



INSTITUT FÜR INFORMATIK
AG MEDIENINFORMATIK

Bachelorarbeit

Prozedurale Modellierung von Schneedecken

Manuel Schwarz

März 2012

Erstgutachter: Prof. Dr. Oliver Vornberger
Zweitgutachter: Prof. Dr. Sigrid Knust

Zusammenfassung

Die vorliegende Arbeit entstand in der Arbeitsgruppe Medieninformatik an der Universität Osnabrück im Bereich Computergrafik und beschäftigt sich mit prozeduraler Modellierung. Kern der Arbeit ist die Generierung von Schneedecken zur naturgetreueren Repräsentation von Wetterverhältnissen basierend auf Wetterdaten. Diese werden mit Hilfe eines angenährten Modells und der Verwendung des sogenannten Marching Cubes Algorithmus realisiert.

Inhaltsverzeichnis

1 Einleitung	1
1.1 Motivation	1
1.2 Zielsetzung	1
1.3 Aufbau der Arbeit	1
2 Grundlagen	3
2.1 Prozedurale Modellierung	3
2.2 Displacement Mapping	3
2.3 Punkt in Polygon - Algorithmus	4
2.4 Marching Cubes - Algorithmus	5
2.4.1 Bedeutung	5
2.4.2 Funktionsweise	6
3 Umsetzung	9
3.1 Ansätze	9
3.1.1 Snow-Map	9
3.1.2 Voxelrepräsentation	11
3.2 Szene auslesen	14
3.2.1 Wavefront-Dateiformat	14
3.2.2 Parser	15
3.3 Szene mit Voxeln füllen	16
3.4 Voxel in Objekt - Algorithmus	18
3.4.1 Aktive Faces berechnen	19
3.4.2 Schnittkanten der Faces mit der Ebene bestimmen	20
3.4.3 Punkt in Polgon - Algorithmus anwenden	22
3.5 Marching Cubes - Algorithmus	23
3.5.1 Imaginären Cube erstellen	24
3.5.2 Schnittkanten mit der Ebene bestimmen	25
3.5.3 TriangleLookupTable	27
3.5.4 Ergebnis	28
3.6 Schneedecke	29
3.6.1 Schnee von oben	31
3.6.2 Schnee von der Seite	33

3.6.3	Schneestabilität	33
4	Reflexion	37
4.1	Zusammenfassung	37
4.2	Fazit und Ausblick	37

Abbildungsverzeichnis

2.1	Beispiel für Displacement Mapping	4
2.2	Punkt in Polygon Algorithmus	4
2.3	Sonderfälle des Punkt in Polygon Algorithmus	5
2.4	Marching Cube	6
2.5	Marching Cube - Schnittmöglichkeiten	7
2.6	Marching Cubes - Polygonmodell eines Kopfes	7
3.1	Lightmap auf einer Wand	9
3.2	Beispiel für transparentes Displacement-Mapping	10
3.3	Überschneidung beim Displacement-Mapping	11
3.4	Voxel in Objektecken	12
3.5	Voxelnachbarschaft	12
3.6	Beispiel eines rechteckigen Objektes im Voxelgrid	13
3.7	Beispiel eines beliebigen Objektes im Voxelgrid	13
3.8	Voxelgitter	18
3.9	Aktive Faces	20
3.10	Ebene-Face Schnittmöglichkeiten	20
3.11	Schnittpunkt von einer Geraden mit einer Ebene	21
3.12	Voxel in Objekt - Algorithmus	23
3.13	Indizierung eines imaginären Würfels	24
3.14	Beispiel Cube	26
3.15	Marching Cubes Ergebnisse	28
3.16	Unterschiedlicher Detailgrad des Marching Cubes Algorithmus	29
3.17	Unterschied zwischen tatsächlicher und angenäherter Oberfläche	30
3.18	Annäherung eines rechteckigen Objekts mit Schneedecke	30
3.19	Anwachsen der Schneedecke	31
3.20	Schneedecke bei senkrechtem Schneefall	33
3.21	Schneedecke bei Schneefall von rechts oben	34
3.22	Schneeverteilung bei einer Punktquelle mit Stabilitätstest	34
3.23	Beispiel der Schneeverteilung bei alternativer Punktquelle	34
3.24	Senkrechter Schneefall ohne und mit Stabilitätstest	35

Kapitel 1

Einleitung

Diese Arbeit befasst sich mit dem Entwurf, der Konzeption und der Entwicklung einer Methode zur Repräsentation von Schneedecken.

1.1 Motivation

Im Rahmen der Masterprojektgruppe "Virtueller Campus" an der Universität Osnabrück im Sommersemester 2011 sowie Wintersemester 2011/12.

1.2 Zielsetzung

Ziel der Arbeit ist es eine richtige Schneedecke mit Hilfe prozeduraler Modellierung zu generieren und somit eine naturgetreue Wetterdarstellung zu gewährleisten.
Insbesondere stehen hierbei die Algorithmen zur Modellierung der Schneedecke im Vordergrund und weniger die Visualisierung.

1.3 Aufbau der Arbeit

Zunächst wird eine kurze Einführung in das Thema Prozedurale Modellierung gegeben. Diese beinhaltet die Entstehung, sowie den Boom Mitte der 80er Jahre sowie die Entwicklung bis heute.

Darauf folgen einige grundlegende Konzepte und Ideen deren Theorie erläutert wird. Im Kapitel Umsetzung werden diese Konzepte Implementiert und angewendet. Besonders im Fokus stehen hier der ins 3-dimensionale übertragene Punkt in Polygon- sowie der Marching Cubes Algorithmus.

Kapitel 2

Grundlagen

Im folgenden Kapitel sollen die dieser Arbeit zugrunde liegenden theoretischen Grundlagen aufgeführt und erklärt werden, bevor sie im Kapitel Umsetzung implementiert werden.

2.1 Prozedurale Modellierung

Zu Beginn soll der Begriff der prozeduralen Modellierung näher beleuchtet werden.

2.2 Displacement Mapping

Das Displacement-Mapping geht aus dem Bump-Mapping hervor und stellt eine Erweiterung dessen dar.[EMP⁺03, S. 9] Robert L. Cook beschreibt das Displacement-Mapping erstmals 1984.[Coo84] Dabei werden Texturen dazu benutzt die Objektoberfläche zu alternieren. Das heißt, es werden nicht nur die Normalen, wie beim Bump Mapping verändert um eine gewünschte Erscheinung der Oberfläche zu simulieren[Bli78], sondern es wird die tatsächliche Objektgeometrie modifiziert. Dies geschieht, indem die Vertices eines Objekts anhand der Werte einer Displacement-Map entlang der Normalen (senkrecht zur Oberfläche) verschoben und die Normalen entsprechend angepasst werden.

In der Abbildung 2.1 ist ein Beispiel für eine solche Displacement-Map und die Auswirkung auf die Oberfläche (Mesh) gegeben. Die Displacement-Map enthält dabei Farbwerte von schwarz (keine Veränderung) bis weiß(maximale Verschiebung des Vertex in Richtung der Normalen). Im dritten Bild ist die veränderte Silhouette der Oberfläche gut zu erkennen. Da bei dieser Vertexverschiebung auch die Normalen neu berechnet werden sieht das Displacement-Mapping oftmals dem Bump-Mapping sehr ähnlich, mit dem entscheidenden Unterschied, dass die Struktur, die durch das Displacement-Mapping erzeugt wird im Umriss des Objekts sichtbar ist.

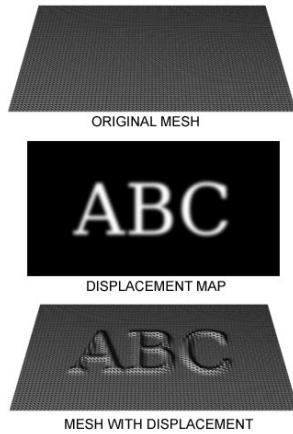


Abbildung 2.1: Beispiel für Displacement Mapping [Wik05]

2.3 Punkt in Polygon - Algorithmus

Der Punkt in Polyon - Algorithmus ist ein Verfahren um festzustellen, ob ein gegebener Punkt in einer Ebene innerhalb oder außerhalb der Grenzen eines Polygons liegt. Diese Methode findet vorwiegend Anwendung beim Verarbeiten geometrischer Daten, wie zum Beispiel in der Computergrafik.

Bereits 1974 wurden zwei Ansätze zur Lösung des Punkt-in-Polygon-Problems für nicht-konvexe Polygone (Ray-Casting und Winkelsumination) vorgestellt.[SSS74, S. 13f.] In dem Artikel von Sutherland wird der Algorithmus dazu verwendet festzustellen welche Oberflächen einer Szene näher am Betrachter liegen und welche durch andere verdeckt werden.

Im Folgenden wird näher auf das Konzept des Ray-Castings eingegangen, welches in Abbildung 2.2 illustriert wird. Punkte werden getestet, indem man einen imaginären "Strahl" in eine belie-

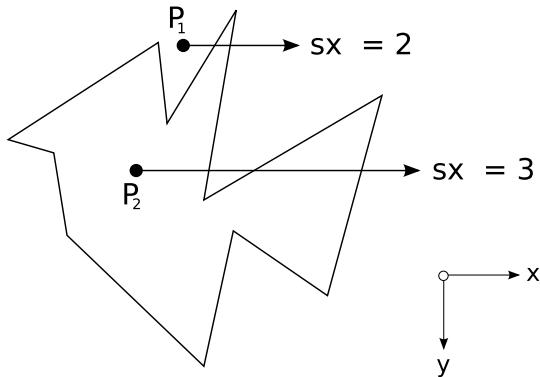


Abbildung 2.2: Punkt in Polygon Algorithmus

bige Richtung schickt und die Schnittpunkte mit den Kanten des Polygons zählt. Ist die Anzahl der Schnittpunkte gerade, so liegt der Punkt außerhalb, ist sie ungerade, so liegt der Punkt innerhalb des Polygons.

Um konsistente Ergebnisse zu erhalten falls ein oder mehrere Polygon-Vertices genau auf dem Test-Strahl liegen, ist es notwendig die Sonderfälle zu betrachten, die in Abbildung 2.3 dargestellt sind. Die Umgehung dieses Problems liegt in der Annahme, dass der Polygon-Vertex infinitisi-

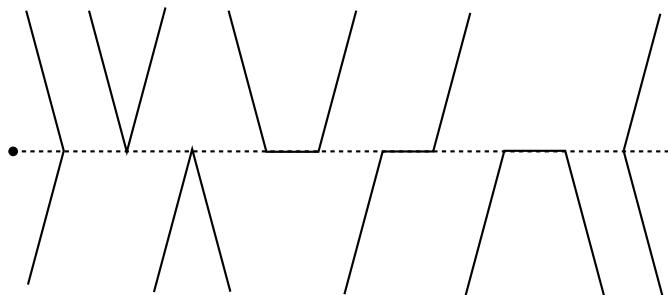


Abbildung 2.3: Sonderfälle des Punkt in Polygon Algorithmus

mal über dem Test-Strahl liegt. Durch dieses Vorgehen wird zusätzlich festgelegt, zu welchem Polygon ein Pixel gehört, sollten zwei Polygone einen oder mehrere Vertices gemeinsam haben. [Vor10, S. 51ff.]

2.4 Marching Cubes - Algorithmus

2.4.1 Bedeutung

Der Marching Cubes Algorithmus wurde im Jahr 1987 im Auftrag der General Electric Company von Lorensen und Cline entwickelt, um ein effizienteres Verfahren zur Visualisierung medizinischer Messdaten bereitzustellen. [LC87] Dabei wurde bis zum damaligen Zeitpunkt meist auf 2-dimensionale Bilddaten von bildgebenden Verfahren wie zum Beispiel der Computertomografie (CT) oder der Magnetresonanztomografie (MRT) zurückgegriffen. Da das Analysieren und Interpretieren dieser 2-dimensionalen Bilder (Scheiben oder Schichten) besondere Schulung und Erfahrung bedarf, bietet der Marching Cubes Algorithmus eine aussagekräftige, 3-dimensionale Repräsentation der evaluierten Messdaten zur Unterstützung von Medizinern in den unterschiedlichsten Bereichen.

