

Universität Passau
Fakultät für Informatik und Mathematik



Bachelorarbeit

**Oberflächenextraktion mittels des Marching
Cubes Algorithmus**

Verfasser:

Georg Seibt

4. Oktober 2014

Betreuer:

Prof. Dr. Tomas Sauer

Institut für Softwaresysteme in technischen Anwendungen der Informatik

Seibt, Georg:

Oberflächenextraktion mittels des Marching Cubes Algorithmus
Bachelorarbeit, Universität Passau, 2014.

Inhaltsverzeichnis

1	Motivation	1
2	Der Marching Cubes Algorithmus	2
2.1	Blackbox - Input und Output	2
2.1.1	Input - Voxelgitter	3
2.1.2	Input - Der Schwellwert	3
2.1.3	Output - Vertex- und Normalenliste	4
2.2	Der Würfel	4
2.2.1	Konfigurationen und der Würfelindex	5
2.2.2	Darstellung der Konfigurationen	5
2.3	Verarbeitung der Würfel	8
2.3.1	Lookup - Kanten und Dreiecke	9
2.3.2	Normalen an den Würfecken	10
2.3.3	Dreiecksvertices - Interpolation der Position und Normalen	10
3	Implementierung - V8	12
3.1	Marching Cubes Klassenübersicht	12
3.1.1	MCRunner	13
3.1.2	MCVolume	15
3.1.3	Cube	17
3.2	GUI Übersicht	18
3.3	Auswahl der Datenquellen	19
3.3.1	DICOM Bilder	19
3.3.2	Zufällige Volumen - Metabälle	20
3.4	Konfiguration des Algorithmus	21
3.4.1	Level	22
3.4.2	Grid Size	24
3.4.3	Ausgabe von Zwischenergebnissen	25
3.4.4	Export nach OBJ und STL	25

3.5	3D Ansicht	26
3.5.1	Visualisierungsmöglichkeiten	27
3.5.2	Navigation	28
4	Anwendungen	30
4.1	Profiling der Anwendung	30
4.2	Verschieden dichte Objektanteile	32
4.3	Einfluss der Grid Size auf den Mesh	33
5	Mögliche Erweiterungen	36
6	Zusammenfassung	38
	Glossar	

Abbildungsverzeichnis

2.1	Ein Voxelgitter	3
2.2	Der Würfel	4
2.3	Eine Beispielkonfiguration	5
2.4	Zwei komplementäre Würfel	6
2.5	Der Würfel mit Index 252 (links) und zwei Rotationen	6
2.6	Die 14 Klassen	7
2.7	Diagonale Seiten eines Würfels	7
2.8	Eine inkorrekte Darstellung, unten die korrigierte Version	8
3.1	Klassendiagramm von <code>MCRRunner</code> und zugehöriger Klassen	13
3.2	Mesh Speicherformat	14
3.3	Speicherung zweier Dreiecke	15
3.4	Klassendiagramm des <code>MCVolume</code> und Implementierungen	16
3.5	Klassendiagramm von Würfel und Vertex Klassen	17
3.6	Das Hauptfenster von <code>V8</code>	18
3.7	Dialog zum Erstellen eines Metaball-Volumens	20
3.8	Ein Volumen aus drei Metabällen	21
3.9	Die Konfigurationsoptionen von <code>V8</code>	22
3.10	Die Vorschauansicht für zwei Schwellwerte	23
3.11	Die Vorschauansicht für zwei Schwellwerte	24
3.12	Die Vorschauansicht für Level 150 und drei Grid Size Werte	24
3.13	Klassendiagramm von <code>MeshView3D</code>	27
4.1	Laufzeit in Abhängigkeit von Grid Size mit <code>ArrayVolume</code>	31
4.2	Laufzeit in Abhängigkeit von Grid Size mit <code>CachedVolume</code>	31
4.3	Verschiedene extrahierte Oberflächen	32
4.4	Artefakte bei Level 100	33
4.5	Anzahl der Dreiecke in Abhängigkeit von Grid Size	34
4.6	Verschiedene extrahierte Oberflächen	35

Kapitel 1

Motivation

Mit der Verbreitung von digitalen, bildgebenden Geräten in der Medizin ist die Verarbeitung der Volumendaten, die diese Geräte produzieren, zu einem wichtigen Feld in der Computergrafik geworden. Zur Anzeige dieser Daten existieren Techniken wie ‚Volume Raycasting‘ oder ‚Splatting‘. Mithilfe dieser Verfahren können zwar die Volumendaten unverfälscht angezeigt werden, sie gelten aber als sehr zeitintensiv, und es existiert keine direkte Hardwarebeschleunigung. Aufgaben wie Schattenwurf, für die in der Oberflächengrafik schnelle Algorithmen existieren, sind für Volumengrafiken hochgradig nicht-trivial. Soll die Voxelgrafik in anderen Bereichen, wie z.B. in Videospielen, eingesetzt werden, so zeigen sich andere Nachteile. Es ist beispielsweise kaum möglich, Animationen in einer Voxel Szene zu realisieren.

Zur Umwandlung von Volumen- in Oberflächengrafiken existieren deshalb effiziente Algorithmen, die eine Oberfläche aus dem Volumen extrahieren und durch Dreiecke annähern. Diese können mit den existierenden Programmen, wie sie im Abschnitt 3.4.4 genannt sind, weiterverarbeitet werden. Moderne Grafikkarten können solche Oberflächen schließlich hocheffizient anzeigen. Ein grundlegender Algorithmus für diese Umwandlung ist der Marching Cubes Algorithmus. In dieser Bachelorarbeit sollen folgende Ziele erreicht werden:

1. Darlegung der Funktionsweise des Marching Cubes Algorithmus
2. Wiederverwendbare Implementierung des Algorithmus
3. Bereitstellung einer GUI, die es ermöglicht, den Algorithmus auf DICOM Volumendaten anzuwenden
4. Hardwarebeschleunigte Darstellung der Oberfläche

Kapitel 2

Der Marching Cubes Algorithmus

In [LC87] wurde der Marching Cubes Algorithmus zur Extraktion von Isoflächen aus Volumendaten vorgestellt. Formal ist diese Extraktion wie folgt definiert:

Aus einer Funktion $\varphi : \mathbb{R}^n \rightarrow \mathbb{R}$ wird, gegeben ein Schwellwert $c \in \mathbb{R}$, eine Isofläche S_c extrahiert, für die gilt:

$$S_c := \{v \in \mathbb{R}^n \mid \varphi(v) = c\}$$

In Implementierungen ist $n = 3$. φ wird auf einem 3D Gitter gesampled und beispielsweise durch ein `float[][][]` repräsentiert. Die resultierende Isofläche S_c wird durch einen Polygon Mesh angenähert.

Marching Cubes stellt die Grundlage für viele moderne Algorithmen dar, wie zum Beispiel für den Dual Marching Cubes Algorithmus, der in [SW04] vorgestellt wurde. Im Folgenden wird der Aufbau und Ablauf des Algorithmus beschrieben.

2.1 Blackbox - Input und Output

Zuerst wird der Input und Output des Algorithmus betrachtet, um einen Überblick zu gewinnen.

2.1.1 Input - Voxelgitter

In diesem 3D Gitter aus Voxeln sind die Werte der Voxel Funktionswerte von φ . Um das Gitter zu erstellen, wird φ in konstanten Abständen in allen drei Dimensionen ausgewertet.

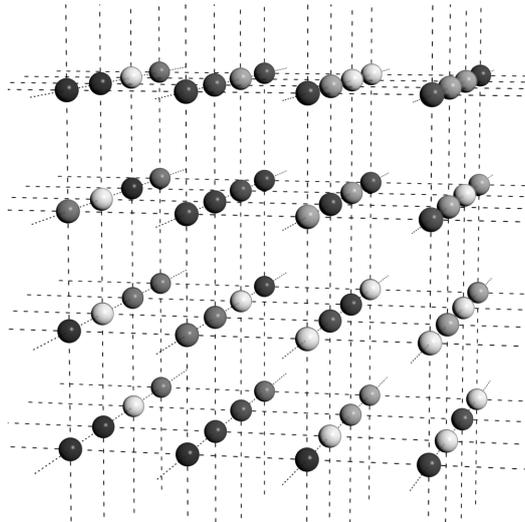


Abbildung 2.1: Ein Voxelgitter

Eine solche Auswertung findet beispielsweise statt, wenn ein Objekt von einem Computertomographen aufgenommen wird. Alle Voxel der gleichen z-Ebene bilden eine Scheibe (eng. ‚Slice‘). Einer der namensgebenden Würfel besteht aus acht Voxeln, die zu zwei übereinanderliegenden Scheiben gehören. Zusätzlich wird für die spätere Berechnung der Normalen eine Scheibe über und eine unter den zwei Scheiben benötigt, die die Voxel des Würfels enthalten. Eine Implementierung des Algorithmus muss somit vier z-Ebenen des Datensatzes vorhalten, während die Würfel in einer Scheibe bearbeitet werden.

2.1.2 Input - Der Schwellwert

Der Schwellwert c definiert, welche Voxel vom Algorithmus als transparent und welche als solide betrachtet werden. Die extrahierte Isofläche S_c ist die Grenze zwischen den soliden Voxeln mit $\varphi(x, y, z) > c$ und den transparenten, für welche $\varphi(x, y, z) \leq c$ gilt. Durch geschickte Anpassung des Schwellwertes können verschiedene Oberflächen aus einem Datensatz extrahiert werden. So

lässt sich beispielsweise entweder die Hautoberfläche oder die Oberfläche der Knochen aus ein und demselben CT Scann darstellen.

2.1.3 Output - Vertex- und Normalenliste

Der Algorithmus produziert einen Polygon Mesh, welcher die Isofläche S_c annähert. Speziell wird ein Mesh aus Dreiecken zurückgegeben. Für jeden Dreiecksvertex wird eine zugehörige Normale berechnet, welche die Orientierung der Oberfläche angibt.

2.2 Der Würfel

Die zentrale Struktur im Marching Cubes Algorithmus ist der Würfel. Als Würfel wird eine Menge von acht Voxeln (den Ecken des Würfels) und zwölf Kanten bezeichnet. Jeweils vier der Voxel liegen in einer Scheibe des Datensatzes.

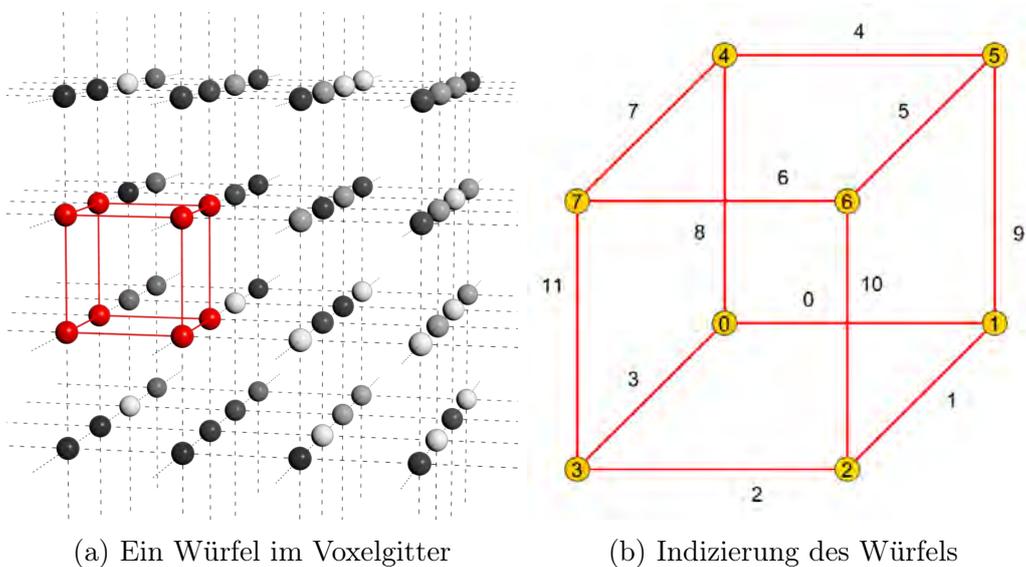


Abbildung 2.2: Der Würfel

Die Abbildung 2.2a zeigt rot markiert einen Würfel der Kantenlänge 1. Die Aufgabe des Marching Cubes Algorithmus ist nun, für jeden Würfel zu entscheiden, ob und wie S_c diesen schneidet. Die Ecken und Kanten des Würfels werden indiziert, um sie im Algorithmus identifizieren zu können.

