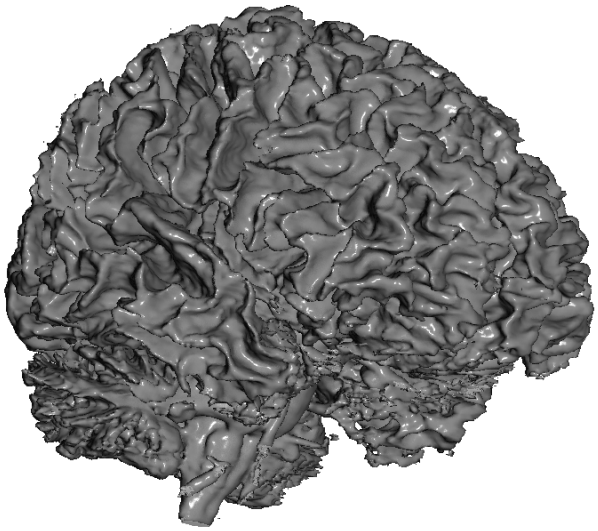
Besondere Lernleistung

„Optimierung des Marching Cubes Algorithmus durch das Span Space Verfahren“



Geistert, David

Klasse

12de2s

Schuljahr

2013/2014

Leipzig, im März 2014

Gymnasium Schkeuditz

Lessingstraße 10

04435 Schkeuditz

Betreuender Fachlehrer: Frau Naundorf (Gymnasium Schkeuditz)

Externer Betreuer: Sebastian Eichelbaum (Universität Leipzig)

Inhaltsverzeichnis

Abkürzungsverzeichnis

2D	Zweidimensional
3D	Dreidimensional
CPU	Central Processing Unit
CT	Computertomographie
GPU	Graphics Processing Unit
MC	Marching Cubes
MRT	Magnetresonanztomographie
RAM	Random Access Memory
SS	Span Space

Tabellenverzeichnis

Abbildungsverzeichnis

Codeverzeichnis

Zusammenfassung

Der Marching Cubes Algorithmus soll mittels des Span Space Verfahrens in seiner Performance optimiert werden. Diese Verbesserung hinsichtlich der Berechnungszeit geschieht auf Kosten eines höheren Arbeitsspeicherverbrauches. Es wurde jedoch festgestellt, dass der unoptimierte Marching Cubes Algorithmus mit heutigen, technologischen Voraussetzungen schneller als der Marching Cubes Algorithmus mit Span Space Verfahren ist.

The performance of the Marching Cubes algorithm should be optimized. The optimization only works because of a higher memory consumption. But in fact it was discovered that the Marching Cube algorithm without optimization is faster than the optimized Marching Cubes algorithm because of the modern technical components.

---

1 Einleitung

1.1 Problemdarstellung

Der Marching Cubes Algorithmus in Open Walnut benötigt auf Grund seiner Vorgehensweise relativ lange die Skalardaten zu verarbeiten. Durch Anwendung des Span Space Verfahrens sollte eine Optimierung des Marching Cubes Algorithmus erfolgen und somit die Performance verbessert werden.

1.2 Skalardaten

Skalardaten sind in der Praxis sehr weit verbreitet. Sie spielen vor allem bei Magnetresonanztomographie (MRT)- bzw. Computertomographie (CT)-Scans eine wesentliche Rolle, denn die Rohdaten des MRT- bzw. CT-Scans werden in Skalardaten abgespeichert. Ohne diese Rohdaten wäre jegliche Analyse des gescannten Objektes unmöglich. Die Skalardaten sind demnach essentiell für die Auswertung von MRT- bzw. CT-Scans [8].
Skalardaten bestehen vereinfacht gesagt nur aus reellen Zahlen, welche eine Menge A von Dreidimensional (3D) Punkten $P_i = (x_i, y_i, z_i)$ beschreiben. Jedem dieser 3D Punkte wird ein gewisser Wert zugeordnet [3, S. 854]. Die Besonderheit ist, dass die Koordinaten nie direkt gespeichert werden. Vielmehr nutzt man ein reguläres Würfelgitter, wodurch die Position eines Voxel „implizit aus der Position zu anderen Voxeln hergeleitet“[10] wird.
In der medizinischen MRT- bzw. CT-Diagnostik ist dieser Grauwert nicht nur eine Farbe, sondern er gibt eine physikalische Größe wie die Dichte von Wasserstoffatomen (MRT) oder Röntgendurchlässigkeit (CT) des gescannten Bereiches an [12]. So ist es möglich Objekte mit gleicher Dichte zu extrahieren. Dies ist in der Medizin von großer Bedeutung, weil so gewisse

Organe, bzw. Gewebeschichten freigestellt und somit getrennt betrachtet werden können.
Die Extraktion von einem Grauwert aus 3D Bildern ist vergleichbar mit der Extraktion von Farbwerten aus Zweidimensional (2D) Bildern.

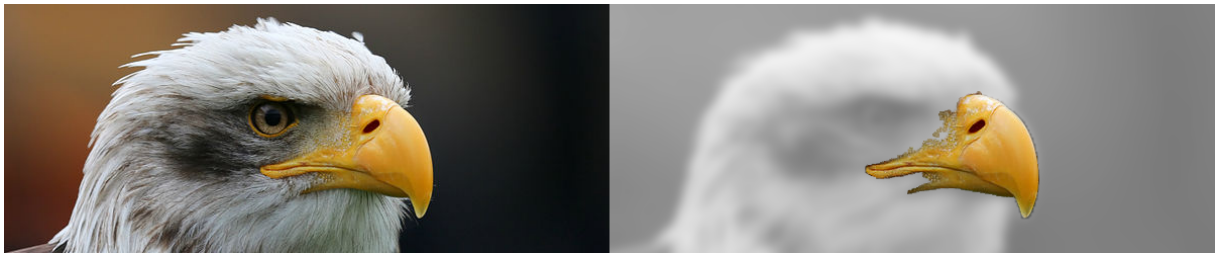


Abbildung 1: extrahierter, gelber Schnabel eines Adlers

So ist es beispielsweise möglich, den farblich markanten, gelben Schnabel aus einem 2D Bild von einem Adler freizustellen (Abb. 1). In 2D Bildern werden dafür alle Pixel mit den selben Farbwert ausgewählt wie z.B. die Pixel des Schnabels. In 3D Bildern ist dieser Prozess nicht anders.
3D Bilder werden auch Voxelgrafiken genannt. Eine Voxelgrafik besteht aus einzelnen Voxeln, d.h. Datenpunkte in 3 Dimensionen. Ein Voxel (zusammengesetzt aus volumetric und pixel) ist in einer 3D-Grafik wie ein Pixel in einer 2D-Grafik. Ein Voxel kann man sich wie einen Würfel vorstellen, da eine dritte Dimension existiert. Desweiteren wird ein Voxel nicht durch zwei Koordinaten, sondern durch drei Koordinaten beschrieben.
Da Computer-Bildschirme keine dritte Dimension besitzen, ist es unmöglich 3D Bilder darzustellen. Durch einen Algorithmus können 3D Bilder auf eine 2D Ebene projiziert werden. Diese 2D Projektionen finden nicht nur in der Visualisierung von medizinischen Daten Anwendungen, sondern aktuelle Computerspiele basieren ausnahmslos auf solch einer Projektion. Sie wandeln komplexe Dreiecksmodelle in ein 2D Bild um. Diese Dreiecksmodelle werden beispielsweise durch den *Marching Cubes (MC) Algorithmus* umgesetzt.

1.3 Marching Cubes Algorithmus

Der „Marching Cubes ist ein Algorithmus zur Berechnung von Isoflächen in der 3D-Computergrafik. Er nähert eine Voxelgrafik durch eine Polygongrafik an.“[9]
Eine Isofläche ist eine Fläche, die im dreidimensionalen Raum benachbarte Punkte mit den selben Werten [5, S. 2-3], welcher auch *Iso-Wert* genannt wird, verbindet. *(Iso kommt aus dem Griechischen und bedeutet „Gleich“)[6].*

Eine Polygongrafik besteht im Gegensatz zu einer Voxelgrafik nicht aus einzelnen Bildpunkten (Voxeln), sondern aus aus Polygonen. Ein Polygon ist ein geschlossene, geometrische Fläche, die aus mindestens drei verbundenen Punkten (vertices) besteht [7]. So ist es möglich, dass ein 3D Bild durch Dreiecke (Polygone) angenähert wird. Dreiecke sind ohne Probleme auf eine 2D-Fläche zu projizieren.

1.3.1 Vorgehensweise

Als Eingabe wird dem Marching Cubes Algorithmus das Gitter und der Schwellwert vorgegeben [3, S. 855]. Das Voxelgitter ist eine Liste aus allen Voxeln eines Objektes. Jedes einzelne Voxel hat vier Attribute - drei Koordinaten und einen Grauwert. Später werden nur Grauwerte dargestellt, die größer als der angegebene Iso-Wert sind.

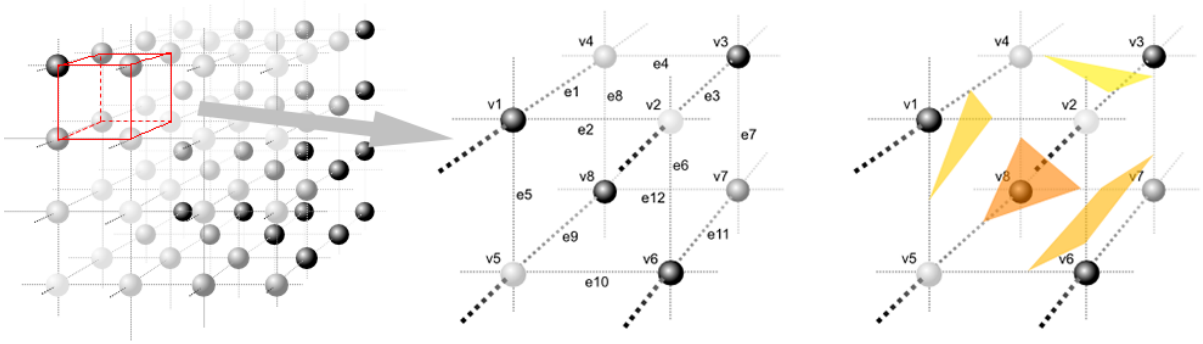


Abbildung 2: Veranschaulichung des „logischen“ Würfels

Anschließend wird ein „logischer“ Würfel (in Abb. 2 durch eine rote Umrandung dargestellt) in das Voxelgitter gelegt, welcher 2x4 gegenüberliegenden Voxel umfasst. Dieser „logische“ Würfel wird *Cube* genannt und ist im mittleren Teil in Abb. 2 zusehen. Jede Ecke (*vertex* bzw. *corner*) des Cubes liegt auf einem dieser Voxel und nimmt deren Wert an. Die Vertices werden in Abb. 2 durch Kugeln visualisiert. Die Kanten des Würfels werden *edges* genannt.
Nun „marschiert“ (marching) der „logische“ Würfel durch alle Voxel. Bei jedem Durchgang prüft der Algorithmus, welche vertices unter bzw. über dem Schwellenwert liegen. Die Vertices, deren Wert kleiner als der Schwellenwert ist, bleiben unmakiert (ihnen wird eine 0 zugeordnet) und die Vertices, deren Wert größer oder gleich dem Schwellenwert ist, werden makiert, indem ihnen eine 1 zugeordnet wird [2, S. 164]. In Abb. 2 sind die Vertices, deren Wert größer als der Schwellwert ist, als schwarz Kugeln dargestellt.
Bei Aneinanderreihung der Vertice-Werte ergibt sich eine achtstellige Zahl, welche den Würfel Index darstellt. Wird der Index des Würfels als eine Binärzahl betrachtet, die in eine Dezimal-

zahl umgewandelt wird, kann der Index einen Wert zwischen 0 und 255 annehmen.