Die vom CT bzw. MRT erzeugten Volumendaten liegen in Form von Voxel-Datenmengen vor. Das heißt, dass in regelmäßigen Abständen die Dichte des Materials ermittelt und gespeichert wird, wodurch ein gleichmäßiges 3-dimensionales Datengitter generiert wird. Die Nachteile eines solchen Modells sind der enorme Speicherbedarf und die langsame Visualisierung im Vergleich zu einfachen Drahtgittermodellen.

Der Marching Cubes Algorithmus ermöglicht nun die zuvor genannten Voxel-Datenmengen durch ein polygonales Oberflächenmodell anzunähern und somit effizient zu visualisieren.

2.4.2 Funktionsweise

Der Marching Cubes Algorithmus verfolgt einen Divide-and-Conquer-Ansatz. Um die gegebenen Volumen-Daten zu Triangulieren betrachtet man zunächst zwei benachbarte bzw. direkt übereinanderliegende Schichten oder Ebenen. In einem nächsten Schritt werden vier benachbarte Voxel auf der unteren Ebene z so miteinander verbunden, dass sie ein Quadrat ergeben. Danach bilden diese mit den genau darüberliegenden Voxeln der Ebene $z+1$ einen imaginären Würfel (siehe Abbildung 2.4). Dabei gibt es meist einen Referenzvoxel von dem aus die restlichen Ku-

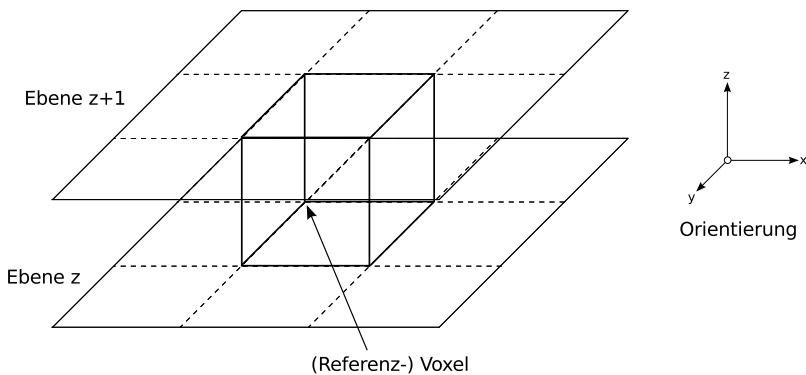


Abbildung 2.4: Marching Cube

busecken mit Hilfe eines Offsets bestimmt werden.

Damit anschließend eine Oberfläche generiert werden kann wird für jeden Eckpunkt geprüft, ob dieser innerhalb oder außerhalb des betrachteten Objektes liegt, was anhand der Dichtewerte bestimmt werden kann. Liegen alle Eckpunkte innerhalb oder außerhalb, so muss dieser Würfel nicht weiter untersucht werden und es wird zum Nächsten "marschiert". Liegt hingegen ein Schnittpunkt des Kubus mit der Objektoberfläche vor, so teilt diese den Marching Cube unweigerlich in Innen- und Außenbereiche auf.

Da der Würfel acht Eckpunkte besitzt und jeder Eckpunkt zwei Zustände haben kann, gibt es genau $2^8 = 256$ Möglichkeiten den Kubus zu zerteilen. Aus Symmetriegründen können diese 256 Fälle auf die in Abbildung 2.5 dargestellten 15 reduziert werden. Sind schließlich die Zustände aller beteiligten Voxel bestimmt, so wird in einer Tabelle, der sogenannten `TriangleLookupTable` nachgesehen, welche Dreiecke in dem vorliegenden Fall gezeichnet werden müssen. Genauer gesagt, werden die Würfelkanten bestimmt, die von der Oberfläche geschnitten werden und wo auf den Kanten die Eckpunkte der zu zeichnenden Dreiecke liegen. In dem Kapitel 3.5 wird dieses Verfahren und die `TriangleLookupTable` genauer erläutert.

Wahlweise können für eine bessere Annäherung an das Ursprungsobjekt die Oberflächenpunkte entsprechend interpoliert werden. Ansonsten wird für die Berechnung der Dreiecke immer von der Mitte einer Würfelkante ausgegangen. Zur besseren Darstellung (Beleuchtung) lassen sich daraufhin die Einheitsnormalen berechnen.

Ein Beispiel für die Visualisierung eines aus 150 Schichten bestehenden MRT-Modells soll die Abbildung 2.6 geben, in der die Annäherung an einen Kopf deutlich zu erkennen ist.

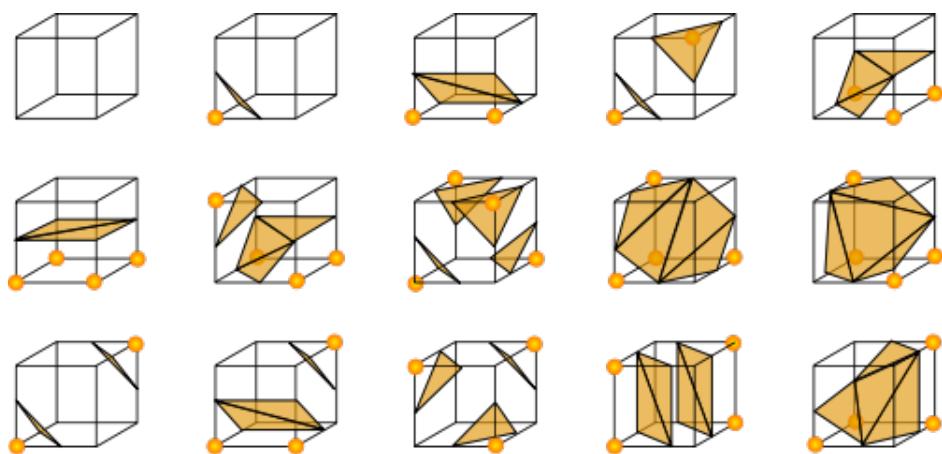


Abbildung 2.5: Marching Cube - Schnittmöglichkeiten [Wik12a]

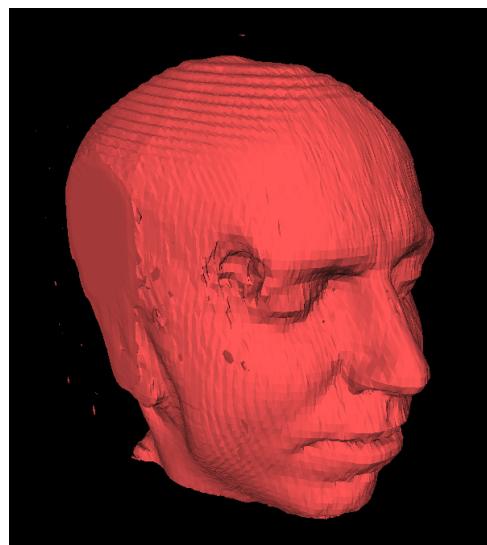


Abbildung 2.6: Marching Cubes - Polygonmodell eines Kopfes [Wik12b]

Kapitel 3

Umsetzung

3.1 Ansätze

Dieses Kapitel befasst sich mit möglichen Lösungsansätzen zur Erstellung einer Schneedecke und stellt drei verschiedene Ansätze dar, von denen der Letztgenannte umgesetzt werden soll.

3.1.1 Snow-Map

Interpretation als Lightmap

Ein erster, naiver Ansatz eine Schneedecke zu modellieren ist es, eine Textur zu erstellen, die die Aufgabe hat die Menge an gefallenem Schnee an jeder Stelle der Szene zu speichern. Diese Textur wird im Folgenden als Snow-Map bezeichnet. Die Snow-Map könnte anschließend ähnlich wie eine Lightmap interpretiert werden. Das heißt, dass in die ursprüngliche Farbe der Objekttextur entsprechend der Menge an Schnee weißhineingemischt wird um die Farbe aufzuhellen. Das Prinzip einer Lightmap ist in Abbildung (3.1) verdeutlicht. Die Abbildung zeigt die Auswirkung

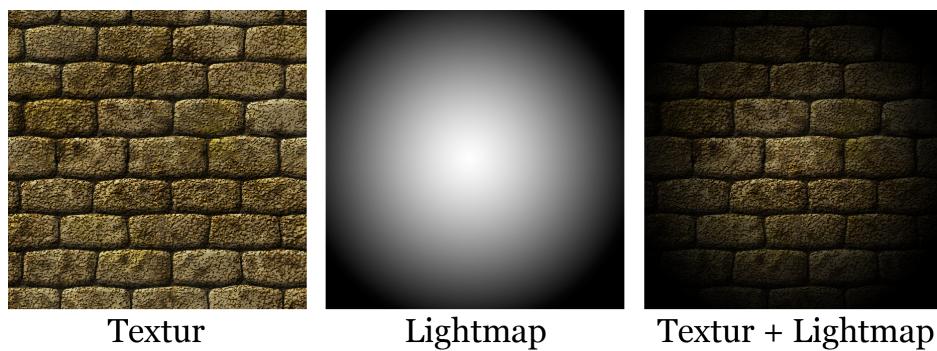


Abbildung 3.1: Lightmap auf einer Wand

einer Lightmap in Form eines Spotlights auf eine Mauer-Textur. Ebenso würde sich die Snow-Map bei der Simulation von Schnee verhalten. Die dabei entstehenden Probleme liegen auf der Hand. Es können lediglich hellere Stellen oder Flächen auf der bereits vorhandenen Oberfläche entstehen. Läuft die Simulation lange genug, bzw. fällt genügend Schnee, so ist nach gewisser Zeit die gesamte Oberfläche der Szene weiß und eine Unterscheidung zwischen viel und wenig Schnee ist nicht mehr möglich. Die Nutzung einer einfachen Textur bedeutet letztlich, dass keine richtige Schneedecke vorhanden ist, wodurch wiederum keine Schneehäufungen oder -ansammlungen deutlich gemacht werden können. Dieser Ansatz führt folglich zu keinem zufriedenstellenden Ergebnis.

Interpretation als Light- und Displacement-Map

Zur Verbesserung des Ergebnisses, das allein mit der Lightmap erzeugt werden kann, wäre es denkbar die Snow-Map als eine Kombination aus Light- und Displacement-Map zu interpretieren. Dabei arbeitet die Lightmap genauso wie im vorherigen Abschnitt beschrieben und die Funktionsweise der Displacement-Map ist dem Kapitel 2.2 zu entnehmen.