2.2.1 Konfigurationen und der Würfelindex

In einem Würfel kann jede Ecke entweder als transparent oder solide klassifiziert werden. Da also jede der acht Ecken einen von zwei Zuständen haben kann, ergeben sich $2^8 = 256$ verschiedene Konfigurationen, in denen sich der Würfel befinden kann. Aus jeder Konfiguration kann ein 8 Bit Bitmuster erzeugt werden, in welchem das Bit i genau dann 1 ist, wenn für den Wert des Voxels mit dem Index i (bezeichnet durch d_i) gilt: $d_i \leq c$. Interpretiert man dieses Bitmuster als natürliche Zahl, erhält man einen Wert zwischen 0 und 255, genannt Würfelindex.

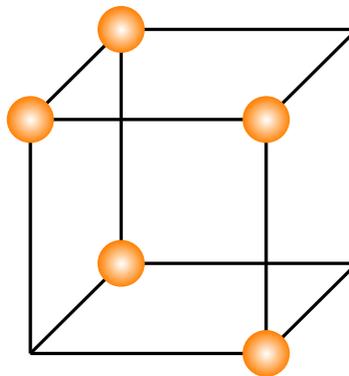
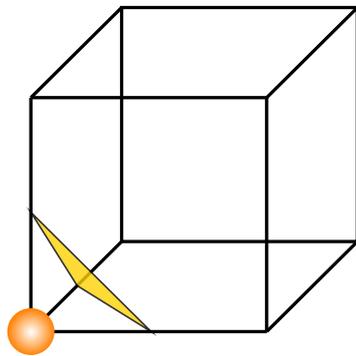


Abbildung 2.3: Eine Beispielkonfiguration

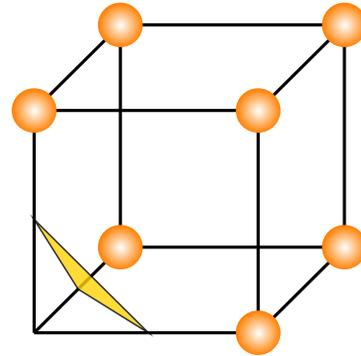
Die Abbildung 2.3 zeigt eine Konfiguration, in der die soliden Ecken orange markiert sind. Berechnet man den Würfelindex der gegebenen Konfiguration, ergibt sich $00101010_2 = 42_{10}$.

2.2.2 Darstellung der Konfigurationen

Zu jeder der 256 Konfigurationen aus 2.2.1 muss eine Darstellung von S_c durch Dreiecke gefunden werden. In [LC87, S. 165] beschreiben die Autoren diesen Vorgang als „... possible but tedious and error-prone.“. Um die 256 Konfigurationen zu reduzieren, wurden Symmetrien des Würfels eingesetzt, um Klassen von Würfeln zu identifizieren, für deren Mitglieder die gleiche Repräsentation genutzt werden kann. Die erste Symmetrie, die ausgenutzt wurde, ist die zwischen komplementären Würfeln. Werden in einem Würfel die Zustände jeder Ecke negiert (solide \leftrightarrow transparent) so bleibt die Repräsentation gleich.



Würfel mit Index 247



Der komplementäre Würfel

Abbildung 2.4: Zwei komplementäre Würfel

Des weiteren wird ausgenutzt, dass die Repräsentation gleich bleibt, wenn der Würfel rotiert wird.

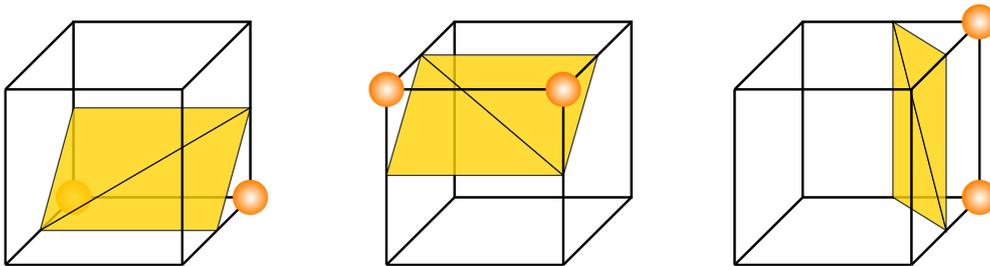


Abbildung 2.5: Der Würfel mit Index 252 (links) und zwei Rotationen

Durch Anwendung dieser Symmetrien ergeben sich 14 Klassen und eine ‚unechte Klasse‘. Die Würfel, in denen entweder alle Ecken solide oder alle transparent sind, produzieren keine Dreiecke, da angenommen wird, dass S_c den Würfel nicht schneidet.

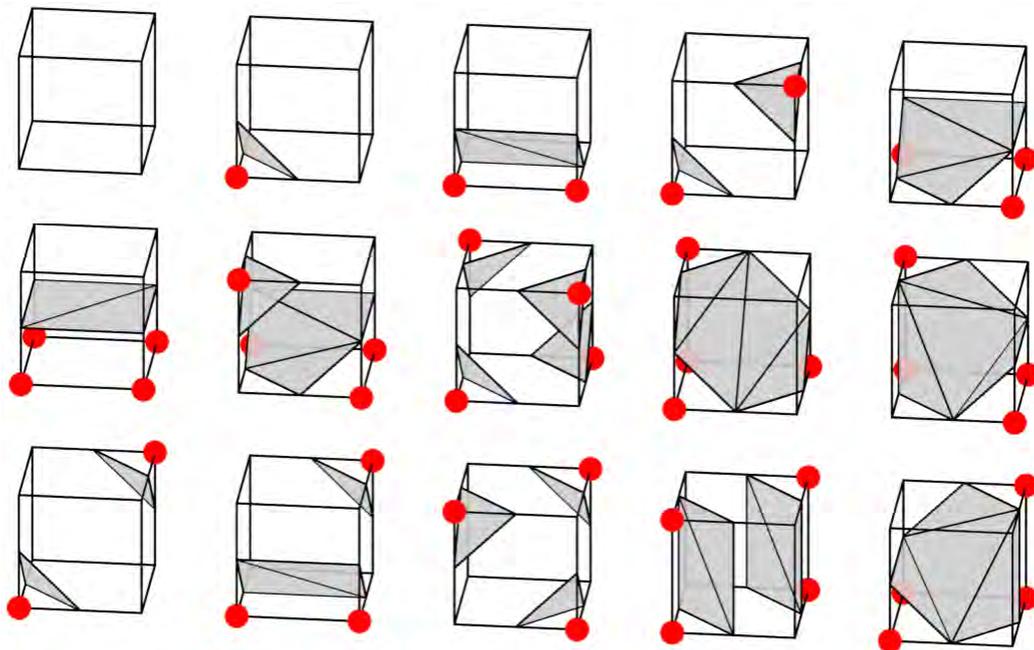


Abbildung 2.6: Die 14 Klassen

Allerdings gibt es durch diese Einteilung in Klassen uneindeutige Fälle. Jeder Würfel, der mindestens eine ‚diagonale‘ Seite hat (siehe 2.7), kann zu Brüchen in der Oberfläche führen. Für solche Fälle ist die Annahme nicht zutreffend, dass für komplementäre Würfel die gleiche Repräsentation durch Dreiecke verwendet werden darf. Diese Fälle erfordern eine separate Behandlung.

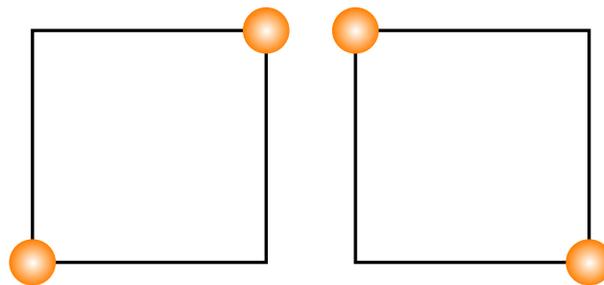


Abbildung 2.7: Diagonale Seiten eines Würfels

In Abb. 2.8 sind (in der rechten Spalte) zwei mögliche Darstellungen für die Konfiguration mit Index 129 gegeben. Die obere wurde gewählt da diese Konfiguration durch Rotation und Negation der mit Index 183 erreicht werden kann. Würden die Würfel der oberen Zeile nebeneinanderliegen ergäbe sich so aber ein Bruch in der Oberfläche. Die Konfiguration 129 muss demnach

anders dargestellt werden. Die korrigierte Version ist in der unteren Zeile zu sehen.

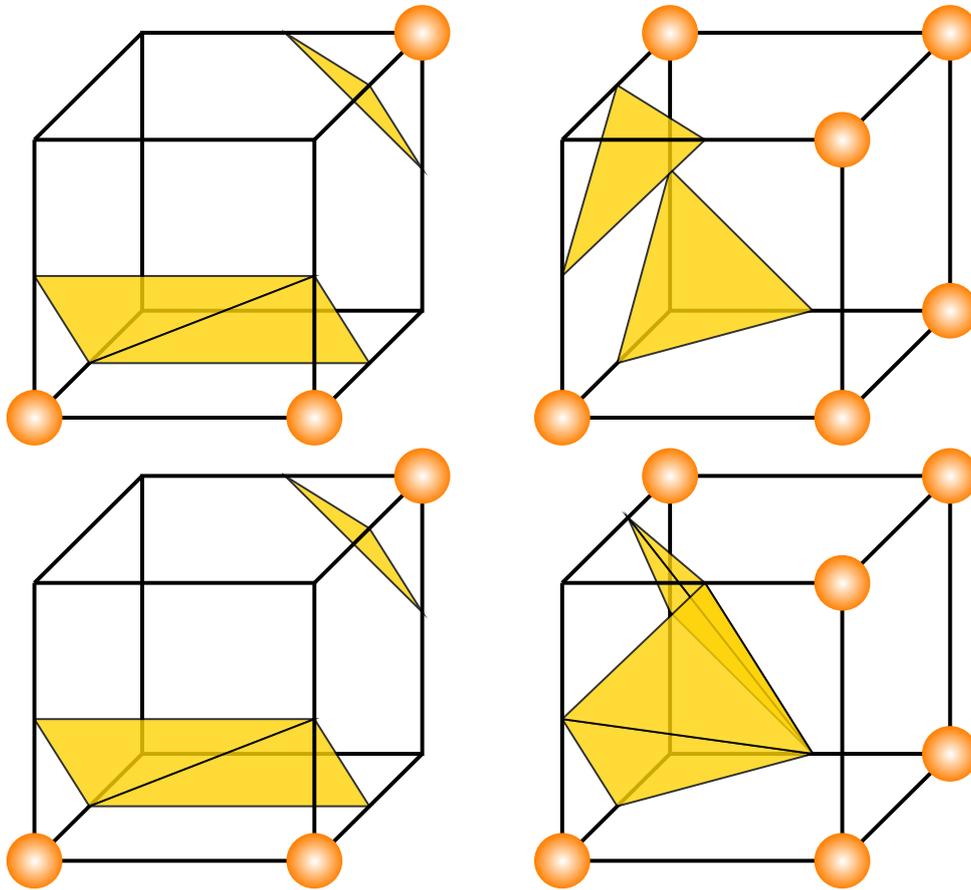


Abbildung 2.8: Eine inkorrekte Darstellung, unten die korrigierte Version

Heutige Implementationen des Marching Cubes Algorithmus stützen sich auf Arbeiten wie [NH91] und [Nie03], in welchen Methoden entwickelt wurden, um diese uneindeutigen Fälle zu entscheiden.