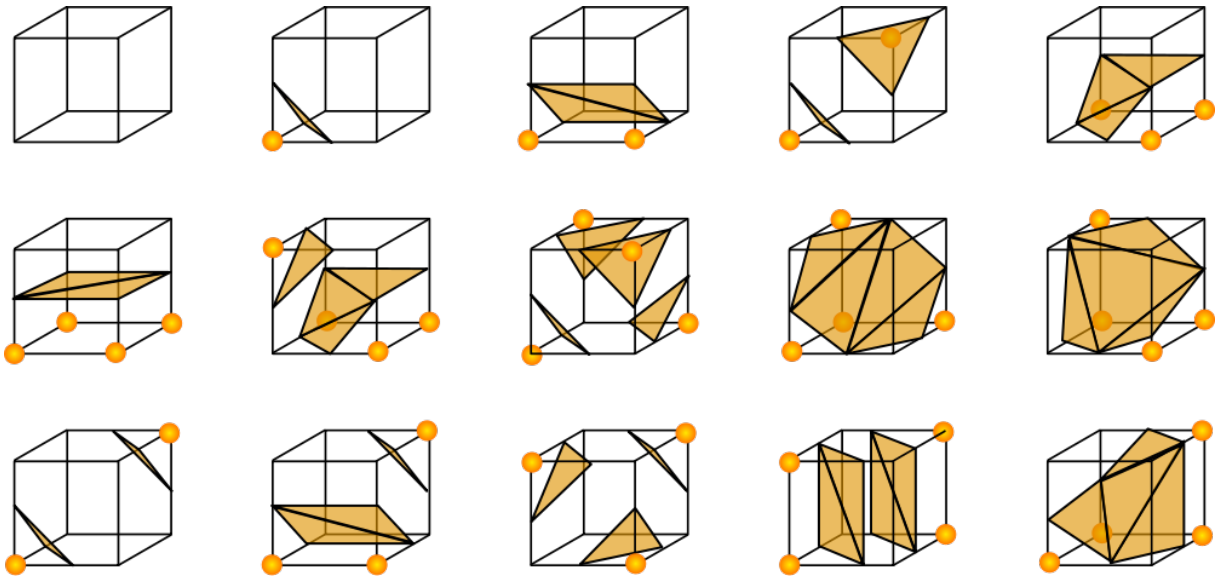


Abbildung 3: Alle 15 Möglichkeiten zur Dreiecksbildung

Dieser Würfel-Index wird in der sogenannten *Triangle Lookup Table* nachgeschlagen. In dieser Tabelle sind die Dreiecke enthalten, die in dem Würfel gebildet werden. Bei Vernachlässigung der Spiegelung der gebildeten Dreiecke, ergeben sich 15 verschiedene Fälle, wie die Dreiecke in dem Würfel angeordnet werden können (siehe Abb. 3).
Die genaue Position der Dreieckspunkte auf den Kanten des Würfels wird durch Interpolation zwischen den zwei angrenzenden Würfeleckpunkten berechnet.
Schließlich gibt der Marching Cubes Algorithmus eine Liste aller erzeugten Dreiecksknoten zurück. Dadurch kann eine Polygongrafik dargestellt werden.

1.3.2 Performance Problem

Die einfache Vorgehensweise des Marching Cubes ist auch sein Verhängnis. Es nimmt sehr viel Zeit in Anspruch, wenn der Algorithmus durch alle Würfel „marschieren“ muss. So werden auch Cubes verarbeitet, die nicht aktiv sind. Als aktive Cubes werden Cubes bezeichnet, welche mindestens ein Voxel besitzen, dessen Werte größer oder gleich des Isowerts ist. Dies ist verschwendete Zeit, da diese inaktiven Voxel ignoriert werden könnten. Unoptimiert hat der Marching Cubes Algorithmus eine Komplexität von $O(n)$.
Der Begriff „Komplexität“ beinhaltet in der Informatik den theoretischen Rechenaufwand eines Algorithmus. Je kleiner der Wert ist, desto weniger Ressourcen verbraucht der Algorithmus. Die Komplexität hilft dabei einzuschätzen, wie effizient Algorithmen sind.

Das Worstcase Szenario bezeichnet den Fall, in dem der Algorithmus die meisten Ressourcen verbraucht.
Die Performance des Marching Cubes Algorithmus sollte mit Reduktion der Berechnungszeit gesteigert werden.

1.4 Open Walnut

Open Walnut ist ein Open Source Programm zur Analyse und Visualisierung von Gehirndaten, welche beispielsweise im Zusammenhang mit MRT bzw. CT-Scans entstanden sind. Open Walnut ist in C++ geschrieben und läuft auf allen gängigen Betriebssystemen. Es ist ein Projekt der Universität Leipzig und wird primär von Sebastian Eichelbaum programmiert.
Bei der Entwicklung wurde besonders auf den großen Funktionsumfang und auf die einfache Bedienung Augenmerk gelegt. Desweiteren ist es erlaubt Open Walnut umzuprogrammieren bzw. auf eigene Probleme anzupassen. Deshalb ist Open Walnut nicht nur als Programm, sondern auch als ein Grundgerüst zum Programmieren eigener Projekte nutzbar.
Durch die große Anzahl von Modulen sind die Einsatzgebiete sehr fassettenreich. Das Modul *„Isosurface“* berechnet ein Bild (das sogenannte Rendern) der Skalardaten mittels des Marching Cubes Algorithmus. Durch einen Schieberegler wird dem User ermöglicht, den Schwellenwert des Marching Cubes Algorithmus zu verändern.
Wird der Schieberegler verändert, weil die Isofläche mit einem andern Schwellenwert sichtbar sein soll, muss der Marching Cubes Algorithmus jedes mal neu durchlaufen. Durch diese ineffiziente Arbeitsweise des Marching Cubes Algorithmus, entstehen je nach Datenmenge relativ lange Wartezeiten.

1.5 Programmiersprache C++

„Ursprünglich wurde C++ von Dr. Bjarne Stroustrup 1979 entwickelt, um Simulationsprojekte mit geringem Speicher- und Zeitbedarf zu programmieren.“ [11, S. 25] Die Programmiersprache wurde auf dem Betriebssystem UNIX entwickelt. Das große Problem war damals, dass keine Programmiersprache existierte, die schnell und umfangreich zugleich war. Deshalb hat Stroustrup die schnelle Programmiersprache C um ein Klassenkonzept erweitert [11, S. 26]. Daraus entstand dann der Name *C++*.
Über die Jahre sind viele Verbesserungen und Erweiterungen dazu gekommen, wie zum Beispiel die Ausnahmebehandlung. Desweiteren wurden viele Bibliotheken entwickelt, welche Funktionen zur Lösung von häufig, auftretenden Problemen bereitstellt.

C++ hat sich heutzutage zu einer der meist verwendeten Programmiersprachen durchgesetzt. Dies kann man auf die Stärken von C++ zurückführen. Eine der bedeutendsten Stärken ist die Erzeugung von hocheffizienten Code und der Einsatz in sehr umfangreichen Projekten [11, S. 27]. Dafür ist C++ sehr schwer zu erlernen und eine lange Einarbeitungszeit ist von Nöten. Bekannte Programme wie FireFox, Skype und VLC sind mit C++ programmiert wurden.

2 Ziel

2.1 Zieldarstellung

Ziel der Arbeit sollte die Performance Verbesserung des Marching Cubes Algorithmus sein. Dafür wurde das Span Space Verfahren genutzt.

2.2 Span Space

Das Span Space Verfahren greift genau da ein, wo der Marching Cubes Algorithmus seine Schwäche hat - das Prüfen *aller* 2x4 Voxel Cubes.
Mit dem Span Space Verfahren muss der Datensatz nur einmal gescannt und sortiert werden, wenn das geschehen ist, können recht schnell die aktiven Würfel abgerufen werden.
Die Grundidee dieses Verfahrens ist, dass der Datensatz in Teilbereiche eingeteilt wird, welche *buckets* genannt werden. Diese Teilbereiche werden gebildet, indem der Datensatz abwechselnd einmal nach den maximalen Wert der Cubes und dem Minimalen sortiert wird. Wenn er fertig sortiert ist, wird er in der Mitte halbiert und es entstehen zwei neue Teilbereiche. Für jeden Teilbereich wird die Spanne zwischen den kleinsten und größten Maximal- bzw. Minimal Werten gespeichert.

