Fachhochschule Aachen
Abteilung Jülich
Fachbereich: Medizintechnik und Technomathematik

Berechnung und Visualisierung von Cavities in $Ge_xSb_xTe_x$-Molekülen von wiederbeschreibbaren Medien

Christoph Hahn
<c.hahn@fz-juelich.de>

Die vorliegende Diplomarbeit wurde in Zusammenarbeit mit der Forschungszentrum Jülich GmbH, Institut für Festkörperforschung, angefertigt.

Diese Diplomarbeit wurde betreut von:

Referent: Prof. Dr. rer. nat. M. Reißel (Fachhochschule Aachen)

Korreferent: Josef Heinen (Forschungszentrum Jülich)

Diese Arbeit wurde von mir selbstständig angefertigt und verfasst. Es sind keine anderen als die angegebenen Quellen und Hilfsmittel benutzt worden.

Christoph Hahn
Jülich, den 25. Februar 2009

Inhaltsverzeichnis

Kapitel 1

Einleitung / Ziel der Diplomarbeit

Der Bedarf an Speicherplatz und höhere Lese-, Schreib- und Löschgeschwindigkeiten von Speichermedien, wie DVDs, steigen in der heutigen Zeit stetig an. Vor dem Hintergrund dieser Entwicklung gilt es die zugrunde liegenden technischen Möglichkeiten weiter auszureizen und Speicherverfahren zu optimieren. Alle optischen Speichermedien funktionieren nach dem Prinzip, dass die gespeicherten digitalen Daten, Bereiche mit unterschiedlichen Reflektions- oder Transmissionsverhalten, mit einem Laser ausgelesen werden können. Bei gepressten CDs/DVDs sind dies Vertiefungen (*pits*) im Oberflächenmaterial selbst oder bei gewöhnlichen einfach beschreibbaren DVDs gebleichte Löscher in einer Farbschicht. Bei wiederbeschreibbaren Medien [DVD-RW, CD-RW und Blue-ray Disc (BD)] jedoch muss der Schreibvorgang reversibel sein. Dafür bieten sich sogenannte *Phase-Change-Materials* (Phasen-Wechsel-Legierungen) mit ihrem *order-disorder phase-change memory effect* an. Diese Legierungen bestehen meist aus den drei Elementen Tellur (Te_x), Antimon (Sb_x) und Germanium (Ge_x), kurz $Ge_xSb_xTe_x$ wobei das $_x$ die Anzahl der im Molekül enthaltenen Anteile des vorangestellten Elements angibt. Solche Legierungen haben eine *amorphe* und eine *kristalline* Phase.

(a) kristallin (b) amorph

Abbildung 1.1: Kristalline und amorphe Struktur eines $GeSbTe$-Moleküls

Die amorphe Phase reflektiert Licht weniger stark als die kristalline Phase, wodurch sie mit den *Pits* und *Lands* von gepressten CDs und DVDs vergleichbar sind. Bei Phasen-

wechsellegierungen wird somit der Wechsel von der kristallinen in die amorphe Phase und umgekehrt als logische 1 interpretiert und ein Nicht-Wechsel als logische 0. Ein Wechsel zwischen diesen beiden Phasen wird durch das Erhitzen einer gewünschten Stelle mit einen Laserstrahl erreicht. Zunächst liegt die gesamte Schicht kristallin vor und wird zum Beschreiben mit einem kurzen sehr starken Laserimpuls hoch erhitzt. Dabei werden Temperaturen erzielt, die über dem Schmelzpunkt des *phase change Materials* liegen (über einen Zeitraum von 5 Nanosekunden, 500-700°C). Anschließend kühlt sich die Stelle sehr schnell ab, so dass **keine** Kristallisation stattfinden kann. Diese dann amorphe/ungeordnete Stelle ist weniger reflektierend als die kristalline und stellt ein *pit* dar. Für den Wechsel in die kristalline Phase, erhitzt man die gewünschte Stelle mit einem längeren, aber weniger starken Laserimpuls (über ca. 50 Nanosekunden). Dabei wird zwar die Kristallisationstemperatur (gelbe gestrichelte Linie in *Abbildung* 1.2) überschritten, nicht aber die Schmelztemperatur.

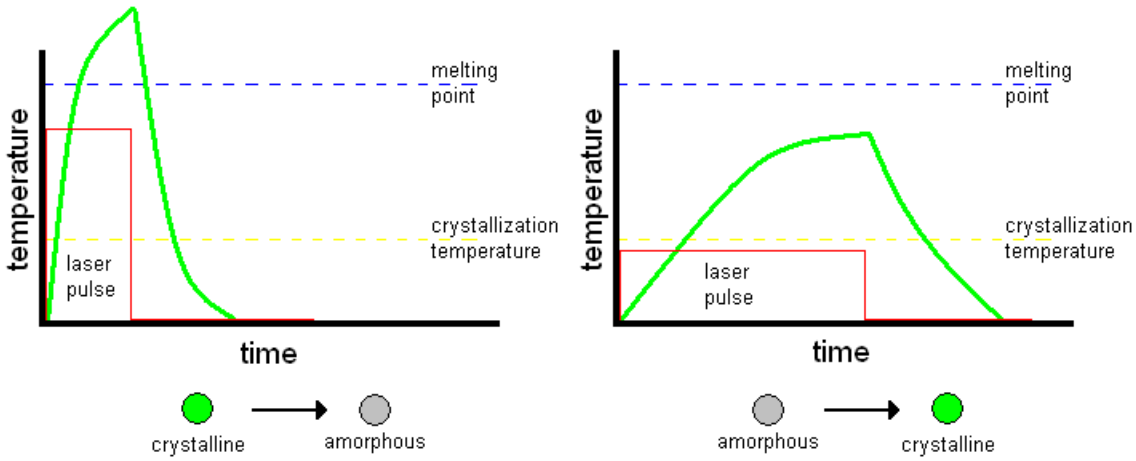


Abbildung 1.2: Wechsel zwischen den 2 Phasen kristallin/amorph

Das amorphe Material rekristallisiert dann bei einer Temperatur von 200-500°C und die Langzeitstabilität der gespeicherten Daten ist sichergestellt. Die erwähnten Zeiten, die benötigt werden, um Daten zu speichern, zu ändern oder zu löschen variieren beim Verändern der Anteile der Elemente in der Legierung.

Aus physikalischer Sicht ist es der Wissenschaft aber noch nicht gelungen, die Beziehung von Lese-/Löschgeschwindigkeit und Mischverhältnis herzuleiten. Jedoch hat man herausgefunden, dass sogenannte *cavities* oder *vacancies* (Hohlstellen/Löcher) innerhalb der Molekülstrukturen diese Zeiten beeinflussen. Verschiedene Legierungen unterscheiden sich in Anzahl und Große der Hohlstellen. Diese Unterschiede beeinflussen das Verhalten beim Wechseln zwischen den beiden Phasen und bestimmen so die Eignung

einer Legierung für die optische Speicherung. Denn nur Legierungen, die schnellere Lese-/Schreibzeiten sowie höhere Datendichten haben, sind aus wirtschaftlicher Sicht interessant.

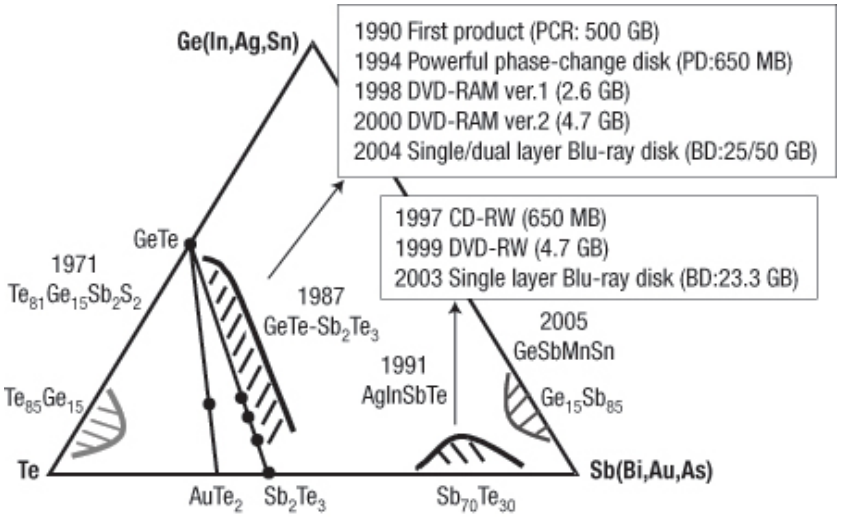


Abbildung 1.3: Legierungspyramide mit Beispielen für die Verwendung

Um das Phänomen der Hohlstellen gezielt nutzen zu können, wird momentan lediglich durch einfaches Ausprobieren verschiedener Mischverhältnisse versucht, die eben erwähnten Eigenschaften zu verbessern.
Wie in vielen Bereichen der Forschung ist es auch hier wichtig, gemessene oder simulierte Daten, in unserem Fall die Moleküle solcher $Te_xSb_xGe_x$-Verbindungen, visuell darzustellen um die Ergebnisse besser analysieren zu können. Durch die Berechnung und Visualisierung der *Cavities* sollen Wissenschaftler in die Lage versetzt werden, das Verhalten von Cavities bei mehrmaligen Wechseln zwischen der *amorphen* und der *kristallinen* Phase zu analysieren und dadurch eventuell wiederkehrende Strukturen nutzen zu können. So spielt man zum Beispiel jetzt schon mit dem Gedanken, wiederkehrende Strukturen von Cavities zur weiteren Datenspeicherung nutzbar zu machen.

Ziel meiner Diplomarbeit ist es, möglichst effizient solche Hohlstellen aus Simulationsdaten[1,2] zu berechnen und in einer 3D-Ansicht darzustellen. Weiterhin werden für die Analyse der Daten verschiedene Eigenschaften wie die Oberfläche, die Volumina und die Mittelpunkte der Cavities benötigt.

[1] [1]: J. Akola and R. O. Jones
Structural phase transitions on the nanoscale: The crucial pattern in the phase-change materials $Ge_2Sb_2Te_5$ and GeTe

[2] [2]: J. Akola and R. O. Jones
Density functional study of amorphous, liquid and crystalline $Ge_2Sb_2Te_5$: homopolar bonds and/or AB alternation?

Kapitel 2

Problemanalyse

2.1 Eingabedaten

Als Eingabedaten liegen die Positionen der Atome in einem $Ge_xSb_xTe_x$-Molekül, als X/Y/Z-Koordinaten in einer Datei mit der Endung *.xyz* vor. Die Koordinaten haben die Einheit *Ångström* ($1\mathring{A} = 10^{-7}mm = 10^{-10}m$) und bestimmen die Position relativ zum Ursprung.
In einer solchen Datei können mehrere Simulationen, sogenannte Frames eines Moleküls zu verschiedenen Zeiten abgespeichert sein. Dabei stehen mehrere Frames direkt hintereinander. Ein Frame besteht aus einer bestimmten Anzahl von Atomen, die in der ersten Zeile angegeben ist. Die zweite Zeile eines Frames dient der Identifikation desselben und kann überlesen werden. Ab der dritten Zeile folgen Angaben über Atome, beginnend mit dem Elementnamen, wie er im Periodensystem zu finden ist (*Ge*=Germanium, *Te*=Tellur, *Sb*=Antimon). Anschließend folgen die *X*-, *Y*- und *Z*-Koordinaten. Alle Daten in einer Zeile sind durch Leerzeichen voneinander getrennt. Die Eingabedateien haben folgendes Format:

Zeile 1: Anzahl der im Molekül vorhandenen Atome
Zeile 2: Frame 1 //Kommentar und wird überlesen
Zeile 3: Elementname X-Koordinate Y-Koordinate Z-Koordinate
Zeile 4: Elementname X-Koordinate Y-Koordinate Z-Koordinate
Zeile 5: Elementname X-Koordinate Y-Koordinate Z-Koordinate
⋮
Zeile X : Anzahl der im Molekül vorhandenen Atome
Zeile X+1: Frame *X* //Kommentar und wird überlesen
Zeile X+2: Elementname X-Koordinate Y-Koordinate Z-Koordinate
⋮
usw.
Die Atomradien für die verschiedenen Elemente sind konstant. Für die drei Elemente Antimon, Tellur und Germanium gilt:

- $R_{GE} = 1.25\ \mathring{A}$
- $R_{SB} = 1.33\ \mathring{A}$
- $R_{TE} = 1.23\ \mathring{A}$

Beispiel:

```
630
Step 1
Ge −11.11281967 12.57160950 3.536616564
Ge 10.09884834 −9.351238251 10.63656902
Ge 8.165349960 2.745560408 5.731359482
Ge −7.719173431 −12.12700844 −10.50668621
Ge −5.364578247 8.738575935 −2.992333412
Ge 6.885195732 1.877170324 12.93306446
Ge 10.44540691 9.887635231 −1.039365888
Ge −2.553694010 10.15852737 −10.89788723
.
.
.
```

Listing 2.1: die ersten 10 Zeilen einer Eingabedatei

Aus diesen Grunddaten sollen dann die Hohlräume berechnet und dargestellt werden.