Zusätzlich zu der dort gegebenen Beschreibung würde man eine Schneogeometrie definieren, die initial mit der Szenengeometrie übereinstimmt und die Werte in der Snow-Map als Transparenzfaktor betrachten. Damit ist gemeint, dass je dunkler der Farbwert in der Snow-Map ist, desto transparenter ist die Schneogeometrie an dieser Stelle, so dass man die Szene darunter noch erkennen kann. Umgekehrt ist die Schneogeometrie undurchsichtiger je heller der Wert der Snow-Map ist. Hierbei ist es wichtig zu bedenken, dass die Schneogeometrie von Anfang an weiß ist und nur die Transparenz die Menge an Schnee wiederspiegelt. Somit ließen sich Erhebungen

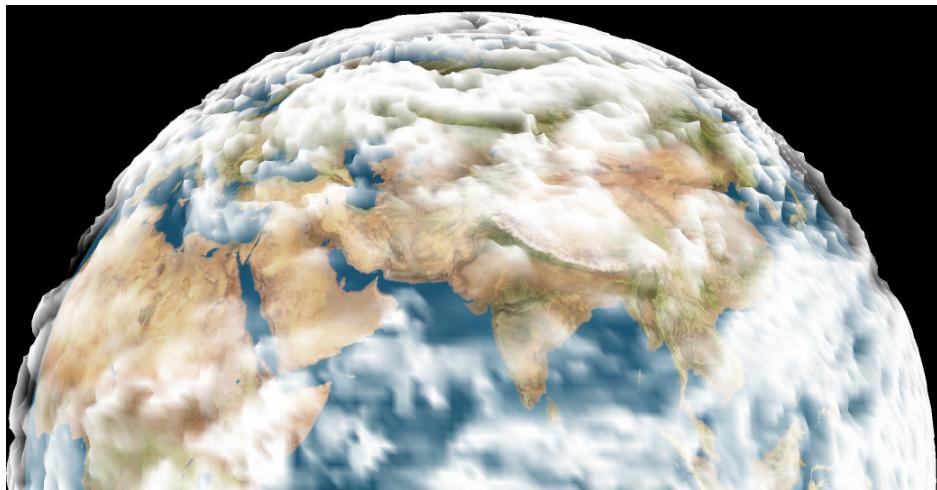


Abbildung 3.2: Beispiel für transparentes Displacement-Mapping [Wen05]

und Schneehäufungen darstellen, die zudem passend eingefärbt wären. Jedoch wird man bei dieser Vorgehensweise ebenfalls mit gewissen Problemen konfrontiert. Zwar ist das Ergebnis bei einem steilen Betrachtungswinkel durchaus zufriedenstellend, jedoch kann es bei der Visualisierung passieren, dass aufgrund der Transparenz bei einem flachen Betrachtungswinkel lediglich

schwebende Pyramiden- bzw. Kegelspitzen zu sehen sind, wie am Rand der Erdkugel in Abbildung 3.2. Des Weiteren können unerwünschte Überschneidungen in der Nähe von Objektecken auftreten. Ein mögliches Problem ist in Abbildung 3.3 aufgezeigt. Aufgrund der sich verändern-

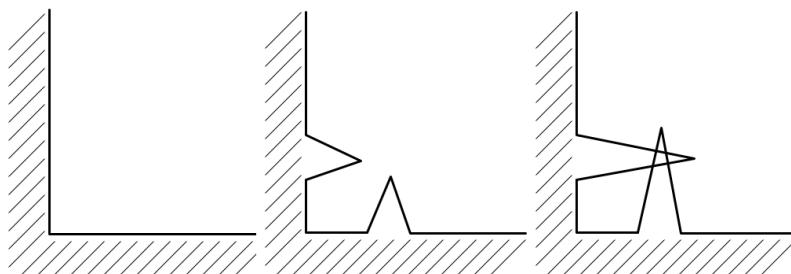


Abbildung 3.3: Überschneidung beim Displacement-Mapping

den Schneogeometrie und eventuell wachsenden Schneemengen könnte es zu Überschneidungen der Schneedecke mit sich selbst an solchen Ecken kommen (siehe Bild 3). Da allerdings weder schwebende Kegelspitzen sichtbar sein, noch seltsame Effekte beim Rendern durch Überschneidungen auftreten sollen, könnte auch dieser Ansatz kein befriedigendes Ergebnis liefern.

3.1.2 Voxelrepräsentation

Die Lösung des Problems der bisher fehlenden Schneedecke beziehungsweise der sich selbst schneidenden Displacement-Oberfläche kann durch die Repräsentation durch Voxel verwirklicht werden.

Dabei wird mit Hilfe von Voxeln eine richtige Schneedecke erzeugt, die auf bzw. über der Szenenoberfläche liegt. Dies wird erreicht, indem der gesamte Raum, den die Szene einnimmt mit Voxeln gefüllt wird, sodass die Schneedecke später überall wachsen kann. Darauffolgend wird jeder Voxel geprüft, ob er innerhalb oder außerhalb eines Objektes liegt, um im Anschluß mit Hilfe des sogenannten Marching Cubes Algorithmus (siehe Kapitel 3.5) eine Schneedecke zu erzeugen (siehe Kapitel 3.6). Demnach ist keine Überschneidung der Displacement-Oberfläche in den Ecken mehr möglich, da die Voxel in einem regelmäßigen Gitter gesetzt werden und jeder Voxel nur einmal existiert. Die Abbildung 3.4 verdeutlicht das gleichmäßige Voxelgrid sowie die Lösung des Überschneidungsproblems. Wächst zum Beispiel von unten und von links der Schnee an, so trifft die Schneeoberfläche auf den selben Voxel und die Schneedecke erweitert sich bis zu diesem Punkt. Folglich kann es keine ungewollten Überschneidungen geben. Die Bereiche "Innen" und "Außen" stehen in der Zeichnung für Gebiete innerhalb sowie außerhalb von einem Objekt.

Im Folgenden wird erläutert, was ein Voxel in diesem Fall genau ist, welche Eigenschaften er hat und welche Bedeutung diese für den weiteren Verlauf der Arbeit haben.

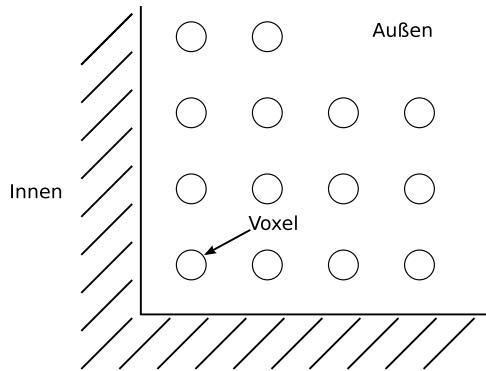


Abbildung 3.4: Voxel in Objektecken

Voxel

Wie zuvor bereits angedeutet sind Voxel im Grunde Punkte im Raum mit gewissen Eigenschaften. Ein Voxel besitzt folglich eine x -, y - und z -Koordinate zur eindeutigen Festlegung der Position. Des Weiteren wird gespeichert, ob ein Voxel sich inner- oder außerhalb eines Objekts befindet, was mit Hilfe des dreidimensionalen Punkt in Polygon Algorithmus ermittelt wird. Zudem wird vorgehalten, ob ein Voxel Schnee beinhaltet. Ist dies der Fall, so wird in einer weiteren Variablen ein gewisser **density**-Wert festgehalten, der aussagt, wie viel Schnee der Voxel enthält.

Als letzte wichtige Information merkt sich jeder Voxel seine bis zu sechs direkten Nachbarn. Die zugrunde liegende Nachbarschaftsbeziehung ist in Abbildung 3.5 dargestellt. Ein Voxel in der

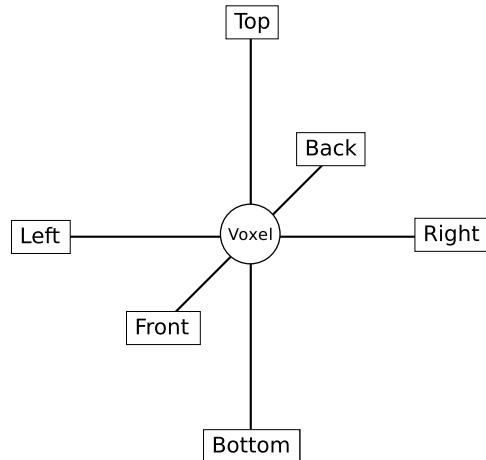


Abbildung 3.5: Voxelnachbarschaft

Szenenmitte hat somit stets 6 Nachbarn, Randvoxel haben 5, Voxel auf einer Szenenkante 4 und die Voxel in den vier Eckpunkten haben lediglich 3 direkte Nachbarn.

1. Ansatz: rechteckige/rasterangepasste Objekte

Ein erster, einfacherer Ansatz ist in Abbildung 3.6 dargestellt. Die Kreise repräsentieren Voxel, wobei diese rot eingefärbt sind, wenn sie in einem Objekt liegen. Zur besseren Übersicht sind die Abbildungen in diesem Abschnitt zweidimensional. Gegeben ist ein rechteckiges Objekt, dessen

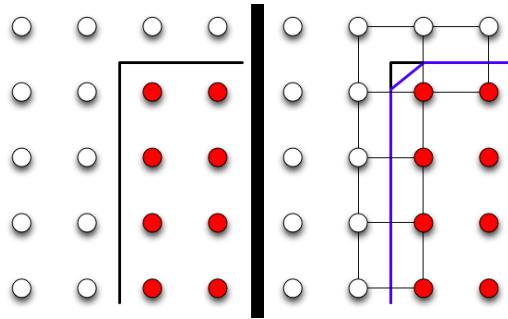


Abbildung 3.6: Beispiel eines rechteckigen Objektes im Voxelgrid

Kanten exakt zwischen den Voxeln liegen (linkes Bild). In diesem Fall liefert der Marching Cubes Algorithmus ein sehr gutes Ergebnis, da er als Grundlage für die Schnittpunktberechnung im Standardfall die Kantenmitte zugrunde legt. Involvierte bzw. geschnittene Würfel sind als solche hervorgehoben. Die resultierende, initial erzeugte Oberfläche ist im rechten Bild durch die blaue Linie gekennzeichnet und stimmt fast exakt mit der Oberfläche des Objekts überein.

2. Ansatz: beliebige Objekte

Der zweite Ansatz verfolgt nun das Ziel das Ergebnis des ersten Ansatzes für beliebige Objekte zu erzeugen. Um folglich eine Oberfläche wie im linken Bild der Abbildung 3.7 möglichst exakt nachzubilden, müssen die Schnittpunkte zwischen den Kuben und der Objektoberfläche entsprechend genau berechnet bzw. interpoliert werden. Setzt man bei einer solchen beliebigen

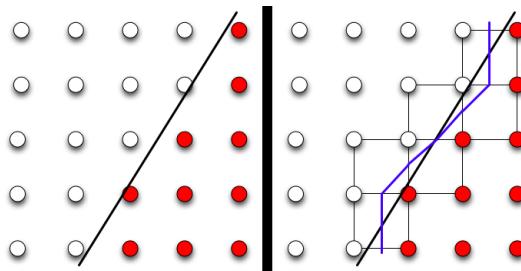


Abbildung 3.7: Beispiel eines beliebigen Objektes im Voxelgrid

Oberfläche die Schnittpunkte allerdings in die Kantenmitte, so erhält man eine initial angenäherte Lösung, die durch die blaue Linie im rechten Bild dargestellt wird. Es ist deutlich zu sehen, dass die Oberfläche teilweise über und teilweise unter der tatsächlichen Objektoberfläche liegt.

Dies kann dazu führen, dass die Schneedecke leicht über dem Objekt schwebt.

Aufgrund der höheren Komplexität des zweiten Ansatzes wird in der vorliegenden Arbeit der erste Ansatz verfolgt und umgesetzt. Dies hat zur Folge, dass die durch das Programm erzeugten Ergebnisse für Objekte optimiert sind, deren Kanten mit dem Voxelgitter übereinstimmen, also vorwiegend rechteckig sind. Natürlich werden auch Ergebnisse für beliebige Objekte geliefert, diese sind allerdings aus den oben genannten Gründen unpräziser. Natürlich erhöht sich die Qualität des Ergebnisses mit steigender Anzahl von Voxeln, welche ab einer bestimmten Menge und Szenengröße jedoch unpraktikabel wird. Zudem spielt dies nur bei der ersten Anwendung des Marching Cubes Algorithmus eine Rolle, denn sobald die Schneedecke anwächst soll diese über dem Objekt liegen. In Kapitel 3.6 wird dieses Problem erneut aufgegriffen und anhand von Applikationsgrafiken illustriert.

3.2 Szene auslesen

Um nun eine Schneedecke simulieren zu können, muss die vorliegende Szene zunächst ausgelernt werden. Das Programm kann Szenen im Wavefront-Format einlesen. Es folgt eine kurze Erklärung dieses Dateiformats sowie die Beschreibung und Funktionsweise des Parsers.

3.2.1 Wavefront-Dateiformat

Eine Wavefront-Datei enthält alle für die Szene relevanten Informationen, wie Vertices, Normalen, Faces, Texturkoordinaten oder benutzte Oberflächenmaterialien. In dem folgenden Dateiausschnitt sind die für diese Applikation erforderlichen Daten beispielhaft aufgeführt.

```
# Blender v2.62 (sub 0) OBJ File: 'cube.obj'
# www.blender.org
v -1.000000 1.000000 1.000000
v -1.000000 -1.000000 1.000000
v 1.000000 -1.000000 1.000000
...
vn 0.000000 0.000000 1.000000
vn 0.000000 0.000000 -1.000000
vn -1.000000 -0.000000 -0.000000
...
f 1//1 2//1 3//1
f 1//1 3//1 4//1
f 5//2 8//2 7//2
...
```

Die ersten beiden Zeilen können ignoriert werden, da es Kommentare sind, deren Inhalt hier irrelevant ist. Es sind lediglich jene Zeilen von Bedeutung, die mit "v", "vn" oder "f" beginnen.