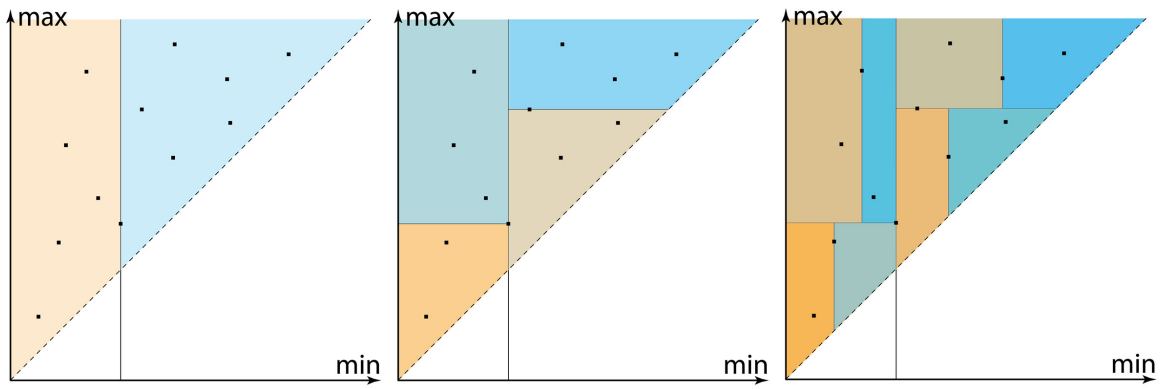


Abbildung 4: Beispielhafte Visualisierung des SpanSpace Verfahren

Beim SpanSpace Verfahren befinden sich einzelnen Cubes als Punkte $P(min|max)$ in einem Ko-

ordinatensystem, welche durch ihren minimalen (X-Achse) bzw. maximalen (Y-Achse) Grauwert angeordnet werden (siehe Abb. 4. In so einem Koordinatensystem werden alle Punkte auf bzw. über der Funktion $f(x) = x$ liegen, weil $P_{min} \leq P_{max}$ gilt. Punkte unter der Funtkion $f(x) = x$ hätten einen größeren „minimal Wert“ als „maximal Wert“, was logisch unmöglich wäre.

In Abb. 4 ist außerdem auch sehr gut der schrittweise Teilungsprozess zu erkennen. Die blaue dargestellten Bereiche in Abb. 4 sind die „Maximal-Bereiche“ und die orangen Bereiche die „Minimal Bereiche“.

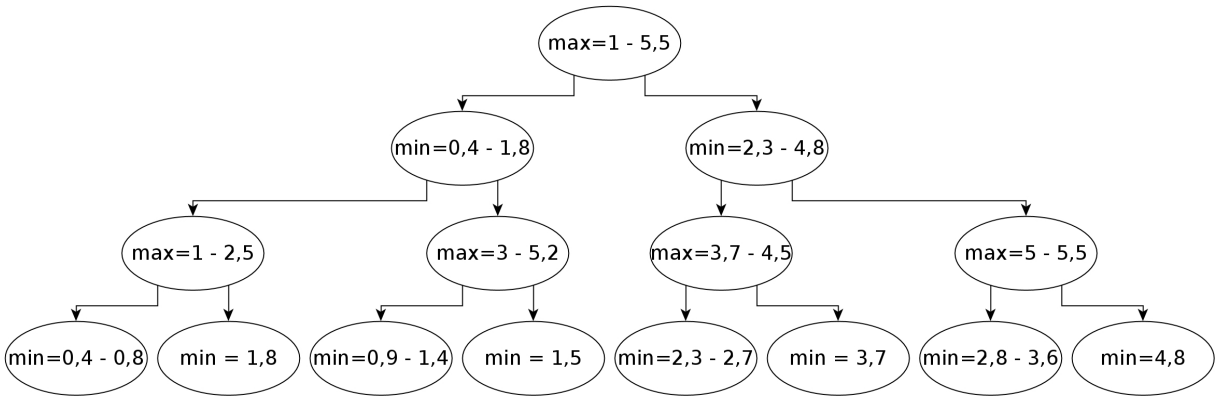


Abbildung 5: Ausschnitt eines KdTree

In der Praxis wird dieser Prozess in einem KdTree umgesetzt. Ein KdTree besteht aus einzelnen Knoten (*„nodes“*). Diese Knoten enthalten, abhängig von der Ebene, den Bereich von den minimalen bzw. maximalen Werten (siehe Abb. 5). Sollte ein Knoten keinen weiteren unteren Knoten (*„subnode“*) besitzen, handelt es sich um ein einzelnes Element. In unserer Anwendung wäre dieses einzelne Element ein Cube.

Ein Kd-Tree hat eine unoptimiert Bildungskomplexität von $O(n*\log(n))$. Der große Performance-Schwachpunkt ist das Sortieren der Werte, da jeder Wert rekursiv sortiert werden muss. Jedoch kann dieser Prozess optimiert werden, indem sichergestellt wird, dass „nach“ dem Median nur Werte sind, die größer als die Werte „vor“ dem Median sind. Diesen Sortieralgorithmus wird in C++ „nth-element“ genannt, welches sehr an den Sotieralgorithmus „Quicksort“ angelehnt ist. „Nth-element“ hat eine Worstcase Komplexität von $O(n^2)$, jedoch ist die durchschnittliche Komplexität linear $O(n)$ [1].

Um die Cubes zu extrahieren, muss von Teilbereich zu Teilbereich „gegangen“ werden, bis nur noch Cubes in der Auswahl sind, deren Minimal- Wert kleiner und deren Maximale-Wert größer als der gesuchte Isowert ist.

Da die Teilbereiche immer abwechselnd nach dem maximalen bzw. minimalen Werten sortiert werden, muss auch abwechselnd gesucht werden. Einem „Maximal-Wert“-sortierten Teilbereich darf nur weiter „verfolgt“ werden, wenn der größte „Maximal-Wert“ größer als der gesuchte Isowert ist. Einem „Minimal-Wert“-sortierten Teilbereich darf hingegen nur „verfolgt“ werden, wenn der kleinste „Minimal-Wert“ kleiner als der gesuchte Isowert ist.
Dieser Algorithmus sucht so lange, bis in den Teilbereichen nur noch einzelne Cubes übrig bleiben. Diese übrig gebliebenen Cubes werden dann an den Marching Cubes Algorithmus weiter gegeben, welcher mit diesen dann eine Isofläche rendert.
Das Suchen der Cubes hat eine Worstcase Komplexität von $O(\sqrt{n} + k)$ [4, S. 73], bei der n die Anzahl aller Cubes ist und k die Anzahl der extrahieren Cubes ist.

2.3 Leistungsverbesserung

2.3.1 Performance Verbesserung

Der normale Marching Cubes Algorithmus hat eine Komplexität von $O(n)$. Wenn das Span-Space Verfahren verwendet wird, erhält man eine Komplexität, welche sich aus der Bildungs- und Suchkomplexität zusammensetzt. Die Worstcase Komplexität wäre $O(n^2 + \sqrt{n} + k)$.
An dieser Komplexität ist bereits auf den ersten Blick erkennbar, dass keine Performance-Verbesserung zu erwarten ist, sondern vielmehr eine starke Verschlechterung.
Man muss jedoch beachten, dass es sehr unwahrscheinlich ist, dass die Grauwerte zufällig so angeordnet sind, dass der Worstcase-Fall eintritt. Deshalb ist davon auszugehen, dass die Komplexität $O(n + \sqrt{n} + k)$ ist.
Außerdem wird in der Regel die Isofläche des Datensatzes mehrfach berechnet. Daraus ergibt sich für den normalen Marching Cubes folgende Komplexität $O_i(i * n)$ und für die SpanSpace Optimierung $O_i(n + i * (\sqrt{n} + k))$. Wenn jedoch tatsächlich der Worstcase-Fall eintreten sollte, wäre die Komplexität des SpanSpace Verfahren $O_i(n^2 + i * (\sqrt{n} + k))$.
Ein durchschnittlicher Datensatz von einem Gehirn hat ca. 5 Millionen Zellen. Davon sind durchschnittlich 300.000 Zellen aktiv, was 6% entspricht. In der folgenden Tabelle sind alle Komplexitäten von diesem Beispiel zusammengefasst.

#	Marching Cubes	SpanSpace Durchschnitt	SpanSpace WorstCase
	$O_i(i * n)$	$O_i(n + i * (\sqrt{n} + k))$	$O_i(n^2 + i * (\sqrt{n} + k))$
i	$O_i(i * 5 * 10^6)$	$O_i(5 * 10^6 + i * (\sqrt{5 * 10^6} + 3 * 10^5))$	$O_i((5 * 10^6)^2 + i * (\sqrt{5 * 10^6} + 3 * 10^5))$
1	$O_1 = 5 * 10^6$	$O_1 = 5,3 * 10^6$	$O_1 = 2,5 * 10^{13}$
5	$O_5 = 2,5 * 10^7$	$O_5 = 6,5 * 10^6$	$O_5 = 2,5 * 10^{13}$
10	$O_{10} = 5 * 10^7$	$O_{10} = 8 * 10^6$	$O_{10} = 2,5 * 10^{13}$

Tabelle 1: Beispielhafte Berechnung der Komplexität des Marching Cubes Algorithmus

Aus diesem Rechenbeispiel ist erkennbar, dass der Worstcase den Rechenaufwand des Algorithmus ins Unermessliche steigern würde (siehe Tabelle 1). Es ist dann in der Analyse zu überprüfen, ob wirklich die durchschnittliche Komplexität eintritt, um zu entscheiden ob das SpanSpace Verfahren wirklich eine Optimierung zur Folge hat.
Wenn jedoch wirklich immer die durchschnittliche Komplexität eintreten sollte, würde ohne Optimierung beim fünffachen Berechnen der Isofläche die ca. vierfache Zeit benötigt werden.

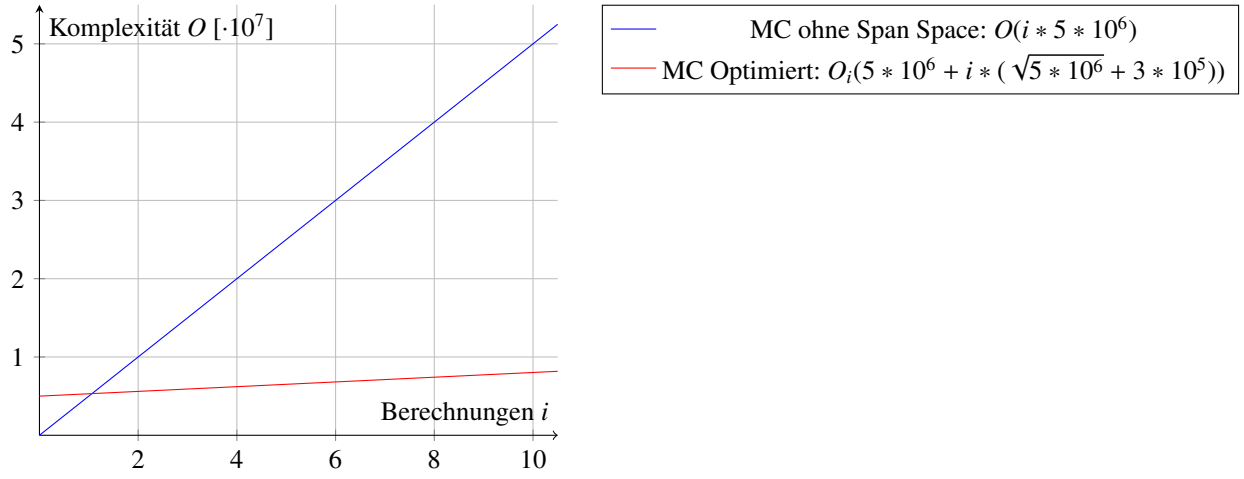


Abbildung 6: Visualisierung der Komplexitäten des un- bzw. optimierten MC

In Abb. 6 wird deutlich, dass der Marching Cubes Algorithmus ohne Span Space sehr stark linear ansteigt. Der optimierte Marching Cubes steigt im Gegensatz dazu, nach dem der KdTree einmal gebildet wurde, sehr schwach linear an.

