Fachhochschul-Bachelorstudiengang
SOFTWARE ENGINEERING
A-4232 Hagenberg, Austria

Bachelorarbeit

zur Erlangung des akademischen Grades
Bachelor of Science in Engineering

Eingereicht von

Peter Paul Ortner

Hagenberg, September 2015

software engineering
IKM Fakultät Hagenberg

Inhalt

Eidesstattliche Erklärung

Ich erkläre eidesstattlich, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen nicht benutzt und die den benutzten Quellen entnommenen Stellen als solche gekennzeichnet habe. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt.

Datum, 07.09.2015 Unterschrift

Fachhochschul-Bachelorstudiengang
SOFTWARE ENGINEERING
A-4232 Hagenberg, Austria

3D Modellierung von Oberflächen mittels Marching Cubes Algorithmus und generische Darstellung mittels OpenGL

Bachelorarbeit
Teil 1

zur Erlangung des akademischen Grades
Bachelor of Science in Engineering

Eingereicht von

Peter Paul Ortner

Begutachter: FH-Prof. DI Dr. Werner Backfrieder

Hagenberg, September 2015

Inhaltsverzeichnis

Kurzfassung

In der computergestützten Bildverarbeitung gibt es eine breite Palette von Möglichkeiten für die Darstellung von dreidimensionalen Objekten. Die wohl am weitesten verbreitete Darstellungsform ist die polygonale Darstellung. Diese Form der Aufbereitung zerlegt ein gegebenes Objekte in Dreiecke.

Ein weiteres Verfahren ist die Methode der Modellierung aus einem Bildvolumen in ein Voxelmodell. Jedoch birgt diese, im Bezug auf die digitale Verarbeitung, einige Nachteile gegenüber der polygonalen Darstellungsform. Vordergründige Probleme hierbei sind der vergleichsweise hohe Speicherverbrauch der Modelle, die Visualisierung benötigt länger und Objektmanipulationen erweisen sich als schwieriger.

Da in der Medizin im Bereich der bildgebenden Systeme, wie der Computertomografie, von Natur aus solche Voxel-Modelle erzeugt werden, besteht die Anforderung, auch diese nach Möglichkeit schnell und Aussagekräftig darzustellen.

In dieser Arbeit wurde, im Bezug auf die oben erläuterte Problematik, der Marching Cubes Algorithmus betrachtet. Dieser Algorithmus ermöglicht es ein Voxelmodell in eine polygonale Darstellung zu überführen.

Diese Arbeit umfasst eine allgemeine Einführung in die Problematik, sowie eine Implementierung des Marching Cubes Algorithmus. Des Weiteren wird auf eine Möglichkeit zur Visualisierung via OpgenGL eingegangen.

Anhand der einfachen aber dennoch effizienten Implementierung des Algorithmus, lässt sich seine Funktionsweise gut verfolgen. Des Weiteren kann gut erkannt werden, dass die gezeigte einfachste Form ein großes Potential für Verbesserungen aufweist. Unter Anbetracht, dass bereits viele Erweiterungen der Grundform entwickelt wurden, ist die weite Verbreitung dieses Algorithmus nicht verwunderlich.

Abstract

Considering the computer based image processing there are multiple possibilities to represent three-dimensional objects. The most common way to illustrate these objects is the polygonal approach. This approach fragments an object into triangles.

An other possible procedure to model objects is to represent them as a voxel grid. But if we consider the ability to process this kind of representation we have to face some disadvantages. The main problems are: the model needs a comparatively high amount of disk space, it takes much longer to show the image and it is difficult to perform manipulations on the object.

In the field of Medical imaging such as computed tomography creates such voxel models, these should be presented quickly and meaningfully.

Considering the set of problems above, this thesis presents the "Marching Cubes Algorighm". This Algorithm enables us to transform voxel objects into a polygonal form.

This thesis gives an insight of the basic terms and provides an implementation of the Marching Cubes Algorithm.

On the base of the simple but efficient implementation it is easy to follow the functionality of the algorithm. The basic form shows the big potential of the algorithm for enhancements. Because of the fact that many of this possible enhancements are already developed, the wide spread of this algorithm is not surprising at all.

Kapitel 1

Einleitung

1.1 Aufgabenstellung

In der medizinischen Diagnostik wird im Gegensatz zu CAD-Konstruktionen die dreidimensionale Gestalt anatomischer Details aus Volumsbildern abgeleitet. Durch vorangegangene Segmentierung werden binäre Objekte erzeugt, d.h. das Objekt ist wie eine Lego-Figur aufgebaut. Mit dem Marching Cubes Algorithmus wird aus diesem binären Volumen eine Oberfläche, welche aus Dreiecken besteht, aufgebaut. Diese Oberfläche wird in einem binären STL-Format persistiert und anschließend mithilfe von generischem Rendering als 3D-Objekt dargestellt.

Anforderungen: C/C++ Implementierung des MC-Algorithmus (Matlab-Version vorhanden), Konversion in STL-Format, OpenGl Visualisierung.

1.2 Motivation

Da moderne Grafikchips auf die Darstellung von polygonalen Modellen ausgelegt sind, ist es sinnvoll, die aus der medizinischen Diagnostik erhaltenen Voxel-Modelle für spätere Verarbeitung in diese Form zu überführen. Ein weiterer Vorteil neben der vereinfachten Verarbeitung und Darstellung von Polygonen liegt in dem vergleichsweise geringen Speicherbedarf eines solchen Objektes.

1.3 Zielsetzung

Ziel dieser Arbeit ist es, die aus bildgebenden Verfahren der Medizin erhaltenden Voxel-Mengen mithilfe des Marching Cubes Algorithmus in eine polygonale Darstellung zu überführen. Als Input werden die Daten, welche

von dem Programm Analyze 7.5[1] erzeugt werden verwendet. Im Speziellen handelt es sich hierbei um die Formate Image (.img) und Header (.hdr). Die Voxel-Menge, welche in der Image-Datei abgelegt ist, wird ausgelesen und mithilfe des Marching Cubes Algorithmus in Polygone zerlegt. Nach erfolgreicher Umwandlung wird die erhaltene Datenmenge via OpenGL dargestellt. Des Weiteren soll das Modell als STL-Datei exportiert werden können. Die gesamte Umsetzung erfolgt in der Programmiersprache C++.

[1] https://rportal.mayo.edu/bir/

Kapitel 2

Grundlagen

In diesem Kapitel wird auf die zugrunde liegende Theorie bezüglich Bildverarbeitung, Marching Cubes, verwendete Dateiformate und OpgenGL eingegangen. Es wird ein grober Überblick über die wichtigsten Begrifflichkeiten der jeweiligen Bereiche geschaffen, um ein grundlegendes Verständnis der Materie zu gewährleisten. Da es sich jeweils um sehr große Bereiche handelt wurde darauf geachtet, nur das Wesentliche welches für das Verständnis dieser Arbeit notwendig ist zu erfassen.

2.1 Bildgebende Verfahren

> "Die Medizinische Bildverarbeitung hat das Ziel, medizinische Bilder und Bildfolgen zur Unterstützung der medizinischen Diagnostik und Therapie aufzubereiten, zu analysieren und zu visualisieren [Hadels, 2000]."

Die verschiedenen medizinischen Verfahren können grob in die Art der erzeugten Bilddaten eingeteilt werden:

- **Schnittbilder:** Verfahren wie Computertomografie, Magnetresonanztomografie oder Röntgentomografie erzeugen diese Art von Bildern.
- **Projektionsbilder:** Diese werden z.B. durch "klassisches" Röntgen erzeugt.
- **Oberflächenabbildungen:** Diese Bilder entstehen unter Andrem durch die Verwendung eines Rastertunnelmikroskops.