Das Präfix „v“ leitet einen Vertex ein, gefolgt von den zugehörigen x-, y- und z-Koordinaten, welche jeweils durch ein Leerzeichen getrennt sind. Analog dazu sind die Vertexnormalen aufgelistet, die durch das Präfix „vn“ eingeleitet werden, mit dem Unterschied, dass die Koordinaten keine Position, sondern eine Richtung angeben.

Sowohl für die Vertices als auch für die Normalen gilt, dass sie nur einmal aufgelistet werden. Sollten beispielsweise an der selben Stelle mehrere Vertices liegen, so wird dieser eine Vertex nicht mehrfach aufgeführt, sondern mehrfach verwendet (z.B. wenn mehrere Faces den selben Vertex als Eckpunkt verwenden). Dies geschieht mit Hilfe einer impliziten Indizierung. Jeder Vertex und jede Normale hat einen eindeutigen Index, der durch ihre jeweilige Position in der Liste gegeben ist.

Implizit sieht die Datei nun wie folgt aus:

```
# Blender v2.62 (sub 0) OBJ File: 'cube.blend'
# www.blender.org
1 v -1.000000 1.000000 1.000000
2 v -1.000000 -1.000000 1.000000
3 v 1.000000 -1.000000 1.000000
...
1 vn 0.000000 0.000000 1.000000
2 vn 0.000000 0.000000 -1.000000
3 vn -1.000000 -0.000000 -0.000000
...
```

Schließlich leitet das Präfix „f“ ein Face ein. Hinter jedem „f“ stehen drei durch Leerzeichen getrennte Strings in dem Format „<vertex_index>//<normal_index>“.

Somit wird festgelegt, welche Vertices miteinander zu einem Face verbunden werden müssen und welche Normale der jeweilige Vertex hat. Beispiel:

```
f 5//2 8//2 7//2
```

Dieses Face besteht folglich aus dem Vertex mit dem Index 5, dem Vertex 8 und dem Vertex 7, welche in genau dieser Reihenfolge miteinander verbunden werden. Zudem haben in diesem Fall alle drei Vertices die Normale mit dem Index 2.

3.2.2 Parser

Mit dem oben beschriebenen Wissen über das vorliegende Dateiformat kann nun ein entsprechender Parser zum Auslesen der Szene entwickelt werden. Dieser liest die Werte ein und speichert sie zur weiteren Verarbeitung in mehreren Arrays.

Dabei wird zunächst die gesamte Datei durchgegangen um die Vorkommen der „v“, „vn“ und „f“ zu zählen und entsprechend große Arrays anzulegen. Diese Arrays sind:

- **vertexArray**: Enthält die Vertex-Koordinaten, die hintereinander in das Array eingetragen werden.

- **normalArray**: Enthält die (Richtungs-) Koordinaten der Normalen, die ebenfalls hintereinander in dem Array stehen.
- **faceArray**: Enthält die Indizes der Vertices die zu einem Face verbunden werden. Ein Face besteht dabei immer aus drei Vertices.
- **normalIndexArray**: Enthält analog zum **faceArray** die Indizes der Normalen.

Nachdem die Arrays erstellt wurden, wird die Datei mit Hilfe der folgenden `while`-Schleife erneut durchgegangen um die Arrays zu füllen.

```

1 while((line = br.readLine()) != null)
2 {
3     if(getPrefix(line).equals(VERTEX_PREFIX))
4     {
5         addVertices(line);
6     }
7     else if(getPrefix(line).equals(FACE_PREFIX))
8     {
9         addFaces(line);
10    }
11    else if(getPrefix(line).equals(NORMAL_PREFIX))
12    {
13        addNormals(line);
14    }
15 }
```

Hierbei werden jeweils die ersten zwei Zeichen jeder Zeile mit den oben genannten Präfixen (hier repräsentiert durch Konstanten) verglichen, wodurch festgestellt wird, ob es sich um einen Vertex, eine Normale oder ein Face handelt. Je nach Übereinstimmung wird die entsprechende Methode aufgerufen, welche die Zeile durchgeht und die einzelnen Werte in das korrekte Array einträgt. Dabei ist zu erwähnen, dass die Methode `addFaces(String line)` beiden Index-Arrays (`faceArray` und `normalIndexArray`) ihre Werte zuweist, da dort immer zwei Informationen in einer Zeile stehen.

3.3 Szene mit Voxeln füllen

Nachdem die Szene nun ausgelesen werden kann und die Arrays mit den benötigten Daten zur Verfügung stehen, soll die Szene mit Voxeln gefüllt werden.

Um ein regelmäßiges Voxelgitter zu erstellen muss als erstes die räumliche Ausdehnung der Voxel bestimmt werden, das heißt die Größe der Szene ist zu ermitteln. Mit Hilfe des Vertex-Arrays lässt sich nun der größte bzw. kleinste in der Szene enthaltene Wert auf jeder der drei Achsen herausfinden. Daraus lassen sich nun drei Kantenlängen berechnen aus denen wiederum ein Quader ermittelt werden kann, der die Szene enthält und mit Voxeln aufgefüllt wird.

Dazu wird das Vertex-Array betrachtet und der jeweils kleinste bzw. größte Wert jeder Koordinatenachse herausgesucht.

Um schließlich den daraus entstehenden Quader mit Voxeln zu füllen wird die folgende Methode verwendet.

```

1  private void fillScene(float[] vertices)
2  {
3      float maxX = calculateMaxX(vertices);
4      float minX = calculateMinX(vertices);
5      float maxY = calculateMaxY(vertices);
6      float minY = calculateMinY(vertices);
7      float maxZ = calculateMaxZ(vertices);
8      float minZ = calculateMinZ(vertices);
9
10     for (float z = minZ; z <= maxZ; z += 1.0f / STEPS)
11     {
12         for (float x = minX; x <= maxX; x += 1.0f / STEPS)
13         {
14             for (float y = minY; y <= maxY; y += 1.0f / STEPS)
15             {
16                 setVoxel(x, y, z);
17             }
18         }
19     }
20 }
```

Man geht entlang jeder Koordinatenachse die zuvor berechnete Kantenlänge ab und setzt einen Voxel an den entsprechenden Koordinaten. Da es sich um drei Dimensionen handelt wird dieses Vorgehen mit drei ineinander geschachtelten `for`-Schleifen realisiert. Die Konstante `STEPS` stellt dabei eine Art Granularitätsfaktor dar. Das heißt pro Längeneinheit im Koordinatensystem werden `STEPS` Voxel gesetzt.

Das Ergebnis ist in Abbildung 3.8 zu sehen. Als Grundlage der Szene wurde ein Bauer eines Schachspiels beispielhaft herangezogen (links im Bild). Die rechte Bildhälfte stellt die ausgelesene Szene dar in der nur die gesetzten Voxel angezeigt werden.

Nachdem die Szene gefüllt und alle benötigten Voxel erzeugt wurden, sind im Anschluß die Nachbarschaftsbeziehungen für jeden Voxel zu setzen. Beispielaufgabe sei einmal die Bestimmung des rechten Nachbarvoxels aufgeführt.

```

1  private void setRightNeighbor(Voxel[] voxels, float distance)
2  {
3      for(int i = 0; i < voxels.length; i++)
4      {
5          float tmp = voxels[i].getX() - this.x;
6          if(tmp <= distance && tmp > 0.0f && this.y == voxels[i].getY() &&
7              this.z == voxels[i].getZ())
```

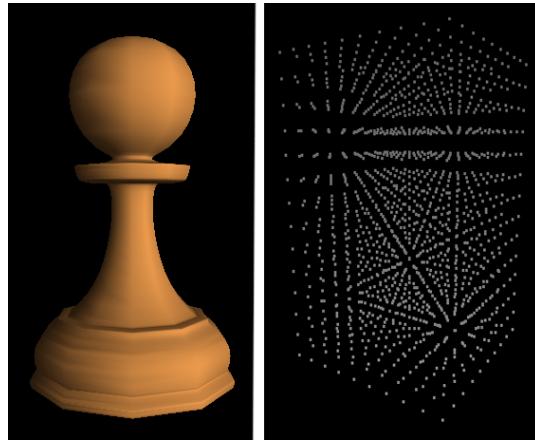


Abbildung 3.8: Voxelgitter

```

8      {
9          this.rightNeighbor = voxels[i];
10         return;
11     }
12   }
13 }
```

Die Methode bekommt das gesamte Voxelgitter sowie die Distanz zwischen zwei benachbarten Voxeln übergeben. Die Distanz hängt von der zuvor bestimmten Voxeldichte ab. Es wird für jeden Voxel der Szene geprüft, ob dieser die richtigen Bedingungen erfüllt. Zunächst wird der Abstand zwischen dem aktuellen Voxel und dem untersuchten auf der x-Achse berechnet (Zeile 5) und in `tmp` gespeichert. Für diesen Abstand muss nun gelten, dass er der vorgegebenen Distanz entspricht und positiv ist, da der rechte Nachbarvoxel auch rechts von dem aktuellen Voxel liegen sollte. Weiterhin ist zu beachten, dass der Nachbarvoxel auf der selben Höhe (gleiche z-Koordinate) sowie direkt neben (gleiche y-Koordinate) liegen muss.

Wurde ein passender Kandidat gefunden, so wird sich dieser gemerkt und die Methode kann verlassen werden, da es höchstens einen rechten Nachbarn geben kann. Die Bestimmung der fünf restlichen Nachbarn erfolgt analog. Dieser Prozess kann je nach Granularität eine gewisse Zeit beanspruchen, da für jeden Voxel das gesamte Voxelgrid linear durchsucht werden muss.

3.4 Voxel in Objekt - Algorithmus

In diesem Abschnitt soll die Funktionsweise des 3-dimensionalen Punkt in Polygon Algorithmus erläutert werden. Dieser dient dazu festzustellen, ob ein Voxel in einem Objekt liegt. Mit dieser Information lässt sich später bestimmen an welchen Stellen der Szene Schnee fallen kann und an welchen nicht.

3.4.1 Aktive Faces berechnen

Es sollen nicht immer alle in der Szene enthaltenen Faces auf Schnittpunkte mit dem aus dem Punkt in Polygon bekannten, imaginären "Strahl" geprüft werden, sondern lediglich die für die jeweilige Ebene relevanten.

Da der Algorithmus die Szene ebenenweise durchgeht müssen die Faces, die die aktuelle Ebene schneiden nur ein Mal pro Ebene ermittelt werden. Das bedeutet, dass alle Voxel in der aktuellen x-y-Ebene die selbe z-Koordinate besitzen und nur mit den gerade aktiven Faces geprüft werden müssen.

```

1  private int[] activeFaces(float z)
2  {
3      int j = 0;
4      int[] activeFaces = new int[this.faces.length];
5
6      for(int i = 0; i < this.faces.length; i += 3)
7      {
8          if(!((this.vertices[this.faces[i]] * 3 - 1) < z) &&
9              (this.vertices[this.faces[i + 1]] * 3 - 1) < z) &&
10             (this.vertices[this.faces[i + 2]] * 3 - 1) < z)) &&
11             !((this.vertices[this.faces[i]] * 3 - 1) > z) &&
12             (this.vertices[this.faces[i + 1]] * 3 - 1) > z) &&
13             (this.vertices[this.faces[i + 2]] * 3 - 1) > z)))
14          {
15              activeFaces[j++] = this.faces[i];
16              activeFaces[j++] = this.faces[i + 1];
17              activeFaces[j++] = this.faces[i + 2];
18          }
19      }
20      return activeFaces;
21  }

```

Welches Face die betrachtete Ebene schneidet bestimmt die z-Koordinate der drei Face-Vertices. Sind alle z-Werte kleiner oder größer als der z-Wert der Ebene, so gibt es offensichtlich keine Schnittkante. Sobald mindestens eine z-Koordinate größer und eine andere kleiner ist als die der Ebene, so muss es einen Schnitt geben und das Face wird zu den aktiven Faces hinzugefügt. Diese Abfrage findet in dem if-Statement im oberen Code-Ausschnitt statt (Zeile 8-13). In der Abbildung 3.9 sind beispielhaft für eine Ebene die aktiven Faces einmal dargestellt. Zudem werden zur besseren Übersicht lediglich die innenliegenden Voxel angezeigt.