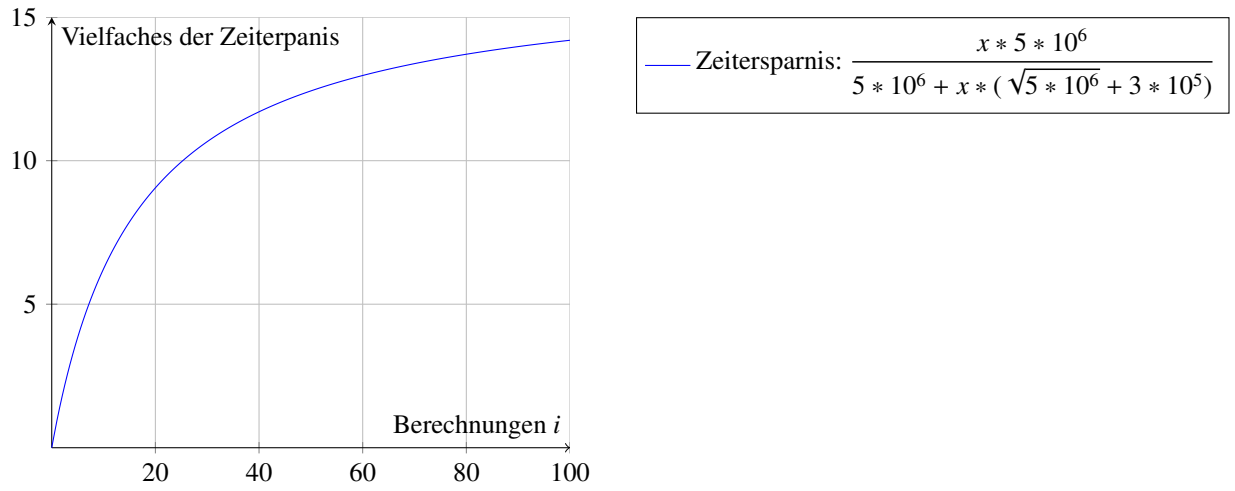


Abbildung 7: Visualisierung der Zeitersparnis

Die Funktion aus Abb. 7 nähert sich an den Wert $\frac{5000}{300+\sqrt{5}} \approx 16,54$ an. Das bedeutet, dass es bei diesem Datensatz nicht möglich ist mehr als die 16,54 fache Zeit gegenüber den unoptimierten Marching Cubes Algorithmus einzusparen.

2.3.2 Speicherbetrachtung

Der Marching Cubes Algorithmus hat in dem Sinne keinen Speicherverbrauch, da er die Eingabedaten direkt verarbeitet.
Das Span Space Verfahren hat im Vergleich zu dem Marching Cubes Algorithmus einen enorm hohen Speicherverbrauch, denn es muss ein KdTree erstellt werden. Der Speicheraufwand eines KdTree ist durch $O(k * n)$ definiert. k gibt die Anzahl der Dimensionen des KdTree an und n die Anzahl der zu speichernden Elemente. Der KdTree des SpanSpace Verfahren hat 2 Dimensionen - die Min und Max Dimension. Daraus ergibt sich $k = 2$.
Jeder Knoten des KdTree besitzt einen *Min* und *Max* Wert. Diese Werte sind als eine Gleitkommazahl (*float*) gespeichert. Dieser Datentyp benötigt 32 Bit Speicher, was 4 Byte entspricht. Desweiteren besitzt jeder Knoten eine X,Y,Z Koordinate, welche als Ganzzahl (*unsigned int*) gespeichert. Dieser Datentyp benötigt auch 32 Bit, 4 Byte Speicher. Außerdem benötigt jeder Knoten zwei Zeiger (*pointer*), welche die Speicheradresse des rechten bzw. linken Unterknoten speichern. Da nicht ein normaler Zeiger verwendet wird, sondern ein Erweiterter aus der *Boost-Bibliothek*, kommen zu den üblichen 64 Bit Speicher, noch weitere 64 Bit Speicher hinzu. In diesen zusätzlichen 64 Bit werden extra Funktionen, wie ein interner „*Referenzzähler*“, welcher für die intelligente Speicherverwaltung genutzt wird, gespeichert.
Aus diesen einzelnen Attributen ergibt sich ein Speicheraufwand von $2 * 4Byte + 3 * 4Byte +$

$2 * 16Byte = 56Byte$ pro Knoten. Daraus ergibt sich ein insgesamter Speicheraufwand von $O(2 * n * 56Byte)$.
Für den Beispiel Datensatz mit 5 Millionen Zellen, würde sich ein Speicheraufwand von $O(2 * 5 * 10^6 * 56Byte) = 534Megabyte$ ergeben.
Daraus ist zu erkennen, dass das Span Space Verfahren den Speicherverbrauch massiv in die Höhe treibt. Es wird also die Schnelligkeit auf Kosten des Speicherverbrauchs verbessert.

3 Umsetzung

Um die Theorie zu überprüfen und die theoretischen Berechnung nachzuweisen, musste das SpanSpace Verfahren in Openwalnut implementiert werden. Der Prozess der Implementierung wird in diesem Kapitel beschreiben.

Open Walnut ist modular aufgebaut, d.h. dass je nach Aufgabe verschiedene Module aus dem Katalog ausgewählt und geladen werden können.
Das Modul, das in Open Walnut für das Rendern der Isofläche zuständig ist, heißt *„Isosurface“*. Dieses Modul liest Skalardaten ein und rendert mittels des Marching Cubes Algorithmus eine Isofläche (Isosurface).

Code 1: Ausschnitt aus dem ursprünglichen Isosurface Modul

```
1  while( !m_shutdownFlag() )
2  {
3      [...]
4      // new data available?
5      if( dataChanged )
6      {
7          [...]
8          if( m_isoValueProp->get() >= m_dataSet->getMax() || m_isoValueProp->get() <= m_dataSet->
               getMin() )
9          {
10             m_isoValueProp->set( 0.5 * ( m_dataSet->getMax() + m_dataSet->getMin() ), true );
11         }
12     }
13     generateSurfacePre( m_isoValueProp->get( true ) );
14     [...]
15 }
```

Der Isowert kann über einen Schieberegler eingestellt werden. Bei jeder Bewegung dieses Reglers, wird dem Modul ein neuer Isowert mitgeteilt und die Isofläche neu berechnet. In der Praxis wird dies über eine *„while“* Schleife in Verbindung mit einer *„if“* Bedingung umgesetzt. In Code 1 Zeile 5 wird geprüft, ob sich die Eingabedaten (Isowert, Skalardaten) verändert haben. Wenn sich etwas verändert hat, wird in Zeile 10 ein neuer Isowert abgespeichert, wenn der jetzige außerhalb der Datenwerte liegt. In Zeile 13 wird die Isofläche mit dem neu festgelegten Isowert neu berechnet.

Die Methode *generateSurfacePre* überprüft, ob die Eingabedaten auch wirklich den zu erwartenden Daten entsprechen. Wenn keine Fehler auftreten, wird der eigentliche Marching Cubes Algorithmus über einen *ValueSetVisitor* aufgerufen.

Der *ValueSetVisitor* hat die Aufgabe den Datentyp der Eingabedaten in einen geeigneten Datentyp für den Marching Cubes Algorithmus umzuwandeln.

Der Marching Cubes Algorithmus marschiert dann durch den kompletten Datensatz. Zuerst wird der Würfel Index für die Vertices berechnet, deren Wert unter dem Isowert liegt.

Code 2: Berechnung des Würfel Indexes

```
1  unsigned int tableIndex = 0;
2  if( ( *vals )[ z * nPointsInSlice + y * nX + x ] < m_tIsoLevel )
3     tableIndex |= 1;
4  if( ( *vals )[ z * nPointsInSlice + ( y + 1 ) * nX + x ] < m_tIsoLevel )
5     tableIndex |= 2;
6  if( ( *vals )[ z * nPointsInSlice + ( y + 1 ) * nX + ( x + 1 ) ] < m_tIsoLevel )
7     tableIndex |= 4;
8  if( ( *vals )[ z * nPointsInSlice + y * nX + ( x + 1 ) ] < m_tIsoLevel )
9     tableIndex |= 8;
10 if( ( *vals )[ ( z + 1 ) * nPointsInSlice + y * nX + x ] < m_tIsoLevel )
11    tableIndex |= 16;
12 if( ( *vals )[ ( z + 1 ) * nPointsInSlice + ( y + 1 ) * nX + x ] < m_tIsoLevel )
13    tableIndex |= 32;
14 if( ( *vals )[ ( z + 1 ) * nPointsInSlice + ( y + 1 ) * nX + ( x + 1 ) ] < m_tIsoLevel )
15    tableIndex |= 64;
16 if( ( *vals )[ ( z + 1 ) * nPointsInSlice + y * nX + ( x + 1 ) ] < m_tIsoLevel )
17    tableIndex |= 128;
```

Die *if* Bedingungen in Code 2 prüfen, ob der Wert der Würfelecken unter dem Isowert liegt. Wenn eine Ecke unter dem Isowert liegt, wird der Würfel Index über den binären Operator *OR* (| =) neu definiert. Über diesen Würfel Index werden die Dreiecke ermittelt, die in dem Würfel gebildet werden (siehe Abb. 3).

Der binäre OR Operator vergleicht die Bits beider Variablen. Sobald einer der beiden zu vergleichenden Bits 1 ist, wird auch das resultierende Bit 1. Wenn beide zu vergleichenden Bits 0 sind, wird das resultierende Bit auch 0. Beispielsweise würde 48|64 → 0110000|1000000 → 1110000 → 112 ergeben.