Das Hauptaugenmerk dieser Arbeit liegt auf der aus den tomographischen Verfahren erhaltenden Schnittbildern, welche als sogenannte Voxel-Modelle gespeichert werden. Ein vollständiges dreidimensionales Bild besteht aus mehreren solcher übereinandergelegten Schnittbildern.

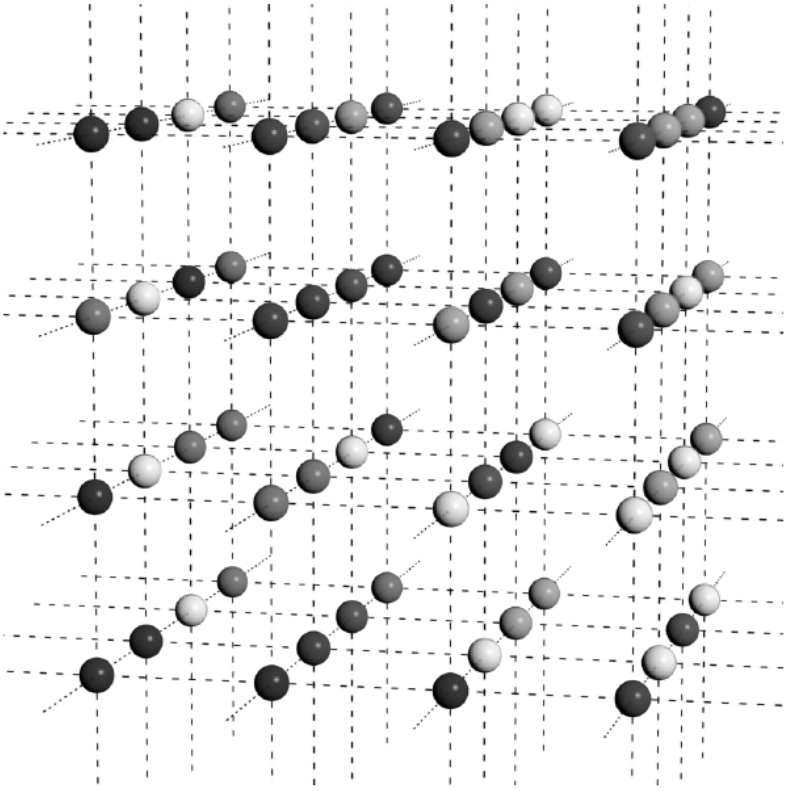

Abbildung 2.1: Ein Voxelgitter [Seibt, 2014].

2.2 Volumengrafik

Unter Volumengrafik versteht man in der Computergrafik die Darstellung von Objekten durch eine Menge von Punkten. Diese Punkte werden auch als Voxel bezeichnet und repräsentieren einen Punkt im Raum.

2.2.1 Voxel

Ein solches Voxel ist als ein einzelner Punkt in einem dreidimensionalen Objekt zu verstehen, welcher eine gewisse Intensität aufweist. Dieser Wert beschreibt die Eigenschaft des Objektes an diesem Punkt und ist essentiell um, wie im Falle dieser Arbeit, bei den tomographischen Verfahren in der Medizin die festeren von den weicheren Teilen eines Körpers zu unterscheiden. Ein mögliches Beispiel wären die unterschiedlichen Dichtewerte von Knochen und Gewebe im menschlichen Körper. In Abbildung 2.1 ist eine solche Voxel-Menge zu sehen. Die verschiedenen Grauwerte der einzelnen Bildpunkte stellen dabei die unterschiedlichen Intensitäten der Voxel dar.

2.3 Marching Cubes

Da der Marching Cubes Algorithmus das zentrale Element dieser Arbeit bildet, wird hier seine Grundform nochmals genau erörtert [Lorensen u. Cline, 1987]. Für weiterführende Betrachtungen und Verbesserungen wird am Ende im Abschnitt "Schwachpunkte" auf entsprechende Literatur verwiesen.

2.3.1 Formale Definition

> "Der Marching Cubes Algorithmus ist ein Algorithmus um eine Isofläche S_c eines Objektes, das in einem Skalarfeld $\varphi : \mathbb{R}^n \to \mathbb{R}$ beschrieben wird durch Dreiecke zu approximieren. Der Isowert c ϵ $\mathbb{R}$ beschreibt die gemeinsame Eigenschaft des Objektes wie z.B. gleiche Dichte, Temperatur oder emittierter Strahlung [Wollmann, 2013]."

Für eine Isofläche S_c, wird eine endliche Menge an Datenpunkten P $\subset$ $\mathbb{R}^n$ x $\mathbb{R}$ zur Approximation von S_c erzeugt [Hansen u. Johnson, 2005].

$$S_c := \{\vartheta \; \epsilon \; \mathbb{R}^n \mid \varphi(\vartheta) = c\} \tag{2.1}$$

In gängigen Implementierungen wird $n = 3$ gesetzt und φ auf ein dreidimensionales Gitter gesampled. Die resultierende Isofläche S_c wird durch einen Polygon Mesh angenähert [Seibt, 2014].

2.3.2 Funktionsweise

Wie der Name des Algorithmus bereits sagt, wird durch die Voxel-Menge "marschiert". Darunter ist das betrachten eines Teilausschnitts des Voxelgitters nach dem nächsten zu verstehen. Die Input-Menge des Algorithmus umfasst acht aneinander grenzende Punkte der Voxel-Menge, welche zusammen einen Würfel bilden sowie eines Schwellwerts für die Dichte. Nach erfolgreicher Verarbeitung wird zum nächste Würfel gewandert ("marschiert") bis die gesamte Datenmenge abgearbeitet wurde.

Vorbereitung

Wie bereits erwähnt wird der Algorithmus auf jeden einzelnen Würfel der gesamten Menge angewendet. Für ein besseres Verständnis zeigt Abbildung 2.2 einen solchen Würfel (rot) in einem Voxelgitter.

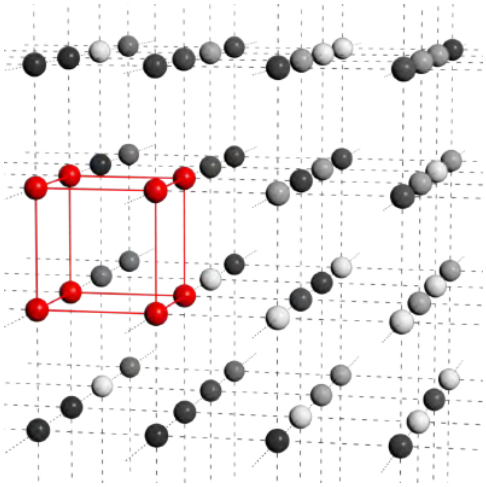

Abbildung 2.2: Ein Wüfel im Voxelgitter [Seibt, 2014].

In Abbildung 2.3 ist als erster Schritt des Algorithmus die Indizierung der Ecken und Kanten des Würfels für die spätere Verarbeitung zu sehen.

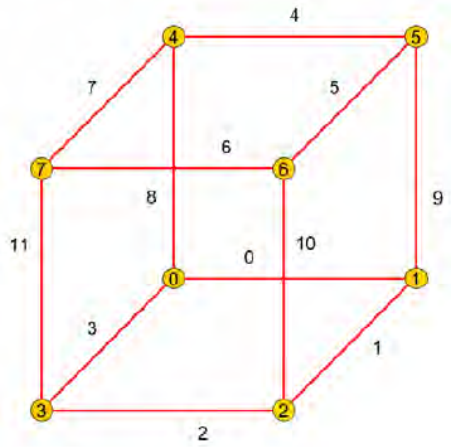

Abbildung 2.3: Indizierung eines Würfels [Seibt, 2014].