In einem nächsten Schritt müssen nun die genauen Schnittkanten der Faces mit der entsprechenden Ebene berechnet werden.

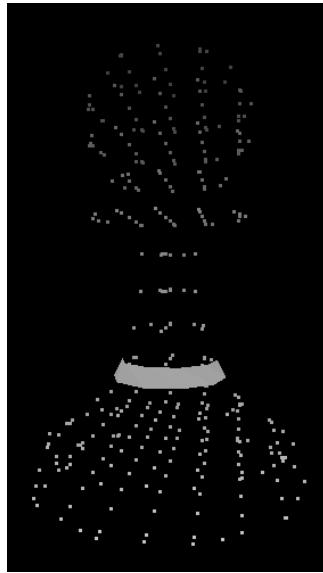


Abbildung 3.9: Aktive Faces

3.4.2 Schnittkanten der Faces mit der Ebene bestimmen

Die Schnittkantenermittlung der aktiven Faces mit der aktuell betrachteten Ebene ist entscheidend um im Folgenden den zweidimensionalen Punkt in Polygon Algorithmus anwenden zu können.

Dazu werden zunächst alle Möglichkeiten ein Face zu schneiden betrachtet (siehe Abbildung 3.10). Es können sechs verschiedene Fälle unterschieden werden. Für die Fallunterscheidung si-

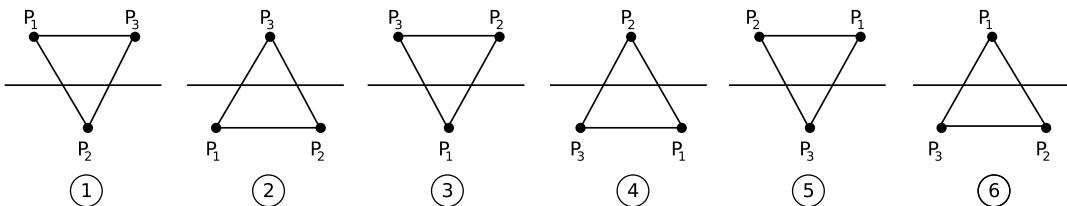


Abbildung 3.10: Ebene-Face Schnittmöglichkeiten

gnifikannt sind die z-Koordinaten der drei Vertices jedes Faces. Es wird geprüft, welcher Vertex oberhalb und welcher unterhalb der Ebene liegt. Aus Symmetriegründen können bei der weiteren Berechnung die Fälle 1 und 4, 2 und 5 sowie 3 und 6 zusammengefasst werden, da jeweils die gleichen Seiten des Face-Dreiecks geschnitten werden.

Um die genaue Schnittkante zwischen dem Face und der Ebene zu berechnen wird bestimmt, welche beiden Seiten des Faces geschnitten werden. Aus den daraus resultierenden Schnittpunkten lässt sich anschließend die Schnittkante zusammensetzen.

Das bedeutet, dass für jedes Face zweimal ein Schnittpunkt zwischen einer Gerade und der aktuellen Ebene durchgeführt werden muss. Umgesetzt wird dies mit Hilfe einfacher Vektorgeo-

metrie. Zur besseren Veranschaulichung ist die Abbildung 3.11 gegeben. Exemplarisch soll sich

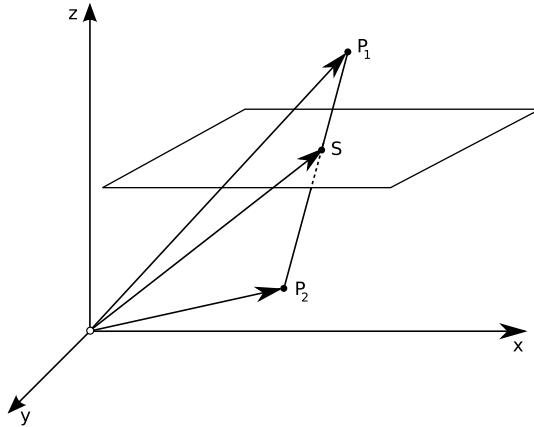


Abbildung 3.11: Schnittpunkt von einer Geraden mit einer Ebene

hier auf eine Seite und deren Schnittpunkt mit einer Ebene beschränkt werden. Gegeben sind die Punkte P_1 und P_2 und gesucht ist der Schnittpunkt S , wobei die z -Koordinate von S durch die Ebene bereits gegeben ist. Insbesondere sind folglich die x - und die y -Koordinate von S von Interesse. Die Punkte können ebenso als Vektoren betrachtet werden, so, wie in der Zeichnung angedeutet. Auch die Kante $\overrightarrow{P_2P_1}$ selbst kann als Vektor dargestellt werden durch

$$\overrightarrow{P_2P_1} = P_1 - P_2$$

Da der Schnittpunkt offensichtlich auf diesem Vektor liegen muss, kann S zunächst wie folgt geschrieben werden:

$$S = \lambda \overrightarrow{P_2P_1}$$

Dabei ist λ ein Skalierungsfaktor für den gilt

$$\lambda = \frac{z^* - z_2}{z_1 - z_2}$$

Der Wert z^* stellt die gegebene z -Koordinate der betrachteten Ebene dar und z_1 bzw. z_2 die z -Koordinaten der beiden Punkte P_1 und P_2 .

Des Weiteren muss berücksichtigt werden, dass S im Moment noch von P_2 und nicht vom Ursprung ausgeht. Um die richtigen Koordinaten von S zu erhalten muss zuletzt noch der Vektor \vec{P}_2 zu dem Ergebnis hinzugefügt werden, sodass sich für die gesuchten x - und y -Koordinaten des Schnittpunktes folgende Formeln aufstellen lassen:

$$x_1^* = \frac{z^* - z_2}{z_1 - z_2} (x_1 - x_2) + x_2$$

$$y_1^* = \frac{z^* - z_2}{z_1 - z_2} (y_1 - y_2) + y_2$$

Analog lassen sich die Koordinaten x_2^* und y_2^* des Schnittpunktes der zweiten Kante berechnen. Zum Schluss werden die Koordinaten aller Schnittkanten in einem `edges`-Array gespeichert, wobei eine Kante immer aus vier Koordinaten besteht, die hintereinander in das Array eingetragen werden.

Mit den so berechneten Schnittkanten kann im Folgenden der Punkt in Polygon Algorithmus im Zweidimensionalen angewandt werden.

3.4.3 Punkt in Polgon - Algorithmus anwenden

Durch die im vorherigen Abschnitt 3.4.2 bestimmten Schnittkanten zwischen der Ebene und den betroffenen Faces kann im Folgenden der in Kapitel 2.3 bereits theoretisch erläuterte, zweidimensionale Punkt in Polygon - Algorithmus angewandt werden.

Die `punktInPolygon`-Methode bekommt hierzu die Koordinaten des zu testenden Voxels und ein Array mir allen Polygonkanten der Ebene übergeben.

```

1  private boolean pointInPolygon(float x, float y, float[] edges)
2  {
3      boolean inside = false;
4      for(int i = 0; i < edges.length; i += 4)
5      {
6          float x1 = edges[i];
7          float y1 = edges[i + 1];
8          float x2 = edges[i + 2];
9          float y2 = edges[i + 3];
10
11         boolean startOver = y1 >= y;
12         boolean endOver = y2 >= y;
13
14         if(startOver != endOver)
15         {
16             float sx = ((float) (y * (x2 - x1) - y1 * x2 + y2 * x1) /
17                         (float) (y2 - y1));
18
19             if(sx >= x)
20             {
21                 inside = !inside;
22             }
23         }
24     }
25     return inside;
26 }
```

Dabei muss für jede Polygonkante geprüft werden, ob sie einen Schnittpunkt mit dem Test-Strahl hat, oder nicht. Da jede Kante aus zwei Punkten, bzw. vier Koordinaten besteht, werden diese

zu Beginn aus dem `edges`-Array geholt und zwischengespeichert. Schließlich gibt es nur zwei relevante Zustände (innerhalb und außerhalb) die in einer Variable `inside` vom Typ `boolean` gespeichert werden können.

Zur effizienteren Berechnung wird der Test-Strahl in Richtung der positiven `x`-Achse geschossen. Um nur die relevanten Kanten auf einen Schnittpunkt mit dem Strahl zu testen, wird vorab geprüft, ob Anfangs- und Endpunkt der aktuellen Polygonkante beide oberhalb oder beide unterhalb des Teststrahls liegen. Ist dies der Fall, so liegt kein Schnittpunkt vor. Dabei wird die Annahme, dass ein Polygonvertex infinitesimal über dem Test-Strahl liegt in Zeile 11 und 12 umgesetzt. Einen Schnittpunkt kann es nur geben, wenn ein Punkt oberhalb und der andere unterhalb des Strahls liegt, wobei dann die `x`-Koordinaten genauer untersucht werden. Liegen beide Punkte rechts von dem zu prüfenden Punkt (d.h. $x_1 > x$ und $x_2 > x$), so schneidet die Kante den Strahl irgendwo und `inside` wechselt den Wert. Sind die `x`-Koordinaten der beiden Punkte kleiner als `x`, dann gibt es keinen Schnittpunkt. Liegt ein Punkt rechts und der andere links von `x`, so muss berechnet werden, ob der Schnittpunkt rechts von dem Textvoxel liegt. Ist dies der Fall, so wechselt `inside` erneut seinen Wert.[Vor10, S. 51ff.]

Das Ergebnis des Punkt-in-Polygon-Algorithmus lässt sich in Abbildung 3.12 gut erkennen. Das erste Bild zeigt die mit Voxeln gefüllte Szene und im zweiten Bild sieht man alle Voxel, die innerhalb der Schachfigur liegen.

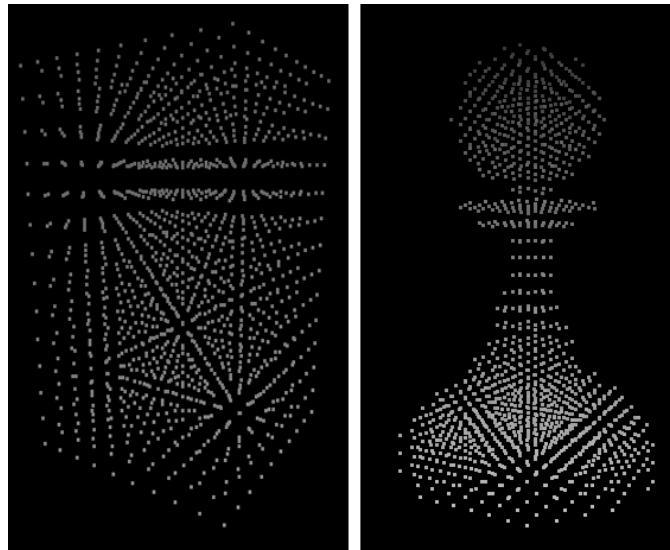


Abbildung 3.12: Voxel in Objekt - Algorithmus

3.5 Marching Cubes - Algorithmus

Der folgende Abschnitt beschreibt die Umsetzung des in Kapitel 2.4 dargestellten Marching Cubes Algorithmus in Java, basierend auf einer Implementation von Cory Gene Bloyd in C++. [Blo94]

Um später zu bestimmen an welchen Stellen der Szene Schnee fallen kann, dient dieser Algorithmus zunächst als Grundlage, um die Oberfläche der Szene bzw. der Objekte innerhalb der Szene nachzubilden.