2.3 Verarbeitung der Würfel

Nach diesen Vorbetrachtungen wird nun erläutert, wie die Entscheidung über den Schnitt, zwischen jedem Würfel und S_c , getroffen wird. In einem Voxelgitter mit Dimensionen $x \times y \times z$ muss für jeden der $(x - 1) \cdot (y - 1) \cdot (z - 1)$

Würfel eine Reihe von Schritten durchlaufen werden. Diese Schritte produzieren schließlich zwischen null und fünf Dreiecke, deren Vertices auf den Kanten des Würfels liegen.

1. Berechnung des Würfelindex anhand der Werte der Würfecken
2. Ermitteln der Kanten des Würfels, auf denen Dreiecksvertices liegen
3. Gruppierung der Dreiecksvertices in Dreiecke
4. Approximation der Normalen an den Würfecken
5. Interpolation der Positionen der Dreiecksvertices
6. Interpolation der Normalen an den Dreiecksvertices
7. Speichern der resultierenden Dreiecke und Normalen

Schritt 1 erfolgt wie in 2.2.1 beschrieben. Die Speicherung der Vertices und Normalen (Schritt 7) kann beliebig erfolgen, allerdings bietet es sich an, jeweils eine Liste für einmalige Vertices und Normalen und zusätzlich eine Liste mit Indices in diese Listen zu verwenden. Dies reduziert den benötigten Speicherplatz da sich die Dreiecke untereinander viele Vertices teilen. Ein Beispiel für ein solches Speicherformat ist in 3.1.1 beschrieben.

2.3.1 Lookup - Kanten und Dreiecke

Um zu ermitteln, ob auf einer Kante des Würfels ein Dreiecksvertex liegt (Schritt 2), genügt es, die Werte der Würfecken, die die Kante verbindet, mit c zu vergleichen. Liegt der Wert einer der Ecken unter (oder gleich) c , und ist der Wert der anderen größer (oder gleich) c , so liegt auf der Kante ein Vertex. Dieser Vergleich muss für jede der 12 Kanten jedes Würfels im Datensatz durchgeführt werden. Da es nur, wie in 2.2.1 beschrieben, 256 Konfigurationen für den Würfel gibt, kann dieser Prozess durch einen Lookup mit Hilfe des Würfelindex ersetzt werden. Implementationen des Algorithmus verwenden oft ein `int []` mit 256 Einträgen, in dem für jeden Würfelindex ein `int` steht. In dem `int`, an der dem Würfelindex entsprechenden Position im Array, ist das Bit i genau dann 1, wenn auf der Kante mit dem Index i ein Dreiecksvertex liegt.

Nachdem nun feststeht, auf welchen Kanten sich Dreiecksvertices befinden, müssen diese in Gruppen von drei Vertices eingeteilt werden, um Dreiecke zu bilden. Auch hier bietet sich ein Lookup über den Würfelindex an. Es wird ein `int [] []` erstellt, in dem für jeden möglichen Würfelindex ein Array

aus Indizes steht. Drei aufeinanderfolgende Indizes in einem solchen Array ergeben ein Dreieck. Diese Arrays bilden die Konfigurationen aus 2.2.2 ab. Durch diesen Vorgang wird Schritt 2 und 3 abgedeckt.

2.3.2 Normalen an den Würfecken

An jeder Würfecke wird eine Normale mittels zentraler Differenzen approximiert. Diese dient der späteren Interpolation der Normalen der Dreiecksvertices. Die Normale $n \in \mathbb{R}^3$ der Würfecke an der Position (i, j, k) ergibt sich, wie folgt:

$$n_x = \frac{\varphi(i-1, j, k) - \varphi(i+1, j, k)}{\Delta x} \quad (2.1)$$

$$n_y = \frac{\varphi(i, j-1, k) - \varphi(i, j+1, k)}{\Delta y} \quad (2.2)$$

$$n_z = \frac{\varphi(i, j, k-1) - \varphi(i, j, k+1)}{\Delta z} \quad (2.3)$$

Hierbei sind Δx , Δy , und Δz die Kantenlängen des Würfels.

2.3.3 Dreiecksvertices - Interpolation der Position und Normalen

Für jede Kante, auf der ein Dreiecksvertex liegt, muss nun berechnet werden, wo sich dieser befindet und was die zugehörige Normale ist. Ein möglicher Ansatz für die Position ist, diese genau mittig zwischen den Würfecken der Kante festzulegen. Diese Vereinfachung führt allerdings zu einem sehr eckigen Mesh. In [LC87] wird deshalb lineare Interpolation verwendet, um Position und Normale des Vertex zu bestimmen.

Verbindet man zwei Datenpunkte (x_0, f_0) und (x_1, f_1) mit einer Strecke, so gilt:

$$f(x) = f_0 + \frac{f_1 - f_0}{x_1 - x_0}(x - x_0)$$

Für f_0 und f_1 werden die Werte von φ an den Ecken des Würfels, die durch die aktuelle Kante verbunden werden, verwendet. Um S_c bestmöglich abzubilden, wird der Vertex genau auf den interpolierten Schnittpunkt zwischen der Oberfläche und der Würfelkante gelegt. Dazu wird $f(x) = c$ gesetzt, um

den Wert des Schnittpunktes x berechnen zu können. Dieser ergibt sich aus folgender Umformung:

$$\begin{aligned} f(x) &= f_0 + \frac{f_1 - f_0}{x_1 - x_0}(x - x_0) \\ \Leftrightarrow \frac{f(x) - f_0}{f_1 - f_0}(x_1 - x_0) + x_0 &= x \\ \Leftrightarrow \frac{f(x) - f_0}{f_1 - f_0}x_1 + \left(1 - \frac{f(x) - f_0}{f_1 - f_0}\right)x_0 &= x \end{aligned}$$

Die x, y, z Werte der Position und Normalen ergeben sich aus der obigen Gleichung, wenn man für x_1 und x_0 die x, y, z Werte der Positionen bzw. Normalen der Würfecken einsetzt. Der Wert des Bruches $\frac{f(x)-f_0}{f_1-f_0}$ muss dabei nur einmal berechnet werden, da er sich für die einzelnen Komponenten der beiden Vektoren (Position und Normale) nicht ändert. Dies trägt zur Optimierung der Interpolationsschritte 5 und 6 bei.

Kapitel 3

Implementierung - V8

Im Rahmen dieser Arbeit ist das Programm ‚V8‘ entstanden. V8 besteht aus einer Java Implementation des Marching Cubes Algorithmus, einer in JavaFX erstellten GUI, mithilfe derer der Algorithmus auf verschiedene Datenquellen angewandt und konfiguriert werden kann, und einer 3D Ansicht für den resultierenden Mesh. Diese ist mittels OpenGL¹ hardwarebeschleunigt und stützt sich auf die *Lightweight Java Game Library* (LWJGL²).

Zuerst soll nun eine Übersicht über die Klassen der Implementierung des Marching Cubes Algorithmus gegeben werden. Anschließend beschreiben die Kapitel 3.3, 3.4, und 3.5 die Visualisierungs- und Konfigurationsmöglichkeiten der GUI.

3.1 Marching Cubes Klassenübersicht

Die Kernkomponente von V8 ist die Implementierung des Marching Cubes Algorithmus im Paket `model.mc_alg`. In diesem Paket befindet sich die Hauptklasse `MCRunner` und alle Hilfsklassen, derer sich diese bedient. Im Folgenden soll ein Überblick über die Verwendungszwecke und Schnittstellen der wichtigsten Klassen gegeben werden.

¹<http://www.opengl.org/>

²<http://lwjgl.org/>

3.1.1 MCRunner

Der Marching Cubes Algorithmus, wie er im Kapitel 2 beschrieben wurde, ist in der Klasse `MCRunner` implementiert. Die Ausführung des Algorithmus findet in der Methode `run()` statt, welche von dem implementierten Interface `Runnable` vorgeschrieben wird. Weitestgehend wurde intern das gleiche Vorgehen, wie schon in [LC87] beschrieben, eingesetzt. Um die Ausführung des Algorithmus zu beschleunigen, wurde zusätzlich ein Cache implementiert, der schon interpolierte Dreiecksvertices vorhält. In [LC87] wurde diese Optimierung zwar beschrieben, aber nicht ausgeführt. Die Tabellen, die der Algorithmus für den Lookup benötigt, wurden in die Klasse `Tables` ausgelagert. Auf sie kann mittels statischer Methoden zugegriffen werden.

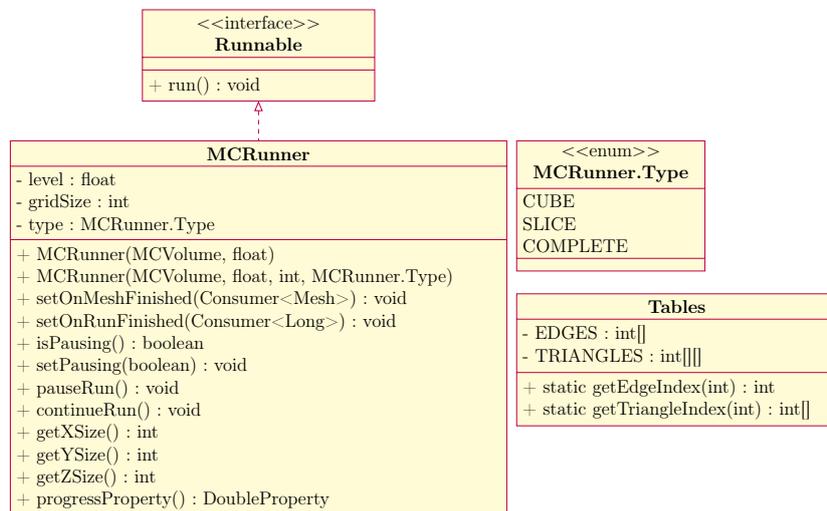


Abbildung 3.1: Klassendiagramm von `MCRunner` und zugehöriger Klassen

Die Klasse wird größtenteils über die Konstruktoreparameter konfiguriert. Der Konstruktor `MCRunner(MCVolume, float)` erwartet nur die Datenquelle für den Algorithmus und den Schwellwert. Die Grid Size wird auf eins eingestellt, und der verwendete Type ist `COMPLETE`. Über den Konstruktor `MCRunner(MCVolume, float, int, MCRunner.Type)` können alle diese Optionen spezifiziert werden. Die Bedeutung der Parameter Grid Size und Type werden im Abschnitt 3.4.2 bzw. 3.4.3 erläutert.

Da das `Runnable` Interface implementiert wird, kann `MCRunner` einfach in einem eigenen Thread ausgeführt werden. Um den aus dem Marching Cubes Algorithmus resultierenden Mesh zu erhalten, muss ein `Consumer<Mesh>` über die Methode `setOnMeshFinished(Consumer<Mesh>)` registriert werden. Dessen `accept(Mesh)` Methode wird aufgerufen, sobald ein Mesh erstellt wurde.

Je nachdem, welcher `Type` gewählt wurde, geschieht dies mehrmals (nach jedem Würfel oder jeder Scheibe des Datensatzes) oder nur einmal, nachdem der Algorithmus abgeschlossen ist. Der `Consumer<Long>`, der mittels der Methode `setOnRunFinished(Consumer<Mesh>)` registriert wurde, wird aufgerufen, nachdem der Marching Cubes Algorithmus abgeschlossen ist. Da es sich bei `Consumer<T>` um ein funktionales Interface handelt, kann beiden `setOnXFinished` Methoden (seit Java 8) ein Lambda Ausdruck oder eine Methodenreferenz übergeben werden. Dies macht die Verwendung in Verbindung mit einer GUI besonders einfach.