Danach werden die exakten Koordinaten der einzelnen, enstehenden Dreiecke mittels Intersektion bestimmt. Diese Dreiecke werden zusammengesetzt und die Grafikkarte erstellt aus diesen dann die Isofläche.

In der optimierten Version wird zuerst die SpanSpace Klasse initialisiert. Diese Klasse kopiert aus den Skalardaten die Wertespanne (Span) aller Zellen in eine separate Liste.

Code 3: Zellen werden aus den Skalardaten in eine separate Liste kopiert

```
1 m_spanSpace.reserve( m_nCellsX * m_nCellsY * m_nCellsZ );
2 for( unsigned int z = 0; z < m_nCellsZ; z++ ){
3     ++*progressSpan;
4     for( unsigned int y = 0; y < m_nCellsY; y++ ){
5         for( unsigned int x = 0; x < m_nCellsX; x++ ){
6             m_spanSpace.push_back( cells_t( getCellMinMax( x, y, z ), x, y, z ) );
7         }
8     }
9 }
```

Dafür wird zuerst ein Vektor (Liste in C++) angelegt mit der Größe des Produktes der X, Y, Z Koordinaten (Code 3, Z. 1). So wird sicher gestellt, dass der reservierte Speicherplatz für den Vektor genau so groß ist, wie die Anzahl der zu speichernden Würfel.

Die *for* Schleifen in den Zeilen 2, 4, 5 in Code 3 arbeiten sich schrittweise durch alle möglichen Koordinaten, um dann ein *cells_t* Struct in den Vektor zu schreiben (Code 3, Z. 6). Durch diesen *cells_t* Struct wird ein einzelner Würfel beschrieben. Der *cells_t* Struct setzt sich aus einen MinMax Struct und ein KdCellId Struct zusammen. Das MinMax Struct beinhaltet den minimalen und den maximalen Wert des Würfels. Das KdCellId Struct beinhaltet die X, Y, Z Koordinate des Würfels.

Nach dem alle Würfel in den Vektor geschrieben wurden, wird die Methode *createTree* aus der *kdTree* Klasse aufgerufen. Diese Methode erstellt den KdTree.

Code 4: Methode zur Erstellung des KdTrees

```
1 template< class T > boost::shared_ptr< SSKdTreeNode< T > > WSpanSpaceKdTree< T >::createTree(
2       kdTree_t& points, unsigned int depth, size_t first, size_t last ){
3     size_t size = last - first;
4     size_t median = size / 2;
5     boost::shared_ptr< SSKdTreeNode< T > > kdNode( new SSKdTreeNode< T >() );
```

```
6     m_depth = depth;
7
8     if( size <= 1 ){
9         kdNode->m_location = points[ first + median ];
10        return kdNode;
11    }
12
13    std::nth_element( points.begin() + first, points.begin() + first + median, points.begin() +
          last,
14                    std::tr1::bind( &WSpanSpaceKdTree<T>::sortValues, this,
15                                    std::tr1::placeholders::_1, std::tr1::placeholders::_2 ) );
16
17    kdNode->m_location = points[ first + median ];
18    kdNode->m_leftTree = createTree( points, depth + 1, first, first + median );
19    kdNode->m_rightTree = createTree( points, depth + 1, first + median, last );
20
21    return kdNode;
22 }
```

Diese Methode verlangt 4 Parameter (Code 4, Z. 2). Der erste Parameter ist die Speicheradresse der Würfel Liste. Es wird nur die Speicheradresse weitergegeben, denn somit muss die Liste nicht für die Benutzung in der Funktion kopiert werden. Dadurch wird eine erhebliche Menge an Arbeitsspeicher eingespart. Der zweite Eingabewert ist die aktuelle Tiefe des KdTrees. Der dritte und vierte Eingabewert enthält einen Start und einen End Indexwert der Würfelliste.
Aus diesen beiden Werten wird in Code 4 Zeile 4 die Differenz gebildet, um auszurechnen, wie viele Würfel sich in der Spanne befinden. In Zeile 5 wird der Median dieser Spanne berechnet. In Zeile 7 wird der Tiefenpararmeter in der Klassenvariable *m_depth* abgespeichert. Diese Klassenvariable wird später zum Sortieren der Werte benutzt.
Wenn sich in dieser Spanne nur ein Würfel befindet, wird kein rechter bzw. linker KdTree Knoten gebildet, sondern es wird nur der einzelne Würfel in dem Knoten gespeichert. Dieser Knoten wird in Zeile 10 zurückgegeben. Durch das *return* wird die Funktion beendet und die Anweisungen darunter werden nicht mehr ausgeführt.
Wenn sich in dieser Spanne mehr als ein Würfel befindet, wird die Spanne mittels der *nth_element* Funktion sortiert. Diese C++ Funktion ist wie folgt definiert *nth_element(first, nth, last, comp)*. First und last sind die zwei Elemente die die Spanne, welche sortiert werden soll, begrenzen. *Nth* ist das „Medianelement“. Durch die Sortierung wird bezweckt, dass alle Elemente vor dem „Medianelement“ kleiner als dieses zu sein haben.
Der Pararmeter *comp* fordert eine Funktion mit 2 Eingabewerte die nach eigenen Sortierkriteri-

en entscheidet, welches der Elemente das größere bzw kleinere ist.

In diesem speziellen Fall, wird eine Hilfsfunktion verwendet, welche auf die eigentliche Funktion verweist. Dieser Schritt muss getan werden, weil die Funktion ein Template enthält um Typkonflite zu vermeiden.

Die Sortierfunktion ist wie folgt definiert *sortValues(cells_t i, cells_t j)*.

Code 5: Funktion zum Sortieren zweier Werte

```
1 template< class T > bool WSpanSpaceKdTree< T >::sortValues( cells_t i, cells_t j )
2 {
3     if( m_depth % 2 == 1 ){
4         return ( i.m_minMax.m_min < j.m_minMax.m_min );
5     }else{
6         return ( i.m_minMax.m_max < j.m_minMax.m_max );
7     }
8 }
```

In Code 5 Zeile 3 findet man den Modulo Operator % zwischen der Variable *m_depth* und der Zahl 2. Der Modulo Operator berechnet den Rest der bei der Division zweier Zahlen entsteht. Bei Berechnung des Modulos von einer beliebigen Zahl und 2 kann entweder 0 oder 1 als Ergebnis geliefert werden. So ist es möglich herauszufinden, ob die gegebene Zahl eine gerade oder ungerade Zahl ist. Denn gerade Zahlen haben bei der Division durch 2 keinen Rest und deshalb ist das Ergebnis der Modulo Berechnung 0. Bei ungeraden Zahlen entsteht jedoch bei der Division von 2 immer der Rest 1.

Wenn *m_depth* ungerade ist, also die Modulo Berechnung 1 zurück gibt, wird Zeile 4 ausgeführt. Bei einem geraden *m_depth* Wert, wird Zeile 6 ausgeführt. Durch die Unterscheidung je nach Tiefe, wird die abwechselnde Sortierung des KdTrees umgesetzt.

In Zeile 4 wird der Minimum Wert der beiden gegebenen Würfen verglichen. In Zeile 6 hingegen wird der Maximale Wert verglichen.

Nach dem die Spanne erfolgreich sortiert wurde, wird in dem Knoten ein rechter und ein linker Baum gespeichert. Dafür wird die Methode createTree aufgerufen. Durch diesen Aufruf wird der KdTree mit einer neuen Spanne rekursiv gebildet.

Nach dem der KdTree vollständig gebildet wurde, wird in dem Isosurface Modul aus dem kleinsten und dem größten auftretenden Wert der Mittelwert gebildet. Dieser Mittelwert wird als Start Isowert genommen. Mit diesem Isowert wird der Marching Cubes Algorithmus das erste mal ausgeführt.

Der Unterschied zur unoptimierten Variante ist, dass jetzt der Methode *generateSurface* der

Marching Cubes Klasse zusätzlich noch die Speicheradresse der zuvor initialisierten SpanSpace Klasse mit gegeben wird.
In der *generateSurface* Methode wird die Methode *findCells* der gegebenen SpanSpace Klasse aufgerufen.

Code 6: findCells Methode der SpanSpace Klasse

```
1  template< class T > boost::shared_ptr< WSpanSpaceBase::cellids_t > WSpanSpace<T>::findCells(
       double iso ){
2      boost::shared_ptr< cellids_t > cellVector( new cellids_t );
3      T isoValue = static_cast< T >( iso );
4      if( m_kdTreeNode ){
5          searchKdMinMax( m_kdTreeNode, isovalue, cellVector );
6      }else{
7          cellVector.push_back( &m_kdTreeNode->m_location.m_id );
8      }
9      return cellVector;
10 }
```

Wie in Code 6 Zeile 1 zuerkennen ist, fordert die Methode nur den Parameter *iso*. Dieser Parameter ist der Isowert, der von dem Isosurface Modul mitgeliefert wird.
In der Methode *findCells* wird nichts weiter gemacht, als eine Liste initialisiert (Code 6, Zeile 2). In Zeile 4 wird geprüft, ob der KdTree mehr als nur einen Würfel besitzt.
Wenn dies nicht der Fall sein sollte, wird der einzelne Würfel in die zuvor initialisierte Liste geschrieben (Code 6, Zeile 7). Sonst wird die eigentlichen Suchfunktion *searchKdMinMax* ausgeführt (Code 6, Zeile 5).

Code 7: searchMinMax Methode der SpanSpace Klasse

```
1  template< class T > void WSpanSpace<T>::searchKdMinMax( boost::shared_ptr< SSKdTreeNode< T > >
       root, double isovalue, cellids_t& cellVector ) const{
2     if( root->m_location.m_minMax.m_min <= isovalue ){
3        if( root->m_location.m_minMax.m_max > isovalue ){
4           if( !root->m_rightTree && !root->m_leftTree ){
5              cellVector.push_back( &root->m_location.m_id );
6           }
7        }
8        if( root->m_rightTree ){
9           searchKdMaxMin( root->m_rightTree, isovalue, cellVector );
10       }
```

```
11      }
12      if( root->m_leftTree ){
13          searchKdMaxMin( root->m_leftTree, isovalue, cellVector );
14      }
15  }
```

In der *searchKdMinMax* Methode, die in Code 7 zusehen ist, wird in Zeile 2 überprüft ob der Isowert größer oder gleich dem minimalen Wert des KdNodes ist. Sollte das der Fall sein, wird in Zeile 3 geprüft, ob der Isowert auch kleiner als der maximale Wert des KdNodes ist. Sollte das auch zutreffen, wird in Zeile 4 noch überprüft, ob der KdNode keine Subnodes hat, also ob das „Ende“ des KdTrees erreicht ist. Wenn das zutrifft, wird der Würfel in die Liste geschrieben.
Desweiteren wird in Zeile 9 der rechte KdTree mit der *searchMaxMin* Methode durchsucht, sofern er existiert und der Isowert größer oder gleich dem Wert des aktuellen KdNodes ist. Der linke Baum wird eben so mit der *searchMaxMin* Methode durchsucht, sofern der linke KdTree existiert (Code 7, Zeile 13).