Gegeben ist das aus Kapitel 3.3 bekannte regelmäßige, dreidimensionale Voxelgitter. Jeder Voxel enthält die Information, ob er innerhalb oder außerhalb eines Objektes liegt. Es wird nun eine neue Oberfläche erstellt, indem man kubusweise durch das Voxelgrid durchgeht, den jeweiligen Würfel auf Schnittpunkte mit der Objektoberfläche untersucht und die entsprechenden Dreiecke zeichnet. Dabei können verschiedene Fälle auftreten. Die Objektoberfläche durchtrennt den Kubus nicht, sie schneidet einen Vertex ab oder teilt ihn beliebig komplex in Innen- und Außenbereiche auf. Die Aufteilung richtet sich nach der Anzahl und dem Index der innen- bzw. außenliegenden Eckvoxeln des Kubus. Insgesamt kann es 256 verschiedene Fälle geben, welche in Abbildung 2.5 bereits illustriert wurden.

Die Bestimmung der Eckpunkte der zu zeichnenden Dreiecke richtet sich nach der Anzahl und dem Index der geschnittenen Würfelkanten. Liegt zum Beispiel ein Voxel innerhalb eines Objekts und ein benachbarter Voxel außerhalb, so muss die Objektoberfläche einen Schnittpunkt mit der Kante zwischen diesen beiden Voxeln haben. Die genaue Position des Schnittpunktes kann linear interpoliert werden, worauf in diesem Fall allerdings nicht eingegangen wird. Stattdessen wird stets der Kantenmittelpunkt als Schnittpunkt gewählt.

3.5.1 Imaginäre Cube erstellen

Der Marching Cubes Algorithmus geht jeden Voxel der Szene durch und bildet mit diesem als Referenzvoxel einen imaginären Würfel. Das bedeutet, dass die restlichen sieben Eckvoxel anhand des ersten bestimmt werden.

Die Indizierung der Voxel von v_0 bis v_7 sowie der Kanten von e_0 bis e_{11} wie sie bei der Umsetzung des Algorithmus verwendet werden, ist in Abbildung 3.13 dargestellt.

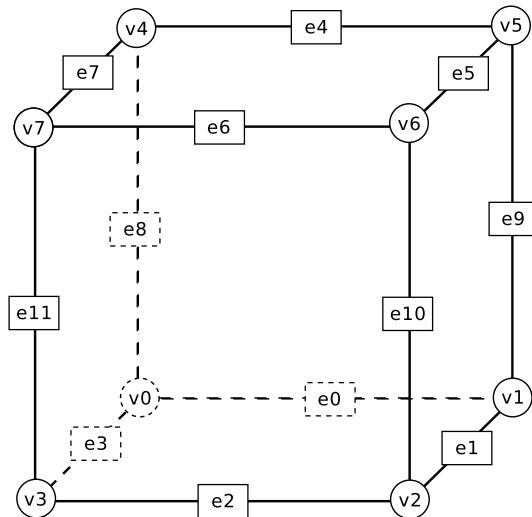


Abbildung 3.13: Indizierung eines imaginären Würfels

Der Referenzvoxel ist dabei immer die hintere linke Ecke des Kubus. Arbeitet man mit Vertices, so können die anderen Eckpunkte mit Hilfe eines Offsets bestimmt werden. In diesem Fall hingegen werden die Nachbarschaftsbeziehungen eines Voxels ausgenutzt um zu dem selben Ergebnis zu kommen.

```

1 Voxel[] cube = new Voxel[8];
2 // cube[0] ist der Referenzvoxel
3 cube[0] = v;
4 cube[1] = v.getRightNeighbor();
5 cube[2] = v.getRightNeighbor().getFrontNeighbor();
6 cube[3] = v.getFrontNeighbor();
7 cube[4] = v.getTopNeighbor();
8 cube[5] = v.getTopNeighbor().getRightNeighbor();
9 cube[6] = v.getTopNeighbor().getRightNeighbor().getFrontNeighbor();
10 cube[7] = v.getTopNeighbor().getFrontNeighbor();

```

Wie in dem Codebeispiel zu sehen ist, werden alle acht Würfecken in einem Voxel-Array gespeichert, wobei `cube[0]` der Referenzvoxel ist.

3.5.2 Schnittkanten mit der Ebene bestimmen

Um im Anschluss die Schnittkanten des Würfels mit einem Objekt zu ermitteln werden alle Eckpunkte darauf getestet ob sie innerhalb eines Objektes liegen oder Schnee beinhalten. Im zweiten Fall wird ein Voxel genauso behandelt wie im ersten, da er dann bereits in der Schneedecke liegt, das bedeutet unterhalb der Schneeoberfläche.

Ist der Test erfolgreich, so wird sich mit einem 8-Bit Index gemerkt, wobei jedes Bit einem Eckvoxel entspricht. Dies geschieht mit einer Bitshift-Operation der 1 um die entsprechende Anzahl an Stellen nach links verschiebt. Ist ein Voxel innenliegend, so wird das Bit auf 1 gesetzt, ansonsten bleibt es 0.

Dieser Teil lässt sich am Besten anhand eines Beispiels erklären. Es sei der Würfel aus Abbildung 3.14 gegeben. Nun schneidet eine Objektoberfläche den Kubus derart, dass `cube[0]` und `cube[1]` (also `v0` und `v1`) innerhalb des Objektes liegen.

Der `iFlagIndex` beträgt folglich 0000 0011 in binärer oder 3 in dezimaler Kodierung. Als nächstes wird in einer Tabelle bzw. einem Array nachgesehen, welche Würfelkanten von der Oberfläche bei gegebenem `iFlagIndex` geschnitten werden.

```

1 // find which edges are intersected by the surface
2 int iEdgeFlags = CubeEdgeFlags[iFlagIndex];

```

Dieses Array heißt `CubeEdgeFlags`, hat 256 Einträge, entsprechend aller Schnittmöglichkeiten und bildet den 8-Bit `iFlagIndex` auf eine 12-Bit kodierte Zahl mit dem Namen `iEdgeFlags` ab.

```

1 //size: int[256]
2 private int[] CubeEdgeFlags =

```

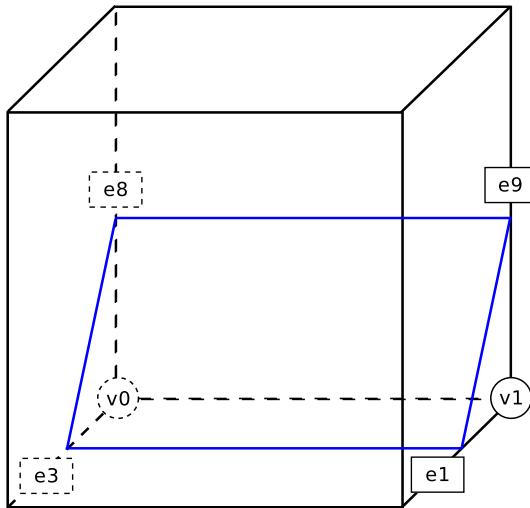


Abbildung 3.14: Beispiel Cube

```

3 {
4     0x000, 0x109, 0x203, 0x30a, 0x406, 0x50f, 0x605, 0x70c, 0x80c, ...
5 }

```

Auch hier entspricht jedes Bit einer bestimmten Kante. Ein Bit ist auf 1 gesetzt, wenn die Objektoberfläche die Kante schneidet und auf 0, falls es keinen Schnittpunkt gibt. Ist `iEdgeFlags == 0`, so wird keine der Würfelkanten geteilt und es kann mit dem nächsten Kubus fortgefahren werden.

Bezogen auf das obere Beispiel bedeutet dies, dass `iEdgeFlags[3] = 0011 0000 1010` in Binär- oder `0x30a` in Hexadezimalform beinhaltet. Das heißt, es werden die Kanten `e1`, `e3`, `e8` und `e9` von der Objektoberfläche geschnitten, wie auch in Abbildung 3.14 zu sehen ist.

Anschließend werden die Koordinaten der Schnittpunkte auf den jeweiligen Kanten berechnet und in dem zwölfelementigen `Vertex`-Array `edgeVertex` gespeichert. Dazu betrachtet man jede Kante und prüft, ob in `iEdgeFlags` an der entsprechenden Stelle eine 1 steht. Ist dies der Fall, so folgt eine Schnittpunktberechnung durch lineare Interpolation. Es wird bei den waagerechten Kanten (`e0` bis `e7`), wie in Abschnitt 3.1.2 bereits angekündigt, der Einfachheit halber von dem Mittelpunkt einer Kante als Schnittpunkt ausgegangen. Lediglich die `z`-Koordinaten der senkrechten Würfelkanten (`e8` bis `e11`) weichen von der Mitte ab und richten sich nach dem `Density`-Wert des zugehörigen, unteren Voxels.

Der folgende Codeausschnitt verdeutlicht das oben beschriebene Vorgehen.

```

1 Vertex[] edgeVertex = new Vertex[12];
2 for(int iEdge = 0; iEdge < 12; iEdge++)
3 {
4     //if there is an intersection on this edge
5     if((iEdgeFlags & (1 << iEdge)) == (1 << iEdge))
6     {

```

```

7     edgeVertex[iEdge] =
8         new Vertex((v.getX() + (VOXEL_OFFSET[EDGE_CONNECTION[iEdge][0]][0] +
9                         0.5f * EDGE_DIRECTION[iEdge][0]) * scale),
10            (v.getY() + (VOXEL_OFFSET[EDGE_CONNECTION[iEdge][0]][1] +
11                         0.5f * EDGE_DIRECTION[iEdge][1]) * scale),
12            (v.getZ() + (VOXEL_OFFSET[EDGE_CONNECTION[iEdge][0]][2] +
13                         v.getDensity() * EDGE_DIRECTION[iEdge][2]) * scale));
14    }
15 }

```

Die Koordinaten eines Schnittpunktes lassen sich mit Hilfe von drei Hilfsarrays berechnen. Zunächst wird im `EDGE_CONNECTION`-Array nachgesehen, welche beiden Voxel die aktuelle Kante verbindet. Danach wird die Entfernung des Schnittpunktes vom Referenzvoxel durch das `VOXEL_OFFSET`-Array bestimmt. Damit der Vertex an der richtigen Stelle platziert wird gibt `EDGE_DIRECTION` die Richtung der Kante an. Der Wert `0.5f` bedeutet, dass der Kantenmittelpunkt als Schnittpunkt gewählt wird. `scale` skaliert schließlich die Positionierung in Abhängigkeit zur Granularität des `Voxel`-Gitters ($\text{scale} = \frac{1}{\text{STEPS}}$).

die Einheitsnormalen an eben diesen Punkten berechnet.

3.5.3 TriangleLookupTable

Nachdem alle Schnittpunkte eines Kubus mit der Objektoberfläche berechnet wurden müssen diese noch in korrekter Reihenfolge zu Dreiecken bzw. `Faces` verbunden werden. Hierzu wird erneut ein Array verwendet, das den selben `iFlagIndex` verwendet wie zuvor `CubeEdgeFlags` und es erlaubt die `Vertex`-Sequenz zur Darstellung der angenährten Oberfläche nachzuschlagen. Das `TriangleLookupTable`-Array ist zweidimensional und besteht aus 256×16 Einträgen, da es 256 Teilungsmöglichkeiten und pro Cube bis zu fünf `Faces` gibt. Die `TriangleLookupTable` hat die folgende Form.

```

1 // size: int[256][16]
2 private int[][] TriangleLookupTable =
3 {
4     {-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
5     {0, 8, 3, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
6     {0, 1, 9, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
7     {1, 8, 3, 9, 8, 1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
8     {1, 2, 10, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
9     ...
10 };

```

Dabei steht die `-1` für einen invaliden Eintrag und zeigt an, dass der Algorithmus für diesen Cube an dieser Stelle abgebrochen werden kann (siehe Zeile 4-5 im unteren Code-Beispiel).

```

1 // Store the triangles that were found. There can be up to five per cube.
2 for(int iTriangle = 0; iTriangle < 5; iTriangle++)
3 {
4     if(TriangleLookupTable[iFlagIndex][3 * iTriangle] < 0)
5         break;
6     for(int iCorner = 0; iCorner < 3; iCorner++)
7     {
8         int iVertex =
9             TriangleLookupTable[iFlagIndex][3 * iTriangle + iCorner];
10        this.faces[cnt++] = edgeVertex[iVertex].getX();
11        this.faces[cnt++] = edgeVertex[iVertex].getY();
12        this.faces[cnt++] = edgeVertex[iVertex].getZ();
13    }
14 }

```

Folglich werden die bis zu fünf Möglichkeiten durchgegangen. Soll ein Face später gezeichnet werden, so wird auf die Vertices des `edgeVertex`-Arrays in richtiger Reihenfolge zugegriffen und diese entgegen dem Uhrzeigersinn in einer Instanzvariable `faces` abgespeichert.