Jede Ecke des Würfels kann aufgrund seines Wertes als Solide bzw. Transparent klassifiziert werden. Unter Solide ist ein erkennbarer Bildpunkt zu verstehen. Ein transparenter Punkt gleicht hingegen der Hintergrundfarbe des Objektes. Folglich sind aufgrund der zwei möglichen Werte jeder Ecke $2^8 = 256$ unterschiedliche Konfigurationen der Eingabemenge möglich. Jede dieser Konfigurationen kann als ein 8 Bit Muster dargestellt werden, wobei gilt, dass jedes Bit i bei welchem der Wert d_i des dazugehörigen Voxel einen gewissen Schwellwert c überschreitet als binäre 1 interpretiert wird. Für die

Werte kleiner gleich des Schwellwertes wird eine Binäre 0 angenommen.

Wird dieses Bitmuster nun als natürliche Zahl betrachtet, ergibt sich eine sogenannter Würfelindex welcher einen Wert zwischen 0 und 255 aufweist. Dieser ist für die weitere Verarbeitung essentiell.

Aufgrund der Symmetrie eines Würfels können durch Rotation, Spiegelung und Inversion die Anzahl der 255 möglichen Anordnungen der Voxel auf 15 unterschiedliche Konfigurationen reduziert werden. Diese 15 Konfigurationen sind in Abbildung 2.4 zu sehen.

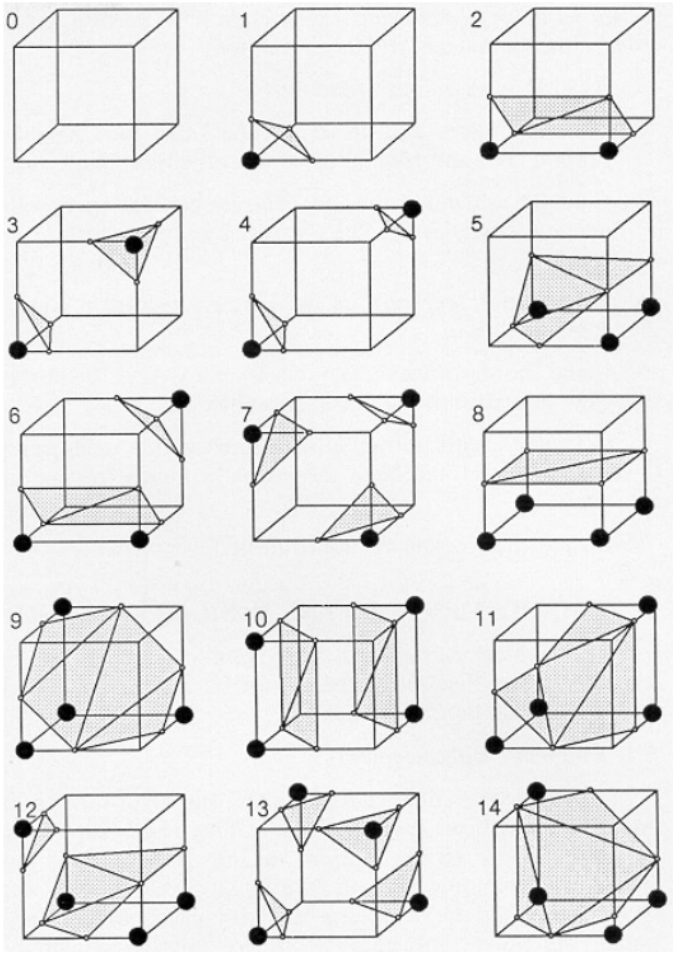

Abbildung 2.4: Marching Cubes Grundkonfigurationen [Lorensen u. Cline, 1987].

Verarbeitung

In diesem Abschnitt wird auf die Verarbeitungsschritte eines jeden Würfels aus dem Voxelgitter eingegangen. Der Ablauf lässt sich in sieben Schritte unterteilen [Lorensen u. Cline, 1987]:

1. Einlesen von vier[1] Scheiben in den Speicher.
2. Betrachten von zwei Scheiben und Bilden eines Würfels aus vier Nachbarn der ersten und weiteren vier Nachbarn der zweiten Scheibe.
3. Berechnen des Index des Würfels durch Vergleichen der Isowerte der Punkte mit dem Schwellwert.
4. Bestimmung der Liste der Dreiecksvertices anhand des Index (Abgleich mit Lookuptable).
5. Interpolation der Positionen der Dreiecksvertices.
6. Interpolation der Normalen an den Dreiecksvertices.
7. Speichern der resultierenden Dreiecke und Normalen.

Wie bereits im Abschnitt "Vorbereitung" erklärt, wird in den ersten drei Punkten ein Würfel aus dem Voxelgitter extrahiert. Im Anschluss wird dieser Indexiert und jede Ecke mit einem Iso-Schwellwert abgeglichen. Der dadurch erhaltene Index wird nun in Punkt 4 für die weitere Verarbeitung benötigt.

Punkt 4 beschreibt, dass für jede der 256 möglichen Kombinationen des Würfels die dazugehörigen Dreiecksvertices bestimmt werden müssen. Dies erfolgt durch Betrachtung zweier benachbarter Punkte des Würfels und deren Abgleich mit dem gegeben Schwellwert. Da eine erneute Auswertung für jeden Würfel äußerst rechenintensiv wäre, wird hierzu auf eine sogenannte Lookup-Tabelle zurückgegriffen, welcher für jeden der 256 Konfigurationen die vorberechneten Werte beinhaltet.

In Schritt 5 wird durch die Interpolation der Dreiecksverices eine Glättung der Oberfläche erreicht.

Punkt 6 dient dazu die Position der Dreiecke im Raum zu bestimmen. Für eine nähere Beschreibung der Berechnung siehe [Lorensen u. Cline, 1987].

Im letzten Schritt können nun die berechneten Dreiecke weiterverwendet werden. Im konkreten Fall heißt dies, dass das Objekt via OpenGL visualisiert und oder das Ergebnis als STereoLithography Datei persistieren wird.

[1] Im ersten Moment scheinen für den Algorithmus nur zwei Scheiben notwendig. Allerdings werden für die Berechnung des Gradienten der Würfel Veritces die zwei weiteren Scheiben benötigt [Lorensen u. Cline, 1987].

2.3.3 Probleme

Die Grundform des Marching Cubes Algorithmus hat einige Schwachpunkte. Im Laufe der Zeit wurden daher Methoden entwickelt diese Schwachstellen zu beheben. In diesem Abschnitt werden einige Probleme aufgezählt und auf Lösungen für diese verwiesen.

Uneindeutigkeit

Die Einteilung in die 15 Grundkonfigurationen wie in Abbildung 2.4 zu sehen, kann zu Uneindeutigkeiten bezüglich der Repräsentation führen. Bei nicht Behandlung dieser kann es zu "Löchern" in der Oberfläche des resultierenden Objektes kommen. Zur Behebung dieses Problems stützen sich moderne Implementierungen des Marching Cubes Algorithmus auf die sogenannte asymptotische Entscheidung[2] oder der trilinearen Interpolation der inneren Voxel[3].

Dreiecksdezimierung

Der Marching Cubes Algorithmus erzeugt von Natur aus eine sehr hohe Anzahl an Dreiecken. Dies wirkt sich negativ auf die Performanz des Algorithmus aus. Daher wurden Methoden entwickelt die es ermöglichen, die Anzahl der zu verarbeitenden Dreiecke zu minimieren [Schroeder u. a., 1992].

Glättung Dreiecksnetz

Der Punkt der Verarbeitung "Interpolation der Positionen der Dreiecksvertices" welcher im Kapitel zuvor erwähnt wurde, dient bereits der Glättung des Dreiecksnetzes, da der Marching Cubes Algorithmus ohne diesen Schritt eine sehr kantige Oberfläche erzeugen würde. Um eine weitere Verfeinerung zu erreichen, bietet sich hier der häufig verwendete Humphrey's Classes Algorithmus an [Vollmer u. a., 2001] .