2.2 Berechnung der Cavities

Da sich innerhalb von Phasenwechsellegierungen immer gleiche Molekülstrukturen wiederholen, lässt sich die Problemstellung mittels periodischer Randbedingung abbilden. Wie im 2D-Beispiel [2.1] zu erkennen, handelt es sich dabei immer um die gleiche Struktur, die sich an ihren Grenzen periodisch fortsetzt.

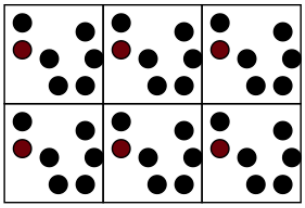


Abbildung 2.1: Periodische Randbedingung im zweidimensionalen Raum

Durch diese Eigenschaft der Phasenwechsellegierungen lässt sich die Analyse einer ganzen Legierung auf die Analyse eines Moleküls dieser Legierung reduzieren.

Die Struktur der Atome in ihrer Molekülverbindung kann (bei $GeSbTe$) durch einen Würfel abgegrenzt werden, dessen Kantenlänge der Ausdehnung des Moleküls entspricht. Die Größe dieses Würfels hängt damit von dem zu untersuchenden Molekül ab. Bei einem $Ge_xSb_xTe_x$ Molekül mit 460 Atomen hat der Würfel beispielsweise im kristallinen Zustand eine Kantenlänge von $24.05\mathring{A}$ und im amorphen Zustand $24.62\mathring{A}$. Die Kantenlänge kann dem Programm als Parameter übergeben werden.

Als nächstes müssen die Hohlräume bestimmt, deren Volumina interpoliert und dargestellt werden. Für die Erzeugung der Volumina aus den Rohdaten bietet sich der sogenannte **Marching-Cubes** Algorithmus an, der unter anderem in der Medizin verwendet wird, um aus schichtweise aufgenommenen Bildern von Körperteilen ein 3D-Modell zu erstellen.
Dieser Algorithmus interpoliert die Oberfläche einer Punktwolke durch ein Dreiecksnetz, das dann mittels OpenGL[1]-Grafikbefehlen in einer 3D-Umgebung visualisiert werden kann.
Die Hohlräume in einer vorliegenden Molekülstruktur werden durch Abarbeiten der im folgenden beschriebenen Schritte berechnet.

[1] OpenGL = (**Open** **G**raphics **L**ibrary), plattform- und programmiersprachenunabhängige Schnittstelle zur 3D-Grafikprogrammierung

2.2.1 Rasterung des Raums

Für die Berechnung der *cavities* wird der Raum in ein Raster von $N \times N \times N$ Teile aufgeteilt.
In unserem Fall genügt eine Unterteilung des Würfels mit $N = 300$, was eine Gesamtzahl der zu betrachtenden Gitterpunkte von $N^3 = 27.000.000$ ergibt und eine ausreichend hohe Auflösung bietet. Aus den Größen $N = 300$ und der Kantenlänge a des Kubus, ergibt sich eine Schrittweite von

$$step = \frac{a}{N}\mathring{A}$$

Durch die Diskretisierung des Raums in N^3 Teile lassen sich die Atome mit ihren Koordinaten, ursprünglich im *float*-Zahlenbereich, in den *Integerraum* abbilden. Dadurch werden viele rechenintensive Gleitkommaoperationen durch schnelle Integerarithmetik ersetzt. Die Abbildung in den Integerraum bringt jedoch auch einen Verlust der Genauigkeit mit sich. Je nach Auflösung variiert der Rundungsfehler und ist bei einer Auflösung von $N = 300$ schon relativ klein. Bei $N = 300$ und der Abgrenzung $a = 24\mathring{A}$ ergibt sich eine Auflösung von

$$step = \frac{a}{N} = \frac{24}{300} = 0.08\mathring{A}$$

mit einem Rundungsfehler von

$$\delta = \pm\frac{step}{2} = \pm 0.04\mathring{A}$$

2.2.2 Bestimmen der Vacancy-Domains

Da die eigentlichen Hohlstellen um deren Mittelpunkt gebildet werden, muss man zunächst diese Mittelpunkte berechnen. Ausgangspunkt dafür sind die Vacancy-Domains, denn in jeder Domain ist ein Mittelpunkt für eine spätere Hohlstelle enthalten.
Zunächst werden alle Punkte im Raum bestimmt, die von jedem Atom einen größeren Abstand als ein festes R_{ex} haben (hier ist $R_{ex} = 2,8\mathring{A}$, das sind 85% der Bindungslänge von *Ge-Te*-Verbindungen). Der Wert für R_{ex} kann genau wie der des abgrenzenden Würfels bei verschiedenen Molekülen variieren. In solchen Fällen kann ein individueller Wert als Programmparameter übergeben werden.

Die so berechneten Punkte gehören zu den Vacancy-Domains. Im roten Bereich **I** in *Abbildung 2.2*, ist der Mittelpunkt eines Beispiel-Domains eingezeichnet.

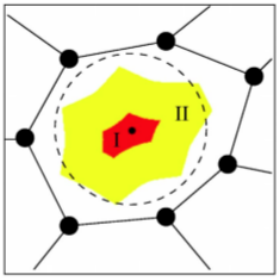


Abbildung 2.2: Schematische Form eines 2D-Vacancy

Dieser Mittelpunkt $\mathbf{P_m}$ ist definiert durch den Punkt, um den die größte Kugel (Kreis im 2D) gelegt werden kann, ohne ein umliegendes Atom zu berühren. Dafür bestimmt man als erstes für jeden Vacancy-Domain-Punkt den minimalen Abstand zu allen umliegenden Atomen:

$$\mathbf{M_{Abstände}} = \{\forall p_I \in \mathbf{P_I}, p_a \in \mathbf{P_{ATOME}} | min(dist(p_I, p_a))\}$$

Dabei ist $\mathbf{M_{Abstände}}$ die Menge aller minimalen Abstände mit den dazugehörigen Punkten. Anschließend hat man zu jedem Vacancy-Domain-Punkt die Kugel mit dem größten Radius, ohne dass ein umliegendes Atom berührt wird. Davon bestimmt man das Maximum und der zu diesem Abstand zugehörige Punkt wird als Mittelpunkt für das Domain benutzt.

$$\mathbf{P_m} = max(\mathbf{M_{Abstände}})$$

2.2.3 Komplettes Volumen bestimmen

Das komplette *Cavity*-Volumen kann dann im Anschluss an die Bestimmung der Mittelpunkte der Domains berechnet werden. Die hierfür benötigten Mittelpunkte und Atome, welche die Domains erzeugen, stehen für die Berechnung in Listen/Vektoren bereit. Um nun die Bildungsvorschrift der Volumina zu verdeutlichen, beschränken wir uns zunächst auf ein zweidimensionales Beispiel.

In *Abbildung* 2.3 stellt der rote Punkt den Mittelpunkt eines Domains dar und die

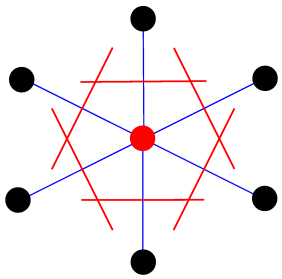
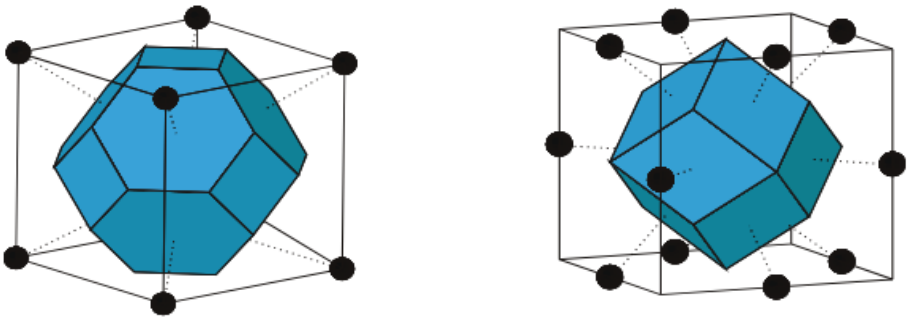


Abbildung 2.3: Wigner-Seitz-Zelle

umliegenden schwarzen Punkte die Atome, die das Domain erzeugen beziehungsweise aufspannen. Dann werden die Verbindungsstrecken zwischen den Atomen und dem Mittelpunkt berechnet. Zeichnet man anschließend die (roten) Mittelsenkrechten der Verbindungsstrecken ein, dann bilden diese die äußeren Grenzen des kompletten *Cavity*-Volumens.
Überträgt man jetzt diese Bildungsvorschrift in den dreidimensionalen Raum, werden die äußeren Grenzen nicht mehr durch Geraden, sondern durch Ebenen, die die Verbindungsstrecke zwischen den Rand-Atomen und Mittelpunkt in der Hälfte der Strecke senkrecht schneiden, gebildet.



(a) Wigner-Seitz-Zelle Beispiel 1

(b) Wigner-Seitz-Zelle Beispiel 2

Abbildung 2.4: Wigner - Seitz - Zellen in 3D

Die Ebenengleichungen lassen sich auf diese Weise leicht aufstellen, denn die Strecke zwischen Atomen und Mittelpunkt ist als Richtungsvektor gegeben.

Setzen wir den Richtungsvektor, der senkrecht auf der gesuchten Ebene steht, in die Normalenform der Ebenengleichung ein, erhält man eine Ebenengleichung der Form:

$$n_1 \cdot x + n_2 \cdot y + n_3 \cdot z = d$$

So ordnet man jedem Rand-Atom eine Ebene zu. Bildet man die Schnittmenge dieser Ebenen in der der Mittelpunkt enthalten ist, dann entspricht die Schnittmenge dem gesuchten Cavity-Volumen.