Code 8: searchMaxMin Methode der SpanSpace Klasse

```
1   template< class T > void WSpanSpace<T>::searchKdMaxMin( boost::shared_ptr< SSKdTreeNode< T > >
        root, double isovalue, cellids_t& cellVector ) const{
2       if( root->m_location.m_minMax.m_max > isovalue ){
3           if( root->m_location.m_minMax.m_min <= isovalue ){
4               if( !root->m_rightTree && !root->m_leftTree ){
5                   cellVector.push_back( &root->m_location.m_id );
6               }
7           }
8           if( root->m_leftTree ){
9               searchKdMinMax( root->m_leftTree, isovalue, cellVector );
10          }
11      }
12      if( root->m_rightTree ){
13          searchKdMinMax( root->m_rightTree, isovalue, cellVector );
14      }
15  }
```

Die *searchMaxMin* Methode funktioniert genau so wie die *searchMinMax* Methode mit dem Unterschied, dass bei der *searchMaxMin* Methode der linke KdTree nur durchsucht wird (Code 8, Zeile 9), wenn dieser existiert und der Isowert kleiner als der maximale Wert des aktuellen KdNodes ist.
Nach dem die Liste mit den aktiven Würfel gefüllt wurde, wird diese zurückgegeben (Code 6, Zeile 9) und der Marching Cubes Algorithmus berechnet daraus die Isofläche.

Es wird hier nicht weiter auf die Implementierung, sondern nur auf die theoretische Arbeitsweise des Marching Cubes Algorithmus eigegangen, da die Implementierung nicht Teil dieser Arbeit ist.
Zusammenfassend kann gesagt werden, dass ein System implementiert wurde, das die einzelnen Würfel in einer Datenstruktur - den KdTree - speichert, um dann später aktive Würfel möglichst schnell zu extrahieren.

4 Analyse

4.1 Performance Analyse

Für die Analyse wurde die Zeit in Millisekunden über eine virtuelle Stoppuhr im Code gemessen. Die Zeitmessung wurde 10 mal wiederholt und aus den gemessenen Werten der Mittelwert gebildet. Mit diesen Mittelwert wurde durch Regression eine Funktion ermittelt.
Es wurde ein Datensatz eines Gehirnes mit 5 Millionen Würfeln verwendet.
Die Analyse fand auf einem Testsystem statt, welches mit einer AMD Phenom(tm) II X6 1100T Central Processing Unit (CPU) und 2xG Skill Intl F3-1866C9-8GSR Random Access Memory (RAM) ausgestattet war.

Bei der ersten Analyse wurde ein Isowert von 100 verwendet, bei dem 361376 (=7,18%) Würfel aktiv waren.
Der Marching Cubes Algorithmus ohne Optimierung brauchte für eine Berechnung eine mittlere Zeit von 248 Millisekunden, die optimierte Version brauchte 321 Millisekunden. Dazu musste für das SpanSpace Verfahren noch der KdTree gebildet werden, was eine mittlere Zeit von 6230 Millisekunden in Anspruch nahm.

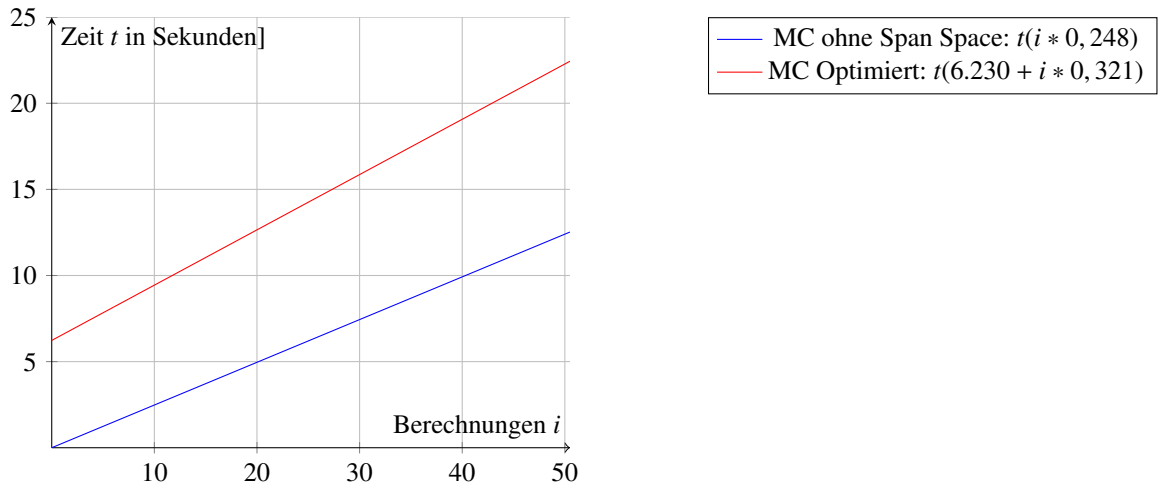


Abbildung 8: Visualisierung der Analyse zwischen optimierten und unoptimierten MC

Aus der Analyse in Abb. 8 wird sichbar, dass die Span Space Optimierung in der Praxis keine zeitliche Verbesserung mit sich gebracht hat. Aufgrund der Tatsache, dass dieses Ergebnis nicht mit den zuvor berechneten Ergebnis übereinstimmt, wurden weitere virtuelle Stoppuhren eingebaut, welche die Zeit der einzelnen Abläufe des Span Space Verfahrens messen.
Diese Messungen haben ergeben, dass das Suchen der aktiven Würfel aus dem KdTree nur 18 Millisekunden gebraucht hat. Das Berechnen des Marching Cubes Algorithmus für die rund 360.000 Würfen hat 303 Millisekunden gedauert.
Daraus ergibt sich das Paradoxon, wie es möglich sein kann, dass die Berechnung des Marching Cubes Algorithmus für 5 Millionen Würfel weniger Zeit in Anspruch nimmt, als die Berechnung für 0,35 Millionen Würfel.
Um diesem Paradoxon auf dem Grund zu gehen, muss die Arbeitsweise des unoptimierten Verfahrens betrachtet werden. Dieses Verfahren geht schrittweise durch die Koordinaten und übergibt diese dem Marching Cubes Algorithmus. Dadurch werden die Koordinaten nach der Position geordnet übergeben.
Als erstes wird der Würfel mit den Koordinaten (0|0|0) übergeben. Danach wird die X Koordinate solange hochgezählt, bis die Letzte X Koordinate erreicht ist. Wenn dies geschehen ist, wird die Y Koordinate um eins erhöht und die Prozedur wiederholt sich mit dem Würfel für die Koordinaten (0|1|0). Dieser Vorgang wird nun solange wiederholt, bis auch die Y Koordinate ihr Maximum erreicht hat und die Z Koordinate um eins erhöht wird.
Das Wichtige hierbei ist, dass die Daten des Datensatzes genau so im Speicher liegen. Der Hauptspeicher des Computers ist 1-Dimensional. Das bedeutet, dass alle X, Y, Z Paare hintereinander der Position nach geordnet im Speicher liegen. CPUs arbeiten so, dass sie die nächsten

Daten im Speicher schon einmal provisorisch in den Cache lädt.
Wenn nun also die Daten für die Nachfolgende X Koordinate geholt werden, dann befindet sich der Teil des Datensatzes bereits im Cache. Dadurch wird enorm Zeit eingespart!
Da die Werte, die das Span Space Verfahren liefert, nicht nach Position geordnet sind, werden jedes mal 3 unterschiedliche Koordinaten übergeben. Das bedeutet, dass der Prozessor für jeden Würfel 3 mal im RAM nachschauen muss, da sich diese Daten noch nicht im Cache befinden. Durch dieses Nachschauen ist das unoptimierte Verfahren im Vergleich zum optimierten Verfahren schneller!
Um eine Caching Gleichheit herzustellen, muss man die Würfel, die das Span Space Verfahren liefert, nach der Position sortieren. Dieser Sortiervorgang braucht zusätzlich Zeit! Im folgenden Diagramm wurde eine Sortierfunktion in das Span Space Verfahren eingebaut und die Bildungszeit des KdTree wurde zum besseren Vergleich nicht berücksichtigt. Außerdem wurde die insgesamte Zeit des Span Space Verfahren mit Sortierung in seine Zwischenschritte zerlegt.