Alternative Datenstruktur

Da ein Volumen Datensatz verhältnismäßig große Ausmaße annehmen kann, bietet sich ein sogenannter Octree für eine vereinfachte Datenhaltung an. In [Wilhelms u. Gelder, 2000] ist die Verwendung einer solchen Struktur zur Generierung von Isoflächen dargestellt.

[2] [Nielson u. Hamann, 1991]
[3] [Nielson, 2003]

2.4 Dateiformate

Die zu dieser Arbeit herangezogenen Dateiformate sind einerseits die von dem Softwarepaket Analyze[4] verwendeten Image und Header Files sowie die sogenannte STereoLithography-Schnittstelle.

2.4.1 Image File (.img)

Diese Datei ist vergleichsweise einfach aufgebaut und enthält ein Objekt bestehend aus (normalerweise) unkomprimierten Pixel Daten [Mayo, 2015]. Jedes Pixel repräsentiert eine Voxel mit dem jeweils dazugehörigen Wert. Das gesamte Objekt kann somit in ein dreidimensionales Array eingelesen werden.

2.4.2 Header File (.hdr)

Diese Datei beschreibt die Ausmaße der Pixel-Datei (.img) sowie ihre Historie [Mayo, 2015].

Die genaue Struktur ist in drei Teilbereiche aufgeteilt [Mayo, 2015]. Der erste Teil ist der sogenannte "header key" und beinhaltet allgemeine Informationen bezüglich der Datei. Der zweite Teil beinhaltet Informationen bezüglich der Dimension der Image-Datei. Der dritte und letzte Abschnitt hält Informationen bezüglich der Historie.

Die jeweiligen Teile sind in den Abbildungen "header key" 2.1, "Image-Datei" 2.3 und "Historie" 2.2 als C-Struktur angeführt.

Programm 2.1: Header key als C-Struktur [Mayo, 2015]

```
 1 struct header_key /* header key */
 2 { /* off + size */
 3   int sizeof_hdr /* 0 + 4 */
 4   char data_type[10]; /* 4 + 10 */
 5   char db_name[18]; /* 14 + 18 */
 6   int extents; /* 32 + 4 */
 7   short int session_error; /* 36 + 2 */
 8   char regular; /* 38 + 1 */
 9   char hkey_un0; /* 39 + 1 */
10 }; /* total=40 bytes */
11
```

[4] https://rportal.mayo.edu/bir/

Programm 2.2: Data history als C-Struktur [Mayo, 2015]

```
1 struct data_history
2 { /* off + size */
3   char descrip[80]; /* 0 + 80 */
4   char aux_file[24]; /* 80 + 24 */
5   char orient; /* 104 + 1 */
6   char originator[10]; /* 105 + 10 */
7   char generated[10]; /* 115 + 10 */
8   char scannum[10]; /* 125 + 10 */
9   char patient_id[10]; /* 135 + 10 */
10  char exp_date[10]; /* 145 + 10 */
11  char exp_time[10]; /* 155 + 10 */
12  char hist_un0[3]; /* 165 + 3 */
13  int views /* 168 + 4 */
14  int vols_added; /* 172 + 4 */
15  int start_field; /* 176 + 4 */
16  int field_skip; /* 180 + 4 */
17  int omax, omin; /* 184 + 8 */
18  int smax, smin; /* 192 + 8 */
19 };
20
```

Programm 2.3: Image Dimension als C-Struktur [Mayo, 2015]

```
1 struct image_dimension
2 { /* off + size */
3   short int dim[8]; /* 0 + 16 */
4   short int unused8; /* 16 + 2 */
5   short int unused9; /* 18 + 2 */
6   short int unused10; /* 20 + 2 */
7   short int unused11; /* 22 + 2 */
8   short int unused12; /* 24 + 2 */
9   short int unused13; /* 26 + 2 */
10  short int unused14; /* 28 + 2 */
11  short int datatype; /* 30 + 2 */
12  short int bitpix; /* 32 + 2 */
13  short int dim_un0; /* 34 + 2 */
14  float pixdim[8]; /* 36 + 32 */
15  /*
16  pixdim[]  specifies  the  voxel  dimensitons:
17  pixdim[1] - voxel width
18  pixdim[2] - voxel height
19  pixdim[3] -  interslice  distance
20   ... etc
21  */
22  float vox_offset; /* 68 + 4 */
23  float funused1; /* 72 + 4 */
24  float funused2; /* 76 + 4 */
25  float funused3; /* 80 + 4 */
26  float cal_max; /* 84 + 4 */
27  float cal_min; /* 88 + 4 */
28  float compressed; /* 92 + 4 */
29  float verified; /* 96 + 4 */
30  int glmax,glmin; /* 100 + 8 */
31 }; /* total=108 bytes */
32
```

2.4.3 STereoLithography (.stl)

> "The STL (STereoLithography) file format, as developed by 3D Systems, has been widely used by most Rapid Prototyping (RP) systems and is supported by all major computer-aided design (CAD) systems [Chua u. a., 1997]."

Eine STL-Datei besteht im Prinzip aus einer Liste von Dreiecken. Jedes Dreieck wird durch seine drei Eckpunkte im Raum sowie durch seinen Normalvektor beschrieben. Dies führt folglich zu einer Summe von 12 Werten pro Dreieck[5].

Zum allgemeinen Verständnis des Aufbaus bietet sich die ASCII-Darstellung der Datei an:

solid name

$$\left\{\begin{array}{l}\textbf{facet normal } n_i\ n_j\ n_k \\ \quad \textbf{outer loop} \\ \quad\quad \textbf{vetex } v1_x\ v1_y\ v1_z \\ \quad\quad \textbf{vetex } v2_x\ v2_y\ v2_z \\ \quad\quad \textbf{vetex } v3_x\ v3_y\ v3_z \\ \quad \textbf{endloop} \\ \textbf{endfacet}\end{array}\right\}^{+}$$

endsolid name

Für die Umsetzung im Programm ist der Binäre Aufbau der Datei von Interesse. Dieser beschreibt die Abbildung der Datei im Speicher[Fabbers, 2015].

Bytes	Data type	Description
80	ASCII	Header. No data significance
4	unsingend long integer	Number of facets in file
DO{		
4	float	i for normal
4	float	j
4	float	k

[5] x, y und z Postion der drei Eckpunkte sowie die drei Werte des Normalvektors

4	float	x for vertex 1
4	float	y
4	float	z
4	float	x for vertex 2
4	float	y
4	float	z
4	float	x for vertex 3
4	float	y
4	float	z
2	unsigend integer	Attribute byte count

}UNTIL(end of data)

2.5 Computergrafik

Computergrafik beschreibt das computergestützte Erstellen und Verarbeiten von Grafiken [Shirley u. Marschner, 2009]. In dieser Arbeit wird auf die Verarbeitung und insbesondere auf die Darstellung von dreidimensionalen Objekten als Polygon-Menge zurückgegriffen. Zu diesem Zweck bieten sich diverse Programmierschnittstellen wie OpenGL[6], Direct3D[7] oder AMD Mantle[8] an, welche für Grafikausgaben genutzt werden können. Aufgrund der Aufgabenstellung dieser Arbeit findet OpenGL Verwendung.

2.5.1 OpenGL

> "OpenGL (for "Open Graphics Library") is a software interface to graphics hardware. The interface consists of a set of several hundred procedures and functions that allow a programmer to specify the objects and operations involved in producing high-quality graphical images, specifically color images of three-dimensional objects [Segal u. Akeley, 2009]."