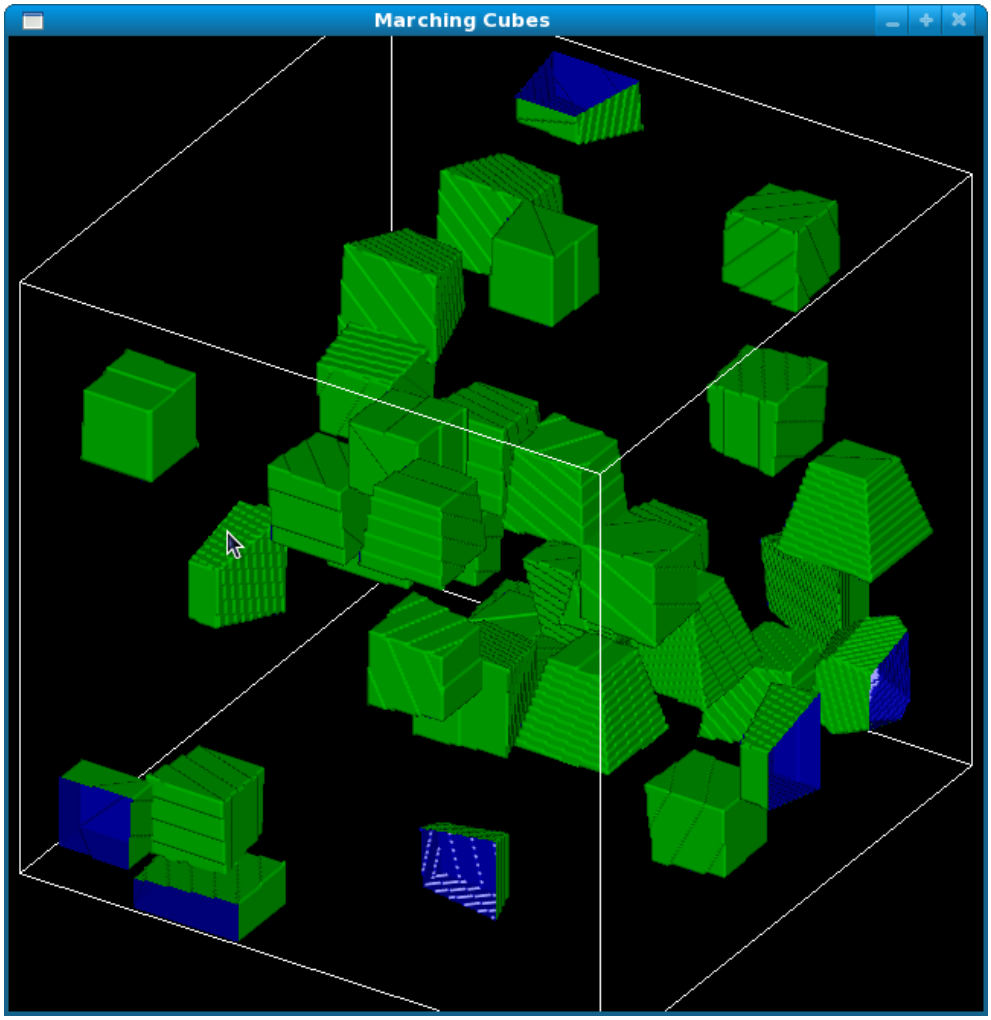
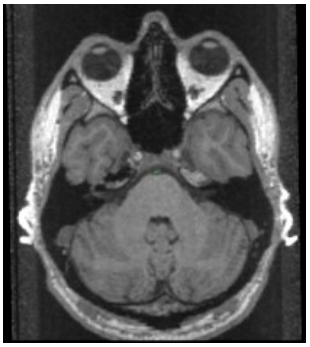


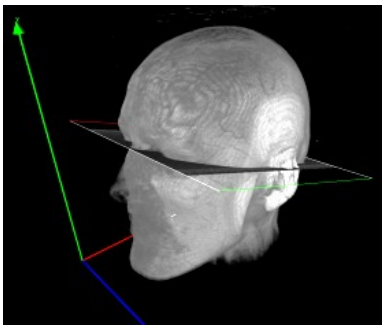
Abbildung 2.5: Komplett berechnete Cavity-Volumen

2.3 Marching-Cubes Algorithmus

Der Marching-Cubes Algorithmus, den sich die Entwickler *William E. Lorensen* und *Harvey E. Cline* im Jahre 1985 patentieren[2] ließen, darf mittlerweile frei verwendet werden. Die Entwickler beschäftigten sich mit einer effizienten Visualisierung von Bilddaten, wie sie beispielsweise bei bildgebenden Verfahren in der Medizin (Computertomographie [CT], Magnetresonanztomographie [MRT]) anfallen. Bei solchen Verfahren entstehen schichtweise Bilder von dreidimensionalen Objekten (Körperteilen). In solchen Bildern werden Materialien wie Knochen und Fett mit der von ihrer Dichte abhängigen Helligkeit dargestellt (vergleiche *Abbildung* 2.6a). Aus einer ganzen Reihe von Schichtbildern können dann mit dem Marching-Cubes Algorithmus 3D-Modelle erstellt werden.



(a) Schichtbild von MRT (2D)



(b) 3D-Modell durch Marching-Cubes

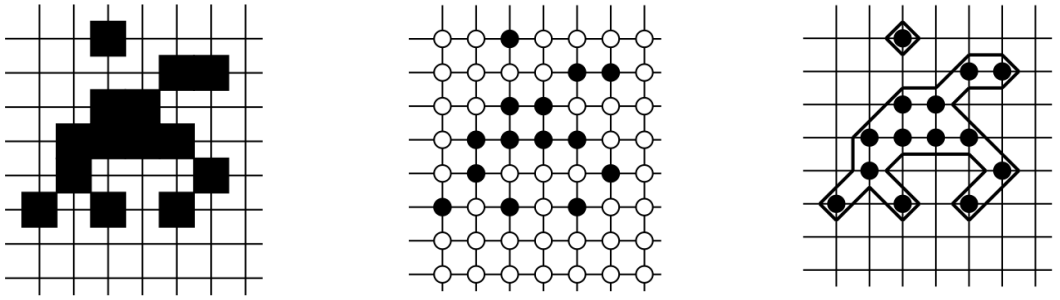
Abbildung 2.6: Visualisierung in der Medizin

Dabei wird im Bezug auf den Algorithmus oft der Begriff *Voxel* benutzt. Dieser Begriff setzt sich aus den Begriffen *Volumen* und *Pixel* zusammen und bezeichnet ein diskretes Volumenelement an einer Stelle im dreidimensionalen Raum.

Um den Marching-Cubes Algorithmus besser verstehen zu können, verdeutlichen wir uns die Vorgehensweise zuerst am zweidimensonalen Marching-Squares Algorithmus. Hier legt man ein Gitter über das zu visualisierende Objekt und besetzt das Gitter an den Kreuzungspunkten mit Dichtewerten. Wie in folgender Abbildung zu sehen, entsteht dann aus einem Objekt [2.7a] ein Punkte-Gitter [2.7b].

[2] Patentiert im Juni 1985. Das Patent war befristet auf 20 Jahre.

Es gibt nur Punkte die innerhalb oder außerhalb der Struktur liegen.



(a) Objekt gerastert

(b) resultierende Voxelgrafik

(c) Berechnete Kontur durch Marching-Squares

Abbildung 2.7: Rasterung, Voxelgrafik und berechnte Kontur

Jetzt kann der Marching-Squares Algorithmus die Kontur um die schwarzen Voxel bestimmen. Dabei existieren $2^4 = 16$ Konfigurationen, wie eine Kontur ein Quadrat des Gitternetzes schneiden kann.

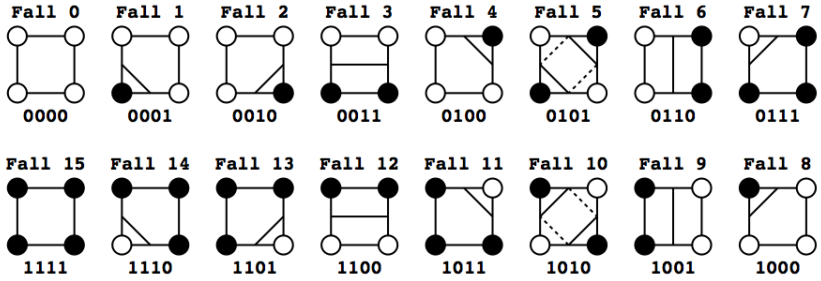


Abbildung 2.8: Die 16 Konfigurationen des Marching-Squares

Wenn anhand dieser Möglichkeiten die Kontur eines Objekts berechnet wird, werden folgende Schritte nacheinander abgearbeitet:

1. Erstellung des Gitternetzes
2. Rasterung des Bildes bzw. Bestimmung der Eckpunkte der Gitternetzquadrate (für jedes Gitternetz ergibt sich anhand der nicht gesetzten (0) bzw. gesetzten (1) Eckpunkte ein 4-Bit Index)
3. Vergleichen der Indizes mit den 16 Konfigurationen. Festlegung, wie die Kontur des jeweiligen Quadrats geschnitten wird.
4. Zeichnen der Kontur

Wenn diese Punkte alle abgearbeitet sind, entsteht aus der Voxelgrafik 2.7b die Kontur in *Abbildung* 2.7c.

Bei unserem dreidimensionalen Problem haben wir in der Voxelgrafik negative und positive Werte. Die negativen Werte werden von dem Marching-Cubes Algorithmus als innerhalb interpretiert und zeigen an, wie tief ein Voxel im zu visualisierenden Objekt liegt. Alle anderen, positiven Voxelwerte werden als *außerhalb* interpretiert und markieren den restlichen nicht zu visualisierenden Raum. Wie beim Marching-Squares Algorithmus wird auch beim Marching-Cubes der Raum gerastert. Für unser Problem muss das Raster sehr fein gewählt werden, in dem dann ein Würfel (engl.: *cube*) schichtweise durch das gerasterte Modell marschiert (engl.: *marching*) und untersucht wird, ob und wie er von einem Objekt geschnitten wird. Anhand des Ergebnisses werden die Ecken des Marching-Cube (MC), je nachdem wie weit diese im Objekt liegen oder auch nicht, mit Dichtewerten besetzt. Anhand eines Schwellwerts wird später entschieden, ab wann eine Ecke des MC den Status „*innerhalb*“ oder „*ausserhalb*“ bekommt. Durch diese zwei Möglichkeiten und die acht Ecken gibt es $2^8 = 256$ verschiedene Schnittmöglichkeiten, die durch Dreiecke dargestellt werden können. Diese 256 verschiedenen Fälle lassen sich aufgrund von symmetrischen Operationen auf lediglich 15 voneinander unabhängige Fälle (*Abbildung* 2.9) reduzieren.

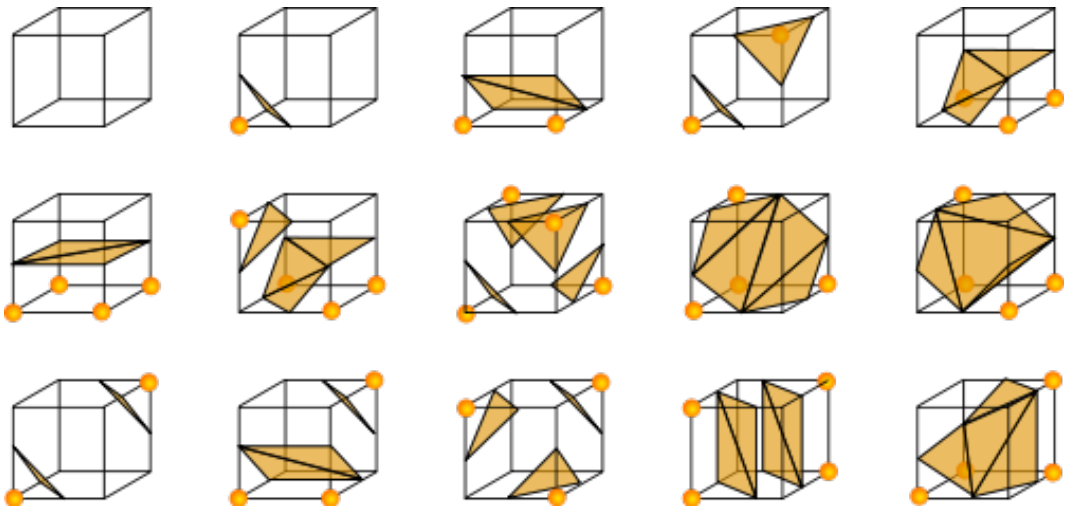
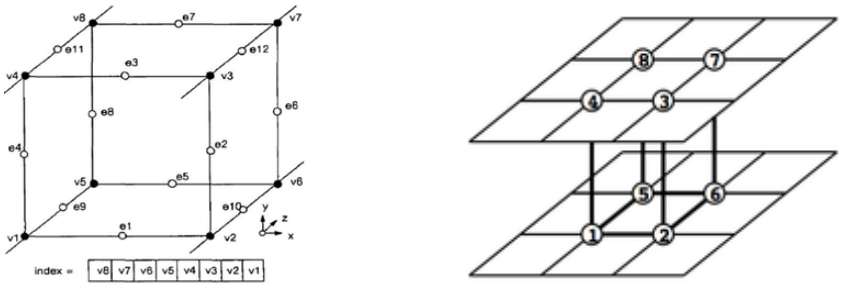


Abbildung 2.9: Die 15 Grundfälle eines Marching-Cube

Wie in der letzten *Abbildung* (2.9) zu sehen ist, werden die Schnittflächen der 256 Konfigurationen durch Dreiecke angenähert. Die 256 Konfigurationen und die für die Visualisierung benötigten Dreiecke sind in einer Liste vorab gespeichert. In der ***T****riangle* ***L****ookup* ***T****able* (**TLT**) steht für jede Schnittmöglichkeit eine Liste mit den Eckpunkten für jedes der erforderlichen Dreiecke.