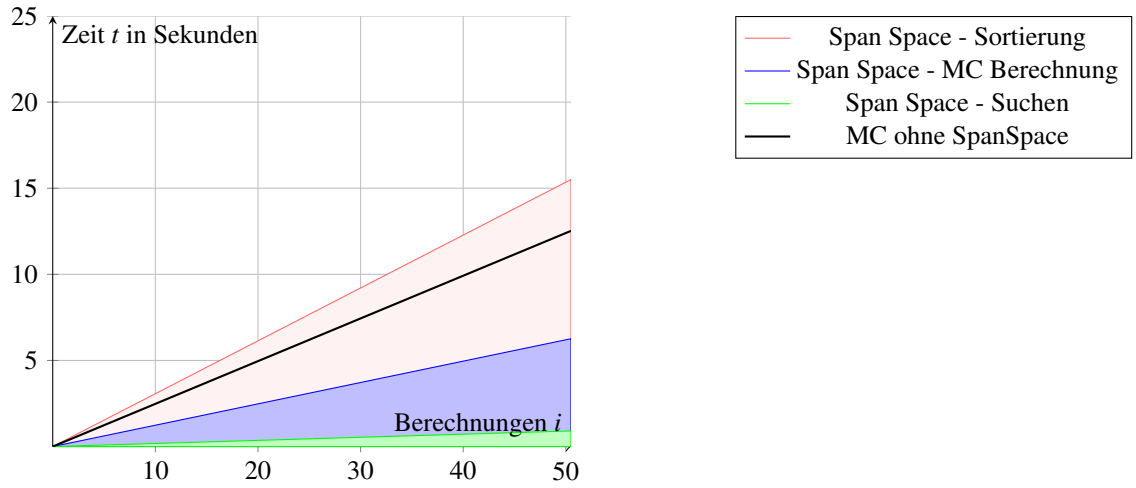


Abbildung 9: Visualisierung der Zeiten der Zwischenscritte des Span Space Verfahrens

Aus dem Diagramm in Abb. 9 wird ersichtlich, dass der Marching Cubes Algorithmus mit dem eigentlichen Span Space Verfahren schneller als der Marching Cubes Algorithmus ohne Span Space ist. Denn das eigentliche Span Space Verfahren setzt sich aus dem Durchsuchen des KdTree (grüne Schraffur) und dem Berechnen des Marching Cubes Algorithmus für die gefundenen Zellen (blaue Schraffur) zusammen. Doch durch die Sortierung (rote Schraffur) ist das Span Space Verfahren insgesamt gesehen langsamer als der Marching Cubes Algorithmus ohne Span Space.
Es wurde außerdem eine weitere Analyse mit einem recht großen Isowert von 190 durchgeführt, bei dem rund 50.000 Würfel aktiv waren, was 1 Prozent an aktiven Würfeln entspricht. In die-

sem Fall hat der Marching Cubes Algorithmus ohne Optimierung 137 Millisekunden pro Berechnung gebraucht und die optimierte Version hingegen nur 36 Millisekunden pro Berechnung. Diese Zeit setzt sich aus 3 Millisekunden Zellen aus dem KdTree suchen, 21 Millisekunden Marching Cubes berechnen und 12 Millisekunden Werte sortieren zusammen. Die Bildungszeit des KdTrees blieb gleich, da der selbe Datensatz verwendet wurde.

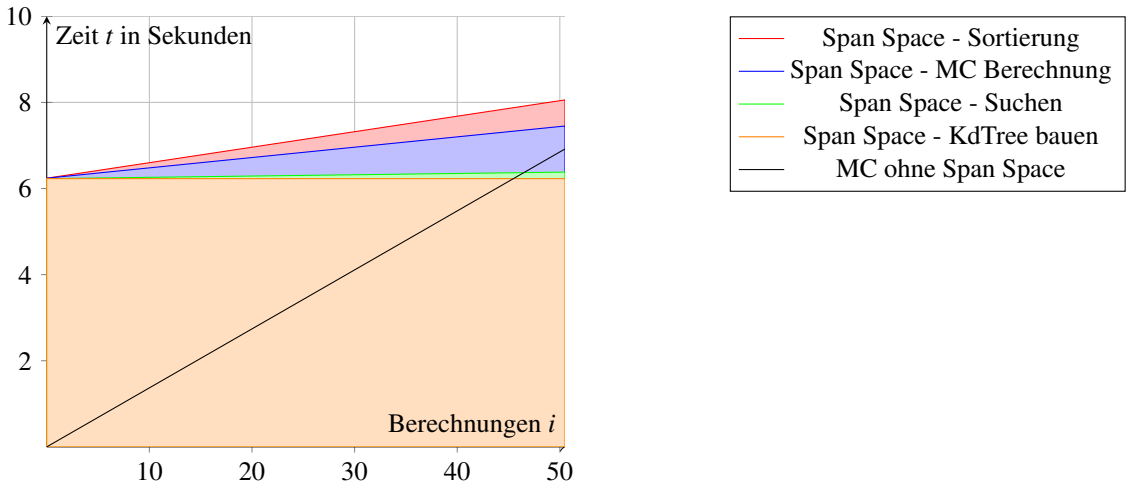


Abbildung 10: Visualisierung des kompletten Zeitverbrauches des Span Space Verfahrens

Aus Abb. 10 wird ersichtlich, dass die Zeit, die durch das schnelle Suchen des Span Space Verfahren eingespart wurde, in keiner Relation zu der Zeit steht, die für das Bauen des KdTree benötigt wird. Jedoch ist diese Zeit belanglos, weil der KdTree pro Datensatz nur einmal erstellt werden muss und es möglich ist unendlich viele Isoflächen mit ihm zu berechnen. Somit ist die Zeit, die für das Erstellen des KdTree benötigt wird, zu vernachlässigen.

Desweiteren muss unbedingt das Szenario analysiert werden, wie sich der unoptimierte Marching Cubes Algorithmus ohne Prozessor Caching im Vergleich zu dem optimierten Marching Cubes Algorithmus ohne Prozessor Caching verhält.

Um dieses Szenario zu realisieren, wurden die Würfel dem Marching Cubes nicht in X Richtung, sondern in Z Richtung übergeben. Da die Werte nicht so im Speicher liegen, greift auch kein Caching.

Für diese Analyse wurde wieder ein Isowert von 100 gewählt, bei dem ca. 7% der Würfel aktiv sind.

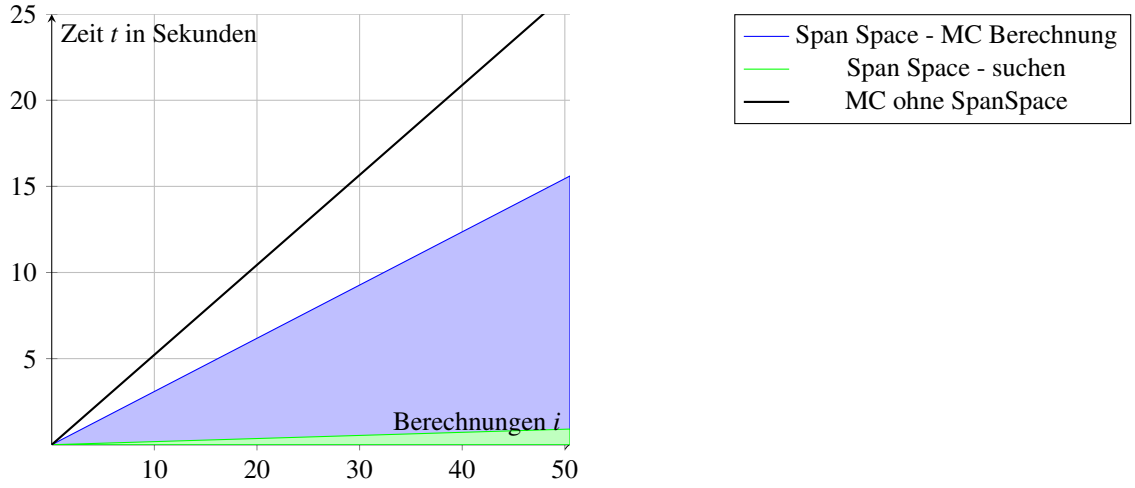


Abbildung 11: Visualisierung der Zeiersparnis des Span Space Verfahrens gegenüber dem MC ohne Caching

Aus dem Diagramm in Abb. 11 wird deutlich, dass das Span Space Verfahren den Marching Cubes Algorithmus optimiert, wenn keine Prozessor Caching aktiv ist.

4.2 Speicher Analyse

Für die Speicher Analyse wurde das Linux Programm *htop* verwendet, mit dem es möglich ist den Speicherverbrauch einzelner Programme zu analysieren.
Der Speicherverbrauch wurde nach 3 verschiedenen Ereignissen gemessen. Die erste Messung erfolgte nach dem Start von Open Walnut, die zweite Messung erfolgte nach dem der Datensatz geladen wurde und die dritte Messung erfolgte nach dem der KdTree erstellt wurde.
Der Speicherverbrauch des KdTree in der Praxis wird aus der Differenz der 3. und 2. Messung errechnet. Der theoretische Speicherverbrauch wurde mithilfe der Formel $O(2 * 56Byte * n)$ berechnet.

Würfel	1. Messung	2. Messung	3. Messung	KdTree Speicher Praxis	KdTree Speicher Theorie
$1 * 10^6$	112MB	129MB	149MB	120MB	106MB
$5 * 10^6$	112MB	144MB	798MB	654MB	534MB
$10 * 10^6$	112MB	208MB	1174MB	1382MB	1068MB

Tabelle 2: Analyse des Speicherverbrauches des Span Space Verfahrens

Wie in Tabelle 2 zu erkennen ist, existiert ein kleiner Unterschied zwischen dem realen und dem theoretischen Speicherverbrauch des KdTree. Dieser Unterschied kann jedoch vernachlässigt werden, da er durch Hintergrundprozesse in Open Walnut entsteht, die nicht bei der Analyse berücksichtigt wurden.

Im Bezug auf die Werte kann davon ausgegangen werden, dass die Implementierung des KdTree korrekt war.

5 Ergebnisse

5.1 Zusammenfassung der Umsetzung

Wie in Abb. 12 zu erkennen ist, ist der Aufbau ohne SpanSpace Optimierung sehr simpel. Bei der Initialisierung des Isosurface Moduls wird der Mittelwert aller Werte als Isowert benutzt und mit der Liste aller Würfel dem Marching Cubes Algorithmus übergeben. Der Suchvorgang funktioniert genau so, bis auf den Unterschied, dass nun nicht der „mittlere“ Isowert verwendet wird, sondern der Isowert, der in dem Isosurface Modul mit dem Schieberegler eingestellt wird.

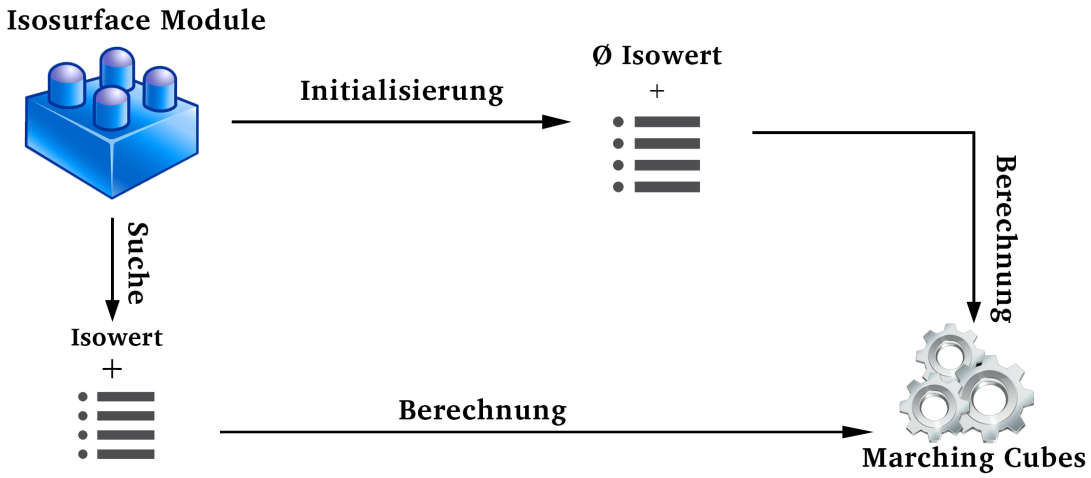


Abbildung 12: Aufbau des Marching Cube Algorithmus ohne SpanSpace Optimierung

Im Gegensatz dazu, ist der Marching Cubes Algorithmus mit Optimierung um einiges komplexer, wie in Abb. 13 zu sehen ist. Bei der Initialisierung des Isosurface Moduls wird die Liste aller Würfel in einen KdTree umgewandelt. Sobald dieser KdTree erstellt ist, wird der KdTree und der „mittlere“ Isowert an das SpanSpace Verfahren übergeben.
Bei der Suche muss der KdTree nicht erneut erstellt werden. Deshalb kann der zuvor generierte KdTree mit dem eingestellten Isowert an das SpanSpace Verfahren direkt übergeben werden. Dieses Verfahren durchsucht den KdTree und gibt eine Liste an den Marching Cubes Algorithmus weiter, welche nur „aktive“ Zellen beinhaltet.