OpenGL ermöglicht eine verhältnismäßig einfache Plattform unabhängige Grafikprogrammierung. Da es sich um eine reine Grafikbibliothek handelt kümmert sich OpenGL nicht um die Verwaltung von Zeichenoberflächen, Renderkontexten oder weitere Puffer. Um OpenGL im Weiteren vernünftig

[6] https://www.opengl.org/

[7] https://msdn.microsoft.com/en-us/library/windows/desktop/bb153256(v=vs.85).aspx

[8] http://www.amd.com/de-de/innovations/software-technologies/technologies-gaming/mantle

mit einem Betriebssystem zu verwenden existieren daher verschiedene Bibliotheken. Die in dieser Arbeit verwendete Bibliothek ist QT. Die Gründe für diese Entscheidung sind die Unabhängigkeit bezüglich Betriebssystem, die Aktualität der Bibliothek sowie die hohe Verbreitung dieser.

Kapitel 3

Implementierung

In dieser Arbeit wurde eine in C++ geschriebenes Applikation entwickelt, welche den Marching Cubes Algorithmus auf eine Voxelimage anwendet und das Resultat via OpgenGL visualisiert. Des Weiteren ist es möglich, das verarbeitete Model als STereoLithography (.stl) Datei zu exportieren.

3.1 Marching Cubes

Der Marching Cubes Algorithmus ist der Kern der entwickelten Applikation. Er ermöglicht die Umrechnung der gegebenen Voxel-Datenmenge in eine polygonale Darstellung, welche sich im späteren Verlauf vergleichsweise einfach darstellen lässt.

3.1.1 Allgemein

Die Implementierung ist eine angepasst Version der von Paul Bourke bereitgestellten Umsetzung [Bourke, 1994]. Die wesentlichen Änderungen sind die Auslagerung der Funktionen in eine eigenen Klasse und das Verwenden anderer Datenstrukturen. Durch die Umstellung auf STL-Behälter und der daraus folgende Verzicht auf C-Strukturen welche zur Laufzeit immer neuen Speicher anfordern, konnte die Geschwindigkeit immens gesteigert werden.

3.1.2 Schnittstelle (Klasse)

Diagramm 3.1: UML-Diagramm der marchingCubes Klasse

«Class» marchingCubes
- mData : d3Buffer - mIsolevel : short int - mResult : vector<TRIANGLE> - generateSlice(offset : int, slice . int) : void - polygoniseCube(grid : GRIDCELL, ref triangles : vector <TRIANGLE>) : void - vertexInterp(p1 : XYZ, p2 : XYZ, valp1 : float, valp2 : float) : XYZ - calcNormal(ref tri : TRIANGLE) : void
+ setAllData(data : d3Buffer, isolevel : short int) : void + perform(offset : int, slice = -1: int) : void + getResult() : vector<TRIANGLE> + generateStlFile(path : string) : bool + setIsolevel(isolevel : short int) : void

Datentypen

Neben den üblichen Datentypen wurden zur leichteren Verarbeitung komplexere Strukturen verwendet.

- **XYZ** beschreibt einen Punkt im dreidimensionalen Raum.
- **GRIDCELL** ist die Repräsentation eines Voxel-Würfels.
- **TRIANGLE** repräsentiert ein Dreieck mithilfe seiner drei Eckpunkte im Raum und seiner Normalen.
- **3dBuffer** bildet ein Voxelgitter als dreidimensionalen Vektor im Speicher ab.

Funktionen

Um ein grundlegendes Verständnis zu bieten, werden hier die öffentlichen Methoden der Klasse kurz beschrieben.

- **setAllData:** Da während der Laufzeit des Programmes nur eine Instanz der *marchingCubes*-Klasse instanziiert wird, ist es notwendig diese, wenn nötig, mit neuen Werten zu initialisieren. Es wird das gesamte Voxelgitter (data) sowie der zu verwendende Schwellwert (isolevel) für die Berechnung mitgegeben.

- **perform:** Diese Funktion führt den Marching Cubes Algorithmus auf dem zuvor via *setAllData* gesetzten Voxelgitter aus. Es bestehen zwei Möglichkeiten des Aufrufs. Zum Einen mit beiden Parametern und zum Anderen mit nur dem ersten (offset) Parameter. Bei der ersten Variante wird nur eine Scheibe des Voxelgitters betrachtet, wobei der erste Parameter die Dicke der Scheibe und der zweite die Position der Scheibe im Gitter kennzeichnet. Bei der zweiten Variante wird der gesamte Input betrachtet, wobei Parameter "offset" die Schrittweite pro Würfel angibt.
- **getResult:** Hier kann nach dem Durchlaufen von *perform* das Ergebnis des Algorithmus abgegriffen werden.
- **setIsolevel:** Hier kann der Schwellwert der Voxeln für den Algorithmus nachträglich verändert werden.
- **generateStl:** Das Ausführen dieser Funktion persistiert das Ergebnis des Marching Cubes Algorithmus als .stl Datei.

Verwendung

Im Programm 3.1 ist die Verwendung der Klasse exemplarisch dargestellt. Die Funktionen *getRawData*, *getIsolevel*, *getOffset* und *getSlice* dienen zur Veranschaulichung und sind nicht im beigelegten Programmcode enthalten.

Programm 3.1: Verwendung der marchingCubes Klasse

```
1   int main(){
2     marchingCubes *mc = new marchingCubes();
3     mc->setAllData(getRawData(),   // 3d vector
4              getIsolevel()); // int
5     mc->perform(getOffset());
6     mc->perform(getOffset(), // slice thicknes
7           getSlice()); // slice number
8     delete(mc);
9     return 0;
10  }
11
```

1. Instanzieren einer *marchingCubes* Instanz.
2. Setzen der zu verarbeitenden Voxelmenge sowie den zu verwendenden Schwellwert.
3. Anwenden des Algorithmus auf die zuvor gesetzte Voxelmenge.
4. Erneutes Anwenden des Algorithmus allerdings mit den Parametern für das Slicing.
5. Freigeben des allokierten Speichers.

3.2 File Formate

Wie bereits im Kapitel 2.4 erwähnt wurden für die Umsetzung die Dateiformate von Analyze 7.5[1] sowie das STereoLithography (.stl) Format verwendet.

3.2.1 Allgemein

Um eine unnötige Komplexität zu vermeiden, wurden für die Analyze-Formate nur lesender und für das STereoLithography Format nur schreibende Zugriff implementiert. Hier bieten sich für spätere Erweiterungen viele Möglichkeiten bezüglich Import/Export. Auch die Unterstützung von weiteren Formaten wäre hier eine sinnvolle Erweiterung.

3.2.2 Image File (.img)

Um das Image File auszulesen, müssen vorher die Dimensionen des Voxelgitters bekannt sein. Diese Information erhält man aus dem Header File. Sind die Dimensionen bekannt können mithilfe einer dreifach geschachtelten for-Schleife die einzelnen Voxel-Werte in eine entsprechende Datenstruktur ausgelesen werden. Hierzu würde sich als Beispiel ein dreidimensionales Array sehr gut eigenen.

[Mayo, 2015] bietet eine beispielhafte Implementierung für das Auslesen einer solchen Image Datei. Diese wurde für den Prototyp angepasst und integriert.

3.2.3 Header File (.hdr)

Diese Datei ist wie bereits in Kapitel 2.4.2 beschrieben aufgebaut. Auch hier ist bietet [Mayo, 2015] eine beispielhafte Implementierung an, welche für die Verwendung angepasst und erweitert wurde.

3.2.4 STereoLithography (.stl)

In Kapitel 2.4.3 ist bereits der binäre Aufbau einer STL-Datei abgebildet. Die Implementierung (Programm 3.2) richtet sich daher genau an diesen formalen Aufbau.