Ein Marching-Cube wird aus zwei übereinanderliegenden Schichten und jeweils vier übereinanderliegenden Pixeln / Punkten gebildet (vergleiche *Abbildung 2.10b*).



(a) Nummerierung eines MC

(b) MC aus den Schichten S_i und S_{i+1}

Abbildung 2.10: Marching-Cube Beispiele

Abbildung 2.10a zeigt die Nummerierung eines MC, dessen Ecken *(vertices)* mit v und Kanten *(edges)* mit e bezeichnet sind. Wegen der zwei verschiedenen Zustände *(innerhalb, außerhalb)* einer Ecke, kann der Gesamtzustand eines MC binär, anhand eines *8Bit*-Wertes (*Bsp.: 00010011 siehe Abbildung 2.11*), wobei die i-te Ziffer eine *1* ist, falls diese i-te Ecke innerhalb des Objekts liegt und *0*, falls die i-te Ecke außerhalb des Objekts liegt, ausgedrückt werden.

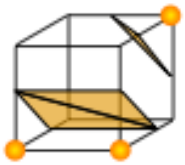


Abbildung 2.11: Ecken 0, 2 und 6 innerhalb, MC-Index: $(00010011)_2$ TLT-Index: $(19)_{10}$

Umgerechnet ins Dezimalsystem ergibt sich der Index $i \in \{0, \cdots, 255\}$ um an der Stelle *TLT(i)* die entsprechenden Dreiecke zur Visualisierung auszulesen. Nach der Bestimmung der Dreiecke müssen nun die Einheitsnormalen der Dreieckspunkte interpoliert werden. Diese dienen der Belichtung der Oberfläche und bewirken ein geglättetes Aussehen.

Hier nochmal die Schritte des Algorithmus kurz zusammengefasst:

1. **Den Marching-Cube bilden** $\Longrightarrow$ Die Werte der acht Ecken bestimmen
2. **Den Index des Cubes errechnen** $\Longrightarrow$ Ist $e_i > \delta$ dann ist die x-te Ziffer, einer 8Bit-Zahl i, eine 1 ansonsten 0 (Im Dezimalsystem: $i \in \{0, \cdots, 255\}$).
3. **Berechne die benötigten Dreiecke** $\Longrightarrow$ Hole die Dreiecke an der Stelle i aus der *TLT(i)*
4. **Oberflächenschnittpunkte interpolieren** $\Longrightarrow$ Bestimme die Position jedes Knotens der eben erzeugten Dreiecke auf den Kanten des Würfels durch lineare Interpolation der anliegenden Ecken.
5. **Einheitsnormalen berechnen und interpolieren** $\Longrightarrow$ Berechne und interpoliere die Einheitsnormalen für jeden Knoten der eben bestimmten Dreiecke.
6. **Speichere die berechneten Daten** $\Longrightarrow$ Die berechneten Dreiecke und Einheitsnormalen für die spätere Visualisierung speichern.

Kapitel 3

Implementierung

3.1 Grundlagen

3.1.1 Programmiersprache & Testsystem

Programmiersprache:	C++
Kompiler:	gcc - GNU project C/C++ compiler version 4.0.1
Betriebssystem:	Mac OS X, Version 10.5.6
Prozessor:	Intel Core 2 Duo 2.4GHz
Speicher:	2GB 800MHz DDR2 SDRAM

3.1.2 Module

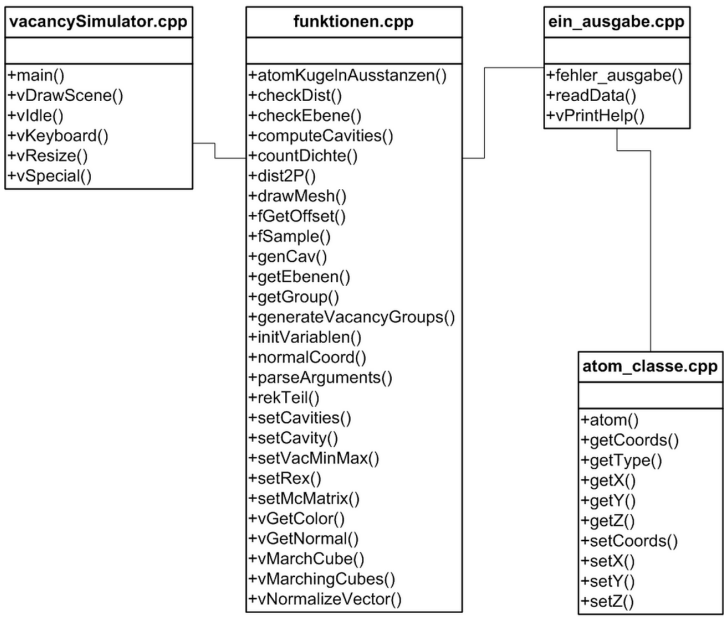


Abbildung 3.1: Modulplan vom Programm *vacancySimulator*

3.1.3 Kompilieren und starten des Programms

Für das Kompilieren des Sourcecodes existiert ein Makefile, welches abhängig vom Modifikationsdatum der einzelnen Dateien erkennt, welche davon neu erstellt werden müssen. Dadurch brauchen die Endnutzer kein weiteres Wissen über Kompilieren und Linken von Programmsystemen mitzubringen.

Das Makefile hat zwei verschiedene Regeln um das Programm zu erstellen. Eine für Linux und die andere für Mac OS Systeme. Die Regel für Mac Systeme ist die default make-Regel.

Erstellt wird das Programm wie folgt:

```
make linux
    oder
make mac
```

Dabei werden die vier Module

- *vacancySimulator.cpp*
- *atom_class.cpp*
- *ein_ausgabe.cpp*
- *funktionen.cpp*

erst einzeln, soweit nötig, kompiliert und in Object-Files mit der Endung *.o* gespeichert. Anschließend werden die Object-Files zusammen zum ausführbaren Programm verlinkt.

Das ausführbare Programm *vacancySimulator* kann mit verschiedenen Optionen gestartet werden, welche per Komandozeilenparameter übergeben werden. Eine Liste der möglichen Optionen kann mit dem Parameter *-h* ausgegeben werden. Mit *-f filename* wird dem Programm die Datei mit den Simulationsdaten übergeben, aus denen die *Cavities* berechnet werden sollen. Mit der Option *-o* wird die 3D-Visualisierung eingeschaltet und mit *-m* werden zusätzlich die Mittelpunkte der Vacancy-Domains eingezeichnet.

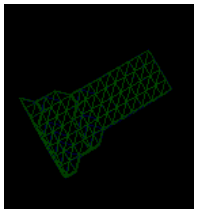
Ausgabe mit der Option *-h*:

```
/*************************************************************
*================>   OPTIONALE ÜBERGABEPARAMETER   <==========*
*************************************************************
*                                                           *
*    -f     file(string)  übergeben der Eingabedatei        *
*    -o                   aktivieren der 3D-Ausgabe         *
*    -m                   einzeichnen der Mittelpunkte      *
*    -del   anzahl(int)   Vacancy-Domains mit weniger       *
*                         als "anzahl" Punkten werden gelöscht *
*    -box   größe(float)  Kantenlänge des abgrenzenden Würfels *
*                         default-Wert ist 24.05 Angström   *
*    -vcut  größe(float)  Entfernung ab wann ein Punkt zum Domain *
*                         gehört. Default-Wert ist R_ex=2.8 *
*                         Angström                          *
*    -saveT datei(string) speichert die berechneten Cavities *
*                         in der angegebenen "datei"        *
*    -readT datei(string) liest aus einer Datei die gespeicherten *
*                         Cavities ein und zeigt diese an   *
*    -v                   wenn angegeben, werden statt der  *
*                         Domains die kompletten Cavities   *
*                         visualisiert                      *
*    -h                   Hilfe ausgeben                    *
*                                                           *
*************************************************************
*================>   WÄHREND DES PROGRAMMABLAUFS   <========*
*************************************************************
*                                                           *
*    w              Drahtgitternetz anzeigen an/aus         *
*    a              Atome anzeigen an/aus                   *
*    m              Mittelpunkte anzeigen an/aus            *
*    l              Wechseln zwischen Belichtung und Farben *
*    +/-            rein- und raus-zoomen                   *
*    =              Zoom auf 0 zurücksetzen                 *
*    Home           Szene drehen an/aus                     *
*                                                           *
/*************************************************************
```

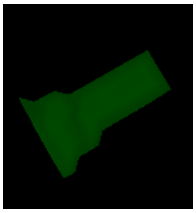
Die Übergabeparameter sind alle optional und können beliebig kombiniert werden. Wenn die Art des Molekülsystems es verlangt, dass ein individueller R_{ex}-Wert oder Kantenlänge des abgrenzenden Würfels gebraucht werden, können diese mit den zwei Optionen *-box* für die Kantenlänge und *-vcut* für R_{ex} dem Programm mitgegeben werden.