Weiter ist es möglich, die Ausführung des Algorithmus zu pausieren beziehungsweise fortzusetzen. Hierfür werden die Methoden `pauseRun()` und `continueRun()` eingesetzt. Diese werden den Thread, der die `run()` Methode des `MCRunner` ausführt, in den `WAIT` Modus versetzen und bei Aufruf der `continueRun()` Methode wieder aufwecken. Es ist auch möglich, den `MCRunner` so zu konfigurieren, dass die Ausführung jedes mal pausiert wird, wenn ein `Mesh` an den `Consumer<Mesh>` weitergegeben wurde. Dies geschieht mittels der Methode `setPausing(boolean)`.

Anpassung des Outputs an OpenGL

Die Klasse `Mesh` dient als Container für zwei `FloatBuffer` und einen `IntBuffer`, die die Informationen über den Mesh enthalten. Um zu vermeiden, dass diese Daten verarbeitet oder umformatiert werden müssen, bevor sie von `MeshView3D` an OpenGL weitergegeben werden können, wurden die Buffer wie folgt formatiert:

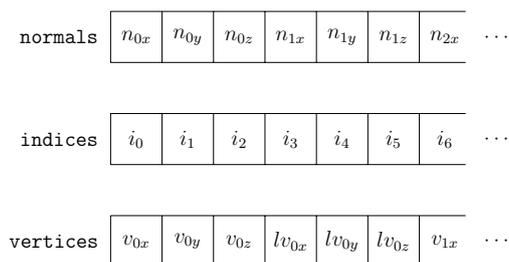


Abbildung 3.2: Mesh Speicherformat

Der `FloatBuffer` namens `vertices` enthält Gruppen von je drei `float`, welche die Position eines Vertex angeben. Zusätzlich enthält der Buffer für jeden Vertex eines Dreiecks eine Gruppe von `float` so, dass eine Linie von (v_{ix}, v_{iy}, v_{iz}) nach $(lv_{ix}, lv_{iy}, lv_{iz})$ die Normale des Vertex mit dem Index i

symbolisiert. In dem Buffer `normals` befinden sich Dreiergruppen von `float`, welche die Normalenvektoren an den Dreiecksvertices angeben. Um aus diesen Daten Dreiecke zu konstruieren, muss der Buffer `indices` durchlaufen werden. Dieser Buffer enthält Indizes von (v_{ix}, v_{iy}, v_{iz}) bzw. (n_{ix}, n_{iy}, n_{iz}) Gruppen. Drei Elemente von `indices` definieren ein Dreieck.

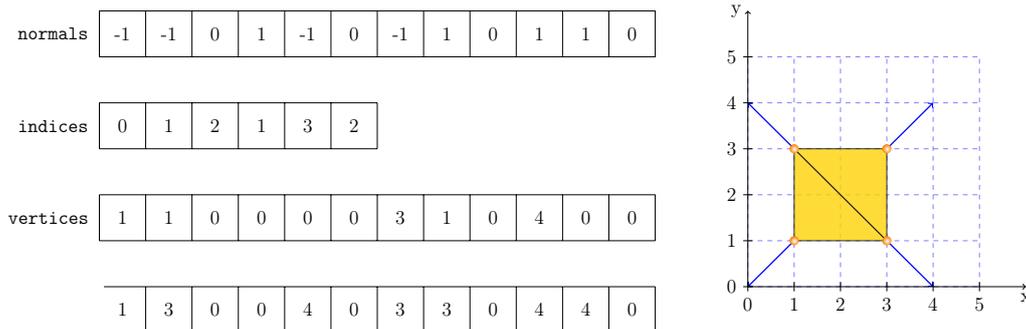


Abbildung 3.3: Speicherung zweier Dreiecke

In Abbildung 3.3 ist der Speicherinhalt der Buffer für zwei Dreiecke dargestellt, die beide in der $X \times Y$ Ebene liegen. Die Normalen der Dreiecksvertices wurden durch blaue Pfeile repräsentiert. Durch die Verwendung des `indices` Buffers müssen die Vertices bei $(3, 1)$ und $(1, 3)$ (und deren Normalen) nur einmal gespeichert werden. Würde diese Speicherform nicht eingesetzt, müssten sie doppelt in den Buffern existieren.

3.1.2 MCVolume

Um `MCRRunner` die Möglichkeit zu geben, beliebige Datenquellen zu verwenden, werden diese durch das Interface `MCVolume` repräsentiert. Mittels der Methoden `getXSize()`, `getYSize()` und `getZSize()` bestimmt `MCRRunner` die Größe des Volumens, über das der Marching Cubes Algorithmus ausgeführt werden soll. Die Methode `getValue(int, int, int)` dient dazu, den Wert des Volumens an einer gegebenen (x, y, z) Position zu erfragen.

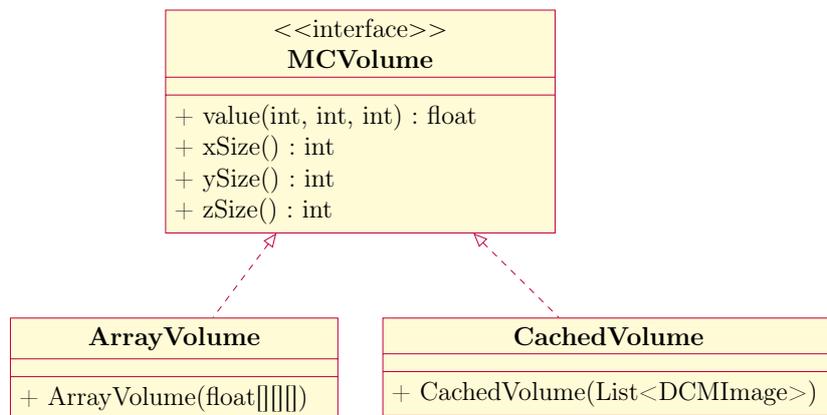


Abbildung 3.4: Klassendiagramm des **MCVolume** und Implementierungen

Für V8 wurde das **MCVolume** Interface auf drei verschiedene Arten implementiert.

ArrayVolume

Das **ArrayVolume** ist die einfachste denkbare Datenquelle für den Algorithmus. Hier wurde eine Containerklasse für ein 3D `float` Array implementiert. Die Methoden, welche die Größe des Volumens zurückgeben, beziehen sich direkt auf die Länge des Arrays und seiner Sub-Arrays; `getValue(int, int, int)` ist einfach als `return data[z][y][x]` implementiert.

CachedVolume

Das **ArrayVolume** bietet einen sehr schnellen Zugriff auf die Volumendaten, allerdings setzt es voraus, dass das komplette Volumen gleichzeitig im Arbeitsspeicher gehalten wird. Wie in Kapitel 2 angemerkt, ist für die Ausführung des Algorithmus allerdings nur die Vorhaltung von vier Scheiben des Datensatzes nötig. Das **CachedVolume** bietet die Möglichkeit, nur die letzten vier (oder eine beliebige andere Anzahl) per `getValue(int, int, int)` angefragten Scheiben im Arbeitsspeicher vorzuhalten. Die Scheiben werden in der Form von **DCMImage** (siehe 3.3.1) Instanzen an das **CachedVolume** übergeben, welches nach Bedarf den zur Umwandlung in ein `float[][]` nötigen Festplattenzugriff durchführt.

Da der Algorithmus die Scheiben der Reihe nach in überlappenden Vierergruppen (zuerst Scheiben null bis drei, dann Scheiben eins bis vier) benötigt,

wird jedes DICOM Bild genau einmal von der Festplatte geladen. Dies führt dazu, dass nur ein Bruchteil des gesamten Datensatzes im Speicher gehalten werden muss, verlangsamt aber den Algorithmus, da das Nachladen der Daten während der Durchführung geschieht.

MetaBallVolume

Die Klasse `MetaBallVolume` war zuerst nur als Sammlung von bzw. Fabrik für `MetaBall` Instanzen gedacht. Um den Marching Cubes Algorithmus auf ein `MetaBallVolume` anzuwenden, wurde dieses in ein `float[][][]` umgewandelt und mittels des `ArrayVolume` an den `MCRRunner` übergeben. Dies bringt die schon beschriebenen Nachteile (Speicherverbrauch) mit sich. Um sie zu umgehen, implementiert auch `MetaBallVolume` das `MCVolume` Interface. Die direkte Verwendung einer `MetaBallVolume` Instanz als Datenquelle erhöht die Laufzeit des Algorithmus allerdings massiv. Die `getValue(int, int, int)` Funktion muss die in Kapitel 3.3.2 beschriebene Gleichung berechnen. Aufgrund dessen wird in `V8` weiterhin die alte Methode verwendet.

3.1.3 Cube

Die Klasse `Cube` modelliert einen der namensgebenden Würfel des Marching Cubes Algorithmus.

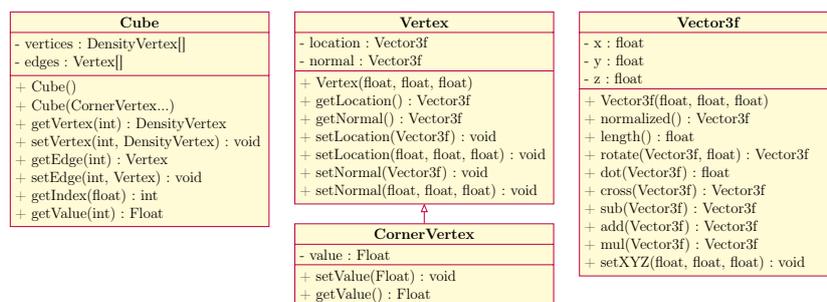


Abbildung 3.5: Klassendiagramm von Würfel und Vertex Klassen

Der `Cube` enthält Arrays von `Vertices`, welche die Ecken und Dreiecksvertices auf den Kanten des Würfels repräsentieren. Zwei Klassen werden verwendet um diese `Vertices` zu modellieren:

- **Vertex**

Der `Vertex` ist eine Containerklasse für die zwei `Vector3f` Instanzen

`location`, welche die Position des Vertex angibt und `normal`, die die Normale an dem Vertex darstellt. Um Referenzkomplikationen vorzubeugen, kopieren alle Setter, die `Vector3f` als Parameter nehmen, die Werte des übergebenen Vektors in die internen Vektoren des `Vertex`.

- **CornerVertex**

Diese Klasse wird verwendet um die Vertices der Würfecken darzustellen. Sie erweitert `Vertex` um ein `float` Feld, welches den Wert des Voxelgitters an der Würfecke speichert, die der `CornerVertex` repräsentiert.

Ein `Cube` enthält acht `CornerVertex` und zwölf `Vertex` Instanzen. Neben seiner Funktion als Datencontainer enthält die `Cube` Klasse die für den Algorithmus zentrale Methode `getIndex(float)`, welche, gegeben den Schwellwert, den Würfelindex aus 2.2.1 berechnet.

3.2 GUI Übersicht

Die folgenden Abschnitte beschäftigen sich mit der grafischen Benutzeroberfläche von V8. Die GUI besteht aus einem Hauptfenster, zwei Vorschauan-sichten, die sich in ihrem eigenen Fenster öffnen, und der 3D Ansicht des Mesh, welche auch ein eigenes Fenster benötigt.

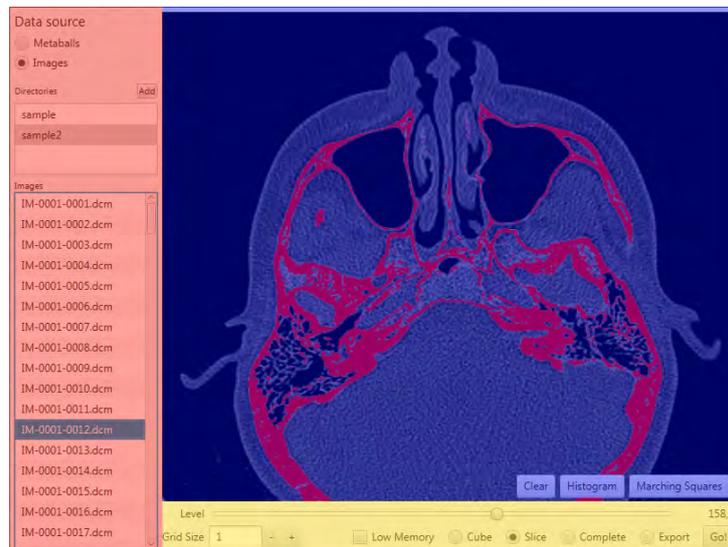


Abbildung 3.6: Das Hauptfenster von V8

Das Hauptfenster ist in drei große Abschnitte unterteilt, welche in Abb. 3.6 Rot, Blau, bzw. Gelb markiert sind. Die Abschnitte haben die folgenden Aufgaben:

- **Rot**

In diesem Abschnitt wird die Datenquelle ausgewählt. Ist ‚Images‘ ausgewählt, so kann über den Button ‚Add‘ ein Ordner hinzugefügt werden, welcher .dcm Bilder enthält. Einer dieser Ordner kann ausgewählt werden, woraufhin die Dateien, die in ihm enthalten sind, in der unteren Liste angezeigt werden. Dort kann eines der Bilder markiert werden, um es anzuzeigen.

