hrle->randomAccess( qv );
21       else definedDataIndex = nRun.index + qi_(left/right) - letzter Break vor dem Run;
22
23       Aktualisiere currentNeighbors
24
25     }
26
27   }
28
29   if ( currentNeighbors.leftNeighbor || currentNeighbors.rightNeighbor )
30     returnValue.push_back( currentNeighbors );
31
32   if ( lower_hrle != NULL ) {
33     definedDataIndex = run.index + qi - letzter Break vor dem Run;
34     returnValue.union( lower_hrle->neighborAccess( qv, axis, definedDataIndex );
35   }
36
37   return returnValue;
38
39 }
```

B.5 Dilationsalgortithmus

Der Dilationsalgortithmus besteht im Prinzip aus vier Methoden: einer 1D Dilation, welches ein einzelnes LLK Segment unabhängig von der restlichen Datenstruktur dilatiert, einem Union-Algorithmus, welcher zwei LLK Segmente inklusive ihrer zugehörigen Segmente in hierarchisch untergeordneten Dimensionen vereint, der Dilation an sich, welche mit Hilfe der ersten beiden Methoden die Dilation des kompletten Objekts durchführt, und einer Methode, die schließlich die definierten Werte der Gitterpunkte initialisiert bzw. auf ihren originalen Wert setzt, falls diese in der Eingabe-Datenstruktur bereits existierten.

B.5.1 1D Dilation

Bei der 1D Dilation werden nur die Läufe und die Unterbrechungen eines einzigen LLK Segments betrachtet, unabhängig von der restlichen Datenstruktur und eventuell abhängigen LLK Segementen in einer möglichen hierarchisch untergeordneten Dimension. Übergeben werden der Funktion ein Segment Index *si* und die Anzahl der Dilationspunkte *n*. Zurück liefert die Funktion eine eindimensionale H-LLK Datenstruktur, welche das dilatierte Segment enthält.

```
1  HierarchicalRLE* dilateSegment( int si, int n ) {
2
3    HierarchicalRLE*  dilatedHRLE = new HierarchicalRLE();
4    vector<int>       dilatedCoordinates;                  // enthält die neue minimale und maximale
5                                                               Koordinate der Defined Runs des Segments
6
7    Ermittle First Run, First Break und Segment Länge des gegebenen Segments
8
9    for ( pos = alle Runs des Segments ) {
10     run = run_types.at(pos);
11
12     if ( run.type == RT_DEFINED ) {
13         dilatedCoordinates.push_back( startCoordinate - n );
14         dilatedCoordinates.push_back( endCoordinate + n );
15     }
16   }
17
18   dilatedHRLE->min = min;
19   dilatedHRLE->max = max;
20   dilatedHRLE->start_indices.push_back(0);
21
22   for (j = 0; j < dilatedCoordinates.size(); j += 2) {
23
24     int definedStartCoordinate = dilatedCoordinates.at(j);
25     int definedEndCoordinate = dilatedCoordinates.at(j+1);
26
27     Aktualisiere gegebenfalls min und max des dilatedHRLE
28
29     if ( definedStartCoordinate > dilatedHRLE->run_breaks.back() ) {
30
31       Überprüfe, welchen Run Typ die Koordinaten zwischen run_breaks.back() und definedStartCoordinate hatten
32       und füge diesen Run hinzu
33
34       dilatedHRLE->run_breacks.push_back( definedStartCoordinate );
35
36     }
37     else {
38
39       dilatedHRLE->run_breacks.pop_back();
40
41     }
42
43     run_type newRun;
44     newRun.type = RT_DEFINED
45
46     if ( definedEndCoordinate < dilatedHRLE->max )
47       dilatedHRLE->run_breaks.push_back( definedEndCoordinate );
48
49   }
50
51   return dilatedHRLE;
52
53 }
```

B.5.2 Union

Die Union erwieß sich als der aufwendigste Teil des Dilationsalgorithmus, da bei ihr das korrekte Zusammenfügen von LLK Blöcken niedrigerer Dimensionen exakt durchgeführt werden muß.
Als Eingabe erhält die Methode eine Liste von LLK Segmenten in Form von separaten H-LLK Datenstrukuren, welche mehrdimensional sein können. Diese Segmente werden während der Union zu einem einzigen zusammengefügt, welches dann zurück gegeben wird,