Sollen die zu berechnenden Cavities berechnet werden und zu einem späteren Zeitpunkt angezeigt werden, können die Ergebnisse in einer Datei gespeichert werden. Dafür wird dem Programm über den Parameter *-saveT* eine Datei zum Speichern mitgeteilt. Existiert diese nicht, wird sie angelegt. Mit *-readT* kann man dem Programm einen Dateinamen zum Einlesen und Visualisieren gespeicherter Cavities geben. Das Speichern und wieder Einlesen der Daten macht aber nicht immer Sinn, denn die Datenmengen sind beachtlich groß. So kostet das Speichern und vor allem das wieder Einlesen der Daten

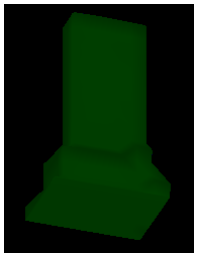
viel Zeit. Manchmal ist es dann vielleicht sinnvoller die Daten neu zu berechnen.
Durch die Diskretisierung des Raums in ein festes Raster kommt es vor, dass sehr kleine Vacancy-Domains entstehen, die eigentlich keinen Sinn ergeben. Durch die Angabe des *-del* Parameters kann dem Programm mitgeteilt werden, dass alle Domains mit weniger als *anzahl* Domain-Punkten ignoriert werden.
Wie im Listing (S. 19) zu sehen, können verschiedene Tastenkombinationen das Verhalten des Programms während der Ausführung verändern. Dazu einige Beispiele:



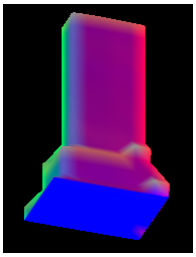
(a) Drahtgitteranzeige an



(b) Drahtgitteranzeige aus



(c) Belichtung



(d) Farben durch Dreiecksnormalen

Abbildung 3.2: Effekt der w-Taste [(a), (b)] und l-Taste [(c), (d)]

3.1.4 Datenstrukturen

Wie in Kapitel 2.1 beschrieben, liegen die Eingabedaten in einem Format vor, in dem die Lage der Atome durch Koordinaten in X-, Y- und Z-Richtung gegeben ist. Darüber hinaus ist deren Elementname angegeben, was wiederum einen bestimmten Atomradius festlegt. Für diese oft wiederkehrenden Informationen bietet sich eine eigene Datenstuktur an. Mit geeigneten Get- und Set-Funktionen kann auf die gewünschten Daten zugegriffen werden. Die daraus folgende Klasse *atom* besitzt folgende Struktur:

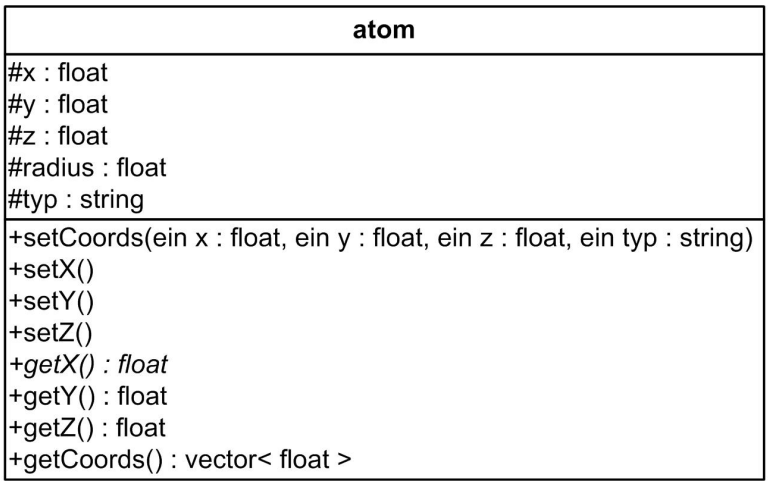


Abbildung 3.3: UML-Klassendiagramm von „atom “

Für das Abspeichern, Markieren und Durchsuchen des Raumes bieten sich dreidimensionale Matrizen an. Da die für die Variablendeklaration benötigten Größen der Matrizen bekannt sind, können diese am Anfang des Programms angelegt und initialisiert werden.

Für Vektoren unbekannter Größe wird eine zusätzliche Bibliothek, die C++ ***S****tandard* ***T****emplate* ***L****ibrary* (kurz **STL**) *vector*, verwendet. Hier muss der Speicherbereich für dynamische Variablen nicht, wie beispielsweise in C, kompliziert selbst verwaltet werden. In der *STL* wird der Speicher intern automatisch verwaltet und durch einfache sprechende Funktionen wie *push_back()*, *insert()*, *size()* und viele mehr, kann man den Vektor erweitern, Elemente einfügen und die aktuelle Größe herausfinden. Der Zugriff erfolgt weiterhin wie gewohnt durch den *[]*-Operator.

So werden zum Beispiel die Atome beim Einlesen nach und nach mit der *push_back()*-Funktion an einen *atom*-Vektor angehängt. Danach stehen die Atome auf der Variablen *atomVector* zur weiteren Verwendung zur Verfügung.

```
vector< atom > atomVector;
```

Für die oft benötigte Speicherung von Vektoren der Form

$$\vec{v} = \begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

werden die Vektoren als Datentypen der Form

```
struct iVec{
    int x;
    int y;
    int z;
};
```

implementiert. Die gleiche Form existiert auch für *float*-Vektoren (fVec).

Für die später benötigten Normalengleichungen von Ebenen hilft folgende Struktur die Gleichung jeder Ebene zu speichern. In einem *ebene*-Element sind dann der Normalenvektor v, die Konstante d sowie die Länge des Normalenvektors l enthalten. Der Typ *ebene* ist wie folgt definiert:

```
struct ebene
{
        iVec v;
        float d;
        float l;
};
```

Der Marching-Cubes-Algorithmus erzeugt zur Darstellung der *Cavities* nicht nur das Dreiecksnetz, sondern auch zu jedem Eckpunkt eines Dreiecks einen Normalen-Vektor und einen eigenen Farbwert. Somit hat jedes Dreieck 3 Punkte, 3 Vektoren und 3 Farbwerte. Damit diese Daten strukturiert und zusammenhängend gespeichert werden können, existiert folgende Datenstruktur:

```
struct GLtri{
    fVec p[3];
    fVec n[3];
    fVec color[3];
};
```

3.2 Implementierung der Berechnung der Vacancy-Domains

In Abschnitt 2.2.2 wurde definiert, dass alle Punkte, die weiter als ein festes R_{ex} von allen Atomen entfernt sind, zu einem Domain gehören.
Während der Implementierung hat sich gezeigt, dass eine iterative Lösung zum Berechnen der Domains einfacher und auch schneller ist, als ein rekursiver Algorithmus. Als Datenstruktur wird eine dreidimensionale integer-Matrix verwendet in der alle Punkte um ein Atom markiert werden, deren Abstand zum Atom kleiner als R_{ex} ist. Dadurch markiert man im Raum Kugeln um die Atome mit dem Radius R_{ex} und es bleiben letztlich diejenigen Punkte unmarkiert, die zu Vacancy-Domains gehören. In *Abbildung 3.4* ist der Zeitpunkt nach dem „Ausstanzen“ zu sehen.

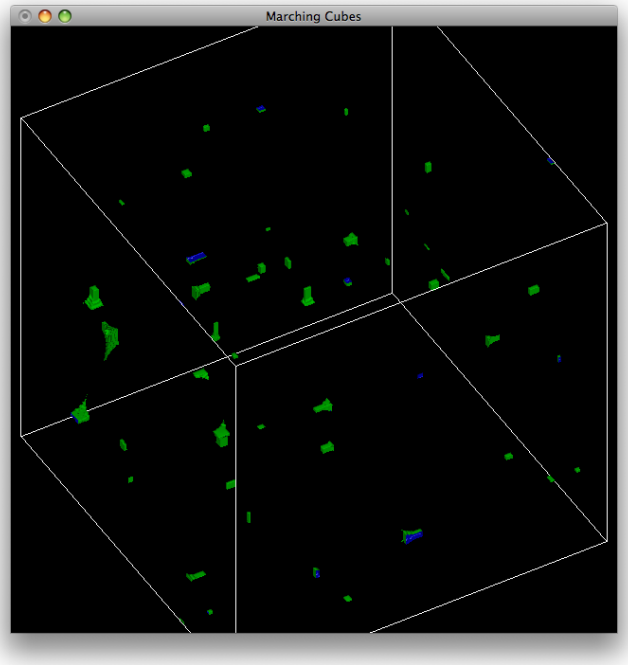


Abbildung 3.4: Visualisierung der Vacancy-Domains nach dem „Ausstanzen“ der Kugeln um die Atome

Programmiertechnisch ergibt das drei Schleifen, die diese Kugeln markieren. Der Rechenaufwand hierfür ist relativ hoch, denn in den Schleifen werden viele Abstände berechnet. Bei beispielsweise 460 Atomen und $R_{ex} = 2.8\mathring{A}$, in den diskreten Raum umgerechnet ergibt das ein $R^*_{ex} = 35$ Längeneinheiten.

Daraus folgen dann

$$(2 * R^*_{ex})^3 * Anzahl_{Atome} = (2 * 35)^3 * 460 = 157780000$$

Abstandsberechnungen.

Nach der Markierung derjenigen Punkte, die zu Vacancy-Domains gehören, weiß man jedoch noch nicht, welche Punkte zu welchem Domain gehören. Das heißt, als zunächst müssen die Punkte gruppiert werden.
Dabei geht man von einem Vacancy-Domain-Punkt aus rekursiv durch den Raum solange weitere Vacancy-Domain-Punkte an den aktuellen Punkt angrenzen. Die dabei neu gefundenen Punkte werden in einer Liste gespeichert, so ist es später möglich, auf ein komplettes Domain und den zugehörigen Punkten zu zugreifen.
Um bei späteren Berechnungen Zeit zu sparen und das Programm weiter zu optimieren, ist es wichtig zu wissen, welche Atome ein Vacancy-Domain abgrenzen bzw. erzeugen. Bestimmt man später die Mittelpunkte der Domains, müssen die Abstände der Domain-Punkte zu den Atomen berechnet werden. Sind dann die Rand-Atome eines Domains nicht bekannt, müssen die Abstände zu jedem der beispielsweise 460 Atomen berechnet werden. Im Vergleich zu den durchschnittlich 4-6 Rand-Atomen würde das einen deutlichen Mehraufwand bedeuten. Um die Rand-Atome möglichst effizient zu identifizieren, sind beim Markieren der Kugeln jeweils die Indizes der entsprechenden Atome benutzt worden.
Wenn man bei dem Verfahren zur Gruppierung der Domains an den Rand eines Domains stößt, muss nur einmal „über den Rand hinaus geschaut“ werden, wo dann der Index des abgrenzenden Atoms steht. Dieser Index wird in einem zweidimensionalen Vektor gespeichert. Da bei dieser Strategie manche Randatome mehrfach erkannt und abgespeichert werden, müssen im Nachhinein doppelte Indizes entfernt werden.

Letztlich gibt die erste Dimension des Rand-Atom-Vektors an, um welches Domain es sich handelt und mit der zweiten Dimension greift man auf die Indizes der Rand-Atome zu. Mit diesen Indizes hat man dann in dem *atom*-Vektor Zugriff auf die Koordinaten der Atome. Durch das Speichern der Indizes wird das redundante Speichern der Koordinaten gespart, da diese ja schon im *atom*-Vektor stehen.

In folgender Grafik ist ein zweidimensionales Beispiel einer Matrix abgbildet, in der die Vacancy-Domains markiert sind. Dabei stellen die Zahlen größer Null die Indizes der Atome dar, die das Domain erzeugen, und die Nullen selbst den Bereich, der zu einem Domain gehört.

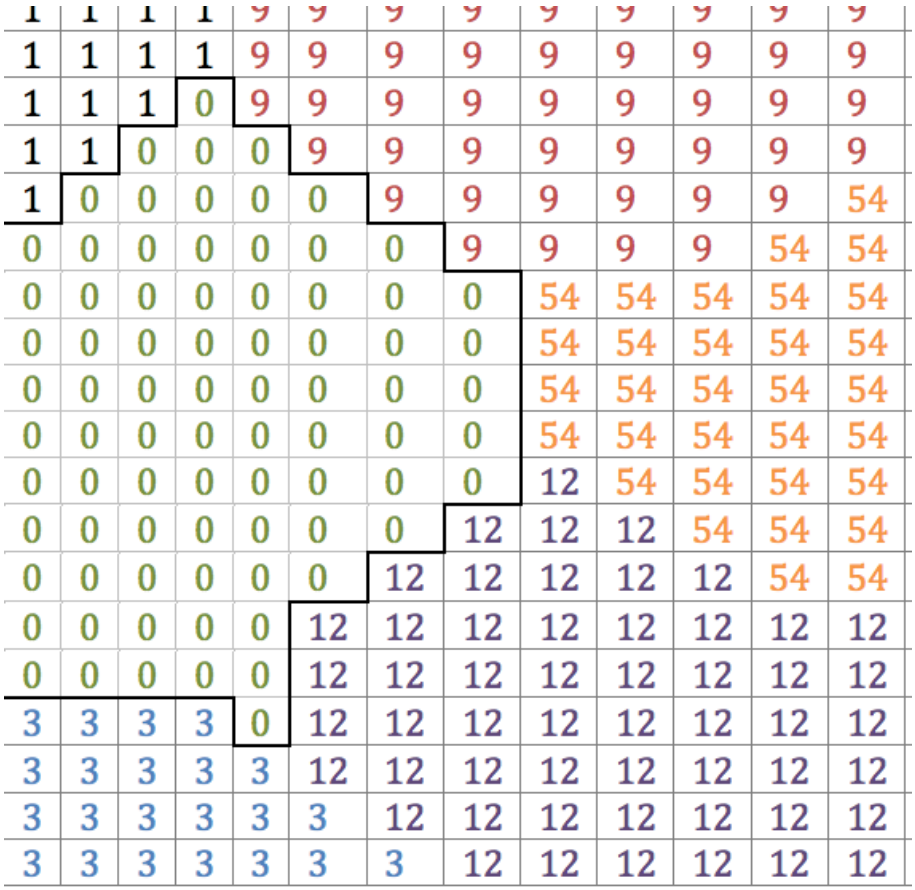


Abbildung 3.5: Zweidimensionales Beispiel einer Matrix, in der die Kugeln um die Atome markiert worden sind. Die grünen Nullen sind hier die nicht markierten Stellen, d.h. diese Stellen gehören zu einem Domain

Da die Nullen für die Interpolation der Oberflächen im Marching-Cubes Algorithmus ungeeignet sind, werden in einer zweiten Matrix, der *dichteMatrix*, an die Stelle der Nullen Dichte-Werte gesetzt. Ein solcher Dichte-Wert gibt an, wieviele Punkte in einer bestimmten Umgebung ebenfalls zu diesem Domain gehören. Die Matrixwerte wurden vorher mit *-1* vorinitialisiert.