[1] Image (.img) und Header (.hdr)

Programm 3.2: Generierung einer STL-Datei

```
1   void fileHandler::CreateStl(std::string path){
2     FILE *fptr = NULL;
3     int sizeResult = mResult.size();
4     if ((fptr = fopen(path.c_str(), "a+b")) == NULL) return;
5
6     char fileHeader[81] = "solid Test Head";
7     char bytes[3] = { 0x00, 0x00 };
8     fwrite(&fileHeader, sizeof(fileHeader)-1, 1, fptr);
9     fwrite(&sizeResult, sizeof(int), 1, fptr);
10    for (int i = 0; i < mResult.size(); i++) {
11      fwrite(&mResult[i].n, sizeof(float), 3, fptr);
12      for (int k = 0; k < 3; k++)
13        fwrite(&mResult[i].p[k], sizeof(float), 3, fptr);
14      fwrite(bytes, 2, 1, fptr);
15    }
16    fclose(fptr);
17  }
18
```

3.2.5 Schnittstelle (Klasse)

Diagramm 3.2: UML-Diagramm der fileHandler Klasse

«Class» fileHandler
- mPath : string - mType : fileType - mHdr : dsr - mData : d3Buffer - readImg(fp : *FILE) : bool - readHdr(fb : *FILE) : void - swapHdr() : void - showHdr() : void
+ fileHandler() + setpath(path : string) : void + setType(type : fileType) : void + setHdr(hdr : dsr) : void + getHdr() : dsr + readFile() : bool + createStl(path : string) : bool + getData() : d3Buffer

Datentypen

Neben den bereits zuvor beschriebenen Datentypen weist die fileHandler-Klasse noch weitere spezielle Typen zur Verarbeitung auf.

- **fileType** repräsentiert den zu verarbeitenden Dateityp als Enumeration. Im gegebenen handelt es sich um die Typen Image und Header.
- **dsr** ist die in Kapitel 2.4.2 beschriebene Repräsentation des Header-Formates als C-Struktur.
- **d3Buffer** wurde bereits im vorhergegangenen Abschnitt (3.1) beschrieben.

Funktionen

Es folgt ein Überblick der öffentlichen Methoden dieser Klasse.

- **fileHandler** der Konstruktor der Klasse initialisiert die Datenkomponenten für die spätere Verwendung.
- **setPath** setzt den Pfad zur verwendeten Datei.
- **setType** spezifiziert welche Dateityp von der Instanz dieser Klasse verwaltet werden soll.
- **setHdr:** Handelt es sich um eine Image-Datei sind dieser die Informationen aus der dazugehörigen Header-Datei mitzuteilen.
- **getHdr:** Hier können nach dem Auslesen eines Header-File die erhaltenen Informationen abgegriffen werden.
- **readFile:** Nach dem Setzen des Pfades und Dateityps kann hier die entsprechende Datei ausgelesen werden.
- **createStl:** Ermöglicht das Erstellen einer .stl Datei aus den vom Marching Cubes Algorithmus gewonnen Dreiecken.
- **getData:** Nach dem Auslesen einer Image-Datei können hier die erhaltenen Daten abgegriffen werden.

Verwendung

In Programm 3.3 ist die beispielhafte Verwendung der fileHandler-Klasse skizziert.

Programm 3.3: Verwendung der fileHandler-Klasse

```
1   int main(){
2     //create the classes
3     fileHandler *hdrFile = new fileHandler();
4     fileHandler *imgFile = new fileHandler();
5
6     // set the filetype
7     hdrFile->setType(hdr);
8     imgFile->setType(img);
9
10    // set the file path
11    hdrFile->setPath(ui->headerPath->text().toStdString());
12    imgFile->setPath(ui->imagePath->text().toStdString());
13
14    // read the header file
15    hdrFile->readFile();
16    // set the header for the image file
17    imgFile->setHdr(hdrFile->getHdr());
18    // read the image file
19    imgFile->readFile();
20
21    // get the image data and pass it to the marching cubes algorithm
22    marchingCubes->setData(imgFile->getData());
23
24    // free the memory
25    delete(hdrFile);
26    delete(imgFile);
27    return 0;
28  }
29
```

1. Erzeugen zweier Instanzen von *fileHanlder* für die Header- und Imagedatei.
2. Setzen der jeweils zu erwartenden Dateitypen für jede Instanz.
3. Setzen der Pfade wo die entsprechenden Daten im Filesystem abgelegt sind.
4. Einlesen der Header-Datei.
5. Setzen der gelesenen Header Informationen in der Image Instanz.
6. Einlesen der Image-Datei.
7. Übergabe des gelesenen Voxelgitters in die *marchingCubes*-Klasse.
8. Freigabe des in Punkt 1 allokierten Speichers.

3.3 OpenGL

Für die grafische Aufbereitung des generierten polygonalen Objektes wird wie bereits mehrfach erwähnt OpenGL verwendet. Um eine möglichst große Unabhängigkeit bezüglich des gewählten Betriebssystemes zu gewährleisten wurde QT als Schnittstelle zwischen System und OpenGL gewählt.

3.3.1 Allgemein

Die Implementierung ermöglicht die Darstellung des generierten Objektes, das Bewegen um die Achsen via Maus sowie das Zoomen via Mausrat.

3.3.2 Schnittstelle (Klasse)

Diagramm 3.3: UML-Diagramm der Visualisation Klasse

«Class» Visualisation : QGLWidget
- rotationX : GLfloat - rotationY : GLfloat - rotationZ : GLfloat - faceColors : QColor[4] - lastPos : QPoint - mObject : vector<TRIANGLE> - mScale : GLfloat - draw(); - qNormalizeAngle(ref angle : int) : void - drawSlice() : void - drawAll() : void - scaleObject() : void
initializeGL() : void # resizeGL(width : int, height : int) : void # paintGL() : void # mousePressEvent(*event : QMouseEvent) : void # mouseMoveEvent(*event : QMouseEvent) : void # wheelEvent(*event : QWheelEvent) : void # setXRotation(angle : int) : void # setYRotation(angle : int) : void # setZRotation(angle : int) : void
+ Visualisation(*parent = 0 : QWidget) : void + setObject(*object : vector<TRIANGLE>) : void + setWiring(enable : bool) : void + updateUI() : void

Funktionen

Es folgt eine kurzer Überblick über die öffentlichen Funktionen der Visualisation-Klasse.

- **Visualisation:** Erstellen der Zeichenoberfläche.
- **setObject:** Setzen des zu zeichnenden Objektes.
- **setWiring:** Ermöglicht das Hin- und Herschalten zwischen den beiden Polygonmodi GL_LINE und GL_FILL siehe Abbildung 3.3.
- **updateUI:** Manuelles update der Zeichenfläche z.B. nach setWiring oder setObjekt, um die Änderungen für den Benutzer sichtbar zu machen.

Verwendung

Die Verwendung dieser Klasse ist vergleichsweise einfach. Eine beispielhafte Implementierung ist in Programm 3.4 zu finden.

Programm 3.4: Exemplarische Verwendung der Visualisation Klasse

```
1   int main(){
2     Visualisation vi = new Visualisation();
3
4     // set the polygon object (result of the marching cubes algorithm)
5     vi->setObject(marchingCubes->getResult());
6     // update ui for the user
7     vi->updateUI();
8     // enable wiring (set polygon mode to GL_LINE)
9     vi->setWiring(true);
10    // update ui for the user
11    vi->updateUI();
12
13    delete(vi);
14  }
15
```

1. Es wird eine neue *Visualisation* Instanz erzeugt.
2. Setzen des polygonalen Objektes welches aus dem Marching Cubes Algorithmus hervor geht.
3. Darstellen des neu gesetzte Objekt mit der Funktion *updateUI*.
4. Aktivieren des Wirings. Hierdurch erhalten wir die in Abbildung 3.3 dargestellt Gitterform.
5. Erneuter Aufruf von *updateUI*.
6. Freigabe der allokierten *Visualisation* Instanz.