- **Blau**

Hier wird das links markierte Bild angezeigt. Zusätzlich enthält dieser Abschnitt Buttons, um die verschiedenen Vorschauansichten zu steuern.

- **Gelb**

Dieser Abschnitt enthält die GUI Elemente zur Konfiguration (siehe 3.4) und den Button, welcher den Algorithmus startet.

Alle Vorschauansichten (auf die im Abschnitt 3.4.1 eingegangen wird) öffnen sich zwar in ihrem eigenen Fenster, reagieren aber dennoch auf die Benutzereingaben im Hauptfenster. Das Histogramm zeigt beispielsweise immer die Daten für das derzeit ausgewählte Bild an. Ebenso reagiert die Marching Squares Vorschau auf Veränderungen von Level oder Grid Size.

3.3 Auswahl der Datenquellen

In dem auf Abb. 3.6 rot markierten Bereich kann ausgewählt werden, welche Datenquelle für den Algorithmus gewünscht ist. In V8 werden zwei verschiedene Datenquellen unterstützt, die nun beschrieben werden sollen.

3.3.1 DICOM Bilder

Digital Imaging and Communications in Medicine oder kurz DICOM ist ein standardisiertes Dateiformat (und Kommunikationsprotokoll), welches im medizinischen Umfeld für den Datenaustausch verwendet wird. Praktisch alle Systeme wie Computertomographen oder Röntgengeräte produzieren Bilder im DICOM Format. In V8 bilden die in solchen Dateien gespeicherten

Dichtewerte die primäre Datenquelle für den Marching Cubes Algorithmus. Mittels der Bibliothek ‚dcm4che‘³ werden die in den .dcm Dateien gespeicherten Scheiben in Grauwertbilder umgewandelt, die mit der JavaFX Klasse `ImageView` angezeigt werden können. Um aus den Bildern ein 3D Array aus Grauwerten zu erhalten, werden die 2D Arrays der Bilder in der Reihenfolge, wie sie aus dem Ordner geladen wurden, als Sub-Arrays eines 3D Array verwendet. Dieses wird an den Algorithmus weitergegeben.

3.3.2 Zufällige Volumen - Metabälle

Ein Metaball (wie beschrieben in [Bli82]) ist eine Funktion, deren Werte verwendet werden, um ein Voxelgitter für den Marching Cubes Algorithmus zu erstellen. Um einen Metaball B zu definieren, wird der Mittelpunkt (x_0, y_0, z_0) und die Intensität I des Metaballs angegeben. Der Wert des Metaballs an der Stelle (x, y, z) ergibt sich aus:

$$B(x, y, z) = \frac{I}{(x - x_0)^2 + (y - y_0)^2 + (z - z_0)^2}$$

Typischerweise enthält ein Volumen mehr als einen Metaball. Für ein Volumen V mit n Metabällen ist der Wert an der Stelle (x, y, z) :

$$V(x, y, z) = \sum_{i=1}^n B_i(x, y, z)$$

Volumen, die sich aus mehreren Metabällen unterschiedlicher Intensitäten und Positionen zusammensetzen, ergeben interessante Formen. In 3.8 ist die aus einem Volumen mit zwei positiven und einem negativen Metaball extrahierte Oberfläche dargestellt.

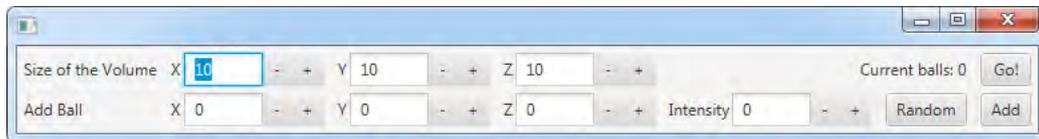


Abbildung 3.7: Dialog zum Erstellen eines Metaball-Volumens

In V8 wird die Klasse `MetaBallVolume` genutzt, um aus einer Menge von `MetaBall` Instanzen ein `float[][][]` zu erstellen. Dieses kann als Datenquelle für den Marching Cubes Algorithmus verwendet werden.

³<http://www.dcm4che.org/>

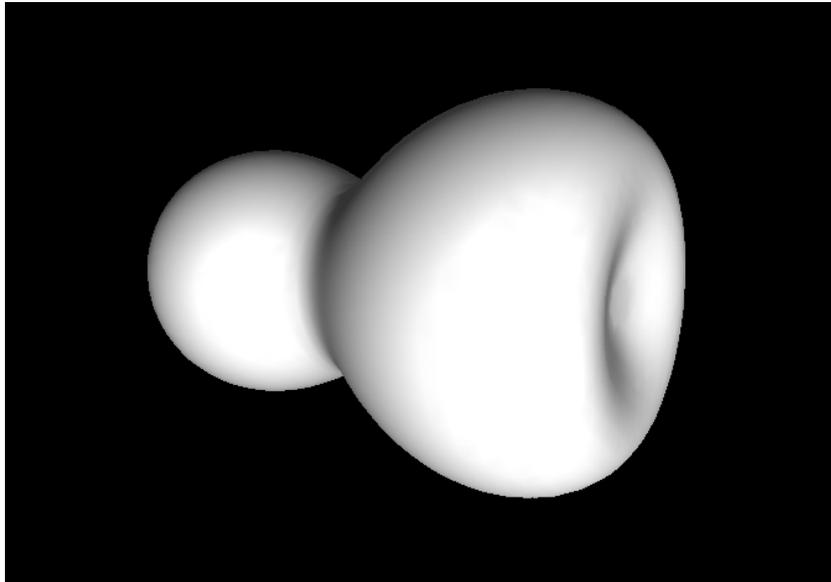


Abbildung 3.8: Ein Volumen aus drei Metabällen

Um direkt aus dem Programm solche Volumen erstellen zu können, bietet V8 den Dialog in Abbildung 3.7 an. Wählt der Benutzer als Datenquelle ‚Metaballs‘, erscheint dieser Dialog nach dem Klick auf den ‚Go‘ Button. Es besteht die Möglichkeit, die Größe des Volumens zu konfigurieren und Metabälle mit manuell spezifizierten oder zufälligen Parametern hinzuzufügen.

3.4 Konfiguration des Algorithmus

Die Konfiguration des Algorithmus geschieht im gelben Bereich des GUI Hauptfensters (Abb. 3.6). Der Marching Cubes Algorithmus an sich hat zwei Parameter, die konfigurierbar sind. Durch die Konfiguration von Grid Size und Level kann die Geschwindigkeit und die extrahierte Oberfläche beeinflusst werden. Zusätzlich lässt sich über die GUI einstellen, wie häufig der derzeitige Mesh an die 3D Ansicht weitergegeben werden soll (mehr dazu in Kapitel 3.4.3). Diese Konfigurationsoptionen können über den Konstruktor der Klasse `MCRunner` übergeben werden.

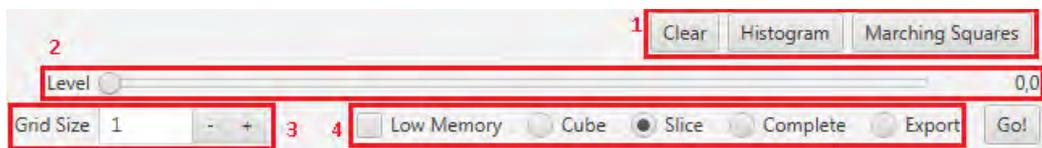


Abbildung 3.9: Die Konfigurationsoptionen von V8

3.4.1 Level

In V8 wird als 'Level' der in 2 eingeführte Schwellwert c bezeichnet. Dieser Wert kann über den in Abbildung 3.9 als 2 markierten Schieberegler eingestellt werden. Eine gute Wahl des Schwellwertes ist maßgeblich dafür, ob der produzierte Mesh die gewünschte Oberfläche darstellt oder nicht. Um dem Benutzer bei der fundierten Wahl des Schwellwertes zu helfen, wurden drei Werkzeuge implementiert, die nun vorgestellt werden sollen.

Histogramm

Ein Histogramm ist, eine Möglichkeit die Häufigkeit des Vorkommens einer Menge von Werten darzustellen. In V8 kann der in Abbildung 3.9 bei 1 enthaltene Button 'Histogramm' betätigt werden, um die Häufigkeitsverteilung der 256 möglichen Grauwerte im aktuellen Bild anzuzeigen. Diese Werte sind die Voxelwerte einer Scheibe des Volumens, mit dem später der Marching Cubes Algorithmus arbeiten wird.

Die Grauwerte des Bildes bewegen sich im Intervall $[0, 255]$. Es hat sich jedoch gezeigt, dass die Werte 0 und 255 meist wesentlich häufiger als alle anderen Werte vorkommen. Würde der Wertebereich des Histogramms so gewählt werden, dass die Anzahl für diese Werte angezeigt werden kann, ließen sich die Häufigkeiten der anderen Werte nur noch schlecht erkennen. Folglich wurden diese separat als Text angezeigt.

Marching Cubes Vorschauansicht

Noch nützlicher als die Analyse der Häufigkeitsverteilung der Pixelwerte ist es, zu betrachten, welche Pixel vom Algorithmus als solide und welche als transparent betrachtet werden. In V8 wurde diese Unterscheidung direkt in das derzeit angezeigte Bild, das eine Scheibe des Datensatzes repräsentiert, integriert.

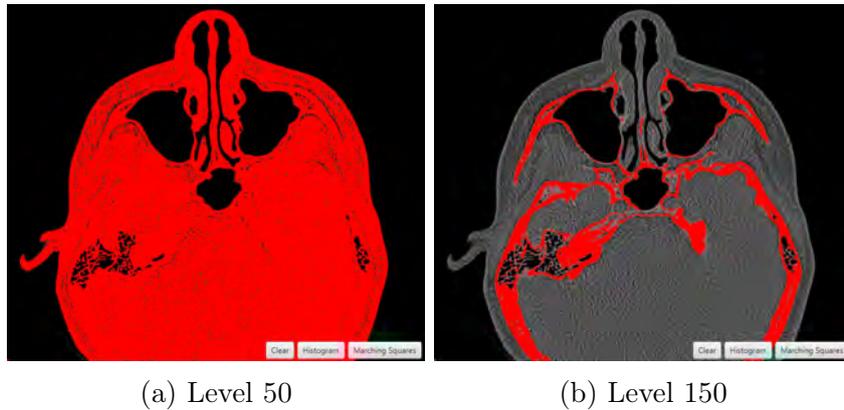


Abbildung 3.10: Die Vorschauansicht für zwei Schwellwerte

Wird das Level über den entsprechenden Schieberegler (Abb. 3.9 Box 2) verändert, färbt V8 die Pixel des Bildes rot, die als solide betrachtet würden. Alle anderen Pixel behalten ihre ursprüngliche Farbe. Das Einfärben der Pixel geschieht kontinuierlich, während der Schieberegler bewegt wird. Das macht es möglich, sehr schnell einen sinnvollen Schwellwert für die gewünschte Oberfläche zu wählen. Über den Button ‚Clear‘ können die roten Pixel auf ihren Grauwert zurückgesetzt werden.