```
1  HierarchicalRLE* hrleUnion( vector<HierarchicalRLE*> hrleVector ) {
2
3    if ( hrleVector.size() == 1 ) return hrleVector.back();
4
5    HierarchicalRLE* segment1 = hrleVector.at(0);
6
7    for ( int i = 1; i < hrleVector.size(); i++ ) {
8
9      HierarchicalRLE* segment2 = hrleVector.at(i);
10     HierarchicalRLE* returnValue = new HierarchicalRLE();
11
12     Setze min und max von returnValue auf das grössere max bzw. min von segment1 und segment2
13     returnValue->start_indices.push_back( 0 );
14
15     Ermittle Start- und Endkoordinaten der ersten Runs der beiden Segmente
16
17     // pos1 und pos2 sind Indices ins Run Type Array der beiden Segmente
18     int pos1 = 0, pos2 = 0;
19
20     while ( ( pos1 < segment1->run_types.size()) || (pos2 < segment2->run_types.size()) ) {
21
22       run1 = segment1->run_types.at(pos1);
23       run2 = segment2->run_types.at(pos2);
24
25       if ( run1.type == run2.type ) {
26
27         if ( run1.type == RT_DEFINED ) {
28           Füge die grössere endCoord zu returnValue->run_breaks hinzu
29
30           for ( qi = alle Koordinaten des grösseren Runs ) {
31             if ( qi > kleinere endCoord ) gehe zum nächsten Run des betroffenen Segments;
32             // wenn die kleinere endCoord überschritten wird, ist der betroffene
33             // Run eventuell nicht mehr RT_DEFINED
34             if ( beide Runs == RT_DEFINED)
35               Füge Union der beiden Subsegmente zu returnValue->lower_hrle hinzu
36             else
37               Füge das Subsegment des Defined Runs zu returnValue->lower_hrle hinzu
38           }
39         }
40         else {
41           Füge die kleinere endCoord zu returnValue->run_breaks hinzu
42         }
43
44         if (returnValue->run_types.back != run1.type)
45           returnValue->run_types.push_back(run1);
46
47       }
48       else {
49
50         if ( run1.type == RT_DEFINED ) {
51           returnValue->run_breaks.push_back(endCoord1);
52
53           for ( qi = startCoord1; qi < endCoord1; qi++ ) {
54             if ( qi > endCoord2 ) pos2++;
55             // bei Überschreiten von endcoord2 könnte run2 wieder RT_DEFINED werden
56             if ( beide Runs == RT_DEFINED)
57               Füge Union der beiden Subsegmente zu returnValue->lower_hrle hinzu
58             else
59               Füge das Subsegment von run1 zu returnValue->lower_hrle hinzu
60           }
61
62           if (returnValue->run_types.back != run1.type)
```

```
63              returnValue->run_types.push_back(run1);
64            }
65
66            else if ( run2.type == RT_DEFINED ) {
67              returnValue->run_breaks.push_back(endCoord2);
68
69              for ( qi = startCoord2; qi < endCoord2; qi++ ) {
70                if ( qi > endCoord1 ) pos1++;
71                // bei Überschreiten von endcoord1 könnte run1 wieder RT_DEFINED werden
72                if ( beide Runs == RT_DEFINED)
73                  Füge Union der beiden Subsegmente zu returnValue->lower_hrle hinzu
74                else
75                  Füge das Subsegment von run2 zu returnValue->lower_hrle hinzu
76              }
77
78              if (returnValue->run_types.back != run2.type)
79                returnValue->run_types.push_back(run2);
80            }
81
82            else {
83              Füge die kleinere endCoord zu returnValue->run_breaks hinzu
84              Falls einer der Runs RT_POSITIVE, füge einen positiven Run zum returnValue hinzu,
85              ansonsen einen negativen (returnValue->run_types.back != run.type)
86            }
87
88          }
89
90          Falls endCoord eines Runs kleiner als returnValue->run_breaks.back(), gehe solange zum
91          nächsten Run (pos1++ bzw. pos2++) im jeweiligen Segment, bis endCoord grösser ist
92
93        }
94
95        if ( ( pos1 < segment1->run_types.size()) || (pos2 < segment2->run_types.size()) ) {
96          Füge verbliebene Runs des jeweiligen Segments zu returnValue hinzu
97        }
98
99        segment1 = returnValue;
100     }
101
102   return returnValue;
103 }
```

B.5.3 Aktualisiren des Werte-Array

Beim Aktualisieren des Werte-Arrays werden die definierten Gitterpunkte der bereits dilatierten H-LLK Niveaumenge mit denen der originalen verglichen. Gitterpunkte, die in beiden Niveaumengen exisitieren, werden auf ihren ursprünglichen Wert gesetzt, neue Gitterpunkte initialisiert.
Übergeben wird der Methode ein Zeiger auf die originale Datenstruktur, ein mit Null initialisierter Vektor von Elternkoordinaren und ein mit Null initialisierter Segment Index *si*.