Im folgenden Beispiel wird die direkte Umgebung eines Punktes untersucht. Mit der direkten Umgebung sind, wie in *Abbildung* 3.6 verdeutlicht, die direkten acht Nachbarzellen eines Punktes gemeint.

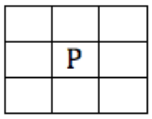


Abbildung 3.6: direkte Umgebung

Also ist der größte theoretische Dichte-Wert die **8**, da der aktuelle Punkt nicht mitgezählt wird.

1	1	9	9
1	0	9	9
0	0	0	9
0	0	0	0

(a) *Matrix*

-1	-1	-1	-1
-1	3	-1	-1
4	6	5	-1
3	5	4	2

(b) *dichteMatrix*

Abbildung 3.7: Vergleich der beiden Matrizen *Matrix* und *dichteMatrix* an der gleichen Stelle

3.3 Marching-Cubes Implementierung

Für den Algorithmus sind fünf Konstanten definiert, die der Übersichtlichkeit im Code und der Optimierung des Programmablaufs dienen.

```
static const int a2iVertexOffset[8][3];
static const int a2iEdgeConnection[12][2];
static const int a2iEdgeDirection[12][3];
static const int aiCubeEdgeFlags[256];
static const int a2iTriangleConnectionTable[256][16];
```

Die ersten drei dienen der Orientierung innerhalb eines Würfels. Nimmt man bei einem Würfel den vorderen, unteren, linken Eckpunkt als Ausgangspunkt, so ergibt die Addition mit den einzelnen Vektoren aus *a2iVertexOffset* jeweils die Position einer der acht Ecken des Würfels. In *a2iEdgeConnection* stehen alle Punktepaare, welche die 12 Kanten des Würfels bilden.
Ob eine Kante nun von Punkt eins nach Punkt zwei zeigt oder umgekehrt, ist in *a2iEdgeDirection* als Richtungsvektor abgespeichert. Die letzten zwei Variablen sind zum Einen, wie schon in Abschnitt 3.7 erläutert, die 256 Möglichkeiten, wie ein Würfel von einem Objekt geschnitten werden kann und zum Anderen die für die Darstellung nötige *Triangle Lookup Table*.

Vorgehensweise (Pseudocode):

```
hole die 8 Werte für die Ecken des Cubes
gehe durch alle 8 Ecken
    ist der Wert der Ecke größer als der Schwellwert
        dann erhöhe den Würfel—Index

ist der Würfel komplett oder garnicht im Objekt enthalten
    dann gehe zum nächsten Cube

gehe über alle Kanten
    wird die momentane Kante geschnitten?
        interpoliere Schnittpunkt
        errechne die Normale des Schnittpunkts

berechne die Dreiecke für den momentanen Würfel
    speichere diese auf die globale Variable "dreiecke"
```

Da die Werte für die acht Ecken des Marchung-Cubes in unserer *dichteMatrix* stehen, benötigen wir eine Funktion, die den Wert an einer gewünschten Stelle zurückgibt. Dies erledigt die Funktion *fSample(x,y,z,dichteMatrix)*. Sie gibt an einer beliebigen Stelle *(x,y,z)* den entsprechenden Wert der *dichteMatrix* zurück. Dieser wird in einem acht Felder langen Vektor gespeichert und daraus anschließend der Würfelindex berechnet. Dieser Index beschreibt eine der 256 möglichen Schnittkonfigurationen. Aufgrund der

Zweierpotenz lässt sich durch den Bit-Shift-Operator „<<“ sehr bequem der Index erstellen. Nehmen wir zur Verdeutlichung nochmal die Abbildung aus Abschnitt 2.11. Wenn jetzt für alle Ecken geprüft wird, ob diese innerhalb des Objekts liegen, wird bei

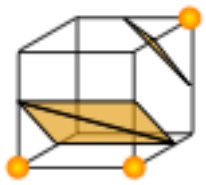


Abbildung 2.11 aus Abschnitt 2.3

der 0ten, 1ten und 6ten Ecke folgende Zeile ausgeführt:

```
    iFlagIndex |= 1<<iVertexTest;
```

Dabei steht in *iVertexTest* die 0, 1 oder 6. Zuerst wird die *1* um *iVertexTest*-Stellen nach links geshiftet (Bsp.: 1 << 6 = 1000000). Anschließend wird das Ergebnis des Shifts und der aktuelle Index mit dem bitweisen Oder-Operator verglichen. Ansonsten gibt es eine **1** zurück.

Erster Schritt:

```
iFlagIndex = 0, iVertexTest = 0;
1 << 0 => 1
0 |= 1 => 1
iFlagIndex = 1;
```

Zweiter Schritt:

```
iFlagIndex = 1, iVertexTest = 1;
1 << 1 => 10
1 |= 10 => 11
iFlagIndex = 11;
```

Dritter Schritt:

```
iFlagIndex = 11, iVertexTest = 6;
1 << 6 => 1000000
11 |= 1000000 => 1000011
iFlagIndex = 1000011;
```

Am Schluss ist

$$iFlagIndex = (1000011)_2 = (67)_{10} \ .$$

Nun kann man sich mit dem so errechneten Index die Dreiecke für die spätere Visualisierung aus der TLT an der 67sten Stelle auslesen und abspeichern. Aus der TLT bekommt man mit der *67* folgende Zeile:

```
{1, 8, 3, 1, 9, 8, 5, 10, 6, −1, −1, −1, −1, −1, −1, −1},
```

In dieser Zeile sind drei Dreiecke eingetragen. Einmal von Kante **1** nach **8** und nach Kante **3** des momentanen Marching-Cubes. Das zweite von Kante **1** nach **9** und nach **8**. Die erste *-1* markiert das Ende der Dreiecksauflistung. Um die Dreiecke explizit durch drei Punkte angeben zu können, müssen die Schnittpunkte der Kanten berechnet/interpoliert werden. Über die Variable *a2iEdgeConnection* erhält man pro Kante die benötigten Punktepaare. Vergleicht man die Dichtewerte an den beiden Punkten wird bei Gleichheit der Dichtewerte die Mitte der Kante als Schnittpunkt gewählt. Sind die Werte jedoch unterschiedlich, wird der Schnittpunkt in Richtung des höheren Dichtewerts verschoben.

Diese Abfolge muss nun für den kompletten Raum wiederholt werden. Entlang der drei Achsen ergeben sich bei $N = 300$ bereits:

$$N^3 = 300^3 = 27.000.000$$

Marching-Cubes Abläufe.

3.4 Berechnung der Mittelpunkte

Die Berechnung der Mittelpunkte erfolgt relativ effizient, da die Vacancy-Domains mit den dazugehörigen Rand-Atome gezielt durchlaufen werden können. Für jedes Domain werden dabei folgende Schritte abgearbeitet:

- berechne für jeden Punkt des Domains die Abstände zu den Rand-Atomen
- wähle für jeden Punkt mit seinen Abständen zu den Rand-Atomen den kleinsten Abstand aus
- setze den Punkt, der den größten Minimalabstand hat, als Mittelpunkt für das Domain

Die Strategie dahinter ist es, um jeden Punkt eine Kugel zu setzen, die kein umliegendes Rand-Atom berührt oder einschließt. Nach Abarbeiten des zweiten Punktes der Liste sind die Radien der einzelnen Kugeln für jeden Vacancy-Domain-Punkt berechnet. In Abbildung 3.8 stellt der rote Kreis einen Vacancy-Domain-Punkt mit dem dazugehörigen maximalen Kreis dar.

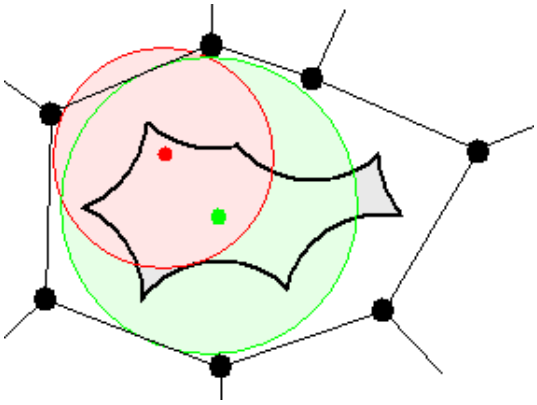


Abbildung 3.8: 2D-Beispiel zur Verdeutlichung der Mittelpunktberechnung

Das Maximum dieser Radien ist somit die größte mögliche Kugel die in das Domain passt (grüner Kreis im 2D-Beispiel [3.8]). Gibt es mehrere maximale Kugeln, wird nur die zuerst gefundene berücksichtigt.

3.5 Berechnung der vollständigen Cavities

Die Bildungsregeln für das Komplettvolumen einer Hohlstelle haben wir in Abschnitt 2.2.3 kennen gelernt. Zunächst brauchen wir für die Volumenbestimmung die Gleichungen der Ebenen, die jeweils senkrecht zu den Strecken zwischen Rand-Atom und Mittelpunkt stehen. Diese werden in einer separaten Funktion berechnet und abgespeichert. Die Ebenen stehen anschließend Domain-weise gruppiert in einer Liste. Dadurch wird bei der Volumenbestimmung nur auf die nötigen Ebenen zugegriffen.

Bevor mit der Markierung der Cavities begonnen werden kann, wird der zu betrachtende Raum eingegrenzt, damit der Aufwand möglichst gering bleibt. Als Abgrenzung werden die Minima und Maxima der Koordinaten der Rand-Atomen genommen. Anschließend geht man iterativ durch den so abgesteckten Raum und bestimmt für jeden Punkt den Abstand zu jeder Ebene der Rand-Atome.

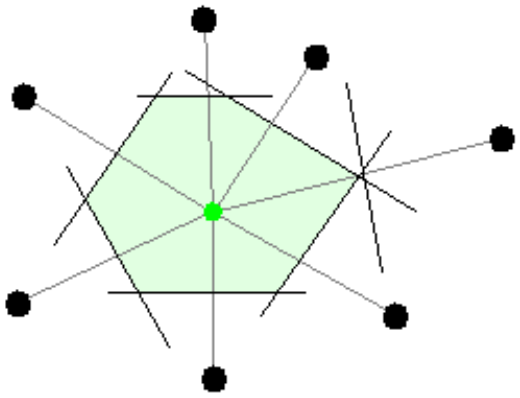


Abbildung 3.9: Wigner-Seitz-Menge, resultierend aus Abbildung 3.8

Da die Ebenen so bestimmt wurden, dass die Rand-Atome auf den Ebenen liegen, gehört ein Punkt zum gesuchten Cavity-Volumen, falls er von jeder Ebene weiter entfernt ist, als die Hälfte der jeweiligen Verbindungsstrecke zwischen Rand-Atom und Mittelpunkt. Diese Vorgehensweise ist sehr rechenintensiv, da pro Punkt durchschnittlich 4-6 Abstände berechnet werden müssen.