Marching Squares Vorschauansicht

Der Marching Squares Algorithmus ist ein Algorithmus, der analog zu Marching Cubes aus 2D Pixelgittern Oberflächen extrahiert. In V8 existieren im Paket `model.ms_alg` die 2D Entsprechungen der Klassen aus dem Paket `model.mc_alg`. Über den Button ‚Marching Squares‘ kann ein Fenster geöffnet werden, in dem das Ergebnis des Marching Squares Algorithmus für das derzeitige Bild angezeigt wird.

Der Marching Squares Algorithmus produziert eine Isofläche wie in Kapitel 2 mit $n = 2$. Es lässt sich deshalb anhand der roten Punkte in Abb. 3.11a leicht erkennen, dass der Marching Cubes Algorithmus einen Mesh mit sehr vielen Artefakten produzieren würde. Gleichzeitig ist klar zu sehen, wie der äußere Rand die Haut um den Schädel modelliert. Der in Abb. 3.11b gewählte Schwellwert hingegen würde einen vergleichsweise einfachen Mesh produzieren, der die Schädelknochen abbildet. Ein weiterer Vorteil dieser Vorschauansicht ist, dass die Grid Size einen Einfluss auf die extrahierte Oberfläche hat. Dieser Effekt lässt sich mit der Ansicht aus 3.4.1 nicht anzeigen.

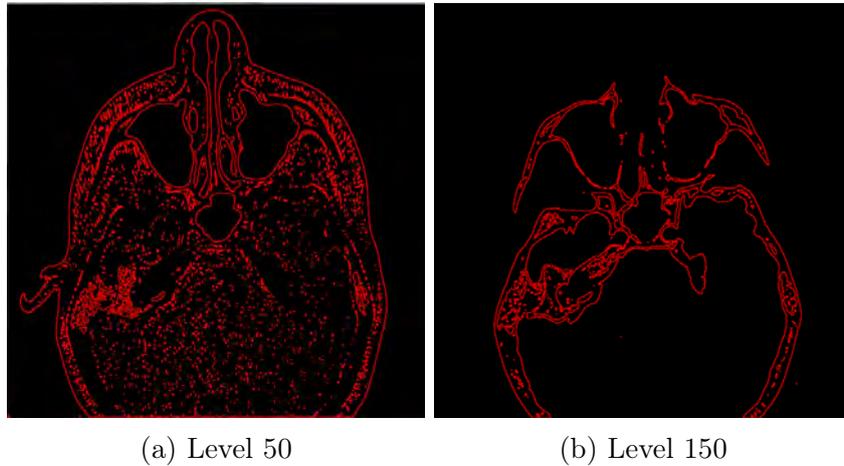


Abbildung 3.11: Die Vorschauansicht für zwei Schwellwerte

3.4.2 Grid Size

Über die ‚Grid Size‘ kann der Abstand zwischen Datenpunkten, die einen Würfel bilden, eingestellt werden. Die Grid Size ist somit die Seitenlänge eines Würfels im Marching Cubes Algorithmus. Die Ecken mit Index 0 und 6 eines Würfels repräsentieren die Voxel des Datensatzes mit den Koordinaten (x, y, z) und $(x + g, y + g, z + g)$, wobei g die Grid Size bezeichnet.

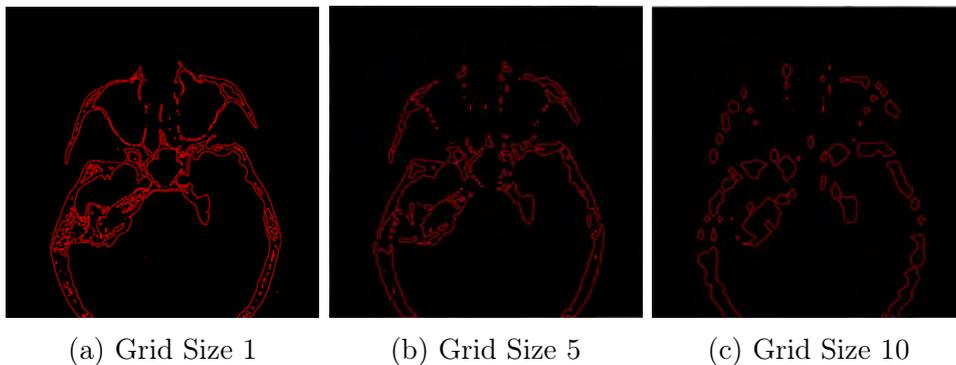


Abbildung 3.12: Die Vorschauansicht für Level 150 und drei Grid Size Werte

Durch die Erhöhung der Grid Size werden folglich Datenpunkte ausgelassen. In Abb. 3.12 ist die resultierende niedrigere Qualität des Mesh zu sehen. In 2.3 wurde die Anzahl der zu berechnenden Würfel (a) als

$$a = (x - 1) \cdot (y - 1) \cdot (z - 1)$$

angegeben.

Durch die Erhöhung der Grid Size kann diese stark reduziert werden. Es gilt allgemein, dass die Anzahl als

$$a = \frac{(x - 1) \cdot (y - 1) \cdot (z - 1)}{g^3}$$

gegeben ist. Die Geschwindigkeit des Algorithmus hängt direkt von dieser Anzahl ab. Liegt ein großer Datensatz vor, so kann eine höhere Grid Size genutzt werden, um eine Vorschau des 3D Mesh zu erstellen. In 4.3 wird ein Beispiel dessen vorgestellt.

3.4.3 Ausgabe von Zwischenergebnissen

Der Mesh, den der Marching Cubes Algorithmus erstellt, wird nach und nach aus den Dreiecken, die aus den einzelnen Würfeln resultieren (siehe 2.3), zusammengebaut. In V8 ist es möglich, den Fortschritt des Mesh zu beobachten, während der Algorithmus läuft. Mittels der Optionen in Abb. 3.9 Box 4 lässt sich einstellen, in welchen Abständen MCRunner den derzeitigen Mesh an die 3D Ansicht weitergeben soll. Die möglichen Abstände sind:

- **Cube** nach jedem Würfel
- **Slice** nach jeder Scheibe des Datensatzes
- **Complete** nachdem der komplette Datensatz abgearbeitet wurde

Da dieses Weitergeben allerdings nicht kostenlos ist, verlangsamen die Optionen Slice und (besonders) Cube die Ausführung des Algorithmus. Slice sollte normalerweise eingesetzt werden, um den Algorithmus zu beobachten. Cube kann für Minimalbeispiele („Was macht der Algorithmus für diese zwei Würfelkonfigurationen?“) verwendet werden.

3.4.4 Export nach OBJ und STL

Zur Transformation und Weiterverarbeitung von 3D Meshes existieren sehr mächtige Programme, wie beispielsweise Maya⁴, Blender⁵ oder MODO⁶. Weiter können 3D Meshes durch Programme wie Slic3r⁷ in Instruktionen für 3D

⁴<http://www.autodesk.de/products/maya/overview>

⁵<http://www.blender.org/>

⁶<http://www.thefoundry.co.uk/products/modo/>

⁷<http://slic3r.org/>

Drucker umgewandelt werden. Um es zu ermöglichen, dass Meshes, die mit V8 erstellt wurden, von solchen Programmen verwendet werden können, existiert in V8 die Möglichkeit, das Ergebnis des Marching Cubes Algorithmus zu exportieren. Es werden zwei Formate angeboten:

- OBJ
Das ‚Wavefront OBJ‘ Format, entwickelt von der Firma ‚Wavefront Technologies‘, kann von allen großen 3D Bearbeitungsprogrammen geöffnet werden. Die Dateierweiterung ist .obj
- STL
Das ‚Surface Tesselation Language‘ Format wurde von ‚3D-Systems, Inc.‘ entwickelt und hat sich als Standardformat für Slicer etabliert. Die Dateierweiterung ist .stl

Da der Algorithmus sehr viele Dreiecke produziert, können diese Exporte hunderte von MB groß sein. Ein OBJ Export ist dabei meist größer als einer im STL Format, da es sich bei OBJ um ein Klartextformat handelt, wohingegen in V8 die binäre Variante des STL Formates gewählt wurde. Beide Formate lassen sich allerdings hervorragend komprimieren. Beispielhaft wurde ein Mesh mit über 3,6 Millionen Dreiecken in beiden Formaten exportiert und mit dem LZMA Verfahren gepackt. Die Größen der Dateien zeigt die folgende Tabelle.

Anzahl der Dreiecke	Größe (Byte)	Gepackte Größe (Byte)
3 664 909	342 478 102	55 388 784
3 664 909	183 245 534	63 965 798

Wenn die Exporte als gepackte Dateien aufbewahrt werden, kann OBJ somit dennoch das effektivere Format sein.

3.5 3D Ansicht

Um den 3D Mesh anzuzeigen, wurde zuerst die JavaFX Klasse `TriangleMesh` verwendet. Dies bot den Vorteil, dass die 3D Ansicht direkt in die JavaFX GUI integriert werden konnte. Für große Meshes, mit mehr als einer Million Dreiecken, war die Performanz dieser Lösung allerdings nicht zufriedenstellend. Es wurde deswegen mit der Klasse `MeshView3D` eine 3D Ansicht eines Mesh in ihrem eigenen Fenster implementiert. `MeshView3D` ist mittels OpenGL hardwarebeschleunigt und kann (mit einer aktuellen Grafikkarte) auch Meshes mit mehreren Millionen von Dreiecken flüssig darstellen.

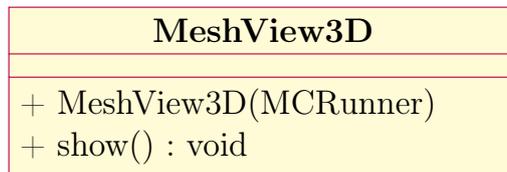


Abbildung 3.13: Klassendiagramm von MeshView3D

Der Klasse `MeshView3D` wird eine Instanz von `MCRunner` übergeben. Die Methode `show()` startet die übergebene Instanz in einem neuen Thread, öffnet ein neues Fenster und blockiert, bis dieses geschlossen wird. Während das Fenster offen ist, wird der letzte vom `MCRunner` übergebene Mesh angezeigt. Es stehen dem Benutzer verschiedene Möglichkeiten der Navigation und Visualisierung des Mesh zur Verfügung, welche nun erläutert werden sollen.

3.5.1 Visualisierungsmöglichkeiten

Die wohl wichtigste Aufgabe des `MeshView3D` ist es, den Mesh möglichst gut darzustellen. Um alle Aspekte des Mesh zu visualisieren, wurden verschiedene Modi der Anzeige implementiert. Außerdem werden Hilfsmittel wie ein Koordinatensystem und Einheitswürfel angeboten, um das Fehlersuchen im Mesh zu vereinfachen.

- **Drahtgittermodell**
Die Dreiecke des Mesh können entweder als Linien, die die Vertices verbinden, oder als ausgefüllte Fläche dargestellt werden.
- **Lichtsimation**
Die OpenGL Lichtsimulation kann abgeschaltet werden. Dies führt dazu, dass alle Dreiecke in der gleichen Farbe (Weiß) dargestellt werden. Ist die Lichtsimulation angeschaltet, wird die Farbe angepasst, als würde der Mesh von einer Lichtquelle angestrahlt.
- **Backface Culling**
Ist Backface Culling angeschaltet, so werden die Rückseiten der Dreiecke des Mesh nicht angezeigt. Der Mesh erscheint so von innen durchsichtig. Da weniger Dreiecke angezeigt werden müssen, führt dies zu einer höheren Performanz.
- **Koordinatensystem**
Ein einfaches Koordinatensystem aus Pfeilen in den Farben Rot, Grün,

und Blau für die X, Y, und Z Achsen kann angezeigt werden.

- **Einheitswürfel**

Es können $6 \times 6 \times 6$ Würfel mit der Kantenlänge 1 um die Kamera angezeigt werden. Dies ist nützlich, um zu verifizieren, dass die Vertices der Dreiecke des Mesh auf den Kanten der Würfel liegen.