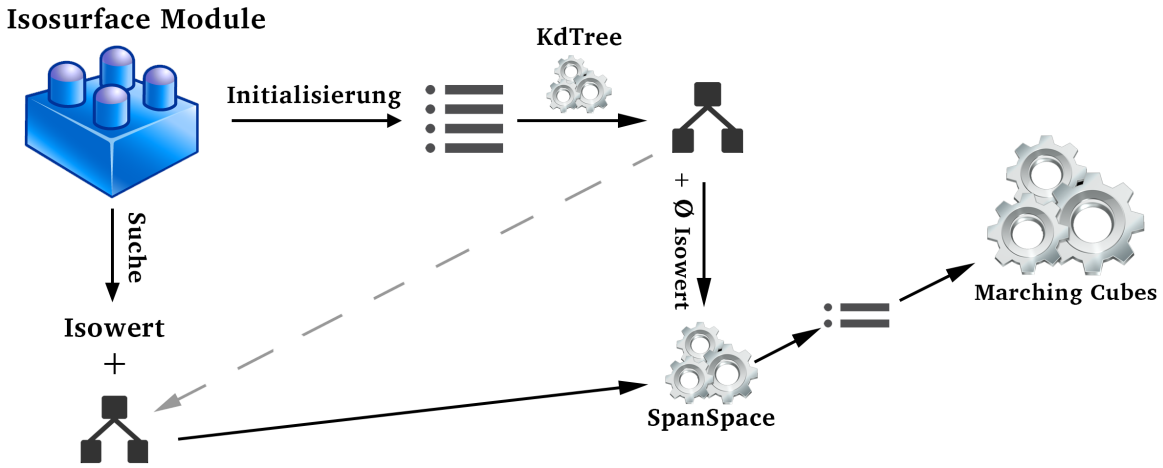


Abbildung 13: Aufbau des Marching Cube Algorithmus mit SpanSpace Optimierung

5.2 Auswertung Analysen

Aus den Analysen wird deutlich, dass die naive Implementierung des Span Space Verfahrens heutzutage keine zeitliche Verbesserung bringt.
Denn bei einem durchschnittlichen Datensatz, bei dem 10% der Würfel aktiv sind, braucht das Span Space Verfahren mindestens genau so lange wie der unoptimierte Marching Cubes Algorithmus. Dies liegt an der Zeit, die zusätzlich zur Sortierung der Werte benötigt wird.
Die Zeit, die für das einmalige Erstellen des KdTree gebraucht wird, ist zwar sehr hoch im Vergleich zu den einmaligen Berechnen des unoptimierten Marching Cubes Algorithmus. Jedoch ist diese Zeit zu vernachlässigen, weil der KdTree eben nur einmal pro Datensatz gebildet werden muss.
Vor 30 Jahren, als die Technologie der Prozessor noch nicht so weit entwickelt wie heute war, war das Span Space Verfahren wirklich eine sinnvolle Verbesserung. Denn damals brauchte der unoptimierte Marching Cubes Algorithmus um einiges mehr an Zeit. Dies lag einmal an den allgemein sehr leistungsschwachen Prozessoren und daran, dass diese Prozessoren noch keine moderne Caching Technologie wie die heutigen besaßen.
Vor 30 Jahren war Rechenleistung auch ein Luxusgut, was man sich kaufen konnte. Deshalb ist es verständlich, dass man probiert hat, jeden Algorithmus maximal zu optimieren.
Abschließend ist zu sagen, dass nicht das Span Space Verfahren zu langsam ist, sondern der Marching Cubes Algorithmus aufgrund neuer Technologie einfach zu schnell ist!

6 Ausblick

Um das Span Space Verfahren wieder rentabel zu machen, muss Zweierlei massiv optimiert werden. Zum Einen muss das Erstellen des KdTree und zum Anderen die Sortierung der Werte nach der Position optimiert werden.
Um das Sortieren der Werte möglichst effektiv zu gestalten, muss ein Algorithmus entwickelt werden, der die Werte schon bei der Erstellung des KdTree so gut wie möglich nach der Position zu ordnet.
Um die Bildungszeit des KdTree zu optimieren, sind viele kleine Optimierungen durchzuführen.
Grundsätzlich sollte man sich von den speziellen Zeigern aus der *Boost Bibliothek* trennen und die Standardzeiger von C++ verwenden. Dies hätte einmal eine Verbesserung hinsichtlich des verbrauchten Arbeitsspeichers zur Folge, weil die Zusatzfunktionen wegfallen würden. Außerdem würde durch Wegfallen der Zusatzfunktionen auch die Performance allgemein verbessert werden.
Eine weitere Verbesserung die unbedingt umgesetzt werden sollte ist, dass der KdTree nicht mit Hilfe von Listen organisiert wird, sondern eine komplett eigene Datenstruktur anstelle von Listen verwendet. So würden nicht die recht langsamen „Vektoren“ aus der Standard Bibliothek verwenden werden und es würden wieder ein paar Millisekunden einspart werden.
Trotz dieser Einsparungen wird es nicht möglich sein an die Schnelligkeit des unoptimierten Marching Cubes zu erreichen, wenn das originale Span Space Verfahren benutzt wird. Würde es jedoch ein Möglichkeit geben, dass der KdTree die Würfel nach der Größe sortiert und so sortiert zurück gibt, wäre das Span Space Verfahren schneller, da so das Prozessor-Interne-Caching ausgenutzt werden kann.
Desweiteren kann der Marching Cubes Algorithmus auch auf die Graphics Processing Unit (GPU) ausgelagert werden, was hinsichtlich der Performance ebenfalls sinnvoll wäre. Denn die GPU ist um einiges schneller als die CPU. Da Grafikkarten kein vergleichbares, automatisches Caching wie die CPU hat, würde sich die Implementierung des Span Space Verfahren hinsichtlich der Performance lohnen.

Literatur

[1] CppReference. *std::nth_element*. Aug. 2013. URL: http://en.cppreference.com/w/cpp/algorithm/nth_element (besucht am 09. 11. 2013).

[2] William E. Lorensen und Harvey E. Cline. "A high resolution 3D surface construction algorithm". In: *Computer Graphics* 21.4 (Juli 1987), S. 163–169.

[3] Timothy S. Newman und Hong Yi. "A survey of the marching cubes algorithm". In: *Computer Graphics* 30.5 (Okt. 2006), S. 854–879.

[4] Yarden Livnat, Han-Wei Shen und Christopher R. Johnson. "A Near Optimal Isosurface Extraction Algorithm Using the Span Space". In: *IEEE Transactions on Visualization and Computer Graphics* 2.1 (1996), S. 73–84. ISSN: 1077-2626. DOI: http://dx.doi.org/10.1109/2945.489388.

[5] Christopher Mielack. "Isosurfaces". In: (Nov. 2009). URL: http://www.zib.de/mielack/isosurfaces.pdf.

[6] Dirk Moosbach. *Wortbedeutung ISO (Deutsch)*. URL: http://www.wortbedeutung.info/ISO/ (besucht am 11. 10. 2013).

[7] John Page. *Polygon - math word definition*. 2009. URL: http://www.mathopenref.com/polygon.html (besucht am 23. 11. 2013).

[8] Webopedia. *voxel*. 2013. URL: http://www.webopedia.com/TERM/V/voxel.html (besucht am 23. 11. 2013).

[9] Wikipedia. *Marching Cubes*. Juli 2013. URL: http://de.wikipedia.org/wiki/ (besucht am 27. 09. 2013).

[10] Wikipedia. *Voxel*. Okt. 2011. URL: http://de.wikipedia.org/wiki/Voxel (besucht am 11. 10. 2013).

[11] Jürgen Wolf. *C++ von A bis Z - das umfassende Handbuch*. 1. Aufl. Bonn: Galileo Press, 2006. ISBN: 978-3-898-42816-3.

[12] Daisy Yuhas. *What's a Voxel and What Can It Tell Us? A Primer on fMRI*. Juni 2012. URL: http://blogs.scientificamerican.com/observations/2012/06/21/whats-a-voxel-and-what-can-it-tell-us-a-primer-on-fmri/ (besucht am 23. 11. 2013).

Eidesstattliche Erklärung

Ich, David Geistert, versichere hiermit, dass ich meine BELL mit dem Thema

Optimierung des Marching Cubes Algorithmus durch das Span Space Verfahren

selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, wobei ich alle wörtlichen und sinngemäßen Zitate als solche gekennzeichnet habe. Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Leipzig, den 30. März 2014

_____

David Geistert

Danksagung

An dieser Stelle möchte ich mich bei allen Personen bedanken, die mich bei der Erstellung dieser Arbeit unterstützt haben.
Ein besonderer Dank gilt meinem Betreuer Herrn Eichelbaum, der mir mit sehr viel Engagement, guten Ideen und unermüdlichen Einsatz meine Bell betreut hat.
Ebenfalls bedanken möchte ich mich bei Frau Dr. Meiler, die durch Ihre Vermittlung diese Bell erst ermöglicht hat.
Bedanken möchte ich mich auch bei meiner Informatik Lehrerin Frau Naundorf und Informatik Lehrer Herrn Gögel für ihre Betreuung.
Ein weiterer Dank gebührt meiner Mutter Ina Geistert für Ihre Unterstützung.