3.4 Benutzeroberfläche

In diesem Abschnitt wird auf die Funktionen der Benutzeroberfläche (UI) des entstandenen Prototypen eingegangen. Erstellt wurde diese mithilfe des QT-Designers. Um eine einfache Benutzung zu gewährleisten wurde sie möglichst schlicht gehalten.

3.4.1 Allgemeiner Aufbau

In Abbildung 3.1 ist die Benutzeroberfläche dargestellt. Es folgt nun eine kurze Beschreibung der einzelnen Schaltflächen.

- **Menu:** In der linken oberen Ecke befindet sich die Menu Schaltfläche welche nur einen Eintrag aufweist. Dieser ermöglicht das Exportieren des generierten Objektes in das .stl Format.
- **Select Header:** Durch das Betätigen des Button ”Select Header” öffnet sich eine Dialogbox zum Auswählen einer .hdr Datei im Dateisystem. Nach erfolgreichem Selektieren einer Datei erscheint der Dateipfad links im Textfeld. Alternativ kann auch der Pfad direkt im Textfeld bearbeitet werden.
- **Select Image:** Die Funktionalität ist Analog zu ”Select Header” für .img Dateien.
- **Generate View:** Durch das Betätigen dieser Schaltfläche wird der Marching Cubes Algorithmus auf die ausgewählten Dateien angewandt. Nach erfolgreicher Berechnung wird das Ergebnis rechts in der Zeichenoberfläche dargestellt.
- **Enable Wiring:** Siehe Abschnitt 3.4.2.
- **Enable Slicing:** Siehe Abschnitt 3.4.3.
- **Zeichenoberfläche:** Die rechte Hälfte der Benutzeroberfläche dient zur dreidimensionalen Darstellung des generierten Objektes. Mithilfe der Maus ist es möglich das angezeigte Objekt zu rotieren. Des Weiteren besteht die Möglichkeit mit dem Mausrat die Größe zu ändern (zoomen).
- **Isolevel Regler:** Der rechts neben der Zeichenfläche angebrachte Regler dient zur Anpassung des Isowertes. Bei jeder Änderung wird das anzuzeigende Objekt mit dem neuen Wert berechnet und angezeigt.
- **Polygon Regler:** Dieser Regler ist unterhalb der Zeichenoberfläche zu finden und hat zwei Funktionen. Die erste Funktionalität ist das Anpassen der Schrittweite für den Marching Cubes Algorithmus. Eine höhere Schrittweite führt zu einer gröberen polygonalen Struktur. Die zweite Funktion ist das Anpassen der Dicke der Scheibe bei aktiviertem ”Slicing”

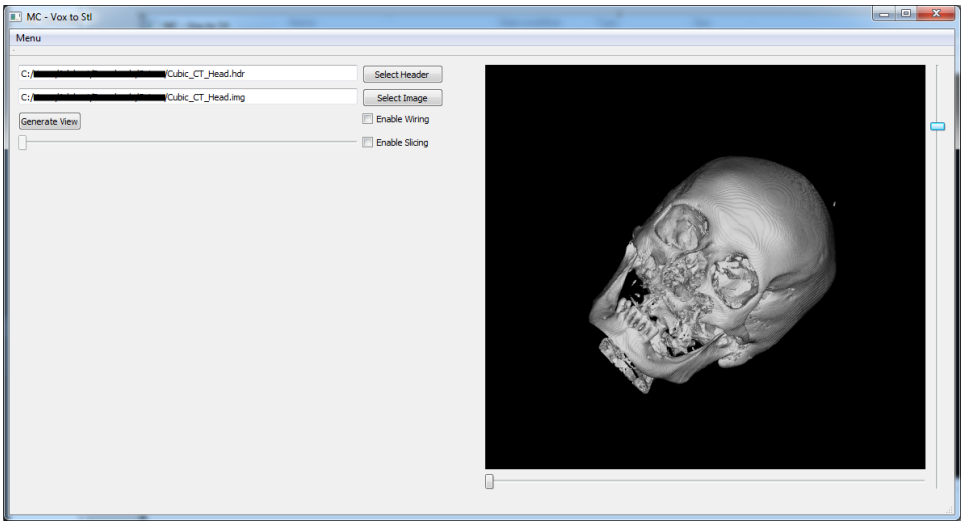

Abbildung 3.1: Übersicht der Benutzeroberfläche

3.4.2 Slicing

Die Schaltfläche ”Enable Slicing” ermöglicht die Darstellung einer einzelnen Scheibe des gesamt Objektes. Durch das Aktivieren der Checkbox wird auch der Regler links aktiviert. Mithilfe von diesem kann das zu betrachtende Scheibenelement verschoben werden. Des Weiteren ändert sich die Funktionalität des unteren Reglers von der Regulierung der Polygongröße zur Regulierung der Dicke der Scheibe.

Vorteil dieser Darstellungsform ist, dass das mit aktivem Slicing das Innenleben eines Objektes betrachtet werden kann.

Das Benutzerinterface mit aktiviertem Slicing ist in Abbildung 3.2 zu sehen.

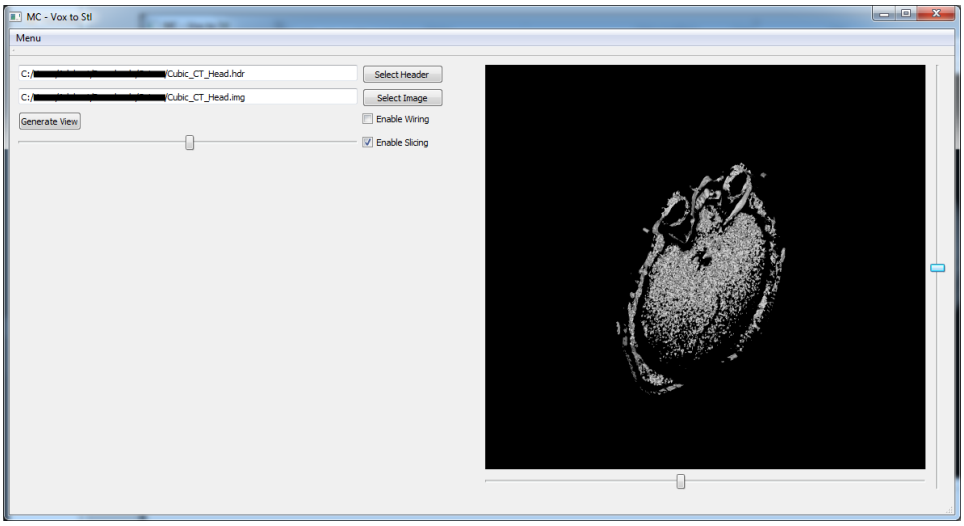

Abbildung 3.2: Programm Slicing aktiviert

3.4.3 Wiring

Durch Aktivieren der Checkbox ”Enable Wiring” kann das Drahtgittermodell des Objektes betrachtet werden. Diese Darstellung ermöglicht es, die polygonale Struktur des Objektes besser zu erkennen. In Abbildung 3.3 ist ein Objekt mit aktivem Wiring zu sehen.