```
1  void updateValueArray( HierarchicalRLE* originaleHRLE, vector<int> *parentCoord, int si  ) {
2
3    vector<int> tmpVector;
4
5    if ( parentCoord == NULL ) parentCoord = new vector<int>;
6
7    Ermittle First Run, First Break und Segment Länge des gegebenen Segments sowie
8    Start- und Endkoordinaten des ersten Runs
9
10   for ( run = alle Runs des gegebenen Segments ) {
11
12     if ( run.type == RT_DEFINED ) {
13       for ( qi =alle Koordinaten des Runs ) {
14
15         parentCoord->push_back( qi );
16
17         if ( this->lower_hrle != NULL ) {
18            definedIndex = tmpRun.index + qi - startCoord;
19             this->lower_hrle->updateValueArray( originalHrle, parentCoord, definedIndex );
20         }
21         else {
22
23           tmpVector.clear();
24
25           for ( int i = (parentCoord->size()-1); i >= 0; i-- )
26               tmpVector.push_back( parentCoord->at(i) );
27
28           originalRun = originalHrle->randomAccess( tmpVector );
29
30           if ( originalRun.type == RT_DEFINED )
31               defined_values.push_back(originalHrle->getDefinedValue( definedIndex des originalRun ));
32           else
33               defined_values.push_back( 1 );
34
35         }
36
37         parentCoord->pop_back();
38
39       }
40     }
41   }
42 }
```

B.5.4 ND Dilation

Die mehrdimensionale Dilation ließ sich nicht in einer Methode bewerkstelligen. Die erste Funktion steuert sozusagen mit Hilfe der vorher beschriebenen Methoden den Ablauf der gesamten Dilation und beginnt mit dem obersten LLK Block, während die zweite die Dilation rekursiv in den niedrigeren Dimensionen fortführt.

```
1  HierarchicalRLE* dialtion( int n ) {
2
3    HierarchicalRLE* dilatedHRLE = this->dilateSegment( 0, n );
4
5    dilatedHRLE = this->dilation( dilatedHRLE, n );
6
7    dilatedHRLE->updateValueArray( this );
8
9    return dilatedHRLE;
10
11 }
```

```
1  HierarchicalRLE* dialtion( HierarchicalRLE* dilatedHRLE, int n ) {
2
3    HierarchicalRLE* dilatedSubHRLE = new HierarchicalRLE();
4    HierarchicalRLE* tmpSegment, tmpSubSegment;
5    vector<HierarchicalRLE*> tmpSubsegments;
6
7    if (lower_hrle != NULL ) {
8      for ( run = alle Runs von dilatedHRLE ) {
9
10       if ( run.type == RT_DEFINED {
11         run.index = dilatedSubHRLE->start_indices.size();
12
13         for ( ym = alle Koordinaten des Runs ) {
14           tmpSubsegments.clear();      // delete subsegments from the last coordinate
15
16             for ( int qi = (ym-pointCount); qi <= (ym+pointCount); qi++) {
17
18               if ( (qi >= dilatedHRLE->min) && (qi < dilatedHRLE->max) ) {
19                 tmpRun = Run an Koordinate qi in der originalen Datenstruktur
20
21                 if ( tmpRun.type == RT_DEFINED ) {
22                   int definedIndex = tmpRun.index + qi - startOfRun;
23
24                   tmpSegment = lower_hrle->dilateSegment( definedIndex, pointCount );
25
26                   if ( this->lower_hrle->lower_hrle != NULL )
27                     tmpSegment = lower_hrle->dilation( tmpSegment, pointCount, definedIndex );
28
29                   tmpSubsegments.push_back( tmpSegment );
30                 }
31               }
32             }
33
34             tmpSubSegment = hrleUnion( tmpSubsegments );
35
36             dilatedSubHRLE->addToHRLE( tmpSubSegment );
37         }
38       }
39     }
40
41     dilatedHRLE->lower_hrle = dilatedSubHRLE;
42   }
43
44   return dilatedHRLE;
45 }
```