3.6 Zusammenfassung

Folgender Ablaufplan fasst die Vorgehensweise des Programms zusammen:

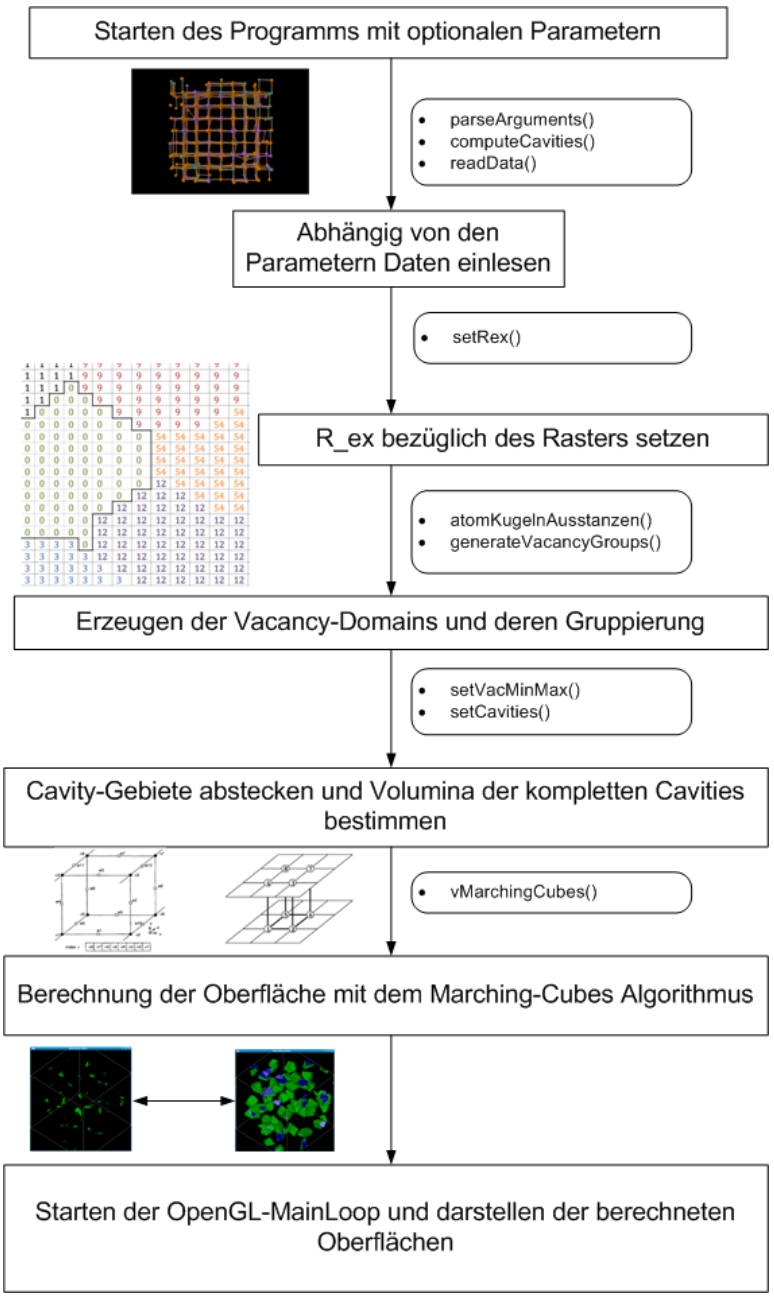


Abbildung 3.10: Zusammenfassung des Programmablaufs - Ablaufdiagramm

Kapitel 4

Ergebnisse

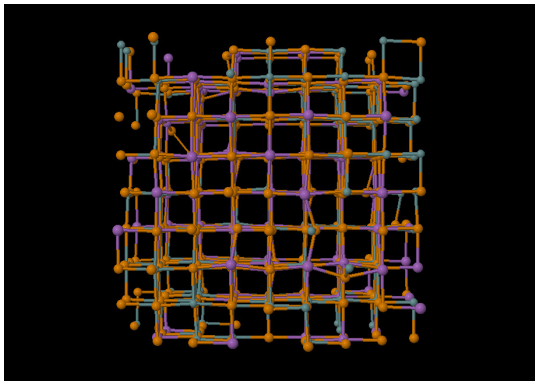
Während der Entwicklung des Programms und dessen Tests hat sich gezeigt, dass die meiste Rechenzeit für das Markieren der Vacancy-Domains und der kompletten Cavities sowie beim Marching-Cubes (MC) Algorithmus benötigt wird. Bei den ersten zwei Programmteilen werden viele Abstände berechnet was auch der Grund für deren lange Laufzeiten ist. Ein Abstand a von zwei Punkten:

$$p_1 = \begin{pmatrix} x1 \\ y1 \\ z1 \end{pmatrix} \text{ und } p_2 = \begin{pmatrix} x2 \\ y2 \\ z2 \end{pmatrix}$$

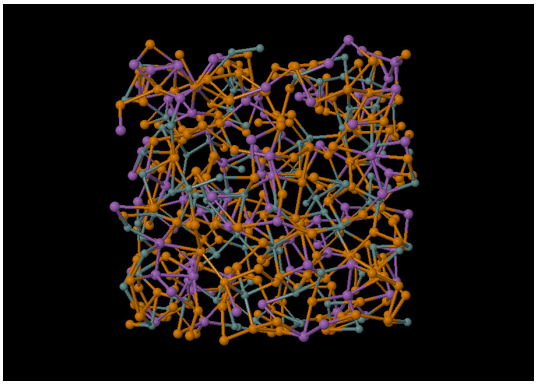
wird im dreidimensionalen Raum wie folgt berechnet:

$$a = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2 + (z_1 - z_2)^2}$$

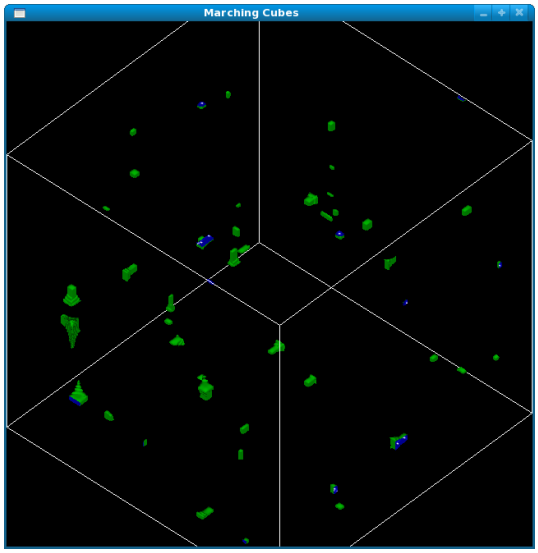
Bei 157.780.000 Abstandsberechnungen (Beispiel S.24) dauert die Markierung der Domains auf dem Testsystem 4-7 Sekunden. Die weit rechenaufwändigere Funktion ist jedoch die zur Bestimmung der ganzen Cavity-Volumina. Auf dem Testsystem werden dafür einige Minuten in Anspruch genommen.
Für den MC-Algorithmus müssen immer 27 Millionen MC-Abläufe durchgerechnet werden da das Raster konstant bleibt. Die Laufzeit des Algorithmus wird durch Punkte die komplett oder nicht in einem zu visualisierenden Objekt liegen, verkürzt. Hier wird der Algorithmus abgebrochen, da keine Oberfläche visualisiert werden muss. Bei sehr großen oder vielen Cavities kann so die Laufzeit um 4-5 Sekunden ansteigen. Alles in allem hängen die Laufzeiten des Programms von den Simulationsdaten ab. Sind die Molekülstukturen kristalliner Art, braucht das Programm weniger Zeit als bei amorphen Strukturen. Das liegt daran, dass in der amorphen Phase einer Phasenwechsellegierung die Cavities um einiges größer sind als in der geordneten kristallinen Phase.



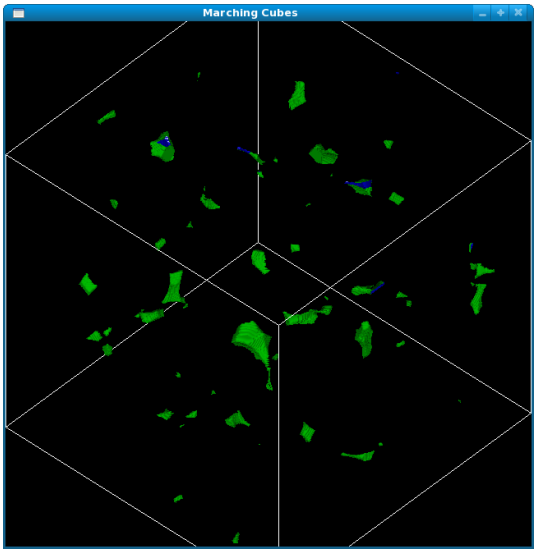
(a) kristallines Molekül



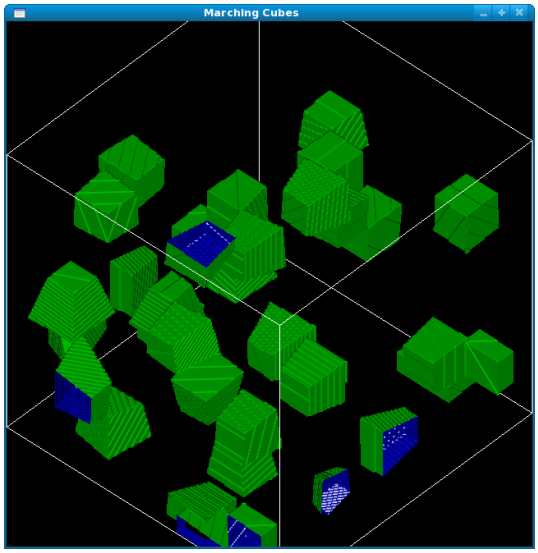
(b) amorphes Molekül



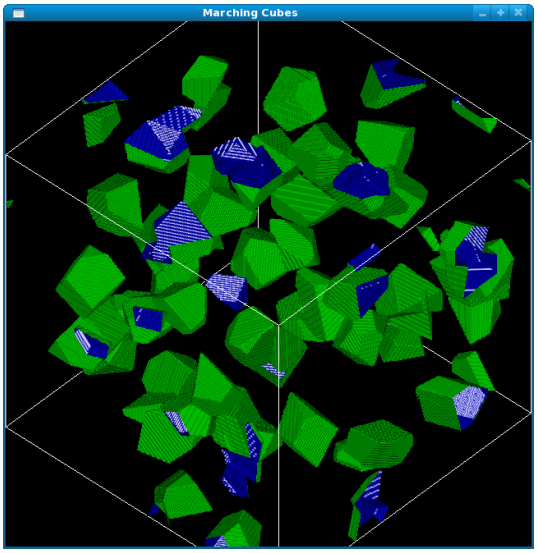
(c) Vacancy-Domains von (a)



(d) Vacancy-Domains von (b)



(e) Cavities von (a)



(f) Cavities von (b)

Abbildung 4.1: Screenshots der Ergebnisse zweier $GeSbTe$-Moleküle

Bei den Beispielen aus der Abbildung 4.1 wurden alle Vacancy-Domains mit weniger als 5 Punkten entfernt. Alle restlichen sind komplett berechnet und dargestellt worden. Die Berechnung für das kristalline Molekül 4.1e hat auf dem Testsystem insgesamt 43 Sekunden gedauert.
Die Berechnung zu Abbildung 4.1f hat bis zur Darstellung der kompletten Cavities insgesamt 162 Sekunden benötigt. Dies zeigt, dass ein amorphes Molekül rechenintensiver ist.