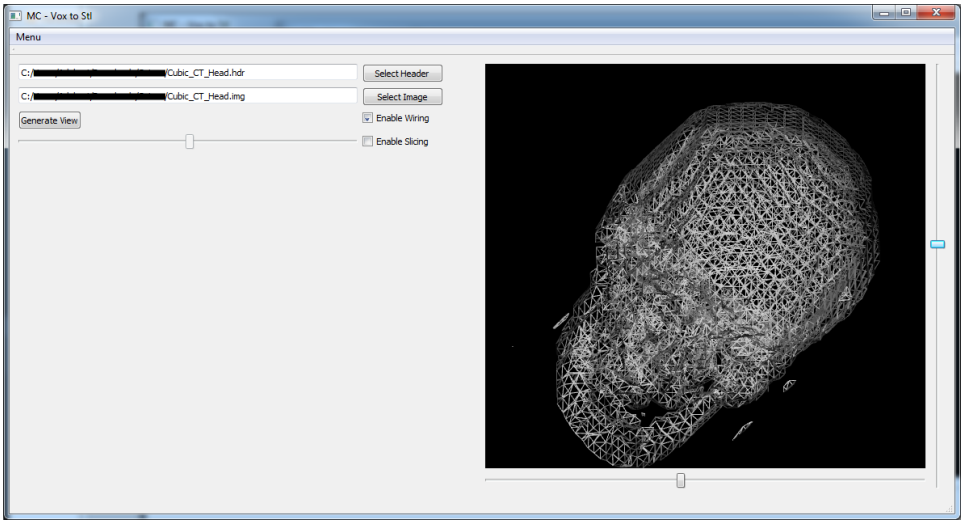

Abbildung 3.3: Programm Wiring aktiviert

Kapitel 4

Zusammenfassung und Ausblick

In der gegenständlichen Arbeit wurde einerseits das Thema Marching Cubes theoretisch beleuchtet, sowie eine mögliche praktische Umsetzung präsentiert. Es folgt eine kurze Zusammenfassung über das erreichte sowie ein Ausblick über mögliche weiterführende Entwicklungen.

4.1 Erweiterungen

Aufgrund des eng geschnürten Zeitplans konnten leider nicht alle Ideen und Ansätze bezüglich der Implementierung umgesetzt werden. Des Weiteren kamen während der Umsetzung Ideen für eine bessere Implementierung auf, welche bei einer Überarbeitung bzw. Erweiterung gut untergebracht wären.

4.1.1 Parallelisieren

Geschuldet der Tatsache, dass das Voxelgitter in einzelne Würfel unterteilt wird und auf jeden dieser der Marching Cubes Algorithmus angewandt wird, bietet sich hier die Möglichkeit für einen parallelen Ansatz in der Implementierung.

Ebenso wäre es aufgrund der längeren Berechnungszeit des Algorithmus wünschenswert, diese nach der Betätigung des "Generate" Buttons im Hintergrund laufen zu lassen, um zu verhindern, dass die Benutzeroberfläche nicht einfriert.

4.1.2 Alternative Algorithmen

Der Marching Cubes Algorithmus wurde zwar bereits im Jahre 1987 vorgestellt, allerdings konnte dieser aufgrund patentrechtlicher Probleme nicht einfach verwendet werden. Daher wurden im Laufe der Zeit ähnliche, dem

Marching Cubes Algorithmus nachempfundene Ansätze entwickelt, wie z.B. Marching Squares oder Marching Thetrahedrons. Um einen direkten Vergleich zwischen allen Ansätzen zu beobachten, könnten diese als mögliche Erweiterung ebenso im Programm eingebaut werden.

4.1.3 Algorithmen zur Verbesserung

Wie bereits in Kapitel 2.3.3 beschrieben, gibt es einige Schwachpunkte bezüglich des normalen Marching Cubes. Um diese Schwächen auszumerzen, gibt es einige interessante Algorithmen, welche ebenfalls im Kapitel 2.3.3 kurz beschrieben sind.

4.1.4 Alternative Dateiformate

Neben den verwendeten Dateiformate gibt es noch eine Vielzahl weiterer Formate zur Darstellung von Volumengrafiken und polygonalen Grafiken. Um eine entsprechend vielseitige Verwendung zu gewährleisten, wäre daher ein Import und Export von mehreren Formaten wünschenswert.

4.2 Zusammenfassung

Das Ziel dieser Arbeit war die Implementierung des 1987 vorgestellten Marching Cubes Algorithmus sowie die Darstellung des generierten polygonalen Modelles via OpenGL. Des Weiteren sollte es möglich sein, die generierten Objekte als STereoLithography Datei zu exportieren, da dies ein weit verbreitetes Format darstellt und von vielen Programmen unterstützt wird.

Diese Ziele wurden erreicht und entsprechen aufbereitet. Des Weiteren wurden Überlegungen bezüglich möglicher Erweiterungen und Verbesserungen der verwendeten Algorithmen angestellt.

Literaturverzeichnis

[Bourke 1994] Bourke, Paul: *Polygonising a scalar field.* http://paulbourke.net/geometry/polygonise/, 1994. – Besucht: 2015-25.08

[Chua u. a. 1997] Chua, Kai Chee ; Gan, K. J. G. ; Tong, Mei: *Interface between CAD and Rapid Prototyping systems. Part 2: LMI — An improved interface.* 13. Springer-Verlag, 1997

[Fabbers 2015] Fabbers: *The StL Format.* http://www.fabbers.com/tech/STL_Format, 2015. – Besucht: 2015-09.08

[Hadels 2000] Hadels, Heinz: *Medizinische Bildverarbeitung.* 2. Springer-Verlag, 2000

[Hansen u. Johnson 2005] Hansen, Charles D. ; Johnson, Chris R.: *The Visualisation Handbook.* Elsevier Academic Press, 2005

[Lorensen u. Cline 1987] Lorensen, William E. ; Cline, Harvey E.: Marching Cubes: A High Resolution 3D Surface Construction Algorithm. In: *Computer Graphics, Volume 21, Number 4* (1987)

[Mayo 2015] Mayo: *ANALYZE 7.5 File Format.* https://rportal.mayo.edu/bir/ANALYZE75.pdf, 2015. – Besucht: 2015-09.08

[Nielson 2003] Nielson, G. M.: On Marching Cubes. In: *Visualization and Computer Graphics, IEEE Transactions on (Volume:9 , Issue: 3)* (2003)

[Nielson u. Hamann 1991] Nielson, Gregory M. ; Hamann, Bernd: The Asymptotic Decider: Resolving the Ambiguity in Marching Cubes. In: *Proceedings of the 2Nd Conference on Visualization* (1991)

[Schroeder u. a. 1992] Schroeder, William J. ; Zarge, Jonathan A. ; Lorensen, William E.: Decimation of triangle meshes. In: *Proceedings of the 19th annual conference on Computer graphics and interactive techniques* (1992)

[Segal u. Akeley 2009] Segal, Mark ; Akeley, Kurt: *The OpenGL Graphics System: A Specification.* 3.1. https://www.opengl.org/registry/doc/glspec31.20090324.pdf, 2009

[Seibt 2014] Seibt, Georg: *Oberflächenextraktion mittels des Marching Cubes Algorithmus.* Passau, 10 2014

[Shirley u. Marschner 2009] Shirley, Peter ; Marschner, Steve: *Fundamentals of Computer Graphics.* 3. Taylor & Francis Ltd, 2009

[Vollmer u. a. 2001] Vollmer, J. ; Mencl, R. ; Müller, H.: Improved Laplacian Smoothing of Noisy Surface Meshes. In: *Computer Graphics Forum Volume 18, Issue 3* (2001)

[Wilhelms u. Gelder 2000] Wilhelms, Jane ; Gelder, Allen V.: Octrees for faster isosurface generation. In: *IEEE TRANSACTIONS ON MEDICAL IMAGING* 19 (2000), S. 739–758

[Wollmann 2013] Wollmann, Thomas S.: *Entwicklung und Implementierung eienes Marching Cube basierten Algorithmus zur punkterhaltenden Triangulation von Konturen mit Subpixel-Auflösung.* Heilbronn, 08 2013