Erneut bezogen auf den obigen Beispieldfall liefert der Aufruf `TriangleLookupTable[3]` die Zeile:

```
{1, 8, 3, 9, 8, 1, -1, -1, -1, -1, -1, -1, -1, -1}
```

Das bedeutet, dass für diesen Cube zwei Faces gezeichnet werden müssen. Zum einen bestehend aus den Vertices mit den Indices (1, 8, 3) und zum anderen aus (9, 8, 1).

3.5.4 Ergebnis

Nachfolgend werden einige Ergebnisse aufgeführt, die mit Hilfe des oben beschriebenen Algorithmus entstanden sind. Die Abbildung 3.15 zeigt im linken Teil die Annäherung an eine Kugel

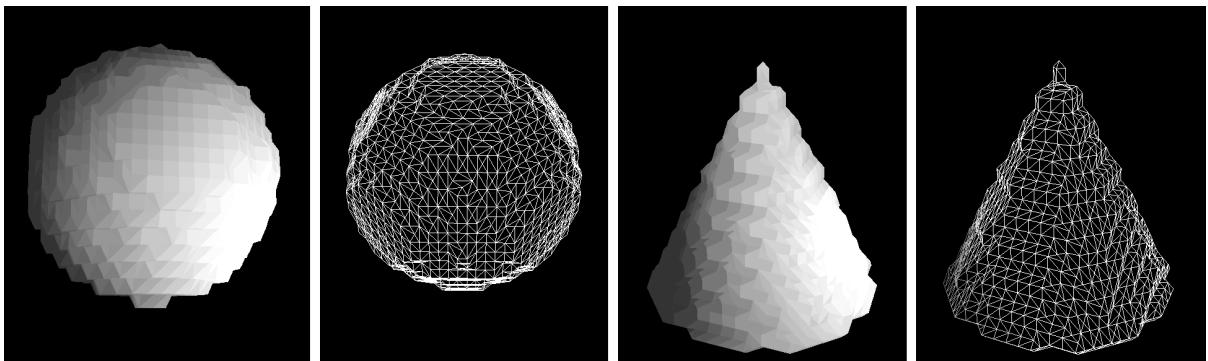


Abbildung 3.15: Marching Cubes Ergebnisse

und im rechten die wiederhergestellte Oberfläche eines Kegels. Damit die erzeugten Dreiecke

besser erkennbar sind, ist neben dem jeweiligen Objekt mit gefüllten **Faces** zusätzlich das entsprechende Drahtgittermodell dargestellt.

Abhängig von der Voxeldichte kann nun der Detailgrad der erzeugten Oberfläche variiert werden. Je mehr Voxel vorhanden sind, desto genauer ist die Annäherung an die tatsächliche Objektoberfläche. Dies illustriert die Abbildung 3.16 recht deutlich. Allen drei gezeichneten Figuren liegt

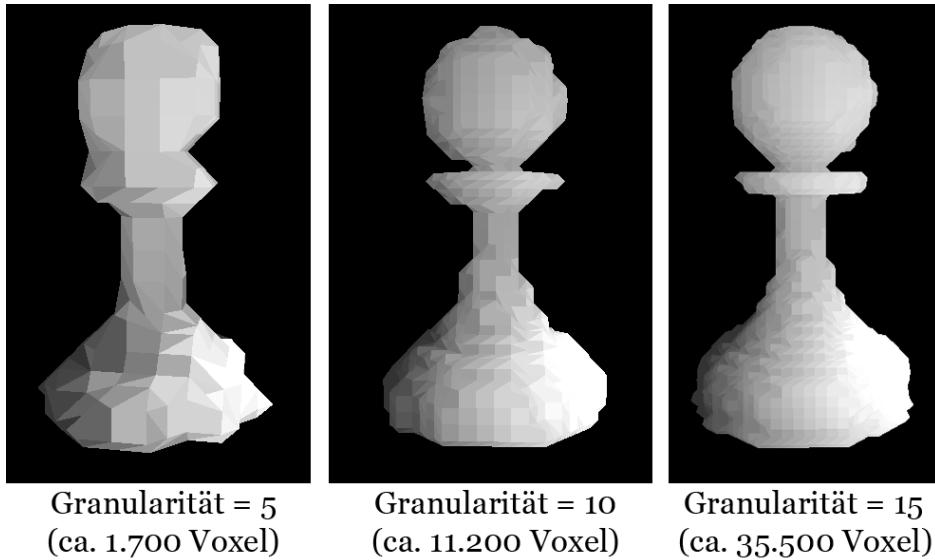


Abbildung 3.16: Unterschiedlicher Detailgrad des Marching Cubes Algorithmus

das selbe Objekt zugrunde. Das erste Bild hat offensichtlich eine sehr geringe Voxeldichte oder auch Granularität von 5. Das heißt pro Einheit auf der jeweiligen Achse werden 5 Voxel gesetzt. Ist die Szene zum Beispiel 2 Einheiten (in x-Richtung) lang, so werden 10 Voxel entlang dieser Achse gesetzt. Das zweite Bild lässt die Schachfigur deutlich besser erkennen, da dort 10 Voxel pro AchsenEinheit gesetzt werden. Im dritten Bild liegt schließlich eine Granularität von 15 vor. Nachdem jetzt die Oberfläche der Szene durch den Marching Cubes Algorithmus angenähert werden kann, ist es im Folgenden möglich Schnee auf Objekten fallen zu lassen und eine Schneedecke zu erzeugen.

3.6 Schneedecke

Die zuvor durch den Marching Cubes Algorithmus erzeugte Oberfläche dient im Nachfolgenden als Grundlage der Schneedecke. Damit diese allerdings wachsen und auf den bereits vorhandenen Oberflächen aufbauen kann, muss zunächst Schneefall simuliert werden. Das Anwachsen der Schneedecke wiederum wird mit Hilfe der Voxel realisiert, mit denen die gesamte Szene gefüllt wurde. Dieses Vorgehen wird im weiteren Verlauf noch genauer erläutert.

Der in diesem Kapitel simulierte Schneefall hat zwei mögliche Richtungen: senkrecht von oben oder schräg von der Seite. Als Abrundung des Ergebnisses wird die Schneestabilität eingeführt. Dabei wird geprüft, ob kleinere Lawinen ausgelöst werden müssen (Verteilung des Schnees auf

benachbarte Voxel), um unrealistisch hohe Schneeaufstürmungen an Objektkanten zu vermeiden. Die folgenden zwei Applikationsbilder (3.17 und 3.18) zeigen den initialen Zustand nach einmaliger Anwendung des Marching Cubes Algorithmus. Die Abbildung 3.17 verdeutlicht das in

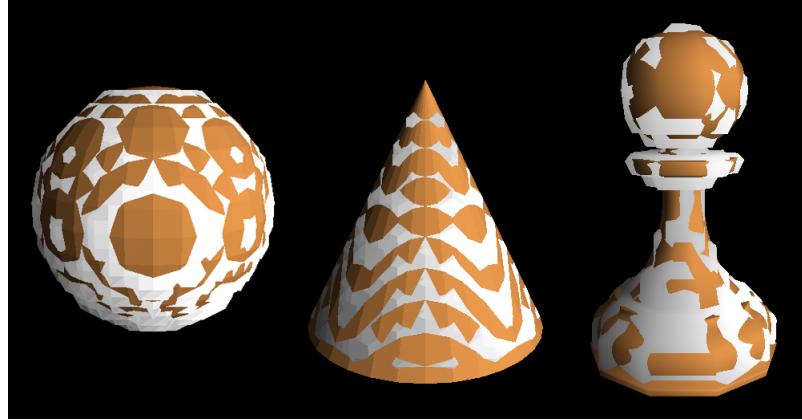


Abbildung 3.17: Unterschied zwischen tatsächlicher und angenäherter Oberfläche

Abschnitt 3.1.2 angesprochene Problem der unpräzisen Annäherung an die tatsächliche Objektoberfläche. Die nachgebildete Oberfläche, die als Grundlage der Schneedecke dienen wird, liegt zu Beginn teilweise über und teilweise unter der Oberfläche des Objekts. Ein besseres Ergebnis kann die vorliegende Implementation des Algorithmus nicht leisten. Die Gründe hierfür sind zum einen die Unausgewogenheit zwischen dem Aufwand der komplexen Berechnung der Schnittpunkte und dem im Gegensatz dazu nur leicht verbesserten Endergebnis und zum anderen die Tatsache, dass die Schneedecke bereits nach wenigen Iterationen auf und über der Objektoberfläche liegt, wodurch die zu Anfang durchscheinende Schneedecke keine Relevanz mehr hat. Im Gegensatz dazu ist die Annäherung an rechteckige Objekte, wie in Abbildung 3.18 zu sehen ist, sehr zufriedenstellend.



Abbildung 3.18: Annäherung eines rechteckigen Objekts mit Schneedecke

3.6.1 Schnee von oben

Das Anwachsen der Schneedecke soll zunächst an einem Beispiel verdeutlicht werden. Dabei wird davon ausgegangen, dass die imaginären Schneeflocken senkrecht von oben fallen. Die Abbildung

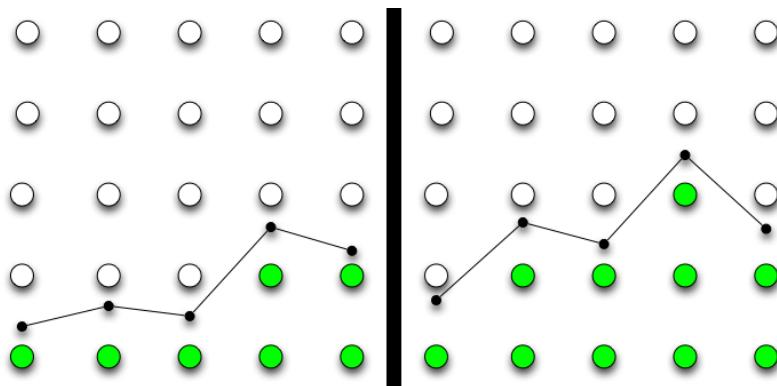


Abbildung 3.19: Anwachsen der Schneedecke

3.19 stellt das das Voxelgitter schematisch und zweidimensional dar. Es werden zusätzlich die in Abschnitt 3.1.2 beschriebenen Eigenschaften der Voxel ausgenutzt. Die grün gefüllten Kreise repräsentieren schneegefüllte Voxel, die kleinen schwarzen Kreise stellen den jeweiligen `density`- oder Füllewert des darunterliegenden Voxel dar und die schwarze Linie illustriert die Schneedecke.

Der `density`-Wert eines Voxel kann zwischen 0 und 1 liegen. Er wird erhöht, wenn eine imaginäre Schneeflocke auf den jeweiligen Voxel trifft. Überschreitet der Füllwert dabei den Wert 1, so wird mit Hilfe der Nachbarschaftsbeziehung der darüberliegende Voxel aktiviert, das heißt sein `snow`-Attribut wird auf `true` gesetzt. Um schließlich sprunghafte Übergänge von einem Voxel zum nächsten bei der Visualisierung des Schneewachstums zu verhindern, wird die Höhe der Schneedecke anhand des `density`-Wertes des entsprechenden Voxels interpoliert.