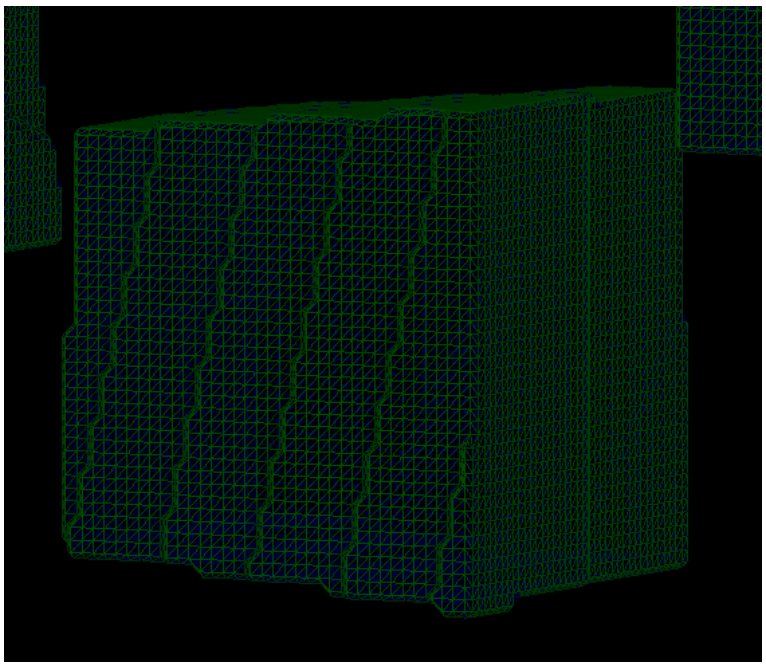


Abbildung 4.2: Ausschnitt aus einem Cavity-Beispiel in der Gitternetzdarstellung

Wie in Abbildung 4.2 zu sehen ist, bestehen die anfallenden Oberflächen größtenteils aus geraden, ebenen Flächen. Zurzeit werden diese geraden Flächen durch eine Vielzahl von Dreiecken visualisiert, deren Berechnung den Marching-Cubes Algorithmus relativ viel Zeit kostet. Durch die große Anzahl von Dreiecken ist das Drehen der Szene im interaktiven OpenGL-Modus ziemlich stockend, was aber durch einen optimierten Algorithmus behoben bzw. verbessert werden kann.

Kapitel 5

Ausblick

Volumen- und Oberflächenberechnung

In dieser Arbeit ist die Berechnung der Maßzahlen der Volumina und Oberflächen nicht enthalten, da dies den Rahmen der Arbeit gesprengt hätte. Es gilt dafür Berechnungen zu ergänzen oder separat zu implementieren.

Volumenbestimmung durch das Monte-Carlo-Verfahren

Die Volumina könnten beispielsweise mit einem *Monte-Carlo-Verfahren*[4] angenähert werden. Dabei würden die Bereiche der Cavities durch einen Würfel abgeschätzt und zufällig eine bestimmte Anzahl an Punkten in diesem Würfel daraufhin untersucht, ob diese innerhalb oder außerhalb der jeweiligen Hohlstelle liegen. Zählt man die Punkte innerhalb und außerhalb kann so auf das Volumen des Cavitys geschlossen werden.

Oberflächenbestimmung

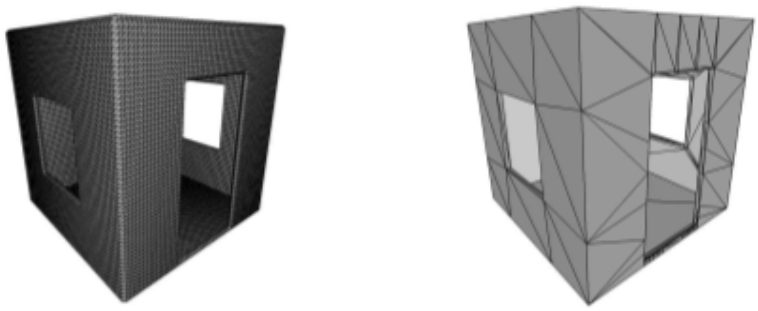
Für die Bestimmung der Oberflächen müssten die Cavities mit einem noch feineren Raster triangularisiert werden. Summiert man die Flächeninhalte der Dreiecke auf, erhält man eine Abschätzung der Oberfläche. Diese Methode wird aufgrund des feineren Rasters noch rechenaufwändiger sein und bedarf einer guten Optimierung oder eines anderen Verfahrens. Da bei den Cavities die Oberflächenstruktur durch Ebenen abgegrenzt werden, könnte man vielleicht auch analytisch vorgehen, indem die Schnittmenge der Ebenen berechnet wird.

Visualisierung

Bei der Auswahl der Methode zur Visualisierung der Cavities ist die Wahl relativ schnell auf den Marching-Cubes Algorithmus gefallen, weil er sich besonders gut für die vorliegenden und anfallenden Datenmengen eignet. Während der Umsetzungsphase und vor allem gegen deren Ende, hat sich jedoch gezeigt, dass die simple Form des Algorithmus doch einige Schwachstellen hat. Es gibt weiterentwickelte kompliziertere Varianten des Algorithmus die verschiedene Möglichkeiten der Optimierung beinhalten.

Dual Marching Cubes

Eine Variante ist der *Dual Marching Cubes* Algorithmus, der statt eines statischen Dreiecksgitters ein adaptives Gitter berechnet. Das spart vor allem bei ebenen Flächen viel Zeit und Resourcen. Ein gutes Beispiel zur Verdeutlichung der unterschiedlichen Dreiecksgitter bietet die folgende Abbildung:



(a) Marching Cubes (b) Dual Marching Cubes

Abbildung 5.1: Vergleich der beiden Dreiecksgitter zur Darstellung der Oberfläche eines Zimmers

Octree-basierte Reduktion der Dreiecke

Eine andere interessante Variante ist die *Octree-Based Decimation of Marching Cubes Surfaces*. Die *Octree*-Struktur ähnelt einem Binärbaum, nur für 3D. Einfach betrachtet, basiert dieses Verfahren auf der Idee, einen Würfel in jeweils acht kleinere Würfel zu zerlegen.

In Verbindung mit dem Marching-Cubes Algorithmus sollen so Gebiete, die weit entfernt von den zu visualisierenden Objekten sind, mit einer geringeren Detailgenauigkeit betrachtet werden. Dabei wird der Würfel bis zu einer gewünschten Größe in eine *Octree*-Struktur unterteilt. Anschließend werden jeweils acht Blätter eines Vaterknotens

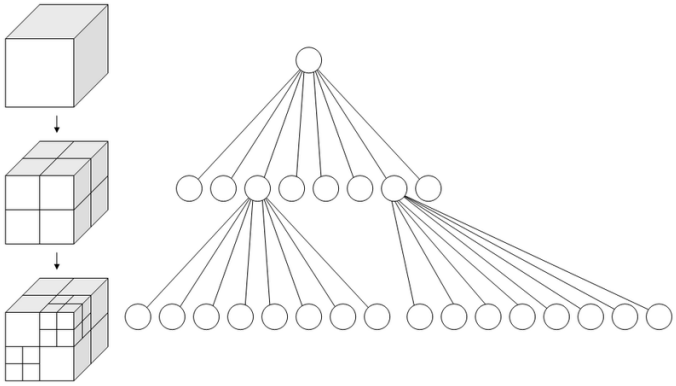


Abbildung 5.2: Die Idee eines Octrees zur Zerlegung eines Würfels

daraufhin untersucht, ob sie ohne großen Qualitätsverlust zu einem größeren Würfel zusammen gefasst werden können.

Parallelisierung

Da sich bei der Problemstellung der Berechnung von Cavities verschiedene Berechnungen und Funktionsaufrufe sehr oft wiederholen und sich die Berechnung zum Beispiel verschiedener Cavities gut aufteilen lässt, bietet sich hier auch die Parallelisierung für Großrechner oder Rechner-Cluster an. Der aktuelle serielle Programmteil, in dem die Kugeln um die Atome ausgestanzt werden, benötigt auf dem Testsystem ca. 4-7 Sekunden. Dabei wird ein Atom nach dem anderen behandelt, obwohl die Daten unabhängig voneinander sind. Bei 460 Atomen und einer Laufzeit von $\mathbf{L_S} = 5s$ braucht der Rechner dabei 0,011 Sekunden pro Atom. Parallelisiert man diesen Teil, ändert sich die Laufzeit linear mit der Anzahl der zur Verfügung stehenden Prozessoren. Sei $\mathbf{L_P}$ die Laufzeit des parallelisierten Programmteils, $\mathbf{L_S}$ die Laufzeit des seriellen Programmteils und $\mathbf{n_p}$ die Anzahl der Prozessoren, dann verhalten sich die Laufzeiten zueinander wie folgt:

$$\mathbf{L_P} = \frac{\mathbf{L_S}}{\mathbf{n_p}}$$

Theoretisch würde der Programmteil mit 460 Prozessoren dann nur $0,011s$ benötigen. Ebenso wie das „Ausstanzen“ der Kugeln ist die Berechnung der Mittelpunkte, die Berechnung der kompletten Cavity-Volumina und der Marching-Cubes Algorithmus parallelisierbar. Es werden dort immer einzelne voneinander unabhängige Domains, Bereiche im Raum oder Marching-Cubes berechnet. Alle diese Möglichkeiten der Parallelisierung würden den Programmablauf enorm beschleunigen.

Abbildungsverzeichnis

Literaturverzeichnis

[1] J. Akola and R. O. Jones
Structural phase transitions on the nanoscale: The crucial pattern in the phase-change materials $Ge_2Sb_2Te_5$ and $GeTe$
Phys. Rev. B **76**, 235201 (2007) [10 pages]

[2] J. Akola and R. O. Jones
Density functional study of amorphous, liquid and crystalline $Ge_2Sb_2Te_5$: homopolar bonds and/or AB alternation?
J. Phys.: Condens. Matter **20**, 465103 (10pp)

[3] Wikipedia: *Marching Cubes*
http://de.wikipedia.org/wiki/Marching_Cubes (Zugriff: Dezember 2008)

[4] Wikipedia: *Monte-Carlo-Simulation*
http://de.wikipedia.org/wiki/Monte-Carlo-Simulation (Zugriff: Dezember 2008)

[5] Wikipedia: *Octree*
http://de.wikipedia.org/wiki/Octree (Zugriff: Dezember 2008)

[6] cplusplus.com - The C++ Resources Network
http://www.cplusplus.com/ (Zugriff: Dezember 2008)

[7] Jürgen Wolf
C++ von A bis Z, Das umfassende Handbuch
Verlag: Galileo Press, ISBN 978-3-89842-816-3

[8] Richard S. Wright, Benjamin Lipchak, Nicholas Haemel
OpenGL(R) SuperBible: Comprehensive Tutorial and Reference (4th Edition)
Verlag: Addison-Wesley Longman, ISBN: 978-0-32149-882-3

[9] Bernd Mohr
Programming in C++ (Jülich, 2003)
FZJ-ZAM-BHB-0154

[10] Michel Goossens
Der LaTeX-Begleiter.
Verlag: Pearson Studium, ISBN: 978-3-82737-166-9

Danksagungen

Besonders möchte ich mich bei

Herrn Prof. Dr. rer. nat. Martin Reißel für die Übernahme des Hauptreferats und Herrn Josef Heinen als Korreferent

sowie für deren fachliche Unterstützung bei der Arbeit bedanken.

Weiterhin bedanke ich mich bei:

- Herrn Dr. Robert O. Jones
- Herrn Dr. Jaakko Akola
- Frau Ursula Funk-Kath