- **Screenshot**

Ein Screenshot der derzeitigen Ansicht kann im Verzeichnis ‚screenshots‘ (im Arbeitsverzeichnis des Programms) abgelegt werden.

Die Tastaturbelegung für die Funktionen sind wie folgt:

Taste	Funktion An/Aus	Taste	Funktion An/Aus
	Drahtgittermodell		Einheitswürfel
	Lichtsimation		Normalenvektoren
	Backface Culling		Screenshot
	Koordinatensystem		

Zusätzlich kann die Ausführung des Marching Cubes Algorithmus über die von MCRunner gebotenen Funktionen gesteuert werden. Die Ausführung lässt sich pausieren bzw. fortsetzen. Des weiteren kann die Ausführung (wenn pausiert) fortgesetzt werden, bis das nächste mal ein Mesh an MeshView3D weitergegeben wird. Je nachdem, wie MCRunner konfiguriert ist, wird demnach der Teil des Mesh zur derzeitigen Anzeige hinzugefügt, welcher den nächsten Würfel bzw. die nächste Scheibe darstellt.

Taste	Funktion
	Marching Cubes pausieren/fortsetzen
	Marching Cubes für einen Cube/Slice fortsetzen

Mithilfe dieser Funktionen kann der Ablauf des Algorithmus komfortabel beobachtet werden.

3.5.2 Navigation

Zur Navigation wird die typische Tastenbelegung verwendet, wie sie in vielen 3D Programmen und Videospielen zur Bewegung der Kamera eingesetzt wird.

Taste	Kamera	Taste	Kamera
	Vorwärts		Nach Oben schwenken
	Rückwärts		Nach Unten schwenken
	Links		Nach Links schwenken
	Rechts		Nach Rechts schwenken
	Rollen gegen Uhrzeigersinn		
	Rollen mit Uhrzeigersinn		

Zusätzlich zu den Pfeiltasten kann die Kamera mit der Maus (bei gedrückter linker Maustaste) geschwenkt werden.

Kapitel 4

Anwendungen

Im folgenden Kapitel soll zuerst die Implementierung des Algorithmus auf ihre Laufzeit und Speicherverbrauch untersucht werden. Dann sollen einige Beispiele gegeben werden, wie sich die Konfigurationsparameter auf den Mesh auswirken. Als Datenquelle wurde ein Beispieldatensatz von ¹ verwendet. Der Datensatz ist die CT Aufnahme eines Paares menschlicher Füße.

Die Größe des Datensatzes beträgt $512 \times 512 \times 250$ `float` Werte. Das System, auf dem die folgenden Werte genommen wurden, hat einen AMD Phenom II X4 965 Prozessor, 20GB RAM und eine ATI Radeon HD 5800 Serien Grafikkarte. Zur Ausführung des Algorithmus wird Java in der Version 1.8.0_20 und die ‚Java HotSpot(TM) 64-Bit Server‘ Virtuelle Maschine verwendet. Diese wird mit den Optionen `-Xms512m -Xmx4g` konfiguriert, so dass der Virtuellen Maschine anfangs 512MB und maximal 4GB Heap Speicher zur Verfügung steht.

4.1 Profiling der Anwendung

Zuerst soll die Laufzeit des Algorithmus in Abhängigkeit von der eingestellten Grid Size betrachtet werden. Dazu wird der `MCRRunner` ohne GUI oder 3D Ansicht mit Grid Size Werten aus dem Intervall $[1, 10]$ ausgeführt.

Zu erkennen ist, dass die Laufzeiten für unterschiedliche Schwellwerte sehr ähnlich sind. Durch die Erhöhung der Grid Size sinkt die Laufzeit rapide ab.

¹<http://www.osirix-viewer.com/datasets/>

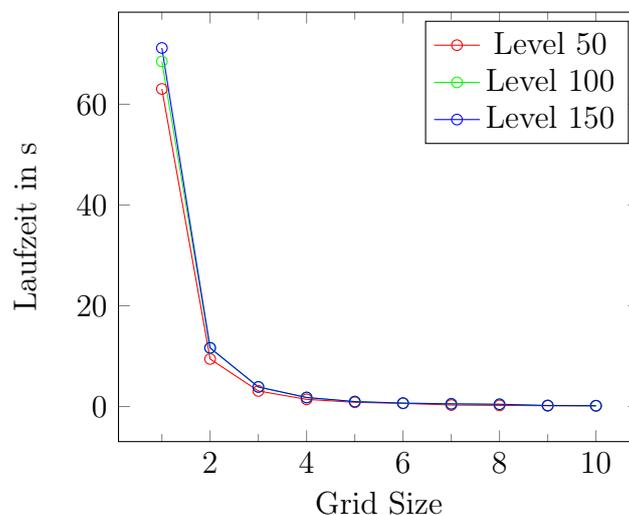


Abbildung 4.1: Laufzeit in Abhängigkeit von Grid Size mit ArrayVolume

Für den Datensatz in 4.1 wurde das `ArrayVolume` als Datenquelle verwendet.

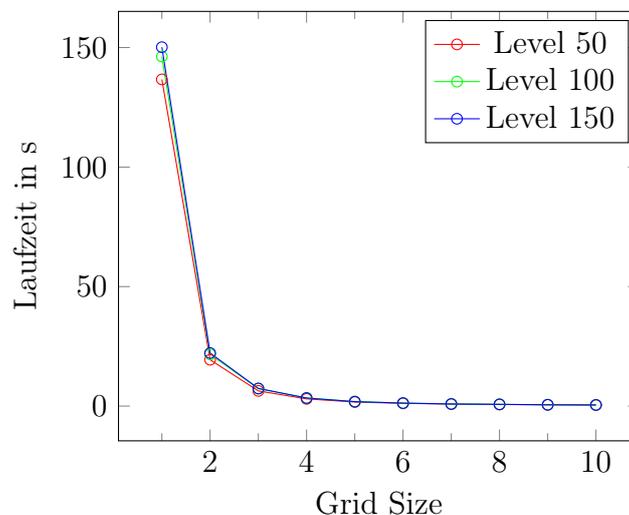


Abbildung 4.2: Laufzeit in Abhängigkeit von Grid Size mit CachedVolume

Wird das `Cached Volume` verwendet, steigt die Laufzeit durch die Festplattenzugriffe während der Ausführung des Algorithmus. Der Laufzeitgewinn ist in beiden Fällen sehr ähnlich für die verschiedenen Level. Dies liegt nahe, da die vom Algorithmus bearbeitete Anzahl von Würfeln nicht von dem Schwellwert, sondern nur von der Grid Size abhängt.

4.2 Verschieden dichte Objektanteile

Eine der interessantesten Anwendungen des Programms ist es, verschiedene Oberflächen aus demselben Datensatz zu extrahieren. Das in der Einleitung beschriebene Volumen wurde dazu mit verschiedenen Schwellwerten an den Algorithmus übergeben. Die resultierenden Oberflächen sind in Abb. 4.3 zu sehen.

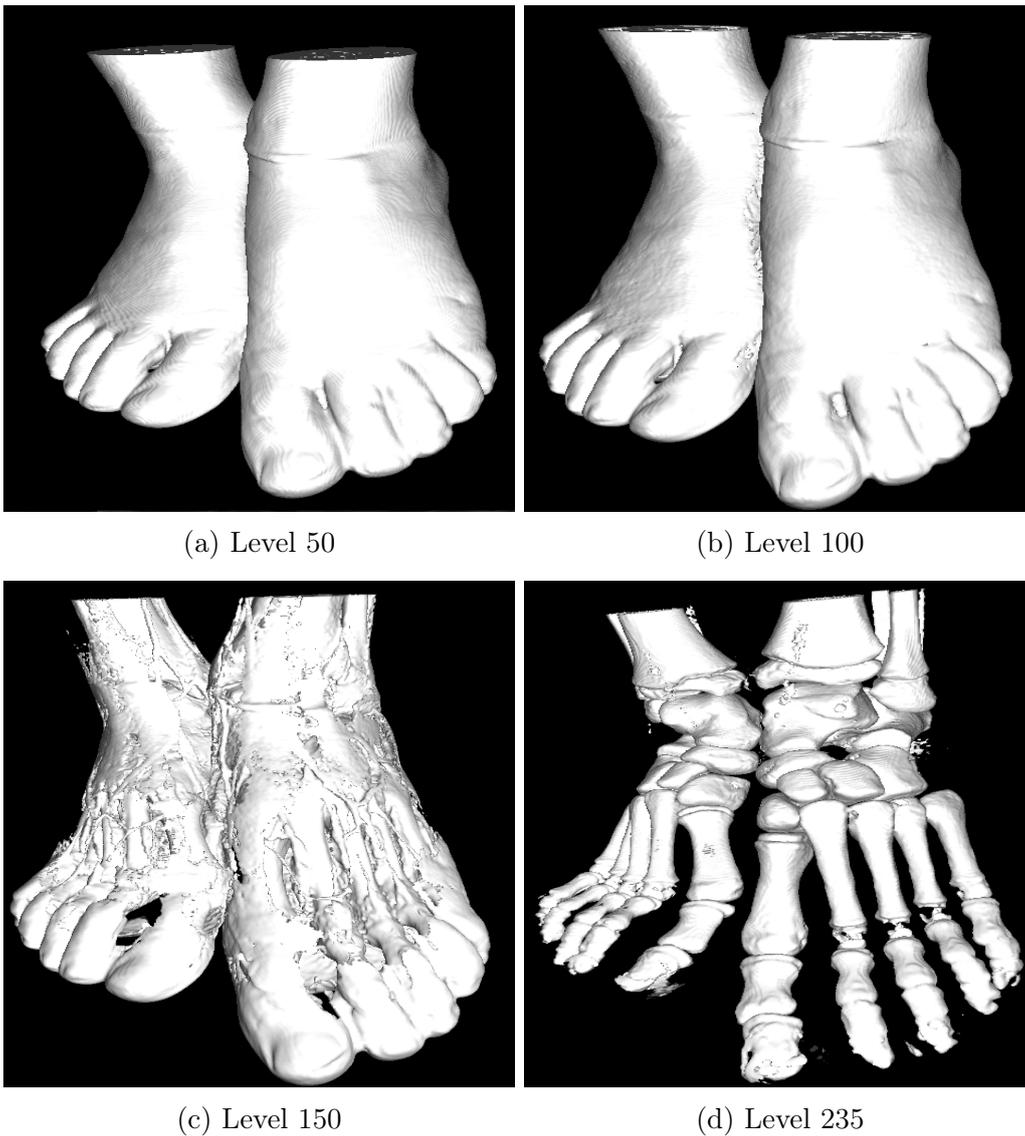


Abbildung 4.3: Verschiedene extrahierte Oberflächen

Mit dem Schwellwert 50 kann, wie in 4.3a zu sehen ist, die Haut als Oberfläche extrahiert werden. Sehr ähnlich sieht die Oberfläche aus, die aus dem Schwellwert 100 resultiert. Hier sind die Zehen etwas deutlicher zu sehen. Bei diesem Level bilden sich allerdings viele kleine Artefakte im Innern der Füße. Diese führen dazu, dass der Mesh aus sehr vielen Dreiecken (mehrere Millionen) besteht. Ein solcher Mesh müsste nachbearbeitet werden, ehe er weiterverwendet werden kann.



Abbildung 4.4: Artefakte bei Level 100

Bei Schwellwert 150 werden größtenteils die Knochen der Füße als solide betrachtet. Erst bei einem Schwellwert von 235 stehen die Knochen alleine da (wie in Abb. 4.3d zu sehen ist).

4.3 Einfluss der Grid Size auf den Mesh

Der Einfluss der Grid Size auf die Qualität des Mesh wurde bereits in 3.4.2 erläutert. Hier soll dieser Einfluss anhand einiger Beispiele illustriert werden. Es wurde der Testdatensatz mit Schwellwert 235 (wie er mit Grid Size 1 in Abb. 4.3d zu sehen ist) und den Größen 2, 4, 8, und 16 an den Algorithmus weitergegeben.