Im weiteren Verlauf wird nun genauer auf die Umsetzung des senkrechten Schneefalls auf Programmebene eingegangen. Zur Simulation von Schneefall oder Schneeflocken müssen als erstes die Voxel der Szene bestimmt werden auf die Schnee fallen kann. Dazu wird das gesamte `Voxel`-Array durchgegangen um anschließend die in Frage kommenden Kandidaten in dem Array `activeVoxels` zu speichern.

```

1 int k = 0;
2 this.activeVoxels = new Voxel[countActiveVoxels(voxels)];
3 // snow from above
4 for(int i = 0; i < voxels.length; i++)
5 {
6     if((voxels[i].getSnow() && // voxels on the surface
7         !voxels[i].getTopNeighbor().getSnow()) ||
8         (!voxels[i].hasBottomNeighbor() && // floor-voxels beside objects
9         !voxels[i].getTopNeighbor().getSnow() && !voxels[i].getSnow()) ||

```

```

10     (voxels[i].isInside() &&           // voxels on an object
11     !voxels[i].getTopNeighbor().isInside()))
12 {
13     activeVoxels[k++] = voxels[i];
14 }
15 }
```

Wird angenommen, dass der Schnee direkt von oben fällt, so legt das längliche `if`-Statement (Zeile 6 - 11) fest, welche Voxel als mögliche Landeplätze bereitgestellt werden. Dies betrifft zum einen alle auf der Schneeeoberfläche liegenden Voxel, das bedeutet, Voxel die selbst Schnee enthalten, ihr oberer Nachbar aber nicht. Zum anderen soll auch neben den Objekten, auf dem Szenenboden Schnee fallen können, wozu die Voxel ohne unteren Nachbarn aktiviert werden. Zuletzt werden noch die Voxel in Betracht gezogen, die auf einem Objekt liegen, das heißt der Voxel selbst liegt innerhalb, sein oberer Nachbar hingegen außerhalb eines Objekts. Sind die Aktiven Voxel bestimmt worden, so kann mit der Simulation des Schneefalls begonnen werden. Dazu bestimmt man zunächst die Anzahl der Schneeflocken, die fallen sollen, bevor die Szene neu gezeichnet wird (in diesem Fall `int snowflakes = 50`).

```

1 int snowflakes = 50;
2 for(int j = 0; j < snowflakes; j++)
3 {
4     int index = (int)(Math.random() * activeVoxels.length);
5     this.activeVoxels[index].raiseDensity(SNOWFLAKE);
6     if(this.activeVoxels[index].getDensity() > 1.0f)
7     {
8         this.activeVoxels[index].getTopNeighbor().setSnow();
9         this.activeVoxels[index] = this.activeVoxels[index].getTopNeighbor();
10    }
11 }
```

Anschließend wird für jede Schneeflocke ein beliebiger Voxel aus dem `activeVoxels`-Array herausgegriffen und dessen `density`-Wert um eine konstante Größe erhöht (z.B. 0.1). Übersteigt die `density` den Wert 1.0, so wird der obere Nachbar des aktuellen Voxels aktiviert und an die Stelle des alten in das `activeVoxels`-Array geschrieben.

Nach einer gewissen Anzahl von Iterationen führt dieses Vorgehen zu folgendem Ergebnis (Abbildung 3.20). Der Pfeil zeigt die Richtung des Schneefalls an. Natürlich ist dies nur ein grobes Modell zur Validierung und Visualisierung eines Ergebnisses. Bei dem oben beschriebenen Vorgehen kann es zum Beispiel vorkommen, dass auch Voxel unterhalb eines Objekts auf den Boden empfänglich für Schnee sind, obwohl diese in der Realität nicht erreichbar wären. Um ein authentischeres Ergebnis zu erhalten müsste ein realitätsnahes Partikelsystem erstellt werden anhand dessen sich Schneeflocken in der Szene bewegen. In diesem Fall müsste der `density`-Wert eines aktiven Voxels nur erhöht werden, wenn eine Schneeflocke in das Einzugsgebiet dieses Voxels fällt und nicht nach dem oben verwendeten Zufallsprinzip.



Abbildung 3.20: Schneedecke bei senkrechtem Schneefall

3.6.2 Schnee von der Seite

Um nun etwas Wind zu simulieren wird zusätzlich zu dem senkrechten noch waagerechter Schneefall hinzugefügt. Die Funktionsweise ist analog zum Schnee von oben, nur das etwas andere Voxel aktiviert werden. Beispielhaft soll der Schnee von rechts auf das Objekt auftreffen.

```

1 if((voxels[i].getSnow() || voxels[i].isInside()) &&
2     voxels[i].hasRightNeighbor() && !voxels[i].getRightNeighbor().getSnow())
3 {
4     activeRightVoxels[k++] = voxels[i];
5 }
```

Dazu wird lediglich geprüft, ob sich ein Voxel auf der Objektoberfläche oder der Schneedecke befindet und einen nicht gefüllten, rechten Nachbarn besitzt. Ist dies der Fall, so kann die Schneedecke nun rechts an das Objekt wachsen. In Kombination mit den senkrecht fallenden Schneeflocken kann somit ein Schneefall von rechts oben nachgestellt werden. Damit das in Abbildung 3.21 dargestellte Ergebnis realistisch wirkt, ist darauf zu achten, dass die Anzahl der Schneeflocken, die von oben fallen größer ist, als die Zahl der von rechts kommenden. Zudem zeigen die ersten beiden Bilder zwei Zustände der Szene mit steigender Schneedecke. Im dritten Bild verdeutlicht das Drahtgittermodell gut die deutlich noch oben und nach rechts gewachsene Schneedecke, da sich die Schachfigur darunter erkennen lässt.

3.6.3 Schneestabilität

Die Idee des Stabilitätstests der Schneedecke stammt aus dem Paper "Computer Modelling Of Fallen Snow" von Paul Fearing in dem er sich ebenfalls mit der Akkumulation von Schnee, sowie der Bildung von Schneedecken beschäftigt, dabei allerdings einen anderen Ansatz verfolgt. [Fea00]



Abbildung 3.21: Schneedecke bei Schneefall von rechts oben

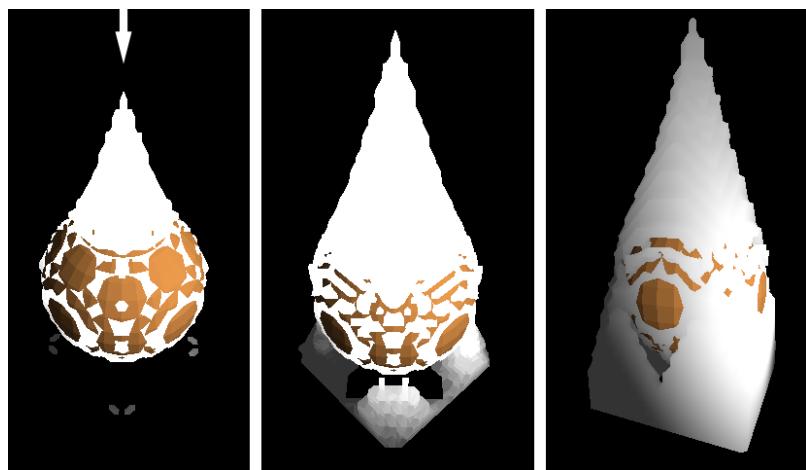


Abbildung 3.22: Schneeverteilung bei einer Punktquelle mit Stabilitätstest



Abbildung 3.23: Beispiel der Schneeverteilung bei alternativer Punktquelle

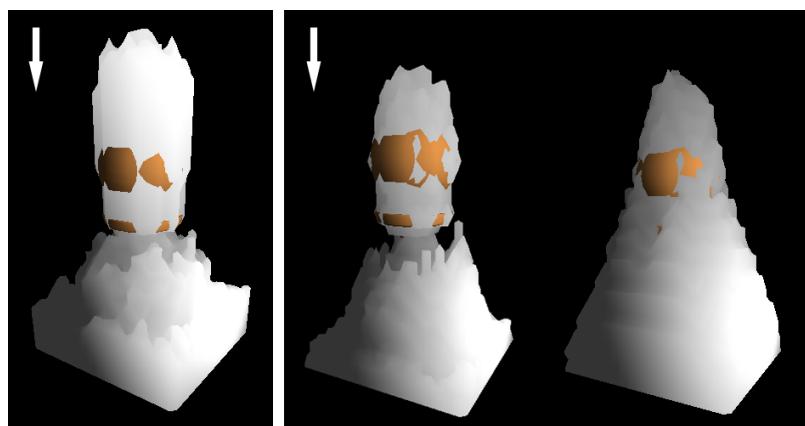


Abbildung 3.24: Senkrechter Schneefall ohne und mit Stabilitätstest

Kapitel 4

Reflexion

4.1 Zusammenfassung

4.2 Fazit und Ausblick

Literaturverzeichnis

- [Bli78] BLINN, James F.: Simulation of wrinkled surfaces. In: *SIGGRAPH Comput. Graph.* 12 (1978), August, Nr. 3, 286–292. <http://dx.doi.org/10.1145/965139.507101>. – DOI 10.1145/965139.507101. – ISSN 0097–8930
- [Blo94] BLOYD, Cory G.: *Marching Cubes Algorithmus in C++*. <http://local.wasp.uwa.edu.au/~pbourke/geometry/polygonise/marchingsource.cpp>, 1994
- [Coo84] COOK, Robert L.: Shade trees. In: *SIGGRAPH Comput. Graph.* 18 (1984), Januar, Nr. 3, 223–231. <http://dx.doi.org/10.1145/964965.808602>. – DOI 10.1145/964965.808602. – ISSN 0097–8930
- [EMP⁺03] EBERT, David S. ; MUSGRAVE, F. K. ; PEACHY, Darwyn ; PERLIN, Ken ; WORLEY, Steven: *Texturing and Modeling. A Procedural Approach*. 3. San Francisco, CA : Morgan Kaufmann Publishers, 2003
- [Fea00] FEARING, Paul: Computer modelling of fallen snow. In: *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*. New York, NY, USA : ACM Press/Addison-Wesley Publishing Co., 2000 (SIGGRAPH '00). – ISBN 1–58113–208–5, 37–46
- [LC87] LORENSEN, William E. ; CLINE, Harvey E.: Marching cubes: A high resolution 3D surface construction algorithm. In: *SIGGRAPH Comput. Graph.* 21 (1987), August, Nr. 4, 163–169. <http://dx.doi.org/10.1145/37402.37422>. – DOI 10.1145/37402.37422. – ISSN 0097–8930
- [SSS74] SUTHERLAND, Ivan E. ; SPROULL, Robert F. ; SCHUMACKER, Robert A.: A Characterization of Ten Hidden-Surface Algorithms. In: *ACM Comput. Surv.* 6 (1974), März, Nr. 1, 1–55. <http://dx.doi.org/10.1145/356625.356626>. – DOI 10.1145/356625.356626. – ISSN 0360–0300
- [Vor10] VORNBERGER, Oliver: *Computergrafik*. Osnabrück : Selbstverlag der Universität Osnabrück, 2010
- [Wen05] WENKE, Henning: *3D Klimadatavisualisierung mit OpenGL*, Universität Osnabrück, Bachelorarbeit, 2005

- [Wik05] WIKIPEDIA: *Displacement Mapping*. <http://upload.wikimedia.org/wikipedia/commons/a/a4/Displacement.jpg>, 2005
- [Wik12a] WIKIPEDIA: *Marching Cubes*. <http://upload.wikimedia.org/wikipedia/commons/thumb/a/a7/MarchingCubes.svg/500px-MarchingCubes.svg.png>, 2012
- [Wik12b] WIKIPEDIA: *Marching Cubes - Polygonmodell eines Kopfes*. <http://upload.wikimedia.org/wikipedia/commons/6/63/Marchingcubes-head.png>, 2012

Erklärung

Ich versichere, dass ich die eingereichte Bachelor-Arbeit selbstständig und ohne unerlaubte Hilfe verfasst habe. Anderer als der vom mir angegebenen Hilfsmittel und Schriften habe ich mich nicht bedient. Alle wörtlich oder sinngemäß den Schriften anderer Autoren entnommenen Stellen habe ich kenntlich gemacht.

Osnabrück, März 2012