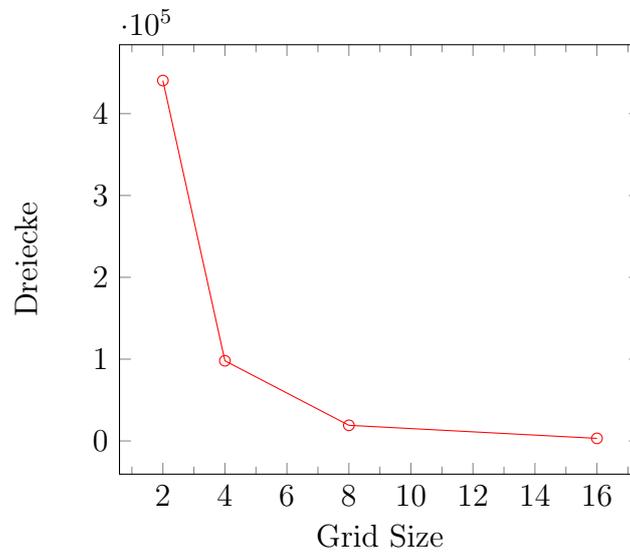
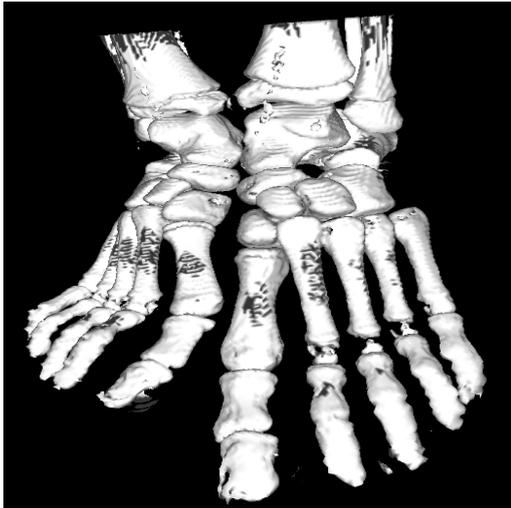
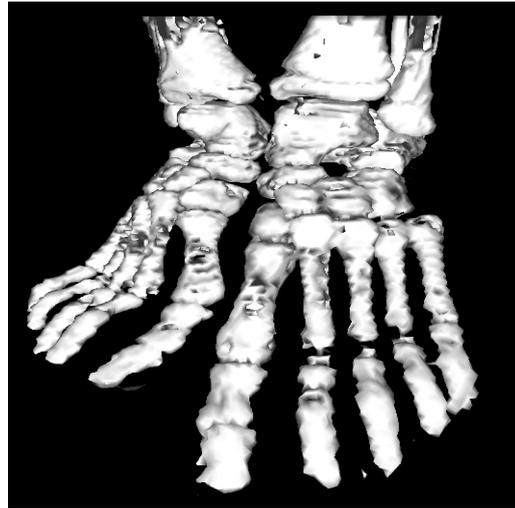


Abbildung 4.5: Anzahl der Dreiecke in Abhängigkeit von Grid Size

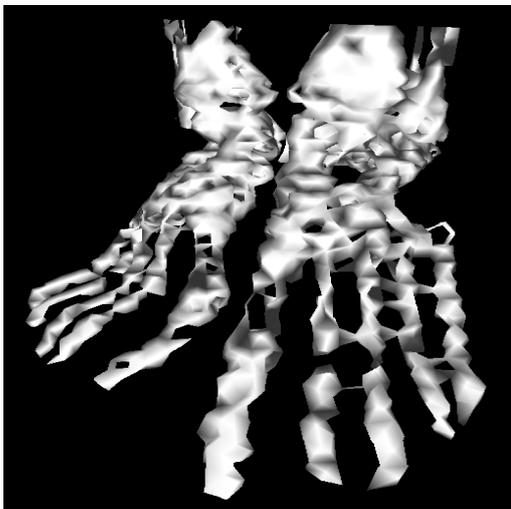
Es ist deutlich zu erkennen, wie die Anzahl der Dreiecke immer niedriger wird. Während sie in Abb. 4.6a noch 440488 beträgt, sinkt sie bei Grid Size vier schon auf 98012 ab. Die Abbildungen 4.6c und 4.6d zeigen Meshes mit 19140 bzw. 3172 Dreiecken.



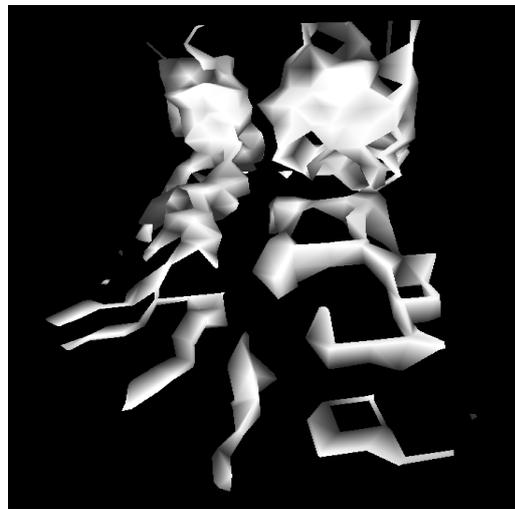
(a) Grid Size 2



(b) Grid Size 4



(c) Grid Size 8



(d) Grid Size 16

Abbildung 4.6: Verschiedene extrahierte Oberflächen

Kapitel 5

Mögliche Erweiterungen

Während der Entwicklung von V8 haben sich mehrere mögliche Erweiterungen ergeben. Die Implementierung dieser Features könnte der Bestand einer anderen Arbeit sein.

- **Verwendung von Shader Programmen**
Die derzeitige Implementation von `MeshView3D` verwendet die veraltete Fixed-Function-Pipeline von OpenGL. Der Mesh wird mithilfe von Vertex Buffer Objekten (mittels der `GL_ARB_vertex_buffer_object` Erweiterung) und indiziertem Rendering per `glDrawElements` angezeigt. Dadurch wird eine gute Performanz erreicht. Mithilfe moderner OpenGL Funktionalität (Vertex- und Fragmentshader) könnte diese noch deutlich erhöht werden. Die visuelle Qualität des Mesh kann so auch massiv verbessert werden.
- **Beliebige Grid Size**
Die Grid Size kann nur als natürliche Zahl spezifiziert werden. Vorstellbar wäre allerdings auch eine Grid Size von beispielsweise 0,5. Die zusätzlich benötigten Volumenwerte würden mithilfe der benachbarten Werte des Datensatzes linear interpoliert werden.
- **MetaBall Volume Designer ausbauen**
Der MetaBall Volume Designer (siehe Abb. 3.7) könnte durch eine Auflistung der derzeitig im Volumen enthaltenen MetaBall Instanzen erweitert werden. Nützlich wäre auch eine Anzeige der Positionen dieser als 3D Plot oder als zwei 2D Plots ($X \times Y$ und $X \times Z$ Projektion). Diese könnten als JavaFX `ScatterChart` implementiert werden und so sogar interaktiv (Veränderung der Position mit der Maus, Intensität per Mousrad) sein.

- **Zusätzliche Input Formate**

V8 beschränkt sich im Moment auf `.dcm` als Datenformat. Der Marching Cubes Algorithmus kann allerdings mit beliebigen Volumendaten arbeiten. Es existieren Dateiformate wie beispielsweise das ‚BINVOX‘¹ Format, um solche Daten zu speichern. Das Importieren solcher Formate wäre wünschenswert.

Des Weiteren könnte der in Kapitel 2 erwähnte ‚Dual Marching Cubes‘ Algorithmus über die bereits vorhandene GUI gesteuert werden. Dieser ist allerdings wesentlich komplexer als der Marching Cubes Algorithmus. Seine Implementierung würde alleine schon den Rahmen einer Bachelorarbeit übersteigen.

¹<http://www.cs.princeton.edu/~min/binvox/binvox.html>

Kapitel 6

Zusammenfassung

Um Volumendaten, die beispielsweise von medizinischen Geräten wie Computertomographen produziert werden, effizient bearbeiten und anzeigen zu können, müssen diese in Oberflächengrafiken umgewandelt werden. Für diese Aufgabe wird oft der Marching Cubes Algorithmus oder andere auf ihm basierende Algorithmen verwendet.

Ziel dieser Arbeit ist die Erläuterung und Implementierung des Algorithmus, die Erstellung einer GUI, die es ermöglicht, medizinische Volumendaten an den Algorithmus weiterzugeben, und eine hardwarebeschleunigte Anzeige der resultierenden Oberfläche. Alle diese Ziele wurden mit der Entwicklung des Programms V8 (abgeleitet von den acht Voxeln in einem Würfel des Marching Cubes Algorithmus) erreicht.

Hierfür wurde zuerst die Funktionsweise des Algorithmus, der in [LC87] vorgestellt wurde, erläutert und in der Klasse `MCRunner` implementiert. Sie und die anderen Klassen im Paket `model.mc_alg` stellen eine eigenständige und wiederverwendbare Implementierung des Algorithmus dar, die keine weiteren Abhängigkeiten neben der Java Klassenbibliothek hat. Zusätzlich wurde der Marching Squares Algorithmus implementiert und kann ebenso eigenständig verwendet werden.

Zur komfortablen Verwendung des Algorithmus wurde eine JavaFX GUI implementiert, die die Konfiguration erleichtert und Werkzeuge bietet, um einen guten Schwellenwert zu wählen. Schließlich existiert mittels der Klasse `MeshView3D` eine durch OpenGL hardwarebeschleunigte Ansicht der extrahierten Oberfläche.

Literatur

- [Bli82] James F. Blinn. »A Generalization of Algebraic Surface Drawing«. In: *ACM Trans. Graph.* 1.3 (Juli 1982), S. 235–256. ISSN: 0730-0301. DOI: 10.1145/357306.357310. URL: <http://doi.acm.org/10.1145/357306.357310>.
- [LC87] William E. Lorensen und Harvey E. Cline. »Marching Cubes: A High Resolution 3D Surface Construction Algorithm«. In: *SIG-GRAPH Comput. Graph.* 21.4 (Aug. 1987), S. 163–169. ISSN: 0097-8930. DOI: 10.1145/37402.37422. URL: <http://doi.acm.org/10.1145/37402.37422>.
- [NH91] Gregory M. Nielson und Bernd Hamann. »The Asymptotic Decider: Resolving the Ambiguity in Marching Cubes«. In: *Proceedings of the 2Nd Conference on Visualization '91*. VIS '91. San Diego, California: IEEE Computer Society Press, 1991, S. 83–91. ISBN: 0-8186-2245-8. URL: <http://dl.acm.org/citation.cfm?id=949607.949621>.
- [Nie03] Gregory M. Nielson. »On marching cubes«. In: *Visualization and Computer Graphics, IEEE Transactions on* 9.3 (2003), S. 283–297.
- [SW04] Scott Schaefer und Joe D. Warren. »Dual Marching Cubes: Primal Contouring of Dual Grids.« In: *Pacific Conference on Computer Graphics and Applications*. IEEE Computer Society, 2004, S. 70–76. ISBN: 0-7695-2234-3. URL: <http://dblp.uni-trier.de/db/conf/pg/pg2004.html#SchaeferW04>.

Glossar

CT Abk. für Computertomograph. 4, 30

DICOM Abk. für Digital Imaging and Communications in Medicine. 19

GUI grafische Benutzeroberfläche. 12, 14, 37, 38

Mesh Untereinander mit Kanten verbundene Punkte, die die Gestalt eines Polyeders definieren. 2, 4, 30

Voxel Gitterpunkt in einem 3D Gitter, entspricht dem Pixel aus der 2D Grafik. 3

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig verfasst habe und keine anderen als die angegebenen Quellen als Hilfsmittel benutzt habe.

Passau, den 4. Oktober 2014

Georg Seibt